Lập trình Python qua các ví dụ và bài tập

Phan V. Phương – THPT Xuân Trường B

https://o2.edu.vn

Mục lục

Chương 0 Lời giới thiệu....4

Chương 1 Mở đầu....5
- 1.1 Lịch sử ngôn ngữ Python....5
- 1.2 Ưu điểm của ngôn ngữ Python....5
- 1.3 Cài đặt....5
- 1.4 Chạy một chương trình Python....7

Chương 2 Các phần tử cơ bản của Python....10
- 2.1 Các từ khóa....10
- 2.2 Tên và cách đặt tên....10
- 2.3 Chú thích....11
- 2.4 Câu lệnh và khối lệnh....11
- 2.5 Một số quy tắc khi viết chương trình....13

Chương 3 Các phép toán....14
- 3.1 Phép gán....14
- 3.2 Phép toán số học....15
- 3.3 Phép toán logic....16
- 3.4 Phép toán so sánh....17

Chương 4 Các kiểu dữ liệu cơ bản....18
- 4.1 Kiểu logic *bool*....18
- 4.2 Kiểu số....19
- 4.3 Kiểu xâu *string*....19
- 4.4 Kiểu danh sách *list*....19
- 4.5 Kiểu từ điển *dict*....20
- 4.6 Kiểu bộ *tuple*....20
- 4.7 Kiểu tập hợp *set*....20
- 4.8 Kiểu *None*....24

Chương 5 Các thủ tục vào ra dữ liệu....25
- 5.1 Thao tác vào ra cơ bản....25
- 5.2 Làm việc với tệp....25

Chương 6 Câu lệnh điều khiển....26
- 6.1 Câu lệnh điều kiện *if*....26
- 6.2 Vòng lặp for....27
- 6.3 Vòng lặp while....27

Chương 7 Hàm....29
- 7.1 Khái niệm hàm....29
- 7.2 Khai báo hàm....29
- 7.3 Lời gọi đến hàm....30
- 7.4 Giá trị trả về của hàm....31
- 7.5 Tham số của hàm....32
- 7.6 Lời gọi đệ quy....32
- 7.7 Biến toàn cục, biến cục bộ....33
- 7.8 Hàm nặc danh....33

Chương 8 Lớp và đối tượng..34
8.1 Khái niệm lớp..34
8.2 Khái niệm đối tượng..34
8.3 Khai báo lớp..34
8.4 Thuộc tính..35
8.5 Phương thức..35
8.6 Tính kế thừa..35
8.7 Tính đa hình..35
8.8 Các phương thức đặc biệt..35
Chương 9 Xử lý ngoại lệ..36
Chương 10 Thư viện..37
10.1 Virtualenv..37
Chương 11 Phụ lục..39
11.1 Mô tả từ khóa qua các ví dụ..39
11.2 Giới thiệu một số thư viện..50

Chương 0 Lời giới thiệu

Trong quá trình tự học về Python, hè năm 2017, tôi có ghi chép lại những kiến thức vụn vặt mà mình thu lượm được từ rất nhiều nguồn tài liệu, bài giảng, video… Thiết nghĩ, Python đang ngày càng chứng tỏ được sức mạnh và vai trò quan trọng của mình trong thế giới các ngôn ngữ lập trình. Và, nguồn tài liệu tiếng Việt về Python còn khá hạn chế, trong khi không phải bạn nào cũng có khả năng đọc hiểu *tốt* tiếng Anh nên tôi mạnh dạn biên soạn tài liệu nhỏ này, coi như một cách ghi bài khi tự học mà thôi.

Về mặt hình thức, tài liệu được soạn bằng **LibreOffice**, vì mới làm quen với phần mềm soạn thảo văn bản này nên tài liệu có một thiếu sót mà tôi chưa khắc phục được. Đó là, phần **Chỉ mục** không sắp xếp được các thuật ngữ theo bảng chữ cái tiếng Việt, chẳng hạn các âm **Đ** lại được cho vào loại kí hiệu và xếp sau cùng, cùng với các kí hiệu khác như + *...

Ngoài ra, thì về mặt nội dung, vì là một tay *mơ* trong thế giới lập trình, và trình độ còn kém cỏi nên tài liệu vẫn còn một vài hạn chế sau:

- Không có kinh nghiệm sử dụng Linux nên các hướng dẫn thực hành trong tài liệu này được thực hiện trên Windows 10.
- Một số chỗ trình bày còn luộm thuộm, khó hiểu, không khoa học hoặc không đầy đủ.
- Đôi khi, để rõ ràng, các từ khóa được dịch sang bằng tiếng Việt, và đi kèm sau đó là nguyên bản bằng tiếng Anh thì từ khóa tiếng Anh sẽ được in nghiêng.
- Bài tập trong cuốn sách này khá ít, do đó bạn có thể vào một số trang để làm thêm bài tập, ở đây xin giới thiệu ba địa chỉ sau:
  - https://www.hackerrank.com
  - https://projecteuler.net
  - http://www.w3resource.com/python-exercises

Mặc dù rất cố gắng nhưng chắc chắn tôi không tránh khỏi sai sót. Rất mong nhận được phản hồi của các bạn để tài liệu ngày càng hoàn thiện hơn. Mọi đóng góp xin gửi về địa chỉ gachduoi@hotmail.com hoặc điện thoại **+84.976.146.213**.

Truy cập website: https://o2.edu.vn/ hoặc https://divin.dev/ để đọc những bài viết mới nhất.

Nam Định, Mùa Thu năm 2021

Chương 1 Mở đầu

1.1 Lịch sử ngôn ngữ Python

Python là một ngôn ngữ lập trình đa năng, được tạo ra bởi **Guido van Rossum** từ năm cuối những năm 1980. Phiên bản đầu tiên được phát hành năm 1991, hiện nay các phiên bản của Python gồm có hai nhánh chính là Python 2.x và Python 3.x. Hiện tại, Python được phát triển trong một dự án mã mở, do tổ chức phi lợi nhuận Python Software Foundation quản lý.

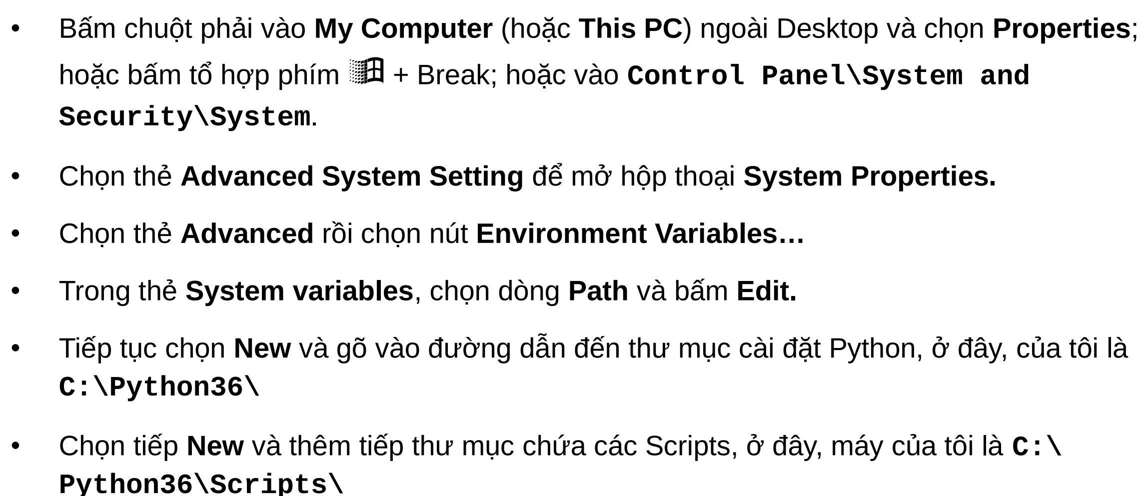
1.2 Ưu điểm của ngôn ngữ Python

Python là một ngôn ngữ lập trình bậc rất cao và có một số đặc điểm nổi bật sau:

- đa mục đích – bạn có thể sử dụng Python để phát triển ứng dụng desktop, web, machine learning, mobile, IoT…;
- hướng đối tượng;
- là ngôn ngữ lập trình dạng thông dịch (Interpreter Language);
- đa nền tảng;
- hoàn toàn tạo kiểu dữ liệu động và dùng cơ chế cấp phát – thu gom bộ nhớ tự động;
- cú pháp đơn giản, dễ đọc, có ít từ khóa, phân tách các khối lệnh (lệnh ghép) bằng khoảng trắng chứ không bằng cặp ngoặc {} như trong C, hoặc các từ khóa `begin, end` như trong Pascal…, không sử dụng dấu chấm phảy `(;)` để kết thúc một lệnh;
- các thư viện hỗ trợ phong phú, có thể tìm được các thư viện phục vụ cho hầu hết mọi nhu cầu của bạn và tất cả đều miễn phí.

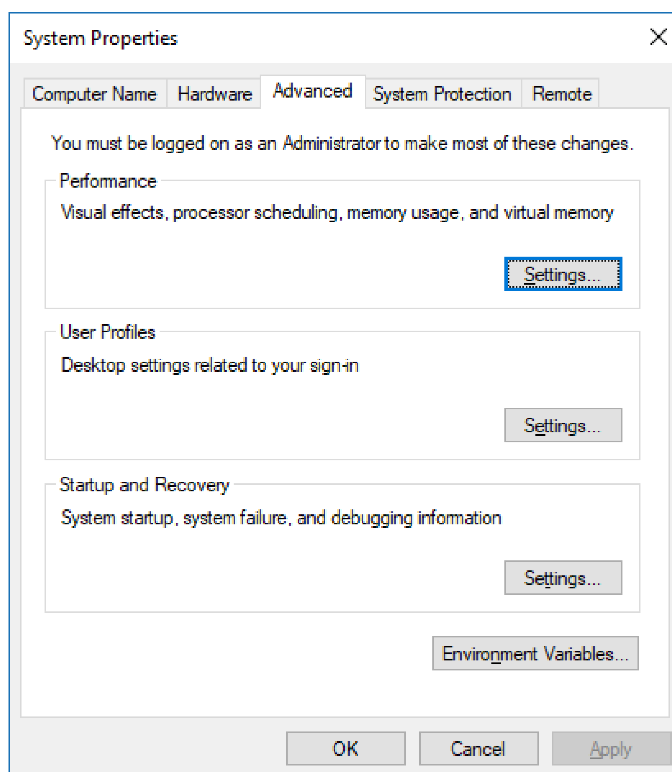
1.3 Cài đặt

Để cài đặt **Python**, bạn vào trang chủ của Python tại `https://python.org/` và tải về phiên bản phù hợp với hệ điều hành đang dùng. Ở đây tôi không đi vào chi tiết cách cài đặt, cá nhân tôi sử dụng phiên bản 3.6 cho Windows 64 bit và cài vào thư mục `C:\Python36`, chỉ lưu ý các bạn khi cài đặt nên tích chọn để đưa **Python** vào biến môi trường (**System Path**). Nếu không, bạn phải thêm thư mục Python vào **System Path** một cách thủ công như sau:



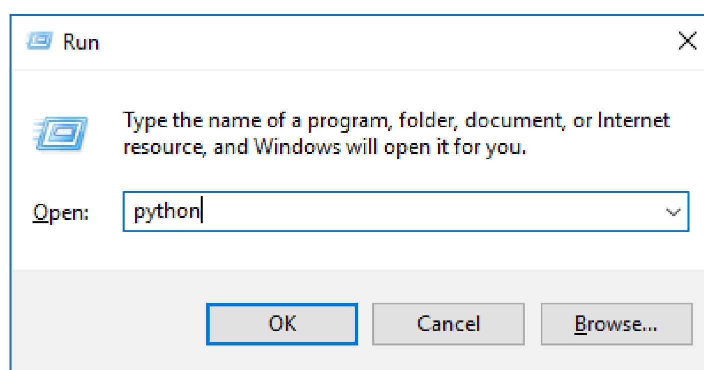
- Bấm chuột phải vào **My Computer** (hoặc **This PC**) ngoài Desktop và chọn **Properties**; hoặc bấm tổ hợp phím ⊞ + Break; hoặc vào **`Control Panel\System and Security\System`**.
- Chọn thẻ **Advanced System Setting** để mở hộp thoại **System Properties.**
- Chọn thẻ **Advanced** rồi chọn nút **Environment Variables…**
- Trong thẻ **System variables**, chọn dòng **Path** và bấm **Edit.**
- Tiếp tục chọn **New** và gõ vào đường dẫn đến thư mục cài đặt Python, ở đây, của tôi là **`C:\Python36\`**
- Chọn tiếp **New** và thêm tiếp thư mục chứa các Scripts, ở đây, máy của tôi là **`C:\Python36\Scripts\`**

- Bấm **OK**.



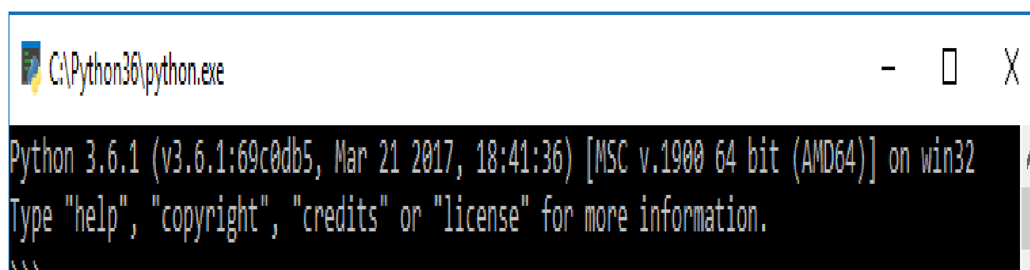
Hình 1: Hộp thoại System Properties

Để kiểm tra đã thêm Python vào **System Path** chưa, bạn mở[1] hộp thoại **Run** của Windows và gõ `python`, sau đó bấm Enter:



Hình 2: Hộp thoại Run

nếu hiện cửa sổ như sau là thành công:



Hình 3: Trình biên dịch Python

1 Để mở hộp thoại Run có thể bấm tổ hợp phím **Windows** + **R**, hoặc bấm Start và gõ run sau đó chọn Run – ở đây tôi sử dụng Windows 10.

Sau khi cài đặt xong trình biên dịch Python, mặc định sẽ có một trình soạn thảo đi kèm là **IDLE**, tuy nhiên trình soạn thảo này khá cơ bản và không hỗ trợ nhiều cho người sử dụng như gợi ý các từ khóa, quản lý project, gỡ lỗi… nên tôi khuyên bạn nên sử dụng thêm một trình soạn thảo như **Notepad++, Sublime Text**, **Visual Studio Code**, **Pycharm**, **Eclipse**… Có rất nhiều chương trình như vậy, cả miễn phí và trả phí, nhưng cá nhân tôi thường sử dụng **Visual Studio Code** của Microsoft, đôi khi cũng sử dụng thêm cả **Sublime Text 3**.

Nếu mới làm quen với Python, bạn có thể cài đặt **Anaconda** tại `https://www.continuum.io` là một môi trường Python đã bao gồm cả trình dịch Python, trình soạn thảo với rất nhiều tính năng cao cấp chuyên dụng giành cho **Data Science,** và được cài sẵn rất nhiều thư viện, đặc biệt là các thư viện cho **Machine Learning**, **Data Science** như `numpy, jupyter, matplotlib`...

1.4 Chạy một chương trình Python

Như đã nói ở trên, Python là ngôn ngữ **thông dịch** – tức là thực hiện một chương trình được viết bởi ngôn ngữ bậc cao bằng cách dịch nó theo từng dòng một –, nên để chạy một chương trình Python, bạn có thể sử dụng một trong hai cách:

- Chạy trực tiếp từng dòng lệnh ở trong chương trình dịch Python,
- Tạo một tệp tin với phần mở rộng là *.py* và chạy tệp này bằng chương trình dịch Python, những tệp này còn được gọi là các kịch bản *script*.

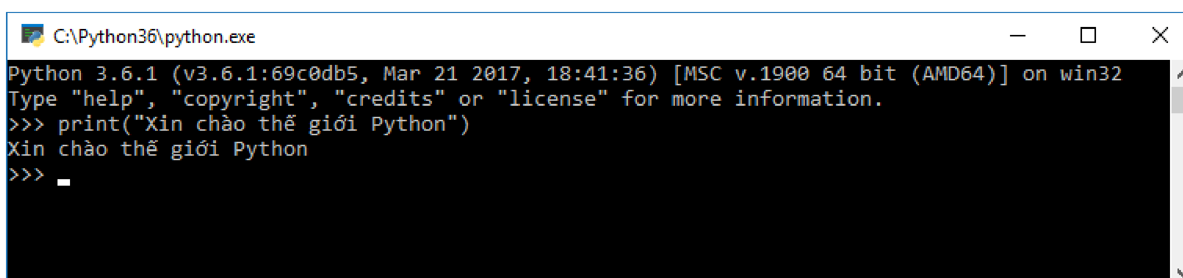
Chúng ta sẽ lần lượt tìm hiểu cả hai cách này.

Chạy trình thông dịch

Cách thứ nhất, bạn chạy trình biên dịch Python tại[2] đường dẫn `C:\Python36\python.exe`, hoặc nếu đã cài đặt Python vào biến môi trường thì chỉ việc mở hộp thoại **Run** hoặc cửa sổ Command Line (từ đây sẽ viết tắt là CMD) và gõ `python`. Nếu thành công, bạn sẽ nhận được một cửa sổ như ở Hình 3. Bây giờ, hãy gõ vào sau dấu nhắc `>>>` dòng lệnh:

```
>>> print("Xin chào thế giới Python!")
```

Sẽ thu được kết quả như hình sau:



Hình 4: Chương trình Python đầu tiên

Chúc mừng! Bạn đã thực hiện thành công chương trình đầu tiên của Python.

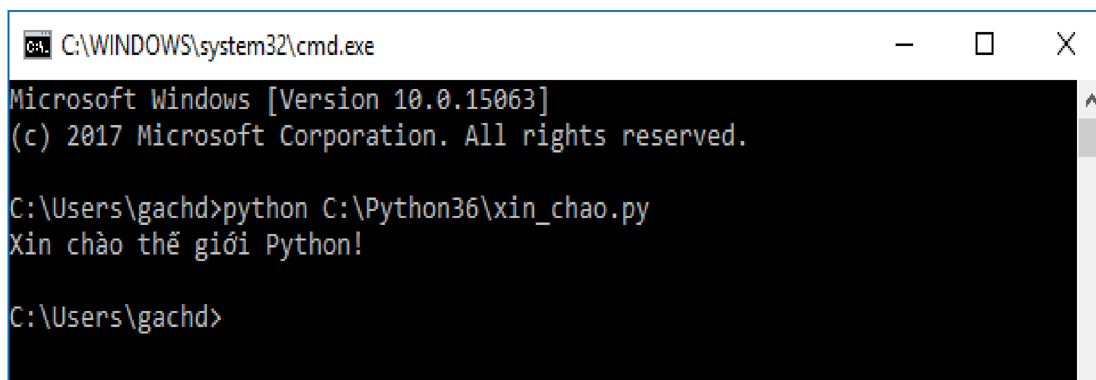
Chạy các script

Cách thứ hai, bạn dùng một trình soạn thảo văn bản bất kì (Notepad chẳng hạn), gõ dòng lệnh

```
print("Xin chào thế giới Python!")
```

2 Ở đây, tôi sử dụng Python phiên bản 3.6, cài đặt vào thư mục C:\Python36\, trong máy của bạn có thể khác.

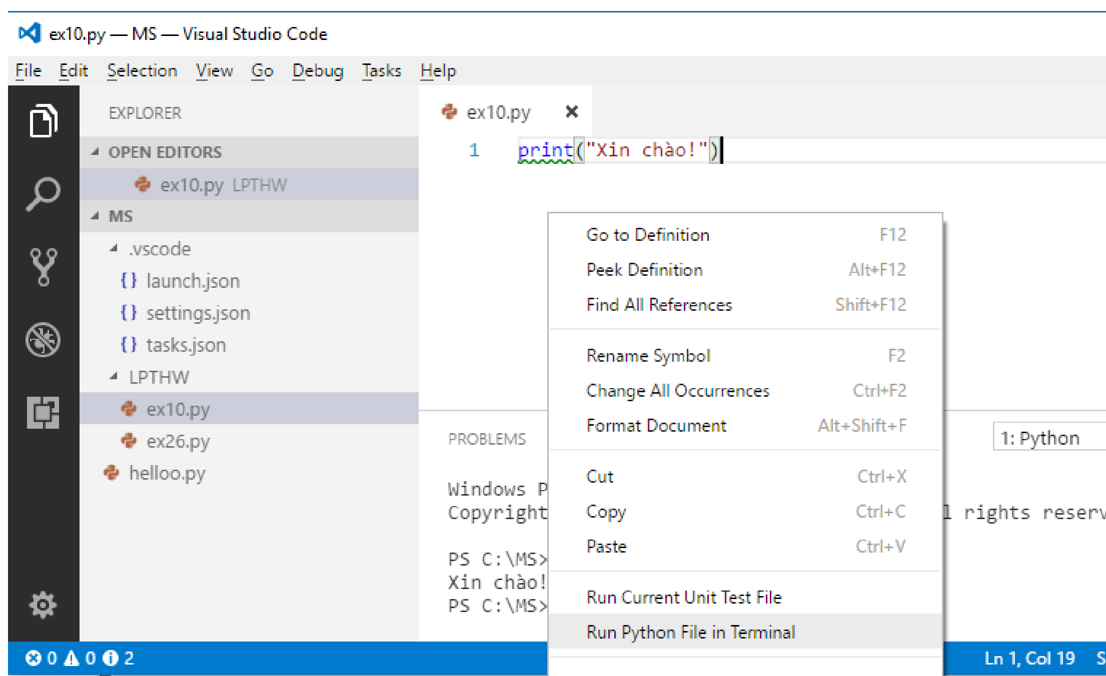
và lưu lại với đuôi mở rộng là .py – mà ta sẽ gọi là các *script*, ví dụ, tôi lưu lại thành tệp xin_chao.py tại thư mục C:\Python36, rồi mở cửa sổ **CMD** và gõ lệnh `python C:\Python36\xin_chao.py` hoặc chỉ cần gõ `C:\Python36\xin_chao.py` sẽ thu được kết quả như hình sau:



Hình 5: Chạy script .py bằng chương trình dịch Python

Ngoài ra, nếu bạn sử dụng **Sublime Text** để soạn *script*, thì có thể chạy một script bằng cách sử dụng phím tắt **Ctrl + B** để build *script* hoặc vào chọn Tools – Build.

Còn nếu sử dụng **Visual Studio Code** để soạn thảo script thì bấm chuột phải vào vùng soạn thảo và chọn **Run Python File in Terminal**.



Hình 6: Chạy script Python trong Visual Studio Code

Trong hai cách trên, cần chú ý rằng, bạn phải lưu *script* vào đĩa cứng trước khi chạy.

Trong tài liệu này, những đoạn mã có dấu `>>>` thì bạn có thể thực hiện trực triếp ở trong chương trình thông dịch Python, mà không cần tạo *script*.

Bài tập

Bài 1. Hãy tự cài đặt chương trình dịch Python phù hợp với hệ điều hành của mình.

Bài 2. Khởi chạy trình thông dịch Python và kiểm tra phiên bản đang sử dụng.

Bài 3. Khởi động chương trình thông dịch Python và tìm hiểu xem các lệnh `help()` có tác dụng gì. Sau đó, sử dụng lệnh `help()` này để tìm hiểu xem kiểu số nguyên int có những phương thức method nào.

Bài 4. Hãy sử dụng nó như một máy tính cầm tay để thực hiện các tính toán đơn giản, với các phép toán cộng +, trừ -, nhân *, chia / và lũy thừa **.

Bài 5. Viết chương trình in ra màn hình dòng chữ sau bằng hai cách, thực hiện trực tiếp trong trình thông dịch Python và viết script.

```
Twinkle, twinkle, little star,
    How I wonder what you are!
                Up above the world so high,
                Like a diamond in the sky.
Twinkle, twinkle, little star,
    How I wonder what you are…
```

Bài 6. Hãy sử dụng trình soạn thảo **Visual Studio Code** và cài thêm các gói hỗ trợ lập trình Python. Google để tìm hiểu thêm.

Chương 2 Các phần tử cơ bản của Python

Như mọi ngôn ngữ lập trình khác, Python có các thành phần cơ bản sau:

- Các từ khóa *keyword*;
- Các biến số và hằng số;
- Các kiểu dữ liệu cơ bản (các cấu trúc dữ liệu được xây dựng sẵn *built-in types*);
- Các câu lệnh và khối lệnh;
- Chú thích.

Trong chương này, chúng ta sẽ giới thiệu qua về các thành phần cơ bản trên. Lưu ý, Python có phân biệt chữ HOA và chữ thường. Đối với Python 3, mặc định, các xâu kí tự *string* sẽ ở dưới dạng unicode. Trong tài liệu này, đa số các ví dụ được viết bằng tiếng Việt không dấu, nhưng bạn hoàn toàn có thể sử dụng tiếng Việt có dấu thoải mái.

2.1 Các từ khóa

Từ khóa *keyword* là các từ dùng riêng của mỗi ngôn ngữ, được dùng cho những mục đích nhất định và *không thể định nghĩa lại* nhằm mục đích khác.

Python có rất ít từ khóa, tất cả chỉ gồm 33 từ khóa, các từ khóa này đều bằng tiếng Anh và viết dưới dạng chữ thường. Tùy vào phiên bản Python bạn sử dụng mà số lượng các từ khóa có thể khác nhau. Để lấy danh sách các từ khóa của phiên bản hiện tại đang dùng, bạn sử dụng thư viện `keyword` và lệnh `print(keyword.kwlist)`

```
>>> import keyword
>>> print(keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not',
'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

2.2 Tên và cách đặt tên

Một chương trình sử dụng rất nhiều **tên** hay còn gọi là **định danh *identification*** để làm tên chương trình, tên hàm, tên biến, tên hằng số… Tên trong Python có độ dài tuỳ ý. Chúng có thể gồm cả chữ cái cả in hoa và in thường, các chữ số và dấu gạch dưới, nhưng bắt buộc phải bắt đầu bằng một chữ cái hoặc dấu gạch dưới. Chúng ta nên đặt tên sao cho có ý nghĩa, và để dễ đọc ta thường ngăn cách giữa các từ bằng dấu gạch dưới, ví dụ `ho_ten, dia_chi`… Vì Python là ngôn ngữ tạo kiểu dữ liệu động, nên đối với các biến, bạn không cần khai báo nó trước khi sử dụng. Lưu ý rằng, như đã nói ở phần trước, Python có phân biệt chữ HOA và chữ thường, nên hai tên a và A là hoàn toàn khác nhau!

Một số quy ước bắt buộc khi đặt tên của Python

- Ký tự bắt đầu của tên phải là một dấu gạch dưới _ hoặc một chữ cái (có thể là chữ hoa hoặc chữ thường). Tiếp theo có thể là một hoặc nhiều ký tự, con số hoặc thậm chí bỏ trống (tức tên chỉ gồm một kí tự).

- Trong tên không được có dấu cách trắng hoặc các ký tự đặc biệt như: `@ , $ . % ^ ! = + - * / % & ~ ` \`
- Tên không được trùng với các *từ khóa* của Python.

Một số quy ước *không bắt buộc* khi đặt tên của Python

- Nên đặt tên có ý nghĩa sao cho dễ hiểu, dễ nhớ.
- Tên của `class` bắt đầu với một ký tự hoa, tiếp theo là chữ thường (có thể viết hoa đầu mỗi từ của tên, ví dụ `MyClass`).
- Tên của biến thường bắt đầu bởi chữ cái in thường.
- Tên bắt đầu với một ký tự gạch dưới duy nhất, được hiểu rằng đây là một định danh *private*.
- Tên bắt đầu với hai ký tự gạch dưới liên tiếp thì đây là một định danh có tính *private* mạnh.
- Nếu tên bắt đầu và kết thúc với hai dấu gạch dưới (chẳng hạn `__init__`), định danh là một cái tên đặc biệt của ngôn ngữ được định nghĩa; hoặc khi ta viết lại một hàm đã có sẵn của Python.

Bài tập

Bài 7. Trong các cách đặt tên như sau, tên nào hợp lệ, tên nào không hợp lệ.

2.3 Chú thích

Khi viết chương trình, đôi khi chúng ta cần ghi chú những điểm quan trọng, để người khác và chính chúng ta sau này, đọc lại sẽ hiểu được mã nguồn một cách tốt nhất. Lúc này chúng ta sẽ ghi chú trực tiếp vào mã nguồn, các ghi chú này gọi là **chú thích *comment***, trong chương trình mà không làm ảnh hưởng tới chương trình, chương trình dịch sẽ bỏ qua các chú thích này.

Một dòng chú thích trong Python bắt đầu với kí tự `#`. Nếu chú thích trên nhiều dòng thì đặt chúng vào trong cặp ngoặc nháy tam, thường sử dụng cặp ba nháy kép `"""`. Ví dụ:

```
# Đây là chú thích trên một dòng
"""
Đây là chú thích trên nhiều dòng
khác nhau
Blah Blah…
"""
```

2.4 Câu lệnh và khối lệnh

Trong một chương trình, ngoài các biến và kiểu dữ liệu của nó; chúng ta còn cần phải có các mô tả những hành động, công việc mà chương trình cần thực hiện. Như trong tiếng Việt, mỗi một câu phải có ý nghĩa nhất định, thì trong Python cũng thế, mỗi một câu lệnh phải để thực hiện một nhiệm vụ nào đó. Các câu lệnh có thể chỉ để thực hiện *một* hành động, nhiệm vụ nào đó, như gán giá trị vào một biến, gọi một hàm… Mỗi lệnh như vậy được gọi là một **câu lệnh** hoặc **lệnh đơn**. Ngoài các lệnh đơn này, Python còn có các câu lệnh ghép (lệnh hợp thành) mà ta sẽ gọi là khối lệnh, các câu lệnh lựa chọn, câu lệnh kiểm tra điều kiện, và các vòng lặp. Chúng được gọi chung là các câu lệnh có cấu trúc mà chúng ta sẽ tìm hiểu ở các chương sau.

Mỗi *câu lệnh* được viết trên một dòng và phải kết thúc bằng kí tự xuống dòng CR. Do đó, muốn kết thúc một *câu lệnh*, ta chỉ việc xuống một dòng mới. Ví dụ, trong hai *câu lệnh* sau, *câu lệnh*

thứ nhất sẽ in ra màn hình dòng chữ: `Toi la Phu Ong`, còn *câu lệnh* thứ hai sẽ in tiếp dòng chữ: `Ban ten la gi?`

```
print("Toi la Phu Ong")
print("Ban ten la gi?")
```

Tuy nhiên, đôi khi có những *câu lệnh* quá dài, để dễ nhìn ta có thể chủ động xuống dòng bằng cách gõ thêm kí tự \ (**backslash**) trước khi xuống dòng mới. Một dòng kết thúc bởi dấu \ thì không thể chứa các chú thích. Ví dụ sau đây, ta chủ động xuống dòng ở chỗ `+ 11`, thì trước khi xuống dòng, ta gõ thêm kí tự `\`, lúc này Python sẽ hiểu là chưa kết thúc một câu lệnh và sẽ chờ đến khi ta gõ `+ 13` và ấn Enter, mới thực hiện câu lệnh và trả về kết quả bằng 91. Mặc dù về mặt hình thức văn bản, ta thấy câu lệnh này được viết trên *hai* dòng, nhưng ta vẫn gọi đây là *một câu lệnh*, hoặc *một dòng lệnh* nếu không gây hiểu lầm.

```
>>> 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 +\
... 11 + 12 + 13
91
```

Vậy khi nào thì nên chủ động xuống dòng? Để cho mã nguồn của chúng ta được đẹp, mỗi dòng *không* nên dài quá 80 kí tự, nên đối với những câu lệnh quá dài ta có thể chủ động xuống dòng để cho mã nguồn dễ đọc hơn[3].
Tuy nhiên, có đôi khi xuống một dòng mới mà không cần kí tự `\`, đó là khi ta liệt kê các phần tử của một danh sách[4] *list*, một bộ *tuple*… Chẳng hạn, khi ta nhập các phần tử của list

```
l = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,
    ... 144, 233]
```

thì trước khi nhập phần tử `144`, ta ấn xuống dòng nhưng Python vẫn đủ thông minh để hiểu là chưa kết thúc một câu lệnh. Và, đối với các chú thích trên nhiều dòng, thì đương nhiên sẽ không cần gõ thêm kí tự \ trước khi xuống dòng mới.

Khi chúng ta cần thực hiện một nhóm các lệnh khác nhau, ví dụ khi giải phương trình bậc hai, thì trong trường hợp biệt số Δ dương, chúng ta phải thực hiện *các* thao tác in ra màn hình thông báo phương trình có hai nghiệm, rồi lần lượt in ra hai nghiệm đó. Khi này, ta cần đến **lệnh ghép** hay *lệnh hợp thành*. Câu lệnh ghép, được hợp thành bởi nhiều câu lệnh đơn nằm trên nhiều dòng khác nhau mà ta gọi là một **khối lệnh** thì Python sẽ sử dụng các khoảng cách trắng để phân biệt. Trong các ngôn ngữ khác, một *khối lệnh* thường được đánh dấu bằng cặp ký hiệu hoặc từ khóa. Ví dụ, trong C/C++, cặp ngoặc nhọn `{ }` được dùng để bao bọc một khối lệnh. Python, trái lại, có một cách rất đặc biệt để tạo khối lệnh, đó là thụt lề ***indent*** các câu lệnh trong khối vào sâu hơn (về bên phải) so với các câu lệnh của khối lệnh cha chứa nó.
Ví dụ, đoạn mã so sánh Δ với số `0`, nếu lớn hơn thì in ra màn hình dòng chữ `“Phương trình có hai nghiệm phân biệt”` và in ra hai nghiệm đó, trái lại thì in ra dòng chữ `“Phương trình không có hai nghiệm phân biệt”`.

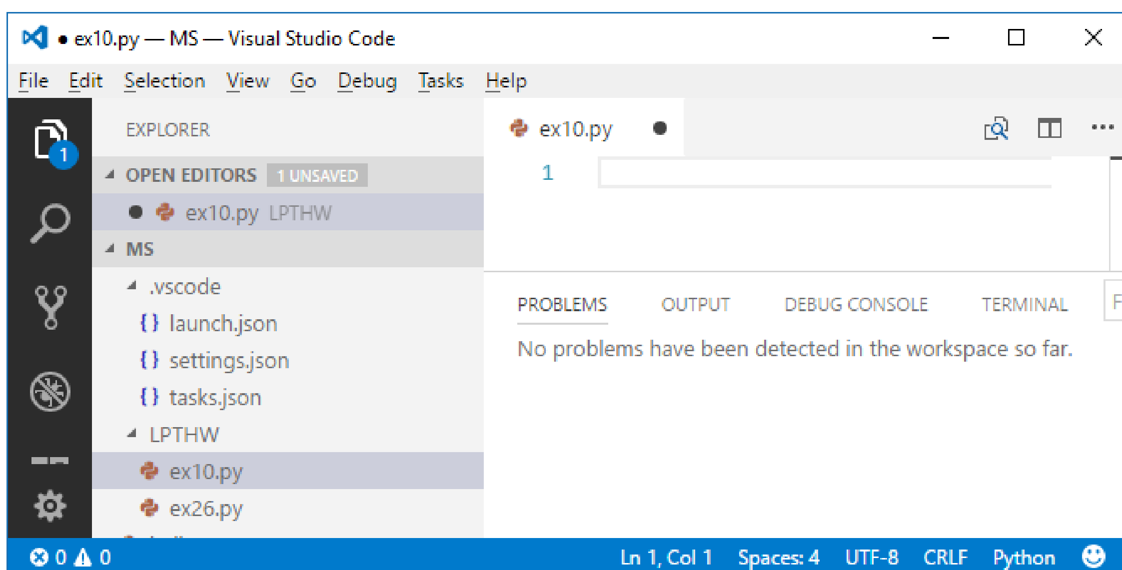
```
#không được dùng kí hiệu Delta nên ta đặt tên sao cho dễ hiểu
if delta > 0:
```

3 Chính vì lí do một dòng không nên quá dài, sẽ gây mỏi mắt, mà các tờ báo giấy mới in thành nhiều cột :)

4 Trong cuốn sách này, hầu hết các từ khóa, thuật ngữ của Python sẽ được giữ nguyên tiếng Anh mà không dịch sang tiếng Việt.

```
    # bắt đầu khối lệnh mới, thụt đầu dòng so với lệnh trước đó
    print("Phương trình có hai nghiệm phân biệt là:")
    print(x_1 = (-b + sqrt(delta)) / (2 * a))
    print(x_1 = (-b - sqrt(delta)) / (2 * a))
# kết thúc khối lệnh, trở về khối lệnh cha bằng cách không thụt đầu dòng
else:
    print("Phương trình không có hai nghiệm phân biệt")
```

Để làm điều này, ta có thể sử dụng dấu cách trắng (space) hoặc dấu nhảy Tab đều được, tuy nhiên từ phiên bản 3.x trở đi, Python chỉ cho phép sử dụng một trong hai loại, không được trộn lẫn (mix) cả hai trong cùng một tệp của chương trình. Thông thường, theo quy ước ngầm, người ta sử dụng 4 dấu cách trắng để thụt các câu lệnh trong khối.
Nếu bạn quen sử dụng phím Tab, có thể cài đặt để mỗi lần gõ phím Tab sẽ tương đương với 4 dấu cách trắng. Chẳng hạn, đối với **Visual Studio Code**, bạn bấm chuột vào vị trí Spaces ở trên thanh trạng thái (Status Bar) và chọn **Indent Using Spaces,** sau đó chọn số lượng dấu cách tương ứng với một lần bấm phím Tab.



Hình 7: Cửa sổ soạn thảo của Visual Studio Code

2.5 Một số quy tắc khi viết chương trình

Để chương trình dễ đọc, gỡ lỗi, vận hành và bảo trì chúng ta nên tuân thủ một số quy tắc sau:

- Nên viết chú thích rõ ràng và sử dụng chú thích thường xuyên, nếu cần thiết có thể ghi rõ ngày tháng sửa, lí do sửa… Đặc biệt, đối với mỗi hàm, mỗi mô-đun ta nên có mô tả chi tiết về nó như mục đích, cách sử dụng… Sau mỗi tên biến, nếu cần cũng nên giải thích biến này sử dụng để làm gì.

  Đối với những chỗ code chưa tối ưu, cần phải sửa chữa, có thể comment

```
# FIXME -- fix these code later
```

Đối với những chỗ cần thêm chức năng, nhiệm vụ… nên comment

```
# TODO -- in future you have to do this
```

- Nên dùng dấu cách thay cho dấu nhảy Tab, và nên dùng 4 dấu cách. Không được trộn lẫn (mix) giữa hai loại dấu này.
- Giữa các hàm cách nhau bởi một dòng trắng.
- Giữa các lớp cách nhau bởi hai dòng trắng.
- Tên biến nên bắt đầu bằng chữ in thường, tên lớp nên bắt đầu bằng chữ in hoa.

Chương 3 Các phép toán

Một biểu thức trong Python là một tập hợp các toán hạng và các toán tử được sắp xếp theo thứ tự nhất định. Toán hạng ở đây chính là các biến số, hằng số, các tên hàm hoặc các con số cụ thể. Còn toán tử chính là các phép toán, gồm có phép gán giá trị, các phép toán số học, phép toán logic, phép so sánh hoặc các phép toán trên các kiểu dữ liệu đặc biệt. Ví dụ trong biểu thức sau

```
2 + 3 * sin(pi)
```

thì `2, 3, sin(pi)` là các toán hạng, còn `+, *` là các toán tử.

Trong phần này, chúng ta chưa tìm hiểu sâu về các kiểu dữ liệu cơ bản của Python. Do đó, nếu có nói đến kiểu số thì bạn hiểu đó là các con số thông thường như số thực[5], số nguyên,… còn nếu nói đến dữ liệu kiểu xâu *string* thì bạn hiểu đó chính là các chữ cái, các từ, các câu… được đặt trong cặp ngoặc nháy đơn, nháy kép.

3.1 Phép gán

Khái niệm biến số, hằng số được hiểu giống như trong toán học, chúng ta sẽ không đi sâu vào các khái niệm này. Mỗi một biến, hằng đều mang một giá trị nào đó. Để **gán giá trị** `value` cho biến số hoặc hằng số có tên `name` ta sử dụng cú pháp

```
name = value
```

Ví dụ sau đây gán giá trị bằng `3` vào biến có tên là `a` và giá trị `3.14` vào biến có tên là `pi`. Sau đó thực hiện phép cộng số học `a + pi`. Ví dụ tiếp theo gán giá trị kiểu string `"Phu"` vào biến có tên là `ho`, giá trị `"Ong"` vào biến có tên là `ten` và thực hiện phép cộng hai xâu (*string*) `ho + ten`

```
>>> a = 3
>>> pi = 3.14
>>> a + pi
6.140000000000000
>>> ho = "Phu"
>>> ten = "Ong"
>>> ho + ten
'PhuOng'
```

Chú ý cần phân biệt *phép gán* `=` với phép *so sánh bằng nhau, so sánh đồng nhất* `==`.

Về biến số, ta có các khái niệm biến toàn cục và biến địa phương. Phần này sẽ thảo luận sau khi tìm hiểu về hàm và lớp.

Chúng ta có thể gán đồng thời nhiều giá trị cho nhiều biến cùng một lúc, ta gọi là **phép gán đa biến** *(multiple assignment)*, sử dụng cú pháp sau

```
name1, name2 = value1, value2
```

Chẳng hạn, hãy xem ví dụ sau

5 Lưu ý rằng Python sử dụng dấu chấm để ngăn cách phần nguyên và phần thập phân, chẳng hạn số π có giá trị gần đúng là π = 3.1415926

```
>>> a, b = 10, 5
>>> ten, tuoi, sdt = 'Phu Ong', 31, '0123456789'
```

Chính nhờ ưu điểm này, mà để hoán đổi giá trị của hai biến, ta có thể chỉ sử dụng một câu lệnh và cũng không cần phải sử dụng thêm các biến trung gian như các ngôn ngữ khác, chỉ đơn giản như sau

```
>>> a, b = b, a
```

Bài tập

Bài 8. Hãy tự đặt 10 tên biến khác nhau và sử dụng trình thông dịch Python để kiểm tra xem tên đó có hợp lệ không.

Gợi ý. Sử dụng phép gán để kiểm tra tên có hợp lệ không, nếu gán thành công là hợp lệ, ngược lại là không hợp lệ.

3.2 Phép toán số học

Python cung cấp các phép toán số học sau:

- **Phép cộng +**
  Tùy vào các toán hạng là kiểu số hay xâu mà phép cộng có tác dụng như phép cộng trong toán học, hoặc sẽ có tác dụng nối hai **xâu** *string* lại với nhau (**phép cộng xâu**). Ví dụ:

```
>>> 1 + 2
3
>>> "Phu" + "Ong"
'PhuOng'
```

- **Phép trừ -**
  Phép trừ có tác dụng như phép trừ trong toán học, áp dụng cho kiểu số.

```
>>> 3 - 6
-3
```

Nếu áp dụng phép trừ cho các toán hạng kiểu tập hợp thì ta có phép lấy hiệu hai tập hợp.

- **Phép nhân ***
  Nếu cả hai toán hạng là kiểu số thì phép nhân có tác dụng như phép nhân trong toán học. Còn nếu một trong hai toán hạng là **string**, toán hạng còn lại là một số nguyên dương **k** thì phép nhân sẽ có tác dụng lặp lại **xâu** *string* đó thêm **k** lần.

```
>>> 2*3
6
>>> "Phu Ong"*3
'Phu OngPhu Ong Phu Ong'
>>> 3*"Phu Ong"
'Phu OngPhu Ong Phu Ong'
```

- **Phép chia /**
  Phép chia áp dụng cho kiểu số. Nếu các toán hạng là số thực hoặc số nguyên thì kết quả trả về luôn là số thực. Lưu ý, trong **Python 2** thì phép chia này có tác dụng như phép chia lấy phần nguyên. Khi đó, muốn kết quả là một số thực bạn phải ép kiểu một trong hai toán hạng sang kiểu số thực *float*, chẳng hạn thay vì viết `1/2` bạn phải viết `1/2.` hoặc `1./2`

```
>>> 1/2
0.5
>>> 1/7
0.14285714285714285
```

- **Phép chia lấy phần nguyên //**

  Phép chia `//` này tương ứng với phép toán `div` trong toán học và các ngôn ngữ khác. Khi chia số nguyên `a` cho số nguyên `b` ta được thương là `q` và số dư là `r`.

```
a = b*q + r
```

Kết quả của phép chia này chính là thương `q` trong định nghĩa trên.

```
>>> 12//4
3
>>> 13//4
3
>>> 15//4
3
```

- **Phép chia lấy phần dư %**

  Khi chia số nguyên `a` cho số nguyên b ta được thương là `q` và số dư là `r`. Thì kết quả của phép chia lấy dư `%` chính là số `r` trong biểu thức đã nêu ở phần trước.

```
>>> 12 % 4
0
>>> 13 % 4
1
>>> 15 % 4
3
```

- **Phép lũy thừa****

  Đây chính là phép tính lũy thừa trong toán học, `a**b` có trả về kết quả là a^b.

```
>>> 2**10
1024
```

Chú ý rằng, `a^b` trong Python không phải là phép tính lũy thừa hoặc mũ, mà là phép toán dịch chuyển bit, xem phần sau để rõ hơn.

Bài tập

Bài 9. Sử dụng trình thông dịch Python và các phép toán số học để kiểm tra xem năm 2017 có là năm nhuận hay không.

Bài 10.

3.3 Phép toán thao tác bit

Thông thường, trong cuộc sống chúng ta sử dụng hệ đếm thập phân, cơ số 10 gồm các 10 chữ số từ 0 đến 9. Nhưng để biểu diễn thông tinh trong máy tính, chúng ta sử dụng các bóng bán dẫn, chỉ có hai trạng thái bật hoặc tắt, tương ứng với hai giá trị 1 và 0 của hệ đếm cơ số 2, tức hệ nhị phân. "Các phép toán trên thao tác bit (bitwise operation) được thực hiện trên một hoặc nhiều chuỗi bit hoặc số nhị phân tại cấp độ của từng bit riêng biệt. Các phép toán này được thực hiện nhanh, ưu tiên, được hỗ trợ trực tiếp bởi vi xử lý, và được dùng để điều khiển các giá trị dùng cho so sánh và tính toán. Đối với các loại vi xử lý rẻ tiền, thường thì các phép toán trên thao tác bit nhanh hơn phép chia đáng kể, đôi khi nhanh hơn phép nhân, và đôi khi nhanh hơn phép cộng đáng kể. Trong khi các vi xử lý hiện đại thường thực hiện phép nhân và phép cộng nhanh tương đương các phép toán trên thao tác bit nhờ vào cấu trúc đường ống lệnh của chúng dài hơn và cũng nhờ vào các lựa chọn trong thiết kế cấu trúc, các phép toán trên thao tác bit thường sử dụng ít năng lượng hơn vì sử dụng ít tài nguyên hơn."[6] Tuy nhiên, trong Python chúng ta ít khi sử dụng các phép toán thao tác trên bit.

Chẳng hạn xét `a = 60` và `b = 13`, chúng ta biểu diễn chúng trong hệ nhị phân sẽ được

```
>>> a = 60
>>> b= 13
>>> bin(a)
'0b111100'
>>> bin(b)
'0b1101'
```

Chúng ta có các phép toán chuyển đổi bit như sau:

- **Phép nghịch đảo ~** có tác dụng đổi các bit từ 0 sang 1 và ngược lại.

```
>>> a = 1988
>>> ~a
-1989
>>> bin(a)
'0b11111000100'
>>> bin(~a)
'-0b11111000101'
```

- Phép
- **Phép dịch chuyển bit.** Xét dãy bit 00110 khi dịch sang trái sẽ được 01100, còn dịch sang phải sẽ được 00011. Phép dịch bit sang trái còn mang ý nghĩa là nhân một số cho 2, còn dịch sang phải mang ý nghĩa chia một số cho 2.
-

6 Phần này lấy từ https://vi.wikipedia.org/wiki/Ph%C3%A9p_to%C3%A1n_thao_t%C3%A1c_bit

3.4 Phép toán logic

Phép toán logic làm việc trên các toán tử logic, tức là các biến, biểu thức chỉ có hai khả năng đúng True hoặc sai False. Xin xem kiểu dữ liệu logic bool để hiểu rõ. Python cung cấp các phép toán logic sau and, or, not.

Phép toán and sẽ trả về kết quả True nếu cả hai toán hạng đều là True. Bảng giá trị chân lý của phép toán and như sau:

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

Phép toán or sẽ trả về kết quả True nếu cả hai toán hạng đều là True. Bảng giá trị chân lý của phép toán or như sau:

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Phép toán not được sử dụng để đảo ngược giá trị chân lý của một biểu thức. Bảng giả trị chân lý của phép toán not như sau:

A	not A
True	False
False	True

Sau đây là một vài ví dụ minh họa, tất cả đều chạy trực tiếp trong trình thông dịch của Python.

```
>>> True and False
False
>>> True or False
True
>>> not False
True
```

3.5 Phép toán so sánh

Kết quả trả về của một phép toán so sánh là True (đúng) hoặc False (sai), hai giá trị này thuộc kiểu dữ liệu bool. Python cung cấp các phép toán so sánh sau đây:

- **Đồng nhất (giống nhau, bằng nhau) ==**

Cần chú ý rằng phép so sánh đồng nhất khác với phép toán is

- **Lớn hơn >**
- **Lớn hơn hoặc bằng >=**
- **Nhỏ hơn <**
- **Nhỏ hơn hoặc bằng <=**
- **Khác nhau !=**

3.6 Thứ tự ưu tiên các phép toán

Khi

Thứ tự ưu tiên các phép toán, lần lượt từ ưu tiên cao xuống thấp, trong Python như sau:

```
+  -  ~
**
*  /  %
+  -
>>  <<
&
^
|
<  <= == >=  >  !=  <>  is
not
and
or
```

Trong bảng trên, nằm cùng hàng với nhau có mức độ ưu tiên bằng nhau, khi đó chúng sẽ được thực hiện lần lượt từ trái qua phải.

Chương 4 Các thủ tục vào ra dữ liệu cơ bản

4.1 Hàm print()

Để in một xâu *string* ra màn hình ta dùng hàm print(). Ví dụ

```
>>> print('Toi la Phu Ong')
Toi la Phu Ong
```

Hàm print() có các tham số sau

```
print(value, sep, end, file, flush)
```

Trong đó, value là giá trị sẽ được in ra màn hình, giá trị này phải là một xâu kí tự, hoặc chỉ được là các giá trị thuộc cùng một kiểu dữ liệu, nếu có nhiều giá trị thì cách nhau bởi dấu phẩy; sep là kí tự ngăn cách giữa các giá trị, end là kí tự khi kết thúc câu lệnh print().

Để in nhiều xâu cùng lúc ta có thể sử dụng các toán tử trên xâu. Ở đây xin giới thiệu qua một số cách, chi tiết xin xem chương dữ liệu kiểu xâu.

Chẳng hạn ta muốn in ra màn hình nội dung của xâu chứa trong biến temp và câu dẫn

```
>>> temp = 'Phu Ong'
>>> print("Tên tôi là", temp)
Tên tôi là Phu Ong
```

Hoặc có thể sử dụng phép nối xâu để in

```
>>> temp = 'Phu Ong'
>>> print("Tên tôi là" + temp)
Tên tôi làPhu Ong
```

Tuy nhiên, nếu sử dụng phép nối xâu, ta thấy kết quả thu được sẽ không có dấu cách giữa các đối số của hàm print như cách trước nữa.

```
>>> print("Ten toi la %s va toi nang %d kg!" % ('Phu Ong', 51))
```

Bản chất của câu lệnh trên là ta đã sử dụng các phép toán định dạng xâu, có thể viết như sau cũng thu được cùng một kết quả.

```
>>> temp = "Ten toi la %s va toi nang %d kg!" % ('Phu Ong', 51)
>>> print(temp)
```

```
>>> print("Total score for %s is %s " % (name, score))
```

Hoặc sử dụng kiểu mới để định dạng xâu:

```
print("Ten toi la {} va toi nang {} kg".format('Phu Ong', 51))
```

Or pass the values as parameters and print will do it:

```
print("Total score for", name, "is", score)
```

If you don't want spaces to be inserted automatically by print, change the sep parameter:

```
print("Total score for ", name, " is ", score, sep='')
```

4.2 Hàm `input()`

Để nhập dữ liệu vào chương trình ta dùng hàm `input()`. Kiểu dữ liệu mặc định chương trình nhận vào sẽ là kiểu xâu.

```
>>> x = input('Ban ten gi? ')
Ban ten gi? Phuong
>>> print(x)
Phuong
```

Để nhập vào một số nguyên hoặc số thực, hoặc một kiểu có cấu trúc phức tạp hơn, thì ta làm thế nào? Chúng ta phải sử dụng các hàm chuyển đổi kiểu của Python. Chẳng hạn để nhập vào một số nguyên, ta dùng hàm `int()` để chuyển sang kiểu số nguyên, dùng hàm `float()` để chuyển sang kiểu số thực. Ví dụ

```
>>> a=input('Xin moi nhap mot so: ')
Xin moi nhap mot so: 13
>>> a*2
'1313'
>>> a = int(input('Xin moi nhap mot so: '))
Xin moi nhap mot so: 13
>>> a*2
26
```

hoặc nhập vào một số thực

```
>>> a = float(input('Xin moi nhap mot so: '))
Xin moi nhap mot so: 1.3
>>> a*2
2.6
```

Đối với các kiểu dữ liệu phức tạp hơn, ta phải sử dụng thêm các hàm để xử lý xâu kí tự nhập vào.

Để nhập một vào nhiều giá trị cùng một lần, các giá trị cách nhau bởi dấu phẩy hoặc một kí tự bất kì, ta dùng vẫn hàm `input()` nhưng phải sử dụng thêm phương thức `split()` của kiểu xâu. Ví dụ, nhập vào hai số nguyên `a, b` cách nhau bởi dấu phẩy.

```
>>> a, b = input("Xin moi nhap vao hai so:").split(',')
```

hoặc cách nhau bởi dấu cách trắng

```
>>> a, b = input("Xin moi nhap vao hai so:").split(' ')
```

dĩ nhiên, sau đó ta muốn sử dụng `a, b` như là các số nguyên thì phải chuyển đổi từ kiểu xâu này sang kiểu số nguyên, vì mặc định nhập vào luôn là kiểu xâu.

Để nhập vào một danh sách *list*, ta nhập vào một xâu, sau đó chuyển xâu đó sang danh sách bằng cách tách rời các phần tử. Ví dụ, người dùng nhập vào một dãy các số nguyên, cách nhau bởi dấu cách trắng, và chúng ta phải chuyển thành một *list*.

```
>>> a = [int(x) for x in input().split()]
3 4 5
>>> a
[3, 4, 5]
```

Nếu cách nhau bởi dấu phẩy

```
>>> a = [int(x) for x in input().split(',')]
3,4,5
>>> a
[3, 4, 5]
```

Ở cách này, chúng ta phải sử dụng thêm vòng lặp for, bạn có thể xem chi tiết ở chương sau. Hoặc có thể sử dụng hàm `map()` để ánh xạ mỗi giá trị với một phần tử của danh sách.

```
>>> s = input()
1 2 3 4 5
>>> numbers = list(map(int, s.split()))
>>> numbers
[1, 2, 3, 4, 5]
```

Bài tập

Bài 11. Viết chương trình nhập vào một số nguyên n và in ta màn hình giá trị bình phương của số đó.

Bài 12. Viết chương trình nhập vào bán kính r của đường tròn, là một số thực, và in ra diện tích của hình tròn đó.

Chương 5 Câu lệnh điều khiển

5.1 Câu lệnh điều kiện *if*

Nếu `<điều kiện>` đúng thì thực hiện `<khối lệnh>`

```
if <điều kiện> :
    <khối lệnh>
```

Nếu `<điều kiện>` đúng thì thực hiện `<khối lệnh 1>`, nếu sai thì thực hiện `<khối lệnh 2>`

```
if <điều kiện> :
    <khối lệnh 1>
else:
    <khối lệnh 2>
```

Nếu `<điều kiện 1>` đúng thì thực hiện `<khối lệnh 1>`, nếu sai thì kiểm tra tiếp `<điều kiện 2>` và cứ như vậy...

```
if <điều kiện 1> :
    <khối lệnh 1>
elif <điều kiện 2>:
    <khối lệnh 2>
    ...
else:
    <khối lệnh n>
```

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

Bài tập

Bài 13. Viết chương trình cho người dùng nhập vào một số nguyên n. Nếu n lẻ thì in ra màn hình dòng chữ `“Số lẻ.”`. Nếu số n chẵn và lớn hơn hoặc bằng 100 thì in ra `“Số chẵn và lớn hơn hoặc bằng 100.”`, ngược lại thì in ra `“Số chẵn và bé hơn 100.”`

Bài 14. Biện luận phương trình bậc nhất `a*x + b = 0.`

Bài 15. Biện luận phương trình bậc hai `a*x**2 + b*x + c = 0.`

Bài 16. Người dùng nhập vào một năm là một số nguyên dương bất kì. Cho biết năm đó có là năm nhuận hay không?

Bài 17. Viết chương trình biện luận phương trình bậc nhất `a*x + b = 0.`

Lời giải.

5.2 Câu lệnh vòng lặp *for*

Vòng lặp **`for`** có cấu trúc:

```
for <tên_biến> in <phạm_vi>:
    <khối lệnh>
```

Trong đó <phạm_vi> có thể là một tập hợp *set*, một danh sách *list*, một xâu *string* hoặc một dãy số nguyên. Ví dụ sau sẽ in ra các phần tử của list hoa_qua

```
hoa_qua = ['chuoi', 'tao', 'xoai', 'cam', 'le']
for qua in hoa_qua:
    print(qua)
```

Trường hợp <phạm_vi> là một xâu kí tự:

```
for letter in 'Python':
    print(letter)
```

Đối với dãy các số nguyên, ta có hàm `range()` để tạo dãy. Ví dụ sau sẽ in ra các số nguyên chẵn bé hơn `100`.

```
for x in range(100):
    if x % 2 == 0:
        print(x)
```

Trong ví dụ trên, số `0` cũng được in ra, lí do là hàm `range(100)` sẽ trả về kết quả là tất cả các số nguyên bắt đầu từ `0` và bé hơn `100`, tức là dãy số `0, 1, 2, 3,…, 99.`

Hàm `range()` có các cách sử dụng sau, `range(n)` trả về các số nguyên từ `0, 1, 2,…` cho đến `n-1`. Nếu ta thêm tham số, `range(a,b)` thì trả về các số nguyên từ a đến `b-1.` Còn `range(a,b,i)` trả về các số nguyên cách nhau `i` đơn vị, bắt đầu từ a cho đến không quá `b-1.`

Để thoát khỏi (kết thúc) *vòng* lặp `for`, ta dùng câu lệnh `break`, để thoát khỏi *lần* lặp hiện tại và bắt đầu lần lặp kế tiếp ta dùng câu lệnh `continue`.

```
for i in range(10):
    if i == 3:
        continue
    if i == 8:
        break
    print(i)

print(i)
```

Kết quả thu được như sau:

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

Ta thấy, khi biến `i` có giá trị bằng 3 thì chương trình sẽ nhảy bỏ qua vòng lặp hiện tại – tức là không in ra giá trị 3 này – mà nhảy qua vòng lặp tiếp theo luôn. Còn khi `i=8` thì chương trình sẽ thoát hoàn toàn khỏi vòng lặp. Sau vòng lặp, ta thấy biến `i` dừng ở giá trị 8.

Bài tập

Bài 18. Viết chương trình kiểm tra tính nguyên tố của một số n do người dùng nhập vào.

Bài 19. Viết chương trình nhập vào một số nguyên và in kết quả ra màn hình dưới dạng số đảo ngược (về thứ tự) của số nguyên vừa nhập đó.

Bài 20. Viết chương trình nhập vào một câu tiếng Anh, đếm số *từ* và ký tự trong câu đó, và in kết quả ra màn hình.
Gợi ý, các từ cách nhau bởi dấu cách trắng.

Bài 21. Vẽ hình chữ nhật đặc có chiều dài `n` và chiều rộng `m`. Ví dụ với `m = 4, n = 5`

```
* * * * *
* * * * *
* * * * *
* * * * *
```

Bài 22. Vẽ hình chữ nhật rỗng có chiều dài `n` và chiều rộng `m`. Ví dụ: `m = 4, n = 5`

```
* * * * *
*       *
*       *
* * * * *
```

Bài 23. Viết chương trình in ra bảng cửu chương dùng cho học sinh cấp 1 như sau:

```
2x2=4  3x2=6  4x2=8  5x2=10 6x2=12 7x2=14 8x2=16 9x2=18
2x3=6  3x3=9  4x3=12 5x3=15 6x3=18 7x3=21 8x3=24 9x3=27
2x4=8  3x4=12 4x4=16 5x4=20 6x4=24 7x4=28 8x4=32 9x4=36
2x5=10 3x5=15 4x5=20 5x5=25 6x5=30 7x5=35 8x5=40 9x5=45
2x6=12 3x6=18 4x6=24 5x6=30 6x6=36 7x6=42 8x6=48 9x6=54
2x7=14 3x7=21 4x7=28 5x7=35 6x7=42 7x7=49 8x7=56 9x7=63
2x8=16 3x8=24 4x8=32 5x8=40 6x8=48 7x8=56 8x8=64 9x8=72
2x9=18 3x9=27 4x9=36 5x9=45 6x9=54 7x9=63 8x9=72 9x9=81
```

Bài 24. Viết chương trình cộng hai số tự nhiên, mỗi số có 100 chữ số.

Bài 25. Viết chương trình cộng hai số tự nhiên `a` và `b` có độ dài tùy ý, chẳng hạn

```
a = "12345678902142141525434325432632632" và
b = "7839496235703058070385781578932759830785642".
```

Lời giải tham khảo cho một số bài tập

Bài

```python
for i in range(2,10):
    for j in range(2,10):
```

```
            if i*j < 10:
                print(str(j)+'x'+str(i)+'='+str(i*j), end = ' ')
            else:
                print(str(j)+'x'+str(i)+'='+str(i*j), end = ' ')
        print()
```

Bài

```
a = "1234567890214214152543432543263263 2"
b = "78394962357030580703857815789327598307856 42"
n = max(len(a), len(b))

def fill_zero(a, n):
    return '0'*(n-len(a)) + a

a = fill_zero(a, n)
b = fill_zero(b, n)
kq = []
so_nho = 0

for i in range(n-1, -1, -1):
    temp = int(a[i]) + int(b[i]) + so_nho
    kq.append(temp % 10)
    so_nho = temp // 10

                      kq = "".join(str(e) for e in kq)
                               kq = kq[::-1]
                                 print(kq)
```

5.3 Câu lệnh vòng lặp *while*

Vòng lặp **while** có cấu trúc:

```
while <điều kiện> :
    <khối lệnh>
```

Trong đó, `<điều kiện>` là các biểu thức logic hoặc các biểu thức trả về kết quả là kiểu *bool*. Để thoát khỏi vòng lặp `while` ta cũng sử dụng lệnh `break`.

Chú ý rằng trong Python không có câu lệnh **repeat... until...**, mà thay vào đó, ta có thể sử dụng vòng lặp `while` với câu lệnh `break` để thoát khỏi vòng lặp, cú pháp như sau.

```
while True:
    <khối lệnh>
    if <điều kiện>:
        break
```

Với cách làm này thì `<khối lệnh>` sẽ luôn luôn được thực hiện, cho đến khi nó gặp `<điều kiện>` thì mới kết thúc vòng lặp. Chẳng hạn, chương trình sau sẽ in ra tất cả các số nguyên chẵn nhỏ hơn `1000` và chia hết cho `3`.

```
i = 0
while True:
    if i % 3 == 0:
        print(i)
    i += 2
    if i == 1000:
        break
```

Tuy nhiên, cách làm trên chỉ để minh hoạ cho cách viết theo phong cách của câu lệnh `repeat until` nên *khá* phức tạp, chúng ta có thể làm ngắn gọn hơn như sau:

```
i = 0
while i < 1000:
    if i % 3 == 0:
        print(i)
    i += 2
```

Bài tập

Bài 26. Chia dãy số nguyên không âm $a_1, a_2, ..., a_n$, với $n > 1$ cho trước thành hai đoạn có tổng các phần tử trong mỗi đoạn bằng nhau.

Bài 27. In ra tất cả các xâu nhị phân[7] có độ dài n, ví dụ với $n = 4$ thì kết quả in ra màn hình là: `0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111`.

Lời giải tham khảo cho một số bài tập

Bài

```
n = 8
s = list(0 for x in range(n+1))
while True:
    i = n
    print("".join(str(x) for x in s), end = " ")
    while ((i >= 0) and (s[i] == 1)):
        i = i - 1
    s[i] = 1
    for j in range(i+1,n+1):
        s[j] = 0
    if i < 0:
        break
```

7 Xâu nhị phân độ dài n là xâu có độ dài bằng n và chỉ được tạo nên từ hai kí tự 0 và 1, ví dụ 10010110 là xâu nhị phân có độ dài bằng 8.

Chương 6 Các kiểu dữ liệu

Trong Python có các kiểu dữ liệu cơ bản, được xây dựng sẵn *built-in* sau:

- Kiểu số như nguyên, số thực, số phức, phân số;
- Kiểu logic bool;
- Kiểu xâu string;
- Kiểu None;
- Kiểu danh sách list, kiểu bộ tuple, kiểu từ điển dict; kiểu tập hợp.

Để kiểm tra kiểu của một dữ liệu, ta dùng hàm `type()`

```
>>> type(1)
<class 'int'>
>>> type("Phu Ong")
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type([1, 2, 3])
<class 'list'>
>>> type({1, 2, 3})
<class 'set'>
>>> type((1, 2, 3))
<class 'tuple'>
```

6.1 Kiểu logic *bool*

Khi thực hiện các phép toán so sánh hoặc các phép toán logic, kết quả trả về chỉ có thể đúng hoặc sai. Khi đó, ta sử dụng kiểu *bool* để biểu diễn. Để biểu diễn giá trị đúng, ta dùng từ khóa `True`, để biểu diễn giá trị sai ta dùng từ khóa `False`. Chúng được gọi là các giá trị chân lý trong Python.

`True` và `False` cũng giống như các số nguyên `0` và `1`. Tuy nhiên, nếu ta ép kiểu từ số nguyên hoặc các kiểu dữ liệu khác sang kiểu bool thì mọi giá trị khác `0` hoặc khác rỗng đều tương ứng với `True`. Hãy xem ví dụ sau.

```
>>> True == 1
True
>>> False == 0
True
>>> True + True
2
>>> True == 2
False
>>> bool(1)
True
>>> bool(10)
True
```

```
>>> bool("")
False
>>> bool("a")
True
```

6.2 Kiểu số

Kiểu số trong Python bao gồm số nguyên `int`; số thực `float`; các số cơ số 2, cơ số 6, cơ số 8; số phức `complex`, phân số.

6.3 Kiểu xâu *string*

Kiểu xâu *string,* đôi khi còn được dịch thành kiểu chuỗi, là một kiểu dữ liệu mà chúng ta thường gặp nhất, dùng để biểu thị các kí tự, các từ, các câu... Để tạo một string, chúng ta có thể dùng một dấu nháy đơn, một dấu nháy kép hoặc ba dấu nháy kép. Khi dùng ba dấu nháy kép, chúng ta có thể viết string trên nhiều dòng mà không cần dùng ký tự thoát `\`. Lưu ý rằng, hai dấu nháy dùng để mở và đóng này phải cùng loại, cùng là nháy đơn, cùng là nháy kép chẳng hạn, không được trộn lẫn.

```
>>> a = 'Xin chao'
>>> a
'Xin chao'
>>> b = "Toi la Phu Ong"
>>> b
'Toi la Phu Ong'
>>> c = """Day la string
... tren
... nhieu dong"""
>>> c
'Day la string\ntren\nnhieu dong'
```

Để sử dụng dấu nháy trong một string, bạn có thể bao nó bằng một cặp dấu nháy khác loại. Chẳng hạn muốn sử dụng dấu nháy kép, thì bạn đặt cả string đó vào trong cặp nháy đơn và ngược lại.

```
>>> "I'm Phuong."
"I'm Phuong."
```

Một cách khác là sử dụng kí tự điều khiển \, chẳng hạn

```
>>> 'I\'m Phuong'
"I'm Phuong"
>>> '\'\"'
'\'"
```

Các phương thức của kiểu string

Đối với kiểu *string*, có một số phương thức – hàm hay sử dụng sau đây.

Hàm `<string>.title()` dùng để chuyển các kí tự đầu của mỗi từ trong `<string>` thành kiểu chữ hoa.

```
>>> s = "i'm phu ong"
>>> s.title()
"I'm Phu Ong"
```

Hàm `<string>.upper()` dùng để chuyển tất cả các kí tự trong `<string>` thành kiểu chữ hoa.

```
>>> s = "i'm phu ong"
>>> s.upper()
"I'M PHU ONG"
```

Hàm `<string>.lower()` dùng để chuyển tất cả các kí tự trong `<string>` thành kiểu chữ thường.

```
>>> s = "I'm Phu Ong"
>>> s.title()
"i'm phu ong"
```

Hàm `<string>.swapkey()` dùng để hoán đổi kí tự hoa thành thường và thường thành hoa củả tất cả các kí tự trong `<string>`.

```
>>> s = "I'm Phu Ong"
>>> s.title()
"i'M pHU oNG"
```

Hàm `<string>.isalnum()` dùng để kiểm tra xem `<string>` có chỉ gồm các kí tự thuộc bảng chữ cái `a-Z` hoặc chữ số `0-9` hay không, tức là có chứa các kí tự đặc biệt như `!@#$%…` hay không.

```
>>> s = "qwerty123"
>>> s.isalnum()
True
>>> s = "qwerty$%"
>>> s.isalnum()
False
```

Chú ý rằng, kể cả xâu của chúng ta có chứa kí tự cách trắng thì kết quả trả về cũng là `False`.

Tương tự, hàm `<string>.isdigit()` dùng để kiểm tra xem `<string>` có phải chỉ chứa các kí tự số `0-9` hay không.

```
>>> s="123"
>>> s.isdigit()
True
>>> s="123.45"
>>> s.isdigit()
False
```

Hàm `<string>.isalpha()` dùng để kiểm tra xem `<string>` có phải chỉ chứa các kí tự thuộc bảng chữ cái `a-Z` hay không, ở ví dụ sau nếu xâu có chứa dấu cách thì kết quả trả về là `False`.

```
>>> s="Alphabet"
>>> s.isalpha()
True
>>> s="Alphabet ABC"
>>> s.isalpha()
False
```

Hàm `<string>.islower()/<string>.isupper()` dùng để kiểm tra xem `<string>` có phải chỉ chứa các kí tự chữ thường/chữ hoa hay không. Tương tự, chúng ta cũng có hàm `<string>.istitle()` để kiểm tra xem các từ trong `<string>` có được viết hoa chữ cái đầu mỗi *từ* hay không.

```
>>> s = 'Toi La Phu Ong'
>>> s.isupper()
False
>>> s.islower()
False
>>> s.istitle()
True
```

Hàm `len(<string>)` trả về độ dài (số lượng kí tự) của `<string>`.

Lưu ý rằng, chỉ số của các kí tự trong một xâu được đánh số từ 0. Chẳng hạn

```
>>> s = "Phuong"
>>> len(s)
6
>>> s[0]
'P'
>>> s[1]
'h'
```

Như ta thấy, xâu `"Phuong"` gồm có 6 kí tự, và kí tự đầu tiên là `'P'` ở vị trí `s[0]` chứ không phải là `s[1]`. Để lấy ra một xâu con của xâu `<string>`, chúng ta sẽ chỉ ra chính xác vị trí bắt đầu và kết thúc của xâu con, sử dụng cú pháp

```
<string>[i:j]
```

trong đó, `i` là vị trí bắt đầu còn `j` là vị trí kết thúc.

```
s = '1234567abcde'
>>> s[1:6]
'23456'
```

Nếu muốn lấy từ vị trí đầu tiên, ta có thể lược bỏ tham số `i`, còn muốn lấy đến vị trí cuối cùng của xâu ta có thể lược bỏ tham số `j`.

```
>>> s = '1234567abcde'
>>> s[:6]
'123456'
>>> s[6:]
```

```
'7abcde'
>>> s[:]
'1234567abcde'
```

Đấy là trong trường hợp chúng ta lấy theo chiều tăng của chỉ số, còn muốn lấy theo chiều giảm của chỉ số hoặc muốn lấy các kí tự ở những vị trí cách quãng nhau, ta dùng cú pháp

```
<string>[i:j:k]
```

trong đó, `i` là vị trí bắt đầu, `j` là vị trí kết thúc còn k là bước nhảy.

```
>>> s = '1234567abcde'
>>> s[1:5:2]
'24'
>>> s[6:0:-1]
'765432'
```

Do đó, muốn đảo ngược một xâu, ta chỉ cần đơn giản là dùng cách lấy toàn bộ xâu đó với bước nhảy `-1`.

```
>>> s = '1234567abcde'
>>> s[::-1]
'edcba7654321'
```

Cần lưu ý rằng, kiểu xâu *string* là kiểu dữ liệu không thay đổi được *immutable*, tức là bạn không thể thay đổi được xâu đã tạo ra như một số ngôn ngữ khác, Pascal chẳng hạn, trừ khi bạn gán giá trị mới cho biến đã tạo đó. Để dễ hiểu, ta xét ví dụ xâu `s = "Phu ong"`, chúng ta không thể thay đổi xâu s thành `"phu ong"` bằng phép gán `s[0] = 'p'` được. Lúc này, muốn thay đổi bắt buộc ta phải *gán* đè nội dung mới cho biến s, và đương nhiên một biến s mới sẽ được tạo ra tại một địa chỉ bộ nhớ mới. Để hiểu rõ hơn, xin mời xem ví dụ sau đây.

```
>>> s = 'Phu Ong'
>>> id(s)
2281456950328
>>> s[0]
'P'
>>> s[0] = 'p'
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s = 'phu ong'
>>> id(s)
2281456951056
>>> s += ' is my name!'
>>> s
'phu ong is my name!'
>>> id(s)
2281456947112
```

Như ta thấy, đầu tiên khi khởi tạo biến `s = 'Phu Ong'` thì máy tính sẽ tham chiếu định danh s đến vùng ô nhớ có địa chỉ `2281456950328`, lưu ý rằng ở máy tính của bạn địa chỉ này có thể

khác. Và chúng ta không thể sửa đổi nội dung xâu trong biến s này, chẳng hạn ta thử sửa chữ P thành p bằng phép gán `s[0] = 'p'`, máy báo lỗi ngay. Sau đó, ta bắt buộc phải thực hiện phép gán mới, nếu muốn biến s tham chiếu đến nội dung khác, đương nhiên sẽ được lưu ở vùng bộ nhớ có địa chỉ mới, lần này là 2281456951056. Thậm chí, khi ta thực hiện các phép toán trên xâu s, tổng quát là các hàm, các phương thức, mà làm cho dữ liệu của biến s thay đổi, thì thực ra là máy đã tham chiếu tên s đến một vùng ô nhớ mới. Điều này khác hoàn toàn với kiểu dữ liệu *mutable,* kiểu danh sách chẳng hạn. Hãy xem ví dụ sau, để tìm hiểu kĩ hơn về kiểu danh sách xin mời xem phần sau.

```
>>> L = [1, 3, 8]
>>> id(L)
2281456980424
>>> L.append(2)
>>> L
[1, 3, 8, 2]
>>> id(L)
2281456980424
```

Như ta đã thấy, sau khi thực hiện phương thức thêm vào *list* L phần tử 2 thì địa chỉ ô nhớ của biến L vẫn như cũ, không hề thay đổi.

Trong Python có hai kiểu dữ liệu là không thay đổi được *immutable* và thay đổi được *mutable*. Kiểu dữ liệu *immutable* gồm có kiểu số nguyên *int*, kiểu số thực *float* và *decimal*, kiểu số phức *complex*, kiểu logic *bool*, kiểu xâu *string*, kiểu bộ *tuple*, kiểu khoảng *range*, kiểu tập hợp đóng băng *frozenset*, kiểu số nguyên *bytes*. Còn kiểu *mutable* gồm có danh sách *list*, từ điển *dict*, tập hợp *set*, mảng số nguyên *bytearray*, và các lớp do người dùng định nghĩa (trừ trường hợp chỉ rõ lớp đó là *immutable*). Chúng ta sẽ lần lượt xét tính thay đổi được hay không của từng kiểu dữ liệu khi học đến chúng.

Ưu điểm lớn nhất của kiểu *immutable* là máy tính sẽ sử dụng bộ nhớ đúng để biểu diễn dữ liệu của biến đó, như vậy sẽ tiết kiệm bộ nhớ hơn là việc phải dành sẵn một dung lượng bộ nhớ lớn hơn dữ liệu hiện tại biến đó đang tham chiếu đến. Chẳng hạn, khi tạo ra một list...

Phương thức `<string>.split("<kí tự>")` dùng để tách `<string>` thành các xâu con, mà mỗi xâu con này phân cách nhau bởi kí tự `<kí tự>`. Kết quả trả về được đặt trong một danh sách *list*, bạn có thể xem phần kiểu dữ liệu danh sách ở phần sau để hiểu rõ hơn về danh sách. Nếu phương thức `split` này không có tham số truyền vào, thì mặc định, Python sẽ sử dụng dấu cách trắng để làm kí tự tách.

```
>>> s = "We all love Python"
>>> s.split(" ")
['We', 'all', 'love', 'Python']
>>> s.split()
['We', 'all', 'love', 'Python']
```

Ở ví dụ trên, chúng ta tách xâu thành các xâu con, mà mỗi xâu con được cách nhau bởi dấu cách trắng. Còn ví dụ sau chúng ta quy định kí tự để tách là dấu chấm.

```
>>> pi = "3.1415926"
>>> pi.split('.')
['3', '1415926']
```

Đôi khi, chúng ta cần tách từng kí tự của một xâu, khi đó ta phải sử dụng phương pháp chuyển đổi một xâu sang danh sách, sử dụng hàm `list()`. Chẳng hạn

```
>>> s = "Phuong"
>>> list(s)
['P', 'h', 'u', 'o', 'n', 'g']
```

Vấn đề này xin tìm hiểu thêm ở phần kiểu dữ liệu danh sách *list*.

Ngược lại của tách, chúng ta có thể nối nhiều xâu thành một xâu bằng cách dùng phương thức `join()`, sử dụng cú pháp sau

```
<kí tự nối>.join(<danh_sách>)
```

Phương thức này sẽ nối các phần tử của `<danh_sách>` lại với nhau thành một xâu mới, giữa các phần tử của `<danh_sách>` ở trong xâu mới sẽ được cách nhau bởi `<kí tự nối>`.

```
>>> "-".join(["Toi","la","Phu","Ong"])
'Toi-la-Phu-Ong'
```

Phương thức `<string>.strip()` dùng để loại bỏ các khoảng trắng (dấu cách, dấu Tab, kí tự xuống dòng) ở đầu và cuối xâu. Tương tự, phương thức `lstrip()` sẽ loại bỏ các khoảng trắng ở bên trái của xâu, tức ở đầu xâu, còn `rstrip()` sẽ loại bỏ ở cuối xâu.

```
>>> s = ' Toi la Phu Ong \n'
>>> s.strip()
'Toi la Phu Ong'
>>> s.lstrip()
'Toi la Phu Ong \n'
>>> s.rstrip()
' Toi la Phu Ong'
```

Như ta thấy, kết quả trả về không ảnh hưởng gì tới biến `s` cả, vì biến `s` có kiểu xâu là kiểu dữ liệu immutable, nên các hàm, phương thức không thể làm thay đổi được dữ liệu lưu trong biến `s`.

Hàm `<string>.find(<string con>)` dùng để tìm kiếm ký tự hoặc xâu `<string con>` trong `<string>`. Kết quả trả về là vị trí đầu tiên xuất hiện `<string con>` trong `<string>`, nếu không tìm thấy thì kết quả là `-1`.

```
>>> s = 'Toi la Phu Ong'
>>> s.find('Phu')
7
>>> s.find('PhuOng')
-1
```

Trong các phương thức tìm kiếm, còn có hai phương thức là `<string>.startswith(<string con>)` và `<string>.endswith(<string con>)` dùng để kiểm tra xem xâu `<string>` có bắt đầu hoặc kết thúc bởi `<string con>` hay không, đương nhiên kết quả trả về là kiểu `bool`.

```
>>> s = 'Toi la Phu Ong'
>>> s.startswith('Toi')
True
>>> s.endswith('g')
True
```

6.4 Kiểu danh sách *list*

Danh sách *list*, hiểu đơn là một tập hợp các đối tượng có chung một tính chất nào đó. Ví dụ danh sách các món ăn trên bàn tiệc, danh sách các bài hát của một album, danh sách học sinh trong một lớp học… Ta thấy ngay, mỗi danh sách gồm có các phần tử khác nhau, để liệt kê chúng, chẳng hạn liệt kê các thực phẩm cần mua cho bữa liên hoan, bạn sẽ sử dụng dấu phẩy để ngăn cách các phần tử này. Python cũng vậy, tuy nhiên, các danh sách các phần tử cần được đặt trong cặp ngoặc vuông [].

Và, cũng giống như kiểu xâu *string*, các chỉ số *index* của một danh sách *list* được đánh số bắt đầu từ `0`.

Kiểu danh sách có thể hiểu gần giống như kiểu mảng *array* của các ngôn ngữ lập trình khác.

Ví dụ sau tạo một danh sách `mon_an` gồm có các món canh cua, cà pháo, cá kho, thịt rán.

```
>>> mon_an = ['canh cua', 'ca phao', 'ca kho', 'thit ran']
```

Không như kiểu xâu *string*, là những đối tượng *immutable*, thì kiểu danh sách là đối tượng *mutable*, tức ta có thể thay đổi các phần tử của một danh sách. Chẳng hạn, thay cà pháo bằng dưa muối, ta sẽ gán giá trị mới cho phần tử `mon_an[1]`.

```
>>> mon_an[1] = 'dua muoi'
>>> mon_an
['canh cua', 'dua muoi', 'ca kho', 'thit ran']
```

Cũng giống như kiểu xâu string, kiểu danh sách list cũng có các phép toán

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

```
['spam', 'eggs', 'bacon', 4]
>>> 3 * a[:3] + ['Boo!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boo!']
```

Dùng MAP, FILTER và REDUCE để xử lý danh sách / list trong Python

1. Giới thiệu

- List là một trong những kiểu dữ liệu được sử dụng rất nhiều trong python.
- Các thao tác thường được thực hiện trên list: xử lý từng phần tử trong list, lọc lấy một số phần tử thỏa điều kiện, tính toán dựa trên tất cả các phần tử của list(vd tính tổng) và trả về kết quả.
- Để đơn giản việc xử lý List, Python hỗ trợ một số hàm có sẵn để thực hiện các tác vụ trên gồm `map()`, `filter()`, `reduce()`

2. Map

`map(func, seq)` `map` sẽ áp dụng hàm func cho mỗi phần tử của seq và trả về list kết quả.

Ví dụ: Tính bình phương các số có trong list a. sử dụng map():

```
my_list = [1, 2, 3, 4, 5]

def binh_phuong(number):
    return number * number

print map(binh_phuong, my_list)
# [1,2,9,16,25]
```

- Trong ví dụ trên, map sẽ tự động áp dụng hàm binh_phuong với mỗi phần tử trong danh sách my_list
- Hàm truyền vào hàm `map` nhận vào một tham số cùng kiểu với phần tử của list
- Có thể sử dụng lamda thay thế cho hàm. Ví dụ trên có thể được viết lại: `print map(lambda x: x*x, my_list)`

ách thông thường:

```
my_list = [1,2,3,4,5]

result = list()
for number in my_list:
    result.append( number*number)

print result # [1,2,9,16,25]
```

3. Filter

- filter(func, list)

- Hàm filter sẽ gọi hàm func với tham số lần lượt là từng phần tử của list và trả về danh sách các phần tử mà func trả về **True**
- func chỉ có thể trả về **True** hoặc **False**

Ví dụ: lọc ra các số chẵn từ danh sách a. Sử dụng filter:

```
my_list = [1, 2, 3, 4, 5]

def so_chan(number):
    if number % 2 == 0:
          return True
    else:
          return False

print filter(so_chan, my_list) # [2,4]
```

Sử dụng lambda

```
print filter(lambda x: x%2 ==0, my_list)# [2,4]
```

b. Không dùng filter

```
my_list = [1, 2, 3, 4, 5]
ket_qua = list()
for number in my_list:
    if number % 2 == 0:
          ket_qua.append(number)
print ket_qua
```

4. Reduce

`reduce(func, seq)` reduce sẽ tính toán với các phần tử của danh sách và trả về kết quả. `func` là một hàm nhận vào 2 tham số có dạng`func(arg1, arg2)` trong đó `arg1` là kết quả tính toán với các phần tử trước, `arg2` là giá trị của phần tử của danh sách đang được tính toán.

Ví dụ: tính tổng bình phương của các phần tử trong mảng a. Su dung reduce

```
data = [1,2,3,4]
def tinh_tong(tong, so):
    return tong + so*so

#su dung ham
print reduce(tinh_tong, data) #30

#su dung lambda
print reduce( (lambda tong, so: tong + so*so), data) #30
```

b. Khong su dung reduce

```
data = [1,2,3,4]
tong = 0
for so in data:
    tong += so*so

print tong #30
```

5. Kết luận

- *Trong bài viết chỉ đưa ra những ví dụ đơn giản nên có thể các bạn chưa thấy được sự tiện dụng của `map, filter, reduce`.*
- *Tuy nhiên khi phải làm việc với list nhiều các bạn sẽ thấy nó rất là tiện đặc biệt là khi sử dụng kèm lambda hoặc tái sử dụng các hàm với map, filter và reduce*

Danh sách của danh sách

Các phần tử của một *list* lại có thể là một *list*, chẳng hạn

```
>>> A = [[1,2,3], [4,5,6], [7,8,9,10,11]]
>>> A[1]
[4, 5, 6]
>>> A[1][2]
6
>>> A[2][4]
11
>>> A[1][5]
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

List A gồm có ba phần tử, mỗi phần tử lại là một list, `[1,2,3], [4,5,6], [7,8,9,10,11]`. Khi đó, để truy cập đến phần tử `j` của list `A[i]` nằm trong *list* `A` ta dùng cú pháp `A[i][j]`. Kiểu dữ liệu này tương ứng với kiểu mảng hai chiều *array*, hoặc là ma trận, trong các ngôn ngữ khác. Tuy nhiên, nếu một ma trận thì, nếu xét riêng từng dòng, bắt buộc các dòng phải có số cột như nhau. Nhưng ở đây thì khác, nếu ta viết lại `list A` ở trên, mỗi phần tử viết trên một dòng thì ta sẽ có dạng gần giống ma trận, do số cột của từng dòng là khác nhau

```
[1,2,3]
[4,5,6]
[7,8,9,10,11]
```

Rõ ràng, kiểu danh sách của danh sách này linh hoạt hơn – nhưng cũng đồng nghĩa với khó kiểm soát hơn – kiểu mảng hai chiều trong các ngôn ngữ khác.

Bài tập

Bài 28. Cho một *list* các số nguyên. Viết chương trình tìm phần tử lớn nhất của *list* đó.

Bài 29. Cho một *list* các số nguyên. Viết chương trình in ra tất cả các phần tử lớn là số chẵn và lớn hơn 10 của *list* đó.

Bài 30. Viết chương trình tính số trung bình các giá trị của một *list* số nguyên.

Bài 31. Tạo một *list* gồm *n* số nguyên không âm ngẫu nhiên.

6.5 Kiểu từ điển *dict*

6.6 Kiểu bộ *tuple*

6.7 Kiểu tập hợp *set*

Trong toán học, ***tập hợp*** là khái niệm không thể định nghĩa. Ta hiểu tập hợp như sự tụ tập các đối tượng nào đó, mà ta gọi là các phần tử. Dữ liệu **kiểu tập hợp** (**set, kiểu set**) cũng giống như vậy, là một *tập* các phần tử trong đó mỗi phần tử chỉ xuất hiện duy nhất một lần. Hiểu đơn giản như kể tên các học sinh trong một lớp, mỗi học sinh đương nhiên chỉ được kể một lần, và ta không cần theo thứ tự nào cả; lúc này, ta cần đến kiểu tập hợp.

Để tạo một *set* ta có cú pháp khá giống với *dictionary*, với các phần tử trong *cặp ngoặc nhọn* cách nhau bởi dấu phẩy. Tuy nhiên, mỗi phần tử của set là một giá trị (thay cho một cặp giá trị như với dictionary), ví dụ:

```
set = {1, 7, 2, 6, 4, 5}
```

Kiểu **set** có các đặc điểm sau:

- **Set** được biểu diễn trong ngoặc nhọn `{}` với các phần tử phân cách bởi dấu phẩy.
- Mỗi phần tử chỉ xuất hiện *một lần* trong *set* cho dù bạn có nhập nó nhiều lần.
- Kiểu set cũng có các phép lấy hợp, lấy giao như trong Toán học.
- Set là dữ liệu thay đổi được *mutable*, tuy nhiên mỗi phần tử trong set thì lại không sửa được *immutable*. Tuy nhiên Python cũng có *immutable set* gọi là **frozenset**.
- Để tạo một *set* **s** rỗng, bạn cần dùng hàm **`set()`** không có thông số. Vì sử dụng `s={}` sẽ tạo một dictionary rỗng.

Các phép toán trên kiểu tập hợp, cho `S, S1, S2` là các tập hợp, `x` là một phần tử:

- `x in S`
  Phương thức kiểm tra phần tử `x` có nằm trong tập `S` hay không. Trả về kết quả `True` nếu tập `S` chứa phần tử `x`, `False` nếu `S` không chứa `x`.

```
>>> S = {1, 3, 2, 8, 10}
>>> 1 in S
True
>>> 4 in S
False
```

- `len(S)`
  Phương thức lấy số lượng phần tử của tập `S`, trả về kết quả là một số nguyên không âm.

```
>>> S = {1, 3, 2, 8, 10}
>>> len(S)
5
```

```
>>> A = set() #tập A là tập rỗng
>>> len(A)
0
```

- `S.add(x), S.remove(x), S.discard(x), S.pop(), S.clear()`
  Phương thức `S.add(x)` sẽ thêm phần tử `x` vào tập `S`.
  Phương thức `S.remove(x), S.discard(x)` sẽ xóa phần tử `x` khỏi tập `S`.

  Phương thức `S.pop()` lấy ra phần tử đứng đầu tập hợp đồng thời xóa phần tử đó khỏi tập `S`.

  Phương thức `S.clear()` xóa tất cả các phần tử của tập `S`.

```
>>> S = {1, 2, 3, 4, 5, 6}
>>> S.add(100) #Thêm phần tử 100 vào tập S
>>> S
{1, 2, 3, 4, 5, 6, 100}
>>> S.remove(1) #Xóa phần tử 1 khỏi tập S
>>> S
{2, 3, 4, 5, 6, 100}
>>> S.pop() #Lấy ra phần tử đầu tiên của tập hợp
2
>>> S
{3, 4, 5, 6, 100}
>>> S.discard(6)
>>> S
{3, 4, 5, 100}
>>> S.clear() #Xóa hết mọi phần tử của S, lúc này S trở thành rỗng
>>> S
set()
```

Chú ý rằng hai thủ tục `remove(x)` và `discard(x)` cùng xóa đi phần tử `x`, nhưng sự khác nhau ở đây là nếu tập `S` không tồn tại phần tử `x` thì phương thức:

- ✔ `discard(x)` sẽ trả về kết quả `None`.
- ✔ `remove(x)` sẽ trả về lỗi `KeyError`.

Hãy xem ví dụ sau để hiểu rõ hơn.

```
>>> S = {'a', 'b', 'c', 'd'}
>>> S.discard('e') #Không hiển thị gì, dùng print để xem kết quả
>>> print(S.discard('e'))
None
>>> S.remove('e')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'e'
```

- `S1 - S2, S1.difference(S2)`

Phép lấy hiệu hai tập hợp `S1` và `S2`, trả về một tập mới chứa các phần tử thuộc tập `S1` và không thuộc tập `S2`. Một phép toán tương tự là `S1.symmetric_difference(S2)` hoặc `S1^S2` có tác dụng như `S2 - S1`.

```
>>> S1 = {1, 2, 3, 4, 5, 6}
>>> S2 = {0, 2, 4, 6, 8, 10}
>>> S1 - S2
{1, 3, 5} #Hiệu của tập S1 và S2
```

- `S1|S2, S1.union(S2)`
  Phép lấy hợp hai tập hợp `S1` và `S2`, trả về một tập mới chứa tất cả các phần tử thuộc tập `S1` và tất cả các phần tử thuộc tập `S2`. Đương nhiên, nếu phần tử nào đó thuộc cả hai tập hợp thì chỉ được kể một lần.

```
>>> S1 = {1, 2, 3, 4, 5, 6}
>>> S2 = {0, 2, 4, 6, 8, 10}
>>> S1|S2
{0, 1, 2, 3, 4, 5, 6, 8, 10} #Hợp của S1 và S2
```

- `S1 & S2, S1.intersection(S2)`
  Phép lấy giao hai tập hợp `S1` và `S2`, trả về một tập mới chứa tất cả các phần tử thuộc đồng thời cả hai tập `S1` và `S2`. Tức là các phần tử chung của hai tập hợp đó.

```
>>> S1 = {1, 2, 3, 4, 5, 6}
>>> S2 = {0, 2, 4, 6, 8, 10}
>>> S1&S2
{2, 4, 6} #Giao của S1 và S2 gồm ba phần tử 2, 4, 6
```

- `S1 <= S2, S1.issubset(S2)`
  Phép kiểm tra `S1` có là tập con của `S2` hay không, trả về `True` nếu đúng và nếu `False` sai. Nhắc lại rằng, tập `S1` là tập con của `S2` nếu mọi phần tử của `S1` đều thuộc tập `S2`.

```
>>> S1 = {1, 2, 3, 4, 5, 6, 7}
>>> S2 = {1, 2, 5}
>>> S3 = {1, 2, 5, 9}
>>> S4 = set()
>>> S1.issubset(S1) #Mọi tập đều là tập con của chính bản thân nó
True
>>> S2.issubset(S1)
True
>>> S3.issubset(S1)
False
>>> S4.issubset(S1) #Tập rỗng là tập con của mọi tập hợp
True
```

- `S1 >= S2, S1.issupperset(S2)`
  Phép kiểm tra `S1` có là tập cha của (tức là có chứa) `S2` hay không, trả về `True` nếu đúng và nếu `False` sai. Nhắc lại rằng, tập `S1` là tập cha của `S2` nếu mọi phần tử của `S2` đều thuộc tập `S1`.

```
>>> S1 = {1, 2, 3, 4, 5, 6, 7}
>>> S2 = {1, 2, 5}
>>> S3 = {1, 2, 5, 9}
>>> S4 = set()
>>> S1.issupperset(S1) #Mọi tập đều là tập cha của chính bản thân nó
```

```
True
>>> S1.issupperset(S2)
True
>>> S1.issupperset(S3)
False
>>> S1.issupperset(S4)
True
```

- `S1 < S2, S1 > S2`
  Phép kiểm tra `S1` có là tập con thực sự, tập cha thực sự của `S2` hay không. Nhắc lại rằng, tập `S1` là tập con thực sự của `S2` nếu `S1` là tập con của `S1` nhưng `S1` và `S2` không trùng nhau.

Bài tập

Bài 32. Cô giáo *Thảo* thống kê chiều cao của các học sinh trong lớp 12A được kết quả như sau:
161 182 161 154 176 170 167 171 170 174 150 142 148 165 170 178 156 145 149 163 162 159 165 165 170 180 155 159 155 153 152 162 180 168 169 168 167 170
1. Hỏi lớp có bao nhiêu học sinh?
2. Tính chiều cao trung bình của các học sinh trong lớp.
3. Liệt kê các chiều cao khác nhau của học sinh trong lớp. Tính giá trị trung bình của chúng.

Bài 33. Liệt kê tập hợp S gồm các kí tự của câu sau, kể cả dấu cách trắng,
tôi là phú ông.
Tập hợp S có bao nhiêu phần tử. Thêm vào tập S các phần tử là các kí tự xuất hiện trong câu sau
tôi rất nghèo tiền bạc, chỉ giàu thời gian thôi.

Bài 34. Cho tập hợp A gồm có các phần tử sau, mỗi phần tử cách nhau bởi dấu cách trắng,
1 a 5 7 f 0 89 g i 11 88 3 p
Liệt kê các tập con có hai phần tử của tập hợp A. Có tất cả bao nhiêu tập con như vậy.

Bài 35. Một lớp có 40 học sinh trong đó có 10 bạn học tiếng Pháp, 14 bạn học tiếng Anh, 6 bạn học cả hai thứ tiếng đó. Hỏi có bao nhiêu học sinh không học tiếng Pháp mà cũng không học tiếng Anh.

Bài 36. Trong một lớp học mỗi học sinh đều chơi thể bóng đá hoặc bóng chuyền. Có 25 học sinh chơi bóng đá, 27 học sinh chơi bóng chuyền và 18 học sinh chơi cả hai. Hỏi lớp đó có bao nhiêu học sinh?

6.8 Kiểu *None*

Để biểu diễn *không có gì cả*, ta dùng kiểu `None`. `None` là một từ khóa đặc biệt trong Python dùng để biểu diễn một giá trị, đối tượng *rỗng*, không có gì cả. Chú ý rằng None khác với `False`, `0` hoặc một *list*, *dictionary*, *string* rỗng. Ví dụ:

```
>>> None == 0
False
>>> None == []
False
>>> None == False
False
>>> x = None
```

```
>>> y = None
>>> x == y
True
```

Một hàm số không có câu lệnh `return` thì mặc định sẽ trả về một đối tượng `None`.

```
def a_void_function():
    a = 1
    b = 2
    c = a + b
x = a_void_function()
print(x)
```

Kết quả trả về là `None`, vì hàm này không có câu lệnh `return`. Tuy nhiên, một hàm số có thể có câu lệnh `return` nhưng vẫn trả về kết quả `None`.

```
def even_function(a):
    if (a % 2) == 0:
        return True
x = even_function(3)
print(x)
```

Hàm trên chỉ trả về kết quả `True` nếu tham số nhận được là số chẵn, nên nếu là số lẻ thì kết quả trả về là `None`.

Chương 7 Hàm

7.1 Khái niệm hàm

Trong lập trình, để thực hiện *một* công việc nào đó, và công việc này lặp đi lặp lại, chúng ta sử dụng *hàm* sẽ tiết kiệm được công sức. Chính bạn đã sử dụng một số hàm đã được xây dựng sẵn trong Python, ví dụ như hàm `print(), type()`... Chẳng hạn, trong chương trình chúng ta có một nhiệm vụ làgiải phương trình bậc hai $ax^2+bx+c=0$ năm lần, thì mỗi lần người dùng sẽ nhập vào các hệ số `a, b, c` từ bàn phím, chương trình sẽ giải phương trình với các hệ số đó và trả lại kết quả là vô nghiệm, có nghiệm kép hay hai nghiệm phân biệt và in các nghiệm đó. Nếu không sử dụng *hàm* ta phải đánh máy đoạn chương trình sau 5 lần!

```
from math import sqrt

a = int(input("a = "))
b = int(input("b = "))
c = int(input("c = "))
d = b**2 – 4*a*c
if d < 0:
    print("Phương trình vô nghiệm.")
elif d == 0:
    print("Phương trình có nghiệm kép x = ", -b /(2*a))
else:
    print("Phương trình có hai nghiệm ", (-b + sqrt(d))/(2*a), " và ",\
    (-b - sqrt(d))/(2*a))
```

Khi đó, chương trình trông sẽ dài dòng phức tạp, hơn nữa, mỗi lần giải một phương trình mới ta lại phải gõ lại đoạn mã này, tính *tái sử dụng* không[8] có. Lúc này, chúng ta cần sử dụng đến *hàm.*

Bài tập

Bài 37. Giải thích sự giống và khác nhau của *hàm* trong Python và trong toán học.

7.2 Khai báo hàm

Để khai báo một hàm chúng ta sử dụng từ khóa `def` với cú pháp:

```
    def name([arg,... arg=value,... *arg, **arg]):
        """Mô tả về hàm nếu (nên) có"""
        <khối lệnh>
        return [giá trị trả về]
```

Trong đó `name` là tên của hàm, được đặt tên theo đúng quy tắc đặt tên và theo sau phải là cặp ngoặc đơn `()`. Bên trong cặp ngoặc đơn, ta có `arg` là các tham số *nếu có*, `value` là giá trị mặc định của tham số – nếu người dùng không cung cấp một giá trị cho tham số khi gọi hàm thì giá trị mặc định này sẽ được sử dụng, `*arg, **arg` là các tham số kiểu *tuple*, tức một bộ các tham số. Một hàm có thể có tham số hoặc không. Cần lưu ý rằng, nếu hàm có tham số nhận các giá trị mặc định, thì các tham số này phải được khai báo sau các tham số không nhận giá trị mặc định. Chẳng hạn, khai báo hàm bậc hai `y=a*x**2+b*x+c` thì thường giá trị mặc định là `a=1`,

8 Bạn chỉ có thể tái sử dụng bằng cách Copy – Paste mà thôi :)

nhưng ta không thể khai báo tham số a lên trước, mà bắt buộc phải khai báo tham số a cuối cùng, như sau chẳng hạn

```
def ham_bac_hai(b, c, a = 1):
    pass
```

Tiếp theo là dấu hai chấm : và xuống dòng để bắt đầu vào phần thân của hàm. Phần này có thể gồm chú thích mô tả về hàm để giúp cho các lập trình viên khác, và chính bản thân chúng ta sau này, hiểu rõ hơn về hàm và khối lệnh chính của hàm để thực hiện nhiệm vụ đề ra.
Cuối cùng là từ khóa `return` để thoát khỏi hàm và trở về chương trình chính, hoặc trả về giá trị của hàm đối với hàm có giá trị trả về; phần này không bắt buộc phải có.

Một *hàm* nếu được khai báo trong một lớp thì còn được gọi là một **phương thức** *method* của lớp.

7.3 Lời gọi đến hàm

Để gọi một hàm có tên là `name` đã được định nghĩa, ta sử dụng câu lệnh `name()`. Hãy quay trở lại ví dụ giải phương trình bậc hai để hiểu rõ hơn.

```
from math import sqrt
def giai_pt():
    a = int(input("a = "))
    b = int(input("b = "))
    c = int(input("c = "))
    d = b**2 - 4*a*c
    if d < 0:
        print("Phương trình vô nghiệm.")
    elif d == 0:
        print("Phương trình có nghiệm kép x = ", -b /(2*a))
    else:
        print("Phương trình có hai nghiệm ", (-b + sqrt(d))/(2*a),
        " và ", (-b - sqrt(d))/(2*a))
```

Như vậy, hàm `giai_pt` ở trên không có tham số truyền vào và cũng không có kết quả trả về, bởi bản thân các lệnh trong hàm đã in ra nghiệm của phương trình rồi. Để gọi hàm này ta chỉ đơn giản là dùng câu lệnh `giai_pt()`.

Ở cách làm này, trong bản thân hàm đã có các câu lệnh để người dùng nhập vào giá trị của các hệ số `a, b, c` của phương trình. Nếu muốn truyền các hệ số khi gọi hàm `giai_pt()` ta có thể viết lại như sau:

```
from math import sqrt
def giai_pt(a, b, c):
    d = b**2 - 4*a*c
    if d < 0:
        print("Phương trình vô nghiệm.")
    elif d == 0:
        print("Phương trình có nghiệm kép x = ", -b /(2*a))
    else:
```

```
        print("Phương trình có hai nghiệm ", (-b + sqrt(d))/(2*a),\
        " và  ", (-b - sqrt(d))/(2*a))
```

Lúc này, để giải phương trình bậc hai với các hệ số `a = 1, b = -3, c = 5` ta chỉ việc gọi hàm bằng cách sử dụng câu lệnh `giai_pt(1, -3, 5)`.

7.4 Giá trị trả về của hàm

Hàm có thể có hoặc không có giá trị trả về. Trong trường hợp hàm có kết quả trả về, tức là có từ khóa `return`, thì hàm cũng có thể được sử dụng như một *biến* để tính toán. Hãy xét một hàm đơn giản, tên là `cong` để cộng hai số nguyên *a* và *b*, định nghĩa như sau:

```
def cong(a, b):
    return a + b
```

Chúng ta sẽ thử gọi hàm này với các tham số truyền vào `a = 3, b =7.`

```
>>> cong(3, 7)
10
>>> cong(3, 7) + 5
15
```

Nếu ta thực hiện chương trình này trong trình thông dịch Python, thì ở lần gọi hàm thứ nhất kết quả trả về là `10`, ở lần gọi thứ hai, bản thân hàm được sử dụng như một biến số để thực hiện tiếp phép cộng, và kết quả trả về là `10`. Tuy nhiên, nếu các bạn soạn một *script* có nội dung như sau

```
def cong(a, b):
    return a + b

cong(3, 7)
cong(4, 7) + 5
```

và chạy script này thì kết quả là không xuất hiện gì cả trên màn hình! Lí do là chương trình vẫn tính `a + b`, và vẫn trả về kết quả này và được sử dụng để tính toán tiếp nhưng không được in ra màn hình. Lúc này, muốn in kết quả ra chúng ta phải sử dụng đến hàm `print` của Python.

```
def cong(a, b):
    return a + b

print(cong(3, 7))
print(cong(4, 7) + 5)
```

Theo sau từ khóa `return` là một giá trị, một biểu thức, một câu lệnh, thậm chí lại là một *hàm*[9] hoặc *không có gì cả*. Nếu không có gì, thì chương trình chỉ đơn giản là thoát khỏi hàm mà thôi và trả về giá trị `None`. Mỗi khi gặp từ khóa `return` thì hàm sẽ kết thúc, chương trình sẽ thoát khỏi hàm và thực hiện câu lệnh tiếp theo. Hãy xem ví dụ sau để hiểu rõ hơn:

9 Xem phần đệ quy hàm để rõ hơn.

```
def test():
    print(1)
    return
    print(2)
```

Chúng ta sẽ chạy thử.

```
>>> test()
1
>>> print(test())
1
None
```

Ta thấy ngay, câu lệnh `print(2)` không được thực thi vì trước đó chương trình đã thực hiện lệnh `return` rồi. Hơn nữa, lệnh `return` này không trả về gì cả, nên nếu ta dùng lệnh `print(test())` thì chương trình sẽ trả về kết quả `None`. Còn số 1 kia được in ra là do bản thân trong hàm có câu lệnh `print(1)` mà thôi.

Bài tập

Bài 38. Viết hàm giải hệ phương trình bậc nhất hai ẩn, sử dụng định thức cấp hai.

Bài 39. Viết chương trình kiểm tra xem số tự nhiên n có phải là số nguyên tố không.

Bài 40. Viết chương trình tìm ước chung lớn nhất của hai số tự nhiên.

7.5 Tham số của hàm

Như đã nói, một hàm có thể có tham số hoặc không.....

Phần này lấy từ blog https://icewolf-blog.herokuapp.com/post/8

Cách truyền tham số dạng *args và **kwargs trong python (abittrary arguments)

Kiểu truyền tham số này thường được sử dụng khi định nghĩa những hàm không biết trước số lượng tham số truyền vào. Thực sự thì không nhất thiết phải là *args và **kwargs. điều quan trọng là tham số có 1 dấu sao * hay là 2 dấu sao **. Đặt tên tham số là *var hay **vars hay bất cứ thứ gì bạn muốn. Nhưng để dễ hiểu thì nên dùng tên chuẩn là *args và **kwargs

1. *args và **kwargs dùng để làm gì?

- Khi khai báo 1 hàm, sử dụng *args và **kwargs cho phép bạn truyền vào nhiều mà không cần biết trước số lượng.
- **Ví dụ**

```
# với giả sử các tham số truyền vào đều là số
```

```
def sum(*args):
    total = 0
    for number in args:
      total += number
  return total

# gọi hàm
sum(1, 2, 3,19)
sum( 1, 100)
```

2. *args và **kwargs khác gì nhau?

- Cho những bạn chưa biêt: Khi gọi hàm trong Python, có 2 kiểu truyền tham số:
    - Truyền tham số theo tên.
    - Truyền tham số bình thường theo thứ tự khai báo đối số. **Ví dụ**

```
def register(name, password):
  ....

#Truyền tham số theo kiểu thông thường, phải theo đúng thứ tự
register( 'Coulson', 'hail_Hydra')

#Truyền tham số theo tên, Không cần phải theo thứ tự khai báo
thao số
register( password='cookHim', name='Skye')
```

- ***args** nhận các tham số truyền bình thường. Sử dụng **args** như một list.
- ****kwargs** nhận tham số truyền theo tên. Sử dụng **kwargs** như một. dictionary**Ví dụ**

```
def test_args(*args):
  for item in args:
     print item

>>>test_args('Hello', 'world!')
Hello
world!

def test_kwargs(*kwargs):
  for key, value in kwargs.iteritems():
     print '{0} = {1}'.format(key, value)

>>test_kwargs(name='Dzung', age=10)
age = 10
name = Dzung
```

3. Thứ tự sử dụng và truyền tham số *args, **kwargs và tham số bình thường

Khi sử dụng phải khai báo đối số theo thứ tự:

đối số xác đinh –> *args –> **kwargs

Đây là thứ tự bắt buộc. Và khi truyền tham số bạn cũng phải truyền theo đúng thứ tự này. Không thể truyền lẫn lộn giữa 2 loại.

Khi sử dụng đồng thời *args **kwargs thì không thể truyền tham số bình thường theo tên

Ví dụ

```
def show_detail(name, *args, **kwargs):
    ...

show_detail(name='Coulson', 'agent', age='40', level='A')
>>> Lỗi

def show_detail_2(name, **kwargs):
    ...

show_detail_2(name=Coulson', age='40', level='A')
>>> Chạy OK
```

7.6 Lời gọi đệ quy

Một hàm có thể gọi lại *chính nó*, lúc này ta gọi là các lời gọi **đệ quy**[10] *recursion*.

Ví dụ, cho cấp số cộng u_n với $u_1 = 5$ và công sai $d = 7$ thì, chẳng hạn, muốn tìm số hạng thứ 20 ta lấy số hạng thứ 19 cộng thêm 7, nhưng muốn tính số hạng thứ 19 ta lại phải lấy số hạng thứ 18 cộng thêm 7 và cứ như thế cho đến số hạng thứ nhất thì đã có sẵn $u_1 = 5$. Tổng quát, ta có công thức truy hồi như sau

Nếu ta viết hàm `cap_so_cong(n)` để tính số hạng thứ n thì trong bản thân hàm này sẽ gọi lại chính nó với tham số truyền vào là `n - 1`:

```
def cap_so_cong(n):
    if n == 1:
          return 5
    else:
          return cap_so_cong(n-1) + 7
```

Lúc này, để tìm số hạng thứ `n` ta chỉ việc gọi hàm và truyền vào tham số `n`, chẳng hạn tính số hạng thứ `50` thì gọi hàm và truyền vào cho nó tham số `n = 50, cap_so_cong(50)`:

```
>>> cap_so_cong(50)
348
```

Một hàm có thể gọi lại chính nó *nhiều* lần, ví dụ, chúng ta xét dãy số Fibonacci

$$\begin{cases} u_1 = 1, u_2 = 1 \\ u_{n+2} = u_{n+1} + u_n \end{cases}$$

Chúng ta có chương trình như sau

10 Thực ra, thử tìm hiểu từ *đệ quy* thì thấy rất khó hiểu, ta cứ hiểu nôm na như là phép quy nạp trong Toán học.

```python
def fibonacci(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Khi chạy chương trình, thử tìm số Fibonacci thứ 30 ta được kết quả

```
>>> fibonacci(30)
832040
```

Tuy nhiên, trong lập trình, chúng ta nên hạn chế các lời gọi đệ quy, vì như thế chương trình sẽ cần rất nhiều bộ nhớ, chẳng hạn, ta tìm số Fibonacci thứ 100 thôi đã phải đợi rất lâu, vì bản thân mỗi lần gọi hàm, chương trình sẽ gọi lại chính nó hai lần và cứ như thế, do đó số lượng lời gọi hàm sẽ tăng lên rất nhiều với tốc độ *cấp số nhân*!

Bài tập

Bài 41. Viết hàm `cap_so_nhan(n)` để tìm số hạng thứ n của một cấp số nhân với số hạng đầu bằng 7 và công bội bằng 2.

Bài 42. Viết hàm `giai_thua(n)` để tính giai thừa[11] của số tự nhiên `n`, chẳng hạn `giai_thua(6)` trả về kết quả `720`.

Bài 43. Tìm tất cả các số tự nhiên hai chữ số mà khi đảo trật tự của hai chữ số đó sẽ thu được một số nguyên tố cùng nhau với số đã cho.

Bài 44. Tìm các số tự nhiên lẻ có ba chữ số. Ba chữ số này, theo trật tự từ trái qua phải tạo thành một cấp số cộng.

Bài 45. Tìm các số tự nhiên có ba chữ số. Ba chữ số này, theo trật tự từ trái qua phải tạo thành một cấp số nhân với công bội là một số tự nhiên khác `0`.

7.7 Biến toàn cục, biến cục bộ

Trong Python không có sự phân biệt giữa tham số và tham số giá trị như trong ngôn ngữ khác, Pascal chẳng hạn, nên tất cả các giá trị được truyền vào một hàm chúng ta đều gọi là **tham số** của hàm. Ở đây, chúng ta chỉ cần phân biệt khái niệm biến toàn cục *global* và biến cục bộ (biến địa phương) *local* mà thôi.

Một biến được định nghĩa bên *trong* một hàm chỉ có thể được sử dụng bên trong hàm đó và được gọi là biến cục bộ. Ví dụ:

7.8 Hàm nặc danh

Chúng ta sử dụng từ khoá `lambda` được dùng để tạo một hàm số nặc danh (một hàm số không có tên). Nó là một hàm số trên một dòng và không có câu lệnh `return`. Nó chỉ bao gồm một biểu thức mà kết quả của biểu thức này chính là giá trị trả về của hàm. Ví dụ:

11 Giai thừa của một số tự nhiên n, kí hiệu là $n!$, được tính bởi công thức

$$n! = n(n-1)(n-2)...2.1$$

```
a = lambda x: x*2
for i in range(1,6):
    print(a(i))
```

Kết quả

```
2
4
6
8
10
```

Ở đây, chúng ta tạo ra một hàm số chỉ trong một dòng lệnh, nó được dùng để nhân đôi giá trị của đối số nhận vào, sử dụng câu lệnh `lambda`. Sau đó, chúng ta dùng hàm này để nhân đôi các giá trị của một *list* gồm các số từ 1 đến 5.

Chương 8 Lớp và đối tượng

8.1 Khái niệm lớp

Trong tiếng Anh, **class** có nghĩa là **lớp**, **loài**. Nên dịch là *lớp* nhiều khi hơi không sát nghĩa lắm, thực ra, chúng ta nên hiểu là một (giống) loài, tương tự như khái niệm *lớp* trong sinh học. Mỗi cá thể trong một loài thường có những đặc điểm (tính chất) chung, giống nhau; và những đặc điểm riêng, khác nhau. Tất cả những đặc tiểm này gọi là các **thuộc tính** của lớp. Ví dụ, xét lớp cá thì đều có những đặc điểm giống nhau là sống ở dưới nước, thở bằng mang, có râu... nhưng xét từng con cá chép cụ thể thì sẽ khác nhau về khối lượng, kích thước… Tất cả những đặc điểm này đều gọi chung là các thuộc tính của *lớp cá chép*. Những con cá chép thì đều thể làm những hành động như bơi, thở, đớp mồi… Những hành động này gọi là các **phương thức** của lớp. Hoặc, như xét lớp các hình chữ nhật thì chúng đều có những *thuộc tính* là có bốn góc vuông, hai đường chéo cắt nhau tại trung điểm mỗi đường, giao điểm này chính là tâm đối xứng, đều có chiều dài và chiều rộng… Nhưng, mỗi một hình chữ nhật cụ thể thì lại có các thuộc tính chiều dài – chiều rộng lại là các con số (có thể) khác nhau, hoặc nếu đặt chúng vào trong một hệ trục tọa độ, để vẽ nó ra trên màn hình máy tính chẳng hạn, thì lại có các tọa độ đỉnh là những bộ số khác nhau...

Từ đây về sau, để thống nhất với các tài liệu hiện có, chúng ta thống nhất sẽ sử dụng từ **lớp** hoặc giữ nguyên từ **class** mà không dịch sang tiếng Việt.

8.2 Khái niệm đối tượng

Đối tượng là các *thực thể* của một lớp nào đó. Ví dụ như **list, dict, tuple**… là các lớp. Khi chúng ta khai báo một biến thuộc các lớp này, thì biến đó chính là đối tượng, tức là một thể hiện cụ thể của lớp đang xét. Trong Python, tất cả mọi thứ đều là đối tượng. Để xem một đối tượng thuộc lớp nào, chúng ta có thể sử dụng hàm `type()` như trong ví dụ sau:

8.3 Khai báo lớp

Để khai báo một *lớp* chúng ta sử dụng từ khóa `class` với cú pháp:

```
class ClassName([arg,... arg=value,... *arg, **arg]):
          """Mô tả về lớp nếu (nên) có"""
          <khối lệnh>
```

Trong đó `ClassName` là tên của hàm, được đặt tên theo đúng quy tắc đặt tên, và thường bắt đầu bởi một chữ cái in hoa.

Ví dụ, chúng ta khai báo lớp số phức `ComplexNumber` như sau:

8.4 Thuộc tính

8.5 Phương thức

8.6 Tính kế thừa

8.7 Tính đa hình

8.8 Các phương thức đặc biệt

Chương 9 Xử lý ngoại lệ

Chương 10 Làm việc với tệp

Chương 11 Thư viện

Phần này sẽ trình bày cách tạo một thư viện cũng như giới thiệu một số thư viện, ngoài các thư viện được phân phối sẵn cùng với chương trình dịch Python, mà *cá nhân* tôi thấy hữu ích và thường xuyên sử dụng.

Trước tiên, để cài đặt một thư viện, bạn có thể dùng nhiều cách, ở đây tôi sử dụng `pip` để cài đặt. Nếu trong lúc cài đặt Python, bạn đã chọn cài luôn `pip` thì không cần làm bước tiếp sau đây, còn nếu chưa, bạn phải vào đây https://pip.pypa.io/en/stable/installing/ để tải về tệp `get-pip.py`, sau đó chạy lệnh sau trong **CMD**

```
python get-pip.py
```

Để kiểm tra đã cài đặt thành công chưa, bạn có thể sử dụng lệnh `pip list` để liệt kê tất cả các thư viện Python đã được cài đặt trong máy của bạn.

11.1 Virtualenv

`Virtualenv`, viết tắt của Virtual Environment (môi trường ảo), là một công cụ tạo ra nhiều môi trường Python độc lập với nhau trên cùng một máy tính. Với mỗi môi trường độc lập này, bạn có thể thêm xóa các thư viện mới mà không làm ảnh hưởng đến các thư viện của môi trường khác. Mỗi môi trường độc lập này sẽ được lưu ở các vị trí khác nhau trên ổ cứng, tùy bạn lựa chọn. Ví dụ, bạn muốn tạo ra một môi trường Python để thử nghiệm thư viện `sympy` chẳng hạn, nhưng không muốn môi trường Python của máy bạn bị ảnh hưởng bởi gói sympy, hoặc bạn sợ xung đột giữa các thư viện khác nhau…

Python 3 được cài đặt sẵn gói `venv` để tạo môi trường làm việc ảo, đối với Python 2 bạn phải sử dụng `virtualenv`, đương nhiên Python 3 sử dụng `virtualenv` cũng được. Ở đây sẽ hướng dẫn với gói `virtualenv`. Để cài đặt `virtualenv`, bạn sử dụng câu lệnh sau trong cửa sổ CMD của Windows

```
pip install virtualenv
```

Khi đó, `pip` sẽ tự động tải về và cài đặt `virtualenv` cho bạn.

Để tạo một môi trường Python mới, bạn sử dụng cửa sổ CMD, chuyển đến thư mục cần tạo, chẳng hạn tôi muốn tạo một môi trường Python để thực hành Django tại thư mục `C:\LearnPython\DjangoTuts` thì trong cửa sổ CMD, tôi chuyển[12] đến thư mục này, và sử dụng lệnh

```
virtualenv [tên_môi_trường]
```

Khi đó, `virtualenv` sẽ tạo ra một thư mục có tên là `[tên_môi_trường]` chứa tất cả những gì cần thiết và cả những thư viện mà bạn đã cài đặt nữa. Nếu muốn tạo ra môi trường mới chỉ gồm những thư viện mặc định của Python, không bao gồm những thư viện bạn tự cài thêm thì dùng lệnh

12 Bạn sử dụng lệnh `cd` của CMD, ví dụ bạn đang ở `C:\Windows\System32` muốn chuyển đến thư mục `C:\LearnPython\DjangoTuts` thì dùng lệnh

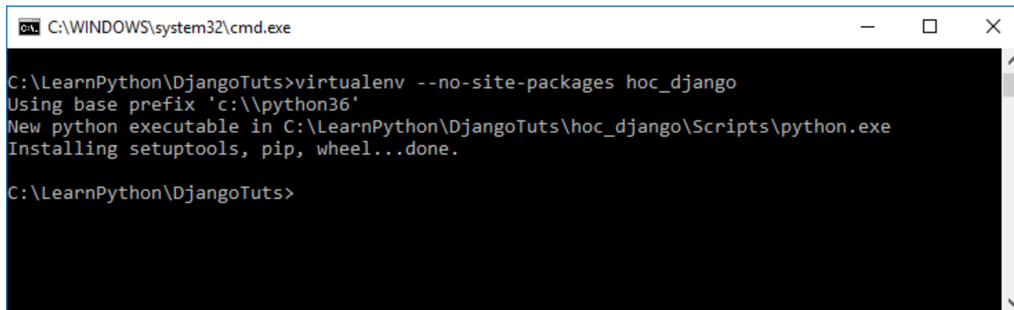
```
C:\Windows\System32>cd C:\LearnPython\DjangoTuts
```

Dĩ nhiên, nếu chưa có thư mục `C:\LearnPython\DjangoTuts` thì CMD sẽ báo lỗi.

```
virtualenv --no-site-packages [tên_môi_trường]
```

Ví dụ, ở đây tôi tạo ra một môi trường tên là hoc_django:

```
C:\LearnPython\DjangoTuts>virtualenv --no-site-packages hoc_django
```



Khi đó, virtualenv sẽ tạo thư mục hoc_django nằm trong thư mục C:\LearnPython\DjangoTuts gồm những gì cần thiết để chạy được Python. Lúc này, để kích hoạt activate môi trường hoc_python, trong **CMD**, bạn chuyển đến thư mục con C:\LearnPython\DjangoTuts\hoc_django\Scripts và chạy lệnh activate

```
C:\LearnPython\DjangoTuts\hoc_django\Scripts>activate
```

Cửa sổ CMD sẽ hiện như sau

```
(hoc_django) C:\LearnPython\DjangoTuts\hoc_django\Scripts>
```

tức là bạn đang làm việc với Python ở trong môi trường hoc_django, và có thể thoải mái cài thêm các thư viện cho riêng môi trường này, chẳng hạn tôi sẽ cài thêm thư viện django

```
(hoc_django) C:\LearnPython\DjangoTuts\hoc_django\Scripts>pip install
django
```

Để thoát khỏi môi trường này, vẫn trong cửa sổ CMD, ta dùng lệnh deactivate.

Để xóa bỏ một môi trường Python độc lập nào đó, bạn chỉ cần xóa thư mục đó. Hoặc dùng lệnh rmdir /s trong CMD.

```
C:\LearnPython\DjangoTuts>rmdir /s hoc_django
```

11.2 Sympy

Khai báo, có ba kiểu

```
from sympy import *
```

hoặc

```
import sympy
```

hoặc

```
from sympy import tên_mô-đun_cụ_thể
```

Thư viện sympy dùng để giải toán

Để khai báo các biến chúng ta dùng lệnh

```
x = Symbol('x')
x, y, z = symbols("x y z")
```

Xuất kết quả dạng LaTex

```
latex(bieu_thuc)
```

Đơn giản một biểu thức dùng lệnh

```
simplify(bieu_thuc)
```

Chuyển một biểu thức sang cú pháp sympy dùng lệnh

```
sympify(bieu_thuc)
```

Tính nguyên hàm của hàm f theo biến x

```
integrate(f, x)
```

Tính tích phân của hàm f theo biến x, trong đoạn [a,b]

```
integrate(f, (x, a, b))
```

Thay thế giá trị

```
>>> expr = cos(x) + 1
>>> expr.subs(x, y)
cos(y)+1
```

Giải phương trình, bất phương trình dùng lệnh solve(biểu thức, biến)

Nếu muốn tìm tập nghiệm của phương trình, bất phương trình thì dùng lệnh `solveset(biểu thức, biến, miền domain=nghiệm)`
miền nghiệm có thể là S.Reals, S.Complex,…

Chuyển một biểu thức sympy sang latex sử dụng câu lệnh

`latex(expr, **settings)`

Các tùy chọn gồm có…. Xin mời xem tại đây:
http://docs.sympy.org/latest/modules/printing.html#sympy.printing.latex.latex

Chương 12 Phụ lục

12.1 Mô tả từ khóa qua các ví dụ

True, False

True (đúng) và False (sai) là các giá trị chân lý trong Python. Chúng là kết quả của các phép toán so sánh hoặc các phép toán logic. Ví dụ,

```
>>> 1 == 1
True
>>> 5 > 3
True
>>> True or False
True
>>> 10 <= 1
False
>>> 3 > 7
False
>>> True and False
False
```

None

and, or , not

as

as được dùng để tạo một bí danh *alias* khi sử dụng một thư viện (module). Nghĩa là chúng ta sẽ đặt và sử dụng một tên *mới* cho thư viện đó.
Ví dụ, Python có một thư viện chuẩn là math. Giả sử chúng ta muốn tính cosin của pi trong thư viện math dưới tên toan_hoc chẳng hạn.

```
>>> import math as toan_hoc
>>> toan_hoc.cos(toan_hoc.pi)
-1.0
```

Ở đây chúng ta đã import thư viện math dưới tên mới là toan_hoc. Và thay vì sử dụng tên math, chúng ta sẽ sử dụng tên toan_hoc để tham chiếu đến thư viện math này. Cách này thường sử dụng khi tên thư viện dài dòng, hoặc có sự thay đổi tên module giữa các phiên bản Python 2 và 3 chẳng hạn, thì khi đổi sang phiên bản khác, chúng ta chỉ cần thay đổi một dòng mã lệnh.

assert

assert được sử dụng để gỡ lỗi.

While programming, sometimes we wish to know the internal state or check if our assumptions are true. `assert` helps us do this and find bugs more conveniently. `assert` is followed by a condition.

If the condition is true, nothing happens. But if the condition is false, `AssertionError` is raised. For example:

```
>>> a = 4
>>> assert a < 5
>>> assert a > 5
Traceback (most recent call last):
    File "<string>", line 301, in runcode
    File "<interactive input>", line 1, in <module>
AssertionError
```

For our better understanding, we can also provide a message to be printed with the `AssertionError`.

```
>>> a = 4
>>> assert a > 5, "The value of a is too small"
Traceback (most recent call last):
    File "<string>", line 301, in runcode
    File "<interactive input>", line 1, in <module>
AssertionError: The value of a is too small
```

At this point we can note that,

```
assert condition, message
```

is equivalent to,

```
if not condition:
    raise AssertionError(message)
```

break, continue

`break` được sử dụng trong vòng lặp `for` và `while` để điều thoát khỏi vòng lặp đó, tức là kết thúc vòng lặp và nhảy sang câu lệnh tiếp theo sau vòng lặp.

`continue` được sử dụng trong vòng lặp `for` và `while` để điều thoát khỏi lần lặp đó và nhảy sang lần lặp kế tiếp của vòng lặp, tức là vòng lặp không kết thúc mà chương trình chỉ bỏ qua lần lặp hiện tại.

```
for i in range(1,11):
    if i == 5:
          break
    print(i)
```

Kết quả thu được là

```
1
2
3
4
```

Ứng với mỗi giá trị `i` từ `1` đến `10` thì chương trình sẽ in ra giá trị `i` hiện tại, nếu gặp điều kiện `i == 5` thì thoát hoàn toàn khỏi vòng lặp. Do đó, chương trình sẽ lặp từ `i = 1` cho đến `i = 4`, đến khi `i = 5` thì điều kiện `i == 5` được thỏa mãn, nên câu lệnh `break` làm chương trình thoát khỏi vòng lặp.

```
for i in range(1,11):
    if i == 5:
        continue
    print(i)
```

Kết quả thu được là

```
1
2
3
4
6
7
8
9
10
```

Ứng với mỗi giá trị `i` từ `1` đến `10` thì chương trình sẽ in ra giá trị `i` hiện tại, nếu gặp điều kiện `i == 5` thì thoát khỏi *lần lặp* đó và bắt đầu lần lặp kế tiếp. Do đó, chương trình sẽ lặp từ `i = 1` cho đến `i = 4`, đến khi `i = 5` thì điều kiện `i == 5` được thỏa mãn, nên câu lệnh `continue` sẽ làm chương trình thoát khỏi lần lặp `i = 5`, thức là không in ra giá trị `i = 5`, mà nhảy sang lần lặp kết tiếp ứng với `i = 6.`

class

`class` được dùng để định nghĩa một lớp trong Python. Một lớp là một tập hợp các phần tử có các thuộc tính và phương thức tương tự nhau. Điều này cũng giống với khái niệm *lớp* trong sinh học.

Class is a collection of related attributes and methods that try to represent a real world situation. This idea of putting data and functions together in a class is central to the concept of object-oriented programming (OOP).

Classes can be defined anywhere in a program. But it is a good practice to define a single class in a module. Following is a sample usage:

```
class ExampleClass:
    def function1(parameters):
        ...
    def function2(parameters):
        ...
```

Để hiểu rõ hơn về *class,* xin xem chương **Lớp và đối tượng.**

def

`def` được dùng để tự định nghĩa một hàm.

Function is a block of related statements, which together does some specific task. It helps us organize code into manageable chunks and also to do some repetitive task.

The usage of `def` is shown below:

```
def function_name(parameters):
       …
```

del

`del` được dùng để xóa tham chiếu đến một đối tượng. Trong Python thì mọi *thứ* đều là đối tượng. Chúng ta có thể xóa tham chiếu đến một biến *variable,* tức là xóa biến đó và không thể sử dụng (tham chiếu) đến biến đó được nữa, sử dụng `del`**.**

```
>>> a = b = 5
>>> del a
>>> a
Traceback (most recent call last):
File "<string>", line 301, in runcode
File "<interactive input>", line 1, in <module>
NameError: name 'a' is not defined
>>> b
5
```

Ở đây, tham chiếu đến biến *a* đã bị xóa. Do đó, chương trình báo lỗi biến *a* chưa được định nghĩa. Còn biến *b* thì vẫn còn tồn tại.

`del` cũng được dùng để xóa một phần tử *item* từ một danh sách *list* hoặc một từ điển *dictionary*:

```
>>> a = ['x','y','z']
>>> del a[1]
>>> a
['x', 'z']
```

if, else, elif

`if, else, elif` được dùng cho câu lệnh kiểm tra điều kiện.

Khi chúng ta muốn kiểm tra một điều kiện và thực thi một khối lệnh nếu điều kiện đó đúng (xảy ra) ta dùng `if` và `elif.` `elif` là dạng viết gọn của `else if`. Khối lệnh sau từ khoá `else` sẽ được thực thi nếu điều kiện sai. Hãy xem một ví dụ đơn giản sau:

```
def if_example(a):
    if a == 1:
          print('Một')
    elif a == 2:
          print('Hai')
```

```
    else:
        print('Số khác')
if_example(2)
if_example(4)
if_example(1)
```

Kết quả

```
Hai
Số khác
Một
```

Ở đây, hàm `if_example` sẽ kiểm tra và in ra màn hình nếu số đưa vào là 1 hoặc 2. Nếu đưa vào hàm những số khác thì chương trình sẽ thực thi câu lệnh sau từ khoá `else`.

Chi tiết về phàn này, xin xem thêm ở chương **Câu lệnh điều khiển.**

except, raise, try

`except, raise, try` are used with exceptions in Python.

Exceptions are basically errors that suggests something went wrong while executing our program. `IOError`, `ValueError`, `ZeroDivisionError`, `ImportError`, `NameError`, `TypeError` etc. are few examples of exception in Python. `try...except` blocks are used to catch exceptions in Python.

We can raise an exception explicitly with the `raise` keyword. Following is an example:

```
def reciprocal(num):
    try:
        r = 1/num
    except:
        print('Exception caught')
        return None
    return r

print(reciprocal(10))
print(reciprocal(0))
```

Output

```
0.1
Exception caught
None
```

Here, the function `reciprocal()` returns the reciprocal of the input number.

When we enter 10, we get the normal output of 0.1. But when we input 0, a `ZeroDivisionError` is raised automatically.

This is caught by our `try…except` block and we return `None`. We could have also raised the `ZeroDivisionError` explicitly by checking the input and handled it elsewhere as follows:

```
if num == 0:
    raise ZeroDivisionError('cannot divide')
```

finally

`finally` is used with `try…except` block to close up resources or file streams.

Using `finally` ensures that the block of code inside it gets executed even if there is an unhandled exception. For example:

```
try:
    Try-block
except exception1:
    Exception1-block
except exception2:
    Exception2-block
else:
    Else-block
finally:
Finally-block
```

Here if there is an exception in the `Try-block`, it is handled in the `except` or `else` block. But no matter in what order the execution flows, we can rest assured that the `Finally-block` is executed even if there is an error. This is useful in cleaning up the resources.

Learn more about exception handling in Python programming.

for

`for` được dùng cho vòng lặp. Chúng ta sử dụng `for` khi chúng ta *biết chính xác* số lần chúng ta muốn lặp, nếu không biết số lần chúng ta lặp thì sẽ sử dụng vòng lặp `while`.

In Python we can use it with any type of sequence like a list or a string. Here is an example in which `for` is used to traverse through a list of names:

```
names = ['John','Monica','Steven','Robin']
for i in names:
    print('Hello '+i)
```

Output

```
Hello John
Hello Monica
Hello Steven
Hello Robin
```

from, import

`import` keyword is used to import modules into the current namespace. `from … import` is used to import specific attributes or functions into the current namespace. For example:

```
import math
```

will import the `math` module. Now we can use the `cos()` function inside it as `math.cos()`. But if we wanted to import just the `cos()` function, this can done using `from` as

```
from math import cos
```

now we can use the function simply as `cos()`, no need to write `math.cos()`.

Learn more on Python modules and import statement.

global

`global` được dùng để khai báo một biến nằm trong một hàm là biến toàn cục, tức là các hàm khác cũng có thể sử dụng biến này.

If we need to read the value of a global variable, it is not necessary to define it as `global`. This is understood.

If we need to modify the value of a global variable inside a function, then we must declare it with `global`. Otherwise a local variable with that name is created.

Following example will help us clarify this.

```
bien_toan_cuc = 10
def read():
    print(bien_toan_cuc)
def write1():
    global bien_toan_cuc
    bien_toan_cuc = 5
def write2():
    bien_toan_cuc = 15

read()
write1()
read()
write2()
read()
```

Kết quả:

```
10
5
5
```

Ở ví dụ này, hàm `read()` chỉ làm nhiệm vụ đọc và in giá trị của `bien_toan_cuc`. Nên chúng ta không cần phải khai báo nó là biến toàn cục. Nhưng hàm `write1()` lại thay đổi giá trị của biến `bien_toan_cuc` nên chúng ta phải khai báo nó là `global`. Ta có thể thấy sự khác nhau của hàm `write1()` và `write1()`, cùng thay đổi giá trị của biến `bien_toan_cuc` nhưng ở hàm `write2()` thì những thay đổi đó chỉ có tác dụng ở trong nội bộ bản thân hàm đó mà thôi, nghĩa là trong bản thân hàm đó thì một biến địa phương, cũng tên là `bien_toan_cuc` đã được tạo ra, và chỉ có phạm vi sử dụng trong hàm đó mà thôi.

in

`in` is used to test if a sequence (list, tuple, string etc.) contains a value. It returns `True` if the value is present, else it returns `False`. For example:

```
>>> a = [1, 2, 3, 4, 5]
>>> 5 in a
True
>>> 10 in a
False
```

The secondary use of `in` is to traverse through a sequence in a `for` loop.

```
for i in 'hello':
    print(i)
```

Output

```
h
e
l
l
o
```

is

`is` is used in Python for testing object identity. While the `==` operator is used to test if two variables are equal or not, `is` is used to test if the two variables refer to the same object.

It returns `True` if the objects are identical and `False` if not.

```
>>> True is True
True
>>> False is False
True
>>> None is None
True
```

We know that there is only one instance of `True`, `False` and `None` in Python, so they are identical.

```
>>> [] == []
True
>>> [] is []
False
>>> {} == {}
True
>>> {} is {}
False
```

An empty list or dictionary is equal to another empty one. But they are not identical objects as they are located separately in memory. This is because list and dictionary are mutable (value can be changed).

```
>>> '' == ''
True
>>> '' is ''
True
>>> () == ()
True
>>> () is ()
True
```

Unlike list and dictionary, string and tuple are immutable (value cannot be altered once defined). Hence, two equal string or tuple are identical as well. They refer to the same memory location.

lambda

`lambda` được dùng để tạo một hàm số nặc danh (một hàm số không có tên). Nó là một hàm số trên một dòng và không có câu lệnh `return`. Nó bao gồm một biểu thức mà kết quả của biểu thức này chính là giá trị trả về của hàm. Ví dụ:

```
a = lambda x: x*2
for i in range(1,6):
    print(a(i))
```

Kết quả

```
2
4
6
8
10
```

Ở đây, chúng ta tạo ra một hàm số chỉ trong một dòng lệnh, nó được dùng để nhân đôi giá trị của đối số nhận vào, sử dụng câu lệnh `lambda`. Sau đó, chúng ta dùng hàm này để nhân đôi các giá trị của một *list* gồm các số từ 1 đến 5.

nonlocal

The use of `nonlocal` keyword is very much similar to the `global` keyword. `nonlocal` is used to declare that a variable inside a nested function (function inside a function) is not local to it, meaning it lies in the outer inclosing function. If we need to modify the value of a non-local variable inside a nested function, then we must declare it with `nonlocal`. Otherwise a local variable with that name is created inside the nested function. Following example will help us clarify this.

```
def outer_function():
    a = 5
    def inner_function():
        nonlocal a
        a = 10
        print("Inner function: ",a)
    inner_function()
    print("Outer function: ",a)
```

```
outer_function()
```

Output

```
Inner function:  10
Outer function:  10
```

Here, the `inner_function()` is nested within the `outer_function`.

The variable *a* is in the `outer_function()`. So, if we want to modify it in the `inner_function()`, we must declare it as `nonlocal`. Notice that *a* is not a global variable.

Hence, we see from the output that the variable was successfully modified inside the nested `inner_function()`. The result of not using the `nonlocal` keyword is as follows:

```
def outer_function():
    a = 5
    def inner_function():
        a = 10
        print("Inner function: ",a)
    inner_function()
    print("Outer function: ",a)

outer_function()
```

Output

```
Inner function:  10
Outer function:  5
```

Here, we do not declare that the variable *a* inside the nested function is `nonlocal`. Hence, a new local variable with the same name is created, but the non-local *a* is not modified as seen in our output.

pass

`pass` is a null statement in Python. Nothing happens when it is executed. It is used as a placeholder.

Suppose we have a function that is not implemented yet, but we want to implement it in the future. Simply writing,

```
def function(args):
```

in the middle of a program will give us `IndentationError`. Instead of this, we construct a blank body with the `pass` statement.

```
def function(args):
    pass
```

We can do the same thing in an empty `class` as well.

```
class example:
    pass
```

return

`return` statement is used inside a function to exit it and return a value.

If we do not return a value explicitly, `None` is returned automatically. This is verified with the following example.

```
def func_return():
    a = 10
    return a

def no_return():
    a = 10

print(func_return())
print(no_return())
```

Output

```
10
None
```

while

`while` is used for looping in Python.

The statements inside a `while` loop continue to execute until the condition for the `while` loop evaluates to `False` or a `break` statement is encountered. Following program illustrates this.

```
i = 5
while(i):
    print(i)
    i = i – 1
```

Output

```
5
4
3
2
1
```

Note that `0` is equal to `False`.

Learn more about [Python while loop].

with

`with` statement is used to wrap the execution of a block of code within methods defined by the context manager.

Context manager is a class that implements `__enter__` and `__exit__` methods. Use of `with` statement ensures that the `__exit__` method is called at the end of the nested block. This concept is similar to the use of `try…finally` block. Here, is an example.

```
with open('example.txt', 'w') as my_file:
    my_file.write('Hello world!')
```

This example writes the text `Hello world!` to the file `example.txt`. File objects have `__enter__` and `__exit__` method defined within them, so they act as their own context manager.

First the `__enter__` method is called, then the code within `with` statement is executed and finally the `__exit__` method is called. `__exit__` method is called even if there is an error. It basically closes the file stream.

yield

`yield` is used inside a function like a `return` statement. But `yield` returns a generator.

Generator is an iterator that generates one item at a time. A large list of value will take up a lot of memory. Generators are useful in this situation as it generates only one value at a time instead of storing all the values in memory. For example,

```
>>> g = (2**x for x in range(100))
```

will create a generator *g* which generates powers of 2 up to the number two raised to the power 99. We can generate the numbers using the `next()` function as shown below.

```
>>> next(g)
1
>>> next(g)
2
>>> next(g)
4
>>> next(g)
8
>>> next(g)
16
```

And so on… This type of generator is returned by the `yield` statement from a function. Here is an example.

```
def generator():
    for i in range(6):
        yield i*i

g = generator()
for i in g:
    print(i)
```

Output

```
0
1
4
9
16
```

```
25
```

Here, the function `generator()` returns a generator that generates square of numbers from `0` to `5`. This is printed in the `for` loop.