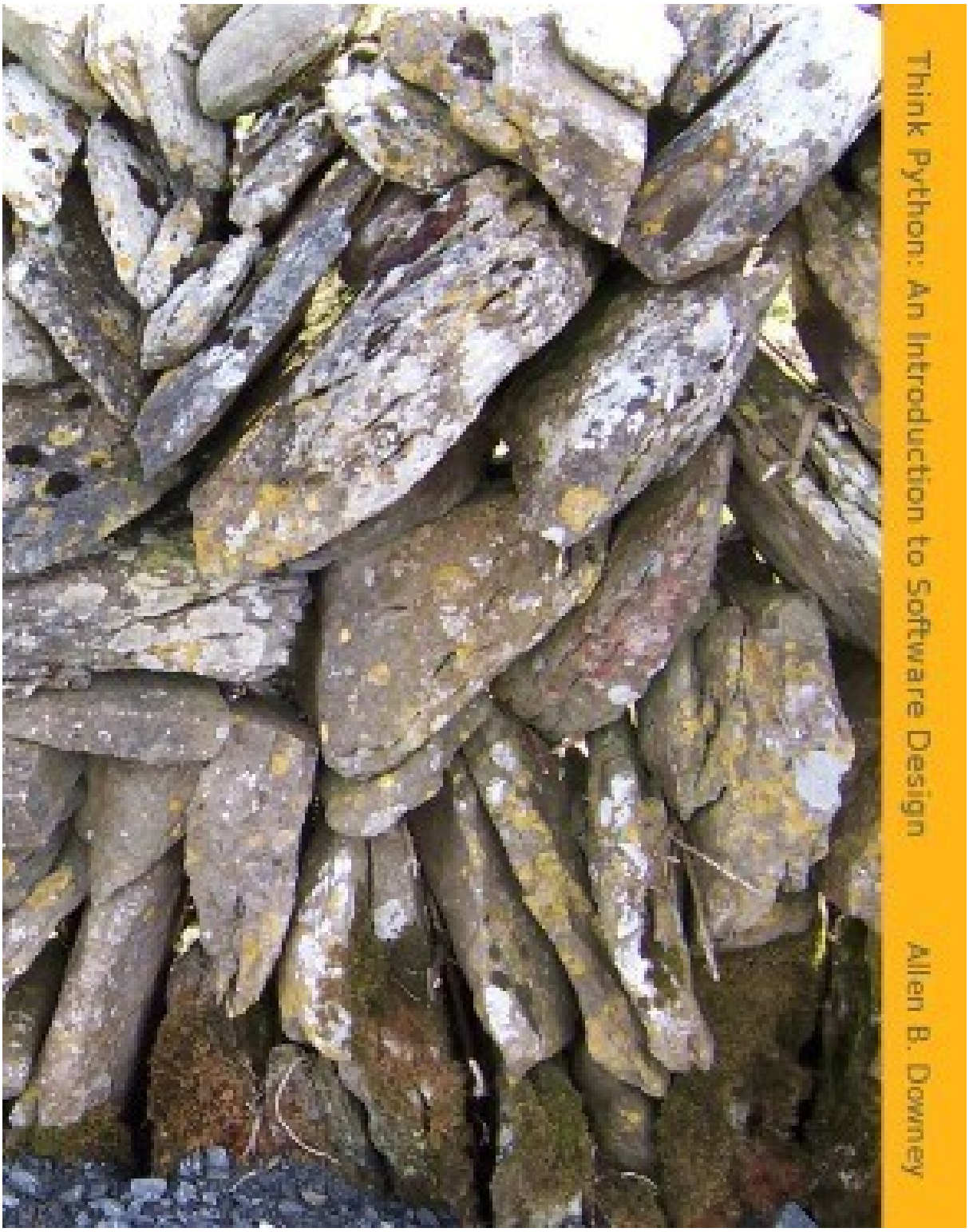
Think Python: How to think like a compter scientist

Allen B. Downey



Sự ra đời kì lạ của cuốn sách này

(Lời giới thiệu của tác giả)

Tháng Một năm 1999 tôi chuẩn bị dạy một lớp học nhập môn lập trình ngôn ngữ Java. Tôi đã từng dạy khoá học này ba lần và cảm thấy không hài lòng. Sinh viên có tỉ lệ thi trượt rất cao, và ngay cả những người qua được, thì điểm cũng không khả quan.

Một trong những vấn đề tôi thấy được là ở những cuốn sách giáo trình. Chúng thường quá dày, với nhiều chi tiết nhỏ nhặt về Java, và không có đủ những hướng dẫn lập trình theo tầm nhìn bao quát. Và chúng đều bị mắc phải hiệu ứng "cửa sập": khởi đầu rất dễ dàng, phát triển từ từ, và đến Chương 5 thì lôi ra đủ mọi kiến thức. Sinh viên sẽ tiếp thu quá nhiều tài liệu, quá gấp gáp, và hậu quả cuối cùng là đến cuối kì thì "chữ thầy trả thầy".

Hai tuần lễ trước khi khoá học bắt đầu, tôi quyết định viết quyển sách của riêng mình. Mục tiêu của tôi là:

- Viết ngắn gọn. Để sinh viên đọc 10 trang thì hay hơn là 50 trang.
- Chú ý đến từ ngữ. Tôi cố gắng hạn chế dùng các thuật ngữ, và mỗi khi dùng lần đầu thì định nghĩa chúng luôn.
- Xây dựng dần dần. Để tránh các tình trạng "cửa sập", tôi đem chia nhỏ những chủ đề khó thành một chuỗi các bước kế tiếp nhau.
- Chú ý đến lập trình thay vì ngôn ngữ lập trình. Tôi chỉ trình bày phần rất nhỏ nhưng thiết yếu của Java và lược qua tất cả phần còn lại.

Nhan đề cuốn sách tôi chọn theo ý thích của riêng mình, là *Cách tư duy như nhà khoa học máy tính*.

Phiên bản ban đầu rất sơ lược, nhưng đã có hiệu quả. Sinh viên nghiêm túc đọc tài liệu, và hiểu rằng khi lên lớp tôi chỉ giảng về những phần khó, còn những chủ đề hay (và quan trọng nhất) là để cho sinh viên luyện tập.

Tôi đã phát hành quyển sách theo giấy phép Văn bản tự do của GNU, theo đó người dùng có thể tự sao chép, sửa đổi và phân phối sách.

Câu chuyện tiếp diễn rất thú vị. Jeff Elkner, một giáo viên trung học dạy tại Virginia, đã chọn lấy cuốn sách của tôi và biên tập với ngôn ngữ Python. Ông đã gửi tôi một bản dịch, và tôi đã có một kinh nghiệm thú vị khi học được Python từ chính sách của mình.

Jeff và tôi đã hiệu đính lại quyển sách, thêm vào một phần ví dụ thực tế của Chris Meyers, và năm 2001 chúng tôi phát hành *Cách tư duy như nhà khoa học máy tính: Học với ngôn ngữ Python*, cũng theo Giấy phép Văn bản tự do của GNU.

Với nhà xuất bản Green Tea, tôi phát hành quyển sách và bắt đầu bán những cuốn sách in, qua Amazon.com và các hiệu sách đại học. Những cuốn sách khác cùng nhà xuất bản Green Tea đều có tại greenteapress.com.

Năm 2003 tôi bắt đầu dạy tại Đại học Olin College, cũng là lần đầu tiên tôi dạy Python. Nét tương phản với Java thật là ấn tượng. Sinh viên đã đỡ vất vả, học được nhiều hơn, tham gia nhiều dự án thú vị hơn, và nói chung đều rất vui vẻ.

Trong khoảng năm năm qua tôi vẫn tiếp tục chỉnh biên cuốn sách, sửa lỗi, cải thiện các ví dụ và thêm vào tư liệu, đặc biệt là các bài tập. Trong năm 2008 tôi đã bắt đầu làm việc với một phiên bản chính—cùng lúc đó tôi có được hợp đồng với một biên tập viên tại Nhà xuất bản Đại học Cambridge. Họ muốn tiếp tục phát hành một ấn bản kế tiếp. Thật kịp thời!

Kết quả là cuốn sách này, bây giờ đã với tên gọi ngắn gọn hơn: *Tư duy trong Python*. Một số sửa đổi bao gồm:

- Tôi đã thêm vào một mục ở cuối mỗi chương, chuyên về gỡ lỗi. Mục này trình bày những kĩ thuật chung để phát hiện và tránh lỗi khi lập trình, và cảnh báo những bẫy nhỏ trong Python.
- Tôi lược bỏ một số phần trong những chương cuối, về tạo lập các danh sách và cấu trúc cây. Mặc dù vẫn thích những chủ đề này, nhưng tôi nghĩ rằng chúng không phù hợp với phần còn lại của cuốn sách.
- Tôi đã thêm vào các bài tập, từ những bài kiểm tra ngắn về độ hiểu bài cho đến một vài chương trình phần mềm thực sự.
- Tôi bổ sung thêm một loạt các chương trình cụ thể—những ví dụ dài hơn với bài tập, lời giải, và biện luận. Một số trong đó dựa trên Swampy, một bộ chương trình Python mà tôi đã soạn thảo cho quá trình dạy trên lớp. Swampy, mã lệnh, và lời giải được tải lên trang thinkpython.com.
- Tôi đã mở rộng các kế hoạch xây dựng chương trình và những kiểu mẫu thiết kế cơ bản.
- Cách dùng của Python trong sách đã dựa vào nhiều điểm đặc thù của ngôn ngữ lập trình này. Dù rằng mục đích chủ yếu của cuốn sách là dạy về lập trình chứ chứ không phải Python, song tôi nghĩ rằng nhờ ngôn ngữ này mà chất lượng cuốn sách đã được nâng cao.

Tôi hi vọng bạn thích đọc cuốn sách này, với mục đích giúp cho bạn học cách lập trình và suy nghĩ theo kiểu một nhà khoa học máy tính.

Allen B. Downey
Needham Massachusett, Hoa Kì.

Chương 1: Cơ chế của chương trình máy tính

Mục đích của cuốn sách này là hướng dẫn bạn suy nghĩ như là một nhà khoa học máy tính. Cách tư duy này kết hợp những ưu điểm của khoa học tự nhiên, trong đó có toán học, với kĩ thuật. Cũng như những nhà toán học, những nhà khoa học máy tính dùng những ngôn ngữ có cấu trúc để diễn đạt ý tưởng (đặc biệt là tính toán). Giống như những kĩ sư, họ cũng làm công việc thiết kế, gắn kết các thành phần tạo nên một hệ thống và đánh giá những ưu khuyết giữa các phương án khác nhau. Giống như những nhà khoa học, họ khảo sát các động thái của hệ thống phức tạp, đề ra các giả thiết, và kiểm định những tính toán.

Kĩ năng quan trọng nhất của nhà khoa học máy tính là **giải quyết vấn đề**. Giải quyết vấn đề chính là cách tạo lập vấn đề, suy nghĩ giải pháp một cách sáng tạo, và trình bày giải pháp một cách rõ ràng và chính xác. Như bạn sẽ thấy, việc học lập trình chính là một cơ hội tuyệt vời để bạn luyện tập những kĩ năng giải quyết vấn đề. Đó là lí do tại sao chương này lại có tên là "Cơ chế của chương trình máy tính".

Một mặt, bạn sẽ được học cách lập trình, vốn bản thân nó là một kĩ năng hữu dụng. Mặt khác, bạn sẽ dùng lập trình như một phương tiện để giải quyết vấn đề. Điều này bạn sẽ dần dần làm được trong quá trình học.

Ngôn ngữ lập trình Python

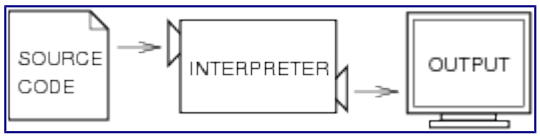
Ngôn ngữ lập trình mà bạn sẽ học là Python. Python là một ví dụ trong số các **ngôn ngữ lập trình bậc cao**; một số ngôn ngữ lập trình bậc cao khác mà bạn có thể biết đến gồm có C, C++, Perl, và Java.

Cũng có những **ngôn ngữ lập trình bậc thấp**, đôi khi mà ta gọi là "ngôn ngữ máy" hoặc "hợp ngữ". Nói nôm na, máy tính chỉ có thể thực hiện các chương trình được viết bằng ngôn ngữ bậc thấp. Vì vậy những chương trình được viết bằng một ngôn ngữ bậc cao cần được xử lý trước khi chúng có thể chạy được. Bước phụ trợ này sẽ tốn thêm thời gian, đây là một nhược điểm của các ngôn ngữ bậc cao.

Tuy vậy, các ưu điểm là rất lớn. Thứ nhất, việc lập trình bằng ngôn ngữ bậc cao dễ hơn rất nhiều. Chương trình được viết bằng ngôn ngữ bậc cao được viết nhanh hơn, nội dung chương trình ngắn hơn, dễ đọc hơn, và nhiều khả năng là chúng chính xác. Thứ hai, các ngôn ngữ bậc cao có tính **khả chuyển** theo nghĩa chạy được trên nhiều hệ máy tính khác nhau mà ít hoặc không cần phải sửa đổi. Các chương trình bậc thấp chỉ có thể chạy trên một loại máy tính và phải được viết lại nếu muốn chạy trên các hệ máy khác.

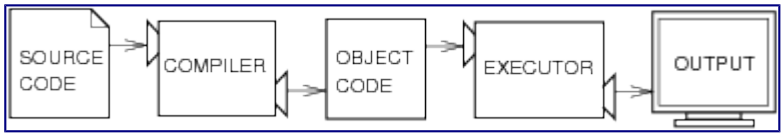
Bởi các ưu điểm nêu trên, hầu hết các chương trình đều được lập trình bằng ngôn ngữ bậc cao. Các ngôn ngữ bậc thấp chỉ được dùng cho một số ít những ứng dụng đặc biệt.

Hai loại chương trình có nhiệm vụ chuyển đổi các ngôn ngữ bậc cao về dạng ngôn ngữ bậc thấp: **trình thông dịch** và **trình biên dịch**. Trình thông dịch đọc một chương trình bậc cao và thực hiện nó theo đúng những gì mà chương trình chỉ định. Nó xử lý chương trình một cách dần dần, nghĩa là đọc câu lệnh đến đâu thì thực hiện tính toán tới đó.



Mã nguồn – Trình thông dịch – Kết quả đầu ra

Còn trình biên dịch thì đọc chương trình và dịch nó hoàn toàn trước khi chương trình bắt đầu chạy. Theo nghĩa đó, chương trình bậc cao được gọi là **mã nguồn**, và chương trình được dịch gọi là **mã đối tượng**, hoặc **chương trình chạy**. Một khi chương trình được biên dịch rồi, bạn có thể thực hiện nó nhiều lần sau này mà không phải dịch nữa.



Mã nguồn – Trình biên dịch – Mã đối tượng – Trình thực thi – Kết quả đầu ra

Python được coi là ngôn ngữ thông dịch vì chương trình Python được thực hiện bởi trình thông dịch. Có hai cách sử dụng trình thông dịch: theo **chế độ tương tác** và **chế độ văn lệnh**. Trong chế độ tương tác, bạn gõ vào các lệnh Python và trình thông dịch sẽ hiện kết quả lên màn hình:

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

Dấu `>>>` ở đây là **dấu nhắc** mà trình thông dịch dùng để thông báo rằng hiện giờ nó đang sẵn sàng đợi lệnh. Nếu bây giờ bạn gõ vào `1 + 1`, thì trình thông dịch sẽ trả lời là `2`.

Mặt khác, bạn cũng có thể lưu mã lệnh trong một file và sử dụng trình thông dịch để thực hiện nội dung của file, mà ta gọi là một **văn lệnh**. Theo quy ước, các văn lệnh Python đều có đuôi là `.py`.

Để thực hiện văn lệnh, bạn phải báo cho trình biên dịch biết tên file. Chẳng hạn, trong cửa sổ lệnh UNIX, bạn cần gõ vào `python dinsdale.py`. Trong các môi trường khác, cách thực hiện văn lệnh có thể khác đi. Bạn có thể tham khảo một số hướng dẫn trên trang Web của Python: `python.org`.

Làm việc trong chế độ tương tác rất thuận tiện nếu bạn cần kiểm tra các đoạn mã ngắn vì bạn có thể gõ trực tiếp và chúng được thực hiện ngay. Nhưng nếu mã lệnh gồm nhiều dòng thì bạn nên lưu chúng trong một file văn lệnh để sau này có thể chỉnh sửa và thực hiện chúng.

Chương trình là gì?

Chương trình là một danh sách các chỉ dẫn cách thực hiện tính toán. Việc tính toán có thể là thuần tuý toán học, chẳng hạn giải hệ phương trình hoặc tìm nghiệm đa thức, nhưng cũng có thể là những phép tính trên các kí hiệu, chẳng hạn tìm kiếm và thay thế chữ trong một văn bản, hoặc (kì lạ hơn) là biên dịch một chương trình.

Dù chi tiết có thể khác nhau tuỳ theo từng ngôn ngữ lập trình, nhưng một số chỉ dẫn luôn có trong mọi ngôn ngữ:

nhập số liệu:
: Là việc lấy số liệu từ bàn phím, file, hoặc một thiết bị khác.

xuất kết quả:
: Hiển thị kết quả trên màn hình hoặc gửi kết quả ra file hoặc một thiết bị khác.

tính toán:
: Thực hiện các phép toán cơ bản như cộng và nhân.

thực hiện có điều kiện:
: Kiểm tra một điều kiện cụ thể và thực hiện danh sách câu lệnh tương ứng với điều kiện đó.

tính lặp:
: Thực hiện lặp lại công việc nhiều lần, thường là với một số thay đổi giữa các lần lặp.

Tin hay không thì tùy bạn, nhưng bất cứ một chương trình nào, dù phức tạp đến đâu, đều được cấu thánh từ những chỉ dẫn đơn giản ở trên. Vì vậy, có thể coi lập trình như việc chia một bài toán lớn, phức tạp thành nhiều bài toán nhỏ hơn cho đến khi từng bài toán nhỏ này đơn giản đến mức có thể được thực hiện theo một trong các chỉ dẫn trên đây.

Điều này có thể còn mơ hồ, nhưng ta sẽ quay lại khi bàn về **thuật toán**.

Gỡ lỗi là gì?

Việc lập trình rất hay mắc phải lỗi. Việc theo dõi, phân tích nguyên nhân gây ra lỗi được gọi là **gỡ lỗi**.

Có ba loại lỗi có thể xuất hiện trong chương trình: lỗi cú pháp, lỗi chạy và lỗi ngữ nghĩa. Để nhanh chóng tìm ra lỗi ta cần phân biệt được chúng.

Lỗi cú pháp

Python chỉ có thể thực hiện được một chương trình với những câu lệnh đúng theo cú pháp; nếu không, trình thông dịch sẽ đưa ra thông báo lỗi. **Cú pháp** nghĩa là cấu trúc của chương trình và các quy tắc về cấu trúc đó. Chẳng hạn, ngoặc đơn phải đi theo từng cặp, như vậy `(1 + 2)` là hợp lệ, nhưng `7)` là một **lỗi cú pháp**.

Trong ngôn ngữ hàng ngày người ta có thể bỏ qua nhiều lỗi cú pháp, nhất là trong cách viết văn thơ. Python thì không như vậy. Nếu trong chương trình có bất cứ lỗi cú pháp nào, Python sẽ hiển thị thông báo lỗi và dừng chạy chương trình. Nếu bạn mới nhập môn lập trình được vài tuần, rất có thể bạn phải dành nhiều thời gian dò tìm lỗi. Khi kinh nghiệm tăng dần lên, bạn sẽ tránh được lỗi tốt hơn và nếu mắc thì cũng phát hiện ra lỗi nhanh hơn.

Lỗi thực thi

Loại lỗi thứ hai là lỗi thực thi; chúng có tên như vậy bởi vì chỉ xuất hiện khi chương trình đã bắt đầu chạy. Những lỗi kiểu này được gọi là **biệt lệ** bởi vì chúng thường chỉ những điều kiện (xấu) bất thường phát sinh.

Với những chương trình đơn giản trong một vài chương đầu tiên, ta ít gặp những lỗi chạy chương trình kiểu như vậy.

Lỗi ngữ nghĩa

Loại lỗi thứ ba là **lỗi ngữ nghĩa**. Trong trường hợp có lỗi kiểu này, chương trình vẫn chạy thông theo nghĩa máy sẽ không phát thông báo lỗi, nhưng sẽ không thực hiện đúng yêu cầu mong muốn, mà sẽ cho kết quả khác. Cụ thể là thực hiện theo đúng những hướng dẫn câu lệnh trong chương trình.

Vấn đề ở đây là chương trình bạn viết sẽ không đúng theo ý muốn của bạn. Ý nghĩa của chương trình bị sai lệch. Việc phát hiện các lỗi ngữ nghĩa đôi lúc rất khó vì bạn cần phải quay ngược lại và nhìn vào kết quả của chương trình để phán đoán xem bản thân chương trình đã thực hiện những gì.

Gỡ lỗi thử nghiệm

Một trong những kĩ năng quan trọng nhất mà bạn sẽ học được, đó là gỡ lỗi. Mặc dù đôi khi rất khó chịu, nhưng việc gỡ lỗi rất cần trí tuệ, chứa đầy thử thách và là một phần thú vị trong lập trình.

Theo một nghĩa nào đó, gỡ lỗi giống như việc điều tra tội phạm. Bạn có trong tay các manh mối, phải suy luận ra các quá trình và sự kiện dẫn đến những hậu quả đang chứng kiến.

Việc gỡ lỗi cũng giống như khoa học thực nghiệm. Mỗi khi có ý kiến về nguyên nhân dẫn đến lỗi sai, bạn sửa chữa chương trình và thực hiện lại. Nếu giả thiết của bạn là đúng thì bạn thu được kết quả của công việc sửa chữa, đồng thời tiến một bước gần hơn tới chương trình đúng. Còn nếu giả thiết là sai thì bạn cần đề ra một giả thiết mới. Sherlock Holmes đã chỉ ra, "Khi bạn đã loại trừ tất cả những điều không thể thì những gì còn lại, dù có mập mờ đến đâu, chính là sự thật". (A. Conan Doyle, *Dấu của bộ tứ*)

Đối với một số người, việc lập trình và gỡ lỗi là giống nhau. Đó là vì lập trình chính là quá trình gỡ lỗi dần dần đến khi bạn có được chương trình mong muốn. Ý tưởng ở đây là bạn nên bắt đầu với một chương trình có một tính năng *nhỏ* nào đó và thực hiện các chỉnh sửa, gỡ lỗi trong suốt quá trình, đến khi bạn có được một chương trình hoàn thiện.

Chẳng hạn, Linux là một hệ điều hành bao gồm hàng nghìn dòng lệnh, nhưng nó chỉ bắt đầu từ một chương trình đơn giản do Linus Torvalds dùng để khám phá chip Intel 80386. Theo Larry Greenfield thì "Một trong những dự án trước đó của Linus là một chương trình có nhiệm vụ chuyển từ việc in AAAA thành BBBB. Sau đó nó dần trở thành Linux". (*The Linux Users' Guide* Beta Version 1 / *Hướng dẫn sử dụng Linux*, phiên bản Beta 1).

Các chương tiếp sau đây sẽ nói thêm về việc gỡ lỗi và các vấn đề thực tế trong lập trình.

Ngôn ngữ hình thức và ngôn ngữ tự nhiên

Ngôn ngữ tự nhiên được mọi người dùng để giao tiếp, ví dụ Tiếng Anh, Tiếng Tây Ban Nha, Tiếng Pháp. Chúng tự do phát triển mà không định theo khuôn mẫu với bất kì mục đích nào (mặc dù có một số trật tự chẳng hạn như ngữ pháp);

Ngôn ngữ hình thức được con người thiết kế để ứng dụng trong những lĩnh vực riêng. Chẳng hạn, kí hiệu toán học chính là một ngôn ngữ hình thức rất hữu dụng để biểu diễn mối quan hệ giữa những biến lượng và con số. Trong hoá học, một loại ngôn ngữ hình thức khác được dùng để biểu diễn cấu trúc hoá học của các phân tử. Và quan trọng nhất:

> **Ngôn ngữ lập trình là những ngôn ngữ hình thức được thiết kế phục vụ mục đích diễn tả quá trình tính toán.**

Các ngôn ngữ hình thức thường có quy định rất chặt chẽ về cú pháp. Chẳng hạn, 3 + 3 = 6 là một biểu thức toán học đúng, nhưng 3 + = 3\#6 thì không. H_2O là một công thức hoá học đúng về cú pháp, còn $_2Zz$ thì không. Các quy tắc cú pháp có hai biểu hiện, về các **nguyên tố** và cấu trúc. Nguyên tố là các thành phần cơ sở của ngôn ngữ, chẳng hạn, các từ, các con số, và các nguyên tố hoá học. Trong ví dụ nêu trên, 3 + = 3\#6 có lỗi sai vì **#** không phải là một nguyên tố hợp lệ trong toán học. Tương tự như vậy, $_2Zz$ không hợp lệ vì không có nguyên tố hoá học nào có kí hiệu là *Zz*.

Loại lỗi cú pháp thứ hai thuộc về dạng cấu trúc của một mệnh đề; nghĩa là cách sắp xếp các nguyên tố. Mệnh đề 3 + = 3\#6 không hợp lệ là vì mặc dù + và = đều là các nguyên tố đúng, nhưng chúng không thể đứng liền kề nhau. Tương tự như vậy, trong một công thức hoá học thì chỉ số phải được đặt sau tên nguyên tố chứ không phải đặt trước.

> Hãy viết một câu có cấu trúc đúng nhưng có chứa những từ (nguyên tố) không đúng. Viết một câu khác trong đó tất cả các từ (nguyên tố) đều đúng nhưng cấu trúc lại không đúng.

Mỗi khi đọc một câu trong ngôn ngữ tự nhiên, hoặc trong ngôn ngữ hình thức, bạn cần hình dung được cấu trúc của câu đó là gì (mặc dù với ngôn ngữ tự nhiên thì việc làm này được thực hiện một cách vô thức). Quá trình này được gọi là **phân tách**.

Chẳng hạn, khi nghe câu "Đồng xu rơi", bạn cần hiểu được "đồng xu" là chủ ngữ còn "rơi" là vị ngữ. Một khi đã phân tích được, bạn hiểu được câu đó nói gì, tức là nắm được ý nghĩa của câu. Giải thiết rằng bạn biết được nghĩa của từng từ riêng biệt (đồng xu, và rơi), bạn sẽ hiểu được hàm ý chung của câu này.

Mặc dù ngôn ngữ hình thức và ngôn ngữ tự nhiên có nhiều đặc điểm chung—nguyên tố, cấu trúc, cú pháp, và ngữ nghĩa—nhưng chúng có một số khác biệt:

tính chính xác:
: Ngôn ngữ tự nhiên chứa đựng sự mập mờ theo nghĩa con người muốn hiểu đúng phải có suy luận tuỳ từng ngữ cảnh. và có thêm các thông tin khác để bổ sung. Các ngôn ngữ hình thức được thiết kế gần như rõ ràng tuyệt đối, tức là mỗi mệnh đề chỉ có đúng một nghĩa, bất kể ngữ cảnh như thế nào.

tính gọn gàng:
: Để loại trừ sự mập mờ và tránh gây hiểu nhầm, ngôn ngữ tự nhiên cần dùng đến nhiều nội dung bổ trợ làm dài thêm nội dung. Trái lại, các ngôn ngữ hình thức có nội dung gọn gàng đến mức tối thiểu.

tính phi văn phong:
: Các ngôn ngữ tự nhiên có chứa nhiều thành ngữ và ẩn dụ. Khi ai đó nói "Đồng xu rơi", có thể tại đó không có đồng xu nào và cũng chẳng có gì vừa rơi.[1] Còn các ngôn ngữ hình thức luôn luôn có nghĩa đúng theo những gì được viết ra.

Chúng ta dùng ngôn ngữ tự nhiên ngay từ thủa nhỏ, nên thường có một thời gian khó khăn ban đầu khi làm quen với ngôn ngữ hình thức. Về phương diện nào đó, sự khác biệt giữa ngôn ngữ hình thức và ngôn ngữ tự nhiên cung như khác biệt giữa thơ ca và văn xuôi, dù hơn thế nữa.

Thơ ca:
: Các từ được dùng với cả chức năng âm điệu bên cạnh chức năng ý nghĩa, và toàn bộ bài thơ/ca tạo ra hiệu quả cảm xúc. Luôn mang tính không rõ ràng, thậm chí còn là chủ định của tác giả.

Văn xuôi:
: Coi trọng ý nghĩa của câu chữ hơn, trong đó phải kể đến vai trò của cấu trúc đối với việc diễn đạt ý nghĩa. Văn xuôi dễ phân tích ngữ nghĩa hơn so với thơ ca nhưng vẫn còn yếu tố không rõ ràng.

Chương trình:
: Ý nghĩa của một chương trình máy tính là rõ ràng và được diễn đạt hoàn toàn thông qua câu chữ, theo đó ta có thể hiểu được trọn ven bằng cách phân tích các từ ngữ (nguyên tố) và cấu trúc.

Khi đọc chương trình (hoặc một ngôn ngữ hình thức nào khác) bạn nên làm như sau. Trước hết, hãy nhớ rằng ngôn ngữ hình thức cô đọng hơn ngôn ngữ tự nhiên, nên phải mất nhiều thời gian để đọc hơn. Mặt khác, cấu trúc cũng rất quan trọng, do đó không nên chỉ đọc qua một lượt từ trên xuống dưới. Bạn cần phải học cách phân tách ngôn ngữ trong trí óc, nhận diện các nguyên tố và diễn giải cấu trúc. Cuối cùng, những chi tiết đóng vai trò quan trọng. Các lỗi dù là nhỏ nhất trong cách viết các từ hoặc dấu câu trong ngôn ngữ hình thức sẽ có thể gây ra khác biệt lớn về ý nghĩa.

Chương trình đầu tiên

Theo thông lệ, chương trình đầu tiên mà bạn viết theo một ngôn ngữ lập trình mới có tên gọi là "Hello, World!" vì tất cả những gì nó thực hiện chỉ là làm hiện ra dòng chữ "Hello, World!" Một chương trình như vậy trong Python được viết như sau:

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

Đây là ví dụ về một **lệnh print**[2], vốn chẳng in gì ra giấy. Nó chỉ hiển thị một giá trị trên màn hình. Trong trường hợp này, kết quả là dòng chữ

```
Hello, World!
```

Cặp dấu nháy đơn trong đoạn chương trình có nhiệm vụ đánh dấu các điểm đầu và cuối của đoạn chữ cần hiển thị; chúng sẽ không xuất hiện trong kết quả.

Người ta có thể đánh giá chất lượng của một ngôn ngữ lập trình bằng độ đơn giản của chương trình "Hello, World!". Theo tiêu chuẩn này, Python xứng đáng đạt điểm cao nhất.

Gỡ lỗi

Nếu có thể đọc cuốn sách này trước máy tính thì rất tốt vì bạn sẽ thử được tất cả các ví dụ trong quá trình đọc. Bạn có thể chạy phần lớn các ví dụ ở chế độ tương tác, nhưng nếu viết mã lệnh trong một file văn lệnh thì sẽ dễ thực hiện các điều chỉnh về sau này.

Mỗi khi thử nghiệm một đặc tính mới cho chương trình, bạn nên phạm lỗi. Chẳng hạn, trong chương trình "Hello, world!", điều gì sẽ xảy ra nếu bạn bỏ bớt một trong hai dấu nháy? Và nếu bỏ cả hai dấu nháy? Nếu bạn viết sai chữ `print`?

Kiểu thử nghiệm này sẽ giúp bạn nhớ những gì bạn đã đọc; nó cũng giúp cho công việc gỡ lỗi, vì lúc đó bạn sẽ biết rằng thông báo lỗi ngụ ý gì. Do đó tốt hơn là cố ý phạm lỗi ngay từ lúc này còn hơn là để sau này vô tình mắc lỗi.

Đôi khi việc lập trình, và đặc biệt là gỡ lỗi, đem đến những cảm xúc mạnh. Nếu bạn đang đánh vật với một lỗi rất khó, bạn có thể nổi xung, đầu hàng hoặc bối rối.

Đã có những chứng cứ cho thấy con người phản ứng tự nhiên lại với máy tính như thể chúng là những người thực.[3]. Khi chúng hoạt động trôi chảy, ta coi chúng như người bạn; và khi chúng rất cứng đầu hoặc thô lỗ, chúng ta phản ứng với chúng như thể với hạng người mang những tính đó.

Chuẩn bị tiếp nhận những phản ứng này có thể giúp bạn biết cách vượt qua chúng. Một cách làm là nghĩ về máy tính như một nhân viên với các ưu điểm năng lực nhất định, như tốc độ và độ chính xác, nhưng kèm theo những nhược điểm riêng, như thiếu sự đồng cảm và thiếu khả năng nắm bắt bức tranh tổng thể.

Còn bạn có vai trò là một người quản lý tốt: hãy tìm cách tận dụng ưu điểm và khắc phục những nhược điểm. Và tìm ra những cách điều khiển cảm xúc khi giải quyết vấn đề, không để cho những phản ứng của bản thân làm ảnh hưởng đến khả năng làm việc hiệu quả.

Học cách gỡ lỗi có thể dễ gây bực bội, nhưng đó lại là kỹ năng rất quý báu và cần thiết cho nhiều hoạt động khác ngoài lập trình. Ở cuối mỗi chương sách đều có một mục gỡ lỗi, như mục này, trong đó tôi muốn chia sẻ những ý kiến bản thân về việc gỡ lỗi. Hi vọng nó sẽ giúp bạn!

Thuật ngữ

giải quyết vấn đề:
: Quá trình thiết lập bài toán, tìm lời giải, và biểu diễn lời giải.

ngôn ngữ bậc cao:
: Ngôn ngữ lập trình như Python được thiết kế nhằm mục đích để con người dễ đọc và viết.

ngôn ngữ bậc thấp:
: Ngôn ngữ lập trình được thiết kế nhằm mục đích để máy tính dễ thực hiện; còn gọi là "ngôn ngữ máy" hoặc "hợp ngữ".

tính khả chuyển:

Đặc tính của chương trình mà có thể chạy trên nhiều loại máy tính khác nhau.

thông dịch:
: Thực hiện chương trình được viết bằng ngôn ngữ bậc cao bằng cách dịch nó theo từng dòng một.

biên dịch:
: Dịch một lượt toàn bộ chương trình viết bằng ngôn ngữ bậc cao sang ngôn ngữ bậc thấp, để chuẩn bị thực hiện sau này.

mã nguồn:
: Chương trình ở dạng ngôn ngữ bậc cao trước khi được biên dịch.

mã đối tượng:
: Sản phẩm đầu ra của trình biên dịch sau khi nó đã dịch chương trình.

chương trình chạy:
: Tên khác đặt cho mã đối tượng đã sẵn sàng được thực hiện.

dấu nhắc:
: Các kí tự được hiển thị bởi trình thông dịch nhằm thể hiện rằng nó đã sẵn sàng nhận đầu vào từ phía người dùng.

văn lệnh:
: Chương trình được lưu trong file (thường chính là chương trình sẽ được thông dịch).

chế độ tương tác:
: Cách dùng trình thông dịch Python thông qua việc gõ các câu lệnh và biểu thức vào chỗ dấu nhắc.

chế độ văn lệnh:
: Cách dùng trình thông dịch Python để đọc và thực hiện các câu lệnh có trong một văn lệnh.

chương trình:
: Danh sách những chỉ dẫn thực hiện tính toán.

thuật toán:
: Quá trình tổng quát để giải một lớp các bài toán.

lỗi:
: Lỗi trong chương trình.

gỡ lỗi:
: Quá trình dò tìm và gỡ bỏ cả ba kiểu lỗi trong lập trình.

cú pháp:
: Cấu trúc của một chương trình.

lỗi cú pháp:
: Lỗi trong chương trình mà làm cho quá trình phân tách không thể thực hiện được (và hệ quả là không thể biên dịch được).

biệt lệ:
: Lỗi được phát hiện khi chương trình đang chạy.

ngữ nghĩa:
: Ý nghĩa của chương trình.

lỗi ngữ nghĩa:
: Lỗi có trong chương trình mà khiến cho chương trình thực hiện công việc ngoài ý định của người viết.

ngôn ngữ tự nhiên:
: Ngôn ngữ bất kì được con người dùng, được trải qua sự tiến hóa tự nhiên.

ngôn ngữ hình thức:
: Ngôn ngữ bất kì được con người thiết kế nhằm mục đích cụ thể, như việc biểu diễn các ý tưởng toán học hoặc các chương trình máy tính; tất cả các ngôn ngữ lập trình đều là ngôn ngữ hình thức.

nguyên tố:

Một trong những thành phần cơ bản trong cấu trúc cú pháp của một chương trình, tương đương với một từ trong ngôn ngữ tự nhiên.

phân tách:
: Việc kiểm tra một chương trình và phân tích cấu trúc cú pháp.

lệnh print:
: Chỉ thị để khiến trình thông dịch Python hiển thị một giá trị lên màn hình.

Bài tập

Dùng một trình duyệt web để truy cập trang web của Python `python.org`. Trang này bao gồm thông tin về Python và các kết nối đến những trang khác có liên quan đến Python; nó cũng giúp bạn tìm kiếm trong tài liệu về Python.

Chẳng hạn, nếu bạn nhập vào `print` ở cửa sổ tìm kiếm thì đường kết nối thứ nhất sẽ xuất hiện như là tài liệu hướng dẫn câu lệnh `print`. Đến đây, có thể bạn không hiểu những gì trong đó viết, nhưng biết cách tìm ra nó là điều tốt nhất.

Khởi động trình thông dịch Python và gõ vào `help()` để khởi động ứng dụng hỗ trợ phần mềm. Hoặc bạn có thể gõ `help('print')` để biết thông tin về câu lệnh `print`.

Nếu như ví dụ này không thực hiện được, có thể bạn sẽ cần phải cài đặt riêng bộ tài liệu về Python hoặc thiết lập một biến môi trường; cụ thể điều này còn phụ thuộc vào hệ điều hành và phiên bản Python mà bạn đang dùng.

Hãy khởi động trình thông dịch Python và dùng nó như một máy tính tay. Cú pháp của Python về các phép tính cũng giống như các kí hiệu toán học thông dụng. Chẳng hạn các dấu `+`, `-`, và `/` để chỉ các phép tính cộng, trừ, và chia, như bạn trông đợi. Kí hiệu cho phép nhân là `*`.

Nếu bạn chạy thi 10 km trong vòng 43 phút 30 giây thì thời gian trung bình mà để bạn chạy được một dặm là bao nhiêu? Tốc độ trung bình của bạn là bao nhiêu dặm mỗi giờ? (Gợi ý: một dặm bằng 1.61 km).

---

1. Thành ngữ tiếng Anh này nghĩa là ai đó đã nhận ra điều gì đó sau một thoáng bối rối. ↩
2. Trong Python 3.0, `print` là một hàm, không phải câu lệnh, và do đó cú pháp sẽ là `print('Hello, World!')`. Không lâu nữa chúng ta sẽ làm quen với các hàm! ↩
3. Xem Reeves and Nass, { The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places}. ↩

Chương 2: Biến, biểu thức và câu lệnh

Giá trị và kiểu

Giá trị là một trong những cái cơ bản mà chương trình cần dùng đến, chẳng hạn như một chữ cái hoặc một con số. Các giá trị mà ta đã thấy đến giờ bao gồm `1`, `2`, và `'Hello, World!'`.

Các giá trị này thuộc về hai **kiểu** khác nhau: `2` là một số nguyên, còn `'Hello, World!'` là một **chuỗi**, được gọi như vậy vì nó là một chuỗi các kí tự ghép lại với nhau. Bạn (và trình thông dịch) có thể nhận ra các chuỗi vì chúng được đặt trong cặp dấu nháy.

Câu lệnh print cũng có tác dụng với các số nguyên.

```
>>> print 4
4
```

Nếu bạn không chắc rằng kiểu của một giá trị là gì, trình thông dịch có thể cho bạn biết.

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)
<type 'int'>
```

Thật không ngạc nhiên rằng chuỗi kí tự (*string*) thuộc về kiểu `str` và các số nguyên (*integer*) thuộc về kiểu `int`. Điều ít hiển nhiên là các số có phần thập phân thuộc về một kiểu có tên là `float`, vì những số này được biểu diễn dưới một dạng được gọi là **dấu phẩy động** (*floating-point*) [ta sẽ tạm gọi là số có phần thập phân trong cuốn sách này].

```
>>> type(3.2)

<type 'float'>
```

Thế còn các giá trị như `'17'` và `'3.2'`? Trông chúng giống như số, nhưng chúng được đặt trong cặp dấu nháy như các chuỗi.

```
>>> type('17')
<type 'str'>
>>> type('3.2')
<type 'str'>
```

Vậy chúng là các chuỗi.

Khi bạn gõ vào những số nguyên lớn, có thể bạn muốn dùng những dấu phẩy để nhóm từng lớp ba chữ số lại với nhau, như `1,000,000`. Đây không phải là một số nguyên hợp lệ trong Python, nhưng vẫn đúng về mặt cú pháp:

```
>>> print 1,000,000
1 0 0
```

À, đó không phải là điều chúng ta mong muốn! Python dịch `1,000,000` như một danh sách các số nguyên được phân cách bởi các dấu phẩy, và khi in ra thì đặt dấu cách giữa các số này.

Đây là ví dụ đầu tiên mà chúng ta thấy một lỗi ngữ nghĩa: đoạn mã chạy mà không có lỗi được thông báo, nhưng nó không thực hiện điều "đúng".

Biến

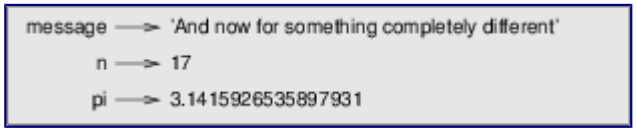
Một trong những tính năng mạnh nhất của một ngôn ngữ lập trình là khả năng thao tác với các **biến**. Biến là một tên gọi tham chiếu đến một giá trị.

Một **lệnh gán** tạo ra biến mới và đặt giá trị cho nó:

```
>>> message = 'And now for something completely different'
>>> n = 17
>>> pi = 3.1415926535897931
```

Ví dụ này có ba lệnh gán. Lệnh thứ nhất gán một chuỗi cho một biến có tên là `message`; lệnh thứ hai gán số nguyên `17` cho `n`; lệnh thứ ba gán giá trị (gần đúng) của π cho `pi`.

Một cách chung để biểu diễn các biến trên giấy là viết ra tên kèm theo một mũi tên chỉ đến giá trị của biến. Dạng hình vẽ này được gọi là **sơ đồ trạng thái** vì nó cho thấy trạng thái của mỗi biến hiện tại là như thế nào (hãy hình dung nó như trạng thái trí não của biến đó). Sơ đồ dưới đây cho thấy kết quả của ví dụ trước:



Để hiển thị giá trị của một biến, bạn có thể dùng lệnh print:

```
>>> print n
17

>>> print pi
3.14159265359
```

Kiểu của một biến là kiểu của giá trị mà biến đó tham chiếu đến.

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

Nếu bạn gõ vào một số nguyên mà bắt đầu với chữ số 0, có thể bạn sẽ nhận được một thông báo lỗi khó hiểu:

```
>>> zipcode = 02492
                   ^
SyntaxError: invalid token
```

Với các số khác có vẻ như mọi việc bình thường, nhưng kết quả rất quái lạ:

```
>>> zipcode = 02132
>>> print zipcode
1114
```

Bạn có thể hình dung điều gì đang xảy ra không? Gợi ý: hãy in các giá trị `01`, `010`, `0100` và `01000`.

Tên biến và từ khoá

Thông thường các lập trình viên chọn tên biến có nghĩa— tự nó nói lên rằng biến được dùng vào việc gì.

Tên biến có độ dài tuỳ ý. Chúng có thể gồm cả chữ cái và số, nhưng bắt buộc phải bắt đầu bằng một chữ cái. Dùng các chữ in cũng được, nhưng tốt nhất là bạn nên bắt đầu tên biến với chữ thường (sau này bạn sẽ biết tại sao).

Dấu gạch dưới (_) có thể xuất hiện trong một tên. Nó thường được dùng trong các tên gồm có nhiều từ, như `my_name` hoặc `airspeed_of_unladen_swallow`.

Nếu bạn đặt một tên biến không hợp lệ, sẽ có lỗi cú pháp:

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

>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` không hợp lệ vì nó không bắt đầu bằng một chữ cái. `more@` không hợp lệ vì nó có chứa một kí tự không hợp lệ, `@`. Nhưng còn `class` tại sao lại sai?

Hoá ra vì `class` là một trong những **từ khoá** của Python. Trình thông dịch sử dụng từ khoá để nhận ra cấu trúc của chương trình, và chúng không thể được dùng để đặt tên biến.

Python có 31 từ khoá[1]:

```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

Bạn có thể ghi lại danh sách trên đây. Nếu trình thông dịch phàn nàn về một tên biến mà bạn không biết tại sao, hãy tra xem nó có nằm trong danh sách này không.

Câu lệnh

Câu lệnh là một đơn vị của mã lệnh mà trình thông dịch Python có thể thực hiện được. Chúng ta đã gặp hai loại câu lệnh: print và lệnh gán.

Khi bạn gõ một câu lệnh ở trong chế độ tương tác, trình thông dịch sẽ thực hiện nó và hiển thị kết quả, nếu có.

Một văn lệnh thường gồm một loạt các câu lệnh hợp thành. Nếu có hơn một câu lệnh thì kết quả của từng câu lệnh sẽ lần lượt được xuất hiện khi câu lệnh đó được thực thi.

Chẳng hạn, đoạn văn lệnh

```
print 1
x = 2
print x
```

cho ta kết quả

```
1
2
```

Câu lệnh gán không tạo ra kết quả.

Toán tử và toán hạng

Toán tử là các kí hiệu đặc biệt để biểu diễn các phép tính như cộng và nhân. Toán tử được áp dụng cho các giá trị được gọi là **toán hạng**.

Các toán tử `+`, `-`, `*`, `/` và `**` biểu thị phép cộng, trừ, nhân, chia, và luỹ thừa như trong ví dụ sau:

```
20+32   hour-1   hour*60+minute   minute/60   5**2   (5+9)*(15-7)
```

Trong một số ngôn ngữ lập trình khác, ^ được dùng để tính luỹ thừa, nhưng với Python đó là một toán tử tính cho bit có tên là XOR. Tôi sẽ không trình bày các toán tử để tính cho bit trong sách này, nhưng bạn có thể đọc thêm về chúng ở trang wiki.python.org/moin/BitwiseOperators.

Toán tử chia có thể không thực hiện điều mà bạn mong đợi:

```
>>> minute = 59
>>> minute/60
0
```

Giá trị của `minute` là 59, và trong đại số thông thường thì 59 chia cho 60 bằng 0.98333, chứ không phải 0. Lí do của sự khác biệt ở đây là Python đã thực hiện phép **chia làm tròn xuống**[2].

Khi cả hai toán hạng đều là số nguyên, kết quả cũng sẽ là một số nguyên; phép chia làm tròn xuống cắt bỏ phần thập phân, vì vậy trong ví dụ này kết quả được làm tròn xuống 0.

Nếu một trong hai toán hạng là một số có phần thập phân, Python sẽ thực hiện phép chia thập phân, và kết quả là một số thập phân (`float`):

```
>>> minute/60.0
0.98333333333333328
```

Biểu thức là một tổ hợp các giá trị, biến, và toán tử. Một giá trị bản thân nó cũng được coi như là một biểu thức, và một biến cũng vậy; vì thế tất cả những cái dưới đây đều là các biểu thức hợp lệ (giả sử rằng biến `x` đã được gán một giá trị):

```
17
x
x + 17
```

Nếu bạn gõ một biểu thức ở trong chế độ tương tác, trình thông dịch sẽ **định lượng** nó và hiển thị kết quả:

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

Nhưng trong một văn lệnh, một biểu thức bản thân nó không có tác dụng gì cả! Đây là một điểm dễ gây nhầm lẫn cho người mới học.

> Hãy gõ những câu lệnh dưới đây vào trình thông dịch Python để xem chúng có tác dụng gì:
>
> ```
> 5
> ```

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

Bây giờ đưa chính các câu lệnh đó vào trong một văn lệnh và chạy nó. Kết quả là gì? Sửa lại văn lệnh bằng cách thay mỗi biểu thức bằng một câu lệnh print tương ứng và chạy lại.

Thứ tự thực hiện

Khi trong biểu thức có nhiều hơn một toán tử, thứ tự định lượng sẽ tuân theo **quy tắc ưu tiên**. Với các toán tử toán học, Python dựa vào quy ước chung trong môn toán. Chữ viết tắt **PEMDAS** là một cách nhớ quy tắc này:

- Cặp ngoặc đơn (**P**arentheses) có thứ tự ưu tiên cao nhất và có thể được dùng để buộc việc lượng giá một biểu thức theo đúng thứ tự mà bạn mong muốn. Vì các biểu thức trong cặp ngoặc đơn được lượng giá trước tiên, `2 * (3-1)` bằng 4, và `(1+1)**(5-2)` bằng 8. Bạn cũng có thể dùng cặp ngoặc đơn để biểu thức trở nên dễ đọc, như với `(minute * 100) / 60`, ngay cả khi không có nó thì kết quả cũng không đổi.
- Phép luỹ thừa (**E**xponentiation) có thứ tự ưu tiên kế tiếp, vì vậy `2**1+1` bằng 3 chứ không phải 4, và `3*1**3` bằng 3 chứ không phải 27.
- Các phép nhân (**M**ultiplication) và chia (**D**ivision) có cùng độ ưu tiên, cao hơn các phép cộng (**A**ddition) và trừ (**S**ubtraction), hai phép sau cũng có cùng độ ưu tiên. Vì vậy `2*3-1` bằng 5 chứ không phải 4, và `6+4/2` bằng 8 chứ không phải 5.
- Các toán tử có cùng độ ưu tiên được định lượng từ trái sang phải. Vì vậy, trong biểu thức `degrees / 2 * pi`, phép chia được thực hiện trước và kết quả sẽ được nhân với `pi`. Để chia cho 2π, bạn có thể dùng cặp ngoặc đơn hoặc viết `degrees / 2 / pi`.

Các thao tác với chuỗi

Nói chung, bạn không thể thực hiện các phép toán đối với chuỗi, ngay cả khi chuỗi trông giống như những con số. Vì vậy các biểu thức sau đây đều không hợp lệ:

```
'2'-'1'    'eggs'/'easy'    'third'*'a charm'
```

Toán tử + có tác dụng với chuỗi, nhưng nó có thể sẽ không hoạt động theo cách bạn mong đợi: nó có nhiệm vụ **nối**, nghĩa là ghép nối tiếp các chuỗi lại với nhau. Chẳng hạn:

```
first = 'throat'
second = 'warbler'
print first + second
```

Kết quả của chương trình trên là `throatwarbler`.

Toán tử * cũng có tác dụng đối với chuỗi; nó có nhiệm vụ lặp lại. Chẳng hạn, `'Spam'*3` là `'SpamSpamSpam'`. Nếu một trong các toán hạng là chuỗi, toán hạng còn lại phải là một số nguyên.

Công dụng của + và * cũng có nghĩa tương tự như với phép cộng và phép nhân. Giống như việc `4*3` tương đương với `4+4+4`, chúng ta trông đợi `'Spam'*3` tương đương `'Spam'+'Spam'+'Spam'`, và thật vậy. Mặt khác, có một sự khác biệt đáng kể giữa kết nối và lặp chuỗi so với các phép cộng và nhân số nguyên. Bạn có thể nghĩ ra một thuộc tính của phép cộng mà phép nối chuỗi không có không?

Chú thích

Khi chương trình trở nên lớn và phức tạp hơn, chúng cũng đồng thời khó đọc hơn. Các ngôn ngữ hình thức rất cô đặc, và nhìn vào một đoạn mã lệnh ta thường khó hình dung ra nó để làm gì, hoặc tại sao.

Vì lí do này, ta nên thêm các ghi chú vào chương trình để giải thích rằng chương trình làm gì bằng ngôn ngữ tự nhiên. Các ghi chú này được gọi là **chú thích**, và đều bắt đầu bằng kí hiệu #:

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

Trong trường hợp này, chú thích xuất hiện riêng trên một dòng. Bạn cũng có thể đặt chú thích ở cuối một dòng:

```
percentage = (minute * 100) / 60     # percentage of an hour
```

Mọi thứ từ dấu # về cuối dòng đều được bỏ qua—nó không làm ảnh hưởng đến tác dụng của chương trình.

Các chú thích rất cần thiết khi chúng đưa thông tin về những tính năng không dễ thấy của đoạn mã lệnh. Thường ta có thể coi rằng người đọc đều hình dung được mã lệnh làm *làm gì*; và tốt hơn là hãy dùng chú thích vào việc giải thích *tại sao*.

Trong đoạn mã lệnh sau, chú thích là thừa và vô dụng:

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

Chú thích sau chứa thông tin hữu dụng hơn mà mã lệnh không có:

```
v = 5     # velocity in meters/second.
```

Việc đặt tên biến hợp lý có thể làm giảm nhu cầu dùng chú thích, nhưng những tên biến dài có thể làm các biểu thức khó đọc, vì vậy nó luôn có sự được–mất giữa hai mặt.

Gỡ lỗi

Cho đến giờ lỗi cú pháp mà bạn thường gặp nhất có lẽ là tên biến không hợp lệ, như `class` và `yield`, vốn trùng với các từ khoá, hoặc `odd~job` và `US$`, vốn có chứa các kí tự không hợp lệ.

Nếu bạn đặt một dấu cách về phía trước một tên biến, Python sẽ nghĩ rằng đó là hai toán hạng mà không kèm theo toán tử nào:

```
>>> bad name = 5
SyntaxError: invalid syntax
```

Với các lỗi cú pháp, dòng chữ thông báo lỗi không giúp được gì nhiều. Những thông báo lỗi thường gặp nhất là `SyntaxError: invalid syntax` (cú pháp không hợp lệ) và `SyntaxError: invalid token` (nguyên tố không hợp lệ), cả hai đều không mang thông tin đáng kể.

Loại lỗi khi chạy chương trình mà có lẽ bạn thường gặp nhất là "use before def"; nghĩa là bạn đã thử dùng một biến trước khi gán cho nó một giá trị. Điều này có thể xảy ra nếu bạn viết nhầm tên biến:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Các tên biến đều phân biệt chữ in và chữ thường, vì vậy, `LaTeX` khác với `latex`.

Cho đến giờ, nguyên do thường gặp nhất gây ra lỗi ngữ nghĩa là thứ tự thực hiện phép tính. Chẳng hạn, để định lượng 1/ 2π, có thể bạn đã toan viết

```
>>> 1.0 / 2.0 * pi
```

Nhưng phép chia lại được thực hiện trước, vì vậy bạn sẽ được π/2, vốn không giống kết quả đúng! Vì Python không có cách nào đoán biết ý của bạn khi viết chương trình nên trong trường hợp này bạn không thấy có thông báo lỗi; bạn chỉ thu được đáp số sai.

Thuật ngữ

giá trị:
: Một trong những đơn vị cơ bản của dữ liệu, cũng như số hoặc chuỗi, mà chương trình thao tác với.

kiểu:
: Loại riêng của các giá trị. Những kiểu mà ta đã gặp bao gồm kiểu số nguyên (`int`), số có phần thập phân (`float`), và chuỗi (`str`).

số nguyên:
: Kiểu dùng để biểu diễn loại số tương ứng.

số có phần thập phân:
: Kiểu dùng để biểu diễn loại số tương ứng.

chuỗi:
: Kiểu dùng để biểu diễn một danh sách các kí tự.

biến:
: Tên được tham chiếu đến một giá trị.

câu lệnh:
: Đoạn mã biểu diễn một lệnh hoặc một hành động. Cho đến giờ, các câu lệnh mà ta đã gặp gồm có lệnh gán và lệnh print.

lệnh gán:
: Lệnh để gán một giá trị cho một biến.

sơ đồ trạng thái:
: Đồ thị biểu diễn một tập hợp các biến và các giá trị mà chúng tham chiếu tới.

từ khoá:
: Từ dành riêng cho trình biên dịch để phân tách một chương trình; bạn không thể dùng những từ khoá như `if`, `def`, và `while` để đặt tên biến.

toán tử:
: Kí hiệu đặc biệt để biểu diễn một phép tính đơn nhất như cộng, nhân, hoặc nối chuỗi.

toán hạng:

Một trong những giá trị mà toán tử thực hiện với.

phép chia làm tròn xuống:
: Phép toán chia hai số và cắt bỏ phần thập phân.

biểu thức:
: Tập hợp các biến, toán tử, và giá trị nhằm biểu diễn một giá trị kết quả duy nhất.

định lượng:
: Giản hoá một biểu thức bằng cách thực hiện các phép tính nhằm thu được một giá trị duy nhất.

quy tắc ưu tiên:
: Tập hợp các quy tắc chi phối thứ tự mà những biểu thức bao gồm nhiều toán tử và toán hạng được định lượng.

nối:
: Ghép nối tiếp hai toán hạng.

chú thích:
: Thông tin trong một chương trình; thông tin này có ích đối với các lập trình viên khác (hoặc người khác đọc mã nguồn) nhưng không làm ảnh hưởng đến việc thực hiện chương trình.

Bài tập

Giả sử ta thực hiện những câu lệnh gán dưới đây:

```
width = 17
height = 12.0
delimiter = '.'
```

Với mỗi biểu thức sau, hãy cho biết giá trị của biểu thức cùng với kiểu của giá trị đó.

1. `width/2`
2. `width/2.0`
3. `height/3`
4. `1 + 2 * 5`
5. `delimiter * 5`

Dùng trình thông dịch Python để kiểm tra kết quả.

Tập luyện cách dùng trình thông dịch Python thay cho máy tính tay:

1. Thể tích của một hình cầu có bán kính r là $4/3\ \pi r^3$. Thể tích của một hình cầu có bán kính bằng 5 là bao nhiêu? Gợi ý: 392.6 là đáp số sai!
2. Coi rằng giá bìa của một cuốn sách là $24.95, nhưng các hiệu sách được mua giảm giá 40%. Tiền vận chuyển là $3 với cuốn sách đầu và 75 xu với mỗi cuốn sách thêm. Tổng số tiền bán sỉ cho 60 bản sách là bao nhiêu?
3. Nếu tôi rời nhà lúc 6:52 sáng và chạy chậm 1 dặm (mỗi dặm hết 8:15), sau đó chạy mức trung bình 3 dặm (mỗi dặm hết 7:12) và tiếp tục chạy chậm 1 dặm, thì

lúc mấy giờ tôi sẽ về đến nhà để ăn sáng?

---

1. Trong Python 3.0, `exec` không còn là một từ khoá, nhưng lại có thêm từ khoá `nonlocal`. ↩
2. Trong Python 3.0, kết quả của phép chia này là một số có phần thập phân (`float`). Một toán tử mới `//` thực hiện phép chia làm tròn. ↩

Chương 3: Hàm

Việc gọi các hàm

Trong lập trình, một **hàm** là một nhóm được đặt tên gồm các câu lệnh nhằm thực hiện một nhiệm vụ tính toán cụ thể. Khi định nghĩa hàm, bạn chỉ định tên của nó và tiếp theo là loạt các câu lệnh. Sau này, bạn có thể "gọi" hàm theo tên của nó.

Ta đã gặp một ví dụ của việc **gọi hàm**:

```
>>> type(32)
<type 'int'>
```

Tên của hàm này là `type`. Biểu thức ở trong cặp ngoặc đơn được gọi là **đối số** của hàm. Kết quả của hàm này là kiểu của đối số.

Người ta thường nói một hàm "lấy" một đối số và "trả về" một giá trị. Giá trị này được gọi là **giá trị trả về**.

Các hàm chuyển đổi kiểu

Python cung cấp các hàm dựng sẵn giúp chuyển đổi một giá trị từ kiểu này sang kiểu khác. Hàm `int` lấy bất kì một giá trị nào và chuyển nó thành một số nguyên nếu có thể, còn nếu không được thì thông báo lỗi:

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

`int` có thể chuyển các giá trị số có phần thập phân sang số nguyên, nhưng nó không làm tròn mà chỉ cắt bỏ phần thập phân:

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

`float` chuyển số nguyên và chuỗi sang số có phần thập phân:

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

Cuối cùng, `str` chuyển đổi số của nó sang một chuỗi:

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

Các hàm toán học

Python có một module (mô-đun) toán cung cấp phần lớn các hàm toán học thông dụng. Một **module** là một file trong đó có tập hợp các hàm liên quan với nhau.

Để sử dụng được module, ta cần phải nhập nó bằng lệnh import:

```
>>> import math
```

Câu lệnh này tạo ra một **đối tượng module** có tên là math. Nếu bạn in đối tượng module này, bạn sẽ nhận được thông tin về nó:

```
>>> print math
<module 'math' from '/usr/lib/python2.5/lib-dynload/math.so'>
```

Đối tượng module chứa các hàm và biến được định nghĩa trong module. Để truy cập một trong các hàm đó, bạn phải chỉ định tên của module và tên của hàm, cách nhau bởi một dấu chấm. Cách này được gọi là **kí hiệu dấu chấm**.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)
>>> radians = 0.7
>>> height = math.sin(radians)
```

Ví dụ thứ nhất thực hiện tính lô-ga cơ số 10 của tỉ lệ giữa tín hiệu và nhiễu. module math cũng cung cấp một hàm có tên là `log` có nhiệm vụ tính lô-ga cơ số `e`.

Ví dụ thứ hai tìm sin của `radians`. Ở đây tên của biến số gợi ý rằng `sin` và các hàm lượng giác khác (`cos`, `tan`, v.v.) nhận đối số có đơn vị radian. Để đổi từ độ sang radian, hãy chai cho 360 và nhân với 2π:

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.707106781187
```

Biểu thức `math.pi` lấy biến `pi` từ module math. Giá trị của biến này xấp xỉ với số π, với độ chính xác khoảng 15 chữ số.

Nếu bạn nắm vững lượng giác, bạn có thể kiểm tra lại kết quả trước bằng cách so sánh nó với căn bậc hai của hai chia đôi:

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

Sự kết hợp

Cho đến giờ, chúng ta mới xét đến các thành phần tạo nên chương trình —biến, biểu thức, và câu lệnh—một cách riêng lẻ, mà chưa nói đến việc kết hợp chúng như thế nào.

Một trong những đặc điểm hữu ích của các ngôn ngữ lập trình là chúng cho lấy những thành phần nhỏ và **kết hợp** chúng lại. Chẳng hạn, đối số của một hàm có thể là bất cứ biểu thức nào, bao gồm cả các toán tử đại số:

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

Và thậm chí cả các hàm được gọi:

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

Hầu như bất kì chỗ nào bạn đặt được một giá trị, bạn cũng sẽ thay được vào đó một biểu thức, chỉ với một ngoại lệ: phía bên trái của một câu lệnh gán phải là một tên biến. Tất cả biểu thức nếu đặt ở bên phía trái đó sẽ phạm lỗi cú pháp[1].

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

Thêm vào các hàm mới

Đến bây giờ, chúng ta mới chỉ dùng những hàm có sẵn trong Python, song thật ra có thể tạo ra những hàm mới. Một **định nghĩa hàm** bao gồm việc chỉ định tên của hàm mới và danh sách các câu lệnh cần được thực hiện khi hàm được gọi.

Sau đây là một ví dụ:

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

`def` là một từ khoá để khẳng định rằng đây là một định nghĩa làm. Tên của hàm là `print_lyrics`. Quy tắc đặt tên hàm cũng như đặt tên biến: chữ cái, số và dấu nối là hợp lệ nhưng kí tự đầu tiên không thể là số. Bạn không thể đặt tên hàm giống như một từ khoá, và cũng nên tránh đặt tên hàm và tên biến trùng nhau.

Tiếp theo tên hàm là cặp ngoặc đơn bên trong không có gì, điều đó có nghĩa là hàm này không lấy đối số nào.

Dòng đầu tiên của định nghĩa hàm được gọi là **đoạn đầu**; phần còn lại là **phần thân**. Phần đầu phải được kết thúc bởi dấu hai chấm và phần thân phải được viết thụt đầu dòng. Theo quy ước, khoảng cách thụt vào luôn là bốn dấu cách (xem Mục [Trình soạn thảo]). Phần thân có thể chứa bao nhiêu câu lệnh cũng được.

Các chuỗi trong câu lệnh print được viết trong cặp dấu nháy kép. Cặp dấu nháy đơn và nháy kép có tác dụng như nhau; người ta thường dùng cặp nháy đơn trừ những trường hợp như sau khi có một dấu nháy đơn xuất hiện trong chuỗi.

Nếu bạn gõ định nghĩa hàm vào ở chế độ tương tác, trình thông dịch sẽ in ra các dấu ba chấm (`...`) nhằm cho bạn biết rằng việc định nghĩa hàm chưa hoàn thành:

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

Để kết thúc hàm, bạn phải gõ thêm một dòng trống (điều này không cần thiết trong một văn lệnh).

Việc định nghĩa hàm sẽ tạo ra một biến có cùng tên.

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

Giá trị của `print_lyrics` là một **đối tượng hàm**; nó có kiểu `'function'`.

Cú pháp của lời gọi hàm mới cũng giống như với các hàm dựng sẵn:

```
>>> print_lyrics()
```

```
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Một khi bạn đã định nghĩa hàm, bạn có thể dùng nó trong một hàm khác. Chẳng hạn, để lặp lại điệp khúc vừa rồi, ta có thể viết một hàm có tên là `repeat_lyrics`:

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

Và sau đó gọi `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.
```

Song đó không phải là cách viết một bài hát theo đúng nghĩa.

Định nghĩa và sử dụng

Lấy lại những đoạn câu lệnh từ mục trước, ta được toàn bộ chương trình sau:

```
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()
```

Chương trình này bao gồm hai định nghĩa hàm: `print_lyrics` và `repeat_lyrics`. Các định nghĩa hàm được thực hiện cũng giống như với các câu lệnh khác, nhưng tác dụng của chúng là tạo ra những đối tượng hàm. Các câu lệnh bên trong hàm không được thực hiện cho đến tận khi hàm được gọi, hơn nữa định nghĩa hàm cũng không tạo ra kết quả.

Bạn có thể nghĩ rằng cần phải tạo ra một hàm trước khi có thể thực hiện nó, và quả đúng như vậy. Nói cách khác, định nghĩa hàm phải được thực hiện trước lần gọi hàm đầu tiên.

> Chuyển dòng lệnh cuối cùng của chương trình vừa rồi lên trên cùng, để cho lời gọi hàm xuất hiện trước định nghĩa hàm. Chạy chương trình để xem bạn nhận được thông báo lỗi gì.
>
> Chuyển lời gọi hàm trở lại cuối cùng và chuyển định nghĩa hàm `print_lyrics` xuống dưới định nghĩa hàm `repeat_lyrics`. Lần này khi chạy chương trình, điều gì sẽ xảy ra?

Luồng thực hiện chương trình

Để đảm bảo chắc chắn rằng một hàm đã được định nghĩa trước lần sử dụng đầu tiên, bạn phải biết thứ tự thực hiện các câu lệnh, còn gọi là **luồng thực hiện** chương trình.

Việc thực hiện luôn được bắt đầu với câu lệnh thứ nhất của chương trình. Các câu lệnh được thực

hiện lần lượt từ trên xuống.

Các định nghĩa hàm không làm thay đổi luồng thực hiện chương trình, nhưng cần nhớ rằng các câu lệnh bên trong của hàm không được thực hiện cho đến tận lúc hàm được gọi.

Mỗi lần gọi hàm là một lần rẽ ngoặt luồng thực hiện. Thay vì chuyển sang câu lệnh kế tiếp, luồng sẽ nhảy tới phần thân của hàm, thực hiện tất cả những câu lệnh ở trong đó, rồi trở lại tiếp tục thực hiện từ điểm mà nó vừa rời đi.

Điều này nghe có vẻ đơn giản, nhưng sẽ khác đi nếu bạn nhận thấy rằng một hàm có thể gọi hàm khác. Khi ở trong phần thân của một hàm, chương trình có thể phải thực hiện những câu lệnh ở trong phần thân của một hàm khác. Nhưng khi đang thực hiện hàm mới đó, chương trình còn phải thực hiện một hàm khác nữa!

May mắn là Python rất giỏi theo dõi vị trí thực hiện của chương trình, vì vậy mỗi khi một hàm được thực hiện xong, chương trình sẽ trở về chỗ mà nó đã rời đi từ hàm gọi ban đầu. Khi trở về cuối chương trình, việc thực hiện kết thúc.

Vậy ý nghĩa của câu chuyện này là gì? Khi đọc một chương trình, bạn không nhất thiết phải đọc từ trên xuống dưới. Đôi khi việc dò theo luồng thực hiện của chương trình sẽ có lý hơn.

Tham số và đối số

Một số các hàm dựng sẵn mà ta đã gặp có yêu cầu đối số. Chẳng hạn, khi gọi hàm `math.sin` bạn cần nhập vào một đối số. Một số hàm còn lấy hơn một đối số: `math.pow` lấy hai đối số là cơ số và số mũ.

Bên trong hàm, các đối số được gán cho các biến được gọi là **tham số**. Sau đây là ví dụ về một hàm do người dùng định nghĩa; hàm này lấy một đối số:

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

Hàm này gán một đối số cho một tham số có tên là `bruce`. Khi hàm được gọi, nó in ra giá trị của tham số hai lần (bất kể tham số là gì).

Hàm này hoạt động được với bất kì giá trị nào có thể in được.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

Quy tắc áp dụng cho các hàm dựng sẵn cũng có thể áp dụng được cho các hàm người dùng tạo ra, vì vậy ta có thể dùng bất kì loại biểu thức nào làm đối số cho `print_twice`:

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

Đối số được ước lượng trước khi hàm số được gọi, vì vậy trong các ví dụ, các biểu thức `'Spam '*4` và `math.cos(math.pi)` đều chỉ được ước lượng một lần.

Bạn cũng có thể dùng một biến cho một đối số:

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

Tên của biến được đưa vào như đối số (`michael`) không có liên quan gì đến tân của tham số (`bruce`). Giá trị nào được gọi về (ở đoạn chương trình gọi) cũng không quan trọng; ở đây trong `print_twice`, chúng ta đều gọi mọi người với tên `bruce`.

Các biến và tham số đều có tính địa phương

Khi tạo ra một biến ở trong hàm, nó mang tính **địa phương**, theo nghĩa rằng nó chỉ tồn tại bên trong hàm số. Chẳng hạn:

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

Hàm này nhận hai đối số, nối chúng lại, và sau đó in ra kết quả hai lần. Sau đây là một ví dụ sử dụng hàm:

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

Khi `cat_twice` kết thúc, biến `cat` bị huỷ bỏ. Nếu cố gắng in nó, ta sẽ nhận được một biệt lệ:

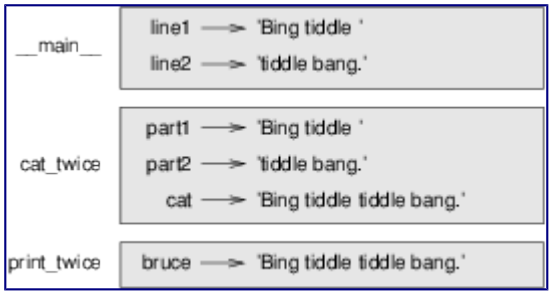
```
>>> print cat
NameError: name 'cat' is not defined
```

Các tham số cũng có tính địa phương. Chẳng hạn, bên ngoài `print_twice`, không có thứ gì được gọi là `bruce` cả.

Biểu đồ ngăn xếp

Để theo dõi xem những biến nào được sử dụng ở đâu, đôi khi sẽ tiện lợi nếu ta vẽ một **biểu đồ ngăn xếp**. Cũng như biểu đồ trạng thái, biểu đồ ngăn xếp cho thấy giá trị của từng biến, đồng thời cho thấy hàm mà mỗi biến thuộc về.

Mỗi hàm đều được biểu diễn bởi một **khung**. Khung là một hình chữ nhật, có tên của hàm số ghi bên cạnh, cùng với các tham số và biến số của hàm được ghi trong đó. Biểu đồ ngăn xếp cho ví dụ trước có dạng như sau:



Các khung được bố trí trong một ngăn xếp cùng với chỉ định hàm nào gọi những hàm nào, và cứ như vậy. Ở ví dụ này, `print_twice` được gọi bởi `cat_twice`, và `cat_twice` được gọi bởi

__main__, vốn là một tên đặc biệt dành cho khung cấp cao nhất. Khi bạn tạo ra một biến không nằm trong bất cứ hàm nào, thì nó sẽ nằm trong __main__.

Mỗi tham số tham chiếu đến giá trị tương ứng với đối số của nó. Do vậy, part1 có cùng giá trị với line1, part2 có cùng giá trị với line2, và bruce có cùng giá trị với cat.

Nếu có một lỗi xảy ra trong quá trình gọi hàm, Python sẽ in ra tên của hàm, cùng với tên của hàm số gọi hàm trước đó, và cứ như vậy cho đến khi trở về __main__.

Chẳng hạn, nếu bạn cố gắng truy cập cat từ bên trong print_twice, bạn sẽ nhận được một thông báo lỗi 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
```

Danh sách các hàm như vậy có tên là **dò ngược**. Nó cho bạn biết file chương trình nào có chứa lỗi, và dòng lệnh nào cũng như những hàm nào được thực hiện lúc bấy giờ. Nó cũng cho biết dòng lệnh gây ra lỗi.

Thứ tự của các hàm trong dò ngược cũng giống như thứ tự của các khung trong sơ đồ ngăn xếp. Hàm số đang được chạy có vị trí dưới cùng.

Các hàm có và không trả lại kết quả

Một số hàm mà chúng ta dùng, như các hàm toán học, đều cho ra kết quả; ta gọi nôm na là **hàm trả lại kết quả**. Các hàm khác, như print_twice, thực hiện một hành động, nhưng không trả lại kết quả nào. Chúng được gọi là **hàm không kết quả**.

Khi bạn gọi hàm trả kết quả, thường thì bạn muốn thực hiện thao tác với kết quả thu được; chẳng hạn, bạn muốn gán nó cho một biến hoặc dùng nó như một phần của biểu thức:

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

Khi bạn gọi một hàm từ chế độ tương tác, Python hiển thị kết quả:

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

Nhưng trong một văn lệnh, nếu bạn gọi một hàm trả kết quả, thì giá trị kết quả này sẽ vĩnh viễn mất đi!

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

Văn lệnh này tính giá trị căn bậc hai của 5, nhưng vì nó không ghi lại và cũng chẳng hiển thị kết quả, nên nó không có tác dụng gì.

Các hàm không kết quả có thể hiển thị thứ gì đó trên màn hình hoặc có những hiệu ứng khác, nhưng chúng không có giá trị được trả về. Nếu bạn cố gắng gán kết quả vào một biến, bạn sẽ được một giá trị đặc biệt gọi là None.

```
>>> result = print_twice('Bing')
Bing Bing
```

```
>>> print result
None
```

Giá trị None không phải là chuỗi 'None'. Nó là một giá trị đặc biệt và có dạng riêng của mình:

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

Các hàm ta đã viết cho đến giờ đều thuộc loại không kết quả. Ta sẽ bắt đầu viết các hàm trả lại kết quả kể từ những chương tiếp sau.

Tại sao lại cần có hàm?

Có thể sẽ không rõ rằng tại sao ta phải cất công chia nhỏ chương trình thành các hàm. Có một số lí do cho điều đó:

- Việc tạo ra một hàm mới sẽ giúp bạn có khả năng đặt tên cho một nhóm các câu lệnh, từ đó làm cho chương trình dễ đọc và gỡ lỗi hơn.
- Các hàm có thể thu gọn một chương trình bằng cách loại bỏ những đoạn mã lệnh trùng lặp. Sau này, nếu bạn sửa đổi chương trình, thì chỉ cần thực hiện sửa ở một chỗ.
- Việc chia một chương trình dài thành những hàm cho phép ta gỡ lỗi từng phần một và sau đó kết hợp lại để được một chương trình tổng thể hoạt động được.
- Các hàm được thiết kế tốt sẽ hữu dụng với nhiều chương trình. Một khi bạn viết ra một hàm và gỡ lỗi xong xuôi, bạn có dùng lại nó.

Gỡ lỗi

Nếu bạn dùng một trình soạn thảo văn bản để viết các văn lệnh, bạn có thể gặp rắc rối liên quan đến các dấu trống và dấu tab. Cách tốt nhất để tránh những vấn đề kiểu này là chỉ dùng dấu trống (không có dấu tab). Hầu hết các trình soạn thảo nhận biết được Python đều thực hiện điều đó một cách mặc định, tuy nhiên một số trình soạn thảo lại không thể.

Các dấu tab và dấu trống thông thường đều vô hình, điều đó làm cho việc gỡ lỗi khó khăn hơn. Vì vậy, bạn hãy tìm một trình soạn thảo có khả năng xử lý tốt vấn đề thụt đầu dòng.

Và cũng đừng quên lưu lại chương trình trước khi chạy nó. Một số phần mềm môi trường phát triển tự động làm việc này, nhưng số khác thì không. Ở trường hợp thứ hai, chương trình mà bạn nhìn thấy ở trên cửa sổ soạn thảo sẽ khác với chương trình được chạy.

Việc gỡ lỗi có thể tốn nhiều thời gian nếu bạn cứ tiếp tục chạy đi chạy lại chương trình không đúng.

Cần đảm bảo chắc rằng mã lệnh bạn đang nhìn thấy chính là mã lệnh được chạy. Nếu bạn không chắc chắn, hãy đặt một câu lệnh như print 'hello' ở đầu chương trình và chạy lại. Nếu không thể thấy chữ hello thì bạn không chạy đúng chương trình cần được chạy!

Thuật ngữ

hàm:
: Chuỗi các câu lệnh được đặt tên; nhằm thực hiện một thao tác có ích nhất định. Các hàm có hoặc không nhận tham số và có thể có hoặc không cho ra kết quả.

định nghĩa hàm:
: Câu lệnh nhằm tạo ra một hàm, đặt tên cho nó, chỉ định các tham số và các lệnh cần được thực hiện.

đối tượng hàm:

Giá trị được tạo ra bởi định nghĩa hàm. Tên của hàm là một biến tham chiếu đến một đối tượng hàm.

đoạn đầu:
: Dòng đầu tiên trong định nghĩa hàm.

phần thân:
: Chuỗi các câu lệnh bên trong định nghĩa hàm.

tham số:
: Tên được dùng bên trong của hàm để tham chiếu đến giá trị được chuyển dưới dạng đối số.

lời gọi hàm:
: Câu lệnh nhằm thực hiện một hàm. Câu lệnh này bao gồm tên hàm, theo sau là danh sách các đối số.

đối số:
: Giá trị được cung cấp cho hàm khi hàm được gọi. Giá trị này được gán cho tham số tương ứng trong hàm.

biến địa phương:
: Biến được định nghĩa bên trong hàm. Một biến địa phương chỉ có thể được dùng bên trong hàm đó.

giá trị được trả về:
: Kết quả của hàm. Nếu một lời gọi hàm được dùng như một biểu thức thì giá trị được trả về chính là giá trị của biểu thức đó.

hàm có kết quả:
: Hàm có trả lại kết quả.

hàm không kết quả:
: Hàm không trả lại kết quả.

module:
: File có chứa một tập hợp các hàm có liên hệ với nhau đồng thời có thể gồm các định nghĩa khác.

câu lệnh import:
: Câu lệnh dùng để đọc một file module và tạo ra một đối tượng module.

đối tượng module:
: Giá trị được tạo ra bởi một câu lệnh `import` để cho phép truy cập đến các giá trị được định nghĩa trong module đó.

kí hiệu dấu chấm:
: Cú pháp để gọi một hàm trong module khác, bằng cách chỉ định tên của module theo sau là một dấu chấm và tên của hàm.

kết hợp:
: Việc dùng một biểu thức như là một phần của biểu thức lớn hơn, hay một câu lệnh như là một phần của câu lệnh lớn hơn.

luồng thực hiện:
: Thứ tự mà theo đó các câu lệnh được thực hiện khi chạy chương trình.

biểu đồ ngăn xếp:
: Cách biểu diễn bằng hình vẽ cho một loạt các hàm chồng xếp lên nhau, trong đó có chỉ ra các biến của chúng và các giá trị mà chúng tham chiếu đến.

khung:
: Hình chữ nhật trong biểu đồ ngăn xếp dùng để biểu diễn lời gọi hàm. Nó bao gồm các biến địa phương và các tham số của hàm.

dò ngược:
: Danh sách các hàm đang được thực hiện, được biểu thị khi có biệt lệ xảy ra.

Bài tập

Python có một hàm dựng sẵn, có tên là `len` để trả lại độ dài của một chuỗi kí tự, chẳng hạn giá trị của `len('allen')` là 5.

Hãy viết một hàm tên là `right_justify` để đọc vào một chuỗi tên là `s` như một tham số và in ra chuỗi bắt đầu với các kí tự trống; số kí tự trống vừa đủ để cho kí tự cuối cùng của chuỗi nằm tại cột thứ 70 trên màn hình.

```
>>> right_justify('allen')
                                                                allen
```

Một đối tượng hàm là một giá trị mà bạn có thể gán vào một biến hoặc chuyển dưới dạng một tham số. Chẳng hạn, `do_twice` là một hàm nhận vào một đối tượng hàm như một tham số và thực hiện hàm tham số này hai lần:

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

Sau đây là một ví dụ có sử dụng `do_twice` để gọi một hàm tên là `print_spam` hai lần.

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

do_twice(print_spam)
```

1. Hãy gõ một văn lệnh thực hiện ví dụ này và chạy kiểm tra.
2. Sửa đổi `do_twice` sao cho nó nhận vào hai đối số— một đối tượng hàm và một giá trị—và gọi hàm hai lần, trong đó có chuyển giá trị như một đối số.
3. Viết một dạng tổng quát hơn cho `print_spam`, đặt tên là `print_twice`, trong đó nhận một chuỗi như tham số và in nó hai lần.
4. Dùng dạng đã chỉnh sửa của `do_twice` để gọi `print_twice` hai lần, trong đó có chuyển `'spam'` như một tham số.
5. Định nghĩa một hàm mới có tên `do_four` nhận vào một đối tượng hàm và một giá trị, sau đó gọi hàm bốn lần, với giá trị đóng vai trò tham biến. Trong phần thân của hàm được định nghĩa chỉ dùng hai câu lệnh chứ không phải là bốn.

Bạn có thể xem cách giải của tôi ở thinkpython.com/code/do_four.py.

Bài tập này[2] có thể được giải bằng những câu lệnh và các đặc điểm khác của ngôn ngữ mà chúng ta đã được biết đến giờ.

1. Hãy viết một hàm để vẽ ô lưới giống như hình sau đây:

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

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

Gợi ý: để in ra nhiều giá trị trên cùng một dòng, hãy dùng dấu phẩy để phân cách các chuỗi:

```
print '+', '-'
```

Nếu lệnh `print` kết thúc bằng dấu phẩy, Python sẽ tạm dừng in tại dòng hiện tại, và những giá trị tiếp theo được in sẽ nằm trên cùng dòng đó.

```
print '+',
print '-'
```

Kết quả của hai lệnh trên là `'+ -'`.

Một câu lệnh `print` tự bản thân nó kết thúc dòng hiện tại và chuyển đến dòng tiếp theo.

2. Hãy dùng hàm vừa định nghĩa để vẽ một lưới tương tự nhưng gồm bốn hàng và bốn cột.

Bạn có thể xem cách giải của tôi ở thinkpython.com/code/grid.py.

---

1. Sau này chúng ta sẽ xét thêm những ngoại lệ của quy tắc này. ↩
2. Dựa theo một bài tập của Oualline, {Practical C Programming, Third Edition}, O'Reilly (1997) ↩

Chương 4: Nghiên cứu cụ thể: thiết kế giao diện

TurtleWorld

Kèm theo cuốn sách này, tôi có viết một bộ module có tên là Swampy. Một trong những module này là TurtleWorld; nó cung cấp một nhóm các hàm phục vụ cho việc vẽ các đường nét bằng cách điều khiển những "con rùa" chạy trên màn hình.

Bạn có thể tải về Swampy từ `thinkpython.com/swampy`; và thực hiện theo những chỉ dẫn cần thiết để cài đặt Swampy vào máy của mình.

Hãy chuyển đến thư mục có chứa `TurtleWorld.py`, tạo ra một file có tên `polygon.py` và gõ vào đoạn mã lệnh sau:

```
from TurtleWorld import *

world = TurtleWorld()
bob = Turtle()
print bob

wait_for_user()
```

Dòng đầu tiên là một dạng của lệnh `import` mà ta đã gặp; thay vì tạo ra một đối tượng module, nó trực tiếp nhập vào các hàm có trong module đó, từ đó bạn có thể truy cập chúng mà không cần dùng kí hiệu dấu chấm.

Dòng kế tiếp tạo ra một đối tượng TurtleWorld để gán vào `world` và một đối tượng Turtle gán vào `bob`. Việc in ra `bob` bằng lệnh `print` sẽ cho ta thông tin kiểu như:

```
<TurtleWorld.Turtle instance at 0xb7bfbf4c>
```

Điều này có nghĩa là `bob` tham chiếu đến một **cá thể** của Turtle được định nghĩa trong module `TurtleWorld`. Trong trường hợp này, "cá thể" có nghĩa là thành viên của một tập hợp; cá thể kiểu Turtle này là một trong số các cá thể của tập hợp các Turtle.

`wait_for_user` lệnh cho TurtleWorld đợi người dùng thực hiện một thao tác, dù trong trường hợp này người dùng không có nhiều lựa chọn khác ngoài việc đóng cửa sổ.

TurtleWorld cung cấp một số hàm phục vụ cho việc "lái" "con rùa": `fd` và `bk` để đi tiến và lùi, `lt` và `rt` để rẽ trái và rẽ phải. Ngoài ra, mỗi con rùa (đối tượng Turtle) đều nắm một cây bút, vốn lại có thể được nhấc hoặc hạ; nếu hạ bút xuống, rùa sẽ để lại nét vẽ khi nó di chuyển. Các hàm `pu` và `pd` tương ứng với nhấc bút hoặc hạ bút.

Để vẽ một góc vuông, hãy thêm các dòng lệnh dưới đây vào chương trình (sau khi tạo ra `bob` và trước khi gọi `wait_for_user`):

```
fd(bob, 100)
lt(bob)
fd(bob, 100)
```

Dòng đầu tiên nhằm lệnh cho `bob` đi tiến 100 bước. Dòng lệnh thứ hai bảo nó rẽ trái.

Khi chạy chương trình này, bạn sẽ thấy `bob` di chuyển trước hết theo hướng đông, và sau đó theo hướng nam, để lại sau nó hai đoạn thẳng.

Bây giờ hãy chỉnh sửa chương trình để vẽ một hình vuông. Xin đừng đọc tiếp trước khi bạn hoàn thành chương trình này!

Cách lặp lại đơn giản

Đôi khi bạn viết mã lệnh kiểu như sau (ở đây không nói đến đoạn lệnh dùng để khởi tạo TurtleWorld và đợi người sử dụng):

```
fd(bob, 100)
lt(bob)

fd(bob, 100)
lt(bob)

fd(bob, 100)
lt(bob)

fd(bob, 100)
```

Chúng ta có thể làm việc này bằng cách viết gọn hơn với một lệnh `for`. Hãy thêm ví dụ sau đây vào `polygon.py` và chạy lại chương trình:

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

Bạn sẽ thấy kết quả gióng như:

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

Đây là cách dùng lệnh `for` đơn giản nhất; sau này chúng ta sẽ tìm hiểu thêm. Nhưng chỉ bằng cách đơn giản này cũng đủ để viết lại chương trình vẽ hình vuông. Xin đừng đọc tiếp trước khi bạn hoàn thành chương trình.

Đây là cách dùng một lệnh `for` để vẽ hình vuông:

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

Cú pháp của lệnh `for` cũng tương tự như một định nghĩa hàm. Nó gồm có một phần đầu kết thúc bởi dấu hai chấm và một phần thân được viết thụt vào so với lề. Phần thân có thể chứa bao nhiêu câu lệnh cũng được.

Một câu lệnh `for` đôi khi còn đươc gọi là một **vòng lặp** vì dòng thực hiện sẽ đi xuôi theo phần thân và vòng ngược trở lại đầu. Trong ví dụ trên, phần thân được chạy qua bốn lần.

Thực ra phiên bản mã lệnh này hơi khác với các bản vẽ hình vuông trước đó ở chỗ nó thực hiện một lần rẽ nữa sau khi vẽ cạnh cuối cùng của hình vuông. Lần rẽ dư thừa này làm thời gian chạy lâu hơn một chút, nhưng nó đơn giản hoá mã lệnh nếu chúng ta có thể thực hiện các công việc gióng nhau bằng vòng lặp. Phiên bản này cũng có tác dụng đưa con rùa về trạng thái khởi đầu, đặt nó về hướng xuất phát.

Bài tập

Tiếp theo đây là một loạt các bài tập có sử dụng TurtleWorld. Chúng có mục đích riêng ngoài việc giải trí. Khi làm các bài tập này, bạn hãy nghĩ xem mục đích riêng đó là gì.

Bên cạnh các bài tập còn có lời giải. Bạn hãy cố hoàn thành (hoặc ít nhất là nỗ lực làm) các bài tập trước khi xem qua lời giải này.

1. Hãy viết một hàm có tên `square` nhận một tham biến tên `t`, vốn là một con rùa. Hàm này dùng con rùa để vẽ một hình vuông.

   Viết một hàm nhằm mục đích chuyển `bob` như một đối số cho `square`, và sau đó chạy lại chương trình.

2. Thêm một tham biến nữa có tên `length` cho `square`. Sửa đổi phần thân của sao cho độ dài của các cạnh là `length`, và sau đó sửa đổi gọi hàm để cung cấp cho một đối số thứ hai. Chạy lại chương trình. Thử chương trình với một loạt các giá trị của `length`.
3. Các hàm `lt` và `rt` đều mặc định thực hiện rẽ 90 độ, nhưng bạn có thể cung cấp đối số thứ hai chứa số độ. Chẳng hạn, `lt(bob, 45)` thực hiện rẽ `bob` 45 độ về bên trái.

   Tạo một bản sao của `square` và đổi tên thành `polygon`. Thêm một tham số có tên là `n` và sửa đổi phần thân để nó vẽ một hình đa giác đều có n cạnh. Gợi ý: Các góc ngoài của một hình n-giác đều cùng bằng 360. 0 / *n* độ.

4. Viết một hàm có tên là `circle` để điều khiển rùa `t`, và bán kính, `r`, như là hai tham số. Hàm này thực hiện vẽ gần chính xác một đường tròn bằng cách gọi hàm `polygon` với các gía trị phù hợp cho chiều dài cạnh và số cạnh. Thử lại hàm của bạn với một loạt các giá trị của `r`.

   Gợi ý: hình dung ra độ dài chu vi đường tròn và đảm bảo rằng `length * n = circumference` (chu vi).

   Một gợi ý khác: nếu `bob` tỏ ra quá chậm, bạn có thể tăng tốc độ bằng cách thay đổi `bob.delay`, vốn là thời gian giữa các lần dịch chuyển tính theo giây. `bob.delay = 0.01` có thể sẽ đủ nhanh.

5. Viết một bản tổng quát hơn so với `circle` gọi là `arc`, trong đó nhận thêm một tham biến `angle`, nhằm chỉ định bao nhiêu phần đường tròn cần được vẽ. `angle` có đơn vị là độ, vì vậy khi `angle=360`, `arc` sẽ vẽ một đường tròn.

Bao bọc

Bài tập thứ nhất yêu cầu bạn đặt đoạn mã vẽ hình vuông vào trong một định nghĩa hàm và sau đó gọi hàm này, trong đó chuyển con rùa như một tham biến. Một cách làm là như sau:

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

square(bob)
```

Các câu lệnh trong cùng, `fd` và `lt` được thụt lề hai lần, nhằm cho thấy chúng ở bên trong vòng lặp `for`, vốn bản thân ở trong định nghĩa hàm. Dòng tiếp theo, `square(bob)`, lại bắt đầu từ lề trái, đó chính là chỗ kết thúc của cả vòng lặp `for` và định nghĩa hàm.

Bên trong hàm, `t` tham chiếu đến cùng con rùa như `bob` tham chiếu, vì vậy `lt(t)` có cùng ý nghĩa

như `lt(bob)`. Vậy tại sao không đặt tên tham biến là `bob`? Đó là vì `t` có thể là bất cứ con rùa nào chứ không riêng gì `bob`, và bạn có thể tạo ra một con rùa thứ hai và chuyển nó như một đối số của `square`:

```
ray = Turtle()
square(ray)
```

Việc gói một đoạn mã vào trong một hàm được gọi là **bao bọc**. Một trong những ưu điểm của việc bao bọc là nó gắn đoạn mã với một tên cụ thể, chính là một kiểu giúp cho việc biên khảo sau này. Một ưu điểm khác là nếu bạn sử dụng lại đoạn mã, việc gọi tên hàm sẽ ngắn gọn hwon nhiều so với việc sao chép và dán toàn bộ phần thân hàm!

Khái quát hoá

Bước tiếp theo là thêm một tham biến `length` vào `square`. Sau đây là một giải pháp:

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

square(bob, 100)
```

Việc thêm một tham số vào một hàm được gọi là **khái quát hoá** vì nó làm cho hàm số trở nên khái quát hơn: trong phiên bản trước, kích thước của hình vuông là cố định, ở phiên bản này nó có thể lớn nhỏ bất kì.

Bước tiếp theo cũng là một cách khái quát hoá. Thay vì việc vẽ hình vuông, `polygon` vẽ một hình đa giác đều với số cạnh bất kì. Sau đây là một lời giải:

```
def polygon(t, n, length):
    angle = 360.0 / n
    for i in range(n):
        fd(t, length)
        lt(t, angle)

polygon(bob, 7, 70)
```

Đoạn mã trên thực hiện vẽ thất giác đều với mỗi cạnh dài bằng 70. Nếu bạn có nhiêu tham biến hơn, sẽ rất dễ quên ý nghĩa của từng tham biến cũng như thứ tự của chúng. Việc đưa vào tên của các tham biến trong danh sách đối số là hợp lệ và thậm đôi khi là cần thiết:

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

Các tên này được gọi là **tham biến từ khoá** vì chúng bao gồm cả tên các tham biến đóng vai trò "từ khoá" (không nên nhầm với các từ khoá dành riêng trong Python như `while` và `def`).

Cú pháp này giúp cho chương trình trở nên dễ đọc hơn. Nó cũng giúp bạn nhớ được rằng các đối số và tham biến hoạt động thế nào: khi bạn gọi một hàm, các đối số được gán cho các tham biến.

Thiết kế giao diện

Bước tiếp theo là viết `circle`, trong đó nhận một tham biến là bán kính `r`. Sau đây là một lời giải đơn giản có sử dụng `polygon` để vẽ đa giác đều 50 cạnh:

```
def circle(t, r):
    circumference = 2 * math.pi * r
```

```
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

Dòng đầu tiên nhằm tính toán chu vi của đường tròn có bán kính `r` theo công thức $2\pi r$. Vì ta dùng `math.pi` nên cần phải nhập `math`. Để cho tiện, các câu lệnh `import` thường được đặt ở đầu đoạn mã lệnh.

`n` là số đoạn thẳng để vẽ gần đúng đường tròn, sao cho `length` là chiều dài mỗi đoạn. Vì vậy, `polygon` sẽ vẽ một đa giác đều có 50 cạnh gần khớp với một đường tròn có bán kính `r`.

Một hạn chế của lời giải này là `n` là một hằng số; điều đó có nghĩa là với những đường tròn lớn, các đoạn thẳng sẽ rất dài, và với những đường tròn nhỏ, chúng ta mất thời gian để vẽ quá nhiều đoạn thẳng ngắn. Một giải pháp là khái quát hoá hàm này bằng cách nhận `n` làm tham số. Điều này giúp cho người dùng (khi gọi `circle`) có quyền lựa chọn tốt hơn, nhưng giao diện của chương trình vì thế cũng kém phần trong sáng.

Giao diện của một hàm là phần tóm tắt cách dùng hàm đó: các tham biến là gì? Hàm được viết nhằm mục đích gì? Và giá trị được trả lại là gì? Một giao diện "trong sáng" có nghĩa là nó "đơn giản nhất tới mức có thể, nhưng không được đơn giản hơn.[1]

Ở ví dụ này, `r` phải thuộc về giao diện vì nó chi phối đường tròn cần được vẽ. Còn `n` thì ít có lí hơn vì nó liên quan đến những chi tiết gắn với *cách* vẽ đường tròn đó.

Thay vì việc làm lộn xộn giao diện, tốt hơn là ta chọn một giá trị hợp lí cho `n` tuỳ thuộc vào chu vi `circumference`:

```
def circle(t, r):
    circumference = 2 * math.pi * r
    n = int(circumference / 3) + 1
    length = circumference / n
    polygon(t, n, length)
```

Bây giờ số cạnh xấp xỉ bằng `circumference/3`, như vậy mỗi cạnh có độ dài xấp xỉ bằng 3, tức là đủ nhỏ để cho đường tròn được đẹp, nhưng cũng đủ lớn để mã lệnh được hiệu quả, và phù hợp với mọi kích cỡ đường tròn.

Chỉnh đốn

Khi viết `circle`, tôi có thể dùng `polygon` vì một đa giác đều nhiều cạnh có thể gần khớp với một đường tròn. Nhưng `arc` thì không phù hợp; ta không dùng được `polygon` hoặc `circle` để vẽ một cung tròn.

Một cách làm khác là bắt đầu bằng một bản sao của `polygon` và biến đổi nó về thành `arc`. Kết quả có thể là như sau:

```
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

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

Nửa sau của hàm này trông giống như `polygon`, nhưng ta không thể dùng lại `polygon` mà không

thay đổi giao diện. Ta có thể khái quát hoá `polygon` để nhận vào tham biến thứ ba là góc, như khi đó `polygon` lại không còn là một tên gọi phù hợp nữa! Thay vào đó, hãy gọi hàm với tên `polyline` để khái quát hơn:

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

Bây giờ ta có thể viết lại `polygon` và `arc` có dùng `polyline`:

```
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)
```

Sau cùng, ta có thể viết lại `circle` có dùng `arc`:

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

Quá trình này—việc sắp xếp lại chương trình để cải thiện giao diện của hàm và giúp cho sử dụng lại mã lệnh— được gọi là **chỉnh đốn**. Trong trường hợp này, ta đã nhận thấy rằng có sự tương đồng trong mã lệnh của `arc` và `polygon`, vì vậy ta đã chỉnh đốn lại bằng cách đưa phần chung này vào trong `polyline`.

Nếu đã có kế hoạch từ trước, có thể ta đã viết `polyline` từ đầu và tránh việc chỉnh đố, nhưng thường thì vào thời điểm bắt đầu dự án bạn không biết rõ để thiết kế được toàn bộ giao diện. Một khi đã bắt tay vào viết mã lệnh, bạn hiểu hơn về vấn đề cần giải quyết. Đôi khi việc chỉnh đốn là một tín hiệu cho thấy bạn đã học được một điều gì đó.

Một kế hoạch phát triển

Một **kế hoạch phát triển** là một quá trình trong việc lập trình. Ở đây ta sẽ dùng kĩ thuật "bao bọc và khái quát hoá". Các bước trong quá trình này gồm có:

1. Bắt đầu bằng việc viết chương trình nhỏ mà không định nghĩa hàm.
2. Một khi chương trình của bạn đã chạy, hãy đóng gói nó vào trong một hàm và đặt tên cho hàm này.
3. Khái quát hoá hàm bằng cách thêm vào các tham số một cách thích hợp.
4. Lặp lại các bước 1–3 đến khi bạn có một tập hợp các hàm hoạt động tốt. Hãy sao chép và dán các đoạn mã lệnh tốt đó để khỏi đánh máy lại (và gỡ lỗi lại).
5. Tìm mọi cơ hội để cải thiện chương trình bằng cách chỉnh đốn. Chẳng hạn, nếu bạn có đoạn mã lệnh tương tự ở một vài chỗ trong chương trình, hãy xét xem có thể chỉnh đốn bằng việc đưa nó vào một hàm chung hay không.

Quá trình này có một số hạn chế—ta sẽ thấy các giải pháp khác trong phần sau quyển sách—nhưng có thể nó sẽ có ích nếu bạn không biết trước được việc chia chương trình thành các hàm như thế nào cho hợp lí. Phương pháp này giúp bạn thiết kế trong lúc bạn viết chương trình.

docstring

docstring (viết tắt của "documentation string") là một chuỗi được đặt ở đầu một hàm có nhiệm vụ giải thích giao diện. Sau đây là một ví dụ:

```
def polyline(t, length, n, angle):
    """Vẽ n đoạn thẳng với chiều dài cho trước và góc
    (tính bằng độ) giữa chúng.  t là một Turtle.
    """
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Docstring này là một chuỗi đặt trong ba dấu nháy, cũng được gọi là chuỗi nhiều dòng vì ba dấu nháy cho phép chuỗi kéo dài qua nhiều dòng liên tiếp.

Tuy rất gọn gàng nhưng docstring này chứa đầy đủ tất cả những thông tin thiết yếu cho người cần dùng đến hàm này. Nó giải thích một cách cô đọng hàm này có nhiệm vụ gì (mà không nói chi tiết rằng hàm thực hiện bằng cách nào). Nó giải thích ảnh hưởng của mỗi tham biến đối với biểu hiện của hàm và mỗi tham biến có kiểu là gì (trong trường hợp không rõ ràng).

Việc ghi chép này là một phần quan trọng trong thiết kế giao diện. Một giao diện được thiết kế tốt phải là giao diện rất dễ diễn giải; nếu bạn gặp khó khăn khi giải thích các hàm mà bạn viết ra thì đó có thể là dấu hiệu cho thấy giao diện của bạn có thể phải được cải thiện.

Gỡ lỗi

Một giao diện cũng tựa như một giao kèo giữa hàm và chương trình gọi. Chương trình đồng ý cung cấp những tham biến nhất định còn hàm thì đồng ý thực hiện một việc nhất định.

Chẳng hạn, `polyline` đòi hỏi bốn đối số. Thứ nhất phải là một Turtle. Thứ hai phải là một số, và nó hẳn phải là một số dương, dù rằng hàm vẫn hoạt động nếu không phải là số dương. Đối số thứ ba phải là một số nguyên; nếu không thì `range` sẽ báo lỗi (điều này còn tùy vào phiên bản Python mà bạn đang dùng). Thứ tư phải là một số, ma ta hiểu là nó tính bằng độ.

Các yêu cầu trên được gọi là những **điều kiện tiền đề** vì chúng cần được đảm bảo là đúng trước khi hàm được thực hiện. Trái lại, các điều kiện ở cuối hàm được gọi là **trạng thái cuối**. Các trạng thái cuối gồm có những hiệu ứng được mong đợi của hàm (như việc vẽ các đoạn thẳng) và bất kì hiệu ứng phụ nào khác (như di chuyển Turtle hoặc thay đổi gì đó trong khung cảnh).

Các điều kiện tiền đề thuộc về trách nhiệm của chương trình gọi. Nếu chương trình vi phạm một điều kiện tiền đề (đã được viết rõ ở docstring) và hàm không thực hiện được việc, thì lỗi thuộc về chương trình gọi chứ không thuộc về hàm.

Thuật ngữ

cá thể:
: Một thành viên của tập hợp. Trong chương này, TurtleWorld là một cá thể của tập hợp các TurtleWorld.

vòng lặp:
: Một phần của chương trình được thực hiện lặp đi lặp lại.

bao bọc:
: Quá trình đưa một danh sách các câu lệnh vào trong bên một định nghĩa hàm.

khái quát hoá:
: Quá trình thay thế một thứ riêng biệt một cách không cần thiết (chẳng hạn một con số) với một thứ khái quát thích hợp (như một biến hoặc tham biến).

đối số từ khoá:
: Một đối số bao gồm cả tên của đối số đó dưới hình thức của một "từ khoá".

giao diện:
: Một đoạn mô tả cách dùng một hàm, bao gồm tên và lời mô tả các đối số và giá trị được trả về.

kế hoạch phát triển:
: Một quy trình để viết chương trình máy tính.

docstring:
: Một chuỗi xuất hiện bên trong định nghĩa hàm nhằm ghi chép lại giao diện của hàm đó.

điều kiện tiền đề:
: Một yêu cầu cần được thoả mãn bởi chương trình gọi trước khi hàm được thực hiện.

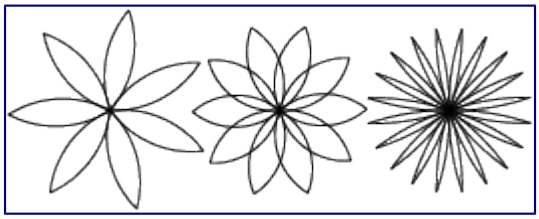
trạng thái cuối:
: Một điều kiện cần được hàm thoả mãn trước khi nó kết thúc.

Bài tập

Tải về mã lệnh trong chương này từ địa chỉ `thinkpython.com/code/polygon.py`.

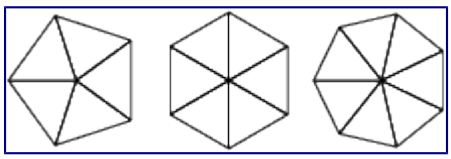
1. Viết các docstring thích hợp cho `polygon`, `arc` và `circle`.
2. Vẽ một sơ đồ ngăn xếp trong đó chỉ ra trạng thái của chương trình khi chạy `circle(bob, radius)`. Bạn có thể tính tay hoặc thêm vào các lệnh `print` kèm theo mã lệnh.
3. Phiên bản `arc` trong Mục {refactor} không chính xác lắm vì cách xấp xỉ đoạn thẳng này luôn nằm ngoài đường tròn đúng. Do đó, con rùa đã dừng lại ở cách đích cuối cùng một vài điểm. Cách làm của tôi đã giảm đi sai lệch này. Hãy đọc mã lệnh và cố gắng hiểu nó. Bạn có thể vẽ biểu đồ và xem cơ chế hoạt động của cách này.

Viết một tập hợp tổng quát gồm các hàm để vẽ những bông hoa như sau:



Bạn có thể tải về một lời giải từ `thinkpython.com/code/flower.py`.

Viết một tập hợp tổng quát gồm các hàm để vẽ những hình như sau:



Bạn có thể tải về một lời giải từ `thinkpython.com/code/pie.py`.

Các chữ cái trong bảng chữ có thể được xây dựng từ một số đủ nhiều các thành phần cơ bản, như những đường thẳng đứng, đường ngang, và đường cong. Hãy thiết kế một bộ phông chữ mà có thể được vẽ với một số tốt thiểu các thành phần cơ bản như vậy; rồi viết các hàm thực hiện việc vẽ chữ cái.

Bạn nên từng hàm riêng cho mỗi chữ cái, với tên hàm như `draw_a`, `draw_b`, v.v., và đặt chung các hàm vào một file có tên là `letters.py`. Bạn có thể tải về một "bộ chữ turtle" từ `thinkpython.com/code/typewriter.py` để so sánh với mã lệnh của bạn.

Bạn có thể tải về một lời giải từ `thinkpython.com/code/letters.py`.

---

1. Nguyên văn: "as simple as possible, but not simpler." (Einstein) ↩

Chương 5: Câu lệnh điều kiện và đệ quy

5.1 Toán tử chia dư

Toán tử chia dư tính với các số nguyên và cho kết quả là phần dư của phép chia số thứ nhất cho số thứ hai. Trong Python, toán tử chia dư có kí hiệu là dấu phần trăm (%). Cú pháp cũng giống như các toán tử khác:

```
>>> thuong = 7 / 3
>>> print thuong
2
>>> sodu = 7 % 3
>>> print sodu
1
```

Như vậy 7 chia cho 3 bằng 2 dư 1.

Toán tử số dư bất ngờ trở nên có ích. Chẳng hạn, bạn có thể kiểm tra xem một số có chia hết cho số khác không—nếu `x % y` bằng không thì `x` chia hết cho `y`.

Hơn nữa, bạn còn có thể lọc ra những chữ số cuối cùng bên phải từ số ban đầu. Chẳng hạn, `x % 10` cho ta số hàng đơn vị của `x` (trong hệ thập phân). Tương tự, `x % 100` cho ta hai chữ số hàng chục và đơn vị.

5.2 Biểu thức Boole

Một **biểu thức Boole** là một biểu thức có giá trị đúng hoặc sai. Các ví dụ sau đây dùng toán tử `==`, để so sánh hai toán hạng và trả lại kết quả `True` (đúng) nếu chúng bằng nhau và `False` (sai) trong trường hợp còn lại:

```
>>> 5 == 5
True

>>> 5 == 6
False
```

`True` và `False` là các giá trị đặc biệt thuộc về kiểu `bool`; chúng không phải là các chuỗi:

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

<type 'bool'>
```

Toán tử `==` là một trong số các **toán tử quan hệ**; các toán tử quan hệ khác gồm có:

```
x != y               # x không bằng y
x > y                # x lớn hơn y
x < y                # x nhỏ hơn y
x >= y               # x lớn hơn hoặc bằng y
x <= y               # x nhỏ hơn hoặc bằng y
```

Mặc dù có thể bạn đã quen thuộc với các toán tử này, song thực ra các kí hiệu Python khác với kí hiệu toán học. Một lỗi phổ biến là dùng nhầm một dấu bằng (=) thay vì viết hai dấu bằng (==). Nhớ

lại rằng = là một toán tử gán, còn == là một toán tử quan hệ. Ngoài ra không có toán tử nào được viết là =< hoặc =>.

5.3 Toán tử lô-gic

Có ba **toán tử lô-gic**: `and`, `or`, và `not`. Nghĩa của các toán tử này giống như nghĩa các từ tương ứng trong tiếng Anh. Chẳng hạn, `x > 0 and x < 10` chỉ đúng khi `x` lớn hơn 0 *và* nhỏ hơn 10.

`n%2 == 0 or n%3 == 0` chỉ đúng khi *một trong hai* điều kiện là đúng; nghĩa là nếu số `n` chia hết cho 2 *hoặc* 3.

Sau cùng, toán tử `not` phủ định một biểu thức Boole. Do vậy `not (x > y)` chỉ đúng khi `x > y` là sai; tức là nếu `x` nhỏ hơn hoặc bằng `y`.

Nói một cách chặt chẽ, các toán hạng đi theo toán tử lô-gic phải là các biểu thức Boole, nhưng Python thì không chặt chẽ vậy. Bất kì con số nào khác không đều được coi như True.

```
>>> 17 and True
True
```

Sự linh hoạt này có thể có lợi, nhưng có một số điểm bất lợi nhỏ khiến người dùng nhầm lẫn. Có thể bạn muốn tránh dùng nó (trừ khi bạn biết chắc rằng mình đang làm gì).

5.4 Thực hiện lệnh theo điều kiện

Để viết được những chương trình thực sự, chúng ta thường cần đến khả năng kiểm tra những điều kiện nhất định và thay đổi biểu hiện tương ứng của chương trình. Các **câu lệnh điều kiện** cung cấp cho ta khả năng này. Dạng đơn giản nhất là lệnh `if`:

```
if x > 0:
    print 'x là số dương'
```

Biểu thức Boole ở cuối lệnh `if` được gọi là **điều kiện**. Nếu nó được thoả mãn thì đoạn lệnh bên trong được thực thi. Nếu không, sẽ chẳng có điều gì xảy ra.

Các lệnh `if` có cùng cấu trúc với các định nghĩa hàm: chúng gồm có một phần đầu và tiếp theo là một lệnh bên trong được viết thụt vào so với lề. Các câu lệnh kiểu như vậy được gọi là **lệnh phức hợp**.

Không có quy định về giới hạn tối đa số câu lệnh ở trong phần thân, nhưng ít nhất phải có một lệnh. Đôi khi ta cần có phần thân mà không chứa một lệnh thực sự nào (thường là chỉ để giữ chỗ cho mã lệnh sau này được viết thêm vào). Trong trường hợp đó, bạn có thể dùng lệnh `pass` để chỉ định "không làm gì cả".

```
if x < 0:
    pass          # cần phải xử lý giá trị âm!
```

5.5 Thực hiện lệnh theo lựa chọn

Dạng thứ hai của lệnh `if` giúp cho việc **thực hiện lệnh theo lựa chọn**, trong đó có hai khả năng và điều kiện được đặt ra để căn cứ vào đó mà lựa chọn thực hiện một trong hai. Cú pháp có dạng như sau:

```
if x%2 == 0:
    print 'x là số chẵn'
else:
```

```
    print 'x là số lẻ'
```

Nếu phần dư của phép chia `x` cho 2 là 0, thì chúng ta biết rằng `x` là số chẵn, và chương trình sẽ hiển thị thông báo điều này. Nếu điều kiện không được thoả mã thì lệnh thứ hai sẽ được thực hiện. Vì điều kiện hoặc là được thoả mãn, hoặc không; nên luôn chỉ có một trong hai phương án được thực hiện. Các phương án này được gọi là **nhánh**, vì chúng là các nhánh rẽ trong luồng thực thi.

5.6 Các điều kiện xâu chuỗi

Đôi khi có nhiều hơn hai khả năng và ta cần nhiều nhánh. Một cách thể hiện quy trình tính toán là dùng các **điều kiện xâu chuỗi**:

```
if x < y:
    print 'x nhỏ hơn  y'
elif x > y:
    print 'x lớn hơn y'
else:
    print 'x bằng y'
```

`elif` là chữ viết tắt của "else if". Một lần nữa, chỉ có đúng một nhánh được thực hiện. Không có giới hạn trên cho số các lệnh `elif`. Nếu có một vế `else`, thì nó phải đứng cuối cùng. Nhưng không nhất thiết cần có vế này.

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

Mỗi điều kiện được kiểm tra lần lượt. Nếu điều kiện thứ nhất sai, điều kiện tiếp theo sẽ được kiểm tra, và cứ như vậy. Nếu mộ ttrong các điều kiện đúng thì nhánh tương ứng được thực hiện; và cả câu lệnh lớn sẽ kết thúc. Ngay cả khi có nhiều hơn một điều kiện được thoả mãn, chỉ có nhánh đúng đầu tiên được thực hiện.

5.7 Các lệnh điều kiện lồng ghép

Một câu lệnh điều kiện có thể được đặt trong một lệnh điều khác. Ta có thể viết lại ví dụ so sánh ba trường hợp như sau:

```
if x == y:
    print 'x bằng y'
else:
    if x < y:
        print 'x nhỏ hơn y'
    else:
        print 'x lớn hơn y'
```

Câu lệnh điều kiện bên ngoài có hai nhánh. Nhánh thứ nhất chỉ chứa một lệnh đơn giản. Nhánh thứ hai lại chứa một câu lệnh `if` khác, mà bản thân nó lại có hai nhánh. Hai nhánh này đều chứa những câu lệnh đơn giản, mặc dù dĩ nhiên chúng có thể là những câu lệnh điều kiện khác.

Tuy cách viết thụt vào trong làm cho cấu trúc rõ ý, nhưng **các lệnh điều kiện lồng ghép** trở nên rất khó để người đọc nhanh. Nhìn chung, tốt hơn là ta nên cố gắng tránh dùng chúng.

Các toán tử lô-gic thường cho ta cách đơn giản hoá các câu lệnh điều kiện lồng ghép. Chẳng hạn, ta

có thể viết lại mã lệnh sau bằng một lệnh điều kiện đơn:

```
if 0 < x:
    if x < 10:
        print 'x là số dương có một chữ số.'
```

Lệnh `print` chỉ được thực hiện một lần nếu ta làm cho nó qua cả hai điều kiện, vì vậy đoạn lệnh sau với toán tử `and` cũng có tác dụng tương tự:

```
if 0 < x and x < 10:
    print 'x là số dương có một chữ số.'
```

5.8 Đệ quy

Việc một hàm gọi một hàm khác là hợp lệ; một hàm gọi chính nó cũng hợp lệ. Mặc dù bề ngoài thì có thể điều này không rõ hay dở ra sao, nhưng thực ra đó chính là một trong những đặc điểm tuyệt vời nhất trong lập trình. Chẳng hạn, hãy xét hàm sau:

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

Nếu `n` bằng 0 hoặc âm, chương trình sẽ in ra chữ, "Bùm!" Còn nếu không, nó sẽ in ra giá trị `n` và sau đó gọi một hàm có tên `countdown`—nghĩa là chính nó—nhưng chuyển vào đối số `n-1`.

Điều gì sẽ xảy ra khi ta gọi hàm kiểu như thế này?

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

Việc thực hiện `countdown` bắt đầu với `n=3`, và do `n` lớn hơn 0, nó đưa ra giá trị 3, và rồi gọi chính nó…

Việc thực hiện `countdown` bắt đầu với `n=2`, và do `n` lớn hơn 0, nó đưa ra giá trị 2, và rồi gọi chính nó…

Việc thực hiện `countdown` bắt đầu với `n=1`, và do `n` lớn hơn 0, nó đưa ra giá trị 1, và rồi gọi chính nó…

Việc thực hiện `countdown` bắt đầu với `n=0`, và do `n` không còn lớn hơn 0, nó đưa ra dòng chữ "Bùm!" và rồi quay về.

Hàm `countdown` ứng với `n=1` quay về.

Hàm `countdown` ứng với `n=2` quay về.

Hàm `countdown` ứng với `n=3` quay về.

Và rồi bạn trở về với `__main__`. Như vậy, toàn bộ kết quả đầu ra như sau:

```
3
2
1
Bùm!
```

Một hàm gọi chính nó được gọi tên là **đệ quy**; quy trình tương ứng cũng được gọi là **đệ quy**.

Với ví dụ tiếp theo đây, ta viết một hàm để in một chuỗi `n` lần.

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

Nếu `n <= 0` thì câu lệnh `return` sẽ kết thúc hàm ngay. Luồng thực hiện của chương trình sẽ lập tức trở về với nơi gọi nó, và phần còn lại của hàm sẽ không được thực hiện.

Phần còn lại của hàm cũng giống như `countdown`: nếu `n` lớn hơn 0, nó sẽ hiển thị `s` và sau đó sẽ gọi chính nó để in lại `s` thêm *n* - 1 lần nữa. Như vậy số dòng kết quả sẽ là `1 + (n - 1)`, tức là bằng `n`.

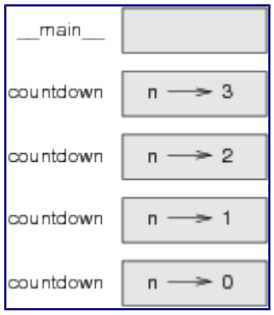
Với những ví dụ đơn giản như trên, có thể sẽ dễ hơn nếu ta dùng một vòng lặp `for`. Nhưng sau này ta sẽ gặp những ví dụ mà ở đó rất khó viết một vòng lặp `for` còn viết bằng đệ quy sẽ dễ hơn, vì vậy việc làm quen với đệ quy từ sớm là rất tốt.

5.9 Biểu đồ ngăn xếp cho các hàm đệ quy

Trong Mục {Biểu đồ ngăn xếp}, chúng ta đã dùng một biểu đồ ngăn xếp để biểu thị trạng thái của một chương trình trong quá trình hàm được gọi. Loại biểu đồ này cũng có thể được dùng để diễn giải hàm đệ quy.

Mỗi khi hàm được gọi, Python tạo ra một "khung" mới cho hàm, trong đó có chứa các biến cục bộ và tham số của hàm. Đối với hàm đệ quy, có thể cùng một thời điểm trên ngăn xếp sẽ tồn tại nhiều khung hàm.

Hình vẽ này minh hoạ một sơ đồ ngăn xếp cho hàm `countdown` khi gọi với `n = 3`:



Như thường lệ, đỉnh của ngăn xếp là một khung cho `__main__`. Nó trống không vì ta không tạo ra bất cứ biến nào trong `__main__` hay chuyển đổi số nào cho nó.

Bốn khung `countdown` có các giá trị khác nhau cho tham biến `n`. Đáy của ngăn xếp, ở đó `n=0`, được gọi là **trường hợp cơ sở**. Nó không thực hiện lời gọi đệ quy, do đó không có thêm khung nào.

> Hãy vẽ một biểu đồ ngăn xếp cho `print_n` được gọi với `s = 'Xin chào'` và `n=2`.
>
> Viết một hàm có tên `do_n` trong đó nhận một đối tượng hàm và một số, `n`, làm hai tham biến, để gọi hàm được chỉ định đúng `n` lần.

5.10 Đệ quy vô hạn

Nếu một quá trình đệ quy không bao giờ đạt đến trường hợp cơ bản, nó tiếp tục thực hiện gọi hàm đệ quy mãi mãi, và chương trình không bao giờ kết thúc. Đây là **đệ quy vô hạn**, và là điều ta thường tránh khi lập trình. Sau đây là một chương trình đơn giản nhất có đệ quy vô hạn:

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

Trong phần lớn các ngôn ngữ lập trình, một chương trình có đệ quy vô hạn sẽ không chạy mãi mãi. Python thông báo một lỗi khi chương trình đạt đến mức độ sâu đệ quy tối đa:

```
  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
```

Lần này kết quả lần vết ngược lớn hơn một chút so với ở chương trước. Khi một lỗi xảy ra, có 1000 khung hàm `recurse` trên ngăn xếp!

5.11 Đầu vào từ bàn phím

Cho đến giờ, các chương trình ta đã viết có phần khuyết điểm ở chỗ không cho người dùng nhập số liệu vào. Lần nào chương trình cũng chạy y nguyên như thế.

Python cung cấp một hàm có sẵn, tên là `raw_input` nhằm thu thập đầu vào từ bàn phím[1]. Khi hàm này được gọi, chương trình tạm dừng và chờ người dùng nhập thông tin từ bàn phím. Đến khi người dùng gõ phím {Return} hoặc {Enter}, chương trình chạy tiếp và `raw_input` trả lại những gì người dùng đã gõ vào dưới dạng một chuỗi kí tự.

```
>>> input = raw_input()
Bạn đang chờ đợi gì?
>>> print input
Bạn đang chờ đợi gì?
```

Trước khi nhận thông tin từ người dùng, có thể sẽ tốt hơn nếu ta in ra một lời nhắc để cho người dùng biết cần nhập vào điều gì. `raw_input` có thể nhận lời nhắc như là một đối số:

```
>>> name = raw_input('Tên bạn là gì?\n')
Tên bạn là gì?
Arthur, Vua xứ Britons!
>>> print name
Arthur, Vua xứ Britons!
```

Chuỗi `\n` ở cuối lời nhắc trên đại diện cho một **newline** (dòng mới), vốn là một kí tự đặc biệt để ngắt dòng. Điều này giải thích tại sao dòng chữ người dùng nhập vào lại xuất hiện phía dưới lời nhắc.

Nếu bạn trông đợi người dùng nhập vào một số nguyên, hãy thử chuyển đổi giá trị thu được sang kiểu `int`:

```
>>> prompt = 'Một con chim én không mang gì có thể bay nhanh bao nhiêu?\n'
>>> speed = raw_input(prompt)
Một con chim én không mang gì có thể bay nhanh bao nhiêu?
17
>>> int(speed)
17
```

Nhưng nếu người dùng nhập vào những thứ khác một chuỗi những chữ số thì có thể sẽ nhận được

thông báo lỗi:

```
>>> speed = raw_input(prompt)
Một con chim én không mang gì có thể bay nhanh bao nhiêu?
Ý của bạn là sao, một con én châu Phi hay châu Âu?
>>> int(speed)
ValueError: invalid literal for int()
```

Trong các phần sau chúng ta sẽ xem xét cách khắc phục lỗi này.

5.12 Gỡ lỗi

Mỗi khi có lỗi, công cụ lần vết ngược trong Python hiển thị rất nhiều thông tin; có thể sẽ quá nhiều đối với người dùng, đặc biệt là khi có nhiều khung trên ngăn xếp. Thường những phần quan trọng nhất cần biết là:

- Lỗi này thuộc loại gì, và
- Nó xuất hiện ở đâu?

Các lỗi cú pháp thường dễ tìm, nhưng cũng có vài chỗ gây bất ngờ. Các lỗi liên quan đến khoảng trắng có thể sẽ khó phát hiện vì trên cửa sổ soạn thảo các dấu cách và dấu tab đều có thể lẫn với nhau.

```
>>> x = 5

>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
SyntaxError: invalid syntax
```

Ở ví dụ này, sự phiền phức là ở chỗ dòng thứ hai được viết thụt vào một dấu cách. Nhưng thông báo lỗi lại chỉ đến `y`; điều này dễ gây ngộ nhận. Nói chung, các thông báo lỗi đều chỉ ra trục trặc được phát hiện ở đâu, nhưng lỗi thực sự lại có thể nằm ở trước đó trong đoạn mã lệnh, đôi khi là ở dòng ngay trước đó.

Điều tương tự cũng đúng với các lỗi runtime (lỗi trong lúc chạy). Chẳng hạn bạn đang thử tính tỉ số tín hiệu so với nhiễu động theo đơn vị đề-xi-ben. Công thức là $SNR_{db} = 10\log_{10}(P_{signal} / P_{noise})$.

Trong Python, bạn có thể viết một đoạn mã giống như sau:

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

Nhưng khi chạy nó, bạn lại nhận được thông báo lỗi[2]:

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

Thông báo lỗi chỉ ra dòng thứ 5, nhưng bản thân dòng đó không có gì sai. Để tìm lỗi thực sự, có thể bạn cần phải in ra giá trị của `ratio`, hoá ra nó bằng 0. Vậy trục trặc xảy ra ở dòng 4, vì việc chia hai số tự nhiên là theo phép chia nguyên bỏ phần dư. Giải pháp khắc phục điều này là biểu thị cả

công suất tín hiệu và công suất nhiễu động dưới dạng các giá trị số thập phân (dấu phẩy động).

Nói chung, các thông báo lỗi sẽ cho bạn biết trục trặc được phát hiện ở đâu, nhưng thường thì đó không phải là nguyên nhân gây ra lỗi.

5.13 Thuật ngữ

toán tử module:
: Toán tử, kí hiệu là dấu phần trăm, (%), được dùng với các số nguyên và trả lại phần dư của phép chia hai số nguyên đó.

biểu thức Boole:
: Biểu thức có giá trị là `True` (đúng) hoặc `False` (sai).

toán tử quan hệ:
: Một trong các toán tử để so sánh các toán hạng của nó: `==`, `!=`, `>`, `<`, `>=`, và `<=`.

toán tử lô-gic:
: Một trong các toán tử để kết hợp các biểu thức Boole: `and`, `or`, và `not`.

câu lệnh điều kiện:
: Câu lệnh để điều khiển dòng thực hiện chương trình tuỳ theo một điều kiện nào đó.

điêu kiện:
: Biểu thức Boole trong một câu lệnh điều kiện để quyết định nhánh nào sẽ được thực hiện.

câu lệnh phức hợp:
: Câu lệnh bao gồm một đoạn đầu và một phần thana. Đoạn đầu kết thúc bởi dấu hai chấm (:). Phần thân được viết thụt vào so với đoạn đầu.

phần thân:
: Một loạt các câu lệnh ở trong một câu lệnh phức hợp.

nhánh:
: Một trong số các phương án trong một câu lệnh điều kiện. Mỗi phương án có thể gồm một loạt các câu lệnh.

câu lệnh điều kiện xâu chuỗi:
: Câu lệnh điều kiện với một chuỗi liên tiếp các nhánh phương án.

câu lệnh điều kiện lồng ghép:
: Câu lệnh điều kiện xuất hiện bên trong của một trong số các nhánh của một câu lệnh điều kiện khác.

đệ quy:
: Quá trình gọi hàm mà hiện thời đang được thực thi.

trường hợp cơ bản:
: Nhánh điều kiện trong một hàm đệ quy mà bản thân không gọi đệ quy.

đệ quy vô hạn:
: Đệ quy mà không có trường hợp cơ bản, hoặc không bao giờ đạt đên trường hợp cơ bản. Đệ quy vô hạn cuối cùng sẽ gây ra lỗi thực thi (runtime error).

5.14 Bài tập

Định lý cuối cùng của Fermat phát biểu rằng không có các số nguyên *a*, *b*, và *c* nào thoả mãn

$a^n + b^n = c^n$
với bất kì giá trị nào của *n* lớn hơn 2.

1. Viết một hàm có tên là `check_fermat` nhận vào bốn tham số—`a`, `b`, `c` và `n`—

rồi kiểm tra xem có thoả mãn định lý Fermat không. Nếu n lớn hơn 2 và hoá ra$a^n + b^n = c^n$
thì chương trình sẽ in ra "Trời, Fermat đã lầm!" Còn nếu không thì chương trình sẽ in ra, "Không, vẫn không đúng".

2. Viết một hàm nhắc người dùng nhập vào các giá trị của Write a function that prompts the user to input values `a`, `b`, `c` và `n`, chuyển chúng sang dạng số nguyên và dùng `check_fermat` để kiểm tra xem liệu chúng có vi phạm định lý Fermat hay không.

Nếu bạn có trong tay ba thanh thẳng, bạn có thể hoặc không thể xếp thành một hình tam giác. Chẳng hạn, nếu một thanh dài 30 cm và hai thanh kia chỉ đều chỉ dài 3 cm, rõ ràng là bạn không thể làm cho hai thanh ngắn nối với nhau được. Với ba thanh có độ dài bất kì, có một cách đơn giản để kiểm tra xem chúng có tạo nên hình tam giác được không.

> "Nếu bất kì một độ dài nào lớn hơn tổng hai độ dài còn lại thì bạn không thể tạo thành tam giác. Ngược lại, bạn có thể. [3]"

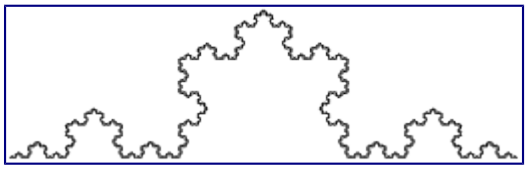
1. Viết một hàm có tên là `is_triangle` nhận vào ba tham số là các số nguyên, sau đó in ra "Yes" hoặc "No", tuỳ theo bạn có thể hay không thể tạo thành hình tam giác từ ba thanh với các độ dài đó.
2. Viết một hàm nhắc người dùng nhập vào độ dài ba thanh, chuyển thành dạng số nguyên, rồi dùng `is_triangle` để kiểm tra xem ba thanh với các độ dài đó có thể được xếp thành tam giác hay không.

Các bài tập tiếp theo đây dùng TurtleWorld từ Chương 4:

Hãy đọc hàm sau đây và thử xem bạn có thể hình dung được mục đích của nó không. Sau đó thì chạy nó (xem các ví dụ ở Chương 4).

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

Đường cong Koch là một hình phân mảnh (fractal) có dạng như hình sau:



Để vẽ một đường cong Koch với độ dài x, tất cả những việc bạn cần làm là

1. Vẽ một đường cong Koch với độ dài x / 3.
2. Quay trái 60 độ.

3. Vẽ một đường cong Koch với độ dài *x* / 3.
4. Quay phải 120 độ.
5. Vẽ một đường cong Koch với độ dài *x* / 3.
6. Quay trái 60 độ.
7. Vẽ một đường cong Koch với độ dài *x* / 3.

Ngoại lệ duy nhất là nếu *x* nhỏ hơn 3. Trong trường hợp đó, bạn chỉ cần vẽ một đoạn thẳng có độ dài *x*.

1. Viết một hàm có tên là `koch` nhận vào các tham số là một Turtle và một độ dài, sau đó dùng Turtle để vẽ một đường cong Koch với độ dài cho trước đó.
2. Viết một hàm có tên là `snowflake` để vẽ ba đường cong Koch nối thành hình một bông tuyết.Bạn có thể xem lời giải của tôi ở `thinkpython.com/code/koch.py`.
3. Có một số cách khái quát hoá đường cong Koch. Hãy xem các ví dụ ở `wikipedia.org/wiki/Koch_snowflake` và viết mã lệnh cho ví dụ mà bạn thích.

---

1. Trong Python 3.0, hàm này có tên là `input`. ↩
2. Với Python 3.0, bạn không còn phải nhận thông báo lỗi nữa; toán tử chia sẽ thực hiện phép chia với số có phần thập phân ngay cả khi các toán hạng là số nguyên. ↩
3. Nếu tổng hai độ dài bằng độ dài thứ ba thì chúng sẽ tạo ra một “tam giác suy biến”. ↩

Chương 6: Các hàm trả lại kết quả

6.1 Các giá trị được trả về

Một số hàm dựng sẵn mà ta đã dùng, như các hàm toán học, đều trả lại kết quả. Việc gọi hàm sẽ tạo ra một giá trị, mà chúng ta thường gán vào một biến hoặc sử dụng như một phần của một biểu thức.

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

Tất cả các hàm chúng ta đã viết đều là hàm rỗng; chúng chỉ in ra thông tin hoặc di chuyển con rùa, nhưng kết quả mà chúng trả về là `None`.

Trong chương này, (cuối cùng thì) chúng ta (cũng) viết những hàm có trả lại kết quả. Ví dụ đầu tiên là `area`, có nhiệm vụ tính diện tích của một hình tròn với bán kính cho trước:

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

Ta đã thấy câu lệnh `return` từ trước rồi, nhưng trong một hàm trả lại kết quả, lệnh `return` bao gồm một biểu thức. Câu lệnh này có nghĩa là: "Lập tức trở về chương trình chính và dùng biểu thức bên cạnh để làm giá trị để trả lại". Vì biểu thức có thể phức tạp tù ý nên chúng ta có thể viết hàm trên gọn lại như sau:

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

Tuy vậy, sự có mặt của các **biến tạm thời** như `temp` thường làm việc gỡ lỗi được dễ dàng hơn.

Đôi khi ta cần có nhiều câu lệnh return, mỗi lệnh ở một nhánh của lệnh điều kiện:

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

Vì các lệnh `return` này ở các nhánh điều kiện độc lập với nhau, luôn chỉ có một trong số đó được thực hiện.

Ngay khi một lệnh return được thực hiện, hàm sẽ kết thúc ngay mà không thực hiện bất cứ lệnh nào tiếp sau nó. Mã lệnh xuất hiện sau dòng lệnh `return`, hay nói chung, trong bất cứ chỗ nào khác của chương trình mà không nằm trong luồng thực hiện thì được gọi là **mã lệnh chết**.

Trong một hàm có trả lại kết quả, ta nên đảm bảo rằng mỗi luồng thực hiện khả dĩ đều dẫn tới một lệnh `return`. Chẳng hạn:

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

Chương trình này không chính xác vì nếu chẳng may `x` bằng 0 thì không có điều kiện nào được

thoả mãn, và hàm sẽ kết thúc mà không gặp phải lệnh `return` nào. Nếu dòng thực hiện đến được cuối của hàm thì giá trị trả về sẽ là `None`, không phải là giá trị tuyệt đối của 0.

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

Tiện thể cũng lưu ý các bạn rằng Python có sẵn một hàm tên là `abs` để tính giá trị tuyệt đối.

Bài tập 1

Viết một hàm tên là `compare` (so sánh) để trả lại `1` nếu `x > y`, `0` nếu `x == y`, và `-1` nếu `x < y`.

6.2 Phát triển tăng dần

Khi bạn viết các hàm lớn hơn, có thể bạn sẽ dành nhiều thời gian để gỡ lỗi.

Để giải quyết các chương trình với mức độ phức tạp ngày càng cao, bạn có thể thử một quy trình gọi là **phát triển tăng dần**. Mục tiêu của phát triển tăng dần là tránh mất thời gian gỡ lỗi bằng cách mỗi lúc chỉ thêm vào và thử nghiệm một đoạn mã lệnh rất ngắn.

Ở ví dụ này, bạn cần tìm khoảng cách giữa hai điểm cho bởi các toạ độ (x_1, y_1) và (x_2, y_2). Theo định lý Py-ta-go, khoảng cách sẽ là:

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

Bước đầu tiên là cân nhắc xem một hàm `distance` trong Python sẽ trông như thế nào. Nói cách khác, các số liệu đầu vào (tham số) và kết quả (giá trị trả lại) là gì?

Trong trường hợp này, số liệu đầu vào mô tả hai điểm; ta có thể biểu thị chúng bằng bốn số. Giá trị cần trả về là khoảng cách, tức là một giá trị số có phần thập phân.

Bạn đã có thể phác thảo ngay ra hàm như sau:

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

Dĩ nhiên là mã lệnh trên chưa tính được khoảng cách; nó luôn trả về số không. Nhưng về mặt cú pháp thì nó đúng, và nó chạy được, nghĩa là bạn có thể thử nghiệm nó trước khi làm cho nó phức tạp thêm.

Để thử nghiệm hàm mới viết, hãy gọi nó với các tham số ví dụ:

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

Sở dĩ tôi chọn các tham số này vì khoảng cách ngang sẽ là 3 và khoảng cách dọc là 4, theo đó thì kết quả sẽ bằng 5 (cạnh huyền của một tam giác có các cạnh là 3-4-5). Khi thử nghiệm một hàm, bạn nên biết trước kết quả đúng.

Đến lúc này, chúng ta có thể khẳng định rằng hàm đã đúng về mặt cú pháp, và chúng ta sẽ thêm mã lệnh vào phần thân. Một bước làm hợp lí tiếp theo là tính các hiệu số $x_2 - x_1$ và $y_2 - y_1$. Đoạn mã tiếp theo sẽ lưu giữ các giá trị trên vào các biến tạm thời và hiển thị chúng.

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

```
    print 'dy bằng', dy
    return 0.0
```

Nếu hàm số hoạt động được, nó sẽ hiển thị `'dx bằng 3'` và `'dy bằng 4'`. Nếu vậy, chúng ta biết rằng hàm đã nhận các đối số đúng và thực hiện chính xác phép tính đầu tiên. Nếu không, chúng ta chỉ cần phải kiểm tra một só ít các dòng lệnh.

Tiếp theo chúng ta tính tổng các bình phương của `dx` và `dy`:

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

Một lần nữa, bạn có thể chạy đoạn mã này và kiểm tra kết quả thu được (nó phải bằng 25). Cuối cùng, bạn có thể dùng `math.sqrt` để tính toán và trả lại kết quả:

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

Nếu đoạn mã trên hoạt động tốt, bạn đã giải quyết xong. Nếu không, bạn có thể sẽ cần phải in giá trị của `result` trước câu lệnh return.

Mã lệnh trên là phiên bản cuối cùng của hàm; nó không hiển thị gì khi được chạy mà chỉ trả lại một giá trị. Các câu lệnh `print` mà chúng ta thêm vào chỉ hữu ích khi gỡ lỗi, nhưng một khi đã viết được hàm rồi thì ta cần phải bỏ chúng đi. Các câu lệnh thêm vào như vậy còn có tên là **dàn giáo** vì nó có ích cho việc xây dựng chương trình nhưng lại không phải là một phần trong sản phẩm cuối cùng.

Khi mới tập lập trình, mỗi lúc bạn chỉ nên viết thêm một hoặc hai dòng lệnh. Sau này khi đã có kinh nghiệm, bạn sẽ viết và gỡ lỗi những khối lệnh lớn hơn. Dù theo cách nào đi nữa, việc phát triển tăng dần sẽ giúp bạn tiết kiệm nhiều thời gian dành cho gỡ lỗi.

Các điểm cơ bản của quy trình này là:

1. Bắt đầu với một chương trình chạy được và thêm vào những thay đổi nhỏ. Bất cứ lúc nào khi gặp lỗi, bạn sẽ phát hiện được ngay lỗi đó ở đâu.
2. Dùng các biến tạm để lưu giữ các giá trị trung gian, từ đó bạn có thể hiển thị và kiểm tra chúng.
3. Một khi chương trình đã hoạt động, bạn có thể dỡ bỏ các đoạn mã "dàn giáo", hoặc rút gọn nhiều câu lệnh về một biểu thức phức hợp, nếu việc này không làm cho chương trình trở nên khó đọc hơn.

Bài tập 2

Hãy dùng cách phát triển tăng dần để viết một hàm tên là `hypotenuse` để trả lại chiều dài của cạnh huyền trong một tam giác vuông khi biết các đối số là chiều dài hai cạnh góc vuông. Hãy ghi lại từng bước phát triển trong quá trình làm.

6.3 Hàm hợp

Đến bây giờ, bạn có thể trông đợi việc gọi một hàm từ bên trong một hàm khác, Việc này được gọi là **hợp** các hàm.

Với ví dụ dưới đây, ta sẽ viết một hàm nhận vào hai điểm là tâm của đường tròn và một điểm trên đường tròn đó, rồi tính diện tích của hình tròn.

Giả sử như toạ độ của tâm điểm được lưu trong các biến `xc` và `yc`, toạ độ điểm trên đường tròn là `xp` và `yp`. Bước đầu tiên sẽ là tìm bán kính của đường tròn, vốn là khoảng cách giữa hai điểm đó. Ta vừa mới viết một hàm, `distance`, để làm việc này:

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

Bước tiếp theo là tìm diện tích của một đường tròn có bán kính đó; chúng ta cũng vừa viết một hàm thực hiện điều này:

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

Kết hợp hai bước này vào trong cùng một hàm, ta thu được:

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

Các biến tạm thời `radius` và `result` có ích cho việc phát triển và gỡ lỗi chương trình, nhưng một khi chương trình đã hoạt động tốt, ta có thể rút gọn nó lại bằng cách kết hợp các lời gọi hàm:

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

Các hàm có thể trả lại giá trị Boole, vốn rất tiện dụng cho việc ẩn giấu các phép kiểm tra phức tạp vào trong một hàm. Chẳng hạn:

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

Một cách làm thông thường là đặt tên các hàm Boole cho giống với các câu hỏi có/không. Chẳng hạn (tiếng Anh: có chia hết) `is_divisible` trả lại `True` hoặc `False` để chỉ định rằng `x` có chia hết cho `y` hay không.

Sau đây là một ví dụ:

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

Kết quả của toán tử `==` là một giá trị Boole, vì vậy ta có thể viết hàm gọn lại bằng cách trả lại giá trị trực tiếp như sau:

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

Các hàm Boole thường được dùng trong các câu lệnh điều kiện:

```
if is_divisible(x, y):
    print 'x chia hết cho y'
```

Có thể bạn sẽ muốn viết:

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

Nhưng phép so sánh thêm là hoàn toàn thừa.

Bài tập 3

Hãy viết một hàm `is_between(x, y, z)` trả lại `True` nếu $x \leq y \leq z$ hoặc `False` trong trường hợp còn lại.

6.4 Nói thêm về đệ quy

Chúng ta mới chỉ tìm hiểu một phần nhỏ của Python, nhưng bạn có thể muốn biết rằng liệu phần nhỏ này có phải là một ngôn ngữ lập trình *đầy đủ* hay không, có nghĩa là dùng nó có thể diễn giải được mọi bài toán hay không. Bất kì chương trình máy tính nào cũng có thể được viết lại chỉ dùng những đặc điểm ngôn ngữ mà chúng ta đã xét đến (thực ra, bạn cần thêm một số lệnh để điều khiển các thiết bị như bàn phím, chuột, ổ đĩa, v.v…, nhưng đó là tất cả những điều cần thiết).

Việc chứng minh nhận định đó là một bài toán khó, lần đầu được Alan Turing đưa ra.[1] Tương ứng với bài toán này là "luận án Turing". Để tìm hiểu cặn kẽ về luận án Turing, bạn nên đọc quyển sách *Introduction to the Theory of Computation* (tạm dịch: "Nhập môn lí thuyết tính toán") của Michael Sipser.

Để cụ thể hoá tác dụng của những kiến thức lập trình mà bạn vừa được học, chúng ta hãy cùng lập một số hàm toán học theo cách đệ quy. Một định nghĩa đệ quy giống như việc định nghĩa vòng quanh; điểm tương đồng là trong phần định nghĩa lại có tham chiếu đến sự vật được định nghĩa. Nhưng cách định nghĩa vòng quanh thực sự thì không mấy có tác dụng:

frabjous:
: Một tính từ được dùng để miêu tả một sự vật frabjous.

Bạn hẳn sẽ bực mình khi thấy một định nghĩa kiểu như vậy trong cuốn từ điển. Ngược lại, khi bạn xem định nghĩa về giai thừa (được kí hiệu bằng dấu !) trong toán học, có thể bạn sẽ thấy:

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

Định nghĩa này phát biểu rằng giai thừa của 0 là 1, và giai thừa của bất kì một giá trị nào khác, n, thì bằng n nhân với giai thừa của n - 1.

Theo đó, 3! bằng 3 nhân với 2!, vốn lại bằng 2 nhân với 1!, vốn bằng 1 nhân với 0!. Gộp tất cả lại, ta có 3! bằng 3 nhân 2 nhân 1 nhân 1, tức là bằng 6.

Nếu bạn có thể phát biểu một định nghĩa có tính đệ quy cho một hàm nào đó thì bạn cũng có thể viết một chương trình Python để tính nó. Bước đầu tiên là xác định các tham số. Trong trường hợp này rõ ràng `factorial` nhận vào một số nguyên:

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

Nếu tham số bằng 0, chúng ta chỉ cần trả lại giá trị 1:

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

Nếu điều đó không xảy ra (đây chính là phần hay nhất), chúng ta thực hiện lời gọi đệ quy để tính giai thừa của *n* - 1 và sau đó nhân nó với *n*:

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

Luồng thực hiện của chương trình này cũng giống như của chương trình `countdown` trong Mục "đệ quy". Nếu ta gọi `factorial` với giá trị 3:

Vì 3 khác 0 nên ta chọn nhánh thứ hai và tính giai thừa của `n-1`…

Vì 2 khác 0 nên ta chọn nhánh thứ hai và tính giai thừa của `n-1`…

Vì 1 khác 0 nên ta chọn nhánh thứ hai và tính giai thừa của `n-1`…

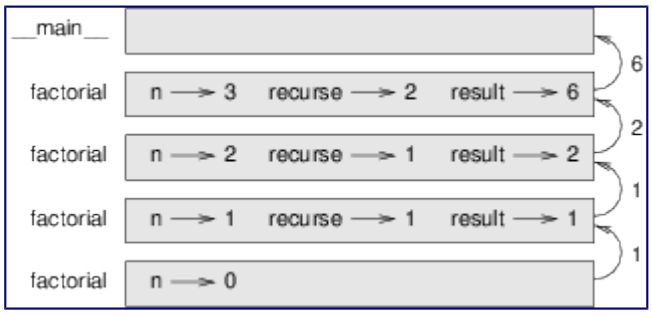
Vì 0 *bằng* 0 nên ta chọn nhánh thứ nhất và trả lại giá trị 1 và không gọi đệ quy thêm lần nào nữa.

Giá trị được trả về, 1, được nhân với *n*, vốn bằng 1, và kết quả được trả lại.

Giá trị được trả về (1) được nhân với *n*, vốn bằng 2, và kết quả được trả lại.

Giá trị được trả về (2) được nhân với *n*, vốn bằng 3, và kết quả, 6 trở thành giá trị trả về của hàm ứng với lúc bắt đầu gọi đệ quy.

Sau đây là nội dung của biểu đồ ngăn xếp khi một loạt các hàm được gọi:



Các giá trị trả lại như ở đây được chuyển về ngăn xếp. Ở mỗi khung, giá trị trả lại chính là giá trị của `result`, vốn là tích của `n` và `recurse`.

Ở khung cuối cùng, các biến địa phương `recurse` và `result` đều không tồn tại, vì nhánh tạo ra chúng không được thực hiện.

6.5 Niềm tin

Việc dõi theo luồng thực hiện của chương trình là một cách đọc mã lệnh, nhưng bạn sẽ nhanh chóng lạc vào mê cung. Một cách làm khác mà tôi gọi là "niềm tin" như sau. Khi bạn dò đến một lời gọi hàm, thay vì việc đi theo luồng thực hiện, hãy *coi như* là hàm đó hoạt động tốt và trả lại kết quả đúng.

Thật ra, bạn đã từng có "niềm tin" này khi dùng các hàm dựng sẵn. Mỗi khi gọi `math.cos` hay `math.exp`, bạn không kiểm tra nội dung bên trong các hàm này. Bạn chỉ việc giả sử rằng chúng

chạy được vì những người lập trình ra các hàm đó đều giỏi.

Cũng như vậy khi bạn gọi các hàm riêng của mình. Chẳng hạn, trong Mục 6.4, chúng ta đã viết một hàm tên là `is_divisible` để xác định xem một số có chia hết cho một số khác không. Một khi chúng ta tự thuyết phục rằng hàm này đã viết đúng—bằng cách kiểm tra và thử mã lệnh—chúng ta có thể sử dụng hàm mà không cần phải xem lại phần thân hàm nữa.

Điều tương tự cũng đúng với các chương trình đệ quy. Khi bạn đến điểm gọi đệ quy, thay vì đi theo luồng thực hiện, bạn cần phải coi rằng lời gọi đệ quy hoạt động tốt (tức là cho kết quả đúng) và sau đó tự hỏi mình "Giả dụ như ta đã tìm được giai thừa của *n* - 1, liệu ta có tính được giai thừa của *n* không?" Trong trường hợp này, rõ ràng là ta sẽ tính được, bằng cách nhân với *n*.

Dĩ nhiên là sẽ có chút kì lạ trong việc ta giả sử rằng hàm hoạt động tốt khi chưa viết xong nó, nhưng chính vì vậy mà ta gọi đó là niềm tin!

6.6 Thêm một ví dụ

Sau `factorial`, một ví dụ thông dụng khác về hàm toán học được định nghĩa theo cách đệ quy là `fibonacci`. Hàm này được xác định như sau: [2]

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(*n*) = fibonacci(*n* - 1) + fibonacci(*n* - 2);

Theo ngôn ngữ của Python, nó có dạng:

```
def fibonacci (n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Nếu bạn thử gắng theo luồng thực hiện ở đây, ngay cả với các giá trị nhỏ của *n*, bạn sẽ đau đầu ngay. Nhưng bằng niềm tin, nếu bạn coi rằng cả hai lời gọi đệ quy đều hoạt động tốt, thì rõ ràng bạn sẽ thu được kết quả đúng khi cộng chúng lại với nhau.

6.7 Kiểm tra kiểu

Điều gì sẽ xảy ra nếu ta gọi `factorial` với đối số bằng 1.5?

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

Dường như đó là đệ quy vô hạn. Nhưng sao có thể xảy ra điều này? Có một trường hợp cơ bản—khi `n == 0`. Nhưng nếu `n` không phải là số nguyên, chúng ta có thể *lỡ mất* trường hợp cơ bản và đệ quy diễn ra mãi mãi.

Ở lời gọi đệ quy thứ nhất, giá trị của `n` bằng 0.5. Ở lần tiếp theo, `n` bằng - 0.5. Từ đó, nó ngày càng nhỏ hơn (càng âm), nhưng sẽ không bao giờ bằng được 0.

Chúng ta có hai sự lựa chọn. Một là thử khái quát hoá hàm `factorial` để làm việc được với các số có phần thập phân, hoặc ta có thể để `factorial` kiểm tra kiểu đối số của nó. Lựa chọn thứ nhất gắn với việc gọi hàm gamma[3] và nó phần nào đã vượt ra ngoài phạm vi quyên sách này. Vì vậy ta sẽ xét đến cách thứ hai.

Ta có thể dùng hàm dựng sẵn `isinstance` để thẩm định kiểu của đối số. Khi đã dùng nó rồi, có thể khẳng định rằng đối số là số dương:

```
def factorial (n):
    if not isinstance(n, int):
        print 'Factorial chỉ được định nghĩa cho số nguyên.'
        return None
    elif n < 0:
        print 'Factorial chỉ được định nghĩa cho số nguyên dương.'
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Trường hợp cơ bản thứ nhất xử lý các số không nguyên; trường hợp thứ hai bắt lỗi các số nguyên âm. Trong cả hai trường hợp, chương trinh đều in ra thông báo lỗi và trả về `None` để biểu thị rằng có điều gì sai đã xảy ra:

```
>>> factorial('fred')
Factorial chỉ được định nghĩa cho số nguyên.
None
>>> factorial(-2)
Factorial chỉ được định nghĩa cho số nguyên dương.
None
```

Nếu vượt qua được cả hai lần kiểm tra, thi *n* chắc chắn là một số nguyên dương, và ta có thể chứng minh rằng lời gọi đệ quy sẽ kết thúc.

Chương trình này minh hoạ cho một dạng lập trình đôi khi được gọi là **chốt bảo vệ**. Hai lệnh điều kiện đầu có vai trò bảo vệ đoạn mã lệnh tiếp theo khỏi những giá trị có thể gây ra lỗi. Những chốt bảo vệ này giúp ta chứng minh được tính đúng đắn của mã lệnh.

6.8 Gỡ lỗi

Việc chia nhỏ một chương trình lớn thành những hàm con tự nó đã tạo ra những điểm kiểm soát để gỡ lỗi. Nếu một hàm không hoạt động, có thể có ba khả năng cần xét đến:

- Các đối số mà hàm nhận vào có vấn đề; một điều kiện đầu bị vi phạm.
- Bản thân hàm có vấn đề; một trạng thái sau bị vi phạm.
- Giá trị trả lại hoặc cách dùng giá trị này có vấn đề.

Để loại trừ khả năng thứ nhất, bạn có thể thêm vào một câu lệnh `print` tại điểm đầu của hàm để hiển thị các giá trị của đối số (và có thể cả kiểu của chúng nữa). Hoặc bạn có thể viết các đoạn mã kiểm tra những điều kiện đầu này một cách tường minh.

Nếu các tham số có vẻ tốt, hãy thêm một lệnh `print` vào trước mỗi lệnh `return` để hiển thị các giá trị được trả lại. Nếu có thể, hãy kiểm tra các kết quả theo cách thủ công. Cân nhắc việc gọi hàm với các giá trị mà ta dễ dàng kiểm tra kết quả (như ở Mục “phát triển tăng dần”).

Nếu hàm số dường như hoạt động tốt, hãy xem lời gọi hàm để chắc rằng giá trị trả lại được dùng và dùng đúng.

Việc thêm các lệnh print vào đầu và cuối một hàm có thể giúp cho luông thực hiện được rõ ràng hơn. Chẳng hạn, sau đây là một dạng của hàm `factorial` với các lệnh print.

```
def factorial(n):
    space = ' ' * (4 * n)
```

```
    print space, 'giai thừa', n
    if n == 0:
        print space, 'trả lại 1'
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print space, 'trả lại', result
        return result
```

`space` là một chuỗi các kí tự trắng để điều khiển mức độ thụt đầu dòng của chữ cần in ra. Sau đây là kết quả của `factorial(5)` :

```
                    giai thừa 5
                giai thừa 4
            giai thừa 3
         giai thừa 2
     giai thừa 1
 giai thừa 0
 trả lại 1
     trả lại 1
         trả lại 2
             trả lại 6
                 trả lại 24
                     trả lại 120
```

Kiểu in kết quả này có thể sẽ rất tốt nếu bạn bị lẫn khi tìm luồng thực hiện. Để dựng các "dàn giáo" một cách hiệu quả cũng cần chút thời gian, nhưng thêm dàn giáo có thể giúp giảm thiểu việc gỡ lỗi.

6.9 Thuật ngữ

biến tạm thời:
: Biến dùng để lưu một giá trị trung gian trong phép tính phức tạp.

đoạn mã chết:
: Phần chương trình không bao giờ được thực hiện, thường là do nó xuất hiện sau một câu lệnh `return`.

`None`:
: Giá trị đặc biệt được trả về từ các hàm không chứa câu lệnh return hoặc một câu lệnh return mà không kèm theo đối số.

phát triển tăng dần:
: Kế hoạch phát triển chương trình trong đó tránh gỡ lỗi bằng việc mỗi lúc chỉ thêm vào và kiểm thử một đoạn mã lệnh ngắn.

dàn giáo:
: Mã lệnh được dùng trong giai đoạn phát triển chương trình nhưng bị bỏ đi ở phiên bản chương trình cuối.

chốt bảo vệ:
: Dạng lập trình trong đó có dùng một câu lệnh điều kiện để kiểm tra và xử lý các trường hợp có thể gây ra lỗi.

6.10 Bài tập

Bài tập 4

Vẽ biểu đồ ngăn xếp cho chương trình dưới đây. Chương trình sẽ in ra cái gì?

```
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):
    sum = x + y + z
    pow = b(sum)**2
    return pow

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

Bài tập 5

Hàm Ackermann, $A(m, n)$, được định nghĩa là[4]:

$$\mathrm{A(m,\ n)} = \begin{cases} n+1 \text{ nếu } m = 0 \\ A(m-1, 1) \text{ nếu } m > 0 \text{ và } n = 0 \\ A(m-1, A(m, n-1)) \text{ nếu } m > 0 \text{ và } n > 0. \end{cases}$$

Viết một hàm có tên là `ack` để tính hàm Ackermann. Sau đó dùng hàm đã viết để tính `ack(3, 4)`, kết quả đúng là 125. Điều gì sẽ xảy ra với các giá trị `m` và `n` lớn hơn?

Bài tập 6

Một từ đối xứng là từ đọc xuôi ngược đều như nhau, chẳng hạn "noon" và "redivider". Theo cách đệ quy, một từ sẽ là đối xứng nếu các chữ cái đầu và cuối là như nhau và phần giữa cũng là một từ đối xứng.

Dưới đây là các hàm nhận vào một đối số chuỗi và trả lại các chữ cái đầu, cuối và giữa:

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

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

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

Chúng ta sẽ xét bản chất của chúng trong Chương "Chuỗi kí tự".

1. Hãy chép lại các hàm trên vào trong file có tên là `palindrome.py` và sau đó thử chúng. Điều gì sẽ xảy ra khi bạn gọi `middle` với một chuỗi chỉ có 2 chữ cái? 1 chữ cái? và chuỗi trống không có chữ cái nào (được viết là `''`)
2. Viết một hàm có tên là `is_palindrome` nhận vào một đối số chuỗi và trả lại `True` nếu đó là một từ đối xứng và `False` nếu không phải. Hãy nhớ rằng bạn có thể dùng hàm dựng sẵn `len` để biết độ dài của chuỗi.

Bài tập 7

Một số, *a*, là lũy thừa của *b* nếu nó chia hết cho *b* và *a* / *b* cũng là một lũy thừa của *b*. Hãy viết một hàm có tên là `is_power` nhận vào các đối số `a` và `b` rồi trả lại `True` nếu `a` là lũy thừa của `b`.

Bài tập 8

Ước số chung lớn nhất (GCD, greatest common divisor) của *a* và *b* là số lớn nhất mà cả *a* và *b* đều chia hết cho nó.[5]

Một cách tìm GCD của hai số là thuật toán Euclid, vốn dựa trên nhận xét rằng nếu như *r* là số dư của phép chia *a* cho *b*, thì $gcd(a, b) = gcd(b, r)$. Với trường hợp cơ bản, ta có thể coi $gcd(a, 0) = a$.

Hãy viết một hàm có tên `gcd` nhận vào hai tham số `a` và `b`, sau đó trả lại ước số chung lớn nhất của chúng. Nếu bạn cần gợi ý, hãy xem http://vi.wikipedia.org/wiki/Thu%E1%BA%ADt_to%C3%A1n_Euclid.

---

1. Turing là một trong những nhà khoa học máy tính đầu tiên (có người cho rằng ông là một nhà toán học, song cũng có nhiều nhà khoa học máy tính thời sơ khai bấy giờ xuất thân từ toán học). ↩
2. Xem vi.wikipedia.org/wiki/Số_Fibonacci. ↩
3. Xem wikipedia.org/wiki/Gamma_function. ↩
4. Xem vi.wikipedia.org/wiki/Hàm_số_Ackermann. ↩
5. Bài tập này dựa theo một ví dụ từ cuốn sách *Structure and Interpretation of Computer Programs* của Abelson and Sussman. ↩

Chương 7: Lặp

Trở về Mục lục cuốn sách

Phép gán nhiều lần

Có thể bạn đã phát hiện thấy rằng, việc gán nhiều giá trị vào cùng một biến là điều hợp lệ. Một phép gán mới làm cho biến hiện tại tham chiếu đến một giá trị mới (và bỏ tham chiếu đến giá trị cũ).

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

Kết quả của chương trình này là `5 7`, vì lần đầu tiên khi `bruce` được in ra, giá trị của nó là 5, và đến lần thứ hai, giá trị là 7. Dấu phẩy đặt ở cuối câu lệnh `print` thứ nhất ngăn cản việc xuống dòng, và đáp số của hai lần tính toán đều xuất hiện trên cùng một dòng.

Sơ đồ trạng thái sau cho thấy cơ chế của một phép **gán nhiều lần**:

Trong việc gán nhiều lần, ta cần đặc biệt lưu ý để phân biệt giữa một toán tử gán và một đẳng thức. Vì Python dùng dấu bằng (=) cho một lệnh gán, ta có thể bị cám dỗ bởi ý nghĩ rằng một câu lệnh kiểu như `a = b` là một đẳng thức. Thực ra thì không phải vậy!

Trước hết, đẳng thức là một hệ thức đối xứng còn lệnh gán thì không phải. Chẳng hạn, trong toán học, nếu $a = 7$ thì $7 = a$. Nhưng trong Python, câu lệnh `a = 7` thì hợp lệ còn `7 = a` thì không.

Hơn nữa, trong toán học, một đẳng thức thì có thể luôn đúng hoặc luôn sai. Nếu bây giờ $a = b$ thì về sau này a sẽ luôn bằng b. Trong Python, một lệnh gán có thể làm cho hai biến bằng nhau, nhưng có thể sẽ không bằng nhau mãi:

```
a = 5
b = a    # bây giờ a và b bằng nhau
a = 3    # a và b không còn bằng nhau
```

Dòng lệnh thứ ba thay đổi giá trị của `a` nhưng không thay đổi giá trị của `b`, vì vậy chúng không còn bằng nhau.

Mặc dù việc gán nhiều lần thường có ích, song bạn cần cẩn thận khi sử dụng chúng. Việc giá trị của các biến thay đổi thường xuyên có thể làm cho mã lệnh trở nên khó đọc và khó gỡ lỗi.

Cập nhật các biến

Một trong các dạng thông dụng nhất của gán nhiều lần là một lệnh **cập nhật**, trong đó giá trị mới của biến phụ thuộc vào giá trị cũ.

```
x = x+1
```

Lệnh này có nghĩa là "lấy giá trị hiện thời của `x`, cộng thêm một, và cập nhật `x` với giá trị mới".

Nếu bạn thử cập nhật một biến mà hiện không tồn tại, bạn sẽ bị báo lỗi, vì Python luôn ước lượng vế phải trước khi gán một giá trị cho `x`:

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

Trước khi có thể cập nhật một biến, bạn cần phải **khởi tạo** nó, thường là bằng một lệnh gán:

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

Việc cập nhật một biến bằng cách cộng thêm 1 được gọi là **tăng**; còn trừ đi 1 được gọi là **giảm**.

Lệnh `while`

Máy tính thường được dùng để tự động hoá các tác vụ có tính chất lặp lại. Việc lặp lại những thao tác giống hệt hoặc tương tự nhau mà không mắc phải lỗi chính là "sở trường" của máy tính đồng thời là việc mà con người làm rất dở.

Ta đã thấy hai chương trình, `countdown` và `print_n`, trong đó dùng đệ quy để thực hiện thao tác lặp lại, hay còn gọi là { lặp}. Vì lặp là một thao tác thường dùng, nên Python có cung cấp một số đặc điểm ngôn ngữ giúp cho việc thực hiện được dễ dàng. Một trong số đó là lệnh `for` mà chúng ta đã gặp trong Mục {lặp}. Chúng ta sẽ trở lại đó vào một dịp khác.

Một cách khác là dùng lệnh `while`. Sau đây là một phiên bản của { countdown} có dùng lệnh `while`:

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

Bạn gần như có thể đọc lệnh `while` giống cách đọc tiếng Anh: "Khi `n` còn lớn hơn 0, hãy hiển thị giá trị của `n` rồi giảm giá trị của `n` đi 1. Khi đạt đến 0, hãy hiển thị chữ `Bùm!`"

Nói một cách hệ thống hơn, luồng thực hiện của câu lệnh `while` như sau:

1. Ước lượng điều kiện, trả lại `True` hoặc `False`.
2. Nếu điều kiện không thoả mãn, thoát khỏi khối lệnh `while` rồi thực hiện câu lệnh kế tiếp.
3. Nếu điều kiện được thoả mãn, thực hiện phần thân và quay trở lại bước 1.

Dạng luồng thực hiện này được gọi là một **vòng lặp** vì sau bước thứ 3 quay trở về bước đầu tiên.

Phần thân của vòng lặp sẽ thay đổi giá trị của một hoặc nhiều biến sao cho cuối cùng thì điều kiện sẽ không thoả mãn và vòng lặp kết thúc. Còn nếu không thì vòng lặp sẽ tiếp diễn mãi mãi, và được gọi là một **vòng lặp vô hạn**. Một câu chuyện đùa về nhà khoa học máy tính xuất phát từ việc quan sát thấy dòng chữ trên hướng dẫn sử dụng lọ dầu gội đầu: "Lather, rinse, repeat", (Xoa dầu, xối nước và lặp lại) là một vòng lặp vô hạn.

Trong trường hợp của `countdown`, chúng ta có thể chứng tỏ rằng vòng lặp sẽ kết thúc vì biết rằng giá trị của `n` là hữu hạn, và mỗi khi lặp lị thì giá trị của `n` sẽ nhỏ đi. Vì vậy cuối cùng nó sẽ phải trở về 0. Với các trường hợp khác, có thể sẽ không dễ chứng tỏ điều đó:

```
def sequence(n):
    while n != 1:
        print n,
        if n%2 == 0:        # n chẵn
            n = n/2
        else:               # n lẻ
            n = n*3+1
```

Điều kện cho vòng lặp này là `n != 1`, vì vậy vòng lặp sẽ tiếp tục đến khi `n` bằng `1`, khi đó điều kiện sẽ bị vi phạm.

Mỗi lần lặp, chương trình sẽ in ra giá trị của `n` và kiểm tra xem nó là chẵn hay lẻ. Nếu là chẵn, `n` được đem chia cho 2. Nếu lẻ giá trị của `n` được thay thế bởi `n*3+1`. Chẳng hạn, nếu tham biến được truyền cho `sequence` là 3, thì chuỗi số thu được sẽ là 3, 10, 5, 16, 8, 4, 2, 1.

Vì `n` đôi khi tăng và đôi khi giảm, nên không có một chứng minh rõ ràng nào cho thấy `n` sẽ đạt được giá trị 1, hay là chương trình kết thúc. Với một số giá trị riêng của `n`, ta có thể chứng minh sự kết thúc này. Chẳng hạn, nếu giá trị ban đầu là một số lũy thừa của 2 thì giá trị của `n` sẽ là chẵn trong mỗi lần lặp và đến khi nó đạt bằng 1. Ở ví dụ trước đây, chuỗi cũng kết thúc theo cách tương tự kể từ số 16.

Câu hỏi được đặt ra là liệu ta có thể chứng minh rằng chương trình sẽ kết thúc với *mọi giá trị dương* của `n` hay không. Cho đến giờ[1], chưa ai có đủ khả năng chứng minh *hoặc* bác bỏ nó!

Viết lại hàm `print_n` ở Mục {đệ quy} dùng cách lặp thay vì đệ quy.

break

Đôi khi bạn chỉ biết được rằng đã đến lúc kết thúc vòng lặp trong khi đang thực hiện một nửa phần thân của vòng lặp đó. Trường hợp này bạn có thể dùng câu lệnh `break` để thoát khỏi vòng lặp.

Chẳng hạn, giả sử như bạn muốn nhận dữ liệu nhập vào bởi người dùng cho đến lúc họ gõ `xong`. Bạn có thể viết như sau:

```
while True:
    line = raw_input('> ')
    if line == 'xong':
        break
    print line

print 'Xong!'
```

Điều kiện lặp là `True`, tức là luôn đúng, vì vậy vòng lặp tiếp diễn đến khi nó gặp phải lệnh break.

Ở mỗi lần lặp, nó nhắc người dùng bằng cách hiện ra kí hiệu '>'. Nếu người dùng gõ vào chữ `xong`, lệnh `break` sẽ giúp thoát khỏi vòng lặp. Còn không thì chương trình sẽ hiện lại bất cứ dòng chữ gì người dùng đã nhập vào, rồi trở lại đầu vòng lặp. Sau đây là một lần chạy thử:

```
> chưa xong
chưa xong

> xong
Xong!
```

Cách viết vòng lặp `while` như thế này rất thường gặp vì bạn có thể kiểm tra điều kiện ở bất kì nơi nào trong vòng lặp (chứ không riêng ở đầu vòng lặp) và có thể khẳng định điều kiện dừng lặp ("hãy dừng ngay khi điều này xảy ra") thay vì chỉ nói một cách phủ định ("cứ tiếp tục đến khi điều đó xảy ra".).

Căn bậc hai

Các vòng lặp thường được dùng trong những chương trình ở đó có tính các trị số bằng cách bắt đầu với một ước lượng rồi liên lục tính lặp để có được xấp xỉ tốt hơn.

Chẳng hạn, một cách tính căn bậc hai là phương pháp Newton. Giả sử bạn muốn tính căn bậc hai của *a*. Nếu bạn bắt đầu với một giá trị ước lượng bất kì, *x*, bạn có thể tính ra một ước lượng khác tốt hơn theo công thức:

y = (x + a/x)/2 $
Chẳng hạn, nếu *a* bằng 4 và *x* bằng 3:

```
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print y
2.16666666667
```

Tức là giá trị mới đã sát hơn kết quả đúng (√4 = 2). Nếu ta tiếp tục lặp lại quá trình này bằng giá trị ước lượng mới, kết quả thu được còn gần đúng nữa:

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

Sau một số lần tính (cập nhật), kết quả ước lượng sẽ gần như chính xác:

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

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

Nhìn chung ta không thể nói trước rằng sẽ cần tính lặp bao nhiêu lần để thu được kết quả đúng, nhưng ta biết rằng khi ta tiến đến giá trị đúng thì các giá trị ước lượng tìm được sẽ không còn dao động nữa:

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

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

Khi `y == x`, ta có thể dừng lại. Sau đây là một vòng lặp bắt đầu bằng một giá trị ước lượng, `x`, và sau đó điều chỉnh nó đến khi giá trị này ngừng dao động:

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

Với phần lớn các giá trị của `a` cách làm này khá hiệu quả, nhưng nói chung việc kiểm tra điều kiện bằng nhau giữa hai số thập phân `float` là nguy hiểm. Các giá trị số có phần thập phân chỉ gần

đúng: hầu hết các số hữu tỉ, như 1 / 3, và vô tỉ, như $\sqrt{2}$, đều không thể biểu diễn được chính xác dưới dạng `float`.

Thay vì kiểm tra xem `x` và `y` có đúng bằng nhau không, cách làm an toàn hơn là dùng hàm có sẵn `abs` để tính giá trị tuyệt đối, hay độ lớn của hiệu giữa hai số này:

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

trong đó `epsilon` nhận một giá trị như `0.0000001` tùy thuộc vào yêu cầu độ chính xác mà ta cần là bao nhiêu.

> Hãy "gói" vòng lặp này vào trong một hàm có tên `square_root` nhận `a` làm tham số, chọn một giá trị hợp lý của `x`, và sau đó trả lại ước lượng xấp xỉ căn bậc hai của `a`.

Thuật toán

Phương pháp Newton là ví dụ cho một **thuật toán**: đó là một quá trình, một cơ chế để giải một lớp các bài toán (trong trường hợp này là bài toán tính căn bậc hai).

Thật không dễ định nghĩa một thuật toán. Có lẽ để tiện hơn, ta sẽ xét một thứ không phải là thuật toán. Khi bạn học cách nhân hai chữ số với nhau, có thể bạn đã thuộc lòng bảng cửu chương. Như vậy bạn đã ghi nhớ được 100 kết quả phép tính riêng biệt. Những kiến thức như vậy không phải là thuật toán.

Nhưng nếu "lười biếng", bạn có thể dùng mẹo để tính nhẩm. Chẳng hạn, để tìm tích số giữa *n* và 9, bạn có thể viết chữ số đầu là *n* - 1 và chữ số sau là 10 - *n*. Mẹo này cho lời giải tổng quát với mọi phép tính giữa số có một chữ số với 9. Đó chính là thuật toán!

Tương tự, kĩ thuật mà bạn đã được học như phép cộng / trừ có nhớ, phép chia có nhớ đều là các thuật toán. Một trong những thuộc tính của thuật toán là để thực hiện chúng thì không cần trí thông minh. Đó là những quá trình trong đó gồm các bước nối tiếp nhau dựa trên một số quy luật đơn giản.

Theo tôi, thật đáng ngạc nhiên là chúng ta lại dành quá nhiều thời gian ở trường để học cách thực thi các thuật toán mà về bản chất thì chúng không có tính trí tuệ chút nào.

Trái lại, quá trình thiết kế các thuật toán mới thú vị, đầy tính thử thách về trí tuệ, và là phần trọng tâm của việc mà ta gọi là lập trình.

Có những việc mà chúng ta làm một cách tự nhiên, chẳng chút khó khăn hay phải nghĩ ngợi gì, lại chính là những điều khó diễn giải thành thuật toán nhất. Một ví dụ điển hình là việc hiểu ngôn ngữ tự nhiên. Tất cả chúng ta đều có khả năng này, nhưng đến nay chưa ai giải thích được là *bằng cách nào* mà chúng ta hiểu nó, ít nhất là giải thích không cần dưới hình thức một thuật toán.

Gỡ lỗi

Khi bạn bắt đầu viết những chương trình lớn, bạn có thể sẽ phải dành nhiều thời gian hơn để gỡ lỗi. Nhiều dòng mã lệnh đồng nghĩa với nhiều khả năng mắc lỗi và nhiều chỗ dễ phát sinh ra lỗi hơn.

Một cách cắt giảm thời gian gỡ lỗi là "phân đôi". Chẳng hạn, nếu chương trình của bạn có 100 dòng lệnh và nếu bạn cần kiểm tra lần lượt từng dòng một sẽ mất 100 bước.

Thay vì vậy, hãy thử chia đôi bài toán. Nhìn vào khu vực giữa của chương trình, kiếm lấy một giá trị trung gian mà bạn có thể kiểm tra được. Thêm vào một câu lệnh `print` (hoặc là một cách nào khác để tạo ra một hiệu ứng giúp ta kiểm tra) rồi chạy chương trình.

Nếu lần kiểm tra ở điểm giữa này mà không đạt, thì chắc chắn nửa đầu chương trình sẽ chứa lỗi. Nếu đạt, thì lỗi nằm ở nửa sau chương trình.

Mỗi khi kiểm tra như thế này, bạn đã chia nửa số dòng lệnh cần thiết phải tìm kiếm. Ít nhất là vết mặt lý thuyết thì sau sáu bước (tức là ít hơn 100), bạn sẽ có thể giảm số dòng lệnh cần kiểm tra xuống còn 1 đến 2 dòng mà thôi.

Trên thực tế, "điểm giữa của chương trình" thường không rõ ràng và đôi khi ta không thể kiểm tra được ở đó. Vì vậy việc đếm số dòng và tìm chính xác điểm giữa là vô nghĩa. Thay vào đó, hãy nghĩ đến những chỗ trong chương trình mà có nhiều khả năng xảy ra lỗi và những chỗ dễ dàng đặt kiểm tra. Sau đó chọn một chỗ kiểm tra mà bạn nghĩ khả năng xảy ra lỗi trước và sau điểm đó là ngang nhau.

Thuật ngữ

phép gán nhiều lần:
: Việc thực hiện hơn một lần gán giá trị cho cùng một biến khi thực thi một chương trình.

cập nhật:
: Một phép gán trong đó giá trị mới của biến phụ thuộc vào giá trị cũ.

khởi tạo:
: Một phép gán gán trị ban đầu cho một biến mà sau này sẽ được cập nhật.

tăng:
: Việc cập nhật bằng cách cộng thêm vào giá trị sẵn có của một biến (thường là thêm 1).

giảm:
: Việc cập nhật bằng cách bớt đi giá trị sẵn có của một biến.

lặp:
: Việc thực hiện liên tục nhiều lần một nhóm các lệnh bằng cách gọi hàm đệ quy hoặc một vòng lặp.

vòng lặp vô hạn:
: Một vòng lặp trong đó điều kiện kết thúc không bao giờ được thỏa mãn.

Bài tập

Để kiểm tra thuật toán căn bậc hai trong chương này, bạn có thể so sánh nó với `math.sqrt`. Hãy viết một hàm có tên `test_square_root` để in ra một bảng như sau:

```
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
```

Cột đầu tiên có chứa một số, *a*; cột thứ hai là căn bậc hai của *a* được tính bằng hàm ở Bài tập {square_root}; cột thứ ba là căn bậc hai được tính từ `math.sqrt`; cột thứ tư là giá trị tuyệt đối của chênh lệch giữa hai kết quả tính được theo hai cách trên.

Hàm dựng sẵn `eval` nhận vào một chuỗi và dùng trình thông dịch Python để định giá trị của nó. Chẳng hạn:

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

Hãy viết một hàm tên là `eval_loop` để lặp lại các thao tác nhắc người dùng nhập vào chuỗi biểu thức, định giá trị nó bằng `eval`, và sau đó in ra kết quả.

Vòng lặp cần tiếp tục đến tận khi người dùng nhập vào `'xong'`, và kết thúc bằng việc trả lại giá trị của biểu thức được định giá trị sau cùng.

Nhà toán học xuất sắc Srinivasa Ramanujan đã tìm ra một chuỗi số vô tận[2] có thể được dùng để phát sinh ra một giá trị xấp xỉ cho số π:

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

Hãy viết một hàm tên là `estimate_pi` dùng công thức này để tính và trả lại giá trị xấp xỉ của π. Hãy dùng một vòng lặp `while` để tính các số hạng trong tổng đến khi số hạng cuối cùng nhỏ hơn `1e-15` (vốn là cách biểu diễn của Python cho 10^{-15}). Bạn có thể kiểm tra kết quả tính được bằng cách so sánh nó với `math.pi`.

Bạn có thể tham khảo đáp án của tôi ở thinkpython.com/code/pi.py.

---

1. Xem wikipedia.org/wiki/Collatz_conjecture. ↩
2. See wikipedia.org/wiki/Pi. ↩

Chương 8: Chuỗi kí tự

Trở về Mục lục cuốn sách

Chuỗi là một danh sách có thứ tự

Chuỗi là một **danh sách có thứ tự** hợp thành từ những kí tự riêng rẽ. Bạn có thể truy cập đến từng kí tự một bằng cách dùng toán tử là cặp ngoặc vuông:

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

Câu lệnh thứ hai nhằm chọn ra chữ cái có thứ tự 1 của `fruit` và gán nó cho `letter`.

Biểu thức nằm ở trong cặp ngoặc vuông được gọi là **chỉ số**. Chỉ số biểu thị cho kí tự mà bạn đang cần trong chuỗi (vì thế mà nó có tên là "chỉ số").

Nhưng đối với bạn kết quả thu được có thể không như mong đợi:

```
>>> print letter
a
```

Với hầu hết chúng ta thì chữ cái thứ nhất của `'banana'` là `b` chứ không phải `a`. Nhưng với nhà khoa học máy tính, chỉ số là độ dời đi so với vị trí đầu của chuỗi; và chữ cái đầu tiên thì có độ dời bằng 0.

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

Như vậy `b` là chữ cái thứ 0 của `'banana'`, `a` là chữ cái thứ 1, và `n` là chữ cái thứ 2.

Bạn có thể dùng bất kì biểu thức nào, bao gồm các biến và toán tử, cho một chỉ số, nhưng cuối cùng giá trị của chỉ số phải là số nguyên. Nếu không thì bạn sẽ gặp lỗi:

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

`len`

`len` là một hàm có sẵn để trả lại số kí tự trong một chuỗi:

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

Để lấy chữ cái cuối cùng của một chuỗi, có thể là bạn đã từng thử đoạn lệnh như sau:

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

Lí do xuất hiện lỗi `IndexError` là vì không có chữ cái nào ở vị trí chỉ số 6 trong chuỗi `'banana'`. Vì chúng ta tính chỉ số từ 0, nên sáu chữ cái sẽ được đánh số từ 0 đến 5. Để có được chữ cái cuối cùng, bạn phải trừ bớt `length` đi 1:

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

>>> print last
a
```

Một cách khác là bạn dùng chỉ số âm, nghĩa là đếm ngược từ điểm cuối của chuỗi. Biểu thức `fruit[-1]` cho ra chữ cái cuối cùng, `fruit[-2]` cho chữ cái áp chót, và cứ như vậy.

Duyệt bằng một vòng lặp `for`

Có nhiều công việc tính toán liên quan đến xử lý một chuỗi theo từ kí tự một. Bắt đầu từ điểm đầu của chuỗi, từng chữ cái được lựa chọn, một số thao tác được thực hiện đối với chữ cái đó, và công việc được tiếp diễn cho các chữ cái còn lại đến hết chuỗi. Kiểu xử lý như thế này được gọi là **duyệt**. Để viết mã lệnh thực hiện việc duyệt, ta có thể dùng vòng lặp `while`:

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

Vòng lặp này để duyệt chuỗi và hiển thị từng chữ cái trên một dòng riêng. Điều kiện lặp là `index < len(fruit)`, như vậy khi `index` bằng với chiều dài của chuỗi, điều kiện bị vi phạm, và phần thân của vòng lặp không được thực hiện. Chữ cái cuối cùng được truy cập đến sẽ tương ứng với chỉ số `len(fruit)-1`, cũng là chữ cái cuối cùng của chuỗi.

> Hãy viết một hàm nhận vào đối số là một chuỗi và hiển thị lần lượt các chữ cái theo thứ tự ngược lại, mỗi chữ cái trên một dòng riêng.

Một cách khác để thực hiện việc duyệt là dùng một vòng lặp `for`:

```
for char in fruit:
    print char
```

Mỗi lần qua vòng lặp, kí tự tiếp theo của chuỗi được gán cho biến `char`. Vòng lặp tiếp diễn đến khi không còn kí tự nào nữa.

Ví dụ tiếp sau đây minh hoạc cho cách dùng phép ghép nối (phép "cộng" chuỗi) và một vòng lặp `for` để tạo ra một danh sách abc (tức là theo đúng thứ tự của bảng chữ cái). Trong cuốn sách của McCloskey, *Make Way for Ducklings* (Dành đường cho vịt con), tên của các chú vịt con là Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. Vòng lặp sau đây sẽ cho ra những cái tên theo đúng thứ tự:

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

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

Kết quả là

```
Jack
Kack
Lack
Mack
Nack
Oack
```

```
Pack
Qack
```

Dĩ nhiên là kết quả trên không hoàn toàn đúng vì "Ouack" và "Quack" đã bị viết chệch đi.

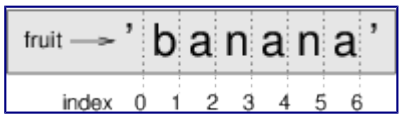
Hãy sửa lại chương trình để có kết quả đúng.

Lát cắt trong chuỗi

Một đoạn trong chuỗi được gọi là **lát cắt**. Việc chọn một lát cắt cũng giống như chọn một kí tự:

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

Toán tử `[n:m]` trả lại phần của chuỗi tính từ kí tự thứ n cho đến kí tự thứ m, trong đó bao gồm kí tự thứ n và không kể kí tự thứ m. Điều này nghe được hợp lý lắm, nhưng nó sẽ giúp bạn hình dung được vị trí của chỉ số là ở *giữa* các kí tự, như trong sơ đồ sau:



Nếu bạn bỏ qua chỉ số thứ nhất (trước dấu hai chấm) thì lát cắt sẽ bắt đầu ở ngay điểm đầu của chuỗi. Nếu bạn bỏ qua chỉ số thứ hai thì lát cắt sẽ kết thúc ở điểm cuối của chuỗi:

```
>>> fruit = 'banana'

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

Nếu chỉ số thứ nhất lớn hơn hoặc bằng chỉ số thứ hai thì kết quả thu được sẽ là một **chuỗi trống**, được biểu thị bằng hai dấu nháy:

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

Một chuỗi trống không chứa kí tự nào và có độ dài bằng 0, nhưng các đặc điểm khác của nó thì cũng tương tự như một chuỗi bất kì.

Biết rằng `fruit` là một chuỗi, thì `fruit[:]` nghĩa là gì?

Chuỗi không thể bị thay đổi

Bạn có thể muốn dùng toán tử `[]` bên vế trái của một lệnh gán, với ý định thay đổi một kí tự trong chuỗi. Chẳng hạn:

```
>>> greeting = 'Hello, world!'

>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

Đối tượng ("object") ở trong trường hợp này chính là chuỗi, còn phần tử ("item") là kí tự mà bạn muốn gán. Tạm thời bây giờ ta coi **đối tượng** cũng giống như một giá trị, nhưng sẽ định nghĩa lại sau. Một **phần tử** là một trong số các giá trị có trong chuỗi.

Lí do gây ra lỗi là ở chỗ các chuỗi đều **không thể thay đổi**, có nghĩa rằng bạn không thể thay đổi một chuỗi hiện có. Việc tốt nhất mà bạn có thể làm được là tạo ra một chuỗi mới như một biến thể của chuỗi ban đầu:

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

Ví dụ này ghép nối một chữ cái mới ở vị trí thứ nhất với lát cát của `greeting`. Nó không có ảnh hưởng gì đến chuỗi ban đầu.

Tìm kiếm

Hàm sau đây nhằm mục đích gì?

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

Theo một nghĩa nào đó, `find` chính là hàm ngược của toán tử `[]`. Thay vì việc nhận một chỉ số và tìm ra kí tự tương ứng, hàm này lại nhận một kí tự và tìm ra chỉ số là vị trí xuất hiện của kí tự đó. Nếu không tìm thấy kí tự, hàm sẽ trả lại `-1`.

Đây là ví dụ đầu tiên mà ta thấy một câu lệnh `return` bên trong vòng lặp. Nếu `word[index] == letter`, hàm sẽ thoát khỏi vòng lặp và trở về ngay lập tức.

Nếu kí tự không xuất hiện trong chuỗi, chương trình sẽ kết thúc vòng lặp một cách bình thường và trả lại `-1`.

Dạng tính toán như thế này—duyệt một danh sách và thoát khi ta tìm được phần tử mà ta cần—được gọi là **tìm kiếm**.

> Chỉnh sửa hàm `find` để nhận thêm tham biến thứ ba nữa, chỉ số trong `word` tại đó việc tìm kiếm được bắt đầu.

Lặp và đếm

Chương trình sau đây đếm số lần xuất hiện của chữ cái `a` trong một chuỗi:

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

Chương trình này giới thiệu một dạng tính toán nữa gọi là **đếm**. Biến `count` được khởi tạo với giá trị 0 và sau đó tăng thêm 1 mỗi lần chữ cái `a` được tìm thấy. Khi vòng lặp kết thúc, `count` sẽ chứa

kết quả—tổng số chữ cái a.

> Hãy gói đoạn mã này vào trong một hàm có tên là count, và sau đó khái quát hóa sao cho nó nhận chuỗi và chữ cái làm các tham biến.
>
> Hãy viết lại hàm sao cho, thay vì duyệt qua chuỗi, hàm sử dụng dạng có 3 tham số của find từ mục trước.

Các phương thức của string (chuỗi)

Một **phương thức** tương tự như một hàm—nó nhận vào các đối số và trả lại một giá trị—nhưng cú pháp lại khác. Chẳng hạn, phương thức upper nhận một chuỗi và trả lại một chuỗi mới trong đó tất cả các chữ cái đều được viết in:

Thay vì dạng cú pháp của hàm upper(word), kiểu cú pháp của phương thức word.upper() được dùng đến.

```
>>> word = 'banana'

>>> new_word = word.upper()
>>> print new_word
BANANA
```

Dạng này của kí hiệu dấu chấm có nêu ra tên của phương thức, upper, và tên của chuỗi mà ta áp dụng phương thức, word. Cặp ngoặc tròn bỏ trống chỉ ra rằng phương thức này không nhận tham biến.

Một lời gọi phương thức được gọi là **kích hoạt**; trong trường hợp này ta nói rằng đã kích hoạt upper đối với word.

Hóa ra rằng, cũng có một phương thức của chuỗi có tên find thực hiện nhiệm vụ giống như hàm mà ta đã viết:

```
>>> word = 'banana'

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

Trong ví dụ này, ta đã kích hoạt find đối với word và sau đó truyền chữ cái cần tìm vào như một tham biến.

Thật ra, phương thức find tổng quát hơn so với hàm của chúng ta; nó có thể tìm được cả các chuỗi con chứ không riêng gì chữ cái:

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

Nó có thể nhận một đối số thứ hai là chỉ số mà tại đó việc tìm kiếm bắt đầu:

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

Và tham số thứ ba là chỉ số mà tại đó dừng việc tìm kiếm:

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

Việc tìm kiếm trên bị thất bại vì b không xuất hiện trong vùng chỉ số từ 1 đến 2 (không bao gồm 2).

> Có một phương thức của chuỗi có tên `count` tương tự như hàm ở trong bài tập trước đây. Hãy đọc tài liệu giải thích hàm này và sau đó viết một lệnh kích hoạt để đếm số chữ cái `a` có trong `'banana'`.

Toán tử `in`

Từ tiếng Anh `in` cũng chính là tên một toán tử nhận vào hai chuỗi và trả lại `True` nếu chuỗi thứ nhật là một chuỗi con của chuỗi thứ hai:

```
>>> 'a' in 'banana'
True

>>> 'seed' in 'banana'
False
```

Chẳng hạn, hàm sau iin ra tất cả các chữ có trong `word1` đồng thời cũng xuất hiện trong `word2`:

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

Với các tên biến được chọn hợp lý thì mã lệnh Python đôi khi đọc lên cũng giống tiếng Anh. Nếu dịch nôm na bằng tiếng Việt thì vòng lặp trên sẽ thành: “với (mỗi) chữ cái trong từ (thứ nhất), nếu chữ cái (này) (cũng có) trong từ (thứ hai) thì in ra chữ cái (này)”.

Và sau đây là kết quả thu được khi bạn so sánh apples (táo) với oranges (cam):

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

So sánh chuỗi

Các toán tử quan hệ cũng có tác dụng đối với chuỗi. Để xem hai chuỗi có bằng nhau hay không:

```
if word == 'banana':
    print  'Được rồi, bananas.'
```

Các toán tử quan hệ khác sẽ giúp ích khi cần xếp các từ theo thứ tự bảng chữ cái:

```
if word < 'banana':
    print 'Từ của bạn,' + word + ', xếp trước banana.'
elif word > 'banana':
    print 'Từ của bạn,' + word + ', xếp sau banana.'
else:
    print 'Được rồi, bananas.'
```

Python không xử lý được chữ in và chữ thường theo cách chúng ta thường làm. Tất cả chữ in đều được xếp trước chữ thường, vì vậy:

```
Từ của bạn, Pineapple, xếp trước banana.
```

Một cách làm thông thường để giải quyết điều này là chuyển đổi chuỗi về một dạng chung, như dạng chữ thường, trước khi thực hiện so sánh. Hãy nhớ điều này khi bạn cần phải tự bảo vệ mình trước một gã vừa vớ lấy Pineapple (quả dứa) làm vũ khí.

Gỡ lỗi

Khi bạn dùng chỉ số để duyệt các giá trị trong một danh s, việc lấy đúng các điểm đầu và cuối danh sách cũng là một mẹo cần lưu ý. Sau đây là một hàm dự định để so sánh hai từ và trả lại `True` nếu từ này là dạng đảo ngược của từ kia; nhưng nó đã có hai lỗi sai:

```
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
```

Lệnh `if` thứ nhất kiểm tra xem các từ có cùng độ dài hay không. Nếu không, ta lập tức trả lại `False`; sau đó trong phần còn lại của hàm, ta coi như các từ đã có cùng độ dài. Đây là một ví dụ cho cách dùng kiểu mẫu chốt chặn trong Mục guardian.

`i` và `j` là các chỉ số: `i` duyệt trên `word1` theo hướng tiến còn `j` duyệt trên `word2` theo hướng lùi. Nếu phát hiện được hai chữ cái không khớp nhau, ta sẽ lập tức trả lại `False`. Nếu như có thể đi qua được toàn bộ vòng lặp và các cặp chữ cái đều khớp nhau thì ta sẽ trả lại `True`.

Nếu kiểm tra hàm này với các từ “pots” và “stop”, có lẽ chúng ta trông đợi kết quả trả về là `True`, nhưng thật ra sẽ nhận được thông báo lỗi IndexError:

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

Để gỡ lỗi kiểu này, hành động trước tiên của tôi là in các giá trị chỉ số ngay trước dòng lệnh có lỗi xảy ra.

```
    while j > 0:
        print i, j        # đặt lệnh print ở đây

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

Bây giờ khi chạy lại chương trình, tôi sẽ thu được thêm thông tin:

```
>>> is_reverse('pots', 'stop')
0 4
...
```

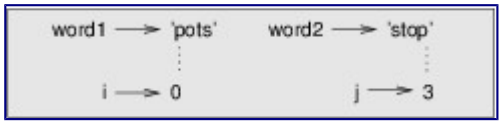
```
IndexError: string index out of range
```

Lần đầu tiên chạy vòng lặp, giá trị của `j` là 4, nghĩa là vượt ra ngoài phạm vi của chuỗi `'pots'`. Chỉ số của kí tự cuối cùng bằng 3, vì vậy giá trị ban đầu của `j` phải là `len(word2)-1`.

Nếu sửa lại lỗi đó và chạy lại chương trình, tôi sẽ thu được:

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

Lần này thì chúng ta có kết quả đúng, nhưng dường như vòng lặp chỉ được thực hiện ba lần, điều này thật đáng ngờ. Để thấy rõ hơn điều gì đã xảy ra, ta cần vẽ một biểu đồ trạng thái. Trong suốt lần lặp thứ nhất, khung chứa `is_reverse` sẽ trông như sau:



Tôi đã sắp xếp các biến trong khung và vẽ thêm các đường đứt nét để cho thấy các giá trị của `i` và `j` dùng để chỉ đến các chữ cái trong `word1` và `word2`.

> Bắt đầu từ biểu đồ này, hãy thực hiện nhẩm chương trình trên giấy, thay đổi các giá trị của `i` và `j` ở mỗi lần lặp. Hãy tìm và sửa lại lỗi sai thứ hai trong hàm này.

Thuật ngữ

đối tượng:
: Thứ mà một biến có thể chỉ định đến. Tạm thời bây giờ bạn có thể coi "đối tượng" và "giá trị" là tương đương nhau.

danh sách:
: Một tập hợp có thứ tự; nghĩa là một tập hợp các giá trị mà mỗi giá trị có thể được chỉ định bởi một chỉ số nguyên.

phần tử:
: Một trong số các giá trị trong một danh s.

số thứ tự:
: Một giá trị số nguyên dùng để chọn một phần tử trong danh s chẳng hạn như một kí tự trong một chuỗi.

lát cắt:
: Một phần của danh sách giới hạn giữa hai số thứ tự.

chuỗi trống:
: Một chuỗi mà không có kí tự nào; nó có độ dài bằng 0 và được biểu thị bằng hai dấu nháy.

(tính) không thể thay đổi:
: Thuộc tính của một danh sách trong đó các phần tử của nó không thể gán giá trị được.

duyệt:
: Việc lặp lại qua các phần tử trong một danh sách, trong khi thực hiện cùng một thao tác với mọi phần tử.

tìm kiếm:
: Một kiểu mẫu duyệt trong đó khi tìm được phần tử mong muốn thì dừng lại.

biến đếm:
: Một biến dùng để đếm thứ gì đó, thường được khởi tạo giá trị không và sau đó tăng dần lên.

phương thức:
: Một hàm được gắn liền với một đối tượng và được gọi theo kiểu cú pháp dấu chấm.

kích hoạt:
: Một câu lệnh làm nhiệm vụ gọi một phương thức.

Bài tập

Một lát cắt của chuỗi có thể nhận một chỉ số thứ ba để chỉ định "kích cỡ của bước"; nghĩa là số khoảng cách giữa các kí tự kế tiếp. Một bước bằng 2 nghĩa là cách một kí tự lấy một; bước 3 nghĩa là cách hai kí tự mới lấy một, v.v…

```
>>> fruit = 'banana'

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

Một bước bằng -1 sẽ duyệt toàn bộ của từ theo hướng ngược lại, vì vậy lát cắt `[::-1]` cho ta một chuỗi lộn ngược lại.

Dùng cách kí hiệu trên để viết một câu lệnh cho hàm `is_palindrome` ở Bài tập palindrome.

Hãy đọc tài liệu hướng dẫn về những phương thức liên quan đến chuỗi tại docs.python.org/lib/string-methods.html. Bạn có thể muốn thử một số phương thức để nắm vững hoạt động của chúng. `strip` và `replace` là hai phương thức đặc biệt có ích.

Tài liệu sử dụng một hình thức cú pháp có thể làm bạn hiểu lầm. Chẳng hạn, trong `find(sub[, start[, end]])`, các cặp ngoặc vuông là để chỉ những đối số có hay không đưa vào đều được. Như vậy `sub` là đối số bắt buộc phải có mặt, còn `start` thì tùy, và nếu bạn có đưa `start` vào, thì `end` vẫn có thể có mặt hoặc không.

Các hàm sau đây đều dự kiến nhằm mục đích kiểm tra xem liệu một chuỗi có bao gồm chữ cái viết thường không, nhưng ít nhất một vài trong số đó bị sai. Với mỗi hàm, hãy mô tả xem tác dụng thực sự của nó là gì (với giả thiết rằng tham biến truyền vào là một chuỗi).

```
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
```

ROT13 là một dạng mã hóa yếu trong đó mỗi chữ cái được "xoay" đi 13 vị trí.[1] Việc xoay một chữ cái có nghĩa là dịch chuyển vị trí trong bảng chữ cái và đến cuối bảng thì quay lộn về đầu. Chẳng hạn chữ 'A' dịch đi 3 thì cho chữ 'D' và 'Z' dịch đi 1 thì cho chữ 'A'.

Hãy viết một hàm có tên là `rotate_word` trong đó nhận các tham biến gồm một chuỗi và một số nguyên, rồi trả lại một chuỗi mới trong đó có chứa các chữ trong chuỗi đầu sau khi đã "xoay" đi số vị trí cho trước.

Chẳng hạn, "cheer" xoay đi 7 là "jolly" và "melon" xoay đi -10 là "cubed".

Bạn có thể cần dùng đến các hàm có sẵn `ord`, which converts a character to a numeric code, and `chr`, which converts numeric codes to characters.

Trên Internet đã có một số câu nói đùa khiêu khích được tạo ra bằng ROT13. Nếu bạn không dễ bị chọc giận, hãy tìm kiếm một số câu nói như vậy và thử giải mã xem.

---

1. Xem wikipedia.org/wiki/ROT13. ↩

Chương 9: Nghiên cứu cụ thể: trò chơi chữ

Trở về Mục lục cuốn sách

9.1 Đọc danh sách các từ

Với những bài tập trong chương này, ta cần đến một danh sách các từ tiếng Anh. Trên mạng hiện nay có rất nhiều danh sách từ, nhưng danh sách thích hợp nhất được Grady Ward sưu tầm và đóng góp cho nguồn tài liệu công cộng, như một phần dự án từ vựng Moby[1]. Nó là một danh sách gồm 113.809 từ chính thức được dùng trong các ô chữ; nghĩa là những từ có nghĩa trong trò chơi ô chữ và các trò đố chữ khác. Trong tuyển tập Moby, tên file danh sách từ là `113809of.fic`; tôi có kèm theo một bản sao của file này với tên gọi đơn giản hơn `words.txt`, trong bộ Swampy.

File này chỉ chứa các kí tự thuần túy, vì vậy bạn có thể mở bằng một trình soạn thảo; nhưng bạn cũng có thể dùng Python để đọc. Hàm có sẵn `open` sẽ nhận vào tên file như một tham biến và trả lại một **đối tượng file** mà bạn dùng nó để đọc nội dung file.

```
>>> fin = open('words.txt')

>>> print fin
<open file 'words.txt', mode 'r' at 0xb7f4b380>
```

`fin` là một cái tên thông dụng để đặt cho đối tượng file nhập vào (input). Chế độ `'r'` chỉ định rằng file này được mở để đọc (reading), khác với `'w'` để ghi (writing).

Đối tượng file cung cấp một số phương thức để đọc, trong đó có `readline`, để đọc các kí tự từ một file cho đến hết dòng, sang đến đầu dòng mới, và trả trả lại kết quả dưới dạng một chuỗi:

```
>>> fin.readline()
'aa\r\n'
```

Từ đầu tiên trong danh mục là "aa", có nghĩa là một loại dung nham. Chuỗi `\r\n` biểu thị hai kí tự trống, một kí tự về đầu dòng (carriage return) và một kí tự nhảy dòng mới (newline), để ngăn cách từ này với từ tiếp theo.

Đối tượng file có nhiệm vụ theo dõi vị trí hiện thời trong file, vì vậy nếu bạn gọi lại `readline` lần nữa, bạn sẽ được từ kế tiếp:

```
>>> fin.readline()
'aah\r\n'
```

Từ tiếp theo là "aah", là một từ hoàn toàn có nghĩa, vì vậy bạn đừng nhìn tôi kiểu như vậy nữa nhé. À, nếu như các dấu trống làm bạn không hài lòng thì hãy dùng phương thức chuỗi có tên là `strip`:

```
>>> line = fin.readline()
>>> word = line.strip()

>>> print word
aahed
```

Bạn cũng có thể dùng một đối tượng file như một bộ phận trong vòng lặp `for`. Chương trình sau sẽ đọc `words.txt` và in ra các từ, mỗi từ trên một dòng:

```
fin = open('words.txt')
for line in fin:
```

```
word = line.strip()
print word
```

Bài tập 1

Hãy viết một chương trình đọc vào `words.txt` và chỉ in ra những từ dài hơn 20 kí tự (không kể dấu trống).

9.2 Bài tập

Trong mục tiếp theo sẽ có lời giải cho các bài tập dưới đây. Vì vậy, bạn cần phải ít ra là cố gắng thử giải chúng trước khi đọc đến đáp án.

Bài tập 2

Vào năm 1939, Ernest Vincent Wright đã xuất bản một cuốn tiểu thuyết gồm 50.000 từ có tên là *Gadsby*, mà trong đó không chứa một chữ cái "e" nào. Việc này thật chẳng dễ dàng vì trong tiếng Anh, "e" là chữ cái thường gặp nhất.

Thực ra khi không dùng chữ cái thông dụng nhất đó thì việc diễn đạt một ý tưởng đơn lẽ cũng là khó rồi. Ban đầu bạn phải thử làm rất chậm, nhưng dần sau này khi cẩn thận và đã có kinh nghiệm nhiều giờ thì bạn sẽ quen hơn.

Thôi, tôi sẽ không nói tiếp về điều đó nữa.

Yêu cầu của bài tập là viết một hàm tên là `has_no_e` nhằm trả lại `True` nếu từ đã cho không chứa chữ cái "e" trong đó.

Hãy chỉnh lại chương trình bạn đã viết trong mục trước để chỉ in ra những từ mà không chứa chữ "e" và tính tỉ lệ phần trăm của các từ trong danh sách mà không chứa chữ "e".

Bài tập 3

Hãy viết một hàm có tên là `avoids`, trong đó nhận vào một từ và một chuỗi các chữ cái bị cấm dùng, sau đó trả lại `True` nếu từ đó không chứa bất kì chữ cái nào trong danh sách bị cấm.

Hãy sửa lại chương trình của bạn để người dùng nhập vào một chuỗi các chữ cái cấm dùng và sau đó in ra tổng số từ mà trong đó không chứa bất kì chữ cái cấm nào.

Bạn có thể tìm được tập hợp 5 chữ cái cấm mà chỉ loại ra số ít nhất các từ vi phạm những chữ cái cấm đó không?

Bài tập 4

Hãy viết một hàm có tên là `uses_only`, trong đó nhận vào một từ và một chuỗi các chữ cái, sau đó trả lại `True` nếu từ đó chỉ cấu thành từ những chữ có trong danh sách chữ cái. Bạn có thể đặt một câu trong đó chỉ dùng các chữ cái `acefhlo` không, ngoài câu "Hoe alfalfa?"

Bài tập 5

Hãy viết một hàm có tên là `uses_all`, trong đó nhận vào một từ và một chuỗi các chữ cái yêu cầu, sau đó trả lại `True` nếu từ đó dùng hết tất cả các chữ cái yêu cầu, mỗi chữ cái ít nhất là một lần. Có bao nhiêu từ dùng tất cả các nguyên âm `aeiou`? Dùng tất cả `aeiouy`?

Bài tập 6

Hãy viết một hàm có tên là `is_abecedarian` để trả lại `True` nếu các chữ cái trong từ xuất hiện xuôi theo thứ tự của bảng chữ cái (chấp nhận các chữ cái lặp lại hai lần). Có tất cả bao nhiêu từ như vậy?

9.3 Tìm kiếm

Tất cả các bài tập trong mục trước đều có điểm chung; chúng đều giải quyết được bằng dạng mẫu tìm kiếm ở Mục find. Ví dụ đơn giản nhất là:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

Vòng lặp `for` duyệt các kí tự có trong `word`. Nếu ta thấy kí tự "e", ta có thể lập tức trả về `False`; còn không thì tiếp tục đến với kí tự tiếp theo. Nếu ta thoát khỏi vòng lặp một cách bình thường, nghĩa là ta không tìm thấy chữ "e", thì ta sẽ trả về `True`.

`avoids` là một dạng khái quát của `has_no_e` nhưng nó cũng có cấu trúc tương tự:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

Chúng ta có thể trả về `False` ngay khi tìm thấy một chữ cái bị cấm; còn nếu chúng ta đến được cuối vòng lặp thì sẽ trả về `True`.

`uses_only` cũng tương tự, chỉ khác là điều kiện được đảo ngược lại:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Thay vì một danh sách các chữ cái bị cấm, ta có một danh sách chữ cái được dùng. Nếu ta tìm thấy một chữ cái trong `word` mà không có trong `available`, ta có thể trả về giá trị `False`.

`uses_all` tương tự, nhưng ta đảo lại vai trò của từ và dẫy các chữ cái:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

Thay vì việc duyệt các chữ cái trong `word`, vòng lặp duyệt các chữ cái được yêu cầu. Nếu bất kì chữ cái yêu cần nào mà không có mặt trong từ thì ta có thể trả về giá trị `False`.

Nếu bạn có thể tư duy như một nhà khoa học máy tính thực thụ, bạn có thể nhận thấy rằng `uses_all` là một trường hợp của một bài toán mà trước đây ta đã giải quyết rồi, và bạn có thể đã viết:

```
def uses_all(word, required):
    return uses_only(required, word)
```

Đó là ví dụ cho một phương pháp xây dựng chương trình gọi là **nhận diện bài toán**, trong đó bạn nhân ra bài toán mà bạn đang giải quyết chính là một trường hợp của bài toán trước đây đã từng làm rồi, và chỉ cần áp dụng cách giải trước đây.

9.4 Lặp với các chỉ số

Tôi đã viết các hàm trong mục trước đây bằng vòng lệnh `for` vì tôi chỉ cần các kí tự có trong chuỗi; tôi không cần làm gì với các chỉ số (thứ tự) của từng kí tự.

Đối với `is_abecedarian` chúng ta phải so sánh các chữ cái liền kề nhau, điều này đòi hỏi phải có mẹo khi dùng một vòng lặp `for`:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

Một cách làm khác là dùng đệ quy:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

hoặc dùng một vòng lặp `while`:

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

Vòng lặp bắt đầu tại `i=0` và kết thúc khi `i=len(word)-1`. Mỗi lần duyệt qua vòng lặp, máy sẽ so sánh chữ cái thứ *i* (mà bạn có thể coi như chữ cái hiện hành) và chữ cái thứ *i*+1 (coi như chữ cái tiếp theo).

Nếu chữ cái tiếp theo mà nhỏ hơn (tức là đứng trước theo bản chữ cái) chữ cái hiện hành, thì ta sẽ phát hiện được sự phá vỡ quy luật sắp xếp, và sẽ trả lại giá trị `False`.

Nếu ta đến được cuối vòng lặp mà không gặp lỗi nào, thì từ đã cho đạt yêu cầu kiểm trả. Để tự tin

khẳng định rằng vòng lặp đã kết thúc đúng, hãy xét một ví dụ như `'flossy'`. Từ này có độ dài là 6, vì vậy vòng lặp kết thúc lần cuối khi `i` bằng 4, ứng với chỉ số của chữ cái áp chót. Ở vòng lặp cuối, máy sẽ so sánh chữ cái áp chót với chữ cái cuối; đây là điều chúng ta muốn.

Sau đây là một dạng khác của `is_palindrome` (xem lại Bài tập palindrome) trong đó có dùng hai biến chỉ số; một bắt đầu từ chữ cái đầu và tăng dần lên; biến kia bắt đầu từ chữ cái cuối và giảm dần xuống.

```
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
```

Hoặc là, nếu bạn nhận thấy rằng đây chỉ là một trường hợp của bài toán đã giải trước đó, thì bạn có thể viết:

```
def is_palindrome(word):
    return is_reverse(word, word)
```

Với giả sử là bạn đã làm Bài tập 9.

9.5 Gỡ lỗi

Kiểm tra chương trình là một việc khó. Các hàm được viết trong chương này đều khá dễ kiểm tra vì bạn có thể kiểm tra kết quả bằng cách tính nhẩm. Tuy vậy, để chọn được toàn bộ tập hợp các từ nhằm kiểm tra chương trình để loại trừ mọi lỗi sẽ là việc rất khó, thậm chí là không thể.

Lấy `has_no_e` làm ví dụ, hiển nhiên là có hai trường hợp để kiểm tra: với những từ có chứa chữ 'e' cần phải trả lại `False`; còn những từ khác thì phải trả lại `True`. Bạn có thể dễ dàng tìm được kết quả trong cả hai trường hợp.

Với mỗi trường hợp, vẫn có những trường hợp nhỏ mà khó nhận ra hơn. Trong số những từ có chứa "e", bạn cần kiểm tra những từ có chứa "e" ở đầu, ở cuối, và ở đoạn giữa của từ. Bạn cần kiểm tra những từ dài, từ ngắn, từ rất ngắn, và cả chuỗi trống nữa. Chuỗi trống chính là ví dụ cho một **trường hợp đặc biệt**, vốn là một trong số các trường hợp không dễ thấy mà ở đó tường tiềm ẩn lỗi.

Ngoài những trường hợp kiểm tra mà bạn tự tạo ra, bạn còn có thể kiểm tra chương trình với một danh sách từ như `words.txt`. Bằng cách lướt qua kết quả tìm được, bạn có thể phát hiện được lỗi, nhưng hãy cẩn thận: bạn chỉ bắt được những lỗi theo một kiểu (ở những từ vốn không nên có trong danh sách kết quả, nhưng lại có) chứ không bắt được lỗi theo kiểu còn lại (những từ đáng ra có trong danh sách kết quả nhưng lại không có).

Nói chung, việc kiểm tra có thể giúp bạn tìm ra lỗi, nhưng thật không dễ tạo ra một bộ trường hợp kiểm tra thật tốt, và dù nếu bạn có làm được vậy đi nữa, cũng không có gì đảm bảo được rằng chương trình sẽ đúng.

Theo lời một nhà khoa học máy tính nổi tiếng: "Việc kiểm tra chương trình có thể được dùng để cho thấy sự tồn tại của lỗi, nhưng không bao giờ có thể dùng để chứng tỏ được sự vắng mặt của chúng!". Nguyên văn:

> Program testing can be used to show the presence of bugs, but never to show their absence!
>
> — Edsger W. Dijkstra

9.6 Thuật ngữ

đối tượng file:
: Một giá trị để biểu thị một file được mở.

nhận diện bài toán:
: Một cách giải quyết bài toán thông qua việc diễn đạt đó như một trường hợp riêng của một bài toán khác đã được giải quyết từ trước.

trường hợp đặc biệt:
: Một trường hợp kiểm tra thuộc vào dạng không điển hình hoặc không dễ thấy (và thường cũng khó xử lý một cách thích hợp).

9.7 Bài tập

Bài tập 7

Câu hỏi này được dựa trên mục đố vui phát sóng trong chương trình *Car Talk*[2]:

> Hãy cho tôi biết một từ trong đó chứa 3 cặp hai chữ cái giống nhau liên tiếp. Tôi sẽ kể ra hai ví dụ gần đạt tiêu chuẩn như vậy. Chẳng hạn, từ "committee", c-o-m-m-i-t-t-e-e. Từ này sẽ đạt yêu cầu nếu chữ 'i' không có ở đó. Hoặc là Mississippi: M-i-s-s-i-s-s-i-p-p-i. Nếu bỏ hết các chữ i trong đó ra, từ này cũng đạt yêu cầu. Nhưng mà có một từ mà tôi biết, nó có 3 cặp đôi chữ cái liên tiếp. Dĩ nhiên là có thể có đến 500 từ tiếng Anh khác như vậy nhưng đến giờ thì tôi chỉ biết được một từ. Đó là từ gì?

Hãy viết một chương trình để tìm ra từ này. Bạn có thể tìm thấy lời giải của tôi ở `thinkpython.com/code/cartalk.py`.

Sau đây là một câu đố khác trong *Car Talk*[3]:

> "Một ngày, khi đang lái xe trên đường cao tốc, tôi đột nhiên chú ý đến đồng hồ công-tơ-mét trên xe. Cũng như các công-tơ-mét khác, nó chỉ sáu chữ số, lấy tròn số dặm đường mà xe đi được. Như vậy, nếu xe tôi đã chạy được 300.000 dặm chẳng hạn, thì tôi sẽ nhìn thấy 3-0-0-0-0-0.
>
> "Bây giờ, trở về chuyện ngày hôm đó, tôi đã thấy một chi tiết thú vị. Tôi nhận thấy rằng bốn chữ số cuối cùng đối xứng nhau; nghĩa là đọc xuôi, đọc ngược đều như nhau. Chẳng hạn, 5-4-4-5 là một chuỗi đối xứng, như vậy công-tơ-mét của tôi có thể đã chỉ 3-1-5-4-4-5.
>
> "Một dặm sau đó, 5 chữ số cuối xếp thành danh sách đối xứng. Chẳng hạn, nó có thể là 3-6-5-4-5-6. Một dặm sau, bốn chữ số giữa trong 6 số xếp thành danh sách 4 đối xứng. Bạn vẫn theo kịp tôi chứ? Này nhé, một dặm sau đó

thì cả chuỗi 6 số là đối xứng!

"Câu hỏi là, lần đầu tôi nhìn công-tơ-mét thì số chỉ của nó bằng bao nhiêu?"

Bài tập 8

Hãy viết một chương trình Python để kiểm tra tất cả các số gồm sáu chữ số sau đó in ra những số thỏa mãn tất cả các điều kiện như vậy. Bạn có thể xem lời giải của tôi tại `thinkpython.com/code/cartalk.py`.

Sau đây là một câu đố khác trong chương trình *Car Talk* mà bạn có thể giải bằng cách tìm kiếm [4]:

> "Gần đây tôi tới thăm mẹ và chúng tôi nhận ra rằng số tuổi của tôi khi viết ngược lại (đổi chỗ hai chữ số) sẽ trở thành tuổi mẹ. Chẳng hạn, nếu mẹ tôi 73 tuổi thì tôi 37. Chúng tôi đã tự hỏi liệu điều này xảy ra được mấy lần nhưng rồi chuyển sang câu chuyện khác mà không tìm ra câu trả lời.
>
> Khi quay trở về nhà tôi đã nhận ra rằng số tuổi của chúng tôi đã có sáu trường hợp đổi chỗ như thế. Đồng thời tôi còn tính được, nếu may mắn thì điều này sẽ xảy ra chỉ trong vài năm tới, và nếu rất may mắn thì nó còn xảy ra thêm một lần nữa. Tóm lại là tổng cộng 8 lần có thể xảy ra. Vậy câu hỏi là, bây giờ tôi đang bao nhiêu tuổi?"

Bài tập 9

Hãy viết một chương trình Python để dò tìm lời giải cho câu đố này. Gợi ý: bạn có thể tận dụng phương thức chuỗi có tên `zfill`.

Bạn có thể xem lời giải của tôi tại `thinkpython.com/code/cartalk.py`.

---

1. `wikipedia.org/wiki/Moby_Project`. ↩
2. `www.cartalk.com/content/puzzler/transcripts/200725`. ↩
3. `www.cartalk.com/content/puzzler/transcripts/200803`. ↩
4. `www.cartalk.com/content/puzzler/transcripts/200813` ↩

Chương 10: Danh sách

Trở về Mục lục cuốn sách

Danh sách là một dãy

Cũng như một chuỗi, **danh sách** cũng là một dãy các giá trị. Trong một chuỗi, các giá trị đó là những kí tự; trong một danh sách, chúng có thể thuộc kiểu bất kì. Các giá trị trong một danh sách được gọi là các **phần tử**.

Có một vài cách khác nhau để tạo ra một danh sách mới; cách đơn giản nhất là bao quanh các phần tử bằng một cặp ngoặc vuông (`[` và `]`):

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
```

Ở ví dụ thứ nhất bên trên, danh sách bao gồm 4 số nguyên. danh sách thứ hai thì có ba chuỗi. Các phần tử trong một danh sách không nhất thiết có cùng kiểu. danh sách sau đây bao gồm một chuỗi, một số có phần thập phân, một số nguyên, và (a ha!) một danh sách khác nữa:

```
['spam', 2.0, 5, [10, 20]]
```

Một danh sách nằm trong một danh sách khác thì được gọi là **lồng ghép**.

Một danh sách không chứa phần tử nào được gọi là danh sách trống; bạn có thể tạo ra một dánh sách trống bằng một cặp ngoặc kép liền nhau, `[]`.

Đúng như bạn có thể nghĩ đến, ta có thể gán các giá trị trong danh sách cho các biến:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']

>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

Cú pháp để thực hiện việc truy cập từng phần tử của một danh sách cũng giống như truy cập từng kí tự trong một chuỗi—đó là dùng toán tử ngoặc vuông. Biểu thức bên trong cặp ngoặc vuông xác định chỉ số. Chú ý rằng chỉ số bắt đầu từ 0:

```
>>> print cheeses[0]
Cheddar
```

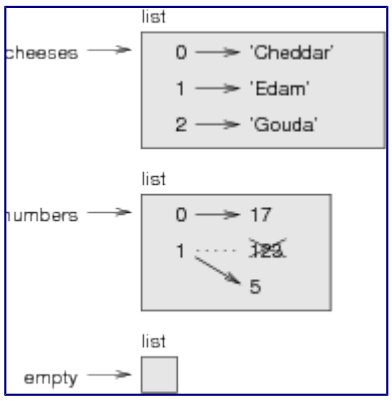
Khác với các chuỗi, danh sách là kiểu dữ liệu thay đổi được. Khi toán tử ngoặc vuông xuất hiện ở vế trái một lệnh gán, nó sẽ chỉ định phần tử của danh sách cần được gán.

```
>>> numbers = [17, 123]

>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

Phần tử có chỉ số 1 của `numbers`, trước đó là phần tử 123, giờ đã được thay bằng 5.

Bạn có thể hình dung danh sách như là một mối quan hệ giữa các chỉ số và phần tử. Mối quan hệ này được gọi là **đánh số**; mỗi chỉ số được "đánh" cho mỗi phần tử. Sau đây là một biểu đồ trạng

thái biểu thị `cheeses`, `numbers` và `empty`:



Các danh sách được biểu thị bởi những khối với từ "list" ghi bên ngoài và các phần tử ở bên trong nó. `cheeses` tương ứng với một danh sách với ba phần tử được đánh chỉ số 0, 1 và 2. `numbers` gồm có hai phần tử; và biểu đồ cho thấy rằng giá trị của phần tử thứ hai đã được gán lại từ 123 thành 5. `empty` ứng với một danh sách không chứa phần tử nào.

Các chỉ số trong dãy cũng có tác dụng như chỉ số của chuỗi:

- Bất kì một biểu thức số nguyên nào cũng có thể được dùng làm chỉ số.
- Nếu bạn cố gắng đọc hoặc ghi một phần tử mà bản thân nó không tồn tại, bạn sẽ gặp phải lỗi `IndexError`.
- Nếu một chỉ số có giá trị âm, nó sẽ được đếm ngược từ phía cuối danh sách.

Toán tử `in` cũng dùng được với các danh sách.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']

>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

Duyệt danh sách

Cách thông dụng nhất để duyệt các phần tử trong một danh sách là dùng một vòng lặp `for`. Cú pháp cũng tương tự như với chuỗi:

```
for cheese in cheeses:
    print cheese
```

Cách này sẽ tốt nếu bạn chỉ cần đọc các phần tử trong danh sách. Nhưng nếu bạn cần ghi hoặc cập nhật các phần tử trong danh sách, bạn phải dùng đến chỉ số. Một cách làm hay dùng là kết hợp các hàm `range` và `len`:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Vòng lặp này duyệt danh sách và cập nhật từng phần tử. `len` trả về số phần tử có trong danh sách. `range` trả về một danh sách các chỉ số từ 0 đến *n* - 1, trong đó *n* là chiều dài của danh sách. Mỗi lần đi qua vòng lặp, `i` sẽ nhận giá trị chỉ số của phần tử kế tiếp. Lệnh gán trong phần thân sẽ dùng `i` để đọc giá trị cũ của phần tử và gán giá trị mới.

Một vòng lặp `for` áp dụng với danh sách trống sẽ không bao giờ thực hiện phân thân câu lệnh:

```
for x in empty:
    print 'Không bao giờ được thực hiện.'
```

Mặc dù một danh sách có thể bao gồm một danh sách khác, danh sách được lồng ghép bên trong vẫn được tính như một phần tử đơn lẻ. Chiều dài của danh sách này là 4:

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

Các phép toán với danh sách

Toán tử + có tác dụng ghép nối các danh sách:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Cũng tương tự như vậy, toán tử * lặp lại một danh sách với số lần định trước:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Ví dụ thứ nhất lặp lại `[0]` bốn lần. Ví dụ thứ hai lặp lại danh sách `[1, 2, 3]` ba lần.

Lát cắt trong danh sách

Toán tử lát cắt cũng có tác dụng với các danh sách:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']

>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Nếu bạn bỏ qua chỉ số thứ nhất, lát cắt sẽ bắt đầu từ vị trí đầu tiên cảu danh sách. Nếu bạn bỏ qua chỉ số thứ hai, lát cắt sẽ kéo dài đến cuối danh sách. Vì vậy nếu bạn bỏ qua cả hai chỉ số, lát cắt sẽ là một bản sao của toàn bộ danh sách.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Vì các danh sách có tính thay đổi cho nên sao chép là việc ta nên làm trước khi thực hiện các thao tác gấp, xoay, hoặc chia nhỏ danh sách.

Một toán tử bên vế trái cảu phép gán có thể cập nhật cùng lúc nhiều phân tử:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']

>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

Python cung cấp các phương thức hoạt động với danh sách. Chẳng hạn, `append` thêm một phân tử mới vào cuối một danh sách:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
```

`extend` nhận vào danh sách như một tham biến và tiếp nối tất cả các phần tử chứa trong danh sách đó:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

Ở ví dụ trên, `t2` không bị thay đổi.

`sort` sắp xếp các phần tử của danh sách từ thấp lên cao:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

Các phương thức với danh sách đều trống; chúng chỉ thay đổi danh sách và trả về `None`. Nếu bạn viết `t = t.sort()`, bạn sẽ thất vọng với kết quả nhận được.

Map, filter và reduce

Để cộng tất cả các số có trong một danh sách lại, bạn có thể dùng vòng lặp như sau:

```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

`total` ban đầu được gán bằng 0. Mỗi lần chạy vòng lặp, `x` nhận một phần tử từ danh sách. Toán tử `+=` giúp ta cập nhật biến theo cách viết gọn gàng nhất. **Câu lệnh gán rút gọn** sau:

```
    total += x
```

tương đương với:

```
    total = total + x
```

Khi vòng lặp được thực hiện, `total` sẽ tích lũy tổng của các phần tử; một biến được dùng theo cách này đôi khi còn được gọi là **biến tích lũy**.

Việc cộng các phần tử trong một danh sách là thao tác rất thường gặp, và vì vậy Python có sẵn một hàm `sum`:

```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

Một thao tác như thế này, thực hiện dồn một dãy các phần tử vào một giá trị đơn lẻ, đôi khi còn

được gọi là **rút gọn**.

Đôi khi bạn muốn duyệt một danh sách trong khi đồng thời lại tạo một danh sách khác. Chẳng hạn, hàm sau đây nhận vào một danh sách các chuỗi và trả về một danh sách mới bao gồm các chuỗi với kiểu chữ in:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` được khởi tạo với một danh sách trống; mỗi lần chạy qua vòng lặp, ta thêm vào đó phần tử tiếp theo. Vì vậy, `res` cũng là một kiểu biến tích lũy khác.

Một thao tác kiểu như `capitalize_all` (chuyển chữ in toàn bộ) đôi khi được coi tương đương như việc **đánh số** vì nó dùng một hàm (trong trường hợp này là phương thức `capitalize`) để "đánh" cho mỗi phần tử trong một dãy.

Một thao tác thường gặp khác là chọn một số các phần tử từ danh sách và trả về một danh sách con. Chẳng hạn, hàm sau đây nhận vào một danh sách các chuỗi và sau đó trả về một danh sách trong đó chỉ bao gồm các chuỗi viết bằng chữ in:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` là một phương thức chuỗi trả về `True` nếu như chuỗi chỉ bao gồm các chữ cái viết in.

Một thao tác kiểu như `only_upper` được gọi là **lọc** vì nó chọn ra một số các phần tử đồng thời lọc bỏ các phần tử khác.

Các thao tác thông dụng nhất với danh sách hầu như đều có thể biểu diễn được dưới dạng kết hợp của đánh số, lọc và rút gọn. Vì những thao tác này quá thông dụng nên Python cung cấp cho ta nhưng đặc điểm ngôn ngữ để hỗ trợ chúng, bao gồm hàm có sẵn `map` và một toán tử có tên là "list comprehension".

> Hãy viết một hàm trong đó nhận vào một danh sách các số và trả về tổng lũy tích, nghĩa là một danh sách mới trong đó phần tử thứ *i* chính là tổng của *i* + 1 phần tử đầu trong danh sách nhận vào. Chẳng hạn, tổng lũy tích của `[1, 2, 3]` là `[1, 3, 6]`.

Xóa các phần tử

Có một vài cách làm khác nhau để xóa phần tử khỏi một danh sách. Nếu đã biết chỉ số của phần tử bạn cần xóa, bạn có thể dùng `pop`:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>> print x
b
```

`pop` thay đổi danh sách và trả lại phần tử mà ta vừa xóa khỏi danh sách. Nếu bạn không cung cấp

chỉ số, nó sẽ xóa và trả lại phần tử cuối cùng.

Nếu không cần giá trị cần được loại khỏi danh sách, bạn có thể dùng toán tử `del`:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

Nếu bạn biết phần tử cần được xóa (nhưng không biết chỉ số), bạn có thể dùng `remove`:

```
>>> t = ['a', 'b', 'c']

>>> t.remove('b')
>>> print t
['a', 'c']
```

Giá trị trả về từ `remove` là `None`.

Để xóa nhiều hơn một phần tử, bạn có thể dùng `del` với một lát cắt chỉ số:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print t
['a', 'f']
```

Như thường lệ, lát cắt sẽ chọn ra tất cả những phần tử cho đến trước (không bao gồm) chỉ số thứ hai.

Danh sách và chuỗi

Một chuỗi là một dãy các kí tự và một danh sách là một dãy các giá trị, nhưng một danh sách các kí tự lại không giống như một chuỗi. Để chuyển từ một chuỗi sang một danh sách các kí tự, bạn có thể dùng `list`:

```
>>> s = 'spam'

>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

Vì `list` cũng giống như tên của một hàm dựng sẵn, nên bạn cần tránh đặt nó làm tên một biến. Tôi cũng tránh dùng `l` vì nó trông rất giống như số `1`. Vì vậy tôi thường dùng `t`.

Hàm `list` phá vỡ một chuỗi thành các kí tự riêng lẻ. Nếu muốn phá vỡ chuỗi thành các từ riêng lẻ, bạn có thẻ dùng phương thức `split`:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
```

Bạn có thể kèm theo một đối số là một **dấu phân cách** chính là kí tự được dùng để ngăn cách giữa các từ. Ví dụ sau đây có sử dụng dấu gạch nối để phân cách:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

`join` là hàm ngược của `split`. Nó nhận vào một dãy các chuỗi và nối chúng lại. `join` là một phương thức chuỗi, vì vậy bạn phải gọi nó từ biến là dấu phân cách và sau đó truyền vào tham biến là danh sách:

```
>>> t = ['pining', 'for', 'the', 'fjords']

>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

Trong trường hợp này, biến phân cách (delimiter) là một kí tự dấu cách, vì vậy `join` đặt dấu cách giữa hai từ liên tiếp. Để nối liền các chuỗi mà không bị ngăn cách, bạn có thể dùng chuỗi rỗng, `''`, vào vai trò của dấu phân cách.

Đối tượng và giá trị

Nếu ta thực hiện các lệnh gán sau:

```
a = 'banana'
b = 'banana'
```

Ta biết rằng cả `a` và `b` đều tham chiếu đến một chuỗi, nhưng ta không biết rằng liệu chúng có tham chiếu đến *cùng* một chuỗi không. Sau đây là hai trường hợp có thể xảy ra:

Trong một trường hợp, `a` và `b` tham chiếu đến hai đối tượng khác nhau nhưng có cùng giá trị. Trong trường hợp thứ hai, chúng cùng tham chiếu đến một đối tượng.

Để kiểm tra xem liệu hai biến có cùng tham chiếu đến một đối tượng hay không, bạn có thể dùng toán tử `is`.

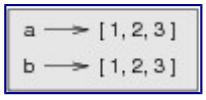
```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

Ở ví d này, Python chỉ tạo ra một đối tượng chuỗi, và cả `a` và `b` đều tham chiếu đến nó.

Nhưng khi bạn tạo ra hai danh sách, bạn sẽ thu được hai đối tượng riêng biệt:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

Vì vậy sơ đồ trạng thái sẽ trông như thế này:



Trong trường hợp này ta nói rằng hai danh sách là **tương đương** nhau, vì chúng có những phần tử giống nhau, nhưng không **đồng nhất**, vì chúng không cùng là một đối tượng. Nếu hai đối tượng đồng nhất nhau thì đồng thời cũng tương đương nhau, nhưng ngược lại nếu chúng tương đương thì không nhất thiết phải đồng nhất.

Đến giờ, ta đã dùng các khái niệm "đối tượng" và "giá trị" để thay thế được cho nhau, nhưng sẽ

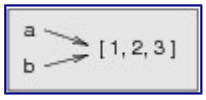
chính xác hơn nếu nói rằng mỗi đối tượng có một giá trị. Nếu bạn thực hiện biểu thức `[1,2,3]`, bạn sẽ có một đối tượng mà giá trị của nó là một chuỗi các số nguyên. Nếu một danh sách khác có những phần tử như vậy, ta nói rằng chúng có cùng giá trị, nhưng hai danh sách này vẫn là những đối tượng khác nhau.

Tham chiếu bội

Nếu `a` tham chiếu đến một đối tượng và bạn gán `b = a`, thì cả hai biến sẽ cùng chỉ đến một đối tượng:

```
>>> a = [1, 2, 3]
>>> b = a

>>> b is a
True
```

Sơ đồ trạng thái sẽ trông như thế này:



Việc gắn một biến với một đối tượng được gọi là **tham chiếu**. Ở ví dụ này, có hai tham chiếu đến cùng một đối tượng

Một đối tượng có nhiều tham chiếu thì cũng có nhiều tên, và chúng ta nói rằng đối tượng này được **tham chiếu bội**.

Nếu như đối tượng được tham chiếu bội có thể thay đổi được, thì những thay đổi thực hiện với tham chiếu này cũng sẽ ảnh hưởng với tham chiếu kia:

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

Mặc dù tính chất này có thể hữu dụng nhưng chắc chắn rất dễ gây lỗi. Nhìn chung, để an toàn ta nên tránh tham chiếu bội khi thao tác với các đối tượng thay đổi được.

Với các đối tượng không thay đổi như chuỗi, tham chiếu bội không còn là vấn đề. Trong ví dụ này:

```
a = 'banana'
b = 'banana'
```

Hầu như không có khác biệt nào trong việc liệu `a` và `b` có tham chiếu đến cùng một chuỗi hay không.

Danh sách với vai trò như đối số

Khi bạn truyền một danh sách vào trong hàm, thì hàm sẽ nhận được một tham chiếu đến danh sách đó. Nếu hàm thay đổi một tham biến danh sách, thì đoạn chương trình gọi sẽ thấy được sự thay đổi đó. Chẳng hạn, `delete_head` xóa phần tử đầu khỏi danh sách:

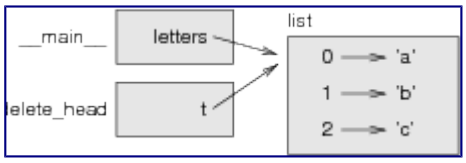
```
def delete_head(t):
    del t[0]
```

Đây là khi hàm này được dùng:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
```

```
>>> print letters
['b', 'c']
```

Tham biến `t` và biến `letters` là các tham chiếu bội đến cùng một đối tượng. Sơ đồ ngăn xếp sẽ trông như sau:



Vì danh sách được dùng chung bởi hai khung riêng biệt nên tôi đã vẽ nó giữa hai khung này.

Điều quan trọng là phân biết được các phép thao tác nhằm thay đổi danh sách và thao tác nhằm tạo những danh sách mới. Chẳng hạn, phương thức `append` thay đổi một danh sách, nhưng toán tử `+` lại tạo ra một danh sách mới:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None

>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3]
>>> t2 is t3
False
```

Khác biệt này có thể trở nên quan trọng khi bạn viết những hàm lẽ ra được dùng để thay đổi danh sách. Chẳng hạn, hàm sau đây *không* xóa được phần tử đầu của danh sách:

```
def bad_delete_head(t):
    t = t[1:]              # SAI!
```

Toán tử lát cắt sẽ tạo ra một danh sách mới và lệnh gán làm cho `t` tham chiếu đến đó, nhưng tất cả những điều này đều không thay đổi gì đến danh sách vốn được truyền vào với vai trò như một đối số.

Một cách làm khác là viết một hàm để tạo ra và trả lại một danh sách mới. Chẳng hạn, `tail` trả lại toàn bộ danh sách chỉ trừ phần tử đầu tiên:

```
def tail(t):
    return t[1:]
```

Hàm này sẽ giữ nguyên, không làm thay đổi danh sách ban đầu. Sau đây là một cách dùng nó:

```
>>> letters = ['a', 'b', 'c']

>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

> Hãy viết một hàm có tên là `chop` trong đó nhận vào một danh sách và thay đổi nó, xóa đi các phần tử đầu và phần tử cuối, rồi trả về `None`.
>
> Sau đó viết một hàm có tên là `middle` trong đó nhận vào một danh sách và trả về một

danh sách mới bao gồm tất cả những phần tử chỉ trừ hai phần tử đầu và cuối của danh sách gốc.

Gỡ lỗi

Sự bất cẩn khi dùng danh sách (và các đối tượng thay đổi được khác) có thể dẫn đến mất thời gian hàng giờ đồng hồ để gỡ lỗi. Sau đây là một số lỗi thường gặp và cách phòng tránh:

1. Đừng quên rằng hầu hết các phương thức làm việc với danh sách đều thay đổi danh sách (trong vai trò đối số) và trả về `None`. Điều này ngược lại với các phương thức chuỗi trong đó trả về một chuỗi mới và giữ nguyên chuỗi cũ.

   Nếu bạn quen viết mã lệnh với chuỗi như sau:

   ```
   word = word.strip()
   ```

   thì cũng rất dễ bị hấp dẫn bởi cách viết mã lệnh này đối với danh sách:

   ```
   t = t.sort()           # SAI!
   ```

   Vì `sort` trả về `None` nên thao tác tiếp theo thực hiện với `t` dường như sẽ bị hỏng.

   Trước khi dùng những phương thức và toán tử thực hiện trên danh sách, bạn cần đọc kĩ tài liệu và sau đó thử chúng ở chế độ tương tác lệnh. Các phương thức và toán tử có chung giữa danh sách và những kiểu dãy khác (như chuỗi) được mô tả ở docs.python.org/lib/typesseq.html. Các phương thức và toán tử chỉ áp dụng được cho những dãy thay đổi được có ở docs.python.org/lib/typesseq-mutable.html.

2. Chọn một cách viết mã lệnh và dùng nó một cách thống nhất.

   Một trong số các nguyên nhân gây ra rắc rối liên quan đến danh sách là có quá nhiều cách thực hiện cùng một công việc. Chẳng hạn để xóa một phần tử khỏi một danh sách, bạn có thể dùng `pop`, `remove`, `del`, hoặc thậm chí một câu lệnh gán với lát cắt.

   Để thêm vào một phần tử, bạn có thể dùng phương thức `append` hoặc toán tử `+` Nhưng đừng quên rằng hai cách sau đây là đúng:

   ```
   t.append(x)
   t = t + [x]
   ```

   và tất cả những cách viết sau đều sai:

   ```
   t.append([x])          # SAI!
   t = t.append(x)        # SAI!
   t + [x]                # SAI!
   t = t + x              # SAI!
   ```

   Hãy thử lại tất cả những ví dụ trên trong chế độ tương tác lệnh và nắm vững được tác dụng của chúng. Lưu ý rằng chỉ có lệnh cuối cùng mới gây ra một lỗi thực thi; còn ba lệnh trên nó đều hợp lệ, nhưng thực hiện việc mà chúng ta không cần đến.

3. Tạo ra các bản sao để tránh tham chiếu bội.

   Nếu muốn dùng một phương thức như `sort` để thay đổi đối số (là danh sách) nhưng cũng cần giữ cả danh sách ban đầu, bạn có thể tạo một bản sao.

   ```
   orig = t[:]
   ```

```
t.sort()
```

Ở ví dụ này bạn cũng có thể dùng hàm có sẵn `sorted`, với trả lại một danh sách mới, được sắp xếp, đồng thời giữ nguyên danh sách ban đầu. Nhưng trong trường hợp đó bạn cần tránh lấy `sorted` để đặt cho tên biến!

Thuật ngữ

danh sách:
: Một dãy các giá trị.

phần tử:
: Một trong số các giá trị của danh sách (hoặc một dãy nói chung).

chỉ số:
: Một giá trị số nguyên để chỉ định một phần tử trong danh sách.

danh sách lồng ghép:
: Một danh sách đóng vai trò là phần tử trong một danh sách khác.

duyệt danh sách:
: Cách truy cập tuần tự từng phần tử trong danh sách.

đánh số:
: Mối quan hệ trong đó từng phần tử của một tập hợp tương ứng với một phần tử trong tập hợp khác. Chẳng hạn, một danh sách là một phép đánh số giữa các chỉ số với các phần tử.

biến tích lũy:
: Một biến được dùng trong vòng lặp để cộng dồn hoặc, nói chung là tích lũy để thu được một kết quả.

gán rút gọn:
: Một lệnh nhằm cập nhật giá trị của một biến bằng cách dùng toán tử kiểu như `+=`.

rút gọn:
: Một dạng mẫu xử lý trong đó bao gồm duyệt một dãy và tính tích lũy với từng phần tử để gộp thành một kết quả cuối cùng.

map:
: Một dạng mẫu xử lý duyệt chuỗi và thực hiện thao tác tính toán với từng phần tử.

lọc:
: Một dạng mẫu xử lý duyệt chuỗi và lựa chọn những phần tử thỏa mãn một điều kiện nào đó.

đối tượng:
: Thứ mà biến có thể tham chiếu đến được. Một đối tượng có kiểu và giá trị riêng của nó.

tương đương:
: Có cùng giá trị.

đồng nhất:
Cùng là một đố tượng (và mặc nhiên là tương đương).

tham chiếu:
Sự liên hệ giữa một biến và giá trị của nó.

tham chiếu bội:
Trường hợp trong đó hai hoặc nhiều biến cùng tham chiếu đến một đối tượng.

dấu phân cách:
Một kí tự hoặc chuỗi được dùng để chỉ định những chỗ mà chuỗi cho trước cần được tách.

Bài tập

Hãy viết một hàm có tên là `is_sorted` trong đó nhận vào tham biến là một danh sách và trả về `True` nếu danh sách đã được xếp teho thứ tự tăng dần và `False` trong trường hợp còn lại. Bạn có thể giả sử rằng (qua khâu xử lý sơ bộ) danh sách chứa các phần tử có thể so sánh được bằng các toán tử quan hệ <, >, v.v.

Chẳng hạn, `is_sorted([1,2,2])` sẽ trả về `True` và `is_sorted(['b','a'])` trả về `False`.

Hai từ là đảo của nhau nếu bạn có thể đảo các chữ cái trong từ này để hình thành nên từ kia. Hãy viết một hàm có tên là `is_anagram` nhận vào hai chuỗi và trả về `True` nếu chúng là đảo của nhau.

Nghịch lý "Ngày sinh nhật":

> Hãy viết một hàm có tên là `has_duplicates` trong đó nhận vào một danh sách và trả về `True` nếu có bất kì phần tử nào xuất hiện nhiều hơn một lần. Danh sách ban đầu không được thay đổi.
>
> Nếu như có 23 sinh viên trong lớp học của bạn, khả năng có được hai người trùng ngày sinh với nhau là bao nhiêu? Bạn có thể ước lượng khả năng (xác suất) này bằng cách phát sinh ra mẫu ngẫu nhiên gồm 23 ngày sinh rồi kiểm tra sự trùng lặp. Gợi ý: Bạn có thể phát sinh những ngày sinh ngẫu nhiên bằng hàm `randint` trong module `random`.

Bạn có thể tìm hiểu thêm về bài toán này tại wikipedia.org/wiki/Birthday_paradox, và có thể xem lời giải của tôi tại thinkpython.com/code/birthday.py.

Hãy viết một hàm có tên là `remove_duplicates` trong đó nhận vào một danh sách và trả về một danh sách mới chỉ gồm các phần tử duy nhất (theo nghĩa không trùng nhau) từ danh sách gốc. Gợi ý: chúng không nhất thiết phải xếp theo cùng thứ tự.

Hãy viết một hàm để đọc file `words.txt` và tạo nên một danh sách với mỗi phần tử cho một từ. Hãy viết hai dạng khác nhau của hàm này, một dạng dùng phương thức `append` và dạng kia dùng cách viết `t = t + [x]`. Dạng nào chạy tốn thời gian

hơn? Tại sao?

Bạn có thể xem lời giải của tôi tại `thinkpython.com/code/wordlist.py`.

Để kiểm tra xem một từ có trong danh sách các từ hay không, bạn có thể dùng toán tử `in`, nhưng cách này sẽ rất chậm vì nó phải tìm qua danh sách theo đúng thứ tự.

Vì các từ đã được xếp theo thứ tự ABC, nên chúng ta có thể tăng tốc bằng cách tìm kiếm nhị phân, điều này cũng hơi giống với cách ta tra từ điển. Ta bắt đầu bằng việc mở giữa quyển từ điển và kiểm tra xem liệu từ cần tra có đứng trước từ nằm giữa trong danh sách hay không. Nếu đúng vậy, ta tìm kiếm nửa đầu của từ điển theo cách tương tự. Còn nếu không đúng, ta sẽ tìm kiếm nửa cuối.

Dù trường hợp nào xảy ra đi nữa, phạm vi tìm kiếm đã giảm xuống chỉ còn một nửa. Nếu danh sách từ ban đầu có 113.809 từ, thì chỉ cần khoảng 17 bước là tìm ra được từ hoặc kết luận rằng từ cần tìm không có trong danh sách.

Hãy viết một hàm có tên là `bisect` nhận vào một danh sách đã được sắp xếp và giá trị cần tìm, sau đó trả lại chỉ số của giá trị trong danh sách nếu có, hoặc là `None` nếu không.

Hoặc bạn cũng có thể đọc tài liệu viết về module `bisect` rồi áp dụng nó!

Hai từ được gọi là một "cặp đảo" nếu như từ này là dạng viết dảo ngược của từ kia. Hãy viết một chương trình để tìm ra tất cả những cặp từ dảo có trong danh sách từ.

Hai từ được gọi là "đan cài" nhau nếu như lần lượt lấy mỗi chữ cái của từng từ ta xếp lại được một từ mới[1]. Chẳng hạn, "shoe" và "cold" đan cài nhau để tạo ra "schooled".

1. Hãy viết một chương trình tìm ra tất cả những cặp từ đan cài. Gợi ý: đừng phát sinh đầy đủ tất cả các cặp từ để kiểm tra!
2. Bạn có thể tìm được từ đan cài ba lần; nghĩa là chia lần lượt các chữ cái thành ba nhóm theo đúng thứ tự thì ta được ba từ khác nhau?

---

1. Bài tập này được bắt nguồn từ một ví dụ trong trang `puzzlers.org`. ↩

Chương 11: Từ điển

Trở về Mục lục cuốn sách

Từ điển giống như một danh sách, nhưng khái quát hơn. Trong một danh sách, các chỉ số phải là số nguyên; trong từ điển chúng có thể thuộc về (gần như) bất kì kiểu dữ liệu nào.

Bạn có thể nghĩ về từ điển như một phép ánh xạ từ một tập hợp các chỉ số (cũng gọi là **khóa**) và một tập hợp các giá trị. Mỗi khóa được đánh cho một giá trị. Sự liên hệ giữa khóa và giá trị được gọi là một **cặp khóa-trị** và đôi khi được gọi là một **mục**.

Ví dụ như chúng ta lập một từ điển ánh xạ giữa các từ tiếng Anh và tiếng Tây Ban Nha, vì vậy tất cả các khóa và các giá trị đều là những chuỗi.

Hàm `dict` sẽ tạo ra một từ điển mới mà không có mục nào. Vì `dict` là tên của một hàm có sẵn, bạn không nên lấy tên này để đặt cho một biến.

```
>>> eng2sp = dict()

>>> print eng2sp
{}
```

Cặp ngoặc nhọn, `{}`, biểu thị một từ điển rỗng. Để thêm các mục vào trong một từ điển, ta có thể dùng cặp ngoặc vuông:

```
>>> eng2sp['one'] = 'uno'
```

Dòng lệnh này tạo ra một mục trong đó ánh xạ khóa `'one'` đến giá trị `'uno'`. Nếu in lại nội dung từ điển, ta sẽ thấy một cặp khóa-trị với dấu hai chấm ngăn cách giữa phần khóa và phần trị:

```
>>> print eng2sp
{'one': 'uno'}
```

Hình thức hiển thị kết quả như vậy cũng dùng được để nhập dữ liệu. Chẳng hạn, bạn có thể tạo một từ điển mới gồm ba mục:

```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
```

Nhưng nếu in `eng2sp`, có thể bạn sẽ thấy ngạc nhiên:

```
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

Thứ tự của các cặp khóa-trị không còn giữ nguyên nữa. Thật ra, nếu bạn gõ lại ví dụ trên vào máy tính của mình, có thể kết quả còn khác nữa. Nói chung, không thể biết trước thứ tự các mục trong một từ điển.

Tuy nhiên điều đó chẳng quan trọng vì các phần tử trong từ điển không bao giờ được chỉ định bởi các chỉ số nguyên. Thay vào đó, bạn dùng các khóa để tra tìm những giá trị tương ứng:

```
>>> print eng2sp['two']
'dos'
```

Khóa `'two'` luôn được ánh xạ đến giá trị `'dos'` vì vậy thứ tự của các mục không quan trọng.

Nếu như khóa cần tìm không xuất hiện trong từ điển, bạn sẽ nhận được một biệt lệ:

```
>>> print eng2sp['four']
```

```
KeyError: 'four'
```

Hàm `len` có tác dụng trên từ điển; nó trả lại số cặp khóa-trị:

```
>>> len(eng2sp)
3
```

Toán tử `in` cũng có tác dụng trên từ điển; nó cho chúng ta biết liệu có một *khóa* trong từ điển với tên gọi cho trước không (chứ không phải là có giá trị với tên gọi như vậy).

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

Để xem rằng có một giá trị nào đó có trong từ điển với tên gọi cho trước không, bạn có thể dùng phương thức `values`, để trả về các giá trị như một danh sách, và sau đó dùng toán tử `in`:

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

Toán tử `in` dùng các thuật toán khác nhau đối với danh sách và với từ điển. Đối với danh sách, nó dùng một thuật toán tìm kiếm, như trong Mục find. Khi danh sách dài hơn, thời gian tìm kiếm cũng tăng lên tỉ lệ thuận với độ dài. Đối với từ điển, Python dùng một thuật toán có tên là **bảng băm** (hashtable) với một tính chất ưu việt: toán tử `in` được thực hiện với thời gian gần như không đổi bất kể độ dài của từ điển là bao nhiêu. Tôi sẽ không giải thích làm thế nào để có được điều đó, nhưng bạn có thể xem thêm ở wikipedia.org/wiki/Hash_table.

> Hãy viết một hàm để đọc vào các từ trong `words.txt` và lưu chúng như những khóa trong một từ điển. Các giá trị bằng bao nhiêu thì không quan trọng. Sau đó bạn có thể dùng toán tử `in` để kiểm tra nhanh xem một chuỗi cần tìm có trong từ điển hay không.
>
> Nếu bạn đã làm Bài tập wordlist1, bạn có thể so sánh tốc độ của lời giải này với toán tử `in` dành cho danh sách cùng phép tìm kiếm nhị phân.

Từ điển như một tập hợp các biến đếm

Giả sử như bạn có một chuỗi và cần đến xem trong chuỗi đó, mỗi chữ cái xuất hiện bao nhiêu lần. Có một vài cách để làm điều đó:

1. Bạn có thể tạo ra 26 biến, mỗi biến cho một chữ cái trong bảng. Sau đó bạn duyệt chuỗi và với mỗi kí tự, tăng biến đếm tương ứng lên một đơn vị, có thể dùng một câu lệnh điều kiện nhiều nhánh.
2. Bạn có thể tạo ra một danh sách gồm 26 phần tử. Sau đó bạn chuyển đổi mỗi chữ cái thành một số (dùng hàm có sẵn `ord`), dùng số này như là một chỉ số trong danh sách, và tăng biến đếm tương ứng lên.
3. Bạn có thể tạo ra một từ điển với các kí tự như những khóa và biến đếm như những giá trị tương ứng. Lần đầu khi bắt gặp một kí tự, bạn thêm một mục vào từ điển. Những lần sau đó thì tăng dần biến đếm cho những mục đã có trong từ điển.

Các cách làm trên đây đều giống nhau về thao tác tính toán, nhưng khác nhau về cách thực hiện tính toán đó.

Cách thực hiện là một phương pháp tính toán; một số cách thực hiện tốt hơn số còn lại. Chẳng hạn,

thực hiện kiểu từ điển có lợi rằng chúng ta không cần biết trước là chữ cái nào sẽ xuất hiện trong chuỗi và chỉ cần dành chỗ cho những chữ cái thực sự xuất hiện.

Đoạn mã có thể được viết như sau:

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

Tên của hàm là **histogram**, chính là một thuật ngữ ngành thống kê để chỉ tập hợp các tần số.

Dòng thứ nhất của hàm nhằm tạo ra một từ điển rỗng. Vòng lặp `for` làm nhiệm vụ duyệt chuỗi. Qua mỗi vòng lặp, nếu kí tự `c` không có trong từ điển, ta sẽ tạo ra một mục mới với khóa `c` và giá trị ban đầu bằng 1 (vì ta đã bắt gặp kí tự này một lần). Nếu `c` đã có trong từ điển thì ta sẽ tăng thêm 1 cho `d[c]`.

Sau đây là tác dụng của nó:

```
>>> h = histogram('brontosaurus')

>>> print h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

Bảng phân bố tần số này cho thấy các chữ cái `'a'` và `'b'` xuất hiện một lần; `'o'` xuất hiện hai lần, v.v.

> Từ điển có một phương thức gọi là `get` nhận vào một khóa và một giá trị mặc định. Nếu khóa này có xuất hiện trong từ điển, `get` sẽ trả lại giá trị tương ứng; ngược lại nó sẽ trả về giá trị mặc định nói trên. Chẳng hạn:
>
> ```
> >>> h = histogram('a')
>
> >>> print h
> {'a': 1}
> >>> h.get('a', 0)
> 1
> >>> h.get('b', 0)
> 0
> ```
>
> Hãy dùng `get` để viết `histogram` theo cách gọn gàng hơn. Bạn cần loại bỏ bớt một lệnh `if`.

Thao tác lặp trong từ điển

Nếu bạn dùng một từ điển trong một lệnh `for`, nó sẽ duyệt các khóa của từ điển này. Chẳng hạn, `print_hist` sẽ in các khóa cùng với giá trị tương ứng:

```
def print_hist(h):
    for c in h:
        print c, h[c]
```

Và kết quả đầu ra sẽ có dạng sau:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Một lần nữa ta thấy các khóa không xếp theo thứ tự cụ thể nào.

Từ điển có một phương thức gọi là `keys` để trả về các khóa của bản thân nó, dưới dạng một danh sách không xếp theo thứ tự.

Hãy sửa `print_hist` để in ra các khóa và giá trị theo thứ tự bảng chữ cái.

Tra ngược

Cho trước một từ điển `d` và khóa `k`, ta dễ dàng tìm được giá trị tương ứng `v = d[k]`. Thao tác này được gọi là **tra**.

Nhưng nếu bạn có `v` và muốn tìm `k` thì sao? Bạn gặp phải hai vấn đề: thứ nhất, có thể có nhiều khóa tương ứng với giá trị `v`. Tùy từng trường hợp cụ thể, bạn có thể chọn một hoặc lập một danh sách chứa tất cả các khóa đó. Thứ hai, không có một dạng cú pháp đơn giản nào giúp **tra ngược**; bạn phải thực hiện tìm kiếm.

Sau đây là một hàm nhận vào một giá trị và trả về khóa đầu tiên ánh xạ tới giá trị đó:

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise ValueError
```

Hàm này là một ví dụ nữa minh họa cho dạng mẫu tìm kiếm, nhưng nó có một đặc điểm mà trước đây ta chưa từng bắt gặp, đó là `raise`. Lệnh `raise` gây ra một biệt lệ; trong trường hợp này là `ValueError`, nói chung thường được dùng để chỉ rằng có điều gì đó không ổn với giá trị của một tham biến.

Nếu ta đến cuối vòng lặp, nghĩa là `v` không xuất hiện trong từ điển như một giá trị, thì ta sẽ gây ra một biệt lệ.

Sau đây là một ví dụ tra ngược thành công:

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)

>>> print k
r
```

Và một ví dụ không thành công:

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in reverse_lookup
ValueError
```

Kết quả nhận được khi bạn gây ra biệt lệ cũng giống như khi Python gây ra: sẽ có thông báo lỗi cùng việc dò ngược về vị trí xảy ra lỗi.

Lệnh `raise` nhận một tham biến tùy chọn là thông bào lỗi cụ thể. Chẳng hạn:

```
>>> raise ValueError, 'gia tri khong xuat hien trong tu dien'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: gia tri khong xuat hien trong tu dien
```

Việc tra ngược chậm hơn nhiều so với tra xuôi; nếu bạn phải thường xuyên thực hiện thao tác này, hoặc khi từ điển rất lớn, chương trình sẽ chạy chậm.

> Hãy sửa `reverse_lookup` sao cho nó lập và trả về một danh sách của *tất cả* các khóa ánh xạ đến `v`, hoặc một danh sách rỗng nếu không có khóa nào như vậy.

Từ điển và danh sách

Danh sách có thể đóng vai trò làm giá trị trong từ điển. Chẳng hạn, nếu bạn có một từ điển ánh xạ những chữ cái đến tần số xuất hiện của chúng, một việc có thể làm là đảo ngược nó; nghĩa là tạo ra một từ điển ánh xạ từ tần số đến chữ cái. Vì một số chữ cái có thể có cùng tần số, nên mỗi giá trị trong từ điển ngược phải là một danh sách chữ cái.

Sau đây là một hàm để đảo ngược từ điển:

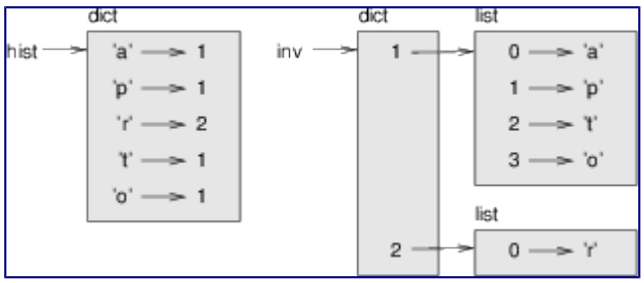
```
def invert_dict(d):
    inv = dict()
    for key in d:
        val = d[key]
        if val not in inv:
            inv[val] = [key]
        else:
            inv[val].append(key)
    return inv
```

Qua mỗi vòng lặp, `key` nhận một khóa từ `d` và `val` nhận giá trị tương ứng. Nếu `val` không có trong `inv`, nghĩa là chúng ta chưa từng bắt gặp nó, vì vậy chúng ta sẽ tạo ra một mục mới và khởi tạo nó là một **danh sách đơn** (một danh sách chỉ chứa một phần tử). Ngược lại nếu đã thấy giá trị này từ trước, ta sẽ điền thêm khóa tương ứng vào danh sách.

Sau đây là một ví dụ:

```
>>> hist = histogram('parrot')

>>> print hist
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inv = invert_dict(hist)
>>> print inv
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

Và sau đây là một sơ đồ chỉ `hist` và `inv`:



Từ điển được biểu diễn bởi một khung vời kiểu `dict` phía trên nó và các cặp khóa-trị bên trong. Nếu như các giá trị là số nguyên, số có phần thập phân hoặc chuỗi, tôi thường vẽ chúng bên trong khung. Còn danh sách thì tôi thường vẽ bên ngoài khung, để giữ cho sơ đồ được đơn giản.

Danh sách có thể là giá trị trong từ điển, như trong ví dụ này, nhưng không thể là khóa. Sau đây là hậu quả nếu bạn thử điều này:

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

Trước đây tôi đã đề cập rằng một từ điển được thực hiện bằng cách dùng một mảng băm và điều đó có nghĩa là các khóa phải **băm được**.

Một **hàm băm** là một hàm nhận vào một giá trị (có kiểu bất kì) và trả lại một số nguyên. Từ điển sử dụng những số nguyên này, gọi là các giá trị hash, để lưu trữ và tra những cặp khóa-trị.

Hệ thống này làm việc tốt nếu các khóa không thay đổi được. Nhưng nếu các khóa thay đổi được, như danh sách, thì mọi việc tồi tệ có thể xảy ra. Chẳng hạn, khi bạn tạo ra một cặp khóa-trị, Python tính hàm băm cho các khóa và lưu trữ chúng ở những địa chỉ tương ứng. Nếu bạn thay đổi khóa và lại băm chúng thì sẽ đến những địa chỉ khác. Trong trường hợp đó bạn có thể có hai mục cho cùng một khóa, hoặc có thể không tìm được khóa; và dù thế nào thì từ điển cũng không hoạt động đúng.

Đó là lí do tại sao khóa phải băm được, và tại sao các kiểu dữ liệu thay đổi được như danh sách thì không dùng được làm khóa. Cách đơn giản nhất để khắc phục hạn chế này là dùng các *bộ*, mà ta sẽ xét đến trong chương sau.

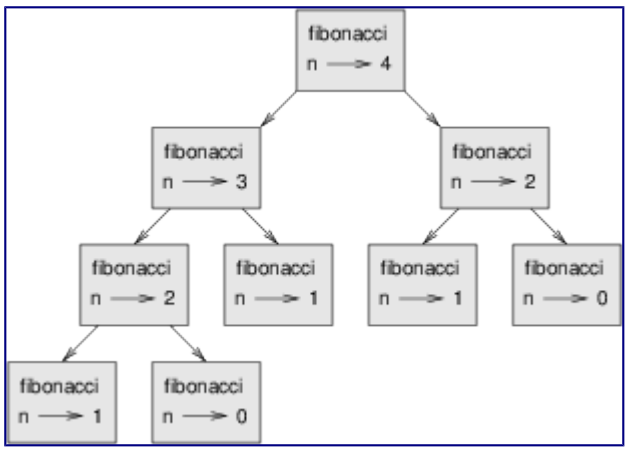
Vì từ điển có tính thay đổi được, bản thân chúng không thể dùng làm khóa, nhưng chúng *có thể* dùng vào vai trò giá trị.

> Hãy đọc tài liệu về phương thức `setdefault` của từ điển và dùng nó để viết một dạng gọn hơn cho `invert_dict`.

Giá trị nhớ

Nếu đã thử nghịch hàm `fibonacci` ở Mục one more example, bạn có thể nhận thấy đối số bạn nhập vào càng lớn thì chương trình chạy càng lâu. Hơn nữa, thời gian chạy của chương trình tăng vọt một cách đáng kể.

Để hiểu được tại sao, hãy xét **đồ thị gọi** này của hàm `fibonacci` với `n=4`:



Một đồ thị gọi cho thấy một loạt những khung biểu diễn hàm, với các đường nối giữa khung hàm này với các khung hàm mà nó gọi đến. Trên đỉnh của đồ thị, `fibonacci` với `n=4` gọi `fibonacci` với `n=3` và `n=2`. Đến lượt mình, `fibonacci` với `n=3` lại gọi `fibonacci` với `n=2` và `n=1`. Và cứ như vậy.

Hãy đến xem có bao nhiêu lần `fibonacci(0)` và `fibonacci(1)` được gọi đến. Đây là một lời giải rất kém hiệu quả, và càng tồi hơn khi đối số càng lớn.

Một cách giải quyết là theo dõi những giá trị đã được tính bằng cách lưu nó vào trong một từ điển. Một giá trị đã tính trước rồi được lưu để sau này dùng được gọi là một **giá trị nhớ**[1]. Sau đây là một cách thực hiện `fibonacci` có dùng giá trị nhớ:

```
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
```

`known` là một từ điển theo dõi các số Fibonacci mà ta đã biết. Nó bắt đầu với hai mục: 0 ánh xạ đến 0 và 1 ánh xạ đến 1.

Mỗi khi `fibonacci` được gọi, nó sẽ kiểm tra `known`. Nếu kết quả đã có ở đấy rồi, nó có thể trả lại lập tức. Còn nếu không, nó phải tính giá trị mới, bổ sung vào từ điển, và trả lại giá trị này.

> Hãy chạy hàm `fibonacci` trên và phiên bản ban đầu của hàm này với một loạt các tham số rồi so sánh thời gian thực hiện của chúng.

Biến toàn cục

Trong ví dụ trước, `known` được tạo ra bên ngoài hàm, vì vậy nó thuộc về một khung đặc biệt có tên là `__main__`. Các biến trong `__main__` đôi khi được gọi là **toàn cục** vì chúng có thể được truy cập từ bất kì hàm nào. Khác với các biến địa phương, mà sẽ biến mất ngay sau khi hàm kết thúc, các biến toàn cục vẫn tồn tại từ một lần gọi hàm sang đến lần gọi tiếp theo.

Các biến toàn cục thường được dùng làm **cờ**; tức là các biến kiểu boole để chỉ định (như "lá cờ") xem môt điều kiện có đúng hay không. Chẳng hạn, một số chương trình dùng một cờ có tên là `verbose` để điều khiển mức độ chi tiết của kết quả được xuất ra:

```
verbose = True
```

```
def example1():
    if verbose:
        print 'Running example1'
```

Nếu bạn thử gán lại giá trị cho một biến toàn cục, bạn có thể sẽ ngạc nhiên. Ví dụ sau đây vốn được viết với ý định theo dõi xem hàm có được gọi hay chưa:

```
been_called = False

def example2():
    been_called = True         # SAI
```

Nhưng nếu chạy nó, bạn sẽ thấy giá trị của `been_called` không hề thay đổi. Vấn đề là hàm `example2` tạo ra một biến địa phương khác có tên là `been_called`. Biến địa phương sẽ biến mất ngay khi hàm kết thúc và không ảnh hưởng gì đến biến toàn cục.

Để gán lại giá trị cho một biến toàn cục bên trong một hàm, bạn phải **khai báo** biến toàn cục trước khi sử dụng nó:

```
been_called = False

def example2():
    global been_called
    been_called = True
```

Lệnh `global` báo cho trình thông dịch biết một thông tin kiểu như, "Trong hàm này, khi nói đến `been_called`, ý của tôi là biến toàn cục; đừng tạo ra một biến địa phương".

Sau đây là một ví dụ cố gắng cập nhật một biến toàn cục:

```
count = 0

def example3():
    count = count + 1          # SAI
```

Nếu chạy nó bạn sẽ thu được:

```
UnboundLocalError: local variable 'count' referenced before assignment
```

Python coi rằng `count` là biến địa phương, nghĩa là bạn sẽ đọc biến này trước khi ghi giá trị vào nò. Để sửa lỗi sai này, một lần nữa bạn phải khai báo `count` là biến toàn cục.

```
def example3():
    global count
    count += 1
```

Nesu biến toàn cục thuộc kiểu thay đổi được, bạn có thể sửa nó mà không phải khai báo:

```
known = {0:0, 1:1}

def example4():
    known[2] = 1
```

Như vậy bạn có thể thêm vào, bỏ bớt, và thay thế các phần tử của một danh sách hoặc từ điển toàn cục, nhưng nếu bạn muốn gán lại giá trị cho biến, bạn phải khai báo nó:

```
def example5():
    global known
```

```
    known = dict()
```

Số nguyên dài

Nếu tính `fibonacci(50)`, bạn sẽ thu được:

```
>>> fibonacci(50)
12586269025L
```

Chữ `L` ở cuối chỉ định rằng kết quả có dạng số nguyên dài [2], hoặc kiểu `long`.

Các giá trị thuộc kiểu `int` bị giới hạn về độ lớn; các số nguyên dài có thể lớn tùy ý, nhưng khi lớn lên chúng sẽ chiếm nhiều dung lượng bộ nhớ và cần nhiều thời gian xử lý.

Các toán tử số học và các hàm trong module `math` cũng dùng được với số nguyên dài, vì vậy nói chung bất kì đoạn mã nào chạy được với số nguyên `int` cũng chạy được với `long`.

Bất cứ khi nào kết quả tính toán quá lớn không thể biểu diễn được bởi số nguyên thường, Python sẽ tự chuyển sang dạng số nguyên dài:

```
>>> 1000 * 1000
1000000

>>> 100000 * 100000
10000000000L
```

Trong trường hợp đầu, kết quả có kiểu `int`; ở trường hợp thứ hai là kiểu `long`.

> PHép lũy thừa các số nguyên lớn là cơ sở của các thuật toán thông dụng để mã hóa khóa công khai. Hãy đọc trang web Wikipedia về thuật toán RSA[3] và viết các hàm thực hiện mã hóa và giải mã một đoạn tin.

Gỡ lỗi

Khi bạn làm việc với các bộ dữ liệu lớn, việc gỡ lỗi bằng cách in ra và kiểm tra dữ liệu thủ công là việc không tưởng. Sau đây là một số gợi ý cho việc gỡ lỗi các bộ dữ liệu lớn:

Thu nhỏ dữ liệu đầu vào:
> Nếu có thể, hãy thu gọn kích thước của bộ dữ liệu. Chẳng hạn, nếu chương trình cần đọc một file kí tự, hãy bắt đầu bằng việc đọc 10 dòng đầu tiên, hoặc với một ví dụ nhỏ nhất mà bạn có thể kiếm được. Bạn có thể tự tạo ra file, hoặc (tốt hơn là) sửa đổi chương trình để nó chỉ đọc `n` dòng đầu tiên.
>
> Nếu có lỗi xảy ra, bạn có thể giảm `n` xuống đến giá trị nhỏ nhất mà còn có lỗi, sau đó tăng dần lên đồng thời tiến hành tìm kiếm và sửa các lỗi.

Kiểm tra các kết quả tóm tắt và kiểu các biến:
> Thay vì in và kiểm tra toàn bộ dữ liệu, bạn có thể cân nhắc in ra phần tóm lược cho dữ liệu thôi; chẳng hạn, số các mục trong một từ điển hoặc tổng của các số trong một danh sách.
>
> Một nguyên nhân thường gây ra lỗi thực thi là khi một giá trị không nhận được kiểu đúng. Để gỡ những lỗi dạng này, thường ta chỉ cần in ra kiểu của một giá trị.

Viết chương trình kiểm tra:

Đôi khi bạn có thể viết đoạn mã lệnh để tự động kiểm tra lỗi. Chẳng hạn, nếu bạn tính trị trung bình của một dãy các số, bạn có thể kiểm tra rằng kết quả không lớn hơn giá trị lớn nhất trong dãy hoặc không nhỏ hơn giá trị nhỏ nhất. Đây được gọi là "kiểm tra độ lành mạnh" vì nó phát hiện được các kết quả vô lí hết mức.

Một dạng kiểm tra nữa nhằm so sánh kết quả giữa hai lần tính khác nhau để xem chúng có nhất quán không. Đây được gọi là "kiểm tra độ nhất quán".

In đẹp kết quả đầu ra:
: Việc định dạng kết qủa đầu ra khi gỡ lỗi có thể đơn giản hóa việc phát hiện lỗi. Ta đã thấy một ví dụ trong Mục factdebug. Module `pprint` cho ta một hàm `pprint` để hiển thị thông tin về các kiểu có sẵn theo hình thức dễ đọc hơn.

Một lần nữa, thời gian mà bạn dành cho việc dựng dàn giáo có thể giúp giảm bớt thời gian mất để gỡ lỗi.

Thuật ngữ

từ điển:
: Dạng ánh xạ từ một tập hợp các khóa đến các giá trị tương ứng của chúng.

cặp khóa-trị:
: Dạng biểu diễn ánh xạ từ một khóa đến một giá trị.

mục:
: Tên gọi khác cho cặp khóa-trị.

khóa:
: Một đối tượng xuất hiện trong từ điển với vai trò bộ phận đầu của một cặp khóa-trị.

trị:
: Một đối tượng xuất hiện trong từ điển với vai trò bộ phận thứ hai của một cặp khóa-trị. Khái niệm này đặc thù hơn so với "giá trị" mà ta dùng trước đây.

thực thi:
: Một cách tiến hành tính toán.

bảng băm:
: Thuật toán được dùng để thực thi các từ điển trong Python.

hàm băm:
: Hàm được dùng bởi một bảng băm để tính ra vị trí cho một khóa.

băm được:
: Kiểu dữ liệu có hàm băm. Các kiểu không thay đổi được như số nguyên, số có phần thập phân, và chuỗi đều băm được; các kiểu thay đổi được như danh sách và từ điển thì không.

tra:
: Thao tác trên từ điển, nhận vào một khóa và tìm ra trị tương ứng.

tra ngược:
: Thao tác trên từ điển, nhận vào một trị và tìm ra một hoặc nhiều khóa có ánh xạ đến nó.

danh sách đơn:
: Danh sách (hoặc một dãy nói chung) có một phần tử duy nhất.

biểu đồ gọi:
: Đồ thị biểu diễn tất cả các khung được tạo ra khi chạy một chương trình, với một mũi tên chỉ từ khung gọi đến khung được gọi.

histogram:
: Tập hợp các biến đếm.

giá trị nhớ:
: Giá trị sau khi tính toán được lưu trữ để tránh phải tính lại sau này.

biến toàn cục:
: Biến dược định nghĩa bên ngoài mọt hàm. Các biến toàn cục có thể được truy cập từ bất kì hàm nào.

cờ:
: Biến kiểu boole được dùng để chỉ định một điều kiện có đúng hay không.

khai báo:
: Lệnh kiểu như `global` để báo cho trình thông dịch biết thông tin về một biến.

Bài tập

Nếu bạn đã làm Bài tập duplicate, bạn đã có một hàm có tên `has_duplicates` để nhận vào một danh sách với vai trò tham biến và trả lại `True` nếu có bất kì đối tượng nào trong danh sách xuất hiện nhiều hơn một lần.

Hãy dùng một từ điển để viết một dạng nhanh hơn và đơn giản hơn cho `has_duplicates`.

Hai từ tạo thành một "cặp xoay" nếu như bạn xoay từ này để được từu kia (xem `rotate_word` ở Bài tập exrotate).

Hãy viết một chương trình đọc vào một danh sách các từ và tìm tất cả những cặp xoay.

Sau đây là một câu đố khác từ chương trình *Car Talk*: [4]

> Câu đố này được ông Dan O'Leary gửi đến. Ông bắt gặp một từ rất thông dụng; từ này có 5 chữ cái và chỉ một âm tiết, với đặc tính riêng như sau. Khi bạn bỏ đi chữ cái thứ nhất, các chữ cái còn lại sẽ hình thành một từ đồng âm với từ gốc. Bây giờ trả lại chữ cái thứ nhất và bỏ đi chữ cái thứ hai, một lần nữa ta lại thu được từ đồng âm với từ gốc. Câu hỏi là, từ gốc là gì?
>
> Bây giờ tôi sẽ lấy thí dụ một từ chưa đạt yêu cầu. Hãy xét từ gồm 5 chữ cái

> sau, ‘wrack.’ W-R-A-C-K. Nếu tôi bỏ đi chữ cái thứ nhất, tôi được một từ bốn chữ cái ’R-A-C-K.’ À đó là một từ đồng âm, theo tiếng Anh. Bây giờ nếu bạn hoàn trả lại chữ ‘w’, và bỏ đi chữ ‘r’, bạn sẽ còn lại từ ‘wack’, cũng là một từ có nghĩa, chỉ có điều là không đồng âm với hai từ trước.
>
> Tuy nhiên vẫn có ít nhất một từ mà Dan và chúng ta đều biết đến, một từ mà sẽ cho ra hai từ đồng âm khác gồm bốn chữ cái, khi ta bỏ đi lần lượt chữ cái thứ nhất và chữ cái thứ hai của từ gốc. Câu hỏi là, từ đó là gì?

Bạn có thể dùng từ điển trong Bài tập wordlist2 để kiểm tra xem một chuỗi có ở trong danh sách các từ hay không.

Để kiểm tra xem hai từ có đồng âm hay không, bạn có thể dùng Từ điển Phát âm CMU. Bạn có thể tải về từ `www.speech.cs.cmu.edu/cgi-bin/cmudict` hoặc `thinkpython.com/code/c06d` và bạn cũng có thể tải về `thinkpython.com/code/pronounce.py`, trong đó có hàm tên là `read_dictionary`. Hàm này đọc vào một từ điển phát âm và trả về một từ điển Python ánh xạ từ mỗi mục từ đến một chuỗi mô tả cách phát âm của nó.

Hãy viết một chương trình liệt kê tất cả các từ thỏa mãn yêu cầu của câu đố. Bạn có thể xem lời giải của tôi tại `thinkpython.com/code/homophone.py`.

---

1. Xem `wikipedia.org/wiki/Memoization`. ↩
2. Ở Python 3.0, kiểu `long` bị rút bỏ, tất cả số nguyên dù lớn đến đâu đều có kiểu `int`. ↩
3. `en.wikipedia.org/wiki/RSA`. ↩
4. `www.cartalk.com/content/puzzler/transcripts/200717`. ↩

Chương 12: Bộ

Trở về Mục lục cuốn sách

Bộ là kiểu dữ liệu không thay đổi

Bộ là một dãy các giá trị. Các giá trị có thể thuộc kiểu dữ liệu bất kì, và chúng đánh thứ tự bởi các số nguyên, và vì vậy bộ khá giống với danh sách. Điểm khác nhau quan trọng là ở chỗ bộ có tính không thay đổi.

Về mặt cú pháp, một bộ là một dãy các giá trị được phân cách bởi dấu phẩy:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Mặc dù không cần thiết, nhưng người ta thường dùng cặp ngoặc tròn để bao lại phần nội dung của một bộ:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

Để tạo ra một bộ với một phần tử duy nhất, bạn phải viết thêm một dấu phẩy ở phía cuối:

```
>>> t1 = 'a',
>>> type(t1)
<type 'tuple'>
```

Một giá trị ở trong cặp ngoặc tròn không phải là một bộ:

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

Một cách làm khác để tạo ra một bộ là dùng hàm có sẵn `tuple`. Khi không có đối số, nó sẽ tạo ra một bộ rỗng:

```
>>> t = tuple()
>>> print t
()
```

Nếu đối số là một dãy (như chuỗi, danh sách hoặc bộ), thì kết quả sẽ là một bộ có chứa các phần tử của dãy đó:

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
```

Vì `tuple` là tên của một hàm có sẵn, bạn cần tránh lấy nó đặt cho tên biến.

Hầu hết các toán tử với danh sách cũng có tác dụng đối với bộ. Toán tử ngoặc vuông để chỉ định một phần tử:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print t[0]
'a'
```

Và toán tử lát cắt để chọn một khoảng các phần tử.

```
>>> print t[1:3]
```

```
('b', 'c')
```

Nhưng nếu bạn thử thay đổi một phần tử trong bộ, bạn sẽ bị báo lỗi:

```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Bạn không thể sửa đổi được các phần tử của bộ, nhưng có thể thay thế một bộ với một bộ khác:

```
>>> t = ('A',) + t[1:]
>>> print t
('A', 'b', 'c', 'd', 'e')
```

Ta thường gặp tình huống cần tráo đổi giá trị của hai biến. Với câu lệnh gán truyền thống, bạn phải dùng một biến tạm thời. Chẳng hạn, để tráo đổi `a` với `b`:

```
>>> temp = a
>>> a = b
>>> b = temp
```

Cách giải này lủng củng; và **phép gán với bộ** sẽ đẹp hơn:

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

Vế trái là một bộ các biến; vế phải là một bộ các biểu thức. Mỗi giá trị được gán cho biến tương ứng với nó. Tất cả các biểu thức ở vế phải đều được định lượng trước khi thực hiện gán.

Số các biến ở vế trái và số các giá trị ở vế phải cần phải bằng nhau:

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

Tổng quát hơn, vế phải có thể là bất cứ dạng dãy nào (chuỗi, danh sách hoặc bộ). Chẳng hạn, để tách một địa chỉ email thành các phần tên người dùng và tên miền, bạn có thể viết:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

Giá trị trả về từ `split` là một dãy gồm hai phần tử; phần tử đầu được gán cho `uname`, phần tử thứ hai cho `domain`.

```
>>> print uname
monty
>>> print domain
python.org
```

Chặt chẽ mà nói, một hàm chỉ có thể trả về một giá trị, nhưng nếu giá trị đó là một bộ, thì tác dụng này cũng như là trả về nhiều giá trị. Chẳng hạn, nếu bạn muốn chia hai số tự nhiên để tìm ra thương và số dư, việc tính `x/y` trước rồi đến `x%y` là không hiệu quả. Cách tốt hơn là đồng thời tính chúng.

Hàm có sẵn `divmod` nhận vào hai đối số và trả về một bộ hai giá trị, là thương và số dư. Bạn có thể lưu kết quả dưới dạng một bộ:

```
>>> t = divmod(7, 3)
>>> print t
(2, 1)
```

Hoặc dùng phép gán với bộ để lưu các phần tử một cách riêng rẽ:

```
>>> quot, rem = divmod(7, 3)
```

```
>>> print quot
2
>>> print rem
1
```

Sau đây là ví dụ một hàm trả lại một bộ:

```
def min_max(t):
    return min(t), max(t)
```

`max` và `min` là các hàm có sẵn để tìm các phần tử lớn nhất và nhỏ nhất trong một dãy. `min_max` tính cả hai và trả lại một bộ hai giá trị.

Bộ đối số có độ dài thay đổi

Số lượng các đối số cho một hàm là thay đổi được. Một tham biến mà tên bắt đầu bởi * sẽ **thu gom** các đối số vào trong một bộ. Chẳng hạn, `printall` nhận vào một số lượng đối số tùy ý và in chúng ra:

```
def printall(*args):
    print args
```

Bạn có thể đặt tên tham biến dùng để thu gom là gì cũng được, nhưng `args` là một cái tên quy ước. Sau đây là cách dùng hàm này:

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

Thao tác bổ sung cho thu gom là **phát tán**. Nếu bạn có một dãy các giá trị và muốn truyền cho một hàm dưới dạng nhiều đối số, bạn có thể dùng toán tử *. Chẳng hạn, `divmod` nhận vào đúng hai tham số; nó sẽ không hoạt động với một bộ:

```
>>> t = (7, 3)
>>> divmod(t)
TypeError: divmod expected 2 arguments, got 1
```

Nhưng nếu bạn phát tán bộ đó thì câu lệnh sẽ có tác dụng:

```
>>> divmod(*t)
(2, 1)
```

Rất nhiều hàm có sẵn sử dụng bộ đối số có chiều dài thay đổi. Chẳng hạn, `max` và `min` có thể nhận bao nhiêu đối số cũng được:

```
>>> max(1,2,3)
3
```

Nhưng `sum` thì không thể.

```
>>> sum(1,2,3)
TypeError: sum expected at most 2 arguments, got 3
```

Hãy viết một hàm có tên `sumall` nhận vào bao nhiêu đối số cũng được và trả về tổng của chúng.

Danh sách và bộ

`zip` là một hàm có sẵn; nó nhận vào nhiều dãy và đan cài chúng vào một danh sách duy nhất[1] chứa các bộ trong đó mỗi bộ có một phần tử của mỗi dãy. (zip là tên gọi của phéc-mơ-tuya có hình ảnh giống với thao tác đan cài này.) Ví dụ sau đan cài một chuỗi và một danh sách:

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
[('a', 0), ('b', 1), ('c', 2)]
```

Kết quả là một danh sách các bộ trong đó mỗi bộ chứa một kí tự của chuỗi và phần tử tương ứng của danh sách ban đầu.

Nếu các dãy không có cùng độ dài thì kết quả sẽ dài bằng dãy ngắn hơn.

```
>>> zip('Anne', 'Elk')
[('A', 'E'), ('n', 'l'), ('n', 'k')]
```

Bạn có thể dùng phép gán bộ trong vòng lặp `for` để duyệt một danh sách các bộ:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print number, letter
```

Mỗi lần lặp, Python sẽ chọn ra bộ tiếp theo trong danh sách và gán các phần tử của nó cho `letter` và `number`. Kết quả đầu ra của vòng lặp này là:

```
0 a
1 b
2 c
```

Nếu kết hợp `zip`, `for` và phép gán bộ, bạn sẽ có một cách viết rất hữu ích để duyệt hai (hoặc nhiều) dãy cùng lúc. Chẳng hạn, `has_match` nhận vào hai dãy, `t1` và `t2`, rồi trả lại `True` nếu có một chỉ số `i` sao cho `t1[i] == t2[i]`:

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

Nếu cần duyệt các phần tử trong một dãy và các chỉ số của chúng, bạn có thể dùng hàm có sẵn `enumerate`:

```
for index, element in enumerate('abc'):
    print index, element
```

Kết quả đầu ra của vòng lặp này là:

```
0 a
1 b
2 c
```

Một lần nữa.

Từ điển và bộ

Từ điển có một phương thức tên là `items` để trả lại một danh sách các bộ, trong đó mỗi bộ là một cặp khóa-trị[2].

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> print t
[('a', 0), ('c', 2), ('b', 1)]
```

Với từ điển, như bạn có thể đoán được, các mục đều không có một thứ tự nào.

Ngược lại, bạn có thể dùng danh sách các bộ để khởi tạo một từ điển mới:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

Kết hợp `dict` với `zip` ta được một cách nhanh gọn để tạo ra một từ điển:

```
>>> d = dict(zip('abc', range(3)))
>>> print d
{'a': 0, 'c': 2, 'b': 1}
```

Phương thức `update` của từ điển cũng nhận vào một danh sách các bộ và gộp chúng lại, theo các cặp khóa-trị, vào từ điển sẵn có.

Kết hợp `items`, phép gán bộ và `for` lại với nhau, bạn có một cách viết chương trình duyệt các khóa và trị của từ điển:

```
for key, val in d.items():
    print val, key
```

Kết quả đầu ra của vòng lặp này là:

```
0 a
2 c
1 b
```

Giống như trước.

Bộ thường được dùng làm khóa trong từ điển (chủ yếu là vì bạn không thể dùng danh sách). Chẳng hạn, một danh bạ điện thoại có thể ánh xạ từ cặp họ, tên đến số điện thoại. Giả sử rằng ta đã định nghĩa `ho`, `ten` và `so`, ta có thể viết:

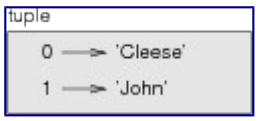
```
danhba[ho,ten] = so
```

Biểu thức trong ngoặc vuông là một bộ. Chúng ta có thể dùng phép gán cho bộ để duyệt từ điển này.

```
for ho,ten in danhba:
    print ho, ten, danhba[ho,ten]
```

Vòng lặp này duyệt các khóa trong `danhba`, vốn là các bộ. Nó gán các phần tử của từng bộ cho `ho` và `ten`, sau đó in ra họ tên và số điện thoại tương ứng.

Có hai cách biểu diễn bộ trong một sơ đồ trạng thái. Cách chi tiết hơn chỉ rõ các chỉ số và phần tử giống như với danh sách. Chẳng hạn, bộ `('Cleese', 'John')` sẽ được biểu diễn như sau:



Nhưng trong một sơ đồ lớn hơn bạn có thể muốn bỏ qua các chi tiết. Chẳng hạn, sơ đồ của một danh bạ điện thoại có thể như sau:

Ở đây các bộ được ghi dưới dạng cú pháp Python, cũng để ghi tắt trên sơ đồ.

Số điện thoại trong sơ đồ là đường dây góp ý cho đài phát thanh BBC, vậy nên bạn đừng gọi số đó.

So sách các bộ

Các toán tử quan hệ cũng có tác dụng với bộ và các dãy khác; Python bắt đầu với việc so sánh phần tử thứ nhất của từng dãy. Nếu chúng bằng nhau thì việc so sánh tiếp tục với các phần tiếp kế sau, và cứ như vậy, cho đến khi tìm được phần tử khác nhau. Các phần tử còn lại thì không được xét đến nữa (dù chúng có lớn đến đâu).

```
>>> (0, 1, 2) < (0, 3, 4)
True
>>> (0, 1, 2000000) < (0, 3, 4)
True
```

Hàm `sort` cũng hoạt động theo cách tương tự. Nó sắp xếp theo phần tử thứ nhất, nhưng nếu chúng bằng nhau thì xếp theo phần tử thứ hai, và cứ như vậy.

Đặc điểm này dựa trên một dạng mẫu có tên **DSU**, vốn là viết tắt của

Decorate
: (“trang trí”) một dãy bằng cách tạo dựng một danh sách các bộ với một hoặc nhiều khóa sắp xếp đi trước những phần tử của dãy,

Sort
: (sắp xếp) danh sách các bộ, và

Undecorate
: (“gỡ bỏ trang trí”) bằng cách kết xuất các phần tử đã được sắp xếp của dãy.

Chẳng hạn, nếu bạn có một danh sách những từ cần sắp xếp từ dài đến ngắn:

```
def sort_by_length(words):
    t = []
    for word in words:
       t.append((len(word), word))

    t.sort(reverse=True)

    res = []
    for length, word in t:
```

```
            res.append(word)
    return res
```

Vòng lặp thứ nhất tạo dựng một danh sách các bộ, trong đó mỗi bộ là một từ và trước đó là chiều dài của từ đó.

`sort` thực hiện so sánh phần tử thứ nhất (tức độ dài) trước, và chỉ khi độ dài như nhau thì mới xét đến phần tử thứ hai. Từ khóa đối số `reverse=True` báo cho `sort` thực hiện theo chiều giảm dần.

Vòn lặp thứ hai duyệt danh sách các bộ và tạo dựng một danh sách các từ theo thứ tự giảm dần về độ dài.

> Ở ví dụ này, để phân định thứ tự giữa các từ cùng độ dài sẽ dựa vào thứ tự của từ đó xếp trong bảng chữ cái. Trong trường hợp khác bạn có thể phân định theo tiêu chí ngẫu nhiên. Hãy sửa lại ví dụ này sao cho những từ cùng độ dài được xếp theo thứ tự ngẫu nhiên. Gợi ý: dùng hàm `random` trong module `random`.

Dãy chứa các dãy

Đến giờ tôi mới tập trung vào danh sách các bộ, nhưng hầu hết các ví dụ trong chương này cũng thực hiện được với danh sách chứa các danh sách, bộ chứa các bộ, và bộ chứa các danh sách. Để tránh phải liệt kê tất cả những sự kết hợp có thể, ta sẽ nói gọn là dãy chứa các dãy.

Trong nhiều trường hợp, những dạng khác nhau của dãy (chuỗi, danh sách và bộ) có thể được dùng hoán đổi cho nhau được. Vì vậy tại sao và bằng cách nào mà bạn lựa chọn dạng này thay vì dạng kia?

Bắt đầu với điều hiển nhiên, chuỗi thì có tính năng hạn chế rất nhiều so với các loại dãy khác vì những phần tử của nó chỉ có thể là kí tự. Chuỗi cũng là kiểu không thay đổi được. Nếu bạn muốn có tính năng thay đổi những kí tự trong chuỗi mà không muốn tạo ra một chuỗi mới, có lẽ bạn sẽ phải dùng một danh sách các kí tự để thay thế.

Danh sách thì thông dụng hơn bộ, chủ yếu là vì chúng thay đổi được. Nhưng cũng có một vài trường hợp bạn muốn dùng bộ hơn:

1. Trong một số trường hợp, như trong câu lệnh `return`, việc tạo ra một bộ thì đơn giản hơn một danh sách, về hình thức cú pháp. Trong những trường hợp khác, có thể bạn nên dùng danh sách.
2. Nếu muốn tạo một dãy để làm khóa cho từ điển, bạn phải dùng một kiểu dữ liệu không thay đổi như bộ hoặc chuỗi.
3. Nếu bạn chuyển một dãy với vài trò đối số cho một hàm, việc dùng bộ sẽ làm giảm khả năng gây lỗi bắt nguồn từ tham chiếu bội.

Vì bộ là kiểu không thay đổi được, chúng không có những phương thức như `sort` và `reverse`, vốn để sửa đổi một danh sách sẵn có. Nhưng Python lại cung cấp các hàm có sẵn `sorted` và `reversed` để nhận vào một chuỗi bất kì làm tham biến và trả lại một danh sách mới với cùng các phần tử theo một thứ tự khác.

Gỡ lỗi

Danh sách, từ điển và bộ thường được gọi chung là **cấu trúc dữ liệu**; trong chương này ta bắt đầu thấy những cấu trúc dữ liệu phứ hợp, như danh sách các bộ, và từ điển có khóa là các bộ và trị là các danh sách. Cấu trúc dữ liệu phức hợp rất có ích, nhưng chúng dễ mắc phải vấn đề mà tôi gọi là **lỗi hình dạng**; nghĩa là những lỗi phát sinh khi một cấu trúc dữ liệu có kiểu, kích thước hoặc cấu

trúc sai. Chẳng hạn, nếu bạn trông đợi một danh sách có chứa một số nguyên mà tôi lại cho bạn một số nguyên (không phải danh sách) thì chương trình không hoạt động.

Để giúp cho việc gỡ lỗi kiểu như vậy, tôi đã viết một module có tên `structshape` trong đó có một hàm cùng tên `structshape`, nhận vào bất kì một cấu trúc dữ liệu nào làm đối số và trả về một chuỗi làm nhiệm vụ tóm lược lại hình dạng của nó. Bạn có thể tải nó về từ thinkpython.com/code/structshape.py

Sau đây là kết quả cho một dãy đơn giản:

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> print structshape(t)
list of 3 int
```

Một chương trình đẹp hơn có thể in ra "list of 3 ints", nhưng không tính đến danh từ số nhiều thì sẽ dễ hơn. Sau đây là một danh sách các danh sách:

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> print structshape(t2)
list of 3 list of 2 int
```

Nếu các phần tử của danh sách không có cùng kiểu thì `structshape` sẽ nhóm chúng lại theo dạng, đúng theo thứ tự:

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> print structshape(t3)
list of (3 int, float, 2 str, 2 list of int, int)
```

Sau đây là một danh sách các bộ:

```
>>> s = 'abc'
>>> lt = zip(t, s)
>>> print structshape(lt)
list of 3 tuple of (int, str)
```

Và sau đây là một từ điển có 3 mục trong đó ánh xạ số nguyên đến chuỗi.

```
>>> d = dict(lt)
>>> print structshape(d)
dict of 3 int->str
```

Nêu bạn vẫn vướng mắc trong việc theo dõi các cấu trúc dữ liệu thì `structshape` có thể sẽ hữu ích.

Thuật ngữ

bộ:
: Dãy các phần tử không thể thay đổi được.

phép gán bộ:
: Phép gán với một dãy ở vế phải và một bộ các biến ở vế trái. Vế phải được định lượng và sau đó các phần tử của nó được gán cho các biến ở vế trái.

thu gom:
: Thao tác tập hợp một bộ các đối số có độ dài thay đổi.

phát tán:
: Thao tác xử lý dãy như một danh sách các đối số.

DSU:
: Viết tắt của “decorate-sort-undecorate”, một dạng mẫu bao gồm tạo dựng một danh sách các bộ, sắp xếp, và kết xuất một phần của kết quả.

cấu trúc dữ liệu:
: Tập hợp các giá trị có liên hệ với nhau, thường được tổ chức dưới dạng danh sách, từ điển, bộ, v.v.

hình dạng (của cấu trúc dữ liệu):
: Dạng tóm lược của kiểu, hình dạng và thành phần của một cấu trúc dữ liệu.

Bài tập

Hãy viết một hàm có tên `most_frequent` nhận vào một chuỗi và in ra các chữ cái theo thứ tự tần số xuất hiện giảm dần. Hãy tìm những đoạn văn viết bằng vài thứ tiếng khác nhau và xem tần số chữ cái của chúng khác nhau thế nào. So sách kết quả bạn tìm được với bảng thống kê tại wikipedia.org/wiki/Letter_frequencies.

Thêm bài về đảo chữ!

1. Hãy viết một chương trình đọc vào danh sách các từ trong một file (xem Mục {wordlist}) và in ra tất cả tập hợp các từ gồm những chữ cái đảo nhau.

   Sau đây là một ví dụ kết quả có thể tìm được:

   ```
   ['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
   ['retainers', 'ternaries']
   ['generating', 'greatening']
   ['resmelts', 'smelters', 'termless']
   ```

   Gợi ý: có thể bạn muốn lập một từ điển trong đó ánh xạ từ một tập hợp những chữ cái đến một danh sách những từ viết được từ những chữ cái đó. Câu hỏi là, bạn biểu diễn tập hợp chữ cái đó như thế nào để chúng có thể làm khóa của từ điển?

2. Sửa chương trình trên để in ra tập hợp các từ đảo lớn nhất trước, sau đến tập hợp lớn thứ hai, và cứ như vậy.
3. Trong trò chơi Scrabble, “bingo” là trường hợp khi bạn dùng hết toàn bộ bảy mảnh chữ trên khay của mình, cùng với một mảnh chữ sẵn có trên bảng, để tạo thành một từ gồm tám chữ cái. Tập hợp 8 chữ cái nào có nhiều khả năng hình thành bingo nhất? Gợi ý: tập hợp này tạo thành được 7 từ.
4. Hai từ tạo thành một “cặp hoán đổi” nếu bạn có thể chuyển từ này thành từ kia chỉ bằng cách đổi chỗ hai chữ cái[3]; chẳng hạn, “converse” và “conserve”. Hãy viết một chương trình để tìm tất cả các cặp hoán đổi trong từ điển. Gợi ý: đừng kiểm tra tất cả những cặp hai từ, cũng đừng kiểm tra tất cả các khả năng hoán đổi.

Bạn có thể tải về một lời giải ở
thinkpython.com/code/anagram_sets.py.

Sau đây là một câu đố Car Talk khác[4]:

> Từ tiếng Anh nào dài nhất có đặc điểm sau, mỗi khi ta lần lượt bỏ bớt một chữ cái thì nó vẫn còn có nghĩa?
>
> Cụ thể, chữ cái bỏ đi có thể ở đầu, cuối, hoặc giữa từ, nhưng sau khi bỏ không được sắp xếp lại vị trí các chữ còn lại. Mỗi khi bỏ đi một chữ, bạn giữ lại một từ tiếng Anh khác. Cứ tiếp tục như vậy, cuối cùng bạn sẽ còn một chữ cái và chữ cái này cũng phải là một từ tiếng Anh có nghĩa— tức là có trong từ điển. Tôi muốn biết rằng từ dài nhất như vậy là gì và nó có mấy chữ cái?
>
> Tôi sẽ cho bạn một từ ngắn làm ví dụ: Sprite. Được chứ? Bắt đầu với *sprite*, bạn bỏ bớt một chữ cái trong từ đó, bỏ chữ r đi, và chúng ta còn lại từ *spite*, sau đó ta bỏ chữ cái e ở cuối, ta còn lại từ *spit*, ta bỏ đi chữ s, ta còn lại từ *pit*, *it*, và *I*.

Hãy viết một chương trình để tìm tất cả những từ có thể lược đi kiểu này, và sau đó tìm ra từ dài nhất trong số đó.

Bài tập này khó hơn một chút so với các bài khác trong cuốn sách này, và tôi có một số gợi ý sau:

1. Bạn có thể sẽ muốn viết một hàm nhận vào một từ và tìm ra danh sách của tất cả các từ có thể hình thành được sau khi bỏ một chữ cái. Đó là những "từ con" của từ ban đầu.
2. Theo cách đệ quy, một từ được gọi là lược bớt được nếu như mọi từ con của nó đều lược bớt được. Ở trường hợp cơ bản, có thể coi rằng một chuỗi rỗng là lược bớt được.
3. Danh sách từ mà tôi cung cấp, `words.txt`, không chứa các từ có một chữ cái. Vì vậy bạn có thể muốn thêm vào "I", "a", và chuỗi rỗng.
4. Để cải thiện tốc độ của chương trình, bạn có thể sẽ cần phải ghi nhớ những từ đã xác định là lược bỏ được.

Bạn có thể xem lời giải của tôi ở thinkpython.com/code/reducible.py.

---

1. Trong Python 3.0, `zip` trả lại một nguồn lặp các bộ, nhưng thường thì người ta vẫn coi một nguồn lặp tựa như một danh sách. ↩
2. Điều này sẽ hơi khác trong Python 3.0. ↩
3. Bài tập này được hình thành từ một ví dụ tại puzzlers.org. ↩
4. www.cartalk.com/content/puzzler/transcripts/200651. ↩

Chương 13: Nghiên cứu cụ thể: lựa chọn cấu trúc dữ liệu

Trở về Mục lục cuốn sách

Phân tích tần số của từ vựng

Như thường lệ, ít nhất là bạn phải cố gắng làm các bài tập sau đây trước khi đọc lời giải của tôi.

Hãy viết một chương trình đọc vào một file, tách mỗi dòng thành các từ riêng lẻ, cắt bỏ các dấu trắng và dấu câu, rồi chuyển chúng về dạng chữ thường.

Gợi ý: module `string` cho ta hai chuỗi, `whitespace` chứa dấu cách, dấu tab, dấu xuống dòng, v.v., và `punctuation` chứa các kí tự dấu câu. Hãy thử làm Python chửi thề xem:

```
>>> import string
>>> print string.punctuation
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```

Có thể bạn cũng muốn dùng các phương thức chuỗi `strip`, `replace` và `translate`.

Hãy thăm dự án Gutenberg (`gutenberg.net`) và tải về cuốn sách ưa thích của bạn dưới dạng file chữ.

Sửa chương trình trong bài tập trước để đọc được quyển sách vừa tải về, bỏ qua đoạn thông tin ở đầu file, và xử lý phần còn lại như bài trước.

Sau đó sửa chương trình để đếm số từ trong quyển sách, và số lần mỗi từ được dùng đến.

In ra số từ khác nhau được dùng trong cuốn sách. So sánh những cuốn sách viết bởi các tác giả khác nhau, viết trong những giai đoạn khác nhau. Tác giả nào dùng lượng từ vựng nhiều nhất?

Hãy sửa chương trình ở bài tập trước để in ra 20 từ được dùng thường xuyên nhất trong cuốn sách.

Hãy sửa chương trình trước để đọc vào một dánh sách các từ (xem Mục {wordlist}) rồi in ra tất cả các từ trong cuốn sách mà không nằm trong danh sách từ. Trong số đó có bao nhiêu lỗi in? Có bao nhiêu từ thông dụng mà *đáng ra* phải có trong danh sách, và bao nhiêu từ thực sự ít gặp?

Số ngẫu nhiên

Với cùng dữ liệu đầu vào, phần lớn các chương trình máy tính đều cho ra kết quả giống hệt sau mỗi lần chạy, vì vậy ta nói chúng có **tính tất định**. Tất định là một đặc tính tốt, vì ta thường trông đợi

những kết quả tính toán phải giống nhau. Tuy nhiên, ở một số ứng dụng, ta muốn việc tính toán phải không định trước được. Các trò chơi trên máy là ví dụ dễ thấy, ngoài ra còn những chương trình khác.

Để làm chương trình hoàn toàn không định trước hóa ra lại là một việc không dễ dàng gì, nhưng có những cách làm cho chúng ít nhất là dường như không định trước. Một trong số đó là dùng những thuật toán phát sinh ra số **giả ngẫu nhiên**. Số giả ngẫu nhiên không phải ngẫu nhiên thực sự vì được phát sinh từ một phép toán tất định, nhưng chỉ nhìn vào những con số thì chúng ta không thể phân biệt chúng với những số ngẫu nhiên thực sự.

Module `random` có các hàm phát sinh ra những số giả ngẫu nhiên (để cho tiện, từ đây tôi sẽ gọi là "ngẫu nhiên").

Hàm `random` trả lại một số có phần thập phân, nằm giữa 0.0 và 1.0 (tính cả 0.0 nhưng không kể 1.0). Mỗi khi gọi `random`, bạn thu được số tiếp theo trong một dãy số dài. Để xem một ví dụ, hãy chạy vòng lặp sau:

```
import random

for i in range(10):
    x = random.random()
    print x
```

Hàm `randint` nhận các tham biến `low` và `high` rồi trả lại một số nguyên nằm trong khoảng từ `low` đến `high` (kể cả hai đầu).

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

Để chọn ngẫu nhiên một phần tử từ một dãy, bạn có thể dùng `choice`:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

Module `random` cũng có các hàm để phát sinh những giá trị ngẫu nhiên từ những dạng phân bố liên tục bao gồm phân bố chuẩn, lũy thừa, gamma và một số dạng khác.

> Hãy viết một hàm có tên `choose_from_hist` nhận vào một histogram như đã đề cập trong Mục {tần số} và trả lại một giá trị ngẫu nhiên từ histogram, được chọn với xác suất tỉ lệ với tần suất xuất hiện. Chẳng hạn, với histogram sau:
>
> ```
> >>> t = ['a', 'a', 'b']
> >>> h = histogram(t)
> >>> print h
> {'a': 2, 'b': 1}
> ```
>
> hàm bạn viết cần trả lại `'a'` với xác suất 2 / 3 và `'b'` với xác suất 1 / 3.

Bảng tần số của từ vựng

Sau đây là một chương trình để đọc vào một file và lập một bảng tần số các từ xuất hiện trong file

đó:

```
import string

def process_file(filename):
    h = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, h)
    return h

def process_line(line, h):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()

        h[word] = h.get(word, 0) + 1

hist = process_file('emma.txt')
```

Chương trình này đọc file `emma.txt`, vốn có nội dung là truyện *Emma* của Jane Austen.

`process_file` lặp qua các dòng trong file, lần lượt chuyển từng dòng vào `process_line`. Bảng tần số `h` được dùng như một biến tích lũy.

`process_line` dùng phương thức chuỗi `replace` để thay thế dấu gạch nối từ với dấu cách trước khi dùng `split` để tách một dòng thành một danh sách các chuỗi. Nó duyệt một dãy các từ và dùng `strip` và `lower` để bỏ các dấu câu và chuyển về dạng chữ thường. (Ở đây ta chỉ nói tắt là "chuyển đổi" chuỗi; hãy nhớ rằng kiểu chuỗi không thể thay đổi được, vì vậy các phương thức như `strip` và `lower` đều trả về chuỗi mới.)

Cuối cùng, `process_line` cập nhật bảng tần số bằng cách tạo ra một mục mới hoặc tăng thêm 1 cho mục sẵn có.

Để đếm tổng số từ trong file, ta cần cộng các tần số trong bảng lại:

```
def total_words(h):
    return sum(h.values())
```

Số các từ khác nhau chính bằng số các mục trong từ điển:

```
def different_words(h):
    return len(h)
```

Sau đây là đoạn mã để in kết quả:

```
print 'Total number of words:', total_words(hist)
print 'Number of different words:', different_words(hist)
```

Và kết quả:

```
Total number of words: 161073
Number of different words: 7212
```

Để tìm những từ thông dụng nhất, ta có thể áp dụng dạng mẫu DSU; `most_common` nhận một bảng tần số và trả lại một danh sách các bộ từ-tần số, được sắp xếp theo thứ tự tần số giảm dần:

```
def most_common(h):
```

```
    t = []
    for key, value in h.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

Sau đây là một vòng lặp in ra mười từ thông dụng nhất:

```
t = most_common(hist)
print 'The most common words are:'
for freq, word in t[0:10]:
    print word, '\t', freq
```

Và đây là kết quả thu được từ truyện *Emma*:

```
The most common words are:
to      5242
the     5204
and     4897
of      4293
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

Ta đã thấy các hàm và phương thức có sẵn, chúng nhận vào một số lượng các đối số thay đổi. Ta còn có thể viết những hàm có đối số tùy chọn. Chẳng hạn, sau đây là một hàm để in ra những từ thông dụng nhất dựa theo bảng tần số

```
def print_most_common(hist, num=10)
    t = most_common(hist)
    print 'The most common words are:'
    for freq, word in t[0:num]:
        print word, '\t', freq
```

Tham biến thứ nhất là bắt buộc; tham biến thứ hai có thể tùy chọn. **Giá trị mặc định** của `num` là 10.

Nếu bạn chỉ cung cấp một đối số:

```
print_most_common(hist)
```

thì `num` sẽ nhận giá trị mặc định. Nếu bạn cung cấp hai đối số:

```
print_most_common(hist, 20)
```

thì thay vào đó `num` sẽ lấy giá trị từ đối số. Nói cách khác, đối số tùy chọn sẽ **chiếm chỗ** giá trị mặc định.

Nếu như một hàm có cả tham số bắt buộc và tham số tùy chọn thì tất cả tham số bắt buộc phải được viết trước, rồi mới đến tham số tùy chọn.

Phép trừ các từ điển

Việc tìm những từ trong cuốn sách mà không có trong danh sách từ `words.txt` là một bài toán có thể bạn nhận ra đó chính là phép trừ tập hợp; nghĩa là ta muốn tìm tất cả các từ trong một tập hợp

(các từ trong cuốn sách) mà không có trong tập hợp kia (các từ trong từ điển).

`subtract` nhận vào hai từ điển `d1` và `d2` rồi trả về một tử điển mới bao gồm tất cả các khóa của `d1` mà không có trong `d2`. Vì chúng ta không thực sụ quan tâm đến các giá trị, ta gán chúng đều bằng None.

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

Để tìm những từ trong cuốn sách mà không có trong `words.txt`, ta có thể dùng `process_file` để lập một bảng tần số cho `words.txt`, rồi sau đó thực hiện phép trừ:

```
words = process_file('words.txt')
diff = subtract(hist, words)

print "The words in the book that aren't in the word list are:"
for word in diff.keys():
    print word,
```

Sau đây là một đoạn kết quả tìm được từ *Emma*:

```
The words in the book that aren't in the word list are:
 rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

Một số từ trong đó là các tên riêng và sở hữu cách. Các từ khác, như "rencontre", hiện tại ít được dùng. Nhưng vẫn có một số ít từ thông dụng mà đáng ra phải có trong danh sách!

> Python có một cấu trúc dữ liệu tên là `set` trong đó có chứa nhiều phép toán thông dụng với tập hợp. Hãy đọc tài liệu tại docs.python.org/lib/types-set.html và viết một chương trình trong đó dùng phép trừ tập hợp để tìm những từ trong cuốn sách mà không có trong danh sách từ.

Những từ ngẫu nhiên

Để chọn một từ ngẫu nhiên từ bảng tần số, thuật toán đơn giản nhất là lập một danh sách với mỗi từ lặp lại nhiều lần, dựa theo tần số quan sát được, và sau đó thì lựa chọn từ danh sách:

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

Biểu thức `[word] * freq` tạo ra một danh sách chứa `freq` lần lặp lại chuỗi `word`. Phương thức `extend` cũng tương tự như `append` ngoại trừ rằng đối số là một dãy.

> Thuật toán này hoạt động được, nhưng không hiệu quả lắm; mỗi khi bạn chọn một từ ngẫu nhiên, nó lại lập lại danh sách, mà danh sách này lớn bằng cuốn sách ban đầu. Một cách cải thiện dễ thấy là lập danh sách một lafn và sau đó chọn nhiều lần, nhưng danh sách vẫn còn lớn.

Một cách làm khác là:

1. Dùng `keys` để lấy một danh sách các từ trong cuốn sách.
2. Lập một danh sách có chứa tổng lũy tích của các tần số từ (xem Bài tập {lũy tích}). Mục từ cuối trong danh sách này là số từ trong cuốn sách này, *n*.
3. Chọn một số ngẫu nhiên từ 1 đến *n*. Dùng cách tìm kiếm nhị phân (Xem Bài tập {phân đôi}) để tìm chỉ số mà tại đó số ngẫu nhiên cần được điền vào cho tổng lũy tích.
4. Dùng chỉ số để tìm ra từ tương ứng trong danh sách các từ

Hãy viết một chương trình dùng thuật toán này để chọn một từ ngẫu nhiên từ cuốn sách.

Phép phân tích Markov

Nếu bạn chọn các từ trong cuốn sách một cách ngẫu nhiên, bạn chỉ có thể có những từ đúng, nhưng không thể có một câu đúng:

```
this the small regard harriet which knightley's it most things
```

Một loạt các từ ngẫu nhiên hiếm khi có nghĩa vì không có mối liên hệ nào giữa những từ kế tiếp. Chẳng hạn, trong một câu thực sự bạn sẽ mong đợi một mạo từ như "the" đi trước một tính từ hoặc danh từ, và có lẽ không đi trước một động từ hoặc trạng từ.

Một cách lượng hóa những mối liên hệ như vậy là phép phân tích Markov[1], trong đó cho trước một dãy các từ, ta phải tìm xác suất của từ đi kế tiếp. Chẳng hạn, bài hát *Eric, the Half a Bee* bắt đầu với:

> 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?

Trong bài này, cụm từ "half the" luôn được tiếp nối bởi từ "bee", nhưng cụm từ "the bee" có thể được nối tiếp bởi "has" hoặc "is".

Kết quả của việc phân tích Markov là một quy tắc ánh xạ từ mỗi tiền tố (như "half the" và "the bee") đến tất cả những hậu tố có thể có (như "has" và "is").

Cho trước quy tắc ánh xạ này, bạn có thể phát sinh một văn bản ngẫu nhiên bằng cách bắt đầu với một tiền tố bất kì và chọn một hậu tố khả dĩ một cách ngẫu nhiên. Sau đó, bạn có thể kết hợp đoạn cuối của tiền tố và hậu tố mới tìm được để hình thành một tiền tố mới cho bước tính toán tiếp theo.

Chẳng hạn, nếu bạn bắt đầu với tiền tố "Half a", thì từ tiếp theo phải là "bee", bởi vì tiền tố này chỉ xuất hiện một lần duy nhất trong văn bản. Tiền tố tiếp theo là "a bee", vì vậy hậu tố tiếp theo có thể là "philosophically", "be" hoặc "due".

Trong ví dụ này độ dài của tiền tố luôn là hai, nhưng bạn có thể phân tích Markov với tiền tố dài bất kì. Độ dài của tiền tố được gọi là "bậc" của phân tích.

Phép phân tích Markov:

1. Hãy viết một chương trình đọc một văn bản từ một file và thực hiện phân tích Markov. Kết quả cần thu được là một từ điển với ánh xạ từ các tiền tố đến một tập hợp các hậu tố có thể có. Tập hợp này có thể là danh sách, bộ, hoặc từ điển, tùy bạn chọn. Bạn có thể chạy thử chương trình với tiền tố có độ dài bằng 2, nhưng bạn nên viết chương trình sao cho dễ thử được với những độ dài khác.
2. Hãy thêm một hàm vào chương trình trên để phát sinh một văn bản ngẫu nhiên dựa trên phép phân tích Markov. Sau đây là một ví dụ từ truyện *Emma* với độ dài tiền tố bằng 2:

   > 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.

   Ở ví dụ này, tôi để nguyên dấu câu đi liền với các từ. Kết quả tương đối đúng về mặt ngữ pháp, nhưng không hoàn toàn. Về ngữ nghĩa nó dường như có nghĩa, nhưng cũng không hoàn toàn.

   Điều gì sẽ xảy ra nếu tôi tăng chiều dài tiền tố? Liệu văn bản ngẫu nhiên lần này có nghĩa hơn không?

3. Một khi chương trình của bạn chạy được, có thể bạn muốn thử trộn lẫn: nếu bạn phân tích văn bản từ nhiều cuốn sách khác nhau, văn bản ngẫu nhiên được phát sinh sẽ hòa trộn từ và cụm từ những cuốn sách ban đầu theo cách khá thú vị.

Cấu trúc dữ liệu

Việc dùng phép phân tích Markov để phát sinh văn bản ngẫu nhiên thật thú vị, nhưng cũng có một mục đích để luyện tập lập trình: kĩ năng chọn cấu trúc dữ liệu. Trong lời giải của bạn cho các bài tập trước, bạn đã phải chọn:

- Cách biểu diễn các tiền tố.
- Cách biểu diễn tập hợp các hậu tố có thể có.
- Cách biểu diễn quy tắc ánh xạ giữa mỗi tiền tố với tập hợp các hậu tố có thể.

Được rồi, điều cuối cùng rất dễ dàng; dạng ánh xạ duy nhất mà chúng ta biết chính là từ điển, và đây là cách lựa chọn rất tự nhiên.

Với các tiền tố, lựa chọn dễ thấy nhất là dùng chuỗi, danh sách các chuỗi, hoặc bộ các chuỗi. Với các hậu tố, một lựa chọn là danh sách; lựa chọn khác là bảng tần số (từ điển).

Bạn lựa chọn bằng cách nào đây? Bước đầu tiên là nghĩ về những phép thao tác cần thực hiện trên từng cấu trúc dữ liệu. Với các tiền tố, ta cần có khả năng bỏ được những từ ở phía đầu và thêm vào những từ ở phía cuối. Chẳng hạn, nếu tiền tố hiện tại là "Half a", và từ tiếp theo là "bee", bạn cần có khả năng tạo thành tiền tố thiếp theo là "a bee".

Lựa chọn ban đầu của bạn có thể là danh sách, vì ta dễ dàng thêm và bỏ các phần tử của nó, nhưng ta cũng cần dùng được các tiền tố như các khóa trong từ điển; vì vậy mà ta phải loại bỏ khả năng dùng danh sách. Khi dùng bộ, bạn không thể thêm vào hoặc xóa đi, nhưng có thể dùng toán tử cộng để hình thành một bộ mới:

```
def shift(prefix, word):
    return prefix[1:] + (word,)
```

`shift` nhận vào một bộ các từ, `prefix`, và một chuỗi, `word`, rồi hình thành một bộ mới chứa tất cả những từ trong `prefix` ngoại trừ từ thứ nhất, và thêm `word` vào cuối.

Với tập hợp các hậu tố, những phép thao tác cần thực hiện gồm có thêm vào một hậu tố mới (hoặc tăng tần số của một hậu tố sẵn có), và chọn một hậu tố ngẫu nhiên.

Việc thêm một hậu tố mới cũng dễ thực hiện trên danh sách như trên bảng tần số. Việc chọn một phần tử ngẫu nhiên từ danh sách thì dễ; chọn từ bảng tần số một cách hiệu quả thì khó thực hiện hơn (xem Bài tập {tần số ngẫu nhiên}).

Đến đây chúng ta mới chỉ chủ yếu bàn về độ khó dễ khi thực hiện, nhưng còn những yếu tố khác cần tính đến khi chọn các cấu trúc dữ liệu. Một trong số đó là thời gian chạy. Đôi khi có một nguyên nhân về mặt lí thuyết khiến ta mong đợi rằng một cấu trúc dữ liệu này nhanh hơn cấu trúc khác; chẳng hạn, tôi đã đề cập rằng toán tử `in` dùng với từ điển thì nhanh hơn dùng với danh sách, ít nhất là khi có rất nhiều phần tử.

Nhưng thường thì bạn sẽ không biết trước được là cách thực hiện nào sẽ nhanh hơn. Một lựa chọn là thực hiện cả hai và xem cách nào tốt hơn. Cách tiếp cận này được gọi là **đánh giá bằng so sánh**. Một cách làm thực tiễn khác là chọn cấu trúc dữ liệu dễ thực hiện nhất, vá sau đó xem nó có đủ nhanh khi dùng trong ứng dụng được viết hay không. Nếu đủ nhanh thì không cần phải sửa nữa. Nếu không thì có những công cụ, như module `profile`, có thể chỉ ra những chỗ trong chương trình mà chạy mất nhiều thời gian nhất.

Yếu tố khác cần tính đến là dung lượng lưu trữ. Chẳng hạn, việc dùng một bảng tần số để tập hợp các hậu tố có thể sẽ chiếm ít dung lượng hơn vì bạn chỉ cần lưu mỗi từ một lần, bất kể từ đó được dùng bao nhiêu lần trong văn bản. Trong một số trường hợp khác, việc tiết kiệm dung lượng cũng có thể làm chương trình của bạn chạy nhanh hơn, và ngược lại, có thể làm chương trình của bạn không chạy được do hết bộ nhớ. Tuy nhiên, với nhiều ứng dụng, dung lượng là vấn đề thứ yếu sau bộ nhớ.

Một suy nghĩ sau cùng: với nội dung thảo luận ở trên, tôi có ngụ ý là chúng ta nên dùng một cấu trúc dữ liệu chung cho cả việc phân tích và phát sinh. Nhưng vì đây là hai khâu riêng biệt, ta cũng có thể dùng một cấu trúc dữ liệu để phân tích sau đó chuyển đổi sang một cấu trúc khác cho việc phát sinh. Cách này tính ra sẽ có lợi nếu như khoảng thời gian tiết kiệm được khi phát sinh lớn hơn thời gian để chuyển đổi.

Gỡ lỗi

Khi gỡ lỗi một chương trình, đặc biệt là khi bạn cố gắng gỡ một lỗi hóc búa, có bốn việc mà bạn nên thử như sau:

đọc:
: Xem xét mã lệnh, tự đọc và diễn giải nó, và kiểm tra xem nó có thực hiện đúng ý định của bạn không.

chạy:
: Thử nghiệm bằng cách thay đổi và chạy các bản mã lệnh khác nhau. Thường khi bạn hiển thị đúng thứ cần thiết ở một vị trí đúng trong chương trình thì bài toán đã rõ ràng, nhưng đôi khi bạn phải dành thời gian để dựng dàn giáo.

tư duy:

Hãy dành thời gian để suy nghĩ! Lỗi xảy ra thuộc loại gì: cú pháp, thực thi, hay ngữ nghĩa? Bạn rút được thông tin gì từ thông báo lỗi, từ kết quả đầu ra của chương trình? Loại lỗi gì có thể gây ra vấn đề mà bạn thấy được? Trước khi có vấn đề xảy ra, bạn đã sửa đổi mã lệnh ở chỗ nào?

rút lui:
: Sẽ có lúc nhất định, cách làm tốt nhất là rút lui, thu lại những thay đổi gần nhất, đến khi bạn trở về trạng thái của chương trình chạy được mà bạn cũng hiểu được. Sau đó bạn có thể bắt đầu phát triển lại chương trình.

Các lập trình viên mới đôi khi cũng bị lún sâu vào một trong những cách làm sau mà quên mất những cách còn lại. Mỗi cách gắn liền với một kiểu nhược điểm riêng.

Chẳng hạn, việc đọc mã lệnh có thể giúp bạn nếu như vấn đề xảy ra thuộc về lỗi chính tả, nhưng sẽ không ích gì nếu như vấn đề ở chỗ hiểu sai khái niệm. Nếu bạn không hiểu chương trình làm việc gì, có thể bạn sẽ phải đọc nó 100 lần mà vẫn không thấy lỗi, vì lỗi nằm ngay trong đầu bạn.

Việc chạy các thử nghiệm có thể sẽ giúp bạn, đặc biệt khi bạn chạy những bài kiểm tra ngắn, đơn giản. Nhưng nếu bạn chạy thử nghiệm mà không nghĩ hoặc đọc mã lệnh, có thể bạn sẽ rơi vào trường hợp mà tôi gọi là "lập trình theo bước ngẫu nhiên", tức là quá trình tạo ra những thay đổi ngẫu nhiên đến khi chương trình chạy đúng. Khỏi cần phải nói, việc lập trình theo ngẫu nhiên có thể sẽ rất tốn thời gian.

Bạn cần phải dành thời gian suy nghĩ. Gỡ lỗi cũng giống như một ngành khoa học thực nghiệm. Bạn cần phải có ít nhất một giả thiết về vấn đề cần giải quyết là gì. Nếu có hai hay nhiều khả năng xảy ra, cố gắng nghỉ ra một phép thử để loại bỏ một khả năng trong số đó.

Việc nghỉ ngơi cũng giúp ích cho suy nghĩ. Việc nói năng cũng vậy. Nếu bạn giải thích vấn đề cho một người khác (hoặc thậm chí cho chính bạn), có thể đôi khi bạn sẽ tìm được lời giải ngay trước lúc kết thúc lời nói.

Nhưng ngay cả những kĩ thuật gỡ lỗi tốt nhất cũng thất bại khi có quá nhiều lỗi, hoặc khi mã lệnh bạn cần sửa quá lớn và phức tạp. Đôi lúc giải pháp tốt nhất là rút lui, làm đơn giản chương trình cho đến khi bạn có được phiên bản chương trình hoạt động được và bạn hiểu được nó.

Các lập trình viên mới thường miễn cưỡng trong việc rút lui vì họ không thể chịu được khi xóa một dòng lệnh (ngay cả khi nó sai). Nếu bạn cảm thấy tiện thì có thể sao chép chương trình ra một file khác trước khi bắt tay vào việc lược bỏ nó. Sau đó bạn có thể dán các phần cũ trở lại, mỗi lúc một ít.

Để tìm ra một lỗi hóc búa đòi hỏi phải đọc, chạy mã lệnh, suy nghĩ, và đôi khi rút lui. Nếu bạn bị sa lầy vào một trong những việc trên thì hãy thử các việc còn lại.

Thuật ngữ

tất định:
: Tính chất của chương trình thực hiện công việc giống nhau ở mỗi lần chạy và cho ra kết quả như nhau.

giả ngẫu nhiên:
: Tính chất của dãy số nhìn có vẻ ngẫu nhiên, nhưng được phát sinh bởi một chương trình tất định.

giá trị mặc định:

Giá trị được gán cho tham biến tùy chọn nếu không nhập vào đối số.

chiếm chỗ:
Sự thay thế giá trị mặc định bởi một đối số.

đánh giá bằng so sánh:
Quá trình chọn lựa giữa các cấu trúc dữ liệu bằng cách thực hiện các phương án rồi kiểm thử chúng theo một mẫu số liệu có thể.

Bài tập

"Hạng" của một từ là vị trí của nó trong một danh sách các từ xếp theo tần số: từ thường gặp nhất có hạng 1, từ thường gặp thứ nhì có hạng 2, v.v.

Định luật Zipf mô tả mối liện hệ giữa hạng và tần số của từ trong ngôn ngữ tự nhiên[2]. Đặc biệt hơn, nó còn dự đoán rằng tần số f của từ có hạng r là:

$$f = cr^{-s}$$

trong đó s và c là các tham số phụ thuộc vào ngôn ngữ và văn bản. Nếu lấy logarit hai vế của phương trình trên, bạn sẽ thu được:

$$\log f = \log c - s \log r$$

Vì vậy khi vẽ đồ thị log$f$ theo log$r$, bạn sẽ được một đường thẳng với độ dốc $-s$ và giao điểm log$c$ với trục tung.

Hãy viết một chương trình để đọc văn bản vào từ một file, đếm tần số các từ, và in ra mỗi từ trên một dòng, theo thứ tự giảm dần của tần số, cùng với log$f$ và log$r$. Hãy dùng chương trình vẽ biểu đồ mà bạn chọn để vẽ đồ thị kết quả và kiểm tra xem liệu nó có hình thành một đường thẳng không. Bạn có thể ước tính giá trị của s không?

---

1. Nghiên cứu cụ thể này được dựa vào một ví dụ của Kernighan and Pike trong cuốn *The Practice of Programming*, 1999. ↩
2. Xem wikipedia.org/wiki/Zipf's_law. ↩

Chương 14: File

Trở về Mục lục cuốn sách

Sự duy trì dữ liệu

Phần lớn các chương trình ta đã thấy đến giờ đều có tính nhất thời về khía cạnh dữ liệu; theo nghĩa chúng chạy trong một thời gian ngắn, tạo ra kết quả nhưng khi kết thúc thì dữ liệu cũng biến mất. Nếu bạn chạy lại, chương trình sẽ bắt đầu với một trạng thái trống rỗng.

Các chương trình khác có tính **duy trì dữ liệu**: chúng chạy trong thời gian dài (hoặc luôn luôn chạy); và giữ lại ít nhất một phần dữ liệu trong một thiết bị lưu dữ liệu vĩnh viễn (ví dụ như ổ đĩa); và sau khi được kết thúc và khởi động lại, chương trình sẽ bắt đầu từ điểm mà chúng tạm dừng trước đây.

Các ví dụ về chương trình duy trì bao gồm các hệ điều hành, luôn được chạy mỗi khi máy tính được bật lên, và trình phục vụ web, vốn được chạy mọi lúc, đợi các yêu cầu gửi đến từ mạng máy tính.

Một trong những cách đơn giản nhất để chương trình có thể duy trì dữ liệu của chúng là bằng cách đọc và ghi các file chữ. Ta đã thấy các chương trình đọc được file chữ; ở chương này ta sẽ thấy chương trình ghi file.

Một cách khác là lưu trạng thái của chương trình vào trong một cơ sở dữ liệu. Ở chương này tôi sẽ trình bày một cơ sở dữ liệu đơn giản và một module, `pickle`, giúp cho việc lưu trữ dữ liệu của chương trình được dễ dàng.

Đọc và ghi

Một file chữ là một dãy các kí tự được lưu trên một thiết bị vĩnh viễn như một ổ đĩa cứng, bộ nhớ flash, hoặc đĩa CD-ROM. Ta đã thấy cách mở và đọc file ở Mục wordlist.

Để ghi một file, bạn cần mở nó theo chế độ `'w'` đặt như tham biến thứ hai:

```
>>> fout = open('output.txt', 'w')

>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

Nếu file đã tồn tại, việc mở nó với chế độ w (ghi) sẽ xóa sạch dữ liệu cũ và bắt đầu với file trống, nên hãy cẩn thận! Nếu file chưa tồn tại, sẽ có một file mới được tạo ra.

Phương thức `write` đưa dữ liệu vào trong file.

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
```

Một lần nữa, đối tượng file theo dõi vị trí hiện thời, vì vậy nếu bạn gọi lại `write` lần nữa, nó sẽ ghi dữ liệu mới vào cuối file.

```
>>> line2 = "the emblem of our land.\n"

>>> fout.write(line2)
```

Khi ghi xong, bạn phải đóng file lại.

```
>>> fout.close()
```

Toán tử định dạng

Đối số của `write` phải là một chuỗi, vì vậy nếu muốn đưa các giá trị khác vào một file, ta cần phải chuyển đổi chúng thành chuỗi. Cách dễ nhất để làm điều đó là dùng `str`:

```
>>> x = 52
>>> f.write(str(x))
```

Một cách khác là dùng **toán tử định dạng**, %. Khi áp dụng nó cho các số nguyên, % là toán tử chia lấy dư. Nhưng khi hạng tử thứ nhất là một chuỗi, % là toán tử định dạng.

Hạng tử thứ nhất là **chuỗi định dạng**, bao gồm một hoặc nhiều **dãy định dạng**, để chỉ định cách định dạng cho toán hạng thứ hai. Kết quả sẽ là một chuỗi.

Chẳng hạn, dãy định dạng `'%d'` có nghĩa là hạng tử thứ hai cần được định dạng như một số nguyên (`d` là viết tắt của "decimal"):

```
>>> camels = 42

>>> '%d' % camels
'42'
```

Kết quả là chuỗi `'42'`, mà ta không nên nhầm nó với giá trị nguyên `42`.

Một dãy định dạng có thể xuất hiện bất cứ đâu trong chuỗi, vì vậy bạn có thể gài một giá trị vào trong một câu:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

Nếu có nhiều dãy định dạng trong chuỗi thì hạng tử thứ hai phải là một bộ. Mỗi dãy định dạng được cặp với một phần tử của bộ theo đúng thứ tự.

Ví dụ sau sử dụng `'%d'` để định dạng một số nguyên `'%g'` để định dạng một số có phần thập phân (đừng hỏi lí do dùng kí hiệu này), và `'%s'` để định dạng một chuỗi:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

Số phần tử của bộ cần phải bằng số dãy định dạng có trong chuỗi. Ngoài ra, kiểu của các phần tử cũng phải hợp với kiểu của chuỗi định dạng:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

Ở ví dụ thứ nhất, không có đủ các phần tử; ở ví dụ thứ hai, phần tử có kiểu sai.

Toán tử định dạng rất đa năng, như có thể khó sử dụng. Bạn hãy tham khảo thêm thông tin ở docs.python.org/lib/typesseq-strings.html.

Tên file và đường dẫn

File được tổ chức trong các **thư mục**. Mỗi chương trình chạy có một "thư mục hiện thời", đó là thư mục mặc định cho hầu hết các thao tác. Chẳng hạn, khi bạn mở một file để đọc, Python tìm nó trong thư mục hiện thời.

Module os có các hàm làm việc với file và thư mục ("os" là viết tắt của "operating system"). os.getcwd trả lại tên của thư mục hiện thời:

```
>>> import os

>>> cwd = os.getcwd()
>>> print cwd
/home/dinsdale
```

cwd là viết tắt của "current working directory". Kết quả của ví dụ này là /home/dinsdale, chính là thư mục chủ của một người dùng có tên dinsdale.

Một chuỗi như cwd để nhận diện một file được gọi là **đường dẫn**. Một **đường dẫn tương đối** bắt đầu từ thư mục hiện thời; một **đường dẫn tuyệt đối** bắt đầu từ thư mục gốc của hệ thống file.

Các đường dẫn ta đã thấy đến giờ chỉ là tên file đơn giản, vì vậy chúng là đường dẫn tương đối đến thư mục hiện thời. Để tìm đường dẫn tuyệt đối đến một file, bạn có thể dùng os.path.abspath:

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

os.path.exists kiểm tra xem một file hoặc thư mục có tồn tại không:

```
>>> os.path.exists('memo.txt')
True
```

Nếu tồn tại, os.path.isdir sẽ kiểm tra xem nó có phải là thư mục không:

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('music')
True
```

Tương tự, os.path.isfile kiểm tra xem nó có phải là file không.

os.listdir trả lại một danh sách các file (và những thư mục khác) trong thư mục đã cho:

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

Để giới thiệu các hàm này, ví dụ sau có nhiệm vụ "dò" một thư mục, in ra tên của tất cả các file, và gọi bản thân nó một cách đệ quy với tất cả các thư mục.

```
def walk(dir):
    for name in os.listdir(dir):
        path = os.path.join(dir, name)

        if os.path.isfile(path):
            print path
        else:
            walk(path)
```

os.path.join nhận vào một thư mục và một tên file rồi nối chúng lại thành một đường dẫn hoàn chỉnh.

Hãy sửa lại walk để nó thay vì in ra các tên file thì trả lại một danh sách các tên.

Module os có một hàm tên là walk cũng giống như hàm trên nhưng linh hoạt hơn. Hãy đọc tài liệu và dùng hàm đó để in ra tên của các file trong thư mục đã cho và các thư

mục con.

Bắt các biệt lệ

Khi bạn đọc và ghi file, một loạt những lỗi có thể xảy ra. Nếu bạn mở một file hiện không tồn tại, bạn sẽ nhận lỗi `IOError`:

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

Nếu bạn không có quyền truy cập một file:

```
>>> fout = open('/etc/passwd', 'w')
IOError: [Errno 13] Permission denied: '/etc/passwd'
```

Và nếu bạn mở một thư mục để đọc, bạn sẽ nhận được lỗi:

```
>>> fin = open('/home')
IOError: [Errno 21] Is a directory
```

Để tránh các lỗi này, bạn có thể dùng các hàm như `os.path.exists` và `os.path.isfile`, nhưng sẽ mất nhiều thời gian và câu lệnh để kiểm tra hết tất cả các trường hợp có thể (nếu coi "`Errno 21`" là chỉ thị báo lỗi, thì ít nhất có 21 điều khác nhau có thể gây ra lỗi).

Tốt hơn là bạn bắt tay trực tiếp thử, và giải quyết các vấn đề nảy sinh, giống như cách mà câu lệnh `try` thực hiện. Cú pháp lệnh này cũng giống như lệnh `if`:

```
try:
    fin = open('bad_file')
    for line in fin:
        print line
    fin.close()
except:
    print 'Something went wrong.'
```

Python bắt đầu bằng việc thực hiện nhánh `try`. Nếu tất cả chạy tốt, nó sẽ bỏ qua nhánh `except` và tiếp tục. Nếu có biệt lệ xảy ra, nó sẽ nhảy khỏi nhánh `try` và thực hiện nhánh `except`.

Việc xử lý một biệt lệ bằng cách dùng lệnh `try` được gọi là **bắt** một biệt lệ. Ở ví dụ này, nhánh `except` in ra một thông báo lỗi không có ích lắm. Nói chung, việc bắt biệt lệ cho bạn một cơ hội sửa lỗi, hoặc thử lại, hoặc ít nhất là kết thúc chương trình một cách tốt đẹp.

Cơ sở dữ liệu

Cơ sở dữ liệu là một file được tổ chức để lưu dữ liệu. Đa số các cơ sở dữ liệu được tổ chức như các từ điển ở chỗ chúng là ánh xạ các khóa đến giá trị. Điểm khác biệt lớn nhất là cơ sở dữ liệu ở trên đĩa (hoặc một thiết bị lưu trữ vĩnh viễn khác), vì vậy nó còn tồn tại sau khi chương trình kết thúc.

Module `anydbm` cung cấp một giao diện phục vụ việc tạo lập và cập nhật file cơ sở dữ liệu. Tôi sẽ lấy một ví dụ cho việc tạo lập cơ sở dữ liệu chứa tiêu đề của các file hình ảnh.

Việc mở một cơ sở dữ liệu cũng giống như mở các file khác:

```
>>> import anydbm

>>> db = anydbm.open('captions.db', 'c')
```

Chế độ `'c'` có nghĩa là cơ sở dữ liệu sẽ được tạo mới nếu như nó chưa tồn tại. Kết quả là một đối tượng cơ sở dữ liệu có thể được dùng giống như một từ điển (ở hầu hết các thao tác). Nếu bạn tạo một mục mới, `anydbm` sẽ cập nhật file cơ sở dữ liệu.

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

Khi bạn truy cập đến một mục, anydbm sẽ đọc file:

```
>>> print db['cleese.png']
Photo of John Cleese.
```

Nếu bạn lại gán cho một khóa đã tồn tại, anydbm sẽ thay thế giá trị cũ:

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> print db['cleese.png']
Photo of John Cleese doing a silly walk.
```

Nhiều phương thức của từ điển, như keys và items, cũng có tác dụng với các đối tượng cơ sở dữ liệu. Việc lặp với lệnh for cũng như vậy.

```
for key in db:
    print key
```

Cũng như các file khác, bạn nên đóng cơ sở dữ liệu khi xong việc:

```
>>> db.close()
```

Bảo lưu dữ liệu

Hạn chế của anydbm là ở chỗ các khóa và trị phải là các chuỗi. Nếu bạn thử dùng bất cứ kiểu nào khác, bạn sẽ nhận được lỗi.

Module pickle[1] có thể khắc phục điều trên. Nó dịch hầu hết các kiểu đối tượng ra kiểu chuỗi phù hợp cho việc lưu trong cơ sở dữ liệu, và sau đó lại dịch các chuỗi trở lại thành đối tượng.

pickle.dumps nhận vào một đối tượng như một tham biến và trả về dạng chuỗi biểu diễn cho nó (dumps và viết tắt của “dump string”):

```
>>> import pickle

>>> t = [1, 2, 3]
>>> pickle.dumps(t)
'(lp0\nI1\naI2\naI3\na.'
```

Dạng này không dễ đọc; nó phù hợp để pickle diễn giải. pickle.loads (“load string”) lại khôi phục đối tượng:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)

>>> print t2
[1, 2, 3]
```

Mặc dù đối tượng mới có cùng giá trị với đối tượng cũ, nói chung là chúng không đồng nhất với nhau:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

Nói cách khác, việc bảo quản và phục hồi sử dụng có cùng tác dụng như sao chép đối tượng.

Bạn có thể dùng `pickle` để lưu các kiểu dữ liệu không phải chuỗi trong cơ sở dữ liệu. Thực ra, sự kết hợp này thông dụng đến nỗi nó được gói trong một module tên là `shelve`.

> Nếu bạn đã làm Bài tập anagrams, hãy sửa lại lời giải sao cho nó tạo ra một cơ sở dữ liệu ánh xạ từ mỗi mục từ trong danh sách đến một danh sách các từ cũng được lập bởi các chữ cái của từ gốc.
>
> Hãy viết một chương trình khác để mở cơ sở dữ liệu và in nội dung ra dưới dạng mà chúng ta có thể đọc được.

Ống dẫn dữ liệu

Hầu hết các hệ điều hành đều cung cấp một giao diện dòng lệnh, cũng được gọi là **lớp vỏ**. Lớp vỏ hệ điều hành thường có các câu lệnh để di chuyển trong hệ thống file và khởi động các ứng dụng. Chẳng hạn, trong Unix, bạn có thể thay đổi thưu mục bằng `cd`, hiển thị nội dung thư mục bằng `ls`, và khởi động một trình duyệt web bằng cách gõ vào một lệnh như `firefox`.

Bất kì chương trình nào bạn khởi động từ lớp vỏ cũng có thể được khởi động từ Python bằng cách dùng một **ống dẫn**. Ống dẫn là đối tượng để biểu thị một quá trình đang chạy.

Chẳng hạn, lệnh Unix `ls -l` thường để hiển thị nội dung của thư mục hiện thời (theo dạng đầy đủ). Bạn có thể khởi động `ls` bằng `os.popen`:

```
>>> cmd = 'ls -l'

>>> fp = os.popen(cmd)
```

Tham biến là một chuỗi trong đó có chứa một lệnh của lớp vỏ. Giá trị được trả lại là một đối tượng có biểu hiện giống như một file đang mở. Bạn có thể đọc kết quả đầu ra từ quá trình `ls` theo từng dòng một bằng `readline` hoặc đọc tất cả một lượt bằng `read`:

```
>>> res = fp.read()
```

Khi đã xong việc, bạn đóng ống dẫn như làm với file:

```
>>> stat = fp.close()
>>> print stat
None
```

Giá trị được trả về là trạng thái cuối cùng của quá trình `ls`; `None` có nghĩa là nó kết thúc bình thường (không có lỗi).

Một cách sử dụng ống dẫn là đọc vào dần dần một file được nén; nghĩa là không giải nén toàn bộ file cùng một lúc. Hàm sau đây nhận vào tên của file nén như một tham biến và trả về một ống dẫn có dùng `gunzip` để giải nén nội dung:

```
def open_gunzip(filename):
    cmd = 'gunzip -c ' + filename
    fp = os.popen(cmd)
    return fp
```

Nếu đọc từng dòng một từ `fp`, bạn không bao giờ phải lưu file được giải nén trên bộ nhớ hoặc đĩa.

Viết các module

Mọi file chứa mã lệnh Python đều có thể được nhập vào như một module. Chẳng hạn, nếu bạn có một file tên là `wc.py` với mã lệnh sau:

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print linecount('wc.py')
```

Nếu bạn chạy chương trình này, nó sẽ đọc chính nó và in ra số dòng trong file, tức là 7. Bạn cũng có thể nhập nó vào như sau:

```
>>> import wc
7
```

Bây giờ bạn có một đối tượng module tên là `wc`:

```
>>> print wc
<module 'wc' from 'wc.py'>
```

Module này có một hàm tên là `linecount`:

```
>>> wc.linecount('wc.py')
7
```

Như vậy đó là cách bạn viết các module trong Python:

Vấn đề duy nhất ở ví dụ này là khi bạn nhập vào module nó chạy đoạn mã lệnh ở dưới cùng. Thông thường khi bạn nhập vào một module, nó định nghĩa các hàm mới như không thực hiện chúng.

Những chương trình được nhập như là module thường có cách viết sau đây:

```
if __name__ == '__main__':
    print linecount('wc.py')
```

`__name__` là một biến có sẵn, nó được đặt mỗi khi chương trình bắt đầu. Nếu chương trình chạy dưới dạng một văn lệnh, `__name__` sẽ có giá trị `__main__`; trong trường hợp đó, đoạn mã lệnh kiểm tra sẽ được thực hiện. Ngược lại, nếu module được nhập vào, mã lệnh kiểm tra sẽ được bỏ qua.

> Hãy gõ ví dụ sau đây vào trong một file tên là `wc.py` và chạy nó như một văn lệnh. Sau đó khởi động trình thông dịch Python và gõ vào `import wc`. Giá trị của `__name__` sẽ là gì khi module được nhập vào?
>
> Lưu ý: Nếu bạn nhập vào một module mà đã được nhập từ trước, Python sẽ không làm gì cả. Nó không đọc lại file nữa, ngay cả khi file này bị thay đổi.
>
> Nếu bạn muốn tải lại một module, bạn có thể dùng hàm có sẵn `reload`, nhưng khi dùng phải cẩn thận, và an toàn nhất là khởi động lại trình thông dịch và nhập lại module một lần nữa.

Gỡ lỗi

Trong quá trình đọc và ghi file, bạn có thể gặp vấn đề liên quan đến dấu trắng. Những lỗi kiểu đó có thể khó gỡ vì các dấu cách, dấu tab và dấu xuống dòng thông thường đều không hiện rõ trên màn hình:

```
>>> s = '1 2\t 3\n 4'

>>> print s
1 2  3
 4
```

Hàm có sẵn `repr` có thể giúp ích. Nó nhận vào bất kì đối tượng nào như một đối số và trả về một chuỗi biểu diễn đối tượng đó. Với các chuỗi, nó biểu diễn dấu trắng bởi các dãy có dấu gạch chéo ngược:

```
>>> print repr(s)
'1 2\t 3\n 4'
```

Điều này có thể giúp ích cho việc gỡ lỗi.

Một vấn đề khác mà bạn có thể gặp phải là các hệ thống khác nhau dùng những kí tự khác nhau để biểu thị chỗ kết thúc dòng. Một số hệ thống dùng kí tự dòng mới, biểu thị bởi `\n`. Một số khác dùng kí tự trở về, kí hiệu `\r`. Một số khác lại dùng cả hai. Nếu bạn chuyển file giữa các hệ thống khác nhau, sự không thống nhất này có thể làm nảy sinh vấn đề.

Với phần lớn các hệ thống, có những trình ứng dụng để chuyển giữa các dạng. Bạn có thể tìm chúng (hoặc đọc thêm về điều này) ở wikipedia.org/wiki/Newline. Hoặc tất nhiên là bạn có thể tự viết một chương trình.

Thuật ngữ

duy trì:
: Tính chất của một chương trình chạy lâu dài và giữ lại ít nhất là một phần dữ liệu của nó trong thiết bị lưu trữ vĩnh viễn.

toán tử định dạng:
: Toán tử %, nhận vào một chuỗi định dạng và một bộ rồi phát sinh ra một chuỗi trong đó bao gồm các phần tử của bộ được định dạng như chỉ định trong chuỗi định dạng.

chuỗi định dạng:
: Chuỗi được dùng với toán tử định dạng, gồm các dãy định dạng.

dãy định dạng:
: Dãy kí tự trong một chuỗi định dạng, như %d, nhằm chỉ rõ cách định dạng cho một giá trị.

file chữ:
: Dãy các kí tự được lưu trong một thiết bị lưu trữ vĩnh viễn, như một ổ cứng.

thư mục:
: Tập hợp được đặt tên chứa các file.

chuỗi:
: Chuỗi để chỉ định một file.

đường dẫn gián tiếp:
Đường dẫn bắt nguồn từ thư mục hiện thời.

đường dẫn trực tiếp:
Đường dẫn bắt nguồn từ thư mục gốc trong hệ thống file.

bắt:
Thao tác ngăn ngừa việc biệt lệ xảy ra làm kết thúc chương trình, được thực hiện bằng các lệnh `try` và `except`.

cơ sở dữ liệu:
Một file mà nội dung được tổ chức như một từ điển với các khóa tương ứng với các giá trị.

Bài tập

Module `urllib` có các phương thức thao tác với các URL và tải về thông tin từ mạng. Ví dụ sau tải về và in ra một thông điệp bí mật từ `thinkpython.com`:

```
import urllib

conn = urllib.urlopen('http://thinkpython.com/secret.html')
for line in conn.fp:
    print line.strip()
```

Hãy chạy đoạn mã lệnh này và thực hiện các chỉ dẫn được hiện ra.

Trong một tập hợp gồm rất nhiều file MP3, có thể có nhiều bản sao của cùng một bài hát lưu trong nhiều thư mục hoặc lưu dưới nhiều tên khác nhau. Mục đích của bài tập này là tìm những bản trùng tên đó.

1. Hãy viết một chương trình để tìm một thư mục và tất cả các thư mục con của nó theo cách đệ quy, và trả về một danh sách các đường dẫn đầy đủ tới tất cả các file có phần đuôi cho trước (như `.mp3`). Gợi ý: `os.path` có một số hàm giúp ích cho việc xử lý file và tên đường dẫn.
2. Để nhận biết được sự trùng lặp, bạn có thể dùng một hàm băm đọc vào file và phát sinh một đoạn tóm tắt cho nội dung file. Chẳng hạn, MD5 (Message-Digest algorithm 5) nhận vào một "thông điệp" dài tùy ý và trả lại một "checksum" 128 bit. Xác suất để hai nội dung khác nhau có thể trả về cùng checksum là rất nhỏ.

   Bạn có thể tìm hiểu về MD5 tại www.wikipedia.org/wiki/Md5. Trong hệ thống Unix, bạn có thể dùng chương trình `md5sum` và một ống dẫn để tính checksum từ Python.

Cơ sở dữ liệu Phim ảnh trên Internet, Internet Movie Database (IMDb) là một tuyển tập trực tuyến lưu giữ các thông tin về những cuốn phim. Cơ sở dữ liệu này cũng sẵn có dưới hình thức văn bản chữ, vì vậy sẽ khá dễ đọc từ Python. Trong bài tập này, các file bạn cần là `actors.list.gz` và `actresses.list.gz`; bạn có thể tải chúng về từ www.imdb.com/interfaces#plain.

Tôi đã viết một chương trình để tách các file này và phân nội dung riêng thành tên diễn

viên, tựa đề phim, v.v. Bạn có thể tải chúng về từ thinkpython.com/code/imdb.py.

Nếu bạn chạy `imdb.py` như một văn lệnh, nó sẽ đọc `actors.list.gz` và in ra mỗi dòng một cặp diễn viên-bộ phim. Hoặc, nếu `import imdb` bạn có thể dùng hàm `process_file` để xử lý file. Các đối số gồm có tên file, một đối tượng hàm và tùy chọn là số các dòng cần xử lý. Sau đây là một ví dụ:

```
import imdb

def print_info(actor, date, title, role):
    print actor, date, title, role

imdb.process_file('actors.list.gz', print_info)
```

Khi bạn gọi `process_file`, nó sẽ mở `filename`, đọc nội dung, và gọi `print_info` một lần cho mỗi dòng trong file. `print_info` nhận các đối số gồm có một tên diễn viên, ngày tháng, tựa đề phim và vai diễn rồi in chúng ra.

1. Hãy viết một chương trình để đọc vào `actors.list.gz` và `actresses.list.gz` rồi dùng `shelve` để lập một cơ sở dữ liệu trong đó ánh xạ từ tên từng diễn viên đến một danh sách các bộ phim của diễn viên đó.
2. Hai diễn viên là "đồng minh tinh" nếu họ diễn cùng trong ít nhất là một bộ phim. Hãy xử lý cơ sở dữ liệu vừa lập ra ở bước trên, và lập một cơ sở dữ liệu thứ hai trong đó ánh xạ tên từng diễn viên đến một danh sách các đồng minh tinh của diễn viên đó.
3. Hãy viết một chương trình để chơi trò "Six Degrees of Kevin Bacon", mà bạn có thể tìm hiểu tại wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon. Vấn đề này rất khó vì nó đòi hỏi phải tìm đường đi ngắn nhất trong một đồ thị. Bạn có thể tìm hiểu về các thuật toán đường đi ngắn nhất tại wikipedia.org/wiki/Shortest_path_problem.

---

1. Pickle = ngâm muối, ướp muối để bảo quản (nghĩa gốc tiếng Anh) ↩

Chương 15: Lớp và đối tượng

Trở về Mục lục cuốn sách

Các kiểu dữ liệu do người dùng định nghĩa

Ta đã dùng nhiều kiểu dữ liệu có sẵn trong Python; bây giờ ta sẽ định nghĩa một kiểu mới. Lấy ví dụ, ta sẽ tạo ra một kiểu gọi là `Point` để biểu diễn một điểm trong không gian hai chiều.

Theo ngôn ngữ toán học, điểm thường được viết trong một cặp ngoặc đơn với dấu phẩy ngăn cách giữa các tọa độ. Chẳng hạn, (0, 0) biểu diễn gốc tọa độ, và (x, y) biểu diễn điểm x đơn vị về bên tay phải và y đơn vị độ dài lên phía trên so với điểm gốc.

Có một số cách để diễn tả một điểm bằng Python:

- Ta có thể lưu giữ các tọa độ một cách riêng rẽ trong hai biến, `x` và `y`.
- Ta có thẻ lưu giữ các tọa độ như những phần tử trong một danh sách hoặc một bộ.
- Ta có thể tạo ra một kiểu mới để diễn tả điểm như những đối tượng.

Việc tạo ra một kiểu mới sẽ phức tạp hơn (một chút) so với những cách còn lại, nhưng những ưu điểm của nó sẽ sớm trở nên rõ ràng sau này.

Một kiểu dữ liệu do người dùng định nghĩa được gọi là **lớp**. Một định nghĩa lớp sẽ có dạng như sau:

```
class Point(object):
    """represents a point in 2-D space"""
```

Đoạn đầu này cho thấy lớp mới này là một `Point`, vốn là một kiểu `object`, và là một kiểu có sẵn.

Phần thân là một chuỗi ghi chú để giải thích mục đích của lớp này. Bạn có thể định nghĩa các biến và hàm bên trong một định nghĩa lớp, nhưng ta sẽ trở lại điều này sau.

Việc định nghĩa một lớp tên là `Point` sẽ tạo ra một đối tượng lớp.

```
>>> print Point
<class '__main__.Point'>
```

Vì `Point` được định nghĩa ở cấp cao nhất, "tên đầy đủ" của nó là `__main__.Point`.

Đối tượng lớp giống như một xưởng chế tạo ra các đối tượng. Để tạo ra một Point, bạn gọi `Point` như thể nó là một hàm.

```
>>> blank = Point()
>>> print blank
<__main__.Point instance at 0xb7e9d3ac>
```

Giá trị trả về là một tham chiếu tới một đối tượng Point, mà ta gán bằng `blank`. Việc tạo ra đối tượng mới được gọi là **tạo cá thể**, và đối tượng là một **cá thể** của lớp này.

Khi bạn in ra một cá thể, Python sẽ nói cho bạn biết cá thể này thuộc lớp nào và nó được lưu vào đâu trong bộ nhớ (đoạn đầu `0x` nghĩa là số tiếp sau sẽ tính theo hệ thập lục phân).

Thuộc tính

Bạn có thể gán các giá trị cho một cá thể bằng cách dùng kí hiệu dấu chấm:

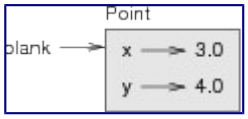
```
>>> blank.x = 3.0
```

```
>>> blank.y = 4.0
```

Cú pháp này cũng giống như khi ta viết lệnh chọn một biến từ một module, như `math.pi` hoặc `string.whitespace`. Tuy vậy, ở trường hợp này, ta đang gán các giá trị cho các phần tử được đặt tên của một đối tượng. Những phần tử này được gọi là **thuộc tính**.

Trong tiếng Anh, thuộc tính là danh từ, được phát âm "AT-trib-ute" với trọng âm đặt vào âm tiết đầu, khác với từ "a-TRIB-ute", là một động từ.

Sơ đồ sau cho thấy kết quả của các phép gán này. Một sơ đồ trạng thái chỉ ra đối tượng và các thuộc tính của nó được gọi là **sơ đồ đối tượng**:



Biến `blank` chỉ đến một đối tượng Point, vốn chứa hai thuộc tính. Mỗi thuộc tính lại chỉ đến một số có phần thập phân.

Bạn có thể đọc giá trị của thuộc tính bằng cú pháp tương tự:

```
>>> print blank.y
4.0

>>> x = blank.x
>>> print x
3.0
```

Biểu thức `blank.x` nghĩa là, "Đến đối tượng mà `blank` chỉ tới và lấy giá trị của `x`". Trong trường hợp này, ta gán giá trị cho một biến tên là `x`. Không hề có sự xung đột giữa biến `x` và thuộc tính `x`.

Bạn có thể dùng kí hiệu dấu chấm như một phần của biểu thức bất kì. Chẳng hạn:

```
>>> print '(%g, %g)' % (blank.x, blank.y)
(3.0, 4.0)
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> print distance
5.0
```

Bạn có thể truyền một cá thể với vai trò của đối số theo cách thông thường. Chẳng hạn:

```
def print_point(p):
    print '(%g, %g)' % (p.x, p.y)
```

`print_point` nhận vào đối số là một điểm và hiển thị nó theo kiểu kí hiệu toán học. Để sử dụng hàm này, bạn có thể truyền `blank` như một đối số:

```
>>> print_point(blank)
(3.0, 4.0)
```

Bên trong hàm, `p` là một tên khác của `blank`, vì vậy nếu hàm làm thay đổi `p` thì `blank` cũng thay đổi theo.

> Hãy viết một hàm tên là `distance` nhận vào hai điểm làm đối số và và trả lại khoảng cách giữa hai điểm đó.

Hình chữ nhật

Đôi khi có thể dễ thấy rằng các thuộc tính của một đối tượng phải là gì, nhưng có những trường hợp bạn phải quyết định. Chẳng hạn, hãy hình dung rằng bạn cần thiết kế một lớp để biểu diễn hình chữ nhật. Bạn sẽ sử dụng những yếu tố gì để đặc trưng cho vị trí và kích thước của một hình chữ nhật. Để đơn giản, ta không xét đến góc quay của hình chữ nhật và coi rằng nó được đặt thẳng đứng hay nằm ngang.

Có ít nhất là hai khả năng sau đây:

- Bạn có thể chỉ định một đỉnh (điểm góc) của hình chữ nhật (hoặc tâm điểm), bề rộng, và chiều cao.
- Bạn có thể chỉ định hai đỉnh đối diện.

Đến đây ta khó nói trước rằng cách nào tốt hơn, vì vậy để làm ví dụ ta sẽ làm theo cách thứ nhất.

Sau đây là định nghĩa lớp:

```
class Rectangle(object):
    """represent a rectangle.
       attributes: width, height, corner.
    """
```

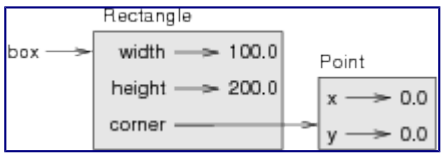
Chuỗi chú thích liệt kê các thuộc tính: `width` và `height` là các số; `corner` là một đối tượng điểm chỉ định đỉnh góc trái phía dưới.

Để biểu diễn một hình chữ nhật, bạn phải tạo cá thể thuộc đối tượng Rectangle và gán giá trị cho thuộc tính:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

Biểu thức `box.corner.x` có nghĩa là, "Đến đối tượng mà `box` chỉ đến và chọn lấy thuộc tính có tên là `corner`; sau đó đến đối tượng đó và chọn lấy thuộc tính có tên là `x`".

Hình vẽ sau cho thấy trạng thái của đối tượng này:



Một đối tượng được gọi là được **nhúng** nếu nó là thuộc tính của một đối tượng khác.

Cá thể với vai trò là giá trị trả về

Các hàm có thể trả về các cá thể. Chẳng hạn, `find_center` nhận vào đối số là một `Rectangle` và trả lại một `Point` trong đó chứa tọa độ của tâm điểm `Rectangle`:

```
def find_center(box):
    p = Point()
    p.x = box.corner.x + box.width/2.0
    p.y = box.corner.y + box.height/2.0
    return p
```

Sau đây là một ví dụ trong đó truyền box như một đối số và gán Point thu được cho center:

```
>>> center = find_center(box)
>>> print_point(center)
(50.0, 100.0)
```

Bạn có thể thay đổi trạng thái của một đối tượng bằng cách thực hiện phép gán với một thuộc tính của nó. Chẳng hạn, để điều chỉnh kích thước của hình chữ nhật mà không làm dịch chuyển nó, bạn có thể thay đổi các giá trị của width và height:

```
box.width = box.width + 50
box.height = box.width + 100
```

Bạn cũng có thể viết các hàm để thay đổi đối tượng. Chẳng hạn, grow_rectangle nhận vào một đối tượng Rectangle và hai số, dwidth và dheight, rồi cộng các số này với bề rộng và chiều cao của hình chữ nhật:

```
def grow_rectangle(rect, dwidth, dheight) :
    rect.width += dwidth
    rect.height += dheight
```

Sau đây là một ví dụ minh họa tác dụng của hàm này:

```
>>> print box.width
100.0

>>> print box.height
200.0
>>> grow_rectangle(box, 50, 100)
>>> print box.width
150.0
>>> print box.height
300.0
```

Ở trong hàm, rect là một tên khác của box, vì vậy nếu hàm làm thay đổi rect, box cũng thay đổi theo.

> Hãy viết một hàm có tên move_rectangle nhận vào một Rectangle và hai số tên là dx và dy. Hàm này cần thay đổi vị trí của hình chữ nhật bằng cách cộng dx với tọa độ x của corner và cộng dy với tọa độ y của corner.

Sao chép

Việc dùng tham chiếu bội có thể làm chương trình khó đọc vì sự thay đổi ở một chỗ có thể gây ảnh hưởng không ngờ được ở chỗ khác. Thật khó có thể theo dõi được tất cả các biến có khả năng chỉ đến cùng một đối tượng.

Sao chép một đối tượng thường là cách làm thay thế cho tham chiếu bội. Module copy có một hàm tên là copy có thể sao chép bất kì đối tượng nào:

```
>>> p1 = Point()
>>> p1.x = 3.0

>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

p1 và p2 chứa dữ liệu giống nhau, nhưng chúng không cùng là một điểm (Point).

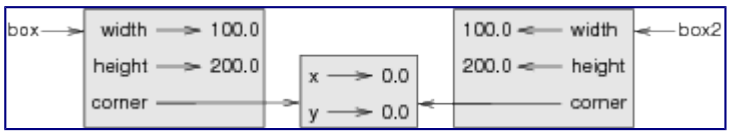
```
>>> print_point(p1)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)

>>> p1 is p2
False
>>> p1 == p2
False
```

Toán tử `is` cho thấy rằng p1 và p2 là các đối tượng khác nhau, như chúng ta mong đợi. Nhưng có lẽ bạn cũng mong đợi rằng == sẽ cho kết quả `True` vì những điểm này có số liệu như nhau. Nếu vậy, bạn sẽ thất vọng khi biết rằng đối với các thể, toán tử == sẽ hoạt động mặc định như toán tử `is`; nó kiểm tra sự đồng nhất của hai đối tượng, chứ không phải sự tương đương giữa chúng. Tính năng này vẫn có thể thay đổi được—sau này ta sẽ biết cách thay đổi.

Nếu dùng `copy.copy` để tạo bản sao cho một Rectangle, bạn sẽ thấy rằng nó sao chép đối tượng Rectangle nhưng không sao chép đối tượng Point được nhúng trong đó.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Sơ đồ đối tượng sẽ có dạng sau:



Thao tác trên được gọi là **sao chép nông** bởi vì nó chỉ sao chép đối tượng và các tham chiếu trong đó, mà không sao chép các đối tượng nhúng.

Trong phần lớn các trường hợp, đây không phải là điều bạn muốn. Ở ví dụ này, việc gọi `grow_rectangle` từ một trong hai Rectangle sẽ không làm ảnh hưởng đến đối tượng kia, nhưng `move_rectangle` từ một đối tượng sẽ làm ảnh hưởng đến cả hai! Hiện tượng này có thể làm ta bối rối và dễ gây lỗi.

Thật may là module `copy` có một phương thức tên là `deepcopy` để sao chép không chỉ bản thân đối tượng mà còn cả những đối tượng mà nó chỉ đến, rồi cả những đối tượng mà *chúng* chỉ đến, và cứ như vậy. Không ngạc nhiên khi người ta gọi thao tác này là **sao chép sâu**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

box3 và box là các đối tượng hoàn toàn riêng biệt.

> Hãy viết một dạng của `move_rectangle` để tạo ra và trả về một Rectangle mới thay vì sửa lại cá thể cũ.

Gỡ lỗi

Khi bắt tay làm việc với các đối tượng, bạn có thể sẽ gặp một số biệt lệ mới. Nếu bạn thử truy cập một thuộc tính không có sẵn, bạn sẽ nhận `AttributeError`:

```
>>> p = Point()
>>> print p.z
AttributeError: Point instance has no attribute 'z'
```

Nếu không biết rõ kiểu của một đối tượng nào đó, bạn có thể hỏi:

```
>>> type(p)
<type '__main__.Point'>
```

Nếu không biết rõ liệu một đối tượng có một thuộc tính nào đó, bạn có thể dùng hàm có sẵn `hasattr`:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

Đối số thứ nhất có thể là một đối tượng bất kì; đối số thứ hai là một *chuỗi* bao gồm tên của thuộc tính.

Thuật ngữ

lớp:
: Kiểu dữ liệu do người dùng định nghĩa. Lời định nghĩa lớp sẽ tạo ra một đối tượng lớp mới.

đối tượng lớp:
: Đối tượng trong đó bao gồm thông tin về một kiểu do người dùng định nghĩa. Đối tượng lớp có thể được dùng để tạo ra các cá thể của kiểu này.

cá thể:
: Đối tượng thuộc về một lớp.

thuộc tính:
: Một trong các giá trị được đặt tên gắn với một đối tượng.

nhúng (đối tượng):
: Đối tượng được lưu trữ như một thuộc tính của một đối tượng khác.

sao chép nông:
: Việc sao chép nội dung của một đối tượng, bao gồm cả những tham chiếu đến các đối tượng nhúng; được thực hiện bởi hàm `copy` trong module `copy`.

sao chép sâu:
: Việc sao chép nội dung của một đối tượng cùng tất cả đối tượng nhúng nếu có, và các đối tượng nhúng bên trong chúng, và cứ như vậy; được thực hiện bởi hàm `deepcopy` trong module `copy`.

sơ đồ đối tượng:
: Sơ đồ biểu diễn các đối tượng, thuộc tính của chúng và các giá trị của thuộc tính.

Bài tập

`World.py` trong Swampy (see Chapter {turtlechap}) có chứa định nghĩa lớp cho một kiểu người dùng định nghĩa tên là `World`. Bạn có thể nhập nó như sau:

```
from World import World
```

Phiên bản này của lệnh `import` nhập vào lớp `World` từ module `World`. Đoạn mã lệnh sau tạo ra một đối tượng World và gọi phương thức `mainloop`, để đợi hoạt động từ phía người dùng.

```
world = World()
world.mainloop()
```

Một cửa sổ sẽ xuất hiện với thanh tiêu đề và một hình vuông trống không. Ta sẽ dùng cửa sổ này để vẽ các điểm, hình chữ nhật, và các hình khác. Hãy thêm các dòng lệnh sau đây trước khi gọi `mainloop` và chạy lại chương trình.

```
canvas = world.ca(width=500, height=500, background='white')
bbox = [[-150,-100], [150, 100]]
canvas.rectangle(bbox, outline='black', width=2, fill='green4')
```

Bạn sẽ thấy một hình chữ nhật màu xanh lá cây với viền đen. Dòng lệnh thứ nhất tạo ra Canvas, vốn xuất hiện như một hình vuông màu trắng trên cửa sổ. Đối tượng Canvas có các phương thức như `rectangle` để vẽ nhiều hình dạng khác nhau.

`bbox` là một danh sách chứa các danh sách biểu diễn "hình bao" của hình chữ nhất. Cặp tọa độ thứ nhất là của đỉnh góc dưới bên trái; cặp thứ hai của đỉnh góc phía trên bên phải.

Bạn có thể vẽ một đường tròn như sau:

```
canvas.circle([-25,0], 70, outline=None, fill='red')
```

Tham số thứ nhất là cặp tọa độ tâm của đường tròn; tham số thứ hai là bán kính.

Nếu bạn thêm dòng này vào chương trình, kết quả thu được sẽ giống như quốc kì của Bangladesh (xem wikipedia.org/wiki/Gallery_of_sovereign-state_flags).

1. Hãy viết một hàm tên là `draw_rectangle` nhận vào một Canvas và Rectangle làm các đối số và vẽ hình biểu diễn Rectangle trên Canvas.
2. Hãy bổ sung một thuộc tính tên là `color` cho các đối tượng Rectangle và sửa đổi `draw_rectangle` sao cho nó dùng thuộc tính màu này làm màu tô.
3. Hãy viết một hàm tên là `draw_point` nhận vào một Canvas và một Point làm các đối số và vẽ hình biểu diễn Point trên Canvas.
4. Hãy định nghĩa một lớp mới có tên là Circle với các thuộc tính phù hợp rồi tạo ra một số cá thể thuộc đối tượng Circle. Hãy viết một hàm tên là `draw_circle` để vẽ các Circle trên Canvas.
5. Hãy viết một chương trình để vẽ hình quốc kì của nước Cộng hòa Séc. Gợi ý: bạn có thể vẽ một hình đa giác như sau:

```
points = [[-150,-100], [150, 100], [150, -100]]
canvas.polygon(points, fill='blue')
```

Tôi đã viết một chương trình ngắn nhằm liệt kê các màu sẵn có; bạn có thể tải nó về từ thinkpython.com/code/color_list.py.

Chương 16: Lớp và hàm

Trở về Mục lục cuốn sách

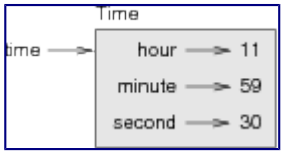
Để lấy một ví dụ khác về kiểu người dùng định nghĩa, ta sẽ định nghĩa một lớp tên là `Time` để ghi lại thời gian trong ngày. Lời định nghĩa của lớp này như sau:

```
class Time(object):
    """represents the time of day.
       attributes: hour, minute, second"""
```

Ta có thể tạo ra một đối tượng `Time` mới và gán các thuộc tính cho số giờ, phút, và giây:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

Sơ đồ trạng thái cho đối tượng `Time` trông như sau:



> Hãy viết một hàm có tên `print_time` nhận vào một đối tượng Time và in nội dung ra dưới dạng `hour:minute:second`. Gợi ý: dãy định dạng `'%.2d'` dùng để in ra một số nguyên với ít nhất là hai chữ số, trong đó có một chữ số 0 đứng đầu nếu cần.
>
> Hãy viết một hàm boole có tên `is_after` nhận vào hai đối tượng Time, `t1` và `t2`, rồi trả lại `True` nếu `t1` xếp sau `t2` về mặt thời gian và `False` nếu ngược lại. Đố bạn thực hiện mà không cần lệnh `if` ?

Hàm thuần túy

Ở các mục tiếp theo, ta sẽ viết hai hàm để cộng các giá trị thời gian. Chúng sẽ đại diện cho hai loại hàm: hàm thuần túy và hàm sửa đổi. Đây cũng là minh họa cho một kế hoạch phát triển mà tôi gọi là **hình mẫu và sửa chữa**, một cách xử lý vấn đề khó bằng cách bắt đầu bằng một hình mẫu đơn giản và dần xử lý những điểm phức tạp.

Sau đây là một hình mẫu đơn giản của `add_time`:

```
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
```

Hàm này tạo ra một đối tượng `Time` mới, khởi tạo các thuộc tính của nó, và trả lại một tham chiếu đến đối tượng mới. Nó được gọi là một **hàm thuần túy** vì không làm thay đổi gì đến đối tượng được truyền đến như đối số; và nó cũng không có hiệu ứng như hiển thị một giá trị hay thu kết quả do người nhập vào, ngoại trừ việc trả lại một giá trị.

Để kiểm tra hàm này, tôi sẽ tạo ra hai đối tượng Time: `start` chứa thời điểm bắt đầu chiếu một bộ

phim, như *Monty Python and the Holy Grail*, cùng với `duration` chứa thời lượng của phim, bằng 1 giờ 35 phút.

`add_time` sẽ tính ra khi nào bộ phim kết thúc.

```
>>> 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
```

Kết quả, `10:80:00` có thể không được như bạn mong đợi. Vấn đề là ở chỗ hàm này không tính được các trường hợp mà số phút hoặc số giây cộng lại lớn hơn 60. Khi đó, ta cần phải "nhớ" đưa số giây còn dư vào trong cột số phút, hoặc số phút còn dư vào trong cột số giờ.

Sau đây là một phiên bản được sửa chữa:

```
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
```

Mặc dù hàm này đã đúng, nó bắt đầu phình to lên. Chúng ta sẽ xét đến một cách khác ngay sau đây.

Hàm sửa đổi

Đôi sẽ hữu ích hơn khi một hàm có thể thay đổi các đối tượng khi chúng được truyền vào như tham biến. Trong trường hợp này, hàm gọi sẽ nhận thấy được các thay đổi đó. Những hàm kiểu này được gọi là **hàm thay đổi**.

Có thể viết `increment`, một hàm để cộng một số giây cho trước vào đối tượng `Time`, dưới dạng một hàm thay đổi theo cách tự nhiên. Sau đây là một bản phác thảo:

```
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
```

Dòng đầu tiên thực hiện phép tính cơ bản; phần còn lại xử lý các trường hợp đặc biệt như ta đã thấy trước đây.

Hàm này có đúng không? Điều gì sẽ xảy ra khi tham biến `seconds` lớn hơn nhiều so với 60?

Trong trường hợp như vậy, việc cộng nhớ một lần là chưa đủ; ta còn phải tiếp tục làm đến khi `time.second` dưới 60. Một cách giải quyết là thay các lệnh `if` bằng các lệnh `while`. Việc này sẽ giúp ta có một hàm đúng, nhưng không hiệu quả lắm.

> Hãy viết một dạng đúng của `increment` mà không chứa vòng lặp nào.

Hàm thuần túy có thể làm được bất cứ công việc nào thực hiện bởi hàm thay đổi. Thật ra, một số ngôn ngữ lập trình chỉ cho phép dùng hàm thuần túy. Đã có một số bằng chứng cho thấy những chương trình chỉ dùng hàm thuần túy thì được phát triển nhanh hơn và ít gây lỗi hơn so với chương trình dùng hàm thay đổi. Nhưng có những lúc dùng hàm thay đổi lại thuận tiện, và chương trình dùng hàm thuần túy lại kém hiệu quả.

Nói chung, tôi khuyên bạn dùng hàm thuần túy mỗi khi có thể, và chỉ dùng hàm thay đổi khi có một ưu điểm rõ rệt. Cách tiếp cận này có thể được gọi là **phong cách lập trình hàm**.

> Hãy viết một hàm thuần túy cho `increment` để tạo ra và trả về một đối tượng Time mới thay vì thay đổi tham biến.

So sánh việc tạo nguyên mẫu với lập kế hoạch

Kế hoạch phát triển mà tôi sẽ trình bày sau đây được gọi là "nguyên mẫu và sửa chữa[1]". Với mỗi hàm, tôi viết một nguyên mẫu để thực hiện những phép tính cơ bản và sau đó chạy thử nó, đồng thời sửa chữa những lỗi phát sinh.

Cách làm này có thể hiệu quả, đặc biệt nếu bạn chưa có hiểu biết sâu sắc về bài toán. Nhưng các sửa chữa dần có thể hình thành mã lệnh phức tạp quá mức cần thiết—vì nó giải quyết nhiều trường hợp đặc biệt—và không đáng tin cậy—vì bạn khó có thể biết được rằng liệu đã tìm được tất cả lỗi hay chưa.

Một cách làm khác là **phát triển theo kế hoạch**, trong đó những nhận định sâu sắc ở cấp độ cao về bài toán trước mắt có thể làm việc lập trình đơn giản hơn nhiều. Trong trường hợp này, nhận định đó là việc một đối tượng Time thật ra là số có 3 chữ số trong hệ cơ số 60 (xem wikipedia.org/wiki/Sexagesimal).! Thuộc tính `second` là "cột 1", thuộc tính `minute` là "cột 60", còn thuộc tính `hour` là "cột 3600".

Khi viết `add_time` và `increment`, ta đã thực hiện rất hiệu quả phép cộng trên hệ 60, đó là lí do tại sao ta đã phải cộng có nhớ khi chuyển sang cột tiếp theo.

Kết quả quan sát này gợi ý cho ta một cách tiếp cận khác đến tổng thể bài toán—ta có thể chuyển đổi tượng Time ra các số nguyên và lợi dụng khả năng làm tính cộng của máy tính.

Sau đây là một hàm để chuyển Time ra số nguyên:

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

Và sau đây là hàm để chuyển số nguyên thành Time (nhớ lại rằng `divmod` chia đối số thứ nhất cho đối số thứ hai và trả về một bộ gồm có thương và số dư).

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

Có thể bạn phải suy nghĩ thêm một chút, chạy thử một vài lần, để tin rằng các hàm đó đều đúng. Một cách chạy thử là kiểm tra rằng `time_to_int(int_to_time(x)) == x` cho nhiều giá trị của `x`. Đây là ví dụ về một phép kiểm tra sự nhất quán.

Một khi đã tin rằng các hàm đều đúng, bạn có thể dùng chúng để viết lại `add_time`:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Bản này ngắn hơn so với bản đầu tiên, và dễ kiểm tra tính đúng đắn hơn.

Hãy viết lại `increment` dùng `time_to_int` và `int_to_time`.

Bằng cách nào đó, các việc đổi từ hệ cơ số 60 sang hệ cơ số 10 và ngược lại khó hơn so với chỉ thao tác với thời gian. Chuyển đổi giữa các hệ cơ số thì trừu tượng hơn; trực giác của chúng ta hợp với việc tính toán các giá trị thời gian hơn.

Nhưng nếu chúng ta hiểu sâu rằng có thể coi thời gian như số trong hệ 60 và dành thời gian để viết các hàm chuyển đổi (`time_to_int` và `int_to_time`), ta sẽ thu được chương trình ngắn hơn, dễ đọc và sửa lỗi, và đáng tin cậy hơn.

Ngoài ra, sau này ta cũng dễ thêm vào các tính năng khác. Chẳng hạn, hãy hình dung việc trừ hai Time để tìm ra khoảng thời gian giữa chúng. Cách làm tự nhiên là thực hiện phép trừ có nhớ. Nhưng việc dùng các hàm chuyển đổi sẽ dễ hơn và có nhiều khả năng sẽ chính xác hơn.

Điều nghịch lý là, đôi khi việc làm bài toán trở nên khó hơn (hoặc tổng quát hơn) hóa ra lại làm nó dễ hơn (vì còn ít trường hợp đặc biệt và ít khả năng gây lỗi).

Gỡ lỗi

Một đối tượng Time sẽ hợp lệ nếu các giá trị của `minutes` và `seconds` đề nằm giữa 0 và 60 (có thể bằng 0 nhưng không bằng 60) và nếu `hours` là số dương. `hours` và `minutes` phải là các số nguyên, nhưng `seconds` được phép có phần thập phân.

Những yêu cầu kiểu như vậy được gọi là các **bất biến** vì chúng cần phải luôn đúng. Nói cách khác, nếu chúng không đúng thì chương trình sẽ có lỗi ở đâu đó.

Việc viết mã lệnh để kiểm tra các bất biến có thể giúp bạn phát hiện các lỗi và tìm ra nguyên nhân gây lỗi. Chẳng hạn, giả sử bạn có một hàm như `valid_time` nhận vào một đối tượng Time và trả lại `False` nếu một bất biến bị vi phạm:

```
def valid_time(time):
    if time.hours < 0 or time.minutes < 0 or time.seconds < 0:
        return False
    if time.minutes >= 60 or time.seconds >= 60:
        return False
    return True
```

Tiếp theo, ở đoạn đầu mỗi hàm, bạn có thể kiểm tra các đối số để chắc rằng chúng đều hợp lệ:

```
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)
```

hoặc có thể dùng lệnh `assert`, vốn để kiểm tra một bất biến và báo biệt lệ nếu có sự vi phạm:

```
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)
```

Lệnh `assert` có ích vì chúng phân biệt rạch ròi giữa đoạn mã giải quyết trường hợp bình thường và mã lệnh để kiểm tra lỗi.

Thuật ngữ

nguyên mẫu và sửa chữa:
: Kế hoạch phát triển trong đó bao gồm việc viết một bản nháp của chương trình, chạy thử, và sửa những lỗi phát sinh.

phát triển theo kế hoạch:
: Kế hoạch phát triển trong đó bao gồm nhận định sâu sắc về bài toán, đồng thời kế hoạch được xác lập kĩ hơn so với các cách phát triển tăng dần và phát triển nguyên mẫu.

hàm thuần túy:
: Hàm không thay đổi bất kì đối tượng nào được nhận làm đối số. Hầu hết các hàm thuần túy đều cho kết quả.

hàm thay đổi:
: Hàm có làm thay đổi (các) đối tượng được nhận làm đối số. Hầu hết các hàm thay đổi đều không cho kết quả.

phong cách lập trình hàm:
: Phong cách thiết kế chương trình trong đó đa số các hàm đều là thuần túy.

bất biến:
: Điều kiện cần luôn được đảm bảo đúng trong quá trình chạy chương trình.

Bài tập

Hãy viết một hàm có tên `mul_time` nhận vào một đối tượng Time và một số, rồi trả về một đối tượng Time mới có chứa tích của Time ban đầu với số đó.

Tiếp theo, dùng `mul_time` để viết một hàm nhận vào đối tượng Time biểu thị thời gian về đích trong một cuộc đua, và một số biểu thị khoảng cách, rồi trả về một đối tượng Time biểu thị thời gian trung bình để đi hết một dặm đường.

Hãy viết một định nghĩa lớp cho đối tượng Date gồm các thuộc tính `day`, `month` và

year. Viết một hàm increment_date nhận vào một đối tượng Date, date và một số nguyên, n, rồi trả về một đối tượng Date mới biểu thị ngày tháng xuất hiện vào n ngày sau date. Gợi ý: Dùng khớp xương bàn tay để nhớ số ngày trong tháng. Đố bạn dùng hàm này để tính với năm nhuận? Xem wikipedia.org/wiki/Leap_year.

Module datetime có các đối tượng date và time; chúng cũng gần giống các đối tượng Date và Time trong chương này, nhưng có tập hợp các phương thức và toán tử rất phong phú. Hãy tìm hiểu tài liệu ở docs.python.org/lib/datetime-date.html.

1. Hãy dùng module datetime để viết một chương trình nhận vào ngày hôm nay và in ra thứ của ngày trong tuần.
2. Hãy viết một chương trình nhận vào ngày sinh của bạn rồi in ra tuổi bạn cùng số ngày, giờ, phút, giây cho đến sinh nhật tiếp theo.

---

1. thuật ngữ gốc có nghĩa là "chắp vá". ↩

Chương 17: Những đặc điểm của lập trình hướng đối tượng

Trở về Mục lục cuốn sách

Python là một **ngôn ngữ lập trình hướng đối tượng**, theo nghĩa là nó có những đặc điểm hỗ trợ lập trình hướng đối tượng.

Thật không dễ định nghĩa lập trình hướng đối tượng là gì, nhưng ta đã thấy một số đặc điểm của nó:

- Các chương trình được cấu thành từ các định nghĩa đối tượng và định nghĩa hàm, và phần lớn việc tính toán đều được diễn đạt trên những thao tác với đối tượng.
- Mỗi định nghĩa đối tượng đều tương ứng với một đối tượng hoặc khái niệm nào đó trong thực tại, và các hàm thao tác trên đối tượng đó tương ứng với cách mà những đối tượng thực tại này tương tác với nhau.

Chẳng hạn, lớp `Time` định nghĩa trong Chương 16 tương ứng với cách mà con người ghi chép thời gian trong ngày, và các hàm ta đã định nghĩa tương ứng với những thao tác mà con người thường tính với thời gian. Tương tự, các lớp `Point` và `Rectangle` tương ứng với những khái niệm toán học là điểm và hình chữ nhật.

Cho đến giờ, ta vẫn chưa lợi dụng những đặc tính sẵn có của Python để phục vụ cho lập trình hướng đối tượng. Những đặc tính này đều không nhất thiết phải có, chúng hầu hết chỉ là dạng cú pháp thay thế cho những câu lệnh ta đã viết. Tuy nhiên trong nhiều trường hợp, dạng thay thế này gọn hơn và chuyển tại ý tưởng về cấu trúc chương trình một cách chính xác hơn.

Chẳng hạn, trong chương trình `Time`, không có mối liên hệ rõ nét nào giữa lời định nghĩa lớp và các định nghĩa hàm theo sau nó. Song khi xem xét kĩ hơn, ta thấy được mỗi hàm đều nhận ít nhất là một đối tượng `Time` làm đối số.

Nhận định trên là cơ sở cho việc hình thành các **phương thức**; một phương thức là một hàm được gắn liền với một lớp cụ thể. Ta đã gặp những phương thức dành cho chuỗi, danh sách, từ điển, và bộ. Trong chương này, ta sẽ định nghĩa các phương thức cho kiểu dữ liệu do người dùng định nghĩa.

Các phương thức về mặt ngữ nghĩa thì cũng giống như hàm, nhưng về cú pháp thì có hai điểm khác biệt:

- Các phương thức được định nghĩa trong lời định nghĩa lớp nhằm biểu thị rõ mối quan hệ giữa lớp và phương thức đó.
- Cú pháp để gọi một phương thức khác với cú pháp để gọi hàm.

Trong một vài mục tiếp theo, ta sẽ lấy các hàm từ hai chương trước và chuyển đổi chúng thành các phương thức. Việc chuyển đổi này chỉ là máy móc; bạn có thể thực hiện được theo một số bước. Khi đã thạo việc chuyển đổi giữa các dạng, bạn có thể chọn được dạng phù hợp nhất với công việc đang làm.

Các đối tượng thực hiện việc in

Ở Chương 16, ta đã định nghĩa một lớp tên là `Time` và trong Bài tập {printtime}, bạn đã viết một hàm có tên `print_time`:

```
class Time(object):
    """represents the time of day.
       attributes: hour, minute, second"""
```

```
def print_time(time):
    print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

Để gọi hàm này, bạn cần phải truyền một đối tượng `Time` như là đối số:

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

Để biến `print_time` thành một phương thức, toàn bộ những gì ta phải làm chỉ là chuyển lời định nghĩa hàm vào trong định nghĩa lớp. Chú ý rằng khoảng thụt đầu dòng được thay đổi.

```
class Time(object):
    def print_time(time):
        print '%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second)
```

Bây giờ có hai cách gọi `print_time`. Cách thứ nhất (ít thông dụng) là dùng cú pháp của hàm:

```
>>> Time.print_time(start)
09:45:00
```

Trong cách dùng kí hiệu dấu chấm này, `Time` là tên của lớp, còn `print_time` là tên của phương thức. `start` được truyền vào làm tham biến.

Cách thứ hai (gọn gàng hơn) là dùng cú pháp của phương thức:

```
>>> start.print_time()
09:45:00
```

Trong cách dùng kí hiệu dấu chấm này, `print_time` (một lần nữa) là tên của phương thức, còn `start` là tên của đối tượng mà phương phức đó được gọi, còn có tên là **chủ thể**. Vai trò của chủ thể trong một lời gọi phương thức cũng như vai trò của chủ ngữ trong một câu văn.

Bên trong phương thức, chủ thể được gán là tham biến thứ nhất. VÌ vậy trong trường hợp này, `start` được gán cho `time`.

Theo quy ước, tham biến thứ nhất của một phương thức được gọi là `self`, vì vậy cách viết `print_time` như sau là thông dụng hơn:

```
class Time(object):
    def print_time(self):
        print '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

Lí do có quy ước này là từ phép ẩn dụ:

- Cú pháp của một lời gọi hàm, `print_time(start)`, hàm ý rằng hàm đó là chủ thể. Nó kiểu như lời yêu cầu "Này `print_time`! Đây là một đối tượng mà cậu cần in ra".
- Trong lập trình hướng đối tượng, các đối tượng là chủ thể. Một lời gọi phương thức như `start.print_time()` giống câu nói "Này `start`! Hãy tự in bản thân cậu đi".

Sự thay đổi về góc nhìn này có vẻ khiêm tốn, nhưng thật không dễ thấy lợi ích mà nó mang lại. Ở những ví dụ ta đã thảo luận, có thể là chưa. Nhưng đôi khi việc chuyển trách nhiệm từ hàm sang đối tượng có thể giúp ta dễ dàng viết được các hàm linh động hơn, cũng dễ bảo trì và sử dụng lại mã lệnh hơn.

Hãy viết lại `time_to_int` (ở Mục {hình mẫu}) dưới dạng phương thức. Có thể sẽ không phù hợp khi viết lại `int_to_time` như vậy; thật không rõ ràng là bạn muốn gọi phương thức đó từ đối tượng nào!

Một ví dụ khác

Sau đây là một dạng của `increment` (trong Mục {phát triển tăng dần}) được viết lại như một phương thức:

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

Dạng này có giả sử rằng `time_to_int` được viết như một phương thức, như trong Bài tập {chuyển đổi}. Đồng thời, cần thấy rằng đó là một hàm thuần túy, không phải một hàm sửa đổi.

Sau đây là cách gọi `increment`:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```

Chủ thể, `start`, được gán cho tham biến thứ nhất, `self`. Đối số, `1337`, được gán cho tham biến thứ hai, `seconds`.

Cơ chế làm việc này có thể gây nhầm lẫn, đặc biệt khi bạn mắc phải lỗi. Chẳng hạn, nếu gọi `increment` với hai đối số, bạn sẽ nhận được:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes exactly 2 arguments (3 given)
```

Thông báo lỗi thoạt đầu nhìn rất khó hiểu, vì chỉ có hai đối số trong ngoặc kép. Nhưng chủ thể cũng được coi là một đối số, nên tổng cộng có ba đối số.

Một ví dụ phức tạp hơn

`is_after` (trong Bài tập {is_after}) khó hơn một chút vì nó nhận vào hai đối tượng Time làm tham biến. Ở trường hợp này theo quy ước thì tham biến thứ nhất được gọi là `self` và tham biến thứ hai là `other`:

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Để dùng phương thức này, bạn phải gọi nó từ một đối tượng và truyền đối tượng kia như một đối số:

```
>>> end.is_after(start)
True
```

Một điều hay ở dạng cú pháp này là cách đọc nó giống với tiếng Anh: “end xảy ra sau start?”

Phương thức init

Phương thức init (gọi tắt của "initialization") là một phương thức đặc biệt được gọi khi một đối tượng được khởi tạo (cá thể hóa). Tên đầy đủ của nó là `__init__` (hai dấu gạch thấp, tiếp theo là `init`, rồi hai dấu gạch thấp khác). Một phương thức init cho lớp `Time` có thể trông như sau:

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

Thường thì các tham biến của `__init__` cũng có tên giống như các thuộc tính. Câu lệnh

```
        self.hour = hour
```

lưu lại giá trị của tham biến `hour` như một thuộc tính của `self`.

Các tham biến đều là tùy chọn, vì vậy nếu gọi `Time` mà không có đối số nào, bạn sẽ thu được các giá trị mặc định.

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

Nếu bạn cấp cho một đối số, nó sẽ thay cho giá trị mặc định của `hour`:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

Nếu bạn cấp hai đối số, chúng sẽ thay cho các giá trị mặc định của `hour` và `minute`.

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

Và nếu bạn cấp ba đối số, chúng sẽ thay thế tất cả ba giá trị mặc định.

> Hãy viết một phương thức init cho lớp `Point` nhận vào các tham biến tùy chọn `x` và `y` rồi gán chúng cho các thuộc tính tương ứng.

Phương thức __str__

`__str__` là một phương thức đặc biệt, cũng như `__init__`. Nó được dùng để trả lại một chuỗi biểu diễn cho một đối tượng.

Chẳng hạn, sau đây là một phương thức `str` cho đối tượng Time:

```
# ben trong lop Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

Khi bạn `print` một đối tượng, Python sẽ gọi phương thức `str`:

```
>>> time = Time(9, 45)
```

```
>>> print time
09:45:00
```

Khi viết một lớp mới, tôi luôn bắt đầu bằng việc viết `__init__`, để làm cho việc cá thể hóa đối tượng dễ dàng hơn, và `__str__`, để cho việc gỡ lỗi được dễ dàng hơn.

> Hãy viết một phương thức `str` cho lớp `Point`. Sau đó tạo một đối tượng Point và in nó ra.

Toán tử đa năng

Bằng việc định nghĩa các phương thức đặc biệt, bạn có thể chỉ định tính năng của các toán tử hoạt động trên kiểu dữ liệu do người dùng định nghĩa. Chẳng hạn, nếu bạn định nghĩa một phương thức có tên `__add__` cho lớp `Time`, bạn có thể dùng toán tử `+` với các đối tượng Time.

Sau đây là một ví dụ về cách định nghĩa nêu trên:

```
# ben trong lop Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

Và sau đây là cách sử dụng nó:

```
>>> start = Time(9, 45)

>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
```

Khi áp dụng toán tử `+` với các đối tượng Time, Python sẽ gọi `__add__`. Khi bạn in ra kết quả, Python gọi `__str__`. Như vậy là có khá nhiều điều đang diễn ra nơi "hậu trường"!

Việc thay đổi tính năng của một toán tử sao cho nó có thể làm việc được với các kiểu do người dùng định nghĩa được gọi là làm cho **toán tử** trở nên **đa năng**. Mỗi toán tử trong Python có một phương thức đặc biệt tương ứng, như `__add__`. Để biết thêm thông tin, hãy xem docs.python.org/ref/specialnames.html.

> Hãy viết một phương thức `add` cho lớp Point.

Cắt cử dựa theo kiểu dữ liệu

Ở mục trước, ta đã cộng hai đối tượng Time, nhưng bạn có lẽ cũng muốn cộng một số nguyên với đối tượng Time. Sau đây là một dạng của `__add__` để kiểm tra kiểu của `other` và gọi `add_time` hoặc `increment`:

```
# ben trong lop 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)
```

Hàm có sẵn `isinstance` nhận một giá trị và một đối tượng lớp rồi trả về `True` nếu giá trị là một cá thể thuộc lớp đó.

Nếu `other` là một đối tượng Time thì `__add__` sẽ gọi `add_time`. Trái lại, nó sẽ giả sử rằng tham biến là một số và gọi `increment`. Thao tác này được gọi là **cắt cử dựa theo kiểu dữ liệu** vì nó cắt cử các phương thức khác nhau để thực hiện phép tính, tuỳ theo kiểu của đối số.

Sau đây là các ví dụ trong đó dùng toán tử + với các kiểu dữ liệu khác nhau:

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print start + duration
11:20:00
>>> print start + 1337
10:07:17
```

Thật không may là kiểu thực hiện phép cộng này không có tính giao hoán. Nếu số nguyên đứng bên trái toán tử, bạn sẽ nhận được

```
>>> print 1337 + start
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

Vấn đề là ở chỗ, thay vì yêu cầu đối tượng Time cộng với một số nguyên, Python lại yêu cầu số nguyên cộng với đối tượng Time, và nó không biết thực hiện điều đó. Nhưng có một giải pháp khéo léo: phương thức đặc biệt `__radd__`, viết tắt của "right-side add". Phương thức này được gọi mỗi khi đối tượng Time xuất hiện bên phải của toán tử +. Sau đây là lời định nghĩa:

```
# ben trong lop Time:

    def __radd__(self, other):
        return self.__add__(other)
```

Và sau đây là cách dùng nó:

```
>>> print 1337 + start
10:07:17
```

Hãy viết một phương thức `add` cho các Point sao cho nó có thể làm việc với từng đối tượng Point hoặc một bộ:

- Nếu toán hạng thứ hai là một Point, phương thức cần trả lại một Point mới với toạ độ *x* bằng tổng các toạ độ *x* của các toán hạng, và tương tự như vậy với các toạ độ *y*.
- Nếu toán hạng thứ hai là một bộ, phương thức cần cộng phần tử thứ nhất của bộ vào toạ độ *x* và phần tử thứ hai vào toạ độ *y*, rồi trả lại kết quả là một Point mới.

Đa hình

Việc cắt cử theo kiểu dữ liệu có thể hữu dụng khi cần thiết, nhưng (thật may là) nó không phải luôn cần đến. Thường bạn có thể tránh được cách đó bằng cách viết các hàm làm việc được với các đối số có kiểu khác nhau.

Nhiều hàm ta viết cho chuỗi thực ra cũng phát huy tác dụng với bất kì kiểu dãy nào. Chẳng hạn, trong Mục {bảng tần số} ta đã dùng `histogram` để đếm số lần mỗi chữ cái xuất hiện trong một từ.

```
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
```

Hàm này cũng dùng được với các danh sách, bộ, và thậm chí cả từ điển, miễn là các phần tử của `s` phải băm được, để chúng có thể dùng như các khoá trong `d`.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Các hàm có khả năng làm việc được với vài kiểu dữ liệu được gọi là có tính **đa hình**. Tính đa hình giúp cho việc tái sử dụng mã lệnh. Chẳng hạn, hàm có sẵn `sum`, để cộng các phần tử trong một dãy, luôn có tác dụng chỉ cần các phần tử trong dãy cộng được với nhau.

Vì các đối tượng Time có phương thức `add`, chúng sẽ dùng được với `sum`:

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print total
23:01:00
```

Nói chung, nếu tất cả các thao tác bên trong một hàm hoạt động được với một kiểu dữ liệu cho trước, thì hàm đó cũng hoạt động được với kiểu nói trên.

Tính đa hình sẽ tốt nhất là lúc không định trước, khi bạn phát hiện thấy một hàm mình đã viết có thể được áp dụng cho một kiểu dữ liệu mà bạn chưa từng có ý định xử lý đến.

Gỡ lỗi

Sẽ hoàn toàn hợp lệ khi thêm các thuộc tính vào đối tượng bất kì chỗ nào trong quá trình thực hiện chương trình; nhưng nếu bạn gắt gao về lý thuyết kiểu dữ liệu thì để cho các đối tượng cùng kiểu có những thuộc tính khác nhau sẽ là cách làm không đáng tin cậy. Tốt hơn là khởi tạo tất cả thuộc tính của đối tượng trong phương thức init.

Nếu không chắc rằng một đối tượng có chứa một thuộc tính cụ thể hay không, bạn có thể dùng hàm có sẵn `hasattr` (xem Mục {hasattr}).

Một cách khác để truy cập các thuộc tính của đối tượng là thông qua thuộc tính đặc biệt `__dict__`, vốn là từ điển có ánh xạ các tên thuộc tính (kiểu chuỗi) đến giá trị:

```
>>> p = Point(3, 4)
>>> print p.__dict__
{'y': 4, 'x': 3}
```

Bạn có thể ghi nhớ hàm sau để tiện cho việc gỡ lỗi:

```
def print_attributes(obj):
    for attr in obj.__dict__:
        print attr, getattr(obj, attr)
```

`print_attributes` duyệt các mục trong từ điển của đối tượng và in ra mỗi tên thuộc tính kèm theo giá trị tương ứng.

Hàm có sẵn `getattr` nhận vào một đối tượng cùng tên của một thuộc tính (kiểu chuỗi) rồi trả lại giá trị của thuộc tính đó.

Thuật ngữ

ngôn ngữ hướng đối tượng:
: Ngôn ngữ cung cấp những đặc điểm, như lớp do người dùng định nghĩa và cú pháp dành cho phương thức, nhằm giúp cho việc lập trình hướng đối tượng.

lập trình hướng đối tượng:
: Phong cách lập trình trong đó dữ liệu và các thao tác trên dữ liệu được tổ chức thành các lớp và phương thức.

phương thức:
: Hàm được định nghĩa bên trong lời định nghĩa lớp và được gọi với các cá thể của lớp đó.

chủ thể:
: Đối tượng mà phương thức được kích hoạt trên đó.

toán tử đa năng:
: Một toán tử trở thành đa năng như `+` khi nó thay đổi tính năng để làm việc với kiểu dữ liệu do người dùng định nghĩa.

cắt cử dựa theo kiểu:
: Dạng mẫu lập trình trong đó kiểm tra kiểu của toán hạng và gọi các hàm khác nhau tùy theo kiểu dữ liệu của toán hạng đó.

đa hình:
: Tính chất của một hàm có thể làm việc với vài kiểu dữ liệu khác nhau.

Bài tập

Bài tập này cũng là câu chuyện cảnh tỉnh về một lỗi phổ biến nhưng khó phát hiện của Python.

Hãy viết định nghĩa cho một lớp có tên `Kangaroo` gồm những phương thức sau:

1. Một phương thức `__init__` để khởi tạo một thuộc tính có tên là

pouch_contents (“túi” của Kangaroo) bằng một danh sách rỗng.
2. Một phương thức có tên put_in_pouch nhận một đối tượng có kiểu tùy ý và bổ sung nó vào pouch_contents.
3. Một phương thức __str__ để trả về chuỗi biểu diễn cho đối tượng Kangaroo và nội dung của pouch_contents.

Hãy kiểm tra mã lệnh vừa viết bằng cách tạo ra hai đối tượng Kangaroo, gán chúng cho các biến có tên kanga và roo, sau đó thêm roo vào trong “túi” của kanga.

Hãy tải về file thinkpython.com/code/BadKangaroo.py. Nó chứa lời giải của bài tập nhưng có một lỗi rất nghiêm trọng. Hãy tìm và sửa lỗi này.

Nếu bị vướng mắc, bạn có thể tải về thinkpython.com/code/GoodKangaroo.py, trong đó giải thích bài toán và chỉ ra một cách giải.

Visual là một module Python cho phép thao tác với đồ thị ba chiều. Nó thường không đi kèm theo bản cài đặt Python, vì vậy bạn có thể sẽ phải cài đặt nó từ kho phần mềm của hệ điều hành, hoặc nếu không có, thì ở vpython.org.

Ví dụ sau đây tạo ra một không gian 3 chiều có bề rộng, chiều dài, chiều cao đều bằng 256 đơn vị, sau đặt điểm gốc (“center”) tại vị trí (128, 128, 128). Tiếp theo một hình cầu màu xanh lam được vẽ ra.

```
from visual import *

scene.range = (256, 256, 256)
scene.center = (128, 128, 128)

color = (0.1, 0.1, 0.9)          # mau gan giong xanh lam
sphere(pos=scene.center, radius=128, color=color)
```

color là một bộ màu RGB; trong đó các thành tố màu Đỏ-Lục-Lam có mức từ 0.0 đến 1.0 (xem wikipedia.org/wiki/RGB_color_model).

Nếu chạy đoạn mã này, bạn sẽ thấy một cửa sổ có nền màu đen và một hình cầu màu xanh lam. Nếu lăn nút chuột giữa lên hoặc xuống, bạn sẽ phóng to hoặc thu nhỏ hình. Bạn còn có thể xoay hình bằng cách di nút chuột phải, nhưng chỉ với một quả cầu trên hình thì khó quan sát được sự khác biệt nào.

Vòng lặp tiếp theo tạo ra một khối lập phương gồm các quả cầu:

```
t = range(0, 256, 51)
for x in t:
    for y in t:
        for z in t:
            pos = x, y, z
            sphere(pos=pos, radius=10, color=color)
```

1. Hãy đưa đoạn mã này vào trong một file lệnh và đảm bảo rằng nó chạy đúng.

2. Hãy sửa lại chương trình sao cho mỗi hình cầu trong khối lập phương có màu tương ứng với vị trí của nó trong không gian màu RGB. Chú ý rằng các tọa độ nằm trong khoảng 0–255, còn bộ RGB thì lại nằm trong khoảng 0.0–1.0.
3. Tải về `thinkpython.com/code/color_list.py` và dùng hàm `read_colors` để phát sinh một danh sách các màu hiện có trên hệ thống máy của bạn, tên của chúng và các giá trị RGB. Với mỗi màu có tên trên, hãy vẽ một quả cầu ở vị trí tương ứng với giá trị RGB của nó.

Bạn có thể tham khảo lời giải của tôi tại
`thinkpython.com/code/color_space.py`.

Chương 18: Thừa kế

Trở về Mục lục cuốn sách

Trong chương này ta sẽ xây dựng các lớp để biểu diễn cho quân bài tây, bộ bài, và phần bài của người chơi trong trò *poker*. Nếu không biết chơi *poker*, bạn có thể xem ở wikipedia.org/wiki/Poker, dù điều này không cần thiết vì tôi sẽ nói những điều cần để làm bài tập.

Thông tin về bộ bài tây thông dụng có ở wikipedia.org/wiki/Playing_cards.

Đối tượng lá bài (Card)

Có 52 lá bài trong một bộ, mỗi lá bài thuộc về một trong bốn chất và một trong 13 bậc. Các chất gồm Pích, Cơ, Rô, và Nhép (theo thứ tự giảm dần trong trò *bridge*). Các bậc gồm có A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, và K. Tùy theo trò chơi mà bạn quân A có thể cao hơn K hoặc thấp hơn 2.

Nếu bạn muốn định nghĩa một đối tượng mới để biểu diễn cho lá bài, rõ ràng các thuộc tính phải là: `rank` (bậc) và `suit` (chất). Còn việc chọn kiểu dữ liệu cho các thuộc tính lại không hiển nhiên. Một khả năng là dùng các chuỗi gồm những từ như `'Spade'` (Pích) cho chất và `'Queen'` cho bậc. Một vấn đề đặt ra với cách làm này là sẽ không dễ so sánh xem lá bài nào có bậc hoặc chất cao hơn.

Một cách khác là dùng số nguyên để **đánh số** cho các bậc và chất. Ở đây, "đánh số" có nghĩa là lập một phép ánh xạ từ số đến chất, cũng như từ số đến bậc. Kiểu đánh số này khác với mã hóa (với hàm ý bí mật).

Chẳng hạn, bảng dưới đây cho thấy các chất và mã số tương ứng:

Spades ↦ 3
Hearts ↦ 2
Diamonds ↦ 1
Clubs ↦ 0

Mã số này giúp so sánh các lá bài dễ hơn; vì chất cao hơn được ánh xạ đến số lớn hơn, và ta có thể so sánh chất bằng cách so các mã số của chúng.

Ánh xạ đối với bậc thì khá dễ thấy; mỗi bậc số thì ánh xạ đến chính số nguyên tương ứng, còn với các bậc chữ:

Jack ↦ 11
Queen ↦ 12
King ↦ 13

Ở đây tôi dùng kí hiệu ↦ để làm rõ rằng phép ánh xạ này không phải là một phần của chương trình Python. Đó là một phần của khâu thiết kế chương trình, nhưng không xuất hiện một cách cụ thể trên mã lệnh.

Lời định nghĩa lớp cho `Card` sẽ như sau:

```
class Card(object):
    """bieu thi cho mot la bai tay."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

Như thường lệ, phương thức init nhận vào một tham biến tùy chọn cho mỗi thuộc tính. Lá bài mặc định là 2 Nhép.

Để tạo ra một Card, bạn gọi `Card` cùng với chất và bậc của lá bài mà bạn muốn.

```
queen_of_diamonds = Card(1, 12)
```

Để in ra đối tượng Card theo cách mà mọi người dễ đọc, ta cần ánh xạ từ mã số đến các bậc và chất tương ứng. Một cách làm tự nhiên là dùng danh sách các chuỗi. Ta gán những danh sách này cho các **thuộc tính lớp**:

```
# ben trong lop 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])
```

Các biến như `suit_names` và `rank_names`, vốn được định nghĩa bên trong một lớp nhưng bên ngoài bất kì một phương thức nào, được gọi là thuộc tính lớp và chúng gắn liền với đối tượng lớp `Card`.

Chúng khác với những biến như `suit` và `rank`, vốn được gọi là **thuộc tính cá thể** vì gắn với một cá thể nhất định.

Cả hai loại thuộc tính đều được truy cập bằng kí hiệu đấu chấm. Chẳng hạn, trong `__str__`, `self` là một đối tượng Card, còn `self.rank` là bậc của nó. Tương tự, `Card` là một đối tượng lớp, còn `Card.rank_names` là một danh sách các chuỗi gắn với lớp này.

Mỗi lá bài đều có `suit` và `rank` riêng của nó, nhưng chung quy chỉ có `suit_names` và một `rank_names`.

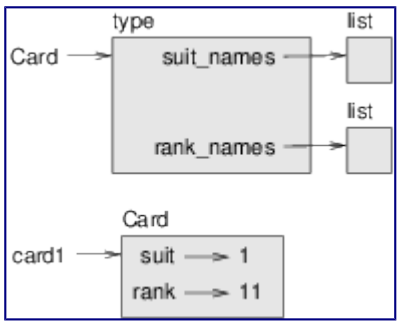
Tóm lại, biểu thức `Card.rank_names[self.rank]` có nghĩa là "dùng thuộc tính `rank` của đối tượng `self` làm chỉ số cho danh sách `rank_names` thuộc lớp `Card`, và chọn lấy chuỗi tương ứng".

Phần tử đầu tiên của `rank_names` là `None` vì không có lá bài nào với bậc bằng 0. Với việc thêm `None` vào để giữ chỗ, ta thu được một phép ánh xạ rất hay trong đó chỉ số 2 ánh xạ trực tiếp đến chuỗi `'2'`, và cứ như vậy. Để tránh cách sửa chữa này, ta cũng có thể dùng một từ điển thay vì danh sách.

Với những phương thức được xét đến giờ, ta có thể tạo và in ra những lá bài:

```
>>> card1 = Card(2, 11)
>>> print card1
Jack of Hearts
```

Sau đây là một sơ đồ cho thấy đối tượng lớp `Card` và một cá thể Card:



`Card` là một đối tượng lớp, vì vậy nó có kiểu là `type`. `card1` có kiểu là `Card`. (Để tiết kiệm chỗ, tôi đã không vẽ nội dung của `suit_names` và `rank_names`).

So sánh các lá bài

Với những kiểu dữ liệu có sẵn, các toán tử quan hệ (<, >, ==, v.v.) giúp so sánh và xác định xem một giá trị lớn hơn, nhỏ hơn hoặc bằng giá trị kia. Với các kiểu dữ liệu do người dùng định nghĩa, ta có thể thay thế tính năng mặc định của những toán tử có sẵn bằng cách cung cấp một phương thức có tên `__cmp__`.

`__cmp__` nhận vào hai tham số, `self` và `other`, rồi trả về một số dương nếu như đối tượng thứ nhất lớn hơn, một số âm nếu như đối tượng thứ hai lớn hơn, và 0 nếu chúng bằng nhau.

Thứ tự đúng đắn của các quân bài thật không rõ ràng. Chẳng hạn, lá bài nào tốt hơn, 3 Nhép hay 2 Rô? Một lá thì có bậc cao hơn, nhưng lá kia lại có chất cao hơn. Để so sánh các quân bài, bạn cần phải quyết định xem giữa bậc và chất, cái nào quan trọng hơn.

Câu trả lời có thể tùy vào trò chơi đang xét, nhưng để giữ cho mọi thứ đơn giản, ta sẽ tạm quy ước là chất quan trọng hơn, như vậy tất cả các quân bài chất Pích sẽ cao hơn các quân chất Rô, và cứ như vậy.

Khi đã xác định như vậy, ta có thể viết `__cmp__`:

```
# ben trong lop Card:

    def __cmp__(self, other):
        # kiem tra chat
        if self.suit > other.suit: return 1
        if self.suit < other.suit: return -1

        # chat giong nhau... kiem tra bac
        if self.rank > other.rank: return 1
        if self.rank < other.rank: return -1

        # bac giong nhau... hai quan bang nhau
        return 0
```

Bạn có thể viết gọn đoạn mã trên bằng cách so sánh các bộ:

```
# ben trong lop Card:

    def __cmp__(self, other):
        t1 = self.suit, self.rank
        t2 = other.suit, other.rank
        return cmp(t1, t2)
```

Hàm có sẵn `cmp` cũng có giao diện giống như của phương thức `__cmp__`: nó nhận hai giá trị và trả về một số dương nếu giá trị thứ nhất lớn hơn, một số âm nếu giá trị thứ hai lớn hơn, và 0 nếu chúng

bằng nhau.

> Hãy viết một phương thức `__cmp__` cho đối tượng Time. Gợi ý: bạn có thể dùng so sánh bộ, nhưng cũng nên cân nhắc dùng phép trừ số nguyên.

Bộ bài

Bây giờ, khi đã có các đối tượng Card, ở bước tiếp theo ta sẽ định nghĩa Deck (bộ bài). Vì một bộ bài được tạo thành từ những quân bài, nên cách làm tự nhiên sẽ là cho mỗi Deck chứa một danh sách các quân bài làm thuộc tính.

Sau đây là đoạn mã định nghĩa `Deck`. Phương thức init tạo ra thuộc tính `cards` và phát sinh một bộ bài tiêu chuẩn gồm 52 lá:

```
class Deck(object):

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

Cách dễ nhất để tạo ra bộ bài là dùng một vòng lặp lồng ghép. Vòng lặp ngoài đếm số chất từ 0 đến 3. Vòng lặp trong đếm số bậc từ 1 đến 13. Mỗi lần lặp sẽ tạo ra một Card mới theo chất và bậc hiện có, rồi bổ sung nó vào `self.cards`.

In bộ bài

Sau đây là phương thức `__str__` của `Deck`:

```
#ben trong lop Deck:

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

Phương thức này cho thấy một cách hiệu quả để tích lũy thành một chuỗi lớn: tạo ra một danh sách các chuỗi rồi dùng `join`. Hàm có sẵn `str` gọi phương thức `__str__` với mỗi lá bài và trả về chuỗi biểu diễn của nó.

Vì ta gọi `join` từ một kí tự xuống dòng, nên các quân bài được phân tách bởi theo dòng. Kết quả sẽ trông như sau:

```
>>> 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
```

Mặc dù kết quả hiện ra trên 52 dòng, nhưng thật ra nó là một chuỗi dài trong đó có các kí tự xuống dòng.

Thêm, bớt, trộn bài và sắp xếp

Để chia bài, ta cần có một phương thức nhằm lấy một lá từ bộ bài và trả lại nó. Phương thức `pop` của danh sách cho ta một cách làm tiện lợi:

```
#ben trong lop Deck:

    def pop_card(self):
        return self.cards.pop()
```

Vì `pop` lấy ra lá bài *cuối cùng* trong danh sách, nên ta chia từ phía dưới của bộ bài. Ngoài đời, người ta sẽ nhướn lông mày khi thấy cách chia bài như vậy[1], nhưng để phục vụ mục đích lập trình thì không sao.

Để thêm một lá bài, ta có thể dùng phương thức `append` của danh sách:

```
#ben trong lop Deck:

    def add_card(self, card):
        self.cards.append(card)
```

Một phương thức như thế này, vốn dùng một hàm khác mà không tính toán gì đáng kể, còn được gọi là **véc-ni**. Từ này là hình ảnh ẩn dụ xuất phát từ nghề mộc, ở đó người ta thường dùng keo dán một lớp gỗ tốt lên trên bề mặt đồ gỗ rẻ tiền hơn.

Trong trường hợp này ta định nghĩa một phương thức "mỏng" để biểu thị một toán tử thao tác với danh sách, vốn rất phù hợp với đối tượng bộ bài.

Lấy một ví dụ khác, ta có thể viết một phương thức của Deck tên là `shuffle` (trộn) dùng hàm `shuffle` từ module `random`:

```
# ben trong lop Deck:

    def shuffle(self):
        random.shuffle(self.cards)
```

Đừng quên việc nhập `random`.

> Hãy viết một phương thức của Deck có tên là `sort` trong đó dùng phương thức `sort` của danh sách để sắp xếp các lá bài trong `Deck`. `sort` sử dụng phương thức `__cmp__` mà ta đã định nghĩa để xác định thứ tụ sắp xếp.

Thừa kế

Đặc điểm của ngôn ngữ mà thường gắn liền nhất với lập trình hướng đối tượng là **thừa kế**. Thừa kế là khả năng định nghĩa một lớp mới dưới dạng một phiên bản được sửa đổi từ một lớp sẵn có.

Sở dĩ gọi là "thừa kế" vì lớp mới sẽ hưởng lại toàn bộ các phương thức từ lớp sẵn có. Bằng việc mở rộng phép ẩn dụ này, lớp sẵn có được gọi là **lớp cha mẹ** và lớp mới được gọi là **lớp con**.

Lấy ví dụ, giả sử ta muốn có một lớp biểu diễn phần bài thuộc về một người chơi. Phần bài này cũng giống bộ bài ở chỗ: chúng đều là tập hợp từ những lá bài, và chúng đều cần các thao tác như thêm và bớt các lá bài.

Một phần bài khác với một bộ bài ở chỗ có những thao tác chúng ta muốn áp dụng cho phần bài nhưng lại không dùng cho bộ bài. Chẳng hạn, trong trò chơi *poker* ta có thể so sánh hai phần bài xem ai thắng. Trong trò chơi *bridge* ta muốn tính điểm cho một phần bài để đặt cược.

Mối liên hệ này giữa các lớp—giống mà khác—dẫn đến sự hình thành tính thừa kế.

Việc định nghĩa của một lớp con cũng giống như các định nghĩa lớp khác, nhưng tên của lớp cha mẹ phải được đặt trong cặp ngoặc đơn:

```
class Hand(Deck):
    """bieu thi phan bai cua moi nguoi choi"""
```

Định nghĩa này chỉ ra rằng `Hand` kế thừa từ `Deck`; nghĩa là ta có thể dùng những phương thức như `pop_card` và `add_card` cho cả Hand lẫn Deck.

`Hand` cũng thừa kế `__init__` từ `Deck`, nhưng đây sẽ không phải là điều ta muốn: thay vì phát sinh toàn bộ 52 quân bài cho một phần, phương thức init của Hands cần khởi tạo `cards` là một danh sách rỗng.

Nếu ta viết một phương thức init bên trong lớp `Hand`, nó sẽ thay thế cho init của lớp `Deck`:

```
# ben trong lop Hand:

    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

Như vậy khi bạn tạo một Hand, Python sẽ gọi phương thức init này:

```
>>> hand = Hand('new hand')
>>> print hand.cards
[]
>>> print hand.label
new hand
```

Nhưng các phương thức khác vẫn được thừa kế từ `Deck`, vì vậy ta có thể dùng `pop_card` và `add_card` để chia bài:

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print hand
King of Spades
```

Một cách tự nhiên, bước tiếp theo sẽ là gói đoạn mã trên vào một phương thức có tên `move_cards`:

```
#ben trong lop Deck:

    def move_cards(self, hand, num):
        for i in range(num):
            hand.add_card(self.pop_card())
```

`move_cards` nhận hai đối số, một đối tượng Hand và số lá bài cần chia. Nó làm thay đổi cả `self` và `hand`, rồi trả về `None`.

Trong một số trò chơi, các lá bài được chuyển từ phần này sang phần kia, hoặc từ một phần trở về bộ bài. Bạn có thể dùng `move_cards` để thực hiện bất cứ thao tác nào kể trên: `self` có thể là Deck hoặc Hand, và `hand`, dù tên nó là vậy, cũng có thể là `Deck`.

Hãy viết một phương thức của Deck có tên là `deal_hands` nhận vào hai tham biến là số phần bài và số lá bài trong mỗi phần, rồi tạo ra các đối tượng Hand mới, chia đủ số lá bài cho mỗi phần, và trả về một danh sách các đối tượng Hand.

Thừa kế là một tính năng có ích. Có những chương trình sẽ trở nên tự lặp lại nếu không dùng thừa kế, và sẽ đẹp hơn khi dùng nó. Thừa kế có ghể giúp sử dụng lại mã lệnh, vì bạn có thể tự sửa tính chất có ở lớp cha mẹ mà không cần phải sửa lớp đó. Trong một số trường hợp, cấu trúc thừa kế phản ánh đúng cấu trúc tự nhiên của vấn đề, từ đó làm cho chương trình trở nên dễ hiểu hơn.

Mặt khác, thừa kế có thể làm cho chương trình khó đọc hơn. Khi một phương thức được kích hoạt, đôi khi không dễ tìm được đoạn định nghĩa của nó. Phần mã lệnh có liên quan đến có thể nằm rải rác trong một số module. Hơn nữa, nhiều việc làm bằng cách thừa kế cũng có thể làm tốt bằng, chưa kể là tốt hơn, khi không thừa kế.

Sơ đồ lớp

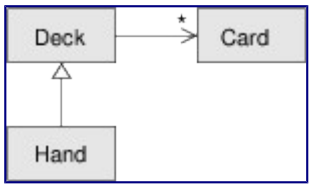
Cho đến giờ ta đã thấy sơ đồ ngăn xếp để chỉ ra trạng thái của một chương trình, và sơ đồ đối tượng để chỉ ra các thuộc tính của một đối tượng cùng những giá trị của chúng. Các sơ đồ kể trên đều biểu diễn một khoảnh khắc trong quá trình thực thi chương trình. Vì vậy khi chương trình chạy cũng thay đổi đi.

Những sơ đồ đó cũng rất chi tiết; và ở khía cạnh nào đó, quá chi tiết. Một sơ đồ lớp, trái lại, biểu diễn một cách trừu tượng cấu trúc chương trình. Thay vì cho thấy các đối tượng riêng biệt, nó cho thấy các lớp và mối liên hệ giữa các lớp đó.

Có một số mối liên hệ giữa các lớp như sau:

- Đối tượng trong một lớp có thể chứa những tham chiếu đến những đối tượng thuộc lớp khác. Chẳng hạn, mỗi Rectangle chứa một tham chiếu đến một Point, và mỗi Deck chứa nhiều tham chiếu đến nhiều Card. Dạng liên hệ này được gọi là **HAS-A** (“có một”), như câu nói, “một Rectangle có một Point”.
- Một lớp có thể thừa kế từ lớp khác. Mối liên hệ này được gọi là **IS-A** (“là một”), như cách nói, “một Hand là một kiểu Deck”.
- Một lớp có thể phụ thuộc vào lớp khác theo nghĩa sự thay đổi của lớp này sẽ yêu cầu những thay đổi ở lớp kia.

Một **sơ đồ lớp** là một cách biểu diễn bằng hình vẽ cho những mối liên hệ như vậy.[2] Chẳng hạn, sơ đồ này cho thấy mối liên hệ giữa `Card`, `Deck` và `Hand`.



Mũi tên với đầu hình tam giác trắng biểu thị một liên hệ IS-A; ở đây nó cho thấy Hand thừa kế từ Deck.

Mũi tên bình thường biểu thị một liên hệ HAS-A; ở đây Deck có những tham chiếu đến các đối tượng Cards.

Dấu sao (`*`) gần đầu mũi tên biểu thị một **phép nhân**; nó cho biết có bao nhiêu Card trong mỗi Deck. Một phép nhân có thể đi kèm với con số, như `52`, một khoảng số, như `5..7` hoặc chỉ một dấu sao đơn lẻ, ngụ ý rằng Deck có thể có số lá bài bất kì.

Một sơ đồ chi tiết hơn có thể chỉ ra rằng Deck thực chất bao gồm một *danh sách* các Card, nhưng

các kiểu dữ liệu có sẵn như danh sách và từ điển thường không được ghi trong sơ đồ lớp.

Hãy đọc `TurtleWorld.py`, `World.py` và `Gui.py` rồi vẽ một sơ đồ lớp để chỉ ra những mối liên hệ giữa các lớp được định nghĩa trong đó.

Gỡ lỗi

Thừa kế có thể làm cho việc gỡ lỗi khó khăn vì khi bạn gọi một phương thức từ một đối tượng, bạn có thể không biết rằng phương thức nào sẽ được kích hoạt.

Giả sử rằng bạn đang viết một hàm để thao tác với các đối tượng Hand. Bạn muốn nó dùng được với mọi loại Hand, như PokerHand, BridgeHand, (các phần bài trong những loại trò chơi khác nhau), v.v. Nếu gọi một phương thức như `shuffle`, bạn có thể sẽ được phương thức định nghĩa trong `Deck`, nhưng chỉ cần một lớp con có phương thức thay thế thì bạn sẽ gọi vào phương thức đó. Bất cứ lúc nào bạn không chắc chắn về luồng thực hiện chương trình, cách làm đơn giản nhất là thêm lệnh print vào đầu những phương thức có liên quan. Nếu `Deck.shuffle` in ra một thông báo chẳng hạn như `Running Deck.shuffle`, thì khi chương trình chạy nó sẽ theo dõi luôn luồng thực hiện.

Một cách làm khác là dùng hàm sau đây, vốn nhận vào một đối tượng và một tên phương thức (dạng chuỗi) và trả về một lớp trong đó có định nghĩa của phương thức:

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

Sau đây là một ví dụ:

```
>>> hand = Hand()
>>> print find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

Như vậy phương thức `shuffle` của Hand này chính là phương thức trong `Deck`.

`find_defining_class` dùng phương thức `mro` để thu được danh sách các đối tượng lớp (kiểu) sẽ được dùng để tìm kiếm phương thức. “MRO” là viết tắt của “method resolution order” (thứ tự phân định phương thức).

Sau đây là một gợi ý về việc thiết kế chương trình: mỗi khi lập một phương thức thay thế, giao diện của phương thức mới cần phải giống như của phương thức cũ. Nó cần phải nhận cùng các tham biến, trả về cùng kiểu, và tuân theo cùng loại điều kiện trước và điều kiện sau. Nếu tuân theo quy tắc này, bạn sẽ thấy hàm nào được lập ra cho cá thể của một lớp cha mẹ, như Deck, cũng sẽ hoạt động được với các cá thể của những lớp con như Hand hoặc PokerHand.

Nếu vi phạm quy tắc này, chương trình của bạn (rất tiếc) sẽ sụp đổ như một đống bài.

Thuật ngữ

đánh số:
: Biểu thị một tập hợp các giá trị bằng một tập hợp giá trị khác bằng cách thiết lập quy tắc ánh xạ giữa chúng.

thuộc tính lớp:
: Thuộc tính gắn với một đối tượng lớp. Thuộc tính lớp được định nghĩa bên trong lời định

nghĩa hàm nhưng ở ngoài các phương thức.

thuộc tính cá thể:
: Thuộc tính gắn với một cá thể của một lớp.

véc-ni:
: Phương thức hoặc hàm cung cấp giao diện khác tới một hàm khác mà không thực hiện tính toán gì đáng kể.

thừa kế:
: Khả năng định nghĩa một lớp mới là một dạng sửa đổi từ một lớp đã định nghĩa từ trước.

lớp cha mẹ:
: Lớp mà từ đó các lớp con kế thừa.

lớp con:
: Lớp mới được tạo ra bằng cách kế thừa từ một lớp sẵn có.

liên hệ IS-A:
: Mối liên hệ giữa một lớp con và lớp cha mẹ của nó.

liên hệ HAS-A:
: Mối liên hệ giữa hai lớp trong đó các cá thể của một lớp chứa tham chiếu đến các cá thể của lớp kia.

sơ đồ lớp:
: Sơ đồ thể hiện các lớp trong một chương trình và mối liên hệ giữa chúng.

phép nhân:
: Cách viết trong sơ đồ lớp dành cho mối liên hệ HAS-A, để chỉ ra có bao nhiêu tham chiếu đến các cá thể của lớp khác.

Bài tập

Sau đây là các khả năng xảy ra đối với một phần bài poker, xếp theo thứ tự tăng dần về giá trị (và giảm dần về xác suất):

đôi:

: hai lá bài có cùng bậc

hai đôi:

: hai đôi cùng bậc với nhau

ba:

: ba lá bài có cùng bậc

straight:

năm lá bài với bậc tăng dần (quân Át có thể tính là cao hoặc thấp, vì vậy cả `A-2-3-4-5` lẫn { 10-Jack-Queen-King-Ace} đều hợp lệ, nhưng `Queen-King-Ace-2-3` thì không.)

flush:

năm lá bài có cùng chất

full house:

ba lá bài có cùng bậc, hai lá còn lại có bậc khác

tứ quý:

bốn lá bài có cùng bậc

dây:

năm lá bafi có bậc tăng dần (đã giải thích ở trên) và có cùng chất

Mục đích của bài tập này là ước tính xác suất rút được những lá bài như vậy trong một phần bài.

1. Tải các file sau từ thinkpython.com/code:

   `Card.py`

   : Một bản đầy đủ của các lớp `Card`, `Deck` và `Hand` trong chương này.

   `PokerHand.py`

   : Một bản chưa đầy đủ gồm một lớp biểu diễn phần bài poker, mà mã lệnh để chạy thử nó.

2. Khi bạn chạy `PokerHand.py`, nó sẽ chia 7 phần bài, mỗi phần có 7 lá, và kiểm tra xem liệu có phần nào chứa một flush hay không. Hãy đọc mã lệnh cẩn thận trước khi tiếp tục.
3. Bổ sung vào `PokerHand.py` các phương thức có tên `has_pair`, `has_twopair`, v.v. để trả về True hoặc False tùy theo phần bài có đạt yêu cầu (có 1 đôi, 2 đôi) hay không. Mã lệnh bạn viết cần chạy được với những "phần bài" chứa số lá bài bất kì (mặc dù số lượng thông dụng nhất là 5 và 7).
4. Viết một phương thức có tên `classify` để phân loại bằng cách tìm ra trường hợp có giá trị cao nhất cho một phần bài và đặt thuộc tính `label` tương ứng cho nó. Chẳng hạn, một phần bài 7 lá có thể chứa cả một flush và một đôi; nó cần được xếp loại "flush".
5. Khi bạn đã chắc rằng các phương thức phân loại viết ra đều đúng, bước tiếp theo là ước tính xác suất của các trường hợp khác nhau. Hãy viết một hàm trong `PokerHand.py` để trộn bộ bài, chia thành các phần, phân loại từng phần, và

đếm số lần mà mỗi loại khác nhau đã xuất hiện.
6. In ra một bảng phân loại cùng các xác suất tương ứng. Chạy chương trình với số lần chia bài tăng dần đến khi kết quả hội tụ về một giá trị có độ chính xác chấp nhận được. So sánh kết quả của bạn với những giá trị liệt kê tại wikipedia.org/wiki/Hand_rankings.

Bài tập này sử dụng TurtleWorld từ Chương {turtlechap}. Bạn sẽ viết mã để điều khiển Turtle (con rùa) chơi *tag* (một trò chơi đuổi bắt). Nếu bạn chưa biết luật chơi, hãy xem wikipedia.org/wiki/Tag_(game).

1. Tải về file thinkpython.com/code/Wobbler.py và chạy nó. Bạn sẽ thấy một TurtleWorld với ba Turtle. Nếu bạn ấn nút {Run}, các Turtle sẽ chạy một cách ngẫu nhiên.
2. Hãy đọc mã lệnh và nắm vững cách hoạt động của nó. Lớp `Wobbler` thừa kế từ `Turtle`, có nghĩa là các phương thức của `Turtle` gồm có `lt`, `rt`, `fd` và `bk` đều có tác dụng với Wobblers.

   Phương thức `step` được gọi bởi TurtleWorld. Nó gọi `steer`, để rẽ Turtle theo hướng mong muốn, `wobble`, để rẽ theo hướng ngẫu nhiên tùy theo mức độ mất trật tự của Turtle và `move`, để đi tiến vài pixel về phía trước, tùy theo tốc độ của Turtle.

3. Hãy tạo một file `Tagger.py`. Nhập tất cả mọi thứ từ `Wobbler`, rồi định nghĩa một lớp tên là `Tagger` để thừa kế từ `Wobbler`. Gọi `make_world` có chuyển đối tượng lớp `Tagger` làm đối số.
4. Thêm phương thức có tên `steer` và `Tagger` để thay thế cho phương thức ở `Wobbler`. Hãy bắt đầu bằng việc viết một bản chương trình để luôn luôn hướng Turtle về gốc tọa độ. Gợi ý: dùng hàm toán học `atan2` và các thuộc tính của Turtle bao gồm `x`, `y` và `heading`.
5. Sửa lại `steer` sao cho các Turtle hoạt động trong phạm vi cho phép. Để gỡ lỗi, có thể bạn sẽ cần dùng đến nút {Step}, có tác dụng gọi `step` mỗi lần cho từng Turtle.
6. Sửa lại `steer` sao cho các Turtle hướng về phía con gần nó nhất. Gợi ý: Turtle có một thuộc tính, `world`, vốn là một tham chiếu đến TurtleWorld mà chúng sống trong đó; còn TurtleWorld lại có một thuộc tính, `animals`, là danh sách tất cả những Turtles có trong vùng.
7. Sửa lại `steer` để các Turtle chơi *tag*. Bạn có thể thêm các phương thức vào `Tagger` đồng thời cũng có thể thay thế đè lên `steer` và `__init__`, nhưng bạn không được sửa đổi hoặc thay đè vào `step`, `wobble` hay `move`. Ngoài ra, `steer` được phép thay đổi phương hướng của Turtle nhưng không được thay đổi vị trí.

   Hãy chỉnh sửa luật chơi và phương thức `steer` để trò chơi hấp dẫn hơn; chẳng hạn, có thể làm cho những Turtle bắt được Turtle sau quá trình đuổi lâu dài.

Bạn có thể tham khảo lời giải của tôi tại thinkpython.com/code/Tagger.py.

---

1. Xem wikipedia.org/wiki/Bottom_dealing ↩

2. Sơ đồ mà tôi dùng ở đây cũng giống như UML (xem wikipedia.org/wiki/Unified_Modeling_Language), với một số lược bớt. ↩

Chương 19: Nghiên cứu cụ thể: Tkinter

Trở về Mục lục quyển sách

GUI

Phần lớn chương trình ta đã gặp đều hoạt động trên nền chữ, nhưng cũng có nhiều chương trình dùng **giao diện đồ họa người dùng**, cũng được gọi là **GUI**.

Python cung cấp một số cách để viết chương trình GUI: dùng wxPython, Tkinter, và Qt. Mỗi cách đều có ưu và nhược điểm, đó cũng là lí do mà Python chưa có quy chuẩn cụ thể về mặt này.

Công cụ mà tôi trình bày trong chương này là Tkinter vì tôi nghĩ rằng đó là thứ dễ nhất mà chúng ta bắt đầu học. Đa số các khái niệm trong chương này cũng dùng được với các module GUI khác.

Có một số cuốn sách và trang web về Tkinter. Một trong những nguồn trực tuyến tốt nhất là *An Introduction to Tkinter* viết bởi Fredrik Lundh.

Tôi đã viết một module có tên `Gui.py` đi kèm theo Swampy. Nó cung cáp một giao diện được đơn giản hóa với các hàm và lớp trong Tkinter. Những ví dụ trong chương này đều dựa theo module đó.

Sau đây là một ví dụ đơn giản nhằm tạo ra và hiển thị một Gui:

Để tạo ra một GUI, bạn cần nhập vào `Gui` rồi cá thể hóa một đối tượng Gui:

```
from Gui import * g = Gui() g.title('Gui') g.mainloop()
```

Khi bạn chạy đoạn mã này, một cửa sổ sẽ xuất hiện với một hình vuông màu xám và tiêu đề chữ Gui. `mainloop` chạy **vòng lặp sự kiện**, để đợi người dùng thực hiện một thao tác và phản hồi một cách tương ứng. Đó là một vòng lặp vô hạn; nó tiếp tục chạy đến khi người dùng đóng cửa sổ, hoặc ấn Control-C, hoặc thực hiện thao tác khiến cho chương trình kết thúc.

Đối tượng Gui này chẳng có gì đáng kể vì nó không chứa một **widget** nào. Widget là thành phần tạo nên GUI; chúng bao gồm:

Nút (Button):
: Một widget, có chứa chữ hoặc hình, để thực hiện một việc khi được nhấn.

Nền (Canvas):
: Một vùng trên đó có thể hiển thị đường nét, hình chữ nhật, hình tròn và các hình khác.

Ô chữ (Entry):
: Một vùng mà người dùng có thể gõ chữ vào đó.

Thanh trượt (Slider):
: Một widget có khả năng điều chỉnh phần hiển thị của một widget khác.

Khung (Frame):
: Một widget để chứa, thường luôn hiển thị, và đựng các widget khác ở trong.

Hình vuông màu xám mà bạn nhìn thấy khi tạo ra Gui là một Frame. Khi bạn tạo ra một widget mới, nó sẽ được thêm vào Frame này.

Nút và những điểm gọi lại

Phương thức `bu` sau đây tạo ra một widget kiểu Button (nút):

```
button = g.bu(text='Press me.')
```

Giá trị trả về từ bu là một đối tượng Button. Hình nút hiện ra trong Frame là hình biểu diễn của đối tượng này; bạn có thể điều khiển nút bằng cách gọi các phương thức của nó.

bu nhận đến 32 tham biến có ảnh hưởng đến hình dáng và tính năng của nút. Các tham biến này được gọi là những **tuỳ chọn**. Thay vì cấp giá trị cho cả 32 tuỳ chọn này, bạn có thể dùng đối số từ khoá, như text='Press me.', để chỉ định những tuỳ chọn nào bạn cần và số còn lại thì dùng các giá trị mặc định.

Khi bạn thêm một widget vào trong Frame, nó sẽ bị "co hẹp;" nghĩa là, Frame sẽ co lại cho bằng kích thước của Button. Nếu bạn thêm nhiều widget nữa, Frame sẽ nở ra để có đủ chỗ.

Phương thức la sau đây sẽ tạo ra một widget kiểu Label:

```
label = g.la(text='Press the button.')
```

Tkinter mặc nhiên chồng chất các widget theo chiều đứng và căn chúng vào giữa. Ta sẽ xét cách thay thế đè lên biểu hiện đó.

Nếu ấn nút, bạn sẽ thấy không ăn thua. Đó là vì bạn chưa "tiếp điện" cho nó; nghĩa là bạn chưa ra lệnh cho nó làm gì!

Tuỳ chọn để điều khiển động thái của nút là command. Giá trị của command là một hàm vốn sẽ được thực hiện khi nút được nhấn. Chẳng hạn, sau đây là một hàm để tạo mới một Label:

```
def make_label(): g.la(text='Thank you.')
```

Bây giờ khi bạn tạo ra nút với hàm này đưa vào làm command của nó:

```
button2 = g.bu(text='No, press me!', command=make_label)
```

Khi bạn ấn nút này, nó sẽ thực hiện make_label và một dòng chữ mới sẽ xuất hiện.

Giá trị của tuỳ chọn command là một đối tượng hàm, và cũng được gọi là một **điểm gọi lại** vì sau khi bạn gọi bu để tạo ra nút, luồng thực hiện sẽ "gọi lại" khi người dùng nhấn nút.

Kiểu luồng thực hiện này là đặc trưng cho **lập trình hướng sự kiện**. Các hành động của người dùng, như ấn nút và gõ phím, được gọi là {sự kiện}. Trong lập trình hướng sự kiện, luồng thực hiện được quyết định bởi người dùng thay vì người lập trình.

Thử thách đối với lập trình hướng sự kiện là việc thiết lập một tập hợp các widget và điểm gọi lại sao cho chúng hoạt động đúng (hoặc ít nhất phải phát sinh những thông báo lỗi hợp lý) cho bất kì chuỗi hành động nào từ phía người dùng.

> Hãy viết một chương trình để tạo ra GUI chỉ có một nút. Khi nút được nhấn, một nút thứ hai sẽ được tạo ra. Khi nút thứ hai này được nhấn, cần tạo ra một dòng chữ viết, "Nice job!".
>
> Điều gì sẽ xảy ra khi bạn ấn các nút nhiều lần? Bạn có thể tham khảo lời giải của tôi tại thinkpython.com/code/button_demo.py

Các widget kiểu Canvas

Một trong những widget năng động nhất là Canvas, có tác dụng tạo ra một vùng để vẽ các đường thẳng, hình tròn và những hình khác. Nếu đã làm Bài tập canvas, bạn đã quen với kiểu widget nền này.

Phương thức ca sau đây sẽ tạo ra Canvas mới:

```
canvas = g.ca(width=500, height=500)
```

`width` và `height` là các kích thước của nền tính theo pixel.

Sau khi tạo ra một widget, bạn vẫn có thể thay đổi các giá trị của tuỳ chọn bằng phương thức `config`. Chẳng hạn, tuỳ chọn `bg` thay đổi màu nền:

```
canvas.config(bg='white')
```

Giá trị của `bg` là một chuỗi chứa tên của một màu. Tập hợp những tên màu hợp lệ lại khác nhau với từng phiên bản Python dành cho các loại máy, nhưng tất cả phiên bản đều có ít nhất là các màu sau:

```
white black red green blue cyan yellow magenta
```

Các hình trên một Canvas được gọi là **item**. Chẳng hạn, phương thức của Canvas có tên `circle` vẽ một hình tròn:

```
item = canvas.circle([0,0], 100, fill='red')
```

Đối số thứ nhất là cặp toạ độ của tâm hình tròn; đối số thứ hai là bán kính.

`Gui.py` có một hệ toạ độ Đề-các tiêu chuẩn với điểm gốc tại tâm của Canvas và trục *y* hướng lên. Nó khác với một số hệ thống đồ thị khác trong đó điểm gốc lại ở góc bên trái phía trên và trục *y* hướng xuống.

Tuỳ chọn `fill` chỉ định rằng hình tròn cần được tô màu đỏ.

Giá trị trả về từ `circle` là một đối tượng Item nhằm cung cấp những phương thức để sửa đổi các item trên nền. Chẳng hạn, bạn có thể dùng `config` để thay đổi bất kì tuỳ chọn nào của hình tròn:

```
item.config(fill='yellow', outline='orange', width=10)
```

`width` là độ dày của viền tính theo pixel; `outline` là màu của viền.

> Hãy viết một chương trình để tạo ra một Canvas và một Button. Khi người dùng ấn Button, cần phải vẽ ra một hình tròn trên nền.

Dãy các toạ độ

Phương thức `rectangle` nhận vào một dãy hai toạ độ để định vị hai góc đối diện của hình chữ nhật. Ví dụ này vẽ một hình chữ nhật màu xanh lá cây với góc trái phía dưới ở gốc tạo độ và góc phải phía trên ở (200,100):

```
canvas.rectangle([[0, 0], [200, 100]], fill='blue', outline='orange', width=10)
```

Cách làm chỉ định các góc nói trên cũng được gọi là chỉ định **hình bao** vì hai điểm góc đó giới hạn hình chữ nhật.

`oval` nhận vào một hình bao và vẽ một hình trái xoan bên trong hình chữ nhật bao đó:

```
canvas.oval([[0, 0], [200, 100]], outline='orange', width=10)
```

`line` nhận vào một dãy các toạ độ và vẽ đoạn thẳng nối các điểm đó. Ví dụ sau vẽ hai cạnh bên của một hình tam giác:

```
canvas.line([[0, 100], [100, 200], [200, 100]], width=10)
```

`polygon` nhận vào cũng các đối số như trên, nhưng nó còn vẽ thêm một đoạn thẳng nối hai điểm

đầu cuối của đa giác (nếu cần) và tô màu nó:

```
canvas.polygon([[0, 100], [100, 200], [200, 100]], fill='red', outline='orange',
width=10)
```

Các widget khác

Tkinter có hai loại widget cho phép người dùng gõ chữ vào: Entry, chứa một dòng chữ, và Text, chứa nhều dòng.

`en` tạo ra một Entry mới:

```
entry = g.en(text='Default text.')
```

Tuỳ chọn `text` cho phép bạn đưa chữ vào Entry một khi nó được tạo ra. Phương thức `get` trả về nội dung của Entry (mà người dùng thay đổi được):

```
>>> entry.get() 'Default text.'
```

`te` tạo ra một widget kiểu Text:

```
text = g.te(width=100, height=5)
```

`width` và `height` là các kích thước của tính theo số kí tự bề rộng và số dòng.

`insert` đặt đoạn chữ vào trong widget kiểu Text:

```
text.insert(END, 'A line of text.')
```

`END` là một chỉ số đặc biệt để biểu thị kí tự cuối trong widget kiểu Text.

Bạn cũng có thể chỉ định một kí tự bằng cách dùng chỉ số dấu chấm, như `1.1`, trong đó số dòng đi trước dấu chấm còn số cột đi sau. Ví dụ sau thêm chữ vào sau kí tự thứ nhất của dòng đầu tiên.

```
>>> text.insert(1.1, 'nother')
```

Phương thức `get` đọc những chữ có trong widget; nó nhận các đối số gồm có chỉ số đầu và cuối. Ví dụ sau trả về toàn bộ chữ trong widget, kể cả kí tự xuống dòng:

```
>>> text.get(0.0, END) 'Another line of text.\n'
```

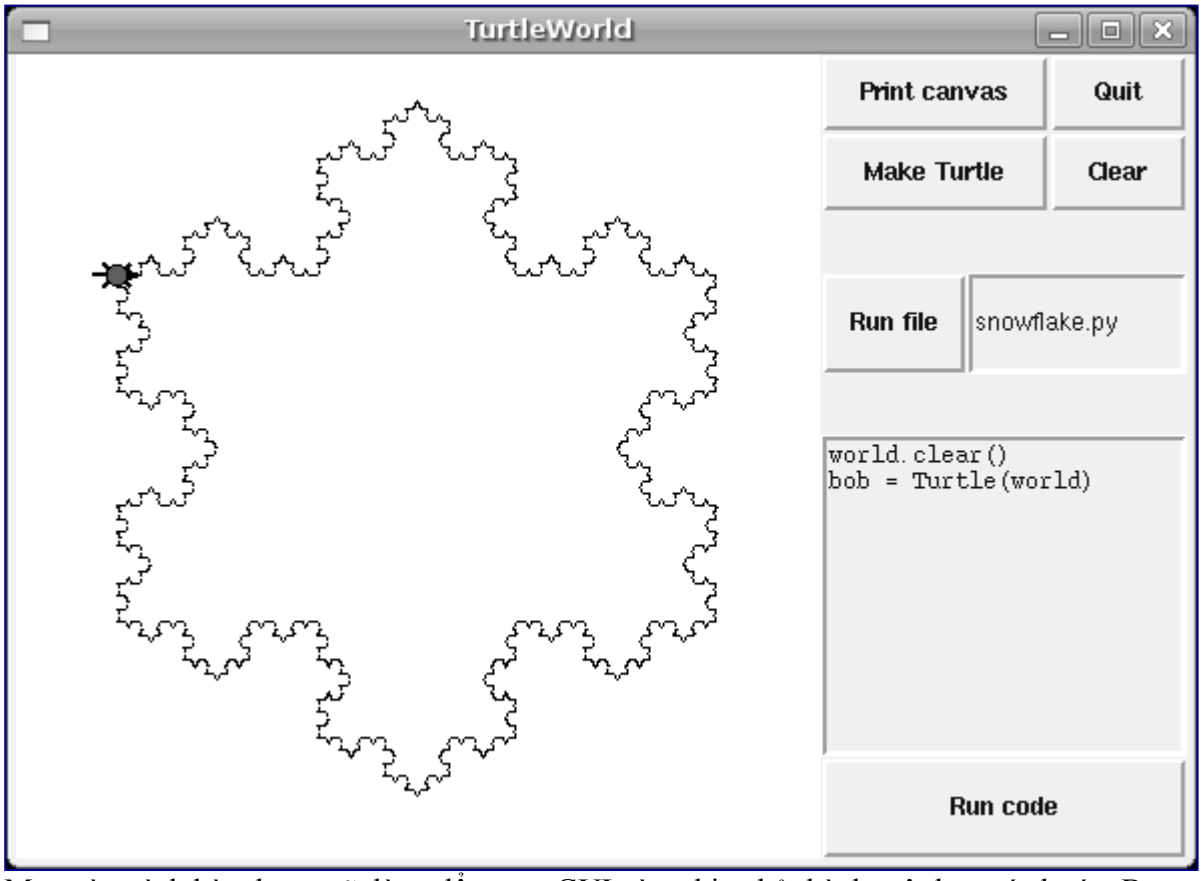
Phương thức `delete` xoá chữ trong widget; ví dụ sau xoá toàn bộ, chỉ để lại hai kí tự đầu tiên:

```
>>> text.delete(1.2, END) >>> text.get(0.0, END) 'An\n'
```

> Hãy sửa lại lời giải của bạn cho Bài tập circle bằng cách thêm vào một widget kiểu Entry và một nút thứ hai. Khi người dùng nhấn nút thứ hai, chương trình sẽ đọc một tên màu từ Entry và dùng nó để đổi cho màu tô hình tròn. Hãy dùng `config` để sửa đổi hình tròn đã có; đừng tạo ra hình mới.
>
> Chương trình của bạn cần giải quyết được các trường hợp khi người dùng thử thay đổi màu của một hình tròn khi nó chưa được tạo ra, và trường hợp khi tên màu không hợp lệ.
>
> Bạn có thể xem lời giải của tôi tại thinkpython.com/code/circle_demo.py.

Xếp các widget

Đến đây ta đã xếp các widget theo một cột thẳng đứng, nhưng trong hầu hết các GUI, sự sắp xếp còn phức tạp hơn nhiều. Chẳng hạn, sau đây là một phiên bản được rút gọn chút ít từ TurtleWorld (xem Chương 4).



Mục này trình bày đoạn mã dùng để tạo ra GUI này, chia nhỏ thành một loạt các bước. Bạn có thể tải về ví dụ trọn vẹn từ thinkpython.com/code/SimpleTurtleWorld.py.

Ở cấp cao nhất, GUI này chứa hai widget—một Canvas và một Frame —được xếp theo hàng ngang. Vì vậy bước thứ nhất là tạo ra hàng đó.

```
class SimpleTurtleWorld(TurtleWorld): """This class is identical to TurtleWorld,
but the code that lays out the GUI is simplified for explanatory purposes."""
def setup(self): self.row() ...
```

`setup` là hàm dùng để tạo ra và sắp xếp các widget. Việc sắp đặt các widget trong GUI còn được gọi là **xếp**.

`row` tạo ra một Frame hàng và làm nó trở nên "Frame hiện hành". Cho đến tận khi Frame này được đóng lại hoặc một Frame khác được tạo ra, tất cả các widget tiếp theo đều được xếp trên một hàng.

Sau đây là đoạn mã dùng để tạo Canvas và Frame cột để chứa những widget khác:

```
 self.canvas = self.ca(width=400, height=400, bg='white') self.col()
```

Widget thứ nhất trong cột là một Frame lưới, trong đó lại chứa bốn nút được xếp theo hai hàng và hai cột:

```
 self.gr(cols=2) self.bu(text='Print canvas', command=self.canvas.dump)
self.bu(text='Quit', command=self.quit) self.bu(text='Make Turtle',
command=self.make_turtle) self.bu(text='Clear', command=self.clear) self.endgr()
```

`gr` tạo ra lưới; đối số ở đây là số cột. Các widget trong lưới được đặt lần lượt từ trái sang phải, từ trên xuống dưới.

Nút thứ nhất sử dụng `self.canvas.dump` làm điểm gọi lại; còn nút thứ hai dùng `self.quit`. Đó là những **phương thức hạn hẹp**, theo nghĩa là chúng được gắn với những đối tượng cụ thể. Khi được kích hoạt, chúng được gọi với các đối tượng đó.

Widget tiếp theo trong cột là một Frame hàng có chứa một Button và một Entry:

```
 self.row([0,1], pady=30) self.bu(text='Run file', command=self.run_file)
self.en_file = self.en(text='snowflake.py', width=5) self.endrow()
```

Đối số thứ nhất cho `row` là một danh sách các trọng số dùng để xác định xem có bao nhiêu khoảng trống phụ thêm được đặt giữa các widget. Danh sách `[0,1]` nghĩa là toàn bộ khoảng trống thêm vào được dành cho widget thứ hai, tức là Entry. Nếu chạy chương trình và thay đổi kích cỡ cửa sổ, bạn sẽ thấy rằng Entry sẽ được phóng to hoặc thu nhỏ, còn Button thì không.

Tùy chọn `pady` "chèn" hàng này theo phương *y*, bằng cách thêm vào 30 pixel khoảng trống cả phía trên lẫn dưới.

`endrow` kết thúc hàng widget này, vì vậy các widget tiếp theo được xếp trong Frame cột. `Gui.py` lưu giữ một chồng các Frame:

- Khi bạn dùng `row`, `col` hoặc `gr` để tạo ra Frame, nó sẽ được xếp lên đỉnh của chồng và trở nên Frame hiện hành.
- Khi bạn dùng `endrow`, `endcol` hoặc `endgr` để đóng một Frame, nó được đẩy khỏi chồng xếp và Frame liền trước đó trên chồng trở thành Frame hiện hành.

Phương thức `run_file` đọc vào nội dung của Entry, dùng nó như một tên file, đọc vào nội dung và chuyển nó đến cho `run_code`. Còn `self.inter` là một đối tượng Interpreter (trình thông dịch) để nhận vào một chuỗi và thực hiện nó như đoạn mã lệnh Python.

```
 def run_file(self): filename = self.en_file.get() fp = open(filename) source =
fp.read() self.inter.run_code(source, filename)
```

Hai widget sau cùng gồm có một Text và một Button:

```
 self.te_code = self.te(width=25, height=10) self.te_code.insert(END,
'world.clear()\n') self.te_code.insert(END, 'bob = Turtle(world)\n')
self.bu(text='Run code', command=self.run_text)
```

`run_text` cũng tương tự như `run_file`, chỉ khác ở chỗ nó lấy đoạn mã từ widget Text thay vì lấy từ file:

```
 def run_text(self): source = self.te_code.get(1.0, END)
self.inter.run_code(source, '<user-provided code>')
```

Thật không may là chi tiết về cách sắp đặt widget rất khác nhau giữa các ngôn ngữ, và giữa các module trong Python. Riêng Tkinter cho phép ba cơ chế sắp đặt widget. Các cơ chế này được gọi là **quản lí không gian**. Cơ chế mà tôi giới thiệu trong mục này được gọi là quản lí không gian dạng "lưới"; các dạng khác là "xếp" và "đặt".

May mắn là phần nhiều các khái niệm trong mục này cũng áp dụng được cho các module lập GUI và các ngôn ngữ khác.

Trình đơn và các *Callable*

Menubutton là một widget trông giống như một nút, nhưng khi bạn nhấn, nó sẽ bật ra một trình

đơn. Sau khi người dùng chọn một mục, trình đơn sẽ biến mất.

Sau đây là đoạn mã nhằm tạo ra một Menubutton để chọn màu (bạn có thể tải nó về từ thinkpython.com/code/menubutton_demo.py):

```
g = Gui() g.la('Select a color:') colors = ['red', 'green', 'blue'] mb =
g.mb(text=colors[0])
```

`mb` tạo ra một Menubutton. Ban đầu, dòng chữ trên nút là tên gọi của màu mặc định. Vòng lặp sau đây tạo ra mỗi mục trên trình đơn ứng với một màu:

```
for color in colors: g.mi(mb, text=color, command=Callable(set_color, color))
```

Đối số thứ nhất của `mi` là Menubutton mà các mục này gắn liền với.

Tùy chọn `command` là một đối tượng Callable; đây là một khái niệm mới. Đến giờ chúng ta đã thấy các hàm và phương thức hạn hẹp được dùng như những điểm gọi lại, vốn sẽ hoạt động tốt nếu bạn không cần chuyển đối số nào vào trong hàm. Với trường hợp ngược lại bạn sẽ phải lập một đối tượng Callable có chứa một hàm, như `set_color`, cùng các đối số của nó, như `color`.

Đối tượng Callable lưu trữ một tham chiếu đến hàm và các đối số, tất cả như những thuộc tính. Sau này, khi người dùng kích chuột vào một mục trên trình đơn, điểm gọi lại sẽ gọi hàm và truyền vào các đối số được lưu trữ.

Có thể viết `set_color` như sau:

```
def set_color(color): mb.config(text=color) print color
```

Khi người dùng chọn một mục trên trình đơn và `set_color` được gọi, nó sẽ đặt cấu hình cho Menubutton để hiển thị màu mới được chọn. Nó cũng in ra tên màu; nếu bạn thử ví dụ này, bạn sẽ thấy đúng là `set_color` được gọi khi bạn chọn một mục (và *không* được gọi khi bạn tạo ra đối tượng Callable).

Bó buộc

Bó buộc là việc gắn một widget, một sự kiện và một điểm gọi lại với nhau: khi một sự kiện (như việc nhấn nút) xảy ra đối với widget, điểm gọi lại sẽ được kích hoạt.

Nhiều widget có những bó buộc mặc định. Chẳng hạn, khi bạn nhấn một nút, sự bó buộc mặc định sẽ thay đổi nền của nút bấm khiến nó trông như bị lõm xuống. Khi bạn nhả tay ra, sự bó buộc sẽ làm phục hồi vẻ bề ngoài của nút và kích hoạt điểm gọi lại được chỉ định ở tùy chọn `command`.

Bạn có thể dùng phương thức `bind` để thay thế đè lên những bó buộc mặc định này, hoặc tạo ra những bó buộc mới. Chẳng hạn, đoạn mã này tạo ra một bó buộc cho nền (bạn có thể tải về các đoạn mã ở mục này từ địa chỉ thinkpython.com/code/draggable_demo.py):

```
ca.bind('<ButtonPress-1>', make_circle)
```

Đối số thứ nhất là một chuỗi chứa sự kiện; sự kiện này được bật lên khi người dùng ấn nút chuột bên trái. Các sự kiện khác liên quan đến thao tác chuột gồm có `ButtonMotion`, `ButtonRelease` và `Double-Button`.

Đối số thứ hai là chuôi nắm sự kiện. Một chuôi nắm sự kiện là một hàm hoặc phương thức hạn hẹp, giống như một điểm gọi lại, nhưng có một khác biệt quan trọng là chuôi nắm sự kiện nhận đối tượng Event làm tham biến. Sau đây là một ví dụ:

```
def make_circle(event): pos = ca.canvas_coords([event.x, event.y]) item =
ca.circle(pos, 5, fill='red')
```

Đối tượng Event bao gồm thông tin về kiểu sự kiện và các chi tiết như tọa độ của con trỏ chuột. Ở ví dụ này thông tin mà ta cần là vị trí của con trỏ khi kích chuột. Các giá trị này được đo bằng "tọa độ pixel", vốn được định nghĩa bởi hệ thống đồ họa bên trong. Phương thức `canvas_coords` viết tắt từ "Canvas coordinates", cũng tương thích với các phương thức của Canvas như `circle`.

Với các widget Entry, cách thông thường là buộc chúng với sự kiện `<Return>`, vốn được bật ra khi người dùng gõ phím {Return} hay {Enter}. Chẳng hạn, đoạn mã sau tạo ra một Button và một Entry.

```
bu = g.bu('Make text item:', make_text) en = g.en() en.bind('<Return>',
make_text)
```

`make_text` được gọi khi Button được ấn hoặc khi người dùng gõ phím {Return} khi đang gõ trong Entry. Để làm được điều này, ta cần một hàm có thể gọi được như một lệnh (không có đối số) hoặc một chuôi nắm sự kiện (với một Event làm đối số):

```
def make_text(event=None): text = en.get() item = ca.text([0,0], text)
```

`make_text` nhận vào nội dung của Entry và hiển thị nó dưới dạng một item kiểu Text bên trong Canvas.

Cũng có thể tạo ra những bó buộc cho các item trong Canvas. Sau đây là một lời định nghĩa lớp cho `Draggable`, vốn là một lớp con của `Item` để cung cấp những bó buộc giúp thực hiện thao tác kéo-và-thả.

```
class Draggable(Item): def __init__(self, item): self.canvas = item.canvas
self.tag = item.tag self.bind('<Button-3>', self.select) self.bind('<B3-
Motion>', self.drag) self.bind('<Release-3>', self.drop)
```

Phương thức init nhận một Item làm tham biến. Nó sao chép lại các thuộc tính của Item rồi tạo những bó buộc cho ba sự kiện: nhấn nút, nút được nhấn khi di chuyển, và nhả nút.

Một chuôi nắm sự kiện, `select`, lưu trữ các tọa độ của sự kiện hiện hành và màu ban đầu của đối tượng, sau đó đổi màu sang vàng:

```
 def select(self, event): self.dragx = event.x self.dragy = event.y self.fill =
self.cget('fill') self.config(fill='yellow')
```

`cget` viết tắt cho "get configuration;" nó nhận vào tên của một tùy chọn dưới dạng chuỗi và trả lại màu hiện thời của tùy chọn đó.

`drag` tính xem đối tượng phải di chuyển bao xa so với điểm khởi đầu, cập nhật các tọa độ được lưu lại, và sau đó di chuyển đối tượng.

```
 def drag(self, event): dx = event.x - self.dragx dy = event.y - self.dragy
self.dragx = event.x self.dragy = event.y self.move(dx, dy)
```

Việc tính toán này được thực hiện trên các tọa độ pixel; ta không cần phải chuyển về tọa độ của Canvas.

Sau cùng, `drop` khôi phục lại màu ban đầu của đối tượng:

```
 def drop(self, event): self.config(fill=self.fill)
```

Bạn có thể dùng lớp `Draggable` để thêm vào tính năng kéo-và-thả cho một đối tượng có sẵn. Chẳng hạn, sau đây là một phiên bản được sửa lại của `make_circle` trong đó dùng `circle` để tạo ra một Item và `Draggable` để khiến nó kéo được:

```
def make_circle(event): pos = ca.canvas_coords([event.x, event.y]) item =
```

```
ca.circle(pos, 5, fill='red') item = Draggable(item)
```

Ví dụ này cho thấy một trong những lợi ích của việc thừa kế: bạn có thể sửa lại tính năng của một lớp cha mẹ mà không cần sửa đổi định nghĩa của nó. Điều này đặc biệt có ích nếu bạn muốn thay đổi biểu hiện được định nghĩa trong một module mà bạn không viết ra.

Gỡ lỗi

Một trong những khó khăn khi lập trình GUI là phải theo dõi những việc nào xảy ra trong khi xây dựng GUI và những việc nào sẽ xảy ra để phản hồi lại sự kiện từ phía người dùng.

Chẳng hạn, khi bạn thiết lập một điểm gọi lại, một lỗi thông thường là gọi hàm thay vì chuyển một tham chiếu chỉ đến nó:

```
def the_callback(): print 'Called.' g.bu(text='This is wrong!',
command=the_callback())
```

Nếu chạy đoạn mã này, bạn sẽ thấy rằng nó gọi `the_callback` lập tức, và *sau đó* mới tạo ra nút. Khi bạn ấn nút, chương trình sẽ không làm gì vì bạn giá trị được trả về từ `the_callback` là `None`. Thường thì bạn sẽ không muốn kích hoạt một điểm gọi lại khi đang xây dựng GUI; nó cần được kích hoạt sau này, để phản hồi lại sự kiện từ phía người dùng.

Một khó khăn khác của lập trình GUI là bạn không có quyền kiểm soát luồng thực hiện của chương trình. Phần nào của chương trình được chạy và thứ tự thực hiện đều được quyết định bởi những thao tác của người dùng. Điều đó có nghĩa là bạn phải thiết kế chương trình để hoạt động đúng với một dãy sự kiện bất kì có thể xảy ra.

Chẳng hạn, GUI trong Bài tập circle2 có hai widget: một cái để tạo ra một Circle và cái kia để đổi màu Circle. Nếu người dùng tạo ra hình tròn rồi mới đổi màu của nó thì không sao. Nhưng điều gì sẽ xảy ra nếu người dùng đổi màu của một hình tròn thậm chí chưa tồn tại? Hay tạo ra nhiều hình tròn?

Càng có nhiều widget, sẽ càng khó hình dung tất cả những dãy sự kiện có thể xảy ra. Một cách kiểm soát sự phức tạp này là bằng cách gói trạng thái của hệ thống vào một đối tượng rồi xem xét:

- Các trạng thái có thể là gì? Ở ví dụ với Circle, ta có thể xét hai trạng thái: trước và sau khi người dùng tạo ra hình tròn thứ nhất.
- Với mỗi trạng thai, những sự kiện nào có thể xảy ra? Ở ví dụ trên, người dùng có thể ấn một trong hai nút, hoặt kết thúc chương trình.
- Với mỗi cặp trạng thái-sự kiện, đâu là kết quả mong muốn? Vì có hai trạng thái và hai nút, nên sẽ có bốn cặp trạng thái-sự kiện cần xét đến.
- Điều gì có thể gây ra một sự chuyển đổi từ trạng thái này sang trạng thái khác? Trong trường hợp này, có một sự chuyển đổi khi người dùng tạo ra hình tròn thứ nhất.

Bạn cũng có thể thấy cần định nghĩa, và kiểm tra, những bất biến cần thỏa mãn bất kể dãy các sự kiện như thế nào.

Phương pháp lập trình GUI này có thể giúp bạn viết mã lệnh đúng mà không mất thời gian thử từng dãy sự kiện có thể từ phía người dùng!

Thuật ngữ

GUI:
: Giao diện đồ họa người dùng.

widget:

Một trong các yếu tố để tạo nên GUI, bao gồm nút, trình đơn, ô chữ, v.v.

tùy chọn:
: Giá trị chi phối bề ngoài hoặc tính năng của một widget.

đối số từ khóa:
: Đối số để chỉ định tên của tham biến như một phần của lời gọi hàm.

điểm gọi lại:
: Hàm gắn liền với một widget và sẽ được gọi khi người dùng thực hiện thao tác.

phương thức hạn hẹp:
: Phương thức gắn với một cá thể riêng.

lập trình hướng sự kiện:
: Phong cách lập trình trong đó luồng thực hiện được quyết định bởi thao tác từ phía người dùng.

sự kiện:
: Thao tác từ phía người dùng, như kích chuột, gõ phím để GUI phản hồi lại.

vòng lặp sự kiện:
: Vòng lặp vô hạn để đợi thao tác từ phía người dùng và phản hồi.

item:
: Yếu tố đồ họa trên widget kiểu Canvas.

hình bao:
: Hình chữ nhật bao chứa một nhóm các item, thường được chỉ định bởi hai góc đối diện.

xếp:
: Sắp xếp và hiển thị các yếu tố của GUI.

quản lí không gian:
: Hệ thống giúp xếp các widget.

bó buộc:
: Hình thức gắn kết giữa một widget, một sự kiện, và một chuôi nắm sự kiện. Chuôi nắm sẽ được gọi khi sự kiện xảy ra đối với widget.

Bài tập

Trong bài tập này, bạn sẽ viết một trình xem ảnh. Sau đây là một ví dụ đơn giản:

```
g = Gui() canvas = g.ca(width=300) photo =
PhotoImage(file='danger.gif') canvas.image([0,0], image=photo)
g.mainloop()
```

`PhotoImage` đọc vào một file và trả lại một đối tượng `PhotoImage` mà Tkinter có thể hiển thị. `Canvas.image` đặt hình ảnh lên nền, căn giữa theo các tọa độ cho trước. Bạn cũng có thể đặt các hình lên nhãn, nút, và một số widget khác:

```
g.la(image=photo) g.bu(image=photo)
```

PhotoImage chỉ có thể xử lý một số ít định dạng ảnh, như GIF and PPM, nhưng ta có thể dùng Python Imaging Library (PIL) để đọc các dạng file khác.

Tên của module trong PIL là `Image`, nhưng Tkinter đã định nghĩa một đối tượng cùng tên. Để tránh sự xung khắc này, bạn có thể dùng `import...as` như sau:

```
import Image as PIL import ImageTk
```

Dòng đầu tiên nhập vào `Image` và đặt cho nó tên địa phương là `PIL`. Dòng thứ hai nhập vào `ImageTk`, vốn có thể chuyển một hình PIL thành dạng PhotoImage của Tkinter. Sau đây là một ví dụ:

```
image = PIL.open('allen.png') photo2 = ImageTk.PhotoImage(image)
g.la(image=photo2)
```

1. Hãy tải về `image_demo.py`, `danger.gif` và `allen.png` từ thinkpython.com/code. Chạy `image_demo.py`. Bạn có thể phải cài đặt `PIL` và `ImageTk`. Chúng có thể đã sẵn có trong kho phần mềm của máy bạn, nhưng nếu chưa thì có thể tải được về từ pythonware.com/products/pil/.
2. Trong `image_demo.py`, hãy đổi tên của PhotoImage thứ hai từ `photo2` thành `photo` và chạy lại chương trình. Bạn cần thấy được PhotoImage thứ hai chứ không phải tấm thứ nhất.Vấn đề là khi bạn gán lại `photo` nó ghi đè lên tham chiếu đến PhotoImage thứ nhất, mà bản thân sau đó sẽ mất đi. Điều tương tự cũng xảy ra khi bạn gán một PhotoImage cho một biến địa phương; nó biến mất khi hàm kết thúc.Để tránh điều này, bạn phải lưu một tham chiếu cho mỗi PhotoImage mà bạn muốn giữ lại. Bạn có thể dùng một biến toàn cục, hoặc lưu lại các PhotoImage trong một cấu trúc dữ liệu hoặc dưới dạng thuộc tính của một đối tượng.

   Biểu hiện này có thể rất khó chịu, chính vì vậy mà tôi báo cho bạn biết (và hình dùng cho thí nghiệm này có chữ “Danger!”).

3. Bắt đầu từ ví dụ này, hãy viết một chương trình để nhận vào tên của một thư mục và lặp qua tất cả các file, hiển thị những file mà PIL coi là hình ảnh. Bạn có thể dùng lệnh `try` để bắt những file mà PIL không nhận là ảnh.Khi người dùng kích chuột vào ảnh, chương trình cần phải hiển thị tấm tiếp theo.
4. PIL có nhiều phương thức xử lý ảnh. Bạn có thể tham khảo toàn bộ tại pythonware.com/library/pil/handbook. Hãy chọn ra một số phương thức, đố bạn lập một GUI để áp dụng những phương thức đó lên tấm ảnh?

Bạn có thể tải về một lời giải đơn giản từ thinkpython.com/code/ImageBrowser.py.

Một trình đồ họa véc-tơ là một chương trình cho phép người dùng vẽ và chỉnh sửa những hình trên màn hình và tạo ra các file kết quả dưới dạng đồ họa véc-tơ như Postscript và SVG[1].

Hãy viết một chương trình xử lý đồ họa véc-tơ bằng Tkinter. Ít nhất nó phải cho phép người dùng vẽ đường thẳng, hình tròn, hình chữ nhật, và phải dùng `Canvas.dump` để phát sinh bản mô tả Postscript cho nội dung trên Canvas.

Đố bạn lập trình cho phép người dùng chọn và thay đổi kích cỡ của các item trên Canvas.

Dùng Tkinter để viết một trình duyệt web đơn giản. Nó phải có một widget kiểu Text cho phép người dùng nhập vào một URL và một Canvas để hiển thị nội dung của trang

web.

Bạn có thể dùng module `urllib` để tải về các file (xem Bài tập urllib) và module `HTMLParser` để tách các thẻ HTML (xem `docs.python.org/lib/module-HTMLParser.html`).

Ít nhất là trình duyệt mà bạn viết phải xử lý được văn bản chữ thuần túy và các liên kết (đường link). Đố bạn lập trình xử lý màu nền, các thẻ định dạng và hình ảnh.

---

1. Xem `wikipedia.org/wiki/Vector_graphics_editor`. ↩

Phụ lục: Gỡ lỗi

Trở về Mục lục cuốn sách

Những kiểu lỗi khác nhau có thể xảy ra trong chương trình, và việc phân biệt chúng rất có ích để giúp tìm được lỗi nhanh hơn:

- Các lỗi cú pháp thường được Python báo khi nó dịch từ mã lệnh sang mã byte. Các lỗi loại này thường để chỉ rằng có điều gì đó không ổn trong cú pháp của chương trình. Chẳng hạn: Quên mất dấu hai chấm ở cuối dòng lệnh `def` sẽ cho ra một thông báo lỗi có phần dư thừa `SyntaxError: invalid syntax`.
- Các lỗi thực thi được trình thông dịch báo nếu có điều không ổn khi chương trình đang chạy. Hầu hết các thông báo lỗi thực thi đều kèm theo thông tin về vị trí lỗi xảy ra và những hàm nào đã được thực hiện. Ví dụ: Một phép đệ quy vô hạn cuối cùng sẽ gây ra lỗi thực thi "maximum recursion depth exceeded".
- Các lỗi ngữ nghĩa là những vấn đề với chương trình được chạy mà không phát sinh thông báo lỗi nhưng không làm đúng công việc như đã định. Chẳng hạn: Một biểu thức không được tính theo thứ tự mà bạn trông đợi và cho kết quả không chính xác.

Bước đầu trong quá trình gỡ lỗi là hình dung ra loại lỗi nào mà bạn cần giải quyết. Mặc dù các mục tiếp sau đây được tổ chức theo loại lỗi, song những kĩ thuật có thể được dùng để giải quyết nhiều trường hợp.

Lỗi cú pháp

Các lỗi cú pháp thường dễ sửa một khi bạn đã hình dung ra chúng. Không may là những thông báo lỗi thường không giúp ích nhiều. Những dòn thông báo thường gặp nhất là `SyntaxError: invalid syntax` và `SyntaxError: invalid token`, cả hai đều không cung cấp nhiều thông tin.

Tuy nhiên dòng thông báo cũng cho bạn biết rắc rối xảy ra ở đâu trong chương trình. Thực ra, nó cho bán biết nơi mà Python phát hiện ra vấn đề, vốn không nhất thiết chính là nơi lỗi xảy ra. Đôi khi lỗi xuất hiện trước vị trí của thông báo lỗi, thường là ngay ở dòng trước.

Nếu bạn xây dựng chương trình dần từng bước, bạn hẳn đã phán đoán hợp lí vị trí xảy ra lỗi. Nó sẽ ở dòng lệnh bạn vừa thêm vào.

Nếu bạn sao chép mã lệnh từ một quyển sách, hãy bắt đầu bằng việc so sánh cẩn thận mã lệnh của bạn với mã lệnh trong cuốn sách. Kiểm tra từng kí tự một. Đồng thời nhớ rằng sách cũng có thể sai, vì vậy nếu bạn thấy có chỗ giống như lỗi, thì hoàn toàn có thể vậy.

Sau đây là một số cách tránh các lỗi cú pháp thông dụng nhất:

1. Hãy chắc rằng bạn không dùng một từ khoá Python làm tên biến.
2. Kiểm tra xem bạn có kèm theo dấu hai chấm ở lệnh đầu các lệnh phức hợp chưa, bao gồm các lệnh `for`, `while`, `if`, và `def`.
3. Hãy chắc rằng các chuỗi trong mã lệnh đều có đủ cặp nháy kép.
4. Nếu bạn có chuỗi rải trên nhiều dòng với cặp ba dấu nháy (nháy đơn hoặc nháy kép), hãy chắc rằng bạn có kết thúc chuỗi một cách đúng đắn. Một chuỗi bỏ lửng có thể gây ra lỗi `invalid token` ở điểm cuối chương trình, hoặc coi phần còn lại của chương trình như một chuỗi đến khi nó bắt gặp chuỗi tiếp theo. Trong trường hợp thứ hai, nó thậm chí còn không báo lỗi gì!
5. Một toán tử mở ngoặc—`(`, `{`, hay `[`—mà quên được đóng sẽ làm cho Python chạy tiếp lấy

dòng bên dưới làm một phần của câu lệnh hiện hành. Thường thì lỗi sẽ xuất hiện ngay ở dòng kế tiếp.
6. Kiểm tra lỗi kinh điển xảy ra khi dùng = thay vì == trong câu lệnh điều kiện.
7. Kiểm tra sự thụt đầu dòng để đảm bảo tính đúng đắn. Python có thể xử lý các dấu cách và dấu tab, nhưng nếu bạn dùng lẫn cả hai loại có thể sẽ có vấn đề. Cách tốt nhất để tránh điều này là dùng một trình soạn văn bản chữ có thể nhận ra ngôn ngữ Python và tự căn lề một cách thống nhất.

Nếu các biện pháp trên đều không có tác dụng, hãy đọc mục tiếp theo…

Tôi vẫn tiếp tục sửa đổi mà chẳng thấy gì khác.

Nếu trình thông dịch báo rằng có lỗi và bạn không nhìn ra thì có thể vì bạn và trình thông dịch không nhìn vào cùng một đoạn mã. Hãy kiểm tra lại môi trường lập trình của bạn để chắc rằng chương trình mà bạn đang soạn thảo cũng chính được chạy bởi Python.

Nếu bạn không chắc chắn, hãy thử cố ý đưa vào một lỗi cú pháp hiển nhiên vào ngay đầu chương trình. Bây giờ chạy lại chương trình. Nếu trình thông dịch không tìm thấy lỗi mới thì rõ ràng là bạn không chạy đoạn mã lệnh mới.

Sau đây là một số trường hợp sai lầm dễ gặp:

- Bạn đã soạn thảo file mà lại quên lưu vào trước khi chạy lại. Một số môi trường lập trinh giúp bạn làm điều này, nhưng số khác thì không.
- Bạn đổi tên file, nhưng vẫn chạy chương trình có tên cũ.
- Một chi tiết trong môi trường phát triển bị thiết lập sai.
- Nếu bạn đang viết một module và dùng lệnh `import`, hãy chắc rằng bạn không đặt tên module giống như một trong những module tiêu chuẩn của Python.
- Nếu bạn đang dùng `import` để đọc một module, hãy nhớ rằng bạn phải khởi động lại trình thông dịch hoặc dùng `reload` để đọc một file đã bị chỉnh sửa. Còn nếu bạn nhập lại module thì máy sẽ chẳng làm gì.

Nếu bạn bị bí và không thể hình dung được chuyện gì đang xảy ra, một cách là bắt đầu lại từ đầu với một chương trình kiểu như “Hello, World!”, và chắc rằng bạn biết cách chạy một chương trình có sẵn. Sau đó dần thêm từng phần của chương trình cần chạy vào chương trình mới này.

Lỗi thực thi

Một khi chương trình của bạn đã đúng về mặt cú pháp, Python có thể biên dịch nó và ít nhất là bắt đầu chạy nó. Điều gì có thể trục trặc nữa?

Chương trình của tôi hoàn toàn không làm gì.

Vấn đề này thường gặp nhất khi file của bạn có chứa các hàm và lớp nhưng không thực sự kích hoạt gì để bắt đầu thực hiện tính toán. Có thể điều này được định sẵn nếu bạn chỉ nhập module này để cung cấp các lớp và hàm.

Nhưng nếu không phải vì lí do đinh sẵn, thì hãy chắc rằng bạn đã gọi một hàm để thực hiện tính toán, hoặc thực thi nó từ dấu nhắc lệnh. Ngoài ra, cũng xem thêm mục “Luồng thực hiện” tiếp theo.

Chương trình bị treo.

Nếu một chương trình dừng lại và hình như không làm gì, nó đã bị “treo”. Thường thì điều này nghĩa là nó mắc phải một vòng lặp vô hạn hoặc đệ quy vô hạn.

- Nếu có một vòng lặp cụ thể mà bạn nghi ngờ có vấn đề, hãy thêm một lệnh `print` ngay trước vòng lặp, để in ra "tien vao vong lap" và một lệnh khác ngay sau vòng lặp, in ra "thoat khoi vong lap".

  Chạy chương trình. Nếu bạn thấy được thông điệp thứ nhất mà không thấy cái thứ hai thì đã có một vòng lặp vô hạn. Xem tiếp mục "Vòng lặp vô hạn" dưới đây.

- Ở hầu hết trường hợp, đệ quy vô hạn sẽ làm cho chương trình chạy một lúc và sau đo hiện ra lỗi "RuntimeError: Maximum recursion depth exceeded". Nếu điều này xảy ra, hãy xem tiếp mục "Đệ quy vô hạn" sau đây.

  Nếu bạn không gặp phải lỗi này nhưng nghi ngờ rằng có vấn đề xảy ra với một phương thức hoặc hàm đệ quy, bạn vẫn có thể sử dụng các kĩ thuật trong mục "Đệ quy vô hạn".

- Nếu không cách nào trong số trên có tác dụng, hãy thử những hàm và phương thức khác có chứa vòng lặp hoặc đệ quy.
- Nếu cách này cũng không có tác dụng thì có thể là bạn chưa hiểu luồng thực hiện của chương trình. Hãy đọc tiếp mục "Flow of Execution" bên dưới.

Vòng lặp vô hạn

Nếu bạn nghĩ rằng bạn có một vòng lặp vô hạn và cho rằng mình đã biết được vòng lặp nào gây ra vấn đề, thì hãy thêm một lệnh `print` tại điểm cuối vòng lặp và in ra giá trị các biến trong điều kiện cùng với giá trị của điều kiện.

Chẳng hạn:

```
while x > 0 and y < 0 :
    # thao tac voi x
    # thao tac voi y

    print  "x: ", x
    print  "y: ", y
    print  "dieu kien: ", (x > 0 and y < 0)
```

Bây giờ khi chạy chương trình, bạn sẽ thấy ba dòng kết quả với mỗi lần chạy qua vòng lặp. Lần cuối cùng chạy qua vòng lặp điều kiện sẽ phải là `false`. Nếu vòng lặp tiếp tục chạy, bạn sẽ nhìn được các giá trị của `x` và `y`, và có thể hình dung được tại sao chúng không được cập nhật đúng.

Đệ quy vô hạn

Trong nhiều trường hợp, một vòng lặp đệ quy sẽ khiến chương trình chạy một lúc và sau đó báo lỗi `Maximum recursion depth exceeded`.

Nếu bạn nghi ngờ rằng một hàm hoặc phương thức nào đó gây ra đệ quy vô hạn, hãy bắt đàu kiểm tra để chắc rằng có một trường hợp cơ sở. Nói cách khác, cần phải có điều kiện nào đó để khiến cho hàm hoặc phương thức trả về mà không gọi đệ quy nữa. Nếu không, bạn cần phải nghĩ lại thuật toán và tìm ra một trường hợp cơ sở.

Nếu có một trường hợp cơ sở nhưng chương trình dường như không đạt đến đó, thì hãy thêm câu lệnh `print` vào điểm đầu của hàm hoặc phương thức để in ra các tham biến. Bây giờ khi chạy chương trình, bạn sẽ thấy một ít dòng kết quả mỗi lần hàm hoặc phương thức được gọi đến, và sẽ thấy ccasc tham biến. Nếu tham biến không thay đổi với xu hướng về trường hợp cơ sở, bạn sẽ có được nhận định về nguyên nhân tại sao.

Luồng thực hiện

Nếu bạn không chắc chắn về luồng thực hiện trong chương trình, hãy thêm các câu lệnh `print` vào điểm đầu của mỗi hàm với thông báo kiểu như "bat dau ham `foo`", trong đó `foo` là tên hàm.

Bây giờ khi chạy chương trình, nó sẽ in ra một dấu vết của mỗi hàm khi được gọi đến.

Khi chạy chương trình tôi nhận được một biệt lệ.

Nếu trong quá trình chạy có trục trặc xảy ra, Python sẽ in một thông báo trong đó có tên của biệt lệ, dòng lệnh có vấn đề, và một dò ngược.

Thông báo dò ngược nhằm chỉ định hàm đang được chạy, và hàm gọi nó, rồi hàm gọi hàm đó, và cứ như vậy. Nói cách khác, nó dò theo một dãy các lời gọi hàm để đến nơi có trục trặc. Nó cũng có chứa số thứ tự dòng trong file nơi mà những lời gọi hàm này diễn ra.

Bước đầu tiên là kiểm tra vị trí trong chương trình nơi mà lỗi xuất hiện đòng thời thử hình dung điều gì đã xảy ra. Sau đây là một số lỗi thực thi thường gặp nhất:

NameError:
: Bạn đang cố gắng dùng một biến không tồn tại trong môi trường hiện hành. Hãy nhớ rằng các biến địa phương chỉ có ý nghĩa cục bộ. Bạn không thể nhắc đến chúng từ bên ngoài hàm mà chúng được định nghĩa.

TypeError:
: Sau đây là một số lí do khả dĩ:

  - Bạn đang cố thử dùng một giá trị không đúng cách. Chẳng hạn: lấy chỉ số của một chuỗi, danh sách, hoặc bộ mà dùng kiểu số liệu không phải số nguyên.
  - Có sự bất đồng giữa các mục bên trong một chuỗi định dạng và các mục được truyền vào để chuyển đổi. Điều này có thể xảy ra khi số các mục không bằng nhau, hoặc thao tác chuyển đổi không hợp lệ.
  - Bạn đang truyền một số lượng không đúng các đối số vào trong hàm hoặc phương thức. Với phương thức, hãy xem định nghĩa cảu nó và kiểm tra rằng tham biến đầu tiên phải là `self`. Sau đó nhìn vào lời gọi phương thức; hãy chắc rằng bạn gọi phuonwg thức với đối tượng có kiểu phù hợp và đã cung cấp các đối số khác một cách đúng đắn.

KeyError:
: Bạn đang cố gắng truy cập một phần tử của từ điển bằng một khóa mà từ điển đó không có.

AttributeError:
: Bạn đang cố gắng truy cập một thuộc tính hoặc phương thức mà chúng không tồn tại. Hãy kiểm tra tên chữ! Bạn có thể dùng `dir` để liệt kê những thuộc tính hiện có.

  Nếu một AttributeError nói rằng đối tượng có kiểu `NoneType`, điều đó nghĩa là nó là `None`. Một nguyên nhân thông thường là quên không trả về giá trị từ một hàm; nếu bạn đã đến điểm cuối của hàm mà không gặp phải câu lệnh `return`, nó sẽ trả về `None`. Một lí do thường gặp khác là việc dùng kết quả từ một phương thức với danh sách, như `sort`, vốn trả về `None`.

IndexError:
: Chỉ số mà bạn đang dùng để truy cập một danh sách, chuỗi, hoặc bộ lại lớn hơn chiều dài của nó trừ đi một. Ngay trước điểm gây ra lỗi, hãy thêm vào câu lệnh `print` để hiển thị giá trị của chỉ số cùng với chiều dài của dãy. Liệu dãy này có kích thước đúng chưa? Chỉ số có đúng không?

Bộ gỡ lỗi (pdb) của Python sẽ giúp ích cho việc dò ra các biệt lệ vì chúng cho phép bạn kiểm tra trạng thái của chương trình ngay trước khi có lỗi. Bạn có thể đọc thêm về pdb ở docs.python.org/lib/module-pdb.html.

Tôi thêm vào quá nhiều lệnh `print` đến nỗi bây giờ tràn ngập kết quả đầu ra.

Một trong những vấn đề khi dùng lệnh `print` để gỡ lỗi là việc bạn có thể bị chìm trong kết quả ra. Có hai cách tiếp tục: đơn giản hóa đầu ra hoặc đơn giản hóa chương trình.

Để giản hóa đầu ra, bạn cần xóa bỏ hoặc đưa vào chú thích những dòng lệnh `print` vốn không có tác dụng, hoặc kết hợp chúng lại, hoặc sửa định dạng đầu ra để dễ hiểu hơn.

Để giản hóa chương trình, có vài cách làm được. Trước hết, hãy giảm quy mô của bài toán xuống. Chẳng hạn, nếu bạn cần tìm kiếm trong danh sách, hãy làm với danh sách *nhỏ*. Nếu chương trình nhận đầu vào từ phía người dùng, hãy cho những dữ liệu vào đơn giản mà gây ra lỗi.

Thứ hai là dọn dẹp chương trình. Hãy bỏ những đoạn mã chết và tổ chức lại chương trình để nó càng dễ đọc càng tốt. Chẳng hạn, nếu bạn nghi rằng vấn đề nằm ở một đoạn nằm sâu trong chương trình, hãy thử viết lại nó với cấu trúc đơn giản hơn. Nếu bạn nghi ngờ rằng có một hàm lớn, hãy thử chẻ nhỏ thành những hàm con và kiểm tra lần lượt.

Thông thường quá trình tìm ra trường hợp thử đơn giản nhất sẽ dẫn bạn đến điểm gây lỗi. Nếu bạn thấy được chương trình chạy được trong một trường hợp nhưng không được trong trường hợp khác, điều đó sẽ là dấu vết cho thấy điều gì đang diễn ra.

Tương tự như vậy, việc viết lại một đoạn mã có thể giúp bạn phát hiện những lỗi nhỏ. Nếu bạn thực hiện sửa đổi mà nghĩ rằng nó không ảnh hưởng gì đến chương trình, và lúc có ảnh hưởng thì đó sẽ là bài học cho bạn.

Lỗi ngữ nghĩa

Theo khía cạnh nhất định, lỗi ngữ nghĩa là thứ khó gỡ nhất, vì trình thông dịch không cung cấp thông tin gì về sự trục trặc. Chỉ có bạn mới biết rằng chương trinh cần phải thực hiện điều gì.

Bước đầu tiên là tạo lập một kết nối giữa nội dung chương trình và biểu hiện mà bạn quan sát được. Bạn cần giả thiết về điều thật sự mà chương trình đang thực hiện. Một trong những yếu tố làm việc này trở nên khó khăn là máy tính chạy quá nhanh.

Bạn sẽ thường muốn làm chậm chương trình lại ngang bằng tốc độ của người, và dùng một số bộ gỡ lỗi nếu có thể. Nhưng thời gian càn thiết để chèn thêm một vài lệnh `print` đúng chỗ thường ngắn hơn so với việc thiết lập bộ gỡ lỗi, chèn thêm và gỡ bỏ các điểm dừng, và "lần bước" tới điểm xảy ra lỗi trong chương trình.

Chương trình tôi viết không hoạt động đúng.

Bạn cần tự hỏi mình những điều sau:

- Có điều gì mà chương trình cần phải làm nhưng dường như nó không làm hay không? Hãy tìm ra đoạn mã lệnh thực hiện tính năng đó và chắc rằng nó được thực thi khi bạn nghĩ rằng lẽ ra nó phải chạy.
- Có điều gì đang diễn ra mà lẽ ra không nên có nó? Hãy tìm đoạn mã trong chương trình mà thực hiện tính năng đó rồi xem liệu nó có được thực khi trong khi đáng lẽ thì không.
- Có đoạn mã nào tạo ra một hiệu ứng mà không như bạn mong đợi không? Hãy chắc rằng bạn hiểu được đoạn mã nghi vấn, đặc biệt khi nó liên quan đến lời gọi các hàm hoặc phương thức trong module Python khác. Hãy đọc các tài liệu về hàm mà bạn đã gọi. Thử dùng chúng

bằng cách viết các trường hợp kiểm tra đơn giản và xem xét kết quả.

Để lập trình, bạn phải có một mô hình tưởng tượng về cách thức hoạt động của chương trình. Nếu bạn viết một chương trình mà không thực hiện đúng việc bạn mong đợi, thì thường là vấn đề không nằm ở chương trình; nó nằm ở mô hình tưởng tượng của bạn.

Cách tốt nhất để sửa mô hình tưởng tượng cho đúng là chia chương trình thành những bộ phận (thường là các hàm và phương thức) rồi kiểm tra chạy thử từng bộ phận một cách độc lập. Một khi bạn thấy sự khác biệt giữa mô hình và thực tế, bạn sẽ có thể giải quyết vấn đề.

Tất nhiên, bạn cần phải xây dựng và chạy thử các bộ phận song song với việc phát triển chương trình. Nếu bạn gặp vướng mắc, hẳn chỉ có một phần rất nhỏ những mã lệnh mới đưa vào mà bạn không chắc rằng nó đúng.

Tôi có một biểu thức lớn và gai góc mà chẳng hoạt động theo sự mong đợi.

Việc viết những biểu thức phức tạp cũng tốt miễn là chúng dễ đọc, nhưng chúng có thể làm việc gỡ lỗi gặp khó khăn. Thông thường nên chẻ nhỏ một biểu thức thành một loạt các lệnh gán cho những biến tạm thời.

Chẳng hạn:

```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

Đoạn này có thể được viết lại thành:

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

Dạng mã lệnh chi tiết thì dễ đọc hơn vì tên biến cho ta bản thân đã giúp giải thích rõ thêm, và cũng dễ gỡ lỗi hơn vì bạn có thể kiểm tra kiểu của những biến trung gian cùng việc hiển thị giá trị của chúng.

Một vấn đề khác có thể xảy ra với những biểu thức lớn là thứ tự thực hiện phép tính có thể không như bạn mong muốn. Chẳng hạn, nếu bạn dịch biểu thức x / 2π sang ngôn ngữ Python, có thể bạn đã viết:

```
y = x / 2 * math.pi
```

Điều này không đúng vì các phép nhân và chia có cùng thứ tự ưu tiên và được lượng giá từ trái sang phải. Vì vậy biểu thức này sẽ tính *x*π / 2.

Một cách hay để gỡ lỗi biểu thức là thêm vào những cặp ngoặc đơn để giúp cho thứ tự lượng giá được rõ ràng:

```
y = x / (2 * math.pi)
```

Mỗi khi bạn không nắm vững thứ tự ước lượng, hãy dùng cặp ngoặc đơn. Chúng không chỉ giúp chương trình đúng đắn hơn (theo nghĩa thực hiện công việc bạn mong đợi), mà còn giúp người khác dễ đọc hơn mà không phải ghi nhớ quy luật thứ tự ưu tiên.

Tôi có một hàm hoặc phương thức không trả về giá trị được mong đợi.

Với lệnh `return` kèm theo một biểu thức phức tạp, bạn không có cơ hội in ra giá trị trước khi được rả về. Một lần nữa, hãy dùng biến tạm thời. Chẳng hạn, thay vì:

```
return self.hands[i].removeMatches()
```

bạn có thể viết:

```
count = self.hands[i].removeMatches()
return count
```

Bây giờ bạn đã có cơ hội hiển thị giá trị của `count` trước khi trả về.

Thật sự tôi rất, rất vướng mắc và cần được giúp đỡ.

Trước hết, hãy thử rời khỏi máy tính trong vài phút. Máy tính phát ra sóng từ gây ảnh hưởng đến não, với các triệu chứng sau:

- Cáu giận.
- Tin tưởng vào lực siêu nhiên ("máy tính này ghét tôi") và những ảo tưởng ("chương trình chỉ chạy khi tôi đội ngược mũ").
- Lập trình bước ngẫu nhiên (nỗ lực lập trình bằng cách viết tất cả các trường hợp chương trình có thể có và chọn ra một phiên bản hoạt động đúng).

Nếu bạn tự thấy mình mắc phải một trong số các triệu chứng trên, hãy đứng dậy và đi dạo. Khi đã tĩnh tâm hẳn, hãy nghĩ lại chương trình. Nó đang làm điều gì? Đâu là các nguyên nhân gây ra biểu hiện đó? Lần cuối cùng chương trình cọn chạy được là lúc nào, và sau đó bạn thực hiện những điều gì?

Đôi khi phát hiện lỗi chỉ là vấn đề thời gian. Tôi thường tìm thấy lỗi trong lúc rời xa khỏi máy tính và để trí óc khuây khỏa. Một số nơi tốt nhất để thoát khỏi máy gồm có trên tàu, khi đi tắm, và trước khi đi ngủ.

Không, tôi thật sự muốn giúp đỡ.

Điều đó xảy ra. Ngay cả những lập trình viên giỏi nhất đôi lúc cũng bị bí. Đôi khi bạn làm một chương trình lâu quá đến nỗi không thể phát hiện ra lỗi. Tìm một người có góc nhìn khác chính là điều cần thiết.

Trước khi yêu cầu giúp đỡ, bạn hãy chuẩn bị kĩ. Chương trình phải càng đơn giản càng tốt, và hãy phân tích trên dữ liệu đầu vào nhỏ nhất có thể gây lỗi. Bạn cần có các lệnh `print` ở những vị trí thích hợp (và kết quả đầu ra phải dễ hiểu). Bạn cần hiểu rõ vấn đề để có thể diễn đạt nó một cách ngắn gọn.

Khi đưa người đến giúp, hãy chắc chắn rằng bạn cung cấp đủ thông tin mà họ cần:

- Nếu có thông báo lỗi, thông báo đó là gì và nó chỉ định phần nào trong chương trình?
- Việc cuối cùng mà bạn thao tác trước khi lỗi này xảy ra là gì? Những dòng lệnh nào bạn vừa mới viết gần đây nhất, hay trường hợp chạy thử gần đây nhất mới bị thất bại là gì?
- Bạn đã thử những biện pháp gì rồi, và thu hoạch được gì?

Khi bạn tìm thấy lỗi, hãy nghĩ trong giây lát xem bằng cách nào có thể giúp bạn tìm ra nó nhanh hơn không. Lần tiếp theo khi bạn thấy điều tương tự, có thể bạn sẽ chóng phát hiện ra lỗi hơn.

Nhớ rằng mục tiêu không chỉ có làm cho chương trình chạy được. Mục tiêu là học cách làm cho chương trình chạy được.

Phụ lục: Lumpy

Trở về Mục lục cuốn sách

Xuyên suốt cuốn sách, tôi đã dùng những biểu đồ nhằm thể hiện trạng thái của các chương trình đang chạy.

Ở Mục 2.2, chúng ta dùng một biểu đồ trạng thái để cho thấy tên và giá trị của các biến. Trong Mục 3.10 tôi có giới thiệu một biểu đồ ngăn xếp, trong đó thể hiện mỗi khung cho một lần gọi hàm. Từng khung đều thể hiện các tham số và biến địa phương của hàm hoặc phương thức. Những biểu đồ ngăn xếp cho hàm đệ quy có ở các Mục 5.9 và 6.5.

Mục 10.2 cho thấy dáng vẻ của một danh sách trong biểu đồ trạng thái, Mục 11.4 cho thấy từ điển, và Mục 12.6 thể hiện hai cách biểu diễn một bộ.

Mục 15.2 giới thiệu biểu đồ đối tượng, trong đó cho thấy trạng thái của các thuộc tính của đối tượng, và thuộc tính của những thuộc tính đó, và cứ thư vậy. Mục 15.3 có biểu đồ đối tượng cho Rectangle và các Point đi kèm trong đó. Mục 16.1 cho thấy trạng thái của một đối tượng Time. Mục 18.2 có một biểu đồ gồm một đối tượng lớp và một thực thể, từng cái lại có thuộc tính riêng của mình.

Sau cùng, Mục 18.8 giới thiệu biểu đồ lớp, trong đó cho thấy những lớp hợp thành chương trình, cùng những mối quan hệ giữa chúng.

Những biểu đồ này đều dựa theo Ngôn ngữ mô hình hóa thống nhất (Unified Modeling Language, UML), vốn là một ngôn ngữ đồ thị được chuẩn hóa, dành cho các kĩ sư phần mềm, phục vụ việc trao đổi thông tin về thiết kế chương trình, đặc biệt là những chương trình hướng đối tượng.

UML là một ngôn ngữ phong phú với nhiều loại biểu đồ thể hiện nhiều loại quan hệ giữa đối tượng và các lớp. Những biểu đồ trình bày trong sách này chỉ là một phần nhỏ của ngôn ngữ trên, nhưng chính là phần ngôn ngữ hay được dùng nhất trên thực tế.

Mục đích của phụ lục này là điểm lại những biểu đồ đã trình bày từ các chương trước, đồng thời giới thiệu Lumpy. Lumpy là chữ viết tắt của "UML in Python," sau khi đảo lại chữ cái; công cụ này là một phần của Swampy, mà bạn đã cài đặt khi thực hiện nghiên cứu cụ thể trong Chương 4 hoặc Chương 19, hay nếu bạn đã làm Bài tập 4,

Lumpy sử dụng module `inspect` của Python để kiểm tra trạng thái của một chương trình đang chạy và phát sinh ra biểu đồ đối tượng (bao gồm cả biểu đồ ngăn xếp) và biểu đồ lớp.

Biểu đồ trạng thái

Hình 1: Biểu đồ trạng thái được Lumpy tạo ra.

Sau đây là một ví dụ có dùng Lumpy để tạo nên một biểu đồ trạng thái.

```
from swampy.Lumpy import Lumpy
```

```
lumpy = Lumpy()
lumpy.make_reference()

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

lumpy.object_diagram()
```

Dòng thứ nhất để nhập lớp Lumpy từ `swampy.Lumpy`. Nếu bạn không cài Swampy theo hình thức một gói phần mềm, thì hãy đảm bảo rằng các file Swampy đều nằm trong đường dẫn tìm kiếm của Python và dùng câu lệnh `import` sau đây thay cho lệnh trong đoạn mã trên:

```
from Lumpy import Lumpy
```

Các dòng tiếp theo tạo nên một đối tượng `Lumpy` và lập một điểm "tham chiếu"; tại đó Lumpy ghi lại những đối tượng mà đã được định nghĩa cho đến điểm này.

Sau đó ta định nghĩa các biến mới rồi kích hoạt `object_diagram`; lệnh này vẽ nên những đối tượng đã được định nghĩa kể từ điểm tham chiếu, mà trong trường hợp này gồm có `message`, `n` và `pi`.

Hình 1 cho thấy kết quả. Kiểu dáng biểu đồ này khác với cái mà tôi đã trình bày trước đây. Chẳng hạn, từng tham chiếu được biểu diễn bằng một vòng tròn viết cạnh tên biến và một đoạn thẳng chỉ đến giá trị. Đồng thời những chuỗi kí tự dài đều được lược bớt. Song những thông tin mà biểu đồ truyền đạt thì vẫn như vậy.

Tên các biến được viết trong một khung có nhãn hiệu `<module>`, vốn để chỉ rằng đây là các biến ở cấp độ module, còn gọi là biến toàn cục.

Bạn có thể tải ví dụ này về từ http://thinkpython.com/code/lumpy_demo1.py. Hãy thử thêm vào một số lệnh gán rồi xem biểu đồ trông như thế nào.

Biểu đồ ngăn xếp

Hình 2: Biểu đồ ngăn xếp

Sau đây là một ví dụ dùng Lumpy để phát sinh một biểu đồ ngăn xếp. Bạn có thể tải về từ địa chỉ http://thinkpython.com/code/lumpy_demo2.py.

```
from swampy.Lumpy import Lumpy

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

lumpy = Lumpy()
lumpy.make_reference()
```

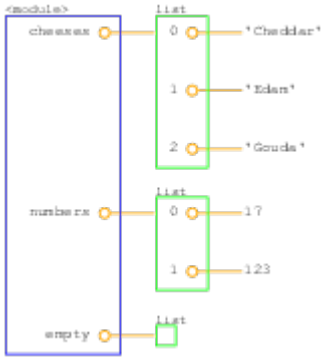
```
countdown(3)
```

Hình 2 biểu diễn kết quả. Từng khung được biểu diễn bằng một hộp với tên hàm viết ở ngoài và tên biến ở trong. Vì hàm này có tính đệ quy, nên mỗi tầng đệ quy chỉ có một khung.

Hãy nhớ rằng biểu đồ ngăn xếp cho thấy trạng thái của chương trình ở một điểm cụ thể trong quá trình thực thi. Để thu được biểu đồ mong muốn, đôi khi bạn phải suy nghĩ xem nên kích hoạt `object_diagram` ở đâu.

Trong trường hợp này tôi kích hoạt `object_diagram` sau khi thực thi trường hợp cơ sở của phép đệ quy. Bằng cách này, biểu đồ ngăn xếp sẽ cho thấy từng tầng một của phép đệ quy. Bạn có thể gọi `object_diagram` nhiều lần để nhận được một loạt các hình ảnh trạng thái về quá trình thực thi của chương trình.

Biểu đồ đối tượng

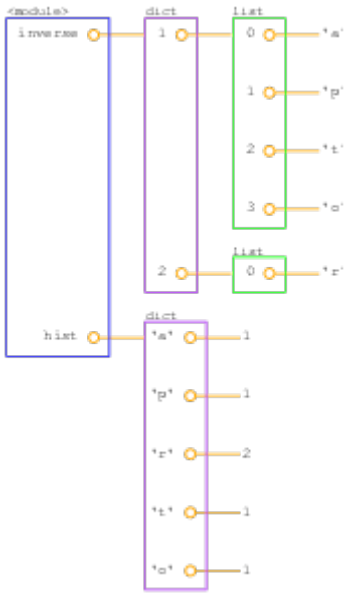


Hình 3: Biểu đồ đối tượng

Ví dụ này sẽ phát sinh ra một biểu đồ đối tượng cho thấy những danh sách ở Mục 10.1. Bạn có thể tải mã lệnh về từ http://thinkpython.com/code/lumpy_demo3.py.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

cheeses = ['Cheddar', 'Edam', 'Gouda']
numbers = [17, 123]
empty = []

lumpy.object_diagram()
```

Hình 3 cho thấy kết quả. Các danh sách được biểu thị bởi một hộp cho thấy rõ ràng các chỉ số được ánh xạ đến các phần tử. Cách biểu diễn này hơi làm người xem lạc hướng, vì thực ra các chỉ số không phải thuộc về danh sách; song dù sao tôi nghĩ rằng cách này dễ đọc biểu đồ hơn. Danh sách rỗng được biểu diễn bằng một hộp trống không.

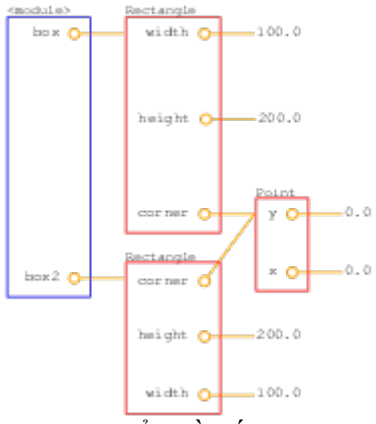


Hình 4: Biểu đồ đối tượng

Và sau đây là một ví dụ cho thấy các từ điển trong Mục 11.4. Bạn có thể tải về từ địa chỉ http://thinkpython.com/code/lumpy_demo4.py.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

hist = histogram('parrot')
inverse = invert_dict(hist)

lumpy.object_diagram()
```

Hình 4 cho thấy kết quả. `hist` là một từ điển để ánh xạ từ những kí tự (chuỗi chỉ có 1 chữ cái) đến những số nguyên; `inverse` ánh xạ từ các số nguyên đến danh sách các chuỗi.



Hình 5: Biểu đồ đối tượng

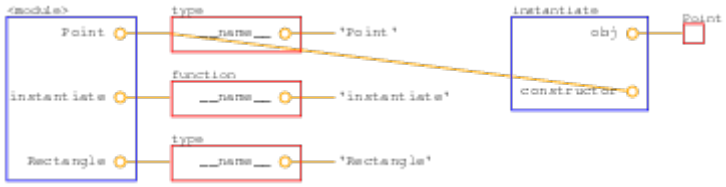
Ví dụ này phát sinh ra một biểu đồ đối tượng cho các đối tượng Point và Rectangle, như ở Mục 15.6. Bạn cũng có thể tải nó về từ http://thinkpython.com/code/lumpy_demo5.py.

```
import copy
from swampy.Lumpy import Lumpy
```

```
lumpy = Lumpy()
lumpy.make_reference()

box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

box2 = copy.copy(box)

lumpy.object_diagram()
```

Hình 5 cho thấy kết quả. `copy.copy` thực hiện "sao chép nông", vì vậy cả `box` lẫn `box2` đều có `width` và `height` riêng, nhưng chúng cùng chia sẻ mối đối tượng Point đi kèm. Hình thức chia sẻ như vậy thường là ổn thỏa đối với các đối tượng bất khả chuyển; song với kiểu khả chuyển, điều này dễ gây nên lỗi.

Các đối tượng hàm và lớp



Hình 6. Biểu đồ đối tượng

Khi dùng Lumpy để lập nên các biểu đồ đối tượng, tôi thường định nghĩa các hàm và lớp trước khi đặt điểm tham chiếu. Bằng cách này, các đối tượng hàm và lớp không xuất hiện trên biểu đồ.

Nhưng nếu bạn truyền các hàm và lớp làm tham số, thì có thể bạn sẽ muốn chúng xuất hiện. Ví dụ sau đây sẽ cho thấy rằng khi chúng xuất hiện thì biểu đồ sẽ như thế nào; bạn có thể tải mã lệnh về từ http://thinkpython.com/code/lumpy_demo6.py.

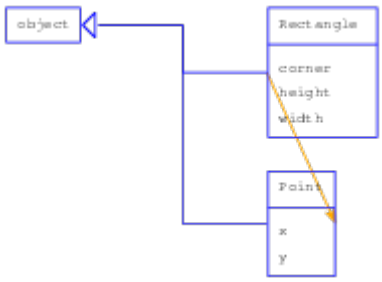
```
import copy
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

class Point(object):
    """Represents a point in 2-D space."""

class Rectangle(object):
    """Represents a rectangle."""

def instantiate(constructor):
    """Instantiates a new object."""
    obj = constructor()
    lumpy.object_diagram()
    return obj

point = instantiate(Point)
```

Hình 6 biểu thị kết quả. Vì ta kích hoạt `object_diagram` bên trong một hàm, ta nhận được một biểu đồ ngăn xếp với một khung dành cho các biến ở cấp độ module và dành cho sự kích hoạt `instantiate`.

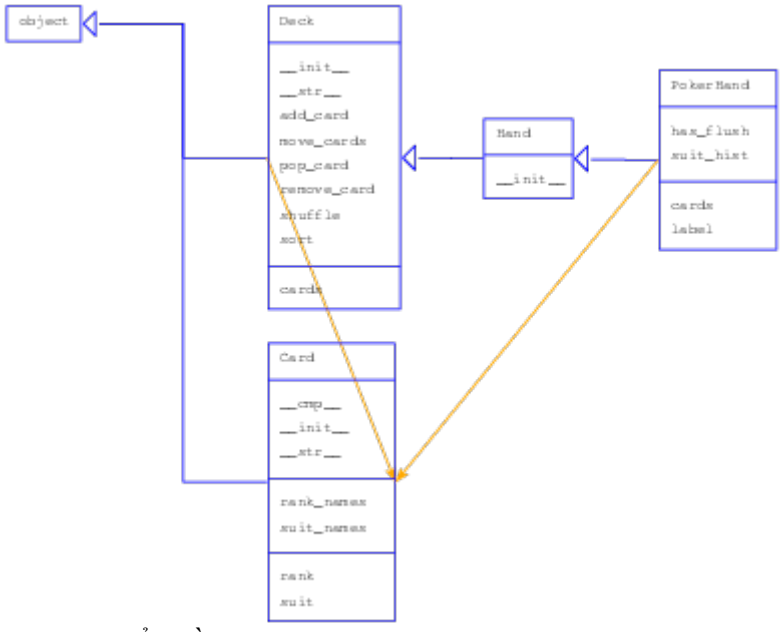
Ở cấp độ module, cả `Point` lẫn `Rectangle` đều tham chiếu đến các đối tượng lớp (mà có kiểu là `type`); `instantiate` tham chiếu đến một đối tượng hàm.

Biểu đồ này có thể làm rõ hai điều dễ gây nhầm lẫn: (1) sự khác biệt giữa đối tượng lớp là `Point`, với thực thể của Point là `obj`, và (2) sự khác biệt giữa đối tượng hàm tạo ra khi định nghĩa `instantiate`, và khung tạo ra khi nó được gọi đến.

Biểu đồ lớp



Hình 7: Biểu đồ lớp



Hình 8: Biểu đồ lớp.

Mặc dù tôi phân biệt giữa biểu đồ trạng thái, biểu đồ ngăn xếp và biểu đồ đối tượng, song chúng đa phần đều giống nhau ở chỗ đều cho thấy trạng thái của một chương trình đang chạy tại một thời điểm.

Biểu đồ lớp thì khác. Chúng cho thấy các lớp hợp thành một chương trình cũng như những mối quan hệ giữa chúng. Chúng không thay đổi theo thời gian vì chúng mô tả chương trình như một tổng thể chứ không phải tại một thời điểm cụ thể nào đó. Chẳng hạn, nếu một thực thể của Lớp A nói chung chứa một tham chiếu đến một thực thể của Lớp B, thì ta nói rằng có một "mối quan hệ HAS-A" giữa hai lớp này.

Sau đây là một ví dụ biểu diễn mối quan hệ HAS-A. Bạn có thể tải nó về

từ http://thinkpython.com/code/lumpy_demo7.py.

```
from swampy.Lumpy import Lumpy

lumpy = Lumpy()
lumpy.make_reference()

box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0

lumpy.class_diagram()
```

Hình 7 biểu diễn kết quả. Từng lớp đều được biểu diễn bằng một hình hộp trong đó chứa tên của lớp, mọi phương thức mà lớp đó cung cấp, mọi biến của lớp, và mọi biến thực thể. Trong ví dụ này, cả `Rectangle` lẫn `Point` đều có biến thực thể, nhưng không có phương thức hay biến lớp.

Mũi tên chạy từ `Rectangle` đến `Point` cho thấy rằng các đối tượng Rectangle chứa một Point kèm theo. Ngoài ra, cả `Rectangle` lẫn `Point` đều thừa kế từ `object`, vốn được biểu diễn bằng một mũi tên có đầu tam giác trong biểu đồ này.

Sau đây là một ví dụ phức tạp hơn có dùng lời giải của tôi cho Bài tập 6. Bạn có thể tải mã lệnh về từ http://thinkpython.com/code/lumpy_demo8.py; bạn cũng sẽ phải cần đến http://thinkpython.com/code/PokerHand.py.

```
from swampy.Lumpy import Lumpy

from PokerHand import *

lumpy = Lumpy()
lumpy.make_reference()

deck = Deck()
hand = PokerHand()
deck.move_cards(hand, 7)

lumpy.class_diagram()
```

Hình 8 cho thấy kết quả. `PokerHand` thừa kế từ `Hand`, mà bản thân nó lại thừa kế từ `Deck`. Cả `Deck` lẫn `PokerHand` đều có Card.

Biểu đồ này không cho thấy được rằng `Hand` cũng có các lá bài, vì trong chương trình không có thực thể nào của Hand. Ví dụ này thể hiện một nhược điểm của Lumpy; nó chỉ biết được những thuộc tính và mối quan hệ HAS-A của những đối tượng nào được tạo nên (được "thực thể hóa").