

Giới thiệu

NỘI DUNG

1. Tại sao nên học lập trình Python?
2. Một số đặc điểm của ngôn ngữ lập trình Python
3. Phiên bản của Python và tài liệu học lập trình
4. Đối tượng học lập trình Python

Tại sao nên học lập trình Python?

Ngôn ngữ lập trình Python do Guido van Rossum phát triển từ những năm 1990. Cho đến nay ngôn ngữ này đã trải qua nhiều thay đổi và được đón nhận rộng rãi.



Biểu tượng của Python

Hiện nay Python là một trong những ngôn ngữ lập trình phổ biến nhất thế giới. Dù có nhiều bảng xếp hạng khác nhau, Python luôn đứng trong top ngôn ngữ lập trình phổ biến nhất cùng với C# và Java. Thêm vào đó, mức độ phổ biến của Python đang có xu hướng tăng.

Python phổ biến không chỉ trong phát triển ứng dụng mà còn cả trong nghiên cứu khoa học. Với nhiều đặc điểm quan trọng, Python được cộng đồng khoa học sử dụng rộng rãi trong các lĩnh vực như IoT, Data Science. Không nhiều ngôn ngữ lập trình có thể so sánh với Python về khía cạnh này.

Nhu cầu việc làm liên quan đến lập trình Python rất lớn. Nhiều công ty sử dụng Python như Google, IBM, EA Games.

Do vậy, việc giảng dạy ngôn ngữ Python đã được nhiều đơn vị đào tạo công nghệ thông tin đưa vào từ rất sớm. Thậm chí, một số nơi còn sử dụng Python khi dạy nhập môn lập trình.

Một số đặc điểm của ngôn ngữ lập trình Python

Ngôn ngữ Python hướng tới sự đơn giản, ngắn gọn, súc tích. Cú pháp của Python đơn giản hơn nhiều so với các ngôn ngữ như C/C++/Java/C#. Chương trình viết bằng Python thường ngắn gọn dễ đọc. Mã nguồn Python gần với ngôn ngữ tự nhiên. Vì vậy, ngôn ngữ Python thường được chọn cho các khóa học nhập môn lập trình.

Python là ngôn ngữ lập trình đa năng mạnh mẽ. Python cho phép phát triển nhiều loại ứng dụng (desktop, web) và hỗ trợ nhiều xu hướng lập trình khác nhau (lập trình hàm, lập trình hướng đối tượng). Vì vậy, Python là một lựa chọn tốt cho dù làm ngôn ngữ thứ nhất hay ngôn ngữ thứ hai.

Python có một cộng đồng đông đảo và tích cực. Là một ngôn ngữ phổ biến, Python có rất nhiều công cụ, thư viện hỗ trợ và tài liệu trợ giúp do cộng đồng đóng góp. Gần như bất kỳ vấn đề gì bạn quan tâm đều đã có giải pháp. Vì vậy việc học và sử dụng Python vô cùng tiện lợi.

Python là một ngôn ngữ kịch bản (script) và hoạt động dựa trên trình thông dịch (interpreter). Do đó Python có thể được sử dụng ở chế độ tương tác (interactive mode) hoặc chế độ kịch bản (script mode). Về điểm này Python rất gần với cách sử dụng của MatLab. Do vậy Python được sử dụng rộng rãi trong nghiên cứu khoa học.

Python hoạt động đa nền tảng. Lập trình viên có thể viết và chạy chương trình Python trên cả Windows, Mac và Linux. Việc cài đặt Python trên các hệ điều hành cũng rất đơn giản. Python thậm chí được cài đặt sẵn trên hầu hết các bản phân phối của Linux.

Phiên bản của Python và tài liệu học lập trình

Phiên bản mới nhất của Python là 3.8. Tuy nhiên có chút phức tạp về phiên bản của Python.

Khi Python 3 ra đời (2008), Python 2 vẫn tiếp tục được phát triển riêng rẽ cho đến năm 2010 (phiên bản 2.7) và hỗ trợ đến 2020. Python 2.7 là phiên bản Python 2 cuối cùng. Như vậy là hiện nay đang có hai "loại" Python tách biệt: Python 2 và Python 3.

Lưu ý rằng, Python 2.x và Python 3.x không tương thích nhau. Nghĩa là code viết trong Python 2.x có thể không chạy với Python 3.x và ngược lại.

Mặc dù phần lớn các tài liệu lập trình Python hiện nay là dành cho Python 3.x, bạn vẫn có thể gặp phải các tài liệu hướng dẫn dành cho Python 2.x.

Để dễ dàng phân biệt tài liệu dành cho phiên bản nào, hãy nhìn vào cách viết của hàm `print`, dùng để xuất dữ liệu ra màn hình console. Trong Python 2.x, `print` là một câu lệnh, còn trong Python 3.x, `print()` là một hàm. Do vậy cách sử dụng `print` là khác nhau:

- Trong Python 2.x: `print 'Hello world from Python'`
- Trong Python 3.x: `print('Hello world from Python')`

Đối tượng học lập trình Python

Python là ngôn ngữ lập trình dành cho nhiều loại đối tượng, từ người chưa từng học lập trình cho đến lập trình viên muốn học một ngôn ngữ thứ hai. Tùy đối tượng mà cách thức giới thiệu nội dung có thể tương đối khác biệt.

Tài liệu này hướng tới các bạn đã từng học một ngôn ngữ lập trình và muốn chuyển sang học Python làm ngôn ngữ thứ hai. Do đó, nội dung sẽ không tập trung vào trả lời câu hỏi *"là cái gì"*, mà sẽ tập trung trả lời *"dùng như thế nào trong Python"*.

Do vậy các chủ đề không tập trung vào giải thích các khái niệm cơ bản của lập trình (như biến, hằng, biểu thức, cấu trúc điều khiển,...). Thay vào đó các chủ đề hướng vào giải thích cách sử dụng của các thành phần của ngôn ngữ cũng như so sánh với tính năng tương đương ở các ngôn ngữ khác.

Tài liệu này cũng chứa nội dung liên quan đến lập trình hướng đối tượng. Bạn nên biết trước những khái niệm cơ bản của lập trình hướng đối tượng (như `class`, `object`, `inheritance`,...) trước khi đọc các nội dung liên quan. Các phần nội dung sẽ không giải thích lại chi tiết về các khái niệm này.

Cài đặt Python và môi trường phát triển ứng dụng

Trong phần này chúng ta sẽ học cách cài đặt Python và môi trường phát triển ứng dụng cho Python. Bạn cũng học cách làm việc với Python Interpreter ở chế độ tương tác và chế độ kịch bản cũng như cơ chế hoạt động của Python.

NỘI DUNG

1. Cài đặt Python
2. Hello world, Python!
 - 2.1. Chế độ tương tác
 - 2.2. Chế độ kịch bản
3. Sử dụng IDLE
4. Các IDE khác cho Python
5. Kết luận

Cài đặt Python

Để có thể làm việc với Python (viết/thực thi ứng dụng) bạn cần cài đặt Python. Do Python hoạt động đa nền tảng, quá trình cài đặt phụ thuộc vào hệ điều hành.

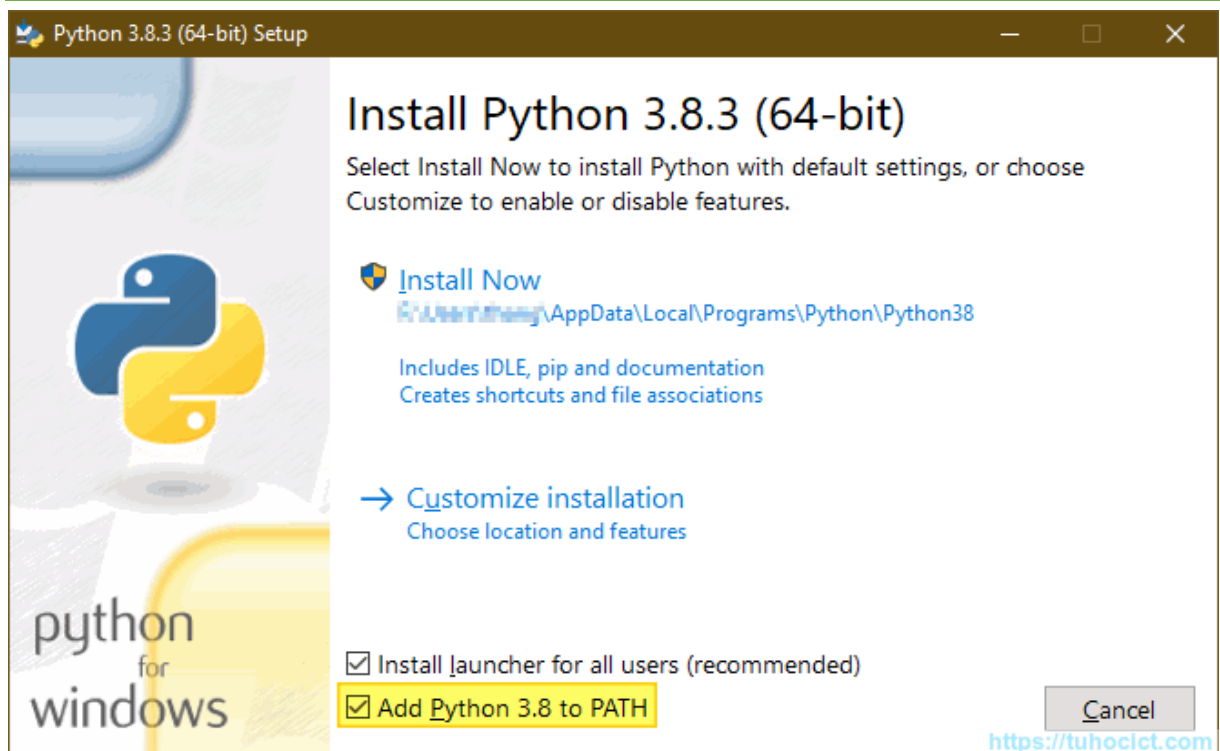
Để đơn giản hóa việc minh họa, chúng ta sẽ chỉ sử dụng hệ điều hành Windows. Việc cài đặt Python trên các hệ điều hành khác bạn có thể rất dễ tìm thấy trong các nguồn tài liệu khác như sách, Internet,...

Tải bản cài đặt Python cho Windows từ link sau:

<https://www.python.org/downloads/windows/>

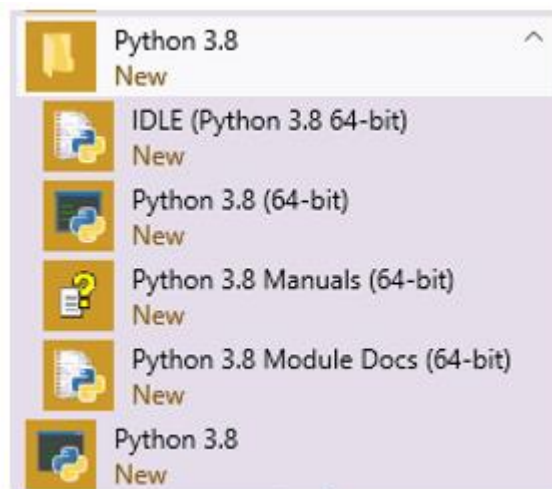
Bạn có thể lựa chọn bản cài đặt offline (executable installer) hoặc cài từ web (web-based installer) 32 hoặc 64 bit. Lưu ý Python không hỗ trợ Windows XP.

Khi chạy chương trình cài đặt lưu ý chọn Add Python to PATH để dễ dàng chạy Python từ dấu nhắc lệnh.



Lưu ý chọn Add Python to PATH

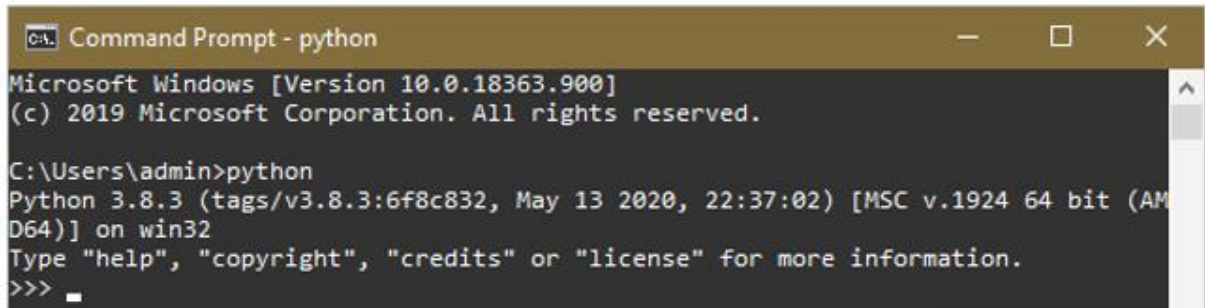
Khi hoàn thành trong Start Menu sẽ xuất hiện folder Python tương ứng:



Python 3.8 là shortcut đến chương trình python.exe – trình thông dịch của Python.

IDLE (Intergrated Development and Learning Environment) là một môi trường tích hợp đơn giản cho Python.

Để kiểm tra, bạn mở Command Prompt của Windows và gõ lệnh `python`. Kết quả thu được như sau:



Anaconda

Bạn có thể lựa chọn cài đặt Python thông qua Anaconda.

Anaconda là một gói phần mềm dành cho nghiên cứu khoa học máy tính sử dụng Python (và ngôn ngữ R) trong đó sử dụng một phiên bản riêng của Python. Gói Anaconda cũng đi kèm Spyder (môi trường phát triển cho Python), Kite (chương trình hỗ trợ viết code Python), Jupyter Notebook (ứng dụng xây dựng tài liệu học tập cho nhiều ngôn ngữ, trong đó có Python).

Nếu lựa chọn cài đặt Anaconda, bạn không cần cài đặt Python và IDE riêng rẽ.

Bạn có thể tải bản cài của Anaconda Individual Edition (miễn phí) từ đây: <https://www.anaconda.com/products/individual>. Có đầy đủ bộ cài cho cả Windows, macOS và Linux.

Khi cài đặt xong Anaconda Individual Edition bạn sẽ có đầy đủ công cụ để bắt đầu học lập trình Python.

Hello world, Python!

Python hoạt động ở hai chế độ: chế độ tương tác (interactive mode) và chế độ kịch bản (script mode).

Chế độ tương tác

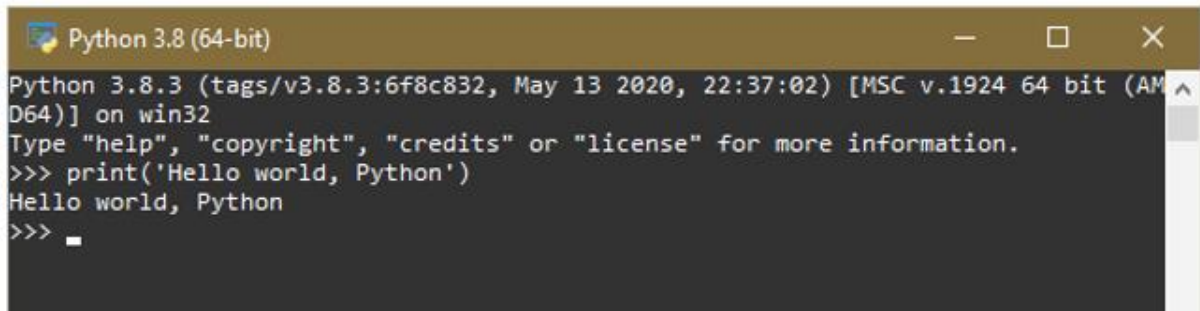
Để thử nghiệm hoạt động của Python ở chế độ tương tác bạn có thể sử dụng Python Interpreter hoặc Python IDLE.

Khi chạy hai chương trình trên bạn sẽ gặp dấu nhắc lệnh `>>>`. Từ vị trí này bạn có thể gõ bất kỳ lệnh Python nào. Trình thông dịch lệnh của Python sẽ dịch và thực thi lệnh ngay khi bạn kết thúc nhập lệnh với phím Enter.

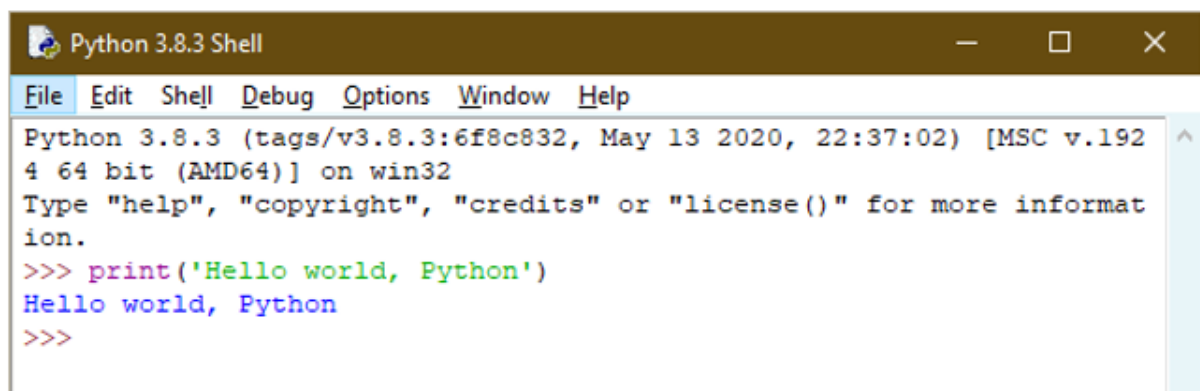
Hãy nhập lệnh sau (và ấn Enter):

```
>>> print('Hello world, Python') # đừng nhập dấu nhắc >>>
```

Bạn thu được kết quả như sau:



Sử dụng Python Interpreter



Sử dụng Python IDLE Shell

Ở **chế độ tương tác** (interactive mode) của Python bạn làm việc theo trình tự: nhập lệnh, ấn Enter => Python Interpreter dịch và thực thi lệnh.

Một số lưu ý khi sử dụng Python Interpreter ở chế độ tương tác:

- Ở chế độ này thông thường mỗi lần bạn chỉ nhập một lệnh (và ấn Enter).
- Nếu lệnh kéo dài trên một dòng, bạn kết thúc dòng bằng ký tự `\` (và ấn Enter).
- Nếu cần nhập nhiều lệnh, bạn phân tách các lệnh bằng ký tự `;` (ấn Enter khi kết thúc nhập lệnh cuối cùng).
- Nếu cần nhập nhiều lệnh trên nhiều dòng, bạn kết thúc mỗi lệnh bằng `;\` (và ấn Enter).

Chế độ tương tác phù hợp với việc học Python cũng như thực hiện các tác vụ tính toán. Để viết chương trình Python, bạn cần sử dụng chế độ kịch bản.

Mặc dù có thể sử dụng Python Interpreter, bạn nên sử dụng IDLE Shell. IDLE Shell hỗ trợ nhắc lệnh và hiển thị màu cho code như ở hình trên. Ngoài ra bạn cũng nên sử dụng một trong các IDE chuyên nghiệp sẽ được giới thiệu ở phần kế tiếp.

Chế độ kịch bản

Ở chế độ này, bạn viết lệnh vào một file mã nguồn có phần mở rộng py và chạy từ Command Prompt qua lệnh python. Hãy cùng thực hiện qua các bước sau:

1. Tạo file mã nguồn hello.py

Nếu sử dụng IDLE, chọn File -> New File (hoặc tổ hợp Ctrl + N) để tạo file mã nguồn mới. Lưu file này trong thư mục D:\Python\hello.py.

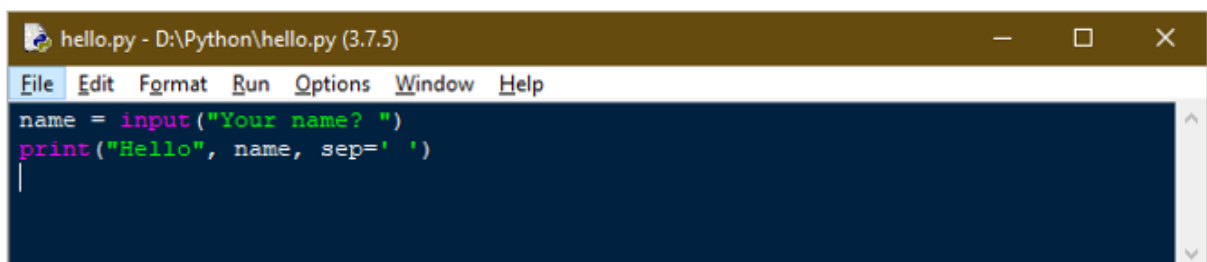
File mã nguồn của Python chỉ là một file văn bản thuần (plaintext) thông thường. Do đó bạn cũng có thể tạo và chỉnh sửa bằng một phần mềm xử lý văn bản như Notepad, Notepad++, Sublime Text.

2. Viết code cho hello.py

Viết code như sau cho hello.py.

```
name = input("Your name? ")
print("Hello", name, sep=' ')
```

Khi sử dụng IDLE code sẽ hiển thị với màu.

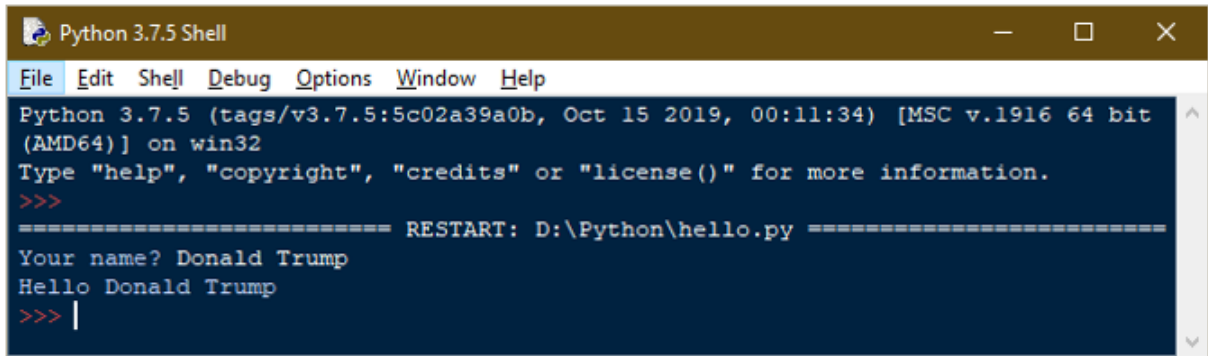


Các chương trình Notepad++ hay Sublime Text đều hỗ trợ hiển thị màu và nhắc code cho code Python. Nhìn chung để học Python bạn chỉ cần Notepad++ hay Sublime Text là đủ.

3. Chạy chương trình

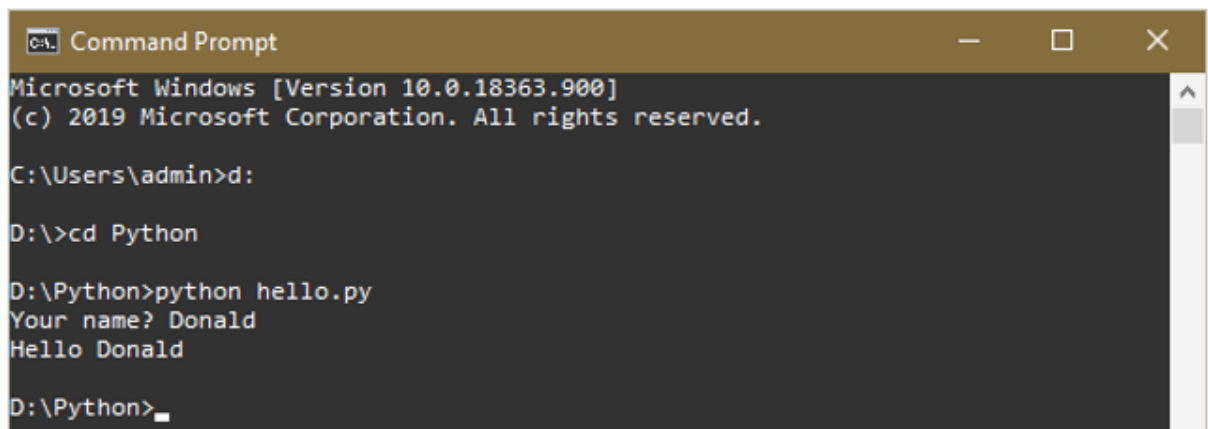
Bạn có một số cách khác nhau để chạy chương trình từ file mã nguồn Python.

Cách đơn giản nhất là chọn lệnh Run -> Run module từ cửa sổ biên tập code của IDLE. Bạn thu được kết quả như sau:



```
Python 3.7.5 Shell
File Edit Shell Debug Options Window Help
Python 3.7.5 (tags/v3.7.5:5c02a39a0b, Oct 15 2019, 00:11:34) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:\Python\hello.py =====
Your name? Donald Trump
Hello Donald Trump
>>> |
```

Cách thứ hai là chạy từ Command Prompt của Windows với lệnh python. Mở Command Prompt và di chuyển đến thư mục D:\Python. Từ dấu nhắc gõ lệnh `python hello.py` (hoặc `py hello.py`). Bạn thu được kết quả như sau:



```
Command Prompt
Microsoft Windows [Version 10.0.18363.900]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\admin>d:

D:\>cd Python

D:\Python>python hello.py
Your name? Donald
Hello Donald

D:\Python>_
```

Cách chạy chương trình ở chế độ kịch bản gần tương tự như chạy chương trình C# qua lệnh dotnet.

Cách thứ ba là thiết lập trong Windows để chương trình Python.exe là chương trình mở mặc định cho file py (tương tự như thiết lập Acrobat Reader làm chương trình mặc định cho file pdf). Khi này nếu bạn click đúp chuột vào file mã nguồn py, file mã nguồn sẽ được Python.exe thực hiện như một chương trình exe.

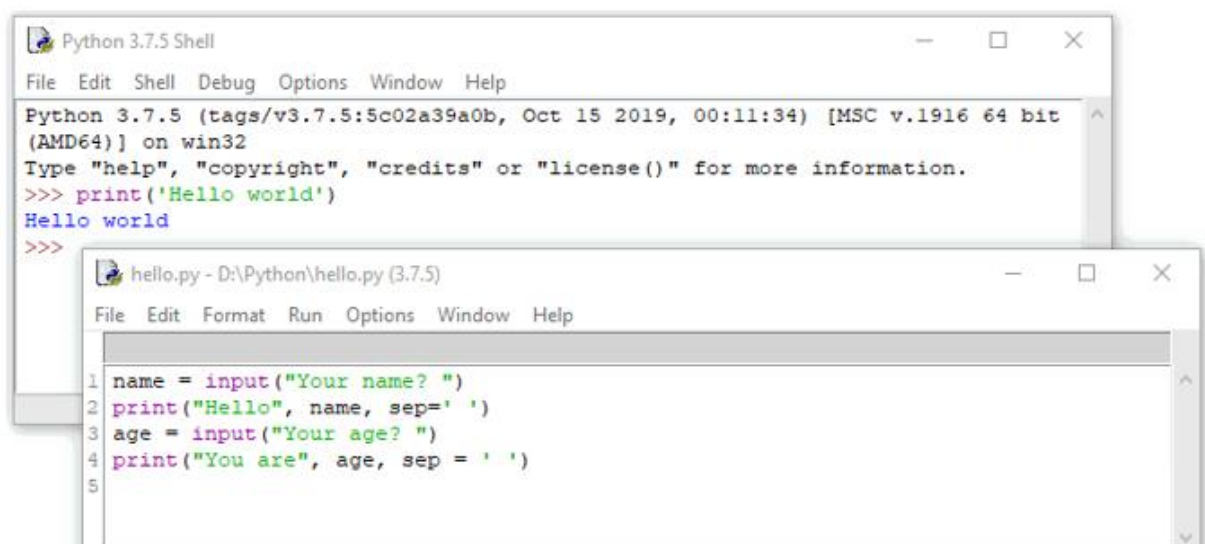


Trên Windows 10 bạn có thể click phải chuột vào file py và chọn Open with -> Python. Nếu không nhìn thấy Python trong danh sách, click Choose another app -> More app -> Look for other app on this PC. Chương trình chạy của Python nằm trong thư mục: {Ổ hệ thống}\Users\{tên user}\AppData\Local\Programs\Python\Python{số phiên bản}.

Bạn cũng có thể dùng {Ổ hệ thống}\Windows\py.exe. py.exe là trình khởi động (launcher) của Python trên Windows. Py.exe sẽ tự lựa chọn bản cài của Python để chạy (vì Python cho phép cài đặt nhiều phiên bản trên hệ thống).

Sử dụng IDLE

IDLE (Integrated Development and Learning Environment) là chương trình cài đặt mặc định cùng Python trên Windows. IDLE rất đơn giản và phù hợp để học lập trình Python. Nhìn chung ở giai đoạn học Python cơ bản bạn chỉ cần sử dụng IDLE là đủ.



IDLE cung cấp cả hai chế độ hoạt động: chế độ tương tác (qua cửa sổ Shell), chế độ kịch bản (qua cửa sổ Editor).

Mặc định IDLE sẽ mở cửa sổ Shell khi khởi động.

Từ cửa sổ Shell, để mở cửa sổ Editor bạn có thể tạo file mới hoặc mở một file script có sẵn.

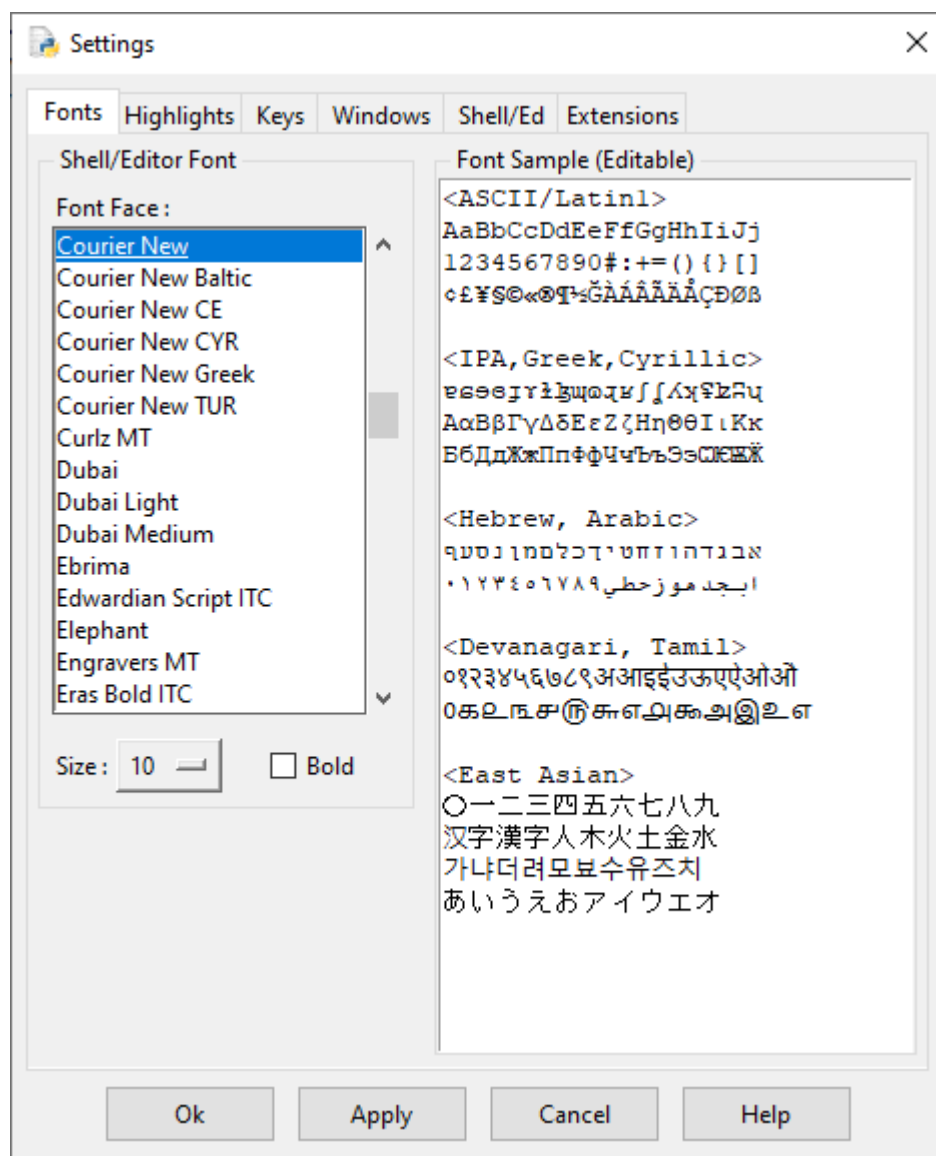
Từ cửa sổ Editor bạn có thể mở cửa sổ Shell bằng cách chọn Run -> Python Shell.

Bạn cũng có thể điều chỉnh để chọn Editor làm cửa sổ mặc định khi khởi động: Chọn Options -> Configure IDLE -> trong tab General chọn "Open Edit Window" mục At Startup.

Để sử dụng nhắc code bạn bấm tổ hợp Ctrl + Space.

Để chạy file script bạn mở file trong Editor và chọn Run -> Run module (hoặc ấn F5).

Tất cả các tùy chỉnh của IDLE đều nằm trong cửa sổ Settings (Options -> Configure IDLE):



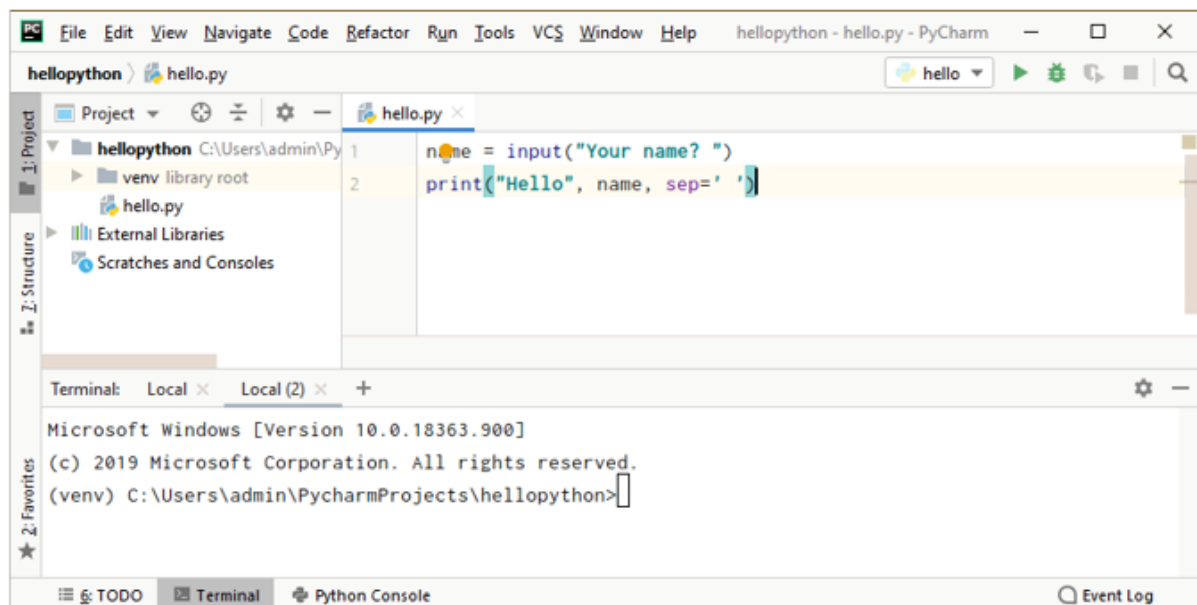
Các IDE khác cho Python

IDLE phù hợp khi học lập trình Python. Tuy nhiên khi làm dự án thực sự với Python bạn sẽ cần đến một IDE chuyên nghiệp hơn.

Hiện nay có khá nhiều IDE dành cho Python. Chúng ta sẽ điểm qua một số IDE phổ biến nhất. Bạn có thể tùy ý lựa chọn IDE theo ý thích. Với việc học lập trình Python, lựa chọn IDE không có nhiều ảnh hưởng.

PyCharm

PyCharm do JetBrains phát triển là IDE cho Python hàng đầu hiện nay. PyCharm có đầy đủ tất cả các tính năng bạn cần để phát triển những dự án phức tạp nhất bằng Python. PyCharm hoạt động đa nền tảng.

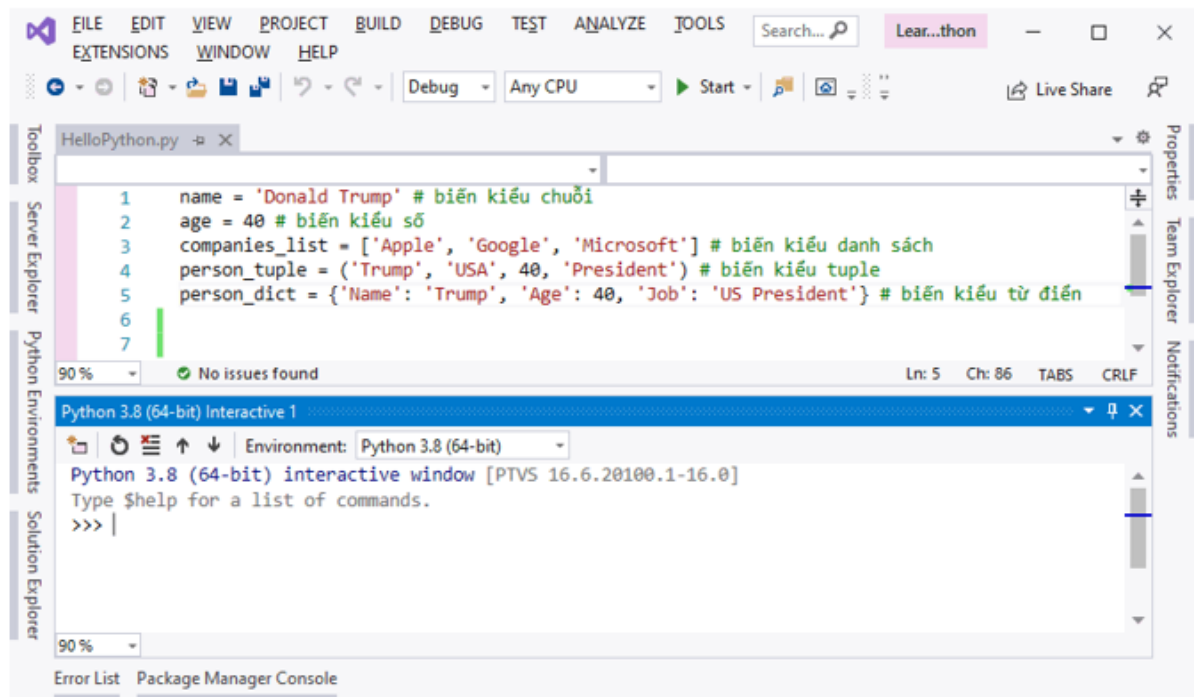


Bạn có thể sử dụng bản PyCharm Community (miễn phí) từ đường link sau:

<https://www.jetbrains.com/pycharm/download/>

Visual Studio

Từ Visual Studio 2019, Microsoft đưa vào workload Python giúp cài đặt tất cả các thành phần cần thiết để làm việc với Python. Đây là một IDE rất chuyên nghiệp dành cho cả việc học và làm project.



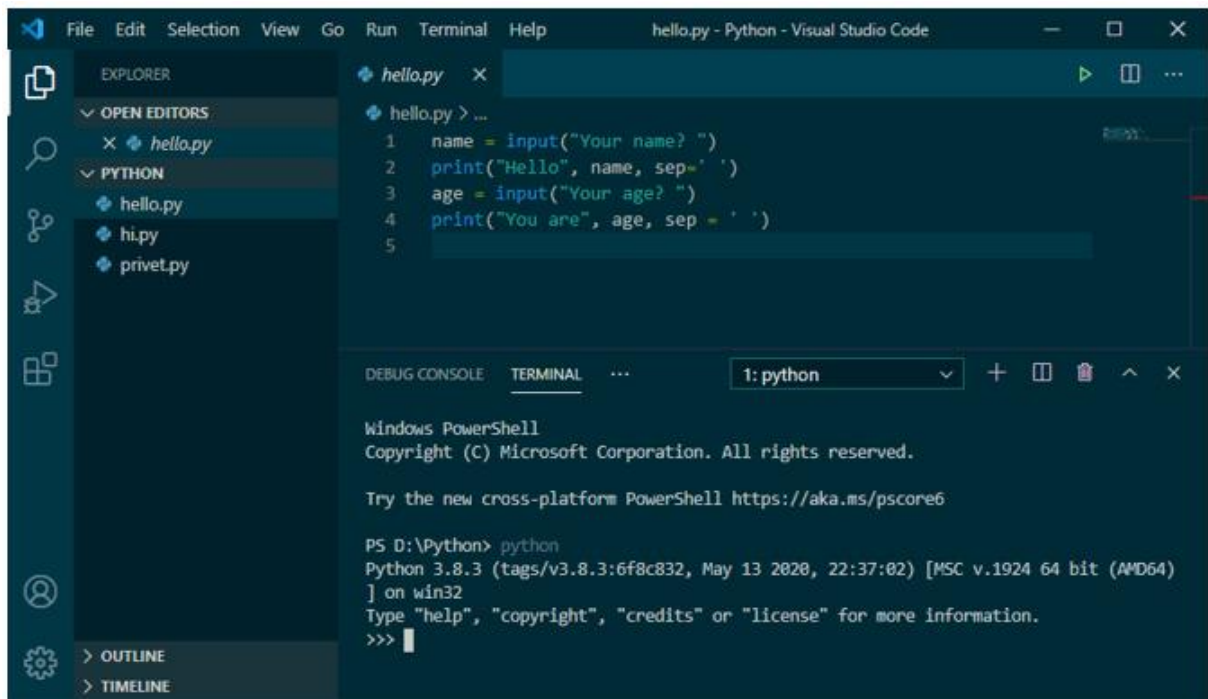
Với những bạn quen thuộc với công nghệ Microsoft, Visual Studio là lựa chọn hàng đầu.

Bạn có thể tải bản Visual Studio Community (miễn phí) từ đường link sau:

<https://visualstudio.microsoft.com/downloads/>

Visual Studio Code

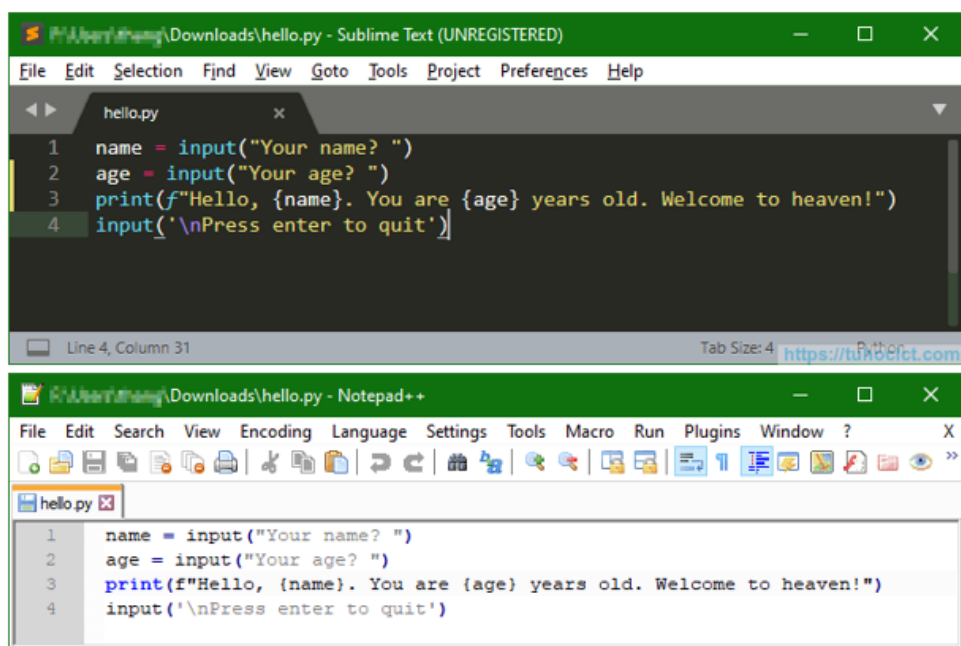
Visual Studio Code hoạt động đa nền tảng. Bạn có thể cài đặt extension Python để hỗ trợ lập trình Python. Khi cài đặt các extension phù hợp, Visual Studio Code hoạt động rất tốt với vai trò một IDE nhẹ và đơn giản cho cả việc học tập và làm việc với Python.



Visual Studio Code do Microsoft phát triển và cung cấp miễn phí tại đường link:

<https://code.visualstudio.com/download>

Nếu không thích IDLE hay các IDE phức tạp, bạn có thể chỉ cần **Notepad++** hay **Sublime Text** là đủ. Bạn có thể tải Notepad++ từ link <https://notepad-plus-plus.org/downloads/>, tải Sublime Text từ link <https://www.sublimetext.com/3>. Hai chương trình này mặc định đều hỗ trợ rất tốt cho Python như hiển thị màu code, nhắc code, thậm chí chạy trực tiếp code Python.



Kết luận

Trong phần này chúng ta đã học cách cài đặt Python và môi trường phát triển ứng dụng cho Python trên Windows. Có nhiều IDE khác nhau cho Python. Với mục đích học tập, việc lựa chọn IDE không có quá nhiều khác biệt. Bạn cũng học cách làm việc với Python Interpreter ở chế độ tương tác và chế độ kịch bản.

Cú pháp Python cơ bản

Ngôn ngữ lập trình Python có nhiều điểm chung với các ngôn ngữ như Perl, C# hay Java. Một số điểm trong Python hoàn toàn khác biệt với các ngôn ngữ khác. Nhìn chung, cú pháp của Python đơn giản hơn so với các ngôn ngữ trên.

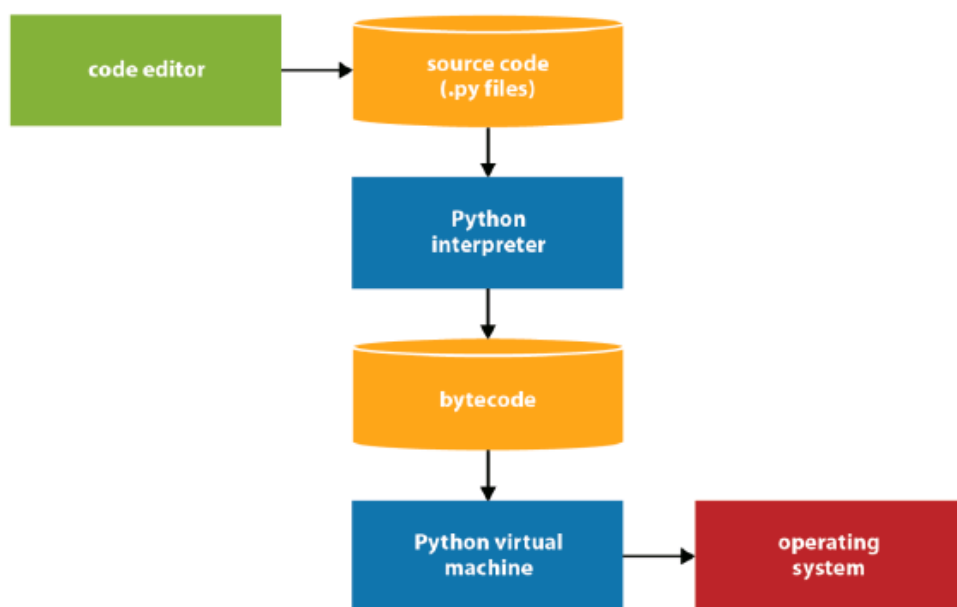
Tài liệu này sẽ cung cấp những nội dung về cú pháp chung của Python mà bạn cần lưu ý và ghi nhớ, đặc biệt là với những bạn có xuất phát điểm là một ngôn ngữ họ C.

NỘI DUNG

1. Cách hoạt động của chương trình Python
2. Định danh và từ khóa trong Python
3. Dòng lệnh và ghi chú trong Python
4. Thụt đầu dòng
5. Khối code trong Python
6. Xuất nhập dữ liệu với console
7. Kết luận

Cách hoạt động của chương trình Python

Hình dưới đây minh họa các bước trong quy trình viết code – dịch – chạy chương trình viết bằng Python.



Mã nguồn của Python là một file văn bản thông thường với phần mở rộng py. File mã nguồn Python có thể được mở và biên tập bởi bất kỳ phần mềm xử lý văn bản nào (như SubLime Text, Notepad++) hay bởi một IDE chuyên dụng (PyCharm, Visual Studio, Visual Studio Code, Spyder, Eclipse,...).

File mã nguồn py được trình thông dịch (**Python Interpreter**) chuyển sang một dạng mã trung gian gọi là **bytecode**. Mã bytecode không phụ thuộc vào platform nào. Đây là một dạng ngôn ngữ lập trình cấp thấp (không dành cho lập trình viên).

Bytecode được chương trình máy ảo (**Python virtual machine**) dịch tiếp thành mã máy tương ứng của platform và thực thi. Với cách thức này, chương trình Python có thể hoạt động trên bất kỳ platform nào với điều kiện đã có chương trình máy ảo Python tương ứng.

Trên thực tế chương trình máy ảo là một phần của trình thông dịch Python và hỗ trợ 3 platform quan trọng nhất hiện nay (Windows, Mac, Linux). Do vậy, chương trình viết bằng Python có thể gọi là chương trình hoạt động đa nền tảng.

Trình thông dịch của Python còn chấp nhận viết code trực tiếp (không cần viết file mã nguồn riêng). Chế độ hoạt động này có tên gọi là chế độ tương tác (**interactive mode**). Khi dịch file mã nguồn, trình thông dịch Python sẽ hoạt động ở chế độ kịch bản (**script mode**).

Định danh và từ khóa trong Python

Định danh là tên gọi của các thành phần trong chương trình Python như tên biến, tên hằng, tên kiểu, tên hàm,... Python đặt ra một số quy tắc chung mà tất cả các định danh phải tuân thủ. Nếu không tuân thủ các quy tắc đặt định danh sẽ dẫn tới lỗi cú pháp.

Quy tắc đặt định danh trong Python như sau:

- Định danh chỉ được phép chứa các ký tự thường (a-z), ký tự hoa (A-Z), chữ số (0-9), dấu gạch chân (_). Các ký tự khác (như @, \$, %, dấu cách,...) không được phép có mặt trong định danh. Có thể sử dụng ký tự unicode trong định danh (ví dụ, có thể đặt tên biến bằng tiếng Việt có dấu).

- Định danh chỉ được phép bắt đầu bằng ký tự (thường/hoa) hoặc dấu gạch chân. Tên gọi bắt đầu bằng ký số là không hợp lệ.
- Định danh không được trùng với một số từ dành riêng cho các mục đích đặc biệt (từ khóa).
- Định danh trong Python phân biệt chữ hoa/thường.

Dưới đây là danh sách từ khóa trong Python mà bạn không được sử dụng làm định danh. Lưu ý rằng tất cả từ khóa trong Python đều viết thường. Bạn tạm thời chưa cần quan tâm đến ý nghĩa của các từ khóa này.

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

Nhìn chung quy tắc đặt định danh trong Python tương tự như trong các ngôn ngữ họ C.

Ngoài các quy tắc (bắt buộc) trên, tùy từng thành phần sẽ có thêm các quy ước riêng khi đặt tên. Ví dụ tên class nên đặt theo kiểu PascalCase. Chúng ta sẽ nói chi tiết về quy ước đặt tên khi tìm hiểu về từng thành phần cụ thể.

Dòng lệnh và ghi chú trong Python

Trong Python, mặc định mỗi lệnh được viết trên một dòng. Kết thúc dòng cũng là báo hiệu kết thúc lệnh. Do vậy các lệnh trong Python không bắt buộc phải dùng dấu báo kết thúc (ký tự `;` trong C/C++/Java/C#). Ví dụ:

```
name = input("Your name? ")
age = int(input("Your age? "))
print(name, age, sep = ' ')
```

Nếu cần viết nhiều lệnh trên cùng một dòng, các lệnh được phân tách bằng ký tự `;` (giống như trong các ngôn ngữ khác). Ví dụ:

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Tuy nhiên lỗi viết này không được khuyến khích trong Python vì nó gây khó đọc code – đi ngược lại triết lý của Python.

Nếu một lệnh quá dài, bạn có thể viết tách nó ra nhiều dòng bằng cách thêm ký tự `\` vào cuối dòng. Ký tự `\` được gọi là **ký tự nối dòng** (line continuation character). Ví dụ:

```
total = item_one + \
        item_two + \
        item_three
```

Một số lệnh chứa các cặp dấu `[]`, `{}`, `()` có thể viết trên nhiều dòng mà không cần ký tự nối dòng `\`. Ví dụ:

```
days = ['Monday',
        'Tuesday',
        'Wednesday',
        'Thursday',
        'Friday']
```

Hiện tượng này được gọi là nối dòng ngầm định (implicit line continuation). Trong trường hợp này Python có thể tự xác định được là lệnh trải trên nhiều dòng.

Ghi chú trong Python được đánh dấu bắt đầu bằng ký tự `#`. Tất cả những gì nằm sau `#` cho đến hết dòng tương ứng được xem là ghi chú và sẽ bị tự động bỏ qua khi Python dịch mã nguồn. Ví dụ:

```
# First comment

print("Hello, Python!") # second comment

name = "Madisetti" # This is again comment

# This is a comment.

# This is a comment, too.

# This is a comment, too.

# I said that already.
```

Trong Python chỉ sử dụng ghi chú trên từng dòng, không có ghi chú nhiều dòng.

Một hoặc một số dòng để trống (hoặc chỉ chứa ghi chú) không có ảnh hưởng gì đến code. Python sẽ bỏ qua các dòng trống này. Sử dụng dòng trống để phân chia code giúp code dễ đọc hơn.

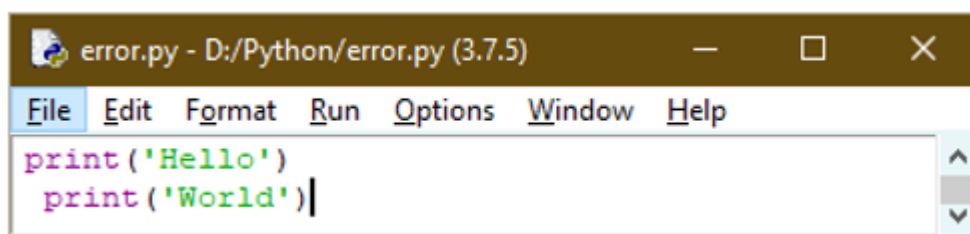
Thụt đầu dòng

Nếu bạn quen thuộc với các ngôn ngữ họ C, bạn sẽ thấy Python hơi khác lạ khi sử dụng dấu cách. Trong các ngôn ngữ họ C, dấu cách đầu dòng, cuối dòng hoặc giữa các phần tử không có giá trị. Nó chỉ giúp code dễ đọc hơn.

Trong Python, **dấu cách ở đầu dòng** lệnh có vai trò quan trọng giúp tạo ra cấu trúc code (thành các **khối code**).

Bạn không được tùy tiện thụt đầu dòng bằng cách dùng dấu cách vì có thể dẫn đến sai logic hoặc sai cú pháp.

Trong hình minh họa dưới đây, dòng code thứ hai sẽ bị báo lỗi *"unexpected intent"* do nó bị thụt vào 1 dấu cách.



Với đặc điểm này, khi viết code Python bạn không được tùy ý viết thụt đầu dòng bằng dấu cách hoặc dấu tab.

Chỉ thụt đầu dòng bằng dấu tab khi bạn cần viết khối code. Dấu cách ở các vị trí khác không có ý nghĩa gì đặc thù trong Python.

Khối code trong Python

Một số lệnh có thể tạo thành một **khối code**. Khối code là một/nhóm lệnh có ranh giới với phần code khác xung quanh.

Lấy ví dụ, nếu bạn muốn lặp lại việc thực hiện một số lệnh, bạn cần đặt những lệnh này vào một khối code trong thân của cấu trúc lặp. Python cần phân biệt được khối code nằm trong thân của vòng lặp với các phần code khác.

Khối code cũng giúp phân chia phạm vi của biến (variable scope) – nơi bạn có thể sử dụng một biến cục bộ.

Để tạo ra khối code, các ngôn ngữ họ C sử dụng cặp dấu { }, Pascal/Delphi sử dụng cặp begin/end, Visual Basic sử dụng nhiều cặp từ khóa (như sub/end sub, if/end if,...).

Một khối code trong Python được tạo thành bằng cách viết các lệnh với cùng số thụt đầu dòng (indentation). Tất cả code với cùng số thụt đầu dòng được xem là nằm trong một khối code.

Thông thường, các IDE Python sử dụng dấu tab để thụt đầu dòng với quy ước một thụt đầu dòng là 4 dấu cách.

Hãy xem ví dụ sau:

```
import sys

                                Khối code chính
try:
    # open file stream
    file = open(file_name, "w")
except IOError:
    print "There was an error writing to", file_name
    sys.exit()                                Khối code thân except
print "Enter '", file_finish,
print "' When finished"
while file_text != file_finish:
    file_text = raw_input("Enter text: ")
    if file_text == file_finish:
        # close the file
        file.close()
        break                                Khối code thân if
    file.write(file_text)
    file.write("\n")                        Khối code thân vòng while
file.close()
file_name = raw_input("Enter filename: ")
if len(file_name) == 0:
    print "Next time please enter something"
    sys.exit()                                Khối code thân if
try:
    file = open(file_name, "r")            Khối code thân try
except IOError:
    print "There was an error reading file"
    sys.exit()                                Khối code thân except
file_text = file.read()
file.close()
print file_text
```

Hình trên minh họa một chương trình Python với các khối code đã được đánh dấu. Bạn chưa cần hiểu nội dung code. Hãy tập trung vào cách phân chia khối code thông qua thụt đầu dòng.

Cách phân chia khối code này giúp Python dễ đọc và ngắn gọn hơn so với các ngôn ngữ lập trình khác.

Xuất nhập dữ liệu với console

Việc học một ngôn ngữ lập trình thường bắt đầu với giao diện console. Khi làm việc với giao diện này, việc xuất nhập dữ liệu có vai trò quan trọng. Python sử dụng các phương thức sau để làm việc với console:

- `print()` – xuất dữ liệu ra console.
- `input()` – nhập dữ liệu từ console.

Hãy xem các ví dụ sau:

```
print('Please wait while the program is loading...') # in ra một chuỗi
print('Hello', 'world', 'from', 'Python') # in nhiều giá trị, tách nhau bằng dấu cách
print('Hello', 'world', 'from', 'Python', sep = '\t') # in tách các giá trị bằng dấu tab
print(['Microsoft', 'Apple', 'Google']) # in danh sách
```

Trong phương thức `print` bạn có thể sử dụng giá trị (như ở trên), biến, hoặc biểu thức.

Nếu dùng chế độ tương tác:

```
>>> print('Please wait while the program is loading...')
```

```
Please wait while the program is loading...
```

```
>>> print('Hello', 'world', 'from', 'Python')
```

```
Hello world from Python
```

```
>>> print('Hello', 'world', 'from', 'Python', sep = '\t')
```

```
Hello world from Python
```


```
>>> print(['Microsoft', 'Apple', 'Google'])
```

```
['Microsoft', 'Apple', 'Google']
```

```
>>>
```

Dưới đây là ví dụ về nhập dữ liệu từ console sử dụng phương thức input:

```
name = input("Your name: ")
age = input("Your age: ")
location = input("Your location: ")
print('Hello', name, age, 'from', location)
```

A screenshot of a Python 3.7.5 Shell window. The window title is "Python 3.7.5 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main text area shows the following output: "Python 3.7.5 (tags/v3.7.5:5c02a39a0b, Oct 15 2019, 00:11:34) [MSC v.1916 64 bit (AMD64)] on win32", "Type 'help', 'copyright', 'credits' or 'license()' for more information.", and a separator line "===== RESTART: D:/Python/input.py =====". Below the separator, the program's output is displayed: "Your name: Donald Trump", "Your age: 50", "Your location: USA", and "Hello Donald Trump 50 from USA". The prompt ">>>|" is visible at the bottom of the text area.

```
Python 3.7.5 (tags/v3.7.5:5c02a39a0b, Oct 15 2019, 00:11:34) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:/Python/input.py =====
Your name: Donald Trump
Your age: 50
Your location: USA
Hello Donald Trump 50 from USA
>>> |
```

Kết luận

Trong phần này chúng ta nhắc đến 3 khái niệm cơ bản nhất trong Python (cũng như trong các ngôn ngữ lập trình khác): định danh, dòng lệnh và khối code. Ngoại trừ định danh, trong Python, dòng lệnh và khối code có nhiều khác biệt với các ngôn ngữ khác, nhất là các ngôn ngữ họ C. Bạn nên lưu ý những vấn đề này trước khi tìm hiểu sâu hơn về Python.

Biến, phép gán và các kiểu dữ liệu cơ sở trong Python

Biến và phép gán là những thành phần cơ bản và thường gặp nhất trong bất kỳ ngôn ngữ lập trình nào. Trong Python, biến, phép gán và kiểu dữ liệu có nhiều điểm khác biệt với C# hay Java do Python là một ngôn ngữ **định kiểu động** và **định kiểu yếu**.

Phần này sẽ trình bày chi tiết vấn đề sử dụng biến, phép gán trong Python. Chúng ta cũng sẽ tìm hiểu sơ lược về các kiểu dữ liệu cơ bản của Python.

NỘI DUNG

1. Biến, phép gán, kiểu dữ liệu trong Python
2. Biến trong Python
3. Phép gán trong Python
4. Các kiểu dữ liệu cơ bản của Python
5. Kết luận

Biến, phép gán, kiểu dữ liệu trong Python

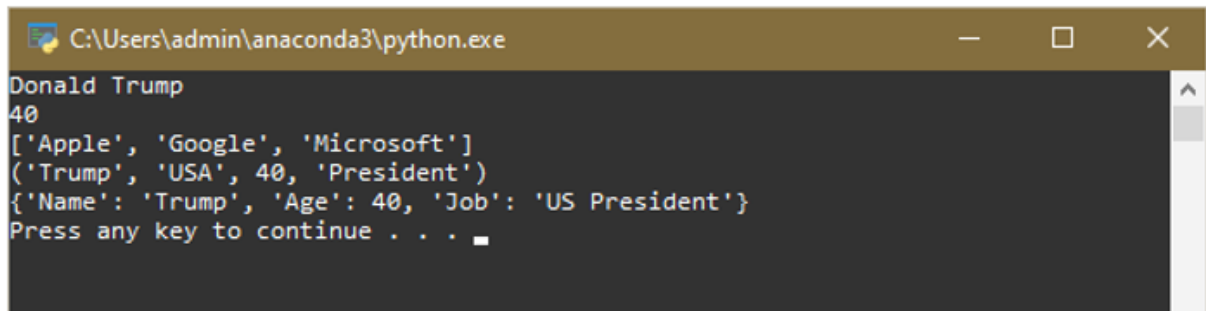
Hãy xem một số lệnh sau:

```
name = 'Donald Trump' # biến kiểu chuỗi
age = 40 # biến kiểu số
companies = ['Apple', 'Google', 'Microsoft'] # biến kiểu danh sách
person_tuple = ('Trump', 'USA', 40, 'President') # biến kiểu tuple
person_dict = {'Name': 'Trump', 'Age': 40, 'Job': 'US President'} # biến kiểu từ điển

print(name)
print(age)
print(companies)
print(person_tuple)
print(person_dict)
```

Bạn có thể tạo file py sử dụng IDLE, PyCharm, hay Visual Studio Code.

Chạy chương trình ở chế độ script bạn sẽ được kết quả như sau:

A screenshot of a Python interpreter window titled 'C:\Users\admin\anaconda3\python.exe'. The window has a dark background and shows the following text: 'Donald Trump', '40', '['Apple', 'Google', 'Microsoft']',>('Trump', 'USA', 40, 'President')',{'Name': 'Trump', 'Age': 40, 'Job': 'US President'}', and 'Press any key to continue . . .'. The text is displayed in a monospaced font with some color coding: 'Donald Trump' is white, '40' is light blue, the list and tuple are light blue, and the dictionary is white. The prompt 'Press any key to continue . . .' is followed by a small white cursor.

Nếu sử dụng chế độ tương tác, bạn nhập lệnh như sau trong Python Interpreter hoặc IDLE:

```
>>> name = 'Donald Trump'

>>> name

'Donald Trump'

>>> age = 40

>>> age

40

>>> companies = ['Apple', 'Google', 'Microsoft']

>>> companies

['Apple', 'Google', 'Microsoft']

>>> person_tuple = ('Trump', 'USA', 40, 'President')

>>> person_tuple

('Trump', 'USA', 40, 'President')

>>> person_dict = {'Name': 'Trump', 'Age': 40, 'Job': 'US President'}

>>> person_dict

{'Name': 'Trump', 'Age': 40, 'Job': 'US President'}

>>>
```

Đây là các lệnh khai báo và gán giá trị cho biến với một số kiểu dữ liệu cơ sở của Python.

Trong ví dụ trên, `name`, `age`, `companies`, `person_tuple`, `person_dict` là các **biến**, dấu `=` là dấu **phép gán**, bên phải dấu `=` là **giá trị của biến**.

Biến trong Python

Định nghĩa và cách hoạt động của biến trong Python có điểm khác biệt với các ngôn ngữ như C#.

Trong Python, biến là một tên gọi đại diện cho một giá trị trong bộ nhớ. Trong đó, giá trị có thể do bạn tự khai báo hoặc được Python tính toán từ một biểu thức.

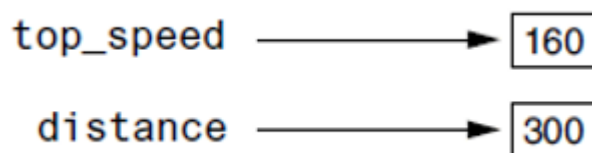
Để dễ hình dung, biến trong Python có nét tương tự như file và shortcut trong Windows. Trong đó, file trên ổ đĩa tương tự như giá trị của biến trong bộ nhớ. Shortcut (lối tắt) đến file tương tự như biến trong Python.

Với hình dung như thế, trong Python:

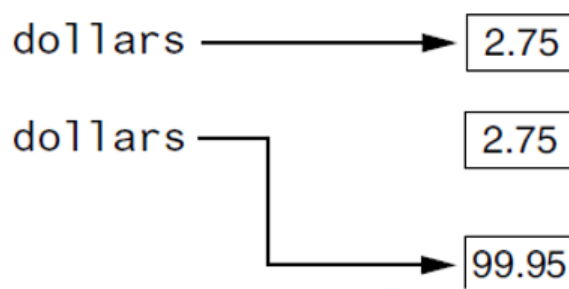
- Nhiều biến có thể “trỏ” đến cùng một giá trị trong bộ nhớ: giống như bạn có thể tạo nhiều lối tắt tới cùng một file, bạn có thể để cho nhiều biến khác nhau cùng trỏ tới một giá trị chung.
- Một biến có thể trỏ sang các giá trị khác nhau: khi tạo ra biến bạn có thể để cho nó trỏ tới giá trị a, sau đó bạn có thể để cho nó trỏ tới giá trị b, trong đó a và b hoàn toàn khác kiểu nhau.

Như vậy, biến trong Python thực tế chỉ là một tên gọi mà ta có thể sử dụng để truy xuất giá trị. Biến trong Python không trực tiếp chứa giá trị.

Biến trong Python gần tương tự như con trỏ (pointer) trong C/C++ hay kiểu tham chiếu (reference) trong C#.



Ví dụ, `top_speed` trỏ tới giá trị 160 (nằm đâu đó trong bộ nhớ), `distance` trỏ tới giá trị 300.



Biến có thể trở sang giá trị khác (bỏ lại giá trị cũ trong bộ nhớ). Giá trị cũ không sử dụng tới sẽ bị thu dọn tự động.

Biến trong Python bắt buộc phải tạo ra cùng phép gán. Giá trị của biến trong bộ nhớ được tự động “thu dọn” khi không sử dụng đến nhờ một tiến trình gọi là Garbage Collection.

Nếu bạn quen thuộc với Java hay C# thì sẽ không xa lạ với Garbage Collection.

Tên biến bắt buộc phải tuân thủ quy tắc đặt định danh trong Python. Ngoài ra có một số kiểu đặt tên biến bạn nên tuân theo.

- Kiểu 1: Ký tự đầu tiên viết thường; các chữ cái trong từ viết thường; nếu tên có nhiều từ thì viết hoa chữ cái đầu mỗi từ tiếp theo. Ví dụ `companySortedList`, `firstName`, `lastName`. Đây là lỗi đặt tên thường gặp trong các ngôn ngữ kiểu C và được gọi là kiểu camelCase.
- Kiểu 2: tất cả ký tự trong tên gọi đều viết thường; nếu tên có nhiều từ thì phân tách bằng dấu gạch chân `_`. Ví dụ `first_name`, `last_name`, `company_sorted_list`. Đây là quy ước dùng phổ biến hơn trong Python.

Có thể hơi lạ một chút: trong Python không có khái niệm hằng (constant) như trong các ngôn ngữ khác. Nếu muốn sử dụng hằng, bạn hãy tạo một biến và đặt tên toàn bộ là chữ hoa để nhớ rằng mình không được thay đổi giá trị của “biến” này!

Phép gán trong Python

Như ở trên đã trình bày về cách hoạt động của biến, phép gán trong Python có nhiệm vụ “ghép đôi” biến với giá trị.

Ví dụ, trong đoạn lệnh sau:

```
a = 1000
b = a
c = b
```

Lệnh gán `a = 1000` sẽ tạo ra một giá trị số nguyên 1000 trong bộ nhớ và ‘a’ được dùng để trỏ tới giá trị này. ‘a’ đại diện cho giá trị 1000 trong

bộ nhớ chữ 'a' không trực tiếp chứa giá trị 1000. Sử dụng 'a' trong chương trình tương đương với sử dụng giá trị 1000 từ bộ nhớ.

Lệnh gán `b = a` tạo tiếp biến `b` trở tới cùng giá trị với `a`, tức là cùng trở tới giá trị 1000 trong bộ nhớ. Khi này `b` và `a` thực tế là như nhau. Tương tự, lệnh gán `c = b` tạo tiếp biến `c` và cùng trở tới giá trị của `b` (và cũng là giá trị của `a`). Như vậy, `a`, `b` và `c` thực tế là cùng một giá trị nhưng khác tên gọi. Giống như cùng một file nhưng bạn có thể tạo nhiều shortcut tới nó.

Trong Python,

1. khai báo biến luôn phải đi kèm gán giá trị;
2. không cần chỉ định kiểu dữ liệu;
3. không cần từ khóa nào khi khai báo biến.

Để tiện lợi, Python cho phép ghép các phép gán như sau:

```
a = b = c = 1000 # tạo ra 3 biến a, b, c và cùng trở tới một giá trị
first_name, last_name, age = "Donald", "Trump", 40 # tạo 3 biến và trở tới 3 giá trị khác nhau (theo thứ tự tương ứng)
```

Bạn có thể thử ở chế độ tương tác như sau:

```
>>> fname, lname, age = "Donald", "Trump", 40
>>> fname
'Donald'
>>> lname
'Trump'
>>> age
40
>>> a = b = c = 1000
>>> a
1000
>>>
>>> b
1000
>>> c
1000
>>>
```

Tức là bạn có thể tạo ra nhiều biến cùng lúc và cho chúng trở tới cùng một giá trị, hoặc bạn cũng có thể đồng thời khai báo nhiều biến trở tới nhiều giá trị khác nhau.

Lưu ý, trong phép gán `a = b = c = 1000`. Nếu bạn tiếp tục gán `a = 2000` sau đó, Python sẽ tạo ra một giá trị mới (2000) và `a` trở sang một giá trị

mới này. Tuy nhiên b và c vẫn trở sang giá trị cũ (1000). Điều tương tự xảy ra nếu giá trị là chuỗi. Số và chuỗi là các kiểu dữ liệu bất biến (immutable) trong Python.

Kiểu dữ liệu không ảnh hưởng đến phép gán. Nghĩa là ban đầu bạn gán cho biến giá trị số, sau này có tiếp tục gán cho biến đó giá trị chuỗi,... Ví dụ:

```
>>> a = 1000 # ban đầu a trở tới giá trị số
>>> a
1000
>>> a = "Donald Trump" # a trở tới giá trị chuỗi
>>> a
'Donald Trump'
>>>
```

Trong ví dụ trên, ban đầu a trở tới giá trị số nguyên, sau đó a lại trở tới giá trị chuỗi ký tự.

Các kiểu dữ liệu cơ bản của Python

Khi khai báo và gán giá trị cho biến bạn không cần chỉ định kiểu dữ liệu. Python tự động xác định kiểu dữ liệu. Trong phần này chúng ta chỉ điểm qua một số nét chính của các kiểu dữ liệu này. Trong các bài học sau sẽ đi vào chi tiết của từng kiểu.

Python có các loại kiểu dữ liệu cơ bản: số, chuỗi, danh sách, tuple, từ điển, boolean.

Kiểu số trong python chia làm 3 loại: số nguyên (int), số thực (float), số phức (complex). Dưới đây là ví dụ về biểu diễn giá trị số trong Python:

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j

0o70	32.3+e18	.876j
-0o470	-90.	- .6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2-E12	4.53e-7j

Số nguyên có thể viết ở cơ số 10 (mặc định), cơ số 8 (bắt đầu bằng 0o), cơ số 16 (bắt đầu bằng 0x). Kiểu float phải có dấu chấm thập phân và có thể viết ở dạng khoa học. Kiểu complex phải có ký tự ảo j (hoặc J).

Chú ý: Python 2.x có hai kiểu số nguyên int và long. Kiểu long cần biểu diễn với hậu tố L (hoặc l). Số nguyên ở hệ cơ số 8 viết bắt đầu bằng 0. Vì vậy, khi đọc tài liệu nên lưu ý xem đó là Python 2 hay Python 3. Hai phiên bản này có nhiều điểm không tương thích.

Kiểu boolean chỉ nhận một trong hai giá trị True hoặc False và là một kiểu con của kiểu số nguyên. Ví dụ:

```
>>> b = True
```

```
>>> b
```

```
True
```

```
>>> b = False
```

```
>>> b
```

```
False
```

```
>>>
```

Lưu ý giá trị boolean phải viết chính xác là True/False. Nếu viết true/false sẽ bị lỗi:

```
>>> b = true # lưu ý giá trị phải là True hoặc False
```

```
Traceback (most recent call last):
```

```
File "<pyshell#0>", line 1, in <module>
```

```
b = true
```

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

```
>>>
```

Kiểu **chuỗi ký tự** được viết trong cặp dấu ngoặc đơn `'`, ngoặc kép `"`, trong cặp ba dấu ngoặc đơn hoặc ba dấu ngoặc kép. Ví dụ dưới đây là những cách khác nhau để viết giá trị chuỗi `"Hello world"` trong Python:

```
s1 = 'Hello world' # trong cặp ngoặc đơn
s2 = "Hello world" # trong cặp ngoặc kép
s3 = '''Hello world''' # trong cặp 3 dấu ngoặc đơn
s4 = """Hello world""" # trong cặp 3 dấu ngoặc kép.
```

Mặc định chuỗi ký tự trong Python chỉ chứa ký tự Unicode. Python không có kiểu ký tự. Ký tự được coi là một chuỗi có 1 phần tử.

Kiểu **danh sách** trong Python tương tự như kiểu mảng trong các ngôn ngữ họ C nhưng không giới hạn kiểu dữ liệu của phần tử cũng như không cố định số phần tử. Cách viết giá trị kiểu danh sách như sau:

```
companies = ['Apple', 'Google', 'Microsoft', 'IBM', 1945, 1975]
```

Tuple là kiểu dữ liệu ít gặp ở các ngôn ngữ họ C. Tuple tương tự như một danh sách cố định chỉ đọc. Bạn cũng có thể hình dung tuple giống như danh sách tham số của phương thức trong các ngôn ngữ họ C. Tuple được viết như sau:

```
tuple = ('Donald Trump', 40, 'USA')
```

Từ điển chứa các cặp khóa : giá trị tương tự như từ điển song ngữ. Giá trị của từ điển được viết như sau:

```
person = {'Name' : 'Donald', 'Age' : 40, 'Job' : 'President'}
```

Trong các nội dung tiếp theo, chúng ta sẽ tìm hiểu chi tiết về từng kiểu dữ liệu.

Kết luận

Trong phần này chúng ta đã xem xét chi tiết việc sử dụng biến trong Python. Nếu bạn đã từng học một ngôn ngữ như C# hay Java, bạn có thể thấy biến, kiểu dữ liệu và phép gán trong Python có nhiều khác biệt.

Những đặc điểm như chúng ta đã biết ở trên khiến Python là một ngôn ngữ định kiểu động (không cần xác định kiểu của biến khi khai báo) và định kiểu yếu (có thể tự do gán giá trị cho biến mà không cần quan tâm đến kiểu). Những điểm này khác biệt với C# hay Java là những ngôn ngữ định kiểu tĩnh và định kiểu mạnh.

Các kiểu dữ liệu số trong Python

Trong phần này chúng ta sẽ xem xét chi tiết các kiểu dữ liệu số trong Python, bao gồm int, float và complex. Chúng ta cũng xem xét một số phép toán và hàm thường gặp với dữ liệu kiểu số. So với các ngôn ngữ lập trình như C# hay Java, các kiểu số trong Python đơn giản hơn (ít kiểu) nhưng lại có điểm đặc thù. Các phép toán cơ bản của Python khá tương tự như trong C# và các ngôn ngữ khác.

NỘI DUNG

1. Giới thiệu chung về các kiểu số trong Python
2. Kiểu số nguyên int
3. Kiểu số thực float
4. Kiểu số phức complex
5. Các phép toán trên kiểu số
6. Kết luận

Giới thiệu chung về các kiểu số trong Python

Số là kiểu dữ liệu cơ bản nhất trong các ngôn ngữ lập trình. Ngôn ngữ Python hỗ trợ sẵn 3 kiểu số cơ bản: int (số nguyên có dấu lớn tùy ý), float (số thực), complex (số phức).

Python căn cứ vào cách viết (literal) của giá trị để tự xác định kiểu cho giá trị số tương ứng. Ví dụ, `a = 10` sẽ tạo ra một số nguyên, `b = 10.` sẽ tạo ra một số thực, `c = 10j` sẽ tạo ra một số phức. Chi tiết về cách viết các loại số sẽ trình bày trong các phần tiếp theo.

Python hỗ trợ tất cả các phép toán số học toán cơ bản (+, -, *, /,...) trên các kiểu số cũng như các phép toán phức tạp hơn (ở dạng các hàm) trong module math (như log, sin, cos,...). Ngoài ra Python cũng rất nổi tiếng về các thư viện hỗ trợ tính toán do cộng đồng cung cấp.

Một đặc điểm quan trọng của kiểu số trong Python là tính bất biến (immutability). Nghĩa là giá trị số một khi đã tạo ra trong bộ nhớ (và có biến tham chiếu tới), nó sẽ không thay đổi được nữa. Nếu bạn thay đổi giá trị của một số, trên thực tế Python sẽ tạo ra một giá trị mới trong bộ nhớ.

Ví dụ nếu ban đầu bạn tạo một giá trị số qua lệnh gán `a = 10`, Python sẽ tạo một giá trị 10 trong bộ nhớ và để a trỏ vào giá trị này. Nếu sau đó bạn lại gán `a = 10j` (số phức), Python không hề thay đổi giá trị số mà a trỏ tới. Thay vào đó Python sẽ tạo một giá trị mới 10j trong bộ nhớ và để a trỏ tới giá trị mới này. Giá trị cũ 10 sẽ không còn được tham chiếu tới nữa. Sau một thời gian Garbage Collector sẽ dọn dẹp giá trị này để thu hồi bộ nhớ.

Kiểu số nguyên int

Kiểu **int** biểu diễn các số nguyên có dấu với độ lớn tùy ý. Kiểu int trong Python không sử dụng số bit cố định để biểu diễn như trong các ngôn ngữ khác. Tùy thuộc vào giá trị cụ thể Python sẽ chọn số bit phù hợp. Giá trị nguyên lớn nhất mà Python biểu diễn được chỉ phụ thuộc vào bộ nhớ.

Python hỗ trợ biểu diễn số nguyên dương, số nguyên âm, số ở dạng thập phân, hệ cơ số 8, hệ cơ số 16.

Khi biểu diễn số ở cơ số 8 bạn dùng tiền tố 0o hoặc 0O (số không và chữ o hoa/thường). Khi biểu diễn số ở hệ 16 thì dùng tiền tố 0x hoặc 0X (số không và chữ x/X)

Dưới đây là ví dụ về cách biểu diễn số nguyên trong Python:

Giá trị	Ghi chú
<code>100</code>	Số nguyên dương
<code>-100</code>	Số nguyên âm
<code>0o100</code> , <code>0O100</code>	Số dương ở cơ số 8 (số 64 cơ số 10)
<code>-0o100</code> , <code>-0O100</code>	Số âm ở cơ số 8
<code>0x100</code> , <code>0X100</code>	Số dương ở cơ số 16 (256 cơ số 10)
<code>-0x100</code> , <code>-0X100</code>	Số âm ở cơ số 16

Chú ý khi viết số ở cơ số 8 tốt nhất là dùng chữ o (thường) do chữ O (hoa) rất dễ nhầm lẫn với số 0.

Python cũng cho phép dùng ký tự gạch chân _ để nhóm các chữ số trong biểu diễn số:

```
>>> i = 1_000_000 # tương đương với 1000000 nhưng dễ đọc hơn nhiều
>>> i
1000000
```

Kiểu boolean trong Python cũng là một kiểu con của kiểu số nguyên. Trong đó 0 tương ứng với False, mọi giá trị nguyên khác tương ứng với True.

Chú ý: Python 2.x có hai kiểu số nguyên int và long. Kiểu long cần biểu diễn với hậu tố L (hoặc l). Số nguyên ở hệ cơ số 8 viết bắt đầu bằng 0. Vì vậy, khi đọc tài liệu nên lưu ý xem đó là Python 2 hay Python 3. Hai phiên bản này có nhiều điểm không tương thích.

Để biến đổi từ các kiểu dữ liệu khác về số nguyên bạn có thể sử dụng hàm `int()`. Ví dụ:

```
>>> age = int(input("Your age: ")) # chuyển chuỗi về số nguyên
Your age: 37
>>> print('You were born in', 2020-age)
You were born in 1983
>>>
```

Kiểu số thực float

Trong Python float dùng để biểu diễn số thực dấu chấm động. Để viết số thực trong Python bạn cần đặt 1 dấu chấm thập phân. Dưới đây các cách biểu diễn giá trị thực trong Python:

Giá trị float	Ghi chú
0.0	Giá trị 0.0 (float) chưa chắc đã bằng 0 (int)
100.0	Số thực dương

Giá trị float	Ghi chú
-100.0	Số thực âm
100e2, -100e2	Cách viết khoa học, bằng $\pm 100 * 10^2 = \pm 10000.0$
100e-2, -100e-2	Cách viết khoa học, bằng $\pm 100 * 10^{-2} = \pm 1.0$
100., -100.	Không cần viết số 0 sau dấu chấm thập phân, tương đương 100.0, -100.0

Bạn có thể xem các thông tin về kiểu float như sau:

```
>>> import sys

>>> sys.float_info

sys.float_info(max=1.7976931348623157e+308,      max_exp=1024,      max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)

>>> sys.float_info.max # giá trị cực đại của kiểu float

1.7976931348623157e+308

>>> sys.float_info.min # giá trị cực tiểu của kiểu float

2.2250738585072014e-308

>>>
```

Bạn cũng có thể dùng ký tự gạch chân để nhóm các chữ số khi viết số thực:

```
>>> i = 1_000_000.001

>>> i

1000000.001
```

Để biến đổi từ kiểu dữ liệu khác về kiểu float bạn có thể sử dụng hàm `float()`.

```
>>> exchange_rate = float(input("Exchange Rate: "))

Exchange Rate:

23.3

>>> print('USD 100 will be VND', exchange_rate * 100)
```

USD 100 will be VND 2330.0

>>>

Trong phép toán mà toán hạng bao gồm cả số thực và số nguyên, kết quả của cả phép toán sẽ là kiểu số thực.

Ví dụ:

```
>>> 1 + 2.0 # kết quả sẽ thuộc kiểu số thực
```

```
3.0
```

```
>>> 2 * 3.0
```

```
6.0
```

```
>>> 3.0 ** 2
```

```
9.0
```

Kiểu số phức complex

Python là một trong số ít các ngôn ngữ hỗ trợ trực tiếp kiểu số phức. Tuy nhiên số phức tương đối ít được sử dụng.

Trong toán học, số phức được biểu diễn ở dạng tổng quát $a + bi$ với i là đơn vị ảo. Trong Python, đơn vị ảo được biểu diễn bằng ký tự j hoặc J .

Như vậy, số phức trong Python cần chứa ký tự j (hoặc J) để biểu diễn phần ảo. Dưới đây là ví dụ về cách biểu diễn số phức trong Python:

Giá trị phức	Ghi chú
3.14j	Chỉ có phần ảo ($0 + 3.14j$)
45.j	Chính là ($0 + 45j$)
3 + 2j	Số phức ($3 + 2j$)
1+3.2e25j	Phần ảo biểu diễn ở dạng khoa học
1+j	<i>Lỗi: j ở đây sẽ hiểu là một biến.</i> Bạn phải viết $1 + 1j$

Lưu ý: j (J) phải đi kèm số thì mới được xem là đơn vị ảo. Nếu đi một mình, j hay J sẽ bị xem là tên biến. Vì vậy $1 + j$ là một biểu thức (với j là một biến) nhưng $1 + 1j$ là một số phức.

Để làm việc riêng với phần thực hoặc phần ảo bạn có thể sử dụng cách sau:

```
>>> c = 1.23 + 4.56j
```

```
>>> c.real
```

```
1.23
```

```
>>> c.imag
```

```
4.56
```

```
>>>
```

`real` và `imag` là hai thuộc tính thành viên của kiểu `complex` giúp bạn trích giá trị của phần nguyên và phần ảo.

Các phép toán trên kiểu số

Python hỗ trợ tất cả các **phép toán số học** cơ bản trên kiểu số. Đa số các phép toán này có hình thức tương tự như trong C, một số có dạng hơi khác.

Phép toán	Ý nghĩa	Sử dụng	Ví dụ
+	Phép cộng số học	$a + b$	
-	Phép trừ số học	$a - b$	
*	Phép nhân số học	$a * b$	
/	Phép chia số học	a / b	
%	Phép chia lấy dư	$a \% b$	$4 \% 2 = 0$, $4 \% 3 = 1$
**	Phép lũy thừa	$a ** b$	$2 ** 3 = 8$

Phép toán	Ý nghĩa	Sử dụng	Ví dụ
//	Phép chia lấy phần nguyên	$a // b$	$9 // 2 = 4$, $11 / 3 = 3$, $-11//3 = -4$

Các **phép so sánh** có thể áp dụng trên tất cả các kiểu số. Kết quả của phép toán so sánh thuộc về kiểu boolean (True/False).

Phép toán	Ý nghĩa	Sử dụng
==	So sánh bằng	$a == b$
!=	So sánh khác	$a != b$
>	So sánh lớn hơn	$a > b$
<	So sánh nhỏ hơn	$a < b$
>=	Lớn hơn hoặc bằng	$a >= b$
<=	Nhỏ hơn hoặc bằng	$a <= b$

Lưu ý trong Python 2.x có hai phép toán so sánh khác != và <>. Trong Python 3.x chỉ còn phép so sánh != (như trong C#).

Python hỗ trợ các **phép toán gán mở rộng** tương tự như trong C#:

Phép toán	Cách dùng	Ý nghĩa
+=	$a += b$	tương đương $a = a + b$
-=	$a -= b$	$a = a - b$
*=	$a *= b$	$a = a * b$
/=	$a /= b$	$a = a / b$
%=	$a \% = b$	$a = a \% b$

Phép toán	Cách dùng	Ý nghĩa
<code>**=</code>	<code>a **= b</code>	<code>a = a ** b</code>
<code>//=</code>	<code>a //= b</code>	<code>a = a // b</code>

Python cung cấp một số **hàm xử lý số** tích hợp sẵn cho kiểu số:

<code>abs(x)</code>	Lấy giá trị tuyệt đối của x
<code>int(x)</code>	Chuyển đổi x sang kiểu số nguyên
<code>float(x)</code>	Chuyển đổi x sang kiểu số thực
<code>complex(re, im)</code>	Tạo số phức với phần thực <i>re</i> , phần ảo <i>im</i> .
<code>divmod(x, y)</code>	Thực hiện cặp phép toán <code>(x // y, x % y)</code> , ví dụ <code>divmod(10, 2)</code> cho kết quả (5, 0)
<code>pow(x, y)</code>	Tương tự <code>x ** y</code>
<code>hex(x)</code>	Trả về dạng biểu diễn cơ số 16 của số x
<code>oct(x)</code>	Trả về dạng biểu diễn cơ số 8 của số x

Nếu bạn cần thực hiện các phép toán phức tạp hơn (như lượng giác, logarit), bạn có thể sử dụng module **math** như sau:

```
import math # sử dụng thư viện math
math.sin(180)
math.log(1)
math.factorial(10)
```

Kết luận

Trong phần này chúng ta đã xem xét chi tiết các kiểu dữ liệu số trong Python, bao gồm int, float và complex. Chúng ta cũng xem xét một số phép toán thường gặp với dữ liệu kiểu số.

So với các ngôn ngữ lập trình như C# hay Java, các kiểu số trong Python đơn giản hơn (ít kiểu). Tuy nhiên, các kiểu số trong Python lại có điểm đặc thù là bất biến, tương tự như string của C#.

Các phép toán cơ bản của Python khá tương tự như trong C# và các ngôn ngữ khác. Python cũng cung cấp thư viện cho các hàm toán học math.

Kiểu chuỗi ký tự (string) trong Python

Chuỗi ký tự (string) là kiểu dữ liệu phổ biến hàng đầu trong Python. Hầu như bất kỳ chương trình nào cũng đều cần dùng đến kiểu string. Bài học này sẽ giới thiệu chi tiết về kiểu chuỗi ký tự (string) trong Python. String trong Python có nhiều điểm tương đồng với chuỗi trong các ngôn ngữ họ C.

NỘI DUNG

1. String trong Python
2. Escape character trong Python string
3. Các phép toán trên kiểu string trong Python
4. Chuỗi định dạng (formatted string) trong Python
5. Các phương thức xử lý chuỗi trong Python
6. Kết luận

String trong Python

Kiểu string trong Python là *chuỗi các ký tự Unicode*. Python cho phép viết giá trị chuỗi theo nhiều cách khác nhau:

Giá trị string	Ghi chú
<code>'Hello world'</code>	Sử dụng cặp dấu nháy đơn
<code>"Hello world"</code>	Sử dụng cặp dấu nháy kép
<code>'''Hello world'''</code>	Sử dụng cặp ''' (3 dấu nháy đơn)
<code>"""Hello world"""</code>	Sử dụng cặp """" (3 dấu nháy kép)
<code>'I want to say "I love you"'</code>	Có thể dùng dấu nháy kép trong chuỗi nằm trong cặp dấu nháy đơn
<code>"My name's Donald"</code>	Có thể dùng dấu nháy đơn trong chuỗi nằm trong cặp dấu nháy kép

Giá trị string	Ghi chú
<code>'My name\'s Donald'</code>	Dùng <code>\'</code> để biểu diễn dấu <code>'</code> trong chuỗi nằm giữa cặp nháy đơn
<code>"I want to say \"I love you\""</code>	Dùng <code>\"</code> để biểu diễn dấu <code>"</code> trong chuỗi nằm giữa cặp nháy kép

Python không có kiểu ký tự (character) như các ngôn ngữ khác. Kiểu ký tự trong Python có thể xem như một chuỗi chỉ chứa 1 ký tự.

Chuỗi ký tự đặt trong cặp dấu nháy đơn có thể chứa dấu nháy kép. Tương tự, nếu chuỗi ký tự đặt trong cặp dấu nháy kép thì có thể chứa cả dấu nháy đơn. Chuỗi ký tự tạo ra bởi cặp dấu nháy đơn hoặc nháy kép bắt buộc phải nằm trên một dòng.

`\'` và `\"` được gọi là các **escape character** biểu diễn cho ký tự `'` và `"`.

Cặp 3 dấu nháy đơn (hoặc cặp 3 dấu nháy kép) cho phép tạo ra chuỗi với nhiều dòng. Ví dụ:

```
text = """Strings are amongst the most popular types in Python.
We can create them simply by enclosing characters in quotes.
Python treats single quotes the same as double quotes.
Creating strings is as simple as assigning a value to a variable."""
print(text)

line = "Hello world"
print(line)
```

Tương tự như các kiểu số trong Python, string cũng là kiểu dữ liệu bất biến (**immutable**). Nghĩa là mọi thao tác cập nhật chuỗi đều dẫn đến tạo chuỗi mới. Điều này cũng tương tự như kiểu string trong C#.

Escape character trong Python string

Nếu bạn đã học C# hẳn đã gặp các ký tự như `\r`, `\n`, `\t`. Các ký tự này được gọi là escape character. Chuỗi trong Python cũng sử dụng các escape character tương tự như thế.

Escape character là một số ký tự có ý nghĩa đặc biệt nếu xuất hiện trong chuỗi. Mỗi escape character là một ký tự. Mỗi ký tự này có thể được biểu diễn ở dạng dấu \ quen thuộc (backslash notation) hoặc ở dạng mã hex.

Dưới đây là một số escape character thường gặp:

Escape character	Mã	Tên gọi
\a	0x07	Bell/Alert
\b	0x08	Backspace
\e	0x1b	Escape
\f	0x0c	Formfeed
\n	0x0a	Newline
\r	0x0d	Carriage return
\s	0x20	Space
\t	0x09	Tab
\v	0x0b	Vertical tab

Đây chỉ là các ký tự thông dụng. Ngoài ra còn một số ký tự khác ít gặp hơn.

Trong các ký tự trên, \r, \n, \t là thường gặp nhất khi in dữ liệu ra console:

- \r – đưa con trỏ văn bản console về đầu dòng
- \t – chèn dấu tab
- \n – bắt đầu một dòng mới.

```
>>> print("Hello\tworld")
```

```
Hello world
```

```
>>> print("Hello\nworld")
```

```
Hello
```

```
world
```

```
>>>
```

Trong cách viết chuỗi, `\'` và `\"` cũng là các escape character biểu diễn cho ký tự `'` và `"`.

Nếu bạn muốn Python bỏ qua tất cả các escape character, bạn có thể viết như sau:

```
>>> print(r'Hello\nworld') # đặt thêm r vào trước giá trị chuỗi
```

```
Hello\nworld
```

Bạn đặt thêm `r` (hoặc `R`) vào trước giá trị của chuỗi. Ký tự `r` (`R`) khi này sẽ biến chuỗi thành **chuỗi thô** (raw string). Trong chuỗi thô mọi ký tự đặc biệt như escape character sẽ bị bỏ qua.

Các phép toán trên kiểu string trong Python

Python cung cấp sẵn một số phép toán trên chuỗi ký tự.

Để dễ minh họa các phép toán này, chúng ta giả sử có hai chuỗi `a = 'Hello'` và `b = 'Python'`.

Phép toán	Ý nghĩa	Ví dụ
<code>+</code>	Phép ghép xâu (concatenation)	<code>a + b</code> cho 'HelloPython'
<code>*</code>	Phép lặp xâu (repetition)	<code>a*2</code> cho 'HelloHello'
<code>[i]</code>	Phép cắt (slice), lấy ký tự ở vị trí <code>i</code>	<code>a[0]</code> cho ký tự 'H'
<code>[i1:i2]</code>	Phép cắt đoạn (range slice) từ vị trí <code>i1</code> đến <code>i2</code> ; có thể bỏ qua <code>i1</code> hoặc <code>i2</code> ; <code>i1</code> và <code>i2</code> âm thì tính từ cuối chuỗi	<code>a[1:4]</code> cho 'ell', <code>a[1:]</code> cho 'ello' (lấy từ ký tự số 1 về cuối), <code>a[-1:]</code> cho 'o' (lấy từ ký tự thứ 1 từ cuối đến hết ký tự cuối cùng), <code>a[1:-1]</code> cho 'ell' (lấy từ ký tự số 1 đến ký tự gần cuối)

Phép toán	Ý nghĩa	Ví dụ
<code>in</code>	Kiểm tra thành viên	<code>'lo' in a</code> cho kết quả True ('Hello' chứa 'lo'), <code>'lol' in a</code> cho kết quả False ('Hello' không chứa 'lol'),
<code>not in</code>	Kiểm tra thành viên (phủ định)	Giống như trên nhưng kết quả ngược lại
<code>r/R</code>	Chuỗi thô	Bạn đã gặp ở phần trên
<code>%</code>	Định dạng chuỗi	Xem phần tiếp theo
<code>==</code>	So sánh xâu (bằng)	<code>a == b</code> cho kết quả False
<code>!=</code>	So sánh xâu (khác)	<code>a != b</code> cho kết quả True

Lưu ý các phép so sánh chuỗi sẽ so từng cặp ký tự từ trái qua phải, ký tự hoa khác ký tự thường. Python cũng áp dụng các phép so sánh `>`, `<`, `>=`, `<=` cho kiểu chuỗi ký tự.

Bạn có thể thử nghiệm các phép toán trên như sau:

```
>>> a, b = 'Hello', 'Python'
```

```
>>> a + b
```

```
'HelloPython'
```

```
>>> a[1:]
```

```
'ello'
```

```
>>> a[-1:]
```

```
'o'
```

```
>>> a[-1:0]
```

```
''
```

```
>>> a[-2:]
```

```
'lo'
```

```
>>> 'lo' in a
```

True

```
>>> 'lol' in a
```

False

```
>>> a * 2
```

```
'HelloHello'
```

```
>>> a[0]
```

```
'H'
```

```
>>>
```

Chuỗi định dạng (formatted string) trong Python

Hãy xem ví dụ sau:

```
>>> name, age = 'Donald', 40
```

```
>>> greeting = 'Welcome, %s, %i years old' % (name, age)
```

```
>>> greeting
```

```
'Welcome, Donald, 40 years old'
```

```
>>> print('Welcome to heaven, %s, %i years old' % (name, age))
```

```
Welcome to heaven, Donald, 40 years old
```

```
>>>
```

Ví dụ này minh họa cách tạo một xâu có định dạng từ một khuôn mẫu và các biến.

Lưu ý các biến phải đặt trong cặp dấu () và phân tách nhau bởi dấu phẩy. Danh sách biến (thực chất là một biến kiểu tuple) phân tách với chuỗi bằng ký tự `%` – phép toán định dạng chuỗi của Python. Thứ tự biến trong tuple phải giống với thứ tự nó xuất hiện trong chuỗi. Trong ví dụ trên, nếu thay đổi thứ tự name và age sẽ gây lỗi.

Dưới đây là các ký tự định dạng trong Python:

Ký tự định dạng	Ý nghĩa
<code>%C</code>	ký tự
<code>%S</code>	chuỗi

<code>%i</code>	số nguyên (có dấu)
<code>%d</code>	số nguyên (có dấu)
<code>%u</code>	số nguyên (không dấu)
<code>%o</code>	số nguyên (cơ số 8)
<code>%x, %X</code>	số nguyên (cơ số 16)
<code>%e, %E</code>	biểu diễn khoa học
<code>%f</code>	số thực

Đây là cách thức tạo chuỗi có định dạng từ Python 2.x. Trong Python 3 bạn vẫn có thể dùng được lỗi viết này.

Nếu không muốn dùng phép toán định dạng `%`, bạn có thể sử dụng hàm `format` như sau:

```
>>> print('Welcome to heaven, {1}, {0} years old'.format(age, name))
```

Welcome to heaven, Donald, 40 years old

Khi sử dụng hàm `format` bạn tạo ra các placeholder trong chuỗi với số thứ tự (tính từ 0). Biến với số thứ tự tương ứng từ phương thức `format` sẽ được đặt vào thay cho placeholder để tạo thành chuỗi hoàn chỉnh.

Lỗi viết này hoàn toàn giống chuỗi định dạng trong C#.

Cách viết này tiện lợi hơn so với sử dụng phép toán định dạng `%`. Bạn không cần nhớ các ký tự định dạng nữa.

Từ Python 3.6 bạn có thể sử dụng một lỗi viết chuỗi định dạng khác:

```
>>> name, age = 'Donald', 40
```

```
>>> print(f'Hello, {name}. You are {age}. Welcome to heaven!')
```

Hello, Donald. You are 40. Welcome to heaven!

Đây là lỗi viết chuỗi định dạng tiện lợi nhất trong Python.

Trong lỗi viết này bạn đặt ký tự `f` vào đầu chuỗi. Bên trong chuỗi bạn có thể sử dụng biến đặt trong cặp `{}`. Loại chuỗi này được gọi là f-string.

Bên trong cặp `{}` bạn có thể sử dụng bất kỳ giá trị hoặc biểu thức nào của Python.

Cách viết này rất giống với string interpolation trong C#.

Các phương thức xử lý chuỗi trong Python

Chuỗi trong Python là một object với nhiều phương thức được xây dựng sẵn. Dưới đây là minh họa một số phương thức thường gặp.

```
>>> greeting = 'Hello, Donald. Welcome to heaven'

>>> greeting
'Hello, Donald. Welcome to heaven'

>>> # các hàm sau đây phải sử dụng từ object của chuỗi

>>> greeting.capitalize() # viết hoa chữ cái đầu tiên, tất cả chữ cái còn lại viết thường
'Hello, donald. welcome to heaven'

>>> greeting.lower() # chuyển về chuỗi viết thường
'hello, donald. welcome to heaven'

>>> greeting.upper() # chuyển về chuỗi viết hoa
'HELLO, DONALD. WELCOME TO HEAVEN'

>>> greeting.title() # viết hoa chữ cái đầu mỗi từ
'Hello, Donald. Welcome To Heaven'

>>> greeting.split() # cắt một chuỗi ra các từ
['Hello,', 'Donald.', 'Welcome', 'to', 'heaven']

>>> greeting = ' Hello, Donald. Welcome to heaven '

>>> greeting.strip() # cắt các khoảng trống ở đầu và cuối chuỗi
'Hello, Donald. Welcome to heaven'

>>> greeting.rstrip() # cắt các khoảng trống ở cuối chuỗi (lề phải)
' Hello, Donald. Welcome to heaven'

>>> greeting.lstrip() # cắt các khoảng trống ở đầu chuỗi (lề trái)
'Hello, Donald. Welcome to heaven '

>>> # một số hàm kiểm tra

>>> my_string = "Hello World"
```



```
>>> my_string.isalnum() #kiểm tra xem chuỗi có chứa toàn chữ số
```

False

```
>>> my_string.isalpha() # chuỗi chứa toàn chữ cái
```

False

```
>>> my_string.isdigit() # chuỗi có chứa chữ số
```

False

```
>>> my_string.isupper() # chuỗi chứa toàn ký tự hoa
```

False

```
>>> my_string.islower() # chuỗi chứa toàn ký tự thường
```

False

```
>>> my_string.isspace() # chuỗi chỉ chứa khoảng trắng
```

False

```
>>> my_string.endswith('d') # kết thúc là d
```

True

```
>>> my_string.startswith('H') # bắt đầu là H
```

True

```
>>># hàm sau đây là hàm toàn cục
```

```
>>> len(greeting) # lấy độ dài chuỗi
```

32

```
>>>
```

Lưu ý: do string trong Python là một class với nhiều phương thức hỗ trợ, bạn cần gọi các phương thức này từ biến (ví dụ `greeting.upper()`). Đây là điểm khác biệt so với các hàm toàn cục như `print` hay `input` (không gọi từ biến).

Kết luận

Phần nội dung này đã giới thiệu chi tiết về kiểu chuỗi ký tự (string) trong Python. Điểm lưu ý là trong Python có nhiều cách khác nhau để viết giá trị kiểu string (so với các ngôn ngữ như C# hay Java). Tuy nhiên, string trong Python có nhiều điểm tương đồng với chuỗi trong các ngôn ngữ họ C như escape sequence và cách viết định dạng (kiểu cũ). Python 3 có cách định dạng rất giống với C#.

Kiểu dữ liệu bool trong Python

Trong phần này chúng ta sẽ xem xét chi tiết cách sử dụng kiểu dữ liệu bool trong Python. Đây là một kiểu dữ liệu đơn giản nhưng rất quan trọng để chuyển sang tìm hiểu về các cấu trúc điều khiển (if-elif-else, while, for) trong Python.

NỘI DUNG

1. Kiểu bool trong Python
2. Chuyển đổi kiểu dữ liệu về bool
3. Các phép toán trên kiểu bool
4. Kết luận

Kiểu bool trong Python

Kiểu bool (Boolean) trong Python là kiểu dữ liệu trong đó chỉ có hai giá trị `True` và `False`. `True` và `False` là hai từ khóa trong Python.

```
>>> a = True
>>> type(a) # hàm cho biết kiểu của biến
<class 'bool'>
>>> b = False
>>> type(b)
<class 'bool'>
```

Nếu bạn đã học một ngôn ngữ trong họ C cần lưu ý giá trị là **True/False** (T và F phải viết hoa). Viết true/false là sai và sẽ bị báo lỗi như sau:

```
>>> a = true # Python sẽ hiểu true là một biến (vốn chưa tồn tại)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'true' is not defined
```

Kiểu bool là kiểu kết quả trả về của các phép so sánh trên số và chuỗi mà bạn đã tìm hiểu trong các bài trước:

```
>>> a = 10; b = 5
>>> a > b
```

True

```
>>> a <= b
```

False

```
>>> a != b
```

True

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

False

```
>>> 0 < b < a < 11 # Python cho phép viết như thế này
```

True

```
>>> a = 'Hello'; b = 'Python'
```

```
>>> a != b
```

True

```
>>> a > b
```

False

```
>>> a < b
```

True

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

False

```
>>>
```

Một số phương thức của str cũng trả về giá trị bool:

```
>>> my_string = "Hello World"
```

```
>>> my_string.isalnum() # kiểm tra xem chuỗi có chứa toàn chữ số
```

False

```
>>> my_string.isalpha() # chuỗi chứa toàn chữ cái
```

False

```
>>> my_string.isdigit() # chuỗi có chứa chữ số
```

False

```
>>> my_string.isupper() # chuỗi chứa toàn ký tự hoa
```

False

```
>>> my_string.islower() # chuỗi chứa toàn ký tự thường
```

False

```
>>> my_string.isspace() # chuỗi chỉ chứa khoảng trắng
```

False

```
>>> my_string.endswith('d') # kết thúc là d
```

True

```
>>> my_string.startswith('H') # bắt đầu là H
```

True

Chuyển đổi kiểu dữ liệu về bool

Python cho phép chuyển đổi giữa các kiểu dữ liệu khác và bool qua hàm `bool()` theo quy tắc sau:

- Giá trị kiểu số (số nguyên, số thực, số phức) thành giá trị `True` nếu số đó khác 0, và `False` nếu số đó bằng 0.
- Giá trị kiểu chuỗi thành giá trị `False` nếu đó là chuỗi rỗng (không có ký tự nào ''), và `True` nếu chuỗi có dù chỉ 1 ký tự.

```
>>> zero_int = 0
```

```
>>> bool(zero_int)
```

False

```
>>> pos_int = 1
```

```
>>> bool(pos_int)
```

True

```
>>> neg_float = -5.1
```

```
>>> bool(neg_float)
```

True

```
>>> ans = 'true'
```

```
>>> bool(ans)
```

True

```
>>> ans = 'hello'
```

```
>>> bool(ans)
```

True

```
>>> ans = "" # chuỗi rỗng (không có ký tự nào)
```

```
>>> bool(ans)
```

False

```
>>> ans = ' ' # chuỗi này chứa 1 dấu cách
```

```
>>> bool(ans)
```

True

Python cũng có thể chuyển đổi giá trị của các kiểu khác về bool. Ví dụ danh sách rỗng -> False, danh sách có phần tử -> True. Chúng ta sẽ tìm hiểu về các kiểu dữ liệu này sau.

Các phép toán trên kiểu bool

Các phép toán trên kiểu bool, còn được gọi là số học Boolean, là các phép toán logic chỉ trả về kết quả True hoặc False. Các phép toán thông dụng nhất bao gồm `and`, `or`, `not`, `==` và `!=`.

```
>>> A = True
```

```
>>> B = False
```

```
>>> A or B
```

True

```
>>> A and B
```

False

```
>>> not A
```

False

```
>>> not B
```

True

```
>>> A == B
```

False

```
>>> A != B
```

True

A	B	not A	not B	A == B	A != B	A or B	A and B
T	F	F	T	F	T	T	F
F	T	T	F	F	T	T	F
T	T	F	F	T	F	T	T
F	F	T	T	T	F	F	F

Bảng chân trị của các phép toán logic Boolean

Bạn có thể ghép các phép toán trên để tạo thành biểu thức phức tạp hơn như sau:

```
>>> A = True
```

```
>>> B = False
```

```
>>> C = False
```

```
>>> A or (C and B)
```

```
True
```

```
>>> (A and B) or C
```

```
False
```

Kết luận

Trong phần này chúng ta đã xem xét chi tiết cách sử dụng kiểu dữ liệu bool trong Python. Đây là một kiểu dữ liệu đơn giản nhưng rất quan trọng để chuyển sang tìm hiểu về các cấu trúc điều khiển (if-elif-else, while, for) trong Python.

Cấu trúc điều kiện if trong Python

Trong một chương trình, bình thường các lệnh sẽ lần lượt được thực hiện theo thứ tự xuất hiện của nó trong file code. Nếu chỉ thực thi lệnh như vậy các chương trình sẽ rất hạn chế. Vì vậy người ta đưa vào các cấu trúc điều khiển có tác dụng làm thay đổi trật tự thực thi lệnh trong chương trình.

Ví dụ, bạn chỉ thực thi lệnh khi đáp ứng một điều kiện nào đó. Bạn cũng có thể muốn lặp đi lặp lại việc thực hiện một nhóm lệnh. Trường hợp thứ nhất người ta gọi là rẽ nhánh, trường hợp thứ hai gọi là vòng lặp.

Python có các cấu trúc điều khiển rẽ nhánh và các cấu trúc lặp tương tự như các ngôn ngữ khác. Trong phần này chúng ta sẽ xem xét cách sử dụng cấu trúc lặp if-elif-else.

NỘI DUNG

1. Chương trình minh họa
2. Cấu trúc rẽ nhánh if
3. Mệnh đề elif và else
4. Các cấu trúc if-elif-else lồng nhau
5. Từ khóa pass
6. Kết luận

Chương trình minh họa

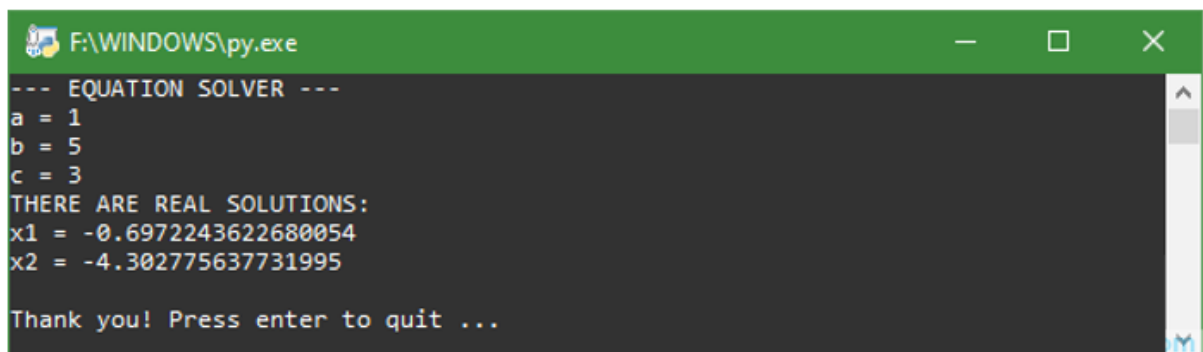
Tạo file equation.py và viết code như sau:

equation.py

```
from math import sqrt # sử dụng hàm tính căn sqrt trong module math
print('--- EQUATION SOLVER ---')
a = float(input('a = '))
b = float(input('b = '))
c = float(input('c = '))
d = b*b - 4*a*c
if d >= 0:
    print('THERE ARE REAL SOLUTIONS:')
```

```
x1 = (-b + sqrt(d))/(2*a)
x2 = (-b - sqrt(d))/(2*a)
print(f'x1 = {x1}')
print(f'x2 = {x2}')
else:
    print('THERE ARE COMPLEX SOLUTIONS BUT I CANNOT SHOW YOU.')
input('\nThank you! Press enter to quit ...')
```

Chạy script trên ở dạng chương trình console (click đúp vào file equation.py hoặc click phải -> Open with -> Python) bạn sẽ được kết quả như sau:



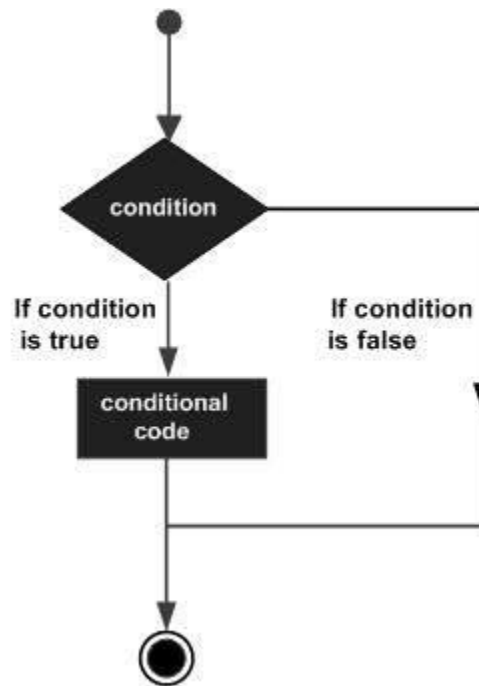
```
--- EQUATION SOLVER ---
a = 1
b = 5
c = 3
THERE ARE REAL SOLUTIONS:
x1 = -0.6972243622680054
x2 = -4.302775637731995
Thank you! Press enter to quit ...
```

Đây là một ví dụ đơn giản minh họa cách sử dụng cấu trúc rẽ nhánh if-else trong Python:

```
if d >= 0:
    print('THERE ARE REAL SOLUTIONS:')
    # code khác
else:
    print('THERE ARE COMPLEX SOLUTIONS BUT I CANNOT SHOW YOU.')
```

Cấu trúc rẽ nhánh if

Cấu trúc rẽ nhánh quyết định xem những lệnh nào sẽ được thực hiện căn cứ vào giá trị (kiểu bool) của một biểu thức điều kiện.



Như trong ví dụ trên, nếu $d \geq 0$ (biểu thức logic điều kiện) thì chúng ta tính nghiệm thực của phương trình và in ra kết quả. Nếu $d < 0$ thì chỉ thông báo là có nghiệm phức nhưng không tính toán được (vì hàm `sqrt` của Python không chấp nhận đối số âm).

Cấu trúc này làm thay đổi luồng thực thi (trật tự thực hiện lệnh) của chương trình. Trật tự thực hiện lệnh phụ thuộc vào giá trị của biểu thức điều kiện. Có những lệnh sẽ không được thực hiện.

Cú pháp cơ bản của cấu trúc này như sau:

```
if <biểu thức điều kiện> :  
    ...  
    # khối code  
    ...
```

Một số ví dụ:

```
age = int(input('Your age: '))  
if(age >= 18):  
    print('Welcome!')  
    print(f'Your birth year is {2020-age}')  
name = input('Your name: ')  
if(name.lower() == 'donald'):
```

```
print('Mr. President!')

print('Welcome to heaven!')
```

Biểu thức điều kiện là một biểu thức có kiểu kết quả là bool. Phần `if` <biểu thức điều kiện>: được gọi là **header** (tiêu đề), phần khối code được gọi là **suite** (thân). Tổ hợp header và suite được gọi là một **clause** (mệnh đề).

```
age = int(input('Your age: '))
if(age >= 18):
    print('Welcome!')
    print(f'Your birth year is {2020-age}')
name = input('Your name: ')
if(name.lower() == 'donald'):
    print('Mr. President!')
    print('Welcome to heaven!')
```

Phần suite là bắt buộc. Tất cả lệnh trong suite phải viết với cùng số thụt đầu dòng. Số lượng thụt đầu dòng không bắt buộc nhưng thường quy ước là 1 thụt đầu dòng = 4 space. Các IDE đều hỗ trợ viết thụt đầu dòng tự động. Bạn cũng có thể tự thụt đầu dòng bằng phím tab hoặc space.

Nếu vô tình làm thay đổi thụt đầu dòng bạn sẽ gặp lỗi cú pháp:

```
age = int(input('Your age: '))
if(age >= 18):
    print('Welcome!')
    print(f'Your birth year is {2020-age}')
else:
    print('young to enter!')
print('Goodbye!')
```

Mệnh đề elif và else

Trong cấu trúc rẽ nhánh đơn giản nhất bạn chỉ cần một mệnh đề if là đủ.

Hãy giả sử bạn viết script kiểm tra tuổi đi học theo các mức sau: (1) dưới 6 tuổi -> trẻ mầm non, (2) từ 6 đến 12 -> học sinh tiểu học, (3) từ 12 đến 15 -> học sinh trung học cơ sở, (4) từ 16 đến 18 -> học sinh trung học, (5) trên 18 tuổi -> đại học / đi làm.

Dĩ nhiên bạn có thể viết 5 lệnh if:

```
age = int(input('Your age: '))
if (0 < age < 6):
    print('Mầm non')
if (6 <= age < 12):
    print('Tiểu học')
if (12 <= age < 15):
    print('Trung học cơ sở')
if (15 <= age < 18):
    print('Trung học phổ thông')
if (18 <= age):
    print('Đại học / đi làm')
```

Cách sử dụng này không sai nhưng có vấn đề. Ví dụ, nếu bạn nhập giá trị 14, Python sẽ kiểm tra tất cả các cấu trúc if. Mặc dù lệnh ở cấu trúc if (12 <= age < 15) được thực thi, Python vẫn tiếp tục thực hiện việc kiểm tra cả 2 cấu trúc if còn lại. Điều này dẫn đến làm thừa việc.

Giờ hãy điều chỉnh lại như sau:

```
age = int(input('Your age: '))
if (0 < age < 6):
    print('Mầm non')
elif (6 <= age < 12):
    print('Tiểu học')
elif (12 <= age < 15):
    print('Trung học cơ sở')
elif (15 <= age < 18):
    print('Trung học phổ thông')
elif (18 <= age <= 100):
    print('Đại học / đi làm')
else:
    print('Bạn còn sống không đấy?')
```

Trong đoạn script trên chúng ta đã vận dụng hai mệnh đề khác của cấu trúc if: mệnh đề `elif` và `else`.

Giả sử bạn nhập giá trị 14. Python sẽ làm như sau:

1. Kiểm tra mệnh đề if -> bỏ qua, vì age nằm ngoài khoảng (0, 6);
2. Kiểm tra mệnh đề elif (6 <= age < 12) -> bỏ qua, vì age nằm ngoài khoảng [6, 12)
3. Kiểm tra mệnh đề elif (12 <= age < 15) -> thực hiện, vì age nằm trong khoảng [12, 15)
4. Bỏ qua hết các mệnh đề còn lại.

Đây là lợi thế của các mệnh đề elif: nếu một mệnh đề phù hợp và được thực hiện, các mệnh đề còn lại bị bỏ qua.

Giả sử bạn nhập giá trị 101 hoặc -1. Theo logic trên, Python sẽ kiểm tra mệnh đề if và tất cả các mệnh đề elif. Tuy nhiên nó không tìm được mệnh đề phù hợp. Khi này Python sẽ thực hiện mệnh đề else.

Mệnh đề else cần viết cuối cùng trong danh sách. Nó là mệnh đề sẽ thực hiện nếu tất cả các mệnh đề khác không phù hợp. Trong ví dụ của chúng ta, trường hợp giá trị tuổi nhỏ hơn 0 hoặc lớn hơn 100 là sẽ được thực hiện trong mệnh đề else.

Các cấu trúc if-elif-else lồng nhau

Hãy xem một ví dụ:

```
age = int(input("Your age: "))
gender = input("Gender (male/female): ")
name = input("Your name: ")
if (age >= 18):
    print('Your age a legal.')
    if (name.isalpha()):
        if (gender.lower() == "male"):
            print(f'Welcome, Mr. {name}!')
        elif (gender.lower() == "female"):
            print(f'Welcome, lady {name}!')
```

```
    else:
        print(f'Welcome, {name}')
    else:
        print('Sorry, who are you?')
else:
    print('You are too young to come here!')
```

Trong ví dụ này bạn yêu cầu người dùng nhập tuổi, họ và giới tính.

Đầu tiên bạn kiểm tra tuổi. Nếu người dùng trên 18 tuổi, bạn tiếp tục kiểm tra tên. Nếu người dùng nhập tên (`name.isalpha()`), bạn tiếp tục kiểm tra giới tính. Nếu là nam (`male`) sẽ in ra lời chào 'Welcome, Mr.', nếu là nữ sẽ in lời chào 'Welcome, lady', nếu không chỉ định giới tính thì chỉ in ra lời chào 'Welcome'.

Nếu người dùng không cung cấp tên thì hỏi lại 'Sorry, who are you?'. Nếu tuổi dưới 18 thì in thông báo 'You are too young to come here!'.

Đây là ví dụ về cách đặt các lệnh `if-elif-else` lồng nhau.

Khi sử dụng các lệnh lồng nhau đặc biệt lưu ý đến thụt đầu dòng: các lệnh nằm trong cùng một suite phải có cùng thụt đầu dòng như nhau.

Từ khóa `pass`

Đây là từ khóa tương đối lạ với các bạn học C#. Từ khóa này được đưa ra do đặc thù của Python khi viết code block.

Hãy cùng thực hiện một ví dụ đơn giản sau:

```
age = int(input('Your age: '))
if(age >= 18):
    print('Welcome!')
else:
    print('Goodbye!')
```

Bạn sẽ gặp ngay lỗi cú pháp (báo ở dòng lệnh `print('Goodbye!')`):
`expected an indented block`.

Đây là điều tương đối lạ với các bạn đã biết C# (và các ngôn ngữ tương tự).

Trong Python, các lệnh như `if`, `while`, `for`, khai báo hàm,... được gọi là các lệnh phức hợp (**compound statement**).

Mỗi lệnh phức hợp chứa một hoặc nhiều mệnh đề (**clause**). Như cấu trúc `if` có thể có nhiều mệnh đề tương ứng với `if`, các `elif`, và `else`.

Mỗi clause tạo ra từ một **header** và một **suite**. Ví dụ, clause tương ứng với `else` có header là `else:` và suite là khối code nằm sau `else:`. Suite là khối code bắt buộc phải có trong clause. Bạn không thể viết header mà không có suite đi kèm.

Vậy nếu như bạn không muốn xử lý gì trong clause thì sao? Giả sử, trong trường hợp mệnh đề `else` bạn không muốn xử lý gì cả nhưng vẫn muốn viết nó, hoặc trong trường hợp tạm thời bạn chưa viết được code xử lý cho nó.

Từ khóa `pass` được sử dụng trong những tình huống như thế này:

```
age = int(input('Your age: '))
if(age >= 18):
    print('Welcome!')
else:
    pass
print('Goodbye!')
```

Khi này bạn vẫn có thể giữ được mệnh đề `else` nhưng nó sẽ không làm gì hết. Từ khóa `pass` được tạo ra chỉ đơn thuần là để đáp ứng yêu cầu cú pháp của Python: mỗi mệnh đề phải có đủ header và suite. Từ khóa `pass` đóng vai trò là một suite hình thức (không làm gì hết).

Đối với cấu trúc `if`, từ khóa `pass` có vẻ hơi vô ích. Tuy nhiên, trong cấu trúc xử lý ngoại lệ bạn sẽ thấy cần đến nó hơn.

Từ khóa `pass` không phải là đặc thù của cấu trúc `if-elif-else`. Đây là từ khóa xuất phát từ cấu trúc chung của các lệnh phức hợp trong Python.

Kết luận

Phần này đã hướng dẫn chi tiết cách sử dụng cấu trúc rẽ nhánh `if-elif-else` trong Python. Nhìn chung ý tưởng và cú pháp của cấu trúc này khá gần với các ngôn ngữ lập trình khác.

Tuy nhiên cần lưu ý về căn lề (thụt đầu dòng) của code trong suite của mỗi clause. Các lệnh trong cùng một suite mà căn lề lệch sẽ bị lỗi cú pháp.

Ngoài ra Python cũng sử dụng từ khóa `pass` để tạo ra một dummy suite (suite không làm gì cả) để phù hợp với cú pháp của ngôn ngữ.

Các cấu trúc lặp trong Python: for và while

Các cấu trúc lặp trong Python bao gồm hai loại: lặp với số bước xác định trước (for) và lặp với số bước không xác định (while). Nhìn chung hai cấu trúc lặp trong Python tương đối gần gũi với các cấu trúc lặp trong C hay C#.

NỘI DUNG

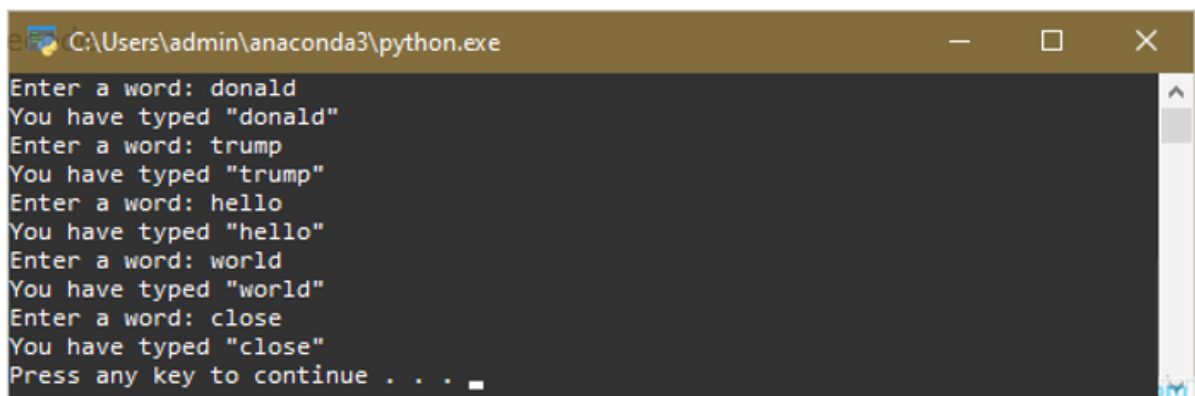
1. Cấu trúc lặp while
2. Cấu trúc lặp for
3. Sử dụng vòng lặp for với các kiểu danh sách khác
4. Điều khiển vòng lặp, break và continue
5. Kết luận

Cấu trúc lặp while

Hãy xem ví dụ sau:

```
word = ''
while (word.lower() != 'close'):
    word = input('Enter a word: ')
    print(f'You have typed "{word}"')
```

Kết quả sau khi thực thi đoạn code trên:

A screenshot of a Windows command prompt window titled "C:\Users\admin\anaconda3\python.exe". The window shows the execution of a Python script. The prompt "Enter a word:" is displayed, and the user has entered "donald", "trump", "hello", "world", and "close" in sequence. For each input, the program outputs "You have typed "[input]"". After the final input "close", the program outputs "Press any key to continue . . .". The window has standard Windows window controls (minimize, maximize, close) in the title bar.

Trong ví dụ này chúng ta liên tục yêu cầu người dùng nhập vào một từ. Nếu người dùng nhập từ 'close' thì sẽ kết thúc vòng lặp (và kết thúc chương trình). Nếu người dùng nhập từ khác thì sẽ yêu cầu nhập lại.

Chúng ta không biết khi nào người dùng sẽ nhập 'close', và cũng có thể là không bao giờ. Vì vậy, vòng lặp while được gọi là vòng lặp với số bước không xác định trước.

Biến word đóng vai trò là biến điều khiển cho biểu thức điều kiện.

Hãy xem một ví dụ khác:

```
# tính tổng các số từ 1 đến 99

sum = 0
i = 1
while i < 100:
    sum += i
    i += 1

print(sum)
```

Trong ví dụ này chúng ta tính tổng các số từ 1 đến 99. Để thực hiện việc này chúng ta sử dụng một biến `i` với vai trò biến đếm. Biến `i` ban đầu nhận giá trị 1. Qua mỗi vòng lặp `i` sẽ tăng thêm 1 đơn vị, và biến `sum` sẽ cộng thêm giá trị bằng `i`. Quá trình này sẽ lặp đi lặp lại chừng nào `i` vẫn nhỏ hơn 100. Đến khi `i` đạt giá trị 100 thì vòng lặp kết thúc.

Cấu trúc while của Python là một lệnh phức hợp với một mệnh đề. Trong đó, header là `while <biểu thức logic>:`. Biểu thức logic là loại biểu thức mà kết quả trả về thuộc kiểu bool (giá trị True hoặc False). Trong các ví dụ trên, biểu thức logic là `i < 100` và `char.lower() != 'close'`.

Chừng nào biểu thức logic còn nhận giá trị True thì sẽ thực hiện danh sách lệnh ở phần suite.

Để tránh tạo thành vòng lặp vô tận, bên trong danh sách lệnh của vòng lặp while phải có lệnh có tác dụng làm thay đổi giá trị của biểu thức logic.

Trong ví dụ thứ nhất, chúng ta liên tục yêu cầu người dùng nhập vào giá trị mới cho biến `char`. Điều này đảm bảo khả năng thoát khỏi vòng lặp.

Trong ví dụ thứ hai, chúng ta làm biến đổi giá trị của `i` qua mỗi vòng lặp (tăng thêm 1 đơn vị). Điều này đảm bảo đến một lúc nào đó `i` sẽ có giá trị lớn hơn 100 và kết thúc vòng lặp.

Cấu trúc lặp for

Hãy xem các ví dụ sau

```
# in ra các số trong phạm vi từ 1 đến 100
for n in range(100):
    print(n + 1, end = '\t')

# in ra các số trong phạm vi từ 100 đến 199
for n in range(100, 200):
    print(n, end = '\t')

# in ra các số lẻ trong phạm vi từ 1 đến 99
for n in range(1, 100, 2):
    print(n, end = '\t')
```

Cấu trúc for cho phép lặp lại việc thực hiện khối code theo một số lần xác định sẵn từ đầu.

Cấu trúc for cũng là một lệnh phức hợp với một clause (khác với cấu trúc rẽ nhánh if có nhiều clause) trong đó header có dạng `for <biến chạy> in range()`. Suite trong các ví dụ trên chỉ chứa một hàm `print()`.

Số lần lặp và cách thức lặp được tạo ra bởi hàm `range()`.

Hàm `range()` trả về một biến chứa một dãy các giá trị nằm trong một khoảng xác định theo quy luật. Hàm này có 3 dạng:

1. `range(stop)`: nhận 1 tham số là giá trị cuối của dãy và trả về dãy số `[0, stop - 1]`. Ví dụ `range(5)` sẽ trả lại dãy số 0, 1, 2, 3, 4. Bước nhảy giữa các giá trị kế tiếp là 1.
2. `range(start, stop)`: nhận 2 tham số là giá trị đầu và cuối của dãy, trả về dãy `[start, stop-1]`. Ví dụ, `range(1, 5)` sẽ trả lại dãy số 1, 2, 3, 4 (bước nhảy giữa các giá trị kế tiếp là 1).
3. `range(start, stop, step)`: tương tự như trường hợp 2 nhưng bước nhảy được xác định bởi biến `step`. Ví dụ, `range(1, 10, 2)` sẽ trả lại dãy 1, 3, 5, 7, 9 (bước nhảy giữa các giá trị kế tiếp là 2).

Trong hàm `range`, `start`, `stop` và `step` có thể nhận cả giá trị âm và dương:

- Nếu `step` nhận giá trị âm (và `start > stop`) sẽ tạo ra dãy giảm dần. Ví dụ `range(0, -5, -1)` sẽ tạo ra dãy 0, -1, -2, -3, -4.

- Nếu `stop < start` và `step` dương, hoặc `stop > start` và `step` âm, sẽ tạo ra chuỗi trống (không có giá trị nào). Ví dụ, `range(2, -3, 1)` và `range(2, 3, -1)` đều không tạo ra phần tử nào.

Với danh sách giá trị mà hàm `range()` tạo ra như trên, cấu trúc `for` sẽ lần lượt lấy từng giá trị gán cho biến chạy (biến `n` trong các ví dụ trên) và thực hiện khối lệnh. Như vậy, số lần lặp đã xác định sẵn từ lúc gọi hàm `range()`.

Sử dụng vòng lặp `for` với các kiểu danh sách khác

Ở trên chúng ta sử dụng `for` với các giá trị lấy từ danh sách tạo ra bởi hàm `range()`. Trong Python, nhiều loại dữ liệu tập hợp có thể sử dụng cùng vòng `for`.

Ví dụ:

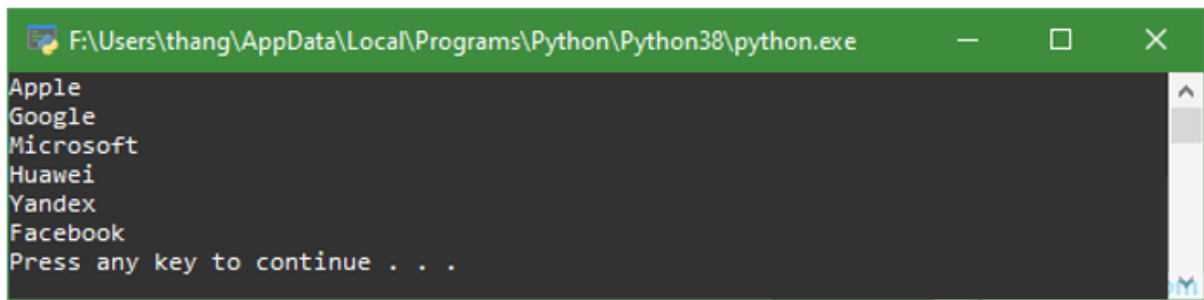
```
for c in 'Hello world':  
    print(c, end = '\t')
```

Trong ví dụ trên danh sách giá trị lại là chuỗi ký tự `'Hello world'`. Biến chạy sẽ lần lượt nhận từng ký tự từ chuỗi, bắt đầu từ ký tự đầu tiên `'H'` đến ký tự cuối cùng `'d'`. Trong ví dụ trên chúng ta chỉ đơn giản là in ký tự tương ứng ra console.

Hãy xem một ví dụ khác:

```
companies = ['Apple', 'Google', 'Microsoft', 'Huawei', 'Yandex', 'Facebook']  
for c in companies:  
    print(c)
```

Trong ví dụ trên, `companies` là một biến kiểu danh sách (list). Kiểu dữ liệu này chúng ta chưa học nhưng bạn có thể hình dung nó tương tự như kiểu mảng của C. Biến kiểu list cũng có thể trở thành danh sách dữ liệu cho vòng lặp `for`. Khi này biến chạy `c` sẽ lần lượt nhận các giá trị trong danh sách theo thứ tự.



```
F:\Users\thang\AppData\Local\Programs\Python\Python38\python.exe
Apple
Google
Microsoft
Huawei
Yandex
Facebook
Press any key to continue . . .
```

Một ví dụ khác:

```
# tính tổng các số dương từ một tuple
total = 0
for num in (-22.0, 3.5, 8.1, -10, 0.5):
    if num > 0:
        total = total + num
```

Trong ví dụ này bạn tính tổng các số dương từ một tuple. Tuple cũng là một kiểu dữ liệu tập hợp mà chúng ta sẽ học chi tiết sau.

Không phải kiểu dữ liệu tập hợp nào cũng có thể làm danh sách lặp cho vòng for. Những kiểu dữ liệu sử dụng được cho vòng for được gọi là các kiểu **iterable**.

Điều khiển vòng lặp, break và continue

Khi làm việc với vòng lặp có thể xuất hiện hai tình huống:

1. Bạn muốn kết thúc vòng lặp sớm hơn dự định hoặc theo điều kiện. Ví dụ, bạn có một danh sách khách hàng và muốn tìm tên một người trong đó. Bạn xem lần lượt từng tên từ đầu danh sách. Nếu gặp tên khách hàng cần tìm, bạn sẽ dừng lại chứ không tiếp tục xem đến hết danh sách nữa. Ở đây bạn phá vỡ hoàn toàn vòng lặp.
2. Bạn muốn bỏ qua một chu kỳ trong vòng lặp để bắt đầu chu kỳ tiếp theo. Ví dụ, bạn dự định chơi 10 ván bài. Ở ván thứ 3 bài xấu quá, bạn quyết định bỏ cuộc để chờ chơi ván tiếp theo. Ở đây bạn không dừng hoàn toàn vòng lặp mà chỉ bỏ không làm những việc nhẽ ra phải làm trong chu kỳ đó và khởi động một chu kỳ mới.

Hai tình huống này được Python giải quyết bằng hai lệnh tương ứng: **break** (phá vỡ vòng lặp, kết thúc sớm) và **continue** (bỏ qua chu kỳ hiện tại, bắt đầu một chu kỳ lặp mới).

Ví dụ với break:

```
# breaking out of a loop early
for item in [1, 2, 3, 4, 5]:
    if item == 3:
        print(item, " ...break!")
        break
    print(item, " ...next iteration")
```

Ví dụ này cho ra kết quả như sau:

```
1 ...next iteration
2 ...next iteration
3 ...break!
```

```
# demonstrating a `continue` statement in a loop
x = 1
while x < 4:
    print("x = ", x, ">> enter loop-body <<")
    if x == 2:
        print("x = ", x, " continue...back to the top of the loop!")
        x += 1
        continue
    x += 1
    print("--reached end of loop-body--")
```

Ví dụ này cho ra kết quả như sau:

```
x = 1 >> enter loop-body <<
--reached end of loop-body--
x = 2 >> enter loop-body <<
x = 2 continue...back to the top of the loop!
x = 3 >> enter loop-body <<
--reached end of loop-body--
```

Kết luận

Trong phần này chúng ta đã xem xét cách sử dụng vòng lặp trong Python, bao gồm vòng lặp for và vòng lặp while. Chúng ta cũng học về cách điều khiển vòng lặp với break và continue.

Có thể nhận xét rằng vòng lặp trong Python không có nhiều khác biệt với vòng lặp ở các ngôn ngữ khác. Đặc biệt, từ khóa break và continue cơ bản là giống hệt như trong C#.

Kiểu danh sách (list) trong Python

Python cung cấp một số cấu trúc dữ liệu đặc biệt gọi là cấu trúc tuần tự (sequence). Đặc thù của loại cấu trúc này là chứa nhiều phần tử, và mỗi phần tử được đánh số thứ tự để dễ truy xuất. Các loại cấu trúc này cũng có chung nhiều phép toán xử lý (như đánh chỉ số, cắt lát, ghép, lặp,...).

Danh sách (list) là một trong những kiểu dữ liệu tuần tự được sử dụng phổ biến nhất của Python. Các kiểu tuần tự thường gặp khác là tuple và dict.

Trong phần này chúng ta sẽ xem xét chi tiết cách sử dụng dữ liệu kiểu list.

NỘI DUNG

1. Ví dụ về list trong Python
2. Kiểu list trong Python
3. Truy xuất phần tử của list
4. Thêm mới, cập nhật và xóa danh sách
5. Một số thao tác khác trên danh sách
6. Các phương thức của lớp list
7. Kết luận

Ví dụ về list trong Python

Hãy bắt đầu với một ví dụ:

```
>>> # lập danh sách ngôn ngữ lập trình phổ biến nhất
>>> languages = ['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#']
>>> type(languages)
<class 'list'>
>>> # in danh sách và kiểu
>>> languages
['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#']
>>> # duyệt danh sách trong vòng for
>>> for language in languages:
...     print(language)
```

...

Python

Java

C/C++

Javascript

Go

R

Swift

PHP

C#

```
>>> # lấy ra phần tử đầu tiên
```

```
>>> print('The most popular language is', languages[0])
```

The most popular language is Python

```
>>> # lấy ra 3 phần tử đầu tiên
```

```
>>> print('The top 3 languages are', languages[0:3])
```

The top 3 languages are ['Python', 'Java', 'C/C++']

```
>>> # thêm phần tử mới
```

```
>>> languages += ['Matlab']
```

```
>>> languages
```

['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#', 'Matlab']

```
>>> # cập nhật phần tử số 9
```

```
>>> languages[9] = 'Perl'
```

```
>>> languages
```

['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#', 'Perl']

```
>>> # xóa phần tử số 9
```

```
>>> del languages[9]
```

```
>>> languages
```

['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#']

```
>>> # xóa các phần tử từ vị trí số 5 về cuối
```

```
>>> del languages[5:]
```



```
>>> languages
```

```
['Python', 'Java', 'C/C++', 'Javascript', 'Go']
```

Nếu muốn chạy ở dạng script bạn dùng code sau:

```
languages = ['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#']

# in danh sách và kiểu
print(type(languages), languages)

# duyệt danh sách trong vòng for
print('Most popular programming languages:')
for language in languages:
    print(language)

# lấy ra phần tử đầu tiên
print('The most popular language is', languages[0])

# lấy ra 3 phần tử đầu tiên
print('The top 3 languages are', languages[0:3])

# thêm phần tử mới
languages += ['Matlab']
print(languages)

# cập nhật phần tử số 9
languages[9] = 'Perl'
print(languages)

# xóa phần tử số 9
del languages[9]
print(languages)

# xóa các phần tử từ vị trí số 5 về cuối
del languages[5:]
print(languages)
```

Ví dụ trên minh họa cách làm việc cơ bản với kiểu list trong Python.

Kiểu list trong Python

`list` là một kiểu dữ liệu rất đa năng trong Python có khả năng chứa một danh sách các phần tử (có thể khác nhau về kiểu dữ liệu).

Python quy định giá trị kiểu list phải viết trong cặp dấu ngoặc vuông, các phần tử tách nhau bởi dấu phẩy. Các phần tử trong danh sách được đánh chỉ số (index). Chỉ số bắt đầu từ 0 (giống trong C, C#) và theo vị trí xuất hiện của phần tử.

Ví dụ, lệnh sau

```
companies = ['Google', 'Apple', 'Microsoft', 'Facebook']
```

sẽ tạo ra danh sách gồm 4 phần tử và biến `companies` trỏ tới danh sách đó. Trong danh sách này, 'Google' được đánh chỉ số 0, 'Apple' được đánh chỉ số 1,... Cả 4 phần tử này đều thuộc kiểu `str` (chuỗi).

Danh sách cũng có thể không chứa phần tử nào. Danh sách không chứa phần tử gọi là danh sách rỗng/trống. Danh sách rỗng được biểu diễn bằng `[]`. Ví dụ, `empty = []`.

Biến kiểu list có thể **chứa các phần tử không cùng kiểu**. Ví dụ, trong cùng một biến list có thể chứa phần tử kiểu chuỗi (`str`) và các kiểu số (`int`, `float`, `complex`).

```
>>> mixed = ['Google', 'Apple', 'Microsoft', 'Facebook', 1945, 1954, 1975]
```

```
>>> mixed
```

```
['Google', 'Apple', 'Microsoft', 'Facebook', 1945, 1954, 1975]
```

Kiểu list trong Python tương tự `List<object>` hoặc `ArrayList` trong C#.

Danh sách có thể chứa cả danh sách khác. Khi này nó được gọi là **danh sách lồng nhau** (nested list). Ví dụ:

```
>>> matrix = [[1, 2], [3, 4]]
```

```
>>> matrix
```

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

```
>>> nested = ['Russia', 'Canada', 'USA', 'China', 'Brazil', [17.09, 9.9, 9.8, 9.5, 8.5]]
```

```
>>> nested
```

```
['Russia', 'Canada', 'USA', 'China', 'Brazil', [17.09, 9.9, 9.8, 9.5, 8.5]]
```

Ngoài việc viết trực tiếp các phần tử, bạn có thể tạo danh sách tự động theo cách sau:

```
>>> # lập danh sách các số có dạng 2^x với x thuộc [0, 9]
>>> pow2 = [2 ** x for x in range(10)]
>>> pow2
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
>>> # lập danh sách các số lẻ từ 1 đến 20
>>> odds = [x for x in range(20) if x % 2 == 1]
>>> odds
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> # ghép cặp từ hai danh sách
>>> phrases = [x+y for x in ['Python ', 'C '] for y in ['Language', 'Programming']]
>>> phrases
['Python Language', 'Python Programming', 'C Language', 'C Programming']
```

Cách tạo danh sách này có tên gọi riêng là **list comprehension**. Đây là cách tốt nhất để tạo ra các danh sách dữ liệu.

Truy xuất phần tử của list

Phép toán cơ bản nhất của danh sách là truy xuất phần tử. Python cho phép truy xuất từng phần tử hoặc nhóm phần tử thông qua phép toán cắt (slice) hoặc cắt đoạn (range slice).

Hãy xem ví dụ sau:

```
>>> countries = ['Russia', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
>>> countries[1]
'Canada'
```

`countries[1]` là phép toán slice (cắt) – trích một phần tử trong danh sách `countries` theo chỉ số. Phép toán này tương tự như phép toán truy cập phần tử của mảng trong ngôn ngữ C#. Lưu ý chỉ số trong Python bắt đầu từ 0 (giống như trong C#).

Nếu bạn sử dụng chỉ số âm, Python sẽ tính từ cuối danh sách. Ví dụ:

```
>>> countries = ['Russia', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
```

```
>>> countries[-1]
```

```
'Australia'
```

```
>>> countries[-2]
```

```
'Brazil'
```

Python cũng cho phép thực hiện phép toán slice trên một nhóm phần tử. Hãy xem ví dụ sau:

```
>>> countries = ['Russia', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
```

```
>>> countries[3:5]
```

```
['China', 'Brazil']
```

```
>>> countries[0:3]
```

```
['Russia', 'Canada', 'USA']
```

```
>>> countries[:3]
```

```
['Russia', 'Canada', 'USA']
```

```
>>> countries[3:]
```

```
['China', 'Brazil', 'Australia']
```

```
>>> countries[:]
```

```
['Russia', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
```

Ở đây chúng ta sử dụng phép toán cắt đoạn (range slice) để trích ra một danh sách con.

Để ý cấu trúc chung của phép toán cắt đoạn có dạng `[start:stop]`. Phép toán này sẽ trả lại một danh sách con, lấy từ phần tử có chỉ số `start` đến phần tử có chỉ số `stop-1`.

Như trong ví dụ trên, `countries[3:5]` sẽ trả lại danh sách con lấy từ phần tử có chỉ số 3 ('China') đến phần tử có chỉ số 5-1 ('Brazil').

Nếu `start = 0` thì có thể bỏ qua (không cần viết), như trường hợp `countries[:3]`: `countries[:3]` tương đương với `countries[0:3]`.

Nếu bỏ qua giá trị `stop` thì nó tương ứng với chỉ số của phần tử cuối cùng, như trường hợp `countries[3:]` sẽ trả về từ phần tử có chỉ số 3 đến hết

danh sách. Cách thức này rất tiện lợi để lấy phần tử mà không cần biết số lượng phần tử.

Cá biệt, nếu bỏ qua cả start và stop thì coi như phép slice này trả lại danh sách con bằng danh sách gốc: `countries[:]`.

start và stop có thể nhận giá trị âm với ý nghĩa là chỉ số tính từ cuối danh sách. Ví dụ:

```
>>> countries = ['Russia', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
```

```
>>> countries[-1] # phần tử thứ nhất từ cuối
```

```
'Australia'
```

```
>>> countries[-2] # phần tử thứ hai từ cuối
```

```
'Brazil'
```

```
>>> countries[-1:] # danh sách con chứa phần tử cuối cùng
```

```
['Australia']
```

```
>>> countries[:-1] # danh sách con loại trừ phần tử cuối cùng
```

```
['Russia', 'Canada', 'USA', 'China', 'Brazil']
```

Với danh sách lồng nhau, bạn phải áp dụng phép toán slice cho cả danh sách con. Ví dụ:

```
>>> matrix = [[1, 2], [3, 4]]
```

```
>>> matrix[0] # trả lại phần tử 0 là một danh sách con
```

```
[1, 2]
```

```
>>> matrix[0][0] # trả lại phần tử 0 của danh sách con
```

```
1
```

```
>>> nested = ['Russia', 'Canada', 'USA', 'China', 'Brazil', [17.09, 9.9, 9.8, 9.5, 8.5]]
```

```
>>> nested[5] # phần tử số 5 là một danh sách
```

```
[17.09, 9.9, 9.8, 9.5, 8.5]
```

```
>>> nested[5][4] # lấy phần tử cuối cùng của danh sách con
```

```
8.5
```

Một khi đã truy xuất được phần tử mong muốn, bạn có thể thực hiện các thao tác như cập nhật hoặc xóa.

Thêm mới, cập nhật và xóa danh sách

Để **cập nhật**, bạn chỉ việc gán giá trị mới cho phần tử được chọn. Ví dụ:

```
>>> countries = ['Russia', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
>>> countries[0] = 'Vietnam' # cập nhật phần tử số 0 bằng 1 giá trị đơn mới
>>> countries
['Vietnam', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
>>> countries[0] = ['Vietnam', 'Laos', 'Cambodia'] # cập nhật phần tử số 0 bằng một danh sách mới
(tạo nested list)
>>> countries
[['Vietnam', 'Laos', 'Cambodia'], 'Canada', 'USA', 'China', 'Brazil', 'Australia']
>>> countries[0] = 'Russia' # cập nhật phần tử 0 (danh sách con) thành một phần tử đơn
>>> countries
['Russia', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
>>> countries[1:3] = ['North America', 'South America'] # cập nhật danh sách con
>>> countries
['Russia', 'North America', 'South America', 'Brazil', 'Australia']
>>> countries[1:3] = ['America'] # cập nhật danh sách con
>>> countries
['Russia', 'America', 'China', 'Brazil', 'Australia']
```

Qua loạt ví dụ trên bạn có thể thấy sự linh hoạt của Python khi cập nhật danh sách. Bạn có thể cập nhật một phần tử hoặc một danh sách con bằng một phần tử hoặc danh sách con khác.

Để **xóa**, bạn dùng lệnh `del` trước phần tử cần xóa, trong đó phần tử cần xóa được lựa chọn bằng phép toán slice hoặc range slice. Ví dụ:

```
>>> countries = ['Russia', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
>>> del countries[0]
>>> countries
['Canada', 'USA', 'China', 'Brazil', 'Australia']
>>> del countries[:2]
>>> countries
```

```
['China', 'Brazil', 'Australia']
```

```
>>> del countries[1:]
```

```
>>> countries
```

```
['China']
```

Để **thêm phần tử mới vào cuối danh sách**, bạn có thể sử dụng phép cộng gán += hoặc phương thức append() như sau:

```
>>> countries = ['China'] # danh sách gốc chỉ có China
```

```
>>> countries += ['Russia', 'USA', 'Canada'] # thêm vào cuối danh sách
```

```
>>> countries
```

```
['China', 'Russia', 'USA', 'Canada']
```

```
>>> countries.append('Brazil') # append() chỉ thêm được 1 phần tử
```

```
>>> countries
```

```
['China', 'Russia', 'USA', 'Canada', 'Brazil']
```

```
>>> countries.extend(['India', 'Argentina', 'Kazakhstan']) # extend thêm 1 danh sách
```

```
>>> countries
```

```
['China', 'Russia', 'USA', 'Canada', 'Brazil', 'India', 'Argentina', 'Kazakhstan']
```

Khi dùng append() bạn chỉ thêm được 1 phần tử. Nếu cần thêm một danh sách, hãy dùng phương thức extend().

Một số thao tác khác trên danh sách

Danh sách là một kiểu dữ liệu có thể duyệt (iterable). Do vậy bạn có thể dùng vòng for để **duyệt danh sách** như sau:

```
>>> countries = ['Russia', 'Canada', 'USA', 'China', 'Brazil']
```

```
>>> for c in countries:
```

```
...     print(c.upper())
```

```
...
```

```
RUSSIA
```

```
CANADA
```

```
USA
```

```
CHINA
```

```
BRAZIL
```

Vòng lặp for sẽ tự động duyệt qua từng phần tử theo thứ tự và đặt giá trị tương ứng vào biến điều khiển. Bạn có thể sử dụng biến điều khiển trong vòng lặp. Trong ví dụ trên, mỗi tên quốc gia sẽ lần lượt đặt vào biến điều khiển c. Chúng ta chuyển nó thành chuỗi in hoa để in ra màn hình.

Danh sách hỗ trợ một số phép toán, bao gồm phép ghép danh sách `+`, lặp danh sách `*`, kiểm tra thành viên `in`.

```
>>> # phép cộng tạo ra một danh sách mới từ các danh sách con
>>> companies = ['Google', 'Apple', 'Microsoft', 'Facebook'] + ['IBM', 'HP', 'Xerox', 'Canon']
>>> companies
['Google', 'Apple', 'Microsoft', 'Facebook', 'IBM', 'HP', 'Xerox', 'Canon']
>>> # phép nhân lặp lại các phần tử của danh sách để tạo thành danh sách mới
>>> haha = ['hah', 'hoh'] * 3
>>> haha
['hah', 'hoh', 'hah', 'hoh', 'hah', 'hoh']
>>> # phép toán kiểm tra thành viên
>>> countries = ['Russia', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
>>> 'Russia' in countries
True
>>> 'Vietname' in countries
False
```

Ngoài các phép toán trên, Python cung cấp một số hàm làm việc với danh sách:

- `len()` – trả về số phần tử của danh sách.
- `max()` – trả về phần tử lớn nhất trong danh sách.
- `min()` – trả về phần tử nhỏ nhất của danh sách.

```
>>> countries = ['Russia', 'Canada', 'USA', 'China', 'Brazil', 'Australia']
>>> len(countries)
6
>>> max(countries)
```


'USA'

```
>>> min(countries)
```

'Australia'

Các phương thức của lớp list

Do list là một class trong Python, nó cung cấp một số phương thức bạn có thể sử dụng. Hãy xem cách sử dụng các phương thức này qua các ví dụ sau:

```
>>> languages = ['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#']
```

```
>>> # append và extend bạn đã gặp ở trên
```

```
>>> languages.append('Visual Basic')
```

```
>>> languages
```

```
['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#', 'Visual Basic']
```

```
>>> languages.extend(['Delphi', 'Pascal'])
```

```
>>> languages
```

```
['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#', 'Visual Basic', 'Delphi', 'Pascal']
```

```
>>> # đếm xem một giá trị xuất hiện bao nhiêu lần
```

```
>>> languages.count('C#')
```

```
1
```

```
>>> # xác định chỉ số của một phần tử
```

```
>>> languages.index('Java')
```

```
1
```

```
>>> languages.index('Python')
```

```
0
```

```
>>> # hoạt động giống như pop trên stack: lấy ra phần tử cuối cùng và loại bỏ nó khỏi danh sách
```

```
>>> languages.pop()
```

'Pascal'

```
>>> languages.pop()
```

'Delphi'

```
>>> languages
```

```
['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#', 'Visual Basic']
```

```
>>> # chèn 1 phần tử vào vị trí chỉ định
>>> languages.insert(0, 'Prolog') # chèn 'Prolog' vào vị trí số 0
>>> languages
['Prolog', 'Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#', 'Visual Basic']
>>> # loại bỏ 1 phần tử khỏi danh sách
>>> languages.remove('Prolog')
>>> languages
['Python', 'Java', 'C/C++', 'Javascript', 'Go', 'R', 'Swift', 'PHP', 'C#', 'Visual Basic']
>>> # đảo ngược vị trí các phần tử
>>> languages.reverse()
>>> languages
['Visual Basic', 'C#', 'PHP', 'Swift', 'R', 'Go', 'Javascript', 'C/C++', 'Java', 'Python']
>>> # sắp xếp danh sách
>>> languages.sort()
>>> languages
['C#', 'C/C++', 'Go', 'Java', 'Javascript', 'PHP', 'Python', 'R', 'Swift', 'Visual Basic']
```

Kết luận

Trong phần này chúng ta đã tìm hiểu chi tiết về kiểu dữ liệu tập hợp đầu tiên: list. Có thể thấy list trong Python là một kiểu dữ liệu rất đa năng và linh hoạt. Bạn có thể lưu trữ bất kỳ dữ liệu bên trong list. List cho phép thực hiện các thao tác thêm-sửa-xóa rất linh hoạt.

Kiểu dữ liệu tuple trong Python

Tuple là một kiểu dữ liệu tập hợp khác trong Python và cũng được sử dụng rất rộng rãi. Mặc dù tuple có điểm tương tự như list, nhưng giữa hai kiểu dữ liệu có nhiều điểm khác biệt về bản chất và mục đích sử dụng. Phần này sẽ hướng dẫn chi tiết về cấu trúc dữ liệu tuple cũng như phân biệt khi nào sử dụng tuple, khi nào sử dụng list.

NỘI DUNG

1. Ví dụ minh họa
2. Kiểu dữ liệu tuple trong Python
3. Các phép toán và hàm trên tuple
4. Tuple hay list trong Python?
5. Kết luận

Ví dụ minh họa

```
>>> vn = ('Vietnam', 330_000, 95_000_000, 2500)
>>> us = ('USA', 9_600_000, 350_000_000, 40_000)
>>> us_info = f"Official name: {us[0]}
... Area: {us[1]} km2
... Population: {us[2]}
... GDP/capital: {us[3]} USD"
>>> print(us_info)
Official name: USA
Area: 9600000 km2
Population: 350000000
GDP/capital: 40000 USD
>>> vn_info = f"Official name: {vn[0]}
... Area: {vn[1]} km2
... Population: {vn[2]}
... GDP/capital: {vn[3]} USD"
>>> print(vn_info)
```

Official name: Vietnam

Area: 330000 km²

Population: 95000000

GDP/capital: 2500 USD

Trong ví dụ trên chúng ta đã tạo ra 2 biến kiểu tuple chứa thông tin địa lý cơ bản về Việt Nam và Mỹ, bao gồm tên tiếng Anh, diện tích, dân số, GDP bình quân đầu người.

```
>>> vn = ('Vietnam', 330_000, 95_000_000, 2_500)
```

```
>>> us = ('USA', 9_600_000, 350_000_000, 40_000)
```

So với list, cách tạo tuple chỉ khác ở cặp dấu () (so với [] của list). Các phần tử của tuple có thể thuộc bất kỳ kiểu nào và cần viết tách nhau bởi dấu phẩy.

Sau đó chúng ta sử dụng các thông tin trên để viết ra đoạn giới thiệu về quốc gia:

```
us_info = f"Official name: {us[0]}
```

```
... Area: {us[1]} km2
```

```
... Population: {us[2]}
```

```
... GDP/capital: {us[3]} USD"
```

```
print(us_info)
```

Hãy để ý cách chúng ta sử dụng các thông tin lưu trong tuple. Bạn vẫn sử dụng phép toán slice [] để truy xuất phần tử. Về hình thức không có gì khác biệt so với sử dụng list trong Python.

Nếu muốn viết ở dạng script hãy dùng code sau:

```
vn = ('Vietnam', 330_000, 95_000_000, 2_500)
us = ('USA', 9_600_000, 350_000_000, 40_000)
vn_info = f"Official name: {vn[0]}
Area: {vn[1]} km2
Population: {vn[2]}
GDP/capital: {vn[3]} USD"
us_info = f"Official name: {us[0]}
Area: {us[1]} km2
Population: {us[2]}
GDP/capital: {us[3]} USD"
print(vn_info)
print(us_info)
```

Kiểu dữ liệu tuple trong Python

Dữ liệu kiểu tuple trong Python có thể chứa nhiều giá trị viết trong cặp dấu ngoặc tròn (). Mỗi giá trị viết tách nhau bởi dấu phẩy.

```
vn = ('Vietnam', 330_000, 95_000_000, 2_500)
```

```
us = ('USA', 9_600_000, 350_000_000, 40_000)
```

Lưu ý: nếu chỉ có 1 phần tử trong tuple bạn vẫn phải viết dấu phẩy sau phần tử duy nhất đó. Nếu không viết dấu phẩy, Python sẽ không coi đó là một tuple. Ví dụ tuple với 1 phần tử: `one_tuple = (100,)`

Một tuple có thể không chứa phần tử nào. Khi đó nó được gọi là một tuple rỗng. Tuple rỗng được biểu diễn là `()`. Ví dụ, `empty_tuple = ()`.

Các giá trị trong một tuple có thể thuộc bất kỳ kiểu dữ liệu nào (kể cả kiểu list hay tuple khác). Thường người ta vẫn dùng tuple để nhóm các dữ liệu khác nhau vào một đơn vị. Như ở trên, chúng ta nhóm tên quốc gia, diện tích, dân số, GDP bình quân vào cùng một tuple.

Trong trường hợp phần tử của một tuple có thể là một tuple khác, người ta gọi là tuple lồng nhau (nested tuple). Ví dụ:

```
>>> # thông tin về Singapore: tên tiếng anh, dân số, các ngôn ngữ thông dụng
```

```
>>> sg = ('Singapore', 5_639_000, ('Chinese', 'English'))
```

Đặc điểm quan trọng của tuple là **tính bất biến** (immutability).

Đặc điểm này khiến tuple **không thể cập nhật** một khi đã được tạo ra. Bạn chỉ có thể lấy một phần của tuple để tạo ra tuple mới chứ không thể cập nhật tuple.

Tính bất biến cũng khiến bạn không thể xóa một phần tuple. Bạn chỉ có thể **xóa toàn bộ tuple** bằng lệnh `del` như sau:

```
>>> sg = ('Singapore', 5_639_000, ('Chinese', 'English'))
```

```
>>> del sg
```

Bạn có thể **truy xuất các phần tử** của tuple qua phép toán *slice* và *range slice* giống hệt như đối với list. Hai phép toán này đã trình bày kỹ trong nội dung về list trong Python. Bạn có thể xem lại nếu không nhớ.

```
>>> sg = ('Singapore', 5_639_000, ('Chinese', 'English'))
```

```
>>> # truy xuất 1 phần tử
```

```
>>> sg[0]
```

```
'Singapore'
```

```
>>> # truy xuất một tuple con sử dụng range slice
```

```
>>> sg[0:2]
```

```
('Singapore', 5639000)
```

```
>>> sg[:2]
```

```
('Singapore', 5639000)
```

```
>>> # đối với nested tuple
```

```
>>> sg[2] # kết quả là một tuple khác
```

```
('Chinese', 'English')
```

```
>>> sg[2][1] # truy xuất phần tử số 1 của tuple lồng
```

```
'English'
```

Các phép toán và hàm trên tuple

Kiểu tuple trong Python cũng hỗ trợ các phép toán giống như với list, bao gồm:

- phép cộng tuple `+` trả về một tuple mới bao gồm các phần tử của cả hai tuple
- lặp tuple `*` trả về một tuple mới bằng cách lặp lại các phần tử của một tuple
- kiểm tra thành viên `in` – trả về True/False tùy thuộc xem giá trị có nằm trong tuple hay không

Hãy xem cách sử dụng các phép toán này qua ví dụ sau:

```
>>> # phép cộng hai tuple trả về một tuple mới
```

```
>>> vn = ('Vietnam', 330_000, 95_000_000, 2500) + ('Vietnamese', ('ASEAN', 'UN', 'APEC'))
```

```
>>> vn
```

```
('Vietnam', 330000, 95000000, 2500, 'Vietnamese', ('ASEAN', 'UN', 'APEC'))
```

```
>>> # kiểm tra thành viên
```

```
>>> 'Vietnam' in vn
```

```
True
```

```
>>> 'Vietnamese' in vn
```

```
True
```

```
>>> # phép lặp tuple
```

```
>>> ('ha', 'ho') * 3
```

```
('ha', 'ho', 'ha', 'ho', 'ha', 'ho')
```

Python cung cấp sẵn các hàm làm việc với tuple, bao gồm `len()` – lấy số phần tử, `max()` – lấy phần tử lớn nhất, `min()` – lấy phần tử nhỏ nhất, `tuple()` – chuyển đổi list về tuple. Hãy xem cách sử dụng các hàm trên qua ví dụ sau:

```
>>> vn = ('Vietnam', 330000, 95000000, 2500, 'Vietnamese', ('ASEAN', 'UN', 'APEC'))
```

```
>>> len(vn)
```

```
6
```

```
>>> org = ('ASEAN', 'UN', 'APEC')
```

```
>>> # min và max chỉ hoạt động nếu các phần tử của tuple có cùng kiểu
```

```
>>> min(org)
```

```
'APEC'
```

```
>>> max(org)
```

```
'UN'
```

Chú ý, `min()` và `max()` chỉ hoạt động nếu tất cả phần tử của tuple có cùng kiểu (ví dụ, cùng là số hoặc cùng là xâu). Nếu không cùng kiểu, phép so sánh của các hàm này không hoạt động được. Thực tế, hai hàm này gần như không có ý nghĩa với tuple.

Lớp tuple cũng cung cấp các phương thức `count()` – đếm số lần xuất hiện của 1 giá trị và `index()` – xác định chỉ số của phần tử:

```
>>> vn = ('Vietnam', 330000, 95000000, 2500, 'Vietnamese', ('ASEAN', 'UN', 'APEC'))
```

```
>>> vn.count('Vietnam')
```

```
1
```

```
>>> vn.index('Vietnamese')
```

```
4
```

Tuple hay list trong Python?

Qua các nội dung của phần này và phần trước đó (list trong Python) bạn có thể thấy có rất nhiều điểm tương tự giữa tuple và list. Tuy nhiên, tuple và list có những điểm phân biệt rất rõ ràng. Sự khác biệt này quyết định giá trị sử dụng của từng kiểu dữ liệu.

Thứ nhất, list là kiểu dữ liệu *khả biến* (mutable) trong khi tuple là kiểu dữ liệu *bất biến* (immutable). Nghĩa là, một khi đã được tạo ra trong bộ nhớ, list có thể thay đổi còn tuple không thể thay đổi. Các thao tác biến đổi trên tuple sẽ đều tạo ra object mới chứ không thay đổi object sẵn có.

Từ khía cạnh nào đó có thể hình dung tuple là phiên bản chỉ đọc thu gọn của list.

Với đặc thù trên, list thường dùng làm kho dữ liệu cho chương trình. Ví dụ, bạn có thể đọc dữ liệu từ file vào list, thực hiện các biến đổi trên list và lưu trở lại file. Dữ liệu lưu trong list được xử lý linh hoạt uyển chuyển hơn. Bạn không thể sử dụng tuple cho mục đích này.

Trong khi đó, tuple được sử dụng để truyền dữ liệu trong chương trình. Ví dụ, nếu một hàm cần trả lại nhiều kết quả, nó có thể trả về một tuple. Kết quả trả về ở dạng tuple nhẹ, nhanh và an toàn hơn. Ở đây mặc dù có thể dùng list nhưng không khuyến khích.

Mặc dù list cho phép lưu trữ dữ liệu thuộc nhiều kiểu khác nhau, người ta thường dùng list để lưu dữ liệu có cùng kiểu. Trong khi đó, tuple thường được dùng để lưu trữ kết quả thuộc các kiểu khác biệt để truyền qua lại trong chương trình.

Ví dụ, nếu khách hàng có các thông tin về họ tên, email, số điện thoại, địa chỉ. Danh sách khách hàng (tập hợp bản ghi) nên được lưu trong list. Nếu cần truyền thông tin về một khách hàng (dữ liệu 1 bản ghi, ví dụ, trả về từ hàm) thì nên dùng tuple chứa các thông tin của khách hàng đó.

Kết luận

Phần này đã hướng dẫn cách sử dụng kiểu dữ liệu tuple trong Python và phân biệt tuple với list. Nhìn chung có thể hình dung tuple như một phiên bản chỉ đọc và đơn giản hơn của list dùng để truyền dữ liệu trong chương trình.

Hàm (Function) trong Python

Việc sử dụng hàm trong Python là rất phổ biến. Trước đây, bạn đã gặp và sử dụng nhiều hàm khác nhau. Python cho phép bạn tự xây dựng hàm riêng và sau đó có thể sử dụng chúng giống như các hàm xây dựng sẵn. Bài học này sẽ hướng dẫn chi tiết các vấn đề liên quan đến xây dựng và sử dụng hàm trong Python.

NỘI DUNG

1. Khái niệm hàm trong Python
2. Xây dựng và sử dụng hàm trong Python
3. Docstring cho hàm
4. Biến cục bộ và phạm vi của biến
5. Đệ quy trong Python
6. Kết luận

Khái niệm hàm trong Python

Trong các ngôn ngữ lập trình đều cung cấp khả năng nhóm các đoạn code lại thành một đơn vị và đặt tên. Một nhóm code có đặt tên như vậy cung cấp khả năng tái sử dụng ở nhiều vị trí, thay vì phải viết lặp lại các đoạn code. Loại đơn vị code như vậy trong các ngôn ngữ lập trình thường được gọi là hàm (function), thủ tục (procedure), chương trình con (sub routine),...

Tương tự, Python cũng cung cấp khả năng xây dựng các khối code được đặt tên. Trong python người ta gọi một đơn vị như vậy là hàm (function).

Việc sử dụng hàm trong Python là rất phổ biến. Trong tài liệu này bạn đã gặp và sử dụng nhiều hàm khác nhau. Để xuất/nhập dữ liệu với giao diện console, bạn đã sử dụng hàm `print()` và `input()`. Để tìm số phần tử của danh sách/tuple hoặc độ dài chuỗi, bạn dùng hàm `len()`. Các hàm trên được gọi chung là hàm xây dựng sẵn (built-in function).

Nhìn chung bạn có thể dễ ý thấy việc sử dụng các hàm xây dựng sẵn có một số điểm sau:

- Hàm có thể trả về kết quả hoặc không: ví dụ, `print()` không trả về giá trị nào, trong khi `len()` trả lại thông tin về độ dài chuỗi.

- Hàm có thể cần thông tin đầu vào hoặc không: ví dụ, `len()` cần đầu vào là một chuỗi/list/tuple, `print()` có thể nhận thông tin đầu vào (là bất kỳ dữ liệu gì) hoặc không cần thông tin gì (khi đó nó in ra một dòng trống).
- Thông tin đầu vào có thể phải đáp ứng yêu cầu về kiểu: ví dụ, để dùng `len()` thì đầu vào phải là 1 kiểu tập hợp (như list, tuple, str) chứ không thể, ví dụ, là một số. Hàm `print()` lại có thể chấp nhận bất kỳ thông tin gì.

Python cho phép bạn tự xây dựng hàm riêng và sau đó có thể sử dụng chúng giống như các hàm xây dựng sẵn. Các hàm do bạn tự xây dựng về bản chất không có gì khác so với các hàm xây dựng sẵn.

Xây dựng và sử dụng hàm trong Python

Hãy xem ví dụ sau đây:

```
def fact(n):  
    '''Calculate the factorial of an input number  
  
    Params:  
  
    n (int) : positive number  
  
    Returns:  
  
    int: value n!  
    ...  
  
    p = 1  
  
    for i in range(1, n+1):  
        p *= i  
  
    return p  
  
print('3! = ', fact(3))  
print('4! = ', fact(4))  
print('5! = ', fact(5))
```

Trong ví dụ trên chúng ta đã định nghĩa hàm `fact()` (tính giai thừa của một số nguyên) và sử dụng hàm này để tính 3!, 4! và 5!. Toàn bộ khối `def fact(n):` cho đến hết `return p` là phần **định nghĩa của hàm `fact()`**. `fact(3)`, `fact(4)`, `fact(5)` là những **lời gọi hàm**.

Hàm được định nghĩa bằng **từ khóa** `def`. Lệnh `def` cũng là một lệnh phức hợp với 1 clause với cấu trúc như sau:

```
def fact(n):
    """
    Calculate the factorial of an input number
    Params: n (int)
    Returns: n! (int)
    """
    p = 1
    for i in range(1, n+1):
        p *= i
    return p
```

Theo sau từ khóa `def` là **tên hàm** và cặp dấu `()`. Những gì đặt ở giữa cặp dấu `()` được gọi là **tham số** (hình thức) của hàm. Nếu có nhiều tham số thì chúng viết tách nhau bởi dấu phẩy. Trong hàm `fact()` chỉ có 1 tham số `n`.

Tên hàm trong Python phải tuân theo quy tắc đặt định danh. Ngoài ra tên hàm thường đặt theo quy ước "underscore notation" giống như tên biến: tên viết chữ thường, nếu có nhiều từ thì viết tách nhau bởi dấu gạch chân.

Lưu ý tham số trong Python không cần chỉ định kiểu dữ liệu.

Từ khóa `def`, tên hàm và danh sách tham số tạo thành header của lệnh.

Ngay sau header là một/một số dòng văn bản gọi là **docstring**. Docstring không bắt buộc và có tác dụng cung cấp tài liệu hỗ trợ cho việc sử dụng hàm. Chúng ta sẽ quay lại với docstring ở phần sau.

Sau docstring là code của thân hàm. Trong code thân hàm bạn có thể sử dụng biến từ tham số (biến `n` trong ví dụ trên).

Nếu hàm trả lại kết quả cho nơi gọi, trong hàm phải có lệnh `return <giá trị>`. Nếu hàm không trả lại kết quả, bạn không cần dùng `return`.

Phần định nghĩa của hàm sẽ không có giá trị gì nếu chúng ta không sử dụng hàm đó trong code. Việc **sử dụng hàm** (gọi hàm) tự xây dựng không có gì khác biệt với các hàm xây dựng sẵn:

```
print('3! = ', fact(3))
print('4! = ', fact(4))
```

Để sử dụng hàm, chúng ta viết tên hàm và viết các giá trị tham số trong ngoặc ().

Một trong những vấn đề quan trọng hàng đầu khi xây dựng và sử dụng hàm là tham số. Tham số cho hàm trong Python bao gồm tham số bắt buộc, tham số mặc định và tham số biến động.

Docstring cho hàm

Docstring là khái niệm riêng trong Python. Docstring là chuỗi ký tự nằm ngay sau header của hàm và đóng vai trò tài liệu hướng dẫn cho hàm.

Docstring được sử dụng cho hàm, class, module và package.

Hãy xem lại ví dụ về hàm tính giai thừa:

```
def fact(n):  
    '''    Calculate the factorial of an input number  
  
    Params:  
  
    n (int) : positive number  
  
    Returns:  
  
    int: value n!  
    ...  
  
    p = 1  
  
    for i in range(1, n+1):  
        p *= i  
  
    return p
```

Một số trình biên tập code hoặc IDE có khả năng đọc docstring để cung cấp hỗ trợ. Ví dụ trong Pycharm:

```

function.py x
1 def fact(n):
2     """ calculate the factorial of an input number
3
4     Params:
5     ~~~~~
6     n (int) : positive number
7
8     Returns:
9     ~~~~~
10    int: value n!
11    """
12    p = 1
13    for i in range(1, n + 1):
14        p *= i
15    return p
16
17 fact(10)
  
```

PEP 8: E305 expected 2 blank lines after class or function definition, found 1

Reformat file Alt+Shift+Enter More actions... Alt+Enter

function
def fact(n: {__add__}) -> int

Calculate the factorial of an input number
Params: n (int) : positive number
Returns: int: value n!

Visual Code cũng có khả năng tương tự.

Nếu hàm (và các đơn vị code khác như class, module, package) được viết docstring đầy đủ, người khác sử dụng code của bạn sẽ dễ dàng hơn. Bản thân bạn cũng dễ dàng hơn khi sử dụng hàm do mình viết.

Nếu người khác sử dụng hàm của bạn viết ở chế độ tương tác có thể sử dụng hàm help như sau:

```
>>> help(fact)
```

Help on function fact in module `__main__`:

```
fact(n)
```

Calculate the factorial of an input number

Params:

```
n (int) : positive number
```

Returns:

```
int: value n!
```

```
>>>
```

Từ đây người dùng có thể dễ dàng hiểu được hàm bạn viết.

Ngoài ra docstring cũng có thể được sử dụng qua thuộc tính `__doc__` mà bất kỳ hàm/class nào đều có:

```
>>> print(fact.__doc__)
```

Calculate the factorial of an input number

Params:

`n (int)` : positive number

Returns:

int: value n!

```
>>>
```

Mặc dù không có quy định nào về cách viết docstring, đa số sử dụng quy ước định dạng như sau:

```
def some_function(argument1):  
    """Summary or Description of the Function  
  
    Parameters:  
  
    argument1 (int): Description of arg1  
  
    Returns:  
  
    int:Returning value  
  
    """
```

Biến cục bộ và phạm vi của biến

Trong một file mã nguồn bạn thường viết nhiều hàm. Trong file code bạn cũng đồng thời có những mã không nằm trong thân hàm nào.

Ngoài thân hàm bạn có thể khai báo và sử dụng biến như đã học. Trong khối code thân của hàm (còn gọi là suite) bạn có thể khai báo các biến mới để sử dụng.

Từ đây phát sinh vấn đề:

- Một biến được khai báo ngoài hàm thì trong thân hàm có thể sử dụng lại biến đó không?
- Một biến khai báo trong thân hàm thì ngoài hàm có thể sử dụng được biến đó không?

- Điều gì xảy ra nếu cả trong thân hàm và ngoài thân hàm khai báo cùng một biến?
- Một biến khai báo trong thân hàm này có thể sử dụng trong thân hàm khác không (các hàm nằm trong cùng một file code)?

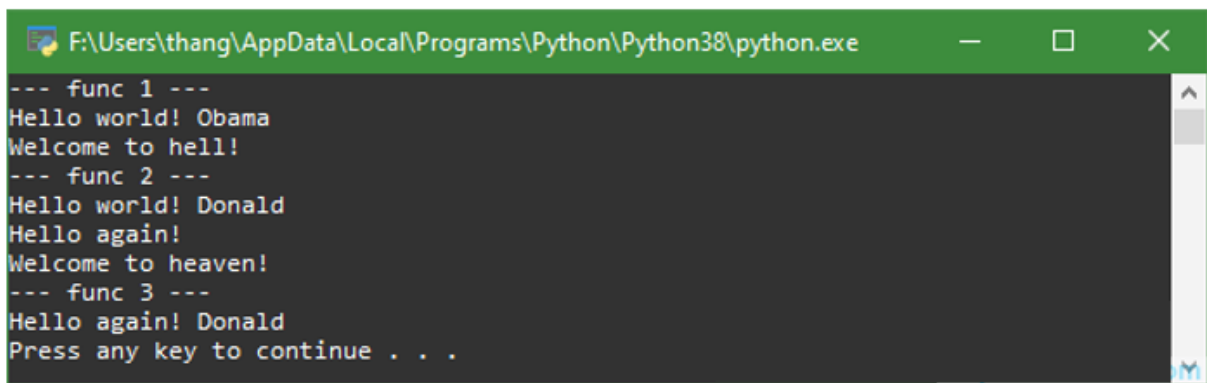
Các vấn đề trên liên quan đến **phạm vi tác dụng** (scope) của biến trong file code. Phạm vi tác dụng của biến là những nơi (trong file code) bạn có thể sử dụng được biến đó.

Trước hết hãy cùng xem ví dụ sau:

```
msg = 'Hello world!'
name = 'Donald'
def func1():
    print('--- func 1 ---')
    name = 'Obama'
    print(msg, name)
    greeting = 'Welcome to hell!'
    print(greeting)

def func2():
    print('--- func 2 ---')
    global msg
    print(msg, name)
    msg = 'Hello again!'
    print(msg)
    greeting = 'Welcome to heaven!'
    print(greeting)
def func3():
    print('--- func 3 ---')
    print(msg, name)

func1()
func2()
func3()
```



```
F:\Users\thang\AppData\Local\Programs\Python\Python38\python.exe
--- func 1 ---
Hello world! Obama
Welcome to hell!
--- func 2 ---
Hello world! Donald
Hello again!
Welcome to heaven!
--- func 3 ---
Hello again! Donald
Press any key to continue . . .
```

Biến trong Python thuộc về một trong hai phạm vi cơ bản: biến cục bộ và biến toàn cục. **Biến toàn cục** được khai báo bên ngoài hàm, trực tiếp trong file code. **Biến cục bộ** được khai báo bên trong hàm.

Trong ví dụ trên, `msg` và `name` khai báo đầu tiên là hai biến toàn cục. Trong hàm `func1` có biến cục bộ `greeting`, trong `func2` cũng khai báo một biến cục bộ cùng tên `greeting`.

Biến cục bộ khai báo trong hàm nào chỉ có thể sử dụng bên trong hàm đó. Hàm khác không nhìn thấy nó. Ngoài phạm vi của hàm, biến cục bộ không còn tồn tại. Biến `greeting` trong `func2` và `func1` là hoàn toàn khác biệt, dù cùng tên. Trong `func3` không biết gì về `greeting`.

Biến toàn cục có thể được sử dụng ở bất kỳ chỗ nào trong file code sau vị trí nó được khai báo, kể cả trong hàm. Như trong hàm `func3` sử dụng biến toàn cục `msg` và `name`, trong `func1` sử dụng biến toàn cục `msg`, trong `func2` sử dụng biến toàn cục `msg` và `name`.

Trong `func1` có điểm đặc biệt: lệnh `name = 'Obama'` không hề tác động lên biến `name` toàn cục. Nó là một lệnh khai báo biến cục bộ `name`. Khi gặp phép gán trong thân hàm, Python sẽ coi đây là lệnh tạo biến cục bộ, ngay cả khi có một biến toàn cục trùng tên.

Khi có lệnh tạo biến cục bộ trùng tên với biến toàn cục thì biến cục bộ sẽ che biến toàn cục. Tức là trong hàm đó Python sẽ chỉ sử dụng biến cục bộ. Như vậy trong `func1`, biến cục bộ `name` sẽ che đi biến toàn cục cùng tên. Biến `msg` vẫn là biến cục bộ (giá trị 'Hello world'). Như vậy kết quả in ra sẽ là 'Hello world! Obama' chứ không phải 'Hello world! Donald'.

Trong `func2` bạn gặp cách dùng đặc biệt: `global msg`. Lệnh này báo hiệu rằng sẽ sử dụng biến `msg` toàn cục. Khi đó, lệnh gán `msg = 'Hello again!'` không tạo ra biến cục bộ `msg` mới. Thay vào đó lệnh này tác

động lên biến msg toàn cục. Như vậy, lệnh gán `msg = 'Hello again!'` tác động lên biến msg toàn cục.

Do func2 tác động lên biến toàn cục msg nên trong lời gọi func3 sẽ in ra 'Hello again! Donald' (biến name toàn cục vẫn giữ nguyên giá trị từ đầu). Lệnh gán `name = 'Obama'` trong func1 sinh ra một biến cục bộ cùng tên name chứ không tác động lên biến toàn cục name.

Đệ quy trong Python

Đệ quy là hiện tượng một hàm gọi lại chính nó. Python cũng cho phép cơ chế này.

Hãy xem lại ví dụ về tính giai thừa. Giai thừa của số nguyên dương n được định nghĩa như sau:

- Nếu $n = 1$ thì $n! = 1$;
- Nếu $n > 1$ thì $n! = n * (n-1)!$

Đây là loại định nghĩa đệ quy. Định nghĩa đệ quy là loại định nghĩa qua chính nó. Như trên người ta định nghĩa $n!$ thông qua $(n-1)!$. Đặc thù của định nghĩa đệ quy là phải có một điều kiện dừng ($n = 1$ thì $n! = 1$).

Với định nghĩa giai thừa như trên chúng ta có thể viết lại hàm fact theo cách khác như sau:

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n - 1)
```

Hàm fact giờ được viết lại theo đúng định nghĩa giai thừa kiểu đệ quy: trong thân hàm fact có lời gọi đến chính hàm fact với tham số $n - 1$.

Mặc dù cách viết đệ quy thường ngắn gọn, việc thực thi chương trình dạng đệ quy tốn bộ nhớ hơn.

Kết luận

Trong phần này bạn đã học chi tiết về cách xây dựng và sử dụng hàm trong Python.

Có thể thấy, việc xây dựng hàm trong Python tương đối đơn giản so với các ngôn ngữ khác. Python hỗ trợ nhiều cách khác nhau để truyền và sử dụng tham số. Python cũng có đặc điểm khác biệt về docstring để tạo documentation.

Tham số cho hàm trong Python

Một trong những vấn đề quan trọng hàng đầu khi xây dựng và sử dụng hàm là tham số. Tham số cho hàm trong Python bao gồm tham số bắt buộc, tham số mặc định và tham số biến động.

NỘI DUNG

1. Tham số cho hàm
2. Keyword argument
3. Tham số mặc định
4. Tham số biến động, *args
5. Tham số biến động với keyword argument, **kwargs
6. Chỉ báo kiểu cho tham số hàm và kết quả
7. Kết luận

Tham số cho hàm

Tham số bắt buộc là loại tham số mặc định của hàm trong Python. Hãy xem ví dụ sau:

```
def equation(a, b, c):  
    '''solving quadratic equation  
    parameters: a, b, c are float  
    return: a tuple (float, float)  
    ...  
    from math import sqrt  
    d = b * b - 4 * a * c  
    if d >= 0:  
        x1 = (- b + sqrt(d)) / (2 * a)  
        x2 = (- b - sqrt(d)) / (2 * a)  
        return (x1, x2)
```

Đây là ví dụ về giải phương trình viết ở dạng hàm với 3 tham số kiểu số (int hoặc float) a, b, c. Kết quả trả về là một tuple chứa hai nghiệm kiểu float.

Trong hàm `equation`, `a`, `b`, `c` là 3 tham số bắt buộc. Các tham số sử dụng trong quá trình khai báo hàm được gọi là **tham số hình thức** (formal parameter). Sở dĩ gọi là tham số hình thức là vì chúng chỉ có tên, không có giá trị. Giá trị của các tham số này chỉ xuất hiện trong lời gọi hàm.

Khi gọi (sử dụng) hàm `equation` bạn bắt buộc phải truyền đủ 3 giá trị tương ứng với 3 tham số `a`, `b`, `c`. Khi này các giá trị truyền cho hàm được gọi là tham số thực, vì giờ nó có giá trị cụ thể.

Bạn có thể trực tiếp truyền giá trị hoặc truyền biến làm tham số thực cho lời gọi hàm:

```
(x1, x2) = equation(1, 2, 1) # truyền trực tiếp giá trị làm tham số (thực)
print('real solutions:')
print(f'x1 = {x1}')
print(f'x2 = {x2}')
aa, bb, cc = 1, 2, 1
(x1, x2) = equation(aa, bb, cc) # truyền biến làm tham số
print(f'x1 = {x1}')
print(f'x2 = {x2}')
```

Dựa vào header chúng ta không biết `a`, `b`, `c` thuộc kiểu dữ liệu nào. Từ tính toán ở thân hàm cho thấy `a`, `b`, `c` phải thuộc kiểu số.

Do Python không yêu cầu chỉ định kiểu khi viết tham số, bạn phải căn cứ vào tài liệu sử dụng của hàm để biết cách dùng đúng. Để hỗ trợ người sử dụng hàm, Python cũng khuyến nghị khi xây dựng hàm nên viết đầy đủ docstring mô tả cho hàm.

Khi sử dụng hàm bạn viết tên hàm và cung cấp danh sách tham số theo yêu cầu. Đặc biệt lưu ý, giá trị các tham số (gọi là tham số thực) phải viết đúng thứ tự (về kiểu) như hàm yêu cầu. Ví dụ, hàm yêu cầu kiểu theo thứ tự (`int`, `string`, `list`) thì bạn phải viết giá trị theo đúng trật tự đó.

Keyword argument

Python cho phép sử dụng một cách truyền tham số khác có tên gọi là **keyword argument**. Trong cách truyền tham số này, bạn phải biết tên của tham số (tên sử dụng trong khai báo hàm).

Hãy xem ví dụ:

```
(x1, x2) = equation(b = 3, c = 2, a = 1)
```

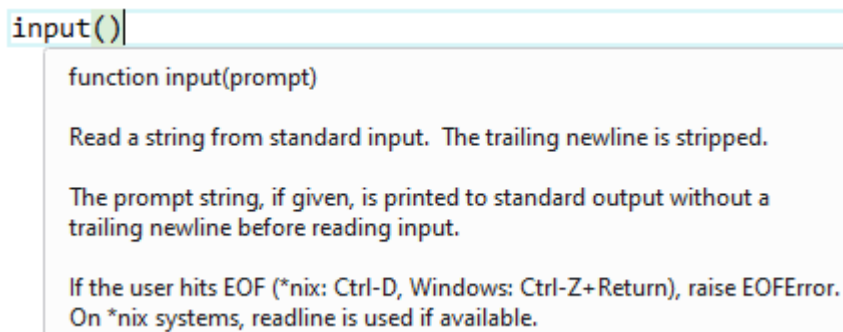
Đây là một cách khác để gọi hàm equation.

Hãy để ý cách viết danh sách tham số (b = 3, c = 2, a = 1). Chúng ta sử dụng tên tham số (a, b, c) và gán cho nó giá trị tương ứng.

Với cách truyền tham số này, chúng ta không cần quan tâm đến thứ tự tham số. Python có thể phân biệt rõ giá trị nào truyền cho tham số nào thông qua tên gọi.

Nếu bạn đã học lập trình C# sẽ thấy cách truyền tham số này rất quen thuộc.

Để sử dụng cách gọi này bạn phải biết tên chính xác của tham số sử dụng trong lời khai báo hàm. Các IDE đều hỗ trợ hiển thị các thông tin này nếu bạn trỏ chuột lên tên hàm.



input()

function input(prompt)

Read a string from standard input. The trailing newline is stripped.

The prompt string, if given, is printed to standard output without a trailing newline before reading input.

If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError. On *nix systems, readline is used if available.

Tham số mặc định

Khi xây dựng hàm, trong một số trường hợp bạn muốn đơn giản hóa lời gọi hàm bằng cung cấp sẵn giá trị của một vài tham số. Nếu người dùng không cung cấp giá trị cho tham số đó thì sẽ sử dụng giá trị cung cấp sẵn.

Lấy ví dụ, hàm print() quen thuộc có bốn tham số mặc định end, sep, file, flush. Tham số end chỉ định ký tự cần in ra khi kết thúc xuất dữ liệu. Ký tự sep chỉ định ký tự cần in ra khi kết thúc in mỗi biến (trong trường hợp in ra nhiều giá trị).

Mặc định end có giá trị '\n', nghĩa là cứ kết thúc xuất dữ liệu thì sẽ chuyển xuống dòng mới, và sep có giá trị ' ' (dấu cách), nghĩa là nếu in ra nhiều giá trị thì các giá trị phân tách nhau bởi dấu cách.

Tuy nhiên trong lời gọi hàm `print()` bạn không cần truyền giá trị cho các biến này. Khi đó, `end` và `sep` đều sử dụng giá trị mặc định. Đó cũng là cách chúng ta sử dụng hàm `print()` bấy lâu nay.

Khi xây dựng hàm bạn cũng có thể chỉ định một vài tham số làm tham số mặc định như vậy.

Hãy xem ví dụ sau:

```
def equation(a, b=0, c=0):  
    from math import sqrt  
    d = b * b - 4 * a * c  
    if d >= 0:  
        x1 = (- b + sqrt(d)) / (2 * a)  
        x2 = (- b - sqrt(d)) / (2 * a)  
    return (x1, x2)
```

Ở đây chúng ta xây dựng lại hàm `equation` nhưng giờ `b` và `c` trở thành hai tham số mặc định.

Trong phương trình bậc hai $ax^2 + bx + c = 0$, ngoại trừ `a` bắt buộc khác 0, `b` và `c` đều có thể nhận giá trị 0. Vì vậy chúng ta đặt giá trị mặc định cho `b` và `c` bằng 0. Trong lời gọi hàm `equation`, nếu không truyền giá trị cho `b` và `c` thì sẽ sử dụng giá trị mặc định.

Với hàm `equation` như trên chúng ta giờ có thể gọi như sau:

```
equation(10) # chỉ cung cấp a = 10 (bắt buộc) bỏ qua b và c  
equation(10, 5) # cung cấp a = 10, b = 5, bỏ qua c  
equation(10, 5, -10) # cung cấp đủ a, b, c  
equation(10, c = -1) # cung cấp a và c (thông qua keyword argument)  
equation(a = 10, c = -9) # cung cấp a và c sử dụng keyword argument
```

Tham số biến động, *args

Khi đọc code trong các tài liệu Python bạn có thể sẽ gặp cách viết tham số `*args` và `**kwargs`. Cách viết này được sử dụng phổ biến như một quy tắc ngầm để chỉ định một loại tham số đặc biệt: **tham số biến động** (variable-length arguments). Nói theo cách khác, đây là loại tham số mà số lượng không xác định.

*args và **kwargs được sử dụng cho hai tình huống khác nhau.

Lấy ví dụ, khi sử dụng hàm print bạn có thể dễ dàng thấy một điểm đặc biệt: bạn có thể truyền rất nhiều biến cho print. Số lượng biến truyền vào cho print không bị giới hạn:

```
>>> print('hello')
```

```
hello
```

```
>>> print('hello', 'world')
```

```
hello world
```

```
>>> print('hello', 'world', 'from Python')
```

```
hello world from Python
```

Đây là một tính năng của hàm trong Python gọi là tham số biến động. Tính năng này cho phép một hàm Python tiếp nhận không giới hạn số lượng biến.

Hãy xem ví dụ sau đây:

```
def sum(start, *numbers):  
    for n in numbers:  
        start += n  
    return start  
  
sum(0, 1, 2) # = 3  
sum(1, 2, 3) # = 6  
sum(0, 1, 2, 3, 4, 5, 6) # = 21
```

Trong ví dụ này chúng ta xây dựng một hàm cho phép cộng một số lượng giá trị bất kỳ vào một giá trị cho trước. Trong hàm sum, start là một tham số bắt buộc.

Hãy để ý cách viết tham số thứ hai `*numbers`. Đây là quy định của Python: nếu một tham số bắt đầu bằng ký tự *, Python sẽ coi nó như một tuple. Theo đó, trong thân hàm bạn có thể sử dụng các giá trị trong tuple này.

Trong ví dụ trên chúng ta giả định rằng trong numbers chỉ chứa giá trị số và chúng ta liên tiếp cộng dồn nó vào biến start.

Từ khía cạnh sử dụng hàm, bạn có thể viết bất kỳ số lượng biến nào trong lời gọi hàm. Với hàm `sum` ở trên, tất cả các biến từ vị trí số 2 trở đi sẽ được đóng vào một tuple để truyền vào hàm.

Do vậy chúng ta có thể viết hàng loạt lời gọi hàm với lượng tham số khác nhau:

```
sum(0, 1, 2) # = 3
sum(1, 2, 3) # = 6
sum(0, 1, 2, 3, 4, 5, 6) # = 21
```

Khi sử dụng tham số biến động cần lưu ý rằng, loại tham số này nên để ở cuối danh sách. Nếu bạn để loại tham số này ở đầu danh sách, bạn sẽ không bao giờ truyền được giá trị cho các tham số khác nữa. Không tin bạn cứ thử mà xem!

Như một quy tắc ngầm, lập trình viên Python thường ký hiệu tham số biến động là `*args`.

Tham số biến động với keyword argument, ****kwargs**

Một tình huống khác xảy ra với tham số biến động là khi bạn muốn sử dụng keyword argument cho phần biến động.

Như ở trên bạn đã hiểu thế nào là keyword argument. Vậy làm thế nào để có thể truyền không giới hạn tham số nhưng theo kiểu keyword argument?

Hãy xem ví dụ sau:

```
def foo(**kwargs):
    for key, value in kwargs.items():
        print(f'{key} = {value}')
foo(a = 1, b = 2, c = 3)
```

Kết quả in ra là

```
a = 1
b = 2
c = 3
```


Qua ví dụ nhỏ này bạn đã thấy hiệu quả của cú pháp `**kwargs`:

- Bạn có thể truyền không giới hạn tham số;
- Mỗi tham số đều có thể truyền ở dạng keyword argument `tên_biến = giá_trị`;

Để sử dụng `**kwargs` trong hàm, bạn cần duyệt nó trong vòng for để lấy ra các cặp khóa (key) và giá trị (value): `for key, value in kwargs.items()`.

Tùy vào từng hàm, bạn có thể sử dụng key và value theo những cách khác nhau.

Giả sử nếu bạn cần tính tổng, bạn có thể xây dựng lại hàm sum như sau:

```
def sum(start: int, **numbers):  
    for _, value in numbers.items():  
        start += value  
    return start  
  
print(sum(start = 0, a = 1, b = 2, c = 3))
```

Tương tự như `*args`, `**kwargs` (viết tắt của keyword arguments) cũng là một tên gọi quy ước được đa số sử dụng. Bạn có thể dùng tên gì tùy thích nhưng nhớ đặt hai dấu `**` phía trước.

Khi có nhiều loại tham số trong một lời khai báo hàm, bạn nên để chúng theo thứ tự như sau:

1. các tham số bắt buộc,
2. các tham số mặc định,
3. `*args`,
4. `**kwargs`.

Chỉ báo kiểu cho tham số hàm và kết quả

Một trong những vấn đề quan trọng hàng đầu khi xây dựng hỗ trợ sử dụng hàm là cung cấp thông tin về kiểu dữ liệu của tham số. Với các ngôn ngữ định kiểu tĩnh như ngôn ngữ họ C thì đây không là vấn đề.

Trong Python, nếu bạn không hỗ trợ cung cấp thông tin về kiểu dữ liệu của tham số, người sử dụng hàm sẽ gặp nhiều khó khăn.

Trong phần về hàm trong Python bạn đã biết khái niệm và vai trò của docstring. Trong Python, mặc dù docstring không bắt buộc nhưng bạn nên sử dụng bất kỳ khi nào có thể.

Tuy nhiên, tài liệu hỗ trợ của hàm hoàn toàn dựa trên docstring có những nhược điểm nhất định. Vấn đề dễ thấy nhất là ở tính tự do của docstring. Không có quy định nào về cách viết docstring.

Thông thường người ta viết docstring theo quy ước như sau:

```
def some_function(argument1):  
    """Summary or Description of the Function  
  
    Parameters:  
  
    argument1 (int): Description of arg1  
  
    Returns:  
  
    int:Returning value  
    """
```

Tuy nhiên bạn có thể viết bất kỳ kiểu nào mình thích.

Điều này gây khó khăn cho các công cụ khi cần xử lý docstring.

Bắt đầu từ Python 3.5 bạn có thể kết hợp thêm **chỉ báo kiểu** (type hint) khi khai báo danh sách tham số của hàm.

Hãy xem cách viết hàm sau:

```
def sum_range(start: int, stop: int, step: int = 1) -> int:  
    '''Tính tổng các số trong khoảng [start, stop)  
  
    start: giá trị đầu khoảng  
  
    stop: giá trị cuối khoảng  
  
    step: khoảng giữa hai giá trị liền nhau  
    ...  
  
    sum = 0  
  
    for i in range(start, stop, step):  
        sum += i  
  
    return sum  
sum_range(1, 100)
```

Lỗi viết này khác biệt ở chỗ sau mỗi tham số sẽ **chỉ dẫn kiểu của tham số đó** (`start: int, stop: int, step: int = 1`), và có cả **chỉ dẫn kiểu kết quả** (`-> int`).

Với cách khai báo này, các công cụ có thể trợ giúp tốt hơn cho người dùng hàm cũng như khi xây dựng hàm.

Dưới đây là ví dụ trong Visual Studio.

```
def sum_range(start: int, stop: int, step: int = 1) -> int:
    '''Tính tổng các số trong khoảng [start, stop)
    start: giá trị đầu khoảng
    stop: giá trị cuối khoảng
    step: khoảng giữa hai giá trị liên nhau
    ...
    sum = 0
    for i in range(start, stop, step):
        sum += i
    return sum

sum_range()
```

function test.sum_range(start: int, stop: int, step: int=1) -> int

Tính tổng các số trong khoảng [start, stop)
start: giá trị đầu khoảng
stop: giá trị cuối khoảng
step: khoảng giữa hai giá trị liên nhau

Bạn nên kết hợp cả docstring và type hint để đạt hiệu quả tốt nhất.

Lưu ý, type hint trong Python chỉ có ý nghĩa trợ giúp, nó không phải là chỉ định kiểu tham số, và cũng không biến Python thành ngôn ngữ định kiểu tĩnh như C#.

Kết luận

Trong phần này chúng ta đã xem xét chi tiết một số vấn đề liên quan đến sử dụng tham số cho hàm, bao gồm tham số bắt buộc, tham số mặc định, tham số biến động. Đây là những tính năng của Python giúp việc xây dựng và sử dụng hàm linh hoạt và đơn giản hơn.

Module và package trong Python

Module và package là những cấp độ quản lý code cao hơn trong Python. Module cho phép lưu trữ hàm (và code khác) trên các file riêng rẽ để sau tái sử dụng trong các file và dự án khác. Package cho phép nhóm các module lại với nhau. Sử dụng module và package giúp bạn dễ dàng quản lý code trong những chương trình lớn cũng như tái sử dụng code về sau.

NỘI DUNG

1. Ví dụ sử dụng module trong Python
2. Module trong Python
3. Sử dụng module với import ...
4. Sử dụng module với from ... import ...
5. Thực thi module
6. Vấn đề đường dẫn khi sử dụng module
7. Sử dụng package trong Python
8. Kết luận

Ví dụ sử dụng module trong Python

Hãy bắt đầu với một ví dụ minh họa. Tạo hai file `my_math.py` và `main.py` trong cùng một thư mục và viết code như sau:

`my_math.py`

```
print('--- start of my_math ---')

PI = 3.14159
E = 2.71828

message = 'My math module'

def sum(start, *numbers):
    '''Calculate the sum of unlimited number

    Params:

        start:int/float, the start sum

        *numbers:int/float, the numbers to sum up

    Return: int/float

    '''
```

```
    for x in numbers:
        start += x
    return start

def sum_range(start, stop, step=1):
    '''    Calculate the sum of intergers

    Params:

        start:int, start range number
        stop:int, stop range number
        step:int, the step between value

    Returns: int
    '''
    sum = 0
    for i in range(start, stop, step):
        sum += i
    return sum

def fact(n):
    '''Calculate the factorial of n

    Params:

        n:int

    Return: int
    '''
    p = 1
    for i in range(1, n + 1):
        p *= i
    return p

print('--- start of my_math ---')
```

main.py

```
import my_math

print(my_math.message)

sum = my_math.sum(0, 1, 2, 3, 4)

sum_range = my_math.sum_range(1, 10)
```

```
fact = my_math.fact(3)

print('sum = ', sum)

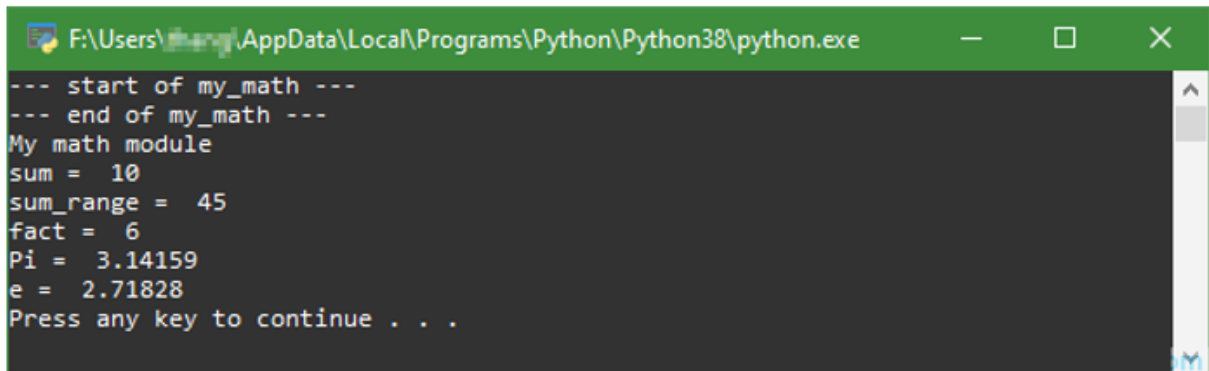
print('sum_range = ', sum_range)

print('fact = ', fact)

print('Pi = ', my_math.PI)

print('e = ', my_math.E)
```

Khi bạn chạy chương trình từ file main.py sẽ có được kết quả như sau:



```
--- start of my_math ---
--- end of my_math ---
My math module
sum = 10
sum_range = 45
fact = 6
Pi = 3.14159
e = 2.71828
Press any key to continue . . .
```

File `my_math.py` là một file script Python hoàn toàn bình thường. Trong file này định nghĩa một số hàm tính toán (`sum`, `sum_range`, `fact`) và một số biến (`PI`, `E`, `greeting`). Điểm khác biệt của file này là chỉ chứa định nghĩa hàm nhưng không sử dụng (gọi) hàm.

Trong file `main.py` chúng ta sử dụng các hàm đã xây dựng từ `my_math.py`.

Để ý lệnh `import my_math` ở đầu file. Đây là lệnh yêu cầu Python tải nội dung của file `my_math.py` khi chạy file script `main.py`.

`my_math` và `main` là hai **module** trong Python.

Module trong Python

Trong Python, bất kỳ file script (file có đuôi py) đều được gọi là một module. Khi bạn tạo một file code Python mới, bạn đang tạo một module. Như vậy, ngay từ những bài học đầu tiên bạn đã xây dựng module Python.

Tên module là tên file (bỏ đi phần mở rộng py). Trong ví dụ ở phần trên, Module `my_math` viết trong file `my_math.py`. Module `main` viết trong file `main.py`.

Mỗi module đều có thể được tải và thực thi bởi Python interpreter. Như ở trên, bạn có thể chạy module `my_math` và `main` riêng rẽ vì chúng đều là các file code Python hợp lệ.

Tuy nhiên, nếu bạn chạy module `my_math` thì sẽ không ra kết quả gì vì trong module này chúng ta chỉ định nghĩa các hàm và biến chứ không hề sử dụng chúng. Đây là điểm khác biệt giữa `my_math` và `main`: trong `my_math` chỉ chứa khai báo (hàm/biến), trong `main` chứa lời gọi.

Như vậy, cách thức xây dựng và sử dụng làm cho các module khác biệt nhau. Có những module được xây dựng ra với vai trò cung cấp thư viện hàm cho module khác sử dụng. Có những module sử dụng hàm định nghĩa trong các module khác.

Để sử dụng hàm/biến/kiểu dữ liệu định nghĩa trong file/module khác, Python sử dụng lệnh `import`. Có hai kiểu viết lệnh `import` khác nhau:

1. `import <module 1>, <module 2>, <module 3> ...`
2. `from <module> import <tên 1>, <tên 2>, ...`

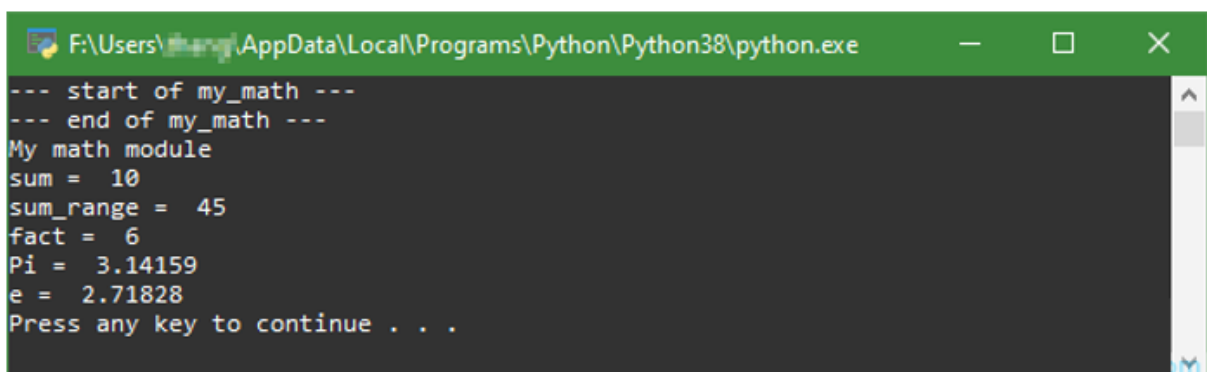
Sử dụng module với import ...

Trong ví dụ minh họa chúng ta đã sử dụng lối viết này:

```
import my_math
```

Cách viết này sẽ tải toàn bộ module `my_math`, gần giống hệt như việc copy nội dung của `my_math.py` đặt vào vị trí của lệnh `import my_math` và chạy chương trình. Điều này có nghĩa là ***tất cả các lệnh trong module đều được thực hiện khi import.***

Bạn có thể thấy điều này khi cập lệnh `print()` viết ở đầu và cuối module `my_math` đều được thực hiện.



```
F:\Users\thang\AppData\Local\Programs\Python\Python38\python.exe
--- start of my_math ---
--- end of my_math ---
My math module
sum = 10
sum_range = 45
fact = 6
Pi = 3.14159
e = 2.71828
Press any key to continue . . .
```

Cách này sẽ **chỉ tải module khi gặp lệnh import lần đầu tiên**. Khi gặp lệnh import lần thứ hai, Python sẽ không tải module nữa. Bạn có thể thấy rất rõ điều này khi cặp lệnh `print()` ở đầu và cuối module `my_math` chỉ được gọi một lần duy nhất trước khi bắt đầu module `main` mặc dù chúng ta viết lệnh `import my_math` hai lần.

Hãy để ý cách chúng ta **gọi hàm và sử dụng biến** (được khai báo trong module `my_math`):

```
sum = my_math.sum(0, 1, 2, 3, 4)
sum_range = my_math.sum_range(1, 10)
fact = my_math.fact(3)
print('Pi = ', my_math.PI)
print('e = ', my_math.E)
print(my_math.message)
```

Tức là, thay vì viết trực tiếp tên hàm/biến, bạn phải cung cấp thêm tên module `my_math`. Cấu trúc chung để sử dụng một hàm/biến trong module khác là `tên_module.tên_hàm()` và `tên_module.tên_biến`.

Bạn có thể **sử dụng alias** trong lệnh import như sau:

```
import my_math as mm
```

Tên `mm` trong lệnh trên được gọi là alias (tên giả) của `my_math`. Khi này thay vì sử dụng tên module `my_math` bạn có thể sử dụng alias `mm` như sau:

```
print(mm.message)
print(mm.sum(1, 2, 3, 4, 5))
print(mm.E)
```

Nghĩa là bạn có thể sử dụng alias thay cho tên module. Bạn có thể **đồng thời sử dụng cả tên module lẫn alias**.

Alias rất tiện lợi khi tên module quá dài hoặc tên module trùng lặp với một hàm/biến có sẵn của Python.

Sử dụng module với `from ... import ...`

Giờ hãy xóa (hoặc comment) toàn bộ code của `main.py` và viết lại code mới như sau:

```
from my_math import sum_range, fact, message
print(message)
print('sum_range = ', sum_range(1, 100, 2))
print('5! = ', fact(5))
```

Ở đây chúng ta gặp cách sử dụng thứ hai của `import`: `from my_math import sum_range, fact, message`.

Khi sử dụng cấu trúc `from .. import` bạn có thể thấy rằng Python vẫn **import toàn bộ code của module** `my_math` (như trường hợp sử dụng `import my_math`). Điều này thể hiện qua việc hai lệnh `print()` ở đầu và cuối module `my_math` đều được thực hiện ở vị trí gọi `from .. import`.

Tuy nhiên, giờ đây bạn **không thể sử dụng được tất cả các hàm/biến của module** `my_math` như trước được nữa. Giờ bạn chỉ có thể sử dụng hàm `sum_range()`, `fact()`, và biến `message`.

Điều đặc biệt là bạn có thể **sử dụng tên ngắn gọn** của các đối tượng này (không cần tên module), giống hệt như khi các đối tượng này được khai báo trực tiếp trong module `main`:

```
print(message)
print('sum_range = ', sum_range(1, 100, 2))
print('5! = ', fact(5))
```

Bạn cũng có thể **chỉ định alias** cho từng tên gọi trong lệnh `from .. import` như sau:

```
from my_math import sum_range as sr, fact as f, message as msg
print(msg)
print('sum_range = ', sr(1, 100, 2))
print('5! = ', f(5))
```

Ở đây khi `import` chúng ta chỉ định alias `sr` cho hàm `sum_range()`, `f` cho hàm `fact()`, `msg` cho biến `message`. Trong code sau đó chúng ta có thể sử dụng alias thay cho tên thật.

Lưu ý, nếu bạn đã đặt alias thì không thể sử dụng tên thật của đối tượng được nữa. Tức là ***không thể đồng thời sử dụng alias và tên thật***.

Bạn cũng có thể sử dụng cách import như sau:

```
from my_math import *
```

Tuy nhiên đây là cách thức ***không được khuyến khích***. Nó dễ dàng làm rối không gian tên của module hiện tại nếu có quá nhiều tên gọi được import.

Về hiệu suất tổng thể thì cả hai cách import gần như là tương đương nhau. Sự khác biệt chỉ nằm ở cách sử dụng các đối tượng từ module. `from...import` tiện lợi hơn nếu bạn chỉ có nhu cầu sử dụng một vài đối tượng từ module. Nó không làm rối không gian tên bằng các hàm không được sử dụng.

Nếu cần sử dụng rất nhiều hàm từ module, bạn nên sử dụng import (kết hợp với alias) sẽ giúp bạn phân biệt rõ hàm của module.

Thực thi module

Khi xây dựng một module bạn có thể phải dự phòng hai tình huống:

1. Module đó được *thực thi trực tiếp* (chạy module đó ở chế độ kịch bản)
2. Module được import vào một module khác, tạm gọi là *thực thi gián tiếp* (khi được import, toàn bộ code của module được thực thi 1 lần).

Mỗi module có một thuộc tính đặc biệt tên là `__name__`. Lưu ý có hai dấu `_` trước và sau `name`. Bạn có thể thử in ra qua lệnh `print(__name__)`.

Khi một module được thực thi trực tiếp, `__name__` sẽ được gán giá trị `'__main__'`. Nếu khi thực thi một module mà `__name__ == '__main__'` thì đây là module chủ của chương trình.

Khi một module được thực thi gián tiếp, `__name__` của nó sẽ có giá trị là tên module (cũng là tên file bỏ đi phần mở rộng `.py`).

Với đặc thù trên, khi xây dựng module trong Python mà cần chạy ở cả hai kiểu (trực tiếp và gián tiếp) người ta thường sử dụng pattern sau đây:

1. Mỗi module sẽ có một hàm `main()` chứa những lệnh cần thực thi theo kiểu trực tiếp;
2. Kiểm tra nếu `__name__ == '__main__'` thì thực thi `main()`.

```
def main():  
    'Thực thi các logic của chương trình'  
    # code cần chạy  
if __name__ == '__main__':  
    main()
```

Khi chạy gián tiếp qua import `__name__` sẽ không thể có giá trị `__main__`, vì vậy logic của `main()` sẽ không thực thi.

Khi một module được chạy gián tiếp qua import, Python cũng thực hiện lưu tạm bản dịch bytecode của nó trong thư mục `__pycache__`. Thư mục `__pycache__` nằm trong cùng thư mục với module được import. Ví dụ, với module `my_math`, khi được import, Python sẽ tạo ra file `__pycache__/my_math.cpython-38.pyc` (nếu bạn sử dụng Python 3.8). Nếu bạn có nhiều phiên bản Python khác nhau, khi chạy với phiên bản nào thì file pyc sẽ có tên tương ứng.

Nhắc lại: khi chạy một script, Python sẽ dịch script này thành bytecode trung gian. Chương trình máy ảo của Python sẽ tiếp tục dịch bytecode thành mã máy để thực thi trên từng hệ điều hành tương ứng.

File bytecode của Python có phần mở rộng là **pyc**.

Khi import một module cũng xảy ra quá trình dịch bytecode như vậy. Tuy nhiên, Python sẽ lưu lại file pyc để lần sau không cần dịch lại. Việc lưu trữ file pyc giúp giảm thời gian tải một chương trình.

Lưu ý, khi trực tiếp thực thi một module Python sẽ không lưu lại file bytecode.

File bytecode không thực thi nhanh hơn. Nó chỉ giảm thời gian tải ứng dụng.

Vì lý do này, khi xây dựng chương trình bằng Python bạn nên tận dụng việc xây dựng và import module, thay vì code trực tiếp trong một module chính lớn. Toàn bộ code nên đưa về các module và để module chính đơn giản nhất có thể.

Vấn đề đường dẫn khi sử dụng module

Bạn có thể ý thấy khi import module bạn không hề chỉ định đường dẫn đến file script. Python thực hiện việc tìm kiếm file module tự động theo các thư mục lưu trong biến sys.path.

Biến sys.path là một danh sách lưu những đường dẫn được đăng ký trong Python. Bạn có thể xem nội dung của sys.path như sau:

```
>>> import sys

>>> sys.path # đây là một list

['.', 'E:\\OneDrive\\ Learn Python\\PythonApps\\HelloPython',
'F:\\Users\\thang\\AppData\\Local\\Programs\\Python\\Python38\\python38.zip',
'F:\\Users\\thang\\AppData\\Local\\Programs\\Python\\Python38\\DLLs',
'F:\\Users\\thang\\AppData\\Local\\Programs\\Python\\Python38\\lib',
'F:\\Users\\thang\\AppData\\Local\\Programs\\Python\\Python38',
'F:\\Users\\thang\\AppData\\Local\\Programs\\Python\\Python38\\lib\\site-packages']

>>> for p in sys.path:

... print(p)

...

.

E:\\OneDrive\\Learn Python\\PythonApps\\HelloPython
F:\\Users\\thang\\AppData\\Local\\Programs\\Python\\Python38\\python38.zip
F:\\Users\\thang\\AppData\\Local\\Programs\\Python\\Python38\\DLLs
F:\\Users\\thang\\AppData\\Local\\Programs\\Python\\Python38\\lib
F:\\Users\\thang\\AppData\\Local\\Programs\\Python\\Python38
F:\\Users\\thang\\AppData\\Local\\Programs\\Python\\Python38\\lib\\site-packages
```

Bạn có thể để ý ngay đường dẫn đầu tiên chính là thư mục làm việc (thư mục hiện hành) nơi bạn đặt file module chính. Ví dụ, nếu bạn chạy module main.py và trong main.py có lệnh import my_math thì Python trước hết sẽ tìm kiếm my_math.py trong thư mục chứa main.py.

Nếu không tìm thấy file tương ứng, Python sẽ lần lượt tìm trong các thư mục còn lại.

Để ý một tình huống khác. Giả sử bạn có module mysql nằm trong file `mysql.py` nằm trong thư mục con `modules\database\` của thư mục hiện hành (tức là đường dẫn tương đối tới file là `modules\database\mysql.py`).

Để import module mysql bạn cần viết như sau: `import modules.database.mysql`. Tức là bạn cần **chỉ định cấu trúc đường dẫn tương đối tới module**. Sự khác biệt ở chỗ **các phần của đường dẫn được phân tách bởi dấu chấm**. Python sẽ tự động ghép đường dẫn này với các đường dẫn lưu trong `sys.path`.

Nếu trong trường hợp các file module của bạn nằm ở một nơi khác (không nằm trong danh sách `sys.path` mặc định), bạn phải có cách chỉ định đường dẫn để Python có thể tìm thấy. Cách đơn giản nhất như sau:

```
import sys
sys.path.append('-- new path --') # thêm thư mục mới vào sys.path
# giờ sẽ import module
import a_module
```

Logic ở đây rất đơn giản. Do `sys.path` chỉ là một list, bạn có thể **tự thêm một đường dẫn mới vào `sys.path`**. Khi đó Python sẽ tìm kiếm cả trong đường dẫn bạn thêm vào.

Ví dụ:

```
>>> import sys
>>> sys.path.append('E:\OneDrive\Learn Python\Modules')
>>> import sample_module
hello world from sample module
>>>
```

Sử dụng package trong Python

Trong Python, một số module trong cùng thư mục (và thư mục con) có thể kết hợp lại để tạo ra một package. Package giúp đơn giản hóa hơn nữa việc sử dụng nhiều module có liên quan.

Hãy cùng thực hiện ví dụ sau:

Tạo thư mục `my_package`. Trong thư mục này tạo hai file `module1.py` và `module2.py` với code như sau:

`module1.py`

```
print('This is the module 1')

def func1():
    print('Module 1 - func 1')
```

`module2.py`

```
print('This is the module 2')

def func2():
    print('Module 2 - func 2')
```

Nếu dừng lại ở đây bạn sẽ có hai module riêng rẽ. Khi sử dụng bạn cần import từng module.

Giờ hãy tạo file `__init__.py` trong thư mục `my_package`. Lưu ý có hai ký tự `_` ở trước và sau `init`. Viết code như sau cho `__init__.py`:

```
from my_package.module1 import *
from my_package.module2 import *

#hoặc cũng có thể import như thế này:
#import my_package.module1
#import my_package.module2
```

Đây là hai lệnh import thông thường mà bạn đã học ở phần trên. Bạn có thể sử dụng kiểu import nào cũng được. Tuy nhiên cần lưu ý về cách viết tên module: `my_package.module1`. Bạn ***cần chỉ định tên thư mục***, nếu không Python sẽ không tìm được module tương ứng.

Tên file `__init__.py` có ý nghĩa đặc thù trong Python. Nếu Python nhìn thấy thư mục nào có file với tên gọi `__init__.py`, nó sẽ tự động coi thư mục này là một package, chứ không còn là thư mục bình thường nữa.

Bên *ngoài* thư mục package hãy tạo module `main.py` và viết code như sau:

```
import my_package as mp
mp.module1.func1()
mp.module2.func2()
from my_package import *
module1.func1();
```

Bạn có thể dễ ý thấy ngay rằng lệnh `import` giờ không hoạt động với từng module nữa mà là với tên thư mục `my_package`.

Tất cả các thư mục chứa file `__init__.py` sẽ trở thành một package. Tên thư mục trở thành tên package và có thể sử dụng cùng lệnh `import` hoặc `from/import`.

Khi `import` một package bạn sẽ đồng thời `import` tất cả các module của nó (được chỉ định trong `__init__.py`).

Nếu sử dụng lệnh `import <tên_package>`, bạn truy cập vào các hàm của từng module qua cú pháp:

```
<tên_package>.<tên_module>.<tên_hàm>() .
```

Nếu sử dụng `from <tên_package> import <tên_hàm>`, bạn có thể sử dụng tên ngắn gọn của hàm như bình thường.

Lưu ý: trong `__init__.py` và trong module sử dụng package **nên dùng cùng một cách import**. Ví dụ, cùng dùng `import` hoặc cùng dùng `from .. import`.

Kết luận

Trong phần này chúng ta đã học chi tiết cách sử dụng module trong Python:

- Mỗi file script nào của Python là một module.
- Trong mỗi module bạn có thể sử dụng hàm và chạy code viết trong một module khác thông qua lệnh `import` hoặc `from .. import`.
- Python tự động tìm kiếm file module dựa trên danh sách thư mục trong `sys.path`.
- Hai lối sử dụng `import` và `from .. import` chủ yếu khác biệt về cách sử dụng tên hàm trong code chứ không khác biệt về hiệu suất.
- Các module trong cùng một thư mục có thể kết hợp thành một package.
- Một thư mục sẽ trở thành package nếu nó chứa file `__init__.py` (với các lệnh `import`).

Giới thiệu chung về class trong Python, constructor

Python là một ngôn ngữ lập trình hướng đối tượng. Mọi phần tử trong chương trình Python thực chất đều là các object từ các class đã xây dựng sẵn: int, float, str, list, function. Python cũng cung cấp khả năng xây dựng các class mới. Nhìn chung các kỹ thuật xây dựng class trong Python tương đối đơn giản hơn so với các ngôn ngữ như C++, C#, Java.

NỘI DUNG

1. Khai báo class trong Python
2. Sử dụng class
3. Các thành phần chính trong class Python
4. Constructor trong Python
5. Đặc điểm của attribute và method trong class Python
6. Kết luận

Khai báo class trong Python

Python là một ngôn ngữ lập trình hướng đối tượng hoàn toàn, nghĩa là mọi phần tử trong chương trình Python đều là object. Một số, một chuỗi, một danh sách,... mà bạn đã biết thực ra đều là object của một class đã được xây dựng sẵn (int, float, str, list,...). Một hàm (xây dựng sẵn hoặc do bạn tự xây dựng bằng từ khóa def) cũng là một object.

Như vậy, khi làm việc với Python, trên thực tế là bạn đã trực tiếp sử dụng class và object. Định nghĩa class và object trong Python hoàn toàn tương tự như trong các ngôn ngữ lập trình khác.

Python cũng cho phép người lập trình tự xây dựng class riêng của mình.

Hãy cùng bắt đầu với một ví dụ:

Tạo module `book.py` và viết code như sau:

```
class Book:
    """A class for e-book"""
    def __init__(self, title: str, authors: str = '', publisher: str = '', year:
int = 2020, edition: int = 1):
        """Hàm tạo của class"""
        self.title = title
```

```
self.authors = authors

self.publisher = publisher

self.year = year

self.edition = edition

def print(self):

    """Print the book infor"""

    print(f"{self.title} by {self.authors}, {self.edition} edition,
{self.publisher}, {self.year}")
```

Đây là code tạo một class mô tả sách, bao gồm các thông tin về tựa sách, tác giả, nhà xuất bản, năm xuất bản, lần tái bản.

Class trong Python được khai báo với từ khóa class theo cấu trúc như sau:

```
class tên_class:

    '''docstring'''

    # class suite
```

Lệnh khai báo class cũng là một lệnh phức hợp với 1 clause. Header bao gồm từ khóa class và tên class, phần suite chứa các lệnh khai báo các thành phần của class.

Tên class phải tuân thủ theo quy tắc đặt định danh chung của Python. Ngoài ra, tên class được đặt theo quy ước PascalCase (viết hoa chữ cái đầu mỗi từ). Tuy nhiên, nếu để ý, các class xây dựng sẵn của Python lại chỉ đặt tên viết thường.

Ngay dưới header là docstring của class. Docstring cung cấp tài liệu hỗ trợ cho việc sử dụng class, tương tự như vai trò của docstring trong hàm. Thông thường docstring của class đơn giản hơn vì chỉ tóm lược mục đích sử dụng của class.

Sau phần docstring là khai báo các thành phần khác của class. Class trong Python có nhiều thành phần khác nhau. Phần tiếp theo của tài liệu sẽ giới thiệu sơ lược về các thành phần này.

Sử dụng class

Với class Book xây dựng như trên, bạn có thể tạo object như sau:

```
from book import Book

b1 = Book('Lập trình hướng đối tượng với Python', 'Nhật Linh', 'Nhà xuất bản Trẻ',
2022, 2)

b1.print()

b2 = Book(title = 'Nhập môn lập trình Python', authors= 'Nhật linh', publisher=
'Nhà xuất bản Trẻ')

b2.print()

b3 = Book('A new book')

b3.print()
```

Để thấy rằng, lệnh tạo object không khác biệt gì so với lời gọi hàm thông thường.

Như vậy, lệnh tạo object đơn giản nhất là sử dụng tên class và cặp dấu (), tương tự như một lời gọi hàm.

Phụ thuộc vào hàm tạo của class, lời gọi lệnh tạo object có thể phức tạp hơn với nhiều tham số, giống như lời gọi hàm thông thường. Hàm tạo sẽ được trình bày chi tiết ở phần sau.

Bạn có thể xây dựng và sử dụng class trong cùng một module. Tuy nhiên, thông thường class nên được xây dựng trong module riêng.

Nếu class xây dựng trong một module/package khác, bạn cần import nó trước khi sử dụng bằng một trong hai cách đã học.

Ví dụ, nếu class Book xây dựng trong module book, bạn có thể sử dụng `from book import Book` hoặc `import book`.

```
from book import Book
```

hoặc

```
import book
```

Nếu sử dụng `from book import Book` bạn có thể sử dụng trực tiếp tên class Book.

Khi sử dụng `import book`, bạn phải sử dụng thêm tên module:

```
b4 = book.Book('Lập trình Python')
```

Các thành phần chính trong class Python

Mặc dù bạn có thể khai báo một class hoàn toàn không chứa thành phần nào, nhưng class như vậy không có giá trị. Class trong Python thường chứa những thành phần sau:

- Constructor (hàm tạo)
- Các attribute (biến)
- Các method (phương thức)
- Các property (thuộc tính)

Hàm tạo (**constructor**) là hàm được gọi trong quá trình tạo object của class. Khác với C# hay Java, hàm tạo trong Python không phải là hàm chạy đầu tiên khi tạo object, và cũng không phải là hàm chịu trách nhiệm tạo object. Hàm tạo trong Python có tác dụng tạo các thuộc tính thể hiện (instance attribute). Chúng ta sẽ học cách làm việc với hàm tạo ở phần sau của tài liệu này.

Attribute (biến/thuộc tính) là thành phần chứa dữ liệu trong class/object. Python phân biệt hai loại attribute: instance attribute và class attribute.

- **Instance attribute** là những biến chứa trạng thái của một object cụ thể và đặc trưng cho object. Trong Python, instance attribute được khai báo và gán giá trị trong hàm tạo.
- **Class attribute** là những biến chứa giá trị đặc trưng cho cả class chứ không đặc trưng cho một object cụ thể. Class attribute có cùng giá trị trong tất cả các object của class. Class attribute có thể được sử dụng, ví dụ, để theo dõi số lượng object của class được tạo ra.

Phương thức (**method**) là thành phần xử lý dữ liệu trong Python. Phương thức thực chất là các loại hàm khác nhau được khai báo trong class. Python phân biệt instance method, class method và static method.

- **Instance method** là những hàm xử lý trạng thái của object. Instance method gắn liền với object và sử dụng các instance attribute (dữ liệu gắn với từng object).
- **Class method** là những hàm xử lý thông tin của class và gắn liền với class. Class method chuyên xử lý class attribute.

- **Static method** là loại phương thức đặc biệt không sử dụng bất kỳ thông tin gì của class hay object mặc dù nằm trong class.

Các thành phần của class trong Python có nhiều điểm tương tự với các ngôn ngữ như C# hay Java. Tuy nhiên, tên gọi trong Python có phần hơi khác. Instance attribute tương tự như biến thành viên trong C# hay Java. Class attribute tương tự như biến thành viên tĩnh trong C#. Class method và static method tương tự như static method trong C#.

Cách gọi tên trong Python rất hệ thống, phân biệt rõ ràng đặc trưng của instance và đặc trưng của class.

Constructor trong Python

Hãy xem lại hàm `__init__()` mà chúng ta đã xây dựng trong ví dụ đầu tiên:

```
def __init__(self, title: str, authors: str = '', publisher: str = '', year:
int = 2020, edition: int = 1):

    """Hàm tạo của class"""

    self.title = title

    self.authors = authors

    self.publisher = publisher

    self.year = year

    self.edition = edition

    self.__private = True

    Book.count += 1
```

`__init__()` là một hàm đặc biệt trong Python: hàm tạo (constructor).

Về mặt hình thức `__init__()` hoàn toàn tương tự như một lệnh khai báo hàm trong Python. Hàm `__init__()` ở trên nhận các tham số `title`, `authors`, `publisher`, `year` và `edition`. Chúng ta cũng sử dụng kỹ thuật chỉ báo kiểu (type hint) và cung cấp giá trị mặc định cho các tham số.

Constructor trong Python bắt buộc phải có tên là `__init__` và phải có ít nhất một tham số, thường đặt tên là `self`. Nếu có nhiều tham số, `self` bắt buộc phải là tham số đầu tiên.

Tên gọi tham số `self` được đặt theo quy ước của Python chứ không bắt buộc, có thể đặt bất kỳ tên gọi nào khác. Những bạn có xuất phát điểm là C++ hay C# thường có xu hướng đặt là `this`.

Constructor và Initializer

Nói một cách chính xác, `__init__()` không phải là constructor theo nghĩa đen của khái niệm này trong lập trình hướng đối tượng. Hàm `__init__()` là một initializer – hàm chịu trách nhiệm khởi tạo các giá trị cho object. Initializer không chịu trách nhiệm khởi tạo object.

Trong Python, hàm `__init__()` không chịu trách nhiệm tạo ra object của class. Python sử dụng một 'magic method' có tên gọi là `__new__()` để tạo object của mỗi class.

Magic method là một số phương thức được Python tự động tạo cùng với class và được Python gọi tự động nhằm thực hiện những công việc đặc biệt.

`__new__()` mới thực sự là constructor của Python class.

Ví dụ, khi gõ lệnh tạo object `b = book()` thì Python sẽ tự động chạy hàm `__new__()` đầu tiên. Kết quả của hàm `__new__()` là object của class `book`. Sau đó Python tiếp tục tự động chạy `__init__()`. Object do `__new__()` tạo ra được truyền sang cho `__init__()` thông qua tham số đầu tiên (*self*) trong danh sách.

Vì lý do này, bạn có thể đặt bất kỳ tên gì cho `self` cũng được nhưng phải để nó ở đầu danh sách tham số.

Vai trò quan trọng hàng đầu của hàm tạo trong Python là tạo và gán giá trị cho instance attribute. Tất cả các tham số còn lại trong danh sách tham số của `__init__()` cung cấp giá trị để tạo ra instance attribute cho object.

Với hàm tạo như trên, bạn có thể tạo object của class `Book` bằng những cách sau:

```
b1 = Book('Lập trình hướng đối tượng với Python', 'Nhật Linh', 'Nhà xuất bản Trẻ', 2022, 2)

b2 = Book(title = 'Nhập môn lập trình Python', authors= 'Nhật linh', publisher= 'Nhà xuất bản Trẻ')

b3 = Book('A new book')
```

Dễ thấy rằng, lệnh tạo object bằng hàm tạo không khác biệt gì so với lời gọi hàm thông thường.

Đặc điểm của attribute và method trong class Python

Attribute và method trong Python có điểm khác biệt với biến thành viên và phương thức trong các ngôn ngữ C++/C#/Java.

Hãy xem ví dụ sau:

```
class Book:
    """A class for e-book"""
b = Book()
b.title = 'Python programming'
b.authors = 'Donald Trump'
b.year = 2020
print(b.title, b.authors, b.year) # kết quả là 'Python programming Donald Trumo
2020'
b2 = Book()
print(b2.title) # lỗi, không có attribute title trong object b2
```

Bạn có thể thấy rất nhiều điều lạ ở đây. Dễ thấy nhất là class Book hoàn toàn trống trơn, chỉ có mỗi docstring. Tuy nhiên sau khi tạo object b, bạn lại có thể dùng phép toán truy xuất phần tử (dot notation) `b.title`, `b.authors`, `b.year`, gán giá trị cho chúng và sau sử dụng lại chúng trong hàm `print()`. Rõ ràng bạn không hề tạo title, authors hay year trong khai báo class Book.

Khi bạn tạo object b2 và thử truy xuất giá trị title (`b2.title`) thì lại gặp lỗi “không tìm thấy attribute title trong object b2”.

title, authors, year được gọi là những **attribute** của object b (nhưng không phải là attribute của b2).

Như vậy, trong Python, attribute là những biến có thể chứa giá trị đặc trưng cho một object. Nó được tạo hoàn toàn độc lập với khai báo class (không cần chỉ định trong khai báo class). Một cách chính xác hơn, biến này được gọi là instance attribute (do liên quan đến object).

Giờ hãy xem một ví dụ khác:

```
class Person:
    pass

putin = Person()

def greeting(msg:str):
    print(msg)

putin.say_hello = greeting

putin.say_hello('Hello world from Python method') # in ra dòng 'Hello world from
Python method'

trump = Person()

trump.say_hello('Welcome to heaven!') # lệnh này sẽ bị lỗi 'không tìm thấy hàm
say_hello'
```

Trong ví dụ này:

1. Chúng ta khai báo một class trống rỗng `Person`, không có bất kỳ thành viên nào.
2. Tiếp theo chúng ta tạo object `putin` của class `Person`.
3. Chúng ta khai báo một hàm độc lập `greeting()` có thể in ra dòng thông báo.
4. Điểm rất đặc biệt là lệnh `putin.say_hello = greeting`. Đây là lệnh tạo ra một phương thức (method) trong object `putin` và gán cho nó hàm `greeting()`. Tức là chúng ta 'gán ghép' `greeting()` với `say_hello()` của `putin`.
5. Sau đó, bạn có thể sử dụng hàm/phương thức `greeting` từ object `putin` nhưng với tên gọi mới `say_hello`. Hàm `say_hello()` được gọi là một **method** của `putin`.
6. Tuy nhiên, nếu bạn tạo object `trump` từ class `Person`, object này lại không có phương thức `say_hello`.

Như vậy, giống như attribute, method trong class Python cũng không bắt buộc phải khai báo trong thân class. Method độc lập với class và object.

Kết luận

Trong phần này chúng ta bắt đầu với lập trình hướng đối tượng trong Python:

- Nhìn chung trong Python vẫn sử dụng các khái niệm cơ bản tương tự như trong các ngôn ngữ lập trình hướng đối tượng khác, mặc dù tên gọi có chút khác biệt.
- Python phân biệt thành phần dữ liệu (attribute) và thành phần xử lý (method) trong class. Đồng thời Python cũng phân biệt rõ các thành viên liên quan đến class và thành viên liên quan đến object. Từ đây dẫn đến sự phân biệt instance attribute/class attribute, instance method/class method.
- Python có điểm đặc biệt trong khởi tạo object và cách xây dựng hàm tạo, có thể gây khó khăn cho những bạn xuất phát từ C++/Java/C#.

Dữ liệu của class Python: instance và class attribute

Mỗi class thường chứa hai loại thành viên quan trọng: thành phần chứa dữ liệu và thành phần xử lý dữ liệu. Trong Python, thành phần chứa dữ liệu được gọi là attribute. Có thể xem attribute của Python tương tự như biến của class trong các ngôn ngữ như C++/Java hay C#.

Python phân biệt hai loại attribute: instance attribute gắn với object và class attribute gắn với chính class.

NỘI DUNG

1. Ví dụ về sử dụng attribute trong Python
2. Instance attribute
 - 2.1. Khai báo instance attribute
 - 2.2. Truy xuất instance attribute
3. Class attribute
4. Public, protected, private trong Python
5. Kết luận

Ví dụ về sử dụng attribute trong Python

Chúng ta bắt đầu với một ví dụ.

Tạo module book.py và viết code như sau:

Ví dụ 1 (book.py)

```
class Book:
    """A class for e-book"""
b = Book()
b.title = 'Python programming'
b.authors = 'Donald Trump'
b.year = 2020
print(b.title, b.authors, b.year) # kết quả là 'Python programming Donald Trumo
2020'
b2 = Book()
print(b2.title) # lỗi, không có attribute title trong object b2
```

Bạn có thể thấy rất nhiều điều lạ ở đây. Dễ thấy nhất là class `Book` hoàn toàn trống trơn. Trong class này chỉ có mỗi docstring. Tuy nhiên sau khi tạo object `b`, bạn lại có thể dùng phép toán truy xuất phần tử (dot notation) `b.title`, `b.authors`, `b.year`, gán giá trị cho chúng và sau sử dụng lại chúng trong hàm `print()`. Rõ ràng bạn không hề tạo `title`, `authors` hay `year` trong khai báo `Book`.

Khi bạn tạo object `b2` và thử truy xuất giá trị `title` (`b2.title`) thì lại gặp lỗi `AttributeError: 'Book' object has no attribute 'title'`

`title`, `authors`, `year` được gọi là những **attribute** của object `b` (nhưng không phải là attribute của `b2`).

Như vậy, trong Python, attribute là những biến có thể chứa giá trị đặc trưng cho một object. Nó được tạo hoàn toàn độc lập với khai báo class (không cần chỉ định trong khai báo class). Một cách chính xác hơn, biến này được gọi là instance attribute (do liên quan đến object).

Giờ hãy cập nhật class `Book` như sau:

Ví dụ 2 (book.py)

```
class Book:
    """A class for e-book"""
    def __init__(self,
                    title: str,
                    authors: str = '',
                    publisher: str = '',
                    year: int = 2020,
                    edition: int = 1):
        """Hàm tạo của class"""
        self.title = title
        self.authors = authors
        self.publisher = publisher
        self.year = year
        self.edition = edition

    def to_string(self, brief = True):
```

```
        """Get the infor and make a formatted string"""

        if brief:

            return f"{self.title} by {self.authors}"

        else:

            return f"{self.title} by {self.authors}, {self.edition} edition, {self.publisher}, {self.year}"

    def print(this, brief = False):

        """Print the book infor"""

        print(this.to_string(brief))

# sử dụng class Book

b1 = Book('Tự học lập trình Python')

b1.authors = 'Nhật Linh'

b1.publisher = 'Nhà xuất bản Trẻ'

b1.year = 2021

b1.print()

b2 = Book('Python programming', 'Trump D.', 'The White house', 2020)

print(b2.title, b2.authors)
```

Trong trường hợp này chúng ta tạo attribute bên trong hàm tạo của class. Bạn có thể thấy rằng giờ cả b1 và b2 đều có chung tổ hợp các attribute như title, authors, publisher, year và edition. Giờ những biến này được gọi là những **instance attribute** của class Book.

Instance attribute

Trong Python, biến thành viên của class được gọi là instance attribute. Đây là các giá trị đặc trưng cho từng object.

Ví dụ, class Book xác định rằng, tất cả sách được đặc trưng bởi tổ hợp giá trị của title, authors, publisher, year và edition. Vậy, cuốn sách thứ nhất có giá trị tổ hợp là 'Tự học lập trình Python' của tác giả 'Nhật Linh', do 'Nhà xuất bản Trẻ' xuất bản năm 2020 và là lần xuất bản đầu tiên. Tổ hợp giá trị này đặc trưng cho riêng 1 cuốn sách (object). Cuốn sách khác sẽ có tổ hợp giá trị khác.

Khai báo instance attribute

Trong class Book, các lệnh tạo (và gán giá trị) biến thành viên của class phải viết trong hàm tạo như sau:

```
def __init__(self, title: str, authors: str = '', publisher: str = '', year: int = 2020, edition: int = 1):  
    """Hàm tạo của class"""  
    self.title = title  
    self.authors = authors  
    self.publisher = publisher  
    self.year = year  
    self.edition = edition
```

Nếu bạn xuất phát từ C++, Java hay C# sẽ thấy cách tạo biến thành viên trong Python hơi khác biệt:

- Biến thành viên trong Python được tạo ra trong hàm tạo chứ không viết trong thân class. Biến được khai báo trong thân class lại thuộc về nhóm class attribute (chúng ta sẽ tìm hiểu sau).
- Biến thành viên được khai báo cùng với tham số self sử dụng phép toán truy xuất thành viên, giống như là các biến này đã có sẵn và bạn chỉ việc gán dữ liệu.

Trong Python, biến self (hay bất kỳ tên tham số nào) đứng đầu trong danh sách tham số của __init__ sẽ trỏ tới object vừa tạo (bởi magic method __new__()). Phép toán truy xuất thành viên (dấu chấm) trên object sử dụng một định danh mới và phép gán sẽ tự động tạo ra một biến thành viên.

Tên biến thành viên được đặt theo quy tắc đặt định danh chung của Python, cũng như theo quy ước đặt tên của biến (cục bộ và toàn cục).

Như vậy, `self.title = 'A new book'` trong `__init__()` sẽ tạo ra biến thành viên title với giá trị 'A new book'.

Tất cả các tham số còn lại của `__init__()` chính là để cung cấp giá trị ban đầu cho các biến thành viên.

Như trong Ví dụ 1 bạn đã thấy, attribute không nhất thiết phải khai báo trong hàm tạo. Bạn có thể khai báo attribute sau khi tạo object. Chỉ có điều, khi này attribute đó chỉ tồn tại trên object cụ thể đó. Nếu bạn tạo ra object mới, nó sẽ không có attribute như object trước đó.

Nếu bạn tạo attribute trong constructor, tất cả object tạo ra từ class sẽ có chung tổ hợp attribute. Điều này phù hợp với khái niệm class/object trong lập trình hướng đối tượng. Do vậy, bạn luôn nên ***tạo instance attribute trong constructor***.

Truy xuất instance attribute

Với các instance attribute tạo ra như trên, bạn có thể truy xuất nó qua tên object trong code ở ngoài class như sau:

```
b3 = Book('')
b3.title = 'Tự học lập trình Python'
b3.authors = 'Nhật Linh'
b3.publisher = 'Nhà xuất bản Trẻ'
b3.year = 2021
b3.edition = 2
```

Việc truy xuất này là hai chiều, nghĩa là có thể gán giá trị hoặc đọc giá trị.

Đối với code ở bên trong class, cách truy cập là tương tự. Hãy xem phương thức `to_string()`:

```
def to_string(self, brief = True):
    """Get the infor and make a formatted string"""
    if brief:
        return f"{self.title} by {self.authors}"
    else:
        return f"{self.title} by {self.authors}, {self.edition} edition, {self.publisher}, {self.year}"
```

Tạm thời chúng ta chưa tìm hiểu kỹ về phương thức này.

Hãy nhìn cách sử dụng biến `title`, `authors`, `edition`, `publisher` và `year` thông qua tham số `self`: `self.title`, `self.authors`, `self.edition`, `self.publisher`, `self.year`.

Bạn không được viết trực tiếp tên attribute mà bắt buộc phải thông qua biến `self`. Nó không có gì khác biệt so với khi truy xuất từ tên object bên ngoài class, cả về hình thức và bản chất. Tham số `self` thực chất là một object kiểu `Book` được Python tự động truyền khi gọi phương thức `to_string()`.

Class attribute

Hãy hình dung yêu cầu sau: khi xây dựng class `Book`, làm thế nào để theo dõi số lượng object đã được tạo ra? Logic đơn giản nhất là tạo ra một biến đếm. Mỗi khi tạo một object mới thì tăng giá trị của biến đếm. Nếu biến đếm và việc tăng giá trị của biến đếm nằm ngoài class thì rất đơn giản. Nhưng nếu chúng ta cần tích hợp logic này vào chính class thì làm như thế nào?

Để ý thấy rằng, một biến đếm cho mục đích trên không thể phụ thuộc vào từng object. Nói cách khác, tất cả các object đều phải sử dụng chung một biến đếm. Để tăng giá trị khi tạo object, phép cộng phải được thực hiện trong constructor.

Python cung cấp một công cụ cho những mục đích tương tự: class attribute. Hãy thay đổi class `Book` như sau:

```
class Book:
    """A class for e-book"""
    count = 0

    def __init__(self, title: str, authors: str = '', publisher: str = '', year:
int = 2020, edition: int = 1):
        """Hàm tạo của class"""
        self.title = title
        self.authors = authors
        self.publisher = publisher
        self.year = year
        self.edition = edition
        self.__private = True
        Book.count += 1
```

Để ý dòng 4 có lệnh khởi tạo biến `count = 0`, và dòng 18 có phép cộng `Book.count += 1`. Trong class Python, `count` là một class attribute.

Class attribute là một biến gắn liền với chính class, có thể được truy xuất từ các object và có giá trị chung cho tất cả các object.

Sự khác biệt giữa class attribute và instance attribute nằm ở chỗ: instance attribute gắn với từng object cụ thể và thể hiện trạng thái riêng của từng object; class attribute gắn với chính class và đặc trưng cho class, hoặc đặc trưng chung cho mọi object.

Với biến `count` xây dựng như trên bạn có thể sử dụng nó thông qua tên class: `Book.count`. Bạn sử dụng lối viết này dù ở trong thân class hay bên ngoài class.

```
b1 = Book('Lập trình hướng đối tượng với Python', 'Nhật Linh', 'Tự học ict', 2022, 2)
print(Book.count)

b2 = Book(title = 'Nhập môn lập trình Python', authors= 'Nhật linh', publisher= 'Nhà xuất bản Trẻ')
print(Book.count)

b3 = Book('')
print(b3.count)
```

Do đặc điểm của class attribute là sử dụng chung trong các object khác nhau, bạn cũng có thể truy xuất giá trị của `count` thông qua tên object: `b3.count`. Tuy nhiên, khi truy xuất qua tên object bạn không thay đổi được giá trị của class attribute. Nói chính xác hơn, thay đổi giá trị của class attribute qua tên object sẽ không lưu lại được.

Public, protected, private trong Python

Trong các ngôn ngữ hướng đối tượng truyền thống như C++, Java hay C#, mỗi thành viên thuộc về một trong các mức truy cập:

- (1) public: tự do truy cập, không có giới hạn gì
- (2) protected: chỉ class con và trong nội bộ class mới có thể truy cập
- (3) private: giới hạn truy cập trong nội bộ class.

Trong Python không có khái niệm về kiểm soát truy cập các thành viên như vậy. Hoặc cũng có thể nói rằng mặc định mọi thành viên của class trong Python đều là public. Nghĩa là code ngoài class có thể tự do truy cập các thành viên này.

Để mô phỏng lại hiệu quả của việc kiểm soát truy cập, Python sử dụng loại kỹ thuật có tên gọi là xáo trộn tên (**name mangling**).

Kỹ thuật này quy ước rằng:

- Nếu muốn biến chỉ được sử dụng trong nội bộ class và các class con (mức truy cập là protected), tên biến cần bắt đầu là `_` (một dấu gạch chân);
- Nếu muốn biến chỉ được sử dụng trong nội bộ class (mức truy cập private), tên biến cần bắt đầu là `__` (hai dấu gạch chân).

Ví dụ, bạn có thể khai báo biến protected và private trong constructor của lớp Book như sau:

```
self.__private = True # private instance attribute
self._protected = False # protected instance attribute
```

Thực ra, loại kỹ thuật này không làm thay đổi được việc truy cập thành viên của class. Nó đơn thuần là chỉ báo để lập trình viên và IDE biết ý định sử dụng của thành viên đó.

Ví dụ, trong class Book như trên, thực tế biến `__private` không hề trở thành private. Một số IDE sẽ trợ giúp che đi biến này trong phần nhắc code. Tuy nhiên bạn vẫn có thể truy xuất nó như bình thường: `b3.__private = False`.

Kết luận

Trong phần này chúng ta đã xem xét cách sử dụng attribute – thành phần lưu trữ dữ liệu trong class Python.

Python phân biệt hai loại attribute: instance attribute gắn với object, class attribute gắn với chính class. Instance attribute tương tự như biến thành viên của C++/Java/C#, còn class attribute tương tự như biến static của các ngôn ngữ này.

Điểm khác biệt lớn trong Python là instance attribute khai báo trong hàm tạo, còn class attribute khai báo trực tiếp trong class.

Đồng thời, các attribute trong Python không phân biệt mức truy cập (public, protected, private) mà sử dụng kỹ thuật name mangling đóng vai trò chỉ báo.

Các loại phương thức (method) trong Python class

Phương thức là những thành viên chịu trách nhiệm xử lý dữ liệu trong class Python. Có 3 loại phương thức trong class: instance method (phương thức thành viên), class method (phương thức class) và static method (phương thức tĩnh). Ngoài ra trong Python còn có property (thuộc tính) – một dạng kết hợp đặc biệt giúp truy xuất dữ liệu.

NỘI DUNG

1. Giới thiệu chung về method trong Python
2. Instance method trong Python
3. Tham số self và truy xuất thành viên class
4. Class method trong Python
5. Static method trong Python
6. Kết luận

Giới thiệu chung về method trong Python

Method trong Python là những hàm được gắn với object hoặc với class. Trong Python, khai báo method có thể thực hiện độc lập với class! Đây là điều mà các bạn xuất phát từ C++/Java/C# rất khó hình dung.

Hãy cùng thực hiện một ví dụ nhỏ:

Ví dụ 1 (person.py)

```
class Person:
    pass

putin = Person()

def greeting(msg:str):
    print(msg)

putin.say_hello = greeting

putin.say_hello('Hello world from Python method') # in ra dòng 'Hello world from
Python method'

trump = Person()

trump.say_hello('Welcome to heaven!') # lệnh này sẽ bị lỗi 'không tìm thấy hàm
say_hello'
```

Trong ví dụ trên:

1. Chúng ta khai báo một class trống rỗng `Person`, không có bất kỳ thành viên nào.
2. Tiếp theo chúng ta tạo object `putin` của class `Person`.
3. Chúng ta khai báo một hàm độc lập `greeting()` có thể in ra dòng thông báo.
4. Điểm rất đặc biệt là lệnh `putin.say_hello = greeting`. Đây là lệnh tạo ra một phương thức (method) trong object `putin` và gán cho nó hàm `greeting()`. Tức là chúng ta 'gán ghép' `greeting()` với `say_hello()` của `putin`.
5. Sau đó, bạn có thể sử dụng hàm/phương thức `greeting` từ object `putin` nhưng với tên gọi mới `say_hello`. Hàm `say_hello()` được gọi là một **method** của `putin`.
6. Tuy nhiên, nếu bạn tạo object `trump` từ class `Person`, object này lại không có phương thức `say_hello`.

Như vậy, giống như attribute, method trong class Python cũng không bắt buộc phải khai báo trong thân class. Method độc lập với class và object.

Tuy nhiên, việc ***khai báo method trong thân class*** giúp cho tất cả các object có chung danh sách method. Điều này phù hợp với đặc điểm của lập trình hướng đối tượng.

Tùy thuộc vào việc method truy xuất được dữ liệu của object cụ thể, truy xuất dữ liệu chung của class, hay hoàn toàn không cần dữ liệu gì, Python phân biệt ra instance method, class method và static method.

Phương thức `say_hello` ở trên có thể được xếp vào loại static method. Và *không nên lạm dụng cách thức này!*

Instance method trong Python

Instance method trong Python là những phương thức có khả năng truy xuất trạng thái của object. Nhắc lại: *trạng thái của object trong Python được lưu trữ trong các instance attribute (biến thành viên)*.

Instance method của Python tương đương với phương thức thành viên trong C# hay Java.

Hãy xem ví dụ sau:

```
class Person:

    def __init__(self, name: str, age: int):

        self.name = name

        self.age = age

    def print(self, format = True):

        """In thông tin ra console

        format: có định dạng thông tin hay không

        """

        if not format:

            print(self.name, self.age)

        else:

            print(f'{self.name}, {self.age} years old')

putin = Person('Putin Vladimir', 60)

putin.print()
```

Phương thức `print()` trong class `Person` là một **instance method**. Về mặt hình thức khai báo, instance method không có gì khác biệt so với một khai báo hàm thông thường (ngoài class).

Khai báo phương thức thành viên sử dụng từ khóa `def` là một lệnh phức hợp gồm một clause. Phần header bao gồm từ khóa `def`, tên phương thức và danh sách tham số.

Tên phương thức được đặt theo quy tắc đặt định danh chung của Python. Ngoài ra tên phương thức cũng được đặt theo quy ước tương tự như với tên hàm: tất cả chữ cái viết thường; nếu có nhiều từ thì phân tách bằng dấu gạch chân.

Danh sách tham số hoàn toàn tương tự như của hàm. Bạn có thể sử dụng tham số bắt buộc, tham số mặc định, tham số biến động và chỉ báo kiểu tham số như khai báo hàm.

Tuy nhiên, danh sách tham số của phương thức bắt buộc phải có ít nhất một tham số, thường đặt tên là `self`. Nếu có nhiều tham số, `self` phải là tham số đầu tiên trong danh sách.

Sau header là **docstring** với vai trò tương tự như docstring của hàm. Giống như hàm, bạn nên kết hợp chỉ báo kiểu (type hint) với docstring.

Trong suite ở thân phương thức bạn có thể ***trả về kết quả với lệnh return***.

Nói tóm lại, khai báo một phương thức hoàn toàn giống như khai báo một hàm thông thường. Tuy nhiên, khi khai báo phương thức có hai điều khác biệt cần lưu ý:

1. Tham số `self` trong danh sách tham số;
2. Cách truy xuất biến thành viên.

Tham số `self` và truy xuất thành viên class

Bạn hẳn đã thấy trong các phương thức của Python, dù là constructor hay instance method đều có ***biến `self` nằm ở đầu danh sách tham số***.

Tại sao hàm tạo và các phương thức thành viên của class trong Python lại phải có biến `self`?

Nếu bạn đã từng làm việc với class trong các ngôn ngữ như C++, Java hay C#, bạn có thể hình dung class như một ngôi nhà, và các phương thức là những người giúp việc sống trong ngôi nhà đó. Những người giúp việc này thực hiện các công việc được giao. Do sống ở ngay trong nhà, họ có thể truy xuất trực tiếp tất cả các những trong nhà nếu được phép. Do vậy, lệnh truy xuất thành viên của class không cần bao gồm tên object.

Trong Python, bạn cần hình dung những người giúp việc sống ở một nơi khác. Do vậy, khi yêu cầu họ làm việc gì, bạn cũng đồng thời phải chỉ rõ ngôi nhà họ sẽ làm việc. Việc chỉ định ngôi nhà tương tự với truyền chính object qua biến `self`.

Việc truy xuất thành viên của class (dù là viết trong code của class hay viết bên ngoài class) đều phải bao gồm tên object.

Như trong ví dụ trên, khi bạn tạo object `putin` và gọi hàm `putin.print()`, object `putin` được truyền cho phương thức `print` qua biến `self`. Tuy nhiên, bạn không cần trực tiếp truyền object `putin` cho `print` do Python làm việc này tự động.

Do vậy, biến `self` là bắt buộc với mọi phương thức thành viên của Python class.

Python đưa ra quy định là tham số đặc biệt `self` phải được viết ở đầu danh sách tham số. Tên `self` chỉ là một quy ước, có thể dùng bất kỳ tên biến hợp lệ nào.

Hãy nhìn cách sử dụng biến `name` và `age`:

```
if not format:
    print(self.name, self.age)
else:
    print(f'{self.name}, {self.age} years old')
```

Bạn không được viết trực tiếp tên attribute mà bắt buộc phải thông qua biến `self`: `self.name`, `self.age`. Nó không có gì khác biệt so với khi truy xuất từ tên object bên ngoài class, cả về hình thức và bản chất.

Nếu bạn vẫn còn thấy khó hiểu với `self`, hãy thử đoạn code sau:

```
putin = Person('Putin Vladimir', 60)
Person.print(putin) # cho kết quả 'Putin, 60 years old'
Person.print(putin, False) # cho kết quả 'Putin 60'
```

Giờ đây bạn không gọi hàm `print` từ object `putin` nữa mà gọi từ class `Person`. Kết quả không có gì khác so với bạn gọi từ object. Sự khác biệt ở chỗ bạn phải tự mình truyền object `putin` cho hàm `print`.

Nhắc lại, trong Python, bạn có thể hình dung method là một hàm thông thường được bạn “vô tình” viết vào trong khai báo của một class. Để hàm có thể truy xuất attribute của object, bạn phải truyền object cho hàm. Tham số `self` (hoặc bất kỳ tên gọi nào) ở đầu danh sách tham số chính là để phục vụ yêu cầu này.

Nếu bạn gọi hàm từ một object, Python sẽ hỗ trợ bạn tự động truyền object đó vào tham số `self`. Nếu bạn gọi theo kiểu khác (như gọi từ class), bạn phải tự mình truyền object cho `self`.

Class method trong Python

Trong bài học trước bạn đã làm quen với class attribute – loại biến chứa giá trị đặc trưng cho class và gắn với class, thay vì gắn với object. Một ví dụ đã được đưa ra để minh họa là đếm số lượng object của class đã được khởi tạo trong chương trình.

Class method cũng có cùng ý tưởng với class attribute. Class method là những phương thức đặc trưng cho class và gắn liền với class, thay vì gắn với object. Class method có nhiệm vụ xử lý dữ liệu lưu trong class attribute.

Hãy xem ví dụ sau:

```
class Person:
    count = 0

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age
        Person.count += 1

    @classmethod
    def show_count(cls):
        """Trả lại số object trong chương trình"""
        print(f"Hiện có {cls.count} object Person trong chương trình")

putin = Person('Putin Vladimir', 60)
trump = Person('Trump Donald', 60)
Person.show_count()
```

Ở đây chúng ta tạo class attribute count và tăng giá trị của count mỗi khi tạo object mới.

Chúng ta cũng xây dựng một phương thức riêng show_count() chuyên để in thông báo về giá trị của count. Hãy để ý cách khai báo phương thức này có điểm đặc thù: @classmethod.

@classmethod trong Python được gọi là một **decorator**. Decorator là một loại hàm đặc biệt có thể nhận một hàm khác làm tham số để bổ sung tính năng cho hàm tham số đó. Chúng ta sẽ tìm hiểu chi tiết về decorator trong một nội dung khác.

Để tạo ra class method từ một method thông thường, chúng ta sử dụng decorator `@classmethod`.

Class method cũng bắt buộc phải có một biến đặc biệt trong danh sách tham số: biến `cls` (viết tắt của class). Biến này có vai trò tương tự như biến `self` của instance method. Điểm khác biệt nằm ở chỗ biến `cls` chứa thông tin về chính class.

Trong ví dụ trên, `count` là một class attribute – chứa thông tin về chính class. Do vậy, có thể truy xuất `count` qua biến `cls`. Trên thực tế, bạn có thể hình dung truy xuất qua biến `cls` cũng chính là truy xuất qua tên class. Tức là `cls.count` hoàn toàn tương đương với `Person.count`.

Trong code, bạn có thể gọi class method qua tên class hoặc qua tên object đều được. Như trong ví dụ trên, bạn có thể gọi `Person.show_count()`, `putin.show_count()`, `trump.show_count()` đều được và trả về cùng một kết quả.

Để tránh lẫn lộn, bạn nên gọi class method qua tên class. Điều này cũng tương tự như truy xuất class attribute nên qua tên class, mặc dù có thể truy xuất được qua tên object.

Static method trong Python

Trong Python cũng hỗ trợ khái niệm static method. Static method là loại phương thức hoàn toàn tự do, không có gì ràng buộc về dữ liệu với class hay object. Chúng được đặt trong class chỉ vì một lí do logic nào đó.

Để so sánh: instance method có ràng buộc về dữ liệu với instance attribute; class method có ràng buộc về dữ liệu với class method.

Cả static method và class method trong Python đều tương đương với static method trong Java hay C#.

Để tạo ra static method trong class Python, chúng ta sử dụng **decorator** `@staticmethod`.

Cùng điều chỉnh class `Person` như sau:

```
class Person:
    count = 0
    def __init__(self, name: str, age: int):
```

```

        self.name = name

        self.age = age

        Person.count += 1

def print(self, format = True):
    """In thông tin ra console

    format: có định dạng thông tin hay không

    """

    if not format:

        print(self.name, self.age)

    else:

        print(f'{self.name}, {self.age} years old')

@classmethod
def show_count(cls):
    """Trả lại số object trong chương trình"""

    print(f"Hiện có {cls.count} object Person trong chương trình")

@staticmethod
def calculate_birth_year(age : int) -> int:

    import datetime as dt

    year = dt.datetime.now().year

    return year - age

# sử dụng phương thức calculate_birth_year như sau:
my_birth_year = Person.calculate_birth_year(37) # kết quả 1983 (năm 2020)

```

Hãy để ý phương thức `calculate_birth_year` được đánh dấu với decorator `@staticmethod`. Phương thức này dùng để tính ra năm sinh dựa trên giá trị tuổi. Phương thức này không sử dụng thông tin của class (biến `count`), cũng không sử dụng thông tin của object (name và age). Nó hoạt động giống như một phương thức độc lập. Khác biệt là nó được khai báo bên trong class `Person`.

`calculate_birth_year` khai báo như trên là một static method.

Cách sử dụng của static method giống như một class method. Bạn gọi nó thông qua tên class và cung cấp tham số cần thiết.

```
my_birth_year = Person.calculate_birth_year(37)
```

Decorator `@staticmethod` biến một phương thức khai báo trong class trở thành một static method. Phương thức `calculate_birth_year` không hề sử dụng bất kỳ thông tin nào của class `Person` cũng như của object tạo từ class này. Nó chỉ có liên hệ về logic với `Person`: giúp tính năm sinh từ giá trị tuổi.

Như vậy, bạn sử dụng static method nếu chúng có mối liên hệ logic nào đó với class nhưng không cần các thông tin của class cũng như của object. Static method cũng giúp nhóm các hàm xử lý vào cùng một class để tiện sử dụng ở client code.

Kết luận

Trong phần này chúng ta đã làm quen với method – thành phần xử lý dữ liệu trong class. Python có ba loại method: instance method gắn với object, class method gắn với class, và static method.

Nếu nhớ lại attribute – thành phần lưu trữ dữ liệu trong class – bạn sẽ thấy, method và attribute trong Python tạo thành một hệ thống rất dễ nhớ:

- instance attribute chứa dữ liệu (trạng thái) riêng cho từng object và được xử lý bởi instance method.
- class attribute chứa dữ liệu chung cho mọi object và đặc trưng cho class – được xử lý bởi class method.
- static method nằm trong class nhưng không sử dụng thông tin của class hay object.

Property (thuộc tính) trong Python

Property là một loại thành viên đặc biệt trong class Python cho phép truy xuất và kiểm soát truy xuất một (instance) attribute cụ thể. Property rất quan trọng và được khuyến khích sử dụng khi xây dựng các class chuyên để chứa dữ liệu (như domain class trong các ứng dụng quản lý). Python cung cấp một số cách khác nhau để khai báo property.

NỘI DUNG

1. Mô hình getter/setter trong Python class
2. Property trong Python, hàm property()
3. Tạo property với decorator
4. Kết luận

Mô hình getter/setter trong Python class

Trong class Python, mặc định mọi biến thành viên (instance attribute) đều là public. Do đó, mọi biến thành viên trong Python đều được truy xuất tự do.

Điều này làm xuất hiện một vấn đề: kiểm soát dữ liệu. Ví dụ, tuổi của con người không thể là một số âm. Tương tự vậy, tên của con người không thể là những cụm ký tự bất kỳ được.

Để thực hiện việc kiểm soát xuất nhập dữ liệu cho object, các ngôn ngữ lập trình hướng đối tượng thường sử dụng mô hình getter/setter. Hãy thực hiện mô hình này trên class Person:

```
class Person:
    def __init__(self, fname: str = '', lname: str = '', age: int = 18):
        self.__fname = fname
        self.__lname = lname
        self.__age = age
    def set_age(self, age: int):
        if age > 0:
            self.__age = age
    def get_age(self):
        return self.__age
```

```
def get_lname(self):
    return self.__lname

def set_lname(self, lname: str):
    if lname.isalpha():
        self.__lname = lname

def get_fname(self):
    return self.__fname

def set_fname(self, fname: str):
    if fname.isalpha():
        self.__fname = fname

def get_name(self):
    return f'{self.__fname} {self.__lname}'

putin = Person()
putin.set_fname('Putin')
putin.set_lname('Vladimir')
putin.set_age(66)
print(putin.get_name())
```

Theo mô hình này, biến cần kiểm soát dữ liệu sẽ đặt mức truy cập là private. Tức là code ngoài class không nên thay đổi giá trị của biến. Trong Python bạn sử dụng name mangling `__age`, `__fname`, `__lname` (hai dấu gạch chân) cho biến private khi khai báo biến thành viên đó trong hàm tạo `__init__()`.

Ứng với mỗi biến private bạn cần xây dựng hai phương thức get/set để xuất/nhập dữ liệu. Ví dụ, với biến thành viên `__age`, bạn xây dựng phương thức `set_age` để nhập dữ liệu cho `__age`, và `get_age` để trả lại dữ liệu chứa trong `__age`. Cặp phương thức get/set như vậy được gọi chung là getter và setter. Tương tự, chúng ta xây dựng `get_fname()/set_fname()` cho `__fname`, `get_lname()/set_lname()` cho `__lname`.

Trong getter và setter, tùy vào yêu cầu kiểm soát bạn có thể thực hiện logic riêng. Ví dụ, trong `set_age()`, bạn chỉ gán giá trị mới cho `__age` nếu giá trị đó lớn hơn 0; trong `set_fname()` và `set_lname()` – điều kiện gán là chuỗi chỉ chứa ký tự chữ cái (kiểm tra `isalpha()` trả lại kết quả True).

Property trong Python, hàm property()

Mô hình getter/setter như trên hoạt động tốt. Nhiều ngôn ngữ khuyến khích vận dụng và phát triển mô hình đó. Ví dụ, trong C# cung cấp một số cú pháp tắt để nhanh chóng tạo ra các cặp getter/setter và gọi loại cú pháp đó là C# property.

Trong Python bạn có thể tiếp tục sử dụng mô hình getter/setter như trên. Tuy nhiên, khi sử dụng mô hình này, trong code của bạn chứa quá nhiều lời gọi hàm khiến code nhìn thiếu tự nhiên và khó đọc.

Python đưa ra giải pháp riêng. Hãy điều chỉnh code như sau:

```
class Person:

    def __init__(self, fname: str = '', lname: str = '', age: int = 18):

        self.__fname = fname

        self.__lname = lname

        self.__age = age

    def set_age(self, age: int):

        if age > 0:

            self.__age = age

    def get_age(self):

        return self.__age

    def get_lname(self):

        return self.__lname

    def set_lname(self, lname: str):

        if lname.isalpha():

            self.__lname = lname

    def get_fname(self):

        return self.__fname

    def set_fname(self, fname: str):

        if fname.isalpha():

            self.__fname = fname

    def get_name(self):

        return f'{self.__fname} {self.__lname}'
```

```
first_name = property(get_fname, set_fname)
last_name = property(get_lname, set_lname)
full_name = property(get_name)
age = property(get_age, set_age)
def print(self, format = True):
    if not format:
        print(self.name, self.age)
    else:
        print(f'{self.full_name}, {self.age} years old')
putin = Person()
putin.first_name = 'Putin'
putin.last_name = 'Vladimir'
putin.age = 66
print(putin.full_name, putin.age)
```

Hãy để ý nhóm lệnh đặc biệt trong class Person:

```
first_name = property(get_fname, set_fname)
last_name = property(get_lname, set_lname)
full_name = property(get_name)
age = property(get_age, set_age)
```

Khi tồn tại nhóm lệnh này, bạn có thể viết code sử dụng class như sau:

```
putin = Person()
putin.first_name = 'Putin'
putin.last_name = 'Vladimir'
putin.age = 66
print(putin.full_name, putin.age)
```

Khi này, `first_name`, `last_name` và `age` trở thành các **property** trong class Python.

Trong class Python, `property` là một dạng giao diện tới các instance attribute để thực hiện xuất/nhập dữ liệu qua bộ getter/setter. Mỗi `property` cung cấp một cách thức tự nhiên để nhập xuất dữ liệu cho một

instance attribute qua phép gán và phép toán truy xuất phần tử thông thường. Property hoàn toàn che đi lời gọi hàm getter/setter thực tế.

Như vậy, khi sử dụng property `first_name`, bạn có thể dùng cách viết tự nhiên `putin.first_name` giống như một biến thành viên thông thường để truy xuất dữ liệu từ biến private `__fname`. Phép gán giá trị cho `first_name` sẽ chuyển thành lời gọi hàm `set_fname()`, lệnh đọc giá trị `first_name` sẽ chuyển thành lời gọi hàm `get_fname()`.

Python cung cấp hai cách để tạo property: sử dụng hàm `property()` và sử dụng `@property` decorator.

Ở trên chúng ta vừa sử dụng hàm `property()`.

Hàm `property()` nhận 3 tham số tương ứng với tên hàm **getter**, **setter** và **deleter**. Kết quả của lời gọi hàm `property` chính là một biến mà bạn có thể sử dụng làm property tương ứng của attribute.

Getter và setter thì bạn đã biết. Còn deleter là hàm được gọi tương ứng với lệnh `del` của Python. Ví dụ, khi gọi lệnh `del putin.first_name` thì sẽ gọi tới deleter tương ứng. Trên thực tế deleter ít được sử dụng hơn.

Nếu thiếu setter, property trở thành chỉ đọc (read-only).

Tạo property với decorator

Một phương pháp đơn giản hơn để tạo property là sử dụng decorator `@property`.

Hãy thay đổi code của Person như sau:

```
class Person:

    def __init__(self, fname: str = '', lname: str = '', age: int = 18):

        self.__fname = fname

        self.__lname = lname

        self.__age = age

    @property
    def age(self):

        return self.__age

    @age.setter
```



```
def age(self, age: int):
    if age > 0:
        self.__age = age

@property
def first_name(self):
    return self.__fname

@first_name.setter
def first_name(self, fname: str):
    if fname.isalpha():
        self.__fname = fname

@property
def last_name(self):
    return self.__lname

@last_name.setter
def last_name(self, lname: str):
    if lname.isalpha():
        self.__lname = lname

@property
def full_name(self):
    return f'{self.__fname} {self.__lname}'

def print(self, format = True):
    if not format:
        print(self.name, self.age)
    else:
        print(f'{self.full_name}, {self.age} years old')

putin = Person()
putin.first_name = 'Putin'
putin.last_name = 'Vladimir'
putin.age = 66
print(putin.full_name, putin.age)
```

Bạn có thể thấy code của `Person` giờ tương đối khác biệt và ngắn gọn hơn. Hãy để ý các dòng code đã được thay đổi. Đây là cách để tạo ra property sử dụng decorator `@property`.

Lấy ví dụ:

```
@property
def age(self):
    return self.__age

@age.setter
def age(self, age: int):
    if age > 0:
        self.__age = age
```

Bạn để ý mấy vấn đề sau:

- Cả getter và setter (và cả deleter) giờ sử dụng chung một tên, và đó cũng là tên của property. Ví dụ nếu bạn cần tạo `age` property để truy xuất giá trị cho attribute `__age` thì cần tạo getter, setter và deleter với cùng một tên `age`.
- Phương thức getter cần đánh dấu với `@property` decorator.
- Phương thức setter cần đánh dấu với cấu trúc `@<tên_property>.setter`, phương thức deleter cần đánh dấu với cấu trúc `@<tên_property>.deleter`. Ví dụ, với `age` property thì setter phải đánh dấu `@age.setter`, deleter phải đánh dấu `@age.deleter`.
- Nếu chỉ có getter, property trở thành read-only. Ví dụ, `full_name` là một property chỉ đọc.

```
@property
def full_name(self):
    return f'{self.__fname} {self.__lname}'
```

Rõ ràng, cấu trúc khai báo property này đơn giản ngắn gọn và dễ đọc hơn.

Kết luận

Trong phần này chúng ta đã làm quen với property trong Python với các ý chính sau:

- Property là loại thành phần tương đối đặc biệt cho phép kết hợp giữa attribute và method để kiểm soát truy xuất dữ liệu cho attribute.
- Python cung cấp hai cú pháp để xây dựng property: sử dụng hàm `property()` và sử dụng decorator `@property`. Trong đó, cú pháp decorator đơn giản gọn nhẹ và dễ đọc hơn.
- Dù sử dụng phương pháp nào, cần lưu ý rằng property thực chất chỉ là giao diện để gọi tới các phương thức getter, setter và deleter cho attribute theo cách thức tiện lợi hơn.

Kế thừa (inheritance) trong Python

Kế thừa là một trong những nguyên lý cơ bản của lập trình hướng đối tượng. Kế thừa là cơ chế tái sử dụng code mà tất cả các ngôn ngữ lập trình hướng đối tượng đều thực thi.

Python cũng hỗ trợ kế thừa. Cơ chế thực hiện kế thừa trong Python có điểm hơi khác so với một số ngôn ngữ khác.

NỘI DUNG

1. Ví dụ về kế thừa trong Python
2. Kế thừa trong Python
3. Ghi đè trong Python
4. Name mangling và kế thừa
5. Kế thừa và đa hình trong Python
6. Kết luận

Ví dụ về kế thừa trong Python

Kế thừa (inheritance) là một công cụ rất mạnh trong lập trình hướng đối tượng cho phép tạo ra các class mới từ một class sẵn có. Qua đó có thể tái sử dụng code của class đã có. Kế thừa giúp giảm thiểu việc lặp code giữa các class.

Hãy cùng thực hiện ví dụ sau đây.

inheritance.py

```
class Person:
    count = 0
    def __init__(self, fname='', lname='', age=18):
        self.fname = fname
        self.lname = lname
        self.age = age
        Person.count += 1
    def print(self):
        print(f'{self.fname} {self.lname} ({self.age} years old)')
    @property
```

```
def full_name(self):
    return f'{self.fname} {self.lname}'

@classmethod
def print_count(cls):
    print(f'{cls.count} objects created')

@staticmethod
def birth_year(age: int) -> int:
    from datetime import datetime as dt
    year = dt.now().year
    return year - age

class Student(Person):
    def __init__(self, fname='', lname='', age=18, group='', specialization=''):
        super().__init__(fname, lname, age)
        self.group = group
        self.specialization = specialization
    def print(self):
        super().print()
        print(f'Group {self.group}/{self.specialization}')
    @property
    def academic_info(self):
        return f'Group {self.group}, Specialization of {self.specialization}'

if __name__ == '__main__':
    trump = Student('Donald', 'Trump', 22, '051311', 'Computer science')
    trump.print()
    print(trump.full_name)
    print(Student.count)
    Student.print_count()
    print(Student.birth_year(37))
    print(trump.academic_info)
```

Kết quả sau khi chạy chương trình như sau:

```
Donald Trump (22 years old)
Group 051311/Computer science
Donald Trump
1
1 objects created
1983
Group 051311, Specialization of Computer science
```

Trong ví dụ trên chúng ta đã xây dựng hai class: Person và Student.

Trong class Person chúng ta tạo các instance attribute (`fname`, `lname`, `age`) trong hàm tạo, một class attribute (`count`), một class method (`print_count`), một static method (`birth_year`) và một property (`full_name`).

Chúng ta xây dựng class thứ hai Student **kế thừa** từ Person. Để ý rằng trong lớp Student chúng ta chỉ định nghĩa hàm tạo và phương thức `print()`.

Biến `trump` được tạo ra từ lớp Student. Tuy nhiên, khi sử dụng biến `trump` chúng ta lại truy xuất được tới các thành viên của lớp Person.

Chúng ta gọi quan hệ giữa Student và Person là quan hệ kế thừa. Trong đó, Student kế thừa Person, hoặc Person sinh ra Student. Lớp Person gọi là **lớp cha** hoặc **lớp cơ sở**. Lớp Student gọi là **lớp con** hoặc **lớp dẫn xuất**.

Kế thừa trong Python

Cú pháp khai báo kế thừa trong Python như sau:

```
class <lớp con>(<lớp cha>):
```

Như trong ví dụ ở phần trên, để khai báo lớp Student kế thừa lớp Person, chúng ta viết phần header như sau:

```
class Student(Person):
```

Trong Python, lớp con kế thừa tất cả mọi thứ từ lớp cha. “Kế thừa” ở đây cần hiểu là tất cả những gì được định nghĩa ở lớp cha sẽ đều có thể sử dụng qua tên class con hoặc qua object của lớp con.

Trong ví dụ trên, ở lớp cha chúng ta xây dựng đầy đủ những thành phần thường gặp ở class Python: constructor (`__init__`), instance attribute (`fname`, `lname`, `age`), class attribute (`count`), property (`full_name`), class method (`print_count`), static method (`birth_year`).

Bạn có thể sử dụng tất cả các thành phần trên của `Person` qua object của `Student` hoặc qua chính class `Student`:

```
trump = Student('Donald', 'Trump', 22, '051311', 'Computer science')
trump.print()

print(trump.full_name)
print(Student.count)
Student.print_count()
print(Student.birth_year(37))
print(trump.academic_info)
```

Python cho phép một class con kế thừa nhiều class cha cùng lúc. Hiện tượng này được gọi là đa kế thừa (**multiple inheritance**).

Đa kế thừa cũng được hỗ trợ trong một số ngôn ngữ như C++. Tuy nhiên, nhìn chung người ta không khuyến khích sử dụng đa kế thừa vì nó có thể dẫn đến những kết quả khó dự đoán. Vì vậy, các ngôn ngữ như C# hay Java không hỗ trợ đa kế thừa.

Mọi class trong Python đều kế thừa từ một class “tổ tông” chung – lớp `object`. Tuy nhiên, khi xây dựng class mới bạn không cần chỉ định lớp cha `object`. Python sẽ làm việc này tự động.

Ghi đề trong Python

Class con khi kế thừa từ class cha sẽ có tất cả những gì định nghĩa trong class cha. Tuy nhiên, đôi khi những gì định nghĩa trong class lại không hoàn toàn phù hợp với class con.

Ví dụ, trong lớp `Person` ở ví dụ trên có định nghĩa phương thức `print()`. Lớp `Student` thừa kế `Person` cũng sẽ thừa kế `print()`. Tuy nhiên, `print()` của `Person` chỉ in ra các thông tin riêng của `Person`, bao gồm `fname`, `lname` và `age`. Trong khi đó `Student` tự định nghĩa thêm hai attribute `group` và `specialization`. Nghĩa là `print()` mà `Student` kế thừa từ `Person` không hoàn toàn phù hợp.

Từ những tình huống tương tự dẫn đến nhu cầu viết lại một phương thức vốn đã được định nghĩa sẵn ở lớp cha. Hoặc cũng có thể diễn đạt theo cách khác: định nghĩa một phương thức mới trong lớp con có cùng tên và tham số với phương thức được kế thừa từ lớp cha (dĩ nhiên, phần thân khác nhau). Trong kế thừa, hiện tượng này được gọi là ghi đè phương thức (**method overriding**).

Trong ví dụ của chúng ta, lớp con Student ghi đè phương thức `print()` của lớp cha Person. Trong lớp con Student định nghĩa một phương thức `print()` có cùng header như phương thức `print()` của Person nhưng với phần suite khác biệt:

```
def print(self):  
    super().print()  
    print(f'Group {self.group}/{self.specialization}')
```

Khi gặp lệnh gọi `print()` từ một object của Student, Python sẽ gọi phương thức `print()` mới của Student.

Python sử dụng cơ chế gọi là Method Resolution Order (MRO) để xác định xem cần phải thực hiện phương thức nào trong kế thừa.

Để ý trong phương thức `print()` mới có lời gọi `super().print()`. Đây là cách Python cho phép gọi tới hàm `print()` cũ của Person. Ở đây chúng ta tận dụng hàm `print()` của lớp Person để in ra các thông tin vốn có sẵn ở lớp Person. Sau đó in bổ sung những thông tin riêng của Student.

Một tình huống ghi đè thường gặp khác liên quan đến constructor.

Do constructor `__init__` được lớp con thừa kế, nếu bạn không có nhu cầu định nghĩa thêm attribute riêng cho class con thì bạn thậm chí không cần xây dựng constructor ở lớp con nữa. Python tự động gọi constructor của lớp cha khi tạo object của lớp con.

Tuy vậy, thông thường lớp con thường định nghĩa thêm attribute của riêng mình. Do vậy, trong lớp con cũng thường định nghĩa constructor của riêng mình. Hiện tượng này được gọi là **ghi đè hàm tạo** (constructor overriding).

Khi ghi đè hàm tạo, bạn sẽ có nhu cầu gọi tới hàm tạo của lớp cha trước khi thêm attribute riêng của lớp con. Chính xác hơn, khi ghi đè hàm tạo, bạn ***bắt buộc phải gọi hàm tạo của lớp cha*** trong hàm tạo của lớp con qua phương thức `super()`:

```
class Student(Person):  
    def __init__(self, fname='', lname='', age=18, group='', specialization=''):  
        super().__init__(fname, lname, age)  
        self.group = group  
        self.specialization = specialization
```

Hàm `super()` trả lại một proxy object – một object tạm của class cha – giúp truy xuất phương thức của class cha từ class con. Hàm `super()` giúp tránh sử dụng trực tiếp tên class cha và làm việc với đa kế thừa.

Khi gộp hàm tạo ghi đè ở lớp con, Python sẽ không tự động gọi hàm tạo của lớp cha nữa mà sẽ chỉ gọi hàm tạo của lớp con khi tạo đối tượng từ lớp con. Nếu bạn không gọi hàm tạo của lớp cha trong hàm tạo mới của lớp con, Python sẽ không thể tạo các attribute cần thiết cho lớp cha và sẽ dẫn đến lỗi.

Name mangling và kế thừa

Trên thực tế không phải tất cả mọi thứ trong lớp cha đều được lớp con kế thừa. Nếu bạn sử dụng name mangling, các thành viên private (trong tên gọi có hai dấu gạch chân) sẽ không được kế thừa.

Name mangling là cơ chế của Python để mô phỏng việc kiểm soát truy cập các thành viên của class. Python quy ước:

- (1) nếu thành viên cần giới hạn truy cập trong phạm vi class thì tên gọi cần bắt đầu bằng hai dấu gạch chân. Ví dụ `__private`.
- (2) nếu thành viên cần truy cập giới hạn trong phạm vi của class hoặc ở class con thì tên gọi cần bắt đầu bằng một dấu gạch chân. Ví dụ `_protected`.

Python hoặc IDE sẽ sinh lỗi hoặc cảnh báo nếu khi sử dụng attribute bạn vi phạm các quy ước trên.

Để thử nghiệm, hãy bổ sung thêm hai attribute sau vào hàm tạo của `Person`:

```
self._protected = True  
self.__private = True
```

Giờ hãy thử nghiệm các lệnh gọi sau:

```
trump = Student('Donald', 'Trump', 22, '051311', 'Computer science')  
print(trump._protected) # True  
print(trump.__private) # lỗi
```

Trong nhóm 3 lệnh trên, lệnh thứ 3 sẽ gây lỗi: `AttributeError: 'Student' object has no attribute '__private'`. Nghĩa là attribute `__private` không được Student kế thừa.

Trong khi đó truy xuất `trump._protected` không gây lỗi. Nghĩa là `_protected` được Student kế thừa.

Lưu ý, việc truy xuất `_protected` như trên sẽ bị IDE như PyCharm cảnh báo **'Access to a protected member of a class'**. Bạn không nên truy xuất protected attribute ở ngoài class cha hoặc ngoài class con.

Như vậy name mangling có khả năng kiểm soát truy cập trong kế thừa.

Kế thừa và đa hình trong Python

Hãy cùng thực hiện ví dụ sau:

polymorphism.py

```
class Person:  
    def __init__(self, firstname: str, lastname: str, birthyear: int):  
        self.first_name = firstname  
        self.last_name = lastname  
        self.birth_year = birthyear  
    def display(self):  
        print(f"{self.first_name} {self.last_name}, born {self.birth_year}")  
class Student(Person):  
    def __init__(self, firstname: str, lastname: str, birthyear: int, group: str):  
        super().__init__(firstname, lastname, birthyear)  
        self.group = group  
    def display(self):
```

```

        print(f"{self.first_name} {self.last_name}, born {self.birth_year}, group {self.group}")

class Teacher:

    def __init__(self, firstname: str, lastname: str, birthyear: int, specialization: str):

        self.first_name = firstname

        self.last_name = lastname

        self.birth_year = birthyear

        self.specialization = specialization

    def display(self):

        print(f"{self.first_name} {self.last_name}, born {self.birth_year}, mastered in {self.specialization}")

def show_info(p):

    p.display()

if __name__ == '__main__':

    putin = Person('Vladimir', 'Putin', 1955)

    trump = Student('Donald', 'Trump', 1965, 'TWH_2017_2021')

    obama = Teacher('Barack', 'Obama', 1975, 'Computer Science')

    show_info(putin)

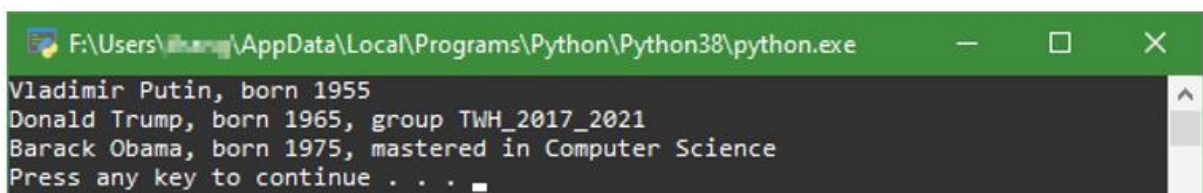
    show_info(trump)

    show_info(obama)

```

Trong ví dụ này chúng ta xây dựng 3 class, Person, Student, Teacher. Trong đó Student kế thừa Person. Cả 3 class này đều có phương thức display(self).

Hãy để ý phương thức show_info(p). Bạn không cần quan tâm p có kiểu gì, chỉ cần p chứa phương thức display() là có thể sử dụng được trong show_info().



```

F:\Users\...AppData\Local\Programs\Python\Python38\python.exe
Vladimir Putin, born 1955
Donald Trump, born 1965, group TWH_2017_2021
Barack Obama, born 1975, mastered in Computer Science
Press any key to continue . . .

```

Việc gọi p.display() như vậy liên quan đến cơ chế đa hình (**polymorphism**).

Trong lập trình hướng đối tượng, kế thừa và đa hình là hai nguyên lý khác nhau.

Đa hình thiết lập mối quan hệ **là** (tiếng Anh gọi là **is-a**) giữa kiểu cơ sở và kiểu dẫn xuất. Ví dụ, nếu chúng ta có lớp cơ sở Person và lớp dẫn xuất Student thì một object của Student cũng là object của Person, kiểu Student cũng là kiểu Person. Nói theo ngôn ngữ thông thường thì sinh viên cũng **là** người. Một anh sinh viên chắc chắn **là** một người.

Trong khi đó, **kế thừa** liên quan đến tái sử dụng code. Lớp con thừa hưởng code của lớp cha.

Một cách nói khác, đa hình liên quan tới quan hệ về ngữ nghĩa, còn kế thừa liên quan tới cú pháp.

Trong các ngôn ngữ như C++, C#, Java, hai khái niệm này hầu như được đồng nhất, thể hiện ở chỗ:

1. class con thừa hưởng các thành viên của class cha (kế thừa, tái sử dụng code);
2. một object thuộc kiểu con có thể gán cho biến thuộc kiểu cha, tức là kiểu cơ sở có thể dùng để thay thế cho kiểu dẫn xuất (đa hình);
3. một phương thức xử lý được object kiểu cha thì sẽ xử lý được object kiểu con.

Tổ hợp kế thừa + đa hình cho phép các ngôn ngữ lập trình hướng đối tượng xử lý object ở dạng tổng quát.

Trong Python điều này vẫn đúng. Tuy nhiên, Python còn mềm dẻo hơn nữa. Python sử dụng nguyên lý có tên gọi là **duck typing** để thực hiện cơ chế đa hình.

Nguyên lý duck typing (nguyên lý con vịt): nếu một con vật nhìn giống như con vịt, có thể đi và bơi như con vịt, thì nó nhất định là một con vịt. Như vậy, theo nguyên lý này, một con thiên nga nhỏ cũng có thể xem là một con vịt!

Mặc dù nghe khá buồn cười nhưng nó thể hiện đặc điểm nhận dạng object của Python: đa hình trong Python đạt được không cần liên quan đến kế thừa. Đa hình trong Python liên quan đến các thành viên của class. Nếu

hai object có các thành viên tương tự nhau thì chúng được xem là thuộc cùng một kiểu. Do vậy chúng có thể sử dụng trong cùng một hàm/phương thức.

Như vậy, theo nguyên lý trên, nếu bạn xây dựng class Person, Student, Teacher với cùng các thành viên, object của hai class này được xem là tương tự nhau và có thể truyền cho cùng một phương thức xử lý. Student và Teacher không cần có bất kỳ quan hệ gì về kế thừa.

Kết luận

Trong phần này chúng ta đã xem xét chi tiết về kế thừa trong Python, bao gồm các vấn đề sau:

- Trong Python, class con có thể kế thừa tất cả mọi thành viên của class cha, ngoại trừ thành viên private (sử dụng name mangling).
- Python hỗ trợ đa kế thừa. Đây là điều khác biệt với C# hay Java nhưng tương tự với C++.
- Python cũng hỗ trợ ghi đè phương thức, bao gồm cả hàm tạo.
- Trong class con có thể truy xuất phương thức của lớp cha qua một proxy tạo ra bởi hàm super().

Decorator (hàm trang trí) trong Python

Bạn đã từng gặp loại cú pháp đặc biệt `@classmethod`, `@staticmethod`, `@property` khi xây dựng class. Trong Python, loại cú pháp này là một cách sử dụng của decorator. **Decorator** là một hàm nhận hàm khác làm tham số để mở rộng khả năng của hàm tham số (nhưng không cần điều chỉnh thân hàm tham số). Phần này sẽ trình bày chi tiết các vấn đề liên quan đến việc xây dựng và sử dụng decorator trong Python.

NỘI DUNG

1. Decorator là gì?
2. Hàm bậc cao
3. Hàm lồng nhau và closure
4. Xây dựng hàm decorator trong Python
5. Một số vấn đề khác với decorator
6. Kết luận

Decorator là gì?

Decorator là một mẫu thiết kế (design pattern) cho phép mở rộng chức năng của một đối tượng mà không cần can thiệp điều chỉnh code của nó. Đây là một trong số các mẫu thiết kế cổ điển.

Trong Python mẫu thiết kế decorator được hỗ trợ và sử dụng rộng rãi.

Khi tìm hiểu về phương thức (method) và thuộc tính (property) trong class Python bạn đã gặp một số cách viết lạ như `@classmethod`, `@staticmethod`, `@property`. Chúng được gọi là những **decorator** (hàm trang trí).

Decorator trong Python là những hàm nhận hàm khác làm tham số để mở rộng khả năng của hàm tham số (nhưng không cần điều chỉnh thân hàm tham số).

Hàm trang trí trong Python chính là một vận dụng của mẫu thiết kế decorator.

Vậy tại sao lại cần decorator trong Python?

Trong quá trình xây dựng hàm bạn có thể sẽ gặp những yêu cầu chung lặp đi lặp lại. Lấy ví dụ, trong một số trường hợp bạn sẽ muốn ghi chú (log) lại quá trình thực hiện của một hàm để theo dõi và debug. Bạn có thể cần khởi tạo một số tài nguyên trước khi hàm có thể hoạt động. Bạn có thể muốn dọn dẹp object sau khi một hàm kết thúc. Bạn có thể muốn đo thời gian thực hiện của một hàm bằng cách tính khoảng thời gian từ lúc bắt đầu đến lúc kết thúc thực hiện hàm.

Dĩ nhiên, bạn có thể viết hết các logic trên vào trong hàm! Tuy nhiên, hãy nghĩ tới tình huống bạn có rất nhiều hàm phải xử lý tương tự nhau. Liệu bạn có muốn copy một đoạn code tới tất cả các hàm cần xử lý?

Khi đó, decorator sẽ giúp bạn bổ sung các tính năng cần thiết vào một hàm mà không cần viết thêm code cho hàm đó. Bạn có thể giữ hàm gốc với các tính năng cơ bản của nó.

Về kỹ thuật, decorator trong Python thực chất là một hàm nhận một hàm khác làm tham số và trả lại kết quả cũng là một hàm. Python có một số hỗ trợ giúp đơn giản hóa việc sử dụng hàm decorator.

Hàm bậc cao

Trong Python, bạn có thể **truyền một hàm làm tham số** của một hàm khác.

Lưu ý: truyền hàm làm tham số không giống truyền lời gọi hàm. Khi truyền hàm làm tham số, bạn chỉ truyền tên hàm. Trong thân hàm chính có thể gọi hàm tham số như một hàm thông thường.

Hãy xem ví dụ sau đây:

high_order_function.py

```
def hello(name):
    print(f'Hello, {name}. Welcome to heaven!')

def hi(name):
    print(f'Hi, {name}. Welcome to hell!')

def greeting(name, func):
    print('Inside high order function:')
    func(name)
```

```
print('---')  
  
greeting('Donald Trump', hello)  
  
greeting('Vladimir Putin', hi)
```

Trong ví dụ trên, hi và hello là hai hàm bình thường. Các hàm này nhận một chuỗi tên người và in ra lời chào.

Trong khi đó, `greeting()` lại là một hàm bậc cao: nó nhận hai tham số (`name` và `func`). Qua hình thức header, `greeting()` không khác gì so với `hi()` hay `hello()`. Tuy nhiên, khi nhìn vào thân hàm, bạn để ý dòng lệnh `func(name)`. Đây là lời gọi hàm chứ không phải là lệnh truy xuất giá trị.

Như vậy, tham số `func` của `greeting()` không phải là một giá trị mà là một hàm. Hàm tham số này có thể được sử dụng bình thường bên trong thân hàm chính.

Một hàm nhận một hàm khác làm tham số được gọi là hàm bậc cao.

Hàm lồng nhau và closure

Trong Python, một hàm có thể trả kết quả là một hàm khác. ***Trả lại một hàm làm kết quả*** có nghĩa là bạn có thể gọi hàm kết quả đó như một hàm khai báo theo kiểu bình thường thông thường.

Hãy xem ví dụ sau:

function_returned.py

```
def hello(name):  
    "đây là một hàm bình thường"  
    print(f'Hello, {name}. Welcome to heaven!')  
  
def welcome():  
    "đây là một hàm bậc cao, kết quả trả về là một hàm khác"  
    return hello  
  
# lời gọi hàm welcome() sẽ tạo ra một hàm mới xxx tương đương với hàm hello()  
xxx = welcome()  
  
# xxx có thể xem như là một tên gọi khác của hello()  
xxx('Obama')
```


Trong ví dụ này, `hello()` là một hàm thông thường in ra lời chào.

Tuy nhiên, `welcome()` lại là một hàm đặc biệt: kết quả trả về của nó không phải là một giá trị mà là một hàm đã khai báo, hàm `hello()`.

Khi đó lời gọi hàm `welcome()` sẽ tạo ra một hàm mới xxx tương đương với hàm `hello()`. Cũng có thể xem xxx như là một tên gọi khác của `hello()`. Bạn sử dụng hàm kết quả `xxx()` giống hệt như sử dụng `hello()`.

Trong Python, bạn có thể khai báo một hàm bên trong hàm khác. Hàm khai báo trong một hàm khác được gọi là **nested function** (hàm lồng nhau).

Hãy cùng xem một ví dụ khác:

closure.py

```
def welcome(greeting='lời chào'):
    def hello(name='tên người'):
        print(f'{greeting}, {name}. Welcome to heaven!')
    return hello
hi = welcome('Hello')
hi('Putin')
```

Lần này thay vì sử dụng một hàm `hello()` xây dựng trước, chúng ta tự định nghĩa một hàm `hello()` mới nằm trong `welcome()`. Hàm `hello()` là hàm lồng (hàm con) của `welcome()`.

Việc khai báo hàm `hello()` trong hàm `welcome()` và trả hàm `hello()` làm kết quả của `welcome()` được gọi là **closure**.

Closure là một hàm khai báo bên trong một hàm khác nhưng sau đó được truyền ra ngoài hàm chứa nó (và có thể được sử dụng bởi code bên ngoài hàm chứa).

Điểm đặc biệt của closure là hàm lồng có thể sử dụng biến và tham số của hàm chứa nó. Trong ví dụ trên, `hello()` sử dụng được tham số `greeting` của `welcome()`.

Xây dựng hàm decorator trong Python

Qua hai phần trên bạn đã hiểu các yếu tố cơ bản tạo ra decorator. Giờ chúng ta cùng vận dụng để tự mình tạo ra một hàm decorator mới.

Hãy cùng thực hiện một ví dụ:

decorator.py

```
def display_decorator(func):  
    def wrapper(str):  
        # logic trước khi chạy hàm func  
        print(f'Log: The function {func.__name__} is executing ...')  
        func(str)  
        # logic sau khi chạy hàm func  
        print('Log: Execution completed.\n')  
    return wrapper  
  
def display(str):  
    print(str)  
  
display = display_decorator(display)  
display('Hello world')  
  
@display_decorator  
def say_hello(str):  
    print(str)  
  
say_hello('Hello, Donald')
```

Trong ví dụ này bạn đã xây dựng hàm bậc cao `display_decorator` nhận một hàm khác làm tham số. Yêu cầu của hàm tham số là nhận một chuỗi tham số.

Trong hàm `display_decorator` bạn tạo một closure `wrapper()`. Trong `wrapper()` bạn thực hiện gọi hàm `func()`, thêm một số logic trước và sau

khi chạy func. Điều đặc biệt là wrapper() cần có chung danh sách tham số với func().

`__name__` là attribute chứa tên của đối tượng. Ở đây chúng ta sử dụng `__name__` để lấy tên chính thức của hàm tham số func.

Hàm `display_decorator` trả lại `wrapper()` làm kết quả.

Trong mẫu decorator, `wrapper()` chính là mẫu chốt vấn đề: để mở rộng tính năng của một hàm `func()`, bạn sẽ thay `func()` bằng `wrapper()`. Điều này thể hiện rõ ở cách sử dụng cơ bản của decorator:

```
def display(str):
    print(str)

display = display_decorator(display) # thay display() bằng wrapper() nhưng gán trở lại tên display
display('Hello world') # hàm vẫn có tên display() nhưng thực tế chính là wrapper()
```

Để đơn giản hóa sử dụng hàm decorator, Python cung cấp cú pháp `@`. Cú pháp này giúp gộp lệnh gọi decorator với lệnh khai báo hàm:

```
@display_decorator
def say_hello(str):
    print(str)

say_hello('Hello, Donald')
```

Hai cách sử dụng decorator là tương đương nhau.

Kết quả chạy chương trình như sau:

```
Log: The function display is executing ...
Hello world
Log: Execution completed.
Log: The function say_hello is executing ...
Hello, Donald
Log: Execution completed.
```

Để ý thấy rằng chức năng chính của hàm `display()` hay `say_hello()` đều chỉ là in ra một chuỗi văn bản. Khi được đánh dấu với decorator, có thêm logic tự động thực hiện trước và sau khi thực thi hàm.

Một số vấn đề khác với decorator

Ở trên chúng ta đã nói về hàm decorator: một hàm nhận một hàm làm tham số và trả về một hàm kết quả.

Python còn mở rộng decorator thành **lớp decorator**. Nghĩa là bạn có thể xây dựng cả một class hoạt động với vai trò decorator cho hàm. Hãy xem ví dụ sau:

```
class_decorator.py

class decoclass(object):

    def __init__(self, f):

        self.f = f

    def __call__(self, *args, **kwargs):

        # logic trước khi gọi hàm f

        print('decorator initialised')

        self.f(*args, **kwargs)

        print('decorator terminated')

        # logic sau khi gọi hàm f

@decoclass

def hello(name):

    print(f'Hello, {name}. Welcome to heaven!')

hello('Obama')
```

Kết quả thực hiện của hàm hello('Obama') sẽ là

```
decorator initialised
Hello, Obama. Welcome to heaven!
decorator terminated
```

Trong ví dụ này, Python sẽ tạo object của decoclass và truyền hàm hello làm tham số khi khởi tạo. Phương thức magic `__call__()` giúp class hoạt động (được gọi) như một hàm (chính xác hơn, một callable object – loại object có thể được gọi giống như hàm).

Nếu logic của bạn quá phức tạp và không thể trình bày trong một hàm, có thể bạn sẽ cần đến lớp decorator.

Nếu hàm wrapper() của bạn đơn giản, bạn có thể **sử dụng hàm *lambda***.

Hàm *lambda* là loại hàm không có tên và phần thân chỉ được chứa một lệnh duy nhất. Hàm *lambda* được khai báo với cú pháp như sau:

```
lambda tham_số: biểu_thức
```

Ví dụ:

```
x = lambda a : a + 10 # lệnh gọi hàm này sẽ là x(5)
x = lambda a, b : a * b # lệnh gọi hàm sẽ là x(5, 6)
```

Bạn có thể sử dụng hàm *lambda* làm wrapper cho decorator như sau:

```
def makebold(f):
    return lambda: "<b>" + f() + "</b>"

@makebold
def say():
    return "Hello"

print(say()) #// kết quả là '<b>Hello</b>'
```

Bạn cũng có thể ghép các decorator lại với nhau thành chuỗi, gọi là **chaining decorators**. Hãy xem ví dụ sau:

```
def makebold(f):
    return lambda: "<b>" + f() + "</b>"

def makeitalic(f):
    return lambda: "<i>" + f() + "</i>"

@makebold
@makeitalic
def say():
    return "Hello"

print(say()) #// kết quả là <b><i>Hello</i></b>
```

Kết luận

Trong phần này bạn đã tìm hiểu chi tiết về decorator trong Python:

1. Decorator trong Python là vận dụng của mẫu thiết kế decorator.
2. Python cung cấp hàm decorator và lớp decorator giúp mở rộng chức năng của một hàm.
3. Decorator là một hàm cấp cao, nhận một hàm làm tham số và trả về một hàm khác.
4. Python cung cấp cú pháp tắt @ để đơn giản hóa sử dụng decorator.
5. Có thể ghép nối các decorator lại với nhau.

Magic method và nạp chồng toán tử trong Python

Như bạn đã biết, Python định nghĩa sẵn một số phép toán trên các kiểu dữ liệu của mình. Ví dụ, phép cộng (+) đã được định nghĩa sẵn cho các kiểu số, kiểu chuỗi ký tự, kiểu danh sách.

Giả sử bạn xây dựng kiểu dữ liệu riêng (class) Matrix (ma trận). Do ma trận cũng có các phép toán như phép cộng, phép nhân, bạn cũng muốn trực tiếp áp dụng các phép toán này cho object của lớp Matrix.

Khi này bạn sẽ cần sử dụng đến kỹ thuật nạp chồng toán tử.

Nạp chồng toán tử (operator overloading) là kỹ thuật **định nghĩa các phép toán sẵn có trên kiểu dữ liệu tự tạo**.

Trong Python nạp chồng toán tử hoạt động tương đối khác với ngôn ngữ như C++/Java/C#. Các toán tử trong Python hoạt động dựa trên một số magic method.

NỘI DUNG

1. Magic method trong Python
2. Ghi đè phương thức `__new__()`
3. Ghi đè phương thức `__str__()` và `__repr__()`
4. Nạp chồng các phép toán số học
5. Kết luận

Magic method trong Python

Nạp chồng toán tử hoạt động dựa trên magic method. Vì vậy trước khi học cách nạp chồng toán tử, chúng ta cần hiểu thế nào là magic method trong Python.

Khái niệm magic method được nhắc đến lần đầu khi bạn tìm hiểu về hàm tạo trong Python.

Magic method là một số phương thức đặc biệt được Python tự động gọi. Lấy ví dụ, khi bạn thực hiện phép cộng, Python sẽ tự động gọi tới phương thức `__add__()`. Khi bạn muốn khởi tạo một object, phương thức `__new__()` được Python tự động gọi để tạo object trước.

Các kiểu dữ liệu xây dựng sẵn của Python có một số magic method. Bạn có thể xem danh sách các phương thức này bằng lệnh `dir(<tên kiểu>)` trong IDLE như sau:

```
>>> dir(int) # danh sách magic method của kiểu int

['_abs_', '_add_', '_and_', '_bool_', '_ceil_', '_class_', '_delattr_', '_dir_', '_divmod_',
'_doc_', '_eq_', '_float_', '_floor_', '_floordiv_', '_format_', '_ge_', '_getattr_',
'_getnewargs_', '_gt_', '_hash_', '_index_', '_init_', '_init_subclass_', '_int_', '_invert_',
'_le_', '_lshift_', '_lt_', '_mod_', '_mul_', '_ne_', '_neg_', '_new_', '_or_', '_pos_',
'_pow_', '_radd_', '_rand_', '_rdivmod_', '_reduce_', '_reduce_ex_', '_repr_',
'_rfloordiv_', '_rlshift_', '_rmod_', '_rmul_', '_ror_', '_round_', '_rpow_', '_rrshift_',
'_rshift_', '_rsub_', '_rtruediv_', '_rxor_', '_setattr_', '_sizeof_', '_str_', '_sub_',
'_subclasshook_', '_truediv_', '_trunc_', '_xor_', 'as_integer_ratio', 'bit_length', 'conjugate',
'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

Một ví dụ nhỏ khác:

```
>>> a = 10

>>> a.__add__(10) # tương đương với gọi a + 10

20

>>> a + 10

20
```

Qua ví dụ này bạn có thể thấy sự tương đương giữa phép toán `+` và lời gọi hàm `__add__()`. Khi bạn sử dụng phép toán `+`, Python tự động thay bạn gọi hàm `__add__()`. Bạn cũng có thể tự mình gọi tới magic method đứng sau phép `+`.

Tóm lại, magic method là những phương thức đứng sau các phép toán thường gặp. Bản thân phép toán chỉ là một dạng cú pháp tiện lợi hơn để gọi tới các phương thức này.

Khi hiểu điều này bạn hẳn có thể nhận ra ngay, thực hiện nạp chồng toán tử trong Python thực chất là định nghĩa lại magic method của phép toán trên class mới. Như vậy trong Python, ***nạp chồng toán tử = ghi đè magic method*** tương ứng.

Ghi đề phương thức `__new__()`

Ghi đề phương thức `__new__()` cho phép bạn ***nạp chồng phép toán tạo object***. Đây chỉ là một ví dụ để bạn hiểu kỹ hơn về class, object và magic method trong Python. Trên thực tế bạn ít khi phải nạp chồng phép toán này.

Hãy xem ví dụ sau đây:

```
class Employee:
    def __new__(cls, *args, **kwargs):
        print('__new__ is being called')
        instance = object.__new__(cls)
        return instance
    def __init__(self, name: str):
        print('__init__ is being called')
        self.name = name
    def print(self):
        print(self.name)
if __name__ == '__main__':
    trump = Employee('Donald Trump')
    trump.print()
    putin = Employee('Vladimir Putin')
    putin.print()
```

Chạy script trên bạn thu được kết quả như sau:

```
__new__ is being called
__init__ is being called
Donald Trump
__new__ is being called
__init__ is being called
Vladimir Putin
```

Hãy để ý đến cách ghi đề phương thức `__new__()`:

```
def __new__(cls, *args, **kwargs):  
    print('__new__ is being called')  
    instance = object.__new__(cls)  
    return instance
```

Để ghi đề `__new__` bạn phải cung cấp một tham số bắt buộc `cls`. Tham số này có vai trò tương tự như tham số trong class method: nó chính là class mà bạn đang cần tạo object. Python sẽ tự động truyền giá trị cho tham số này.

Hai tham số còn lại `*args` và `**kwargs` giúp bạn viết `__init__()` với tham số tùy ý. Sở dĩ phải có `*args` và `**kwargs` ở đây là vì `__new__` đòi hỏi cùng tham số với `__init__`. Do `__new__` được gọi tự động và `__init__` xây dựng sau, chúng ta sử dụng `*args` và `**kwargs` cho `__new__()` để bao phủ hết các khả năng truyền tham số có thể gặp ở `__init__()`.

Trong thân hàm `__new__()` bạn bắt buộc phải gọi tới phương thức `__new__()` của lớp cha `object`. Nhắc lại: `object` là lớp cha của mọi class trong Python. Lời gọi `__new__()` của `object` sẽ sinh ra object mới và bạn cần trả lại object này.

Kế tiếp, Python sẽ gọi `__init__` và tự động truyền object mới sang phương thức `__init__()`.

Khi chạy chương trình bạn có thể thấy rõ, khi gặp lệnh tạo object, `__new__()` được gọi tự động trước, sau đó mới đến `__init__()`.

Ghi đề phương thức `__str__()` và `__repr__()`

Phương thức `__str__()` được gọi tự động khi bạn phát lệnh in object `print()`. Phương thức này sẽ chuyển object về một chuỗi ký tự phù hợp cho việc in.

Phương thức `__repr__()` được gọi tự động khi bạn muốn xuất biến ở chế độ tương tác.

Khi xây dựng một class, nếu muốn tiện lợi cho việc in object của class, bạn nên định nghĩa phương thức `__str__()` và `__repr__()`.

Hãy xem ví dụ sau:

```
class Vector:

    """A class for vector"""

    def __init__(self, x:float, y:float):

        self.x = x

        self.y = y

    def __str__(self):

        return f'({self.x}, {self.y})'

    def __repr__(self):

        return f'({self.x}, {self.y})'
```

Đây là ví dụ về một class Vector hai chiều đơn giản.

Thông thường, vector trong toán học hay được in ra ở dạng (x, y, z, ..). Chúng ta cũng muốn rằng khi dùng hàm print() trên object của Vector, kết quả in ra cũng có dạng (x, y).

Để làm việc này, chúng ta cần định nghĩa phương thức magic __str__() trong class Vector:

```
def __str__(self):

    return f'({self.x}, {self.y})'
```

Phương thức này cần trả về một chuỗi ký tự.

__str__() rất giống với phương thức ToString() trong C#.

Với class Vector như trên, bạn có thể sử dụng như sau:

```
>>> v1 = Vector(0, 1); v2 = Vector(2, 3)

>>> print(v1)

(0, 1)

>>> print(v2)

(2, 3)
```

Tuy nhiên, nếu chỉ có __str__(), khi bạn nhập lệnh:

```
>>> v1 = Vector(1, 2)

>>> v1

<__main__.Vector object at 0x00000183425A2DC0>
```

Để giao diện tương tác cũng in ra kết quả như khi dùng hàm `print()`, bạn cần định nghĩa lại phương thức `__repr__()`:

```
def __repr__(self):  
    return f'({self.x}, {self.y})'
```

Phần thân `__repr__()` và `__str__()` cơ bản là giống nhau.

Nạp chồng các phép toán số học

Các phép toán số học `+`, `-`, `*`, `/` tương ứng với các magic method `__add__()`, `__sub__()`, `__mul__()`, `__div__()`.

Phép toán `+` `-` (âm dương) tương ứng với `__pos__()` và `__neg__()`.

Để nạp chồng các phép toán này, bạn cần định nghĩa/ghi đè các magic method tương ứng.

Hãy xem ví dụ sau đây:

```
class Vector:  
    """A class for vector"""  
  
    def __init__(self, x:float, y:float):  
        self.x = x  
        self.y = y  
  
    def __str__(self):  
        return f'({self.x}, {self.y})'  
  
    def __repr__(self):  
        return f'({self.x}, {self.y})'  
  
    def __add__(self, v):  
        x = self.x + v.x  
        y = self.y + v.y  
        return Vector(x, y)  
  
    def __sub__(self, v):  
        x = self.x - v.x  
        y = self.y - v.y  
        return Vector(x, y)  
  
    def __mul__(self, n):  
        x = self.x * n
```

```

        y = self.y * n

    return Vector(x, y)

def __neg__(self):
    return Vector(self.x * -1, self.y * -1)

```

Với class Vector như trên bạn có thể thực hiện các phép toán cơ bản như sau:

```

>>> v1 = Vector(1, 2)
>>> v2 = Vector(3, 4)
>>> v1 + v2 # phép cộng
(4, 6)
>>> v1 - v2 # phép trừ
(-2, -2)
>>> v1 * 2 # phép nhân vô hướng
(2, 4)
>>> -v1 # nghịch đảo
(-1, -2)

```

Dưới đây là danh sách các hàm toán học cơ bản và magic method tương ứng của chúng

<code>__add__(self, other)</code>	phép cộng +
<code>__sub__(self, other)</code>	phép trừ -
<code>__mul__(self, other)</code>	phép nhân *
<code>__floordiv__(self, other)</code>	phép chia lấy phần nguyên //
<code>__div__(self, other)</code>	phép chia /
<code>__mod__(self, other)</code>	phép chia lấy phần dư %
<code>__pow__(self, other[, modulo])</code>	phép tính lũy thừa **
<code>__lt__(self, other)</code>	phép so sánh nhỏ hơn <

<code>__le__(self, other)</code>	phép so sánh nhỏ hơn hoặc bằng <code><=</code>
<code>__eq__(self, other)</code>	phép so sánh bằng <code>==</code>
<code>__ne__(self, other)</code>	phép so sánh khác <code>!=</code>
<code>__ge__(self, other)</code>	phép so sánh lớn hoặc bằng <code>>=</code>

Kết luận

Trong phần này chúng ta đã làm quen với magic method và ứng dụng của chúng trong nạp chồng toán tử:

- Các toán tử trong Python đều tương ứng với một magic method.
- Python tự động gọi magic method khi gặp phép toán tương ứng.
- Phép toán là cú pháp khác để sử dụng magic method. Có thể trực tiếp gọi magic method nếu muốn.
- Để nạp chồng phép toán cho class mới cần ghi đè hoặc định nghĩa magic method tương ứng trong class.

Iterator và Iterable trong Python

Trong các phần trước bạn đã biết các kiểu dữ liệu cơ bản của Python như list, tuple. Bạn cũng biết về hàm range() và vòng lặp for. Tất cả chúng đều thực thi một mẫu thiết kế chung – mẫu iterator.

Iterator là một mẫu thiết kế rất quan trọng giúp bạn làm việc với dữ liệu. Ví dụ bạn có thể tạo ra các cấu trúc dữ liệu tuần tự mới. Các cấu trúc dữ liệu dạng này phù hợp cho việc tạo ra các chuỗi số, làm việc với file, hoặc giao tiếp qua mạng.

NỘI DUNG

1. Collection và iteration
2. Ví dụ về xây dựng Iterable và Iterator class
3. Iterable trong Python
4. Iterator trong Python
5. Cách hoạt động của vòng lặp for
6. Kết luận

Collection và iteration

Trong Python và các ngôn ngữ khác có thể phân ra hai loại dữ liệu: loại dữ liệu chỉ chứa một giá trị (như các loại số và bool), và loại dữ liệu chứa nhiều giá trị.

Loại dữ liệu chứa nhiều giá trị bạn đã gặp bao gồm list, tuple, dict, string, set. Chúng được gọi chung là dữ liệu **collection** (tập hợp) hay **container** (hộp chứa).

Một đặc điểm phổ biến của container/collection là khả năng truy xuất lần lượt từng phần tử. Lấy ví dụ, bạn có thể lần lượt duyệt qua lần lượt từng phần tử của một danh sách (list) hoặc một tập hợp toán học (set).

Người ta gọi quá trình **truy xuất lần lượt từng phần tử** của container là **iteration** (vòng lặp). Tại mỗi thời điểm trong iteration, bạn chỉ có thể truy xuất một bộ phận của dữ liệu.

Dữ liệu dạng container có thể chia làm hai loại:

1. Một số cho phép đánh chỉ số và truy xuất ngẫu nhiên đến từng phần tử, ví dụ như list, tuple, string. Các kiểu dữ liệu này được gọi là các kiểu dữ liệu **sequence** (kiểu dữ liệu tuần tự).
2. Một số loại dữ liệu khác không cho đánh chỉ số và không thể truy xuất ngẫu nhiên như dict, set, file.

Khi làm việc với các loại dữ liệu dạng tập hợp như vậy bạn có thể hình dung theo hai kiểu:

1. toàn bộ dữ liệu đồng thời tải vào bộ nhớ,
2. tại mỗi thời điểm bạn chỉ tải và truy xuất một bộ phận của dữ liệu.

Bạn có thể so sánh thế này:

- Khi đọc dữ liệu từ một file văn bản, bạn có thể lựa chọn tải toàn bộ văn bản vào bộ nhớ ngay lập tức, hoặc cũng có thể lựa chọn mỗi lần chỉ tải một đoạn văn bản (lấy ký tự /n làm mốc) để xử lý.
- Khi tạo ra một dãy số (chẳng hạn chuỗi Fibonacci), bạn có thể lựa chọn tạo ra tất cả các số (trong một giới hạn nào đó) ngay lập tức, hoặc cũng có thể lựa chọn mỗi lần chỉ tạo ra một số, và chỉ tạo ra số khi chương trình cần đến (ví dụ, để in ra hoặc để tính toán).

Cách thứ nhất có lợi về tốc độ truy cập vì mọi thứ nằm hết trong bộ nhớ nhưng đồng thời lại tốn bộ nhớ hơn. Cách này đặc biệt không phù hợp cho việc xử lý lượng dữ liệu quá lớn hoặc dữ liệu không xác định được vị trí kết thúc (như đọc dữ liệu từ mạng hoặc tạo chuỗi số).

Cách thứ hai có nhược điểm về tốc độ truy xuất, vì mỗi khi cần truy xuất, dữ liệu mới được tạo ra hoặc được tải vào bộ nhớ. Tuy nhiên cách này có thể xử lý được lượng dữ liệu không giới hạn. Đây là cách Python lựa chọn để áp dụng cho các kiểu dữ liệu tập hợp như list, tuple, string, cũng như vận dụng trong vòng lặp for.

Các kiểu dữ liệu tập hợp trong Python cũng được gọi chung là **iterable**. Để sử dụng dữ liệu iterable, bạn cần đến một **iterator** tương ứng.

Ví dụ về xây dựng Iterable và Iterator class

Hãy cùng thực hiện ví dụ sau:

fibonacci.py

```
class FibonacciIterable:

    def __init__(self, count=10):
        self.count = count

    def __iter__(self):
        return FibonacciIterator(self.count)

class FibonacciIterator:

    def __init__(self, count=10):
        self.a, self.b = 0, 1
        self.count = count
        self.__i = 0

    def __next__(self):
        if self.__i > self.count:
            raise StopIteration()

        value = self.a

        self.a, self.b = self.b, self.a + self.b

        self.__i += 1

        return value

    # def __iter__(self):
    #     return self

def test1():

    print('Test 1:', end=' ')

    iterable = FibonacciIterable(15)

    for f in iterable:
        print(f, end=' ')

    print()
```

```
def test2():
    print('Test 2:', end=' ')
    iterator = FibonacciIterator(15)
    for f in iterator:
        print(f, end=' ')
    print()

def test3():
    print('Test 3:', end=' ')
    iterator = FibonacciIterator(15)
    while True:
        try:
            f = next(iterator)
            print(f, end=' ')
        except StopIteration:
            break
    print()

def test4():
    print('Test 4:', end=' ')
    iterable = FibonacciIterable(15)
    iterator = iter(iterable)
    while True:
        try:
            f = next(iterator)
            print(f, end=' ')
        except StopIteration:
            break
    print()

if __name__ == '__main__':
    test1()
```

```
#test2()

test3()

test4()
```

Trong ví dụ này chúng ta xây dựng các class giúp tạo ra `count` phần tử đầu tiên của dãy số Fibonacci.

Nếu bạn chưa biết: dãy số Fibonacci có hai phần tử đầu tiên lần lượt là 0 và 1. Phần tử thứ ba trở đi có giá trị bằng tổng giá trị hai phần tử trước nó.

Kết quả của cả 3 hàm test là 15 giá trị đầu tiên của chuỗi Fibonacci:

```
Test 1: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
Test 3: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
Test 4: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

Trong ví dụ fibonacci.py bạn xây dựng hai class, Fibonacci**Iterable** và Fibonacci**Iterator**. Hai class này lần lượt thuộc về hai loại khác biệt: iterable và iterator.

Sự phân chia iterable và iterator như vậy tuân theo **mẫu thiết kế iterator**. Mẫu thiết kế này giúp phân tách thùng chứa (container) với dữ liệu thực sự chứa trong nó. Trong đó, container là iterable, còn dữ liệu thực sự chính là iterator. Nghĩa là, cùng một container nhưng bạn có thể cho nó “chứa” dữ liệu khác biệt nhau.

Iterable trong Python

Iterable trong Python là loại object container/collection.

Theo mẫu thiết kế, về hình thức thì iterable có khả năng **trả lại lần lượt từng giá trị**. Nghĩa là trong client code bạn có thể lần lượt duyệt từng phần tử của nó.

Tuy nhiên, iterable thực tế chỉ đơn giản là một loại thùng chứa (container). Dữ liệu thực thụ sẽ do **iterator** tải hoặc tạo ra. Do vậy, mỗi iterable phải có một iterator tương ứng.

Về quy định, để tạo ra một iterable, trong class phải **chứa phương thức `__iter__()`**. Khi có mặt phương thức `__iter__()`, class tương ứng được coi là một iterable và có thể **sử dụng trực tiếp trong vòng lặp for**.

Phương thức `__iter__()` phải **trả lại một object thuộc kiểu iterator**. Phương thức này sẽ được gọi tự động để lấy iterator, ví dụ, khi sử dụng trong vòng lặp `for`.

Hãy nhìn lại code của `FibonacciIterable` bạn sẽ thấy các lý thuyết trên đều được áp dụng:

```
class FibonacciIterable:

    def __init__(self, count=10):

        self.count = count

    def __iter__(self):

        return FibonacciIterator(self.count)
```

Hàm tạo của class này nhận giá trị `count` – số lượng phần tử của dãy sẽ tạo.

Class này xây dựng phương thức `__iter__()`. Phương thức này chỉ làm nhiệm vụ trả lại một object của iterator `FibonacciIterator`.

Cũng để ý rằng, `FibonacciIterable` là class sẽ được sử dụng trong client code còn `FibonacciIterator` sẽ ở “hậu trường”. Người sử dụng sẽ chỉ biết đến iterable chứ không biết đến iterator. Như vậy, mặc dù client code cung cấp biến `count` cho iterable, thực tế giá trị này sẽ truyền sang cho iterator để sử dụng. Tự bản thân iterable không sử dụng.

Iterator trong Python

Iterator là loại object có nhiệm vụ tạo ra dữ liệu cho iterable. Tuy nhiên, tự bản thân iterator lại không phải là một kiểu dữ liệu container. Iterator phải phối hợp với iterable thì mới tạo ra được một container có dữ liệu và có thể duyệt được dữ liệu.

Tổ hợp iterable-iterator có thể hình dung giống như một chiếc thùng (iterable), và bạn có thể bỏ vào đó gạch hoặc ngói hoặc bất kỳ vật gì. Việc bỏ vào thùng vật gì sẽ do iterator quyết định.

Iterator object bắt buộc phải xây dựng **phương thức `__next__()`**. Nhiệm vụ của phương thức này là tạo ra phần tử cho iterable. Phương thức này cũng được gọi tự động nếu client code cần lấy dữ liệu. Trong phương thức `__next__()`, nếu không còn phần tử nào nữa thì cần phát ra ngoại lệ `StopIteration`.

Hãy xem lại code của lớp FibonacciIterator:

```
class FibonacciIterator:

    def __init__(self, count=10):

        self.a, self.b = 0, 1

        self.count = count

        self.__i = 0

    def __next__(self):

        if self.__i > self.count:

            raise StopIteration()

        value = self.a

        self.a, self.b = self.b, self.a + self.b

        self.__i += 1

        return value

# def __iter__(self):
#     return self
```

Khi khởi tạo object sẽ gán giá trị hai phần tử đầu tiên của dãy ($a = 0$, $b = 1$), nhận giá trị biến count (số lượng phần tử cần tạo), và gán biến đếm `__i = 0`.

Mỗi lần gọi phương thức `__next__()` sẽ kiểm tra biến đếm. Nếu biến đếm vượt quá count sẽ phát ra ngoại lệ `StopIteration`. Đây là yêu cầu bắt buộc của `__next__()` nếu muốn kết thúc duyệt dữ liệu. Nếu biến đếm trong giới hạn thì trả lại giá trị hiện tại của `a`, đồng thời cập nhật giá trị mới cho `a`, `b` ($a = b$, còn $b = a + b$). Logic tạo ra `a` và `b` hoàn toàn theo định nghĩa của chuỗi Fibonacci.

Thông thường, trong iterator có thể thực thi cả `__iter__()` giúp một iterator cũng đồng thời là một iterable. Trong trường hợp này `__iter__()` của iterator chỉ cần trả lại chính object đó (`self`). Tuy nhiên, để giúp bạn dễ dàng phân biệt iterable và iterator, chúng ta tạm thời comment phương thức `__iter__()` trong iterator.

Cách hoạt động của vòng lặp for

Iterable là loại object mà bạn có thể trực tiếp sử dụng với vòng lặp for của Python.

Mặc dù bạn đã học cách sử dụng vòng lặp for trước đây. Tuy nhiên, hẳn bạn đã nhận ra rằng vòng lặp for của Python không thực sự giống như vòng lặp for của C/C++/Java hay C#. Vòng lặp for của Python, về hình thức, thì tương tự như for của các ngôn ngữ kiểu C nhưng lại hoạt động giống như foreach của C#/C++/Java.

Hãy xem lại hàm Test4:

```
def test4():  
    print('Test 4:', end=' ')  
    iterable = FibonacciIterable(15)  
    iterator = iter(iterable)  
    while True:  
        try:  
            f = next(iterator)  
            print(f, end=' ')  
        except StopIteration:  
            break  
    print()
```

Mặc dù ở đây sử dụng vòng lặp while, đó chính là mô phỏng hoạt động của vòng lặp for trên thực tế.

Logic hoạt động của vòng lặp for như sau:

- Gọi hàm tích hợp iter(). Hàm này thực tế sẽ gọi __iter__() của iterable để lấy object iterator tương ứng.
- Trên iterator liên tục gọi hàm next() để lấy dữ liệu cụ thể. Hàm này sẽ gọi tới __next__() của iterator.
- Thực hiện các xử lý cần thiết trên dữ liệu vừa lấy được.
- Nếu gặp ngoại lệ StopIteration thì dừng vòng lặp.

Bằng cách này bạn có thể tự mình tạo ra một 'vòng lặp' riêng thay cho for.

Dĩ nhiên, cách sử dụng iterable này rất cồng kềnh, chỉ phục vụ cho mục đích tìm hiểu nguyên lý hoạt động của vòng for.

Trên thực tế, vòng lặp for của Python tự động thực hiện tất cả các thao tác trên nếu bạn cung cấp cho nó một iterable. Đây là trường hợp của hàm test1():

```
def test1():  
    print('Test 1:', end=' ')  
    iterable = FibonacciIterable(15)  
    for f in iterable:  
        print(f, end=' ')  
    print()
```

Bạn tạo ra một object iterable từ class FibonacciIterable và cấp thẳng cho lệnh for. Lệnh for sẽ tự động thực hiện theo logic chúng ta đã chỉ ở trên. Mỗi lần gọi next() sẽ trả giá trị về cho biến tạm f để chúng ta sử dụng.

Trong trường hợp bạn không xây dựng iterable mà chỉ có iterator, bạn không thể trực tiếp sử dụng iterator trong vòng lặp for. Khi đó bạn phải sử dụng theo kiểu của hàm test3:

```
def test3():  
    print('Test 3:', end=' ')  
    iterator = FibonacciIterator(15)  
    while True:  
        try:  
            f = next(iterator)  
            print(f, end=' ')  
        except StopIteration:  
            break  
    print()
```

Ở đây bạn phải làm thủ công với vòng lặp while theo đúng logic đã trình bày ở trên.

Như đã nói, iterator cũng thường xây dựng luôn cả phương thức `__iter__()` giúp cho iterator đồng thời đóng vai trò của iterable. Nếu bạn bỏ dấu comment của phương thức `__iter__()` trong `FibonacciIterator`, bạn có thể sử dụng hàm `test2`:

```
def test2():  
  
    print('Test 2:', end=' ')  
  
    iterator = FibonacciIterator(15)  
  
    for f in iterator:  
        print(f, end=' ')  
  
    print()
```

Ở đây object iterator đồng thời đóng luôn vai trò của iterable. Do vậy bạn dùng được nó trong vòng lặp `for`.

Kết luận

Trong phần này chúng ta đã tìm hiểu về iterable và iterator. Đây là một chủ đề khó và rất hay gây nhầm lẫn. Không có nhiều tài liệu trên mạng giải thích được rõ ràng vấn đề này.

- Iterable và iterator là hai bộ phận của mẫu thiết kế iterator giúp phân tách thùng chứa (iterable) và dữ liệu (iterator).
- Iterable được sử dụng bởi client code trong vòng lặp `for`; Iterator được vòng lặp này tự động tạo ra từ iterable và sử dụng để lấy dữ liệu.
- Iterable phải thực thi phương thức `__iter__()` nhằm chỉ định iterator.
- Iterator phải thực thi phương thức `__next__()` để lấy dữ liệu.
- Thông thường iterator cũng thực thi luôn cả `__iter__()`, do vậy iterator có thể đồng thời đóng vai trò iterable.

Generator trong Python

Lớp generator trong Python

Hãy cùng thực hiện một ví dụ:

(ví dụ 1) - fibonacci.py

```
class FibonacciIterable:

    def __init__(self, count=10):

        self.a, self.b = 0, 1

        self.count = count

    def __iter__(self):

        i = 0

        while True:

            if i > self.count:

                break

            yield self.a

            self.a, self.b = self.b, self.a + self.b

            i += 1

def test1():

    iterable = FibonacciIterable(20)

    for f in iterable:

        print(f, end=' ')

if __name__ == '__main__':

    test1()
```

Trong ví dụ 1 này chúng ta sử dụng một cách khác để tạo ra chuỗi fibonacci. Hãy để ý ở đây chúng ta chỉ xây dựng một class FibonacciIterable.

Trong hàm tạo, chúng ta gán hai giá trị đầu tiên của chuỗi, a = 0 và b = 1. Biến count chứa số lượng phần tử cần tạo ra.

Điểm mấu chốt của class `FibonacciIterable` là phương thức `__iter__()`. Đây là phương thức magic mà bất kỳ class iterable nào đều phải thực thi. Nói theo cách khác, nếu một class thực thi phương thức `__iter__()`, nó sẽ trở thành một iterable.

Logic của `__iter__()` không phức tạp:

1. Tạo một biến đếm `i` và một vòng lặp `while`.
2. Nếu biến đếm vượt quá giá trị của `count` thì phá vỡ vòng lặp.
3. Nếu biến đếm trong giới hạn cho phép thì trả lại giá trị của phần tử hiện tại của chuỗi, cập nhật giá trị cho phần tử tiếp theo, và tăng giá trị biến đếm.

Hãy để ý rằng, `FibonacciIterable` không hề có iterator đi kèm, hoặc cũng có thể nói rằng lớp `FibonacciIterable` đóng vai trò của của iterable và iterator.

Cấu trúc này đơn giản hơn rất nhiều tổ hợp iterable (thực thi `__iter__()`) và iterator (thực thi `__next__()`) mà bạn đã học trong bài trước (xem lại bài học iterable và iterator trong Python).

Python gọi loại cấu trúc sinh ra chuỗi giá trị sử dụng từ khóa `yield` như vậy là các **generator**. Bạn có thể thực thi generator trong class hoặc trong hàm. Trong ví dụ 1, `FibonacciIterable` là một class generator.

Từ khóa `yield`

Quay lại ví dụ 1, hãy để ý đến điểm đặc thù của `__iter__()`: **từ khóa `yield`**. Đây là một từ khóa đặt biệt trong Python.

Từ khóa `yield` trong Python hoạt động tương tự như `return` ở chỗ nó ***trả lại giá trị cho hàm gọi***, tuy nhiên lại ***không kết thúc việc thực hiện*** hàm/phương thức.

Khi gặp `yield`, Python sẽ trả lại giá trị tương ứng cho hàm gọi và tạm dừng (pause) thực hiện hàm/phương thức. *Tạm dừng* ở đây mang đúng nghĩa đen: mọi trạng thái của hàm/phương thức vẫn được duy trì và có thể tiếp tục hoạt động trở lại. Khi hàm gọi cần sử dụng đến giá trị tiếp theo, `__iter__()` sẽ được hồi phục từ trạng thái tạm dừng.

Khi phục hồi từ trạng thái tạm dừng, hàm/phương thức tiếp tục thực hiện từ vị trí nó dừng lại trước đó chứ không bắt đầu lại. Như vậy, cả quá trình thực hiện của hàm/phương thức với vòng lặp và yield tạo ra cả một chuỗi dữ liệu.

Có thể thấy, mô hình hoạt động của yield rất tương đồng với `__next__` của iterator.

Đặc điểm này của yield giúp nó có thể sinh ra iterator từ phương thức hoặc hàm mà không cần xây dựng class riêng thực thi phương thức `__next__()` như trong mô hình iterator thông thường.

yield trong Python hoạt động giống như yield return trong C# hay Java.

Trong ví dụ 1, sử dụng từ khóa yield trong `__iter__()` biến FibonacciIterable đồng thời thành (1) iterable và (2) iterator. Điều này bạn có thể thấy trong hàm `test1()`, khi object của FibonacciIterable được sử dụng trực tiếp trong vòng lặp for (với vai trò iterable) trong khi chúng ta không hề xây dựng class iterator nào.

Một đặc điểm quan trọng của yield là **thực thi trễ** (lazy execution). Thực thi trễ thể hiện rằng chỉ khi nào nơi gọi hàm thực sự cần sử dụng dữ liệu (ví dụ, để in ra hoặc tính toán), hàm tương ứng mới được thực thi.

Bạn có thể sử dụng nhiều yield trong một hàm/phương thức. Hãy xem ví dụ sau:

```
class YieldSample:
    def __iter__(self):
        i = 10
        yield i + 10
        yield i * 10
        yield i / 10
        yield i - 10

if __name__ == '__main__':
    for value in YieldSample():
        print(value)
```

Kết quả thực hiện của ví dụ trên như sau:

```
20
100
1.0
0
```

Mỗi lần gặp `yield`, phương thức sẽ trả lại kết quả tương ứng và tạm dừng chờ khi chương trình cần dữ liệu sẽ chạy tiếp.

Nếu sử dụng `yield` cùng `__iter__()` trong class sẽ tạo ra một tổ hợp iterable và iterator mà bạn có thể trực tiếp sử dụng cùng vòng lặp `for`. Class như vậy được gọi là class generator.

Một đặc điểm quan trọng là bạn cũng có thể sử dụng `yield` trong hàm để tạo thành hàm generator. Hàm generator thậm chí còn giúp giảm lược hơn nữa việc tạo ra các cấu trúc dữ liệu dạng iterable/container.

Hàm generator trong Python

Hãy cùng xem ví dụ sau:

Ví dụ 2 - `function_generator.py`

```
def fibonacci(count: int = 15):
    a, b = 0, 1
    i = 0
    while True:
        if i > count:
            break
        yield a
        a, b = b, a + b
        i += 1

def yield_return():
    i = 10
    yield i + 10
    yield i * 10
    yield i / 10
    yield i - 10
```

```
def test1():
    iterable = fibonacci(15)
    for f in iterable:
        print(f, end=' ')
    print()

def test2():
    for value in yield_return():
        print(value)

if __name__ == '__main__':
    test1()
    test2()
```

Kết quả thực hiện của ví dụ trên như sau:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
20
100
1.0
0
```

Đây là ví dụ về cách xây dựng hàm generator sử dụng từ khóa yield.

Quy tắc sử dụng và cách hoạt động của yield trong hàm không có gì khác biệt với khi sử dụng trong phương thức của class.

Tuy nhiên, khi sử dụng với hàm, Python sẽ biến hàm đó trở thành một tổ hợp iterable + iterator. Bạn có thể trực tiếp sử dụng hàm này trong vòng lặp for. Hàm này giờ hoạt động giống hệt như class generator cũng như tổ hợp iterable+iterator.

Đây là cách thức nhanh gọn và đơn giản nhất để tạo ra iterator trong Python.

Biểu thức generator

Trong trường hợp bạn cần tạo ra một generator đơn giản để truyền làm tham số cho một hàm khác, bạn có thể sử dụng biểu thức generator.

Biểu thức generator là một biểu thức mà kết quả trả về là một iterator. Bạn cũng có thể hình dung biểu thức generator tương tự như một hàm generator đơn giản không tên và chỉ có tham hàm.

Hãy xem ví dụ sau:

```
squares = (x*x for x in range(1, 20, 2))  
  
for s in squares:  
    print(s, end=' ')
```

Trong ví dụ trên, `(x*x for x in range(1, 20, 2))` là một biểu thức generator. Kết quả trả về của biểu thức này được biến `squares` trở tới. Biến `squares` là một iterator. Nói chính xác hơn, `squares` là một object của class generator.

Để ý rằng, cú pháp của biểu thức generator rất giống với list comprehension, ngoại trừ cặp dấu ngoặc tròn.

Kết luận

Trong phần này bạn đã làm quen với generator trong Python với các ý sau:

- Generator là một cú pháp đơn giản hơn giúp bạn tạo ra iterable và iterator thay cho mô hình iterator cơ bản.
- Có thể sử dụng generator với class, function và expression.
- Generator trong Python sử dụng từ khóa `yield`.

Map, Filter, Reduce – lập trình hàm với Python

Map, filter và reduce là ba hàm đặc thù của lập trình hàm. Điểm chung của chúng là cho phép áp dụng một hàm xử lý trên một danh sách dữ liệu: map áp dụng một hàm trên dữ liệu của một danh sách; filter áp dụng hàm cho danh sách và chỉ lấy những kết quả phù hợp điều kiện; reduce biến một danh sách về một giá trị.

Ba loại hàm đặc biệt này cho phép viết code xử lý dữ liệu ngắn gọn, súc tích, hiệu quả hơn mà không cần đến các kỹ thuật lập trình thông thường với vòng lặp và rẽ nhánh.

NỘI DUNG

1. Imperative và functional
2. Sử dụng hàm map() trong Python
3. Hàm filter() trong Python
4. Hàm reduce() trong Python
5. Sử dụng hàm lambda
6. Kết luận

Imperative và functional

Giả sử bạn đang có một danh sách X chứa các giá trị như sau:

```
X = [x/100 for x in range(100, 1000)] #dải giá trị [1, 10] với bước 0,01
```

Bạn muốn tạo ra một danh sách khác, Y, trong đó mỗi phần tử của danh sách Y được tính theo hàm $y = ax^2 + bx + c$, với x là giá trị của phần tử tương ứng trong X.

Theo lối suy nghĩ thông thường, bạn sẽ:

1. tạo một danh sách trống Y = []
2. định nghĩa hàm parabolic
3. sử dụng vòng lặp với danh sách X
4. Ứng với mỗi phần tử của x sẽ tính ra y thông qua gọi hàm parabolic
5. Thêm giá trị y vào cuối danh sách Y

“Thuật toán” trên được minh họa qua code Python như sau:

Ví dụ 1 - imperative.py

```
X = [x/100 for x in range(100, 1000)]
Y = []

def parabolic(x, a=1.0, b=0.0, c=0.0):
    return a * x * x + b * x + c

for x in X:
    y = parabolic(x)
    Y.append(y)
```

Lỗi lập trình này được gọi là lập trình mệnh lệnh (imperative programming).

Lỗi lập trình này không sai nhưng dài dòng và thủ công. Bạn phải tự mình quản từng bước trong quá trình thực hiện thuật toán đặt ra.

Trong các tài liệu trước bạn đã biết về các cấu trúc dữ liệu dạng container như list, tuple, dict. Bạn cũng học cách xây dựng các container riêng qua bài học về iterable/iterator và generator trong Python.

Bạn có thể đã thắc mắc, tại sao Python lại phải tạo các cấu trúc rắc rối như iterator?

Trên thực tế, đó là các cấu trúc dữ liệu rất mạnh để sử dụng trong lập trình hàm (functional programming) – một xu hướng lập trình rất mạnh trong xử lý dữ liệu. Map, filter và reduce là ba loại hàm thông dụng trong xu hướng này.

Sử dụng hàm map() trong Python

Giờ hãy thay thế toàn bộ code ở trên như sau:

Ví dụ 2 - map.py

```
X = [x / 100 for x in range(100, 1000)]

def parabolic(x, a=1.0, b=0.0, c=0.0):
    return a * x * x + b * x + c

Y = list(map(parabolic, X))

for y in Y:
    print(y, end=' ')
```


Đoạn code mới vẫn giữ lại phần khai báo X và hàm parabolic nhưng phần tính toán Y đã hoàn toàn thay đổi. Chúng ta sử dụng hàm map() thay cho vòng lặp for và chuyển đổi kết quả về dạng danh sách qua hàm list().

Để ý cách sử dụng hàm map(): `map(parabolic, X)`. Trong đó, map() là một hàm built-in của Python. Bạn có thể trực tiếp sử dụng nó. Bạn truyền tên hàm parabolic chứ không truyền lời gọi hàm (lời gọi hàm sẽ là parabolic(x, a, b, c)).

Do hàm parabolic đòi hỏi 1 tham số, bạn chỉ cần truyền 1 iterable cho hàm map(). Nếu hàm yêu cầu nhiều hơn 1 tham số, bạn cần cung cấp số lượng iterable tương ứng. Ví dụ, nếu hàm cần 2 tham số, bạn phải cung cấp 2 iterable. Trong mỗi lượt thực hiện, map() sẽ lấy từ mỗi iterable một phần tử.

Về ý nghĩa, sử dụng hàm map() là tương tự như phiên bản imperative bạn đã thấy ở ví dụ 1: (1) lần lượt duyệt qua từng phần tử x của X; (2) ứng với mỗi phần tử x áp dụng hàm parabolic để thu được kết quả y; (3) thêm giá trị y vào object kết quả cuối cùng.

Sự khác biệt nằm ở chỗ: (1) quá trình được Python thực hiện tự động; (2) kết quả cuối cùng là một object kiểu map chứ không phải list. Do vậy bạn cần chuyển đổi từ kiểu map về kiểu list hoặc kiểu khác (nếu cần).

map() của Python tương tự như LINQ Select của C#.

Kiểu map cũng đồng thời là một generator nên bạn cũng có thể sử dụng nó trong vòng lặp for:

```
for y in Y:
    aprint(y, end=' ')
```

Giá trị của y cùng kiểu với kiểu trả về của hàm parabolic (float).

Khi sử dụng map() cần lưu ý:

1. Cú pháp chung như sau: `map(<hàm>, *<iterable>)`, tức là <hàm> sẽ lần lượt đem áp dụng cho từng phần tử của <iterable>. Bạn cần truyền tên hàm chứ không phải lời gọi hàm.
2. Ở vị trí <hàm> bạn có thể sử dụng hàm toàn cục hoặc phương thức của class. Không có gì khác biệt giữa chúng.

3. Phụ thuộc vào lượng tham số của <hàm>, bạn cần truyền số lượng <iterable> tương ứng. Nếu <hàm> có 1 tham số, bạn cần truyền 1 <iterable>. Nếu <hàm> có 2 tham số, bạn cần truyền 2 <iterable>.
4. Hàm map() sẽ lấy từng bộ giá trị từ các <iterable>. Ví dụ, với hàm 2 tham số (và 2 <iterable>), lượt 1 sẽ áp dụng <hàm> với phần tử 1 trong <iterable> thứ nhất và phần tử 1 trong <iterable> thứ hai, lượt 2 sẽ áp dụng <hàm> với phần tử 2 trong <iterable> thứ nhất và phần tử 2 trong <iterable> thứ hai,....
5. Kết quả trả về của hàm map() là một object kiểu map (<class 'map'>). Kiểu map cũng là một generator.
6. Phần tử của danh sách map sẽ có cùng kiểu với kiểu trả về của <hàm>.
7. Sử dụng map() không phải là phiên bản functional của for. Hàm map() chỉ quan tâm đến kết quả trả về của <hàm>. map() sẽ bỏ qua những hiệu ứng phụ (side-effect) như các lệnh in trong <hàm>.

* Một đặc thù của lập trình hàm là mọi hàm phải trả về kết quả. Trong Python, nếu hàm không có return, Python sẽ coi kết quả trả về của hàm là None. None trong Python là một từ khóa – giá trị. Các lệnh như xuất dữ liệu được gọi là hiệu ứng phụ (side-effect) trong lập trình hàm.

Hàm filter() trong Python

Hãy bắt đầu bằng một ví dụ:

Ví dụ 3 - filter.py

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def print(self):
        print(f'{self.name} ({self.age} years old)')
    @staticmethod
    def elder(person):
```

```
        if person.age >= 75:
            return True
        return False

presidents = [Person('Putin', 65),
               Person('Trump', 80),
               Person('Obama', 70),
               Person('Bush', 75),
               Person('Clinton', 55),
               Person('Yeltsin', 65),
               Person('Nixon', 90)]

def elder(person):
    if person.age >= 75:
        return True
    return False

if __name__ == '__main__':
    old_presidents = filter(Person.elder, presidents)
    for p in old_presidents:
        p.print()
```

Chạy chương trình cho kết quả:

```
Trump (80 years old)
Bush (75 years old)
Nixon (90 years old)
```

Trong ví dụ trên chúng ta xây dựng một class đơn giản Person với hai attribute name và age. Trong class này chúng ta định nghĩa static method elder. Phương thức này trả về giá trị True nếu age >= 75.

Để thử nghiệm, chúng ta tạo danh sách presidents chứa các object thuộc kiểu Person.

Yêu cầu đặt ra là cần lọc danh sách presidents để lấy ra những người từ 75 tuổi trở lên.

Theo logic thông thường (imperative), bạn sẽ dùng vòng lặp for trên presidents. Ứng với mỗi phần tử bạn áp dụng hàm elder() hoặc phương thức Person.elder(). Nếu thu được kết quả True, bạn sẽ đưa object tương ứng vào danh sách kết quả.

Thay vì làm thủ công như trên, trong Python bạn có thể áp dụng giải pháp của lập trình hàm với hàm filter(): `old_presidents = filter(Person.elder, presidents)`.

Cách sử dụng hàm filter() rất giống với hàm map() bạn đã làm quen ở phần trước. Tuy nhiên có vài điểm sau cần lưu ý:

1. filter() cũng là một **hàm built-in**, do vậy bạn có thể sử dụng nó ngay lập tức trong bất kỳ module nào.
2. Hàm filter() luôn yêu cầu **hai tham số: filter(<func>, <iterable>)**, trong đó <func> là hàm sẽ áp dụng trên danh sách <iterable>. Có thể là hàm toàn cục hoặc phương thức của class. Trong ví dụ 3 ở trên, ở vị trí của <func> bạn có thể dùng hàm elder() hoặc phương thức Person.elder(), ở vị trí <iterable> là danh sách presidents kiểu list.
3. **Kiểu phần tử** của <iterable> phải trùng với kiểu đầu vào cũng <func>. Điều này là đương nhiên vì <func> sẽ lần lượt nhận từng giá trị của <iterable> để kiểm tra. Trong ví dụ 3, phần tử của presidents có kiểu Person, và hàm elder()/phương thức Person.elder() đều xử lý tham số kiểu Person.
4. Hàm <func> phải trả về **kết quả kiểu boolean**. Nếu bạn để <func> trả về kết quả khác, filter() sẽ trả lại nguyên vẹn danh sách <iterable>. Trong ví dụ 3, elder()/Person.elder() trả về True nếu age >= 75 và False nếu ngược lại.
5. Do filter() chỉ nhận một <iterable>, hàm <func> cũng chỉ được phép **có 1 tham số bắt buộc**. Trong ví dụ 3, hàm elder()/Person.elder() chỉ nhận 1 tham số có kiểu Person.
6. Dữ liệu trả về của hàm filter() có **kiểu filter**. Hàm filter() áp dụng <func> trên từng phần tử của <iterable> và lấy tất cả các phần tử mà <func> trả về giá trị True. Các phần tử này được đưa vào một object có kiểu filter.

7. Kiểu ***filter cũng là một generator***, do đó bạn có thể sử dụng kết quả trong vòng lặp for hoặc chuyển nó thành các kiểu dữ liệu tập hợp phù hợp (như list).

Hàm filter() của Python hoạt động giống như phương thức Where của C# LINQ.

Hàm reduce() trong Python

Trước hết hãy xem ví dụ về hàm tính giai thừa của một số:

Ví dụ 4 - reduce.py

```
from functools import reduce

# imperative factorial function
def imp_fact(n: int):
    p = 1
    for i in range(1, n + 1):
        p *= i
    return p

# functional factorial function
def fun_fact(n: int):
    def mult(x, y): return x * y
    return reduce(mult, range(1, n + 1))

def test1():
    n = 3
    f = imp_fact(n)
    print(f'{n}! = {f}')

def test2():
    n = 3
    f = fun_fact(n)
    print(f'{n}! = {f}')

if __name__ == '__main__':
    test1()
    test2()
```

Trong ví dụ này chúng ta xây dựng hai hàm tính giai thừa, một hàm theo kiểu imperative (imp_fact) và một hàm theo kiểu functional (fun_fact).

Để ý trong hàm imp_fact, ở vòng lặp for chúng ta thực hiện phép toán nhân dồn (cumulative) các số liên tiếp trong khoảng [1, n+1) cho p để thu được giá trị giai thừa theo định nghĩa của phép toán này.

Phép toán nhân dồn (và các phép dồn tương tự) được thực hiện bởi hàm reduce() trong fun_fact(): `reduce(mult, range(1, n + 1))` với mult() là hàm nhân hai số được định nghĩa trước đó.

Khác biệt với map() và filter(), hàm reduce() không phải là hàm built-in. Để sử dụng hàm reduce(), bạn phải import nó từ package functools: `from functools import reduce`. Bạn nên đặt lệnh import ở đầu file script.

Cách hoạt động của reduce với mult và range(1, n+1) trong ví dụ trên như sau:

1. Giả sử $n = 4$, khi đó `range(1, n+1)` sẽ cho danh sách phần tử bao gồm các giá trị 1, 2, 3, 4.
2. Áp dụng mult cho hai phần tử đầu tiên: `mult(1, 2)` thu được kết quả 2.
3. Lấy kết quả trên (2) làm tham số thứ nhất và lấy phần tử tiếp theo của danh sách (3) làm tham số thứ hai cho mult: `mult(2, 3)` thu được kết quả 6.
4. Tiếp tục lấy kết quả trên (6) làm tham số thứ nhất, lấy phần tử tiếp theo của danh sách (4) làm tham số thứ hai cho mult: `mult(6, 4)` cho kết quả 24.
5. Danh sách không còn giá trị => kết thúc và trả lại giá trị 24.

Cách thức hoạt động của reduce hoàn toàn tương tự như cách chúng ta sử dụng phép nhân dồn trong vòng for bình thường:

```
# nhân dồn với vòng lặp
for i in range(1, n+1):
    p = p * i # hay p *= i
```

Khi sử dụng `reduce()` cần lưu ý:

1. Cú pháp chung của `reduce` là: `reduce(<func>, <iterable>)`.
2. `<func>` phải là một hàm có hai tham số.
3. Tham số của `<func>` phải cùng kiểu với giá trị trong `<iterable>`.
4. Kết quả trả về của `<func>` cũng phải cùng kiểu với giá trị trong `<iterable>` (và cùng kiểu với tham số đầu vào của chính `<func>`)
5. Kết quả trả về của `reduce` là một giá trị cùng kiểu với kết quả trả về của `<func>`.

Sử dụng hàm `lambda`

Trong các ví dụ trên bạn có thể thấy, `map()`, `filter()` và `reduce()` đều nhận một hàm khác làm tham số thứ nhất. Do vậy, bạn đều phải định nghĩa hàm tương ứng.

Tuy nhiên, trong rất nhiều trường hợp các hàm này đều đơn giản, thường chỉ có 1 biểu thức duy nhất, ví dụ như `return a*x*x + b*x + c` hay `return x * y`. Một đặc điểm khác nữa là các hàm này không được sử dụng ở những chỗ khác trong code.

Việc định nghĩa hàng loạt hàm "mini" sử dụng một lần như vậy làm code khó theo dõi.

Python cho phép định nghĩa trực tiếp hàm ở nơi cần dùng hàm tham số với **hàm `lambda`**.

Hãy cùng làm lại các ví dụ ở các phần trên:

```
def fun_fact(n: int):  
    # sử dụng hàm lambda x, y: x * y  
    return reduce(lambda x, y: x * y, range(1, n + 1))  
  
# sử dụng hàm lambda p : True if p.age >= 75 else False  
old_presidents = filter(lambda p: True if p.age >= 75 else False, presidents)  
  
# sử dụng hàm lambda x, a=1.0, b=0.0, c=0.0: a * x * x + b * x + c  
Y = map(lambda x, a=1.0, b=0.0, c=0.0: a * x * x + b * x + c, X)
```

Trong đoạn code trên chúng ta đã sử dụng một loạt hàm lambda:

```
lambda x, y: x * y  
lambda p : True if p.age >= 75 else False  
lambda x, a=1.0, b=0.0, c=0.0: a * x * x + b * x + c
```

Hàm lambda là những hàm không có tên và thân chỉ chứa 1 biểu thức duy nhất.

Cú pháp chung của hàm lambda là: `lambda <danh sách tham số> : <biểu thức kết quả>`

Trong đó, lambda là từ khóa bắt buộc. Danh sách tham số tương tự như của một hàm thông thường. Biểu thức kết quả không cần từ khóa `return`. Kết quả tính của biểu thức kết quả chính là kết quả thực hiện hàm lambda tương ứng.

Hàm lambda có thể được khai báo trực tiếp ở nơi cần dùng, do vậy tránh được nhu cầu xây dựng các hàm mini sử dụng một lần.

Kết luận

Trong phần này bạn đã làm quen với một số hàm thông dụng trong lập trình hàm:

1. `map()` áp dụng một hàm với danh sách. Có thể sử dụng `map()` thay cho vòng lặp `for`.
2. `filter()` áp dụng một hàm với danh sách và chỉ lấy ra những phần tử phù hợp. Có thể sử dụng `filter()` thay cho vòng lặp `for` và cấu trúc rẽ nhánh `if`.
3. `reduce()` áp dụng một hàm với danh sách để thu được một giá trị duy nhất. Có thể sử dụng `reduce()` cho phép toán cộng dồn, nhân dồn.
4. Có thể sử dụng hàm lambda làm tham số thay cho hàm thông thường.
5. Nhìn chung các phương pháp của lập trình hàm hơi khó hiểu hơn nhưng cách viết gọn nhẹ và xúc tích.

Lỗi và xử lý ngoại lệ trong Python

NỘI DUNG

1. Các loại lỗi trong Python
2. Cơ chế xử lý ngoại lệ mặc định của Python
3. Lớp ngoại lệ xây dựng sẵn trong Python
4. Tự xử lý ngoại lệ với try .. except
5. Sử dụng cấu trúc try .. except
6. Kết luận

Các loại lỗi trong Python

Trong Python có thể phân biệt hai loại lỗi: lỗi cú pháp và lỗi thực thi.

Lỗi cú pháp (syntax error) là loại lỗi phát sinh do viết code không tuân thủ theo quy tắc của ngôn ngữ lập trình. Lấy ví dụ:

```
>>> print "hello" #lỗi cú pháp khi sử dụng hàm print()
```

SyntaxError: Missing parentheses in call to 'print'. Did you mean `print("hello")`?

Trong Python 2, đây là một lệnh chính xác. Tuy nhiên đây lại là một lỗi cú pháp trong Python 3. Python 3 chuyển print từ một lệnh (statement) trở thành một hàm thông thường. Do vậy phải gọi print với cú pháp dành cho lời gọi hàm: `print('Hello world!')`.

Loại lỗi này thường gặp ở những người mới học lập trình Python. Đôi khi, chúng cũng xuất hiện do người viết code vô tình gõ nhầm. Lỗi này cũng hay xảy ra khi chạy code viết cho Python 2.x trong Python 3.x (và ngược lại) do hai phiên bản này có một số khác biệt về cú pháp.

Lỗi cú pháp thường dễ phát hiện và sửa chữa. Trình thông dịch Python hoặc các công cụ hỗ trợ của IDE có thể phát hiện ra lỗi cú pháp ngay từ lúc viết code. Khi gặp lỗi cú pháp, trình thông dịch Python sẽ đưa ra thông báo **SyntaxError** với thông tin chi tiết về lỗi liên quan.

Do đặc thù của ngôn ngữ thông dịch, dù có lỗi cú pháp, Python vẫn thực thi file script nhưng sẽ dừng lại ở vị trí lỗi. Điều này khác biệt với các ngôn ngữ biên dịch (như C#) – dừng biên dịch khi gặp lỗi cú pháp, chương trình chỉ chạy sau khi biên dịch.

Lỗi thực thi (runtime error) là loại lỗi phát sinh trong quá trình chạy chương trình. Lỗi thực thi cũng được gọi là **ngoại lệ** (exception). Ví dụ, trong chương trình bạn vô tình thực hiện phép chia cho 0, hoặc truy xuất một file không tồn tại trên ổ đĩa.

Đây là loại lỗi liên quan đến logic của chương trình. Loại lỗi này rất khó phát hiện. Ít nhất thì bạn không thể phát hiện chúng ở giai đoạn viết code. Chúng cũng chỉ xuất hiện khi thực thi trong những điều kiện nhất định.

Ứng với mỗi loại lỗi thực thi, Python sử dụng một class riêng để mô tả. Ví dụ, để mô tả lỗi chia cho 0, Python sử dụng class `ZeroDivisionError`. Python cung cấp rất nhiều class như vậy. Chúng được gọi chung là ngoại lệ xây dựng sẵn (**built-in exception**). Các class này có đặc điểm là tên gọi kết thúc bằng 'Error'.

Một dạng "lỗi" đặc biệt của Python được gọi là **lỗi đánh giá** (assert error). Đây là một loại "lỗi" liên quan đến việc giá trị của một biểu thức không đạt yêu cầu.

Ví dụ, bạn yêu cầu người dùng nhập tuổi. Do tuổi phải là một số nguyên dương, nếu người dùng nhập một giá trị âm, đây có thể xem là một assert error.

Assert error được dùng trong debug và test. Python cung cấp một cú pháp riêng cho assert error.

Cơ chế xử lý ngoại lệ mặc định của Python

Một cách thức xử lý chung mà Python (và các ngôn ngữ lập trình khác) áp dụng khi gặp loại lệ là **dừng thực hiện chương trình và phát ra thông tin về lỗi** mà nó gặp phải. Đây là cách thức **xử lý ngoại lệ** (exception handling) mặc định của Python.

Trong cơ chế xử lý ngoại lệ mặc định, Python sẽ

- (1) dừng thực hiện chương trình ở lệnh lỗi,
- (2) tạo object của class exception tương ứng,
- (3) in ra thông báo lỗi từ object exception với thông tin chi tiết về vị trí và loại lỗi.

Ví dụ, khi gặp phải lỗi chia cho 0, Python sẽ dừng thực thi tại dòng lệnh lỗi và phát ra thông báo `ZeroDivisionError: division by zero`. Cụ thể hơn, nếu bạn chạy file script sau:

```
a = 10
b = 0
print(a/b)
print(a)
print(b)
```

Bạn sẽ gặp thông báo lỗi:

```
Traceback (most recent call last):
  File "exception.py", line 3, in
    print(a/b)
ZeroDivisionError: division by zero
```

Theo thông báo này, trong file `exception.py`, dòng thứ 3, lệnh `print(a/b)` gặp lỗi chia cho 0.

Từ thông báo này bạn có thể biết chính xác lỗi ở đâu và lỗi gì. Nó rất có ích ở giai đoạn debug chương trình.

Lớp ngoại lệ xây dựng sẵn trong Python

Ứng với mỗi loại lỗi thực thi, Python sử dụng một class riêng để mô tả. Ví dụ, để mô tả lỗi chia cho 0, Python sử dụng class `ZeroDivisionError`. Để mô tả lỗi truy xuất file không tồn tại, Python sử dụng class `FileNotFoundError`.

Python cung cấp rất nhiều class như vậy. Chúng được gọi chung là ngoại lệ xây dựng sẵn (**built-in exception**). Các class này có đặc điểm là tên gọi kết thúc bằng 'Error'.

Dưới đây là một số lớp ngoại lệ thường gặp.

IndexError – xuất hiện khi truy xuất phần tử của danh sách theo chỉ số

```
>>> # IndexError - xuất hiện khi truy xuất phần tử của danh sách theo chỉ số
>>> L1=[1,2,3]
>>> L1[3] # không có chỉ số 3
```

Traceback (most recent call last):

File "<pyshell#18>", line 1, in <module>

L1[3]

IndexError: list index out of range

ModuleNotFoundError – xuất hiện khi import một module không tồn tại

>>> # ModuleNotFoundError - xuất hiện khi import một module không tồn tại

>>> **import** notamodule # không có module này

Traceback (most recent call last):

File "<pyshell#10>", line 1, in <module>

import notamodule

ModuleNotFoundError: No module named '**notamodule**'

ImportError – xuất hiện khi định danh trong lệnh import không chính xác

>>> **from** *math* **import** cube # không có đối tượng nào tên là cube trong module math

Traceback (most recent call last):

File "<pyshell#16>", line 1, in <module>

from *math* **import** cube

ImportError: cannot **import** name '**cube**'

TypeError – xuất hiện khi thực hiện phép toán trên những kiểu dữ liệu không phù hợp

>>> '**2**' + 2 # không thể cộng một chuỗi với một số

Traceback (most recent call last):

File "<pyshell#23>", line 1, in <module>

'**2**' + 2

TypeError: must be str, **not** int

ValueError – xuất hiện khi biến đổi kiểu

>>> **int**('xyz') # không thể chuyển đổi chuỗi 'xyz' thành kiểu số

Traceback (most recent call last):

File "<pyshell#14>", line 1, in <module>

```
int('xyz')
```

ValueError: invalid literal **for** int() **with** base 10: 'xyz'

NameError – xuất hiện khi sử dụng một định danh chưa được định nghĩa

```
>>> age # định danh 'age' chưa được định nghĩa
```

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

age

NameError: name 'age' is **not** defined

Danh sách tất cả các ngoại lệ xây dựng sẵn của Python:

<https://docs.python.org/3.8/library/exceptions.html>

Tự xử lý ngoại lệ với try .. except

Cách thức xử lý ngoại lệ mặc định chỉ phù hợp ở giai đoạn debug. Nếu để cơ chế này hoạt động trên ứng dụng đã triển khai sẽ rất không phù hợp, tạo cảm giác thiếu ổn định cho ứng dụng. Ngoài ra, người dùng cuối thường cũng không biết làm gì với các thông báo lỗi đó.

Vì vậy, trước khi triển khai ứng dụng, bạn cần xây dựng cơ chế xử lý ngoại lệ riêng để đảm bảo ứng dụng hoạt động ổn định.

Cơ chế xử lý ngoại lệ riêng sẽ ngắt chế độ xử lý ngoại lệ mặc định trên một nhóm code nhất định và thay thế bằng cách thức xử lý khác do người lập trình tự xây dựng.

Python cung cấp cú pháp try .. except để bạn tự mình xử lý ngoại lệ. Cú pháp chung của cấu trúc này như sau:

```
try:
    # thực hiện các lệnh có nguy cơ lỗi
except:
    # thực hiện nếu phát sinh lỗi trong khối try
else:
    # thực hiện nếu KHÔNG phát sinh lỗi trong khối try
finally:
    # LUÔN thực hiện, dù trong khối try có phát sinh lỗi hay không
```

Cấu trúc này bao gồm 4 khối: `try`, `except`, `else`, `finally`. Trong đó `try` và `except` là bắt buộc, `else` và `finally` là tùy chọn.

Khối `try` chứa các lệnh nguy hiểm. Lệnh nguy hiểm là những lệnh tiềm ẩn khả năng phát sinh lỗi thực thi. Ví dụ, phép chia là một lệnh nguy hiểm vì tiềm ẩn khả năng chia cho 0. Truy xuất file là một lệnh nguy hiểm vì tiềm ẩn khả năng truy xuất file không tồn tại... Cấu trúc `try .. except` bắt buộc phải có và chứa duy nhất 1 khối `try`.

Khối `except` chứa các lệnh sẽ được thực thi nếu phát sinh lỗi trong khối `try`. Cấu trúc này bắt buộc phải có ít nhất một khối `except`. Có thể xây dựng nhiều khối `except` trong cùng một cấu trúc `try .. except`. Chúng ta sẽ nói kỹ hơn về cách xây dựng khối `except` trong phần tiếp theo.

Khối `else` chứa các lệnh sẽ được thực thi nếu KHÔNG phát sinh lỗi trong khối `try`. Nói cách khác, khối `else` và khối `except` luôn đối lập nhau: nếu đã thực hiện `except` (có lỗi) thì không thực hiện `else`; nếu không thực hiện `except` (không có lỗi) thì sẽ thực hiện `else`. Khối `else` là không bắt buộc.

Khối `finally` chứa các lệnh luôn luôn được thực thi, bất kể có lỗi hay không. Đây cũng là khối không bắt buộc.

Sử dụng cấu trúc `try .. except`

Ở phần trước chúng ta đã nói đến lý thuyết chung về cấu trúc `try .. except`. Tiếp theo đây chúng ta sẽ nói chi tiết về cách sử dụng cấu trúc này.

Hãy cùng thực hiện một ví dụ:

Ví dụ 1

```
try:
    a=5
    b='0'
    print(a/b)
except:
    print('Xảy ra lỗi gì đó.')
print("Ngoài khối try .. except.")
```

Đoạn code trên là ví dụ sử dụng cấu trúc try .. except cơ bản nhất. Cách sử dụng này bắt tất cả các loại lỗi có thể phát sinh trong khối try. Nói cách khác, lỗi sử dụng này sẽ thực thi khối except nếu có bất kỳ lỗi gì phát sinh trong khối try. Tuy nhiên, ở trong khối except không biết bất kỳ thông tin gì về lỗi xảy ra. Cách sử dụng này thực tế sẽ ẩn đi bất kỳ lỗi thực thi nào. Nhìn chung chúng ta *không nên sử dụng lỗi viết này*.

Hãy xem một ví dụ khác:

Ví dụ 2

```
try:
    a=5
    b='0'
    print (a+b)
except TypeError:
    print('Phép toán không hợp lệ')
print("Ngoài khối try .. except.")
```

Sự khác biệt của lỗi sử dụng này là có thêm tên kiểu ngoại lệ trong phần except.

Với lỗi viết này chỉ những lỗi liên quan đến TypeError (lỗi không đồng bộ kiểu trong phép toán) mới bị bắt. Các loại lỗi khác sẽ bị bỏ qua. Nghĩa là, chỉ khi nào phát sinh lỗi TypeError thì khối code except mới được thực thi. Nếu phát sinh lỗi khác thì vẫn thực hiện cơ chế xử lý lỗi mặc định của Python.

Bạn cũng có thể ghép nhiều khối except với nhau như sau:

```
try:
    a=5
    b=0
    print (a/b)
except TypeError as te:
    print(f'Unsupported operation: {te.args}')
except ZeroDivisionError as zde:
    print (f'Division by zero not allowed: {zde.args}')
print ('Out of try except blocks')
```

Khi này mỗi khối `except` sẽ chịu trách nhiệm cho một lỗi cụ thể.

Bạn có thể sử dụng thông tin chi tiết của lỗi như sau:

```
try:
    print("try block")
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")
    x=0
    y=0
print ("Out of try, except, else and finally blocks." )
```

Hãy để ý phần `as <biến>` ở phần header của mỗi khối `except`. Với lỗi viết này, object exception sẽ được trả tới bởi `<biến>` tương ứng và bạn có thể sử dụng `<biến>` trong khối `except`.

Cuối cùng, bạn có thể xem cách sử dụng khối `else` và `finally` như sau:

```
try:
    print("try block")
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
```



```
print("else block")

print("Division = ", z)

finally:

    print("finally block")

    x=0

    y=0

print ("Out of try, except, else and finally blocks." )
```

Kết luận

Trong phần này bạn đã làm quen với các loại lỗi và cơ chế xử lý ngoại lệ trong Python:

- Python có hai loại lỗi chính là lỗi cú pháp và lỗi thực thi, trong đó lỗi thực thi còn được gọi với tên quen thuộc là ngoại lệ (exception).
- Ứng với mỗi loại lỗi Python xây dựng một class riêng, gọi là các exception class.
- Python cung cấp cơ chế xử lý ngoại lệ mặc định bằng cách dừng chương trình và thông báo lỗi và vị trí gây lỗi.
- Nếu không muốn sử dụng cơ chế xử lý lỗi mặc định, bạn có thể tự xây dựng cách xử lý ngoại lệ riêng với cấu trúc try – except – else – finally.