Bài chỉ dẫn Python

Bài chỉ dẫn Python

Guido van Rossum

Python Software Foundation
Thư điện tử: docs@python.org

Fred L. Drake, Jr., biên tập viên

do Nguyễn Thành Nam, Lê Hồng Việt và Lương Trọng Đức của nhóm Python cho người Việt dịch

Phiên bản 2.5
Ngày 19, tháng 09, năm 2006

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách đề nghị thay đổi.

Bài chỉ dẫn Python

Lời tựa

Tóm tắt:

Python là một ngôn ngữ dễ học, và mạnh mẽ. Nó có những cấu trúc dữ liệu cấp cao hiệu quả và hướng lập trình đối tượng đơn giản. Cú pháp tao nhã và kiểu dữ liệu động của Python, cùng với bản chất thông dịch biến nó thành một ngôn ngữ bậc nhất để viết kịch bản (scripting) và phát triển ứng dụng nhanh trong nhiều lĩnh vực và trên hầu hết mọi hệ thống.

Trình thông dịch Python và bộ thư viện chuẩn đầy đủ được cung cấp miễn phí ở dạng nguồn hoặc nhị phân cho mọi hệ thống chính từ trang chủ Python, http://www.python.org/, và có thể được phát tán tùy thích. Trang chủ đó cũng phân phối và liên kết nhiều mô-đun Python khác, các chương trình và công cụ, cũng như các tài liệu thêm.

Trình thông dịch Python có thể được mở rộng dễ dàng với những chức năng và kiểu dữ liệu được viết trong C hoặc C++ (hoặc ngôn ngữ nào đó có thể gọi được từ C). Python cũng phù hợp dùng làm ngôn ngữ mở rộng cho các ứng dụng mà người dùng có thể cải biến.

Bài chỉ dẫn này giới thiệu với người đọc bằng một cách dễ hiểu những khái niệm cơ bản và các tính năng của ngôn ngữ và hệ thống Python. Để tận dụng tốt nhất chỉ dẫn này, bạn nên có trình thông dịch Python sẵn sàng để thực tập. Nhưng bạn cũng không nhất thiết cần đến nó để đọc tài liệu này vì mọi ví dụ đều ngắn và dễ hiểu cả.

Để tìm hiểu thêm về các mô-đun và đối tượng chuẩn, xem qua tài liệu *Tham khảo thư viện Python* . *Sổ tay tham khảo Python* chứa định nghĩa ngôn ngữ chính quy hơn. Để viết các phần mở rộng bằng C hoặc C++, bạn nên đọc *Mở rộng và Nhúng trình thông dịch Python* và *Tham khảo API cho Python/C*. Và cũng có nhiều sách khác nói sâu hơn về Python.

Bài chỉ dẫn này không nhằm vào việc nói về mọi tính năng, hoặc thậm chí là

mọi tính năng hay dùng. Thay vào đó, nó giới thiệu nhiều chức năng đáng lưu ý của Python và đem lại cho bạn một cách nhìn về kiểu cách và hương vị của ngôn ngữ này. Sau khi đọc xong, bạn sẽ có thể đọc và viết các mô-đun và chương trình Python, và bạn sẽ sẵn sàng tìm hiểu tiếp về những mô-đun Python khác được nhắc đến trong *Tham khảo thư viện Python*.

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liêu này... về cách đề nghị thay đổi.

Bài chỉ dẫn Python

Mục lục

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách đề nghị thay đổi.

Bài chỉ dẫn Python

1. Khai vị

Nếu bạn làm việc nhiều với máy vi tính, một lúc nào đó bạn sẽ nhận thấy bạn muốn tự động hóa một số việc. Ví dụ, bạn muốn thực hiện một phép tìm kiếm và thay thế với nhiều tập tin văn bản, hoặc đổi tên và sắp xếp một loạt các tập tin ảnh theo một cách phức tạp. Có thể bạn muốn viết cơ sở dữ liệu tùy biến nho nhỏ, hoặc một ứng dụng với giao diện đồ họa đặc biệt, hay một trò chơi đơn giản.

Nếu bạn là một người chuyên viết phần mềm, bạn có thể làm việc với nhiều thư viện C/C++/Java nhưng bạn nhận thấy thường lặp đi lặp lại việc viết/biên dịch/thử/biên dịch là quá tốn thời gian. Có thể bạn viết một bộ các thử nghiệm cho các thư viện ấy và nhận ra rằng viết mã lệnh để thử nghiệm là một việc chán ngấy. Hoặc có thể bạn viết một chương trình cần sử dụng một ngôn ngữ mở rộng, và bạn không muốn thiết kế, xây dựng cả một ngôn ngữ mới cho ứng dụng của mình.

Python chính là ngôn ngữ lập trình bạn cần.

Bạn có thể viết một kịch bản UNIX hoặc một bó lệnh (batch file) Windows cho công việc kiểu này thế nhưng, ngôn ngữ kịch bản chỉ tốt cho việc chuyển các tập tin lòng vòng và sửa đổi các dữ liệu văn bản, nó không thích hợp cho một ứng dụng với giao diện đồ họa hoặc một trò chơi. Bạn cần viết một chương trình bằng C/C++/Java, nhưng nó có thể tiêu tốn nhiều thời gian cho việc phát triển thậm chí từ bản nháp đầu tiên của chương trình. Sử dụng Python đơn giản hơn, chạy được cả trên Windows, MacOS X, và các hệ điều hành UNIX , và nó cũng giúp bạnh hoàn thành công việc nhanh hơn.

Sử dụng Python thì đơn giản, nhưng nó là một ngôn ngữ lập trình thực thụ, cung cấp nhiều cấu trúc hơn và hỗ trợ các chương trình lớn hơn so với các ngôn ngữ kịch bản hoặc bó lệnh Windows. Mặt khác, Python cũng hỗ trợ nhiều phép kiểm tra lỗi hơn C, và, là một *ngôn ngữ bậc-rất-cao*, nó có sẵn các kiểu dữ liệu cấp cao, như các mảng và các từ điển linh hoạt. Chính vì nhiều kiểu dữ liệu tổng quát của nó Python được ứng dụng rộng rãi hơn Awk hoặc thậm chí là Perl trong nhiều loại công việc khác nhau, do đó có nhiều việc làm bằng Python cũng dễ dàng như làm bằng các ngôn ngữ khác.

Python cho phép bạn chia nhỏ chương trình của mình ra thành các mô-đun để có thể sử dụng lại trong các chương trình Python khác. Nó có sẵn rất nhiều các mô-đun chuẩn để bạn có thể sử dụng làm cơ sở cho chương trình của mình -- hoặc như các ví dụ để bắt đầu học lập trình bằng Python. Một vài mô-đun trong số chúng cung cấp các chức năng như tập tin I/O (vào/ra), các lệnh gọi hàm hệ thống, các socket, và thậm chí các giao tiếp với các công cụ giao diện đồ họa như Tk.

Python là một ngôn ngữ thông dịch, điều đó giúp bạn tiết kiệm thời gian trong quá trình phát triển chương trình vì việc biên dịch hay liên kết là không cần thiết. Bộ thông dịch có thể được dùng một cách tương tác, làm cho việc thử nghiệm các tính năng của ngôn ngữ trở nên dễ dàng, viết các chương trình bỏ đi, hoặc thử các chức năng trong việc phát triển chương trình từ dưới lên. Nó cũng là một máy tính cầm tay tiện lợi.

Python cho phép viết các chương trình nhỏ gọn và dễ hiểu. Các chương trình viết bằng Python thường ngắn hơn so với các chương trình viết bằng C, C++ hoặc Java, vì nhiều lý do:

- các kiểu dữ liệu cao cấp cho phép bạn thực hiện nhanh các thao tác phức tạp chỉ với một lệnh đơn giản;
- phát biểu lệnh được nhóm lại bằng khoảng cách thụt đầu dòng thay vì đóng mở với các dấu ngoặc;
- không cần khai báo biến hoặc tham số trước khi sử dụng.

Python có tính *mở rộng*: nếu bạn biết lập trình C thì rất dễ để bổ sung các hàm có sẵn hoặc mô-đun vào bộ thông dịch, cũng như việc thực hiện các thao tác quan trọng ở tốc độ tối đa, hoặc liên kết các chương trình Python với các thư viện chỉ được cung cấp dưới dạng nhị phân (ví dụ như các thư viện đồ họa của một vài nhà sản xuất). Một khi bạn đã thực sự móc nối, bạn có thể liên kết bộ thông dịch Python vào trong các ứng dụng viết bằng C và sử dụng nó như một tính năng mở rộng hoặc một ngôn ngữ lệnh cho ứng dụng đó.

Cũng xin nói luôn, tên của ngôn ngữ này được đặt sau khi BBC phát chương trình ``Monty Python's Flying Circus'' và nó không có liên quan gì với những loài bò sát bẩn thỉu. Những tham khảo mang tính trào phùng tới Monty Python trong tài liệu không chỉ được cho phép, mà còn được cổ vũ.

Bây giờ khi tất cả các bạn đã bị kích thích về Python, bạn sẽ muốn khám phá nó kỹ hơn. Cách học một ngôn ngữ tốt nhất là hãy sử dụng nó, bài chỉ dẫn này mời gọi bạn hãy vừa thử trình thông dịch Python khi bạn vừa đọc.

Trong chương tiếp theo, các phương thức sử dụng bộ thông dịch sẽ được giải thích. Điều này không đơn thuần là thông tin, nhưng còn là cơ bản cho việc thử các ví dụ được trình bày về sau.

Phần tự học còn lại sẽ giới thiệu các tính năng khác nhau của ngôn ngữ Python và hệ thống thông qua các ví dụ, bắt đầu với các biểu thức đơn giản, các câu lệnh và các kiểu dữ liệu, đi qua các hàm và các mô-đun, và kết thúc là tiếp cận với các khái niệm cao cấp như biệt lệ và các lớp do người dùng tự định nghĩa.

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách đề nghị thay đổi.

Bài chỉ dẫn Python

2. Sử dụng trình thông dịch Python

2.1 Chạy trình thông dịch

Bộ thông dịch Python thường được cài đặt là /usr/local/bin/python trên các máy tính đã cài đặt sẵn; bổ sung /usr/local/bin vào đường dẫn tìm kiếm của vỏ (shell) UNIX sẽ giúp khởi động nó từ mọi nơi bằng một lệnh đơn giản

```
python
```

trong vỏ. Vì nơi mà trình thông dịch được cài đặt là một tùy chọn nên khi cài đặt trình thông dịch có thể sẽ được đặt ở một nơi khác; hãy hỏi quản trị hệ thống của bạn. (ví dụ /usr/local/python cũng là một vị trí hay được dùng để cài.)

Trên các máy tính dùng Windows, Python thường được cài đặt vào C:\Python24, dù vậy bạn vẫn có thể thay đổi vị trí cài đặt khi chạy chương trình cài đặt. Để bổ sung thư mục này vào đường dẫn, bạn có thể gõ lệnh sau lên dấu nhắc lệnh trong cửa sổ DOS:

```
set path=%path%;C:\python24
```

Gõ một ký tự kết thúc tập tin (end-of-file character) (`Control-D` trên UNIX, `Control-Z` trên Windows) tại dấu nhắc của bộ thông dịch sẽ thoát khỏi bộ thông dịch và trả về trạng thái kết thúc chương trình là 0 (không) cho hệ điều hành, bạn cũng có thể thoát khỏi bộ thông dịch bằng các lệnh sau: "`import sys; sys.exit()`".

Tính năng soạn thảo theo dòng của bộ thông dịch thường không phức tạp lắm. Trên UNIX, bất cứ ai đã cài đặt bộ thông dịch đều có thể bật chế độ hỗ trợ cho thư viện GNU readline, điều này sẽ bổ sung tính năng soạn thảo tương tác tốt hơn cũng như các tính năng lịch sử lệnh. Có thể kiểm tra việc hỗ trợ tính năng soạn thảo dòng bằng cách nhấn Control-P tại dấu nhắc đầu tiên của Python. Nếu có tiếng bíp, bộ thông dịch của bạn có hỗ trợ soạn thảo dòng; xem phụ lục A để biết về các phím. Nếu không có gì xảy ra, hoặc kí tự `P` hiện lên, thì tính năng soạn thảo dòng không được hỗ trợ; bạn chỉ việc dùng phím lùi (backspace) để xóa kí tự hiện ra.

Bộ thông dịch Python hoạt động khá giống với vỏ UNIX : khi được gọi với một đầu nhập chuẩn đã kết nối với một thiết bị đầu cuối (tty device), nó đọc và thực hiện các lệnh một cách tương tác; khi được gọi với tham số là một tên tập tin hoặc với đầu vào chuẩn là một tập tin, nó đọc và thực hiện *kịch bản* chứa trong tập đó.

Một cách nữa để khởi động bộ thông dịch là lệnh "**python -c** *command* [arg] ...", sẽ thực thi một hoặc nhiều câu lệnh trong *command*, giống như tùy chọn **-c** của vỏ. Vì các câu lệnh của Python thường chứa các khoảng trống hoặc các kí tự đặc biệt, chọn lựa an toàn nhất là bao *command* bằng dấu nháy kép (").

Một số mô-đun cũng có thể được dùng như kịch bản. Chúng có thể được gọi bằng cách sử dụng cú pháp "**python -m** *module* [arg] ...", lệnh này sẽ thực hiện tập tin nguồn *module* như khi bạn chỉ ra tên tập tin và đường dẫn đầy đủ trên dòng lệnh.

Xin lưu ý rằng có sự khác biệt giữa "python file" và "python <file". Trong trường hợp sau, các yêu cầu vào (input request) từ chương trình, ví dụ như lời gọi tới input() và raw_input(), được cung cấp từ *file*. Vì tập tin này đã được đọc đến cuối bởi trình phân tích (parser) trước khi chương trình thực thi, chương trình sẽ gặp phải cuối tập tin (end-of-file) ngay lập tức. Trong trường hợp đầu (là cách bạn sẽ hay dùng) các yêu cầu vào được cung cấp từ bất kỳ tập tin hoặc thiết bị nào được kết nối vào đầu vào chuẩn của trình thông dịch Python.

Khi tập tin kịch bản (script file) được sử dụng, đôi khi sẽ rất hữu dụng nếu có thể chạy chương trình và chuyển sang chế độ tương tác ngay sau đó. Điều này thực hiện được bằng cách truyền **-i** trước script (python -i script). (Phương pháp này không hoạt động nếu chương trình được đọc từ đầu vào chuẩn, lí do của chuyện này đã được giải thích trong đoạn trước.)

2.1.1 Truyền thông số

Bộ thông dịch nhận biết tên chương trình và các tham số khác được truyền vào chương trình trong biến sys.argv, dưới dạng một danh sách các chuỗi. Độ dài tối thiểu là một; khi không có kịch bản hoặc thông số truyền vào, sys.argv[0] là một chuỗi rỗng. Khi tên kịch bản được truyền vào là '-' (có nghĩa là đầu vào chuẩn), sys.argv[0] được gán thành '-'. Khi **-c** *command* được sử dụng, sys.argv[0] được gán thành '-c'. Khi **-m** *module* được sử dụng, sys.argv[0] được gán là tên đầy đủ của mô-đun đã nạp. Các tùy chọn sau **-c** *command* hoặc **-m** *module* không được sử dụng bởi trình thông dịch Python mà truyền vào sys.argv để cho *command* hay *module* xử lý.

2.1.2 Chế độ tương tác

Khi các lệnh được đọc từ một tty, trình thông dịch được xem là đang trong *chế độ tương tác*. Trong chế độ này nó nhắc lệnh tiếp theo với *dấu nhắc chính (primary prompt)*, thường là ba dấu lớn hơn (">>> "); với các dòng tiếp nối (continuation line), nó sẽ nhắc với *dấu nhắc thứ (secondary prompt)*, mặc định là ba dấu chấm ("... "). Bộ thông dịch sẽ in một thông báo chào mừng, số hiệu phiên bản và thông báo bản quyền trước khi hiện dấu nhắc:

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Các dòng nối tiếp được dùng khi nhập vào các cấu trúc nhiều dòng. Hãy xem ví dụ dưới, chú ý câu lệnh `if` :

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

2.2 Trình thông dịch và môi trường của nó

2.2.1 Xử lý lỗi

Khi xảy ra một lỗi, bộ dịch in ra thông báo lỗi và lần ngược ngăn xếp (stack trace). Trong chế độ tương tác, nó sẽ trả lại dấu nhắc chính; khi đầu vào là một tập tin, nó sẽ thoát với mã lỗi khác 0 sau khi in ra lần ngược ngăn xếp. (Các biệt lệ được xử lý bởi vế `except` trong một câu lệnh `try` không phải là các lỗi chúng ta nói đến ở đây.) Một số lỗi là nghiêm trọng không điều kiện và gây ra lỗi thoát với giá trị lỗi khác 0; điều này áp dụng cho các trường hợp mâu thuẫn nội tại và một vài trường hợp tràn bộ nhớ. Tất cả các thông báo lỗi đều được xuất ra dòng xuất lỗi chuẩn (standard error stream); kết xuất bình thường sẽ được xuất ra dòng xuất chuẩn (standard output - xin được hiểu là màn hình, tập tin...).

Gõ kí tự ngắt (thường là Control-C hoặc DEL) vào dấu nhắc chính hoặc dấu nhắc thứ sẽ bỏ những gì đã nhập vào và trở về dấu nhắc chính.[2.1] Gõ kí tự ngắt trong khi một lệnh đang được thực thi sẽ gây ra biệt lệ `KeyboardInterrupt` , trường hợp này có thể được xử lý bằng câu lệnh `try` .

2.2.2 Các kịch bản Python khả thi

Trên các hệ thống UNIX họ BSD, các kịch bản Python có thể được thực thi trực tiếp, như các kịch bản vỏ (shell script), bằng cách thêm dòng

```
#! /usr/bin/env python
```

(giả sử rằng bộ thông dịch đã có trong PATH của người dùng) ở đầu kịch bản và đặc thuộc tính thực thi (executable mode) cho tập tin đó. Ký hiệu "#!" phải là hai ký tự đầu tiên của tập tin. Trên các nền khác, dòng đầu tiên này phải kết thúc bằng một ký tự xuống dòng kiểu UNIX ("\n"), không phải Mac OS ("\r")

hay Windows ("\r\n"). Lưu ý rằng dấu thăng "#", được dùng để bắt đầu một chú thích trong Python.

Kịch bản có thể được đặt quyền thực thi bằng cách dùng lệnh **chmod** :

```
$ chmod +x myscript.py
```

2.2.3 Bảng mã mã nguồn

Có thể sử dụng các bảng mã khác bảng ASCII trong các tập tin nguồn Python. Cách tốt nhất là thêm các dòng chú thích đặc biệt vào ngay sau dòng #! để định nghĩa bảng mã trong tập tin:

```
# -*- coding: encoding -*-
```

Với khai báo này, mọi ký tự trong tập tin nguồn sẽ được xem như từ bảng mã *encoding*, và vì vậy ta có thể viết các chuỗi Unicode trực tiếp trong bảng mã đó. Danh sách các bảng mã có thể được tìm thấy ở *Tham khảo thư viện Python*, trong phần codecs.

Ví dụ, để viết ký tự biểu diễn đồng Euro, ta có thể sử dụng bảng mã ISO-8859-15, kí hiệu Euro có số thứ tự 164 trong bảng mã. Đoạn chương trình sau sẽ in ra giá trị 8364 (mã Unicode tương ứng với kí tự biểu diễn Euro) và thoát:

```
# -*- coding: iso-8859-15 -*-

currency = u"€"
print ord(currency)
```

Nếu bộ soạn thảo của bạn hỗ trợ lưu tập tin theo UTF-8 với *đánh dấu thứ tự byte* UTF-8 (UTF-8 byte order mark - BOM), bạn có thể dùng nó thay thế cho một khai báo bảng mã. IDLE hỗ trợ sự tương thích này nếu Options/General /Default Source Encoding/UTF-8 được thiết lập. Chú ý rằng ký hiệu này không được các phiên bản Python 2.2 trở về trước nhận biết, và cũng không được hệ điều hành nhận biết là các tập tin kịch bản với các dòng #! (chỉ được dùng trên các hệ UNIX).

Với việc sử dụng UTF-8 (thông qua kí hiệu cũng như khai báo bảng mã), các kí tự trong hầu hết các ngôn ngữ trên thế giới có thể được sử dụng đồng thời trong các chuỗi nguyên bản và các chú thích. Sử dụng các kí tự phi chuẩn ASCII trong các định danh thì không được hỗ trợ. Để hiển thị đúng các kí tự, bộ soạn thảo của bạn nhất thiết phải nhận biết tập tin UTF-8 và buộc phải sử dụng các phông chữ hỗ trợ tốt các kí tự này.

2.2.4 Tập tin khởi tạo tương tác

Khi bạn sử dụng Python ở chế độ tương tác, sẽ rất tiện lợi khi có một số lệnh chuẩn luôn được thực hiện mỗi khi bộ thông dịch khởi động. Bạn có thể thực hiện việc này bằng cách thiết lập một biến môi trường có tên PYTHONSTARTUP với giá trị là tên của tập tin bạn chứa các câu lệnh khởi tạo. Cách này tương tự như chức năng .profile của vỏ UNIX .

Tập tin này chỉ được đọc trong phiên làm việc tương tác, không có tác dụng với các kịch bản, và khi /dev/tty được chỉ định rõ là nguồn lệnh (nếu không thì trường hợp này cũng giống như một phiên làm việc tương tác). Nó được thực thi trong cùng vùng tên (namespace) mà các lệnh tương tác được thực thi, cho nên các đối tượng nó định nghĩa, hoặc nhập vào (import) có thể được dùng mà không cần xác nhận trong phiên làm việc tương tác. Bạn cũng có thể thay đổi dấu nhắc `sys.ps1` và `sys.ps2` trong tập tin này.

Nếu bạn muốn đọc các tập khởi động bổ sung từ thư mục hiện tại, bạn có thể lập trình điều này trong tập tin khởi động với mã như "`if os.path.isfile('.pythonrc.py'): execfile('.pythonrc.py')`". Nếu bạn muốn dùng tập tin khởi động trong một kịch bản, bạn phải chỉ rõ điều này trong kịch bản:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    execfile(filename)
```

Ghi chú

... dấu nhắc chính.[2.1]

Gói GNU Readline có một lỗi có thể ngăn cản điều này.

Bài chỉ dẫn Python

3. Giới thiệu sơ về Python

Trong ví dụ say, đầu vào và đầu ra được phân biệt bằng sự hiện diện của dấu nhắc (">>> " và "... "): để lặp lại ví tụ, bạn cần nhập vào mọi thứ sau dấu nhắc, khi dấu nhắc xuất hiện; các dòng không bắt đầu bằng một dấu nhắc là kết quả xuất từ trình thông dịch. Lưu ý rằng dấu nhắc thứ (secondary prompt) trên một dòng riêng nó có nghĩa là bạn phải nhận một dòng trống; dòng này dùng để kết thúc một lệnh nhiều dòng.

Nhiều ví dụ trong tài liệu này, ngay cả những ví dụ nhập từ dòng lệnh tương tác, có cả chú thích. Các chú thích trong Python bắt đầu bằng một dấu thăng, "#", và kéo dài tới hết dòng. Một chú thích có thể xuất hiện ở đầu dòng, hoặc theo sau khoảng trắng hoặc mã, nhưng không phải trong một chuỗi. Một dấu thăng trong một chuỗi chỉ là một dấu thăng.

Một vài ví dụ:

```
# this is the first comment
SPAM = 1                 # and this is the second comment
                         # ... and now a third!
STRING = "# This is not a comment."
```

3.1 Dùng Python như là máy tính

Hãy thử một vài lệnh Python đơn giản. Khởi động trình thông dịch và chờ dấu nhắc chính, ">>> ". (Không lâu đâu.)

3.1.1 Số

Trình thông dịch đóng vài trò là một máy tính đơn giản: bạn nhập một biểu thức và nó sẽ trả về giá trị. Cú pháp biểu thức rất dễ hiểu: các toán tử +, -, * và / hoạt động như trong hầu hết các ngôn ngữ khác (ví dụ Pascal hay C); dấu ngoặc tròn dùng để gộp nhóm. Ví dụ:

```
>>> 2+2
4
>>> # This is a comment
... 2+2
4
>>> 2+2  # and a comment on the same line as code
4
>>> (50-5*6)/4
5
>>> # Integer division returns the floor:
```

```
... 7/3
2
>>> 7/-3
-3
```

Dấu bằng ("=") được dùng để gán một giá trị vào một biến. Sau đó, không có giá trị nào được hiện ra trước dấu nhắc tương tác kế:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Một giá trị có thể được gán vào nhiều biến cùng một lúc:

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> y
0
>>> z
0
```

Python hoàn toàn hỗ trợ dấu chấm động; các toán tử với các toán hạng khác kiểu chuyển toán hạng số nguyên thành dấu chấm động:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Số phức cũng được hỗ trợ; số ảo được viết với hậu tố "`j`" hoặc "`J`". Các số phức với phần thực khác không được viết "(*real*+*imag*`j`)", hoặc có thể được tạo ra với hàm "`complex(`*real*`, `*imag*`)`".

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Các số phức luôn được thể hiện bởi hai số chấm động, phần thực và phần ảo. Để lấy các phần từ một số phức z, dùng z`.real` và z`.imag`.

```
>>> a=1.5+0.5j
>>> a.real
```

```
1.5
>>> a.imag
0.5
```

Các hàm chuyển đổi từ chấm động sang số nguyên (`float()`, `int()` và `long()`) không dùng được với số phức -- không có một cách chính xác nào để chuyển đổi một số phức thành một số thực. Dùng `abs(`z`)` để lấy độ lớn (magnitude) (như là một số chấm động) hoặc `z.real` để lấy phần thực.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)  # sqrt(a.real**2 + a.imag**2)
5.0
>>>
```

Trong chế độ tương tác, biểu thức được in ra cuối cùng được gán vào biến _. Khi bạn dùng Python như là máy tính, nó sẽ giúp bạn tiếp tục các phép tính dễ hơn, ví dụ:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
>>>
```

Biến này nên được coi là chỉ đọc từ phía người dùng. Không nên gán một giá trị vào biến này trực tiếp -- bạn sẽ tạo một biến cục bộ riêng với cùng tên, che đi biến có sẵn với cách thức (behavior) diệu kỳ của nó.

3.1.2 Chuỗi

Ngoài số, Python còn làm việc được với chuỗi, có thể được biểu hiện theo nhiều cách. Chúng có thể được kẹp trong dấu nháy đơn, đôi:

```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
```

```
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "\"Yes,\" he said."
'"Yes," he said.'
>>> '"Isn\'t," she said.'
'"Isn\'t," she said.'
```

Các chuỗi có thể phủ nhiều dòng theo nhiều cách. Các dòng tiếp tục (continuation line) có thể được dùng, với một dấu suyt huyền là ký tự cuối cùng trên một dòng cho biết rằng dòng kế là sự nối tiếp của dòng này:

```
hello = "This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
    Note that whitespace at the beginning of the line is\
 significant."

print hello
```

Lưu ý rằng các dòng mới vẫn cần được chèn trong chuỗi với \n; ký tự dòng mới theo sau dấu suyt huyền sẽ bị bỏ qua. Ví dụ này sẽ in ra:

```
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

Tuy nhiên, nếu ta làm cho chuỗi trực tiếp thành chuỗi ``thô'', các dãy \n sẽ không được chuyển thành các dòng mới, nhưng dấu suyt huyền ở cuối dòng, và ký tự dòng mới trong nguồn, sẽ đều được thêm vào trong chuỗi như dữ liệu. Cho nên, ví dụ:

```
hello = r"This is a rather long string containing\n\
several lines of text much as you would do in C."

print hello
```

sẽ in:

```
This is a rather long string containing\n\
several lines of text much as you would do in C.
```

Hoặc, các chuỗi có thể được vây quanh trong một cặp nháy ba: """ hoặc '''. Cuối mỗi dòng không cần thêm dấu suyt huyền khi dùng nháy ba, và chúng sẽ có mặt trong chuỗi.

```
print """
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
"""
```

xuất ra:

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Trình thông dịch in ra kết quả của các tác vụ chuỗi theo cùng cách như khi chúng được nhập vào: trong dấu nháy, và với các ký tự dấu nháy hay đặc biệt khác được thoát nghĩa (escape) bằng dấu suyt huyền, để hiện giá trị thực. Chuỗi được kèm trong dấu nháy đôi nếu chuỗi chứa một dấu nháy đơn và không chứa dấu nháy đôi, ngoài ra nó sẽ được chứa trong các dấu nháy đơn. (Câu lệnh `print` , được giải thích sau, có thể dùng để viết các chuỗi không có dấu nháy hoặc thoát nghĩa.)

Các chuỗi có thể được nối với nhau với toán tử `+` , và được lặp lại với *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Hai chuỗi trực tiếp kế nhau được tự động nối với nhau; dòng đầu tiên bên trên có thể được biết "`word = 'Help' 'A'`"; việc này chỉ có tác dụng với hai chuỗi trực tiếp (string literal), không có tác dụng với các biểu thức chuỗi bất kỳ khác:

```
>>> 'str' 'ing'                   #  <-  This is ok
'string'
>>> 'str'.strip() + 'ing'   #  <-  This is ok
'string'
>>> 'str'.strip() 'ing'     #  <-  This is invalid
  File "<stdin>", line 1, in ?
    'str'.strip() 'ing'
                  ^
SyntaxError: invalid syntax
```

Các chuỗi có thể được chỉ mục (subscript hoặc index); như trong C, ký tự đầu tiên của một chuỗi có chỉ mục 0. Không có kiểu ký tự riêng; một ký tự chỉ đơn giản là một chuỗi có độ dài là một. Như trong Icon, chuỗi con có thể được chỉ định theo *cách viết cắt lát (slice notation)*: hai chỉ mục phân cách bởi một dấu hai chấm.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Các chỉ mục cắt lát có giá trị mặc định hữu dụng; chỉ mục đầu tiên có giá trị mặc định là không, chỉ mục thứ hai mặc định là kích thước của chuỗi đang bị cắt.

```
>>> word[:2]    # The first two characters
'He'
>>> word[2:]    # Everything except the first two characters
'lpA'
```

Không như C, các chuỗi Python không thể bị thay đổi. Phép gán vào một vị trí chỉ mục trong một chuỗi sẽ gây ra lỗi:

```
>>> word[0] = 'x'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> word[:1] = 'Splat'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Tuy nhiên, việc tạo một chuỗi với nội dung gộp chung cũng dễ và hiệu quả:

```
>>> 'x' + word[1:]
'xelpA'
>>> 'Splat' + word[4]
'SplatA'
```

Đây là một tính chất bất biến hữu dụng khác của tác vụ cắt lát: `s[:i] + s[i:]` bằng `s`.

```
>>> word[:2] + word[2:]
'HelpA'
>>> word[:3] + word[3:]
'HelpA'
```

Các chỉ mục cắt lát giảm sinh (degenerate) được xử lý rất khéo: một chỉ mục quá lớn sẽ được thay bằng kích thước chuỗi, một giới hạn trên nhỏ hơn giới hạn dưới trả về một chuỗi rỗng.

```
>>> word[1:100]
'elpA'
>>> word[10:]
''
>>> word[2:1]
''
```

Các chỉ mục có thể là số âm, để bắt đầu đếm từ bên phải. Ví dụ:

```
>>> word[-1]     # The last character
'A'
>>> word[-2]     # The last-but-one character
'p'
>>> word[-2:]    # The last two characters
'pA'
>>> word[:-2]    # Everything except the last two characters
```

```
'Hel'
```

Nhưng lưu ý rằng `-0` thật ra cũng là `0`, cho nên nó không bắt đầu đếm từ bên phải!

```
>>> word[-0]     # (since -0 equals 0)
'H'
```

Các chỉ mục cắt lát âm ngoài phạm vi thì bị thu ngắn, nhưng đừng thử kiểu này với các chỉ mục một phần từ (không phải cắt lát):

```
>>> word[-100:]
'HelpA'
>>> word[-10]    # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

Cách tốt nhất để nhớ hoạt động của cắt lát là nghĩ về các chỉ mục như đang trỏ vào *giữa* các ký tự, với cạnh trái của ký tự đầu tiên là 0. Sau đó cạnh phải của ký tự cuối cùng của một chuỗi của n ký tự có chỉ mục n, ví dụ:

```
 +---+---+---+---+---+
 | H | e | l | p | A |
 +---+---+---+---+---+
 0   1   2   3   4   5
-5  -4  -3  -2  -1
```

Các số hàng đầu cho biết vị trí của các chỉ mục 0...5 trong chuỗi; dòng thứ hai cho biết các chỉ mục âm tương ứng. Một lát từ i tới j chứa toàn bộ các ký tự giữa các cạnh đánh số i và jtương ứng.

Với các chỉ mục không âm, chiều dài của lát là hiệu của các chỉ mục, nếu cả hai đều trong giới hạn. Ví dụ, độ dài của `word[1:3]` là 2.

Hàm có sẵn `len()` trả về độ dài của một chuỗi:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Xem thêm:

Các kiểu dãy
: Chuỗi, và các chuỗi Unicode được nhắc đến trong mục kế, là ví dụ của *các kiểu dãy*, và hỗ trợ các tác vụ chung được hỗ trợ bởi các kiểu đó.

Các phương thức chuỗi
: Cả chuỗi và chuỗi Unicode hỗ trợ một số lớn các phương thức nhằm vào chuyển đổi (transform) và tìm kiếm.

Các tác vụ định dạng chuỗi
Các tác vụ định dạng chuỗi được gọi khi các chuỗi và chuỗi Unicode là toán hạng bên trái của toán tử % được bàn đến chi tiết hơn ở đây.

3.1.3 Chuỗi Unicode

Bắt đầu với Python 2.0, một kiểu dữ liệu mới để chứa dữ liệu văn bản được cung cấp cho nhà lập trình: đối tượng Unicode. Nó có thể được dùng để chứa và thay đổi dữ liệu Unicode (xem http://www.unicode.org/) và tích hợp tốt với các đối tượng chuỗi đã có, bằng việc tự chuyển đổi khi cần.

Unicode có lợi điểm là cung cấp một số thứ tự (ordinal) cho mọi ký tự trong các bản thảo dùng trong các văn bản xưa và nay. Trước kia, chỉ có 256 số thứ tự cho các ký tự bản thảo. Các văn bản xưa bị giới hạn vào một trang mã (code page) dùng để ánh xạ các số vào các ký tự bản thảo. Điều này dẫn đến nhiều lẫn lộn đặc biệt là trong ngữ cảnh quốc tế hóa (internationalization, hay được viết tắt là "i18n" -- "i" + 18 ký tự + "n") phần mềm. Unicode giải quyết các vấn đề này bằng các định nghĩa một trang mã cho mọi bản thảo.

Tạo một chuỗi Unicode trong Python dễ như tạo một chuỗi thường:

```
>>> u'Hello World !'
u'Hello World !'
```

Ký tự "u" đằng trước dấu nháy cho biết đây là một chuỗi Unicode cần được tạo. Nếu bạn muốn thêm các ký tự đặc biệt trong chuỗi, bạn có thể làm vậy bằng cách viết thoát nghĩa *Unicode-Escape* của Python. Ví dụ:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

Dãy thoát nghĩa \u0020 cho biết chèn một ký tự Unicode với thứ tự 0x0020 (ký tự khoảng trắng) vào một vị trí đã định.

Các ký tự khác được thông dịch theo thứ tự tương ứng của chúng như là thứ tự Unicode. Nếu bạn có các chuỗi trực tiếp trong bảng mã Latin-1 chuẩn được dùng ở nhiều nước phương Tây, bạn sẽ thấy rằng 256 ký tự đầu của bảng mã Unicode giống như 256 ký tự của Latin-1.

Cho các chuyên gia, Python cũng hỗ trợ các chuỗi Unicode thô. Bạn phải dùng tiền tố 'ur' để bảo Python dùng bảng mã *thoát-nghĩa-Unicode-thô (Raw-Unicode-Escape)* . Nó sẽ chỉ áp dụng phép chuyển đổi \uXXXX bên trên nếu có một số lẻ các dấu suyt huyền phía trước ký tự 'u' nhỏ.

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\\u0020World !'
```

Chế độ thô sẽ hữu dụng trong trường hợp bạn phải nhập thật nhiều dấu suyt huyền, như khi bạn dùng trong các biểu thức chính quy (regular expression).

Ngoài những bảng mã chuẩn này, Python cung cấp một tập hợp các cách khác để tạo các chuỗi Unicode từ một bảng mã đã biết.

Hàm có sẵn `unicode()` cung cấp truy cập vào tất cả bộ mã/giải mã (codec - COder and DECoder) Unicode đã đăng ký. Một vài bảng mã phổ thông mà các bộ chuyển mã này có thể chuyển gồm *Latin-1*, *ASCII*, *UTF-8*, và *UTF-16*. Hai bảng mã sau cùng là các bảng mã có kích thước thay đổi và chứa mỗi ký tự Unicode trong một hoặc nhiều byte. Bảng mã mặc định thường được thiết lập là ASCII, nó cho phép các ký tự từ 0 tới 127 và cấm các ký tự khác. Khi một chuỗi Unicode được in, viết vào tập tin, hoặc chuyển với `str()`, sự chuyển đổi diễn ra với bảng mã mặc định này.

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xf6\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0·
```

Để chuyển một chuỗi Unicode thành một chuỗi 8-bit bằng một bảng mã nào đó, các đối tượng Unicode cung cấp một phương thức `encode()` nhận một thông số, tên của bảng mã. Bạn nên dùng tên bảng mã viết thường.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

Nếu bạn có dữ liệu trong một bảng mã nào đó và muốn tạo ra một chuỗi Unicode tương ứng từ nó, bạn có thể dùng hàm `unicode()` với tên bảng mã là thông số thứ hai.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xf6\xfc'
```

3.1.4 Danh sách

Python biết một số kiểu dữ liệu *gộp (compound)* , dùng để nhóm các giá trị với nhau. Kiểu linh hoạt nhất là *danh sách (list)*, có thể được viết như là một danh sách các giá trị phân cách bởi dấu phẩy ở giữa ngoặc vuông.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Cũng như các chỉ mục chuỗi, chỉ mục danh sách bắt đầu từ 0, và danh sách có thể được cắt lát, gộm và vân vân:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Không như chuỗi, là những đối tượng *immutable (bất biến, không thể thay đổi)*, ta có thể thay đổi các phần tử của một danh sách:

```
>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]
```

Gán vào các cắt lát cũng có thể làm được, và nó có thể thay đổi kích thước của danh sách hoặc xóa sách nó.

```
>>> # Replace some items:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Remove some:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Insert some:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> # Insert (a copy of) itself at the beginning
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
>>> # Clear the list: replace all items with an empty list
>>> a[:] = []
>>> a
[]
```

Hàm có sẵn `len()` cũng áp dụng vào danh sách:

```
>>> len(a)
8
```

Có thể lồng các danh sách (tạo danh sách chứa các danh sách khác), ví dụ:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # See section 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Lưu ý trong ví dụ, `p[1]` và `q` thật ra chỉ tới cùng đối tượng! Chúng ta sẽ nói về *nghĩa của đối tượng (object semantics)* trong các chương sau.

3.2 Những bước đầu lập trình

Dĩ nhiên, chúng ta có dùng Python cho các tác vụ phức tạp khác. Ví dụ ta có thể viết một dãy con ban đầu của dãy *Fibonacci* như sau:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Ví dụ này giới thiệu một vài tính năng mới.

- Dòng đầu tiên chứa một *phép gán đa biến (multiple assignment)*: các biến `a` và `b` đồng loạt nhận giá trị mới 0 và 1. Trong dòng cuối nó được dùng một lần nữa, cho thấy rằng các biểu thức ở phía phải được xác định trước khi bất kỳ phép gán nào được thực hiện. Các biểu thức phía phải được định giá từ trái qua phải.

- Vòng lặp `while` thực thi miễn là điều kiện (ở đây: `b < 10`) vẫn là đúng

(true). Trong Python, như C, mọi giá trị số nguyên khác không là đúng; không là sai (false). Điều kiện cũng có thể là danh sách, chuỗi, hoặc bất kỳ kiểu dãy nào; chiều dài khác không là đúng, dãy rỗng là sai. Phép so sánh dùng trong ví dụ là một phép so sánh đơn giản. Các toán tử so sanh chuẩn được viết như trong C: < (nhỏ hơn), > (lớn hơn), == (bằng), <= (nhỏ hơn hoặc bằng), >= (lớn hơn hoặc bằng) và != (không bằng).

- Vòng lặp *Thân* vòng lặp được *thụt vào*: các thục vào là cách của Python để nhóm các câu lệnh. Python không (chưa) cung cấp một công cụ soạn thảo dòng nhập thông minh, cho nên bạn phải nhập vào một tab hoặc khoảng trắng cho mỗi dòng thụt vào. Trong thực tế, bạn sẽ chuẩn bị đầu vào phức tạp hơn cho Python với một trình soạn thảo; đa số chúng đều có chức năng tự động thụt vào. Khi một câu lệnh ghép (compound statement) được nhập vào một cách tương tác, nó phải được theo sau bởi một dòng trống để chỉ ra sự kết thúc (vì bộ phân tích không thể khi nào bạn nhập dòng cuối). Lưu ý rằng mỗi dòng của một khối phải được thụt vào như nhau.

- Vòng lặp `print` (câu lệnh) viết ra giá trị của biểu thức nó được cung cấp. Nó khác với việc chỉ viết các biểu thức bạn muốn viết (như chúng ta đã làm trong các ví dụ máy tính trước) trong việc xử lý nhiều biểu thức và chuỗi. Các chuỗi được in ra không có dấu nháy, và một khoảng trắng được chèn vào giữa các phần tử, để bạn có thể định dạng chúng đẹp hơn, ví dụ:

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

Dấu phẩy sau cùng tránh dòng mới sau khi xuất:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Lưu ý rằng trình thông dịch chèn một dòng mới trước khi nó in ra dấu nhắc kế nếu dòng trước chưa xong.

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách để nghị thay đổi.

4. Bàn thêm về luồng điều khiển

Ngoài câu lệnh `while` vừa giới thiệu, Python có các câu lệnh điều khiển luồng từ các ngôn ngữ khác, với chút sửa đổi.

4.1 `if` câu lệnh

Có lẽ loại câu lệnh biết đến nhiều nhất là câu lệnh `if` . Ví dụ:

```
>>> x = int(raw_input("Please enter an integer: "))
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
...
```

Có thể không có hoặc có nhiều phần `elif` , và phần `else` là không bắt buộc. Từ khóa `` `elif' `` là viết tắt của `` `else if' ``, và dùng để tránh thụt vào quá nhiều. Dãy `if` ... `elif` ... `elif` ... dùng thay cho câu lệnh `switch` hay `case` tìm thấy trong các ngôn ngữ khác.

4.2 `for` câu lệnh

`for` trong Python khác một chút với C hoặc Pascal. Thay vì lặp qua một dãy số (như trong Pascal), hoặc cho phép người dùng tự định nghĩa bước lặp và điều kiện dừng (như C), câu lệnh `for` của Python lặp qua các phần tử của một dãy bất kỳ (một danh sách, hoặc một chuỗi), theo thứ tự mà chúng xuất hiện trong dãy. Ví dụ:

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

Rất nguy hiểm nếu bạn sửa đổi dãy trong khi bạn đang lặp qua nó. Nếu bạn cần sửa đổi một danh sách khi đang lặp (ví dụ như để nhân đôi các phần tử nào đó) bạn sẽ cần phải lặp qua một bản sao của nó. Cách viết cắt miếng làm cho việc này đơn giản:

```
>>> for x in a[:]: # make a slice copy of the entire list
...    if len(x) > 6: a.insert(0, x)
...
>>> a
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 `range()` hàm

Nếu bạn cần lặp qua một dãy số, hàm có sẵn `range()` trở nên tiện dụng. Nó tạo ra danh sách chứa các dãy số học:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Điểm dừng được chỉ định không bao giờ là một phần của danh sách tạo ra; `range(10)` tạo danh sách 10 giá trị, là những chỉ mục hợp lệ cho các phần tử của một dãy có độ dài 10. Bạn cũng có thể tạo dãy bắt đầu từ một số khác, hoặc chỉ rõ mức tiến khác (ngay cả mức lùi; đôi khi nó còn được gọi là `bước', `step'):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Để lặp qua các chỉ mục của một dãy, gộp `range()` và `len()` như sau:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

4.4 break và continue câu lệnh, và else vế lặp

`break` , như trong C, nhảy ra khỏi phạm vòng lặp `for` hay `while` nhỏ nhất chứa nó.

`continue` , cũng được mượn từ C, tiếp tục lần lặp kế của vòng lặp.

Các câu lệnh lặp có thể có vế `else` ; nó được thực thi khi vòng lặp kết thúc vì danh sách lặp đã cạn (với `for`) hoặc khi điều kiện là sai (với `while`), và không được thực thi khi vòng lặp kết thúc bởi câu lệnh `break` . Ví dụ vòng lặp sau tìm các số nguyên tố:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...     else:
...         # loop fell through without finding a factor
...         print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

4.5 pass câu lệnh

`pass` không làm gì cả. Nó có thể được dùng khi cú pháp cần một câu lệnh nhưng chương trình không cần tác vụ nào. Ví dụ:

```
>>> while True:
...       pass # Busy-wait for keyboard interrupt
...
```

4.6 Định nghĩa hàm

Chúng ta có thể tạo một hàm in ra dãy Fibonacci:

```
>>> def fib(n):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Từ khóa `def` khai báo một *định nghĩa* hàm . Nó phải được theo sau bởi tên hàm, và một danh sách các thông số chính quy trong ngoặc đơn. Các câu lệnh tạo nên thân hàm bắt đầu từ dòng kế tiếp, và bắt buộc phải được thụt vào. Câu lệnh đầu tiên của thân hàm có thể là một chuỗi; chuỗi này là chuỗi tài liệu, hoặc *docstring*của hàm.

Có những công cụ sử dụng docstrings để tự động sinh tài liệu trực tuyến hoặc để in, hoặc cho phép người dùng duyệt mã một cách tương tác; việc thêm docstrings vào mã rất được khuyến khích, cho nên bạn hãy tạo thói quen tốt đó cho mình.

Việc thực thi một hàm tạo ra một bảng ký hiệu mới dùng cho các biến cục bộ của hàm. Chính xác hơn, mọi phép gán biến trong một hàm chứa giá trị vào bảng ký hiệu cục bộ; và các tham chiếu biến sẽ trước hết tìm trong bảng ký hiệu cục bộ rồi trong bảng ký hiệu toàn cục, và trong bảng các tên có sẵn. Do đó, các biến toàn cục không thể được gán giá trị trực tiếp trong một hàm (trừ khi được đặt trong câu lệnh `global`), mặc dù chúng có thể dược tham chiếu tới.

Thông số thật sự của một lệnh gọi hàm được tạo ra trong bảng ký hiệu cục bộ của hàm được gọi khi nó được gọi; do đó các thông số được truyền theo *truyền theo giá trị (call by value)* (mà *giá trị* luôn là một *tham chiếu đối tượng*, không phải là giá trị của đối tượng).[4.1] Khi một hàm gọi một hàm khác, một bảng ký hiệu cục bộ được tạo ra cho lệnh gọi đó.

Một định nghĩa hàm tạo tên hàm trong bảng ký hiệu hiện tại. Giá trị của tên hàm có một kiểu được nhận ra bởi trình thông dịch là hàm do người dùng định nghĩa. Giá trị này có thể được gán vào một tên khác và sau đó có thể được sử dụng như một hàm. Đây là một cách đổi tên tổng quát:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Bạn có thể nói rằng `fib` không phải là một hàm (function) mà là một thủ tục (procedure). Trong Python, cũng như C, thủ tục chẳng qua là hàm không có giá trị trả về. Thật sự, nói rõ hơn một chút, thủ tục cũng trả về giá trị mặc dù là một giá trị vô nghĩa. Giá trị này được gọi là `None` (nó là một tên có sẵn). In ra giá trị `None` thường bị trình thông dịch bỏ qua nếu nó là giá trị duy nhất được in ra. Bạn có thể thấy nó nếu bạn muốn:

```
>>> print fib(0)
None
```

Bạn cũng có thể dễ dàng viết một hàm trả về một danh sách các số của dãy Fibonacci thay vì in nó ra:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while b < n:
...         result.append(b)    # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # call it
>>> f100                # write the result
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Ví dụ này cho thấy một vài tính năng mới của Python:

- `return` trả về với một giá trị từ một hàm. `return` không có thông số biểu thức đi kèm trả về `None`. Rớt ra khỏi một thủ tục cũng trả về `None`của hàm.
- Câu lệnh `result.append(b)` gọi một *phương thức* của đối tượng danh sách. `result`. Một phương thức là một hàm `thuộc về' một đối tượng và có tên `obj.methodname`, với `obj` là một đối tượng nào đó (có thể là một biểu thức), và `methodname` là tên của một phương thức được định nghĩa bởi kiểu của đối tượng. Các kiểu khác nhau định nghĩa các phương thức khác nhau. Phương thức của các kiểu khác nhau có thể có cùng tên mà không dẫn đến sự khó hiểu. (Bạn có thể định nghĩa kiểu đối tượng và phương thức cho riêng bạn, dùng *lớp*, như sẽ được bàn đến ở các chương sau.) Phương thức `append()` dùng trong ví dụ này được định nghĩa cho các đối tượng danh sách; nó thêm một phần tử mới vào cuối danh sách. Trong ví dụ này, nó tương đương với "`result = result + [b]`", nhưng hiệu quả hơn.

4.7 Bàn thêm về định nghĩa hàm

Bạn cũng có thể định nghĩa các hàm với số lượng thông số thay đổi. Có ba dạng, và chúng có thể được dùng chung với nhau.

4.7.1 Giá trị thông số mặc định

Dạng hữu dụng nhất là để chỉ định một giá trị mặc định cho một hoặc nhiều thông số. Dạng này tạo một hàm có thể được gọi với ít thông số hơn là nó được định nghĩa để nhận. Ví dụ:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return True
        if ok in ('n', 'no', 'nop', 'nope'): return False
        retries = retries - 1
        if retries < 0: raise IOError, 'refusenik user'
        print complaint
```

Hàm này có thể được gọi như sau: `ask_ok('Do you really want to quit?')` hoặc như sau: `ask_ok('OK to overwrite the file?', 2)`của hàm.

Ví dụ này giới thiệu từ khóa `in` . Nó kiểm tra xem một dãy có chứa một giá trị nào đó không.

Các giá trị mặc định được định giá tại nơi hàm được định nghĩa trong phạm vi *định nghĩa (defining scope)* , do đó

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

sẽ in 5của hàm.

Cảnh báo quan trọng: Giá trị mặc định chỉ được định giá một lần. Điểm này quan trọng khi mặc định là một giá trị khả biến như danh sách, từ điển hoặc các đối tượng của hầu hết mọi lớp. Ví dụ, hàm sau gộp các thông số truyền vào nó từ các lời gọi sau đó:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Sẽ in ra

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

Nếu bạn không muốn mặc định được dùng chung trong các lời gọi sau, bạn có thể viết một hàm như thế này:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2 Thông số từ khóa

Các hàm cũng có thể được gọi theo thông số từ khóa (keyword argument) theo dạng "*keyword* = *value*". Ví dụ, hàm sau:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

có thể được gọi theo bất kỳ cách nào:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

nhưng những lời gọi sau đều không hợp lệ:

```
parrot()                     # required argument missing
parrot(voltage=5.0, 'dead')  # non-keyword argument following keyword
parrot(110, voltage=220)     # duplicate value for argument
parrot(actor='John Cleese')  # unknown keyword
```

Nói chung, một danh sách thông số phải có bất kỳ thông số vị trí (positional argument) theo sau bởi bất kỳ thông số từ khóa, các từ khóa phải được chọn từ tên thông số chính quy. Các thông số chính quy không nhất thiết phải có giá trị mặc định. Không thông số nào có thể nhận một giá trị nhiều hơn một lần -- tên thông số chính quy tương ứng với thông số vị trí không thể được dùng làm từ khóa trong cùng một lời gọi. Sau đây là một ví dụ sai vì giới hạn này:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Khi thông số chính quy cuối có dạng ***name* , nó nhận một từ điển (dictionary) chứa tất cả các thông số từ khóa trừ những từ khóa tương ứng với thông số chính quy. Điểm này có thể được dùng chung với một thông số chính quy ở dạng **name* (bàn đến trong mục con sau) và nhận một bộ (tuple) chứa các thông số vị trí sau danh sách thông số chính quy. (**name* bắt buộc phải xuất hiện trước ***name*.) Ví dụ, nếu ta định nghĩa một hàm như sau:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, '?'
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments: print arg
    print '-'*40
    keys = keywords.keys()
```

```
    keys.sort()
    for kw in keys: print kw, ':', keywords[kw]
```

Nó có thể được gọi như vậy:

```
cheeseshop('Limburger', "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           client='John Cleese',
           shopkeeper='Michael Palin',
           sketch='Cheese Shop Sketch')
```

và dĩ nhiên nó sẽ in ra:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----------------------------------------
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Lưu ý rằng phương thức `sort()` của danh sách các tên thông số từ khóa được gọi trước khi in nội dung của từ điển `keywords` ; nếu điều này không được thực hiện, thứ tự các thông số được in ra không xác định.

4.7.3 Danh sách thông số bất kỳ

Cuối cùng, một lựa chọn ít dùng nhất để chỉ định rằng một hàm có thể được gọi với bất kỳ số thông số. Các thông số này sẽ được gói và trong một bộ. Trước các thông số không xác định, không hoặc nhiều hơn các thông số chính quy có thể có mặt.

```
def fprintf(file, format, *args):
    file.write(format % args)
```

4.7.4 Tháo danh sách thông số

Trường hợp ngược xảy ra khi các thông số đã nằm trong một danh sách hoặc một bộ nhưng cần được tháo ra cho lời gọi hàm cần những thông số vị trí riêng. Ví dụ, hàm có sẵn `range()` cần nhận các thông số riêng *start* và *stop* . Nếu chúng không được cung cấp riêng lẻ, viết lệnh gọi hàm với toán tử *để tháo các thông số này ra khỏi một danh sách hoặc bộ:

```
>>> range(3, 6)             # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args)            # call with arguments unpacked from a list
[3, 4, 5]
```

Theo cùng một kiểu, từ điển có thể cung cấp các thông số từ khóa với toán tử **:

```
>>> def parrot(voltage, state='a stiff', action='voom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

4.7.5 Dạng lambda

Theo yêu cầu chung, một vài tính năng thường thấy trong các ngôn ngữ lập trình hàm như Lisp đã được thêm vào Python. Với từ khóa `lambda` , các hàm vô danh (anonymous function) có thể được tạo ra. Đây là một hàm trả vè tổng của hai thông số: "`lambda a, b: a+b`". Dạng lambda có thể được dùng ở bất kỳ

nơi nào cần đối tượng hàm. Cú pháp của chúng giới hạn ở một biểu duy nhất. Về ý nghĩa, chúng chỉ là một cách viết gọn của một định nghĩa hàm bình thường. Giống như các định nghĩa hàm lồng nhau, dạng lambda có thể tham chiếu các biến từ phạm vi chứa nó:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

4.7.6 Chuỗi tài liệu

Có những quy luật đang hình thành về nội dung và định dạng của các chuỗi tài liệu.

Dòng đầu tiên cần phải luôn luôn là một tóm tắt ngắn, xúc tích về mục đích của đối tượng. Để dễ hiểu, nó không nên chỉ ra cụ thể tên hoặc kiểu của đối tượng vì chúng có thể có ở hình thức khác (từ khi tên là một động từ diễn tả hoạt động của hàm). Dòng này cần bắt đầu bằng một chữ hoa và kết thúc bằng một dấu chấm.

Nếu có nhiều dòng trong chuỗi tài liệu, dòng thứ hai nên là một dòng trống, rõ ràng phân biệt tóm tắt và phần còn lại. Các dòng sau nên là một hoặc nhiều đoạn hướng dẫn về cách gọi, các hiệu ứng phụ, v.v...

Bộ phân tích ngữ pháp Python không lọc thụt hàng từ các chuỗi đa dòng (multi-line string literal) trong Python, so nên các công cụ xử lý tài liệu cần phải lọc thụt hàng nếu cần. Việc này được làm theo một cách chung. Dòng không trống đầu tiên *sau* dòng đầu tiên của chuỗi xác định mức thụt vào cho toàn bộ chuỗi tài liệu. (Ta không thể dùng dòng đầu tiên vì nó thường nằm kế dấu nháy đầu chuỗi cho nên mức thụt vào của nó không được xác định trong cách viết chuỗi.) Khoảng trắng ``tương đương'' với mức thụt vào này được bỏ khỏi mỗi đầu dòng trong chuỗi. Không nên có các dòng thụt vào ít hơn, nhưng nếu gặp phải, toàn bộ khoảng trắng đầu của chúng nên được bỏ đi. Tính tương đương của khoảng trắng cần được kiểm tra sau khi mở rộng tab (thông thường thành 8 khoảng trắng).

Đâu là ví dụ của một docstring đa dòng:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

Ghi chú

... đối tượng).[4.1]

Thật ra, *gọi theo tham chiếu đối tượng (call by object reference)* có thể là một diễn giải tốt hơn, vì nếu một đối tượng khả đổi được truyền vào, nơi gọi sẽ nhận được các thay đổi do nơi được gọi tạo ra (ví dụ như các phần tử được thêm vào danh sách).

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách để nghị thay đổi.

Bài chỉ dẫn Python

5. Cấu trúc dữ liệu

Chương này diễn giải kỹ hơn một vài điều bạn đã học được, và cũng nói thêm về một số điều mới.

5.1 Bàn thêm về danh sách

Kiểu dữ liệu danh sách (kiểu list) có một số phương thức khác. Đây là toàn bộ các phương thức của đối tượng danh sách:

append(*x*)
Thêm một phần tử vào cuối danh sách; tương đương với `a[len(a):] = [`*x*`]`.

extend(*L*)
Nới rộng danh sách bằng cách chèn vào tất cả các phần tử của danh sách chỉ định; tương đương với `a[len(a):] =` *L*.

insert(*i*, *x*)
Chèn một phần tử vào vị trí chỉ định. Thông số đầu là chỉ mục của phần tử sẽ bị đẩy lùi, cho nên `a.insert(0,` *x*`)` chèn vào đầu danh sách, và `a.insert(len(a),` *x*`)` tương đương với `a.append(`*x*`)`.

remove(*x*)
Bỏ ra khỏi danh sách phần tử đầu tiên có giá trị là *x*. Sẽ có lỗi nếu không có phần tử như vậy.

pop([*i*])
Bỏ khỏi danh sách phần tử ở vị trí chỉ định, và trả về chính nó. Nếu không chỉ định vị trí, `a.pop()` bỏ và trả về phần tử cuối trong danh sách. (Ngoặc vuông xung quanh *i* trong khai báo hàm cho biết thông số đó là không bắt buộc, không có nghĩa là bạn cần gõ dấu ngoặc vuông ở vị trí đó. Bạn sẽ thấy cách viết này thường xuyên trong *Tham khảo thư viện Python*.)

index(*x*)
Trả về chỉ mục của phần tử trong danh sách mà có giá trị là *x*. Sẽ có lỗi nếu không có phần tử như vậy.

count(*x*)
Trả về số lần *x* xuất hiện trong danh sách.

sort()
Sắp xếp các phần tử trong danh sách, ngay tại chỗ.

reverse()
Đảo ngược thứ tự các phần tử trong danh sách, ngay tại chỗ.

Một ví dụ có sử dụng hầu hết các phương thức của danh sách:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

5.1.1 Dùng danh sách như ngăn xếp

Các phương thức của danh sách làm cho nó rất dễ sử dụng như là ngăn xếp (stack), là nơi mà phần tử cuối được thêm vào là phần tử đầu được lấy ra (``vào sau, ra trước'' hay ``last-in, first-out''). Để thêm phần tử vào đỉnh của ngăn xếp, dùng `append()`. Để lấy một phần tử từ đỉnh của ngăn xếp, dùng `pop()` mà không chỉ định chỉ mục. Ví dụ:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Dùng danh sách như hàng đợi

Bạn cũng có thể thuận tiện dùng danh sách như là hàng đợi (queue), nơi mà phần tử được thêm vào đầu tiên là phần tử được lấy ra đầu tiên (``vào trước, ra trước'' hay ``first-in, first-out''). Để thêm một phần tử vào cuối hàng đợi, dùng `append()`. Để lấy một phần tử từ đầu hàng đợi, dùng `pop()` với `0` là chỉ mục. Ví dụ:

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

5.1.3 Công cụ lập trình hướng hàm

Có sẵn ba hàm rất hữu dụng khi dùng với danh sách: `filter()`, `map()`, và `reduce()`.

"`filter(`*function*`,` *sequence*`)`" trả về một dãy chứa các phần tử từ dãy mà *function*`(`*item*`)` có giá trị đúng. Nếu *sequence* là một `string` hoặc `tuple`, thì kết quả trả về sẽ có cùng kiểu; ngược lại, sẽ luôn luôn là một `list`. Ví dụ, để tìm một vài số nguyên tố:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

"`map(`*function*`,` *sequence*`)`" gọi *function*`(`*item*`)` với mỗi phần tử trong dãy và trả về một danh sách các giá trị trả về. Ví dụ, để tính một vài số lập phương:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Có thể truyền vào nhiều dãy; hàm đó phải nhận từng ấy thông số với mỗi phần tử trong mỗi dãy là một thông số (hoặc `None` nếu dãy nào đó ngắn hơn dãy còn lại). Ví dụ:

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

"`reduce(`*function*`,` *sequence*`)`" trả về giá trị duy nhất được tạo ra từ việc gọi hàm

nhị phân *function* với thông số là hai phần tử đầu của dãy, rồi sau đó với giá trị trả về này với phần tử kế, và cứ thế. Ví dụ, để tính tổng của các số từ 1 đến 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

Nếu chỉ có một phần tử trong dãy, giá trị của nó sẽ được trả về; nếu dãy rỗng, biệt lệ sẽ được nâng.

Có thể truyền thêm thông số thứ ba để cho biết giá trị ban đầu. Trong trường hợp đó, giá trị này sẽ được trả về nếu dãy rỗng, và hàm sẽ được áp dụng cho giá trị ban đầu, và giá trị của phần tử đầu của dãy, rồi với giá trị được trả về với giá trị của phần tử kế, và cứ thế. Ví dụ,

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Đừng dùng định nghĩa của ví dụ này về `sum()`: vì việc cộng các con số là một nhu cầu chung, một hàm có sẵn `sum(`*sequence*`)` đã được cung cấp, và hoặc động y như vậy. Từ phiên bản 2.3.

5.1.4 Gộp danh sách

Việc gộp danh sách (list comprehension) cung cấp một cách xúc tích để tạo danh sách mà không cần dùng tới `map()`, `filter()` hoặc `lambda`. Kết quả là khai báo danh sách kiểu này thường dễ hiểu hơn những danh sách tạo ra từ những cách kia. Mỗi gộp danh sách chứa một biểu thức, theo sau bởi vế `for` , rồi không có hoặc có các vế `for` hoặc `if` . Kết quả sẽ là một danh sách được trả về từ việc định giá biểu thức trong ngữ cảnh của các vế `for` và `if` theo sau nó. Nếu biểu thức trả về một tuple, nó phải được đặt trong ngoặc.

```
>>> freshfruit = ['  banana', '  loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
>>> [[x,x**2] for x in vec]
```

```
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in vec]      # error - parens required for tuples
  File "<stdin>", line 1, in ?
    [x, x**2 for x in vec]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

Cách gộp danh sách uyển chuyển hơn nhiều so với `map()` và có thể được áp dụng cho các biểu thức phức tạp và các hàm lồng nhau:

```
>>> [str(round(355/113.0, i)) for i in range(1,6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.2 `del` câu lệnh

Có một cách để bỏ một phần tử ra khỏi danh sách dựa trên chỉ mục của nó, thay vì giá trị: câu lệnh `del` . Cách này khác với phương thức `pop()`trả về một giá trị. Câu lệnh `del` cũng có thể được sử dụng để bỏ các miếng cắt (slice) khỏi danh sách hoặc xóa toàn bộ danh sách (điều mà chúng ta đã làm trước đó bằng cách gán một danh sách rỗng vào miếng cắt). Ví dụ:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` cũng có thể được dùng để xóa hoàn toàn các biến:

```
>>> del a
```

Tham chiếu tới tên `a` sau đó sẽ tạo ra lỗi (ít nhất cho đến khi một giá trị khác được gán vào cho nó). Chúng ta sẽ thấy các cách dùng khác với `del` sau này.

5.3 Bộ và dãy

Chúng ta đã thấy rằng danh sách và chuỗi có nhiều thuộc tính chung, như là có chỉ mục, và các toán tử cắt miếng. Chúng là hai ví dụ của *dãy (sequence)* kiểu dữ liệu. Vì Python là một ngôn ngữ đang phát triển, các kiểu dữ liệu dãy khác có thể được thêm vào. Có một kiểu dãy chuẩn khác: *bộ (tuple)*.

Một tuple gồm một số các giá trị phân cách bởi dấu phẩy, ví dụ:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Khi xuất ra, tuple luôn luôn được kèm giữa hai dấu ngoặc nhằm để cho các tuple lồng nhau có thể được thông dịch chính xác; chúng có thể được nhập vào với ngoặc hoặc không, mặc dù thông thường chúng ta vẫn cần các dấu ngoặc (nếu tuple là một phần của một biểu thức lớn hơn).

Tuple được dùng nhiều. Ví dụ: cặp tọa độ (x, y), bản ghi nhân viên từ cơ sở dữ liệu, v.v... Giống như chuỗi, tuple không thể bị thay đổi: không thể gán giá trị mới cho từng phần tử của tuple (mặc dù bạn có thể đạt được cùng kết quả với cắt miếng và ghép dãy). Cũng có thể tạo tuple chứa các đối tượng khả biến ví dụ như danh sách.

Vấn đề đặc biệt là trong việc tạo nên tuple chứa 0 hoặc một phần tử: cú pháp ngôn ngữ có một vài điểm riêng để thực hiện việc này. Tuple rỗng được tạo nên bởi một cặp ngoặc rỗng; tuple một phần tử được tạo bởi một giá trị theo sau bởi một dấu phẩy (việc cho giá trị đơn lẻ vào trong ngoặc không đủ để tạo tuple). Xấu, nhưng hiệu quả. Ví dụ:

```
>>> empty = ()
>>> singleton = 'hello',    # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Câu lệnh `t = 12345, 54321, 'hello!'` là một ví dụ của *việc đóng gói tuple (tuple packing)*: các giá trị `12345`, `54321` và `'hello!'` được gói lại vào trong một tuple. Và quá trình ngược:

```
>>> x, y, z = t
```

Và nó được gọi là *tháo dãy*. Việc tháo dãy yêu cầu danh sách các biến bên trái có cùng số phần tử như độ lớn của dãy. Chú ý rằng phép đa gán (multiple assignment) thật ra chỉ là sự tổng hợp của việc gói tuple và tháo dãy.

Có một điểm không đối xứng ở đây: việc gói nhiều giá trị luôn luôn tạo một tuple, nhưng phép tháo ra có thể được áp dụng cho mọi dãy (chuỗi, danh sách, tuple).

5.4 Tập hợp

Python cũng có một kiểu dữ liệu cho *tập hợp (set)*. Một tập hợp là một nhóm các phần tử không lặp. Ứng dụng cơ bản bao gồm việc kiểm tra hội viên và bỏ các phần tử trùng lặp. Các đối tượng tập hợp cũng hỗ trợ các toán tử như hợp, giao, hiệu, và hiệu đối xứng.

Đây là một ví dụ ngắn:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)               # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                 # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                 # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                             # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                             # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                             # letters in both a and b
set(['a', 'c'])
>>> a ^ b                             # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

5.5 Từ điển

Một kiểu dữ liệu hữu dụng khác được đưa vào Python là *từ điển (dictionary)*. Từ điển được tìm thấy trong các ngôn ngữ khác như ``bộ nhớ kết hợp (associative memory)'' hoặc ``mảng kết hợp (associative array)''. Không như dãy được chia chỉ mục từ một khoảng số, từ điển được chia chỉ mục từ *các khóa*, có thể là bất kỳ kiểu không đổi nào; chuỗi và số luôn luôn có thể làm khóa. Tuple có thể được

dùng làm khóa nếu nó chỉ chứa chuỗi, số, hoặc tuple; nếu tuple chứa bất kỳ một đối tượng khả biến nào thì nó không thể được dùng làm khóa. Bạn không thể dùng danh sách làm khóa, vì danh sách có thể được thay đổi ngay tại chỗ với phép gán vào chỉ mục, phép gán miếng cắt, hoặc các phương thức khác như `append()` và `extend()`.

Dễ nhất là nghĩ về từ điển như một tập hợp không thứ tự của các bộ *khóa: giá trị* , với điều kiện là khóa phải là duy nhất (trong cùng một từ điển). Một cặp ngoặc nhọn tạo một từ điển rỗng: `{}`. Đặt một loạt các cụm khóa:giá trị phân biệt bởi dấu phẩy vào trong ngoặc nhọn tạo nên các cặp khóa:giá trị ban đầu cho từ điển; đây cũng là cách mà từ điển được xuất ra.

Công việc chính của từ điển là chứa một giá trị vào một khóa nào đó và lấy lại giá trị từ khóa đó. Cũng có thể xóa một cặp khóa:giá trị với `del`. Nếu bạn chứa vào một khóa đã có sẵn, giá trị cũ sẽ bị mất. Lấy giá trị từ một khóa không tồn tại sẽ gây nên lỗi.

Phương thức `keys()` của đối tượng từ điển trả về một danh sách các khóa đã được dùng trong từ điển, theo một thứ tự bất kỳ (nếu bạn muốn chúng được sắp xếp, chỉ cần áp dụng phương thức `sort()` vào danh sách các khóa). Để kiểm tra xem một khóa có trong từ điển hay không, có thể dùng phương thức `has_key()` hoặc từ khóa `in` .

Đây là một ví dụ nhỏ về cách dùng từ điển:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
>>> 'guido' in tel
True
```

Phương thức `dict()` dùng để tạo từ điển trực tiếp từ các danh sách các cụm khóa-giá trị chứa trong tuple. Khi các cụm có một mẫu nào đó, việc gộp danh sách có thể chỉ ra ngắn gọn danh sách khóa-giá trị.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in (2, 4, 6)])     # use a list comprehension
{2: 4, 4: 16, 6: 36}
```

Ở phần sau của bài chỉ dẫn chúng ta sẽ tìm hiểu về các biểu thức bộ tạo thích hợp hơn với việc cung cấp các cặp khóa-giá trị vào hàm khởi tạo `dict()` .

Khi mà khóa là những chuỗi đơn giản, đôi khi nó sẽ dễ hơn nếu chỉ định các cụm bằng thống số từ khóa:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

5.6 Kỹ thuật lặp

Khi lặp qua từ điển, khóa và giá trị tương ứng có thể được lấy ra cùng lúc bằng phương thức `iteritems()` .

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

Khi lặp qua một dãy, vị trí chỉ mục và giá trị tương ứng có thể được lấy ra cùng lúc bằng hàm `enumerate()` .

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
0 tic
1 tac
2 toe
```

Để lặp qua hai hoặc nhiều dãy cùng lúc, các phần tử có thể được ghép với nhau bằng hàm `zip()` .

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your %s?  It is %s.' % (q, a)
...
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

Để lặp qua một dãy theo thứ tự đảo, đầu tiên chỉ định dãy đó theo thứ tự xuôi, rồi gọi hàm `reversed()` .

```
>>> for i in reversed(xrange(1,10,2)):
...     print i
...
```

```
9
7
5
3
1
```

Để lặp qua một dãy theo thứ tự đã sắp xếp, dùng hàm `sorted()` và nó sẽ trả về một danh sách đã sắp xếp trong khi vẫn để danh sách gốc nguyên vẹn.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

5.7 Bàn thêm về điều kiện

Điều kiện dùng trong các câu lệnh `while` và `if` có thể chứa bất kỳ toán tử nào, không chỉ là phép so sánh.

Các toán tử so sánh `in` và `not in` kiểm tra xem giá trị có mặt (hoặc không có mặt) trong một dãy. Toán tử `is` và `is not` so sánh xem hai đối tượng có phải cùng là một đối tượng hay không; việc này chỉ quan trọng đối với các đối tượng khả biến như danh sách. Mọi toán tử so sánh có cùng độ ưu tiên, thấp hơn của các toán tử số.

So sánh có thể được nối với nhau. Ví dụ như, `a < b == c` kiểm tra xem `a` nhỏ hơn `b` và hơn nữa `b` bằng với `c`.

Phép so sánh có thể được ghép với nhau bằng toán tử Boolean `and` và `or`, và kết quả của phép so sánh (hoặc của mọi biểu thức Boolean) có thể được đảo ngược với `not`. Các toán tử này có độ ưu tiên thấp hơn các toán tử so sánh; giữa chúng, `not` có độ ưu tiên cao nhất và `or` thấp nhất, để cho `A and not B or C` tương đương với `(A and (not B)) or C`. Như mọi khi, dấu ngoặc đơn có thể được dùng để cho biết kết cấu đúng ý.

Các toán tử Boolean `and` và `or` còn được gọi là *đoản mạch (short-circuit)* toán tử: toán hạng của chúng được đánh giá từ trái qua phải, và việc định giá dừng lại ngay khi kết quả được xác định. Ví dụ như, nếu `A` và `C` là đúng nhưng `B` là sai, `A and B and C` không định giá biểu thức `C`. Khi dùng như một giá trị chung chung và không phải như một Boolean, giá trị trả về của một toán tử đoản mạch là thông số được định giá cuối cùng.

Có thể gán kết quả của một phép so sánh hoặc một biểu thức Boolean vào một biến. Ví dụ,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Chú ý rằng trong Python, khác với C, phép gán không thể có mặt trong biểu thức. Các lập trình viên C sẽ không hài lòng với việc này, nhưng nó tránh một nhóm lớn các lỗi thường gặp trong chương trình C: nhập vào = trong một biểu thức khi mà == được nhằm tới.

5.8 So sánh dãy và các kiểu khác

Đối tượng dãy có thể được so sánh với đối tượng khác cùng kiểu dãy. Sự so sánh dùng *từ điển* thứ tự: đầu tiên hai phần tử đâu được so sánh, và nếu chúng khác nhau thì kết quả được xác định; nếu chúng bằng nhau thì hai phần tử kế sẽ được so sánh và cứ thế, cho đến cuối một trong hai dãy. Nếu hai phần tử được so sánh lại là hai phần dãy cùng kiểu, phép so sánh từ điển lại được thực hiện đệ quy như vậy. Nếu mọi phần tử trong hai dãy đều bằng nhau thì chúng được coi là bằng nhau. Nếu một dãy là dãy con ban đầu của dãy kia, thì dãy ngắn hơn sẽ là dãy bé hơn. Thứ tự từ điển đối với chuỗi sử dụng thứ tự ASCII cho từng ký tự. Một vài ví dụ về việc so sánh dãy cùng kiểu:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Lưu ý rằng so sánh các đối tượng khác kiểu cũng được chấp nhận. Kết quả có thể đoán được nhưng ngoài ý muốn: các kiểu được xếp theo thứ tự tên của chúng. Do đó, một danh sách (list) luôn nhỏ hơn một chuỗi (string), và một chuỗi luôn nhỏ hơn một tuple, v.v.... [5.1] Các kiểu số lẫn lộn được so sánh theo giá trị của chúng, do đó 0 bằng 0.0, v.v...

Chú thích

... v.v...[5.1]

Không nên dựa vào luật so sánh các đối tượng khác kiểu vì nó có thể bị thay đổi ở phiên bản tương lai của ngôn ngữ.

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách đề nghị thay đổi.

Bài chỉ dẫn Python

6. Mô-đun

Nếu bạn thoát khỏi trình thông dịch và chạy nó lại, những gì bạn đã định nghĩa (hàm và biến) đều bị mất. Do đó, nếu bạn muốn viết một chương trình dài hơn, thì tốt nhất bạn nên dùng một trình soạn thảo để chuẩn bị đầu vào cho trình thông dịch và chạy nó với tập tin vào này. Việc này được gọi là tạo *kịch bản (script)*. Khi chương trình của bạn trở nên dài hơn, bạn sẽ muốn tách nó ra thành nhiều tập tin để dễ duy trì. Bạn sẽ muốn dùng một hàm thuận tiện mà bạn đã viết trong nhiều chương trình mà không cần phải chép lại định nghĩa của nó vào các chương trình đó.

Để hỗ trợ việc này, Python có một cách đặt các định nghĩa vào một tập tin và dùng chúng trong một kịch bản hoặc trong một phiên làm việc với trình thông dịch. Tập tin này được gọi là *mô-đun (module)*; các định nghĩa từ một mô-đun có thể được *nhập (import)* vào các mô-đun khác hoặc vào mô-đun *chính* (tập hợp các biến mà bạn có thể truy cập tới trong một kịch bản được chạy ở cấp cao nhất và trong chế độ máy tính).

Mô-đun là một tập tin chứa các định nghĩa và câu lệnh Python. Tên tập tin là tên của mô-đun với đuôi .py được gắn vào. Trong một mô-đun, tên của mô-đun (là một chuỗi) có thể được truy cập qua một biến toàn cục `__name__`. Ví dụ, dùng trình soạn thảo của bạn để tạo một tập tin đặt tên là fibo.py trong thư mục hiện tại với nội dung sau:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Bây giờ chạy trình thông dịch Python và nhập mô-đun này với dòng lệnh sau:

```
>>> import fibo
```

Việc này sẽ không nhập trực tiếp các tên hàm định nghĩa trong `fibo` vào bảng

ký hiệu (symbol table); nó chỉ nhập tên mô-đun `fibo` mà thôi. Dùng tên mô-đun bạn có thể truy cập các hàm:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Nếu bạn định dùng một hàm thường xuyên, thì bạn có thể gán nó vào một tên cục bộ:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Bàn thêm về mô-đun

Mô-đun có thể chứa các câu lệnh khả thi cũng như các định nghĩa hàm. Các câu lệnh này nhằm mục đích khởi tạo mô-đun. Chúng sẽ chỉ được chạy lần *đầu* mô-đun được nhập ở đâu đó.[6.1]

Mỗi mô-đun có một bảng ký hiệu riêng của nó và được dùng như bảng toàn cục đối với mọi hàm được định nghĩa trong mô-đun. Do đó, tác giả của một mô-đun có thể sử dụng các biến toàn cục trong mô-đun mà không phải lo lắng về việc trùng lặp với các biến toàn cục của người dùng. Mặt khác, nếu bạn biết bạn đang làm gì, bạn có thể truy cập vào các biến toàn cục của mô-đun với cùng một cách dùng để truy cập các hàm của nó. `modname.itemname`.

Mô-đun có thể nhập các mô-đun khác. Thông thường (nhưng không bắt buộc) ta hay để tất cả các lệnh `import` ở đầu một mô-đun (hoặc kịch bản). Các tên của mô-đun bị nhập được đặt trong bảng ký hiệu toàn cục của mô-đun nhập nó.

Có một biến thể của câu lệnh `import` để nhập nhiều tên trực tiếp từ một mô-đun vào trong bảng ký hiệu của mô-đun nhập. Ví dụ:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Câu lệnh này không đưa tên mô-đun bị nhập vào bảng ký hiệu cục bộ (do đó trong ví dụ này, `fibo` chưa được định nghĩa)

Và một biến thể khác để nhập tất cả các tên từ một mô-đun:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Câu lệnh này nhập tất cả mọi tên trừ những tên bắt đầu bằng dấu gạch chân (_).

6.1.1 Đường dẫn tìm mô-đun

Khi mô-đun tên spam được nhập vào, trình thông dịch sẽ tìm một tập tin tên spam.py trong thư mục hiện tại, rồi trong danh sách các thư mục được chỉ định bởi biến môi trường PYTHONPATH. Biến này có cùng cú pháp như là biến môi trường PATH, cùng chứa một danh sách tên các thư mục. Khi PYTHONPATH chưa được thiết lập, hoặc khi tập tin không được tìm thấy, việc tìm kiếm sẽ tiếp tục tìm trong đường dẫn mặc định tùy theo khi cài Python; trên UNIX, đường dẫn này thường là .:/usr/local/lib/python.

Thật ra, mô-đun được tìm trong danh sách các thư mục chỉ định bởi biến sys.path đã được thiết lập từ thư mục chứa kịch bản nguồn (hoặc thư mục hiện tại), PYTHONPATH và các mặc định khi cài đặt. Điều này cho phép các chương trình Python thay đổi hoặc thay thế đường dẫn tìm mô-đun. Lưu ý rằng vì thư mục chứa kịch bản đang chạy nằm trong đường dẫn tìm kiếm, tên của kịch bản nhất thiết phải không trùng với tên các mô-đun chuẩn, nếu không thì Python sẽ cố sức nạp kịch bản như là một mô-đun khi mô-đun đó được nhập vào. Thông thường điều này sẽ gây lỗi. Xem mục 6.2, ``Các mô-đun chuẩn,'' để biết thêm chi tiết.

6.1.2 Các tập tin Python ``đã dịch''

Như một cách quan trọng để tăng tốc quá trình khởi động của các chương trình ngắn có dùng nhiều mô-đun chuẩn, nếu tập tin có tên spam.pyc tồn tại trong thư mục mà spam.py được tìm thấy, tập tin này được giả định là phiên bản đã được ``biên dịch byte'' (byte-compile) của mô-đun spam. Thời gian thay đổi của phiên bản spam.py dùng để tạo spam.pyc được lưu lại trong spam.pyc, và tập tin .pyc sẽ bị bỏ qua nếu chúng không khớp nhau.

Thông thường, bạn không cần làm gì cả để tạo tập tin spam.pyc . Khi nào spam.py được biên dịch thành công, Python sẽ thử ghi phiên bản đã biên dịch ra spam.pyc. Nếu việc ghi này thất bại thì cũng không có lỗi gì xảy ra; nếu vì lý do gì đó mà tập tin không được ghi đầy đủ, tập tin spam.pyc sẽ bị đánh dấu là không hợp lệ và sẽ bị bỏ qua sau này. Nội dung của tập tin spam.pyc không phụ thuộc vào hệ thống, do đó một thư mục mô-đun Python có thể được chia xẻ với nhiều máy trên các kiến trúc khác nhau.

Một vài mẹo cho chuyên gia:

- Khi trình thông dịch Python được chạy với cờ **-O** , mã tối ưu được tạo và lưu trong các tập tin .pyo . Trình tối ưu hóa hiện tại không giúp gì nhiều, nó chỉ bỏ đi các lệnh assert . Khi **-O** được dùng, *tất cả* mã byte (bytecode) đều được tối ưu; .pyc bị bỏ qua và .py được biên dịch ra mã byte tối ưu.

- Truyền hai cờ **-O** vào trình thông dịch Python (**-OO**) sẽ khiến trình biên dịch mã byte thực hiện những tối ưu mà trong những trường hợp hiếm hoi có thể dẫn đến hỏng hóc trong chương trình. Hiện tại chỉ có các chuỗi `__doc__` được bỏ đi khỏi mã byte, làm cho tập tin .pyo gọn hơn. Vì một vài chương trình phụ thuộc vào các chuỗi này, bạn nên chỉ dùng tùy chọn này nếu bạn rõ bạn đang làm gì.

- Một chương trình không chạy nhanh hơn khi nó được đọc từ một tập tin .pyc hay .pyo so với khi nó được đọc từ tập tin .py ; một điểm nhanh hơn duy nhất ở các tập tin .pyc hay .pyo là tốc độ chúng được nạp.

- Khi một kịch bản được chạy bằng cách nhập tên nó ở dòng lệnh, thì mã byte của kịch bản không bao giờ được ghi vào tập tin .pyc hay .pyo . Do đó, thời gian khởi động của một kịch bản có thể được giảm xuống bằng cách chuyển hầu hết mã của nó thành mô-đun và tạo một kịch bản nhỏ để nhập mô-đun đó. Bạn cũng có thể chạy tập tin .pyc hay .pyo trực tiếp từ dòng lệnh.

- Có thể dùng tập tin spam.pyc (hay spam.pyo khi **-O** được dùng) và không có tập tin spam.py của cùng một mô-đun. Việc này được tận dụng để phân phối một thư viện Python dưới một dạng hơi khó để dịch ngược.

- Mô-đun `compileall` có thể tạo các tập tin .pyc (hoặc các tập tin .pyo khi **-O** được dùng) cho mọi mô-đun trong một thư mục.

6.2 Các mô-đun chuẩn

Python có một thư viện các mô-đun chuẩn, được nói tới trong một tài liệu khác, *Tham khảo thư viện Python* (từ nay gọi là ``Tài liệu tham khảo'''). Một vài mô-đun được chuyển thẳng vào trình thông dịch; chúng cung cấp các tác vụ không nằm trong phạm vi chính của ngôn ngữ nhưng cũng được tạo sẵn vì hiệu quả cao hoặc để truy cập vào những chức năng của hệ điều hành ví dụ như các lệnh gọi hệ thống. Tập hợp các mô-đun này là một tùy chọn cấu hình lệ thuộc vào hệ thống. Ví dụ, mô-đun amoeba chỉ có trên các hệ hỗ trợ Amoeba. Cần phải nhắc đến mô-đun: `sys`, được đưa vào trong mọi trình thông dịch Python. Các biến `sys.ps1` và `sys.ps2` định nghĩa những chuỗi được dùng như dấu nhắc chính và phụ:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Hai biến này chỉ được định nghĩa khi trình thông dịch chạy ở chế độ tương tác.

Biến `sys.path` là một danh sách các chuỗi quyết định đường dẫn tìm kiếm các mô-đun của trình thông dịch. Nó được khởi tạo theo đường dẫn mặc định từ biến môi trường PYTHONPATH, hoặc từ một giá trị có sẵn nếu PYTHONPATH không được thiết lập. Bạn có thể sửa nó bằng cách dùng các công cụ trên danh sách:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 dir() hàm

Hàm có sẵn `dir()` được dùng để tìm các tên một mô-đun định nghĩa. Nó trả về một danh sách các chuỗi đã sắp xếp:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__
 '__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',
 'builtin_module_names', 'byteorder', 'callstats', 'copyright',
 'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',
 'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenfl
 'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags
 'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdo
 'version', 'version_info', 'warnoptions']
```

Không có thông số, `dir()` liệt kê các tên bạn đã định nghĩa:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__doc__', '__file__', '__name__', 'a', 'fib', 'fibo',
```

Lưu ý rằng nó liệt kê mọi loại tên: biến, mô-đun, hàm, v.v...

`dir()` không liệt kê tên của các hàm và biến có sẵn. Nếu bạn muốn có danh sách của chúng, thì chúng được định nghĩa trong mô-đun chuẩn `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'DeprecationWarn
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FloatingPointError', 'FutureWarning', 'IOError', 'ImportError',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
```

```
'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UserWarning', 'ValueError', 'Warning', 'WindowsError',
'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
'__name__', 'abs', 'apply', 'basestring', 'bool', 'buffer',
'callable', 'chr', 'classmethod', 'cmp', 'coerce', 'compile',
'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter', 'float',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex',
'id', 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
'len', 'license', 'list', 'locals', 'long', 'map', 'max', 'min',
'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit', 'range',
'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 Gói

Gói (package) là một cách để cấu trúc vùng tên mô-đun của Python bằng cách dùng ``tên mô-đun có chấm''. Ví dụ, tên mô-đun `A.B` chỉ ra mô-đun con tên "`B`" trong một gói tên "`A`". Cũng như việc sử dụng mô-đun giúp tác giả của các mô-đun khác nhau không phải lo lắng về các biến toàn cục của nhau, việc sử dụng tên mô-đun có chấm giúp tác giả của các gói đa mô-đun như NumPy hay Python Imaging Library không phải lo lắng về tên mô-đun của họ.

Giả sử bạn muốn thiết kế một tập hợp các mô-đun (một ``gói'') nhằm vào việc xử lý các tập tin và dữ liệu âm thanh. Có nhiều định dạng tập tin âm thanh (thường được nhận dạng dựa vào phần mở rộng, ví dụ: .wav, .aiff, .au), do đó bạn sẽ cần tạo và duy trì một tập hợp luôn tăng của các mô-đun cho việc chuyển đổi giữa các định dạng khác nhau. Cũng có nhiều tác vụ khác nhau bạn muốn thực hiện với dữ liệu âm thanh (ví dụ như tổng hợp, thêm tiếng vang, áp dụng chức năng làm bằng, tạo ra hiệu ứng nổi), do đó bạn sẽ không ngừng viết một loạt các mô-đun để thực hiện các tác vụ này. Sau đây là một cấu trúc minh họa cho gói của bạn (được trình bày theo cách của một hệ tập tin phân cấp)

```
Sound/                          Top-level package
      __init__.py               Initialize the sound package
      Formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
```

```
Effects/                Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
Filters/                Subpackage for filters
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

Khi nhập một gói, Python tìm trong các thư mục từ `sys.path` để tìm thư mục con của gói.

Các tập tin __init__.py là cần thiết để cho Python biết các thư mục chứa các gói; việc này được đặt ra để tránh các thư mục với tên chung, ví dụ như "`string`", vô tình che mất mô-đun hợp lệ xuất hiện sau trong đường dẫn tìm kiếm. Trong trường hợp đơn giản nhất, __init__.py có thể chỉ là một tập tin rỗng, nhưng nó cũng có thể thực thi các mã thiết lập của gói hoặc thiết lập biến `__all__` , sẽ được nhắc đến sau.

Người dùng gói này có thể nhập từng mô-đun riêng lẻ từ gói, ví dụ:

```
import Sound.Effects.echo
```

Nó sẽ nạp mô-đun con `Sound.Effects.echo`. Nó phải được tham chiếu bằng tên đầy đủ.

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Cách nhập mô-đun con khác là:

```
from Sound.Effects import echo
```

Nó cũng nạp luôn mô-đun con `echo`, và làm cho nó có thể được truy cập mà không cần phần tên gói, do đó nó có thể được dùng như sau:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Một biến thể khác là nhập hàm hoặc biến mình muốn một cách trực tiếp:

```
from Sound.Effects.echo import echofilter
```

Một lần nữa, lệnh này nạp mô-đun con `echo`, nhưng nó làm hàm `echofilter()` có thể được sử dụng trực tiếp:

```
echofilter(input, output, delay=0.7, atten=4)
```

Lưu ý rằng khi sử dụng `from` *package* `import` *item*, *item* có thể hoặc là mô-đun con (hoặc gói con) của gói, hoặc là một tên nào khác được định nghĩa trong

gói, chẳng hạn như một hàm, lớp, hoặc biến. Câu lệnh `import` trước hết kiểm tra xem *item* có được định nghĩa trong gói; nếu không, nó giả định rằng đó là mô-đun và thử nạp nó. Nếu không thể tìm thấy mô-đun, biệt lệ `ImportError` sẽ được nâng.

Ngược lại, khi dùng cú pháp như `import` *item.subitem.subsubitem*, mỗi tiểu mục trừ tiểu mục cuối phải là một gói; tiểu mục cuối có thể là một mô-đun hoặc một gói nhưng không thể là một lớp, hay hàm, hay biến được định nghĩa trong tiểu mục trước.

6.4.1 Nhập * từ một gói

Bây giờ chuyện gì xảy ra khi bạn viết `from Sound.Effects import *`? Tốt nhất, bạn sẽ hy vọng mã này sẽ tìm các mô-đun có trong gói và nhập tất cả chúng vào. Đáng tiếc là tác vụ này không chạy ổn định lắm trên hệ Mac và Windows vì hệ thống tập tin không luôn chứa thông tin chính xác về phân biệt hoa/thường trong tên tập tin. Trên các hệ này, không có cách nào bảo đảm để biết được nếu một tập tin ECHO.PY cần được nhập vào như là một mô-đun `echo`, `Echo` hay `ECHO`. (Ví dụ, Windows 95 có một thói quen là hiện mọi tên tập tin với ký tự đầu tiên viết hoa.) Hạn chế tên 8+3 của DOS cũng tạo ra thêm một vấn đề thú vị với các tên mô-đun dài.

Giải pháp duy nhất là để tác giả gói chỉ định rõ chỉ mục của gói. Câu lệnh `import` dùng cách thức sau: nếu mã __init__.py của gói có định nghĩa một danh sách tên `__all__`, nó sẽ được dùng làm danh sách của các tên mô-đun cần được nhập khi gặp `from` *package* `import *`. Tác giả gói sẽ phải tự cập nhật danh sách này mỗi khi phiên bản mới của gói được phát hành. Tác giả gói cũng có thể không hỗ trợ nó, nếu họ không thấy cách dùng `importing *` là hữu dụng đối với gói của họ. Ví dụ, tập tin Sounds/Effects/__init__.py có thể chứa đoạn mã sau:

```
__all__ = ["echo", "surround", "reverse"]
```

Điều này có nghĩa là `from Sound.Effects import *` sẽ nhập ba mô-đun được chỉ định từ gói `Sound`.

Nếu `__all__` không được định nghĩa, câu lệnh `from Sound.Effects import *` does *không* nhập mọi mô-đun con từ gói `Sound.Effects` vào vùng tên hiện tại; nó chỉ đảm bảo rằng gói `Sound.Effects` đã được nhập (có thể chạy các mã khởi tạo trong __init__.py) và sau đó nhập các tên được định nghĩa trong gói. Nó bao gồm các tên (và mô-đun con đã được nạp) được định nghĩa bởi __init__.py. Nó cũng gồm các mô-đun con của gói đã được nạp bởi câu lệnh `import` trước. Xem đoạn mã này:

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

Trong ví dụ này, các mô-đun `echo` và `surround` được nhập vào vùng tên hiện

tại vì chúng được định nghĩa trong gói `Sound.Effects` khi câu lệnh `from...import` được thực thi. (Cũng sẽ có cùng kết quả khi `__all__` được định nghĩa.)

Lưu ý là bình thường việc nhập * từ một mô-đun hay gói được coi là xấu, vì nó hay dẫn tới các mã khó đọc. Tuy nhiên, khi dùng trong một phiên làm việc tương tác thì nó giúp tiết kiệm công nhập phím, và một vài mô-đun được thiết kế để chỉ xuất các tên theo một vài mẫu cụ thể.

Nhớ rằng, không có gì sai trái với việc dùng `from Package import specific_submodule`! Đúng ra, đây là cách viết được khuyến khích trừ khi mô-đun nhập (importing module) cần dùng mô-đun con có cùng tên ở một gói khác.

6.4.2 Tham chiếu nội trong gói

Các mô-đun con thường cần tham chiếu lẫn nhau. Ví dụ, Mô-đun `surround` có thể sử dụng mô-đun `echo` . Trong thực tế, những tham chiếu này quá phổ biến đến nỗi câu lệnh `import` đầu tiên sẽ tìm trong gói chứa (containing package) trước khi tìm trong đường dẫn tìm kiếm mô-đun chuẩn. Do đó, mô-đun `surround` có thể đơn giản dùng `import echo` hay `from echo import echofilter`. Nếu mô-đun được nhập (imported module) không thể tìm ra trong gói chứa (là gói mà mô-đun hiện tại là một mô-đun con), câu lệnh `import` tìm một mô-đun cấp cao nhất có cùng tên.

Khi các gói được cấu trúc thành các gói con (như gói `Sound` trong ví dụ), không có đường tắt để tham chiếu tới các mô-đun con của các gói kế cận - tên đầy đủ của gói con phải được chỉ định. Ví dụ, nếu mô-đun `Sound.Filters.vocoder` cần dùng mô-đun `echo` trong gói `Sound.Effects` , nó có thể dùng `from Sound.Effects import echo`.

Bắt đầu từ Python 2.5, ngoài việc nhập tương đối hiểu ngầm (implicit relative import) đã nói, bạn có thể viết lệnh nhập tương đối xác định (explicit relative import) với kiểu `from module import name` của câu lệnh nhập. Các lệnh nhập tương đối xác định dùng chấm ở đầu để chỉ định các gói hiện tại và gói cha có mặt trong câu lệnh nhập. Ví dụ từ mô-đun `surround` , bạn sẽ dùng:

```
from . import echo
from .. import Formats
from ..Filters import equalizer
```

Lưu ý cả hai lệnh nhập tương đối xác định và hiểu ngầm đều dựa vào tên của mô-đun hiện tại. Vì tên của mô-đun chính luôn luôn là `"__main__"`, mô-đun được nhằm để dùng như mô-đun chính của một chương trình Python nên luôn luôn cùng câu lệnh nhập tuyệt đối.

6.4.3 Gói trong nhiều thư mục

Gói còn có thêm một thuộc tính đặc biệt khác, `__path__`. Nó được khởi tạo là danh sách chứa tên của thư mục chứa tập tin `__init__.py` của gói trước khi mã trong tập tin đó được thực thi. Biến này có thể được thay đổi; làm như vậy sẽ ảnh hưởng các tìm kiếm mô-đun và gói con chứa trong gói này trong tương lai.

Mặc dù tính năng này không thường được dùng, nó có thể được tận dụng để mở rộng tập hợp các mô-đun tìm thấy trong một gói.

Ghi chú

... đâu đó.[6.1]

[6.1] Thật ra các định nghĩa hàm cũng là `các câu lệnh' được `thực thi'; việc thực thi câu lệnh này nhập tên hàm vào bảng ký hiệu toàn cục của mô-đun.

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách để nghị thay đổi.

Bài chỉ dẫn Python

7. Vào và ra

Có nhiều các để thể hiện đầu ra của một chương trình; dữ liệu có thể được in ra ở dạng người đọc được, hoặc viết vào một tập tin để dùng sau này. Chương này sẽ bàn về một vài khả năng đó.

7.1 Định dạng ra đẹp hơn

Chúng ta đã gặp hai cách để viết giá trị: câu lệnh biểu thức *và* câu lệnh `print` . (Cách thứ ba là dùng phương thức `write()` của đối tượng tập tin; tập tin đầu ra chuẩn có thể được tham chiếu tới theo `sys.stdout`. Xem Tham khảo thư viện Python để biết thêm chi tiết.)

Thông thường bạn sẽ muốn điều khiển cách định dạng đầu ra của bạn nhiều hơn là chỉ đơn giản là in các giá trị phân cách bởi khoảng trắng. Có hai cách để định dạng đầu ra của bạn; cách thứ nhất là bạn tự xử lý các chuỗi; dùng phép cắt miếng của chuỗi và phép ghép chuỗi bạn có thể tạo bất kỳ bố cục nào bạn có thể nghĩ ra. Mô-đun chuẩn `string` chứa một vài công cụ để đệm chuỗi khít vào chiều ngang cột; chúng ta sẽ xem xét nó lát nữa. Cách thứ hai là dùng toán tử `%` với một chuỗi là thông số bên trái. Toán tử `%` thông dịch thông số trái như là một chuỗi định dạng kiểu `sprintf()` (như trong C) để áp dụng vào thông số bên phải; và trả về một chuỗi từ tác vụ định dạng này.

Một câu hỏi vẫn còn đó: làm sao để chuyển giá trị thành chuỗi? May mắn thay Python có một cách để chuyển bất kỳ giá trị nào thành chuỗi: truyền nó vào hàm `repr()` hay hàm `str()` . Dấu nháy ngược (``) tương đương với `repr()`, nhưng chúng không còn được dùng trong mã Python mới, và rất có thể sẽ không còn trong các phiên bản ngôn ngữ sau này.

`str()` nhằm để trả về cách thể hiện giá trị một cách dễ đọc, trong khi `repr()` nhằm để tạo cách thể hiện mà trình thông dịch có thể đọc (hoặc sẽ sinh lỗi `SyntaxError` nếu không có cú pháp tương đương). Đối với các đối tượng không có cách thể hiện cho người đọc, `str()` sẽ trả về cùng giá trị như `repr()`. Nhiều giá trị, ví dụ như số, hoặc cấu trúc như danh sách và từ điển, có cùng cách thể hiện với cả hai hàm này. Riêng chuỗi và số chấm động có hai cách thể hiện khác biệt.

Một vài ví dụ:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
```

```
>>> str(0.1)
'0.1'
>>> repr(0.1)
'0.10000000000000001'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs')))
"(32.5, 40000, ('spam', 'eggs'))"
>>> # reverse quotes are convenient in interactive sessions:
... `x, y, ('spam', 'eggs')`
"(32.5, 40000, ('spam', 'eggs'))"
```

Đây là hai cách để viết một bảng bình phương và lập phương:

```
>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...     # Note trailing comma on previous line
...     print repr(x*x*x).rjust(4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1,11):
...     print '%2d %3d %4d' % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Chú ý một khoảng trắng được thêm vào giữa các cột theo cách hoạt động của `print` : nó luôn luôn thêm khoảng trắng giữa các thông số.)

Ví dụ này biểu diễn phương thức `rjust()` của các đối tượng chuỗi, nó canh phải một chuỗi vào trong trường với độ rộng xác định bằng cách thêm khoảng trống vào bên trái. Các phương thức tương tự khác gồm `ljust()` và `center()`. Các phương thức này không viết gì cả, chúng chỉ trả về một chuỗi mới. Nếu chuỗi nhập vào quá dài, chúng cũng không cắt nó đi, mà trả về nó nguyên vẹn; điều này thường sẽ phá hỏng bố cục của bạn, nhưng vẫn tốt hơn là nói dối về một giá trị. (Nếu bạn thật sự muốn cắt bỏ bạn có thể thêm phép cắt miếng, như "`x.ljust(n)[:n]`".)

Có một phương thức khác, `zfill()`, nó đệm không vào bên trái một chuỗi số. Nó hiểu các dấu cộng và trừ:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

Dùng toán tử % sẽ giống như vậy:

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

Nếu có nhiều hơn một định dạng trong chuỗi, bạn cần truyền một bộ ở toán hạng bên phải, như trong ví dụ này:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '%-10s ==> %10d' % (name, phone)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

Hầu hết các định dạng hoạt động như trong C và yêu cầu bạn truyền kiểu thích hợp; nếu không, bạn sẽ nhận biệt lệ thay vì đổ nhân (core dump). Định dạng `%s` dễ dãi hơn: nếu thông số tương ứng không phải là một đối tượng chuỗi, nó sẽ được chuyển thành chuỗi qua hàm có sẵn `str()` . Việc dùng * để truyền vào độ rộng và mức chính xác như là một thông số nguyên riêng cũng được hỗ trợ. Các định dạng C `%n` và `%p` không được hỗ trợ.

Nếu bạn có một chuỗi định dạng rất dài và không muốn cắt ra, có thể bạn sẽ muốn tham chiếu tới các biến sắp được định dạng qua tên, thay vì vị trí. Việc này có thể được thực hiện theo dạng `%(name)format`, như ví dụ sau:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
```

```
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % table
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Cách này đặc biệt hữu dụng khi đi kèm với hàm có sẵn `vars()` mới, nó trả về một từ điển chứa tất cả các biến cục bộ.

7.2 Đọc và viết tập tin

`open()` trả về một đối tượng tập tin, và thường được dùng với hai thông số: "`open(`*filename*`, `*mode*`)`".

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

Thông số thứ nhất là một chuỗi chứa tên tập tin. Thông số thứ hai là một chuỗi khác chứa một vài ký tự xác định cách thức tập tin sẽ được dùng. *mode* có thể là `'r'` khi tập sẽ chỉ được đọc, `'w'` chỉ được ghi (tập tin cùng tên đang có sẽ bị xóa), và `'a'` mở tập tin để thêm vào cuối; mọi dữ liệu ghi vào tập tin sẽ được tự động thêm vào cuối. `'r+'` mở tập tin để đọc và ghi. Thông số *mode* là không bắt buộc; `'r'` sẽ được giả định nếu nó bị bỏ qua.

Trong Windows và Macintosh, `'b'` thêm vào *mode* mở tập tin ở chế độ nhị phân, cho nên cũng có các chế độ khác như `'rb'`, `'wb'`, và `'r+b'`. Windows phân biệt rõ các tập tin văn bản và nhị phân; ký tự hết dòng (end-of-line) trong các tập tin văn bản được tự động thay đổi một chút khi dữ liệu được đọc hay ghi. Việc thay đổi sau bức bình phong (behind-the-scene) như vậy không ảnh hưởng các tập tin văn bản ASCII, nhưng nó sẽ phá dữ liệu nhị phân như trong các tập tin JPEG hay hàm EXE . Cần cẩn thận dùng chế độ nhị phân khi đọc và ghi các tập tin như vậy.

7.2.1 Phương thức của đối tượng tập tin

Các ví dụ trong mục này sẽ giả sử một đối tượng tập tin `f` đã được tạo.

Để đọc nội dung tập tin, gọi `f.read(`*size*`)`, nó đọc một số lượng dữ liệu và trả về một chuỗi. *size* là một thông số số nguyên không bắt buộc. Khi *size* bị bỏ qua hoặc âm, toàn bộ nội dung tập tin sẽ được đọc và trả về; bạn sẽ gặp vấn đề nếu tập tin lớn gấp đôi bộ nhớ của máy bạn. Ngược lại, nhiều nhất *size* byte sẽ được đọc và trả về. Nếu đã đến cuối tập tin, `f.read()` sẽ trả về một chuỗi rỗng (`""`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` đọc một dòng từ tập tin; ký tự dòng mới (`\n`) được giữ lại ở cuối chuỗi, và sẽ chỉ bị bỏ qua ở dòng cuối của tập tin nếu tập tin không kết thúc bằng một dòng mới. Điều này làm giá trị trả về rõ ràng; nếu `f.readline()` trả về một chuỗi rỗng có nghĩa là đã đụng cuối tập tin, trong khi một dòng trống thì

được biểu diễn bởi `'\n'`, một chuỗi chỉ chứa duy nhất một ký tự dòng mới.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

`f.readlines()` trả về một danh sách tất cả các dòng trong tập tin. Nếu truyền một tham số không bắt buộc *sizehint*, nó sẽ đọc nhiêu đó byte từ tập tin và thêm một chút đủ để hoàn tất một dòng, và trả về các dòng đã đọc được. Điều này thường được dùng để đọc một cách hiệu quả từng dòng một trong một tập tin lớn mà không cần phải nạp toàn bộ tập tin vào bộ nhớ. Chỉ có các dòng toàn vẹn mới được trả về.

```
>>> f.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

Một cách khác để đọc các dòng là lặp qua đối tượng tập tin. Nó rất tiết kiệm bộ nhớ, nhanh, và có mã đơn giản:

```
>>> for line in f:
        print line,

This is the first line of the file.
Second line of the file
```

Cách này đơn giản hơn nhưng không cho bạn điều khiển cách đọc. Vì hai cách này quản lý bộ đệm dòng khác nhau, chúng không nên được dùng chung.

`f.write(`*string*`)` viết nội dung của *string* vào tập tin, trả về `None`.

```
>>> f.write('This is a test\n')
```

Để viết một thứ khác không phải là chuỗi, nó sẽ cần được chuyển thành một chuỗi trước:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

`f.tell()` trả về một số nguyên cho biết vị trí hiện tại của đối tượng tập tin, tính theo byte từ đầu tập tin. Để di chuyển vị trí, dùng "`f.seek(`*offset*`, `*from_what*`)`". Vị trí được tính từ tổng của *offset* và điểm tham chiếu; điểu tham chiếu được xác định bởi thông số *from_what* . Giá trị *from_what* 0 tính từ đầu tập tin, 1 dùng vị trí hiện tại, và 2 tính từ vị trí cuối tập tin. *from_what* có thể bị bỏ qua và mặc định là 0, điểm tham chiếu là đầu tập tin.

```
>>> f = open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Go to the 6th byte in the file
```

```
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

Khi bạn đã dùng xong, gọi `f.close()` để đóng nó lại và giải phóng tài nguyên hệ thống đã sử dụng khi mở tập tin. Sau khi gọi `f.close()`, mọi cách dùng đối tượng tập tin sẽ tự động thất bại.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Các đối tượng tập tin có thêm các phương thức phụ như `isatty()` và `truncate()` không được thường xuyên dùng; tham khảo tài liệu thư viện để biết thêm về các đối tượng tập tin.

7.2.2 `pickle` mô-đun

Các chuỗi có thể được ghi hoặc đọc dễ dàng từ một tập tin. Các số cần một ít cố gắng hơn, vì phương thức `read()` chỉ trả về chuỗi, và cần được truyền vào một hàm như `int()`, nó sẽ nhận một chuỗi như `'123'` và trả về giá trị số 123 của nó. Tuy nhiên, khi bạn muốn lưu các kiểu dữ liệu phức tạp hơn như danh sách, từ điển, hoặc các đối tượng, việc này trở nên rắc rối hơn nhiều.

Thay vì để người dùng luôn viết và gỡ rối mã để lưu các kiểu dữ liệu phức tạp, Python cung cấp một mô-đun chuẩn gọi là `pickle`. Đây là một mô-đun tuyệt diệu có thể nhận hầu hết mọi đối tượng Python (ngay cả một vài dạng mã Python!), và chuyển nó thành một chuỗi; quá trình này được gọi là *giầm (pickling)*. Tạo lại đối tượng từ một chuỗi được gọi là *vớt (unpickling)*. Giữa việc giầm và vớt, biểu diễn dạng chuỗi của đối tượng có thể được lưu vào tập tin, hoặc gửi qua mạng đến một máy ở xa.

Nếu bạn có một đối tượng `x`, và một đối tượng tập tin `f` đã được mở để ghi vào, cách đơn giản nhất để giầm đối tượng chỉ cần một dòng mã:

```
pickle.dump(x, f)
```

Để vớt đối tượng ra, nếu `f` là một đối tượng tập tin đã được mở để đọc:

```
x = pickle.load(f)
```

(Có những biến thể khác, dùng khi giầm nhiều đối tượng hoặc khi bạn không muốn viết dữ liệu đã giầm vào tập tin; tham khảo toàn bộ tài liệu về `pickle` trong *Tham khảo thư viện Python*.)

`pickle` là cách chuẩn để làm cho các đối tượng Python có thể được lưu và dùng lại bởi các chương trình khác, hoặc bởi lần chạy khác của cùng chương trình;

thuật ngữ trong ngành gọi là đối tượng *bền* . Vì `pickle` được sử dụng rộng rãi, nhiều tác giả khi mở rộng Python đã cẩn thận để đảm bảo rằng các kiểu dữ liệu mới ví dụ như ma trận có thể được giầm và vớt đúng đắn.

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách đề nghị thay đổi.

Bài chỉ dẫn Python

8. Lỗi và biệt lệ

Đến bây giờ chũng ta cũng chỉ mới nhắc đến các thông điệp lỗi, nhưng nếu bạn đã thử qua các ví dụ thì bạn có thể đã gặp nhiều hơn. Có (ít nhất) hai loại lỗi khác biệt: *lỗi cú pháp* và *biệt lệ*.

8.1 Lỗi cú pháp

Lỗi cú pháp, còn biết đến như lỗi phân tích (parsing error), có lẽ là phàn nàn lớn nhất bạn gặp phải khi vẫn đang học Python:

```
>>> while True print 'Hello world'
  File "<stdin>", line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

Bộ phân tích lặp lại dòng gây lỗi và hiển thị một mũi tên nhỏ trỏ vào điểm đầu tiên lỗi được phát hiện. Lỗi nằm ở dấu hiệu *phía trước* mũi tên: trong ví dụ trên, lỗi được phát hiện ở từ khóa `print`, vì thiếu một dấu hai chấm (`":"`) ở trước đó. Tên tập tin vào số dòng được hiển thị để bạn biết tìm lỗi ở chỗ nào nếu đầu vào là từ một kịch bản.

8.2 Biệt lệ

Cho dù một câu lệnh hoặc biểu thức là đúng đắn, nó vẫn có thể tạo lỗi khi thực thi. Những lỗi bị phát hiện trong lúc thực thi được gọi là *biệt lệ* và không tai hại một cách vô điều kiện: bạn sẽ học cách xử lý chúng trong các chương trình Python. Hầu hết các biệt lệ đều được xử lý bởi chương trình và dẫn đến kết quả là các thông điệp lỗi như ở đây:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Dòng cuối cùng của thông điệp lỗi cho biết chuyện gì xảy ra. Biệt lệ có nhiều kiểu, và kiểu được hiển thị như là một phần của thông điệp: các kiểu trong ví dụ là `ZeroDivisionError`, `NameError` và `TypeError`. Chuỗi được hiển thị như là kiểu biệt lệ là tên của biệt lệ có sẵn vừa xảy ra. Điều này đúng với tất cả các biệt lệ có sẵn,

nhưng không nhất thiết đúng với các biệt lệ do người dùng định nghĩa (mặc dù đó là một quy ước hữu dụng). Các tên biệt lệ chuẩn là những từ định danh có sẵn (không phải là từ khóa).

Phần còn lại cho biết chi tiết về kiểu biệt lệ và chuyện gì gây ra nó.

Phần trước của thông điệp lỗi cho biết hoàn cảnh khi xảy ra biệt lệ, ở dạng lần ngược ngăn xếp (stack traceback). Bình thường nó chứa một lần ngược ngăn xếp liệt kê các dòng nguồn; tuy nhiên, nó sẽ không hiển thị các dòng đọc từ đầu vào chuẩn.

Tham khảo thư viện Python liệt kê các biệt lệ có sẵn và ý nghĩa của chúng.

8.3 Xử lý biệt lệ

Chúng ta có thể viết những chương trình xử lý những biệt lệ được chọn. Hãy xem ví dụ sau, nó yêu cầu người dùng nhập vào dữ liệu cho tới khi một số nguyên được nhập, nhưng cũng cho phép người dùng ngưng chương trình (dùng `Control-C` hoặc phím tắt khác mà hệ điều hành hỗ trợ); lưu ý rằng sự ngắt quãng do người dùng tạo nên được đánh dấu bởi việc nâng biệt lệ `KeyboardInterrupt` .

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops!  That was no valid number.  Try again..."
...
```

`try` (câu lệnh) hoạt động như sau.

- Trước tiên, *vế try* (các câu lệnh giữa từ khóa `try` và `except`) được thực thi.
- Nếu không có biệt lệ nào xảy ra, vế *except* được bỏ qua và câu lệnh `try` kết thúc.
- Nếu trong khi thực thi vế try xảy ra biệt lệ, phần còn lại của vế được bỏ qua. Sau đó nếu kiểu biệt lệ hợp với kiểu được chỉ định sau từ khóa `except` , thì vế except được thực thi, và rồi việc thực thi tiếp tục sau câu lệnh `try` .
- Nếu biệt lệ xảy ra không hợp với biệt lệ được chỉ định ở vế except, nó sẽ được truyền ra các câu lệnh `try` bên ngoài; nếu không có đoạn mã xử lý nào, nó là một *biệt lệ không được xử lý* và việc thực thi dừng lại với một thông báo như trên.

A `try` (câu lệnh) có thể có nhiều hơn một vế except, để chỉ rõ cách xử lý cho những biệt lệ khác nhau. Nhiều nhất là một đoạn xử lý (handler) sẽ được thực thi. Các đoạn xử lý chỉ xử lý biệt lệ xảy ra trong vế try tương ứng, không xử lý các biệt lệ trong các đoạn xử lý khác của cùng câu lệnh `try` . Vế except có thể định danh nhiều biệt lệ trong một bộ (tuple), ví dụ:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Vế except cuối cùng có thể bỏ qua tên biệt lệ, có tác dụng như là một thay thế (wildcard). Phải hết sức cẩn trọng khi dùng nó, vì nó có thể dễ dàng che đi lỗi lập trình thật! Nó cũng có thể được dùng để in thông báo lỗi và sau đó nâng biệt lệ lại (re-raise exception) (nhằm cho phép nơi gọi xử lý biệt lệ):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError, (errno, strerror):
    print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

`try` ... `except` (câu lệnh) có một *vế else*không bắt buộc, mà khi có mặt sẽ phải đi sau tất cả các vế except. Nó dùng cho mã sẽ được thực thi nếu vế try không nâng biệt lệ nào. Ví dụ:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

Việc dùng vế `else` tốt hơn là thêm mã vào vế `try` vì nó tránh việc vô tình bắt một biệt lệ không được nâng từ mã được bảo vệ trong câu lệnh `try` ... `except` .

Khi một biệt lệ xảy ra, nó có thể có một giá trị gắn liền, còn được biết đến như là *thông số*của biệt lệ. Sự có mặt và kiểu của thông số phụ thuộc vào kiểu biệt lệ.

Vế except có thể chỉ định một biến sau một (hoặc một bộ) tên biệt lệ. Biến đó được gán với một trường hợp biệt lệ (exception instance) với các thông số chứa trong `instance.args`. Để thuận tiện, trường hợp biệt lệ định nghĩa `__getitem__` và `__str__` để cho các thông số có thể được truy xuất và in ra trực tiếp mà không phải tham chiếu `.args`.

Nhưng việc dùng `.args` đã không được khuyến khích. Thay vào đó, cách dùng tốt nhất là truyền một thông số đơn lẻ vào một biệt lệ (có thể là một bộ nếu có nhiều thông số) và gán nó vào thuộc tính `message` . Ta cũng có thể tạo một biệt lệ trước và thêm các thuộc tính vào nó trước khi nâng.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception, inst:
...     print type(inst)     # the exception instance
...     print inst.args      # arguments stored in .args
```

```
...     print inst              # __str__ allows args to printed directly
...     x, y = inst             # __getitem__ allows args to be unpacked directly
...     print 'x =', x
...     print 'y =', y
...
<type 'instance'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Nếu biệt lệ có một thông số, nó sẽ được in ra như là phần cuối (`chi tiết') của thông điệp của những biệt lệ không được xử lý.

Các phần xử lý biệt lệ không chỉ xử lý các biệt lệ xảy ra ngay trong vế try, mà còn xử lý cả biệt trong những hàm được gọi (trực tiếp hoặc gián tiếp) trong vế try. Ví dụ:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError, detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero
```

8.4 Nâng biệt lệ

`raise` (câu lệnh) cho phép nhà lập trình ép xảy ra một biệt lệ được chỉ định. Ví dụ:

```
>>> raise NameError, 'HiThere'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```

Thông số đầu tiên cho `raise` chỉ định biệt lệ sẽ được nâng. Thông số (tùy chọn) thứ hai chỉ định thông số của biệt lệ. Hoặc là, các dòng trên có thể được viết `raise NameError('HiThere')`. Cả hai dạng đều đúng, nhưng người ta có vẻ chuộng dạng thứ hai hơn.

Nếu bạn cần xác định xem một biệt lệ có được nâng chưa nhưng không định xử lý nó, dạng đơn giản hơn của câu lệnh `raise` cho phép bạn nâng lại (re-raise) biệt lệ:

```
>>> try:
...     raise NameError, 'HiThere'
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

8.5 Biệt lệ tự định nghĩa

Các chương trình có thể đặt tên biệt lệ riêng bằng cách tạo một lớp biệt lệ mới. Các biệt lệ thường nên kế thừa từ lớp `Exception` , trực tiếp hoặc gián tiếp. Ví dụ:

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError, e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Trong ví dụ này, mặc định `__init__` của `Exception` đã được định nghĩa lại. Cách thức mới chỉ đơn giản tạo thuộc tính *value* . Nó thay thế cách thức mặc định tạo thuộc tính *args* .

Các lớp biệt lệ có thể được định nghĩa để làm bất kỳ việc gì như các lớp khác, nhưng chúng thường là đơn giản và chỉ cung cấp một số thuộc tính để chứa thông tin về lỗi cho các phần xử lý biệt lệ. Khi tạo một mô-đun mà có thể nâng vài lỗi khác biệt, cách thông thường là tạo một lớp cơ sở cho các biệt lệ được định nghĩa bởi mô-đun đó, và kế thừa từ đó để tạo những lớp biệt lệ cụ thể cho những trường hợp lỗi khác nhau:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
```

```
            previous -- state at beginning of transition
            next -- attempted new state
            message -- explanation of why the specific transition is not allowed
        """

        def __init__(self, previous, next, message):
            self.previous = previous
            self.next = next
            self.message = message
```

Đa số biệt lệ được định nghĩa với tên tận cùng bằng ``Error'', tương tự như cách đặt tên của các biệt lệ chuẩn.

Nhiều mô-đun chuẩn định nghĩa biệt lệ riêng cho chúng để thông báo những lỗi mà có thể xảy ra trong các hàm chúng định nghĩa. Thông tin thêm về các lớp được trình bày trong chương 9, ``Lớp''.

8.6 Định nghĩa cách xử lý

`try` (câu lệnh) có một vế không bắt buộc khác với mục đích định nghĩa những tác vụ dọn dẹp (clean-up action) mà sẽ được thực hiện trong mọi trường hợp. Ví dụ:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

A *vế finally* luôn được thực thi trước khi rời khỏi câu lệnh `try` , cho dù có xảy ra biệt lệ hay không. Khi một biệt lệ đã xảy ra trong vế `try` và không được xử lý bởi vế `except` (hoặc nó đã xảy ra trong một vế `except` hay `else`), nó sẽ được nâng lại sau khi vế `finally` đã được thực thi. Vế `finally` cũng được thực thi ``trên đường ra'' khi bất kỳ vế nào của câu lệnh `try` được bỏ lại thông qua câu lệnh `break`, `continue` hay `return` . Một ví dụ phức tạp hơn:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
```

```
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Như bạn có thể thấy, vế `finally` được thực thi trong mọi trường hợp. `TypeError` được nâng vì chia hai chuỗi không được xử lý bởi vế `except` và vì thế nên được nâng lại sau khi vế `finally` đã được thực thi.

Trong các ứng dụng thực thế, vế `finally` được dùng để trả lại những tài nguyên ngoài (như tập tin, hoặc kết nối mạng), cho dù việc sử dụng tài nguyên có thành công hay không.

8.7 Định nghĩa xử lý có sẵn

Một số đối tượng định nghĩa các tác vụ dọn dẹp chuẩn để thực thi khi một đối tượng không còn được cần đến, cho dù việc xử dụng đối tượng là thành công hay thất bại. Xem qua ví dụ sau, nó thử mở một tập tin và viết nội dung của nó ra màn hình.

```
for line in open("myfile.txt"):
    print line
```

Vấn đề với đoạn mã trên là nó để tập tin ngỏ trong một thời gian không xác định sau khi đoạn mã đã kết thúc. Đây không phải là vấn đề gì trong các đoạn kịch bản đơn giản, nhưng có thể là một vấn đề phức tạp đối với các ứng dụng lớn hơn. Câu lệnh `with` cho phép các đối tượng như tập tin được dùng theo một cách đảm bảo chúng sẽ được dọn dẹp đúng lúc và đúng đắn.

```
with open("myfile.txt") as f:
    for line in f:
        print line
```

Sau khi câu lệnh được thực thi, tập tin *f* luôn được đóng lại, cho dù gặp phải vấn đề trong khi xử lý các dòng. Các đối tượng khác mà cung cấp những tác vụ dọn dẹp định nghĩa sẵn sẽ cho biết về điểm này trong tài liệu của chúng.

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách đề nghị thay đổi.

Bài chỉ dẫn Python

9. Lớp

Chỉ cần một ít cú pháp và từ khóa mới, Python đã có thể hỗ trợ lớp. Nó là sự trộn lẫn giữa C++ và Modula-3. Cũng như mô-đun, các lớp tron Python không đặt rào cản tuyệt đối giữa định nghĩa lớp và người sử dụng, mà thay vào đó nó dựa vào sự lịch thiệp trong cách dùng mà ``không phá định nghĩa.'' Tuy nhiên, các tính năng quan trọng nhất của lớp vẫn được giữ lại trọn vẹn: cách kế thừa lớp hỗ trợ nhiều lớp cơ sở, lớp con có thể định nghĩa lại bất kỳ phương thức nào của các lớp cơ sở của nó, và một phương thức có thể gọi một phương thức cùng tên của một lớp cơ sở. Các đối tượng có thể chứa một lượng dũ liệu riêng bất kỳ.

Theo thuật ngữ C++, mọi thành viên lớp (kể cả thành viên dữ liệu) là *public*(công cộng), và mọi thành viên hàm là *virtual*(ảo). Không có bộ khởi tạo (constructor) hoặc bộ hủy (destructor) đặc biệt. Cũng như Modula-3, không có cách viết tắt nào để tham chiếu tới các thành viên của một đối tượng từ các phương thức của nó: hàm phương thức được khai báo với thông số thứ nhất thể hiện chính đối tượng đó, và được tự động truyền vào qua lệnh gọi. Như trong Smalltalk, các lớp cũng là các đối tượng theo một nghĩa rộng: trong Python, mọi kiểu dữ liệu là đều là các đối tượng. Điều này cho phép nhập (import) và đổi tên. Không như C++ và Modula-3, các kiểu có sẵn có thể được dùng như các lớp cơ sở để mở rộng bởi người dùng. Và như trong C++ nhưng không giống Modula-3, đa số các toán tử có sẵn với cú pháp đặc biệt (các toán tử số học, truy cập mảng, v.v...) có thể được định nghĩa lại trong các trường hợp cụ thể của lớp.

9.1 Vài lời về thuật ngữ

Những từ chuyên ngành dùng ở đây theo từ vựng của Smalltalk và C++.

Các đối tượng có tính cá thể (individuality), và nhiều tên (trong nhiều phạm vi, scope) có thể được gắn vào cùng một đối tượng. Trong các ngôn ngữ khác được gọi là tên lóng (alias). Nó thường không được nhận ra khi dùng Python lần đầu, và có thể được bỏ qua khi làm việc với các kiểu bất biến cơ bản (số, chuỗi, bộ). Tuy nhiên, tên lóng có một ảnh hưởng đối với ý nghĩa của mã Python có sử dụng các đối tượng khả biến như danh sách, từ điển, và đa số các kiểu thể hiện các vật ngoài chương trình (tập tin, cửa sổ, v.v...). Nó thường được dùng vì tên lóng có tác dụng như là con trỏ theo một vài khía cạnh nào đó. Ví dụ, truyền một đối tượng vào một hàm rẻ vì chỉ có con trỏ là được truyền, và nếu một hàm thay đổi một đối tượng được truyền vào, thì nơi gọi sẽ thấy các thay đổi đó -- thay vì cần hai kiểu truyền thông số như trong Pascal.

9.2 Phạm vi trong Python và vùng tên

Trước khi giới thiệu lớp, chúng ta sẽ cần hiểu phạm vi (scope) và vùng tên (namespace) hoạt động như thế nào vì các định nghĩa lớp sẽ sử dụng chúng. Kiến thức về vấn đề này cũng rất hữu dụng với những nhà lập trình Python chuyên nghiệp.

Bắt đầu với một vài định nghĩa.

A *namespace (vùng tên)* (vùng tên) là ánh xạ từ tên vào đối tượng. Đa số các vùng tên được cài đặt bằng từ điển Python, nhưng điều đó thường là không quan trọng (trừ tốc độ), và có thể sẽ thay đổi trong tương lai. Các ví dụ vùng tên như: tập hợp các tên có sẵn (các hàm như `abs()`, và các tên biệt lệ có sẵn); các tên toàn cụ trong một mô-đun; các tên nội bộ trong một phép gọi hàm. Theo nghĩa đó tập hợp các thuộc tính của một đối tượng cũng là một vùng tên. Điều quan trọng cần biết về vùng tên là tuyệt đối không có quan hệ gì giữa các vùng tên khác nhau; ví dụ hai mô-đun khác nhau có thể cùng định nghĩa hàm ``maximize'' mà không sợ lẫn lộn -- người dùng mô-đun phải thêm tiền tố tên mô-đun trước khi gọi hàm.

Cũng xin nói thêm là từ *thuộc tính* được dùng để chỉ mọi tên theo sau dấu chấm -- ví dụ, trong biểu thức `z.real`, `real` là một thuộc tính của đối tượng `z`. Nói đúng ra, tham chiếu tới tên trong một mô-đun là các tham chiếu tới thuộc tính: trong biểu thức `modname.funcname`, `modname` là một đối tượng mô-đun và `funcname` là một thuộc tính của nó. Trong trường hợp này, việc ánh xạ giữa các thuộc tính của mô-đun và các tên toàn cục được định nghĩa trong mô-đun thật ra rất đơn giản: chúng dùng chung một vùng tên! [9.1]

Thuộc tính có thể là chỉ đọc, hoặc đọc ghi. Trong trường hợp sau, phép gán vào thuộc tính có thể được thực hiện. Các thuộc tính mô-đun là đọc ghi: bạn có thể viết "`modname.the_answer = 42`". Các thuộc tính đọc ghi cũng có thể được xóa đi với câu lệnh `del` . Ví dụ, "`del modname.the_answer`" sẽ xóa thuộc tính `the_answer` từ đối tượng tên `modname`.

Các vùng tên được tạo ra vào những lúc khác nhau và có thời gian sống khác nhau. Vùng tên chứa các tên có sẵn được tạo ra khi trình thông dịch Python bắt đầu, và không bao giờ bị xóa đi. Vùng tên toàn cục của một mô-đun được tạo ra khi định nghĩa mô-đun được đọc; bình thường, vùng tên mô-đun cũng tồn tại cho tới khi trình thông dịch thoát ra. Các câu lệnh được thực thi bởi lời gọi ở lớp cao nhất của trình thông dịch, vì đọc từ một kịch bản hoặc qua tương tác, được coi như một phần của mô-đun gọi là `__main__`, cho nên chúng cũng có vùng tên riêng. (Các tên có sẵn thật ra cũng tồn tại trong một mô-đun; được gọi là `__builtin__`.)

Vùng tên nội bộ của một hàm được tạo ra khi hàm được gọi, và được xóa đi khi hàm trả về, hoặc nâng một biệt lệ không được xử lý trong hàm. Dĩ nhiên, các lời gọi hàm đệ quy có vùng tên riêng của chúng.

A *phạm vi* là một vùng văn bản của một chương trình Python mà một vùng tên có thể được truy cập trực tiếp. ``Có thể truy cập trực tiếp'' có nghĩa là một tham chiếu không đầy đủ (unqualifed reference) tới một tên sẽ thử tìm tên đó trong vùng tên.

Mặc dù phạm vi được xác định tĩnh, chúng được dùng một cách động. Vào bất kỳ một lúc nào, có ít nhất ba phạm vi lồng nhau mà vùng tên của chúng có thể được truy cập trực tiếp: phạm vi bên trong cùng, được tìm trước, chứa các tên nội bộ; các vùng tên của các hàm chứa nó, được tìm bắt đầu từ phạm vi chứa nó gần nhất (nearest enclosing scope); phạm vi giữa (middle scope), được tìm kế, chứa các tên toàn cục của mô-đun; và phạm vi ngoài cùng (được tìm sau cùng) là vùng tên chứa các tên có sẵn.

Nếu một tên được khai báo là toàn cục, thì mọi tham chiếu hoặc phép gán sẽ đi thẳng vào phạm vi giữa chứa các tên toàn cục của mô-đun. Nếu không, mọi biến được tìm thấy ngoài phạm vi trong cùng chỉ có thể được đọc (nếu thử khi vào các biến đó sẽ tạo một biến cục bộ *mới* trong phạm vi trong vùng, và không ảnh hưởng tới biến cùng tên ở phạm vi ngoài).

Thông thường, phạm vi nội bộ tham chiếu các tên nội bộ của hàm hiện tại (dựa vào văn bản). Bên ngoài hàm, phạm vi nội bộ tham chiếu cùng một vùng tên như phạm vi toàn cục: vùng tên của mô-đun. Các định nghĩa lớp đặt thêm một vùng tên khác trong phạm vi nội bộ.

Điểm quan trọng cần ghi nhớ là phạm vi được xác định theo văn bản: phạm vi toàn cục của một hàm được định nghĩa trong một mô-đun là vùng tên của mô-đun đó, cho dù mô-đun đó được gọi từ đâu, hoặc được đặt tên lóng nào. Mặt khác, việc tìm tên được thực hiện lúc chạy -- tuy nhiên, định nghĩa ngôn ngữ đang phát triển theo hướng xác định tên vào lúc ``dịch'', cho nên đừng dựa vào việc tìm tên động! (Thực ra thì các biến nội bộ đã được xác định tĩnh.)

Một điểm ngộ của Python là các phép gán luôn gán vào phạm vi trong cùng. Phép gán không chép dữ liệu -- chú chỉ buộc các tên và các đối tượng. Xóa cũng vậy: câu lệnh "`del x`" bỏ ràng buộc `x` khỏi vùng tên được tham chiếu tới bởi phạm vi nội bộ. Thực tế là mọi tác vụ có thêm các tên mới đều dùng phạm vi nội bộ: điển hình là các câu lệnh nhập và các định nghĩa hàm buộc tên mô-đun hoặc tên hàm vào phạm vi nội bộ. (Lệnh `global` có thể được dùng để cho biết một biến cụ thể là ở phạm vi toàn cục.)

9.3 Cái nhìn đầu tiên về lớp

Lớp thêm một ít cú pháp mới, ba kiểu đối tượng mới, và một ít ngữ nghĩa mới.

9.3.1 Cú pháp định nghĩa lớp

Kiểu đơn giản nhất của việc định nghĩa lớp nhìn giống như:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Định nghĩa lớp, cũng như định nghĩa hàm (câu lệnh`def`) phải được thực thi trước khi chúng có hiệu lực. (Bạn có thể đặt một định nghĩa hàm trong một nhánh của lệnh `if` , hoặc trong một hàm.)

Trong thực tế, các câu lệnh trong một định nghĩa lớp thường là định nghĩa hàm, nhưng các câu lệnh khác cũng được cho phép, và đôi khi rất hữu dụng. Các định nghĩa hàm trong một lớp thường có một dạng danh sách thông số lạ, vì phải tuân theo cách gọi phương thức.

Khi gặp phải một định nghĩa lớp, một vùng tên mới được tạo ra, và được dùng như là phạm vi nội bộ -- do đó, mọi phép gán vào các biến nội bộ đi vào vùng tên này. Đặc biệt, các định nghĩa hàm buộc tên của hàm mới ở đây.

Khi rời khỏi một định nghĩa lớp một cách bình thường, một *đối tượng lớp* được tạo ra. Đây cơ bản là một bộ gói (wrapper) của nội dung của vùng tên tạo ra bởi định nghĩa lớp. Phạm vi nội bộ ban đầu (trước khi vào định nghĩa lớp) được thiết lập lại, và đối tượng lớp được buộc vào đây qua tên lớp đã chỉ định ở định nghĩa lớp, (`ClassName` trong ví dụ này).

9.3.2 Đối tượng lớp

Các đối tượng lớp hỗ trợ hai loại tác vụ: tham chiếu thuộc tính và tạo trường hợp (instantiation).

Tham chiếu thuộc tính dùng cú pháp chuẩn được dùng cho mọi tham chiếu thuộc tính trong Python: `obj.name`. Các tên thuộc tính hợp lệ gồm mọi tên trong vùng tên của lớp khi đối tượng lớp được tạo ra. Do đó, nếu định nghĩa lớp có dạng như sau:

```
class MyClass:
    "A simple example class"
    i = 12345
    def f(self):
        return 'hello world'
```

thì `MyClass.i` và `MyClass.f` là những tham chiếu thuộc tính hợp lệ, trả về một số nguyên và một đối tượng hàm, theo thứ tự đó. Các thuộc tính lớp cũng có thể gán vào, cho nên bạn có thể thay đổi giá trị của `MyClass.i` bằng phép gán. `__doc__` cũng là một thuộc tính hợp lệ, trả về chuỗi tài liệu của lớp: `"A simple example class"`.

Class *instantiation* (tạo trường hợp lớp) dùng cùng cách viết như gọi hàm. Hãy tưởng tượng một đối tượng lớp là một hàm không thông số trả về một trường hợp của lớp. Ví dụ (với lớp trên):

```
x = MyClass()
```

tạo một *trường hợp* mới của lớp và gán đối tượng này vào biến nội bộ `x`.

Tác vụ tạo trường hợp (``gọi'' một đối tượng lớp) tạo một đối tượng rỗng. Nhiều lớp thích tạo đối tượng với các trường hợp được khởi tạo ở một trạng thái đầu nào đó. Do đó một lớp có thể định nghĩa một phương thức đặc biệt tên `__init__()`, như sau:

```
    def __init__(self):
        self.data = []
```

Khi một lớp định nghĩa một phương thức `__init__()` , việc tạo trường hợp lớp sẽ tự động gọi `__init__()` ở trường hợp lớp mới vừa được tạo. Trong ví dụ nạy, một trường hợp đã khởi tạo mới có thể được tại ra từ:

```
x = MyClass()
```

Dĩ nhiên, `__init__()` (phương thức) có thể nhận thêm thông số. Trong trường hợp đó, các thông số đưa vào phép tạo trường hợp lớp sẽ được truyền vào `__init__()`. Ví dụ,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Đối tượng trường hợp

Chúng ta có thể làm được gì với những đối tượng trường hợp? Tác vụ duy nhất mà các đối tượng trường hợp hiểu được là tham chiếu thuộc tính. Có hai loại tên thuộc tính hợp lệ, thuộc tính dữ liệu và phương thức.

data attributes (thuộc tính dữ liệu lớp) tương ứng với ``biến trường hợp'' trong Smalltalk, và ''thành viên dữ liệu'' trong C++. Thuộc tính dữ liệu không cần được khai báo; như các biến nội bộ, chúng tự động tồn tại khi được gán vào. Ví dụ, nếu `x` là một trường hợp của `MyClass` được tạo ra ở trên, đoạn mã sau in ra giá trị `16`, mà không chừa lại dấu vết:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

Loại tham chiếu thuộc tính trường hợp khác là một *method (phương thức)*. Một phương thức là một hàm ``của'' một đối tượng. (Trong Python, từ phương thức không chỉ riêng cho trường hợp lớp: các kiểu đối tượng khác cũng có thể có phương thức. Ví dụ, đối tượng danh sách có phương thức tên `append`, `insert`, `remove`, `sort`, v.v... Tuy nhiên, trong phần sau chúng ta sẽ chỉ dùng từ phương thức để chỉ các phương thức của đối tượng trường hợp lớp, trừ khi được chỉ định khác đi.)

Các tên phương thức hợp lệ của một đối tượng trường hợp phụ thuộc vào lớp của nó. Theo định nghĩa, mọi thuộc tính của một lớp mà là những đối tượng hàm định nghĩa các phương thức tương ứng của các trường hợp của lớp đó. Trong ví dụ của chúng ta, `x.f` là một tham chiếu phương thức hợp lệ, vì `MyClass.f` là một hàm, nhưng `x.i` không phải, bởi vì `MyClass.i` không phải. Nhưng `x.f` không phải là một thứ như `MyClass.f` -- nó là một *method object (đối tượng phương thức)*, không phải là một đối tượng hàm.

9.3.4 Đối tượng phương thức

Thông thường, một phương thức được gọi ngay sau khi nó bị buộc:

```
x.f()
```

Trong `MyClass` , nó sẽ trả về chuỗi `'hello world'`. Tuy nhiên, cũng không nhất thiết phải gọi một phương thức ngay lập tức: `x.f` là một đối tượng phương thức, và có thể được cất đi và gọi vào một thời điểm khác. Ví dụ:

```
xf = x.f
while True:
    print xf()
```

sẽ tiếp tục in "`hello world`" mãi mãi.

Chuyện gì thật sự xảy ra khi một phương thức được gọi? Bạn có thể đã nhận ra rằng `x.f()` được gọi với không thông số, mặc dù định nghĩa hàm của `f` chỉ định một thông số. Chuyện gì xảy ra với thông số đó? Python chắc chắn nâng một biệt lệ khi một hàm cần một thông số được gọi suông -- cho dù thông số đó có được dùng hay không đi nữa...

Thật ra, bạn cũng có thể đã đoán ra được câu trả lời: điểm đặc biệt của phương thức

là đối tượng đó được truyền vào ở thông số đầu tiên của hàm. Trong ví dụ của chúng ta, lời gọi `x.f()` hoàn toàn tương đương với `MyClass.f(x)`. Nói chung, gọi một hàm với một danh sách *n* thông số thì tương đương với việc gọi hàm tương ứng với một danh sách thông số được tạo ra bằng cách chèn đối tượng của phương thức vào trước thông số thứ nhất.

(Hiểu đơn giản là `obj.name(arg1, arg2)` tương đương với `Class.name(obj, arg1, arg2)` trong đó `obj` là đối tượng trường hợp của lớp `Class`, `name` là một thuộc tính hợp lệ không phải dữ liệu, tức là đối tượng hàm của lớp đó.)

9.4 Một vài lời bình

Thuộc tính dữ liệu sẽ che thuộc tính phương thức cùng tên; để tránh vô tình trùng lặp tên, mà có thể dẫn đến các lỗi rất khó tìm ra trong các chương trình lớn, bạn nên có một quy định đặt tên nào đó để giảm thiểu tỉ lệ trùng lặp. Các quy định khả thi có thể gồm viết hoa tên phương thức, đặt tiền tố vào các tên thuộc tính dữ liệu (ví dụ như dấu gạch dưới _), hoặc dùng động từ cho phương thức và danh từ cho các thuộc tính dữ liệu.

Các thuộc tính dữ liệu có thể được tham chiếu tới bởi cả phương thức lẫn người dùng đối tượng đó. Nói một cách khác, lớp không thể được dùng để cài đặt các kiểu dữ liệu trừu tượng tuyệt đối. Trong thực tế, không có gì trong Python có thể ép việc che dấu dữ liệu -- tất cả đều dựa trên nguyên tắc. (Mặt khác, cài đặt Python, được viết bằng C, có thể dấu các chi tiết cài đặt và điều khiển truy cập vào một đối tượng nếu cần; điều này có thể được dùng trong các bộ mở rộng Python viết bằng C.)

Người dùng nên dùng các thuộc tính dữ liệu một cách cẩn thận -- người dùng có thể phá hỏng những bất biến (invariant) được giữ bởi các phương thức nếu cố ý sửa các thuộc tính dữ liệu. Lưu ý rằng người dùng có thể thêm các thuộc tính dữ liệu riêng của hộ vào đối tượng trường hợp mà không làm ảnh hưởng tính hợp lệ của các phương thức, miễn là không có trùng lặp tên -- xin nhắc lại, một quy tắc đặt tên có thể giảm bớt sự đau đầu ở đây.

Không có cách ngắn gọn để tham chiếu tới thuộc tính dữ liệu (hoặc các phương thức khác!) từ trong phương thức. Điều này thật ra giúp chúng ta dễ đọc mã vì không có sự lẫn lộn giữa biến nội bộ và biến trường hợp.

Thông số đầu tiên của phương thức thường được gọi là `self`. Đây cũng chỉ là một quy ước: tên `self` hoàn toàn không có ý nghĩa đặc biệt trong Python. (Tuy nhiên xin nhớ nếu bạn không theo quy ước thì mã của bạn sẽ có thể trở nên khó đọc đối với người khác, và có thể là *trình duyệt lớp* được viết dựa trên những quy ước như vậy.)

Bất kỳ đối tượng hàm nào mà là thuộc tính của một lớp sẽ định nghĩa một phương thức cho các trường hợp của lớp đó. Không nhất thiết định nghĩa hàm phải nằm trong định nghĩa lớp trên văn bản: gán một đối tượng hàm vào một biến nội bộ trong lớp cũng được. Ví dụ:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
```

```
    f = f1
    def g(self):
        return 'hello world'
    h = g
```

Bây giờ `f`, `g` và `h` đều là thuộc tính của lớp `C` mà tham chiếu tới các đối tượng hàm, và do đó chúng đều là phương thức của các trường hợp của `C` -- `h` hoàn toàn tương đương với `g`. Chú ý rằng kiểu viết này thường chỉ làm người đọc càng thêm khó hiểu mà thôi.

Phương thức có thể gọi phương thức khác thông qua thuộc tính phương thức của thông số `self` :

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Phương thức có thể tham chiếu tới các tên toàn cục theo cùng một cách như các hàm thông thường. Phạm vi toàn cục của một phương thức là mô-đun chứa định nghĩa lớp. (Phạm vi toàn cục không bao giờ là lớp!) Trong khi bạn ít gặp việc sử dụng dữ liệu toàn cục trong một phương thức, có những cách dùng hoàn toàn chính đáng: ví dụ như hàm và mô-đun được nhập vào phạm vi toàn cục có thể được sử dụng bởi phương thức, cũng như hàm và lớp được định nghĩa trong đó. Thông thường, lớp chứa các phương thức này được định nghĩa ngay trong phạm vi toàn cục, và trong phần kế đây chúng ta sẽ thấy tại sao một phương thức muốn tham chiếu tới chính lớp của nó!

9.5 Kế thừa

Dĩ nhiên, một tính năng ngôn ngữ sẽ không đáng được gọi là ``lớp'' nếu nó không hỗ trợ kế thừa. Cú pháp của một định nghĩa lớp con như sau:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Tên `BaseClassName` phải đã được định nghĩa trong một phạm vi chứa định nghĩa lớp con. Thay vì tên lớp cơ sở, các biểu thức khác cũng được cho phép. Điều này rất hữu ích, ví dụ, khi mà lớp cơ sở được định nghĩa trong một mô-đun khác:

```
class DerivedClassName(modname.BaseClassName):
```

Việc thực thi định nghĩa lớp con tiến hành như là lớp cơ sở. Khi một đối tượng lớp được tạo ra, lớp cơ sở sẽ được nhớ. Nó được dùng trong việc giải các tham chiếu thuộc tính: nếu một thuộc tính không được tìm thấy ở trong lớp, việc tìm kiếm sẽ tiếp tục ở lớp cơ sở. Luật này sẽ được lặp lại nếu lớp cơ sở kế thừa từ một lớp khác.

Không có gì đặc biệt trong việc tạo trường hợp của các lớp con: `DerivedClassName()`

tạo một trường hợp của lớp. Các tham chiếu hàm được giải như sau: thuộc tính lớp tương ứng sẽ được tìm, đi xuống chuỗi các lớp cơ sở nếu cần, và tham chiếu phương thức là hợp lệ nếu tìm thấy một đối tượng hàm.

Lớp con có thể định nghĩa lại các phương thức của lớp cơ sở. Bởi vì phương thức không có quyền gì đặc biệt khi gọi một phương thức của cùng một đối tượng, một phương thức của lớp cơ sở gọi một phương thức khác được định nghĩa trong cùng lớp cơ sở có thể là đang gọi một phương thức do lớp con đã định nghĩa lại. (Người dùng C++ có thể hiểu là mọi phương thức của Python là `virtual`.)

Một phương thức được định nghĩa lại trong lớp con có thể muốn mở rộng thay vì thay thế phương thức cùng tên của lớp cơ sở. Có một cách đơn giản để gọi phương thức của lớp sơ sở: chỉ việc gọi "`BaseClassName.methodname(self, arguments)`". Đôi khi điều này cũng có ích cho người dùng. (Lưu ý rằng đoạn mã chỉ hoạt động nếu lớp cơ sở được định nghĩa hoặc nhập trực tiếp vào phạm vi toàn cục.)

9.5.1 Đa kế thừa

Python cũng hỗ trợ một dạng đa kế thừa hạn chế. Một định nghĩa lớp với nhiều lớp cơ sở có dạng sau:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Luật duy nhất cần để giải thích ý nghĩa là luật giải các tham chiếu thuộc tính của lớp. Nó tuân theo luật tìm theo chiều sâu, và tìm trái qua phải. Do đó, nếu một thuộc tính không được tìm ra trong `DerivedClassName`, nó sẽ được tìm trong `Base1`, rồi (đệ quy) trong các lớp cơ sở của `Base1`, rồi chỉ khi nó không được tìm thấy, nó sẽ được tìm trong `Base2`, và cứ như vậy.

(Đối với một số người tìm theo chiều rộng -- tìm `Base2` và `Base3` trước các lớp cơ sở của `Base1` -- có vẻ tự nhiên hơn. Nhưng, điều này yêu cầu bạn biết một thuộc tính nào đó của `Base1` được thật sự định nghĩa trong `Base1` hay trong một trong các lớp cơ sở của nó trước khi bạn có thể biết được hậu quả của sự trùng lặp tên với một thuộc tính của `Base2`. Luật tìm theo chiều sâu không phân biệt giữa thuộc tính trực tiếp hay kế thừa của `Base1`.)

Ai cũng biết rằng việc dùng đa kế thừa bừa bãi là một cơn ác mộng cho bảo trì, đặc biệt là Python dựa vào quy ước để tránh trùng lặp tên. Một vấn đề cơ bản với đa kế thừa là một lớp con của hai lớp mà có cùng một lớp cơ sở. Mặc dù dễ hiểu chuyện gì xảy ra trong vấn đề này (trường hợp sẽ có một bản chép duy nhất của ``các biến trường hợp'' của các thuộc tính dữ liệu dùng bởi lớp cơ sở chung), nó không rõ cho lắm nếu các ý nghĩa này thật sự hữu ích.

9.6 Biến riêng

Có một dạng hỗ trợ nho nhỏ nho các từ định danh riêng của lớp (class-private identifier). Các từ định danh có dạng `__spam` (ít nhất hai dấu gạch dưới ở đầu, nhiều

nhất một dấu dạch dưới ở cuối) được thay thế văn bản (textually replace) bằng `_classname__spam`, trong đó `classname` là tên lớp hiện tại với các gạch dưới ở đầu cắt bỏ. Việc xáo trộn tên (mangling) được thực hiện mà không quan tâm tới vị trí cú pháp của định danh, cho nên nó có thể được dùng để định nghĩa các trường hợp, biến, phương thức, riêng của lớp, hoặc các biến toàn cục, và ngay cả các biến của trường hợp, riêng với lớp này trên những trường hợp của lớp *khác* . Nếu tên bị xáo trộn dài hơn 255 ký tự thì nó sẽ bị cắt đi. Bên ngoài lớp, hoặc khi tên lớp chỉ có ký tự gạch dưới, việc xáo trộn tên sẽ không xảy ra.

Xáo trộn tên nhằm cung cấp cho các lớp một cách định nghĩa dễ dàng các biến và phương thức ``riêng'', mà không phải lo về các biến trường hợp được định nghĩa bởi lớp con, hoặc việc sử dụng biến trường hợp bởi mã bên ngoài lớp. Lưu ý rằng việc xáo trộn tên được thiết kế chủ yếu để tránh trùng lặp; người quyết tâm vẫn có thể truy cập hoặc thay đổi biến riêng. Và điều này cũng có thể có ích trong các trường hợp đặc biệt, như trong trình gỡ rối, và đó là một lý do tại sao lỗ hổng này vẫn chưa được vá.

Lưu ý rằng mã truyền vào `exec`, `eval()` hoặc `execfile()` không nhận tên lớp của lớp gọi là tên lớp hiện tại; điều này cũng giống như tác dụng của câu lệnh `global` , tác dụng của nó cũng bị giới hạn ở mã được biên dịch cùng. Cùng giới hạn này cũng được áp dụng vào `getattr()`, `setattr()` và `delattr()`, khi tham chiếu `__dict__` trực tiếp.

9.7 Những điều khác

Đôi khi nó thật là hữu ích khi có một kiểu dữ liệu giống như Pascal ``record'' hoặc C ``struct'', gói gọn vài mẩu dữ liệu vào chung với nhau. Một định nghĩa lớp rỗng thực hiện được việc này:

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Với mã Python cần một kiểu dữ liệu trừu tượng, ta có thể thay vào đó một lớp giả lập các phương thức của kiểu dữ liệu đó. Ví dụ, nếu bạn có một hàm định dạng một vài dữ liệu trong một đối tượng tập tin, bạn có thể định nghĩa một lớp với các phương thức `read()` và `readline()` lấy dữ liệu từ một chuỗi, và truyền vào nó một thông số.

Các đối tượng phương trức trường hợp cũng có thuộc tính: `m.im_self` là một đối tượng trường hợp với phương thức `m`, và `m.im_func` là đối tượng hàm tương ứng với phương thức.

9.8 Biệt lệ cũng là lớp

Các biệt lệ được định nghĩa bởi người dùng cũng được định danh theo lớp. Bằng cách này, một hệ thống phân cấp biệt lệ có thể được tạo ra.

Có hai dạng lệnh `raise` mới:

```
raise Class, instance

raise instance
```

Trong dạng đầu, `instance` phải là một trường hợp của kiểu `Class` hoặc là lớp con của nó. Dạng thứ hai là rút gọn của:

```
raise instance.__class__, instance
```

Lớp trong vế `except` tương thích với một biệt lệ nếu nó cùng lớp, hoặc là một lớp cơ sở (nhưng chiều ngược lại thì không đúng -- một vế `except` dùng lớp con sẽ không tương thích với một biệt lệ lớp cơ sở). Ví dụ, đoạn mã sau sẽ in B, C, D theo thứ tự đó:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Nếu các vế `except` được đặt ngược (với "`except B`" ở đầu), nó sẽ in B, B, B -- vế `except` phù hợp đầu tiên được thực thi.

Khi một thông điệp lỗi được in, tên lớp của biệt lệ được in, theo sau bởi dấu hai chấm và một khoảng trắng, và cuối cùng là trường hợp đã được chuyển thành chuỗi bằng hàm có sẵn `str()`.

9.9 Bộ lặp

Bây giờ có lẽ bạn đã lưu ý rằng hầu hết các đối tượng chứa (container object) có thể được lặp qua bằng câu lệnh `for` :

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

Kiểu truy xuất này rõ ràng, xúc tích, và tiện lợi. Bộ lặp (iterator) được dùng khắp nơi

và hợp nhất Python. Đằng sau màn nhung, câu lệnh `for` gọi `iter()` trên đối tượng chứa. Hàm này trả về một đối tượng bộ lặp có định nghĩa phương thức `next()` để truy xuất và các phần tử trong bộ chứa (container). Khi không còn phần tử nào, `next()` nâng biệt lệ `StopIteration` để yêu cầu vòng lặp `for` kết thúc. Ví dụ sau cho thấy cách hoạt động:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()

Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    it.next()
StopIteration
```

Chúng ta đã hiểu giao thức bộ lặp, nên chúng ta có thể thêm cách thức bộ lặp (iterator behavior) vào lớp của chúng ta một cách dễ dàng. Định nghĩa một phương thức `__iter__()` trả về một đối tượng với một phương thức `next()` . Nếu lớp có định nghĩa `next()`, thì `__iter__()` chỉ cần trả về `self`:

```
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
...     print char
...
m
a
p
s
```

9.10 Bộ tạo

Bộ sinh (generator) là một công cụ đơn giản và mạnh mẽ để tạo các bộ lặp. Chúng được viết như những hàm thông thường nhưng dùng câu lệnh `yield` khi nào chúng muốn trả về dữ liệu. Mỗi lần `next()` được gọi, bộ sinh trở lại nơi nó đã thoát ra (nó nhớ

mọi dữ liệu và câu lệnh đã được thực thi lần cuối). Một ví dụ cho thấy bộ sinh có thể được tạo ra rất dễ dàng:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

>>> for char in reverse('golf'):
...     print char
...
f
l
o
g
```

Bất kỳ việc gì có thể được thực hiện với bộ sinh cũng có thể được thực hiện với các bộ lặp dựa trên lớp như đã bàn đến ở phần trước. Điều khiển bộ sinh nhỏ gọn là các phương thức `__iter__()` và `next()` được tự động tạo ra.

Một tính năng chính khác là các biến nội bộ và trạng thái thực thi được tự động lưu giữa các lần gọi. Điều này làm cho hàm dễ viết hơn và rõ ràng hơn là cách sử dụng biến trường hợp như `self.index` và `self.data`.

Thêm vào việc tự động tạo và lưu trạng thái chương trình, khi các bộ tạo kết thúc, chúng tự động nâng `StopIteration`. Cộng lại, các tính năng này làm cho việc tạo các bộ lặp không có gì khó hơn là viết một hàm bình thường.

9.11 Biểu thức bộ tạo

Một vài bộ sinh đơn giản có thể được viết một cách xúc tích như các biểu thức bằng cách dùng một cú pháp giống như gộp danh sách (list comprehension) nhưng với ngoặc tròn thay vì ngoặc vuông. Các biểu thức này được thiết kế cho những khi bộ sinh được sử dụng ngay lập tức bởi hàm chứa nó. Biểu thức bộ sinh gọn hơn nhưng ít khả chuyển hơn là các định nghĩa bộ sinh đầy đủ và thường chiếm ít bộ nhớ hơn là gộp danh sách tương đương.

Ví dụ:

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> from math import pi, sin
>>> sine_table = dict((x, sin(x*pi/180)) for x in range(0, 91))

>>> unique_words = set(word  for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
```

```
>>> list(data[i] for i in range(len(data)-1,-1,-1))
['f', 'l', 'o', 'g']
```

Ghi chú

... vùng tên![9.1]

Trừ một chuyện. Các đối tượng mô-đun có một thuộc tính chỉ đọc gọi là `__dict__` trả về một từ điển dùng để cài đặt vùng tên của mô-đun; tên `__dict__` là một thuộc tính nhưng không phải là một tên toàn cục. Rõ ràng, sử dụng nó vi phạm tính trừu tượng của cài đặt vùng tên, và nên được giới hạn vào những chuyện như gỡ rối.

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách đề nghị thay đổi.

Bài chỉ dẫn Python

10. Giới thiệu sơ về bộ thư viện chuẩn

10.1 Giao tiếp với hệ thống

os (mô-đun) cung cấp hàng loạt các hàm dùng cho việc giao tiếp với hệ điều hành:

```
>>> import os
>>> os.system('time 0:02')
0
>>> os.getcwd()      # Return the current working directory
'C:\\Python24'
>>> os.chdir('/server/accesslogs')
```

Nhớ dùng kiểu lệnh "`import os`" thay vì "`from os import *`". Điều này khiến cho `os.open()` không che hàm `open()` sẵn có của python. Hai hàm này hoạt động khác nhau rất nhiều.

Các hàm sẵn có `dir()` và `help()` là các công cụ trợ giúp tương tác hữu ích khi làm việc với các mô-đun lớn như `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Đối với các công việc quản lý file và thư mục thông thường, mô-đun shutil cung cấp một giao diện mức cao hơn và dễ dùng hơn:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

10.2 Ký tự thay thế tập tin

glob (mô-đun) cũng hỗ trợ việc tạo danh sách các tập tin từ việc tìm kiếm thư mục dùng ký tự thay thế (wildcard):

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Thông số dòng lệnh

Các kịch bản phổ dụng thường phải xử lý các tham số dòng lệnh. Các tham số này được lưu thành một danh sách ở mô-đun sys trong thuộc tính *argv* . Ví dụ, kết quả sau đây thu được từ việc chạy lệnh "`python demo.py one two three`" từ dòng lệnh:

```
>>> import sys
```

```
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

getopt (mô-đun) xử lý *sys.argv* theo các nguyên tắc của hàm UNIX `getopt()` . Nếu cần các thao tác linh hoạt và hữu hiệu hơn, chúng ta có thể dùng mô-đun optparse .

10.4 Chuyển hướng luồng ra và kết thúc chương trình

sys (mô-đun) cũng có các thuộc tính cho *stdin*, *stdout*, và *stderr*. Cái cuối rất hữu dụng trong việc sinh ra các cảnh báo và thông báo lỗi và việc hiển thị chúng ngay cả khi *stdout* đã được định hướng lại:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

Cách thoát khỏi một kịch bản một cách trực tiếp nhất là dùng "`sys.exit()`".

10.5 Khớp mẫu chuỗi

re (mô-đun) cung cấp các công cụ biểu thức chính quy dùng cho việc xử lý chuỗi ở mức cao. Biểu thức chính quy cung cấp các phương án súc tích và tối ưu cho các thao tác tìm kiếm và xử lý chuỗi phức tạp:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Đối với các chức năng xử lý chuỗi cơ bản thì các phương thức của đối tượng chuỗi được ưa chuộng hơn bởi chúng dễ đọc và dễ gỡ rối hơn:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Toán học

math (mô-đun) cung cấp các hàm xử lý về toán dấu chấm động của thư viện C mức dưới:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

random (mô-đun) hỗ trợ việc tạo ra các lựa chọn ngẫu nhiên:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
```

```
>>> random.random()    # random float
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

10.7 Truy cập internet

Python cung cấp một vài mô-đun khác nhau cho việc truy cập internet và xử lý các giao thức internet. Hai mô-đun đơn giản nhất là urllib2 dành cho việc thu thập dữ liệu từ các URL và smtplib dành cho việc gửi thư điện tử:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line:  # look for Eastern Time
...         print line

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
"""To: jcaesar@example.org
From: soothsayer@example.org

Beware the Ides of March.
""")
>>> server.quit()
```

10.8 Ngày và giờ

datetime (mô-đun) cung cấp các lớp dành cho viêc xử lý ngày tháng và thời gian từ đơn giản tới phức tạp. Mô-đun này có hỗ trợ các phép toán về ngày tháng, tuy nhiên nó chú trọng tới việc truy cập các thành phần ngày tháng một cách hiệu quả giúp cho việc định dạng chúng. Mô-đun này cũng hỗ trợ các đối tượng có thể phân biệt được các vùng thời gian.

```
# dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

# dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Nén dữ liệu

Python cung cấp một số mô-đun hỗ trợ trực tiếp các định dạng nén và lưu trữ dữ liệu phổ

biến như: zlib, gzip, bz2, zipfile, và tarfile.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Đo lường hiệu suất

Một vài người dùng Python rất quan tâm đến việc tìm hiểu sự khác bệt về hiệu năng giữa các phương án khác nhau của cùng một vấn đề. Python cung cấp một công cụ đo đạc để thỏa mãn nhu cầu này.

Ví dụ, chúng ta thường muốn sử dụng tính năng gói bộ và mở gói bộ thay cho phương pháp thông thường trong việc hoán đổi tham số. Mô-đun timeit cho thấy phương pháp này có hiệu năng nhỉnh hơn phương pháp thông thường:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

So sánh với độ phân biệt về thời gian và sự chính xác cao của `timeit`, các mô-đun profile và `pstats` cung cấp các công cụ cho việc xác định các đoạn mã tiêu tốn nhiều thời gian trong các khối mã lớn hơn.

10.11 Quản lý chất lượng

Một phương pháp để phát triển phần mềm chất lượng cao là viết các hàm kiểm tra cho từng hàm khi viết các hàm và chạy các hàm kiểm tra một cách thường xuyên trong quá trình phát triển phần mềm.

doctest (mô-đun) cung cấp công cụ cho việc rà soát một mô-đun và thẩm định các hàm kiểm tra nhúng trong tài liệu của chương trình. Việc xây dựng các đoạn kiểm tra được thực hiện đơn giản bằng cách cắt và dán một đoạn gọi hàm thông thường kèm theo kết quả của hàm đó vào tài liệu chương trình. Việc này cải thiện đáng kể tài liệu chương trình bởi nó cung cấp cho người dùng một ví dụ về việc sử dụng hàm và cho phép mô-đun doctest kiểm tra tính đúng đắn của hàm này so với tài liệu:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)
```

```
    import doctest
    doctest.testmod()   # automatically validate the embedded tests
```

unittest (mô-đun) không dễ dùng như mô-đun doctest , nhưng nó hỗ trợ các hàm kiểm tra toàn diện hơn và lưu giữ chúng trong một tập tin riêng biệt:

```
    import unittest

    class TestStatisticalFunctions(unittest.TestCase):

        def test_average(self):
            self.assertEqual(average([20, 30, 70]), 40.0)
            self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
            self.assertRaises(ZeroDivisionError, average, [])
            self.assertRaises(TypeError, average, 20, 30, 70)

    unittest.main() # Calling from the command line invokes all tests
```

10.12 Kèm cả pin

Python được gắn với ý tưởng ``kèm pin''. Điều này được thể hiện bởi các tính năng mạnh mẽ và đa dạng của các gói lớn hơn của nó. Ví dụ như:

- xmlrpclib và SimpleXMLRPCServer (mô-đun) giúp cho việc cài đặt các lệnh gọi thủ tục từ xa (remote procedure call) trở nên dễ dàng hơn bao giờ hết. Khác với cái tên, chúng ta có thể sử dụng mô-đun này mà không cần các kiến thức cụ thể về xử lý XML.
- email (gói) là một thư viện hỗ trợ việc quản lý các thư điện tử, bao gồm các văn bản MIME và các văn bản dựa trên RFC 2822 khác. Không trực tiếp gửi và nhận thông điệp như smtplib và poplib , gói email có một tập các công cụ dành cho việc xây dựng và mã hóa các cấu trúc thông điệp phức tạp (bao gồm cả tập tin đính kèm) và cài đặt mã hóa internet và giao thức tiêu đề.
- xml.dom và xml.sax (gói) hỗ trợ rất tốt cho việc phân tích định dạng phổ biến này. Tương tự, mô-đun csv hỗ trợ việc đọc ghi trực tiếp trên một định dạng văn bản chung. Kết hợp lại, các mô-đun và gói kể trên đơn giản hóa rất nhiều việc trao đổi dữ liệu giữa các trình ứng dụng của python và các chương trình khác.
- Việc hỗ trợ quốc tế hóa được thực hiện bởi một vài mô-đun, bao gồm gettext, locale, và gói codecs .

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này... về cách đề nghị thay đổi.

Bài chỉ dẫn Python

11. Giới thiệu sơ về bộ thư viện chuẩn - Phần II

Bài giới thiệu thứ hai này nói về các mô-đun chuyên sâu nhằm đáp ứng các tác vụ lập trình chuyên nghiệp. Các mô-đun này ít khi xuất hiện ở các kịch bản nhỏ.

11.1 Định dạng ra

repr (mô-đun)cung cấp một phiên bản `repr()` đã được tùy biến để hiển thị vắn tắt các đối tượng chứa (container) lớn hoặc lồng nhau nhiều mức:

```
>>> import repr
>>> repr.repr(set('supercalifragilisticexpialidocious'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

pprint (mô-đun) hỗ trợ việc kiểm soát việc in các đối tượng sẵn có hoặc các đối tượng do người dùng định nghĩa một cách tinh vi hơn theo một phương thúc mà rình thông dịch có thể hiểu được. Khi kết quả hiển thị ngắn hơn một dòng thì "pretty printer" sẽ thêm các dấu xuống dòng và dấu thụt vào đầu dòng khiến cho cấu trúc dữ liệu được thể hiện rõ rệt hơn:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

textwrap (mô-đun) định dạng đoạn văn bản sao cho vừa với độ rộng của màn hình:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print textwrap.fill(doc, width=40)
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

locale (mô-đun) sử dụng một cơ sở dữ liệu các định dạng dữ liệu dựa trên đặc điểm của từng khu vực. Hàm định dạng của locale có thuộc tính tạo nhóm, cho phép định dạng trực tiếp các con số với các dấu phân chia nhóm:

```
>>> import locale
```

```
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format("%s%.*f", (conv['currency_symbol'],
...       conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Tạo mẫu

string (mô-đun) cung cấp một lớp `Template` với một cú pháp đơn giản, có thể dễ dàng thay đổi bởi người dùng. Điều này cho phép người dùng có thể tùy biến trình ứng dụng mà không phải thay đổi nó.

Định dạng này sử dụng các tên biến giữ chỗ bắt đầu với "$" sau đó là một tên biến hợp lệ của Python (chữ cái, chữ số và dấu gạch dưới). Tên giữ chỗ được đặt trong ngoặc, điều này khiến nó có thể được nối tiếp bởi các ký tự khác mà không cần có khoảng trống ở giữa. Viết "$$" sẽ tạo ra một ký tự thoát đơn "$":

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

`substitute` (phương thức) nâng `KeyError` nếu biến giữ chỗ không được cung cấp bởi một từ điển hay một tham số nhập từ bàn phím. Đối với các trình ứng dụng kiểu nhập liệu vào thư điện tử, dữ liệu nhập bởi người dùng có thể có thiếu sót, khi đó phương thức `safe_substitute` sẽ thích hợp hơn -- nó giữ nguyên các biến giữ chỗ nếu dữ liệu bị thiếu:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  . . .
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Các lớp con của `Template` có thể định nghĩa một phần tử phân chia tùy biến. Ví dụ một ứng dụng đổi tên ảnh hàng loạt của một trình duyệt ảnh có thể sử dụng dấu phần trăm để làm các biến giữ chỗ ngày tháng, số thứ tự ảnh, hay định dạng ảnh:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
```

```
...     newname = t.substitute(d=date, n=i, f=ext)
...     print '%s --> %s' % (filename, newname)

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Một ứng dụng khác của tạo mẫu là việc tách biệt lô-gíc chương trình ra khỏi các chi tiết về các định dạng đầu ra khác nhau. Điều này giúp cho việc thay thế các khuôn mẫu tùy biến cho các tập tin XML, các báo cáo bằng văn bản thông thường và các báo cáo bằng HTML.

11.3 Làm việc với bản ghi dữ liệu nhị phân

struct (mô-đun) cung cấp các hàm `pack()` và `unpack()` để dùng với các định dạng bản ghi nhị phân với chiều dài không cố định. Ví dụ sau đây minh họa phương pháp lặp qua các thông tin trong phần tiêu đề của một tập tin ZIP (ký hiệu `"H"` và `"L"` biểu diễn các số không dấu hai byte và bốn byte):

```
import struct

data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('LLLHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print filename, hex(crc32), comp_size, uncomp_size

    start += extra_size + comp_size     # skip to the next header
```

11.4 Đa luồng

Phân luồng là kỹ thuật phân tách các tác vụ khác nhau của một chương trình khi chúng không bị ràng buộc trật tự với nhau. Các luồng được sử dụng để tăng tính đáp ứng của các chương trình ứng dụng đòi hỏi việc nhập liệu từ người dùng diễn ra cùng lúc với các tiến trình nền khác. Một trường hợp tương tự là việc chạy các tác vụ vào ra song song với các tác vụ tính toán ở một luồng khác.

Đoạn mã sau đây minh họa cách mô-đun threading chạy các tác vụ nền trong khi chương trình chính vẫn đang chạy:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
```

```
        def run(self):
            f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
            f.write(self.infile)
            f.close()
            print 'Finished background zip of: ', self.infile

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print 'The main program continues to run in foreground.'

background.join()    # Wait for the background task to finish
print 'Main program waited until background was done.'
```

Khó khăn lớn nhất của các chương trình ứng dụng đa luồng là việc điều phối các luồng phải chia sẻ dữ liệu hay các tài nguyên khác. Về mặt này, mô-đun `threading` hỗ trợ một số hàm đồng bộ sơ cấp như các khóa (lock), các sự kiện (event), các biến điều kiện (condition) và các cờ hiệu (semaphore).

Mặc dù các công cụ kể trên rất mạnh, nhung những lỗi thiết kế nhỏ có thể dẫn tới các vấn đề rất khó có thể tái tạo được. Do đó, phương pháp được ưa chuộng trong việc điều phối các tác vụ là tập hợp các truy cập tới một tài nguyên vào một luồng, sau đó sử dụng mô-đun `Queue` để nạp các yêu cầu từ các luồng khác tới luồng đó. Các trình ứng dụng sử dụng đối tượng `Queue` cho việc giao tiếp và điều phối giữa các luồng có ưu điểm là dễ thiết kể hơn, dễ đọc hơn và đáng tin cậy hơn.

11.5 Nhật ký

`logging` (mô-đun) cung cấp một hệ thống ghi nhật ký (logging) linh hoạt và có đầy đủ các tính năng. Trong trường hợp đơn giản nhất, một thông điệp nhật ký được gửi tới một tập tin hay tới `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Đoạn mã trên sẽ cho kết quả sau:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Theo mặc định, các thông điệp chứa thông tin dành cho việc gỡ rối bị chặn lại và đầu ra được gửi tới kênh báo lỗi chuẩn. Các thông điệp này còn có thể được chuyển tiếp tới thư điện tử, gói tin, socket hay máy chủ HTTP. Các bộ lọc có thể chọn các cơ chế chuyển tiếp tùy theo mức độ ưu tiên của thông điệp: `DEBUG`, `INFO`, `WARNING`, `ERROR`, và `CRITICAL`.

Hệ thống nhật ký có thể được cấu hình trực tiếp bên trong Python hoặc nạp từ một tập tin cấu hình mà người dùng có thể sửa đổi được, nhằm tùy biến việc ghi nhật ký mà không phải sửa đổi trình ứng dụng.

11.6 Tham chiếu yếu

Python hỗ trợ việc quản lý bộ nhớ một cách tự động (bao gồm việc đếm tham chiếu với hầu hết các đối tượng và việc thu dọn rác). Vùng nhớ được giải phóng nhanh chóng sau khi tham chiếu cuối cùng đến nó kết thúc.

Phương pháp này tỏ ra hiệu quả với hầu hết các trình ứng dụng sử dụng Python, tuy vậy đôi khi ta có nhu cầu theo dõi một đối tượng chừng nào chúng được sử dụng ở một chỗ khác. Tuy vậy việc theo dõi này lại tạo ra một tham chiếu đến đối tượng đó, khiến bản thân nó trở thành một tham chiếu vĩnh viễn. Mô-đun weakref cho phép theo dõi một đối tượng mà không cần phải tạo một tham chiếu tới đối tượng đó. Khi đối tượng không còn cần dùng nữa, nó sẽ tự động bị loại ra khỏi bảng tham chiếu yếu và một hàm gọi ngược (callback) sẽ được gọi tới đối tượng `weakref`. Các ứng dụng phổ biến có chứa các đối tượng được lưu tạm và đòi hỏi chi phí khởi tạo cao:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...             self.value = value
...     def __repr__(self):
...             return str(self.value)
...
>>> a = A(10)                   # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a            # does not create a reference
>>> d['primary']                # fetch the object if it is still alive
10
>>> del a                       # remove the one reference
>>> gc.collect()                # run garbage collection right away
0
>>> d['primary']                # entry was automatically removed
Traceback (most recent call last):
  File "<pyshell#108>", line 1, in -toplevel-
    d['primary']                # entry was automatically removed
  File "C:/PY24/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Công cụ làm việc với danh sách

Kiểu danh sách sẵn có của Python có thể đáp ứng được nhu cầu về nhiều kiểu cấu trúc dữ liệu khác nhau. Tuy vậy chúng ta đôi khi cần tới các kiểu cấu trúc khác, tùy thuộc vào các mục tiêu cụ thể về hiệu năng.

array (mô-đun) cung cấp đối tượng `array()` giống như một danh sách chứa các dữ liệu cùng kiểu và lưu giữ chúng gọn hơn. Ví dụ sau cho thấy một mảng các con số được lưu giữ dưới dạng các số không dấu hai byte (mã "H") thay vì mỗi phần tử chiếm 16 byte như trong một danh sách thông thường chứa các đối tượng số nguyên của Python:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

collections (mô-đun) cung cấp đối tượng `deque()` giống như một danh sách nhưng có

các thao tác thêm vào và lấy ra đầu bên trái nhanh hơn, nhưng việc tìm kiếm ở giữa thì lại chậm hơn. Các đối tượng này thích hợp cho việc cài đặt các hàng đợi và tìm kiếm cây theo chiều rộng:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print "Handling", d.popleft()
Handling task1

unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
        unsearched.append(m)
```

Ngoài các cấu trúc thay thế cho danh sách, thư viện chuẩn còn cung cấp các công cụ như mô-đun bisect chứa các hàm thao tác trên danh sách đã được sắp xếp:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

heapq (mô-đun) cung cấp các hàm để cài đặt đống (heap) dựa trên các danh sách thông thường. Phần tử có giá trị thấp nhất luôn được giữ ở vị trí đầu tiên. Hàm này rất có ích trong các trình ứng dụng đòi hỏi việc truy cập tới phần tử nhỏ nhất mà không cần phải sắp xếp lại toàn bộ danh sách:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

11.8 Số học dấu chấm động thập phân

decimal (mô-đun) cung cấp kiểu dữ liệu `Decimal` cho các phép toán dấu chấm động hệ thập phân. So với cấu trúc dấu chấm động nhị phân `float` sẵn có của Python, lớp này rất có ích trong các trình ứng dụng về tài chính và các công việc đòi hỏi việc biểu diễn chính xác các số thập phân, kiểm soát mức độ chính xác cũng như việc làm tròn các con số theo các quy định đề ra, theo dõi vị trí của dấu chấm động hay trong các trình ứng dụng mà người sử dụng mong muốn thu được kết quả khớp với kết quả tính toán bằng tay.

Ví dụ như, việc tính 5% thuế trên một cuộc gọi điện giá 70 cent sẽ cho kết quả khác nhau nếu sử dụng các phép toán dấy chấm động hệ thập phân và nhị phân. Sự khác biệt trở nên rõ rệt nếu kết quả được làm tròn tới cent:

```
>>> from decimal import *
>>> Decimal('0.70') * Decimal('1.05')
```

```
Decimal("0.7350")
>>> .70 * 1.05
0.73499999999999999
```

`Decimal` giữ một số không ở vị trí cuối cùng, nó tự động quy kết quả kiểu có bốn chữ số sau dấu chấm nếu các thừa số của phép nhân có hai chữ số sau dấu chấm. `Decimal` thực hiện các phép toán tương tự như cách chúng được tính bằng tay, nhờ đó tránh được các vấn đề gặp phải khi dấu chấm động hệ nhị phân không thể biểu diễn chính xác các giá trị thập phân.

Việc biểu diễn chính xác các con số giúp cho lớp `Decimal` có thể thực hiện được các phép tính modulo và các so sánh bằng, điều mà dấu chấm động hệ nhị phân không làm được:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal("0.00")
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

`decimal` (mô-đun) cung cấp các phép toán với độ chính xác cao, tùy thuộc vào đòi hỏi của người dùng:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal("0.142857142857142857142857142857142857")
```

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liệu này.. về cách đề nghị thay đổi.

Bài chỉ dẫn Python

12. Tiếp theo?

Đọc xong bài chỉ dẫn này có lẽ đã làm tăng thêm sự thích thú của bạn với ngôn ngữ Python -- và bạn cũng muốn nhanh chóng áp dụng Python để giải quyết các vấn đề trong đời sống. Thế bây giờ bạn nên học tiếp từ đâu?

Bài chỉ dẫn này là một phần của bộ tài liệu của Python. Những tài liệu khác trong bộ này gồm:

- *Tham khảo thư viện Python*:

  Bạn nên duyệt qua cẩm nang này vì nó cung cấp tài liệu tham khảo đầy đủ (dù là ngắn gọn) về các kiểu, hàm, và các mô-đun trong bộ thư viện chuẩn. Bộ phân phối Python chuẩn có *rất* nhiều mã bổ sung. Có những mô-đun để đọc hộp thư UNIX , lấy tài liệu từ HTTP, sinh số ngẫu nhiên, phân tích thông số dòng lệnh, viết các ứng dụng CGI, nén dữ liệu, và rất nhiều tác vụ khác. Lướt qua Tham khảo thư viện sẽ cho bạn biết những gì đang có.

- *Cài đặt các mô-đun Python* giải thích làm thế nào để cài các mô-đun ngoài được viết bởi các người dùng Python khác.

- *Tham khảo ngôn ngữ*: Một tài liệu chi tiết về cú pháp và ngữ nghĩa của Python. Khó đọc, nhưng rất hữu dụng vì nó là hướng dẫn đầy đủ của chính ngôn ngữ.

Các tài nguyên Python khác:

- http://www.python.org: Trang mạng Python chính. Nó có mã, tài liệu, và chỉ dẫn tới các trang liên quan tới Python trên mạng. Trang này được chép ra ở khắp nơi trên thế giới như Châu Âu, Nhật, và Úc; các bản sao có thể nhanh hơn là trang chính, tùy thuộc vào vị trí địa lý của bạn.

- http://docs.python.org: Truy cập nhanh tới tài liệu Python.

- http://cheeseshop.python.org: Chỉ mục gói Python (Python Package Index), tên lóng là Cửa hàng Phô-mai (Cheese Shop), là một chỉ mục của các mô-đun Python do người dùng tạo để tải về. Một khi bạn bắt đầu tung mã của bạn ra ngoài, bạn có thể đăng ký nó ở đây để người khác có thể tìm thấy nó.

- http://aspn.activestate.com/ASPN/Python/Cookbook/: Sách dạy nấu ăn Python (Python Cookbook) là một tập hợp lớn các mã mẫu, các mô-đun lớn hơn, và các kịch bản hữu dụng. Các đóng góp quan trọng được thu thập vào một quyển sách tên là *Python Cookbook* (O'Reilly & Associates,

ISBN 0-596-00797-3.)

Để hỏi các câu hỏi liên quan tới Python và thông báo lỗi, bạn có thể gửi với nhóm tin comp.lang.python, hoặc gửi chúng tới danh sách thư tín python-list@python.org. Nhóm tin và danh sách thư tín được nối chung với nhau, nên các tin gửi ở nơi này sẽ tự động có mặt ở nơi kia. Có khoảng 120 bài gửi một ngày (cao nhất là vài trăm), hỏi (và trả lời) câu hỏi, đề xuất các tính năng mới, và thông báo các mô-đun mới. Trước khi gửi, hãy nhớ xem qua danh sách Các câu hỏi thường hỏi (Frequently Asked Question) (còn được gọi là FAQ), hoặc xem qua thư mục Misc/ của bản phân phối nguồn Python. Kho lưu trữ nhóm tin có ở http://mail.python.org/pipermail/. FAQ trả lời nhiều câu hỏi hay gặp thường xuyên, và có thể câu hỏi của bạn cũng có ở trong này.

Phiên bản 2.5, tài liệu được cập nhật ngày 19, tháng 09, năm 2006.
Xem Về tài liêu này... về cách đề nghị thay đổi.

Python Tutorial

This Appendix was left untranslated.

A. Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the *GNU Readline* library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the UNIX and Cygwin versions of the interpreter.

This chapter does *not* document the editing facilities of Mark Hammond's PythonWin package or the Tk-based environment, IDLE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.

A.1 Line Editing

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: `C-A` (Control-A) moves the cursor to the beginning of the line, `C-E` to the end, `C-B` moves it one position to the left, `C-F` to the right. Backspace erases the character to the left of the cursor, `C-D` the character to its right. `C-K` kills (erases) the rest of the line to the right of the cursor, `C-Y` yanks back the last killed string. `C-underscore` undoes the last change you made; it can be repeated for cumulative effect.

A.2 History Substitution

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. `C-P` moves one line up (back) in the history buffer, `C-N` moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the `Return` key passes the current line to the interpreter. `C-R` starts an incremental reverse search; `C-S` starts a forward search.

A.3 Key Bindings

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called ~/.inputrc. Key bindings have the form

```
key-name: function-name
```

or

```
"string": function-name
```

and options can be set with

```
set option-name value
```

For example:

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Note that the default binding for `Tab` in Python is to insert a `Tab` character instead of Readline's default filename completion function. If you insist, you can override this by putting

```
Tab: complete
```

in your ~/.inputrc. (Of course, this makes it harder to type indented continuation lines if you're accustomed to using `Tab` for that purpose.)

Automatic completion of variable and module names is optionally available. To enable it in the interpreter's interactive mode, add the following to your startup file:[A.1]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

This binds the `Tab` key to the completion function, so hitting the `Tab` key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `"."` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression.

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the

startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environment. You may find it convenient to keep some of the imported modules, such as os, which turn out to be needed in most sessions with the interpreter.

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it:  "export PYTHONSTARTUP=/max/home/itamar/.pystartup" in bash.
#
# Note that PYTHONSTARTUP does *not* expand "~", so you have to put in the
# full path to your home directory.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

A.4 Commentary

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

Footnotes

... file:[A.1]
Python will execute the contents of a file identified by the PYTHONSTARTUP environment variable when you start an interactive interpreter.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.

Python Tutorial

This Appendix was left untranslated.

B. Floating Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the decimal fraction

```
0.125
```

has value 1/10 + 2/100 + 5/1000, and in the same way the binary fraction

```
0.001
```

has value 0/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction 1/3. You can approximate that as a base 10 fraction:

```
0.3
```

or, better,

```
0.33
```

or, better,

```
0.333
```

and so on. No matter how many digits you're willing to write down, the result will never be exactly 1/3, but will be an increasingly better approximation of 1/3.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, 1/10 is the infinitely repeating fraction

```
0.0001100110011001100110011001100110011001100110011...
```

Stop at any finite number of bits, and you get an approximation. This is why you see things like:

```
>>> 0.1
0.10000000000000001
```

On most machines today, that is what you'll see if you enter 0.1 at a Python prompt. You may not, though, because the number of bits used by the hardware to store floating-point values can vary across machines, and Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

instead! The Python prompt uses the builtin `repr()` function to obtain a string version of everything it displays. For floats, `repr(`*float*`)` rounds the true decimal value to 17 significant digits, giving

```
0.10000000000000001
```

`repr(`*float*`)` produces 17 significant digits because it turns out that's enough (on most machines) so that `eval(repr(`x`))` == x exactly for all finite floats x, but rounding to 16 digits is not enough to make that true.

Note that this is in the very nature of binary floating-point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

Python's builtin `str()` function produces only 12 significant digits, and you may wish to use that instead. It's unusual for `eval(str(`x`))` to reproduce x, but the output may be more pleasant to look at:

```
>>> print str(0.1)
0.1
```

It's important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly 1/10, you're simply rounding the *display* of the true machine value.

Other surprises follow from this one. For example, after seeing

```
>>> 0.1
0.10000000000000001
```

you may be tempted to use the `round()` function to chop it back to the single digit you expect. But that makes no difference:

```
>>> round(0.1, 1)
0.10000000000000001
```

The problem is that the binary floating-point value stored for "0.1" was already the best possible binary approximation to 1/10, so trying to round it again can't make it better: it was already as good as it gets.

Another consequence is that since 0.1 is not exactly 1/10, summing ten values of 0.1 may not yield exactly 1.0, either:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.99999999999999989
```

Binary floating-point arithmetic holds many surprises like this. The problem with "0.1" is explained in precise detail below, in the "Representation Error" section. See *The Perils of Floating Point* for a more complete account of other common surprises.

As that says near the end, ``there are no easy answers.'' Still, don't be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic, and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. `str()` usually suffices, and for finer control see the discussion of Python's `%` format operator: the `%g`, `%f` and `%e` format codes supply flexible and easy ways to round float results for display.

B.1 Representation Error

This section explains the ``0.1'' example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is

the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect:

```
>>> 0.1
0.10000000000000001
```

Why is that? 1/10 is not exactly representable as a binary fraction. Almost all machines today (November 2000) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 "double precision". 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form *J*/2**_N_ where *J* is an integer containing exactly 53 bits. Rewriting

```
 1 / 10 ~= J / (2**N)
```

as

```
J ~= 2**N / 10
```

and recalling that *J* has exactly 53 bits (is >= 2**52 but < 2**53), the best value for *N* is 56:

```
>>> 2**52
4503599627370496L
>>> 2**53
9007199254740992L
>>> 2**56/10
7205759403792793L
```

That is, 56 is the only value for *N* that leaves *J* with exactly 53 bits. The best possible value for *J* is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6L
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

```
>>> q+1
7205759403792794L
```

Therefore the best possible approximation to 1/10 in 754 double precision is that over 2**56, or

```
7205759403792794 / 72057594037927936
```

Note that since we rounded up, this is actually a little bit larger than 1/10; if we had not rounded up, the quotient would have been a little bit smaller than 1/10. But in no case can it be *exactly* 1/10!

So the computer never ``sees'' 1/10: what it sees is the exact fraction given above, the best 754 double approximation it can get:

```
>>> .1 * 2**56
7205759403792794.0
```

If we multiply that fraction by 10**30, we can see the (truncated) value of its 30 most significant decimal digits:

```
>>> 7205759403792794 * 10**30 / 2**56
100000000000000005551115123125L
```

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.100000000000000005551115123125. Rounding that to 17 significant digits gives the 0.10000000000000001 that Python displays (well, will display on any 754-conforming platform that does best-possible input and output conversions in its C library -- yours may not!).

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.

Python Tutorial

This Appendix was left untranslated.

C. History and License

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes

Release	Derived from	Year	Owner	GPL compatible?
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes
2.3.3	2.3.2	2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes
2.4.1	2.4	2005	PSF	yes
2.4.2	2.4.1	2005	PSF	yes
2.4.3	2.4.2	2006	PSF	yes
2.5	2.4	2006	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.5

1. This LICENSE AGREEMENT is between the Python Software Foundation (``PSF''), and the Individual or Organization (``Licensee'') accessing and otherwise using Python 2.5 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.5 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., ``Copyright © 2001-2006 Python Software Foundation; All Rights Reserved'' are retained in Python 2.5 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.5 or any part thereof, and wants to make the

derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.5.

4. PSF is making Python 2.5 available to Licensee on an ``AS IS'' basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.5 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.5 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.5, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.5, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com (``BeOpen''), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (``Licensee'') accessing and otherwise using this software in source or binary form and its associated documentation (``the Software'').

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an ``AS IS'' basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES,

EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the ``BeOpen Python'' logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (``CNRI''), and the Individual or Organization (``Licensee'') accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., ``Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved'' are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): ``Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located

on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013.''

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an ``AS IS'' basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the ``ACCEPT'' button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
```

```
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OW
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
```

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this softwar
```

```
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
GAI_ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, TH
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPO
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR GAI_ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQU
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON GAI_ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GAI_AN
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
  --------------------------------------------------------------------
 /                      Copyright (c) 1996.
|          The Regents of the University of California.
|                      All rights reserved.
|
|   Permission to use, copy, modify, and distribute this software for
|   any purpose without fee is hereby granted, provided that this en-
|   tire notice is included in all copies of any software which is or
|   includes  a  copy  or  modification  of  this software and in all
|   copies of the supporting documentation for such software.
|
|   This  work was produced at the University of California, Lawrence
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48
|   between  the  U.S.  Department  of  Energy and The Regents of the
|   University of California for the operation of UC LLNL.
|
|                              DISCLAIMER
|
|   This  software was prepared as an account of work sponsored by an
|   agency of the United States Government. Neither the United States
|   Government  nor the University of California nor any of their em-
|   ployees, makes any warranty, express or implied, or  assumes  any
|   liability  or  responsibility  for the accuracy, completeness, or
|   usefulness of any information,  apparatus,  product,  or  process
|   disclosed,   or  represents  that  its  use  would  not  infringe
|   privately-owned rights. Reference herein to any specific  commer-
|   cial  products,  process,  or  service  by trade name, trademark,
|   manufacturer, or otherwise, does not  necessarily  constitute  or
|   imply  its endorsement, recommendation, or favoring by the United
|   States Government or the University of California. The views  and
|   opinions  of authors expressed herein do not necessarily state or
|   reflect those of the United States Government or  the  University
```

```
|   of  California,  and shall not be used for advertising or product
 \  endorsement purposes.
  ---------------------------------------------------------------------
```

C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
      http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
      references to Ghostscript; clarified derivation from RFC 1321;
      now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
      added conditionalization for C++ compilation from Martin
      Purschke <purschke@bnl.gov>.
```

```
1999-05-03 lpd Original version.
```

C.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.6 Cookie management

The `Cookie` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
```

```
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.7 Profiling

The `profile` and `pstats` modules contain the following notice:

```
Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software
and its associated documentation for any purpose (subject to the
restriction in the following sentence) without fee is hereby granted,
provided that the above copyright notice appears in all copies, and
that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of InfoSeek not be used in
advertising or publicity pertaining to distribution of the software
without specific, written prior permission.  This permission is
explicitly restricted to the copying and modification of the software
to remain in Python, compiled Python, or other languages (such as C)
wherein the modified or derived code is exclusively imported into a
Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY
SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER
RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.8 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
```

```
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserve


Permission to use, copy, modify, and distribute this Python software ar
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies
and that both that copyright notice and this permission notice appear i
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining
distribution of the software without specific, written prior permissior
```

C.3.9 UUencode and UUdecode functions

The uu module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with python standard
```

C.3.10 XML Remote Procedure Calls

The xmlrpclib module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh
```

```
By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.

Python Tutorial

This Appendix was left untranslated.

D. Glossary

>>>
The typical Python prompt of the interactive shell. Often seen for code examples that can be tried right away in the interpreter.

...
The typical Python prompt of the interactive shell when entering code for an indented code block.

BDFL
Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

byte code
The internal representation of a Python program in the interpreter. The byte code is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to byte code can be avoided). This ``intermediate language'' is said to run on a ``virtual machine'' that calls the subroutines corresponding to each bytecode.

classic class
Any class which does not inherit from `object`. See *new-style class*.

coercion
The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` builtin function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

complex number
An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has builtin support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

descriptor
Any *new-style* object that defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, writing *a.b* looks up the object *b* in the class dictionary for *a*, but if *b* is a descriptor, the defined method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

dictionary
An associative array, where arbitrary keys are mapped to values. The use of `dict` much resembles that for `list`, but the keys can be any object with a `__hash__()` function, not just integers starting from zero. Called a hash in Perl.

duck-typing
Pythonic programming style that determines an object's type by inspection of its method or attribute signature rather than by explicit relationship to some type object ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. Instead, it typically employs `hasattr()` tests or *EAFP* programming.

EAFP
Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style that is common in many other languages such as C.

__future__
A pseudo module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to `2`. If the module in which it is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression `11/4` would evaluate to `2.75`. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

generator
A function that returns an iterator. It looks like a normal function except that values are returned to the caller using a `yield` statement instead of a `return` statement. Generator functions often contain one or more `for` or `while` loops that `yield` elements back to the caller. The function execution is stopped at the `yield` keyword (returning the result) and is resumed there when the next

element is requested by calling the `next()` method of the returned iterator.

generator expression
An expression that returns a generator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

GIL
See *global interpreter lock*.

global interpreter lock
The lock used by Python threads to assure that only one thread can be run at a time. This simplifies Python by assuring that no two processes can access the same memory at the same time. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of some parallelism on multi-processor machines. Efforts have been made in the past to create a ``free-threaded'' interpreter (one which locks shared data at a much finer granularity), but performance suffered in the common single-processor case.

IDLE
An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment that ships with the standard distribution of Python. Good for beginners, it also serves as clear example code for those wanting to implement a moderately sophisticated, multi-platform GUI application.

immutable
An object with fixed value. Immutable objects are numbers, strings or tuples (and more). Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

integer division
Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to `2` in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *__future__*.

interactive
Python has an interactive interpreter which means that you can try out things and immediately see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

interpreted
Python is an interpreted language, as opposed to a compiled one. This means that the source files can be run directly without first creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

iterable
A container object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the builtin function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator
An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data is available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code that attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

LBYL
Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many `if` statements.

list comprehension
A compact way to process all or a subset of elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing hex numbers (0x..) that are even and in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

mapping
A container object (such as `dict`) that supports arbitrary key lookups using the special method `__getitem__()`.

metaclass
The class of a class. Class definitions create a class name, a class dictionary,

and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

mutable
Mutable objects can change their value but keep their `id()`. See also *immutable*.

namespace
The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and builtin namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules respectively.

nested scope
The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class
Any class that inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

Python3000
A mythical python release, not required to be backward compatible, with telepathic interface.

__slots__
A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence
An *iterable* which supports efficient element access using integer indices via the `__getitem__()` and `__len__()` special methods. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a

sequence because the lookups use arbitrary *immutable* keys rather than integers.

Zen of Python
Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing ``import this'' at the interactive prompt.

Release 2.5, documentation updated on 19th September, 2006.
See About this document... for information on suggesting changes.