O'REILLY®

Dạo một vòng Python

Jake VanderPlas

O'REILLY®

Một số nguồn bổ trợ

4 cách đơn giản để học tập và bắt kịp xu hướng

Bản tin lập trình
Cập nhận những bản tin về lập trình mới nhất hàng tuần qua thư điện tử.
oreilly.com/programming/newsletter

Chương trình trực tiếp miễn phí
Tìm hiểu những chủ đề về lập trình từ những chương trình trực tiếp chuyên sâu.
webcasts.oreilly.com

O'Reilly Radar
Những phân tích chuyên sâu về những công nghệ mới.
radar.oreilly.com

Hội nghị
Đắm mình trong không gian học thuật trong những hội nghị sắp tới của O'Reilly.
conferences.oreilly.com

Dạo một vòng python (A Whirlwind Tour of Python)
Jake VanderPlas

In tại Hoa Kỳ.

Xuất bản bởi Tập đoàn O'Reilly Media. 1005 Cao tốc Gravenstein phía Bắc, Sebastopol, California, 95472.

Có thể mua sách O'Reilly cho mục đích giáo dục, kinh doanh hoặc khuyến mãi. Phiên bản trực tuyến cũng có sẵn tại "*http://safaribooksonline.com*". Để biết thêm thông tin chi tiết, xin hãy liên hệ với bộ phận kinh doanh của của chúng tôi: 800-998-9938 hoặc *corporate@oreilly.com.*

Biên tập: Dawn Schanafelt

Thiết kế nội dung: David Futato

Sản xuất: Kristen Brown

Thiết kế bìa: Karen Montgomery

Sửa bản in: Jasmine Kwityn

Minh họa: Rebecca Demarest

Tháng 8, 2016: Ấn bản đầu tiền

Lịch sử chỉnh sửa ấn bản đầu tiên
2016-08-10: Xuất bản lần đầu.

Mục lục

Dạo một vòng Python

Lời mở đầu

Được hình thành vào cuối những năm tám mươi như là một ngôn ngữ giảng dạy và kịch bản, Python đã trở thành một công cụ thiết yếu cho nhiều lập trình viên, kỹ sư, nhà nghiên cứu và nhà khoa học dữ liệu trong các học viện và ngành công nghiệp. Là một nhà thiên văn tập trung vào việc xây dựng và thúc đẩy các công cụ mở miễn phí cho ngành khoa học chuyên sâu về dữ liệu, tôi thấy Python phù hợp một cách gần như hoàn hảo cho các loại vấn đề mà tôi gặp phải hàng ngày, cho dù đó là việc trích xuất ý nghĩa từ các bộ dữ liệu thiên văn lớn, thu thập và chuyển đổi dữ liệu nguồn của trang Web hay tự động hóa các nhiệm vụ nghiên cứu hàng ngày.

Sự hấp dẫn của Python đến từ sự đơn giản và vẻ đẹp của nó, cũng như sự tiện lợi của hệ sinh thái lớn gồm các công cụ miền chuyên biệt đã được xây dựng trong đó. Ví dụ, hầu hết câu lệnh Python trong tính toán khoa học và khoa học dữ liệu được xây dựng xung quanh một nhóm các gói (package) hoàn chỉnh và hữu ích.

- NumPy cung cấp khả năng lưu trữ và tính toán hiệu quả cho các mảng dữ liệu đa chiều.
- SciPy chứa một loạt các công cụ số như tích phân và nội suy.
- Pandas cung cấp một đối tượng DataFrame cùng với các phương thức mạnh mẽ để điều khiển, lọc, nhóm và chuyển đổi dữ liệu.
- Matplotlib cung cấp một giao diện hữu ích để tạo ra các phác họa và bản vẽ.
- Scikit-Learn cung cấp một bộ công cụ thống nhất để đưa vào dữ liệu các thuật toán cho phép máy tính khả năng tự học hỏi.
- IPython/Jupyter cung cấp một thiết bị đầu cuối nâng cao và môi trường tương tác máy tính hữu ích cho phân tích thăm dò, cũng như tạo ra các tài liệu tương tác, thực thi. Ví dụ, bản thảo của cuốn sách này được soạn hoàn toàn với Jupyter notebook.

Không kém phần quan trọng đó là một số lượng lớn các công cụ và các gói đi kèm: nếu có một nhiệm vụ phân tích khoa học hoặc dữ liệu mà bạn muốn thực hiện, rất có thể có ai đó đã viết sẵn một gói để làm điều đó cho bạn.

Tuy nhiên, để khai thác sức mạnh của hệ sinh thái khoa học dữ liệu này, trước tiên, đòi hỏi phải làm quen với ngôn ngữ Python. Tôi thường bắt gặp các sinh viên và đồng nghiệp có nền tảng (đôi khi là hiểu biết sâu rộng) ở một số ngôn ngữ như, MAT MATLAB, IDL, R, Java, C ++, v.v., và muốn có một cái nhìn ngắn gọn nhưng toàn diện về ngôn ngữ Python tương xứng với trình độ hiểu biết của họ, hơn là bắt đầu từ con số không. Cuốn sách này phù hợp với nhu cầu đó.

Dù vậy, cuốn sách này không nhắm tới việc trở thành một cuốn cẩm nang toàn diện về lập trình nói chung hay ngôn ngữ Python nói riêng; nếu đó là thứ bạn đang kiếm tìm, hãy thử điểm qua một số gợi ý tham khảo được liệt kê tại mục “Tài nguyên cho việc học thêm” trang 98. Thay vào đó, cuốn sách này sẽ cung cấp những kiến thức sơ khai về cú pháp và ý nghĩa của chúng, kiểu dữ liệu tích hợp sẵn (built-in data) và cấu trúc, định nghĩa hàm, câu lệnh điều khiển, và các khía cạnh khác của ngôn ngữ này. Mục tiêu của tôi là mang đến cho độc giả một nền tảng vững chắc để từ đó khám phá những bộ môn khoa học máy tính như đã nói ở trên.

Sử dụng dòng lệnh mẫu

Tài liệu bổ sung (mã lệnh mẫu, IPython notebooks, v.v.) đều có thể tải về từ đường dẫn https://github.com/jakevdp/WhirlwindTour OfPython/.

Cuốn sách này sẽ giúp bạn hoàn thành công việc của mình. Nhìn chung, bạn có thể sử dụng những đoạn mã có trong cuốn sách này vào trong chương trình hay tài liệu của mình. Bạn không cần phải xin phép chúng tôi trừ phi bạn cần sao chép những đoạn mã quan trọng. Ví dụ, bạn không cần phải xin phép khi sử dụng một hay nhiều đoạn mã trong cuốn sách này vào chương trình của mình, bán hay phân phối CD-ROM chứa các ví dụ từ sách O’Reilly, trả lời các câu hỏi hay bằng cách viện dẫn từ cuốn sách này. Nhưng việc kết hợp nhiều đoạn mã quan trọng từ cuốn sách này vào hồ sơ sản phẩm của bạn càn phải được cho phép.

Chúng tôi đánh giá cao sự ghi nhận, nhưng không bắt buộc. Một sự ghi nhận bao gồm tựa đề, tên tác giả, nhà xuất bản và mã ISBN của cuốn sách này. Ví dụ: *“A Whirlwind Tour of Python* by Jake VanderPlas (O’Reilly). Copyright 2016 O’Reilly Media, Inc., 978-1-491-96465-1.”

Nếu bạn cảm thấy việc sử dụng những đoạn mã của mình nằm ngoài những sự cho phép ở trên, đừng ngần ngại liên lạc với chúng tôi qua địa chỉ permissions@oreilly.com.

Cài đặt và một số vấn đề thực tế

Việc cài đặt Python và bộ thư viện cho phép tính toán khoa học rất đơn giản cho dù bạn sử dụng Windows, Linux hay Mac OS X. Phần này sẽ đưa ra một số cân nhắc khi thiết lập máy tính.

Python 2 và Python 3

Cuốn sách này sử dụng cú pháp của Python 3, chứa các cải tiến về ngôn ngữ không tương thích với những phiên bản 2. x của Python. Mặc dù Python 3.0 ra mắt lần đầu vào năm 2008, nhưng phải mất rất lâu nó mới được chấp nhận, đặc biệt là trong cộng đồng các nhà khoa học và phát triển website. Lý do chính là cần thời gian để các gói và bộ công cụ thiết yếu tương thích với ngôn ngữ mới. Tuy nhiên, từ đầu năm 2014, phiên bản ổn định của các công cụ quan trọng bậc nhất trong hệ sinh thái khoa học dữ liệu đã hoàn toàn tương thích với cả Python 2 và 3, và đó là lý do mà cuốn sách này sử dụng cú pháp Python 3. Dù trong trường hợp đó, đa số những đoạn mã trong cuốn sách này cũng sẽ hoạt động mà không cần phải chỉnh sửa trên Python 2: trong trường hợp có những cú pháp không tương thích với Py2, tôi sẽ cố gắng lưu ý nó một cách rõ ràng nhất.

Cài đặt với conda

Có rất nhiều cách để cài đặt Python, nhưng cách mà tôi muốn đề xuất (đặc biệt nếu bạn muốn sử dụng những công cụ khoa học dữ liệu như đã nói ở trên) là thông qua bộ phân phối đa nền tảng Anaconda. Anaconda có hai dạng:

- Miniconda cung cấp một trình thông dịch Python, cùng với một công cụ dòng lệnh gọi là conda hoạt động như một trình quản lý gói đa nền tảng hướng đến các gói Python, tương tự như công cụ `apt` hoặc `yum` rất quen thuộc với người dùng Linux.
- Anaconda bao gồm cả Python và `conda`, thêm vào đó là một bộ các gói được cài đặt sẵn hướng tới tính toán khoa học.

Tất cả những gói đi kèm theo Anaconda đều có thể cài đặt một cách thủ công trên Miniconda, vì lý do này, tôi khuyên bạn nên bắt đầu với Miniconda.

Bắt đầu bằng việc tải xuống và cài đặt Miniconda (hãy chắc rằng bạn chọn đúng phiên bản cho Python 3) và sau đó là cài đặt IPython:

```
[~] $ conda install ipython-notebook
```

Để biết thêm thông tin về conda, bao gồm cả thông tin về tạo lập và sử dụng môi trường conda, hãy tham khảo tài liệu về Miniconda đã được liên kết ở trên.

"Triết lý" của Python

Những người yêu thích Python thường có thể nhanh chóng chỉ ra Python "trực quan", "đẹp đẽ" hay "thú vị" ra sao. Tôi có xu hướng đồng ý, và cũng nhận ra rằng cái đẹp, trực quan, thú vị đó thường đi cùng với sự quen thuộc. Nhưng đối với những người đã quen với những ngôn ngữ khác, những nhận xét hoa mỹ trên hơi có phần tự mãn. Tuy nhiên, tôi hy vọng rằng nếu bạn cho Python một cơ hội, bạn sẽ thấy được những dấu ấn đó đến từ đâu.

Và nếu bạn thật sự muốn đi sâu tìm hiểu về triết lý lập trình, thứ đã thu hút những người muốn sử dụng thuần thục sức mạnh lập trình của Python, đó là một điều thú vị nho nhỏ tồn tại trong trình thông dịch Python. Hãy nhắm mắt lại, ngẫm nghĩ một chút, và chạy dòng lệnh import this

```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to
guess.
There should be one--and preferably only one--obvious
way to do it.
Although that way may not be obvious at first unless
you're Dutch.
Now is better than never.
Although never is often better than *right* now.
```

```
If the implementation is hard to explain, it's a bad
idea.
If the implementation is easy to explain, it may be a
good idea.
Namespaces are one honking great idea--let's do more of
those!
```

Bản dịch:

```
Đẹp tốt hơn xấu.
Rõ ràng tốt hơn che đậy.
Đơn giản tốt hơn phức tạp.
Phức tạp tốt hơn rắc rối.
Dàn phẳng tốt hơn lồng nhau.
Thưa tốt hơn dày.
Dễ đọc.
Trường hợp đặc biệt cũng đủ phá vỡ quy tắc.
Mặc dù thực tế đè bẹp nguyên bản.
Không bao giờ lặng lẽ bỏ qua lỗi.
Trừ phi đã bắt nó im lặng.
Trước một điều mơ hồ, từ chối sự cám dỗ.
Nên có một, và tốt nhất là chỉ một cách để làm điều đó.
Mặc dù ban đầu nó có thể sẽ không rõ ràng trừ khi bạn là
người Hà Lan.
Bây giờ tốt hơn không bao giờ.
Mặc dù không bao giờ thường tốt hơn “ngay” lúc này.
Nếu khó có thể giải thích cách thực hiện, đó là một ý
tưởng tồi.
Nếu dễ dàng giải thích được, đó có thể là một ý tưởng
hay.
Không gian tên là một ý tưởng tuyệt vời, hãy làm ra
nhiều hơn!
```


Với những lưu ý trên, hãy cùng bắt đầu hành trình với ngôn ngữ Python!

Cách để khởi chạy một đoạn mã Python

Python là một ngôn ngữ linh hoạt, và có rất nhiều cách để sử dụng nó tùy vào từng nhiệm vụ cụ thể. Một điều để phân biệt Python với những ngôn ngữ lập trình khác là nó là ngôn ngữ *thông dịch (interpreted)* chứ không phải *biên dịch (compiled)*. Có nghĩa là nó thực thi từng dòng lệnh, điều này cho phép lập trình

tương tác theo một cách mà ta không thể làm trực tiếp đối với những ngôn ngữ biên dịch như Fortran, C, hay Java. Phần này sẽ giới thiệu bốn cách chính để khởi chạy một đoạn mã Python: *trình thông dịch* Python, *trình thông dịch* IPython, thông qua các *tập lệnh độc lập*, hoặc trong *Jupyter notebook*.

Trình thông dịch Python (Python interpreter)

Cách đơn giản nhất để khởi chạy một đoạn mã Python là thực thi từng dòng lệnh với *trình thông dịch Python*. Trình thông dịch Python có thể khởi động bằng cách cài đặt ngôn ngữ Python (xem phần trước) và nhập `python` vào môi trường dòng lệnh (command prompt) (Terminal của hệ điều hành Mac OS X và Unix/Linux, hoặc ứng dụng Command Prompt của Windows):

```
$ python
Python 3.5.1 |Continuum Analytics, Inc.| (default, Dec
7...
Type "help", "copyright", "credits" or "license" for
more...
>>>
```

Với trình thông dịch này, bạn có thể nhập và thực thi từng dòng lệnh. Dưới đây là ví dụ sử dụng trình thông dịch như một máy tính đơn giản, thực hiện các phép tính và gán giá trị cho biến:

```
>>> 1 + 1
2
>>> x = 5
>>> x * 3
15
```

Trình thông dịch làm cho việc chạy thử các đoạn mã Python nhỏ và các chuỗi hành động ngắn trở nên tiện lợi.

Trình thông dịch IPython (The IPython interpreter)

Nếu dành nhiều thời gian với trình thông dịch Python đơn giản, bạn sẽ nhận thấy nó thiếu rất nhiều tính năng của một môi trường phát triển tích hợp hoàn chỉnh. Một trình thông dịch thay thế có tên là *IPython* được cài đặt cùng với Anaconda, bao gồm một loạt các cải tiến thuận tiện cho việc thông dịch Python cơ bản. Có thể khởi động bằng cách nhập lệnh ipython vào môi trường dòng lệnh:

```
$ ipython Python 3.5.1 |Continuum Analytics, Inc.|
(default, Dec 7...
Type "copyright", "credits" or "license" for more
information.

IPython 4.0.0 -- An enhanced Interactive Python.
?             -> Introduction and overview of IPython's
features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?    -> Details about 'object', use 'object??' for
extra...

In [1]:
```

Điểm khác nhau trực quan nhất giữa trình thông dịch Python và trình thông dịch nâng cao IPython nằm ở dấu nhắc lệnh: Python sử dụng dấu >>> làm mặc định, trong khi IPython đánh số cho dòng lệnh (ví dụ, `In [1]:`). Nhưng bằng cách nào thì chúng ta cũng có thể thực thi từng dòng lệnh như bình thường:

```
In [1]: 1 + 1
Out[1]: 2

In [2]: x = 5

In [3]: x * 3
Out[3]: 15
```

Không chỉ đầu vào được đánh số mà cả đầu ra cũng được đánh số. IPython còn có sẵn một loạt các tính năng hữu ích khác; một số gợi ý để tìm hiểu thêm được viết tại mục “Tài nguyên cho việc học thêm” trang 98.

Các tập lệnh Python độc lập

Chạy từng dòng lệnh trong đoạn mã Python có thể hữu ích trong một số trường hợp, nhưng đối với chương trình có độ phức tạp cao, sẽ thuận tiện hơn nếu lưu đoạn mã vào tập tin và thực hiện tất cả cùng lúc. Theo quy ước, các tập lệnh Python được lưu trong các tập tin có phần mở rộng là .py. Ví dụ, hãy tạo ra một tập lệnh có tên test.py có chứa những dòng lệnh sau:

```
# file: test.py
print("Running test.py")
x = 5
print("Result is", 3 * x)
```

Để khởi chạy tập tin này, hãy đảm bảo rằng nó nằm trong thư mục hiện hành và nhập lệnh python "tên tập tin" vào môi trường dòng lệnh:

```
$ python test.py
Running test.py
Result is 15
```

Đối với những chương trình phức tạp hơn, việc tạo các tập lệnh độc lập như thế này là điều bắt buộc.

Jupyter notebook

Một kết hợp hữu ích của thiết bị đầu cuối tương tác và tập lệnh độc lập chính là *Jupyter notebook*, một định dạng tài liệu cho phép mã thực thi, văn bản được định dạng, đồ họa và thậm chí các tính năng tương tác kết hợp thành một tài liệu đơn nhất. Mặc dù khởi đầu chỉ với định dạng Python, nhưng nó đã được làm cho tương thích với một số lượng lớn các ngôn ngữ lập trình và hiện là một phần thiết yếu của *Dự án Jupyter (Jupyter Project)*. Công cụ này hữu ích cả trên phương diện môi trường phát triển và chia sẻ công việc thông qua những bản ghi có giá trị tính toán và định hướng dữ liệu là một tổ hợp của mã, số liệu, dữ liệu và văn bản.

Hướng dẫn nhanh về Cú pháp ngôn ngữ Python

Python ban đầu được phát triển như một ngôn ngữ giảng dạy, nhưng việc dễ sử dụng và cú pháp rõ ràng đã giúp cho nó được đón nhận bởi những người mới và cả những chuyên gia. Cú pháp Python rõ ràng tới mức một số người gọi nó là "một dạng Giả mã (pseudocode) có thể thực thi được", và theo kinh nghiệm của tôi thì việc đọc và hiểu một tập lệnh Python thường dễ hơn nhiều so với đọc một tập lệnh tương tự được viết trong C. Và chúng ta sẽ bắt đầu thảo luận về các tính năng chính của Cú pháp Python.

Cú pháp ám chỉ cấu trúc của ngôn ngữ (nghĩa là những gì cấu thành nên một chương trình chính xác). Lúc này, chúng ta sẽ không tập trung vào ý nghĩa của chúng (ý nghĩa của các từ và ký hiệu trong cú pháp), nhưng sẽ quay lại vấn đề này sau.

Xem xét ví dụ về đoạn mã dưới đây:

```
In [1]: # thiết lập biến midpoint
        midpoint = 5

        # tạo hai danh sách trống
        lower = []; upper = []

        # chia dãy số thành thấp và cao
        for i in range(10):
            if (i < midpoint):
                lower.append(i)
            else:
                upper.append(i)

        print("lower:", lower)
        print("upper:", upper)

lower: [0, 1, 2, 3, 4]
upper: [5, 6, 7, 8, 9]
```

Tập lệnh này hơi ngớ ngẩn, nhưng nó minh họa ngắn gọn một số khía cạnh quan trọng của cú pháp Python. Hãy xem qua nó thảo luận về một số đặc điểm cú pháp của Python.

Ghi chú được đánh dấu bởi dấu

Tập lệnh bắt đầu bằng một ghi chú:

```
# thiết lập midpoint
```

Ghi chú trong Python được thể hiện bằng một dấu thăng `(#)`, và bất kỳ thứ gì đứng sau dấu thăng trên dòng đó đều bị trình thông dịch bỏ qua. Điều này có nghĩa là có thể có các ghi chú độc lập như trên, cũng như các ghi chú trong một câu lệnh. Ví dụ:

```
x += 2  # viết tắt cho x = x + 2
```

Python không có bất kỳ cú pháp nào cho các ghi chú đa dòng, chẳng hạn như cú pháp `/* ... */` được sử dụng trong C và C ++, tuy nhiên các chuỗi đa dòng

thường được sử dụng để thay thế cho các ghi chú đa dòng (đọc thêm tại “Thao tác chuỗi và biểu thức chính quy” trang 76).

Chấm dứt câu lệnh khi kết thúc dòng

Dòng tiếp theo trong tập lệnh là

```
midpoint = 5
```

Đây là một phép gán, trong đó ta tạo một biến có tên là `midpoint` và gán cho nó giá trị là 5. Có thể để ý rằng kết thúc của câu lệnh này chính là kết thúc của dòng lệnh. Điều này trái ngược với các ngôn ngữ như C và C ++, trong đó mọi câu lệnh phải kết thúc bằng dấu chấm phẩy `(;)`.

Trong Python, nếu muốn một câu lệnh tiếp tục đến dòng tiếp theo, có thể sử dụng dấu \ để chỉ ra điều này:

```
In [2]: x = 1 + 2 + 3 + 4 +\
          5 + 6 + 7 + 8
```

Cũng có thể tiếp tục biểu thức trên dòng tiếp theo trong ngoặc đơn mà không cần sử dụng \ để đánh dấu:

```
In [3]: x = (1 + 2 + 3 + 4 +
          5 + 6 + 7 + 8)
```

Hầu hết các hướng dẫn về quy chuẩn Python đều đề xuất cách làm thứ hai để nối tiếp dòng (trong ngoặc đơn) hơn là cách đầu tiên (sử dụng dấu \).

Dấu chấm phẩy có thể chấm dứt một câu lệnh

Đôi khi nó có thể hữu ích để đặt nhiều câu lệnh trên cùng một dòng. Phần tiếp theo của tập lệnh là:

```
lower = []; upper = []
```

Ví dụ này cho thấy dấu chấm phẩy `(;)` quen thuộc trong C có thể được sử dụng tùy ý trong Python để đặt hai câu lệnh trên cùng một dòng. Về mặt chức năng, nó hoàn toàn tương đương với cách viết:

```
lower = []
upper = []
```

Sử dụng dấu chấm phẩy để đặt nhiều câu lệnh trên cùng một dòng thường không được khuyến khích trong hầu hết các hướng dẫn về quy chuẩn Python, mặc dù đôi khi nó có vẻ khá thuận tiện.

Thụt lề: Vấn đề về khoảng trắng!

Tiếp theo, cùng đến với khối mã chính:

```
for i in range(10):
    if i < midpoint:
        lower.append(i)
    else:
        upper.append(i)
```

Đây là một câu lệnh điều khiển tổng hợp bao gồm một vòng lặp và một điều kiện - chúng ta sẽ xem xét các loại câu lệnh này một chút. Bây giờ, hãy xét đến một điều mà có lẽ là tính năng gây tranh cãi nhất trong cú pháp Python: khoảng trắng cũng có nghĩa!

Trong các ngôn ngữ lập trình, một khối mã là một tập hợp của các câu lệnh nên được coi là một đơn vị. Ví dụ như trong C, các khối mã được biểu thị bằng ngoặc nhọn:

```
// C code
for(int i=0; i<100; i++)
   {
      // cặp dấu ngoặc nhọn biểu thị khối mã
      total += i;
   }
```

Trong Python, khối mã được biểu thị bằng cách thụt lề:

```
for i in range(100):
    # thụt lề biểu thị khối mã
    total += i
```

Khối mã luôn đứng sau dấu hai chấm (`:`) của dòng liền trước đó.

Việc thụt lề giúp tạo ra sự đồng nhất, dễ đọc mà nhiều người thấy hấp dẫn ở Python. Nhưng nó có thể gây nhầm lẫn cho những người không quen thuộc; ví dụ, hai đoạn mã sau sẽ cho ra kết quả khác nhau:

```
>>> if x < 4:
...     y = x * 2
...     print(x)
```

```
>>> if x < 4:
...     y = x * 2
... print(x)
```

Trong đoạn mã bên trái, `print(x)` nằm trong khối thụt lề và sẽ chỉ được thực hiện nếu `x` bé hơn `4`. Trong đoạn mã trên bên phải, `print(x)` nằm ngoài khối, và sẽ được thực hiện bất kể giá trị của `x`!

Việc Python cho phép các khoảng trắng có nghĩa thường gây ngạc nhiên cho các lập trình viên đã quen với các ngôn ngữ khác, nhưng trong thực tế, nó có thể giúp đoạn mã đồng nhất và dễ đọc hơn nhiều so với các ngôn ngữ không thực hiện việc thụt các khối mã. Nếu thấy việc sử dụng khoảng trắng của Python là không thể chấp nhận được, tôi khuyến khích bạn hãy dùng thử như tôi đã làm, bạn có thể sẽ đánh giá cao nó.

Cuối cùng, nên lưu ý rằng lượng khoảng trắng được sử dụng để thụt lề cho các khối mã là tùy thuộc vào người dùng, miễn là nó đồng nhất trong toàn bộ tập lệnh. Theo quy ước, hầu hết các hướng dẫn về quy chuẩn đều khuyên nên thụt lề cho các khối mã bằng bốn khoảng trắng, và đó cũng là quy ước trong cuốn sách này. Lưu ý rằng nhiều trình chỉnh sửa văn bản như Emacs và Vim có chế độ thụt lề bằng bốn khoảng trắng cho Python một cách tự động.

Khoảng trắng *nằm trong* một dòng không quan trọng

Mặc dù việc *khoảng trắng có nghĩa* đúng với khoảng trắng *đứng trước* các dòng (biểu thị một khối mã), thì khoảng trắng *trong* các dòng của đoạn mã Python lại không quan trọng. Ví dụ, cả ba biểu thức này đều tương đương nhau:

```
In [4]: x=1+2
        x = 1 + 2
        x           =           1     +                 2
```

Lạm dụng tính linh hoạt này có thể dẫn đến các vấn đề về khả năng đọc mã trong thực tế, lạm dụng khoảng trắng thường là một trong những biểu hiện chính của việc làm xáo trộn đoạn mã (mà một số người làm để giải trí). Sử dụng khoảng trắng một cách hiệu quả có thể giúp đoạn mã dễ đọc hơn nhiều, đặc biệt là trong

trường hợp các toán tử nối tiếp nhau, so sánh hai biểu thức lũy thừa số mũ âm sau:

```
x=10**-2
```

và

```
x = 10 ** -2
```

Nhìn qua cũng có thể thấy cách thứ hai với khoảng trắng dễ đọc hơn nhiều. Hầu hết các hướng dẫn về quy chuẩn Python đều khuyên rằng hãy sử dụng một khoảng trắng xung quanh toán tử nhị phân và không nên dùng khoảng trắng xung quanh toán tử đơn nguyên. Chúng ta sẽ thảo luận thêm về các toán tử Python trong phần “Ngữ nghĩa Python cơ bản: Biến và các đối tượng” trang 15.

Dấu ngoặc đơn để phân nhóm hoặc gọi

Trong đoạn mã sau, ta thấy hai cách sử dụng dấu ngoặc đơn. Đầu tiên, chúng có thể được sử dụng theo cách điển hình để nhóm các câu lệnh hay các toán tử:

```
In [5]: 2 * (3 + 4)
Out [5]: 14
```

Chúng cũng có thể được dùng để chỉ ra một *hàm* đang được gọi. Trong đoạn mã tiếp theo, hàm `print()` dùng để hiển thị nội dung của một biến. Lệnh gọi hàm được biểu thị bằng một cặp dấu ngoặc đơn với các *đối số* cho hàm ở trong đó:

```
In [6]: print('first value:', 1)
first value: 1
In [7]: print('second value:', 2)
second value: 2
```

Một số hàm có thể được gọi mà không cần đến đối số, trong trường hợp đó vẫn phải sử dụng cặp dấu ngoặc đơn để biểu thị một hàm. Một ví dụ cho điều này là phương thức `sort()` của danh sách:

```
In [8]: L = [4,2,3,1]
        L.sort()
        print(L)

        [1, 2, 3, 4]
```

Cặp dấu `( )` sau `sort` biểu thị rằng hàm sẽ được thực thi, và điều đó là bắt buộc dù có đối số hay không.

Lưu ý với hàm print()

Hàm `print()` là một sự thay đổi giữa hai phiên bản Python 2.x và Python 3.x. Trong Python 2, `print` là một câu lệnh, nghĩa là có thể viết:

```
# Chỉ dành cho Python 2!
>> print "first value:", 1
first value: 1
```

Vì một lý do nào đó, những nhà phát triển đã quyết định rằng trong Python 3, `print()` sẽ trở thành một hàm, vì vậy phải sử dụng như sau:

```
# Chỉ dành cho Python 3!
>>> print("first value:", 1)
first value: 1
```

Đây là một trong nhiều cấu trúc không tương thích ngược giữa Python 2 và 3. Trong quá trình viết cuốn sách này, việc bắt gặp những ví dụ được viết trên cả hai phiên bản rất thường xảy ra, và sự hiện diện của câu lệnh `print` thay vì hàm `print()` thường là dấu hiệu đầu tiên cho thấy đó là đoạn mã Python 2.

Kết thúc và tiếp tục học hỏi

Trên đây là những tìm hiểu ngắn gọn về các tính năng thiết yếu của cú pháp Python; Mục đích của nó là cung cấp một bảng tham chiếu tốt khi đọc mã trong các phần sau. Nhiều lần tôi đã đề cập đến các "hướng dẫn về quy chuẩn" Python, thứ có thể giúp các nhóm viết mã theo một kiểu nhất quán. Hướng dẫn về quy chuẩn được sử dụng rộng rãi nhất trong Python được gọi là PEP8 và có thể được tìm thấy tại *https://www.python.org/dev/peps/pep-0008/*. Sẽ rất hữu ích nếu đọc qua nó khi bắt đầu học Python! Các quy chuẩn được đề xuất chứa đựng sự khôn ngoan của nhiều bậc thầy Python và hầu hết các đề xuất này đều vượt qua những phân tích sư phạm: đó là những đề xuất dựa trên kinh nghiệm có thể giúp bạn tránh được những lỗi và sai sót trong mã của mình.

Ngữ nghĩa Python cơ bản: Biến và các đối tượng

Phần này sẽ bắt đầu đề cập đến những ngữ nghĩa cơ bản của ngôn ngữ Python. Trái ngược với *cú pháp* đã được đề cập ở phần trước, *ngữ nghĩa* của một ngôn ngữ liên quan đến ý nghĩa của những câu lệnh. Như cách đã thảo luận về cú pháp, ta cũng sẽ tìm hiểu trước về một số cấu trúc ngữ nghĩa trong Python giúp cho bạn có một khung tham chiếu tốt hơn để hiểu được những đoạn mã trong các phần sau.

Phần này sẽ đề cập đến ngữ nghĩa của *biến* và các *đối tượng*, đó là những cách chính mà ta sẽ dùng để lưu trữ, tham chiếu và vận hành trên dữ liệu trong tập lệnh Python.

Biến Python là con trỏ

Gán biến trong Python rất dễ dàng chỉ bằng cách đặt tên biến bên trái dấu bằng (=):

```
# gán 4 cho biến x
x = 4
```

Điều này có vẻ đơn giản, nhưng nếu bạn hiểu sai về cách toán tử này hoạt động thì cách hoạt động của Python sẽ có vẻ khó hiểu. Chúng ta sẽ nhanh chóng đào sâu vào đó.

Trong nhiều ngôn ngữ lập trình, các biến được coi là những bình chứa để đặt dữ liệu vào. Ví dụ như khi viết vào C

```
// C code
int x = 4;
```

về cơ bản, bạn đang định nghĩa một "bình chứa dữ liệu" có tên x và đặt giá trị là 4 vào đó. Ngược lại, trong Python, các biến được coi là tốt nhất khi làm con trỏ thay vì bình chứa. Vì vậy, trong Python, khi viết

```
x = 4
```

về cơ bản, bạn đang định nghĩa một con trỏ có tên x trỏ đến một số bình chứa khác mang giá trị 4. Lưu ý một hệ quả của điều này: bởi vì các biến Python chỉ trỏ đến các đối tượng khác nhau, nên không cần phải "khai báo" biến, hay thậm chí không cần biến phải luôn trỏ đến cùng một loại thông tin. Đây là lý do mà người

ta gọi Python là một ngôn ngữ động: tên biến có thể trỏ đến bất kỳ loại đối tượng nào. Vì vậy, trong Python, có thể làm những việc như:

```
In [1]: x = 1          # x là số nguyên
        x = 'hello'    # giờ x là chuỗi
        x = [1, 2, 3] # giờ x là danh sách
```

Trong khi người sử dụng có thể phải chỉ rõ loại dữ liệu trong khai báo (type-safety) đối với những ngôn ngữ tĩnh giống C,

```
int x = 4;
```

thì ngôn ngữ động như Python giúp người dùng viết nhanh và dễ đọc.

Có một hệ quả của cách tiếp cận "biến là con trỏ" mà bạn phải nhận thức được. Nếu có hai tên biến cùng chỉ đến một đối tượng có thể thay đổi, thì việc thay đổi biến này cũng thay đổi biến kia! Ví dụ: hãy khởi tạo và sửa đổi danh sách:

```
In [2]: x = [1, 2, 3]
        y = x
```

Ta vừa tạo hai biến là x và y cùng chỉ đến một đối tượng. Bởi vì thế, nếu sửa đổi danh sách thông qua một trong hai tên của nó, thì "danh sách kia" cũng bị sửa đổi theo:

```
In [3]: print(y)
[1, 2, 3]
In [4]: x.append(4)  # thêm 4 vào danh sách được trỏ bởi
x
        print(y)     # danh sách được trỏ bởi y cũng thay
đổi
[1, 2, 3, 4]
```

Việc này hoặc có thể hơi khó hiểu nếu xem biến là các bình chứa dữ liệu. Nhưng nếu nghĩ theo hướng các biến là con trỏ tới các đối tượng, thì việc này lại trở nên hợp lý.

Cũng lưu ý rằng nếu chúng ta sử dụng = để gán giá trị khác cho x , thì điều này sẽ không ảnh hưởng đến giá trị của y. Phép gán đơn giản chỉ là thay đổi đối tượng mà biến trỏ tới.

```
In [5]: x = 'something else'
```

```
        print(y)   # y không thay đổi
[1, 2, 3, 4]
```

Một lần nữa, điều này hoàn toàn hợp lý nếu xem x và y là con trỏ và toán tử = như là một hành động thay đổi đối tượng mà cái tên đó trỏ tới.

Bạn có thể tự hỏi liệu ý tưởng con trỏ này có làm cho các phép toán số học trong Python khó theo dõi hay không, nhưng Python được thiết kế để không gặp vấn đề này. Số, chuỗi, và các kiểu dữ liệu đơn giản là bất biến: ta không thể thay đổi giá trị của chúng, ta chỉ có thể thay đổi những đối tượng mà biến trỏ đến. Ví dụ: hoàn toàn an toàn khi thực hiện phép tính như sau:

```
In [6]: x = 10
        y = x
        x += 5  # thêm 5 vào giá trị của x, và gán lại
vào x
        print("x =", x)
        print("y =", y)
x = 15
y = 10
```

Khi gọi x += 5, ta không sửa đổi giá trị của "đối tượng 5" mà x đang trỏ vào, thay vào đó, ta đã thay đổi đối tượng x trỏ đến. Chính vì lý do này, giá trị của y không bị ảnh hưởng bởi phép tính.

Mọi thứ đều là đối tượng

Python là ngôn ngữ lập trình hướng đối tượng và trong Python mọi thứ đều là đối tượng. Hãy cùng tìm hiểu ý nghĩa của nó là gì. Trước đó ta đã biết biến đơn giản là những con trỏ, và tên biến không hề mang thông tin về kiểu dữ liệu. Điều này dẫn đến việc một số người có tuyên bố sai lầm, rằng Python là một ngôn ngữ không có kiểu dữ liệu. Nhưng thực tế thì không phải như vậy! Xét ví dụ dưới đây:

```
In [7]:  x = 4
         type(x)
Out [7]: int

In [8]:  x = 'hello'
         type(x)
Out [8]: str
```

```
In [9]:  x = 3.14159
         type(x)
Out [9]: float
```

Python vẫn có kiểu dữ liệu; tuy nhiên kiểu dữ liệu trong Python không liên kết với tên biến mà liên kết với chính các đối tượng.

Với những ngôn ngữ lập trình hướng đối tượng như Python, mỗi đối tượng là một thực thể chứa dữ liệu cùng với siêu dữ liệu và\hoặc chức năng liên quan. Trong Python, mọi thứ đều là đối tượng, có nghĩa là mọi thực thể đều có một số siêu dữ liệu (được gọi là thuộc tính) và các chức năng liên quan (được gọi là phương thức). Các thuộc tính và phương thức này được truy cập thông qua cú pháp với dấu chấm.

Ví dụ: trước đó ta thấy rằng danh sách sẽ có phương thức `append`, phương thức này sẽ thêm một mục vào danh sách và được truy cập thông qua cú pháp dấu chấm `(.)`:

```
In [10]: L = [1, 2, 3]
         L.append(100)
         print(L)

[1, 2, 3, 100]
```

Trong khi các đối tượng ghép như danh sách có các thuộc tính và phương thức không còn xa lạ gì, thì điều bất ngờ là trong Python, ngay cả các kiểu đơn giản cũng có các thuộc tính và phương thức kèm theo. Ví dụ, kiểu dữ liệu số sẽ có thuộc tính real và imag mang dữ liệu về phần thực và phần ảo của một giá trị số nếu đó được xem là số phức:

```
In [11]: x = 4.5
         print(x.real, "+", x.imag, 'i')
4.5 + 0.0i
```

Phương thức cũng như thuộc tính, ngoại trừ việc đó là những hàm có thể gọi thông qua một cặp dấu ngoặc đơn. Ví dụ, Số thực dấu phẩy động (floating-point numbers) có phương thức tên là `is_integer` để kiểm tra xem giá trị đó có phải là số nguyên hay không:

```
In [12]:  x = 4.5
          x.is_integer()
```

```
Out [12]: False
In [13]:  x = 4.0
          x.is_integer()
Out [13]: True
```

Khi nói rằng mọi thứ trong Python đều là đối tượng, thì nó thực sự có nghĩa *mọi thứ* đều là đối tượng, kể cả thuộc tính và phương thức của đối tượng cũng chính là đối tượng với các thông tin về "kiểu dữ liệu" (type) của riêng mình:

```
In [14]:  type(x.is_integer)
Out [14]: builtin_function_or_method
```

Ta sẽ thấy rằng lựa chọn theo thiết kế mọi thứ đều là đối tượng của Python cho phép một số cấu trúc ngôn ngữ rất thuận tiện.

Ngữ nghĩa Python cơ bản: Toán tử

Trong phần trước, ta đã bắt đầu xem xét ngữ nghĩa của các biến và đối tượng Python; phần này chúng ta sẽ đi sâu vào ngữ nghĩa của các toán tử khác nhau có trong ngôn ngữ. Đến cuối phần này, bạn sẽ có các công cụ cơ bản để bắt đầu so sánh và tính toán trên dữ liệu trong Python.

Phép tính số học

Python thực hiện bảy toán tử nhị phân cơ bản, hai trong số đó có thể gấp đôi lên để sử dụng như một toán tử đơn nguyên. Tất cả được tóm tắt trong bảng sau:

Toán tử	Tên	Mô tả
a + b	Cộng	Tổng a và b
a – b	Trừ	Hiệu của a và b
a * b	Nhân	Tích của a và b
a / b	Chia hết	Chia a cho b
a // b	Chia lấy phần nguyên	Chia a cho b, bỏ phần thập phân
a % b	Chia lấy phần dư	Lấy phần dư của phép chia a cho b
a ** b	Lũy thừa	Lũy thừa bậc b của a
-a	Số đối	Số đối của a
+a	Cộng đơn	a không thay đổi (hiếm khi sử dụng)

Những toán tử trên có thể được sử dụng và kết hợp theo nhiều cách trực quan, sử dụng dấu ngoặc đơn để nhóm các phép tính. Ví dụ:

```
In [1]:  # cộng, trừ, nhân
         (4 + 8) * (6.5 - 3)
Out [1]: 42.0
```

Phép chia lấy phần nguyên thực chất là phép chia hết với kết quả có phần thập phân được bỏ đi:

```
In [2]: # True division
        print(11 / 2)
5.5
In [3]: # Floor division
        print(11 // 2)
5
```

Toán tử chia lấy phần nguyên được thêm vào Python 3; nên lưu ý rằng nếu làm việc trong Python 2 thì toán tử chia (/) sẽ hoạt động như phép chia lấy phần nguyên đối với kiểu số nguyên và chia hết đối với kiểu số thực dấu phẩy động.

Cuối cùng, tôi sẽ đề cập rằng một toán tử số học thứ tám đã được thêm vào Python 3.5: toán tử `a @ b`, có chức năng chỉ ra *ma trận* của `a` và `b`, dùng để sử dụng trong các gói đại số tuyến tính khác nhau.

Phép toán thao tác bit

Ngoài các phép tính số học, Python còn có các toán tử để thực hiện các phép toán thao tác bit logic trên các số nguyên. Chúng ít được sử dụng hơn nhiều so với các phép tính số học, nhưng nó rất hữu ích khi biết rằng chúng tồn tại. Sáu toán tử thao tác bit được tóm tắt trong bảng sau:

Toán tử	Tên	Mô tả
a & b	AND	Các bit được đinh nghĩa trong cả a và b
a | b	OR	Các bit được đinh nghĩa trong a hoặc b
a ^ b	XOR	Các bit được đinh nghĩa trong a hoặc b nhưng không cùng được định nghĩa trong cả hai
a << b	Dịch trái bit	Dịch các bit của a sang trái b đơn vị
a >> b	Dịch phải bit	Dịch các bit của a sang phải b đơn vị
~a	NOT	Đảo ngược bit của a

Các toán tử thao tác bit này chỉ có ý nghĩa về mặt biểu diễn nhị phân của số học mà ta có thể xem bằng cách sử dụng hàm tích hợp `bin` :

```
In [4]:  bin(10)
Out [4]: '0b1010'
```

Kết quả có tiền tố 0b, là tiền tố biểu thị hệ nhị phân. Các chữ số còn lại chỉ ra rằng số 10 được biểu diễn dưới dạng tổng:

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

Tương tự, có thể viết:

```
In [5]:  bin(4)
Out [5]: '0b100'
```

Giờ hãy sử dụng toán tử OR, ta có thể tìm ra số được tổ hợp bit giữa 4 và 10

```
In [6]:  4 | 10
Out [6]: 14
In [7]:  bin(4 | 10)
Out [7]: '0b1110'
```

Các toán tử thao tác bit này không có hữu ích tức thời như các toán tử số học, nhưng sẽ rất hữu ích nếu hiểu cách thức mà nó hoạt động. Cụ thể, những người sử dụng các ngôn ngữ khác đôi khi sử dụng XOR (a ^ b) trong khi thực sự phải là lũy thừa (a ** b).

Phép gán

Ta đã biết rằng một biến có thể được gán với một giá trị bằng cách dùng toán tử "=". Ví dụ:

```
In [8]: a = 24
        print(a)
24
```

Có thể sử dụng những biến này cùng với những toán tử đã tìm hiểu ở trên để tạo ra các biểu thức. Ví dụ, để cộng thêm 2 vào biến `a` ta làm như sau:

```
In [9]:  a + 2
Out [9]: 26
```

Ta cũng có thể cập nhật cho biến này một giá trị mới; trong trường hợp đó, ta có thể kết hợp phép cộng và phép gán như sau `a = a + 2`. Bởi vì việc kết hợp phép tính số học và phép gán rất thường gặp, Python cũng bao gồm những toán tử tích hợp cho tất cả những phép tính số học:

```
In [10]: a += 2  # tương đương với a = a + 2
         print(a)
26
```

Mỗi một toán tử nhị phân đã tìm hiểu ở trên đều có một toán tử gán tương ứng được tóm tắt như dưới đây:

```
a += b    a -= b      a *= b     a /= b
a //= b   a %= b      a **= b    a &= b
a |= b    a ^= b      a <<= b    a >>= b
```

Mỗi phép tính tương ứng đều có một phép gán theo sau: có thể hiểu rằng với mỗi toán tử # bất kỳ, biểu thức `a #= b` sẽ tương đương với `a = a # b`. Đối với những đối tượng có thể thay đổi như danh sách, mảng hoặc DataFrame, những phép gán tăng cường này thật sự mang lại những khác biệt tinh tế so với cách viết dài dòng kia: chúng sửa đổi những nội dung có sẵn thay vì tạo ra những đối tượng mới để chứa kết quả.

Phép so sánh

Một dạng phép toán rất hữu dụng khác là phép so sánh những giá trị khác nhau. Python cung cấp một bộ những toán tử so sánh trả về giá trị kiểu Boolean là `True` và `False`. Những phép so sánh được liệt kê trong bảng dưới đây:

Toán tử	Mô tả
a == b	a bằng b
a != b	a khác b
a < b	a bé hơn b
a > b	a lớn hơn b
a <= b	a bé hơn hoặc bằng b
a >= b	a lớn hơn hoặc bằng b

Những toán tử này có thể kết hợp với toán tử số học và toán tử thao tác bit để thực hiện hầu như không giới hạn những phép thử số học. Ví dụ: ta có thể kiểm tra một số có phải là số lẻ hay không bằng cách kiểm tra số dư của phép chia cho 2 có bằng 1 không:

```
In [11]:  # 25 là số lẻ
          25 % 2 == 1
Out [11]: True
In [12]:  # 66 là số chẵn
          66 % 2 == 1
Out [12]: False
```

Ta có thể xâu chuỗi nhiều phép so sánh để kiểm tra nhiều quan hệ phức tạp hơn:

```
In [13]:  # kiểm tra a có nằm giữa 15 và 30 hay không
          a = 25
          15 < a < 30
Out [13]: True
```

Hãy thử động não một chút với phép so sánh sau:

```
In [14]:  -1 == ~0
Out [14]: True
```

Toán tử ~ có tác dụng đảo ngược bit, và hiển nhiên là khi đảo bit số 0 ta sẽ được -1. Nếu bạn thắc mắc tại sao lại như vậy, hãy tra cứu sơ đồ mã hóa *bù hai* mà Python sử dụng để mã hóa số nguyên có dấu và thử nghĩ xem chuyện gì sẽ xảy ra nếu đảo bit tất cả số nguyên theo cách này.

Hàm Boolean

Khi làm việc với kiểu dữ liệu Boolean, Python cung cấp những toán tử để kết hợp nhiều giá trị bằng những khái niệm chuẩn của “and”, “or” và “not”. Thật vậy, những toán tử này được biểu diễn bằng những từ `and`, `or` và `not`:

```
In [15]:  x = 4
          (x < 6) and (x > 2)
Out [15]: True
In [16]:  (x > 10) or (x % 2 == 0)
Out [16]: True
In [17]:  not (x < 6)
```

```
Out [17]: False
```

Những người am hiểu về đại số Boolean có thể nhận thấy toán tử XOR không xuất hiện; tất nhiên, nó có thể được xây dựng theo nhiều cách bằng những câu lệnh tổ hợp của các toán tử khác. Mặt khác, dưới đây là một mẹo nhỏ để có thể sử dụng XOR của giá trị Boolean:

```
In [18]:  # (x > 1) xor (x < 10)
          (x > 1) != (x < 10)
Out [18]: False
```

Hàm Boolean sẽ trở nên cực kỳ hữu dụng khi ta bắt đầu tìm hiểu về những *câu lệnh điều khiển* như câu lệnh điều kiện và câu lệnh lặp.

Một điều đôi lúc gây khó hiểu là khi nào thì sử dụng những toán tử Boolean (`and, or, not`), và khi nào thì sử dụng câu toán tử thao tác bit `(&, |, ~)`. Câu trả lời nằm ngay trong tên gọi của chúng: toán tử Boolean được sử dụng khi muốn xác định các giá trị Boolean (đúng/sai) của toàn bộ câu lệnh. Phép toán thao tác bit được sử dụng khi muốn làm việc trên những bit đơn lẻ hoặc những thành phần của các đối tượng trong một vấn đề.

Toán tử định danh và toán tử thành viên

Giống như `and`, `or` và `not`, Python cũng có những toán tử dạng văn xuôi để kiểm tra danh tính và thành viên được tóm tắt trong bảng dưới đây:

Toán tử	Mô tả
a is b	Đúng nếu a và b là hai đối tượng giống hệt nhau
a is not b	Đúng nếu a và b là hai đối tượng khác nhau
a in b	Đúng nếu a là một phần của b
a not in b	Đúng nếu a không phải một phần của b

Toán tử định danh: is và is not

Toán tử định danh `is` và `is not` dùng để kiểm tra *danh tính của đối tượng.* Danh tính của đối tượng khác với tính ngang bằng, ta có thể thấy qua ví dụ dưới đây:

```
In [19]:  a = [1, 2, 3]
```

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

In [20]:  a == b
Out [20]: True

In [21]:  a is b
Out [21]: False

In [22]:  a is not b
Out [22]: True
```

Thế nào là đối tượng giống hệt nhau? Xem ví dụ sau:

```
In [23]:  a = [1, 2, 3]
          b = a
          a is b
Out [23]: True
```

Sự khác nhau giữa hai trường hợp trên là trong trường hợp thứ nhất, a và b trỏ đến hai *đối tượng khác nhau,* trong khi ở trường hợp thứ hai chúng lại trỏ đến *cùng một đối tượng.* Như ta đã tìm hiểu ở những phần trước, biến trong Python là con trỏ. Toán tử `is` kiểm tra liệu hai biến có trỏ đến cùng một đối tượng không, thay vì kiểm tra xem những đối tượng đó chứa giá trị nào. Với suy nghĩ này, trong nhiều trường hợp, người mới thường sử dụng `is`, nhưng thực sự ý của họ là ==.

Toán tử thành viên

Toán tử thành viên kiểm tra quan hệ trong các đối tượng tổ hợp. Ví dụ:

```
In [24]:  1 in [1, 2, 3]
Out [24]: True
In [25]:  2 not in [1, 2, 3]
Out [25]: False
```

Những toán tử thành viên này là một ví dụ của việc Python dễ sử dụng hơn so với các ngôn ngữ bậc thấp hơn như C. Trong C, quan hệ thường được xác định bằng cách xây dựng thủ công một vòng lặp qua toàn bộ danh sách và kiểm tra tính ngang bằng của từng giá trị. Với Python, đơn giản chỉ cần gõ ra những điều muốn biết như cách viết văn xuôi trong tiếng Anh.

Kiểu dữ liệu tích hợp: Những dữ liệu đơn giản

Khi tìm hiểu về biến và đối tượng trong Python, chúng ta đã nhắc đến việc mọi đối tượng trong Python đều có thông tin về kiểu dữ liệu kèm theo. Bây giờ chúng ta sẽ tóm tắt qua về những kiểu dữ liệu đơn giản được tích hợp sẵn do Python cung cấp. Ta nói "kiểu dữ liệu đơn giản" để phân biệt với những kiểu dữ liệu tổ hợp mà ta sẽ tìm hiểu trong phần sau.

Kiểu dữ liệu đơn giản của Python được tóm tắt trong Bảng 1-1.

Bảng 1-1. Những kiểu dữ liệu vô hướng của Python

Kiểu dữ liệu	Ví dụ	Mô tả
int	x = 1	Số nguyên
float	x = 1.0	Số thực dấu phẩy động
complex	x = 1 + 2j	Số phức (số có phần thực và phần ảo)
bool	x = True	Boolean: Đúng hoặc Sai
str	x = 'abc'	Chuỗi: ký tự và chữ số
NoneType	x = None	Biểu diễn đối tượng không có giá trị

Ta sẽ điểm qua một lượt các kiểu dữ liệu này.

Số nguyên

Kiểu dữ liệu số cơ bản nhất là số nguyên. Mọi chữ số không có dấu thập phân đều là số nguyên:

```
In [1]:  x = 1
         type(x)
Out [1]: int
```

Số nguyên trong Python thực sự có một chút khác biệt so với số nguyên trong những ngôn ngữ như C. Số nguyên trong C có độ chính xác cố định và thường xuyên bị tràn ở một số giá trị (thường là gần 2^{31} hoặc 2^{63}, phụ thuộc vào hệ thống của người dùng). Số nguyên trong Python có độ chính xác tương đối, do vậy nó có thể thực hiện được những tính toán gây tràn ở những ngôn ngữ khác:

```
In [2]:  2 ** 200
Out [2]:
160693804425899027554196209234116260252220299378279283530
1376
```

Một tính năng tiện dụng khác của số nguyên trong Python là mặc định chuyển kết quả phép chia thành kiểu số thực dấu phẩy động:

```
In [3]:  5 / 2
Out [3]: 2.5
```

Lưu ý đây là một tính năng của Python 3, trong Python 2, cũng như nhiều ngôn ngữ tĩnh khác như C, phép chia số nguyên sẽ bỏ đi toàn bộ phần thập phân và trả về số nguyên"

```
# Đặc tính của Python 2
>>> 5 / 2
2
```

Trong Python 3, có thể sử dụng toán tử chia lấy phần nguyên để thực hiện điều này:

```
In [4]:  5 // 2
Out [4]: 2
```

Cuối cùng, lưu ý rằng mặc dù Python 2.x có cả hai kiểu dữ liệu là `int` và `long` những Python 3 đã kết hợp đặc tính của hai kiểu dữ liệu này thành một kiểu dữ liệu duy nhất là `int`.

Số thực dấu phẩy động

Số thực dấu phẩy động có thể lưu trữ phân số. Chúng có thể được biểu diễn dưới dạng thập phân hoặc dạng số mũ:

```
In [5]: x = 0.000005
        y = 5e-6
        print(x == y)
True

In [6]: x = 1400000.00
        y = 1.4e6
        print(x == y)
True
```

Dưới dạng số mũ, ký kiệu e hay E có thể đọc là "...nhân mười mũ...", như vậy 1.4e6 được hiểu là 1.4 x 10^6.

Một số nguyên có thể chuyển đổi thành số thực với hàm tạo đối tượng `float`:

```
In [7]:  float(1)
Out [7]: 1.0
```

Độ chính xác của số thực dấu phẩy động

Một điều cần phải chú ý về số thực dấu phẩy động đó là độ chính xác của nó có giới hạn, dẫn đến việc kiểm tra sự ngang bằng cho ra kết quả không ổn định. Ví dụ:

```
In [8]:  0.1 + 0.2 == 0.3
Out [8]: False
```

Tại sao lại có trường hợp này? Hóa ra đó không phải là đặc tính của Python mà do định dạng chính xác cố định của bộ lưu trữ dấu phẩy động nhị phân được hầu hết - nếu không nói là tất cả, các nền tảng máy tính khoa học sử dụng. Mọi ngôn ngữ lập trình sử dụng số thực dấu phẩy động đều lưu trữ chúng với một số lượng bit cố định, và điều này dẫn đến một số chữ số chỉ được biểu diễn gần đúng. Ta có thể thấy điều này bằng cách in ba giá trị với độ chính xác cao:

```
In [9]: print("0.1 = {0:.17f}".format(0.1))
        print("0.2 = {0:.17f}".format(0.2))
        print("0.3 = {0:.17f}".format(0.3))
0.1 = 0.10000000000000001
0.2 = 0.20000000000000001
0.3 = 0.29999999999999999
```

Ta thường quen với ký hiệu số trong hệ thập phân (cơ số 10), nên mỗi phân số phải được biểu thị dưới dạng tổng các lũy thừa của 10:

$$1/8 = 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 5 \cdot 10^{-3}$$

Với cách biểu thị cơ số 10 quen thuộc, ta biểu thị số này dưới dạng thập phân thông thường là 0.125.

Máy tính thường lưu trữ giá trị bằng ký hiệu nhị phân, vì vậy mỗi số được biểu thị dưới dạng tổng các lũy thừa của 2:

$$1/8 = 0 . 2^{-1} + 0 . 2^{-2} + 1 . 2^{-3}$$

Với cách biểu diễn cơ số 2, ta có thể viết là 0.001_2, trong đó số 2 biểu thị ký hiệu nhị phân. Giá trị $0.125 = 0.001_2$ cũng đồng thời là một số mà cả ký hiệu nhị phân và thập phân có thể biểu thị bằng một số chữ số hữu hạn.

Với cách biểu thị cơ số 10 quen thuộc, có thể bạn đã quen với việc không thể biểu thị một số bằng một số chữ số hữu hạn. Ví dụ, chia 1 cho 3 trong hệ thập phân:

$$1/3 = 0.33333333333...$$

Dãy số 3 sẽ tiếp tục kéo dài mãi, có nghĩa là muốn biểu diễn thương số trên đúng nghĩa thì phải cần vô hạn chữ số!

Tương tự, có những số biểu diễn dưới dạng nhị phân cũng cần vô hạn chữ số. Ví dụ:

$$1/10 = 0.00011001100110011..._2$$

Giống như ký hiệu thập phân cần vô hạn chữ số để hoàn toàn biểu thị được phép tính 1/3 thì ký hiệu nhị phân cũng yêu cầu vô hạn chữ số để biểu thị phép tính 1/10. Python cắt ngắn việc biểu diễn này ở mức 52 bit ngoài bit khác không đầu tiên trên hầu hết các hệ thống.

Lỗi làm tròn với các giá trị dấu phẩy động là sai số không thể tránh khỏi khi làm việc với số thực dấu phẩy động. Cách tốt nhất để đối phó với điều này là luôn nhớ rằng số thực dấu phẩy động chỉ gần đúng, và không bao giờ phụ thuộc vào những phép thử ngang bằng với các giá trị dấu phẩy động.

Số phức

Số phức là số với phần thực và phần ảo. Ta đã từng thấy số nguyên và số thực trước đó và có thể sử dụng chúng để cấu thành một số phức:

```
In [10]:  complex(1, 2)
Out [10]: (1+2j)
```

Hoặc ta có thể sử dụng hậu tố j trong biểu thức để biểu thị phần ảo:

```
In [11]:  1 + 2j
Out [11]: (1+2j)
```

Số phức có một loạt các thuộc tính và phương thức thú vị mà ta có thể trình bày ngắn gọn như sau:

```
In [12]:  c = 3 + 4j
In [13]:  c.real  # phần thực
Out [13]: 3.0
In [14]:  c.imag  # phần ảo
Out [14]: 4.0
In [15]:  c.conjugate()  # số phức liên hợp
Out [15]: (3-4j)
In [16]: abs(c)  # Mô đun của số phức--nghĩa là,
sqrt(c.real ** 2 + c.imag ** 2)
Out [16]: 5.0
```

Kiểu dữ liệu chuỗi

Chuỗi trong Python được tạo ra bởi cặp dấu ngoặc đơn hoặc ngoặc kép:

```
In [17]: message = "what do you like?"
         response = 'spam'
```

Python có rất nhiều hàm và phương thức cực kỳ hữu dụng để thao tác với chuỗi:

```
In [18]:  # độ dài chuỗi
          len(response)
Out [18]: 4

In [19]:  # Viết hoa chuỗi. Xem thêm str.lower()
          response.upper()
Out [19]: 'SPAM'

In [20]:  # Viết hoa ký tự đầu tiên của chuỗi. Xem thêm
str.title()
          message.capitalize()
Out [20]: 'What do you like?'

In [21]:  # nối chuỗi với dấu +
          message + response
Out [21]: 'what do you like?spam'
```

```
In [22]:  # nối chuỗi nhiều lần
          5 * response
Out [22]: 'spamspamspamspamspam'

In [23]:  # Truy xuất ký tự riêng lẻ (chỉ mục bắt đầu từ
          0)
          message[0]
Out [23]: 'w'
```

Để tìm hiểu thêm về đánh chỉ mục trong Python, hãy xem phần “Danh sách” trang 33.

Kiểu dữ liệu None

Python có một kiểu dữ liệu đặc biệt là `NoneType`, kiểu dữ liệu này chỉ có một giá trị duy nhất: `None`. Ví dụ:

```
In [24]:  type(None)
Out [24]: NoneType
```

Kiểu None được sử dụng ở rất nhiều nơi, nhưng có lẽ thông dụng nhất là sử dụng để làm giá trị trả về mặc định của hàm. Ví dụ, hàm `print()` trong Python 3 không trả về bất cứ thứ gì, nhưng ta vẫn có thể lấy được giá trị:

```
In [25]: return_value = print('abc')
         abc
In [26]: print(return_value)
         None
```

Tương tự như vậy, bất kỳ hàm nào trong Python không trả về giá trị nào thì trong thực tế sẽ trả về None.

Kiểu Boolean

Kiểu dữ liệu Boolean là kiểu dữ liệu đơn giản với hai giá trị khả dĩ: True và False, được trả về bằng các toán tử so sánh đã tìm hiểu trước đó:

```
In [27]:  result = (4 < 5)
          result
Out [27]: True
In [28]:  type(result)
```

```
Out [28]: bool
```

Lưu ý rằng giá trị Boolean có phân biệt chữ hoa và chữ thường: không giống như những ngôn ngữ khác, True và False phải viết hoa chữ cái đầu!

```
In [29]: print(True, False)
True False
```

Boolean cũng có thể cấu thành bằng cách dùng hàm tạo đối tượng `bool()`: giá trị thuộc bất kỳ kiểu dữ liệu nào cũng có thể chuyển đổi thành Boolean thông qua các quy tắc có thể dự đoán được. Ví dụ, bất kỳ kiểu số học nào đều là False nếu bằng 0, ngược lại là True:

```
In [30]:  bool(2014)
Out [30]: True
In [31]:  bool(0)
Out [31]: False
In [32]:  bool(3.1415)
Out [32]: True
```

Việc chuyển đổi Boolean của None luôn là False:

```
In [33]:  bool(None)
Out [33]: False
```

Đối với chuỗi, kết quả hàm `bool(s)` luôn là False đối với chuỗi trống và True với những chuỗi khác:

```
In [34]:  bool("")
Out [34]: False
In [35]:  bool("abc")
Out [35]: True
```

Đối với những tổ hợp mà ta sẽ tìm hiểu trong phần sau, việc biểu diễn dạng Boolean cho tổ hợp trống là False và với những tổ hợp còn lại là True:

```
In [36]:  bool([1, 2, 3])
Out [36]: True
In [37]:  bool([])
Out [37]: False
```

Cấu trúc dữ liệu tích hợp

Ta đã tìm hiểu một số kiểu dữ liệu đơn giản: `int`, `float`, `complex`, `bool`, `str`, v.v.. Python cũng có một số kiểu dữ liệu hỗn hợp được tích hợp sẵn, hoạt động như những bình chứa cho các kiểu dữ liệu khác. Những kiểu hỗn hợp đó là:

Tên kiểu	Ví dụ	Mô tả
list	[1, 2, 3]	Tổ hợp có thứ tự
tuple	(1, 2, 3)	Tổ hợp bất biến có thứ tự
dict	{'a':1, 'b':2, 'c':3}	Tổ hợp không có thứ tự các cặp khóa, giá trị
set	{1, 2, 3}	Tổ hợp không có thứ tự các giá trị duy nhất

Các dấu ngoặc đơn, vuông, nhọn đều mang ý nghĩa riêng biệt, biểu thị loại tổ hợp nào sẽ được tạo ra. Ta sẽ cùng điểm qua những cấu trúc dữ liệu trên:

Danh sách (list)

Danh sách là kiểu dữ liệu tổ hợp *có thứ tự* và *có thể thay đổi* cơ bản trong Python. Nó có thể được định nghĩa bằng cách đặt những giá trị được ngăn cách bằng dấu phẩy vào giữa cặp dấu ngoặc vuông, dưới đây là danh sách những số nguyên tố:

```
In [1]: L = [2, 3, 5, 7]
```

Danh sách có một số tính chất và phương thức hữu ích. Một số trường hợp phổ biến và hữu ích hơn cả.

```
In [2]:  # Độ dài của danh sách
         len(L)
Out [2]: 4
In [3]:  # Nối giá trị vào cuối danh sách
         L.append(11)
         L
Out [3]: [2, 3, 5, 7, 11]
In [4]:  # Thêm danh sách bổ sung
         L + [13, 17, 19]
Out [4]: [2, 3, 5, 7, 11, 13, 17, 19]
In [5]:  # sort() phương thức sắp xếp
         L = [2, 5, 1, 6, 3, 4]
         L.sort()
         L
Out [5]: [1, 2, 3, 4, 5, 6]
```

Ngoài ra, còn rất nhiều phương thức tích hợp sẵn dành cho danh sách và đều có đầy đủ trong tài liệu Python trực tuyến.

Mặc dù các giá trị của danh sách được đưa ra trong những ví dụ ở trên đều cùng một kiểu dữ liệu, thì một trong những tính năng mạnh mẽ của đối tượng hỗn hợp trong Python là chúng có thể chứa được những đối tượng thuộc *bất kì* kiểu dữ liệu nào. Ví dụ:

```
In [6]: L = [1, 'two', 3.14, [0, 3, 5]]
```

Tính linh hoạt này chính là hệ quả của hệ thống ngôn ngữ động. Phải rất đau đầu mới có thể tạo ra một hỗn hợp như vậy trong những ngôn ngữ động như C. Ta có thể thấy danh sách thậm chí còn có thể chứa một danh sách khác như là một phần tử. Kiểu dữ liệu linh hoạt như vậy là một trong những đặc tính thiết yếu giúp việc lập trình với Python nhanh chóng và dễ dàng hơn nhiều.

Ta đã cùng tìm hiểu về các thao tác của danh sách nói chung; một phần quan trọng khác đó là truy xuất vào các phần tử bên trong nó. Trong Python, những điều này được thực hiện thông qua *lập chỉ mục* và *trích xuất* mà chúng ta sẽ tìm hiểu ngay sau đây.

Lập chỉ mục và trích xuất danh sách

Python cung cấp quyền truy cập vào các phần tử trong dữ liệu hỗn hợp thông qua việc *lập chỉ mục* cho các phần tử đơn và *chia nhỏ* nhiều phần tử. Chúng ta sẽ biết được rằng cả hai đều được biểu thị bằng cú pháp dấu ngoặc vuông. Quay trở lại với danh sách số những nguyên tố trước đó:

```
In [7]: L = [2, 3, 5, 7, 11]
```

Python sử dụng *chỉ mục bắt đầu từ số 0 (zero-based)*, vì vậy ta có thể truy xuất phần tử thứ nhất và thứ hai bằng cách sử dụng cú pháp sau:

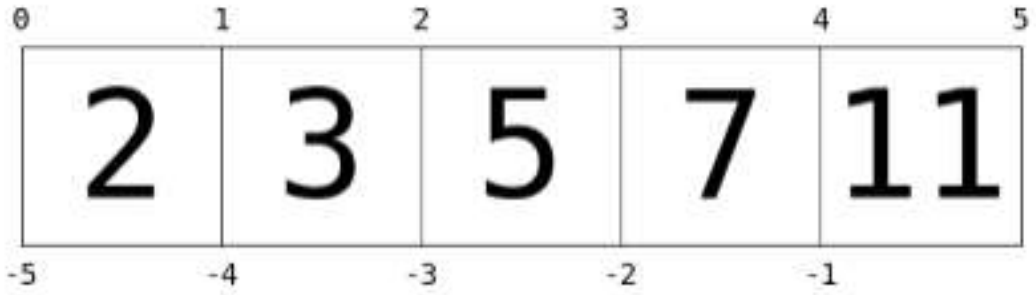
```
In [8]:  L[0]
Out [8]: 2
In [9]:  L[1]
Out [9]: 3
```

Phần tử cuối cùng của danh sách có thể truy xuất bằng số âm, bắt đầu từ -1:

```
In [10]:  L[-1]
Out [10]: 11
```

```
In [12]:  L[-2]
Out [12]: 7
```

Có thể hình dung sơ đồ chỉ mục như sau:



Các giá trị trong danh sách được thể hiện bằng số lớn trong các ô vuông; chỉ mục được thể hiện bằng những số nhỏ ở trên và dưới. Trong trường hợp này, `L[2]` trả về 5, vì đó là giá trị liền kề tại chỉ mục 2.

Trong khi *lập chỉ mục* là phương tiện để tìm ra một giá trị từ danh sách, thì *chia nhỏ* là phương tiện truy xuất nhiều giá trị với danh sách phụ. Sử dụng dấu hai chấm để đánh dấu điểm bắt đầu và kết thúc (không bao gồm điểm kết thúc) của danh sách phụ. Ví dụ, để lấy ba phần tử đầu tiên của danh sách, ta có thể làm như sau:

```
In [12]:  L[0:3]
Out [12]: [2, 3, 5]
```

Lưu ý vị trí của 0 và 3 trong sơ đồ trước đó và cách mà thao tác chia nhỏ chỉ lấy các giá trị giữa các chỉ số. Giả sử chúng ta bỏ qua chỉ mục đầu tiên là 0, ta có thể làm như sau:

```
In [13]:  L[:3]
Out [13]: [2, 3, 5]
```

Tương tự, nếu ta bỏ qua chỉ số cuối, nó sẽ được mặc định là độ dài của danh sách. Do đó, có thể truy xuất ba phần tử cuối bằng cách:

```
In [14]:  L[-3:]
Out [14]: [5, 7, 11]
```

Cuối cùng, có thể chỉ định một số nguyên thứ 3 để biểu thị cho độ rộng của bước nhảy, ví dụ, để chọn những phần tử cách nhau hai vị trí ta làm như sau:

```
In [15]:  L[::2]  # tương đương với L[0:len(L):2]
Out [15]: [2, 5, 11]
```

Một phiên bản đặc biệt hữu ích của việc này là chỉ định một bước nhảy âm, nó sẽ đảo ngược danh sách lại:

```
In [16]:  L[::-1]
Out [16]: [11, 7, 5, 3, 2]
```

Cả thao tác đánh chỉ mục cũng như chia nhỏ có thể được sử dụng để chỉnh sửa những phần tử cũng như truy xuất chúng. Cú pháp:

```
In [17]: L[0] = 100
         print(L)
[100, 3, 5, 7, 11]

In [18]: L[1:3] = [55, 56]
         print(L)
[100, 55, 56, 7, 11]
```

Một cú pháp chia nhỏ tương tự cũng thường được sử dụng trong nhiều gói định hướng khoa học dữ liệu, bao gồm NumPy và Pandas (đã được nhắc đến ở phần mở đầu)

Chúng ta vừa tìm hiểu về danh sách trong Python và cách truy xuất các phần tử trong kiểu dữ liệu hỗn hợp có thứ tự, bây giờ hãy cũng đến với ba kiểu dữ liệu hỗn hợp còn lại mà ta đã đề cập trước đó.

Bộ dữ liệu (Tuple)[1]

Bộ dữ liệu khá tương đồng với danh sách, nhưng được định nghĩa bằng cặp dấu ngoặc đơn thay vì ngoặc vuông:

```
In [19]: t = (1, 2, 3)
```

Chúng cũng có thể được định nghĩa mà không cần bất kỳ dấu ngoặc nào:

```
In [20]: t = 1, 2, 3
         print(t)
```

[1] tuple và set đều có nghĩa là bộ, nên từ đây sẽ dùng thuật ngữ "bộ dữ liệu" cho tuple và "bộ" cho set

```
1, 2, 3)
```

Giống như danh sách, bộ dữ liệu cũng có độ dài và những phần tử có thể được trích xuất bằng lập chỉ mục với ngoặc vuông:

```
In [21]:  len(t)
Out [21]: 3
In [22]:  t[0]
Out [22]: 1
```

Tính năng chính để phân biệt bộ dữ liệu đó là *sự bất biến*: có nghĩa là một khi được tạo ra, kích thước và nội dung bên trong nó không thể bị thay đổi:

```
In [23]: t[1] = 4
---------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-23-141c76cb54a2> in <module>()
----> 1 t[1] = 4

TypeError: 'tuple' object does not support item
assignment

In [24]: t.append(4)
---------------------------------------------------------
AttributeError            Traceback (most recent call last)
<ipython-input-24-e8bd1632f9dd> in <module>()
----> 1 t.append(4)


AttributeError: 'tuple' object has no attribute 'append'
```

Bộ dữ liệu thường được sử dụng trong chương trình Python; một trường hợp phổ biến đó là trong các hàm có nhiều giá trị trả về. Ví dụ, phương thức `as_integer_ratio()` của đối tượng dấu phẩy động trả về một tử số và mẫu số; cặp giá trị này trả về dưới dạng một bộ dữ liệu:

```
n [25]:  x = 0.125
         x.as_integer_ratio()
Out [25]: (1, 8)
```

Những giá trị trả về này có thể được chỉ định một cách riêng biệt như sau:

```
In [26]: numerator, denominator = x.as_integer_ratio()
         print(numerator / denominator)
0.125
```

Những lý luận về việc lập chỉ mục và chia nhỏ đã tìm hiểu trước đó cũng hoạt động với bộ dữ liệu, cùng với một nhóm các phương thức khác. Tìm đọc Tài liệu về Cấu trúc dữ liệu để có một danh sách các phương thức đầy đủ hơn.

Từ điển (Dictionary)

Từ điển là tổ hợp cực kì linh hoạt thể hiện tương quan giữa các cặp khóa (key) và giá trị (value), và cấu thành nền tảng của nhiều hoạt động trong Python. Chúng được tạo thông qua danh sách các cặp khóa : giá trị `(key : value)` được phân tách bằng dấu phẩy đặt trong cặp dấu ngoặc nhọn:

```
In [27]: numbers = {'one':1, 'two':2, 'three':3}
```

Các mục được truy xuất và chỉnh sửa thông qua cú pháp lập chỉ mục giống như danh sách và bộ dữ liệu, ngoại trừ việc chỉ mục không bắt đầu từ số 0 mà chính là khóa hợp lệ trong nhật ký:

```
In [28]:  # Truy xuất dữ liệu thông qua khóa
          numbers['two']
Out [28]: 2
```

Các mục mới có thể thêm vào từ điển bằng cách sử dụng chỉ mục:

```
In [29]: # Thêm một cặp khóa/giá trị
         numbers['ninety'] = 90
         print(numbers)
{'three': 3, 'ninety': 90, 'two': 2, 'one': 1}
```

Lưu ý rằng từ điển không duy trì bất cứ trật tự có nghĩa nào cho các thông số đầu vào mà ta tự quyết định trật tự đó. Việc không có thứ tự cho phép từ điển có thể triển khai rất hiệu quả, do đó việc truy xuất các phần tử ngẫu nhiên rất nhanh chóng, bất kể kích thước của từ điển (nếu hứng thú về cách thức hoạt động của nó, hãy đọc về khái niệm *bảng băm (hash table)*). Tài liệu Python có một danh sách đầy đủ về các phương thức của từ điển.

Bộ (Set)

Tổ hợp cơ bản thứ tư là bộ, trong đó chứa tổ hợp của những mục độc nhất và không có thứ tự. Chúng định nghĩa rất giống danh sách và bộ dữ liệu, ngoại trừ việc nó dùng cặp dấu ngoặc nhọn như từ điển:

```
In [30]: primes = {2, 3, 5, 7}
         odds = {1, 3, 5, 7, 9}
```

Nếu bạn quen với tập hợp trong toán học, bạn cũng sẽ quen với các phép hợp, giao, hiệu, hiệu đối xứng và các phép toán khác. Bộ trong Python cũng được tích hợp sẵn các phép toán đó thông qua các phương thức hoặc toán tử. Đối với mỗi phép toán, chúng tôi sẽ đưa ra ví dụ cho cả hai cách:

```
In [31]:  # hợp: phần tử xuất hiện ở một trong hai bộ
          primes | odds       # toán tử
          primes.union(odds) # phương thức tương đương
Out [31]: {1, 2, 3, 5, 7, 9}

In [32]:  # giao: phần tử xuất hiện trong cả hai bộ
          primes & odds                # toán tử
          primes.intersection(odds) # phương thức tương
đương

Out [32]: {3, 5, 7}
In [33]:  # hiệu: phần tử thuộc bộ primes nhưng không
thuộc bộ odds
          primes - odds            # toán tử
          primes.difference(odds) # phương thức tương
đương

Out [33]: {2}
In [34]:  # hiệu đối xứng: phần tử chỉ xuất hiện ở một
trong hai bộ
          primes ^ odds                       # toán tử
          primes.symmetric_difference(odds) # phương
thức tương đương
Out [34]: {1, 2, 9}
```

Còn nhiều phép toán và phương thức dành cho bộ. Hãy tham khảo tài liệu Python trực tuyến để có được danh sách hoàn chỉnh.

Cấu trúc dữ liệu chuyên biệt

Python còn có một số cấu trúc dữ liệu khác có thể có ích cho bạn; chúng có thể được tìm thấy trong mô-đun tích hợp `collections`. Mô-đun `collections` được ghi chép đầy đủ trong tài liệu Python trực tuyến, và bạn cũng có thể đọc thêm về nhiều đối tượng khác nhau có trong đó.

Cụ thể, tôi đã có dịp tìm thấy những điều rất hữu ích dưới đây:

`collections.namedtuple`

: Giống như bộ dữ liệu (tuple) nhưng mỗi giá trị đều có tên

`collections.defaultdict`

: Giống như từ điển nhưng những khóa không xác định sẽ mang giá trị mặc định do người dùng chỉ định.

`collections.OrderedDict`

: Giống như từ điển nhưng thứ tự các khóa được duy trì

Một khi bạn đã biết về các kiểu dữ liệu tổ hợp chuẩn được tích hợp sẵn, việc sử dụng những chức năng mở rộng này rất trực quan, và tôi khuyến khích việc đọc về cách sử dụng của chúng.

Luồng điều khiển (Control Flow)

Trong lập trình, *luồng điều khiển (Control Flow)* là thứ giúp cho các chương trình thực sự hoạt động. Không có nó, một chương trình chỉ đơn giản là một chuỗi các câu lệnh được thực hiện một cách tuần tự. Với luồng điều khiển, bạn có thể thực thi một số khối mã nhất định một cách có điều kiện và/hoặc lặp đi lặp lại: những khối mã cơ bản này có thể được kết hợp với nhau để tạo ra các chương trình tinh vi!

Ta sẽ tìm hiểu khái quát về các câu lệnh điều kiện (bao gồm `if`, `elif` và `else`) và các câu lệnh lặp (bao gồm `for` và `while`, cùng với đó là `break`, `continue`, và `pass`)

Câu lệnh điều kiện: if, elif và else

Câu lệnh điều kiện, thường được gọi là câu lệnh *if-then*, cho phép lập trình viên thực thi các đoạn mã nhất định phụ thuộc vào một số điều kiện Boolean. Dưới đây là một ví dụ cơ bản của câu lệnh điều kiện trong Python:

```
In [1]: x = -15
        if x == 0:
           print(x, "is zero")
        elif x > 0:
           print(x, "is positive")
        elif x < 0:
           print(x, "is negative")
        else:
           print(x, "is unlike anything I've ever
seen...")
-15 is negative
```

Đặc biệt lưu ý việc sử dụng dấu hai chấm `(:)` và khoảng trắng để biểu thị các khối mã riêng biệt.

Python cũng sử dụng `if` và `else` như trong các ngôn ngữ thường thấy khác; một từ khóa riêng biệt của nó là `elif`, viết gọn của "else if". Trong các mệnh đề điều kiện, `elif` và `else` là các khối mã tùy chọn; ngoài ra, có thể tùy biến số lượng câu lệnh `elif` như mong muốn.

Vòng lặp for

Vòng lặp trong Python là cách để lặp lại việc thực thi một số câu lệnh. Ví dụ, nếu muốn in ra từng mục trong một danh sách, ta có thể sử dụng vòng lặp `for`:

```
In [2]: for N in [2, 3, 5, 7]:
            print(N, end=' ') # in tất cả lên cùng một
dòng

2 3 5 7
```

Hãy để ý đến sự đơn giản của vòng lặp `for`: ta tự chỉ định biến mà ta muốn sử dụng, tập hợp mà ta muốn chạy vòng lặp trên đó, và sử dụng toán tử `in` để liên kết chúng lại một cách trực quan và dễ đọc. Chính xác hơn, đối tượng bên phải toán thử `in` có thể là bất cứ trình lặp (iterator) nào trong Python.

Một trình lặp có thể được hiểu như là một tập suy rộng, và ta sẽ tìm hiểu về chúng trong phần "Trình lặp" ở trang 57.

Ví dụ, một trong những trình lặp thường gặp nhất trong Python là `range`, đối tượng sẽ tạo ra một tập hợp các số:

```
In [3]: for i in range(10):
            print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
```

Lưu ý rằng phạm vi mặc định của đối tượng bắt đầu tại 0, và theo quy ước, điểm cuối của phạm vi đó không được bao gồm trong đầu ra. Đối tượng phạm vi cũng có thể là những giá trị phức tạp hơn:

```
In [4]:  # khoảng từ 5 đến 10
         list(range(5, 10))
Out [4]: [5, 6, 7, 8, 9]

In [5]:  # khoảng từ 0 đến 10 với bước nhảy là 2
         list(range(0, 10, 2))
Out [5]: [0, 2, 4, 6, 8]
```

Bạn có thể để ý thấy ý nghĩa của đối số `range` rất giống với cú pháp chia nhỏ mà ta đã tìm hiểu trong phần "Danh sách" trang 33.

Lưu ý rằng tác dụng của `range()` là một trong những khác biệt giữa Python 2 và Python 3: Trong Python 2, `range()` tạo ra một danh sách, trong khi với Python 3, `range()` tạo ra một đối tượng lặp.

Vòng lặp while

Một kiểu vòng lặp khác trong Python là vòng lặp while, nó sẽ tiến hành lặp cho đến khi thỏa mãn một số điều kiện:

```
In [6]: i = 0
        while i < 10:
            print(i, end=' ')
            i += 1
0 1 2 3 4 5 6 7 8 9
```

Đối số của vòng lặp `while` được xét như một câu lệnh Boolean, và vòng lặp sẽ thực thi cho đến khi câu lệnh mang giá trị là False.

Tinh chỉnh vòng lặp với break và continue

Có hai câu lệnh hữu ích được sử dụng trong vòng lặp để tinh chỉnh cách mà chúng thực thi:

- Câu lệnh `break` để thoát hoàn toàn khỏi vòng lặp
- Câu lệnh `continue` để bỏ qua phần còn lại của lần lặp hiện tại và tiến đến lần lặp tiếp theo.

Chúng có thể sử dụng trong cả vòng lặp `for` và `while`.

Dưới đây là một ví dụ sử dụng `continue` để in ra một chuỗi số lẻ. Trong trường hợp này, chỉ cần câu lệnh `if-else` là đã có thể cho ra kết quả, nhưng đôi lúc câu lệnh continue có thể là một cách thuận tiện hơn để diễn tả ý tưởng:

```
In [7]: for n in range(20):
            # kiểm tra xem n có chẵn không
            if n % 2 == 0:
                continue
            print(n, end=' ')
1 3 5 7 9 11 13 15 17 19
```

Dưới đây là một ví dụ về câu lệnh `break` được sử dụng cho một tác vụ phức tạp hơn. Vòng lặp này sẽ in ra tất cả chữ số trong dãy Fibonacci cho đến một giá trị nhất định:

```
In [8]: a, b = 0, 1
        amax = 100
        L = []

        while True:
            (a, b) = (b, a + b)
            if a > amax:
                break
            L.append(a)

        print(L)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Hãy chú ý tới vòng lặp while True mà chúng tôi sử dụng, nó sẽ lặp mãi mãi trừ phi ta thêm vào câu lệnh break!

Vòng lặp chứa khối mã else

Một cách làm hiếm khi được sử dụng trong Python, đó là sử dụng câu lệnh `else` cùng với vòng lặp `for` hoặc `while`. Chúng ta đã tìm hiểu về khối mã `else` trước đó: nó sẽ thực thi nếu tất cả câu lệnh `if` và `elif` mang giá trị `False`. Vòng lặp chứa else có là một câu lệnh khó hiểu hơn cả trong Python; Tôi sẽ xem nó như là một câu lệnh `nobreak`: có nghĩa là khối mã else chỉ thực thi nếu vòng lặp kết thúc một cách tự nhiên, mà không vướng phải câu lệnh `break`.

Một ví dụ về việc có thể nó sẽ hữu ích, xét quá trình thực hiện (không tối ưu) *Sàng nguyên tố Eratosthenes*, hay còn được biết đến như hàm tìm số nguyên tố dưới đây:

```
In [9]: L = []
        nmax = 30

        for n in range(2, nmax):
            for factor in L:
                if n % factor == 0:
                    break
            else: # không break
                L.append(n)
        print(L)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Câu lệnh `else` chỉ thực thi nếu không có nhân tố nào chia hết cho số cho sẵn. Câu lệnh `else` hoạt động tương tự với vòng lặp `while`.

Định nghĩa và sử dụng Hàm

Cho đến nay, tập lệnh của chúng ta chỉ là những khối mã đơn giản. Một cách để tổ chức khối mã Python và giúp cho chúng dễ đọc cũng như dễ sử dụng lại là đưa yếu tố quan trọng vào các *hàm (function)* có thể tái sử dụng. Ta sẽ tìm hiểu khái quát hai cách để tạo ra hàm: câu lệnh `def`, hữu ích với bất cứ loại hàm nào, và câu lệnh `lambda`, hữu ích với việc tạo ra những hàm không tên ngắn.

Sử dụng Hàm

Hàm (Function) là những nhóm mã có tên và có thể gọi bằng cách sử dụng dấu ngoặc đơn. Ta đã bắt gặp hàm trước đó. Ví dụ, `print` trong Python 3 là một hàm:

```
In [1]: print('abc')
abc
```

`print` ở đây là tên hàm, còn "abc" là *đối số (argument)* của hàm.

Ngoài ra còn có những đối số được đặt tên với một từ khóa *(keyword arguments)*. Một đối số như vậy của hàm `print()` (Python 3) là `sep`, đối số này sẽ cho biết ký tự hay những ký tự nào được sử dụng để phân tách các mục:

```
In [2]: print(1, 2, 3)
1 2 3
In [3]: print(1, 2, 3, sep='--')
1--2--3
```

Khi hai loại đối số được sử dụng cùng nhau thì đối số với từ khóa phải nằm ở cuối.

Định nghĩa Hàm

Hàm thậm chí còn trở nên hữu ích hơn khi ta tự định nghĩa hàm cho riêng mình, tổ chức hàm để sử dụng nhiều nơi. Trong Python, hàm được định nghĩa bằng câu lệnh `def`. Ví dụ, ta có thể tóm gọn lại đoạn mã về dãy Fibonacci lúc trước như sau:

```
In [4]: def fibonacci(N):
            L = []
            a, b = 0, 1
            while len(L) < N:
                a, b = b, a + b
                L.append(a)
            return L
```

Giờ ta đã có hàm tên là `fibonacci` với một đối số `N` duy nhất, thực hiện một số hành động với đối số này và trả về một giá trị; trong trường hợp này là một danh sách `N` số Fibonacci đầu tiên:

```
In [5]:  fibonacci(10)
Out [5]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Nếu đã quen với những ngôn ngữ mạnh như C, bạn sẽ ngay lập tức chú ý đến việc không hề có thông tin về kiểu dữ liệu gắn với đầu vào và đầu ra của hàm. Hàm trong Python có thể trả về bất kỳ đối tượng nào, đơn giản hay hỗn hợp, điều này có nghĩa là những cấu trúc có thể khó với ngôn ngữ khác thì lại rất đơn giản với Python.

Ví dụ, nhiều giá trị trả về cùng lúc đơn giản chỉ cần đặt trong một bộ dữ liệu được phân ra bằng dấu phẩy:

```
In [6]: def real_imag_conj(val):
            return val.real, val.imag, val.conjugate()
        r, i, c = real_imag_conj(3 + 4j)
        print(r, i, c)

3.0 4.0 (3-4j)
```

Giá trị mặc định của đối số

Thông thường vào lúc định nghĩa hàm, có một số giá trị mà ta muốn sự dụng trong *hầu hết* mọi trường hợp, nhưng ta cũng muốn để cho người dùng lựa chọn linh hoạt. Trong trường hợp này, ta có thể sử dụng *giá trị mặc định (default values)* cho đối số. Xét hàm `fibonacci` lúc trước. Sẽ ra sao nếu ta muốn người dùng có thể tùy chỉnh được những giá trị khởi đầu của dãy? Ta có thể làm như sau:

```
In [7]: def fibonacci(N, a=0, b=1):
            L = []
            while len(L) < N:
                a, b = b, a + b
                L.append(a)
            return L
```

Với một đối số duy nhất, kết quả của phép gọi hàm vẫn giống hệt như trước:

```
In [8]:  fibonacci(10)
Out [8]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Giờ ta có thể sử dụng lệnh gọi hàm để tìm hiểu một số thứ, như là tác động khi ta sử dụng những giá trị khởi đầu mới:

```
In [9]:  fibonacci(10, 0, 2)
Out [9]: [2, 2, 4, 6, 10, 16, 26, 42, 68, 110]
```

Giá trị cũng có thể được chỉ định bằng tên nếu muốn, trong trường hợp đó, thứ tự của các giá trị được đặt tên không quan trọng:

```
In [10]:  fibonacci(10, b=3, a=1)
Out [10]: [3, 4, 7, 11, 18, 29, 47, 76, 123, 199]
```

*args và **kwargs: Những đối số linh hoạt

Có thể đôi lúc bạn muốn viết một hàm mà lại không biết người dùng sẽ đưa vào bao nhiêu đối số. Trong trường hợp này, bạn có thể sử dụng các dạng đặc biệt là `*args` và `**kwargs` để bao hàm hết tất cả đối số được đưa vào. Xem ví dụ dưới đây:

```
In [11]: def catch_all(*args, **kwargs):
             print("args = ", args)
             print("kwargs = ", kwargs)

In [12]: catch_all(1, 2, 3, a=4, b=5)
         args = (1, 2, 3)
         kwargs = {'a': 4, 'b': 5}

In [13]: catch_all('a', keyword=2)
         args = ('a',)
         kwargs = {'keyword': 2}
```

Tên gọi `args` và `kwargs` ở đây không quan trọng, mà là ký hiệu * đằng trước chúng. `args` and `kwags` chỉ là tên của biến thường được dùng theo quy ước, là viết tắt của “arguments” và “keyword arguments”. Sự khác biệt nằm trong những ký hiệu hoa thị: một dấu * trước biến có nghĩa là “triển khai như một tổ hợp”, trong khi hai dấu * trước biến có nghĩa là “triển khai như một từ điển”. Trong thực tế, cú pháp này không chỉ được sử dụng trong việc định nghĩa hàm mà còn có thể sử dụng trong gọi hàm!

```
In [14]: inputs = (1, 2, 3)
         keywords = {'pi': 3.14}
```

```
catch_all(*inputs, **keywords)

args = (1, 2, 3)
kwargs =  {'pi': 3.14}
```

Hàm ẩn danh (lambda)

Trước đó ta đã tìm hiểu khái quát về cách định nghĩa hàm phổ biến nhất bằng câu lệnh `def`. Bạn sẽ có khả năng bắt gặp cách định nghĩa ngắn gọn khác là hàm sử dụng một lần với câu lệnh `lambda`. Nó sẽ trông như thế này:

```
In [15]:  add = lambda x, y: x + y
          add(1, 2)
Out [15]: 3
```

Hàm lambda này tương đương với

```
In [16]: def add(x, y):
             return x + y
```

Vậy tại sao lại sử dụng một thứ như vậy? Nó xuất phát từ thực tế rằng *mọi thứ đều là đối tượng* trong Python, thậm chí ngay cả hàm cũng vậy! Có nghĩa là hàm có thể dùng như là đối số của hàm khác.

Để ví dụ cho điều này, hãy giả sử ta có một số dữ liệu được lưu trong một danh sách từ điển:

```
In [17]: data =
[{'first':'Guido', 'last':'Van Rossum', 'YOB':1956},
 {'first':'Grace', 'last':'Hopper',     'YOB':1906},
 {'first':'Alan',  'last':'Turing',     'YOB':1912}]
```

Giờ giả sử ta muốn sắp xếp lại dữ liệu này, Python có hàm `sorted` để làm điều đó:

```
In [18]:  sorted([2,4,3,5,1,6])
Out [18]: [1, 2, 3, 4, 5, 6]
```

Nhưng từ điển không có thứ tự: ta cần một cách để chỉ cho hàm biết làm thế nào để sắp xếp dữ liệu đó. Ta có thể làm được bằng cách chỉ định hàm `key`, khi truyền cho hàm đó một mục thì nó sẽ trả về khóa sắp xếp cho mục đó:

```
In [19]:  # sắp xếp theo thứ tự alphabet của tên
          sorted(data, key=lambda item: item['first'])
Out [19]:
[{'YOB': 1912, 'first': 'Alan', 'last': 'Turing'},
 {'YOB': 1906, 'first': 'Grace', 'last': 'Hopper'},
 {'YOB': 1956, 'first': 'Guido', 'last': 'Van Rossum'}]

In [20]:  # sắp xếp theo thứ tự năm sinh
          sorted(data, key=lambda item: item['YOB'])
Out [20]:
[{'YOB': 1906, 'first': 'Grace', 'last': 'Hopper'},
{'YOB': 1912, 'first': 'Alan', 'last': 'Turing'},
{'YOB': 1956, 'first': 'Guido', 'last': 'Van Rossum'}]
```

Mặc dù những hàm key này có thể tạo ra theo cách thông thường bằng cú pháp `def`, nhưng cú pháp `lambda` tỏ ra thuận tiện hơn cho những hàm ngắn sử dụng một lần như vậy.

Lỗi và Ngoại lệ

Không cần biết kĩ năng lập trình của bạn cao đến đâu, sẽ có lúc bạn mắc phải những sai sót. Những sai sót như vậy có ba kiểu cơ bản:

Lỗi cú pháp (Syntax error)

Lỗi xảy ra khi đoạn mã không hợp lệ với Python (đa số dễ sửa)

Lỗi thực thi (Runtime error)

Lỗi xảy ra khi cú pháp hợp lệ nhưng đoạn mã không thể thực thi, có thể do dữ liệu đầu vào của người dùng không hợp lệ (đôi khi dễ sửa)

Lỗi ngữ nghĩa (Semantic error)

Lỗi logic: mã thực thi bình thường, nhưng kết quả không đúng như mong đợi (thường rất khó để xác định và sửa chữa)

Sau đây chúng ta sẽ tập trung vào cách để xử lý lỗi thực thi. Ta sẽ thấy rằng Python xử lý các lỗi thực thi thông qua khung (framework) *xử lý ngoại lệ* của nó.

Lỗi thực thi (Runtime error)

Nếu đã từng thực hiện đoạn mã nào với Python, bạn có thể đã bắt gặp những lỗi thực thi. Chúng có thể xảy ra theo nhiều cách.

Ví dụ, nếu cố tham chiếu vào một biến không xác định:

```
In [1]: print(Q)
---------------------------------------------------------
NameError                 Traceback (most recent call last)

<ipython-input-3-e796bdcf24ff> in <module>()
----> 1 print(Q)

NameError: name 'Q' is not defined
```

Hay cố thực hiện một thao tác không được định nghĩa:

```
In [2]: 1 + 'abc'
---------------------------------------------------------
TypeError                 Traceback (most recent call last)

<ipython-input-4-aab9e8ede4f7> in <module>()
----> 1 1 + 'abc'

TypeError: unsupported operand type(s) for +: 'int' and
'str'
```

Hay thực hiện một phép tính có kết quả không xác định về mặt toán học:

```
In [3]: 2 / 0
---------------------------------------------------------
ZeroDivisionError                 Traceback (most recent
call last)

<ipython-input-5-ae0c5d243292> in <module>()
----> 1 2 / 0

ZeroDivisionError: division by zero
```

Hay cố truy xuất vào một tổ hợp phần tử không tồn tại:

```
In [4]: L = [1, 2, 3]
        L[1000]
---------------------------------------------------------
IndexError                Traceback (most recent call
last)

<ipython-input-6-06b6eb1b8957> in <module>()
      1 L = [1, 2, 3]
----> 2 L[1000]

IndexError: list index out of range
```

Lưu ý rằng với mỗi trường hợp, Python không đơn giản chỉ là thông báo rằng đã có lỗi xảy ra, mà còn cung cấp một ngoại lệ bao gồm thông tin chính xác về những gì đã xảy ra cùng với dòng mã nơi xảy ra lỗi. Có được quyền truy cập vào lỗi như thế này là điều vô cùng hữu ích khi cố gắng truy tìm nguồn gốc của các vấn đề trong đoạn mã của bạn.

Nắm bắt ngoại lệ: try và except

Công cụ chính mà Python cung cấp để xử lý các ngoại lệ thực thi là mệnh đề `try…except`. Cấu trúc cơ bản của nó là:

```
In [5]:
try:
    print("this gets executed first")
except:
    print("this gets executed only if there is an
error")

this gets executed first
```

Lưu ý rằng khối mã thứ hai ở đây không được thực thi bởi vì khối mã thứ nhất không trả về lỗi nào. Hãy đặt một câu lệnh có vấn đề vào khối mã `try` và xem chuyện gì xảy ra:

```
In [6]: try:
            print("let's try something:")
            x = 1 / 0 # ZeroDivisionError
```

```
        except:
            print("something bad happened!")

        let's try something:
        something bad happened!
```

Ta có thể thấy khi cõ lỗi xảy ra trong câu lệnh `try` (trong trường hợp này là `ZeroDivisionError`), lỗi đã được phát hiện, và câu lệnh `except` được thực thi.

Cách này thường được sử dụng để kiểm tra đầu vào của người dùng trong hàm hay một đoạn mã khác. Ví dụ, ta có thể muốn có một hàm bắt lỗi chia cho số 0 và trả lại một giá trị khác; có thể là một con số lớn thích hợp như 10^{100}:

```
In [7]:  def safe_divide(a, b):
             try:
                 return a / b
             except:
                 return 1E100
In [8]:  safe_divide(1, 2)
Out [8]: 0.5
In [9]:  safe_divide(2, 0)
Out [9]: 1e+100
```

Dù vậy vẫn còn một vấn đề với đoạn mã này: chuyện gì sẽ xảy ra khi một ngoại lệ khác xuất hiên? Ví dụ, đây có lẽ không phải là điều mà ta dự tính:

```
In [10]:  safe_divide(1, '2')
Out [10]: 1e+10
```

Phép chia một số nguyên cho một chuỗi gây ra lỗi `TypeError` mà đoạn mã của chúng ta đã bắt được và giả định đó là `ZeroDivisionError`! Vì lý do này, tốt hơn hết là nắm bắt các ngoại lệ một cách *rõ ràng:*

```
In [11]:  def safe_divide(a, b):
              try:
                  return a / b
              except ZeroDivisionError:
                  return 1E100
In [12]:  safe_divide(1, 0)
```

```
Out [12]: 1e+100
In [13]:  safe_divide(1, '2')
---------------------------------------------------------

TypeError                 Traceback (most recent call last)

<ipython-input-15-2331af6a0acf> in <module>()
----> 1 safe_divide(1, '2')

<ipython-input-13-10b5f0163af8> in safe_divide(a, b)
      1 def safe_divide(a, b):
      2     try:
----> 3         return a / b
      4     except ZeroDivisionError:
      5         return 1E100

TypeError: unsupported operand type(s) for /: 'int' and
'str'
```

Bây giờ ta chỉ bắt mỗi lỗi chia cho số 0 và bỏ qua tất cả các lỗi khác.

Đưa ra ngoại lệ: raise

Ta đã thấy được giá trị mà các ngoại lệ mang lại khi sử dụng ngôn ngữ Python. Nó cũng có giá trị tương đương khi đưa ra những ngoại lệ trong chính mã của bạn, để cho người dùng (mà trước tiên là chính bạn!) có thể tìm ra nguyên nhân gây ra lỗi.

Cách để tự đưa ra những ngoại lệ là dùng câu lệnh `raise`. Ví dụ:

```
In [14]: raise RuntimeError("my error message")
---------------------------------------------------------
RuntimeError                 Traceback (most recent call last)

<ipython-input-16-c6a4c1ed2f34> in <module>()
----> 1 raise RuntimeError("my error message")

RuntimeError: my error message
```

Một ví dụ để thấy việc này có thể sẽ có ích, hãy quay trở lại với hàm `fibonacci` mà ta đã định nghĩa trước đó:

```
In [15]: def fibonacci(N):
             L = []
             a, b = 0, 1
             while len(L) < N:
                 a, b = b, a + b
                 L.append(a)
             return L
```

Một vấn đề tiềm tàng ở đây là giá trị đầu vào có thể âm. Hiện tại điều này không gây ra bất cứ lỗi nào trong hàm của chúng ta, những có lẽ ta sẽ muốn cho người dùng biết rằng số `N` âm không được hỗ trợ. Theo quy ước, khi gặp những lỗi xuất phát từ việc giá trị tham số không hợp lệ, lỗi `ValueError` sẽ được đưa ra:

```
In [16]:  def fibonacci(N):
              if N < 0:
                  raise ValueError("N must be non-
negative")
              L = []
              a, b = 0, 1
              while len(L) < N:
                  a, b = b, a + b
                  L.append(a)
              return L

In [17]:  fibonacci(10)
Out [17]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

In [18]:  fibonacci(-10)
---------------------------------------------------------
RuntimeError                 Traceback (most recent call
last)
<ipython-input-20-3d291499cfa7> in <module>()
----> 1 fibonacci(-10)

<ipython-input-18-01d0cf168d63> in fibonacci(N)
      1 def fibonacci(N):
      2     if N < 0:
```

```
----> 3         raise ValueError("N must be non-
negative")
      4     L = []
      5     a, b = 0, 1


ValueError: N must be non-negative
```

Giờ thì người dùng đã biết chính xác tại sao đầu vào không hợp lệ, và thậm chí còn có thể sử dụng khối mã `try…except` để xử lý nó!

```
In [19]: N = -10
         try:
             print("trying this...")
             print(fibonacci(N))
         except ValueError:
             print("Bad value: need to do something
else")

         trying this...
         Bad value: need to do something else
```

Tìm hiểu sâu hơn về ngoại lệ

Phần này tôi muốn đề cập về một số trường hợp mà bạn có thể gặp phải. Tôi sẽ không đi sâu vào chi tiết về các trường hợp này và tại sao lại sử dụng chúng, thay vào đó chỉ đơn giản cung cấp cú pháp để bạn có thể tự tìm hiểu thêm.

Truy xuất thông báo lỗi

Đôi lúc trong câu lệnh `try...except`, bạn muốn thao tác với chính các thông báo lỗi. Điều này có thể được thực hiện với từ khóa `as`:

```
In [20]: try:
             x = 1 / 0
         except ZeroDivisionError as err:
             print("Error class is:  ", type(err))
             print("Error message is:", err)

         Error class is:   <class 'ZeroDivisionError'>
```

```
Error message is: division by zero
```

Với cách này, bạn có thể điều chỉnh những tác vụ xử lý ngoại lệ trong hàm của mình.

Định nghĩa ngoại lệ

Ngoài các ngoại lệ có sẵn, có thể tự định nghĩa các ngoại lệ thông qua *lớp kế thừa (class inheritance).* Ví dụ bạn muốn có một trường hợp đặc biệt của `ValueError`, có thể làm như sau:

```
In [21]: class MySpecialError(ValueError):
             pass
         raise MySpecialError("here's the message")
---------------------------------------------------------
MySpecialError          Traceback (most recent call last)

<ipython-input-23-92c36e04a9d0> in <module>()
      2     pass
      3
----> 4 raise MySpecialError("here's the message")

MySpecialError: here's the message
```

Đoạn mã dưới đây cho phép khối `try…except` chỉ bắt một loại lỗi:

```
In [22]: try:
             print("do something")
             raise MySpecialError("[informative error
message here]")
         except MySpecialError:
             print("do something else")

do something
do something else
```

Bạn có thể thấy việc này hữu ích khi phát triển mã tùy chỉnh hơn.

try…except…else…finally

Ngoài `try` và `except`, bạn còn có thể sử dụng từ khóa `else` và `finally` để điều chỉnh nhiều hơn việc xử lý ngoại lệ. Cấu trúc cơ bản như sau:

```
In [23]: try:
             print("try something here")
         except:
             print("this happens only if it fails")
         else:
             print("this happens only if it succeeds")
         finally:
             print("this happens no matter what")

try something here
this happens only if it succeeds
this happens no matter what
```

Công dụng của `else` ở đây đã quá rõ ràng, nhưng còn `finally` để làm gì? Mệnh đề finally luôn thực thi *bất kể điều gì xảy ra*: tôi thường thấy nó được sử dụng để thực hiện dọn dẹp sau khi công việc hoàn thành.

Trình lặp (Iterator)

Thông thường, một phần quan trọng của việc phân tích dữ liệu là lặp đi lặp lại một phép tính tương tự nhau một cách tự động. Ví dụ, bạn muốn chia những cái tên trong một bảng thành họ và tên, hay chuyển đổi ngày tháng sang định dạng chuẩn. Một trong những hướng giải quyết của Python cho vấn đề này là cú pháp *trình lặp*. Ta đã từng thấy loại cú pháp này với trình lặp `range`:

```
In [1]: for i in range(10):
            print(i, end=' ')
0 1 2 3 4 5 6 7 8 9
```

Ta sẽ đi sâu hơn một chút về vấn đề này. Trong Python 3, range không phải là một danh sách, mà là một thứ được gọi là *trình lặp*, và tìm hiểu cách hoạt động của nó là chìa khóa để hiểu được một lượng lớn các hàm Python hữu ích.

Lặp thông qua danh sách

Có lẽ cách dễ nhất để hiểu về trình lặp là lặp thông qua một danh sách. Xét ví dụ dưới đây:

```
In [2]: for value in [2, 4, 6, 8, 10]:
            # thực hiện một số phép tính
            print(value + 1, end=' ')

3 5 7 9 11
```

Cú pháp quen thuộc `for x in y` cho phép ta lặp lại một số phép tính với mỗi giá trị của danh sách. Trong thực tế, việc cú pháp này rất giống với mô tả Tiếng Anh *(for [each] value in [the] list)* của nó là một trong những điều giúp cho Python trở thành một ngôn ngữ trực quan và dễ sử dụng.

Nhưng những giá trị mà ta thấy không hẳn là thứ hoạt động. Khi viết một đoạn mã như `for val in L`, trình thông dịch Python sẽ kiểm tra xem đó có phải là một giao diện trình lặp (iterator interface) hay không. Bạn có thể tự mình kiểm tra bằng hàm tích hợp `iter`:

```
In [3]:  iter([2, 4, 6, 8, 10])
Out [3]: <list_iterator at 0x104722400>
```

Chính đối tượng lặp này cung cấp hàm theo yêu cầu của vòng lặp `for`. Đối tượng `iter` là một bình chứa cho phép truy xuất vào giá trị tiếp theo miễn nó hợp lệ, giá trị này có thể đưa ra bằng hàm tích hợp `next`:

```
In [4]: I = iter([2, 4, 6, 8, 10])
In [5]: print(next(I))
2
In [6]: print(next(I))
4
In [7]: print(next(I))
6
```

Vậy mục đích của cách tiếp cận gián tiếp này là gì? Hóa ra điều này cực kì hữu ích khi cho phép Python thao tác với những đối tượng không phải danh sách như là một danh sách.

range(): Danh sách không phải lúc nào cũng là danh sách

Có lẽ ví dụ phổ biến nhất của việc lặp gián tiếp trong Python 3 chính là hàm `range()` (hay `xrange()` trong Python 2), hàm này không trả về một danh sách mà là một đối tượng `range()` đặc biệt:

```
In [8]:  range(10)
Out [8]: range(0, 10)
```

`range` cũng giống như danh sách, là một trình lặp

```
In [9]:  iter(range(10))
Out [9]: <range_iterator at 0x1045a1810>
```

Vì vậy Python hiểu và thao tác với nó *như là* một danh sách:

```
In [10]: for i in range(10):
             print(i, end=' ')

0 1 2 3 4 5 6 7 8 9
```

Lợi ích của trình lặp gián tiếp là *danh sách đầy đủ không bao giờ được tạo ra một cách rõ ràng!* Ta có thể thấy rõ được điều này khi thực hiện một phép tính với range, nó có thể khiến bộ nhớ bị quá tải nếu ta thực sự tạo ra nó (lưu ý rằng trong Python 2, `range` tạo ra một danh sách, vì vậy việc chạy đoạn mã dưới đây không phải là ý tưởng tốt!)

```
In [11]: N = 10 ** 12
         for i in range(N):
             if i >= 10: break
             print(i, end=', ')

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Nếu `range` thực sự tạo ra một danh sách với một nghìn tỷ giá trị, nó sẽ ngốn hàng chục terabyte bộ nhớ máy: một sự lãng phí khi thực tế là chúng ta bỏ qua tất cả giá trị trừ 10 giá trị đầu tiên!

Trong thực tế, không có lý do gì để trình lặp phải kết thúc cả! Thư viện `intertools` của Python chứa hàm `count`, hàm này hoạt động như một khoảng vô hạn:

```
In [12]: from itertools import count

         for i in count():
             if i >= 10:
```

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

0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Nếu chúng ta không có lệnh phá vỡ vòng lặp, nó sẽ đếm mãi cho đến khi bị làm gián đoạn hay bị ngắt một cách thủ công (ví dụ như sử dụng ctrl-C).

Những trình lặp hữu ích

Cú pháp trình lặp này gần như được sử dụng rất phổ biến trong những kiểu tích hợp của Python cũng như nhiều đối tượng khoa học cụ thể mà ta sẽ tìm hiểu trong những phần sau. Phần này chúng ta sẽ tìm hiểu khái quát về nhiều trình lặp hữu ích hơn nữa trong Python.

enumerate

Nếu bạn không chỉ muốn lặp lại những giá trị trong một mảng, mà còn muốn theo dõi cả chỉ số của nó. Bạn có thể làm theo cách này:

```
In [13]: L = [2, 4, 6, 8, 10]
         for i in range(len(L)):
             print(i, L[i])

0 2
1 4
2 6
3 8
4 10
```

Mặc dù nó vẫn hoạt động, nhưng Python có cung cấp một cú pháp đơn giản hơn để làm điều này là trình lặp `enumerate`:

```
In [14]: for i, val in enumerate(L):
             print(i, val)

0 2
1 4
2 6
3 8
4 10
```

Đây là một cách "Python" hơn để liệt kê những chỉ số và giá trị trong một danh sách.

zip

Một trường hợp khác là bạn có nhiều danh sách và muốn lặp chúng cùng lúc. Tất nhiên là bạn có thể lặp thông qua chỉ số một cách không "Python" chút nào như trong ví trụ ta đã thấy trước đó, hoặc một cách tốt hơn là sử dụng trình lặp `zip` để ghép những đối tượng lặp được:

```
In [15]: L = [2, 4, 6, 8, 10]
         R = [3, 6, 9, 12, 15]
         for lval, rval in zip(L, R):
             print(lval, rval)
2 3
4 6
6 9
8 12
10 15
```

Bất kể có bao nhiêu đối tượng cũng có thể ghép lại với nhau, và nếu chúng có độ dài khác nhau thì đối tượng ngắn nhất sẽ quyết định độ dài của `zip`.

map và filter

Trình lặp `map` nhận một hàm và áp dụng nó cho những giá trị trong một trình lặp:

```
In [16]: # tìm 10 số chính phương đầu tiên
         square = lambda x: x ** 2
         for val in map(square, range(10)):
             print(val, end=' ')

0 1 4 9 16 25 36 49 64 81
```

Trình lặp `filter` cũng tương tự như vậy, ngoại trừ việc nó chỉ áp dụng cho những giá trị mà hàm lọc trả về True:

```
In [17]: # tìm giá trị chia hết cho 2 trong phạm vi 10
         is_even = lambda x: x % 2 == 0
         for val in filter(is_even, range(10)):
             print(val, end=' ')
```

```
0 2 4 6 8
```

Hàm `map` và `filter`, cùng với hàm `reduce` (trong mô-đun `functools` của Python) là những thành phần cơ bản của phong cách *lập trình hàm*, một phong cách lập trình có nhiều người ủng hộ mặc dù không phải là một phong cách lập trình chiếm ưu thế trong thế giới Python (xem ví dụ thư viện pytoolz).

Trình lặp có chức năng như đối số của hàm

Trong phần "*args và **kwargs: Những đối số linh hoạt" ở trang 47. ta đã biết rằng `*args` và `**kwargs` có thể được sử dụng để đưa những tổ hợp và từ điển vào các hàm. Nhưng thực ra cú pháp `*args` không chỉ hoạt động với tổ hợp mà với bất kỳ trình lặp nào:

```
In [18]: print(*range(10))
0 1 2 3 4 5 6 7 8 9
```

Vì vậy, ví dụ ta có thể gói gọn ví dụ về trình lặp map lúc trước thành:

```
In [19]: print(*map(lambda x: x ** 2, range(10)))
0 1 4 9 16 25 36 49 64 81
```

Thủ thuật này giúp ta trả lời được câu hỏi lâu đời trong những diễn đàn học Python: tại sao không có hàm `unzip()` có tác dụng trái ngược với hàm `zip()`? Nếu tập trung suy nghĩ một chút, bạn có thể nhận ra rằng trái ngược với `zip()` là... `zip()`! Chìa khóa chính là `zip()` có thể nối nhiều số của trình lặp hoặc tổ hợp với nhau. Quan sát ví dụ sau:

```
In [20]: L1 = (1, 2, 3, 4)
         L2 = ('a', 'b', 'c', 'd')

In [21]: z = zip(L1, L2)
         print(*z)
(1, 'a') (2, 'b') (3, 'c') (4, 'd')

In [22]: z = zip(L1, L2)
         new_L1, new_L2 = zip(*z)
         print(new_L1, new_L2)
(1, 2, 3, 4) ('a', 'b', 'c', 'd')
```

Suy nghĩ về điều này một lúc. Nếu hiểu được tại sao nó hoạt động thì bạn đã đi được một quãng xa trong hành trình tìm hiểu trình lặp Python!

Trình lặp chuyên dụng: itertools

Ta đã có cái nhìn tóm tắt về trình lặp vô hạn: `range`, `itertools.count` trước đây. Mô-đun `itertools` chứa một loạt các trình lặp hoàn toàn hữu ích; nó đáng để bạn khám phá xem những gì có thể dùng được. Xét hàm `itertools.permutations` như một ví dụ, nó sẽ cho ra toàn bộ hoán vị của một tập hợp:

```
In [23]: from itertools import permutations
         p = permutations(range(3))
         print(*p)
(0, 1, 2) (0, 2, 1) (1, 0, 2) (1, 2, 0) (2, 0, 1) (2, 1,
0)
```

Tương tự, hàm `itertools.combinations` cho ra kết quả là toàn bộ tổ hợp chứa `N` giá trị của một danh sách:

```
In [24]: from itertools import combinations
         c = combinations(range(4), 2)
         print(*c)

(0, 1) (0, 2) (0, 3) (1, 2) (1, 3) (2, 3)
```

Một trình lặp liên quan là `product` sẽ cho ra kết quả là các cặp giá trị của hai hay nhiều đối tượng lặp:

```
In [25]: from itertools import product
         p = product('ab', range(3))
         print(*p)

('a', 0) ('a', 1) ('a', 2) ('b', 0) ('b', 1) ('b', 2)
```

Còn rất nhiều trình lặp hữu ích có trong `itertools`: danh sách đầy đủ cùng với các ví dụ có thể tìm thấy tại tài liệu Python trực tuyến.

Danh sách tổng quát (List Comprehensions)

Nếu bạn đọc mã Python đủ nhiều, bạn rồi sẽ gặp cấu trúc ngắn gọn và hiệu quả hơn được biết đến với cái tên *danh sách tổng quát (list comprehension)*. Đây là một tính năng của Python mà tôi đoán là bạn sẽ thích nếu bạn chưa bao giờ dùng trước đây, nó sẽ trông như thế này:

```
In [1]:  [i for i in range(20) if i % 3 > 0]
Out [1]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

Kết quả của đoạn mã trên là một danh sách số không phải bội số của 3. Mặc dù ví dụ này có vẻ hơi mơ hồ lúc đầu, nhưng khi đã quen với sự phát triển của Python, đọc và viết hàm tổng quát sẽ trở thành một đặc tính.

Danh sách tổng quát cơ bản

Danh sách tổng quát là một cách đơn giản để tối giản việc tạo danh sách cho vòng lặp for thành một dòng ngắn gọn và dễ đọc. Ví dụ, dưới đây là một vòng lặp tạo ra một danh sách 12 số chính phương đầu tiên:

```
In [2]:  L = []
         for n in range(12):
             L.append(n ** 2)
         L
Out [2]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Danh sách tổng quát tương đương đoạn mã trên là:

```
In [3]:  [n ** 2 for n in range(12)]
Out [3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Như nhiều câu lệnh Python khác, bạn gần như có thể đọc được ý nghĩa câu lệnh này như một văn bản tiếng Anh: “tạo một danh sách gồm những bình phương của số n với n lên đến 12”.

Cú pháp cơ bản là `[expr for var in iterable]`, với `expr` là bất kỳ biểu thức hợp lệ nào, `var` là tên biến, và `iterable` là bất kỳ đối tượng lặp nào trong Python.

Lặp nhiều lần

Đôi lúc bạn muốn tạo một danh sách không chỉ với một mà là hai giá trị. Để làm điều này, đơn giản chỉ cần thêm một câu lệnh `for` vào biểu thức tổng quát:

```
In [4]:  [(i, j) for i in range(2) for j in range(3)]
Out [4]: [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1,
2)]
```

Chú ý rằng biểu thức `for` thứ hai hoạt động như một nội chỉ số (interior index), thay đổi nhanh nhất trong danh sách kết quả. Loại cấu trúc này có thể mở rộng thành ba, bốn, hay thậm chí nhiều trình lặp hơn trong biểu thức tổng quát, tuy nhiên điều này sẽ khiến đoạn mã mất đi tính dễ đọc vốn có.

Điều kiện trình lặp

Bạn có thể kiểm soát vòng lặp bằng cách thêm một điều kiện vào cuối biểu thức. Trong ví dụ đầu tiên của phần này, ta đã cho vòng lặp chạy qua toàn bộ số từ 1 đến 20 nhưng bỏ đi những số là bội số của 3. Hãy cùng xem lại ví dụ này một lần nữa:

```
In [5]:  [val for val in range(20) if val % 3 > 0]
Out [5]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

Biểu thức (`i % 3 > 0`) mang giá trị True trừ khi val chia hết cho 3. Một lần nữa, ý nghĩa của biểu thức này có thể ngay lập tức diễn đạt bằng tiếng Anh: “Tạo ra một danh sách những giá trị của các số từ 1 đến 20 và không chia hết cho 3”. Một khi đã quen với điều này, đây sẽ là cách đơn giản hơn nhiều để viết và hiểu so với cú pháp vòng lặp tương đương:

```
In [6]:  L = []
         for val in range(20):
             if val % 3:
                 L.append(val)
         L
Out [6]: [1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19]
```

Điều kiện giá trị

Nếu đã từng lập trình với C, bạn có thể quen với điều kiện đơn dòng được kích hoạt bởi toán tử ?:

```
int absval = (val < 0) ? -val : val
```

Python cũng có công cụ tương tự, thường được sử dụng nhiều nhất trong danh sách tổng quát, hàm `lambda`, và những nơi khác cần đến một biểu thức đơn giản:

```
In [7]:  val = -10
         val if val >= 0 else -val
Out [7]: 10
```

Ta thấy rằng đây là một bản sao đơn giản tính năng của hàm tích hợp `abs()`, nhưng cấu trúc của nó cho phép bạn làm một số điều thú vị với danh sách tổng quát. Điều này sẽ trở nên phức tạp, nhưng bạn có thể làm được những điều như:

```
In [8]:  [val if val % 2 else -val
          for val in range(20) if val % 3]
Out [8]: [1, -2, -4, 5, 7, -8, -10, 11, 13, -14, -16,
17, 19]
```

Lưu ý chỗ ngắt dòng trong danh sách tổng quát ngay trước biểu thức `for`: điều này hoàn toàn hợp lệ trong Python, và thường là một cách hay để chia nhỏ các danh sách dài giúp dễ theo dõi hơn. Hãy xem qua đoạn mã: những gì ta đang làm là tạo ra một danh sách, bỏ qua bội số của 3 và lấy số đối những số là bội của 2.

Một khi đã hiểu được cơ chế của danh sách tổng quát, sẽ rất đơn giản để chuyển qua những dạng tổng quát khác. Cú pháp phần lớn là giống nhau; điểm khác nhau duy nhất chính là việc bạn sử dụng dấu ngoặc nào.

Ví dụ, với dấu ngoặc nhọn, bạn có thể tạo ra một set với *bộ tổng quát*:

```
In [9]:  {n**2 for n in range(12)}
Out [9]: {0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121}
```

Nhớ rằng set là một tập hợp không trùng lặp. Bộ tổng quát cùng tuân theo quy tắc này và loại bỏ những mục trùng lặp:

```
In [10]:  {a % 3 for a in range(1000)}
Out [10]: {0, 1, 2}
```

Với một điều chỉnh nhỏ, bạn có thể thêm dấu hai chấm `(:)` để tạo ra *từ điển tổng quát:*

```
In [11]:  {n:n**2 for n in range(6)}
Out [11]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Cuối cùng, nếu sử dụng dấu ngoặc đơn thay vì ngoặc vuông, bạn sẽ có được thứ có tên là *biểu thức trình tạo (generator expression):*

```
In [12]:  (n**2 for n in range(12))
Out [12]: <generator object <genexpr> at 0x1027a5a50>
```

Một biểu thức trình tạo về bản chất là một danh sách tổng quát mà những phần tử trong đó được tạo ra khi cần thiết thay vì tạo ra tất cả cùng lúc, và sự đơn giản này chính là sức mạnh của tính năng ngôn ngữ mà ta sẽ tìm hiểu ở phần tiếp theo.

Trình tạo (Generators)

Chúng ta sẽ đi sâu vào trình tạo Python, bao gồm những *biểu thức trình tạo (generator expressions)* và *hàm trình tạo (generator functions)*.

Biểu thức trình tạo (Generator Expression)

Sự khác nhau giữa danh sách tổng quát và biểu thức trình tạo đôi khi rất mập mờ; phần này ta sẽ tóm lược những khác biệt giữa hai khái niệm trên.

Danh sách tổng quát sử dụng ngoặc vuông, trong khi biểu thức trình tạo sử dụng ngoặc đơn

Đây là một danh sách tổng quát tiêu biểu:

```
In [1]:  [n ** 2 for n in range(12)]
Out [1]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Còn đây là một biểu thức trình tạo tiêu biểu:

```
In [2]:  (n ** 2 for n in range(12))
Out [2]: <generator object <genexpr> at 0x104a60518>
```

Để ý rằng việc in ra biểu thức trình tạo không in ra những nội dung trong đó; một cách để in ra nội dung của một trình tạo là đưa nó vào một hàm tạo list:

```
In [3]:  G = (n ** 2 for n in range(12))
         list(G)
```

```
Out [3]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Danh sách là một tổ hợp các giá trị, trong khi trình tạo là một công thức để xuất ra chúng

Khi tạo một danh sách, thực chất là bạn đang tạo ra một tổ hợp các giá trị, và cần tốn một lượng bộ nhớ cho việc đó. Trong khi đó với trình tạo bạn, không phải xây dựng một danh sách các giá trị mà là công thức để xuất ra các giá trị đó. Cả hai đều có cùng một giao diện trình lặp như ta có thể thấy dưới đây:

```
In [4]: L = [n ** 2 for n in range(12)]
        for val in L:
            print(val, end=' ')
0 1 4 9 16 25 36 49 64 81 100 121

In [5]: G = (n ** 2 for n in range(12))
        for val in G:
            print(val, end=' ')
0 1 4 9 16 25 36 49 64 81 100 121
```

Sự khác biệt là biểu thức trình tạo không thực sự tạo ra các giá trị cho đến khi cần thiết. Điều này không chỉ tạo sự hiệu quả về mặt bộ nhớ mà còn hiệu quả về mặt tính toán! Cũng đồng nghĩa với việc là trong khi *kích thước* của danh sách bị giới hạn bởi lượng bộ nhớ khả dụng thì *kích thước* của biểu thức trình tạo là vô hạn!

Một ví dụ về một biểu thức trình tạo vô hạn có thể được tạo ra bằng cách sử dụng trình lặp `count` được định nghĩa trong `itertools`:

```
In [6]:  from itertools import count
         count()
Out [6]: count(0)

In [7]:  for i in count():
             print(i, end=' ')
             if i >= 10: break
0 1 2 3 4 5 6 7 8 9 10
```

Trình lặp `count` sẽ tiếp tục đếm cho tới khi được lệnh dừng; điều này làm nó trở nên thuận tiện trong việc tạo ra những trình tạo chạy vô hạn:

```
In [8]:
factors = [2, 3, 5, 7]
G = (i for i in count() if all(i % n > 0 for n in
factors))
for val in G:
    print(val, end=' ')
    if val > 40: break

1 11 13 17 19 23 29 31 37 41
```

Có thể nhận thấy rằng nếu mở rộng danh sách các yếu tố một cách thích hợp, ta sẽ có được một trình tạo số nguyên tố sử dụng thuật toán Sàng nguyên tố Eratosthenes. Ta sẽ sớm tìm hiểu nhiều hơn về vấn đề này.

Danh sách có thể được lặp nhiều lần; biểu thức trình tạo sử dụng một lần

Đây là một trong những vấn đề tiềm tàng của biểu thức trình tạo. Với danh sách, ta có thể dễ dàng thực hiện việc này:

```
In [9]: L = [n ** 2 for n in range(12)]
        for val in L:
            print(val, end=' ')
        print()
        for val in L:
            print(val, end=' ')

0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121
```

Một biểu thức trình lặp, nói cách khác, được sử dụng hết sau một lần lặp:

```
In [10]:  G = (n ** 2 for n in range(12))
          list(G)
Out [10]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]

In [11]:  list(G)
Out [11]: []
```

Điều này có thể rất hữu ích bởi vì vòng lặp có thể bị tạm dừng và bắt đầu lại:

```
In [12]: G = (n**2 for n in range(12))
         for n in G:
             print(n, end=' ')
             if n > 30: break

         print("\ndoing something in between")

         for n in G:
             print(n, end=' ')
```

```
0 1 4 9 16 25 36
doing something in between
49 64 81 100 121
```

Một trường hợp hữu dụng của tính năng này là khi làm việc với những tổ hợp dữ liệu trên ổ cứng; có nghĩa là hoàn toàn có thể dễ dàng phân tích chúng theo từng đợt và để cho trình tạo quản lý những dữ liệu chưa được xử lý.

Hàm trình tạo (Generator Function): sử dụng yield

Ta đã biết rằng danh sách tổng quát là cách tốt nhất để tạo ra những danh sách tương đối đơn giản, trong khi sử dụng vòng lặp `for` sẽ tốt hơn trong những trường hợp phức tạp hơn. Điều đó cũng đúng với biểu thức trình tạo: ta có thể tạo ra những trình tạo phức tạp hơn bằng cách sử dụng *hàm trình tạo (generator function)*, tận dụng khả năng của câu lệnh `yield`.

Dưới đây là hai cách để cùng tạo ra một danh sách:

```
In [13]: L1 = [n ** 2 for n in range(12)]

         L2 = []
         for n in range(12):
             L2.append(n ** 2)

         print(L1)
         print(L2)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Tương tự, dưới đây là hai cách để xây dựng trình tạo tương đương:

```
In [14]: G1 = (n ** 2 for n in range(12))

         def gen():
             for n in range(12):
                 yield n ** 2

         G2 = gen()
         print(*G1)
         print(*G2)
```

```
0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121
```

Hàm trình tạo là một hàm, mà thay vì sử dụng `return` để trả về giá trị trong một lần thì sử dụng `yield` để xuất ra một chuỗi các giá trị (có thể vô hạn). Như trong biểu thức trình tạo, trạng thái của trình tạo được bảo toàn giữa những vòng lặp từng phần, nhưng nếu muốn một khởi động lại trình tạo thì đơn giản chỉ cần gọi lại hàm một lần nữa.

Ví dụ: Trình tạo số nguyên tố

Đây là ví dụ yêu thích của tôi về hàm trình tạo: một hàm tạo ra một chuỗi không giới hạn những số nguyên tố. Một thuật toán cổ điển cho việc này là Sàng nguyên tố Eratosthenes, nó sẽ hoạt động kiểu như sau:

```
In [15]: # Tạo danh sách các phần tử
         L = [n for n in range(2, 40)]
         print(L)
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, \
 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, \
 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]

In [16]: # Loại bỏ tất cả bội số của giá trị đầu tiên
         L = [n for n in L if n == L[0] or n % L[0] > 0]
         print(L)
[2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, \
 29, 31, 33, 35, 37, 39]
```

```
In [17]: # Loại bỏ tất cả bội số của giá trị thứ hai
         L = [n for n in L if n == L[1] or n % L[1] > 0]
         print(L)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37]

In [18]: # Loại bỏ tất cả bội số của giá trị thứ ba
         L = [n for n in L if n == L[2] or n % L[2] > 0]
         print(L)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37]
```

Nếu lặp lại hành động này đủ số lần trên một danh sách đủ lớn, ta có thể xuất ra bao nhiêu số nguyên tố mà ta muốn.

Hãy gói gọn lập luận này vào một hàm trình tạo:

```
In [19]: def gen_primes(N):
             """Generate primes up to N"""
             primes = set()
             for n in range(2, N):
                 if all(n % p > 0 for p in primes):
                     primes.add(n)
                     yield n


         print(*gen_primes(70))
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67
```

Tất cả chỉ có vậy! Mặc dù đây chắc chắn không phải là cách thực hiện hiệu quả của thuật toán Sàng nguyên tố Eratosthenes, nó thể hiện sự tiện lợi của cú pháp hàm trình tạo trong việc xây dựng những tập hợp phức tạp hơn.

Mô-đun và Gói

Một tính năng của Python giúp nó hữu dụng với một lượng lớn các tác vụ là có "bộ công cụ đi kèm", nghĩa là thư viện chuẩn của Python chứa những công cụ hữu dụng cho một lượng lớn tác vụ. Trên hết, có một hệ sinh thái rộng lớn gồm các công cụ và gói của bên thứ ba cung cấp với nhiều chức năng chuyên biệt hơn. Trong phần này, ta sẽ tìm hiểu việc nhập các mô-đun thư viện chuẩn, những công cụ để cài đặt các mô-đun của bên thứ ba và mô tả về cách để tự tạo ra những mô-đun của riêng mình.

Tải Mô-đun: câu lệnh import

Để tải các mô-đun tích hợp và mô-đun từ bên thứ ba, Python cung cấp câu lệnh `import`. Có một số cách để sử dụng câu lệnh mà ta sẽ đề cập ngắn gọn ở đây, từ cách được hầu hết mọi người khuyên dùng đến những cách ít được sử dụng.

Nhập mô-đun tường minh

Nhập tường minh một mô-đun giữ nội dung của nó trong một vùng tên (namespace). Vùng tên sau đó được sử dụng để viện dẫn đến nội dung của nó với một dấu . nằm giữa. Ví dụ, dưới đây ta sẽ nhập vào mô-đun tích hợp `math` và tính sin của số Pi:

```
In [1]:  import math
         math.cos(math.pi)
Out [1]: -1.0
```

Nhập mô-đun tường minh bằng tên lóng

Đối với những mô-đun có tên dài, việc sử dụng tên đầy đủ của mô-đun mỗi lần truy xuất một vài nội dung rất bất tiện. Vì lý do này, ta thường sử dụng cú pháp `import ... as ...` để tạo ra một tên lóng (alias) ngắn gọn cho vùng tên. Ví dụ, gói NumPy (Số học Python), một gói từ bên thứ ba phổ biến hữu dụng cho khoa học dữ liệu, được nhập vào dưới tên lóng là `np` theo quy ước:

```
In [2]:  import numpy as np
         np.cos(np.pi)
Out [2]: -1.0
```

Nhập tường minh nội dung mô-đun

Đôi lúc thay vì nhập vùng tên mô-đun, bạn chỉ muốn nhập một số mục trong đó. Điều này có thể thực hiện với cú pháp `from ... import ...`. Ví dụ, ta có thể chỉ nhập hàm `cos` và hằng số `pi` từ mô-đun `math`:

```
In [3]:  from math import cos, pi
         cos(pi)
Out [3]: -1.
```

Nhập bao hàm nội dung mô-đun

Cuối cùng, trường hợp này đôi lúc hữu ích cho việc nhập toàn bộ nội dung của mô-đun vào một vùng tên địa phương (local namespace). Điều này có thể thực hiện với cú pháp `from ... import *`:

```
In [4]:  from math import *
         sin(pi) ** 2 + cos(pi) ** 2
Out [4]: 1.0
```

Cú pháp này nên được sử dụng hạn chế. Vấn đề là cách nhập này đôi lúc có thể ghi đè lên tên hàm mà bạn không có chủ ý, và độ bao hàm của câu lệnh gây khó khăn trong việc xác định những nội dung đã bị thay đổi.

Ví dụ, Python có một hàm tích hợp `sum` có thể dùng cho nhiều phép tính khác nhau:

```
In [5]: help(sum)
        Help on built-in function sum in module
        builtins:
        sum(...)
            sum(iterable[, start]) -> value

            Return the sum of an iterable of numbers
            (NOT strings) plus the value of parameter
            'start' (which defaults to 0).
            When the iterable is empty, return start.
```

Hàm này có thể sử dụng để tính tổng của một tập hợp, bắt đầu với một giá trị cụ thể (-1 trong ví dụ dưới đây):

```
In [6]:  sum(range(5), -1)
Out [6]: 9
```

Giờ hãy quan sát kết quả khi ta thực hiện *gọi cùng một hàm* sau khi nhập * từ numpy:

```
In [7]:  from numpy import *
In [8]:  sum(range(5), -1)
Out [8]: 10
```

Kết quả bị lệch 1 đơn vị! Lý do là câu lệnh `import *` *thay thế* hàm tích hợp `sum` với hàm `numpy.sum`, với một ký hiệu lời gọi (call signature) khác: với hàm `sum`, ta tính tổng giá trị trong `range(5)` cộng thêm -1; với hàm `numpy.sum`, ta tính tổng giá trị trong `range(5)` dọc theo trục cuối cùng (được biểu thị bằng -1). Đây là trường hợp có thể phát sinh nếu không cẩn thận khi dùng `import *`, chính vì lý do này mà tốt nhất nên hạn chế sử dụng nó trừ khi bạn biết rõ mình đang làm gì.

Nhập từ Thư viện chuẩn Python

Thư viện chuẩn của Python chứa rất nhiều mô-đun tích hợp hữu dụng có thể tiếp cận một cách đầy đủ trong tài liệu Python. Bất kỳ mô-đun nào trong số đó đều có thể được nhập vào với câu lệnh `import`, sau khi tìm hiểu bằng hàm trợ giúp đã được thảo luận ở những phần trước. Dưới đây là danh sách không đầy đủ một số mô-đun mà bạn có thể muốn tìm hiểu:

os và sys	Các công cụ để giao tiếp với hệ điều hành, bao gồm điều hướng các cấu trúc thư mục tệp và thực thi các lệnh shell
math và cmath	Các hàm toán học và phép toán với số thực và số phức
itertools	Các công cụ để xây dựng và tương tác với các trình vòng lặp và trình tạo
functools	Các công cụ hỗ trợ lập trình hàm
random	Các công cụ tạo số giả ngẫu nhiên
pickle	Các công cụ để duy trì đối tượng: lưu đối tượng và tải đối tượng từ ổ cứng
json và csv	Các công cụ để đọc những tệp định dạng JSON và CSV
urllib	Các công cụ để thực hiện gửi yêu cầu HTTP và những yêu cầu web khác.

Bạn có thể tìm thêm thông tin về những mô-đun này và nhiều hơn nữa trong tài liệu thư viện chuẩn của Python: https://docs.python.org/3/library/.

Nhập mô-đun từ bên thứ ba

Một trong những điều làm cho Python trở nên hữu ích, đặc biệt trong thế giới khoa học dữ liệu, là hệ sinh thái của các mô-đun từ bên thứ ba. Chúng có thể được nhập như những mô-đun tích hợp, nhưng trước tiên các mô-đun phải được cài đặt trên hệ thống. Sổ đăng ký tiêu chuẩn cho các mô-đun như vậy là Chỉ số gói Python (viết tắt là PyPI), có sẵn tại địa chỉ http://pypi.python.org/. Để thuận

tiện, Python có kèm theo một chương trình gọi là pip (viết tắt đệ quy của "pip installs packages"), sẽ tự động tìm nạp những gói được phát hành và công bố trên PyPI (nếu sử dụng Python 2, pip phải được cài đặt riêng). Ví dụ: nếu muốn nếu muốn cài đặt `supersmoother`, tất cả những gì cần làm là nhập nội dung sau vào dòng lệnh:

```
$ pip install supersmoother
```

Mã nguồn của gói sẽ được tự động tải xuống từ kho lưu trữ PyPI và gói được cài đặt trong đường dẫn chuẩn của Python (giả định rằng bạn có quyền cài đặt trên máy tính đang sử dụng).

Để biết thêm thông tin về PyPI và trình cài đặt pip, hãy tham khảo tài liệu tại http://pypi.python.org/.

Thao tác chuỗi và biểu thức chính quy

Một điểm sáng của ngôn ngữ Python là thao tác với chuỗi. Phần này sẽ trình bày một số phương thức tích hợp liên quan đến chuỗi và các phép định dạng của Python, trước khi chuyển sang hướng dẫn về các *biểu thức chính quy (regular expression)*. Các thao tác chuỗi như vậy thường xuyên xuất hiện trong lĩnh vực khoa học dữ liệu và là một lợi thế lớn của Python trong lĩnh vực này.

Chuỗi trong Python có thể định nghĩa bằng cách sử dụng dấu nháy đơn hoặc nháy kép (chức năng tương đương nhau):

```
In [1]:  x = 'a string'
         y = "a string"
         x == y
Out [1]: True
```

Ngoài ra, có thể định nghĩa chuỗi đa dòng bằng cú pháp ba dấu nháy:

```
In [2]: multiline = """
        one
        two
        three
        """
```

Cùng điểm qua một số công cụ thao tác chuỗi trong Python.

Thao tác chuỗi đơn giản trong Python

Đối với thao tác cơ bản trên chuỗi, các phương thức tích hợp của Python có thể cực kỳ tiện lợi. Nếu có đã từng làm việc với C hoặc một ngôn ngữ cấp thấp khác, bạn có thể sẽ thấy sự đơn giản của các phương thức Python cực kỳ mới mẻ. Ta đã được giới thiệu về kiểu chuỗi của Python và một số các phương thức trước đó; ở phần này ta tìm hiểu sâu hơn một chút.

Định dạng chuỗi: Điều chỉnh hình thức

Python giúp dễ dàng hơn trong việc điều chỉnh hình thức của chuỗi. Ở đây ta sẽ xem xét các phương thức `upper()`, `lower()`, `capitalize()`, `title()`, và `swap case()`, sử dụng chuỗi lộn xộn sau đây làm ví dụ:

```
In [3]: fox = "tHe qUICk bROWn fOx."
```

Để chuyển đổi toàn bộ chuỗi thành chữ hoa hay chữ thường, có thể sử dụng phương thức `upper()` hoặc `lower()` tương ứng:

```
In [4]:  fox.upper()
Out [4]: 'THE QUICK BROWN FOX.'
In [5]:  fox.lower()
Out [5]: 'the quick brown fox.'
```

Một kiểu định dạng phổ biến là viết hoa chữ cái đầu tiên của mỗi từ, hoặc là chữ cái đầu tiên của mỗi câu. Điều này có thể được thực hiện với các phương thức `title()` và `capitalize()`:

```
In [6]:  fox.title()
Out [6]: 'The Quick Brown Fox.'
In [7]:  fox.capitalize()
Out [7]: 'The quick brown fox.'
```

Các hình thức có thể được hoán đổi bằng phương thức `swapcase()`:

```
In [8]:  fox.swapcase()
Out [8]: 'ThE QuicK BrowN FoX.'
```

Định dạng chuỗi: Thêm và xóa khoảng trắng

Một nhu cầu phổ biến khác là loại bỏ khoảng trắng (hoặc các ký tự khác) nằm ở đầu hoặc cuối chuỗi. Phương pháp cơ bản để loại bỏ các ký tự là phương thức `strip()`, loại bỏ khoảng trắng ở đầu và cuối dòng:

```
In [9]:  line = '          this is the content          '
         line.strip()
Out [9]: 'this is the content'
```

Để xóa khoảng trắng chỉ ở bên phải hoặc bên trái, hãy sử dụng `rstrip()` hoặc `lstrip()`, tương ứng:

```
In [10]:  line.rstrip()
Out [10]: '          this is the content'
In [11]:  line.lstrip()
Out [11]: 'this is the content
```

Để xóa các ký tự không phải khoảng trắng, có thể đưa ký tự mong muốn vào trong phương thức `strip()`:

```
In [12]:  num = "000000000000435"
          num.strip('0')
Out [12]: '435'
```

Ngược lại với thao tác này, việc thêm khoảng trắng hoặc các ký tự khác vào chuỗi có thể được thực hiện bằng cách sử dụng các phương thức `centre()`, `ljust()` và `rjust()`.

Ví dụ: ta có thể sử dụng phương thức `centre()` để căn giữa một chuỗi trong một lượng khoảng trắng cho trước:

```
In [13]:  line = "this is the content"
          line.center(30)
Out [13]: '     this is the content      '
```

Tương tự, `ljust()` và `rjust()` sẽ căn trái hoặc căn phải chuỗi trong một lượng khoảng trắng cho trước:

```
In [14]:  line.ljust(30)
Out [14]: 'this is the content           '
```

```
In [15]:  line.rjust(30)
Out [15]: '           this is the content'
```

Có thể đưa vào các phương thức trên bất kỳ ký tự nào để lập đầy khoảng trắng. Ví dụ:

```
In [16]:  '435'.rjust(10, '0')
Out [16]: '0000000435'
```

Bởi vì điền số không là một nhu cầu phổ biến, Python cũng cung cấp `zfill()`, là một phương thức đặc biệt đệm vào bên phải chuỗi một dãy số không:

```
In [17]:  '435'.zfill(10)
Out [17]: '0000000435'
```

Tìm và thay thế các chuỗi con

Nếu muốn tìm một ký tự nào đó trong một chuỗi, những phương thức `find()`/`rfind()`, `index()`/`rindex()` và `replace()` là những phương thức tích hợp tốt nhất .

`find()` và `index()` rất giống nhau, ở chỗ chúng đều tìm kiếm một ký tự hoặc chuỗi con xuất hiện đầu tiên trong một chuỗi và trả về chỉ mục của nó

```
In [18]:  line = 'the quick brown fox jumped over a lazy
dog'
          line.find('fox')
Out [18]: 16
In [19]:  line.index('fox')
Out [19]: 16
```

Sự khác biệt duy nhất giữa `find()` và `index()` là hoạt động của chúng khi không tìm thấy chuỗi cần tìm kiếm; `find()` trả về -1 , trong khi `index()` trả về `ValueError`:

```
In [20]:  line.find('bear')
Out [20]: -1
In [21]: line.index('bear')
---------------------------------------------------------

ValueError           Traceback (most recent call last)
```

```
<ipython-input-21-4cbe6ee9b0eb> in <module>() ----> 1
line.index('bear')

ValueError: substring not found
```

Phương thức `rfind()` và `rindex()` cũng hoạt động tương tự, ngoại trừ việc chúng bắt đầu tìm kiếm từ cuối chứ không phải đầu chuỗi:

```
In [22]:  line.rfind('a')
Out [22]: 35
```

Đối với trường hợp kiểm tra một chuỗi con có nằm ở đầu hoặc cuối chuỗi, Python cung cấp các phương thức `startedwith()` và `endswith()`:

```
In [23]:  line.endswith('dog')
Out [23]: True
In [24]:  line.startswith('fox')
Out [24]: False
```

Để thay thế một chuỗi đã cho bằng một chuỗi mới, có thể sử dụng phương thức `replace()`. Hãy thay thế '`brown`' với '`red`' trong ví dụ sau:

```
In [25]:  line.replace('brown', 'red')
Out [25]: 'the quick red fox jumped over a lazy dog'
```

Hàm `repalce()` trả về một chuỗi mới và thay thế tất cả các chuỗi xác định bằng chuỗi đầu vào.

```
In [26]:  line.replace('o', '--')
Out [26]: 'the quick br--wn f--x jumped --ver a lazy d—
g'
```

Để tiếp cận một cách linh hoạt hơn với chức năng `replace()`, hãy đọc về biểu thức chính quy trong "Khớp mẫu linh hoạt với Biểu thức chính quy" trang 83.

Chia tách và phân vùng chuỗi

Nếu muốn tìm một chuỗi con *và sau đó* tách chuỗi dựa trên vị trí của nó, phương thức `partition()` và/hoặc `split()` là hai công cụ phù hợp. Cả hai sẽ trả về một tập hợp các chuỗi con.

Phương thức `partition()` trả về một bộ dữ liệu với ba phần tử: chuỗi con trước đứng trước vị trí (chuỗi) phân tách đầu tiên, chuỗi phân tách và chuỗi con đúng sao nó:

```
In [27]:  line.partition('fox')
Out [27]: ('the quick brown ', 'fox', ' jumped over a
lazy dog')
```

Phương thức `rpartition()` cũng tương tự, nhưng bắt đầu tìm kiếm từ bên phải chuỗi.

Phương thức `split()` có lẽ hữu ích hơn; nó tìm tất cả các chuỗi phân tách và trả về các chuỗi con đứng giữa. Mặc định nó sẽ phân tách dựa trên nhưng khoảng trắng, trả về danh sách những từ đơn lẻ trong một chuỗi:

```
In [28]:  line.split()
Out [28]: ['the', 'quick', 'brown', 'fox', 'jumped',\
          'over', 'a', 'lazy', 'dog']
```

Một phương thức liên quan là `splitlines()` sẽ phân tách dựa trên những dấu ngắt dòng . Hãy làm điều này với một bài thơ haiku phổ biến được cho là của nhà thơ thế kỷ 17, Matsuo Bashō:

```
In [29]: haiku = """matsushima-ya
         aah matsushima-ya
         matsushima-ya"""

         haiku.splitlines()

['matsushima-ya', 'aah matsushima-ya', 'matsushima-ya']
```

Lưu ý rằng nếu muốn hoàn tác phương thức `split()`, có thể sử dụng phương thức `join()` , phương thức này trả về một chuỗi được xây dựng từ một chuỗi phân tách và một đối tượng lặp:

```
In [30]:  '--'.join(['1', '2', '3'])
Out [30]: '1--2--3'
```

Một cú pháp phổ biến là sử dụng ký tự đặc biệt `\n` (ngắt dòng) để khôi phục các dòng đã bị phân tách trước đó:

```
In [31]: print("\n".join(['matsushima-ya', 'aah
matsushima-ya',
                          'matsushima-ya']))

matsushima-ya
aah matsushima-ya
matsushima-ya
```

Chuỗi định dạng

Trong các phương thức trước, ta đã học cách trích xuất các giá trị từ chuỗi và định dạng chuỗi theo ý muốn. Một công dụng khác của những phương thức chuỗi là *biểu diễn* giá trị của những kiểu dữ liệu khác dưới dạng chuỗi. Tất nhiên, có thể sử dụng hàm `str()` để biểu diễn chuỗi; ví dụ:

```
In [32]:  pi = 3.14159
          str(pi)
Out [32]: '3.14159'
```

Với những định dạng phức tạp hơn, có thể sử dụng phép tính với chuỗi như đã tìm hiểu ở phần "Ngữ nghĩa Python cơ bản: Toán tử" trang 19:

```
In [33]:  "The value of pi is " + str(pi)
Out [33]: 'The value of pi is 3.14159'
```

Một cách linh hoạt hơn để làm điều này là sử dụng những *chuỗi định dạng (format string)*, đó là những chuỗi chứa các điểm đặc biệt được đánh dấu (biểu thị bằng dấu ngoặc nhọn) mà giá trị chuỗi định dạng sẽ được chèn vào. Đây là một ví dụ:

```
In [34]:  "The value of pi is {}".format(pi)
Out [34]: 'The value of pi is 3.14159'
```

Bên trong điểm `{}` được đánh dấu có thể bao gồm thông tin chính xác về những thứ bạn muốn chèn. Nếu đưa vào đó một chữ số, nó sẽ tham chiếu đến chỉ mục của đối số cần chèn:

```
In [35]:
"""First letter: {0}. Last letter: {1}.""".format('A',
'Z')
Out [35]: 'First letter: A. Last letter: Z.'
```

Nếu đưa vào một chuỗi, nó sẽ tham chiếu đến từ khóa của bất kỳ đối số được đặt tên nào:

```
In [36]:
"""First: {first}. Last: {last}.""".format(last='Z',
first='A')
Out [36]: 'First: A. Last: Z.'
```

Cuối cùng, đối với những đầu vào là số, có thể đưa vào những mã định dạng để kiểm soát cách chuyển đổi giá trị thành chuỗi. Ví dụ: để in một số dưới dạng dấu phẩy động với ba chữ số sau dấu thập phân, có thể làm như sau:

```
In [37]:  "pi = {0:.3f}".format(pi)
Out [37]: 'pi = 3.142'
```

Như đã biết, số ở đây `0` đề cập đến chỉ số của giá trị được chèn. Dấu `:` đánh dấu mã định dạng mà chuỗi sẽ tuân theo. `.3f` mã hóa độ chính xác mong muốn: Ba chữ số sau dấu thập phân, định dạng dấu phẩy động.

Kiểu diễn tả định dạng này rất linh hoạt và các ví dụ ở đây hầu như chỉ là một phần nhỏ trong số các tùy chọn định dạng có sẵn. Để biết thêm về cú pháp của những chuỗi định dạng, tham khảo phần "Format Specification" trong tài liệu Python trực tuyến.

Khớp mẫu linh hoạt với Biểu thức chính quy

Các phương thức của kiểu `str` trong Python cung cấp một bộ công cụ mạnh mẽ để định dạng, phân tách và xử lý dữ liệu chuỗi. Nhưng thậm chí còn có các công cụ mạnh hơn trong mô-đun *biểu thức chính quy (regular expression)* tích hợp của Python. Biểu thức chính quy là một chủ đề rất rộng; có những cuốn sách mà toàn bộ nội dung chỉ viết về chủ đề này (bao gồm Làm chủ biểu thức chính quy, Tái bản lần thứ 3 của Jeffrey E.F. Friedl), vì vậy khó có thể bao quát hết chỉ trong một tiểu mục.

Mục tiêu của tôi là cung cấp cho bạn khái niệm về các loại vấn đề có thể được giải quyết bằng cách sử dụng các biểu thức chính quy, cũng như làm sao để sử dụng chúng trong Python. Tôi sẽ đề xuất một số tài liệu tham khảo để tìm hiểu thêm về vấn đề này tại "Tài nguyên cho việc học thêm" trang 98.

Về cơ bản, biểu thức chính quy là một phương thức *khớp mẫu linh hoạt (flexible pattern matching)* trong chuỗi. Nếu thường xuyên sử dụng môi trường dòng

lệnh, có lẽ bạn đã quen với kiểu khớp mẫu linh hoạt này qua ký tự * hoạt động như một ký tự đại diện. Ví dụ: ta có thể liệt kê tất cả các sổ ghi chép IPython (tệp có phần mở rộng *.ipynb*) mà trong tên có chứa chuỗi "Python" bằng cách sử dụng ký tự `*` để đại diện cho tất cả những ký tự khác:

```
In [38]: !ls *Python*.ipynb
01-How-to-Run-Python-Code.ipynb 02-Basic-Python-
Syntax.ipynb
```

Các biểu thức chính quy khái quát hóa ý tưởng về "ký tự đại diện" này thành một loạt những cú pháp khớp chuỗi linh hoạt. Các biểu thức chính quy trong Python được chứa trong mô-đun tích hợp `re`; một ví dụ đơn giản, sử dụng nó để sao chép chức năng của phương thức `split()`:

```
In [39]:  import re
          regex = re.compile('\s+')
          regex.split(line)
Out [39]: ['the', 'quick', 'brown', 'fox', 'jumped', \
          'over', 'a', 'lazy', 'dog']
```

Đầu tiên ta đã *biên dịch* một biểu thức chính quy, sau đó sử dụng nó để *phân tách* một chuỗi. Giống như phương thức `split()` của Python trả về một danh sách tất cả các chuỗi con ở giữa các khoảng trắng, phương thức `split()` của biểu thức chính quy trả về một danh sách tất cả các chuỗi con ở giữa những kết quả khớp với mẫu đầu vào.

Trong trường hợp này, đầu vào \s+: \s là một ký tự đặc biệt khớp với bất kỳ khoảng trắng nào (dấu cách, khoảng trống (tab), ngắt dòng, v.v.) và dấu + là ký tự diễn tả rằng thực thể đứng trước nó có thể xuất hiện *một hoặc nhiều lần*. Do đó, biểu thức chính quy này khớp với bất kỳ chuỗi con nào bao gồm một hoặc nhiều khoảng trắng.

Phương thức `split()` ở đây về cơ bản là một hàm tiện dụng được xây dựng dựa trên hoạt động *khớp mẫu* này; phương thức cơ bản hơn là `match()` sẽ cho biết liệu phần đầu của chuỗi có khớp với mẫu hay không:

```
In [40]: for s in ["      ", "abc  ", "  abc"]:
             if regex.match(s):
                 print(repr(s), "matches")
             else:
                 print(repr(s), "does not match")
```

```
'     ' matches
'abc  ' does not match
'  abc' matches
```

Giống như `split()`, có những hàm tiện dụng tương tự để tìm ra kết quả khớp đầu tiên (như `str.index()` hoặc `str.find()`) hoặc để tìm kiếm và thay thế (như `str.replace()`). Ta sẽ sử dụng lại chuỗi trước đó:

```
In [41]: line = 'the quick brown fox jumped over a lazy
dog'
```

Với ví dụ này, ta có thể thấy rằng phương thức `regex.search()` hoạt động rất giống `str.index()` hay `str.find()`:

```
In [42]:  line.index('fox')
Out [42]: 16
In [43]:  regex = re.compile('fox')
          match = regex.search(line)
          match.start()
Out [43]: 16
```

Tương tự, phương thức `regex.sub()` hoạt động giống như `str.replace()`:

```
In [44]:  line.replace('fox', 'BEAR')
Out [44]: 'the quick brown BEAR jumped over a lazy dog'
In [45]:  regex.sub('BEAR', line)
Out [45]: 'the quick brown BEAR jumped over a lazy dog'
```

Nghĩ rộng ra một chút thì các hoạt động khác với chuỗi cũng có thể được thể hiện bằng các biểu thức chính quy.

Một ví dụ phức tạp hơn

Bạn có thể thắc mắc rằng, tại sao lại sử dụng những cú pháp dài dòng và phức tạp của các biểu thức chính quy thay vì các phương thức chuỗi đơn giản và trực quan hơn? Ưu điểm là biểu thức chính quy linh hoạt hơn rất nhiều.

Ta sẽ xét một ví dụ phức tạp hơn: lọc địa chỉ email. Tôi sẽ bắt đầu bằng cách viết một biểu thức chính quy (hơi khó hiểu), và sau đó diễn tả những gì đang diễn ra:

```
In [46]: email = re.compile('\w+@\w+\.[a-z]{3}')
```

Với đoạn mã này, ta có thể nhanh chóng trích xuất những thứ trông giống địa chỉ email từ một dòng trong tài liệu:

```
In [47]:  text = "To email Guido, try guido@python.org \
          or the older address guido@google.com."
          email.findall(text)
Out [47]: ['guido@python.org', 'guido@google.com']
```

(Lưu ý rằng những địa chỉ này hoàn toàn được dựng nên; có lẽ có nhiều cách tốt hơn để liên lạc với Guido).

Ta có thể làm được nhiều hơn, như là thay thế những địa chỉ email này bằng một chuỗi khác để ẩn địa chỉ trong đầu ra:

```
In [48]:  email.sub('--@--.--', text)
Out [48]: 'To email Guido, try --@--.-- or the older
address --@--.--.'
```

Cuối cùng, lưu ý rằng nếu thực sự muốn lọc được *tất cả* địa chỉ email, biểu thức chính quy trước đó là quá đơn giản. Ví dụ, nó chỉ khớp với các địa chỉ được tạo thành từ các ký tự chữ và số kết thúc với một trong những tên miền phổ biến. Vì vậy, nếu trong địa chỉ có một dấu chấm như dưới đây đồng nghĩa với việc ta chỉ tìm được một phần của nó:

```
In [49]:  email.findall('barack.obama@whitehouse.gov')
Out [49]: ['obama@whitehouse.gov']
```

Điều này cho thấy những biểu thức chính quy có thể gây khó khăn như thế nào nếu không cẩn thận! Nếu tìm kiếm trên mạng, bạn có thể tìm thấy một số gợi ý về những biểu thức chính quy khớp với *tất cả* email hợp lệ, nhưng chúng sẽ khó hiểu hơn rất nhiều so với những biểu thức đơn giản được sử dụng ở đây!

Cú pháp biểu thức chính quy cơ bản

Cú pháp của biểu thức chính quy là một chủ đề quá rộng cho phần này. Tuy nhiên, thông thạo một phần nhỏ cũng có thể tạo nên ảnh hưởng lớn: Tôi sẽ giới thiệu một số cấu trúc cơ bản và cung cấp một số nguồn hoàn chỉnh hơn để bạn có thể tìm hiểu thêm. Hy vọng rằng những kiến thức khái quát dưới đây sẽ giúp bạn sử dụng tài nguyên một cách hiệu quả.

Chuỗi đơn giản được khớp trực tiếp

Nếu xây dựng một biểu thức chỉnh quy với một chuỗi những ký tự và chữ số, nó sẽ khớp với chính chuỗi đó:

```
In [50]:  regex = re.compile('ion')
          regex.findall('Great Expectations')
Out [50]: ['ion']
```

Một số ký tự mang ý nghĩa đặc biệt

Trong khi những chữ cái hoặc chữ số đơn giản khớp trực tiếp thì một số ít ký tự mang những ý nghĩa đặc biệt trong biểu thức chính quy. Bao gồm:

```
. ^ $ * + ? { } [ ] \ | ( )
```

Ta sẽ sớm tìm hiểu về ý nghĩa của một số ký tự trong đó. Trước đó, bạn nên biết rằng nếu muốn khớp trực tiếp với những ký tự này, bạn có thể *bỏ qua* ý nghĩa của chúng bằng dấu gạch chéo ngược (\):

```
In [51]:  regex = re.compile(r'\$')
          regex.findall("the cost is $20")
Out [51]: ['$']
```

Chữ `r` đứng đầu trong `r'\$'` biểu thị một *chuỗi thô*, trong những chuỗi Python thông thường, dấu gạch chéo ngược được sử dụng để biểu thị những ký tự đặc biệt. Ví dụ, khoảng trống (tab) được biểu diễn bằng `\t`:

```
In [52]: print('a\tb\tc')
a    b    c
```

Sự thay thế đó không được thực hiện trong một chuỗi thô:

```
In [53]: print(r'a\tb\tc')
a\tb\tc
```

Vì lý do này, khi sử dụng dấu gạch chéo ngược trong một biểu thức chính quy thì nên sử dụng một chuỗi thô.

Ký tự đặc biệt có thể khớp với nhóm ký tự

Trong những biểu thức chính quy, ngoài việc có thể bỏ qua ý nghĩa của những ký tự đặc biệt để biến chúng thành những ký tự thường, thì ký tự `\` còn có thể được sử dụng để trao cho những ký tự thường một ý nghĩa đặc biệt. Những ký tự đặc biệt này khớp với các nhóm ký tự xác định mà ta đã thấy trước đây. Trong đoạn mã lọc địa chỉ email lúc trước, ta đã sử dụng ký tự `\w`, đây là một dấu hiệu đặc biệt khớp với *bất kỳ ký tự chữ và số nào*. Tương tự như vậy, trong ví dụ đơn giản về `split()`, ta cũng đã gặp `\s`, một dấu hiệu đặc biệt biểu thị mọi *ký tự khoảng trắng.*

Kết hợp chúng lại với nhau, ta có thể tạo ra một biểu thức chính quy khớp với *hai chữ cái/chữ số có khoảng trắng nằm giữa:*

```
In [54]:  regex = re.compile(r'\w\s\w')
          regex.findall('the fox is 9 years old')
Out [54]: ['e f', 'x i', 's 9', 's o']
```

Ví dụ này bắt đầu cho thấy sức mạnh và tính linh hoạt của các biểu thức chính quy.

Bảng sau liệt kê một số ký tự hữu ích:

Ký tự	Mô tả
\d	Khớp tất cả ký tự chữ số
\D	Khớp tất cả ký tự không phải chữ số
\s	Khớp tất cả khoảng trắng
\S	Khớp tất cả ký tự không phải khoảng trắng
\w	Khớp tất cả ký tự chữ số
\W	Khớp tất cả ký tự không phải chữ số

Đây không phải là một danh sách hoặc mô tả đầy đủ; để biết thêm chi tiết, hãy xem tài liệu về cú pháp biểu thức chính quy trong Python.

Dấu ngoặc vuông khớp với các nhóm ký tự tùy chỉnh.

Nếu các nhóm ký tự dựng sẵn không đủ cụ thể cho bạn, có thể sử dụng dấu ngoặc vuông để chỉ định bất kỳ bộ ký tự nào mà bạn muốn. Ví dụ: đoạn mã sau sẽ khớp với tất cả nguyên âm viết thường:

```
In [55]:  regex = re.compile('[aeiou]')
          regex.split('consequential')
Out [55]: ['c', 'ns', 'q', '', 'nt', '', 'l']
```

Tương tự, có thể sử dụng dấu gạch ngang để chỉ định một phạm vi: ví dụ, `[a-z]` sẽ khớp với bất kỳ chữ cái viết thường nào và `[1-3]` sẽ khớp với 1 , 2 hoặc 3. Ví dụ, bạn có thể cần trích xuất từ một tài liệu những mã số cụ thể gồm một chữ in hoa và một chữ số đứng sau. Có thể làm như sau:

```
In [56]:  regex = re.compile('[A-Z][0-9]')
          regex.findall('1043879, G2, H6')
Out [56]: ['G2', 'H6']
```

Ký tự đại diện khớp với những ký tự lặp đi lặp lại.

Giả sử bạn muốn lọc ra những chuỗi gồm ba ký tự chữ số liên tiếp, có thể viết: `\w\w\w`. Bởi vì đây là nhu cầu phổ biến nên có một cú pháp để khớp với những ký tự lặp đi lặp lại: ngoặc nhọn chứa số ở bên trong:

```
In [57]:  regex = re.compile(r'\w{3}')
          regex.findall('The quick brown fox')
Out [57]: ['The', 'qui', 'bro', 'fox']
```

Ngoài ra còn có những ký tự đánh dấu khớp với bất kỳ số lần lặp nào. Ví dụ, ký tự + biểu thị những gì đứng trước nó sẽ được lặp lại một hoặc nhiều lần:

```
In [58]:  regex = re.compile(r'\w+')
          regex.findall('The quick brown fox')
Out [58]: ['The', 'quick', 'brown', 'fox']
```

Dưới đây là bảng các dấu hiệu lặp có sẵn để sử dụng trong các biểu thức chính quy:

Ký tự	Mô tả	Ví dụ
?	Thực thể đứng trước khớp 0 hoặc 1 lần	ab? khớp với a hoặc ab
*	Thực thể đứng trước khớp 0 hoặc nhiều lần	ab* khớp với a, ab ,abb, abbb
+	Thực thể đứng trước khớp 1 hoặc nhiều lần	ab+ khớp với ab, abb, abbb... nhưng không phải a
{n}	Thực thể đứng trước khớp n lần	ab{2} khớp với abb

{m,n}	Thực thể đứng trước khớp m tới n lần	ab{2,3} khớp với abb hoặc abbb

Với những kiến thức cơ bản này, hãy quay trở lại bộ lọc email của chúng ta:

```
In [59]: email = re.compile(r'\w+@\w+\.[a-z]{3}')
```

Giờ ta đã hiểu được ý nghĩa của đoạn mã này: ta muốn một hoặc nhiều ký tự chữ số `(\w+)`, tiếp theo là dấu (@), tiếp theo là một hoặc nhiều ký tự chữ số `(\w+)`, tiếp theo là một dấm chấm (\. - Lưu ý sự cần thiết của dấu gạch chéo ngược), tiếp theo là ba chữ cái viết thường.

Nếu muốn sửa đổi đoạn mã này để khớp với địa chỉ email của Obama, ta có thể làm bằng cách sử dụng ký hiệu dấu ngoặc vuông:

```
In [60]:  email2 = re.compile(r'[\w.]+@\w+\.[a-z]{3}')
          email2.findall('barack.obama@whitehouse.gov')
Out [60]: ['barack.obama@whitehouse.gov']
```

Ta đã thay đổi `\w+` thành `[\w.]+`, như vậy nó sẽ khớp với tất cả ký tự chữ số hoặc dấu chấm. Với biểu thức này, nó có thể khớp với nhiều loại địa chỉ email hơn (mặc dù không phải tất cả - liệu bạn có thể chỉ ra thiếu sót của biểu thức này không?)

Dấu ngoặc đơn chỉ ra các *nhóm* cần trích xuất.

Đối với các biểu thức chính quy tổng hợp như biểu thức lọc email trên, ta thường muốn trích xuất các thành phần của chúng hơn là toàn bộ kết quả. Điều này có thể được thực hiện bằng cách sử dụng dấu ngoặc đơn để *nhóm* kết quả:

```
In [61]:  email3 = re.compile(r'([\w.]+)@(\w+)\.([a-
z]{3})')
In [62]:  text = "To email Guido, try guido@python.org"\
                 "or the older address
guido@google.com."
          email3.findall(text)
Out [62]:
[('guido', 'python', 'org'), ('guido', 'google', 'com')]
```

Như ta thấy, việc nhóm này trích xuất ra một danh sách các thành phần con của địa chỉ email.

Ta có thể *đặt tên* cho các thành phần được trích xuất bằng cú pháp `(?P<name>)`, trong trường hợp đó, các nhóm có thể được trích xuất dưới dạng từ điển Python:

```
In [63]:
email4 = re.compile(r'(?P<user>[\w.]+)@(?P<domain>\w+)'\
                    '\.(?P<suffix>[a-z]{3})')
match = email4.match('guido@python.org')
match.groupdict()

Out [63]: {'domain': 'python', 'suffix': 'org', 'user':
'guido'}
```

Kết hợp những ý tưởng trên (cũng như một số cú pháp regexp mạnh mẽ khác chưa đề cập ở đây) cho phép trích một cách xuất linh hoạt và nhanh chóng thông tin từ các chuỗi trong Python.

Một số tài nguyên về biểu thức chính quy

Những kiến thức đã tìm hiểu trên đây chỉ là một phần rất nhỏ của chủ đề rộng lớn này. Nếu muốn tìm hiểu thêm, tôi sẽ đề xuất các tài nguyên sau:

Tài liệu về gói re của Python

> Tôi nhận thấy rằng mình nhanh chóng quên đi cách sử dụng các biểu thức thông thường mỗi lần sử dụng chúng. Bây giờ khi đã có nền tảng vững chắc, tôi nhận ra trang này là một tài nguyên vô cùng quý giá để ôn lại ý nghĩa của từng ký tự hoặc chuỗi cụ thể trong một biểu thức chính quy.

Tài liệu chính thức của Python về biểu thức chính quy (Regular Expression HOWTO)

> Một cách tiếp cận chi tiết hơn về biểu thức chính quy

Làm chủ biểu thức chính quy (Mastering Regular Expressions - O'Reilly, 2006)

> Một cuốn sách hơn 500 trang về chủ đề này. Nếu muốn một tài liệu thực sự hoàn chỉnh về chủ đề này, đây là tài nguyên cho bạn.

Để biết một số ví dụ về thao tác chuỗi và biểu thức chính quy đang hoạt động ở quy mô lớn hơn, tham khảo "Pandas: Định hướng dữ liệu theo cột được gắn nhãn" trang 94, tại đó chúng ta xem xét áp dụng các loại biểu thức này trên các bảng dữ liệu chuỗi trong gói Pandas.

Một cái nhìn tổng quát về những công cụ khoa học dữ liệu

Nếu muốn có một bước đệm và đi xa hơn trong việc sử dụng Python trong tính toán khoa học và khoa học dữ liệu, có một số gói sẽ làm điều đó trở nên dễ dàng hơn. Phần này sẽ giới thiệu, đưa ra cái nhìn tổng quát về một vài công cụ quan trọng hơn và cho bạn một khái niệm về tính ứng dụng của chúng. Nếu đang sử dụng môi trường Anaconda hoặc Miniconda đã được đề xuất từ đầu cuốn sách này, bạn có thể cài đặt những gói liên quan bằng dòng lệnh dưới đây:

```
$ conda install numpy scipy pandas matplotlib scikit-
learn
```

Hãy lần lượt điểm qua từng công cụ trong số đó.

NumPy: Số học Python

NumPy cung cấp một cách hiệu quả để lưu trữ và thao tác các mảng dày đặc đa chiều trong Python. Các tính năng quan trọng của NumPy là:

Nó cung cấp một cấu trúc `ndarray` , cho phép lưu trữ và thao tác hiệu quả các véc-tơ, ma trận và các bộ dữ liệu chiều cao hơn.

Nó cung cấp một cú pháp dễ đọc và hiệu quả cho việc vận hành trên dữ liệu này, từ số học phần tử đơn giản cho đến các phép toán đại số tuyến tính phức tạp hơn.

Trong trường hợp đơn giản nhất, mảng NumPy trông rất giống danh sách Python. Ví dụ, đây là một mảng chứa số trong phạm vi từ 1 đến 9 (so sánh với hàm tích hợp `range()` của Python):

```
In [1]:  import numpy as np
         x = np.arange(1, 10)
         x
Out [1]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Mảng trong NumPy cung cấp cả kho lưu trữ dữ liệu hiệu quả, cũng như các phép tính hiệu quả trên dữ liệu. Ví dụ, để bình phương từng phần tử của mảng, ta có thể áp dụng toán tử ** trực tiếp vào mảng:

```
In [2]:  x ** 2
Out [2]: array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])
```

So sánh điều này với danh sách tổng quát kiểu Python rườm rà hơn cho cùng một kết quả:

```
In [3]:  [val ** 2 for val in range(1, 10)]
Out [3]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Không giống như danh sách Python (bị giới hạn ở một chiều), mảng NumPy có thể là mảng đa chiều. Ví dụ, ở đây chúng ta sẽ định hình lại mảng x thành một mảng 3x3:

```
In [4]:  M = x.reshape((3, 3))
         M
Out [4]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

Mảng hai chiều là một đại diện của ma trận và NumPy biết cách thực hiện hiệu quả các thao tác điển hình trên ma trận. Ví dụ: bạn có thể tính ma trận chuyển vị bằng cách sử dụng `.T`:

```
In [5]:  M.T
Out [5]: array([[1, 4, 7],
                [2, 5, 8],
                [3, 6, 9]])
```

hoặc tích vector ma trận bằng np.dot:

```
In [6]:  np.dot(M, [5, 6, 7])
Out [6]: array([ 38,  92, 146])
```

và thậm chí cả những phép tính phức tạp hơn như phân tách trị riêng:

```
In [7]: np.linalg.eigvals(M)
Out [7]: array([  1.61168440e+01,  -1.11684397e+00,  -
1.30367773e-15])
```

Thao tác đại số tuyến tính như vậy làm cơ sở cho nhiều phân tích dữ liệu hiện đại, đặc biệt khi nói đến các lĩnh vực học máy và khai thác dữ liệu.

Để biết thêm thông tin về NumPy, xem “Tài nguyên cho việc học thêm” trang 98.

Pandas: Dữ liệu định hướng theo cột được gắn nhãn

Pandas là một gói mới hơn nhiều so với NumPy, và trên thực tế nó được xây dựng dựa trên đó. Những gì Pandas cung cấp là một giao diện được gắn nhãn cho dữ liệu đa chiều, dưới dạng đối tượng DataFrame sẽ tạo cảm giác quen thuốc với người dùng R và các ngôn ngữ liên quan. DataFrames trong Pandas trông giống như thế này:

```
In [8]:
import pandas as pd
df = pd.DataFrame({'label': ['A', 'B', 'C', 'A', 'B',
'C'],
                   'value': [1, 2, 3, 4, 5, 6]})
df
Out [8]:    label value
          0  A     1
          1  B     2
          2  C     3
          3  A     4
          4  B     5
          5  C     6
```

Giao diện Pandas cho phép bạn thực hiện những việc như chọn cột theo tên:

```
In [9]:  df['label']
Out [9]: 0    A
         1    B
         2    C
         3    A
         4    B
         5    C
         Name: label, dtype: object
```

Áp dụng hoạt động chuỗi lên các mục của chuỗi:

```
In [10]:  df['label'].str.lower()

Out [10]: 0    a
          1    b
          2    c
```

```
3    a
4    b
5    c
Name: label, dtype: object
```

Tính tổng với các mục số học:

```
In [11]:  df['value'].sum()
Out [11]: 21
```

Và, có lẽ quan trọng nhất, thực hiện các phép nối và nhóm kiểu cơ sở dữ liệu một cách hiệu quả:

```
In [12]:  df.groupby('label').sum()
Out [12]:       value
          Label
          A     5
          B     7
          C     9
```

Trong một dòng, ta đã tính tổng của tất cả các đối tượng cùng chia sẻ một nhãn, dài dòng hơn (và kém hiệu quả hơn nhiều) so với cách sử dụng các công cụ được cung cấp trong NumPy và Python.

Để biết thêm thông tin về việc sử dụng Pandas, hãy xem các tài nguyên được liệt kê trong “Tài nguyên cho việc học thêm” trang 98.

Matplotlib: khoa học trực quan theo phong cách MATLAB

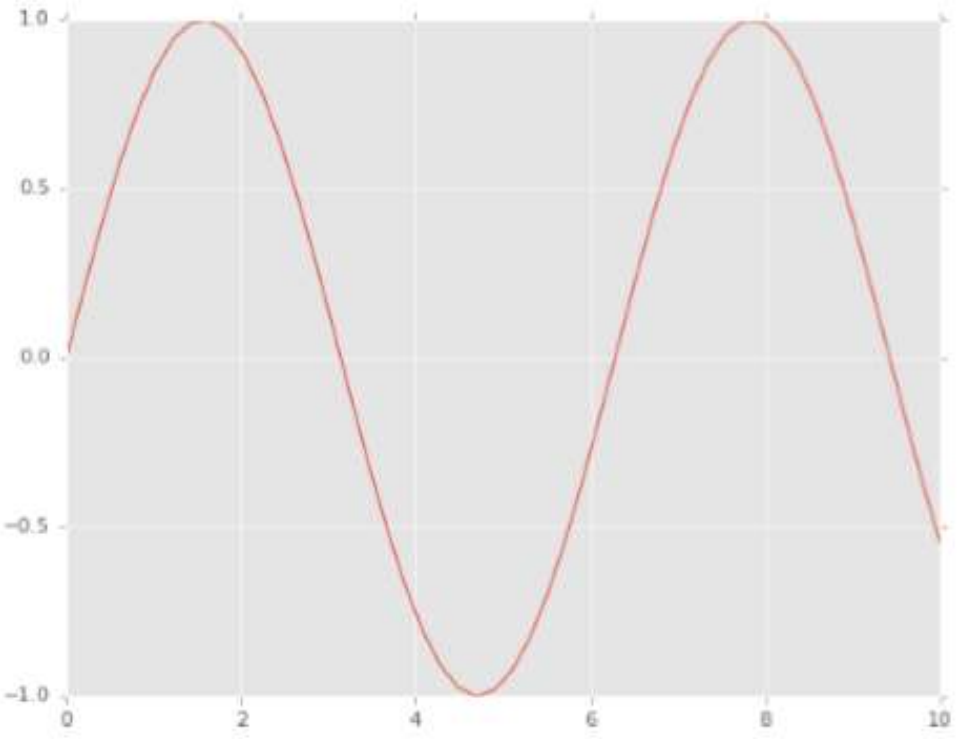
Matplotlib hiện là gói khoa học trực quan phổ biến nhất trong Python. Ngay cả những người đề xướng cũng thừa nhận rằng giao diện của nó đôi khi quá rườm rà, nhưng nó là một thư viện mạnh mẽ để tạo ra một loạt biểu đồ.

Để sử dụng Matplotlib, ta có thể bắt đầu bằng cách bật chế độ sổ ghi chép (để sử dụng trong Jupyter notebook) và sau đó nhập gói vào dưới tên `plt`:

```
In [13]: # khởi chạy nếu sử dụng Jupyter notebook
         %matplotlib notebook
In [14]:
import matplotlib.pyplot as plt
plt.style.use('ggplot')   # tạo biểu đồ theo kiểu R trong
ggplot
```

Bây giờ, hãy để cùng nhau tạo ra một số dữ liệu (tất nhiên là mảng NumPy) và vẽ kết quả:

```
In [15]: x = np.linspace(0, 10)   # phạm vi giá trị từ  0
đến 10
         y = np.sin(x)            # sin của những giá trị
         plt.plot(x, y);          # biểu đồ đường
```



Nếu chạy đoạn mã này trực tiếp, bạn sẽ thấy một biểu đồ có thể tương tác được cho phép bạn xoay, thu phóng và cuộn để khám phá dữ liệu.

Đây là ví dụ đơn giản nhất về biểu đồ Matplotlib; để biết về phạm vi rộng lớn của các loại biểu đồ có sẵn, hãy xem trong thư viện trực tuyến Matplotlib, cũng như các tài liệu tham khảo khác được liệt kê trong “Tài nguyên cho việc học thêm” trang 98.

SciPy: Python khoa học

SciPy là một tập hợp các hàm khoa học được xây dựng trên NumPy. Ban đầu nó như một tập hợp các trình bao bọc Python được biết đến là thư viện Fortran nổi tiếng dùng để tính toán số học và đã phát triển từ đó. Gói được sắp xếp như một tập hợp các mô-đun con, mỗi mô-đun thực hiện một số lớp thuật toán số học. Đây là một ví dụ không đầy đủ về một số mô-đun quan trong khoa học dữ liệu:

scipy.fftpack	Biến đổi Fourier nhanh
scipy.integrate	Thuật toán số học
scipy.interpolate	Nội suy số học
scipy.linalg	Hàm đại số tuyến tính
scipy.opt	Tối ưu hóa số lượng các chức năng
scipy.spzzy	Lưu trữ ma trận thưa và đại số tuyến tính
scipy.stats	Hàm phân tích thống kê

Ví dụ: hãy xét nội suy đường cong trơn của một số dữ liệu:

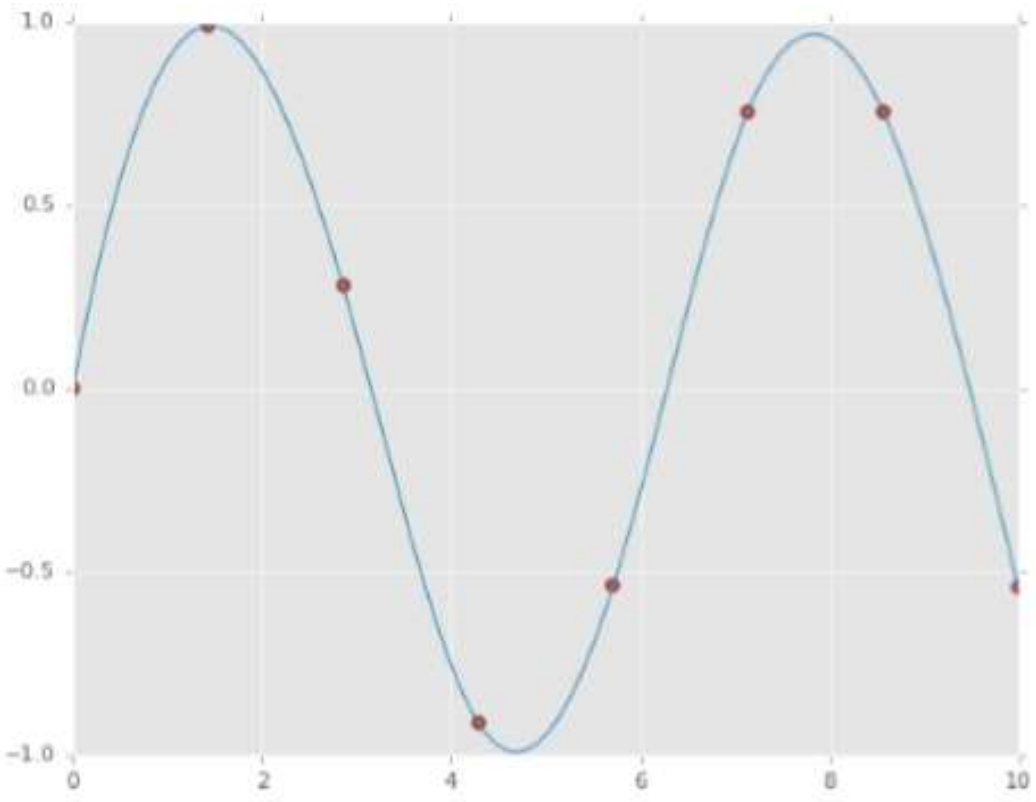
```
In [16]: from scipy import interpolate

         # chọn 8 điểm trong khoảng giữa 0 và 10
         x = np.linspace(0, 10, 8)
         y = np.sin(x)

         # tạo một hàm nội suy bậc ba
         func = interpolate.interp1d(x, y, kind='cubic')

         # nội suy trên một lưới với 1.000 điểm
         x_interp = np.linspace(0, 10, 1000)
         y_interp = func(x_interp)

         # vẽ đồ thị từ kết quả
         plt.figure()  # đồ thị mới
         plt.plot(x, y, 'o')
         plt.plot(x_interp, y_interp);
```



Những gì chúng ta thấy là một nội suy trơn giữa các điểm.

Các gói khoa học dữ liệu khác

Được xây dựng dựa trên các công cụ này là một loạt các gói khoa học dữ liệu khác, bao gồm các công cụ chung như Scikit-Learn cho học máy, Scikit-Image để phân tích hình ảnh và StatsModels cho mô hình thống kê, cũng như các gói dành riêng cho miền như AstroPy cho thiên văn học và vật lý thiên văn, NiPy cho hình ảnh thần kinh, và nhiều hơn nữa.

Dù có gặp phải loại vấn đề khoa học, số học hay thống kê, luôn có thể có một gói Python ngoài kia có thể giúp bạn giải quyết nó.

Tài nguyên cho việc học thêm

Phần này sẽ kết thúc chuyến tham quan của chúng ta với ngôn ngữ Python. Tôi hy vọng là nếu bạn đọc đến đây, bạn đã có một khái niệm về cú pháp, ngữ nghĩa, hoạt động và chức năng thiết yếu được cung cấp bởi ngôn ngữ Python, cũng như một số khái niệm về phạm vi của công cụ và cấu trúc mã mà bạn có thể khám phá thêm.

Tôi đã cố gắng bao quát các phần và cú pháp trong ngôn ngữ Python sẽ hữu ích cho một nhà khoa học dữ liệu sử dụng Python, nhưng đây không phải là một

hướng dẫn hoàn chỉnh. Nếu bạn muốn tìm hiểu sâu hơn về ngôn ngữ Python và cách sử dụng nó một cách hiệu quả, thì đây là một số tài nguyên mà tôi đề xuất:

Thông thạo Python (Fluent Python) của Luciano Ramalho

> Đây là một cuốn sách O 'Reilly tuyệt vời tìm hiểu các thói quen và phong cách tốt nhất cho Python, bao gồm cả việc tận dụng tối đa thư viện chuẩn.

Bắt đầu với Python (Dive Into Python) của Mark Pilgrim

> Đây là một cuốn sách trực tuyến miễn phí cung cấp hướng dẫn cơ bản về ngôn ngữ Python.

Khổ luyện Python (Learn Python the Hard Way) của Zed Shaw

> Cuốn sách này tiếp cận theo cách "học từ thực tiễn" và tập trung vào việc phát triển thứ có thể là kỹ năng hữu ích nhất của một lập trình viên: Tìm kiếm mọi thứ mà bạn không biết bằng Google.

Tài liệu tham khảo thiết yếu về Python (Python Essential Reference) của David Beazley

> Con quái vật 700 trang này được viết rất tốt và bao gồm hầu hết mọi thứ cần biết về ngôn ngữ Python và các thư viện tích hợp của nó. Để có một hướng dẫn tập trung vào ứng dụng Python, hãy tìm đọc *Sách công thức Python (Python Cookbook)* của Beazley .

Để tìm hiểu thêm về các công cụ Python cho khoa học dữ liệu và tính toán khoa học , tôi khuyên bạn nên đọc các cuốn sách sau:

Cẩm nang khoa học dữ liệu Python (The Python Data Science Handbook) của chính tôi

> Cuốn sách này bắt đầu chính xác tại nơi công trình này dừng lại, và cung cấp một hướng dẫn toàn diện các công cụ thiết yếu trong hằng hà sa số các công cụ khoa học dữ liệu của Python, từ việc khai thác và điều khiển dữ liệu cho đến học máy.

Tính toán hiệu quả trong Vật lý (Effective Computation in Physics) của Katie Huff và Anthony Scopatz

> Cuốn sách này có thể áp dụng cho những người không ở trong thế giới nghiên cứu vật lý. Đây là hướng dẫn từng bước, cơ bản về điện toán khoa

học, bao gồm phần giới thiệu tuyệt vời về nhiều công cụ được đề cập trong cuốn sách này.

Phân tích dữ liệu với Python (Python for Data Analysis) của Wes McKinney, cha để của gói Pandas

Cuốn sách này nói chi tiết về thư viện Pandas, cũng như cung cấp thông tin hữu ích về một số công cụ khác cho phép nó.

Cuối cùng, để có cái nhìn rộng hơn vượt ra khỏi cuốn sách này, tôi đề xuất những thứ sau:

Tài nguyên Python của O'Reilly

O 'Reilly có một số cuốn sách tuyệt vời về Python và các chủ đề chuyên ngành trong thế giới Python.

PyCon, SciPy và PyData

Các hội nghị PyCon, SciPy và PyData thu hút hàng ngàn người tham dự mỗi năm và lưu trữ phần lớn các chương trình của họ mỗi năm dưới dạng video trực tuyến miễn phí. Những thứ này đã biến thành một bộ tài nguyên đáng kinh ngạc để tìm hiểu về Python, các gói Python và các chủ đề liên quan. Hãy tìm kiếm video của cả bài thuyết trình và bài hướng dẫn: các bài thuyết trình thường ngắn hơn, điểm qua các gói mới hoặc giao diện mới cho các gói cũ. Các hướng dẫn có thể lên tới vài giờ, bao gồm việc sử dụng các công cụ được đề cập ở đây cũng như những công cụ khác.

Giới thiệu về tác giả

Jake VanderPlas là một người dùng và nhà phát triển lâu năm của khoa học Python. Ông hiện đang làm giám đốc nghiên cứu liên ngành tại Đại học Washington, thực hiện dự án nghiên cứu thiên văn học của riêng mình và dành thời gian tư vấn cho các nhà khoa học địa phương từ nhiều lĩnh vực.

Đôi lời cùng bạn đọc

Bản dịch này được cung cấp miễn phí tại **trituenhantao.io** là trang web chia sẻ thông tin, kiến thức, kinh nghiệm học tập và triển khai các chương trình và dự án sử dụng trí tuệ nhân tạo trên thế giới. Chúng tôi cố gắng hết mình trong việc truyền tải các kiến thức dưới dạng đơn giản và thực tế nhất, với ngôn ngữ dễ hiểu nhất đối với tất cả mọi người. Chúng tôi vui mừng giới thiệu đến bạn cuốn sách ***"Dạo một vòng Python"*** của tác giả **Jake VanderPlas**. Hi vọng sau quyển sách này các bạn có thể tự trang bị cho mình những kiến thức và kỹ năng cần thiết để sử dụng thành tạo ngôn ngữ đầy quyền năng – Python. Đối với những bạn quan tâm đến Trí tuệ nhân tạo, quyển sách này là bàn đạp để bạn có thể lĩnh hội tốt hơn những chủ đề chuyên sâu của lĩnh vực này. Hãy thường xuyên truy cập và ủng hộ website của chúng tôi https://trituenhantao.io để chúng ta có những câu chuyện thú vị hơn các bạn nhé!