Tài liệu hướng dẫn nhanh và ngắn gọn về sử dụng ngôn ngữ Python

Norman Matloff
University of California, Davis

26 tháng 1 năm 2006

Mục lục

1 Khái quát chung

1.1 Các ngôn ngữ script là gì

Các ngôn ngữ lập trình như C và C++ cho phép lập trình viên viết mã lệnh ở mức độ rất chi tiết và mang lại tốc độ thực hiện nhanh chóng. Tuy nhiên trong hầu hết các ứng dụng thực tiễn, tốc độ thực hiện không phải là yếu tố quan trọng, và trong nhiều trường hợp, người sử dụng sẵn sàng viết mã lệnh bằng một ngôn ngữ cấp cao hơn. Chẳng hạn, với các chương trình xử lí file văn bản, đơn vị cơ bản trong C/C++ là kí tự, còn với những ngôn ngữ như Perl và Python thì đó là các dòng văn bản và các từ trong mỗi dòng. Dĩ nhiên là có thể xử lí các dòng văn bản và các từ trong C/C++, nhưng ta cần cố gắng nhiều hơn nếu muốn đạt được một kết quả tương tự.

Thực ra thuật ngữ "scripting language" (ngôn ngữ *kịch bản*) chưa từng được định nghĩa chính thức, nhưng sau đây là một số đặc điểm của nó:

- được sử dụng thường xuyên trong lĩnh vực quản trị hệ thống, lập trình Web và "mô hình hoá hệ thống" và tinh chỉnh phần mềm theo yêu cầu người sử dụng ngay trong quá trình sản xuất phần mềm.
- khai báo biến một cách tự nhiên (với các biến nguyên, dấu phẩy động và chuỗi kí tự thường rất ít hoặc không có sự khác biệt). Các mảng có thể trộn lẫn các kiểu biến khác nhau, chẳng hạn kiểu nguyên và kiểu chuỗi kí tự. Các hàm có thể trả giá trị kiểu mảng (thay vì scalar). Các kiểu mảng này có thể dùng làm chỉ số đếm trong các vòng lặp v.v...
- nhiều các phép tính cấp cao được xây dựng sẵn trong ngôn ngữ, chẳng hạn kết nối chuỗi kí tự và push/pop các *stack*.
- được thông dịch thay vì biên dịch thành các ngôn ngữ máy.

1.2 Tại sao nên dùng Python?

Ngày nay ngôn ngữ kịch bản phổ biến nhất có lẽ là Perl. Tuy vậy, vẫn có nhiều người, trong đó có tác giả, ưa chuộng Python hơn vì ngôn ngữ này rõ ràng và tinh tế hơn. Đối với những người phát triển hệ thống Google thì Python là ngôn ngữ rất quen thuộc.

Những người ủng hộ Python, thường được gọi là các *Pythonista*, cho rằng ngôn ngữ này trong sáng và tiện dụng đến mức ta có thể dùng nó cho mọi khâu lập trình chứ không riêng gì viết *script*. Họ tin rằng Python hay hơn C và C++.[1] Cá nhân tôi cho rằng người ta đề cao C++; một số thành phần của ngôn ngữ này không khớp với nhau. Java là ngôn ngữ hay hơn, nhưng nó yêu cầu mỗi biến phải có một kiểu nhất định. Đặc điểm này, theo tôi, đã tăng độ phức tạp trong việc lập trình. Tôi rất vui khi thấy Eric Raymond, một người nổi tiếng trong giới phát triển phần mềm mã nguồn mở cũng khẳng định những điều tương tự về C++, Java và Python.

[1] Một ngoại lệ cần nhắc lại là ta không xét đến chương trình yêu cầu tốc độ cao.

2 Hướng dẫn sử dụng tài liệu

2.1 Kiến thức cơ bản cần có

Bất kì ai với một chút kinh nghiệm lập trình đều có thể tiếp cận được những nội dung được trình bày trong tài liệu.

Tài liệu mở đầu với phần *Hướng đối tượng* 7, thuận tiện cho người có kiến thức cơ bản về ngôn ngữ lập trình hướng đối tượng như C++ hoặc Java. Nếu thiếu phần kiến thức này, bạn vẫn có thể đọc các phần này, mặc dù có thể phải chậm hơn so với những người biết C++ hoặc Java; bạn chỉ cần nắm chắc những ví dụ thay vì các thuật ngữ.

Sẽ có một đôi chỗ tôi sẽ trình bày riêng nếu máy bạn sử dụng hệ điều hành Unix. Nhưng thực tế kiến thức Unix cũng không cần thiết vì bạn có thể dùng Python trên cả Windows và Macintosh chứ không riêng gì Unix.

2.2 Hướng tiếp cận

Hướng tiếp cận của ta ở đây tương đối khác so với phần lớn các sách (hay các trang web hướng dẫn) về Python. Cách tiếp cận thông thường là trình bày từng chi tiết từ đầu đến cuối. Chẳng hạn như việc liệt kê hết tất cả các giá trị của một tham số trong một câu lệnh Python.

Trong tài liệu tôi tránh dùng cách này. Một lần nữa, mục tiêu là cho phép người đọc có thể nhanh chóng nắm bắt được nền tảng của Python, từ đó có thể tìm hiểu sâu vào một vấn đề cụ thể theo nhu cầu.

2.3 Những phần nào cần đọc và đọc khi nào

Theo tôi, bạn nên bắt đầu đọc phần *key*, sau đó sử dụng thử Python. Trước hết thử nghiệm thao tác với dấu nhắc lệnh Python (Phần *dấu nhắc lệnh*). Sau đó thì tự tay viết một số chương trình ngắn; đó có thể là các chương trình hoàn toàn mới, hoặc chỉ thay đổi một chút từ những chương trình trong các phần tiếp theo của tài liệu. [2]

Điều này sẽ giúp bạn hiểu cách dùng ngôn ngữ một cách cụ thể hơn. Nếu mục đích chính của bạn khi sử dụng Python là viết những đoạn mã ngắn mà không cần sử dụng thư viện Python, Như vậy có lẽ cũng đủ. Tuy vậy, phần lớn người đọc cần nghiên cứu thêm với kiểu thức cơ bản về các đặc điểm lập trình hướng đối tượng và các *mô-đun/gói* chương trình Python. Vì vậy tiếp theo bạn nên đọc Phần 12

Đó là nền tảng rất chắc để bạn sử dụng tốt Python. Cuối cùng, bạn có thể sẽ nhận thấy rằng nhiều lập trình viên Python sử dụng những đặc điểm lập trình theo hàm của Python, và bạn cũng muốn hiểu chương trình của người khác hoặc có thể muốn chính mình sử dụng đặc điểm này. Nếu vậy, phần 14 chính là nơi bạn bắt đầu.

Đừng quên các phụ lục! Những phụ lục quan trọng là A và A.3.

[2] File nguồn của chương trình có thể download tại địa chỉ `http://heather.cs.ucdavis.edu/~matloff/Python/PythonIntro.tex`, do đó bạn không cần phải tự tay gõ chương trình. Bạn có thể soạn thảo một bản sao của file này, lưu lại những dòng lệnh của chương trình mà bạn cần, hoặc dùng chuột thực hiện copy-paste đối với những hướng dẫn có liên quan.

Nhưng nếu bạn muốn tự tay gõ nội dung những chương trình ví dụ này thì cần đảm bảo gõ chính xác những gì có trong tài liệu, đặc biệt là các chỗ thụt đầu dòng. Điều này rất quan trọng, sau này ta sẽ bàn đến.

Tôi cũng có một số tài liệu hướng dẫn lập trình Python theo mục đích cụ thể chẳng hạn: lập trình mạng, kiểu lặp / chuỗi số phát sinh... Xem thêm `http://heather.cs.ucdavis.edu/~matloff/python.html`.

3 Một ví dụ hướng dẫn trong 5 phút

3.1 Mã lệnh của chương trình ví dụ

Dưới đây là một đoạn chương trình ngắn, đơn giản. Giả sử ta cần tính giá trị của hàm

$$g(x) = \frac{x}{1-x^2}$$

ứng với x = 0.1, 0.2, ..., 0.9. Để thực hiện điều này, có thể dùng đoạn chương trình như sau:

```
for i in range(10):
   x = 0.1*i
   print x
   print x/(1-x*x)
```

lưu vào một file, chẳng hạn **fme.py**, sau đó chạy chương trình bằng cách gõ lệnh sau tại dấu nhắc hệ thống (trong Windows, bạn có thể double-click vào tên file)

```
python fme.py
```

Kết quả cho ra có dạng như sau:

```
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
0.5
0.666666666667
0.6
0.9375
0.7
1.37254901961
0.8
2.22222222222
0.9
4.73684210526
```

3.2 Kiểu dữ liệu dạng danh sách (list)

Chương trình nêu trên hoạt động như thế nào? Trước hết, hàm **range()** của Python chính là một ví dụ về một *list* (mảng 1 chiều),[3] mặc dù về hình thức không rõ ràng lắm. Với Python, *list* đóng vai trò cơ bản, cho nên khi gặp phải từ "list" ở tài liệu này, bạn nên hiểu nó là cấu trúc dữ liệu kiểu như mảng thay vì một định nghĩa từ vựng trong Tiếng Anh.

Hàm **range()** của Python trả lại kết quả một list gồm các số nguyên liên tiếp, trong trường hợp ví dụ nêu trên l [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. Chú ý rằng đây là cách viết chính thức của list trong Python. List gồm một chuỗi các đối tượng (có thể là đủ mọi kiểu chứ không riêng chỉ kiểu số, được phân cách bởi các dấu phẩy giữa cặp ngoặc vuông.

Như vậy, câu lệnh **for** trong ví dụ kể trên tương tự với:

```
for i in [0,1,2,3,4,5,6,7,8,9]:
```

Chắc bạn cũng đoán được, lệnh này thực hiện mỗi vòng lặp 10 lần, lần đầu tiên i = 0, sau đó là i = 1,...

Python cũng có cấu trúc lặp **while** (cho dù không có**until**). Ngoài ra, lệnh **break** giống như trong C/C++ cho phép sớm thoát khỏi vòng lặp. Chẳng hạn:

```
>>> x = 5
>>> while 1:
...     x += 1
...     if x == 8:
...         print x
...         break
...
8
```

3.3 Khối lệnh trong Python

Bây giờ hãy chú ý đến dấu hai chấm tưởng như vô dụng ở cuối dòng lệnh **for**, nơi bắt đầu của mỗi khối lệnh. Khác với các ngôn ngữ kiểu như C/C++ và ngay cả Perl đều sử dụng cặp ngoặc nhọn để giới hạn khối lệnh, Python sử dụng dấu hai chấm và cách viết thụt đầu dòng tiếp theo để làm việc này. Giờ tôi sẽ dùng dấu hai chấm để thông tin đến bộ dịch lệnh Python,

> Chào bộ dịch lệnh Python, bạn khoẻ chứ? Tôi muốn báo cho bạn biến rằng, bằng cách thêm vào dấu hai chấm này, một khối lệnh mới được bắt đầu ở dòng tiếp theo. Tôi thụt đầu dòng đó cũng như hai dòng tiếp theo, để báo cho bạn biến rằng ba dòng này hợp thành một khối lệnh.

Trong tài liệu này tôi viết thụt dòng một khoảng cách bằng 3 chữ cái (không nhất thiết là 3, miễn là thống nhất). Chẳng hạn, nếu tôi phải viết[4]

[3] Ở đây có thể tạm gọi là "mảng", tuy nhiên sau này bạn sẽ thấy chúng còn linh hoạt hơn kiểu dữ liệu tương tự của các ngôn ngữ C/C++.

[4] Ở đây **g()** là một hàm đã định nghĩa như ở ví dụ trên.

```
for i in range(10):
   print 0.1*i
      print g(0.1*i)
```

thì bộ dịch lệnh Python sẽ thông báo lỗi, nói rằng tôi mắc lỗi cú pháp.[5] Chúng ta chỉ được phép thụt cột thêm một lần nữa nếu có một khối lệnh con thuộc khối lệnh khác, chẳng hạn:

```
for i in range(10):
   if i%2 == 1:
      print 0.1*i
      print g(0.1*i)
```

Ở đây tôi chỉ in ra những gì tương ứng với i là số lẻ của % là toán tử"mod", giống như trong C/C++.[6]

Bên cạnh đó, cần chú ý những câu lệnh Python kiểu như **print a or b or c**, nếu biểu thức a đúng (tức là khác 0) thì chỉ mình nó được in ra còn b và c thì không; điều này thường thấy ở Python. Cũng cần nhắc lại, chú ý dấu hai chấm ở cuối lệnh **if**, và hai dòng lệnh **print** được viết thụt cột so với dòng lệnh **if.**

Bên cạnh đó cũng cần chú ý rằng khác với C/C++/Perl, không có dấu hai chấm ở cuối các câu lệnh bình thường của Python. Mỗi dòng là một câu lệnh mới. Nếu bạn cần viết một dòng dài, có thể dùng dấu sổ ngược như dưới đây:

```
x = y + \
        z
```

3.4 Python cũng có chế độ gõ lệnh tương tác

Một đặc điểm rất hay của Python là khả năng hoạt động với chế độ tương tác (dòng lệnh). Thường thì bạn cũng không dùng nó khi lập trình, nhưng đó là một cách tiện lợi, nhanh chòng thử mã lệnh để xem nó hoạt động ra sao. Khi bạn không chắc về tác dụng của một thứ gì đó có thể bạn sẽ nói "Phải thử nó xem sao!", và chế độ tương tác dòng lệnh cho phép làm việc này nhanh chóng, dễ dàng.

Trong tài liệu này, chúng ta cũng sử dụng kiểu tương tác dòng lệnh này như một cách minh hoạ nhanh chóng cho một đặc điểm của Python.

Thay vì chạy chương trình từ dấu nhắc hệ thống (batch mode - chạy toàn bộ chương trình một lần), ta có thể gõ lệnh để máy thực hiện dưới chế độ tương tác:

```
% python
>>> for i in range(10):
...     x = 0.1*i
...     print x
...     print x/(1-x*x)
```

[5] Hãy ghi nhớ: Những người mới bắt đầu sử dụng Python thường bị mắc lỗi kiểu như thế này.

[6] Hầu hết các toán tử thường dùng của C đều có trong Python, kể cả toán tử so sánh bằng (==) như bạn thấy. Còn kí hiệu **0x** dành cho hệ số 16, và cũng giống như FORTRAN, kí hiệu ****** dành cho hàm mũ. Bên cạnh đó lệnh **if**có thể kèm theo **else** như thông thường, và bạn có thể viết gọn **else if** thành **elif**. Các toán tử Boolean gồm có **and**, **or** và **not**.

```
...
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
0.5
0.666666666667
0.6
0.9375
0.7
1.37254901961
0.8
2.22222222222
0.9
4.73684210526
>>>
```

Ở đây tôi khởi động Python, dấu nhắc tương tác >>> xuất hiện. Sau đó tôi chỉ việc gõ các lệnh vào, từng dòng một. Bất cứ lúc nào khi tôi ở trong một khối lệnh thì dấu nhắc có dạng đặc biệt, "...". Và khi tôi gõ một dòng trống ở cuối mã lệnh thì máy hiểu rằng tôi nhập xong các dòng lệnh và bắt đầu chạy.[7]

Trong khi ở chế độ tương tác lệnh, bạn có thể tìm các câu lệnh được gõ vào lần trước bằng các phím mũi tên lên xuống, không cần gõ lại. [8]

Để thoát khỏi chế độ dòng lệnh, ấn Ctrl-D (Ctrl-Z trong Windows).

In tự động: Trong chế độ tương tác dòng lệnh, khi ta tham chiếu hoặc tạo mới một đối tượng, hoặc ngay cả một biểu thức mà chưa cần gán tên biến cho nó thì giá trị của biểu thức sẽ được in ra (không cần gõ lệnh **print**). Chẳng hạn:

```
>>> for i in range(4):
...     3*i
...
0
3
6
9
```

[7]Chế độ tương tác lệnh cho phép chúng ta thực hiện từng dòng lệnh Python và tính toán giá trị từng biểu thức đơn lẻ. Ở đây, chúng ta gõ vào và thực hiện một lệnh lặp **for**. Chế độ tương tác lệnh không cho phép chúng ta gõ vào cả một chương trình. Về mặt kĩ thuật, thực ra ta vẫn có thể thực hiện điều này bằng cách bắt đầu bằng một dòng lệnh kiểu như "if 1:", đưa cả chương trình vào một câu lệnh **if** lớn, nhưng dù sao thì đây cũng không phải là cách thuận tiện để gõ nội dung của một chương trình lớn.

[8]Với Pythonwin là Ctrl+mũi tên.

Một lần nữa, chú ý là điều này cũng đúng với đối tượng nói chung, không chỉ với biểu thức, ví dụ

```
>>> open('x')
<open file 'x', mode 'r' at 0x401a1aa0>
```

Ở đây ta mở file **x**, tức là tạo mới một đối tượng kiểu file. Vì chúng ta chưa gán tên biến (chẳng hạn **f**) cho nó, `f=open('x')` (để sau này còn gọi đến), đối tượng file được in ra.

3.5 Dùng Python như một máy tính bỏ túi

Điều này có nghĩa là ngoài các công dụng khác ra, Python có thể được sử dụng như một máy tính tiện dụng (mà tôi cũng thường dùng như vậy). Chẳng hạn để tính 105% của số 88.88 là bao nhiêu, tôi có thể gõ

```
% python
>>> 1.05*88.88
93.323999999999998
```

Ta có thể chuyển đổi nhanh chóng giữa hệ thập phân và hệ đếm 16:

```
>>> 0x12
18
>>> hex(18)
'0x12'
```

Nếu cần các hàm toán học, trước hết ta phải nhập (import) thư viện toán của Python. điều này giống với khai báo #include của C/C++ trong mã nguồn để kết nối thư viện với mã máy. Sau đó chúng ta cần gọi tên các hàm theo sau tên của thư viện. Chẳng hạn, các hàm **sqrt()** and **sin()** phải được dẫn trước bởi **math**:[9]

```
>>> import math
>>> math.sqrt(88)
9.3808315196468595
>>> math.sin(2.5)
0.59847214410395655
```

4 Một ví dụ hướng dẫn trong 10 phút

4.1 Mã lệnh của ví dụ

Chương trình này đọc vào 1 file văn bản mà tên file được nhập vào dấu nhắc lệnh, sau đó in ra số dòng và số từ trong file đó:

[9] Một phương pháp tránh viết từ **math** đứng trước sẽ được trình bày trong phần 10.1.2.

```
# đọc vào file text có tên trong tham số dòng lệnh,
# và báo cáo số dòng và số từ trong file

import sys

def checkline():
   global l
   global wordcount
   w = l.split()
   wordcount += len(w)

wordcount = 0
f = open(sys.argv[1])
flines = f.readlines()
linecount = len(flines)
for l in flines:
   checkline()
print linecount, wordcount
```

Chẳng hạn, file chương trình có tên là **tme.py**, và ta có một file text **x** với nội dung:

```
This is an
example of a
text file.
```

(File này có 5 dòng, trong đó dòng đầu và dòng cuối đều trống.)

Nếu chạy chương trình với file nói trên, ta nhận được kết quả:

```
python tme.py x
5 8
```

Nhìn bề ngoài, dạng mã lệnh trông giống như của C/C++. Trước tiên là lệnh **import**, tương tự với **#include** (với sự liên kết tương ứng lúc biên dịch), như đã nói ở trên. Tiếp theo là định nghĩa một hàm, sau đó là phần chương trình chính. Về cơ bản, đây là một cách nhìn dễ hiểu nhưng hãy ghi nhớ rằng bộ dịch lệnh Python sẽ xử lí các câu lệnh theo thứ tự, bắt đầu từ đầu chương trình. Chẳng hạn, khi dịch lệnh **import**, có thể sẽ kèm theo việc thực hiện một số các câu lệnh khác, nếu như mô-đun được *import* có một số câu lệnh tự do trong đó (điều này sẽ đề cập sau). Việc xử lí câu lệnh **def** không thực hiện lệnh gì ngay, thay vào đó chỉ là khai báo hàm có thể được thực hiện.

Với ví dụ thứ hai này, ta có thể thêm một số đặc điểm mới so với ví dụ đầu tiên:

- sử dụng các tham số dòng lệnh
- cơ chế xử lí file

- thêm về các list
- định nghĩa hàm
- nhập thư viện
- Phạm vi của biến

Các đặc điểm này được giới thiệu trong các phần tiếp theo:

Các tham số dòng lệnh

Trước hết, hãy xét đến **sys.argv**. Python giới thiệu một mô-đun (module) (thư viện) tên là **sys**, mà một trong những thành vin của nó là biến **argv**. Thực ra **argv** là một *danh sách*, giống như một thành phần có tên tương tự trong C/C++.[10] Phần tử 0 của danh sách chính là tên file lệnh, trong trường hợp này là **tme.py**, và các phần tử tiếp theo được viết theo quy tắc như C/C++. Trong ví dụ này, khi ta chạy chương trình với file tên là **x** thì `sys.argv[1]` chính là chuỗi 'x' (các chuỗi trong Python thường được đặt trong dấu nháy đơn). Bởi vì thư viện (mô-đun) `sys` không được nhập tự động khi Python khởi động, ta cần phải `import` nó.

Trong cả C/C++ lẫn Python, các tham số dòng lệnh tất nhiên là các chuỗi kí tự. Nếu ta cần các con số thì phải dùng lệnh đổi. Chẳng hạn, muốn có số nguyên ta dùng `int()` (với C là `atoi()`).[11]. Với số thực ta đổi bằng `float()`.[12]

4.2 Giới thiệu về xử lí file

Hàm **open()** cũng giống như trong C/C++. Nó có nhiệm vụ tạo ra một đối tượng **f** thuộc lớp **file**.

Hàm `readlines()` của lớp `file` trả về giá trị một *danh sách* chứa các dòng trong file đó. Mỗi dòng là một chuỗi kí tự, mỗi chuỗi như vậy là một phần tử của *danh sách*. Bởi vì file này có 5 dòng nên giá trị được trả lại từ hàm `readlines()` là một *danh sách* gồm 5 phần tử

```
['','This is an','example of a','text file','']
```

(Ở cuối mỗi chuỗi nêu trên đều có một kí tự xuống dòng, dù không được hiển thị cụ thể)

5 Khai báo hay không khai báo, Phạm vi, Hàm, v.v...

5.1 Không có phần khai báo

Trong Python, các biến không được khai báo. Mỗi biến được tạo thành khi có lệnh gán đầu tiên cho nó. Chẳng hạn, trong chương trình `tme.py` nêu trên, biến `flines` không tồn tại cho tận lúc câu lệnh:

```
flines = f.readlines()
```

[10] Tuy vậy, trong Python không có cả **argc** như trong C/C++. Python, một ngôn ngữ hướng đối tượng, coi danh sách như những đối tượng. Do vậy, số phần tử của danh sách cũng gắn luôn vào trong đối tượng đó. Nếu cần biết số phần tử của **argv**, ta có thể dùng **len(argv)**.

[11] `int()` của Python như `floor()` của C/C++

[12] Trong C/C++, dùng `atof`, hoặc `sscanf()`

được thực hiện.

Hơn nữa, một biến chưa được gán giá trị gì thì nó nhận giá trị `None` (ta có thể gán `None` cho một biến, hoặc dùng `None` để kiểm tra biến bằng lệnh `if`, v.v...).

5.2 Biến toàn cục và biến cục bộ

Thực ra Python không hề có biến toàn cục theo đúng nghĩa như C/C++, tức là phạm vi của biến là cả chương trình. Chúng ta sẽ bàn về điều này sau trong phần 10.1.5, nhưng ở đây ta giả sử mã nguồn chỉ gói gọn trong một file .py. Trong trường hợp này, Python có biến toàn cục rất giống với C/C++.

Python cố gắng suy diễn phạm vi của một biến dựa trên vị trí của nó trong mã lệnh. Nếu như một hàm bao gồm bất kể mã lệnh nào mà gán giá trị cho một biến, thì biến đó được coi là cục bộ. Bởi vậy, trong đoạn mã lệnh có hàm `checkline()`, Python sẽ cho rằng **l** và **wordcount** là các biến cục bộ của `checkline()`, nếu ta không nói rõ gì thêm (bằng từ khoá `global`, sẽ đề cập sau).

Việc sử dụng các biến toàn cục sẽ giản hoá những gì nêu ra ở đây, và cá nhân tôi tin rằng những lời chỉ trích về các biến toàn cục là không đáng tin cậy. (Xem `http://heather.cs.ucdavis.edu/~matloff/globals.html`.) Trên thực tế một trong những lệnh vực chính của lập trình, *threads*, việc sử dụng các biến toàn cục rất quan trọng (bắt buộc).

Tuy vậy, bạn có thể ít nhất là muốn nhóm lại tất cả các biến toàn cục thành một *lớp*, như tôi làm. Xem Phụ lục A.5.

Định nghĩa hàm; trong Python có kiểu biến không?

Dĩ nhiên từ khoá `def` dùng để định nghĩa hàm. Một lần nữa bạn cần chú ý rằng dấu hai chấm và cách viết thụt đầu dòng được sử dụng nhằm tạo ra một khối lệnh ở trong phần nội dung của hàm. Một hàm có thể trả về giá trị bằng cách dùng từ khoá `return`, chẳng hạn:

```
return 8888
```

Tuy nhiên, để phù hợp với tính chất của Python, hàm cũng không có một kiểu xác định cho dù nó có trả về một giá trị và đối tượng trả về có thể là bất cứ loại nào: số nguyên, danh sách, ...

6 Một số hàm có sẵn trong Python

Hàm `len` cho số lượng các phần tử có trong một *danh sách*, trong trường hợp ví dụ nêu trên là số dòng trong file (vì `readlines()` trả về một list mà mỗi phần tử là một dòng trong file đó).

Phương thức split() là một thành phần trong lớp string.[13] Một trường hợp thường gặp là chia cắt một chuỗi kí tự thành danh sách gồm những từ đơn lẻ.[14] Do đó nếu ta có `l` là 'This is an', dùng lệnh `checkline()` thì danh sách **w** sẽ là ['This','is','an']. (Nếu trong trường hợp dòng đầu tiên là dòng trống thì danh sách `w` sẽ là danh sách trống, [].)

Kiểu của biến / giá trị

[13] Các hàm thành phần của một lớp được gọi là phương thức.

[14] Dấu cách được mặc định sử dụng như kí tự chia cắt, mặc dù các kí tự/chuỗi khác cũng có thể đóng vai trò này.

Là một ngôn ngữ kịch bản điển hình, Python không khai báo những kiểu biến (nguyên, thực) như trong C/C++. Tuy vậy, bộ dịch lệnh Python cũng theo dõi kiểu của tất cả các đối tượng trong bộ nhớ.

Các kiểu biến trong Python gồm các kí hiệu số (còn gọi là *vô hướng*), dãy (danh sách hoặc tuple) và các từ điển (sẽ được trình bày trong phần 6.2.3), các lớp, các hàm, v.v...

6.1 Các chuỗi so với các giá trị số

Khác với Perl, Python có sự phân biệt giữa kiểu số và chuỗi kí tự biểu diễn số đó. Các hàm **eval()** và **str()** có thể được sử dụng để chuyển đổi qua lại. Chẳng hạn:

```
>>> 2 + '1.5'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> 2 + eval('1.5')
3.5
>>> str(2 + eval('1.5'))
'3.5'
```

Ngoài ra còn có **int()** để chuyển đổi từ dạng chuỗi sang số nguyên, và **float()**, để chuyển đổi từ dạng chuỗi sang số thực.

```
>>> n = int('32')
>>> n
32
>>> x = float('5.28')
>>> x
5.2800000000000002
```

Xem thêm phần **??**

Các danh sách là trường hợp đặc biệt của *dãy*, chúng đều có kiểu mảng nhưng còn một số điều khác biệt. Tuy vậy cần chú ý các điểm chung sau (một số sẽ được giải thích ngay dưới đây) đều có thể được áp dụng cho bất kì kiểu dãy nào:

- việc sử dụng cặp ngoặc vuông để chỉ định từng phần tử riêng lẻ (chẳng hạn **x[i]**
- hàm có sẵn **len()** sẽ cho số phần tử có trong dãy [15]
- các phép toán dạng *lát cắt* (để chiết xuất dãy con).
- sử dụng **+** và ***** để thực hiện ghép nối và nhân bản.

[15] Hàm này cũng có thể áp dụng cho kiểu từ điển.

6.2 Danh sách (mảng)

Như đã nói trước, các danh sách được biểu thị bằng cặp ngoặc vuông và các dấu phẩy. Chẳng hạn, Câu lệnh:

```
x = [4,5,12]
```

sẽ gán **x** cho một mảng 3 phần tử xác định.

Các mảng có thể điều chỉnh kích thước, bằng cách dùng các hàm **append()** (bổ sung) hoặc **extend()** (mở rộng) của lớp **list**. Chẳng hạn, sau câu lệnh trên nếu ta viết tiếp:

```
x.append(-2)
```

thì **x** sẽ có giá trị bằng [4,5,12,-2].

Một số toán tử khác cũng có thể áp dụng cho danh sách, một số trong đó được minh hoạ trong đoạn lệnh sau:

```
>>> x = [5,12,13,200]
>>> x
[5, 12, 13, 200]
>>> x.append(-2)
>>> x
[5, 12, 13, 200, -2]
>>> del x[2]
>>> x
[5, 12, 200, -2]
>>> z = x[1:3] # ‘‘cắt lát’’ mảng: các phần tử từ 1 đến 3-1=2
>>> z
[12, 200]
>>> yy = [3,4,5,12,13]
>>> yy[3:]  # tất cả các phần tử bắt đầu từ thứ tự 3 trở đi
[12, 13]
>>> yy[:3]  # tất cả các phần tử từ đầu cho đến trước phần tử
thứ 3
[3, 4, 5]
>>> yy[-1]  # nghĩa là ‘‘1 phần tử tính từ đầu bên phải’’
13
>>> x.insert(2,28)  # điền thêm 28 vào vị trí 2
>>> x
[5, 12, 28, 200, -2]
>>> 28 in x  # kiểm tra phần tử có thuộc mảng không, 1=có, 0=không
1
>>> 13 in x
0
>>> x.index(28)  # tìm vị trí của phần tử có tên tương ứng trong
danh sách)
```

```
2
>>> x.remove(200)  # khác so với ‘‘delete’’, vì nó vì chỉ ra giá trị
phần tử cần xóa thay vì vị trí của nó
>>> x
[5, 12, 28, -2]
>>> w = x + [1,"ghi"] # ghép các danh sách với nhau
>>> w
[5, 12, 28, -2, 1, ’ghi’]
>>> qz = 3*[1,2,3]  # lặp lại các phần tử của danh sách
>>> qz
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> x = [1,2,3]
>>> x.extend([4,5])
>>> x
[1, 2, 3, 4, 5]
>>> y = x.pop(0)  # xoá và trả lại giá trị vừa xoá khỏi danh sách
>>> y
1
>>> x
[2, 3, 4, 5]
```

Chúng ta cũng có thể thấy toán tử**in** trong ví dụ trên được dùng trong một vòng lặp **for**.

Một danh sách có thể bao gồm các thành phần khác loại, thậm chí cả các danh sách khác.

Cách lập trình thông dụng của Python bao gồm một số“mẹo Python” liên quan đến kiểu dãy, chẳng hạn một cách đơn giản tráo đổi giá trị hai biến **x** và **y**:

```
>>> x = 5
>>> y = 12
>>> [x,y] = [y,x]
>>> x
12
>>> y
5
```

Các mảng nhiều chiều có thể được mô tả như một danh sách gồm các danh sách bên trong nó, chẳng hạn:

```
>>> x = []
>>> x.append([1,2])
>>> x
[[1, 2]]
>>> x.append([3,4])
>>> x
[[1, 2], [3, 4]]
>>> x[1][1]
4
```

6.2.1 Các Tuple

Tuple cũng giống như các dãy, nhưng là loại *không hoán vị được*, nghĩa là không thể thay đổi được. Chúng được ngoặc tròn thay vì ngoặc vuông.[16]

Ta có thể dùng các phép toán tương tự như phần trước đối với tuple, trừ các toán tử gây nên sự thay đổi tuple. Do đó chẳng hạn:

```
x = (1,2,'abc')
print x[1]  # in số 2
print len(x)  # in số 3
x.pop()  # không thực hiện được do làm thay đổi tuple
```

Một hàm rất hữu dụng là **zip**, cho phép xâu chuỗi các thành phần tương ứng từ các danh sách khác nhau để tạo thành một tuple mới, chẳng hạn

```
>>> zip([1,2],['a','b'],[168,168])
[(1, 'a', 168), (2, 'b', 168)]
```

6.2.2 Chuỗi kí tự

Chuỗi kí tự chính là tuple của các phần tử kí tự. Tuy nhiên chúng được viết giữa cặp dấu nháy kép thay vì cặp ngoặc tròn, và có khả năng linh hoạt hơn so với một tuple tương ứng. Chẳng hạn:

```
>>> x = 'abcde'
>>> x[2]
'c'
>>> x[2] = 'q'  # không có hiệu lực với chuỗi, vì tính không thể thay đổi
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> x = x[0:2] + 'q' + x[3:5]
>>> x
'abqde'
```

(Bạn có thể hỏi tại sao lệnh gán cuối cùng

```
>>> x = x[0:2] + 'q' + x[3:5]
```

không vi phạm tính không thể thay đổi. Lí do là **x** thực sự là một con trỏ và ta chỉ đơn giản là trỏ nó đến một chuỗi kí tự mới được tạo thành từ các chuỗi cũ. Xem thêm phần 8.)

Như đã lưu ý, chuỗi có nhiều tính năng hơn là tuple gồm các kí tự

[16] Cặp ngoặc tròn chỉ bắt buộc trong trường hợp để tránh gây nhầm lẫn, chẳng hạn như các tham số trong hàm. Một dấu phẩy có thể sử dụng trong trường hợp tuple rỗng, nghĩa là **(,)**.

```
>>> x.index('d')  # như mong đợi
3
>>> 'd' in x  # như mong đợi
1
>>> x.index('cd')  # một bất ngờ thú vị
2
```

Như vậy chúng ta thấy được, hàm **index** của lớp **str** được *overload*, và do đó linh hoạt hơn.

Có nhiều các hàm tiện dụng khác trong lớp **str**. Chẳng hạn, ta thấy đã hàm **split()** ở phần trước. Hàm ngược của nó là **join()**. Khi dùng hàm này với một chuỗi, và kèm theo một loạt các chuỗi tham biến khác, kết quả thu được sẽ là một loạt các chuỗi kí tự tham biến được nối với nhau bằng chuỗi ban đầu.[17]

```
>>> '---'.join(['abc','de','xyz'])
'abc---de---xyz'
>>> q = '\\n'.join(('abc','de','xyz'))
>>> q
'abc\\nde\\nxyz'
>>> print q
abc
de
xyz
```

Một số ví dụ tiếp theo:[18]

```
>>> x = 'abc'
>>> x.upper()
'ABC'
>>> 'abc'.upper()
'ABC'
>>> 'abc'.center(5) # căn giữa chuỗi trong một khoảng rộng 5 kí tự
' abc '
```

Lớp **str** được dựng sẵn trong các phiên bản mới của Python. Với các phiên bản cũ bạn cần thêm câu lệnh:

```
import string
```

Lớp **string** này đến giờ vẫn tồn tại, và lớp mới **str** không hoàn toàn sao chép từ nó.

6.2.3 Sắp xếp

Hàm **sort()** của Python có thể được áp dụng cho bất cứ kiểu dãy nào. Với các kiểu không phải vô hướng, ta dùng hàm so sánh trả lại các giá trị âm, bằng 0 hoặc dương, bằng các kí hiệu $<$, $=$ or $>$. Sau đây là minh họa trong đó một mảng gồm các mảng được sắp xếp theo phần tử thứ hai.

[17] Ví dụ dưới đây cho thấy cách dùng "mới" của **join()**. Ngày nay các phương thức xử lí chuỗi là thành phần sẵn có của Python. Xem thêm so sánh giữa "mới" và "cũ" dưới đây.

[18] Có rất nhiều hàm xử lí chuỗi kí tự trong mô-đun **re** ("regular expression").

```
>>> x = [[1,4],[5,2]]
>>> x
[[1, 4], [5, 2]]
>>> x.sort()
>>> x
[[1, 4], [5, 2]]
>>> def g(u,v):
...     return u[1]-v[1]
...
>>> x.sort(g)
>>> x
[[5, 2], [1, 4]]
```

(Dùng hàm "lambda" có thể thực hiện dễ dàng hơn. Xem thêm phần 14.1.)

Kiểu Từ điển

Từ điển là các **mảng liên kết**. Phần sau sẽ bàn đến nghĩa kĩ thuật của nó; nhưng trên quan điểm lập trình thuần túy, nghĩa là ta có thể thiết lập một mảng với chỉ số không cần là số nguyên. Câu lệnh

```
x = {'abc':12,'sailing':'away'}
```

gán **x** cho một mảng 2 phần tử với **x['abc']** bằng 12 và **x['sailing']** bằng 'away'. Ta nói rằng **'abc'** và **'sailing'** là các *khóa* (key), còn 12 và 'away' là các *giá trị* (value). Các khóa có thể là bất cứ đối tượng nào thuộc loại không biến đổi, như số, tuple hoặc chuỗi.[19] Việc sử dụng khóa là các tuple tương đối phổ biến trong các chương trình Python, và bạn cần chú ý rằng ta có thể tận dụng khả năng này.

Xét sâu xa, **x** ở đây có thể là một mảng 4 phần tử và việc thực hiện lệnh kiểu như

```
w = x['sailing']
```

sẽ yêu cầu bộ dịch lệnh Python duyệt qua mảng đó để tìm từ khoa 'sailing'. Nếu tiến hành theo cách tìm kiểu tuyến tính sẽ chậm, cho nên cấu trúc bên trong sẽ có dạng bảng hash. Đây là lí do tại sao kiểu từ điển của Python (tương tự Perl) thực ra được gọi là **hash**.

Sau đây là các ví dụ sử dụng một số hàm thành viên của lớp *từ điển*:

```
>>> x = {'abc':12,'sailing':'away'}
>>> x['abc']
12
>>> y = x.keys()
>>> y
['abc', 'sailing']
>>> z = x.values()
>>> z
```

[19] Bây giờ bạn có thể hiểu tại sao Python lại phân biệt giữa tuple và danh sách. Nếu cho phép các khóa thay đổi thì thực sự sẽ gay go, và có thể dẫn tới chương trình có chứa nhiều lỗi.

```
[12, 'away']
x['uv'] = 2
>>> x
{'abc': 12, 'uv': 2, 'sailing': 'away'}
```

Chú ý cách ta bổ sung thêm phần tử một vào **x** ở vị trí gần cuối.

Nhập số liệu từ bàn phím

Hàm **raw_input()** sẽ cho hiện ra dấu nhắc và đọc vào giá trị mà ta gõ từ bàn phím. Chẳng hạn:

```
name = raw_input("enter a name: ")
```

sẽ xuất hiện "enter a name:", Sau đó đọc chữ ta gõ vào, và lưu nội dung này vào biến **name**. Chú ý rằng dữ liệu người dùng nhập vào được trả lại dưới dạng chuỗi, và cần phải chuyển đổi nếu như ta muốn nhập số.

Trong trường hợp không muốn có dấu nhắc, chỉ cần không chỉ định nó:

```
>>> y = raw_input()
3
>>> y
'3'
```

6.2.4 Tác dụng của __name__

Trong một số trường hợp, ta rất cần biết một mô-đun có phải được chạy trực tiếp không hay là thông qua import. Điều này có thể xác định được bằng cách dùng biến có sẵn **__name__** của Python như sau.

Bất cứ khi nào bộ dịch lệnh Python đang hoạt động thì nó được gọi là chương trình cấp cao nhất (**top-level program**). Chẳng hạn nếu bạn gõ

```
% python x.py
```

thì mã lệnh trong **x.py** là chương trình cấp cao nhất. Tương tự như vậy, nếu bạn đang chạy ở chế độ tương tác dòng lệnh, thì lệnh bạn trực tiếp gõ vào cũng là chương trình cấp cao nhất.

Đối với bộ dịch lệnh thì chương trình tương tác được gọi là **__main__**, còn mô-đun hiện đang được chạy gọi là **__name__**. Do dó muốn kiểu tra một mô-đun có phải được chạy không hay là được nhập (import) từ câu lệnh khác, ta phải kiểu tra xem **__name__** có phải là **__main__** hay không.

Chẳng hạn, ta thêm câu lệnh:

```
print  __name __
```

vào ví dụ đầu tiên, trong phần 3.1 của file **fme.py**:

```
print  __name __
for i in range(10):
   x = 0.1*i
   print x
   print x/(1-x*x)
```

Hãy chạy chương trình này 2 lần. Lần đầu ta chạy nó trực tiếp:

```
% python fme.py
 __main __
0.0
0.0
0.1
0.10101010101
0.2

0.208333333333
0.3
0.32967032967
... [phần kết quả còn lại không viết ra đây]
```

Bây giờ xem điều gì sẽ xảy ra nếu ta chạy nó từ môi trường tương tác dòng lệnh:

```
>>>  __name __
' __main __'
>>> import fme
fme
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
... [phần kết quả còn lại không viết ra đây]
```

Câu lệnh trong mô-đun của chúng ta

```
print  __name __
```

lần thứ nhất sẽ in ra **__main__**, nhưng lần thứ hai sẽ in ra **fme**.

Thông thường (theo thói quen trong lập trình C), mỗi "chương trình chính" được tập hợp vào trong một hàm tên là **main()**. Ta sẽ thay đổi ví dụ trên theo cách này thành file **fme2.py**:

```
def main():
   for i in range(10):
      x = 0.1*i
      print x
      print x/(1-x*x)

if  __name __ == ' __main __':
   main()
```

Lợi ích của cách làm này là khi ta nhập mô-đun này, mã lệnh sẽ không được thực hiện ngay. Thay vào đó, **fme2.main()** phải được gọi, hoặc là bằng nhiều mô-đun hoặc là bằng bộ dịch lệnh tương tác. Trường hợp thứ hai được minh họa bằng ví dụ sau:

```
>>> import fme2
>>> fme2.main()
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
...
```

Bên cạnh các điểm khác, điều này rất quan trọng trong việc sử dụng các công cụ gỡ lỗi (Section A). Do dó hãy tạo thói quen thường xuyên thiết lập truy cập vào **main()** như vậy trong chương trình.

7 Lập trình hướng đối tượng (Object-Oriented Programming), OOP

Đối lập với Perl, bản thân Python đã có tính hướng đối tượng ngay trong gốc rễ. Do dó nó có giao diện hướng đối tượng (OOP) rõ hơn, đẹp hơn.

Ví dụ mã lệnh chương trình

Ta sẽ minh họa bằng việc xây dựng một lớp có nhiệm vụ xử lí file văn bản. Sau đây là nội dung file **tfe.py**:

```
class textfile:
   ntfiles = 0  # đếm số đối tượng file text
   def  __init __(self,fname):
      textfile.ntfiles += 1
      self.name = fname  # tn
      self.fh = open(fname)  # ``handle'' của file
```

```
      self.lines = self.fh.readlines()
      self.nlines = len(self.lines)  # số dòng
      self.nwords = 0  # số chữ
      self.wordcount()
   def wordcount(self):
      "dem so chu trong file"
      for l in self.lines:
         w = l.split()
         self.nwords += len(w)
   def grep(self,target):
      "dem tat ca cac dong co chua chu can tim"
      for l in self.lines:
         if l.find(target) >= 0:
            print l

a = textfile('x')
b = textfile('y')
print "So file text duoc mo la", textfile.ntfiles
print "Sau day la mot so thong tin ve chung (ten, so dong, so chu):"
for f in [a,b]:
   print f.name,f.nlines,f.nwords
a.grep('example')
```

Ngoài file **x** được sử dụng trong phần 4 ở trên, tôi cũng có file **y** gồm 2 dòng. Sau đây là kết quả thu được khi chạy chương trình:

```
% python tfe.py
So file text duoc mo la 2
Sau day la mot so thong tin ve chung (ten, so dong, so chu):
x 5 8
y 2 5
example of a
```

7.1 Từ khóa "self"

Hãy cùng xem xét lớp **textfile**. Điều đầu tiên cần lưu ý là sự xuất hiện của từ khóa **self**, với nghĩa là thực thể hiện hành của lớp, tương tự như **this** đối với C++ và Java. Do đó **self** là một con trỏ tới thực thể hiện tại của lớp.

7.2 Các biến thực thể

Trong thuật ngữ của lập trình hướng đối tượng, một biến thực thể **x** của một lớp là một biến thành viên mà với nó một thực thể của lớp có một giá trị riêng biến của biến đó. Trong thế giới của C++ và Java, bạn biết đến chúng dưới tên biến không tĩnh (non-static). Thuật ngữ *biến thực thể* là một thuật ngữ chung trong lập trình hướng đối tượng, không phải là riêng cho từng ngôn ngữ.

Để thấy được chúng hoạt động như thế nào đối với Python, trước hết hãy nhớ lại là mỗi biến của Python được hình thành khi ta gán cho nó một giá trị. Bởi vậy, một biến thực thể trong một lớp của Python cũng không tồn tại cho đến tận khi nó được gán giá trị.

```
self.name = fname  # tên
```

được thực hiện, biến thành viên **name** cho thực thể hiện tại của lớp được tạo ra, và nó được gán giá trị đã định sẵn.

Các biến lớp

Một biến lớp **v**, có chung giá trị đối với tất cả các thực thể trong lớp đó [20] được thiết kế theo cách của một số tham chiếu vào **v** trong mã nguồn, phía bên trong lớp nhưng không phải là trong bất cứ phương thức nào của lớp đó. Ví dụ như đoạn mã trên: [21]

```
ntfiles = 0  # đếm số đối tượng file văn bản
```

Chú ý rằng mỗi biến lớp **v** của một lớp **u** được tham chiếu với dạng **u.v** trong phạm vi các phương thức của lớp đó và trong các đoạn mã bên ngoài phạm vi lớp. Với đoạn mã bên trong lớp nhưng ngoài phương thức, có thể viết gọn tham chiếu là **v**. Bây giờ hãy dành chút thời gian rà soát lại đoạn chương trình trên, và xem các ví dụ về điều này với biến **ntfiles** của chúng ta.

7.2.1 Tạo lớp và xóa lớp

Hàm khởi tạo lớp phải có tên là **__init()__**. Tham biến **self** là bắt buộc, và sau đó bạn có thể tham chiếu tham biến khác, như tôi trình bày trong ví dụ trên, là một tên file.[22]

Hàm xóa lớp là **__del()__**. Chú ý rằng hàm này chỉ được thực hiện khi việc thu gom rác trong bộ nhớ đã hoàn tất, nghĩa là khi toàn bộ các biến trỏ tới đối tượng đã bị mất.

7.3 Các phương thức với thực thể

Phương thức **wordcount()** là một phương thức thực thể theo nghĩa được áp dụng riêng cho một đối tượng cho trước của lớp.[23]

Khác với C++ và Java với **this** là một tham biến ngầm định cho các phương thức thức thì Python lại làm sáng tỏ quan hệ này một cách thông minh; tham biến **self** là bắt buộc.

[20] Một lần nữa, trong thế giới của C++ và Java, nó được biết đến với tên biến *tĩnh*.

[21] Ví dụ này ta đặt đoạn mã vào phần đầu của lớp, nhưng thực tế nó có thể đứng ở cuối lớp, hoặc giữa hai phương thức, miễn là không phải ở trong một phương thức nào. Trong trường hợp đặt trong một phương thức, **ntfiles** sẽ được coi như một biến địa phương trong phương thức đó, đây là điều ta hoàn toàn không mong muốn.

[22] Tham biến **self** là bắt buộc, nhưng tên của nó có thể khác. Khác với C++/Java, **self** không phải là một tên dành riêng (từ khóa). Bạn có thể sử dụng bất cứ tên gọi nào miễn là phải thống nhất, như trong phương thức **__init()__** ở ví dụ dưới đây. Nhưng về hình thức sẽ không hay nếu ta sử dụng từ khác.

[23] Một lần nữa, trong thuật ngữ của C++/Java, đây là những phương thức không-*tĩnh*.

7.4 Docstring

Có một chuỗi kí tự trong dấu nháy kép, "xác định số chữ trong file", ngay phần đầu của **wordcount()**. Nó được gọi là *docstring*. Đây là một dạng chú thích, nhưng có tác dụng khi chương trình được thực hiện, do đó có thể được sử dụng với mục đích kiểu như gỡ lỗi. Bên cạnh đó nó cho phép người dùng chỉ có trong tay các method dưới dạng biên dịch (không phải mã nguồn), chẳng hạn như trong phần mềm thương mại, khả năng truy cập đến "chú thích." Sau đây là một ví dụ về cách truy cập, sử dụng file **tf.py** nêu trên:

```
>>> import tf
>>> tf.textfile.wordcount. __doc __
'dem so chu trong file'
```

Một docstring thường kéo dài trên vài dòng; do vậy ta dùng dấu 3 nháy để tạo loại chuỗi này.

Phương thức **grep()** là một phương thức thực thể khác, cái này kèm với tham biến **self** bên cạnh nó

Cần nói thêm là các tham biến phương thức trong Python chỉ có thể truyền theo giá trị, theo nghĩa của C thì các hàm có tác dụng phụ đối với tham số chỉ khi tham số là con trỏ (Python không có kiểu con trỏ theo đúng nghĩa, nhng nó có các tham chiếu, xem phần 8.)

Cần chú ý rằng **grep()** tận dụng một trong số các phép toán chuỗi của Python, đó là **find()**. Nó tìm kiếm chuỗi tham biến bên trong chuỗi đối tượng, sau đó trả lại chuỗi số có lần xuất hiện đầu tiên, hay -1 nếu không tìm thấy.[24]

7.5 Các phương thức lớp

Trước phiên bản 2.2, Python không hỗ trợ chính thức các phương thức lớp, nghĩa là các phương thức không áp dụng cho từng đối tượng cụ thể của lớp. Nay Python có hai cách (hơi khác nhau) thực hiện việc này, đó là sử dụng các hàm **staticmethod()** và **classmethod()**. Tôi sẽ trình bày cách sử dụng cuả hàm thứ nhất, với ví dụ mở rộng mã lệnh trong phần 7 đối với lớp **textfile**:

```
class textfile:
   ...
   def totfiles():
      print "tong so file text la", textfile.ntfiles
   totfiles = staticmethod(totfiles)

...

# ở trong phần "main"
...
textfile.totfiles()
...
```

Chú ý rằng các phương thức lớp không có tham biến **self**.

[24] Chuỗi cũng có thể được xử lí như một danh sách các kí tự. Chẳng hạn, 'geometry' có thể coi là một danh sách gồm 8 phần tử và dùng **find()** với chuỗi con 'met' sẽ cho kết quả là 3.

Cần chú ý rằng phương thức này có thể được gọi ngay cả khi không có thực thể nào của lớp **textfile**. Trong ví dụ nêu ra ở đây, 0 sẽ là kết quả được in ra, vì không có file nào được đếm.[25]

7.6 Các lớp suy diễn

Tính kế thừa là một phần quan trọng trong triết lí của Python. Một câu lệnh kiểu như

```
class b(a):
```

khởi tạo định nghĩa một lớp con **b** của một lớp **a**. Ta cũng có thể thực hiện kế thừa nhiều lần.

Chú ý rằng khi **constructor** của một lớp suy diễn được gọi thì **constructor** của lớp cơ sở lại *không* được gọi cùng. Nếu bạn muốn kích hoạt constructor của lớp này, bạn phải tự làm lấy, chẳng hạn

```
class b(a):
   def  __init __(self,xinit):  # constructor của lớp b
      self.x = xinit  # định nghĩa và khởi tạo một biến thực thể x
      a. __init __(self)  # gọi constructor của lớp cơ sở
```

Theo các văn bản hướng dẫn chính thức của Python thì"[Theo cách nhìn của C++] các phương thức của Python đều thực sự là ảo." Nếu bạn muốn mở rộng thay vì ghi đè lên một phương thức của lớp cơ sở, bạn có thể tham chiếu đến lớp cơ sở bằng cách thêm vào phía trước tên của lớp cơ sở, như trong ví dụ đang xét, là **a.__init__(self)**.

7.7 Một lưu ý về lớp

Mỗi thực thể lớp trong Python hoạt động như một từ điển. Chẳng hạn, trong chương trình **tfe.py** ở trên, đối tượng **b** được thực hiện như một từ điển.

Ngoài các điểm khác ra, diều này cũng có nghĩa là bạn có thể thêm các biến thành phần vào thực thể lớp trong lúc chương trình đang chạy, nghĩa là rất lâu sau khi tạo ra thực thể. Chẳng hạn, trong phần "main" của chương trình, ta có thể có một câu lệnh kiểu như:

```
b.name = 'zzz'
```

8 Tầm quan trọng của việc hiểu được tham chiếu đối tượng

Một biến được gán giá trị có dạng thay đổi được thực chất là một con trỏ tới đối tượng đã cho. Chẳng hạn, xét đoạn lệnh sau:

[25] Chú ý rằng số 0 khác với giá trị **None** của Python. Ngay cả khi ta chưa tạo các thực thể của lớp **textfile**, mã lệnh `ntfiles = 0` vẫn sẽ được thực hiện khi ta lần đầu chạy chương trình. Như đã đề cập đến từ trước, bộ dịch lệnh Python thực hiện file từ dòng thứ nhất trở đi. Khi nó dịch đến dòng `class textfile:` thì tiếp theo sẽ thực hiện một mã lệnh bên ngoài các phương thức trong định nghĩa lớp.

```
>>> x = [1,2,3]
>>> y = x  # x và y bây giờ đều chỉ tới [1,2,3]
>>> x[2] = 5  # điều này có nghĩa là y[2] cũng chuyển thành 5 !
>>> y[2]
5
>>> x = [1,2]
>>> y = x
>>> y
[1, 2]
>>> x = [3,4]
>>> y
[1, 2]
```

Trong đoạn đầu chương trình, **x** và **y** đều là các tham chiếu đến một danh sách, một đối tượng kiểu thay đổi được. Câu lệnh

```
x[2] = 5
```

thay đổi một phần của đối tượng đó, nhưng **x** vẫn chỉ tới đối tượng này. Mặt khác, với đoạn lệnh

```
x = [3,4]
```

bây giờ **x** lại tự thay đổi, khi đó nó chỉ tới một đối tượng khác, trong khi **y** vẫn chỉ tới đối tượng ban đầu.

Nếu như trong ví dụ trên ta chỉ muốn copy tham chiếu của danh sách từ **x** tới **y**, ta có thể dùng cách cắt lát, chẳng hạn

```
y = x[:]
```

Như vậy **y** và **x** sẽ chỉ tới các đối tượng khác nhau, và khi đó mặc dù các đối tượng này tạm thời có cùng giá trị nhưng nếu đối tượng được trỏ bởi **x** có thay đổi thì đối tượng trở bởi **y** sẽ không thay đổi.

Một vấn đề quan trọng tương tự nảy sinh với các tham biến trong lệnh gọi hàm. Bất kì tham biến nào trỏ tới một đối tượng loại có thể thay đổi thì nó có thể thay đổi giá trị của đối tượng từ bên trong của hàm, chẳng hạn:

```
>>> def f(a):
...     a = 2*a
...
>>> x = 5
>>> f(x)
>>> x
5
>>> def g(a):
...     a[0] = 2*a[0]
...
```

```
>>> y = [5]
>>> g(y)
>>> y
[10]
```

Các tên hàm cũng là tham chiếu đến các đối tượng. Thứ mà chúng ta vẫn nghĩ là tên hàm thực chất chính là một con trỏ–loại thay đổi được—trỏ tới mã lệnh của hàm đó. Chẳng hạn,

```
>>> def f():
...     print 1
...
>>> def g():
...     print 2
...
>>> f()
1
>>> g()
2
>>> [f,g] = [g,f]
>>> f()
2
>>> g()
1
```

Có thể xóa các đối tượng bằng lệnh **del**.

Nếu bạn muốn thực hành thêm về các kí hiệu này, hãy xem phần A.4.

9 So sánh các đối tượng

Ta có thể dùng toán tử $<$ để so sánh các dãy, chẳng hạn

```
if x < y:
```

Đối với các danh sách **x** and **y**, việc so sánh là theo từng thành phần lần lượt từ trái qua phải. Chẳng hạn,

```
>>> [12,'tuv'] < [12,'xyz']
True
>>> [5,'xyz'] > [12,'tuv']
False
```

Tất nhiên với các chuỗi cũng là dãy nên ta cũng có thể so sánh chúng:

```
>>> 'abc' < 'tuv'
```

```
True
>>> 'xyz' < 'tuv'
False
>>> 'xyz' != 'tuv'

True
```

Có thể thực hiện so sánh với các đối tượng không phải dãy, như các thực thể lớp, bằng cách định nghĩa một hàm **__cmp()__** trong lớp đó.

Ta có thể sắp xếp tinh vi hơn bằng cách kết hợp hàm **sort()** của Python với một hàm dùng riêng **cmp()**.

10 Các mô-đun và gói chương trình

Bạn đã nghe thường xuyên rằng nguyên tắc quan trọng trong thiết kế phần mềm là viết mã lệnh theo hình thức "mô-đun", nghĩa là chia nhỏ nó thành các thành phần khác nhau, từ trên xuống, và làm cho mã lệnh có thể được sử dụng lại, chẳng hạn viết nó theo cách tổng quát để về sau này bạn hoặc người khác có thể tận dụng lại chúng trong một số chương trình khác. Không giống với các kiểu làm phần mềm tràng giang đại hải, cách này mới là thực sự đúng đắn! :-)

10.1 Mô-đun

Một **mô-đun** là một tập hợp các lớp, các hàm thư viện, v.v... vào trong một file. Khác với Perl, không cần thực hiện gì thêm để chuyển một file thành một mô-đun. Bất kể file nào cũng phần mở rộng **.py** đều là mô-đun![26]

10.1.1 Ví dụ mã lệnh chương trình

ï¿1/2lm ví dụ hãy lï¿1/2 lï¿1/2 **textfile** tï¿1/2ví dụta lm ï¿1/2trn. Bây giờ ta ï¿1/2 nï¿1/2vo một file *ring*, **tf.py**, vï¿1/2 nội dung sau:

```
# file tf.py

class textfile:
   ntfiles = 0  # đếm số đối tượng file text
   def  __init __(self,fname):
      textfile.ntfiles += 1
      self.name = fname  # tên
      self.fh = open(fname)  # handle của file
      self.lines = self.fh.readlines()
      self.nlines = len(self.lines)  # số dòng
      self.nwords = 0  # số chữ
      self.wordcount()
```

[26] Chú ý rằng tên file cũng phải bắt đầu bằng một chữ cái, không phải chữ số hay kí tự nào khác.

```
   def wordcount(self):
      "đếm số chữ trong file"
      for l in self.lines:
         w = l.split()
         self.nwords += len(w)
   def grep(self,target):
      "in ra tất cả các dòng chứa chữ cần tìm"
      for l in self.lines:
         if l.find(target) >= 0:
            print l
```

(Chú ý rằng mặc dù mô-đun của ta ở đây chỉ bao gồm một lớp, nhưng thực tế ta vẫn có thể có vài lớp, cộng thêm các biến toàn cục, mã lệnh chạy được mà không nằm trong bất cứ hàm nào, v.v...) File chương trình thử **tftest.py**, bây giờ sẽ có dạng:

```
# file tftest.py

import tf

a = tf.textfile('x')
b = tf.textfile('y')
print "số lượng file text đang mở là:", tf.textfile.ntfiles
print "Sau đây là một số thông tin về chúng (tên, số dòng, số từ):"
for f in [a,b]:
   print f.name,f.nlines,f.nwords
a.grep('example')
```

10.1.2 Lệnh `import` làm việc thế nào?

Bộ dịch lệnh Python khi thấy lệnh **import tf**, sẽ tìm nội dung của file **tf.py**. [27] Bất kì mã lệnh nào trong **tf.py**, nếu có thể chạy được sẽ được thực hiện, trong trường hợp này là:

```
ntfiles = 0  # đếm số đối tượng file text
```

(Mã lệnh thực hiện ở trong mô-đun có thể sẽ không nằm trong các lớp. Hãy xem điều gì sẽ xảy ra nếu ta thực hiện **import fme2** trong một ví dụ phần 6.2.4 dưới đây.)

Sau đó khi bộ dịch lệnh gặp tham chiếu đến **tf.textfile**, nó sẽ tìm mục có tên là **textfile** nằm trong mô-đun **tf**, nghĩa là nằm trong file **tf.py**, và sau đó sẽ thực hiện xử lí.

Một cách làm khác có thể là:

```
from tf import textfile

a = textfile('x')
```

[27] Trong trường hợp này, ta có thể chuyển hai file vào cùng một thư mục, nhưng sau này sẽ chỉ định đường dẫn tìm kiếm.

```
b = textfile('y')
print "số file văn bản đang mở là", textfile.ntfiles
print "Sau đây là một số thông tin về chúng (tên, số dòng, số chữ:"
for f in [a,b]:
   print f.name,f.nlines,f.nwords
a.grep('example')
```

Bằng cách này ta phải gõ ít hơn, bởi thay vì"tf.textfile", ta chỉ gõ "textfile", làm cho mã lệnh đỡ rối. Nhưng cũng có tranh luận rằng cách này sẽ không an toàn (điều gì sẽ xảy ra nếu **tftest.py** của một số mục khác cùng tên **textfile**?) và không rõ ràng (vị trí ban đầu của **textfile**ở trong **tf** có thể sẽ giúp ta phân định rõ hơn trong trường hợp các chương trình lớn).

Câu lệnh

```
from tf import *
```

sẽ nhập tất cả mọi thứ trong **tf.py** theo cách này.

Trong bất kì trường hợp nào, bằng cách tách riêng lớp **textfile**, ta tăng cường mô-đun hóa mã lệnh tạo điều kiện cho việc tái sử dụng sau này.

10.1.3 Mã lệnh được biên dịch

Tương tự trường hợp của Java, bộ dịch lệnh Python sẽ biên dịch mã lệnh về dạng *bytecode* cho máy ảo Python. Nếu như mã lệnh được import thì mã lệnh dịch được lưu vào file với đuôi **.pyc**, do đó ta không phải dịch lại lần nữa.

Vì các mô-đun cũng là các đối tượng nên tên của các biến, hàm, lớp, v.v... của mỗi mô-đun cũng là các thuộc tính của mô-đun đó. Do vậy chúng được lưu lại trong file **.pyc**, và sẽ được hiển thị, chẳng hạn, khi bạn dùng hàm **dir()** trong mô-đun đó (Xem phần A.3.1).

10.1.4 Các vấn đề hỗn hợp

Các dòng lệnh "tự do" (không thuộc hàm nào) trong mô-đun sẽ được thực hiện ngay khi mô-đun được nhập.

Mô-đun là đối tượng. Chúng có thể được sử dụng như các tham số cho các hàm, trả lại giá trị từ hàm...

Danh sách **sys.modules** cho thấy tất cả các mô-đun đã nhập vào chương trình đang chạy.

10.1.5 Chú ý về biến toàn cục

Python không thực sự hỗ trợ các biến toàn cục như C/C++. Một mô-đun được nhập của Python sẽ không có quyền truy cập tới các biến toàn cục trong mô-đun ra lệnh nhập.

Chẳng hạn, ta xét hai file **x.py** và **y.py**.

```
# x.py
```

```
import y

def f():
   global x
   x = 6

def main():
   global x
   x = 3
   f()
   y.g()

if  __name __ == ' __main __':  main()
```

and **y.py**:

```
# y.py

def g():
   global x
   x += 1
```

Biến **x** trong **x.py** vẫn xuất hiện xuyên suốt mô-đun **x.py**, nhưng không phải trong **y.py**. Trên thực tế dòng lệnh

```
   x += 1
```

trong file thứ hai sẽ gây ra lỗi "global name 'x' is not defined." (biến toàn cục 'x' không được định nghĩa).

Thực sự là biến toàn cục trong mô-đun chỉ là một thuộc tính (nghĩa là một thực thể thành viên) của mô-đun đó cũng như vai trò của một biến lớp trong phạm vi lớp đó.

10.2 Che giấu dữ liệu

Python không có dạng che giấu dữ liệu nào tương tự như **private** và construct như trong C++. Tuy vậy nó cũng hỗ trợ một phần việc kiểu này.

Nếu bạn thêm vào một dấu gạch thấp phía trước tên biến trong một mô-đun thì biến đó sẽ không được nhập trong trường hợp ta dùng lệnh from...import. Chẳng hạn nếu mô-đun **tf.py** trong phần 10.1.1 của một biến **z**, thì câu lệnh:

```
from tf import *
```

sẽ có nghĩa là **z** được truy cập trực tiếp thay vì phải viết **tf.z**. Mặt khác, nếu ta đặt tên biến này là **_z** thì câu lệnh trên sẽ không làm cho biến này có khả năng được truy cập theo **_z** nữa; chúng ta phải viết **tf._z**. Dĩ

nhiên là biến vẫn có thể được nhìn thấy ngoài mô-đun nhưng bằng cách yêu cầu tiền tố **tf.** ta có thể tránh gây nhầm lẫn với những biến có tên giống nó trong các mô-đun khác được nhập cùng.

Hai dấu gạch thấp sẽ cho *mangling*, với mỗi dấu gạch thấp cộng với tên của mô-đun phía trước.

10.3 Các gói chương trình

Như đã đề cập ở trên, ta có thể đưa nhiều lớp vào trong cùng một mô-đun, nếu như các lớp có quan hệ chặt chẽ. Suy rộng ra cho trường hợp nếu có một vài mô-đun có quan hệ với nhau. Khi nội dung của chúng không gắn bó chặt chẽ ta có thể tống chúng vào một mô-đun rất lớn, nhưng nếu chúng có quan hệ cần thiết phải nhập theo một cách nào khác thì lần này có thể sử dụng **gói chương trình**.

Chẳng hạn, bạn muốn viết một số thư viện hỗ trợ cho phần mềm Internet no ï¿1/2 vừa viết. Bạn có một mô-đun **web.py** với các lệnh viết cho các chương trình thực hiện truy cập web và một mô-đun khác, **em.py** là phần mềm thao tác với e-mail. Thay vì kết hợp chúng trong một mô-đun lớn, bạn có thể đặt chúng riêng trong 2 file ở trong cùng thư mục, chẳng hạn **net**.

Muốn làm cho thư mục trở thành một gói chương trình, chỉ cần một file có tên **__init__.py** trong thư mục đó. File này có thể để trống, hoặc gồm một số lệnh khởi tạo trong trường hợp các chương trình phức tạp hơn.

Muốn nhập các mô-đun này, bạn viết lệnh kiểu như:

```
import net.web
```

Gặp lệnh này, bộ thông dịch Python sẽ tìm file có tên **web.py** nằm trong thư mục **net**. Thư mục **net** trong trường hợp này phải nằm trong một đường dẫn tìm kiếm (search path) của Python. Chẳng hạn nếu **net** có vị trí là:

```
/u/v/net
```

thì thư mục **/u/v** cần đặt trong đường dẫn tìm kiếm của Python. Trường hợp dùng hệ điều hành Unix với C shell, bạn có thể gõ lệnh:

```
setenv PYTHONPATH /u/v
```

Nếu bạn có một số thư mục đặc biệt kiểu này, hãy xâu chuỗi chúng lại phân cách bằng các dấu hai chấm:

```
setenv PYTHONPATH /u/v:/aa/bb/cc
```

Thư mục hiện hành được lưu trữ trong **sys.path**. Nó lại bao gồm một danh sách các chuỗi, mỗi chuỗi là một thư mục, phân cách bởi dấu hai chấm. Bạn có thể in chúng hoặc dùng lệnh thay đổi chúng, cũng như các biến khác.[28]

Các thư mục gói thường có các thư mục con, thư mục con của con và cứ thế. Mỗi thư mục như vậy phải có một file **__init__.py**.

[28] Lưu ý rằng trước hết cần phải nhập **sys**.

11 Xử lí lỗi

Cũng cần nói rằng các hàm có sẵn và thư viện của Python không có kiểu lỗi như C, trả lại mã lệnh để kiểm tra xem chúng có chạy thông hay không. Thay vào đó bạn cần sử dụng cơ chế bắt ngoại lệ **try/except**, chẳng hạn:

```
try:
   f = open(sys.argv[1])
except:
   print 'Lỗi mở file:',sys.argv[1]
```

Các ngoại lệ không chỉ gồm thao tác với file. Hãy xem đoạn mã sau:

```
try:
   i = 5
   y = x[i]
except:
   print 'Không có chỉ số', i
```

12 Phần hỗn hợp

12.1 Chạy mã lệnh Python mà không có trực tiếp mở bộ thông dịch

Chẳng hạn ban cần mở file **x.py**. Từ đầu đến giờ ta đã xem xét cách chạy nó bằng lệnh [29]

```
% python x.py
```

Nhng nếu trong dòng đầu tiên của file **x.py** bạn viết:

```
\#! /usr/bin/python
```

và sử dụng lệnh **chmod** của Unix để tạo ra file **x.py** chạy được, thì có thể chạy **x.py** bằng cách chỉ cần gõ

```
% x.py
```

diều này rất cần thiết, chẳng hạn trong trường hợp gọi chương trình từ một trang Web.

Hay hơn nữa, bạn có thể bắt Unix tìm vị trí của Python trong môi trường bằng cách đặt câu lệnh sau vào dòng đầu của file **x.py**:

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

Cách này linh hoạt hơn, vì các hệ điều hành khác nhau có thể xếp Python vào những thư mục khác nhau.

[29] Phần này chỉ dành riêng cho hệ điều hành Unix.

12.2 In kết quả không có dấu xuống dòng hoặc dấu trống

Khi gõ lệnh **print**, máy sẽ tự động in một dấu xuống dòng. Nhằm loại bỏ nó, hãy thêm vào cuối câu lệnh một dấu phẩy, chẳng hạn:

```
print 5,  # chưa in ra gì cả
print 12  # bây giờ in ra '5 12' tiếp đó là dấu xuống dòng
```

Thông thường ta dùng **print** với dấu phẩy trong một vòng lặp. Cuối cùng sau khi thực hiện vòng lặp in chuỗi kết quả, muốn xuống dòng chỉ cần thêm một lệnh **print** không.

Lệnh **print** tự động phân tách các phần tử bằng dấu cách. Nhằm triệt tiêu dấu cách đó hãy dùng toán tử nối chuỗi, **+**, và có thể là hàm **str()**, ví dụ

```
x = 'a'
y = 3
print x+str(y)  # in ra 'a3'
```

Bên cạnh đó **str(None)** sẽ cho ra **None**.

12.3 Định dạng chuỗi

Python hỗ trợ kiểu "printf()" của C, ví dụ

```
print "các thừa số của 15 là %d and %d" % (3,5)
```

sẽ cho ta

```
các thừa số của 15 là 3 và 5
```

Chú ý cách viết '(3,5)' thay vì '3,5'. Trong trường hợp sau, toán tử **%** sẽ nghĩ rằng các toán hạng của nó chỉ là 3, trong khi nó cần một tuple gồm hai phần tử. Hãy nhớ rằng trong mọi trường hợp không gây hiểu lầm thì cặp ngoặc của tuple có thể được bỏ đi, nhưng tiếc là trường hợp này lại không như vậy.

Cách định dạng này hay, nhưng ngoài lệnh in nó còn tiện dụng hơn trong các công việc xử lí chuỗi nói chung. Trong câu lệnh:

```
print "các thừa số của 15 là %d và %d" % (3,5)
```

phần

```
"các thừa số của 15 là %d và %d" % (3,5)
```

là một toán tử chuỗi, nó tạo ra một chuỗi mới; lệnh **print** chỉ đơn giản là in ra chuỗi mới đó.

Chẳng hạn:

```
>>> x = "%d tuổi" % 12
```

Biến **x** bây giờ là chuỗi '12 tuổi'.

Đây là một cách thường dùng và rất hiệu quả.[30]

12.4 Các tham biến có tên trong hàm

Xét hàm sau đây:

```
# ne.py; minh họa cho các tham biến có tên

def namedexample(x,y=2,z='abc'):
   print x,y,z
```

Ở đây **x** là một tham biến thường, còn **y** và **z** là các *tham biến có tên*. Điều đó có nghĩa là nếu chúng không được chỉ rõ trong lệnh gọi hàm thì chúng sẽ là các giá trị mặc định lần lượt là 2 và 'abc'. Nếu ta muốn gán giá trị cụ thể cho chúng khi gọi hàm, ta phải nêu tên chúng. Dưới đây là ví dụ về một số cách gọi hàm này:

```
# testne.py; minh họa cho các tham biến có tên

import ne

ne.namedexample(5)
ne.namedexample(5,z=12)
ne.namedexample(5,y=3,z=12)
```

Và sau đây là kết quả được in ra:

```
% python testne.py
5 2 abc

5 2 12
5 3 12
```

Điều này thực sự có ích trong các trường hợp khi ta cần một hàm với nhiều tham biến nhưng tham biến lại thường có cùng một giá trị

[30] Một số lập trình viên C/C++ có thể nhận thấy cách này giống như lệnh **sprintf()** của thư viện C.

13 Ví dụ về các cấu trúc dữ liệu trong Python

Dưới đây là một lớp có nhiệm vụ xử lí một cây nhị phân.

```
# bintree.py, một mô-đun quản lí các cây nhị phân đã sắp xếp, các
# giá trị được lưu có dạng bất kì miễn là tồn tại một quan hệ sắp xếp

# Ví dụ này chỉ giới thiệu lệnh nhằm chèn thêm và in cây nhị phân,
# nhưng bạn có thể bổ sung lệnh xóa, v.v...

class treenode:
   def  __init __(self,v):
      self.value = v;
      self.left = None;
      self.right = None;
   def ins(self,nd):  # chèn thêm nhánh nd vào cây có gốc đặt tại self
      m = nd.value
      if m < self.value:
         if self.left == None:
            self.left = nd
         else:
            self.left.ins(nd)
      else:
         if self.right == None:
            self.right = nd
         else:
            self.right.ins(nd)
   def prnt(self):  # in cây con có gốc tại self
      if self.value == None: return
      if self.left != None: self.left.prnt()
      print self.value
      if self.right != None: self.right.prnt()

class tree:
   def  __init __(self):
      self.root = None
   def insrt(self,m):
      newnode = treenode(m)
      if self.root == None:
         self.root = newnode
         return
      self.root.ins(newnode)
```

Và sau đây là đoạn chạy thử

```
# trybt1.py:  chạy thử file bintree.py
```

```
# usage:  python trybt.py cac_so_can_dien_vao

import sys
import bintree

def main():
    tr = bintree.tree()
    for n in sys.argv[1:]:

       tr.insrt(int(n))
    tr.root.prnt()

if  __name __ == ' __main __': main()
```

Một điều hay của Python là ta có thể sử dụng lại mã lệnh cho các đối tượng không phải kiểu số miễn là chúng có thể so sánh được. (Hãy nhớ lại phần 9.) Do vậy, ta có thể áp dụng các lớp **tree** và **treenode** một cách tương tự với kiểu chuỗi:

```
# trybt2.py:  chạy thử file bintree.py

# usage:  python trybt.py cac_chuoi_can_them

import sys
import bintree

def main():
    tr = bintree.tree()
    for s in sys.argv[1:]:
       tr.insrt(s)
    tr.root.prnt()

if  __name __ == ' __main __': main()

% python trybt2.py abc tuv 12
12
abc
tuv
```

hay thậm chí là

```
# trybt3.py:  chạy thử file bintree.py

import bintree

def main():
    tr = bintree.tree()
```

```
    tr.insrt([12,'xyz'])
    tr.insrt([15,'xyz'])
    tr.insrt([12,'tuv'])
    tr.insrt([2,'y'])
    tr.insrt([20,'aaa'])
    tr.root.prnt()

if  __name __ == ' __main __': main()
```

```
% python trybt3.py
[2, 'y']
[12, 'tuv']
[12, 'xyz']
[15, 'xyz']
[20, 'aaa']
```

13.1 Sử dụng các kiểu cú pháp đặc biệt

Trong ví dụ của phần 7, đáng chú ý là dòng lệnh sau:

```
for f in [a,b]:
```

trong đó **a** và **b** là các đối tượng kiểu file văn bản. Điều này cho thấy thực tế các phần tử của danh sách không nhất thiết phải có kiểu vô hướng.[31] Quan trọng hơn, nó cho thấy phương pháp sử dụng Python hiệu quả sẽ giúp ta tránh khỏi các vòng lặp kiểu cổ điển và các phép toán trên mảng như trong C. Chính điều này giúp cho mã lệnh trở nên rõ ràng, sáng sủa và đây là một ưu điểm thực sự của Python.

Nói chung bạn không nên sử dụng vòng lặp **for** kiểu C/C++ — nghĩa là một biến chỉ số chẳng hạn, **j** được kiểm tra so với giới hạn trên, như $\mathbf{j} < \mathbf{10}$, và việc tăng biến này sau mỗi vòng lặp (chẳng hạn **j++**).

Thực sự là bạn có thể tránh các vòng lặp dạng minh bạch, và bạn nên làm Như vậy mỗi khi có thể được. Chẳng hạn, đoạn lệnh

```
self.lines = self.fh.readlines()
self.nlines = len(self.lines)
```

trong cùng một chương trình sẽ sáng sủa hơn nhiều so với cách viết như trong C, vốn gắn với việc dùng một vòng lặp. Vòng lặp này sẽ in lần lượt từng dòng trong file, và tăng biến **nlines** sau mỗi vòng lặp.[32]

Một cách khác cũng rất hay nhằm tránh các vòng lặp là sử dụng đặc điểm **lập trình hàm** của Python, được bàn đến trong phần 14.

Tận dụng các cú pháp riêng của Python, theo các *Pythonista*, là cách giải quyết vấn đề theo *kiểu Python (Pythonic).*

[31] Và chúng cũng không nhất thiết phải có kiểu giống nhau. Có thể trộn lẫn nhiều kiểu trong cùng một danh sách.

[32] Bạn cũng cần lưu ý cách tham chiếu tới một đối tượng khác từ trong một đối tượng **self.fh**.

14 Các đặc điểm của lập trình hàm

Các đặc điểm này cho ta cách giải quyết vấn đề nhanh chóng và kín kẽ dù rằng ta vẫn có thể giải quyết theo các cách khác cơ bản hơn, nhưng mã lệnh sẽ không ngắn gọn và dễ viết, dễ sửa bằng.

Ngoài đặc điểm đầu tiên kể ra ở đây (các hàm Lambda), thì các đặc điểm còn lại sẽ triệt tiêu sự xuất hiện tường minh của các vòng lặp và các chỉ định đến từng phần tử trong dãy. Như đã nêu trong phần 13.1, điều này làm cho mã lệnh sáng sủa hơn.

14.1 Các hàm Lambda

Các *hàm Lambda* giúp ta có cách mới định nghĩa các hàm rất ngắn. Chúng giúp bạn tránh làm rối mã lệnh với những dòng lệnh chỉ được gọi đúng có một lần. Chẳng hạn:

```
>>> g = lambda u:u*u
>>> g(4)
16
```

Hãy chú ý rằng đây KHÔNG phải là cách sử dụng điển hình của các hàm lambda; đó chỉ là một minh họa cho cú pháp. Thường người sử dụng hàm lambda không định nghĩa nó lẻ loi như vậy; thay vào đó là bên trong các hàm như **map()** và **filter()** như dưới đây.

Sau đây là một ví dụ khác thực tế hơn, viết lại ví dụ sắp xếp trong phần 6.2.3:

```
>>> x = [[1,4],[5,2]]
>>> x
[[1, 4], [5, 2]]
>>> x.sort()
>>> x
[[1, 4], [5, 2]]
>>> x.sort(lambda u,v: u[1]-v[1])
>>> x
[[5, 2], [1, 4]]
```

Dạng tổng quát của hàm lambda là:

```
lambda thamBiến 1, thamBiến 2, ...: biểuThức
```

Vậy là nó cho phép ta đưa vào nhiều tham biến, tuy vậy phần thân hàm chỉ được phép là một biểu thức mà thôi.

14.2 Mapping

Hàm **map()** có tác dụng chuyển đổi từ một chuỗi này sang chuỗi khác bằng cách áp dụng cùng một hàm lần lượt đối với mọi phần tử trong dãy, ví dụ:

```
>>> z = map(len,["abc","clouds","rain"])
>>> z
[3, 6, 4]
```

Như vậy là với cách này chúng ta đã tránh phải viết ra một vòng lặp **for**, do đó mã lệnh sẽ sáng sủa hơn một chút, dễ đọc và viết hơn.[33]

Trong ví dụ nêu trên ta đã sử dụng hàm có sẵn, **len()**. Ta cũng có thể sử dụng hàm mình tự viết; điều này thường được thực hiện qua hàm lambda, chẳng hạn:

```
>>> x = [1,2,3]
>>> y = map(lambda z: z*z, x)
>>> y
[1, 4, 9]
```

Việc bắt buộc thân hàm lambda chỉ được có một biểu thức là một điều kiện khá hạn chế, chẳng hạn nó khong cho phép sử dụng cấu trúc if-then-else. Nếu bạn thực sự muốn làm như vậy, bạn có thể sử dụng mẹo. Ví dụ như bạn muốn

```
if u > 2: u = 5
```

ta có thể làm như sau:

```
>>> x = [1,2,3]
>>> g = lambda u: (u > 2) * 5 + (u <= 2) * u
>>> map(g,x)
[1, 2, 5]
```

Rõ ràng đây không phải là một biện pháp khả thi trừ những trường hợp đơn giản. Đối với các trường hợp phức tạp, chúng ta có thể sử dụng một hàm không phải lambda. Chẳng hạn, sau đây là một cách cải tiến cho chương trình trong phần 4.1:

```
import sys

def checkline(l):
   global wordcount
   w = l.split()
   wordcount += len(w)

wordcount = 0
f = open(sys.argv[1])
flines = f.readlines()
linecount = len(flines)
map(checkline,flines)  # thay thế vòng lặp 'for' cũ
print linecount, wordcount
```

[33] Một lần nữa chú ý rằng nếu ta không gán cho **z**, danh sách [3,6,4] vẫn sẽ được in ra. Trong chế độ tương tác lệnh, bất kì biểu thức nào của Python mà không phải lệnh gán đều in ra giá trị của kết quả.

Chú ý rằng **l** bây giờ là một tham biến cho **checkline()**.

Dĩ nhiên là đoạn mã này có thể được rút ngắn thêm, với phần trung tâm của chương trình chính nói trên được đổi thành:

```
map(checkline,open(sys.argv[1]).readlines)
```

Nhưng điều này sẽ dẫn đến việc đọc và gỡ rối chương trình rất khó khăn.

14.3 Lọc

Hàm **filter()** hoạt động giống như **map()**, chỉ khác là nó lấy những phần tử trong dãy thỏa mãn một điều kiện nhất định. Hàm áp dụng **filter()** phải trả giá trị boolean (true hoặc false). Chẳng hạn:

```
>>> x = [5,12,-2,13]
>>> y = filter(lambda z: z > 0, x)
>>> y
[5, 12, 13]
```

Một lần nữa, điều này cho phép ta tránh viết vòng **for** và một lệnh **if**.

14.4 List Comprehension

Điều này cho phép ta rút ngắn một vòng lặp **for** mà cho kết quả là một danh sách. Lấy ví dụ:

```
>>> x = [(1,-1), (12,5), (8,16)]
>>> y = [(v,u) for (u,v) in x]
>>> y
[(-1, 1), (5, 12), (16, 8)]
```

Viết thế này sẽ gọn hơn so với đầu tiên là gán [] cho **y**, sau đó là vòng lặp **for** mà trong đó ta gọi **y.append()**

14.5 Chiết giảm

Hàm **reduce()** được sử dụng để thực hiện cộng dồn hoặc một phép toán số học kiểu như vậy đối với toàn bộ danh sách. Chẳng hạn,

```
>>> x = reduce(lambda x,y: x+y, range(5))
>>> x
10
```

Ở đây **range(5)** dĩ nhiên là [0,1,2,3,4]. Trước hết, hàm **reduce()** cộng hai phần tử đầu tiên của [0,1,2,3,4], tức là 0 đóng vai trò **x** và 1 đóng vai trò **y**. Tổng của chúng là 1. Sau đó thì tổng đó, 1, lại đóng vai trò của

x và phần tử tiếp theo của [0,1,2,3,4], tức là 2, đóng vai trò của **y**, chúng cho ta kết quả là 3, và cứ như vậy. Cuối cùng **reduce()** hoàn thành xong nhiệm vụ của nó và trả lại kết quả là 10.

Một lần nữa, điều này cũng cho phép ta tránh một vòng lặp **for**, cùng với một câu lệnh khởi tạo **x** bằng 0 trước vòng **for**.

15 Các biểu thức phát sinh

Gợi nhớ lại rằng một lợi thế lớn của thể lặp là ta có thể thường xuyên sử dụng chúng mà không tạo ra một danh sách lớn trong bộ nhớ. Các biểu thức phát sinh mới được giới thiệu trong Python 2.4, cho ta sự cải tiến tương tự đối với list comprehension.

Xét ví dụ ngắn sau đây, trong đó ta ôn lại list comprehension và giới thiệu các biểu thức phát sinh.

```
>>> x = [1,2,4,12,5]
>>> [i for i in x]
[1, 2, 4, 12, 5]
>>> [i for i in x if i > 4]
[12, 5]
>>> sum(i for i in x if i > 4)
17
>>> min(i for i in x if i > 4)
5
```

Nói cách khác, một biểu thức phát sinh có cùng dạng như một list comprehension, chỉ khác ở hai điểm sau:

- biểu thức được nằm trong cặp ngoặc tròn thay vì ngoặc vuông
- hàm nào áp dụng được cho các thể lặp cũng có thể được áp dụng cho biểu thức[34]

Tiện thể cần nói, hàm **min()** cũng có thể được sử dụng với các tham biến thông thường:

```
>>> min(5,12,3)
3
>>> y = [5,12,4]
>>> min(y)
4
```

Hơn nữa, nó còn có thể được sử dụng đối với bất cứ thứ gì so sánh được theo thứ tự trước-sau:

```
>>> min('def','cxyg')
'cxyg'
>>> min([4,'abc'],[8,5],[3,200])
[3, 200]
```

Dĩ nhiên là **max()** cũng tương tự. Nếu ta kết hợp chúng lại trong các biểu thức phát sinh thì sẽ rất có lợi.

[34]Chú ý rằng cặp ngoặc tròn xung quanh biểu thức cũng đóng vai trò ngoặc cho hàm.

A Gỡ lỗi

Bạn đừng nên gỡ lỗi bằng cách chỉ thêm các dòng lệnh **print** vào trong chương trình. Thay vào đó, hãy sử dụng công cụ gỡ lỗi. Nếu bạn không thường xuyên sử dụng công cụ này thì sẽ phải lập trình rất khó khăn và tốn nhiều thời gian. Xem bài trình bày của tôi tại địa chỉ `http://heather.cs.ucdavis.edu/~matloff/debug.html`.

A.1 Công cụ gỡ lỗi sẵn có trong Python, PDB

Bộ gỡ lỗi sẵn có trong Python, PDB, tương đối sơ khai, nhưng tôi vẫn nêu ra dưới đây theo sự tăng dần mức độ hữu dụng:

- Dạng cơ bản
- Dạng cơ bản tăng cường sử dụng thêm macro có chiến lược
- Dạng cơ bản kết hợp với bộ gỡ lỗi đồ họa DDD.

A.1.1 Dạng cơ bản

Bạn sẽ tìm thấy PDB trong thư mục con **lib** trong hệ thống file Python. Chẳng hạn, trên hệ điều hành Unix, nó có thể ở một chỗ như là **/usr/lib/python2.2**, **/usr/local/lib/python2.4**,v.v... Để gỡ lỗi một script **x.py**, hãy gõ lệnh:

```
% /usr/lib/python2.2/pdb.py x.py
```

(Nếu như **x.py** có các tham số dòng lệnh, chúng phải được đặt sau tên **x.py** trên dòng lệnh.)

Dĩ nhiên là vì bạn sẽ sử dụng PDB nhiều, tốt hơn là hãy đặt cho nó bí danh (ví dụ trong Unix):

```
alias pdb /usr/lib/python2.2/pdb.py
```

như vậy bạn chỉ cần viết một cách đơn giản

```
% pdb x.py
```

Một khi bạn đã ở trong PDB, hãy đặt điểm dừng đầu tiên, chẳng hạn ở dòng 12:

```
b 12
```

Ấn **c** ("continue"), bằng cách này bạn vào file **x.py** và dừng lại ở chỗ có điểm dừng. Sau đó tiếp tục như bình thường, với các thao tác tương tự như GDB:[35]

[35]Trong một số trường hợp, các lệnh này sẽ không hoạt động cho đến tận khi bạn thực sự vào bên trong mã nguồn cần gỡ lỗi. Chẳng hạn, nếu gõ **l** ("list" các dòng) quá sớm sẽ gây ra thông báo lỗi.

- **b** (“break”) : đặt một điểm dừng
- **tbreak** : đặt một điểm-dừng-một-lần
- **l** (“list”) : liệt kê một số dòng trong mã nguồn
- **n** (“next”) : nhảy đến dòng tiếp theo, bỏ qua mã lệnh trên dòng hiện tại
- **s** (“subroutine”) : nhảy đến dòng tiếp theo nhưng có duyệt qua từng lệnh của hàm
- **c** (“continue”) : tiếp tục thực hiện cho đến tận khi gặp điểm dừng mới
- **w** (“where”) : nhận báo cáo về tình trạng stack
- **u** (“up”) : di chuyển lên một mức trong stack, chẳng hạn để truy vấn một biến địa phương tại đó
- **d** (“down”) : di chuyển xuống một mức trong stack
- **j** (“jump”) : nhảy đến một dòng khác mà *không* can thiệp đến mã lệnh được thực hiện
- **h** (“help”) : xem hướng dẫn trực tuyến (ngắn gọn) (chẳng hạn **h b** để xem hướng dẫn về câu lệnh **b**. Đơn giản hơn, gõ **h** để xem danh sách toàn bộ các lệnh.
- **q** (“quit”) : thoát khỏi PDB

Bạn nên dùng lệnh **help pdb** ít nhất một lần. Nó sẽ cho biết toàn bộ thông tin về các tính năng khác nhau của PDB.

Khi vào PDB, dấu nhắc `(Pdb)` xuất hiện.

Nếu bạn có một chương trình gồm nhiêu file, các điểm dừng sẽ có dạng `module_tên:số_TT_dòng`. Chẳng hạn, bạn có chương trình chính là **x.py**, và nó nhập file **y.py**. Bạn thiết lập một điểm dừng ở dòng thứ 8 trong file **y.py** như sau:

```
(Pdb) b y:8
```

Dù vậy, cần chú ý rằng điều này chỉ thực hiện được khi **y** thực sự được nhập từ **x**.

Chú ý rằng khi đang chạy PDB, bạn cũng đồng thời chạy Python trong chế độ tương tác dòng lệnh. Do đó, bạn có thể gõ bất kì dòng lệnh Python nào trên dấu nhắc của PDB. Bạn có thể đặt các biến, gọi hàm, v.v... Điều này rất có lợi trong một vài trường hợp sẽ được chỉ ra sau đây.

Chẳng hạn, dù PDB đã có lệnh **p** nhằm in ra giá trị của các biến và biểu thức, thường thì nó không cần thiết. Vì sao? Trong chế độ tương tác lệnh của Python ta chỉ cần gõ tên của biến hay biểu thức là kết quả sẽ được hiển thị trên màn hình—giống hệt như lệnh **p** đã làm. Như vậy, ta không cần phải gõ ‘p’.

Vậy là nếu **x.py** bao gồm một biến **ww** và khi chạy PDB; thay vì gõ

```
(Pdb) p ww
```

bạn chỉ cần gõ

```
ww
```

và giá trị của **ww** sẽ được in ra trên màn hình.[36]

Nếu bạn có chương trình với một cấu trúc dữ liệu phức tạp, bạn có thể viết một hàm để in ra màn hình toàn bộ các phần khác nhau của cấu trúc đó. Sau này, vì PDB cho phép bạn thực hiện bất kì câu lệnh Python nào từ dấu nhắc PDB, bạn có thể đơn giản là gọi hàm đó từ dấu nhắc, từ đó có thể in được một cách cụ thể hơn đối với từng ứng dụng.

Sau khi chương trình của bạn kết thúc hoặc là trong PDB hay một lỗi thực hiện, bạn có thể chạy lại nó mà không cần thoát khỏi PDB–điều này quan trọng khi bạn không muốn mất các điểm dừng–bởi việc ấn **c**. Chú ý rằng bạn có thể thường xuyên (dù không phải luôn luôn) bắt buộc chương trình phải kết thúc, thông qua lệnh **j**, nhảy đến một câu lệnh nào đó ở cuối chương trình, và sau đó ấn **c**. Hơn nữa, nếu tiếp sau đó bạn đã sửa đổi mã lệnh thì những sửa đổi này sẽ được phản ánh trong PDB.[37]

A.1.2 Lưu ý về các lệnh Next và Step

Nếu bạn ra lệnh cho PDB thực hiện Step đơn kiểu như **n** khi đang dừng trên một dòng lệnh Python có nhiều thao tác, bạn cần phải lặp lại lệnh **n** nhiều lần (hoặc thiết lập một điểm dừng tạm thời để nhảy qua dòng đó).

Thí dụ,

```
for i in range(10):
```

thực hiện hai nhiệm vụ. Thứ nhất là gọi hàm **range()**, và sau đó đặt biến **i**, bởi vậy bạn cần phải ra lệnh **n** hai lần.

Thế còn lệnh này thì sao?

```
y = [(y,x) for (x,y) in x]
```

Nếu chẳng hạn **x** có 10 phần tử thì bạn phải thực hiện lệnh **n** 10 lần! Ở đây chắc chắn là bạn muốn thiết lập một điểm dừng tạm thời để giải quyết vấn đề.

A.1.3 Sử dụng Macro của PDB

Rõ ràng là đặc tính quá đơn giản của PDB có thể được khắc phục một phần bằng cách tận dụng lệnh **alias**. Cá nhân tôi khuyến khích dùng lệnh này. Chẳng hạn, bạn gõ

```
alias c c;;l
```

[36] Tuy vậy, nếu tên của biến lại trùng với một lệnh của PDB (hoặc từ viết tắt của nó), thì lệnh này sẽ được thực hiện. Thí dụ, nếu bạn có biến **n**, thì việc gõ **n** sẽ cho kết quả là thực hiện lệnh **n[ext]** thay vì là in ra giá trị của **n**. Để in dược giá trị này, bạn phải gõ **p n**.

[37] PDB, như đã thấy, chỉ là một chương trình Python. Khi bạn khởi động lại, nó sẽ nhập lại mã nguồn của bạn.

Bên cạnh đó, lí do các điểm dừng của bạn được lưu lại là vì chúng cũng là các biến trong PDB. Cụ thể hơn, chúng được lưu trong biến thàn viên có tên **breaks** trong lớp **Pdb** thuộc **pdb.py**. Biến này được thiết lập như một từ điển, với các khóa là tên của các file nguồn **.py**, và các mục là các danh sách của điểm dừng.

Bằng cách này, mỗi khi đỗ lại tại một điểm dừng, bạn sẽ tự động thấy đoạn mã tiếp theo. Đặc điểm này khắc phục rất nhiều cho khiếm khuyết về mặt giao diện đồ họa của PDB.

Điều này thực tế quan trọng đến nỗi bạn nên đặt nó và file khởi động PDB, trong Unix có tên là **$HOME/.pdbrc**.[38] Bằng cách này **alias** luôn có hiệu nghiệm. Bạn cũng có thể làm tương tự đối với các lệnh **n** và **s**:

```
alias c c;;l
alias n n;;l
alias s s;;l
```

Trong PDB cũng có một lệnh **unalias.**

Bạn có thể viết các macro khác dành riêng cho các chương trình đang được gỡ lỗi. Chẳng hạn, một lần nữa hãy giả dụ rằng bạn có một biến tên là **ww** ở trong **x.py**, và bạn muốn kiểm tra giá trị của nó mỗi lần chương trình gỡ lỗi tạm dừng, ở các điểm dừng chẳng hạn. Muốn vậy ta thay đổi bí danh (alias) trên thành:

```
alias c c;;l;;ww
```

Bạn có thể viết bí danh cho bất kì lệnh Python nào, bao gồm cả các tham số. Chẳng hạn trong mã nguồn của bạn có lớp **C**, một trong các biến thành viên của nó là V. Giả sử có một danh sách **L** bao gồm các đối tượng trong lớp **C**. Như vậy ta có thể có một bí danh kiểu như:

```
(Pdb) alias pl for I in L: print I.V
```

Để định nghĩa mỗi bí danh, ta có thể có các tham số, được tham chiếu đến bởi: **%1**, (**%2** v.v...

Hoặc, bạn cũng có thể định nghĩa một hàm nhằm mục đích gỡ lỗi trong chương trình nguồn, chẳng hạn **Dbg()**, và sau đó tạo một bí danh để lần sau đỡ phải gõ:

```
alias dbg Dbg()
```

A.1.4 Sử dụng __dict__

Trong phần A.2.8 dưới đây, ta sẽ cho thấy rằng nếu **c** là một đối tượng thuộc một lớp nào đó, thì gõ **o.__dict__** sẽ in ra toàn bộ các biến thành viên trong đối tượng này. Bạn có thể kết hợp đặc điểm này với chức năng đặt bí danh của PDB, chẳng hạn:

```
alias c c;;l;;o __dict __
```

A.2 Sử dụng PDB với DDD

DDD, có sẵn trong nhiều hệ thống Unix (và có thể download tự do nếu hệ thống của bạn chưa có nó), là một giao diện đồ họa cho nhiều chương trình gỡ lỗi, chẳng hạn GDB (cho C/C++), JDB (cho Java), trình gỡ lỗi có sẵn trong Perl, v.v... Nó còn có thể dùng cho PDB của Python, giúp ta nâng cao hiệu quả khi làm việc.

[38] Python cũng sẽ kiểm tra sự tồn tại của một file như vậy trong thư mục hiện thời.

A.2.1 Chuẩn bị

Ứng dụng đầu tiên của DDD cho PDB được Richard Wolff thực hiện. Ông đã chỉnh sửa file **pdb.py** nhằm mục đích này, tạo thành file PDB mới có tên **pydb.py**. Ban đầu dành cho Python 1.5, tuy nhiên sau này Richard đã cung cấp bản cập nhật quý báu bao gồm các file sau đây:

```
http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydb.py
http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydbcmd.py
http://heather.cs.ucdavis.edu/~matloff/Python/DDD/pydbsupt.py
```

Đặt hai file vào một chỗ nào đó trong thư mục tìm kiếm của bạn, chẳng hạn **/usr/bin**. Đảm bảo rằng bạn cho chúng quyền thực hiện.

A.2.2 Khởi động DDD và mở chương trình

Để bắt đầu, chẳng hạn gỡ lỗi file **fme2.py** trong Phần 6.2.4, trước hết hãy đảm bảo rằng **main()** được thiết lập như được trình bày trong phần đó.

Khi bạn kích hoạt DDD, hãy bảo nó sử dụng PYDB:

```
ddd --debugger /usr/bin/pydb.py
```

Sau đó trong cửa sổ dòng lệnh của DDD, tức là cửa sổ dòng lệnh con của PDB (gần phía dưới cùng), gõ:

```
(pdb) file fme2.py
```

hoặc chon File | Open Source và click đúp chuột vào tên file.

Sau này, khi cần sửa đổi mã nguồn, bạn hãy gõ lệnh

```
(pdb) file fme2.py
```

Hoặc chọn File | Open Source hay File | Open Recent và click đúp vào tên file. Các điểm dừng trong lần chạy cuối cùng sẽ được giữ nguyên.

A.2.3 Các điểm dừng

Khi đặt một điểm dừng, click phải chuột vào đâu đó trong khoảng trống của dòng trong mã nguồn của bạn và chọn Set Breakpoint (hoặc Set Temporary Breakpoint (đặt điểm dừng tạm thời) hoặc Continue to Until Here (tiếp tục đến tận đây), tuy từng trường hợp).

A.2.4 Chạy chương trình nguồn của bạn

Để chạy, hãy chọn Program | Run, điền các thông số trong chương trình của bạn (nếu có) vào hộp "Run with Argument", và click Run trong cửa sổ đó. Bạn sẽ được đề nghị ấn tiếp Continue, mà bạn có thể làm bằng cách ấn Program | Run |, nhưng tiện lợi hơn cả là click vào Continue trong cửa sổ nhỏ có nhiệm vụ tóm tắt dòng lệnh. (Tuy vậy đừng sử dụng lệnh Run ở đó).

Sau đó bạn có thể click Next, Step, Cont v.v... Dấu trỏ đến dòng lệnh đang được thực hiện sẽ có hình chữ 'I', mặc dù hơi mờ nếu như con trỏ chuột không nằm trong phạm vi mã nguồn của cửa sổ DDD.

Cũng cần nói thêm, bạn đừng gọi **sys.argv** trong đoạn mã lệnh đứng tự do của một lớp. Khi chương trình của bạn lần đầu được nạp, mọi đoạn mã lệnh đứng tự do trong lớp sẽ được thực hiện. Nhưng bởi vì các tham số dòng lệnh vẫn chưa được nạp nên kết quả là có lỗi "index out of range" (thứ tự vượt ngoài khoảng cho phép). Hãy tránh điều này bằng các đưa mã lệnh có chứa **sys.argv** hoặc là vào trong một hàm của lớp, hoặc là ra hẳn phía ngoài lớp.

A.2.5 Theo dõi các biến

Bạn có thể xem giá trị của một biến bằng cách di con trỏ chuột tới bất kì chỗ nào có tên biến đó trong cửa sổ mã nguồn.

Như đã đề cập đến trong Phần A.1.4, nếu **o** là một đối tượng của một lớp nào đó, thì việc gõ **o.__dict__** sẽ in ra toàn bộ các biến thành viên trong lớp này. Trong DDD, bạn có thể làm điều này dễ dàng hơn như sau. Đơn giản là đặt biểu thức đó sau dấu chú thích:

```
# o. __dict __
```

và sau đó mỗi khi muốn xem xét giá trị các biến thành viên của **o**, hãy đưa con trỏ chuột đến biểu thức đó trong dấu chú thích!

Tận dụng các đặc điểm của DDD, ta có thể cho hiển thị một biến một cách liên tục. Đơn giản là click phải chuột lên bất kì chỗ nào có biến đó, và sau đó chọn Display.

A.2.6 Các vấn đề hỗn hợp

Được phát triển với ý định ban đầu dành cho C/C++, DDD không phải luôn phù hợp với Python. Nhưng vì DDD chuyển đổi các lệnh mà bạn click thành các lệnh cho PDB, như bạn thấy trong cửa sổ dòng lệnh của DDD; nên bất cứ việc nào bạn không thể thực hiện với DDD, bạn đều có thể gõ các lệnh PDB vào cửa sổ dòng lệnh.

Cũng cần nói thêm là PYDB, phiên bản sửa đổi của PDB được dùng trong DDD, không có lệnh **alias**.

A.2.7 Một số hỗ trợ gỡ lỗi có sẵn trong Python

Có một số hàm có sẵn của Python mà bạn có thể thấy hữu dụng trong quá trình gỡ lỗi.

A.2.8 Thuộc tính `__dict__`

Hãy nhớ lại rằng các thực thể lớp được vận hành như những từ điển. Nếu có một thực thể lớp tên là **i**, bạn có thể xem từ điển của nó thông qua **i.__dict__**. Nó sẽ cho thấy giá trị của toàn bộ các biến thành viên trong lớp.

A.2.9 Hàm `id()`

Đôi lúc bạn cần biết địa chỉ thực tế của một đối tượng nào đó trong bộ nhớ. Chẳng hạn, có hai biến mà bạn nghĩ rằng chúng cùng chỉ đến một đối tượng, nhưng không chắc chắn. Phương thức **id()** sẽ cho bạn địa chỉ của đối tượng. Chẳng hạn:

```
>>> x = [1,2,3]
>>> id(x)
-1084935956
>>> id(x[1])
137809316
```

(Đừng băn khoăn về địa chỉ âm, vì thực tế là địa chỉ này có số quá lớn và được coi là một số nguyên bù (*two's complement*)).

A.2.10 Các công cụ / IDE gỡ lỗi khác

Bộ gỡ lỗi Winpdb mới The new Winpdb debugger(`www.digitalpeers.com/pythondebugger/`),[39] có vẽ rất tốt. Đặc biệt là đối với tôi vì nó được dựa trên một bộ gỡ lỗi cũ hơn có tên là RPDB vốn xử lí các `thread` của Python rất tốt. Tuy vậy, tài liệu của Winpdb, đến tận tháng 1 năm 2006 còn rất sơ sài, và tôi chưa có dịp để dùng nó. Hãy tìm bản cập nhật (nếu có), tại `http://heather.cs.ucdavis.edu/~matloff/rpdb.html`.

Cá nhân tôi không thích các môi trường phát triển tích hợp (IDE). Chúng thường tải rất chậm và thường không cho tôi dùng chương trình soạn thảo văn bản ưa thích,[40] và theo quan điểm của tôi các IDE không có từng ấy chức năng. Tuy vậy, nếu bạn thích IDE thì sau đây là một số gợi ý:

Bạn có thể muốn thử IDLE, công cụ IDE kèm sẵn với gói Python, dù rằng tôi thấy nó có nhiều khiếm khuyết. Nhưng ngoài ra tôi cũng nghe ý kiến tốt về các IDE miễn phí sau: Boa Constructor, `http://boa-constructor.sourceforge.net/`; Eclipse với phần mở rộng Python, `http://www.python.org/moin/EclipsePythonIntegration`; và JEdit, `http://www.jedit.org/`. Tôi có một bài viết hướng dẫn về Eclipse tại `http://heather.cs.ucdavis.edu/~matloff/eclipse.html`. Còn trong số các phần mềm thương mại, tôi luôn ưa chuộng Wing Edit, `http://wingware.com/wingide/platforms`, công cụ mà tôi sử dụng một thời gian ngắn cho Java.

[39] Không như tên gọi của nó, công cụ này không chỉ dành riêng cho các máy chạy Microsoft Windows.

[40] Tôi dùng vim, nhưng điều căn bản là tôi muốn dùng một chương trình soạn thảo cho tất cả mọi việc—lập trình, soạn thảo, e-mail, phát triển trang Web, v.v...

A.3 Tài liệu trực tuyến

A.3.1 Hàm dir()

dir() là một hàm rất tiện dụng cho phép ta xem nhanh những thành phần có trong một đối tượng hoặc hàm. Bạn nên sử dụng nó thường xuyên.

Để minh họa, trong ví dụ của Phần 7, chẳng hạn chúng ta dừng lại ở dòng:

```
print "Số file văn bản đang mở là: ", textfile.ntfiles
```

Sau đó đột nhiên chúng ta muốn kiểm tra một số điều với hàm **dir()**, chẳng hạn:

```
(Pdb) dir()
['a', 'b']
(Pdb) dir(textfile)
['__doc__', '__init__', '__module__', 'grep', 'wordcount', 'ntfiles']
```

Ngay sau khi bạn khởi động Python, có rất nhiều thứ đã được tải. Chúng ta hãy xem:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
'Exception', 'False', 'FloatingPointError', 'FutureWarning', 'IOError',
'ImportError', 'IndentationError', 'IndexError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None',
'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError',
'OverflowWarning', 'PendingDeprecationWarning', 'ReferenceError',
'RuntimeError', 'RuntimeWarning', 'StandardError', 'StopIteration',
'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_',
'__debug__', '__doc__', '__import__', '__name__', 'abs', 'apply',
'basestring', 'bool', 'buffer', 'callable', 'chr', 'classmethod', 'cmp',
'coerce', 'compile', 'complex', 'copyright', 'credits', 'delattr',
'dict', 'dir', 'divmod', 'enumerate', 'eval', 'execfile', 'exit',
'file', 'filter', 'float', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'intern', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long', 'map',
'max', 'min', 'object', 'oct', 'open', 'ord', 'pow', 'property', 'quit',
'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

Vậy đó, có một danh sách tất cả các hàm có sẵn và những thuộc tính khác mà bạn thấy!

Nếu bạn muốn biết những hàm nào và thuộc tính nào khác có trong kiểu từ điển:

```
>>> dir(dict)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__',
'__doc__', '__eq__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setitem__', '__str__', 'clear', 'copy',
'fromkeys', 'get', 'has_key', 'items', 'iteritems', 'iterkeys',
'itervalues', 'keys', 'pop', 'popitem', 'setdefault', 'update',
'values']
```

Giả dụ chúng ta muốn tìm xem các phương thức và thuộc tính nào gắn với kiểu chuỗi. Như bạn đã biết qua phần 6.2.2, chuỗi đã trở thành một lớp có sẵn trong Python, bởi vậy ta không thể gõ:

```
>>> dir(string)
```

Mà thay vào đó phải dùng một đối tượng chuỗi bất kì:

```
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__str__', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'index', 'isalnum', 'isalpha',
'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower', 'lstrip', 'replace', 'rfind', 'rindex', 'rjust', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

A.3.2 Hàm help()

Chẳng hạn, hãy tìm hiểu về phương thức **pop()** của danh sách:

```
>>> help(list.pop)

Help on method_descriptor:

pop(...)
    L.pop([index]) -> item -- remove and return item at index (default
last)
(END)
```

và phương thức **center()** của chuỗi:

```
>>> help(''.center)
Help on function center:

center(s, width)
    center(s, width) -> string

    Return a center version of s, in a field of the specified
    width. padded with spaces as needed.  The string is never
    truncated.
```

Ấn 'q' để thoát khỏi trang hướng dẫn.

Bạn cũng có thể lấy thông tin bằng cách sử dụng **pydoc** tại dấu nhắc lệnh của Unix, chẳng hạn:

```
% pydoc string.center
[...như trên]
```

A.4 Giải thích về biện pháp xử lí biến của lớp cũ

Trước khi Python ra phiên bản 2.2, không có một hỗ trợ nào dành cho các phương thức của lớp. Nhưng có một biện pháp xử lí đơn giản được trình bày ở `http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52304`.

Bây giờ vì Python đã cho phép các phương thức của lớp (thực tế là hai loại), chúng ta không cần một biện pháp xử lí như vậy trong lập trình, nhưng nó là ví dụ hay cho ta thấy nguyên tắc hoạt động của tham chiếu đối tượng như thế nào. Chúng ta hãy tìm hiểu điều này:

Trong chương trình **tfe.py** của phần 7, có thể thêm

```
class ClassMethod:
   def __init__(self,MethodName):
      self.__call__ = MethodName

class textfile:
   (lots of stuff omitted)
   def printntfiles():
      print textfile.ntfiles
   tmp = ClassMethod(printntfiles)
   printntfiles = tmp

textfile.printntfiles()
```

(Tôi đã thêm vào một dòng—dòng với một phép gán cho **tmp**—để rõ ràng hơn, như bạn sẽ thấy sau này.)

Nhớ lại trong phần 8, các hàm trong Python đều là các (con trỏ đến) các đối tượng, cũng giống như bất cứ thứ gì khác, và do đó đều có thể gán được. Chẳng hạn, trong đoạn mã dưới đây:

```
>>> def f(x):
...     return x*x
...
>>> g = f
>>> g(3)
9
```

f thực chất chỉ là một con trỏ đến một "đối tượng hàm", vì vậy nếu ta gán **f** cho **g**, thì **g** cùng chỉ đến đối tượng đó, và do vậy **g** cũng là hàm cùng tính năng.

Mặt khác, cũng cần lưu ý rằng các thực thể lớp đều có thể gọi được. Đó chính là phương thức **__call__**(), phần ngầm định của tất cả mọi lớp. Do đó đơn giản là chúng ta vờ như thực thể lớp là một hàm, và sau đó gọi nó. Khi đó, phương thức **__call__**() sẽ được kích hoạt đối với **self** (nghĩa là thực thể lớp mà chúng ta đang "gọi") với bất kì tham số nào được nhập vào. Phương thức **__call__**() được mặc định là rỗng, nhưng chúng ta có thể cho một thông số và sử dụng theo mục đích riêng.

Thí dụ, file nguồn sau đây có tên là **yyy.py**:

```
class u:
   def __init__(self):
      self.r = 8
   def __call__(self,z):
      return self.r*z

def main():
   a = u()
   print a(5)
   a.r = 13
   print a(5)

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

```
python yyy.py
40
65
```

Bây giờ hãy xem dòng lệnh sau đây trong file nguồn đã áp dụng biện pháp xử lí:

```
tmp = ClassMethod(printntfiles)
```

Nó có tác dụng tạo ra một thực thể của lớp **ClassMethod** và gán nó cho **tmp**. Tham số là **printntfiles**, bởi vậy dòng lệnh kiến tạo:

```
self.__call__ = MethodName
```

sẽ có nghĩa là **tmp.__call__** được đặt cho **printntfiles**. Hãy tạm quên đi rằng cả **tmp.__call__** và **printntfiles** đều là hàm; chỉ chú ý là chúng ta đã gán một con trỏ (**printntfiles**) cho cái kia (**tmp.__call__**).

Tuy vậy cả hai cái này dĩ nhiên đều là hàm, và do đó **tmp.__call__** bây giờ alf một con trỏ tới mã của hàm ban đầu **printntfiles**, tức là tới mã:

```
print textfile.ntfiles
```

Và dòng tiếp theo

```
printntfiles = tmp
```

có nghĩa là biến **printntfiles** bây giờ không còn chỉ đến mã lệnh của nó nữa! Thay vào đó, nó chỉ đến một thực thể của lớp **ClassMethod**.

Hãy nhớ rằng, bây giờ **printntfiles** bản thân nó là một biến lớp ở trong lớp **textfile**, nghĩa là nó có tên đầy đủ **textfile.printntfiles**. Ban đầu nó là một hàm, nhưng vì bất cứ thứ gì đều có thể gán được, nên bây giờ nó chỉ đơn giản là một biến, chỉ đến một thực thể của lớp **ClassMethod**.

Điều cần nói là nếu ta gọi **textfile.printntfiles**—mà ta có thể thực hiện được, vì mọi lớp của Python đều "gọi được"!—thì ta đã kích hoạt phương thức **__call__()** của thực thể đó. Và phương thức này, như đã thấy ở trên, chỉ đến mã nguồn gốc ban đầu của hàm **printntfiles**, nghĩa là:

```
print textfile.ntfiles
```

Do vậy mà đoạn mã đó sẽ được thực hiện!

Tòm lại một lệnh có cú pháp như một lời gọi phương thức của một lớp, như

```
textfile.printntfiles()
```

thực tế sẽ đóng vai trò như một phương thức lớp, dù rằng trong Python không có hình thức này.

A.5 Đưa tất cả các biến toàn cục vào trong một lớp

Như đã nói trong phần 5, thay vì sử dụng từ khóa **global**, chúng ta có thể làm rõ ràng và có tổ chức hơn bằng cách nhóm tất cả các biến toàn cục vào một lớp. Ở đây, trong file **tmeg.py**, ta sẽ làm như vậy, một cải tiến so với file **tme.py**:

```
# đọc vào file text có tên đã được chỉ định trong dòng lệnh,
# và thông báo số dòng và số từ có trong file

import sys

def checkline():
   glb.linecount += 1
```

```
   w = glb.l.split()
   glb.wordcount += len(w)

class glb:
   linecount = 0
   wordcount = 0
   l = []

f = open(sys.argv[1])
for glb.l in f.readlines():
   checkline()
print glb.linecount, glb.wordcount
```

Chú ý rằng khi chương trình được chạy lần đầu, lớp **glb** sẽ được thực hiện, và thậm chí trước cả khi khởi động **main()**.