LẬP TRÌNH PYTHON DÀNH CHO KỸ THUẬT

TÀI LIỆU THAM KHẢO

Lưu hành nội bộ

BIÊN SOẠN: NGUYỄN ĐÌNH QUÝ - BỘ MÔN KHOA HỌC MÁY TÍNH - ĐẠI HỌC XÂY DỰNG HÀ NỘI

MỤC LỤC

MỞ ĐẦU

Lập trình là việc điều khiển máy tính thực hiện một nhiệm vụ nào đó bằng các dòng lệnh. Lập trình được sinh ra để giải quyết các vấn đề từ đơn giản đến phức tạp để tiết kiệm thời gian của con người. Lập trình được ứng dụng rộng rãi trong nhiều lĩnh vực khác nhau đặc biệt là trong các lĩnh vực về Khoa học - Kỹ thuật. Ngày nay, lập trình không chỉ là công việc của các lập trình viên, ai cũng đều nên biết về lập trình và biến nó thành trợ thủ đắc lực cho công việc của mình. Huyền thoại Steve Jobs đã từng nói rằng: “Ai cũng nên học lập trình“.

Xuất phát từ những lý do quan trọng của việc học lập trình, tôi biên soạn tài liệu "Lập trình python dùng trong kỹ thuật" tập trung vào việc giới thiệu và hướng dẫn sử dụng python - ngôn ngữ phổ biến vì đơn giản và dễ tiếp cận với tất cả mọi người và giới thiệu, hướng dẫn sử dụng các thư viện dùng trong các ngành kỹ thuật như **numpy** và **scipy**.

Ngoài ra cuốn sách cũng cung cấp jupyter notebook đi kèm từng phần để người đọc dễ dàng thực hành các nội dung trong cuốn sách.

Trong tài liệu này, tôi có tham khảo một số giáo trình nước ngoài về python và bài viết của một số blog. Tài liệu mang tính tổng hợp và biên soạn để người học có thêm tài nguyên để tiếp cận với lập trình python.

Chương 1

Chương này tập trung vào giới thiệu về Python, các phiên bản và sự so sánh giữa các phiên bản. Ngoài ra hướng dẫn cách cài đặt Python để người mới tiếp cận có thể dễ dàng cài đặt được để sử dụng.

1.1 Mô hình tính toán.

1.1.1 Giới thiệu

Ngày nay, các hệ thống và quy trình phức tạp đang được nghiên cứu và phát triển thông qua các mô phỏng trên máy tính, các mẫu máy bay mới như A380 được thiết kế và thử nghiệm hoàn toàn thông qua việc mô phỏng trên máy tính. Với khả năng tính toán ngày càng cao, các siêu máy tính, cụm máy tính và thậm chí cả các máy tính cá nhân như máy bàn hay laptop và điện thoại di động có tốc độ xử lý ngày càng được nâng cấp vượt bậc giúp cho các mô hình tính toán có thể thực hiện được nhiều nhiệm vụ phức tạp.

1.1.2 Mô hình tính toán

Để nghiên cứu một quy trình bằng mô phỏng máy tính, chúng ta phân biệt hai bước: bước đầu tiên là phát triển một mô hình của hệ thống thực. Khi nghiên cứu chuyển động của một vật nhỏ, chẳng hạn như một đồng xu, dưới tác dụng của trọng lực, chúng ta có thể bỏ qua ma sát của không khí: mô hình của chúng ta -chỉ phụ thuộc lực hấp dẫn và quán tính của đồng xu, tức là (t) = F / m = −9,81m / s2 - là một giá trị gần đúng của hệ thực.

Mô hình thông thường sẽ cho phép chúng ta thể hiện hành vi của hệ thống (ở một số dạng gần đúng) thông qua các phương trình toán học, thường liên quan đến phương trình vi phân thông thường (ODE) hoặc phương trình vi phân riêng (PDE).

Trong khoa học tự nhiên như vật lý, hóa học và kỹ thuật liên quan, thường không quá khó để tìm một mô hình phù hợp, mặc dù các phương trình kết quả có xu hướng rất khó giải và trong hầu hết các trường hợp có thể không giải được bằng phân tích.

Mặt khác, trong các đối tượng không được mô tả bằng toán học và phụ thuộc vào hành vi của các đối tượng mà hành động của chúng không thể dự đoán một cách xác định (chẳng hạn như con người), thì việc tìm ra một mô hình tốt để mô tả thực tế sẽ khó hơn nhiều. Theo quy luật chung, trong những nguyên tắc này, các phương trình kết quả dễ giải hơn, nhưng chúng khó tìm hơn và cần phải đặt câu hỏi về tính hợp lệ của một mô hình . Ví dụ điển hình là các nỗ lực mô phỏng nền kinh tế, việc sử dụng các nguồn lực toàn cầu, hành vi của một đám đông đang hoảng loạn, mô phỏng quá trình lây nhiễm để dự đoán số người mắc bệnh trong đại dịch, v.v.

Cho đến nay, chúng ta chỉ mới thảo luận về sự phát triển của các mô hình để mô tả thực tế, và việc sử dụng các mô hình này không nhất thiết phải liên quan đến máy tính. Trên thực tế, nếu phương trình của mô hình có thể được giải bằng cách phân tích, thì người ta nên làm điều này và viết ra lời giải cho phương trình.

Trong thực tế, hầu như không phải bất kỳ phương trình mô hình nào của các hệ thống có thể được giải thích bằng cách phân tích, và lúc máy tính ra đời: sử dụng phương pháp số, ít nhất chúng ta có thể nghiên cứu mô hình cho một tập hợp các điều kiện biên cụ thể. Đối với ví dụ đã xét ở trên, chúng ta có thể không dễ dàng nhận thấy từ một nghiệm số rằng vận tốc của đồng xu dưới tác dụng của trọng lực sẽ thay đổi tuyến tính theo thời gian (chúng ta có thể đọc dễ dàng từ giải pháp phân tích có sẵn cho hệ thống đơn giản này: $v(t) = t \cdot 9,81 m/s^2 + v_0$.

Giải pháp số có thể được tính toán bằng máy tính sẽ bao gồm dữ liệu cho thấy vận tốc thay đổi như thế nào theo thời gian đối với một vận tốc ban đầu cụ thể v_0 (v_0 là điều kiện biên).

Chương trình máy tính sẽ tổng hợp một danh sách gồm cặp giá trị (1) thời gian t_i và giá trị (2) giá trị cụ thể của vận tốc v_i đã tại thời gian t_i. Bằng cách lập bảng tất cả các giá trị t_i và v_i ta sẽ tìm ra một đường cong thông qua dữ liệu, chúng ta có thể hiểu được xu hướng từ dữ liệu (dĩ nhiên là chúng ta có thể thấy được xu hướng từ giải pháp phân tích).

Tên gọi mô hình tính toán bắt nguồn từ hai bước: (i) lập mô hình, tức là tìm mô tả mô hình của một hệ thống thực và (ii) giải các phương trình mô hình kết quả bằng cách sử dụng các phương pháp tính toán vì đây là cách duy nhất có thể giải được các phương trình.

1.1.3. Lập trình hỗ trợ mô hình tính toán

Có rất nhiều thư viện lập trình hỗ trợ giải quyết các mô hình tính toán. Để đáp ứng nhu cầu nghiên cứu hoặc trực quan hóa dữ liệu (visualizing) người dùng không cần quá nhiều về kiến thức lập trình. Trong môi trường nghiên cứu và học thuật, mọi người thường sử dụng các thư viện có sẵn, tuy nhiên nhiều bài toán vẫn cần phải thực hiện lập trình nếu các thư viện có sẵn không giải quyết được.

Khi đó, kỹ năng lập trình là bắt buộc. Nói chung sẽ rất hữu ích nếu hiểu hơn về về cách xây dựng phần mềm và các ý tưởng cơ bản về kỹ thuật phần mềm khi chúng ta sử dụng ngày càng nhiều thiết bị được điều khiển bằng phần mềm.

Người ta thường cho rằng không có gì máy tính có thể làm được mà con người chúng ta không thể làm được. Tuy nhiên, máy tính có thể làm điều đó nhanh hơn nhiều và cũng ít mắc lỗi hơn nhiều. Do đó, không có phép tính nào mà một máy tính thực hiện mà không thể được thực hiện bởi con người nhưng có thể mất nhiều thời gian để thực hiện hơn máy tính.

Hiểu cách xây dựng một mô phỏng máy tính bao gồm: (i) tìm mô hình (thường điều này có nghĩa là tìm các phương trình thể hiện một cách đúng đắn), (ii) biết cách giải các phương trình này bằng tính toán số học, (ii) triển khai các hàm tính toán để thực hiện giải pháp (bằng lập trình).

1.2. Vì sao sử dụng python

Trọng tâm thiết kế trên ngôn ngữ Python là sự đơn giản, ví dụ như:

- **Ngôn ngữ lập trình đơn giản, dễ học** : Python có cú pháp rất đơn giản, rõ ràng. Nó dễ đọc và viết hơn rất nhiều so với những ngôn ngữ lập trình khác như C++, Java, C#. Python làm cho việc lập trình trở nên thú vị. Nó cho phép bạn tập trung vào những giải pháp chứ không phải cú pháp.
- **Miễn phí – mã nguồn mở**: Bạn có thể tự do sử dụng và phân phối Python, thậm chí là dùng nó cho mục đích thương mại. Python có một cộng đồng rộng lớn, không ngừng cải thiện nó mỗi lần cập nhật.
- **Khả năng đa nền tảng**: Các chương trình Python có thể di chuyển từ nền tảng này sang nền tảng khác và chạy nó mà không có bất kỳ thay đổi nào. Nó chạy liền mạch trên hầu hết tất cả các nền tảng như Windows, macOS, Linux.
- **Khả năng mở rộng và có thể nhúng**: Giả sử một ứng dụng đòi hỏi sự phức tạp rất lớn, bạn có thể dễ dàng kết hợp các phần code bằng C, C++ và những ngôn ngữ khác (có thể gọi được từ C) vào code Python.

- **Ngôn ngữ thông dịch cấp cao**: Không giống như C/C++, với Python, bạn không phải lo lắng những nhiệm vụ khó khăn như quản lý bộ nhớ, dọn dẹp những dữ liệu vô nghĩa,... Khi chạy code Python, nó sẽ tự động chuyển đổi code sang ngôn ngữ máy tính có thể hiểu. Bạn không cần lo lắng về bất kỳ hoạt động ở cấp thấp nào.
- **Thư viện tiêu chuẩn lớn để giải quyết những tác vụ phổ biến**: Python có một số lượng lớn thư viện tiêu chuẩn giúp cho công việc lập trình của bạn trở nên dễ thở hơn rất nhiều, đơn giản vì không phải tự viết tất cả code. Ví dụ: Bạn cần kết nối cơ sở dữ liệu MySQL trên Web server? Bạn có thể nhập thư viện MySQLdb và sử dụng nó. Những thư viện này được kiểm tra kỹ lưỡng và được sử dụng bởi hàng trăm người. Vì vậy, bạn có thể chắc chắn rằng nó sẽ không làm hỏng code hay ứng dụng của mình.
- **Hướng đối tượng**: Mọi thứ trong Python đều là hướng đối tượng. Lập trình hướng đối tượng (OOP) giúp giải quyết những vấn đề phức tạp một cách trực quan. Với OOP, bạn có thể phân chia những vấn đề phức tạp thành những tập nhỏ hơn bằng cách tạo ra các đối tượng.

Vì Python là một ngôn ngữ thông dịch và chạy chậm hơn nhiều lần so với mã đã biên dịch, nên chúng ta có thể hỏi tại sao lại sử dụng một ngôn ngữ 'chậm' như vậy để mô phỏng máy tính? Lý do là việc viết code đôi khi mất thời gian hơn là việc thực thi, nên chúng ta có thể chấp nhận việc code chạy chậm hơn một chút thay vì sử dụng ngôn ngữ lập trình phức tạp mất nhiều thời gian để xây dựng chương trình.

1.3. Các phiên bản python

Hiện nay cộng đồng đang sử dụng 2 phiên bản phổ biến đó là python 3.x và python 2.x. Có chút khác biệt giữa 2 phiên bản.

Python 2

Python sử dụng một trình thông dịch để chạy mã. Không giống như ngôn ngữ dựa trên trình biên dịch, trình thông dịch Python không duyệt toàn bộ mã cùng một lúc. Thay vào đó, nó đọc từng dòng một và nếu trình thông dịch tìm thấy lỗi, nó sẽ dừng trước đó và đưa ra thông báo lỗi cho người dùng. Python 2 đã tồn tại lâu hơn, vì vậy nó có nhiều thư viện hơn. Phiên bản phổ biến nhất của Python 2 là Python 2.7

wiki.python.org đã có một bài viết rất chi tiết về sự khác biệt giữa 2 phiên bản Python, mỗi phiên bản đều có những ưu điểm so với phiên bản kia, tuy nhiên:

“Python 2.x is legacy, Python 3.x is the present and future of the language”.

Có thể hiểu rằng Python 3.x được thiết kế để trở thành tương lai của ngôn ngữ Python. Tuy nhiên Python 2.x đã được cộng đồng tin dùng tới mức có lẽ phải mất khá nhiều thời gian nữa để Python 3.x có thể hoàn toàn thay thế được người đàn anh “legacy”.

Để hiểu rõ hơn về câu chuyện Python 2&3 này, chúng ta sẽ bắt đầu một chút từ lịch sử của Python. Được phát triển từ cuối những năm 1980 và lần đầu tiên được biết đến rộng

rãi vào năm 1991, cha đẻ của Python Guido van Rossum đã tạo ra ngôn ngữ này khi mà ông ấy quá rảnh rỗi trong dịp giáng sinh. Từ thời điểm diễn đàn về Python (comp.lang.python) được mở ra trên Usenet vào năm 1994, Python đã nhanh chóng trở thành một trong những ngôn ngữ phổ biến nhất trong cộng đồng lập trình viên mã nguồn mở.

Có lẽ bắt đầu từ việc chưa hài lòng với một phiên bản ngôn ngữ được thiết kế hơi lộn xộn trong lúc "rảnh rỗi", Guido van Rossum luôn muốn dọn dẹp Python 2.x cho đẹp đẽ và ngăn nắp hơn qua nhiều phiên bản. Tuy nhiên việc phải luôn tương thích ngược với các phiên bản cũ hơn đã cản trở những nâng cấp toàn diện của ngôn ngữ này, bao gồm cả việc tăng hiệu suất, thêm tính năng cũng như thay đổi cú pháp ...

Python3.x được tạo ra như một rẽ nhánh mới và không tương thích với phiên bản cũ. Phiên bản đầu tiên của Python 3.x được phát hành vào 2008, phiên bản cuối cùng của Python 2.x (Python 2.7) được phát hành vào giữa năm 2010, đây được coi là phiên bản kết thúc của Python 2.x, từ sau phiên bản này không có thêm cập nhật lớn nào nữa cho Python 2.x.

Trong khi đó, Python 3.x vẫn được tiếp tục phát triển và hoàn thiện, nhiều phiên bản ổn định đã được phát hành bao gồm: v3.3 (2012), v3.4 (2014), v3.5 (2015), v3.7 (2017).

Nên sử dụng phiên bản nào? Tùy vào mục đích sử dụng cũng như môi trường chạy chương trình của bạn.

Trong trường hợp bạn không thể update lên Python 3.x (khi bạn làm việc với một server đang chạy ổn định Python 2.x chẳng hạn), hoặc khi một thư viện nào đó quan trọng chưa hỗ trợ Python 3.x, đó là khi bạn sử dụng Python 2.x

Trong trường hợp bạn muốn sử dụng những tính năng mới của Python 3.x (dĩ nhiên), thì tại sao bạn lại không thử "future of Python".

Nói tóm gọn, Python 2.x có thư viện và cộng đồng, Python 3.x có tương lai.

Nếu để lựa chọn cho người mới bắt đầu, thì hiển nhiên Python 3 sẽ có lợi thế hơn. Trên thực tế thì số lượng thư viện hỗ trợ Python 3 đã rất nhiều, số còn lại thường là các thư viện lỗi thời, ít người dùng hoặc không được duy trì phát triển nữa.

Python 2.x có phải đã quá cũ và nhiều lỗi?

Không hẳn, rất nhiều cải tiến của Python 3.x đã được cập nhật ngược lại cho phiên bản Python 2.x (Python 2.6, Python 2.7). Về cơ bản, nếu bạn viết code tốt thì sẽ không có nhiều khác biệt khi sử dụng Python 2.x và Python 3.x.

Python 2 và 3, những điểm khác biệt cần chú ý Khác biệt giữa Python 2 và 3 không phải chỉ là về mặt cú pháp, có rất nhiều thay đổi ở bên trong giúp cho Python trở thành ngôn ngữ "tốt hơn" qua từng phiên bản.

1.4 Cài đặt Python

Cài đặt và chạy Python trên Windows Tải Python tại đây:
https://www.python.org/downloads/, chọn phiên bản bạn cần.

Nhấp đúp vào file vừa tải về để cài đặt. Tại đây có 2 tùy chọn, bạn chọn một cái để cài.

Install Now: Mặc định cài Python vào ổ C, cài sẵn IDLE (cung cấp giao diện đồ họa để làm việc với Python), pip và tài liệu, tạo shortchut,... Customize installation: Cho phép bạn chọn vị trí cài và tính năng cần thiết.

Cài đặt và chạy Python trên Ubuntu Cài đặt các denpendency sau:

```
sudo apt-get install build-essential `
checkinstall
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev
libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-
dev
```

Tải Python tại đây: https://www.python.org/downloads/

Trong terminal, đi đến thư mục chứa file tải về và chạy lệnh sau để giải nén file. Tên file sẽ khác nhau tùy thuộc vào bản bạn tải về.

```
tar -xvf Python-3.6.2.tgz
```

Đi đến thư mục đã giải nén, nhập lệnh:

```
cd Python-3.6.0
```

Thực hiện các lệnh sau để biên dịch mã nguồn Python trên Ubuntu.

```
./configure
make
make install
```

Chương 2: Tính toán python

2.1 Chạy chương trình đầu tiên

Python là ngôn ngữ thông dịch. Chúng ta có thể viết chương trình bằng cách viết các lệnh trong tập tin văn bản và lưu tập tin lại dưới dạng chương trình Python có đuôi mở rộng là **.py**. Ví dụ `hello.py` Hoặc chúng ta có thể viết lệnh python trực tiếp trong cửa sổ dòng lệnh của Python. Python sẽ ngay lập tức thực hiện lệnh và xuất kết quả ra màn hình. Cách này hữu ích cho chúng ta trong việc học và hiểu về Python. Luồng thực hiện của Python trong cửa sổ dòng lệnh sẽ là đọc lệnh (Reading), thực hiện tính toán (Evaluating), in kết quả đã tính toán ra màn hình (Printing), sau đó lặp đi lặp lại luồng thực hiện đó (Loop). Đây chính là cấu trúc vòng lặp REPL (Reading-Evaluating-Printing-Loop) trong python. Để mở trình thông dịch và chạy chương trình chúng ta làm như sau:

- Trên Windows: Mở IDLE (vào start, gõ tìm IDLE, sau đó click để mở)
- Hoặc trong windows, bạn mở cửa sổ dòng lệnh MS-DOS, sau đó gõ python.exe để mở.
- Trên Linux/Unix hoặc Mac OSX: Bạn mở phần mềm có tên là `terminal` trong Applications/Utilities, trong cửa sổ dòng lệnh terminal, bạn gõ `python`, sau đó nhấn Enter.

Khi cửa sổ dòng lệnh python hiện ra, dòng đầu tiên sẽ có ký tự `>>>` để chờ người dùng gõ câu lệnh cho python thực hiện.

```
>>>
```

Chúng ta có thể thực hiện thử các câu lệnh, ví dụ `2+8`. Sau đó nhấn Enter để thực hiện lệnh:

```
>>> 2+8
10
>>>
```

Khi chúng ta nhấn enter, Python sẽ thực hiện tính toán biểu thức (2+8) và hiển thị kết quả đã tính được (10) ra dòng tiếp theo. Sau đó tiếp tục hiển thị lời nhắc `>>>` để người dùng nhập lệnh tiếp theo. Việc thực hiện câu lệnh trong màn hình console thường được gọi là vòng lặp REPL, bởi vì biểu thức được đọc, tính toán, in kết quả ra màn hình sau đó vòng lặp tiếp theo được thực hiện.

2.2 Tính toán trong python

Một số toán tử cơ bản trong python có thể kể ra như: cộng (+), trừ (-), nhân (*), chia (/) và toán tử luỹ thừa (**)

> Toán tử trong lập trình là công cụ dùng để thực hiện các phép tính hoặc chức năng giống như hàm nhưng có hình thức khác với hàm, được thể hiện dưới dạng ký hiệu, biểu tượng nhằm thực hiện các biểu thức.

```
>>> 10+10000 10010
>>> 42 -1.5
40.5
>>> 47*11 517
>>> 10/0.5 20.0
>>> 2**2
4
>>> 2**3 8
>>> 2**4 16
>>> 2+2
4
>>> # Đây là ký tự dùng để comment
... 2+2
4
>>> 2+2 # đây là comment trên cùng dòng code
4
```

Trong đoạn câu lệnh trên các bạn có thể thấy ký tự dùng để comment code. Khi gặp ký tự comment `(#)`, python sẽ bỏ qua và không thực hiện câu lệnh sau dấu `#`. Comment thường được dùng để chú thích các dòng code, giúp việc tài liệu hoá được dễ dàng, giúp ích nhiều cho việc bảo trì và đọc lại code sau này. Ngoài ra có thể tính căn bậc n của một số $\sqrt[n]{x} = x^{1/n}$, chúng ta có thể tính $\sqrt{3} = 1.732050...$ bằng cách sử dụng toán tử **

```
>>> 3**0.5
1.732050...
```

Hoặc có thể dùng ngoặc đơn để nhóm như sau:

```
>>> 2*10+5
25
>>> 2*(10+5)
30
```

2.3 Phép chia số nguyên

Chúng ta sẽ thử phép chia với số nguyên như sau:

```
>>> 15/6
3
```

Rõ ràng kết quả phép chia 15/6 phải là 2.5 nhưng trong python lại đưa ra kết quả là 3. Điều này không phải do lỗi của Python, nếu thực hiện phép chia này trong các ngôn ngữ lập trình khác, ví dụ như C/C++ sẽ đều cho kết quả như vậy. Trong phép chia cần một lưu ý như sau: Đó là kết quả của phép chia số nguyên cho một số nguyên thì luôn là số nguyên. Điều này lý giải kết quả phép chia của 15 (số nguyên) cho 6 (số nguyên) sẽ là 3 - cũng là một số nguyên. Phép chia này còn được gọi là chia lấy phần nguyên. Chúng ta sẽ thảo luận trong phần **3.2**.

2.3.1 Tránh chia lấy nguyên trong python

Có 2 cách để thực hiện phép chia 2 số nguyên cho ra một số thực đúng kết quả của phép chia như sau:

1. Sử dụng *future division* trong Python: Từ phiên bản python 3.0 khi sử dụng future division sẽ cho phép kết quả của phép chia là một số có dấu phẩy động để thể hiện số thực. Trong phiên bản python cũ (2.x) chúng ta có thể import thư viện bằng câu lệnh `from future import division`:

   ```
   >>> 21/7 3
   >>> 15/6 2
   >>> from __future__ import division
   >>> 15/6 2.5
   >>> 21/7
   3.0
   ```

   > Nếu muốn import thư viện trong chương trình python, các bạn nên chèn câu lệnh import vào phần đầu của file.

2. Một cách khác đó là chúng ta đưa số chia hoặc số bị chia về kiểu dữ liệu float. Kết quả phép chia sẽ luôn là kiểu dữ liệu float. Chúng ta có thể viết số 15 là **15.0**, **15.** hoặc ép kiểu sang kiểu dữ liệu float bằng cách sử dụng hàm `float(15)`.

   ```
   >>> 15/6 2
   >>> 15./6 2.5
   >>> 15.0/6 2.5
   >>> float (15)/6
   2.5
   >>> 15/6. 2.5
   >>> 15/ float (6) 2.5
   >>> 15./6.
   2.5
   ```

   > **Ép kiểu** là cách thực hiện biến đổi từ một kiểu dữ liệu này sang một kiểu dữ liệu khác. Python cung cấp các wrapper method để ép dữ liệu sang

> các kiểu khác nhau như: `float(), int()...`

2.3.2 Tại sao lại cần chú ý phép chia số nguyên trong python

Phép chia số nguyên có thể dẫn đến một số bug bất ngờ như: Giả sử để viết code tính giá trị trung bình $m = (x + y)/2$ của 2 số nguyên x và y có thể viết như sau:

```
m = (x + y) / 2
```

Khi chúng ta thử lần lượt với các giá trị `x = 0.5, y = 0.5`. Dòng trên sẽ cho ra kết quả đúng là `m = 0.5` (bởi vì `0.5 + 0.5 = 1.0`), do vậy `1.0/2 = 0.5`. Hoặc khi thử với `x = 10, y = 30`, bởi vì `10 + 30 = 40, 40/2 = 20` cho ra kết quả vẫn đúng `m = 20`. Tuy nhiên khi thực hiện phép tính với `x = 0 và y = 1`, đoạn code trên vẫn cho ra kết quả `m = 0` (vì 0 + 1 = 1 và 1/2 cho kết quả bằng 0) trong khi `m = 0.5` mới là kết quả đúng. Như vậy để tính toán một cách an toàn đối với phép toán chia đó là thực hiện việc ép kiểu hoặc đổi kiểu như sau:

- `m = (x + y) / 2.0`
- `m = float(x + y) / 2`
- `m = (x + y) * 0.5`

Vấn đề về việc chia số nguyên đều gặp phải trong các ngôn ngữ lập trình khác như: `C, C++ và Fortran`.

2.4 Các hàm toán học trong python

Python là một ngôn ngữ lập trình được sử dụng trong nhiều lĩnh vực, vì vậy python cung cấp các hàm toán học phổ biến như `sin, cos, exp, log` và rất nhiều hàm khác trong module `math`. Chúng ta có thể sử dụng các hàm trong module này bằng cách thêm module `math` như sau:

```
>>> import math
>>> math.exp(1.0)
2.7182818284590451
```

Sử dụng hàm `dir` giúp chúng ra biết module này có những hàm nào:

```
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs',
'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh']
```

Sau đó dùng hàm help để xem thông tin về hàm/module cũng như cách sử dụng:

```
>>> help(math.exp)
Help on built-in function exp in module math: exp (...)
exp(x)
Return e raised to the power of x.
```

Module toán học trong python cũng cung cấp các hằng số như `π` và `e` như sau:

```
>>> math.pi 3.1415926535897931
>>> math.e 2.7182818284590451 >>> math.cos(math.pi) -1.0
>>> math.log(math.e)
1.0
```

2.5 Biến trong python

Biến trong python có thể được sử dụng để lưu một giá trị hoặc một đối tượng. Trong python, tất cả các thành tố như hàm, module, files... đều là các đối tượng. Một biến được tạo bằng cách gán một giá trị:

```
>>> x = 0.5
>>>
```

Khi biến x được tạo bằng cách gán giá trị 0.5 cho x, ta có thể sử dụng biến x:

```
>>> x*3 0.5
>>> x**2 0.25
>>> y = 111 >>> y+222
333
```

Một biến được ghi đè nếu một giá trị mới được gán vào:

```
>>> y = 0.7
>>> math.sin(y) ** 2 + math.cos(y) ** 2 1.0
```

Toán tử gán ('=') được sử dụng để gán một giá trị cho một biến:

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

Một giá trị có thể được gán cho nhiều biến cùng lúc như sau:

```
>>>x=y=z=0 # khởi tạo biến x, y và z với giá trị 0
>>> x
0
>>> y
0
>>> z
0
```

Các biến phải được tạo (gán giá trị) trước khi chúng được sử dụng, nếu không sẽ xuất hiện lỗi:

```
>>> # thử sử dụng biến chưa được tạo
... n
Traceback (most recent call last): File "<stdin >", line 1, in
<module >
NameError: name 'n' is not defined
```

> Về bản chất thì biến đại diện cho địa chỉ của một ô nhớ (vùng nhớ) trong máy tính. Khi thực hiện khởi tạo hay khai báo biến tức là ta đang gán giá trị của ô nhớ cho giá trị khởi tạo. Nếu không có giá trị khởi tạo thì biến coi như chưa được khai báo.

Ví dụ khi ta thực hiện khai báo:

```
>>> x = 0.5
```

Đầu tiên python sẽ tạo ra một đối tượng có giá trị bằng 0.5, đối tượng này nằm trong bộ nhớ và có một địa chỉ nào đó. Tiếp đến python sẽ gán tên biến x cho đối tượng có giá trị 0.5 vừa tạo ra, nghĩa là x được tham chiếu đến ô nhớ của đối tượng có giá trị 0.5.

2.6 Một số toán tử gán bất thường

Trong các chương trình máy tính chúng ta hay thấy các câu lệnh kiểu như sau:

```
x = x + 1
```

Nếu như biểu thức này trong toán học,

$$x = x + 1$$

ta trừ cả 2 vế cho x sẽ được:

$$0 = 1$$

Chúng ta thấy điều này có vẻ không đúng, có gì đó sai sai ở đây.

Câu trả lời đó là "biểu thức" trên trong lập trình không phải là biểu thức toán học mà là một phép gán. Chúng được thực hiện như sau:

1. Thực hiện tính toán giá trị ở vế phải
2. Gán giá trị vừa tính được ở vế phải cho biến x ở vế trái.

Trong khoa học máy tính người ta sử dụng ký hiệu cho toán tử gán để tránh nhầm lẫn với biểu thức trong toán học:

```
x ← x+1
```

Áp dụng 2 bước trên để tính toán thử câu lệnh: `x = x + 1`

1. Thực hiện tính toán giá trị ở vế phải: Giả sử `x = 4`, trong trường hợp này giá trị của vế phải `x+1` sẽ là `5`.
2. Gán giá trị vừa tính được ở vế phải cho biến x ở vế trái: Thực hiện gán giá trị `5` vừa tính được ở vế phải cho biến x.

Thực nghiệm lại với câu lệnh python:

```
>>> x = 4
>>> x = x + 1
>>> print(x)
5
```

2.6.1 Toán tử +=

Tăng biến x với một giá trị cố định c nào đó được sử dụng rất phổ biến, nên có thể viết gọn hơn dưới dạng:

```
x += c
```

Thay vì:

```
x = x + c
```

Thử với python như sau:

```
>>> x = 4
>>> x += 1
>>> print(x)
5
```

Tương tự như toán tử tăng, chúng ta cũng có thể sử dụng với phép nhân như `+=` hoặc toán tử trừ `-=` và chia cho một hằng số `/=`

> Chú ý thứ tự của toán tử `+` và `=` :
>
> - `x += 1` : Tăng x lên 1
> - `x =+1` : Gán x cho giá trị dương 1.

Chương 3: Các kiểu dữ liệu và cấu trúc dữ liệu

3.1 Xác định các kiểu dữ liệu

Python cung cấp rất nhiều kiểu dữ liệu khác nhau. Để xác định kiểu dữ liệu của một biến, ta sử dụng hàm `type()` như sau:

```
>>> a = 45
>>> type(a)
<type 'int'>
>>> b = 'Đây là một chuỗi'
>>> type(b)
<type 'str'>
>>> c = 2 + 1j
>>> type(c) <type 'complex'>
>>> d = [1, 3, 56]
>>> type(d)
<type 'list'>
```

3.2 Kiểu số

Python cung cấp kiểu dữ liệu số bao gồm các số nguyên, số thực và số phức. Ngoài ra python cũng cung cấp kiểu dữ liệu `long integers` cho phép biểu diễn số nguyên với phạm vi biểu diễn lớn.

3.2.1 Số nguyên

Kiểu dữ liệu số nguyên `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. Để chuyển đổi một chuỗi số sang số nguyên ta có thể sử dụng hàm `int()`.

```
>>> a = '34' # chuỗi số gồm 2 ký tự 3 và 4
 >>> x = int(a) # x là một số nguyên
```

Hàm `int()` cũng chuyển đổi một số thực sang số nguyên:

```
>>> int (7.0)
7
>>> int (7.9)
7
```

> Chú ý hàm `int` sẽ bỏ đi phần thập phân đằng sau dấu phẩy và chuyển phần nguyên đằng trước dấu phẩy thành số nguyên. Để làm tròn một số thực thành số nguyên ta có thể sử dụng hàm `round()`:

```
>> round (7.9)
8.0
>> int(round(7.9))
8
```

3.2.2 Kiểu long

Trong một số ngôn ngữ lập trình, việc biểu diễn số nguyên chỉ cho phép biểu diễn số với một giá trị trong khoảng có thể biểu diễn được. Trong python, module `sys` cho phép chúng ta xác định giá trị nguyên lớn nhất mà python có thể biểu diễn được:

```
>> import sys
>> sys.maxint
2147483647
```

Như vậy python chỉ có thể biểu diễn được số nguyên int có giá trị lớn nhất là 2147483647. Nếu vượt quá giá trị này, python sẽ chuyển kiểu dữ liệu sang kiểu dữ liệu long.

```
>> type(sys.maxint)
<type 'int'>
>> sys.maxint+1
2147483648L
>> type(sys.maxint+1)
<type 'long'>
```

Kiểu dữ liệu long hoạt động như một số nguyên bình thường nhưng bất kỳ phép tính số học nào liên quan đến kiểu long được thực hiện ở cấp độ phần mềm mà không phải trong CPU. Điều này giúp tránh hiện tượng tràn số nhưng chúng ta nên chú ý việc xử lý tính toán đối với kiểu dữ liệu long thì chậm hơn nhiều so với tính toán dựa trên số nguyên. Số nguyên kiểu `long` càng dài thì RAM sử dụng càng lớn và càng tốn thời gian xử lý của CPU.

3.2.3 Kiểu dữ liệu số thực

Trong Python float dùng để biểu diễn số thực dấu phẩy động. Để viết số thực trong Python ta cần đặt 1 dấu chấm thập phân.

```
>>> import sys
>>> 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
>>>
```

Ta có thể chuyển đổi một chuỗi chứa dấu phẩy động thành kiểu dữ liệu float như sau:

```
>>> a = '35.342'
>>> b = float(a)
>>> print(b)
35.342
>>> print(type(b))
<type 'float'>
```

3.2.4 Số phức

Python là một trong số ít các ngôn ngữ (giống như Fortran và Matlab) đều cung cấp kiểu dữ liệu số phức. Tuy nhiên số phức tương đối ít được sử dụng. Sau đây là một vài ví dụ về việc 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:

```
>>> x = 1 + 3j
>>> x
(1+3j)
>>> abs(x) #Tính module của số phức
3.1622776601683795
>>> x.imag
3.0
>>> x.real
1.0
>>> x * x
(-8+6j)
>>> x * x.conjugate()
(10+0j)
>>> 3 * x
(3+9j)
```

> 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 của kiểu số thực giúp bạn trích giá trị của phần nguyên và phần ảo.

> Khi cần thực hiện tính toán với số phức ta sẽ dùng module `cmath` (Complex MATHematics).

```
>> import cmath
>> cmath.sqrt(x)
(1.442615274452683+1.0397782600555705j)
```

3.2.5 Hàm áp dụng cho tất cả các kiểu dữ liệu số

hàm `abs()` trả về giá trị tuyệt đối của một số (hay còn gọi là modulus):

```
>>> a = -45.463
>>> print(abs(a))
45.463
```

> Hàm abs() được sử dụng cho cả kiểu số phức.

3.3 Kiểu dữ liệu tuần tự

Kiểu String (chuỗi) (3.3.1, kiểu List (danh sách) (3.3.2) và kiểu dữ liệu Tuple (3.3.3) được gọi là kiểu dữ liệu tuần tự. Các phần tử trong kiểu dữ liệu này được đánh chỉ mục (3.3.4) và cắt lát (3.3.5) với cách thực hiện giống nhau.

Kiểu dữ liệu Tuple và String là các kiểu dữ liệu "immutable" (không thay đổi được - nghĩa là ta không thể thay đổi các phần tử của chuỗi hay tuple). Còn List là kiểu dữ liệu có thể thay đổi được vì ta có thể dễ dàng thay đổi một phần tử bất kỳ trong List. Kiểu dữ liệu tuần tự có chung các toán tử như sau:

a[i]	Trả về phần tử thứ i của a
a[i:j]	Trả về danh sách các phần tử thứ i đến phần tử thứ j – 1
len(a)	Trả về số lượng các phần tử trong a
min(a)	Trả về giá trị nhỏ nhất trong a
max(a)	Trả về giá trị lớn nhất trong a
x in a	Trả về True nếu trong a có giá trị x
a+b	Nối a và b
n * a	Tạo ra n bản copy của a

3.3.1 Kiểu dữ liệu String

Python cho phép viết giá trị chuỗi theo nhiều cách khác nhau. Chuỗi ký tự trong python được bao quanh bởi dấu ngoặc kép đơn hoặc dấu ngoặc kép. Python coi các lệnh trích dẫn đơn và kép là như nhau. Ví dụ: 'Hello' tương đương với "Hello".

```
# Tất cả những ví dụ dưới đây là tương đương với nhau.

# Dùng nháy đơn
my_string = 'Xin chào Python!'
print(my_string)

# Dùng nháy kép
my_string = "Xin chào Python!"
print(my_string)

# Dùng ba nháy đơn trên nhiều dòng
my_string = '''Xin chào
            Python'''
print(my_string)

# Dùng ba nháy kép trên nhiều dòng
my_string = """Xin chào
            Python"""
print(my_string)
```

```
Xin chào Python!
Xin chào Python!
Xin chào
            Python
Xin chào
            Python
```

String được xem như là một list các ký tự nên ta có thể truy cập đến các ký tự thông qua chỉ mục của nó. Chỉ mục bắt đầu từ 0, và nếu ta cố gắng truy cập một ký tự nằm ngoài phạm vi chỉ mục thì sẽ xuất hiện lỗi IndexError. Chỉ mục phải là một số nguyên, không thể sử dụng float hoặc các loại khác vì điều này sẽ dẫn đến TypeError. Chỉ số -1 đề cập đến phần tử cuối cùng, -2 cho phần tử cuối cùng thứ hai, .. (đây là cách tính ngược của chuỗi).

Chúng ta có thể truy cập một loạt các phần tử trong một chuỗi bằng cách sử dụng toán tử slicing (dấu hai chấm).

```
str = 'ABCDEFGHIKLMNOPQ'
print('str = ', str)

# Ký tự đầu tiên
print('str[0] = ', str[0])

# Ký tự cuối cùng
print('str[-1] = ', str[-1])

#Lấy các kí tự thứ 2 đến thứ 6
print('str[1:5] = ', str[1:5])

# Lấy các kí tự từ thứ 6 đến ky tự thứ 3 tính từ cuối chuỗi
print('str[5:-2] = ', str[5:-2])
```

```
str =  ABCDEFGHIKLMNOPQ
str[0] =  A
str[-1] =  Q
str[1:5] =  BCDE
str[5:-2] =  FGHIKLMNO
```

Phép toán nối hai hoặc nhiều chuỗi thành một chuỗi thì ta gọi là phép nối chuỗi. Để nối chuỗi thì ta sử dụng toán tử +. Nếu muốn lặp lại chuỗi nhiều lần thì dùng toán tử *. Hãy xem ví dụ dưới đây:

```
str1 = 'Hello'
str2 ='World!'

# Sử dụng +
print('str1 + str2 = ', str1 + str2)

# Sử dụng *
print('str1 * 3 =', str1 * 3)
```

```
str1 + str2 =  HelloWorld!
str1 * 3 = HelloHelloHello
```

Để kiểm tra một chuỗi con có xuất hiện trong chuỗi cha hay không thì ta dùng toán tử in và not in.

```
# Trả về True
print('q' in 'nguyendinhquy')

# Trả về False
print('q' not in 'nguyendinhquy')
```

Ngoài ra String cung cấp một số hàm hữu ích để xử lý chuỗi, ví dụ như hàm `upper()` trả về chuỗi ký tự sau khi được chuyển sang chữ viết hoa:

```
>>> a = "Đây là đoạn văn bản demo"
>>> a.upper()
'ĐÂY LÀ ĐOẠN VĂN BẢN DEMO'
```

Để biết những phương thức xử lý chuỗi trong python ta có thể xem trong tài liệu của python bằng hàm `dir` và hàm `help`. Hai phương thức này dùng để xem các thông tin về các phương thức xử lý chuỗi, ví dụ `dir()` trả về danh sách các phương thức, `help` dùng để xem cách sử dụng.

3.3.2 Kiểu dữ liệu List

Trong Python, List là một loại dữ liệu đặc biệt, nó gồm nhiều phần tử và mỗi phần tử là một dữ liệu riêng biệt. Ví dụ danh sách số nguyên:

```
>>> a = [34, 12, 54]
```

Hoặc danh sách chuỗi:

```
>>> a = ['chó', 'mèo', 'chuột']
```

Một danh sách trống được biểu diễn bằng dấu []

```
>>> a = []
```

Kiểu dữ liệu của a là List

```
>>> type(a)
<type 'list'>
>>> type ([])
<type 'list'>
```

Giống như *String*, để lấy ra số phần tử của List, ta có thể dùng hàm `len()`:

```
>>> a = ['chó', 'mèo', 'chuột']
>>> len(a)
3
```

Khác với các ngôn ngữ lập trình biên dịch như C/C++, Java... chỉ cho phép định nghĩa một mảng gồm các phần tử có cùng kiểu, Python cho phép một List có thể lưu các dữ liệu khác kiểu như:

```
>>> a = [123, 'vịt', -42, 17, 0, 'voi']
```

Trong python, List là một đối tượng, vì thế trong List có thể là một List khác (vì List là danh sách các đối tượng)

```
a = [1, 4, 56, [5, 3, 1], 300, 400]
```

Bạn có thể nối 2 List bằng toán tử +:

```
>>> [3, 4, 5] + [34, 35, 100]
[3, 4, 5, 34, 35, 100]
```

Hoặc thêm một phần tử vào cuối List:

```
>>> a = [34, 56, 23]
>>> a.append(42)
>>> print(a)
[34, 56, 23, 42]
```

Bạn có thể xoá một phần tử trong List bằng cách dùng phương thức `remove()` và truyền vào phần tử cần xoá. Ví dụ:

```
>>> a = [34, 56, 23, 42]
>>> a.remove(56)
>>> print a [34, 23, 42]
```

Hàm range() Một kiểu List đặc biệt được dùng thường xuyên với vòng lặp for gồm các phần tử nguyên chạy từ 0 đến n. Hàm `range(n)` cho phép tạo ra một mảng các số nguyên chạy từ 0 đến n-1 (không gồm giá trị n) như vậy . Sau đây là một vài ví dụ:

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

Hàm range được dùng phổ biến với vòng lặp. Ví dụ, để in ra các số bình phương từ 0 đến 10 ($0^2, 1^2, 2^2, 3^2, \ldots, 10^2$ ta có thể làm như sau:

```
>>> for i in range (11):
... print i ** 2 ...
0
1
4
9
16
25
36
49
64
81
100
```

Hàm range cho phép truyền thêm tham số cho phép tạo danh sách các số nguyên từ giá trị bắt đầu (start), hoặc có thêm giá trị bước nhảy (step) cho biết khoảng cách giữa các số được tạo ra. Chúng thường được viết là `range([start],stop,[step])` với các tham số trong dấu ngoặc vuông là tham số tuỳ chọn. Sau đây là một số ví dụ:

```
>>> range(3, 10) # start=3 – Tạo ra các số nguyên bắt đầu từ 3
và nhỏ hơn 10
[3, 4, 5, 6, 7, 8, 9]
>>> range(3, 10, 2) # start=3, step=2 – Tạo ra các số nguyên
bắt đầu từ 3 và nhỏ hơn 10, khoảng cách mỗi số là 2
[3, 5, 7, 9]
>>> range(10, 0, -1) # start=10,step=-1 – Tạo ra các số nguyên
bắt đầu từ 10 về 0, khoảng cách mỗi số là -1
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

3.3.3 Kiểu dữ liệu Tuple

Tuple là một kiểu dữ liệu gồm các đối tượng tuần tự. Tuple giống với List, chỉ khác một điểm đó là Tuple thì không thể thay đổi được các giá trị của các phần tử (immutable).

Tuple có thể gồm các phần tử có bất kỳ kiểu dữ liệu nào. Ví dụ:

```
>>> a = (12, 13, 'dog')
>>> a
(12, 13, 'dog')
>>> a[0]
12
```

Khi định nghĩa một tuple không nhất thiết phải sử dụng dấu ngoặc đơn để nhóm các phần tử, chỉ cần liệt kê các phần tử và phân cách nhau bởi dấu phẩy:

```
>>> a = 100, 200, 'duck'
>>> a
(100, 200, 'duck')
```

> Việc sử dụng dấu ngoặc đơn để nhóm các phần tử của tuple giúp dễ dàng trong việc đọc code. Đây là quy ước các bạn nên tuân theo.

Ngoài ra Tuple cũng được sử dụng để thực hiện việc gán cùng lúc nhiều giá trị cho nhiều biến:

```
>>> x, y = 10, 20
>>> x
10
>>> y
20
```

Hoặc được sử dụng để hoán đổi giá trị của 2 biến trên cùng 1 dòng:

```
>>> x = 1
>>> y = 2
>>> x, y = y, x
>>> print(x)
2
>>> print(y)
1
```

Tuple rỗng được khai báo như sau:

```
>>> t = ()
>>> len(t) 0
>>> type(t)
<type 'tuple'>
```

Để khai báo tuple chỉ gồm 1 phần tử ta làm như sau:

```
>>> t = (42,)
>>> type(t) <type 'tuple'>
```

```
>>> len(t)
1
```

Dấu phẩy được thêm vào để phân biệt khai báo biến là tuple thay vì là biến kiểu int. Nếu không có dấu phẩy thì biến được hiểu là biến int có giá trị là 42 thay vì là một tuple có một phần tử.

```
>>> t = (42)
>>> type(t)
<type 'int'>
```

Ví dụ sau giúp chúng ta thấy tuple không thể thay đổi được giá trị các phần tử:

```
>>> a = (12, 13, 'dog') >>> a[0]
12
>>> a[0] = 1
Traceback (most recent call last):
File "<stdin >", line 1, in ?
TypeError: object doesn't support item assignment
```

> Tính bất biến là sự khác biệt giữa một tuple và một list. Chúng ta nên sử dụng tuple khi chúng ta không muốn thay đổi nội dung của các phần tử. Lưu ý đối với các hàm python trả về nhiều giá trị, hãy trả về các giá trị này bằng một tuple.

3.3.4 Chỉ mục trong kiểu dữ liệu tuần tự

Ta có thể truy cập vào một phần tử trong một list bằng cách truy cập thông qua chỉ mục (hay vị trí) của phần tử trong list thông qua cặp dấu ngoặc vuông ([và])

```
>>> a = ['chó', 'mèo', 'chuột']
>>> a[0]
'chó'
>>> a[1] 'mèo'
>>> a[2] 'chuột'
```

> Chú ý trong python (cũng giống C/C++ nhưng khác với Fortran, Pascal, Matlab) vị trí phần tử đầu tiên được bắt đầu từ 0. Chỉ số phần tử cuối cùng của list là "-1". Tương tự, chỉ số "-2" là chỉ số phần tử thứ 2 tính từ cuối.

```
>>> a = ['chó', 'mèo', 'chuột']
>>> a[-1]
'chuột'
>>> a[-2]
'mèo'
```

> Nếu muốn, bạn có thể coi chỉ số a `[-1]` là một ký hiệu viết tắt cho `[len(a) - 1]`.
>
> Nhớ rằng các chuỗi (string), giống như List đều là một kiểu dữ liệu tuần tự nên có thể được đánh chỉ số theo cách tương tự:
>
> ```
> >> a = "Hello World!"
> >> a[0] 'H'
> >> a[1]
> 'e'
> >> a[10] 'd'
> >> a[-1] '!'
> >> a[-2] 'd'
> ```

3.3.5 Cắt lát

Cắt lát dữ liệu được dùng để lấy ra nhiều hơn một phần tử của một danh sách, chuỗi... Ví dụ:

```
>>> a = "Hello World!"
>>> a[0:3]
'Hel'
```

Bằng việc lấy a[0:3], ta lấy ra 3 phần tử đầu tiên bắt đầu từ phần tử 0.

Tương tự như vậy:

```
>>> a[1:4] 'ell'
>>> a[0:2]
'He'
>>> a[0:6] 'Hello '
```

Ta có thể dùng chỉ mục âm cho phần tử tính từ phần tử cuối:

```
>>> a[0:-1]
'Hello World'
```

Ta cũng có thể loại bỏ chỉ mục đầu hoặc cuối, loại chỉ số này sẽ trả về tất cả các phần tử tính từ đầu hoặc từ cuối của dãy. Sau đây là một số ví dụ:

```
>>> a = "Hello World!"
>>> a[:5]
'Hello'
>>> a[5:] ' World!'
>>> a[-2:] 'd!'
>>> a[:] 'Hello World!'
```

> Chú ý rằng `a[:]` sẽ tạo ra một giá trị được copy từ biến a chứ không tham chiếu đến biến `a`.

3.3.6 Kiểu dữ liệu Dictionary

Kiểu dữ liệu Dictionary trong Python là một tập hợp các cặp key-value không có thứ tự, có thể thay đổi và lập chỉ mục (truy cập phần tử theo chỉ mục).

Dictionary được khởi tạo với các dấu ngoặc nhọn {} và chúng có các khóa và giá trị (key-value). Mỗi cặp key-value được xem như là một phần tử. Key cho phần tử đó phải là duy nhất, trong khi đó value có thể là bất kỳ kiểu giá trị nào. Phần tử phải là một kiểu dữ liệu không thay đổi (immutable) như chuỗi, số hoặc tuple.

Key và value được phân biệt riêng rẽ bởi một dấu hai chấm (:). Các phần tử phân biệt nhau bởi một dấu phẩy (,). Các item khác nhau được bao quanh bên trong một cặp dấu ngoặc móc đơn tạo nên một Dictionary trong Python

Ta có thể truy cập các phần tử của Dictionary bằng cách sử dụng khóa (key) của nó, bên trong dấu ngoặc vuông, ví dụ:

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
}
print(dictCar["model"])
```

```
Honda Civic
```

Ngoài ra ta cũng có thể sử dụng hàm get() để truy cập vào phần tử của Dictionary trong Python như trong ví dụ sau:

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
}
print(dictCar.get("model"))
```

```
Honda Civic
```

Ta có thể thay đổi giá trị của một phần tử cụ thể bằng khóa của nó:

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
}
dictCar["year"] = 2020
print(dictCar)
```

```
{'brand': 'Honda', 'model': 'Honda Civic', 'year': 2020}
```

Ta có thể duyệt qua một Dictionary bằng cách sử dụng vòng lặp for .

Khi duyệt một Dictionary bằng vòng lặp for, giá trị trả về là các khóa, khi đó bạn có thể dùng hàm get() để lấy giá trị của khóa.

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
}
for x in dictCar:
    print(x, ": ", dictCar.get(x))
```

```
brand :  Honda
model :  Honda Civic
year :  1972
```

Ta cũng có thể sử dụng hàm values() để trả về các giá trị của Dictionary:

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
}
for x in dictCar.values():
    print(x)
```

```
Honda
Honda Civic
1972
```

Để xác định xem một khóa `(key)` được chỉ định có tồn tại trong từ điển hay không, hãy sử dụng từ khóa `in`:

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
}
if "model" in dictCar:
    print("Khoa \"model\" co ton tai.")
else:
    print("Khoa \"model\" khong ton tai.")
```

```
Khoa "model" co ton tai.
```

Để xác định có bao nhiêu phần tử (cặp khóa-giá trị) trong Dictionary, hãy sử dụng hàm len().

```
dictCar = {
```

```
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
}
print(len(dictCar))
```

```
3
```

Thêm một phần tử vào Dictionary được thực hiện bằng cách sử dụng khóa mới và gán giá trị cho nó:

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
}
dictCar["color"] = "yellow"
print(dictCar)
```

```
{'brand': 'Honda', 'model': 'Honda Civic', 'year': 1972, 'color': 'yellow'}
```

Có nhiều phương pháp để loại bỏ các phần tử của một Dictionary.

Hàm pop() xóa phần tử với khoá (key) được chỉ định:

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
}
dictCar.pop("model")
print(dictCar)
```

```
{'brand': 'Honda', 'year': 1972}
```

Hàm popitem() xóa phần tử cuối cùng (trong các phiên bản trước 3.7, một mục ngẫu nhiên được xóa).

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
    "year": 1972
}
dictCar.popitem()
print(dictCar)
```

```
{'brand': 'Honda', 'model': 'Honda Civic'}
```

Lệnh del sẽ xóa phần tử với key được chỉ định:

```
dictCar = {
    "brand": "Honda",
    "model": "Honda Civic",
```

```
    "year": 1972
}
del dictCar["model"]
print(dictCar)
```

```
{'brand': 'Honda', 'year': 1972}
```

3.4 Truyền tham số cho hàm

Như trong C/C++ có 2 cách truyền tham số cho một hàm: Truyền tham số dạng tham trị và truyền tham số dạng tham chiếu. Trong python cách truyền tham số tương tự như truyền tham chiếu trong các ngôn ngữ lập trình như Java hay C/C++.

3.4.1 Tham số bắt buộc

Tham số bắt buộc là loại tham số mặc định của hàm trong Python. Hãy xem ví dụ sau:

```
def equation(a, b, c):
    '''Giải phương trình bậc 2
    Tham số: a, b, c là các số thực
    Trả về: một 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('Nghiệm của phương trình:')
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}')
```

```
Nghiệm của phương trình:
x1 = -1.0
x2 = -1.0
x1 = -1.0
x2 = -1.0
```

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 tài liệu mô tả cho hàm phía trên 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.

3.4.2 Tham số dạng từ khoá

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). 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.

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.

3.4.3 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.

Chúng ta xem ví dụ sau:

```
from math import sqrt

def equation(a, b=0, c=0):
  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

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
```

```
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
```

3.4.4 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 tham số 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 để ý 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
```

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
```

3.4.5 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 này 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)
```

```
a = 1
b = 2
c = 3
```

Qua ví dụ nhỏ này ta thấy hiệu quả của cú pháp `**kwargs`:

- Ta 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))
```

```
6
```

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.

Chương 4: Vào ra trong Python

Chương này sẽ tập trung vào các tác vụ nhập xuất dữ liệu trong python (python3). Bao gồm cách sử dụng hàm `print()` để in ra màn hình, định dạng dữ liệu và chuỗi khi in, nhập dữ liệu từ bàn phím. Chương này cũng giới thiệu cách đọc/ghi dữ liệu từ tệp tin.

4.1 In ra màn hình

Chúng ta sử dụng hàm print() để xuất dữ liệu ra thiết bị chuẩn (màn hình). Chúng ta cũng có thể dùng hàm này để xuất dữ liệu ra một tệp (4.2)

Ví dụ 1:

```
print('Lập trình Python dành cho kỹ thuật')
```

Hoặc ví dụ 2:

```
a = 200
print('Giá trị:', a)
```

Kết quả ví dụ 2:

```
Giá trị: 200
```

Trong ví dụ print() thứ hai, chúng ta có thể nhận thấy rằng khoảng trắng đã được thêm vào giữa chuỗi ký tự và giá trị của biến a. Đây là định dạng in mặc định trong Python, tuy nhiên, chúng ta cũng có thể thay đổi bằng cách thay đổi giá trị các tham số:

```
print(objects_to_print, sep=' ', end='\n', file=sys.stdout,
flush=False)
```

Trong đó `objects_to_print` là các đội đối tượng cần in ra màn hình (các chuỗi, số...), `sep` là ký tự phân cách giữa các đối tượng và `end` là ký tự kết thúc, giá trị mặc định là ký tự xuống dòng (\n), file là đối tượng muốn in ra (sys.stdout nghĩa là in ra màn hình console) Ví dụ:

```
print(230, 250)
print(230, 250, sep='*')
```

```
print(230, 250, sep='#', end='&')
```

```
230 250
230*250
230#250&
```

Định dạng chuỗi muốn in ra Đôi khi chúng ta muốn định dạng chuỗi để khi in ra dễ nhín hơn. Để thực hiện chúng ta sử dụng phương thức `str.format()` : Ví dụ để in ra giá trị biến `a` và `b` trong một chuỗi:

```
a = 100;
b = 200;
print('Giá trị của a là: {} và b là: {}'.format(a,b))
```

```
Giá trị của a là: 100 và b là: 200
```

Ngoài ra chúng ta có thể sử dụng từ khóa của các tham số để định dạng chuỗi.

```
print('Lập trình {lang} {prop}'.format(lang = 'Python', prop = 'trong kỹ t
```

```
Lập trình Python trong kỹ thuật
```

Chúng ta cũng có thể định dạng chuỗi như kiểu sprintf() được sử dụng trong ngôn ngữ lập trình C. Chúng ta sử dụng toán tử dấu phần trăm (%) để thực hiện điều này:

```
a = 100.50
print('Giá trị là %.2f' %a)
```

```
Giá trị là 100.50
```

4.2 Nhập dữ liệu từ bàn phím

Trong các đoạn code ví dụ từ đầu cuốn sách này, giá trị của các biến đều đã được gán cố định một giá trị bằng toán tử gán.

Để cho phép sự linh hoạt, chúng ta có thể muốn lấy giá trị đầu vào từ người dùng. Trong Python, chúng ta có thể sử dụng hàm `input()` để nhập dữ liệu từ bàn phím.

Thông thường hàm `input()` có thêm tham số để chỉ dẫn người dùng cần nhập thông tin gì.

Cấu trúc hàm `input()` có dạng như sau:

```
input([prompt])
```

> Trong đó prompt là chuỗi chỉ dẫn mà chúng ta muốn hiển thị trên màn hình cho người dùng biết. Tham số này là tùy chọn, không nhất thiết phải truyền giá trị.

Ví dụ:

```
a = input('Nhập một số nguyên: ')
print(a)
```

Kết quả sẽ là:

```
>>> Nhập một số nguyên:
200
```

> Ở đây, chúng ta có thể thấy rằng giá trị 200 đã nhập ở dạng chuỗi ký tự, không phải kiểu dữ liệu số. Để chuyển thành kiểu dữ liệu số, chúng ta có thể sử dụng hàm `int()` hoặc `float()`. Ví dụ:

```
a = input('Nhập một số nguyên: ')
print(type(a))
print(int(a))
```

```
Nhập một số nguyên: 200
<class 'str'>
200
```

4.3 Nhập xuất dữ liệu từ File

File hay còn gọi là tệp, tập tin. File là tập hợp của các thông tin được đặt tên và lưu trữ trên bộ nhớ máy tính như đĩa cứng, đĩa mềm, CD, DVD,... Trong python, file được dùng để lưu trữ dữ liệu. Các thao tác với file bao gồm nhập và xuất dữ liệu.

Khi muốn đọc hoặc ghi file, chúng ta cần phải mở file trước. Khi hoàn thành, file cần phải được đóng lại để các tài nguyên được gắn với file được giải phóng.

Do đó, trong Python, một thao tác với file diễn ra theo thứ tự sau.

- Mở file
- Đọc hoặc ghi file
- Đóng file

4.3.1 Mở File trong Python

Trong Python, có một hàm được xây dựng sẵn phục vụ cho việc mở file: `open()`. Hàm này trả về đối tượng file hay còn gọi là "handle" vì bạn có thể thực hiện các hoạt động đọc, ghi, sửa đổi trên file đó.

```
>>> f = open("test.txt") # mở file cùng thư mục với file hiện tại
>>> f = open("C:/Python33/README.txt") # mở file ở thư mục khác, đường dẫn đầy đủ
```

Bạn có thể xác định cách thức mà tập tin được mở ra để làm gì như read, write, append,... Đây là thông số tùy chọn có thể có hoặc không. Ngoài ra bạn cũng có thể định rõ file mở ra dạng văn bản hay dạng nhị phân.

Chế độ truy cập file mặc định là read (r). Khi dùng mode này chúng ta sẽ nhận được giá trị chuỗi trả về dạng văn bản.

Mặt khác nếu giá trị trả về ở dạng byte thì tệp được mở ra là hình ảnh hoặc exe.

Dưới đây là danh sách các chế độ mode khác nhau khi mở một file:

MODE	MÔ TẢ
'r'	Chế độ chỉ được phép đọc.
'r+'	Chế độ được phép đọc và ghi
'rb'	Mở file chế độ đọc cho định dạng nhị phân. Con trỏ tại phần bắt đầu của file
'rb+' 'r+b'	Mở file để đọc và ghi trong định dạng nhị phân. Con trỏ tại phần bắt đầu của file
'w'	Mở file để ghi. Nếu file không tồn tại thì sẽ tạo mới file và ghi nội dung, nếu file đã tồn tại thì sẽ bị cắt bớt (truncate) và ghi đè lên nội dung cũ
'w+'	Mở file để đọc và ghi. Nếu file không tồn tại thì sẽ tạo mới file và ghi nội dung, nếu file đã tồn tại thì sẽ bị cắt bớt (truncate) và ghi đè lên nội dung cũ
'wb'	Mở file để ghi cho dạng nhị phân. Nếu file không tồn tại thì sẽ tạo mới file và ghi nội dung, nếu file đã tồn tại thì sẽ bị cắt bớt (truncate) và ghi đè lên nội dung cũ
'wb+' 'w+b'	Mở file để đọc và ghi cho dạng nhị phân. Nếu file không tồn tại thì sẽ tạo mới file và ghi nội dung, nếu file đã tồn tại thì sẽ bị cắt bớt (truncate) và ghi đè lên nội dung cũ
'a'	Mở file chế độ ghi tiếp. Nếu file đã tồn tại rồi thì nó sẽ ghi tiếp nội dung vào cuối file, nếu file không tồn tại thì tạo một file mới và ghi nội dung vào đó.
'a+'	Mở file chế độ đọc và ghi tiếp. Nếu file đã tồn tại rồi thì nó sẽ ghi tiếp nội dung vào cuối file, nếu file không tồn tại thì tạo một file mới và ghi nội dung vào đó.
'ab'	Mở file chế độ ghi tiếp ở dạng nhị phân. Nếu file đã tồn tại rồi thì nó sẽ ghi tiếp nội dung vào cuối file, nếu file không tồn tại thì tạo một file mới và ghi nội dung vào đó.
'ab+' 'a+b	Mở file chế độ đọc và ghi tiếp ở dạng nhị phân. Nếu file đã tồn tại rồi thì nó sẽ ghi tiếp nội dung vào cuối file, nếu file không tồn tại thì tạo một file mới và ghi nội dung vào đó.
'x'	Mở file chế độ ghi. Tạo file độc quyền mới (exclusive creation) và ghi nội dung, nếu file đã tồn tại thì chương trình sẽ báo lỗi
'x+'	Mở file chế độ đọc và ghi. Tạo file độc quyền mới (exclusive creation) và ghi nội dung, nếu file đã tồn tại thì chương trình sẽ báo lỗi
'xb'	Mở file chế độ ghi dạng nhị phân. Tạo file độc quyền mới và ghi nội dung, nếu file đã tồn tại thì chương trình sẽ báo lỗi
'xb+' 'x+b'	Mở file chế độ đọc và ghi dạng nhị phân. Tạo file độc quyền mới và ghi nội dung, nếu file đã tồn tại thì chương trình sẽ báo lỗi
'b'	Mở file ở chế độ nhị phân
't'	Mở file ở chế độ văn bản (mặc định)

```
f = open("test.txt") # mở file mode 'r' hoặc 'rt' để đọc
f = open("test.txt",'w') # mở file mode 'w' để ghi
f = open("img.bmp",'r+b') # mở file mode 'r+b' để đọc và ghi dạng nhị phân
```

Khi làm việc với các tệp ở chế độ văn bản, bạn nên chỉ định loại bảng mã được dùng để mã hoá văn bản.

```
f = open("test.txt",mode = 'r',encoding = 'utf-8')
```

4.3.2 Đóng file

Sau khi thực hiện xong các thao tác với file thì bạn cần đóng nó lại.

Đóng file để đảm bảo quy chế đóng mở và giải phóng bộ nhớ cho chương trình nên điều này là cần thiết.

Việc đóng file được xây dựng trong Python bằng hàm close().

Python cũng tự động đóng một file khi đối tượng tham chiếu của file đã được tái gán cho một file khác. Tuy nhiên, sử dụng phương thức close() để đóng một file vẫn tốt hơn.

```
f = open("test.txt",encoding = 'utf-8')
# thực hiện các thao tác với tệp
f.close()
```

4.3.3 Ghi File trong Python

Để ghi một file ta cần mở file bằng cú pháp để ghi, sử dụng mode write 'w', append 'a' hoặc mode tạo độc quyền 'x'

Bạn cần cẩn thận với chế độ 'w', vì nó ghi đè lên nội dung nếu file đã tồn tại, các dữ liệu trước đó sẽ bị xóa.

Nếu bạn ghi vào file dạng nhị phân các chuỗi văn bản hoặc chuỗi dạng byte thì kết quả trả về sẽ là số kí tự được ghi vào file.

```
with open("test.txt",'w',encoding = 'utf-8') as f:
  f.write("Lập trình kỹ thuật\n")
  f.write("Giáo trình python\n")
  f.write("Ví dụ về đọc ghi file\n")
```

Với ví dụ trên, chương trình sẽ tạo một file có tên là test.txt nếu tệp chưa tồn tại, nếu tồn tại rồi sẽ bị ghi đè lên. Sử dụng các kí tự '\n' để phân biệt các dòng với nhau.

4.3.3 Đọc File trong Python

Tương tự ghi file, để đọc một file ta cần mở file bằng cú pháp để đọc, sử dụng mode read 'r'.

Dùng `read(size)` Sử dụng phương thức read(size) để lấy về dữ liệu có kích thước bằng size. Nếu để trống tham số này thì nó sẽ đọc hết file hoặc nếu file quá lớn thì nó sẽ đọc đến khi giới hạn của bộ nhớ cho phép.

```
f = open("test.txt",'r',encoding = 'utf-8')
a = f.read(12) # đọc 12 kí tự đầu tiên
print('Nội dung 11 kí tự đầu là:\n', (a))
b = f.read(35) # đọc 35 kí tự tiếp theo
print('Nội dung 35 kí tự tiếp theo là:\n', (b))
c = f.read() # đọc phần còn lại
print('Nội dung phần còn lại là:\n', (c))
```

```
Nội dung 11 kí tự đầu là:
 Lập trình kỹ
Nội dung 35 kí tự tiếp theo là:
  thuật
Giáo trình python
Ví dụ về đ
Nội dung phần còn lại là:
 ọc ghi file
```

Dùng `readline()`

Phương thức này cho phép đọc từng dòng trong file:

```
f = open("test.txt",'r',encoding = 'utf-8')
a = f.readline()
print ('Nội dung dòng đầu: ', (a))
b = f.readline()
print ('Nội dung dòng 2: ', (b))
c = f.readline()
print ('Nội dung dòng 3: ', (c))
```

```
Nội dung dòng đầu:  Lập trình kỹ thuật

Nội dung dòng 2:  Giáo trình python

Nội dung dòng 3:  Ví dụ về đọc ghi file
```

Dùng `readlines()`

Phương thức readlines() trả về toàn bộ các dòng còn lại trong file và trả về giá trị rỗng khi kết thúc file.

```
f = open("test.txt",'r',encoding = 'utf-8')
a = f.readline()
print ('Nội dung dòng đầu: ', (a))
b = f.readlines()
```

```
print ('Nội dung các dòng còn lại: \n', (b))
c = f.readlines()
print ('Nội dung các dòng còn lại: \n', (c))
```

```
Nội dung dòng đầu:  Lập trình kỹ thuật

Nội dung các dòng còn lại:
 ['Giáo trình python\n', 'Ví dụ về đọc ghi file\n']
Nội dung các dòng còn lại:
 []
```

4.3.4 Một số phương thức làm việc với File trong Python

Có rất nhiều phương thức khác nhau để làm việc với file được tích hợp sẵn trong Python, trong đó có một vài phương thức đã được tìm hiểu ở trên.

Bảng dưới đây là danh sách đầy đủ các phương thức dưới dạng text, các bạn tham khảo thêm.

PHƯƠNG THỨC	MÔ TẢ
close()	Đóng một file đang mở. Nó không thực thi được nếu tập tin đã bị đóng.
fileno()	Trả về một số nguyên mô tả file (file descriptor)
flush()	Xóa sạch bộ nhớ đệm của luồng file.
isatty()	Trả về TRUE nếu file được kết nối với một thiết bị đầu cuối.
read(n)	Đọc n kí tự trong file.
readable()	Trả về TRUE nếu file có thể đọc được.
readline(n=-1)	Đọc và trả về một dòng từ file. Đọc nhiều nhất n byte/ký tự nếu được chỉ định.
readlines(n=-1)	Đọc và trả về một danh sách các dòng từ file. Đọc nhiều nhất n byte/ký tự nếu được chỉ định.
seek(offset,from=SEEK_SET)	Thay đổi vị trí hiện tại bên trong file.
seekable()	Trả về TRUE nếu luồng hỗ trợ truy cập ngẫu nhiên.
tell()	Trả về vị trí hiện tại bên trong file.
truncate(size=None)	Cắt gọn kích cỡ file thành kích cỡ tham số size.
writable()	Trả về TRUE nếu file có thể ghi được.
write(s)	Ghi s kí tự vào trong file và trả về.
writelines(lines)	Ghi một danh sách các dòng và file.

Chương 5: Điều khiển luồng

5.1 Luồng cơ bản

Khi python thực hiện xử lý code, trình thông dịch sẽ bắt đầu từ dòng đầu tiên của chương trình và sau đó xử lý tuần tự các dòng phía sau. Chúng ta thử nghiệm với chương trình đơn giản như sau:

```
1  def f(x):
2    """Hàm tính và trả về giá trị x*x"""
3   return x * x
4
5  print("Chương trình chính bắt đầu từ đây")
6  print("4 * 4 = %s" % f(4))
7  print("Dòng lệnh cuối cùng -- bye")
```

Quy tắc cơ bản là các câu lệnh được xử lý từ trên xuống dưới. Nếu một số lệnh được viết trên cùng một dòng (cách nhau bởi;), thì chúng được xử lý từ trái sang phải (việc viết nhiều câu lệnh trên một dòng sẽ khiến chương trình khó đọc code - nên tránh).

Trong ví dụ này, trình thông dịch bắt đầu ở trên cùng (dòng 1). Nó tìm từ khóa def và ghi nhớ rằng hàm f được định nghĩa ở đây. (Nó sẽ chưa thực thi phần thân của hàm, dòng số 3 chỉ được thực hiện chúng ta gọi hàm.)

Trình thông dịch thấy khoảng thụt lề nơi phần thân của hàm dừng lại: thụt lề ở dòng 5 khác với thụt lề của dòng đầu tiên trong thân hàm (line2), và do đó thân hàm đã kết thúc và việc thực thi sẽ tiếp tục với dòng số 5. Ở dòng 5, trình thông dịch sẽ in đầu ra `Chương trình chính bắt đầu từ đây`. Sau đó, dòng 6 được thực hiện. Lúc này lời gọi hàm f(4) sẽ gọi hàm f(x) được định nghĩa ở dòng 1. Trong đó x sẽ nhận giá trị 4.

Sau đó, hàm f được thực thi và tính toán và trả về 4 * 4 ở dòng 3. Giá trị 16 được sử dụng trong dòng 6 để thay thế f(4) và sau đó chuỗi %s của đối tượng 16 được in như một phần của lệnh in ở dòng 6. Trình thông dịch sau đó chuyển sang dòng 7 trước khi chương trình kết thúc.

Luồng hoạt động của chương trình như mô tả bên trên được coi là luồng cơ bản. Trong chương này chúng ta sẽ tìm hiểu kỹ hơn về các câu lệnh dùng để điều khiển luồng trong chương trình.

5.1.1 Kiểu dữ liệu bool và biểu thức điều kiện

Kiểu dữ liệu bool

Trong Python có một kiểu dữ liệu chỉ mang một trong hai giá trị là `True` hoặc `Fale`. Đây là kiểu dữ liệu `bool`.

```
>>> a = True
>>> print(a)
True
>>> type(a)
<type 'bool'>
>>> b = False
```

```
>>> print(b)
False
>>> type(b)
<type 'bool'>
```

Kiểu dữ liệu `bool` có thể sử dụng các toán tử logic như `and`, `or` và `not`:

```
>>> True and True #toán tử logic and
True
>>> True and False
False
>>> False and True
False
>>> True and True True
>>> c = a and b
>>> print(c)
False
```

Biểu thức điều kiện

Biểu thức điều kiện là các biểu thức trả về giá trị True hoặc False. Biểu thức điều kiện là sự kết hợp của các toán tử điều kiện, ví dụ:

```
>>> x = 30 # gán giá trị 30 cho biến x
>>> x > 15 # kiểm tra xem x có lớn hơn 15 không
True
>>> x > 42 # x có lớn hơn 42 không
False
>>> x == 30 # x có bằng 30 không?
True
>>> x == 42 # x có bằng 42 không?
False
>>> not x == 42 # is x not the same as 42?
True
>>> x != 42 # x có khác giá trị 42 không?
True
>>> x > 30 # x lớn hơn 30 không?
False
>>>x >=30 #x có lớn hơn hoặc bằng 30 không?
True
```

5.2 Luồng điều khiển If-then-else

Việc ra quyết định là cần thiết khi chúng ta muốn thực thi một đoạn code chỉ khi nó thỏa mãn kiện nào đó. Lệnh if...elif...else được sử dụng trong Python để phục vụ cho mục đích này.

5.2.1 Lệnh if

```
if điều kiện:
   khối lệnh
```

Ở đây, chương trình đánh giá điều kiện và sẽ thực hiện các lệnh trong khối lệnh khi điều kiện là True. Nếu điều kiện False thì khối lệnh sẽ không được thực hiện.

Trong Python, khối lệnh của lệnh if được viết thụt lề vào trong. Khối lệnh của if bắt đầu với một khoảng thụt lề và dòng không thụt lề đầu tiên sẽ được hiểu là kết thúc lệnh if.

```
num = 3
if num > 0:
   print(num, "là số dương.")
print("Thông điệp này luôn được in.")

num = -1
if num > 0:
   print(num, "là số dương.")
print("Thông điệp này cũng luôn được in.")
```

```
3 là số dương.
Thông điệp này luôn được in.
Thông điệp này cũng luôn được in.
```

Trong ví dụ trên, `num > 0` là điều kiện, khối lệnh của if được thực thi khi thỏa mãn điều kiện. Khi `num` bằng 3, kiểm tra điều kiện, thấy đúng, khối lệnh của if được thực thi. Khi num bằng -1, không khỏa mãn điều kiện, khối lệnh của `if` bị bỏ qua và thực hiện lệnh `print()` cuối cùng.

Chú ý kỹ hơn một chút, ta sẽ thấy rằng lệnh `print()` không được viết thụt lề, điều này nói lên rằng, `print()` nằm ngoài khối lệnh if, nên nó sẽ được thực hiện, bất kể điều kiện là gì.

5.2.2 Lệnh if...else

Lệnh if...else kiểm tra điều kiện và thực thi khối lệnh if nếu điều kiện đúng. Nếu điều kiện sai, khối lệnh của else sẽ được thực hiện. Thụt đầu dòng được sử dụng để tách các khối lệnh.

```
if điều kiện:
    Khối lệnh của if
else:
    Khối lệnh của else
```

Ví dụ:

```
# Kiem tra xem so am hay duong
# Va hien thi thong bao phu hop

num = 3
```

```
if num >= 0:
   print("So duong hoac bang 0")
else:
   print("So am")

num1=-1

if num1 >= 0:
   print("So duong hoac bang 0")
else:
   print("So am")
```

```
So duong hoac bang 0
So am
```

5.3 Vòng lặp For

Để lặp qua từng phần tử trong một kiểu dữ liệu dạng danh sách hay một chuỗi, ta có thể sử dụng vòng lặp for. Cú pháp của vòng lặp for như sau:

```
for bien_lap in chuoi_lap:
  Khối lệnh của for
```

Trong cú pháp trên, chuoi_lap là chuỗi cần lặp, bien_lap là biến nhận giá trị của từng mục bên trong chuoi_lap trên mỗi lần lặp. Vòng lặp sẽ tiếp tục cho đến khi nó lặp tới mục cuối cùng trong chuỗi.

Khối lệnh của for được thụt lề để phân biệt với phần còn lại của code.

```
#Lặp chữ cái trong giaotrinhpythonkythuat
for chu in 'python':
    print('Chữ cái hiện tại:', chu)

#Lặp từ trong chuỗi
chuoi = ['bố','mẹ','em']
for tu in chuoi:
    print('Anh yêu', tu)
```

```
Chữ cái hiện tại: p
Chữ cái hiện tại: y
Chữ cái hiện tại: t
Chữ cái hiện tại: h
Chữ cái hiện tại: o
Chữ cái hiện tại: n
Anh yêu bố
Anh yêu mẹ
Anh yêu em
```

Ví dụ: Để tính tổng tất cả các phần tử của List

```
# Tính tổng tất cả các số trong danh sách A
# Danh sách A
```

```
A = [1, 3, 5, 9, 11, 2, 6, 8, 10]
# Biến để lưu trữ tổng các số là tong, gán giá trị ban đầu bằng 0
tong = 0
# Vòng lặp for, a là biến lặp
for a in A:
     tong = tong+a
# Đầu ra: Tổng các số là 55
print("Tổng các số là", tong)
```

```
Tổng các số là 55
```

5.4 Vòng lặp while

Trong Python, while được dùng để lặp lại một khối lệnh, đoạn code khi điều kiện kiểm tra là đúng. while dùng trong những trường hợp mà chúng ta không thể dự đoán trước được số lần cần lặp là bao nhiêu.

Cú pháp của while trong Python:

```
while điều_kiện_kiểm_tra:
      Khối lệnh của while
```

Trong vòng lặp while, `điều_kiện_kiểm_tra` sẽ được kiểm tra đầu tiên. Khối lệnh của vòng lặp chỉ được nạp vào nếu `điều_kiện_kiểm_tra` là `True`. Sau một lần lặp, `điều_kiện_kiểm_tra` sẽ được kiểm tra lại. Quá trình này sẽ tiếp tục cho đến khi `điều_kiện_kiểm_tra` là False. Trong Python mọi giá trị khác 0 đều là `True`, `None` và `0` được hiểu là `False`. Đặc điểm này của `while` có thể dẫn đến trường hợp là `while` có thể không chạy vì ngay lần lặp đầu tiên `điều_kiện_kiểm_tra` đã False. Khi đó, khối lệnh của `while` sẽ bị bỏ qua và phần code ngay sau đó sẽ được thực thi.

Ví dụ:

```
count = 0
n = 0
while (count < 8):
      print ('Số thứ', n,' là:', count)
      n = n + 1
      count = count + 1
print ("Hết rồi!")
```

```
Số thứ 0  là: 0
Số thứ 1  là: 1
Số thứ 2  là: 2
Số thứ 3  là: 3
Số thứ 4  là: 4
Số thứ 5  là: 5
Số thứ 6  là: 6
Số thứ 7  là: 7
Hết rồi!
```

5.5 Xử lý ngoại lệ

Ngoại lệ có thể là bất kỳ điều kiện bất thường nào trong chương trình mà phá vỡ luồng thực thi chương trình đó. Bất cứ khi nào một ngoại lệ xuất hiện, mà không được xử lý, thì chương trình ngừng thực thi và vì thế code không được thực thi.

Python đã định nghĩa sẵn rất nhiều ngoại lệ, mà đã được trình bày trong chương Standard Exception. Trong mục này chúng ta sẽ tìm hiểu cách xử lý ngoại lệ cũng như cách tạo các Custom Exception (ngoại lệ tuỳ chỉnh) như thế nào.

Dưới đây là cú pháp của khối try....except...else trong Python:

```
try:
   #Các câu lệnh thự hiện có thể sinh ra ngoại lệ (exception);
except ExceptionI:
   #Nếu ngoại lệ là loại ExceptionI, Thì sẽ thực thi khối lệnh
này
except ExceptionII:
   #Nếu ngoại lệ là loại ExceptionII, Thì sẽ thực thi khối lệnh
này
else:
   #Nếu không phát sinh ngoại lệ thì thực hiện khối code này
```

Dưới đây là một số điểm bạn cần lưu ý:

- Phần code khả nghi mà có khả năng sinh ra ngoại lệ cần được đặt trong khối try.
- Sau khối try là lệnh except. Có thể có một hoặc nhiều lệnh except với một khối try .
- Lệnh except xác định exception mà xảy ra. Trong trường hợp mà exception đó xảy ra, thì lệnh tương ứng được thực thi phụ thuộc vào tên loại Exception.
- Ở cuối khối try, có thể có thêm lệnh else. Nó được thực thi khi không có exception nào xảy ra.

Ví dụ sau mở một file để ghi trong khi bạn không có quyền ghi, do đó nó sẽ tạo một exception:

```
try:
   fh = open("testfile", "r")
   fh.write("Ví dụ về ngoại lệ!!")
except IOError:
   print("Error: Không tìm thấy file")
else:
   print("Ghi file thành công")
```

```
Error: Khong tim thay file
```

Ví dụ sau mở một file, ghi nội dung vào file này và sau đó đóng file, tất cả hoạt động đều thành công:

```
try:
```

```
    fh = open("testfile", "w")
    fh.write("Ví dụ về ngoại lệ!!")
except IOError:
    print("Error: Không tìm thấy file")
else:
    print("Ghi file thành công")
    fh.close()
```

```
Ghi file thành công
```

Mệnh đề except mà không xác định Exception trong Python

Lệnh except cũng có thể được sử dụng mà không cần xác định loại exception nào. Lệnh try-except này bắt tất cả exception mà xuất hiện. Lưu ý khi sử dụng loại try-except sẽ làm code khó bảo trì vì khi phát sinh ngoại lệ chúng ta sẽ không biết ngoại lệ này là ngoại lệ gì, xuất phát ở đâu bởi vì nó bắt tất cả exception được sinh ra.

Try...Finally

Try...finally là một cách khác để viết lệnh try trong Python.

Finally còn được gọi là mệnh đề clean-up/termination vì nó luôn luôn chạy bất kể có lỗi nào xảy ra trong block try.

Thường thì các câu lệnh trong finally được dùng để thực hiện công việc giải phóng tài nguyên.

Ví dụ cho các hoạt động của file để minh họa rõ về finally:

Sau khi thực hiện xong các thao tác với file trong Python thì bạn cần đóng nó lại. Đóng file để đảm bảo quy chế đóng mở và giải phóng bộ nhớ cho chương trình.

```
try:
 f = open("test.txt",encoding = 'utf-8')
 # thực hiện các thao tác với tệp
finally:
 f.close()
```

Bằng cách này, ta có thể yên tâm file được đóng đúng ngay cả khi phát sinh ngoại lệ khiến chương trình dừng đột ngột.

Một ví dụ khác có ngoại lệ:

```
mauso = input("Bạn hãy nhập giá trị mẫu số: ")
try:
 ketqua = 15/int(mauso)
 print("Kết quả là:",ketqua)
finally:
 print("Bạn đã nhập số không thể thực hiện phép tính.")
```

`Finally` luôn luôn chạy bất kể có lỗi xảy ra hay không.

Khi bạn nhập input là 5, chương trình trả về kết quả:

```
Bạn hãy nhập giá trị mẫu số: 5
Kết quả là: 3.0
Bạn đã nhập số không thể thực hiện phép tính.
```

Và khi input là 0, kết quả hiển thị:

```
Bạn hãy nhập giá trị mẫu số: 0
Bạn đã nhập số không thể thực hiện phép tính.
```

Chương 6: Hàm và module

Các hàm cho phép chúng ta nhóm một số câu lệnh thành một khối để thực hiện một tác vụ cụ thể nào đó. Hàm giống như một hộp đen, chúng ta không biết bên trong hộp đen hoạt động thế nào, chúng ta chỉ cần truyền vào tham số sau đó hàm thực hiện các xử lý để trả về kết quả. Ví dụ, hàm math.sqrt: chúng ta không biết hàm này tính căn bậc hai như thế nào, nhưng chúng ta biết về mặt giao diện: nếu chúng ta chuyển x vào hàm, nó sẽ trả về (một giá trị gần đúng của) $\sqrt{x}$.

Việc trừu tượng hoá hàm này là một điều hữu ích khi lập trình: đó là một kỹ thuật để chia chương trình thành các thành phần (hộp đen) nhỏ hơn, tất cả đều hoạt động cùng nhau thông qua các chức năng được xác định rõ, nhưng không cần biết về các xử lý bên trong của nhau. Hàm giúp phân chia chương trình tốt hơn, và cho phép tái sử dụng lại mã nguồn.

Chúng ta có thể nhóm các hàm lại với nhau để tạo thành module, module cũng được dùng để tạo ra các thư viện.

6.1 Hàm

Python cung cấp nhiều các hàm dựng sẵn (built-in-function), ngoài ra ta có thể tự định nghĩa các hàm của riêng mình. Những hàm này còn được gọi là hàm do người dùng định nghĩa.

- Hàm sau khi được định nghĩa sẽ không tự thực thi.
- Hàm chỉ thực thi khi được gọi đến.

> Khi định nghĩa hàm ta nên đặt tên hàm là một động từ, vì hàm thể hiện một hành động, một tác vụ của chương trình.

Một số quy tắc khi định nghĩa hàm trong Python

Trong Python, chúng ta định nghĩa hàm theo quy tắc sau:

- Định nghĩa hàm sẽ bắt đầu với từ khóa `def`, sau đó là tên * hàm và cặp dấu ngoặc ( )
- Cặp dấu ( ) sẽ chứa các tham số của hàm (nếu có)
- Câu lệnh đầu tiên của hàm có thể là một lệnh tùy chọn, để mô tả về hàm (còn gọi là `docstring`)
- Thân của hàm sẽ bắt đầu với một dấu : và được thụt lề. Lệnh `return` dùng để thoát ra khỏi hàm, và trả lại giá trị từ hàm.

```
def my_function(arg1, arg2, ..., argn):
  """Phần docstring để mô tả về hàm."""
  # Phần thực thi của hàm
  return result # tuỳ chọn

#Phần bên ngoài hàm
some_command
```

Trong đó:

- my_function là tên hàm do người dùng định nghĩa (thường là động từ)
- arg1, arg2, ..., argn là các tham số của hàm

Truyền tham số cho hàm

> Để truyền tham số cho hàm, bạn đọc có thể đọc trong phần (3.4 - Truyền tham số cho hàm)

Gọi hàm Hàm được gọi thông qua tên, khi hàm được gọi thì lúc đó code trong phần thân hàm mới được xử lý.

Khi gọi hàm thì phụ thuộc vào từng hàm mà cần truyền tham số phù hợp.

Ví dụ về việc gọi hàm:

```
my_function(1,2,3,4)
```

6.2 Module

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.

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.

File 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 ---')
```

File 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)
```

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.

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 from ... import ... Ta có thể chỉ định sẽ import hàm nào trong module bằng from ... import ... Ví dụ như sau:

```
from my_math import sum_range, fact, message

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). Nó giúp bạn phân biệt rõ hàm của module.

6.3 Hàm vô danh

Trong Python, một hàm vô danh là một hàm được định nghĩa mà không có tên. Trong khi các hàm bình thường sẽ được định nghĩa bằng các từ khóa def trong Python, thì các hàm vô danh sẽ được định nghĩa bằng từ khóa lambda. Vì vậy các hàm vô danh còn được gọi là các Lambda Function.

Làm cách nào sử dụng Lambda Function trong Python?

Một lambda function trong Python sẽ có cú pháp như sau:

Cú pháp của Lambda Function trong python lambda arguments: expression

Các hàm lambda có thể có bất kỳ số đối số nào nhưng chỉ có một biểu thức. Biểu thức được đánh giá và trả về. Các lambda function có thể được sử dụng ở bất cứ chỗ nào yêu cầu các đối tượng hàm.

Ví dụ về Lambda Function trong python

Dưới đây là một ví dụ về lambda function để xử lý việc tăng gấp đôi giá trị đầu vào:

Ví dụ về việc sử dụng các Lambda Function trong Python

```
double = lambda x: x * 2
print(double(5))
```

```
Kết quả:
10
```

Ở ví dụ trên `lambda x: x * 2` sẽ là hàm lamba. Ở đây `x` là đối số vào `x*2` là biểu thức được đánh giá và trả về.

Function này sẽ không có tên. Nó trả về một object được định danh là `double`. Giờ thì chúng ta có thể gọi nó như một hàm bình thường. Ví dụ:

```
double = lambda x: x * 2
```

Nó sẽ tương tự như các khai báo function như sau:

```
def double(x):
   return x * 2
```

Sử dụng Lambda Function trong Python

Chúng ta thường sử dụng lambda function khi mà cần một hàm không tên trong một khoảng thời gian ngắn.

Trong Python, chúng ta thường sử dụng lambda function làm đối số cho một hàm bậc cao hơn (một hàm mà có thể nhận các hàm khác làm đối số). Lambda function thường được sử dụng cùng với các hàm tích hợp sẵn như filter(), map()...

6.4 Map

Hàm map() tích hợp sẵn trong Python có tác dụng duyệt tất cả các phần tử của một iterable (list, tuple, dictionary...) qua một hàm cho trước và trả về một list kết quả sau khi thực thi.

```
map(function, iterable, ...)
```

Các tham số của hàm `map()`:

- `function`: Hàm thực thi cho từng phần tử trong `iterable`.
- `iterable`: một `list`, `tuple`, `dictionary`... muốn duyệt. Bạn có thể truyền nhiều iterablecho hàm `map()`.

Giá trị trả về từ `map()`

Hàm map() duyệt tất cả các phần tử của iterable qua function và trả về một list kết quả sau khi thực thi.

Giá trị trả về từ map() được gọi là map object. Đối tượng này có thể được truyền vào các hàm list() (để tạo list trong Python), hay set() (để tạo một set các phần tử mới)...

Ví dụ:

```
def binhphuong(n):
```

```
  return n*n

number = (25, 100, 225, 400)
ketqua = map(binhphuong, number)

# chuyen map object thanh list
print(list(ketqua))
```

```
[625, 10000, 50625, 160000]
```

Ở ví dụ trên, mỗi phần tử trong tuple ban đầu đều được bình phương lên.

Cách sử dụng hàm lambda với map() Vì map() luôn cần tham số để truyền vào, vậy nên các hàm lambda thường xuyên được sử dụng với map().

Trong Python, hàm lambda hay hàm vô danh được định nghĩa mà không có tên. Nếu các hàm bình thường được định nghĩa bằng cách sử dụng từ khóa def, thì hàm vô danh được định nghĩa bằng cách sử dụng từ khóa lambda.

```
number = (5, 10, 15, 20)
result = map(lambda x: x*x, number)

# chuyen map object thanh list
sobinhphuong = list(result)
print(sobinhphuong)
```

Chạy chương trình, kết quả trả về là:

```
[25, 100, 225, 400]
```

6.5 Map function

Map là một function với 2 tham số đầu vào:

```
r= map(func,sep)
```

Map sử dụng một function f, array là input sau đó output array. Map sẽ apply hàm f này vào các phần tử trong mảng. Có thể hiểu map có ý tưởng giống for để duyệt qua các phần tử.

```
temp_c = [10, 3, -5, 25, 1, 9, 29, -10, 5]
temp_K = list(map(lamda x: x+ 273.5, tempc))
list(temp_K)
```

Để hiểu hơn về map, chúng ta thử giải quyết bài toán sau:

```
list_words = [“big”,”small”, “able”, “about”, “hairdresser”, “laboratory”]
```

Sử dụng map để đếm số ký tự của mỗi phần tử.

Trong python có hàm len() để lấy ra độ dài của phần tử. Vậy sử dụng map như sau:

```
print(list(map(len,list_words)))
```

6.6 Reduce function

Cũng như map function, reduce function là function có 2 tham số đầu vào là function f và một dữ liệu tuần tự. Thay vì duyệt qua từng phần tử thì reduce sẽ combine mỗi 2 phần tử của list bằng function f đầu vào. Chúng ta hãy xem xét ví dụ tính tổng dưới đây.

```
# define list
numbers = [1,4,6,2,9,10]

def sum(x,y)
    return x+y

from functools import reduce
print(reduce(sum,numbers)
```

Kết quả sẽ là: ((((1+4)+6)+2)+9)+10 = 32

Để hiểu hơn về reduce thì chúng ta làm ví dụ sau: Hãy tìm số lớn nhất bằng cách sử dụng reduce

```
from functools import reduce

# khởi tạo danh sách
lis = [ 1 , 3, 5, 6, 2, ]

# dùng reduce để tìm giá trị lớn nhất
print ("Giá trị lớn nhất của danh sách là : ",end="")
print (reduce(lambda a,b : a if a > b else b,lis))
```

```
Giá trị lớn nhất của danh sách là : 6
```

6.7 List Comprehension

List comprehension là cách tạo ra một list dữ liệu mới dựa vào dữ liệu cũ theo một điều kiện nào đó.

```
newlist = [expression for item in iterable if condition ==
True]
```

Trong đó:

- `expression` chính là biểu thức trả về
- `iterable` là tập dữ liệu iterator

- `condition` là điều kiện trả về, phần tử nào có `condition = True` thì biểu thức `expression` tương ứng sẽ được trả về.

Giả sử có một mảng các số như sau:

```
numbers = [1,3,86,4,54,54,45,65,6,57,67,87,82]
```

Để tạo 1 list mới từ list number:

- **Cách thứ nhất**: Sử dụng vòng lặp bình thường

```
numbers = [1,3,86,4,54,54,45,65,6,57,67,87,82]

new_numbers = []
for n in numbers:
    if n % 2 == 0:
        new_numbers.append(n)

print(new_numbers)
```

```
[86, 4, 54, 54, 6, 82]
```

- **Cách thứ hai**: Sử dụng `list comprehension`.

```
numbers = [1,3,86,4,54,54,45,65,6,57,67,87,82]

new_numbers = [x for x in numbers if x % 2 == 0]

print(new_numbers)
```

```
[86, 4, 54, 54, 6, 82]
```

Ví dụ: Tạo mảng mới với giá trị gấp đôi mảng cũ.

```
numbers = [1,3,5,7,9]

new_numbers = [x*2 for x in numbers]

# Kết quả: [2, 6, 10, 14, 18]
print(new_numbers)
```

Chương 7: Numerical Python (numpy)

NumPy (Numerical Python) là một thư viện Python mã nguồn mở được sử dụng trong nhiều lĩnh vực của khoa học kỹ thuật. Người dùng NumPy gồm tất cả mọi người từ người mới học đến các nhà nghiên cứu có kinh nghiệm. NumPy API được sử dụng rộng rãi trong Pandas, SciPy, Matplotlib, scikit-learning, scikit-image và hầu hết các thư viện tính toán trong khoa học dữ liệu, học máy.

Thư viện NumPy chứa các cấu trúc dữ liệu ma trận và mảng đa chiều. Numpy cung cấp ndarray, một đối tượng mảng n chiều với các hàm để tính toán tương đối hiệu quả. NumPy có thể được sử dụng để thực hiện nhiều phép toán khác nhau trên mảng. Nó bổ sung các cấu trúc dữ liệu mạnh mẽ hơn cho Python để đảm bảo việc tính toán hiệu quả với các mảng và ma trận, đồng thời nó cung cấp một thư viện khổng lồ các hàm toán để tính toán dựa trên các mảng và ma trận.

7.1 Cài đặt numpy

Để xem chi tiết cách cài đặt numpy bạn có thể vào link Xem cách cài đặt

Nếu đã cài đặt python, bạn chỉ cần cài đặt `numpy` theo cách sau đây:

```
conda install numpy
```

Hoặc

```
pip install numpy
```

7.2 Sử dụng numpy

Để sử dụng numpy bạn cần thực hiện `import` thư viện như sau:

```
import numpy as np
```

Định danh `numpy` bằng `np` giúp việc đọc code của numpy được rõ ràng hơn. Đây là quy ước phổ biến mà nhiều người sử dụng, bạn nên tuân theo để bất kỳ ai làm việc với code của bạn có thể dễ dàng hiểu.

NumPy cung cấp nhiều cách để tạo mảng và thao tác dữ liệu số bên trong nhanh chóng và hiệu quả. Mặc dù List trong Python có thể chứa các kiểu dữ liệu khác nhau, nhưng tất cả các phần tử trong mảng NumPy phải có chung kiểu dữ liệu. Các phép toán được thực hiện trên mảng sẽ kém hiệu quả nếu các phần tử trong mảng có các kiểu dữ liệu khác nhau.

Tại sao sử dụng NumPy?

Một trong những lý do khiến NumPy rất quan trọng đối với các tính toán trong Python là vì Numpy được thiết kế để mang lại hiệu quả trên các mảng dữ liệu lớn. Có một số lý do cho điều này:

Mảng NumPy nhanh hơn và nhỏ gọn hơn so với List Python. Một mảng tốn ít bộ nhớ hơn và thuận tiện khi sử dụng. Ngoài ra các phép toán dựa trên mảng cũng nhanh hơn nhiều so với việc thực hiện tương tự trên List.

- NumPy lưu trữ dữ liệu trong một khối bộ nhớ có các ô nhớ liền kề, độc lập với các đối tượng Python khác. Thư viện thuật toán của NumPy được viết bằng ngôn ngữ C có thể hoạt động trên bộ nhớ mà không cần kiểm tra kiểu. Mảng NumPy cũng sử dụng ít bộ nhớ hơn nhiều so với các chuỗi Python tích hợp sẵn.
- Các phép toán NumPy thực hiện các phép tính phức tạp trên toàn bộ mảng mà không cần đến các vòng lặp for Python (Vòng lặp for thường có hiệu suất thấp)

7.2.1 Tạo mảng

Cách đơn giản nhất để tạo mảng đó là sử dụng hàm `array`. Hàm này chấp nhận tham số đầu vào là dữ liệu tuần tự (hoặc 1 mảng khác) và trả về một mảng NumPy gồm dữ liệu truyền vào. Ví dụ sau dùng để tạo mảng từ 1 list trong python:

```
>>> data1 = [6, 7.5, 8, 0, 1]
>>> arr1 = np.array(data1)
>>> arr1
array([6. , 7.5, 8. , 0. , 1. ])
```

Để tạo ra mảng nhiều chiều ta cũng có thể thực hiện truyền vào 1 list lồng nhau:

```
>>> data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
>>> arr2 = np.array(data2)
>>> arr2
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Các bạn có thể thấy data2 là một list lồng nhau (list bên trong list), mảng array2 có 2 chiều với số chiều tương tự như list. Chúng ta có thể xem số chiều bằng thuộc tính `ndim` và `shape`:

```
>>>arr2.ndim
2
>>>arr2.shape
(2, 4)
```

Khi chuyển từ list sang mảng Numpy, Numpy đã tự động chọn kiểu dữ liệu, ta có thể xem kiểu dữ liệu của mảng qua thuộc tính dtype như sau:

```
arr1.dtype
dtype('float64')

arr2.dtype
dtype('int64')
```

Chúng ta có thể chỉ định kiểu dữ liệu bên trong mảng thay vì `Numpy` chọn kiểu dữ liệu tự động, để thực hiện ta sử dụng tham số `dtype` như sau:

```
>>> data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
>>> arr2 = np.array(data2, dtype=int32)
>>> arr2
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Các kiểu dữ liệu của numpy cung cấp:

Kiểu	Code	Mô tả
`int8, uint8`	`i1, u1`	Số nguyên 8-bit (1 byte)
`int16, uint16`	`i2, u2`	Số nguyên 16 bit
`int32, uint32`	`i4, u4`	Số nguyên 32 bit
`int64, uint64`	`i8, u8`	Số nguyên 64 bit
`float16`	`f2`	Bit dấu, số mũ 5 bit, phần định trị 10 bit
`float32`	`f4 or f`	bit dấu, số mũ 8 bit, phần định trị 23 bit
`float64`	`f8 or d`	float: bit dấu, số mũ 11 bit, phần định trị 52 bit
`float128`	`f16 or g`	Extended-precision floating point
`complex64`, `complex128`, `complex256`	`c8, c16, c32`	Số phức được biểu diễn bởi 32, 64, hoặc 128 bit float
`bool`	?	Chỉ gồm 2 giá trị `True` hoặc `False`
`object`	O	Kiểu đối tượng Python, có thể là bất kỳ loại đối tượng nào
`string_`	S	Chuỗi ký tự theo bảng mã ASCII
`unicode_`	U	Chuỗi ký tự Unicode

> Đừng lo lắng về việc ghi nhớ các Dtypes Numpy, đặc biệt nếu bạn là người dùng mới. Thường chỉ cần phải quan tâm đến loại dữ liệu bạn đang cần nhớ các kiểu cơ bản như float, số phức, số nguyên, boolean, string hoặc object Python nói chung. Các kiểu dữ liệu cụ thể sẽ giúp bạn tổ chức bộ nhớ tốt hơn.

Ngoài `np.array`, có một số chức năng khác để tạo mảng mới. Ví dụ: `zeros` và `ones` tạo ra các mảng 0 hoặc 1, tương ứng với độ dài hoặc kích thước cho trước. `empty` tạo ra một mảng mà không cần khởi tạo các giá trị của nó thành bất kỳ giá trị cụ thể nào.

```
>>> np.zeros(10)
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

>>> np.zeros((3, 6))
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

```
>>> np.empty((2, 3, 2))
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],
       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

> Sẽ không an toàn nếu giả sử rằng `np.empty` sẽ trả về một mảng gồm tất cả các số không. Trong một số trường hợp, nó có thể trả về các giá trị "rác" chưa được khởi tạo.

`arange` là hàm dùng để tạo mảng giống như `range` trong python:

```
>>> np.arange(15)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
14])
```

Hàm	Mô tả
`array`	Chuyển đổi dữ liệu từ các kiểu dữ liệu khác của python ((list, tuple, array, or other sequence type) sang `ndarray`, có thể chỉ định kiểu dữ liệu sẽ chuyển đổi sang qua tham số `dtype`.
`asarray`	Chuyển đổi dữ liệu sang `ndarray`, nhưng không copy dữ liệu nếu dữ liệu truyền vào là `ndarray`
`arange`	Giống hàm `range` trong python nhưng trả về `ndarray` thay vì List
`ones, ones_like`	Tạo ra mảng gồm các số 1 với kiểu dữ liệu `dtype` truyền vào; `ones_like` tạo ra mảng số 1 có số chiều và kiểu dữ liệu giống với mảng truyền vào
`zeros, zeros_like`	Giống như `ones` và `ones_like` nhưng nó tạo ra mảng các số 1
`empty, empty_like`	Tạo ra mảng mới và khởi tạo bộ nhớ nhưng các phần tử mảng chưa khởi tạo dữ liệu, khác với `ones` và `zeros` là sẽ tạo trước các phần từ là 0 hoặc 1
`eye, identity`	Tạo ra ma trận vuông N × N là ma trận đơn vị (ma trận có các phần tử trên đường chéo bằng 1, các phần tử khác bằng 0)

7.2.2 Phép toán số học với Numpy

Mảng rất quan trọng vì chúng cho phép bạn thực hiện các phép toán một cách đồng thời trên các phần tử của dữ liệu mà không cần viết bất kỳ vòng lặp `for` nào. Các thực hiện này NumPy gọi là vectơ hóa. Bất kỳ phép toán số học nào giữa các mảng có kích thước bằng nhau đều áp dụng phép toán dựa trên `element-wise`:

```
>>> arr = np.array([[1., 2., 3.], [4., 5., 6.]])
>>> arr
array([[1., 2., 3.],
       [4., 5., 6.]])

#Nhân 2 mảng
```

```
>>> arr * arr
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])

#Trừ 2 mảng
>>> arr - arr
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Phép toán vô hướng với vector có thể thực hiện bằng các áp dụng với từng phần tử của vector:

```
#Lấy 1 chia cho mảng
>>> 1 / arr
array([[1.    , 0.5   , 0.3333],
       [0.25  , 0.2   , 0.1667]])

# Nhân mảng với 0.5
>>>arr ** 0.5
array([[1.    , 1.4142, 1.7321],
       [2.    , 2.2361, 2.4495]])
```

Hoặc thực hiện so sánh 2 mảng có cùng kích thước:

```
>>> arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])

>>> arr2
array([[ 0.,  4.,  1.],
       [ 7.,  2., 12.]])

#So sánh từng phần tử của 2 mảng.
>>> arr2 > arr
array([[False,  True, False],
       [ True, False,  True]])
```

7.2.3 Cắt lát và chỉ mục

Chỉ số trong NumPy là một chủ đề lớn, ví dụ như có nhiều cách để chọn ra một tập con của mảng hoặc từng phần tử riêng rẽ. Mảng 1 chiều rất hơn giản, về cơ bản nó giống như List:

```
arr = np.arange(10)
```

```
arr
```

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

```
arr[5]
```

```
5
```

```
arr[5:8]
```

```
array([5, 6, 7])
```

```
arr[5:8] = 12
```

```
arr
```

```
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

Như bạn có thể thấy, nếu gán một giá trị vô hướng cho một vùng chọn, như `arr[5: 8] = 12`, giá trị này sẽ được gán cho toàn bộ vùng chọn. Một điểm khác biệt quan trọng đầu tiên so với list trong Python là khi sử dụng các vùng chọn thì từ list gốc chỉ có thể tạo ra một list con được chứ ko thay đổi được dữ liệu của list gốc. Còn trong numpy thì khi trích xuất mảng con thì nếu có bất kỳ thay đổi nào của mảng con sẽ làm thay đổi mảng gốc ban đầu. Lấy ví dụ như sau:

```
>>> arr_slice = arr[5:8]
>>> arr_slice
array([12, 12, 12])
```

Khi ta thay đổi giá trị trong `arr_slice` thì mảng gốc ban đầu sẽ thay đổi theo:

```
>>>arr_slice[1] = 12345
>>> arr
#Kết quả:
array([    0,     1,     2,     3,     4,    12, 12345,    12,
8,
           9])
```

Sử dụng [:] sẽ thực hiện gán dữ liệu cho tất cả các phần tử của mảng:

```
>>> arr_slice[:] = 64
#in ra
>>> arr
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

Nếu chưa quen với NumPy bạn có thể thấy lạ vì điều này, đặc biệt nếu bạn đã sử dụng các ngôn ngữ lập trình khác để thao tác với mảng. Các ngôn ngữ khác sẽ tạo ra một mảng mới thay vì tham chiếu đến mảng cha. Nghĩa là mảng con khi lấy ra trong Numpy sẽ vẫn là một phần của mảng cha, nếu có thay đổi gì với mảng con thì mảng cha sẽ thay đổi. Vì NumPy đã được thiết kế để có thể làm việc với các mảng rất lớn, bạn có thể tưởng tượng ra các vấn đề về hiệu suất và bộ nhớ nếu NumPy lúc nào cũng sao chép dữ liệu ra mảng mới.

> Nếu bạn muốn một bản sao của vùng chọn của `ndarray` thay vì một dạng tham chiếu, bạn có thể sao chép mảng bằng hàm copy() — ví dụ: `arr[5:8].copy()`.

Với mảng nhiều chiều, mỗi phần tử có chỉ số đơn không chỉ là một số mà có thể là 1 một mảng con. Ví dụ như:

```
>>> arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Lấy ra phần tử ở vị trí 2:
>> arr2d[2]
array([7, 8, 9])
```

Do đó, các phần tử riêng lẻ có thể được truy cập thông qua đệ quy hoặc có thể lấy theo một danh sách các chỉ số dạng list, các vị trí được phân tách bằng dấu phẩy để chọn các phần tử riêng lẻ. Ví dụ như:

```
>>> arr2d[0][2]
3
#Lấy phần tử theo chỉ số dạng list:
>>> arr2d[0, 2]
3
```

Trong mảng nhiều chiều chúng ta có thể lấy ra mảng con của một mảng bằng việc sử dụng "cắt mảng":

```
>>> arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10,
11, 12]]])

>>> arr3d
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Mảng con `arr3d[0]` là một mảng 2 × 3:

```
>>> arr3d[0]
array([[1, 2, 3],
       [4, 5, 6]])
```

Chúng ta có thể gán mảng con bởi một số, lúc này tất cả các phần tử của mảng con sẽ được thay bằng giá trị của của số gán. Ví dụ:

```
>>> old_values = arr3d[0].copy()
>>> arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10,
11, 12]]])
>>> arr3d[0] = 42
>>> arr3d
```

```
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

> Chú ý lại một lần nữa: Mảng con lấy ra qua chỉ mục chính là một phần của mảng cha ban đầu, mọi thay đổi với mảng con sẽ làm thay đổi mảng cha.

Cắt lát mảng Giống như list 1 chiều trong Python, NumPy có thể cắt lát một mảng thành mảng con với cú pháp tương tự.

Ví dụ với mảng 1 chiều

```
>>> arr
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])

>>> arr[1:6]
array([ 1,  2,  3,  4, 64])
```

Ví dụ với mảng 2 chiều:

```
>>> arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> arr2d
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

>>> arr2d[:2]   # Cắt lát mảng từ hàng đầu đến hết hàng thứ 2.
#Kết quả
array([[1, 2, 3],
       [4, 5, 6]])
```

Chúng ta có thể cắt theo nhiều chiều khác như sau:

```
>>> arr2d[:2, 1:]
array([[2, 3],
       [5, 6]])
```

Chỉ mục kiểu Boolean

Hãy xem xét một ví dụ trong đó chúng ta có dữ liệu trong một mảng và một mảng tên có trùng lặp. Ở đây ta sẽ sử dụng hàm `randn` trong `numpy.random` để tạo một số dữ liệu ngẫu nhiên theo phân phối chuẩn. Mảng gồm các số ngẫu nhiên `data` có số dòng tương ứng với số phần tử trong mảng `name` như sau:

```
>>> names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will',
'Joe', 'Joe'])

>>> data = np.random.randn(7, 4)
```

```
>>> names
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
dtype='<U4')
>>> data
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Bây giờ ta sẽ xem các phần tử nào trong mảng có tên là 'Bob':

```
>>> names == 'Bob'
array([ True, False, False,  True, False, False, False])
```

Như vậy vị trí các phần tử có giá trị là `True` tương ứng với vị trí các phần tử có tên là 'Bob' trong mảng `name`. Ở ví dụ này phần tử ở vị trí 0 và 3 có giá trị là True. Bây giờ ta sẽ lấy ra các phần tử của mảng `data` có vị trí tương ứng với giá trị True trong mảng `name == 'Bob'` như sau:

```
>>> data[names == 'Bob']
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

Như vậy là NumPy cho phép chúng ta lấy ra mảng con bằng cách truyền vào mảng giá trị Boolean.

Ngoài ra chúng ta có thể gán các phần tử của mảng bằng một giá trị khác theo điều kiện như sau:

```
>>> data[data < 0] = 0 #Gán các phần tử có giá trị < 0 thành
giá trị 0.
>>> data
array([[0.0929, 0.2817, 0.769 , 1.2464],
       [1.0072, 0.    , 0.275 , 0.2289],
       [1.3529, 0.8864, 0.    , 0.    ],
       [1.669 , 0.    , 0.    , 0.477 ],
       [3.2489, 0.    , 0.    , 0.1241],
       [0.3026, 0.5238, 0.0009, 1.3438],
       [0.    , 0.    , 0.    , 0.    ]])
```

7.3 Các phép toán trên ma trận

Các phép toán trên ma trận là những tính toán cơ bản khi làm việc với học máy. Trong phần này, tôi không đề cập đầy đủ tất cả các phép toán của nó, mà chỉ đề cập tới các phép toán cơ bản để có thể sài được với học máy cơ bản.

7.3.1. Nhân ma trận với một vô hướng

Nhân ma trận với một số (vô hướng) là phép nhân số đó với từng phần tử của ma trận.

$$\alpha[A_{ij}]_{mn} = [\alpha.\, A_{ij}]_{mn}$$

Ví dụ:

$$5\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 5 & 10 & 15 \\ 20 & 25 & 30 \end{bmatrix}$$

Các tính chất:

- Tính giao hoán: $\alpha A = A\alpha$
- Tính kết hợp: $\alpha(\beta A) = (\alpha\beta)A$
- Tính phân phối: $(\alpha + \beta)A = \alpha A + \beta A$

Ngoài ra, nhân ma trận với 1 sẽ không làm thay đổi ma trận: $1A = A$, còn nhân với 0 sẽ biến ma trận thành ma trận không $0A = \varnothing$.

Cách biểu diễn với Numpy:

```
# create matrix a
a = np.array([(1, 2, 3), (4, 5, 6)])
print(a)
# [[1 2 3]
#  [4 5 6]]

# Multiplying a by 5
b = 5 * a
print(b)
# [[ 5 10 15]
#  [20 25 30]]

# Multiplying a by -1
b = -1 * a
print(b)
# [[-1 -2 -3]
#  [-4 -5 -6]]
```

7.3.2. Cộng 2 ma trận

Là phép cộng từng phần tử tương ứng của 2 ma trận cùng cấp với nhau.

$$[A_{ij}]_{mn} + [B_{ij}]_{mn} = [A_{ij} + B_{ij}]_{mn}$$

Ví dụ:

$$\begin{bmatrix} 5 & 10 & 15 \\ 20 & 25 & 30 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 6 & 12 & 18 \\ 24 & 29 & 36 \end{bmatrix}$$

Các tính chất:

- Tính giao hoán: $A + B = B + A$
- Tính kết hợp: $A + (B + C) = (A + B) + C$
- Tính phân phối: $\alpha(A + B) = \alpha A + \alpha B$

Ngoài ra, dễ dàng thấy rằng cộng một ma trận với ma trận không thì không làm thay đổi ma trận đó: $A + \varnothing = A$.

Từ phép nhân ma trận với một số, ta có thể định nghĩa được phép trừ ma trận là phép trừ từng phần tử tương ứng trong ma trận: $A - \lambda B = A + (-\lambda)B, \lambda \in \mathbb{R}$.

Cách biểu diễn với Numpy:

```
# create matrix a
    a = np.array([(1, 2, 3), (4, 5, 6)])
    print(a)
    # [[1 2 3]
    #  [4 5 6]]

    # create matrix b
    b = np.array([(0, 5, 25), (4, 9, 9)])
    print(b)
    # [[0 5 25]
    #  [4 9  9]]

    # sum of a and b
    c = a + b
    print(c)
    # [[ 1  7 28]
    #  [ 8 14 15]]

    c = np.add(a, b)
    print(c)
    # [[ 1  7 28]
    #  [ 8 14 15]]

    # subtraction of a and b
    c = np.subtract(a, b)
    print(c)
    # [[  1  -3 -22]
    #  [  0  -4  -3]]

    c = a - b
    print(c)
    # [[  1  -3 -22]
    #  [  0  -4  -3]]
```

7.3.3. Nhân 2 ma trận

Nhân 2 ma trận là phép lấy tổng của tích từng phần tử của hàng tương ứng với cột tương ứng. Phép nhân này chỉ khả thi khi số cột của ma trận bên trái bằng với số hàng của ma trận bên phải. Cho 2 ma trận $[A]_{mp}$ và $[B]_{pn}$, tích chúng theo thứ tự đó sẽ là một ma trận có số hàng bằng với số hàng của A và số cột bằng với số cột của B: $[AB]_{mn}$.

$$C_{ij} = AB_{ij} = \sum_{k=1}^{p} A_{ik}B_{kj} \quad , \forall i = \overline{1,m}; j = \overline{1,n}$$

Ví dụ:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 22 & 28 \\ 49 & 64 \end{bmatrix} \qquad (1)$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad (2)$$

Các tính chất:

- Tính kết hợp: $A(BC) = (AB)C$
- Tính phân phối: $A(B + C) = AB + AC$, $(A + B)C = AC + BC$.

Lưu ý là phép nhân 2 ma trận không có tính chất giao hoán: $AB \neq BA$.

Nếu bạn để ý ở công thứ 2 phía trên thì sẽ thấy rằng việc nhân với ma trận đơn vị không làm thay đổi ma trận đó: $AI = IA = A$.

Cách biểu diễn với Numpy:

```
# create matrix A
    A = np.array([(1, 2, 3), (4, 5, 6)])
    print(A)
    # [[1 2 3]
    #  [4 5 6]]

    # create matrix B
    B = np.array([(0, 5), (4, 9), (9, 0)])
    print(B)
    # [[0 5]
    #  [4 9]
    #  [9 0]]

    # product of A and B
    C = A.dot(B)
    print(C)
    # [[35 23]
    #  [74 65]]
```

```
    C = np.dot(A, B)
    # [[35 23]
    #  [74 65]]
```

7.3.4. Chuyển vị ma trận

Chuyển vị là phép biến cột thành hàng và hàng thành cột của một ma trận. Cho ma trận $[A]_{mn}$ thì chuyển vị của nó là $[B_{ij}]_{nm} = [A_{ji}]^{\intercal}_{mn}$ ($\intercal$ là kí hiệu của phép chuyển vị) có $B_{ij} = A_{ji} \quad , \forall i, j$. Ví dụ:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^{\intercal} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Các tính chất:

- $(A^{\intercal})^{\intercal} = A$
- $(A + B)^{\intercal} = A^{\intercal} + B^{\intercal}$
- $(AB)^{\intercal} = B^{\intercal}A^{\intercal}$
- $(cA)^{\intercal} = cA^{\intercal}$

Ngoài ra, ta có thể thực hiện phép nhân số học với 2 véc-tơ để thu được 1 vô hướng bằng cách nhân với chuyển của 1 trong 2 véc-tơ: $ab = a^{\intercal}b$. Phép nhân kiểu này còn được gọi là phép nhân vô hướng, tức là tổng của tích mỗi phần tử tương ứng của 2 ma trận.

```
# create matrix a
    A = np.array([[1, 2, 3], [4, 5, 6]])

    # transpose matrix of A
    B = A.transpose()
    print(B)
    # [[1 4]
    #  [2 5]
    #  [3 6]]

    # vector inner product
    a = np.arange(10)
    b = np.ones(10)
    c = a.dot(b)
    print(c)
    # 45.0

    c = np.inner(a, b)
    print(c)
    # 45.0
```

Từ đoạn mã trên, ta có thể thấy rằng với Numpy, ta có thể thực hiện ngay được phép nhân vô hướng của 2 véc-tơ (inner product) mà không cần phải chuyển vị ma trận.

7.3.5 Ma trận nghịch đảo

Ma trận của ma trận vuông khả nghịch A cấp n là ma trận B sao cho tích của chúng là ma trận đơn vị cùng cấp: $AB = I_n$. Ma trận khả nghịch được kí hiệu là: A^{-1}. Tức là $AA^{-1} = I_n$.

Các tính chất:

- $(A^{-1})^{-1} = A$
- $(kA)^{-1} = k^{-1}A^{-1} \quad \forall k \neq 0$
- $(AB)^{-1} = B^{-1}A^{-1}$
- $(A^\intercal)^{-1} = (A^{-1})^\intercal$

Ngoài ra nếu để ý sẽ thấy ma trận đơn vị luôn có nghịch đảo là chính nó: $I_n^{-1} = I_n$, còn ma trận không không tồn tại nghịch đảo - hay ta gọi nó là không khả nghịch.

Cách biểu diễn với Numpy:

```
# create matrix a
    A = np.array([[1., 2.], [3., 4.]])

    # inverse matrix of A
    B = np.linalg.inv(A)
    print(B)
    # [[-2.   1. ]
    #  [ 1.5 -0.5]]
```

7.3.6 Phép nhân từng phần tử Hadamard

Là phép nhân từng phần tử tương ứng của 2 ma trận cùng cấp với nhau.

$$[A_{ij}]_{mn} \circ [B_{ij}]_{mn} = [A\_ijB_{ij}]_{mn}$$

Ví dụ:

$$\begin{bmatrix} 5 & 10 & 15 \\ 20 & 25 & 30 \end{bmatrix} \circ \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 5 & 20 & 45 \\ 80 & 115 & 180 \end{bmatrix}$$

Các tính chất:

- Tính giao hoán: $A \circ B = B \circ A$
- Tính kết hợp: $A \circ (B \circ C) = (A \circ B) \circ C$
- Tính phân phối: $A \circ (B + C) = A \circ B + A \circ C$

```python
# create matrix a
a = np.array([(1, 2, 3), (4, 5, 6)])
print(a)
# [[1 2 3]
#  [4 5 6]]

# create matrix b
b = np.array([(0, 5, 25), (4, 9, 9)])
print(b)
# [[0 5 25]
#  [4 9  9]]

# Multiplying of a and b
c = a * b
print(c)
# [[ 0 10 75]
#  [16 45 54]]

c = np.multiply(a, b)
print(c)
# [[ 0 10 75]
#  [16 45 54]]
```

7.3.7 Các phép toán theo từng phần tử (Hadamard) khác

Ngoài phép nhân Hadamard theo từng phần tử, ta cũng có các phép biến đổi khác tương tự như:

Phép chia Hadamard: $[A_{ij}]_{mn} \oslash [B_{ij}]_{mn} = [A_{ij}/B_{ij}]_{mn}$.

Phép lũy thừa Hadamard: $[A_{ij}]_{mn}^{p} = [A_{ij}^{p}]_{mn} \quad , \forall p \in \mathbb{R}$

Từ phép lũy thừa với số mũ phân số, ta có thể viết lại dưới dạng phép khai căn Hadamard: $\sqrt[p]{[A_{ij}]_{mn}} = [\sqrt[p]{A_{ij}}]_{mn} \quad , \forall p \in \mathbb{N}$

```python
# create matrix a
    a = np.array([(1., 2., 3.), (4., 5., 6.)])
    print(a)
    # [[1. 2. 3.]
    #  [4. 5. 6.]]

    # create matrix b
    b = np.array([(0., 5., 25.), (4., 9., 9.)])
    print(b)
    # [[0. 5. 25.]
    #  [4. 9.  9.]]

    # divide
    c = b / a # c = np.divide(b, a)
```

```
print(c)
# [[ 0.          2.5         8.33333333]
#  [ 1.          1.8         1.5       ]]

# power of 2
c = a ** 2
print(c)
# [[  1.   4.   9.]
#  [ 16.  25.  36.]]
```

7.3.8. Norm

Trong không gian véc-tơ, Norm là một công cụ để đo độ dài của véc-tơ hay nói cách khác là đo khoảng cách giữa 2 điểm trong không gian. Đáng lẽ ta phải tìm hiểu không gian véc-tơ là gì các tính chất của nó ra sao để có thể hiểu rõ được độ dài của các véc-tơ và quan hệ của chúng ra sao, nhưng tạm thời lúc này ta hiểu nôm na nó là một tập chứa các véc-tơ và khoảng cách 2 điểm như một đường bay nối thẳng 2 điểm đó như ở không gian 2 chiều.

Thường một norm cấp p (kí hiệu: L^p) hay sử dụng trong học máy được mô tả bằng công thức:

$$\|x\|_p = \left(\sum_i |x_i|^p \right)^{\frac{1}{p}}$$

Trong đó $p \in \mathbb{R}$ và $p \geq 1$, còn x_i là phần tử thứ i của véc-tơ. Dễ nhận thấy rằng một norm không thể nào nhỏ hơn 0 được, vì không có có khoảng cách nào là âm.

Ví dụ trong không gian Euclide n chiều, ta có độ dài của véc-tơ chính là một norm cấp 2: (L^2): $\|x\| = \sqrt{\sum_{i=1}^{n} x_i^2}$. Trong đó x_i là phần tử thứ i của véc-tơ, hay nói cách khác là tọa độ trên trục thứ i tương ứng của véc-tơ trong không gian.

Một norm cấp 2 thường được kí hiệu đơn giản là $\|x\|$ chứ không phải là $\|x\|_2$. Nguyên nhân là do chúng rất hay được sử dụng trong các bài toán, nên phần định danh dưới được bỏ đi cho tiện làm việc. Nếu để ý sẽ thấy, với công thức trên ta có thể dễ dàng tính được L^2 của một véc-tơ bằng phép nhân vô hướng của chúng: $\|x\| = x * x^\intercal$.

Đôi lúc ta cũng sử dụng cả norm cấp 1 để đo khoảng cách giữa các phần tử tại vị trí 0 và vì trị rất gần với 0 thay vì norm cấp 2. Vì norm cấp 2 lấy bình phương từng khoảng cách lên sẽ cho số rất nhỏ do khoảng cách giữa chúng đã rất nhỏ rồi, việc này dẫn tới sự triệt tiêu khoảng cách khi tính toán. Những lúc này ta sẽ sử dụng L^1 để tính toán, vì nó chỉ lấy trị tuyệt đối khoảng cách: $\|x\|_1 = \sum_i |x_i|$.

Cũng có khi ta phải dùng giả norm bậc 0 (L^0) để đo độ dài của véc-tơ khi nó quá bé để có thể thoải mái tính toán với các norm khác. Lưu ý rằng, ta không thực sự có L^0 mà khi nhắc tới nó ta phải hiểu ngầm với nhau rằng chúng là giả norm. L^0 được tính đơn giản bằng cách đếm số lượng các phần tử khác 0 của véc-tơ.

Một dạng norm khác cũng được sử dụng phổ biến trong học máy là L^∞ hay còn được gọi là norm lớn nhất (max norm) được đo bằng cách lấy trị tuyệt đối của phần tử lớn nhất: $\|x\|_\infty = \max_i |x_i|$.

Đó là với cách tính norm cho véc-tơ, thế còn norm áp dụng cho ma trận thì sao? Norm cho ma trận được tính bằng nhiều phương pháp khác nhau, nhưng trong học máy thường ta chỉ dùng tới chuẩn norm Frobenius - tương tự như L^2 cho véc-tơ, như sau:

$$\|A\|_F = \sqrt{\sum_{i,j} A_{ij}^2}$$

Để thực hiện việc tính norm với Numpy, ta có thể sử dụng hàm `norm` trong gói `linalg` (gói đại số tuyến tính) như sau:

```
# for vector
v = np.arange(10)
print(v)
# [0 1 2 3 4 5 6 7 8 9]
print(np.linalg.norm(v))
# 16.881943016134134

# for matrix
A = v.reshape(2, 5)
print(A)
# [[0 1 2 3 4]
#  [5 6 7 8 9]]
print(np.linalg.norm(A))
# 16.881943016134134
```

Chương 8: Visualization trong python

Visualization hiểu một cách đơn giản là hình ảnh hóa dựa trên dữ liệu. Khái niệm của visualization rất ngắn gọn nhưng trên thực tế visualization lại là một mảng rất rộng và có thể coi là một lĩnh vực kết hợp của khoa học và nghệ thuật bởi nó vừa lên quan đến đồ họa (sử dụng hình học để diễn tả kết quả), vừa liên quan đến khoa học thống kê (sử dụng con số để nói lên vấn đề). Nhờ có visualization, chúng ta có thể dễ dàng đưa ra các so sánh trực quan, tính toán tỷ trọng, nhận biết trend, phát hiện outlier, nhận diện đặc điểm phân phối của biến tốt hơn. Từ đó hỗ trợ quá trình nắm thông tin và đưa ra quyết định tốt hơn. Trong các kĩ năng của data scientist thì visualization là một trong những kĩ năng cơ bản và quan trọng nhất. Thế nhưng nhiều data scientist lại chưa nhận

diện được điều này và thường xem nhẹ vai trò của visualization. Trước đây tôi cũng đã từng mắc sai lầm như vậy. Qua kinh nghiệm nhiều năm xây dựng mô hình và phân tích kinh doanh đã giúp tôi nhìn nhận lại vai trò của visualization. Chương này được tổng hợp nội dung từ blog *Phamdinhkhanh.github.io*

Nhắc đến visualization chúng ta không thể không nói đến một số dạng biểu đồ cơ bản như: line, barchart, pie, area, boxplot.

Trong đó:

- **line**: Là biểu đồ đường kết nối các điểm thành 1 đường liền khúc.
- **barchart**: Biểu diễn giá trị của các nhóm dưới dạng cột.
- **pie**: Biểu đồ hình tròn biểu diễn phần trăm của các nhóm.
- **area**: Biểu đồ biểu diễn diện tích của các đường.
- **boxplot**: Biểu đồ biểu diễn các giá trị thống kê của một biến trên đồ thị bao gồm: Trung bình, Max, Min, các ngưỡng percent tile 25%, 50%, 75%.

Sau đây chúng ta sẽ học cách sử dụng các dạng biểu đồ này trên matplotlib.

8.1 Biểu đồ line

Biểu đồ line là biểu đồ biểu diễn các giá trị dưới dạng những đường. Trên `matplotlib`. Line được vẽ thông qua `plt.plot()`. Sau đây ta cùng biểu diễn giá chứng khoán thông qua biểu đồ line.

Lấy dữ liệu chứng khoán của apple

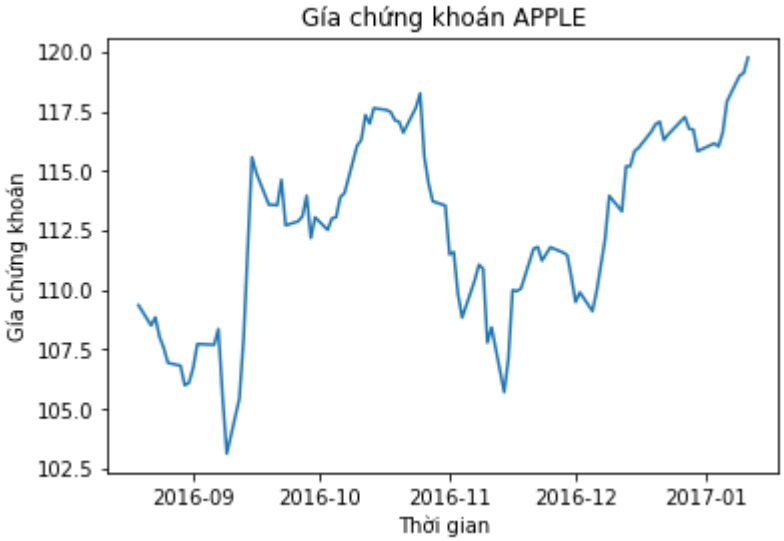
```
import matplotlib.pyplot as plt
import pandas as pd

import datetime
import pandas_datareader.data as web
from pandas import Series, DataFrame


start = datetime.datetime(2010, 1, 1)
end = datetime.datetime(2017, 1, 11)

df = web.DataReader("AAPL", 'yahoo', start, end)
df.tail()
```

	High	Low	Open	Close	Volume	Adj Close
Date						
---	---	---	---	---	---	---
2017-01-05	116.860001	115.809998	115.919998	116.610001	22193600.0	111.727715
2017-01-06	118.160004	116.470001	116.779999	117.910004	31751900.0	112.973305

	High	Low	Open	Close	Volume	Adj Close
2017-01-09	119.430000	117.940002	117.949997	118.989998	33561900.0	114.008080
2017-01-10	119.379997	118.300003	118.769997	119.110001	24462100.0	114.123047
2017-01-11	119.930000	118.599998	118.739998	119.750000	27588600.0	114.736275

Biểu diễn giá chứng khoán dưới dạng biểu đồ line

```
plt.plot(df['Close'].tail(100))
plt.ylabel('Gía chứng khoán')
plt.xlabel('Thời gian')
plt.title('Gía chứng khoán APPLE')
```

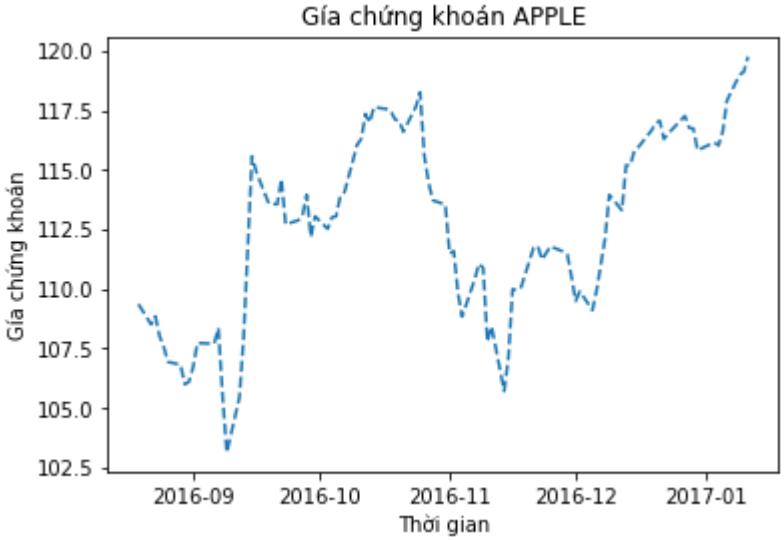


Thay đổi định dạng line

Nếu muốn thay đổi định dạng của line chúng ta sẽ sử dụng thêm 1 tham số khác là linestype. Một số line styles thông dụng: {'-', '–', '-.', ':', ''}

- `-` : Đường nét liền.
- `--` : Đường nét đứt dài.
- `-.` : Đường line nét đứt dài kết hợp với dấu chấm.
- `:` : Đường line gồm các dấu chấm. Chẳng hạn để thay đổi line từ dạng đường nét liền sang nét đứt:

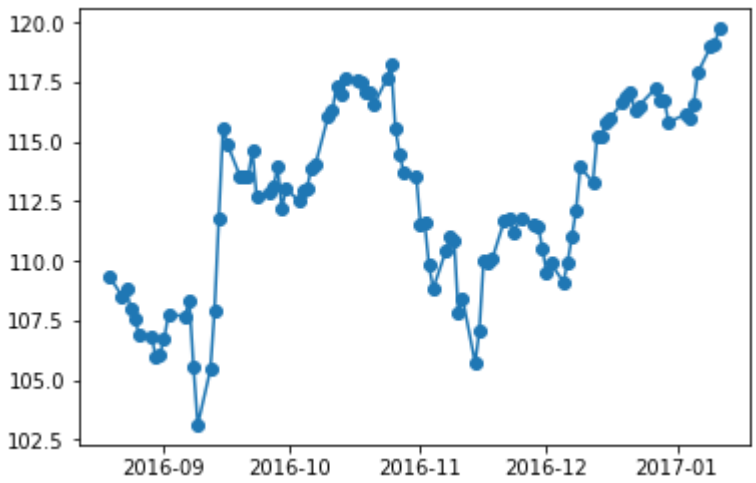
```
plt.plot(df['Close'].tail(100), linestyle = '--')
plt.ylabel('Gía chứng khoán')
plt.xlabel('Thời gian')
plt.title('Gía chứng khoán APPLE')
```



Kết hợp line và point Bên cạnh line chúng ta còn có thể đánh dấu các điểm mút bằng các point. Hình dạng của point có thể là hình tròn, vuông hoặc tam giác và được khai báo thông qua tham số marker. Các giá trị của marker sẽ tương ứng như sau:

- `^` : Hình tam giác
- `o` : Hình tròn.
- `s` : Hình vuông (s tức là square). Bên dưới là một số kết hợp của linstyle và marker.

```
plt.plot(df['Close'].tail(100), linestyle = '-', marker = 'o')
```

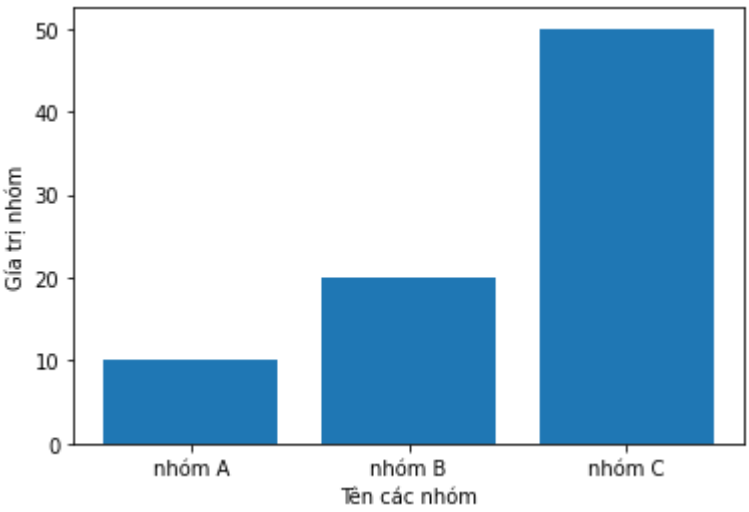


8.2 Biểu đồ barchart

Biểu đồ barchart là dạng biểu đồ có thể coi là phổ biến nhất và được dùng chủ yếu trong trường hợp so sánh giá trị giữa các nhóm thông qua độ dài cột. Để biểu diễn biểu đồ barchart trong python chúng ta sử dụng hàm `plt.bar()`. Các tham số truyền vào bao gồm tên các nhóm (tham số x) và giá trị của các nhóm (tham số height).

```
plt.bar(x = ['nhóm A', 'nhóm B', 'nhóm C'], height = [10, 20, 50])
plt.xlabel('Tên các nhóm')
plt.ylabel('Gía trị nhóm')
```

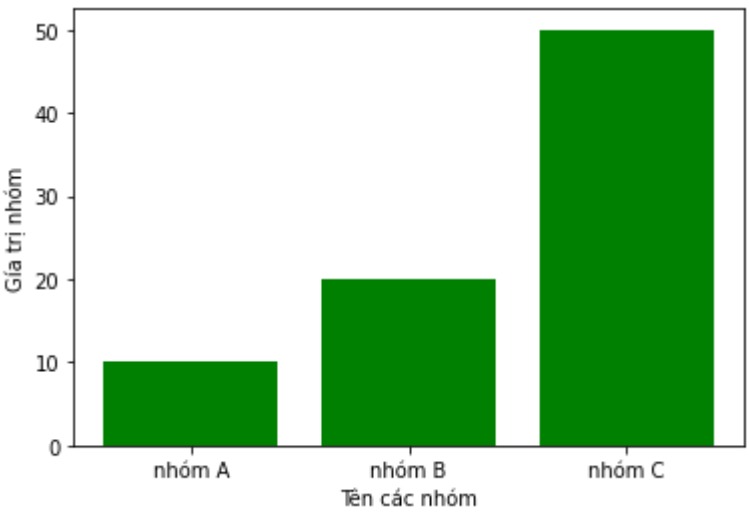
```
Text(0, 0.5, 'Gía trị nhóm')
```



Thay đổi màu sắc các nhóm.

```
plt.bar(x = ['nhóm A', 'nhóm B', 'nhóm C'], height = [10, 20, 50], color =
plt.xlabel('Tên các nhóm')
plt.ylabel('Gía trị nhóm')
```

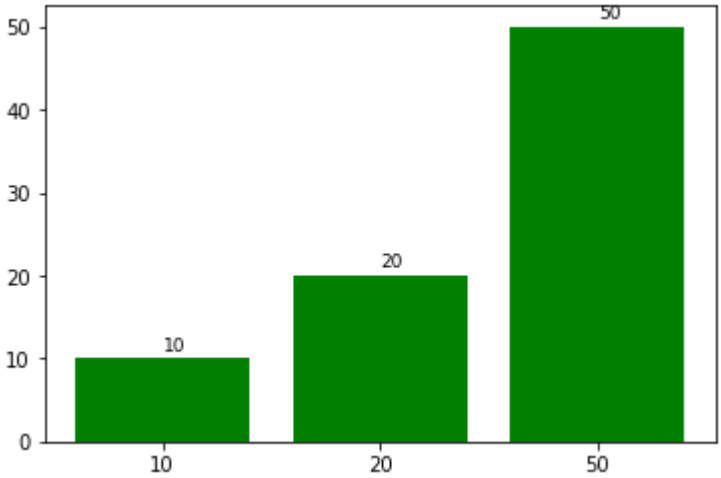
```
Text(0, 0.5, 'Gía trị nhóm')
```



Thêm nhãn giá trị cho các cột bằng tham số `plt.text()`. Trong đó tham số x và y của `plt.text()` qui định tọa độ điểm bắt đầu của rectangle chứa label tên của nhóm. s chứa tên labels của nhóm và size qui định kích thước của text.

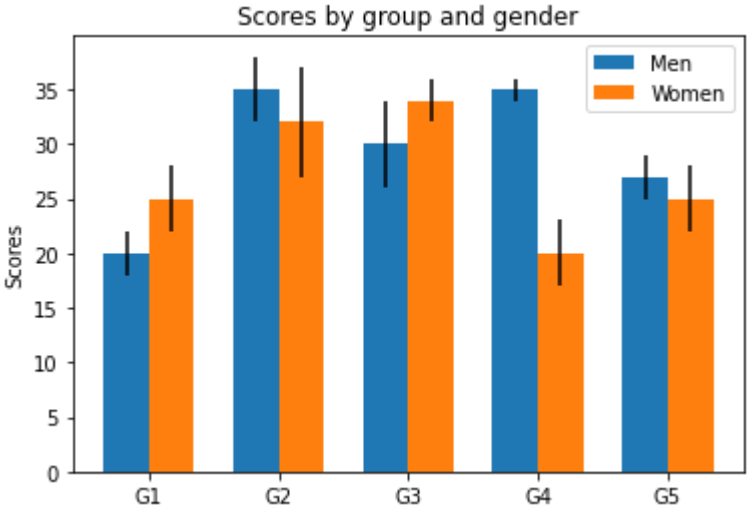
```
x_values = [0, 1, 2]
y_values = [10, 20, 50]
data_labels = ['10', '20', '50']
plt.bar(x = data_labels, height = y_values, color = 'green')
```

```
for i in range(len(data_labels)): # your number of bars
  plt.text(x = x_values[i], #takes your x values as horizontal positioning
  y = y_values[i]+1, #takes your y values as vertical positioning argument
  s = data_labels[i], # the labels you want to add to the data
  size = 9)
```



Chúng ta cũng có thể vẽ biểu đồ của 2 biến trở lên là các barchart liền kề nhau.

```
import numpy as np

men_means, men_std = (20, 35, 30, 35, 27), (2, 3, 4, 1, 2)
women_means, women_std = (25, 32, 34, 20, 25), (3, 5, 2, 3, 3)

ind = np.arange(len(men_means))  # the x locations for the groups
width = 0.35  # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(ind - width/2, men_means, width, yerr=men_std,
                label='Men')
rects2 = ax.bar(ind + width/2, women_means, width, yerr=women_std,
                label='Women')

# Add some text for labels, title and custom x-axis tick labels, etc.
ax.set_ylabel('Scores')
ax.set_title('Scores by group and gender')
ax.set_xticks(ind)
ax.set_xticklabels(('G1', 'G2', 'G3', 'G4', 'G5'))
ax.legend()

plt.show()
```



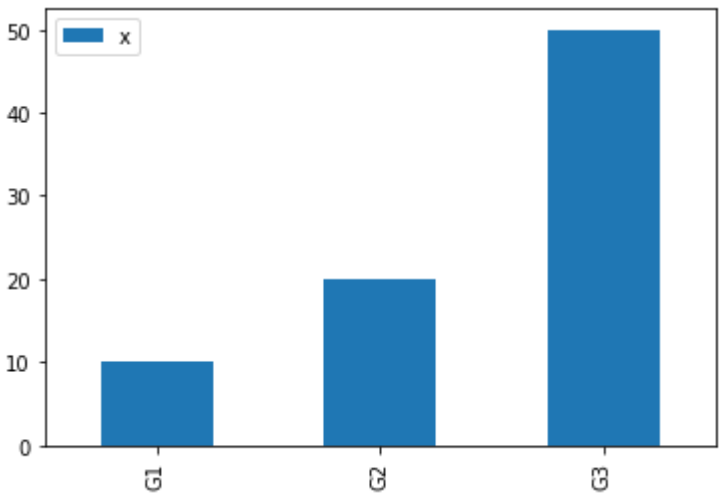
Ta cũng có thể biểu diễn biểu đồ thông qua dataframe.

```
df = pd.DataFrame({'x': [10, 20, 50]}, index = ['G1', 'G2', 'G3'])
df
```

	x
G1	10
G2	20
G3	50

```
df.plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5fdbb3b110>
```



8.3. Biểu đồ tròn

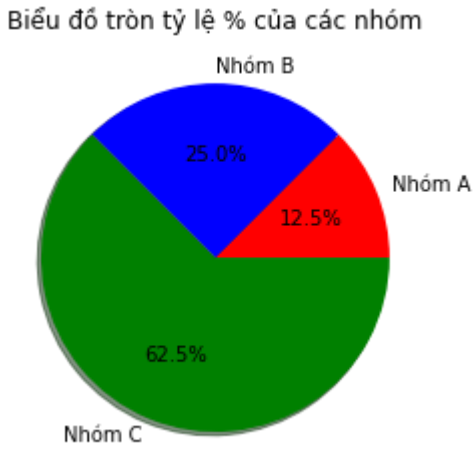
Biểu đồ tròn được sử dụng để visualize tỷ lệ phần trăm các class. Ưu điểm của biểu đồ này là dễ dàng hình dung được giá trị % mà các class này đóng góp vào số tổng. Nhưng

nhược điểm là không thể hiện số tuyệt đối.

Để tạo biểu đồ tròn trong mathplotlib.

```
import numpy as np
plt.pie(x = np.array([10, 20, 50]), # giá trị của các nhóm
        labels = ['Nhóm A', 'Nhóm B', 'Nhóm C'], # Nhãn của các nhóm
        colors = ['red', 'blue', 'green'], # Màu sắc của các nhóm
        autopct = '%1.1f%%', # Format hiển thị giá trị %
        shadow = True
       )
plt.title('Biểu đồ tròn tỷ lệ % của các nhóm')
```

```
Text(0.5, 1.0, 'Biểu đồ tròn tỷ lệ % của các nhóm')
```

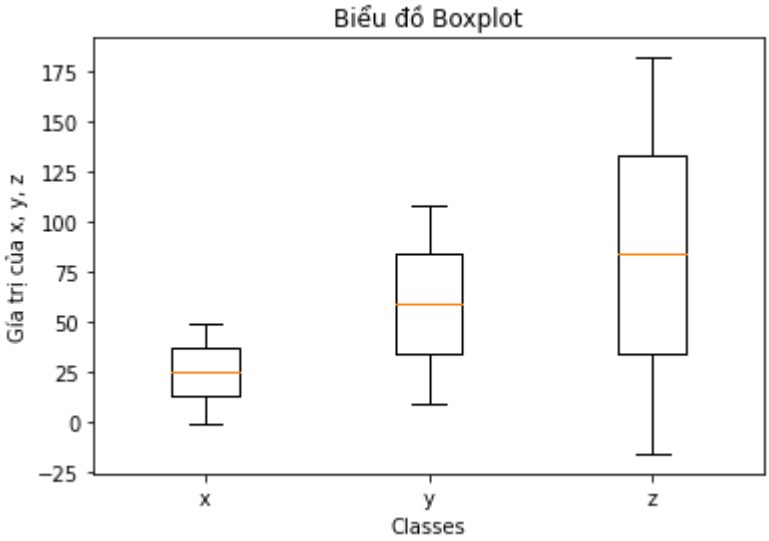


8.4 Biểu đồ boxplot

Biểu đồ boxplot sẽ cho ta biết đặc trưng về phân phối của 1 biến dựa trên các giá trị trung bình, min, max, các khoảng phân vị 25%, 50%, 75%. Đây là biểu đồ được sử dụng nhiều trong chứng khoán và thống kê học để so sánh các biến với nhau.

```
import numpy as np
x = np.random.randn(100) + np.arange(0, 100) * 0.5
y = np.random.randn(100) + np.arange(0, 100) * 1.0 + 10
z = np.random.randn(100) + np.arange(0, 100) * 2 - 15

plt.boxplot([x, y, z],
            labels = ['x', 'y', 'z'],
            showfliers = True)

plt.title('Biểu đồ Boxplot')
plt.xlabel('Classes')
plt.ylabel('Gía trị của x, y, z')
```

```
Text(0, 0.5, 'Gía trị của x, y, z')
```



8.5 Vẽ biểu đồ trên dataframe

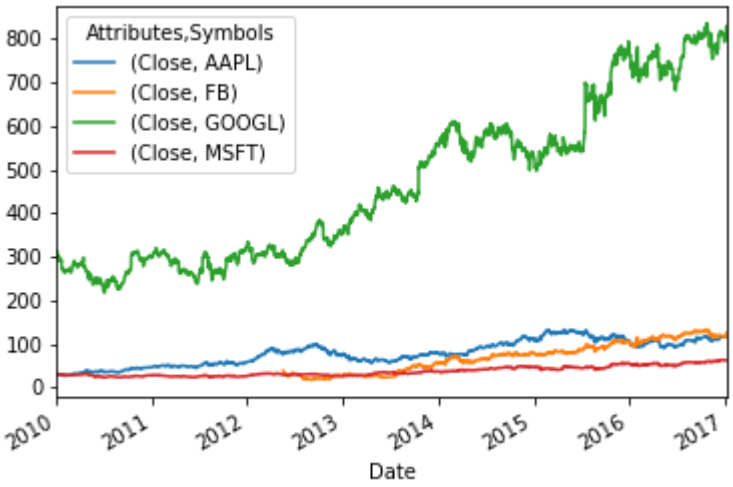
Định dạng dataframe của pandas không chỉ hỗ trợ các truy vấn và thống kê dữ liệu có cấu trúc nhanh hơn mà còn support vẽ biểu đồ dưới dạng matplotlib-based. Sau đây chúng ta cùng sử dụng dataframe để vẽ các đồ thị cơ bản.

```
import pandas as pd
import datetime
import pandas_datareader.data as web
from pandas import Series, DataFrame


start = datetime.datetime(2017, 1, 1)
end = datetime.datetime(2020, 1, 11)

df = web.DataReader(["AAPL", "GOOGL", "MSFT", "FB"], 'yahoo',
start, end)
# Chỉ lấy giá close
df = df[['Close']]
df.tail()
```

Attributes	Close			
Symbols	AAPL	FB	GOOGL	MSFT
---	---	---	---	---
Date				
---	---	---	---	---
2017-01-05	116.610001	120.669998	813.020020	62.299999
2017-01-06	117.910004	123.410004	825.210022	62.840000
2017-01-09	118.989998	124.900002	827.179993	62.639999
2017-01-10	119.110001	124.349998	826.010010	62.619999

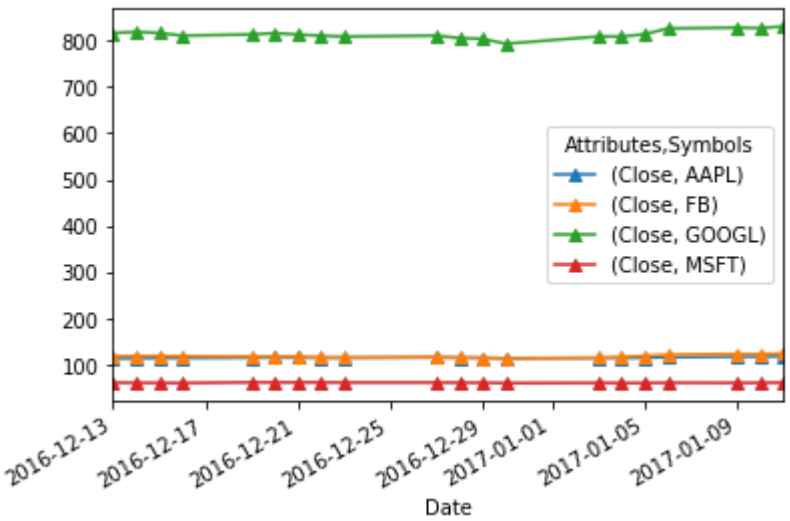
Attributes	Close			
2017-01-11	119.750000	126.089996	829.859985	63.189999

```
df.plot()
```



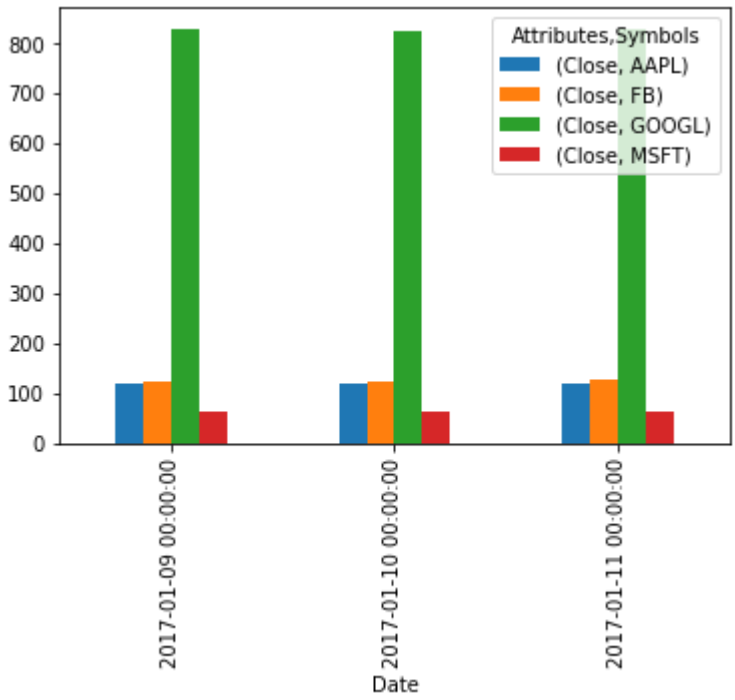
Biểu đồ line kết hợp point

```
df.tail(20).plot(linestyle = '-', marker = '^')
```



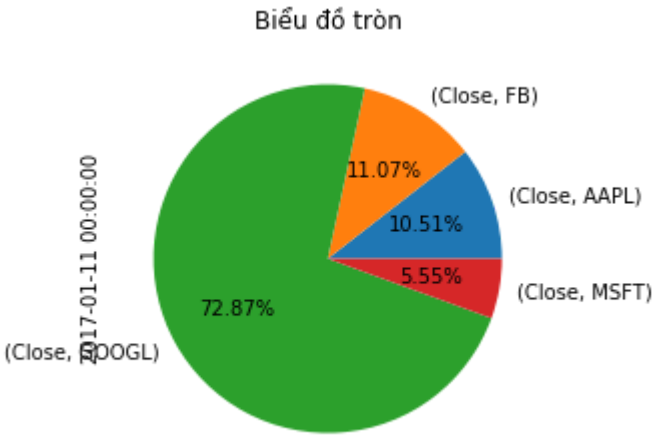
Biểu đồ barchart

```
df.tail(3).plot.bar()
```



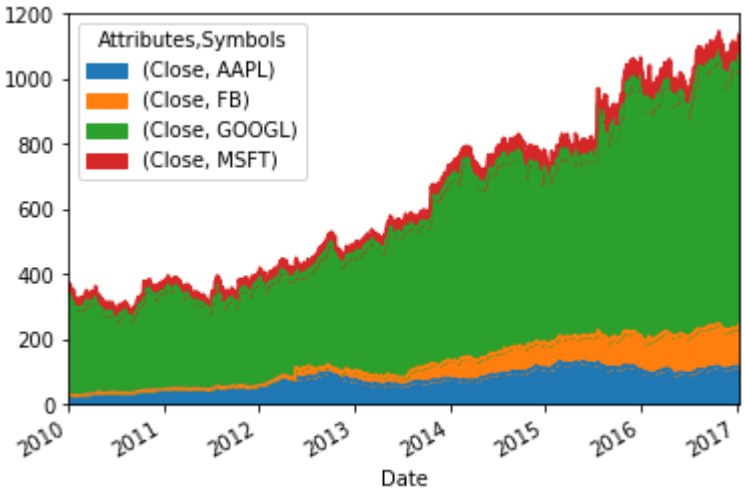
Biểu đồ tròn

```
df.iloc[-1, :].plot.pie(autopct = '%.2f%%')
plt.title('Biểu đồ tròn')
```



Biểu đồ diện tích

```
df.plot.area()
```



Vùng có diện tích càng lớn thì khoảng chênh lệch về giá theo thời gian của nó càng lớn và các vùng có diện tích nhỏ hơn cho thấy các mã chứng khoán ít có sự chênh lệch về giá theo thời gian.

8.6 Các biểu đồ biểu diễn phân phối.

8.6.1 Density plot

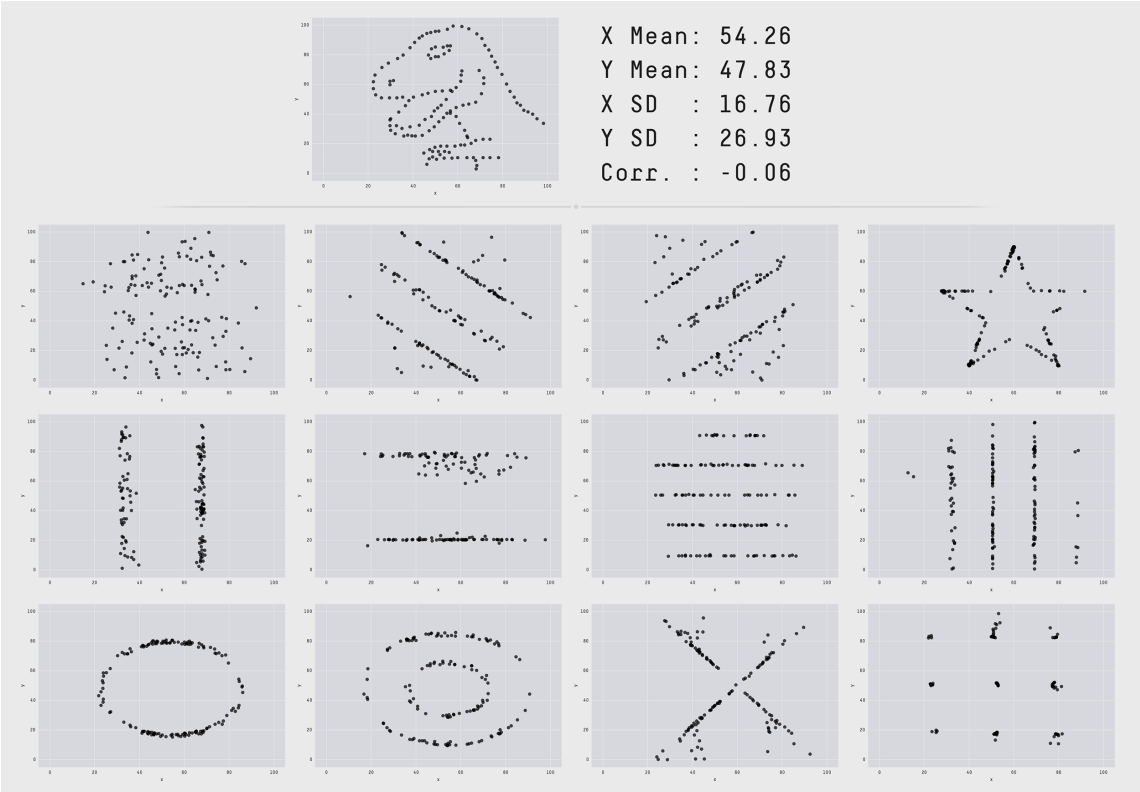
Mỗi một bộ dữ liệu đều có một đặt trưng riêng của nó. Để mô hình hóa những đặc trưng này, thống kê học sử dụng thống kê mô tả như tính mean, max, median, standard deviation, percentile. Để tính thống kê mô tả cho một dataset dạng pandas dataframe trong python đơn giản ta sử dụng hàm describe().

```
from sklearn import datasets
iris = datasets.load_iris()

X = iris.data
y = iris.target

import pandas as pd
dataset = pd.DataFrame(data = X, columns = iris['feature_names'])
dataset['species'] = y
print('dataset.shape: ', dataset.shape)

dataset.describe()
```

```
dataset.shape:  (150, 5)
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333	1.000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

Tuy nhiên không phải lúc nào thống kê mô tả là duy nhất đối với một bộ dữ liệu. Một ví dụ tiêu biểu về phân phối hình `chú khủng long`.

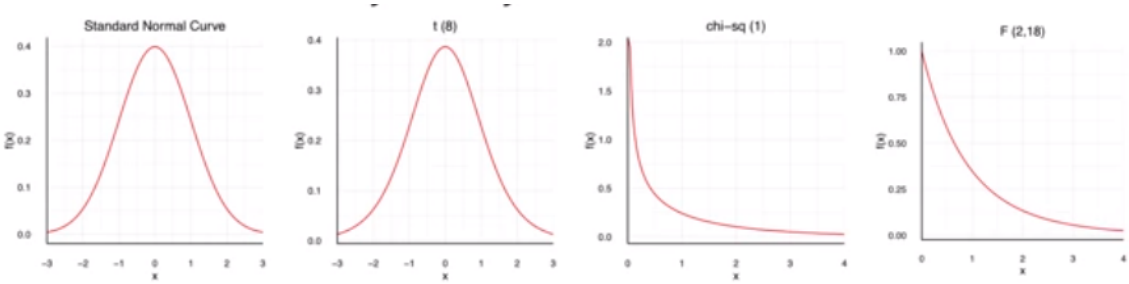


> Hình 1: Đồ thị hình chú khủng long và các hình bên dưới có hình dạng hoàn toàn khác biệt nhau nhưng đều dựa trên 2 chuỗi X,Y có chung thống kê mô tả mean, phương sai và hệ số tương quan.

Do đó không nên hoàn toàn tin tưởng vào thống kê mô tả mà bên cạnh đó chúng ta cần visualize phân phối của dữ liệu.

Trong thống kê mỗi một bộ dữ liệu đều được đặc trưng bởi một hàm mật độ xác suất (pdf - probability density function). Các phân phối điển hình như standard normal, T-

student, poisson, fisher, chi-squared đều được đặc trưng bởi những hình dạng đồ thị phân phối của hàm mật độ xác suất khác nhau.



> Đồ thị hàm mật độ xác suất của những phân phối xác suất `standard normal, T-student, poisson, fisher, chi-squared`

Về mặt lý thuyết (theoreotical) những phân phối này đều dựa trên những phương trình xác định.

Trong thực nghiệm (empirical) nhiều bộ dữ liệu cho thấy có hình dạng tương đồng với những phân phối này.

Để tìm ra một hình dạng tương đối cho hàm mật độ xác suất của một bộ dữ liệu chúng ta sẽ sử dụng phương pháp KDE (kernel density estimate)

KDE là gì?

Hãy tưởng tượng tại mỗi một quan sát ta có đường cong phân phối đặc trưng. Hàm kernel sẽ giúp xác định hình dạng của đường cong trong khi độ rộng của đường cong được xác định bởi bandwidth - h. Phương pháp KDE sẽ tính tổng của các đường cong chạy dọc theo trục x để hình thành nên đường cong mật độ xác suất tổng quát cho dữ liệu.

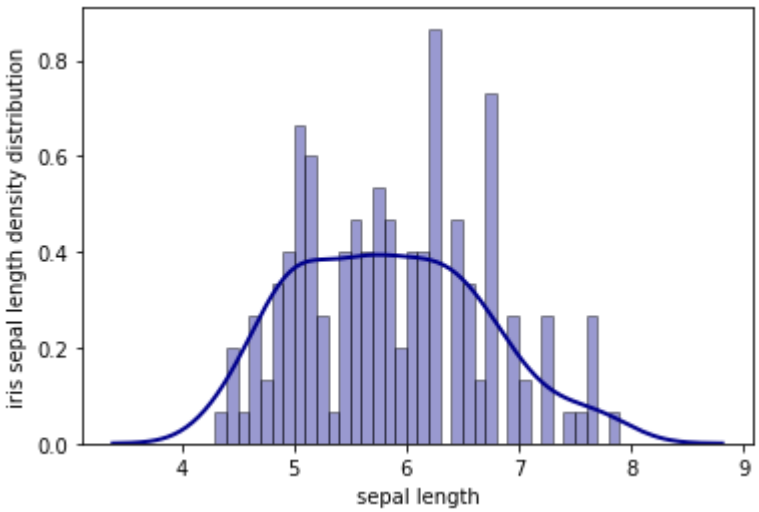
Ngoài ra hình dạng `bandwidth – h` sẽ giúp xác định mức độ khái quát hoặc chi tiết của đường cong. Nếu ta muốn đường cong smoothing hơn thì cần thiết lập h lớn hơn và đường cong mấp mô hơn thì h cần nhỏ hơn. Tuy nhiên bạn đọc cũng không cần quá quan tâm đến bandwidth vì cách tốt hơn là sử dụng giá trị mặc định được tính trong matplotlib.

Bên dưới ta sẽ thực hành vẽ hàm mật độ xác suất của độ dài các đài hoa thông qua hàm `distplot()` của package `seaborn`.

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.distplot(dataset['sepal length (cm)'],
             hist = True,
             bins=int(180/5),
             kde = True,
```

```
            color = 'darkblue',
            hist_kws={'edgecolor':'black'},
            kde_kws={'linewidth':2})
# Khai báo tiêu đề cho trục x
plt.xlabel('sepal length')
# Khai báo tiêu đề cho trục y
plt.ylabel('iris sepal length density distribution')
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: Futur
eWarning: `distplot` is a deprecated function and will be removed in a futu
re version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function fo
r histograms).
  warnings.warn(msg, FutureWarning)
```



Tham số quan trọng nhất của hàm số là kde = True để xác nhận chúng ta sử dụng phương pháp KDE để tính toán đường cong hàm mật độ. Các tham số khác như color, hist_kws, kde_kws chỉ là những tham số râu ria qui định màu sắc, format, kích thước. Ngoài ra hist = True để thiết lập đồ thị histogram mà chúng ta sẽ tìm hiểu bên dưới.

8.6.2 Histogram plot

Histogram là biểu đồ áp dụng trên một biến liên tục nhằm tìm ra phân phối tần suất trong những khoảng giá trị được xác định trước của một biến.

Có 2 cách tạo biểu đồ histogram theo các khoảng giá trị đó là:

- Phân chia các khoảng giá trị có độ dài bằng nhau và độ dài được tính toán từ số lượng bins khai báo.
- Tự định nghĩa các khoảng giá trị dựa trên bins_edge là các đầu mút của khoảng.

Biểu đồ histogram có thể được visualize qua package mathplotlib. Các biểu đồ của mathplotlib được thể được setup dưới nhiều style đồ họa khác nhau (thay đổi về theme, kiểu chữ, ... nhưng về bản chất vẫn là các đối tượng của mathplotlib). Trong đó seaborn, một matplotlib-based package xuất sắc được phát triển bởi Michael Waskom là một

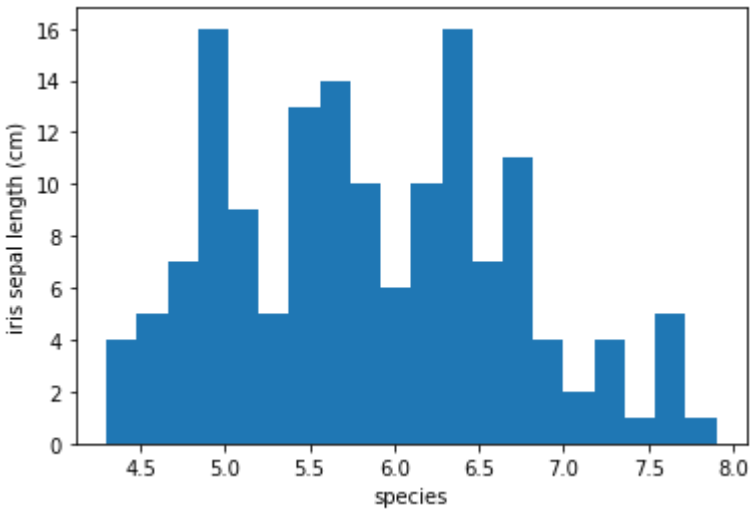
trong những style được ưa chuộng nhất. Trong phần này chúng ta sẽ setup style của đồ thị dưới dạng seaborn.

Bên dưới là biểu đồ histogram của độ rộng đài hoa visualize theo 2 cách: Khai báo bins và khai báo bins edge.

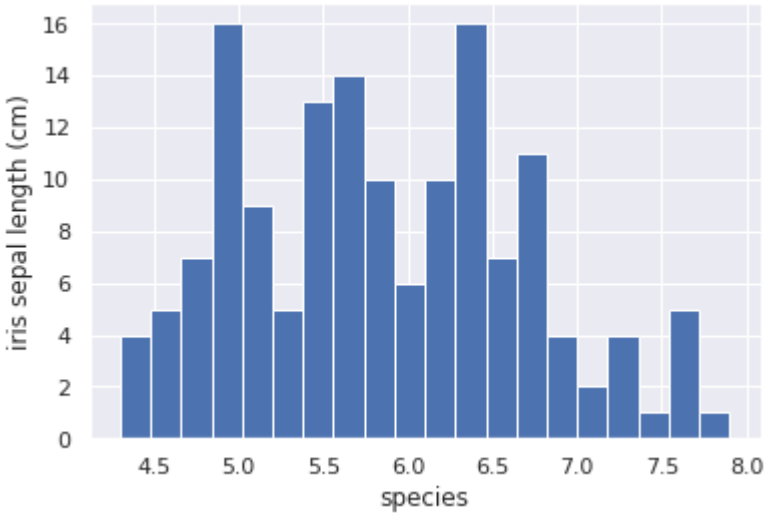
Đồ thị histogram theo số lượng bins = 20

Nếu không set style hiển thị mặc định là seaborn đồ thị sẽ là:

```
import matplotlib.pyplot as plt

plt.hist(dataset['sepal length (cm)'], bins = 20)
# Khai báo tiêu đề cho trục x
plt.xlabel('species')
# Khai báo tiêu đề cho trục y
plt.ylabel('iris sepal length (cm)')
plt.show()
```




```
import matplotlib.pyplot as plt
import seaborn as sns

# Setup style của matplotlib dưới dạng seaborn

sns.set()
plt.hist(dataset['sepal length (cm)'], bins = 20)
# Khai báo tiêu đề cho trục x
plt.xlabel('species')
# Khai báo tiêu đề cho trục y
plt.ylabel('iris sepal length (cm)')
plt.show()
```

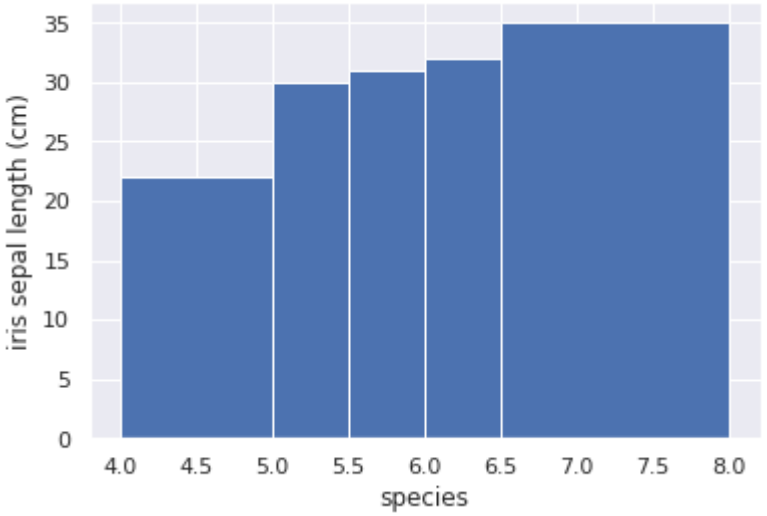


Ta thấy hình của đồ thị được chuyển sang màu xám nhạt và giữa các cột histogram có viền trắng phân chia nhìn rõ ràng hơn. Đây là những thay đổi về đồ họa rất nhỏ nhưng giúp đồ thị trở nên đẹp mắt hơn so với mặc định của mathplotlib.

Đồ thị histogram theo bin edges

Các bin edges được khai báo thông qua cũng cùng tham số bins, giá trị được truyền vào khi đó là 1 list các điểm đầu mút. Từ đó giúp đồ thị linh hoạt hơn khi có thể hiệu chỉnh độ dài các bins tùy thích.

```
import matplotlib.pyplot as plt
import seaborn as sns

bin_edges = [4, 5, 5.5, 6, 6.5, 8]
plt.hist(dataset['sepal length (cm)'], bins = bin_edges)
# Khai báo tiêu đề cho trục x
plt.xlabel('species')
# Khai báo tiêu đề cho trục y
plt.ylabel('iris sepal length (cm)')
plt.show()
```



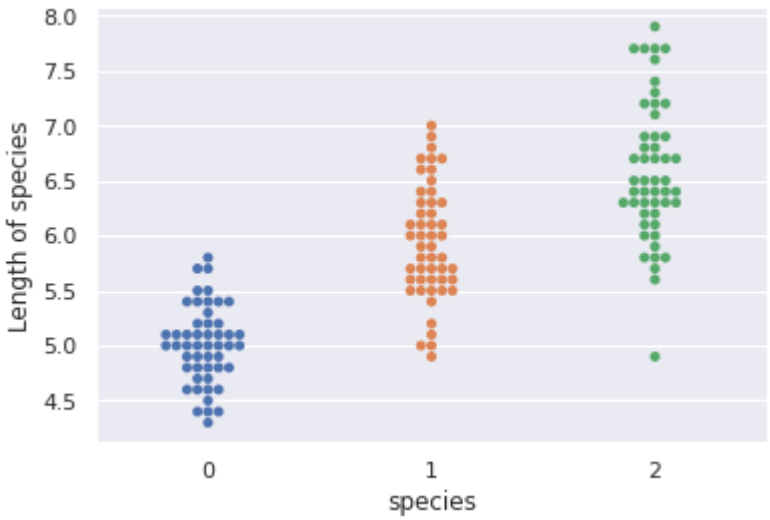
Ta thấy nhược điểm của histogram đó là đồ thị sẽ bị thay đổi tùy theo số lượng bins được thiết lập hoặc list các đầu mút range được khai báo. Do đó để nhận biết được hình dạng phân phối của dữ liệu, một biểu đồ khác thường được sử dụng thay thế đó chính là swarn plot.

8.2.6 Swarn plot

Swarn plot là biểu đồ point biểu diễn các giá trị dưới dạng các điểm. Các giá trị trên đồ thị bằng đúng với giá trị thật của quan sát. Do đó không xảy ra mất mát thông tin như histogram. Thông qua swarn plot ta có thể so sánh được phân phối của các class khác nhau trên cùng một đồ thị.

Hãy hình dung qua ví dụ cụ thể khi visualization dữ liệu iris theo chiều dài, rộng cánh hoa và đài hoa.

```
import seaborn as sn
import matplotlib.pyplot as plt

sn.swarmplot(x = 'species', y = 'sepal length (cm)', data = dataset)
plt.xlabel('species')
plt.ylabel('Length of species')
plt.show()
```



Muốn thay nhãn của các x = [0, 1, 2] sang target_names = [‘setosa’, ‘versicolor’, ‘virginica’] ta sử dụng hàm plt.xticks().

```
import seaborn as sn
import matplotlib.pyplot as plt

sn.swarmplot(x = 'species', y = 'sepal length (cm)', data = dataset)
plt.xlabel('species')
# Thêm plt.xticks() để thay nhãn của x
plt.xticks(ticks = [0, 1, 2], labels = ['setosa', 'versicolor', 'virginica
plt.ylabel('Sepal length of species')
plt.show()
```

Từ biểu đồ ta nhận thấy độ dài đài hoa có sự khác biệt ở cả 3 giống hoa iris. Trung bình độ dài của đài hoa tăng dần từ setosa, versicolor đến virginica. Vì swarm là đồ thị giữ nguyên giá trị thực của trục y nên các điểm outliers được thể hiện đúng với thực tế trên từng class. Thông tin thể hiện trên biểu đồ swarm dường như là không có sự mất mát so với biểu đồ bins hoặc density.

Chương 9: SciPy

Scipy (đọc là /'saɪpaɪ'/ "Sigh Pie") là phần mềm nguồn mở cho toán học, khoa học và kỹ thuật. Thư viện SciPy được xây dựng dựa trên thư viện NumPy, cung cấp thao tác mảng N chiều thuận tiện và nhanh chóng. SciPy gồm các gói con (submodule) cho đại số tuyến tính, tối ưu hóa, tích hợp và thống kê.

NumPy và SciPy rất dễ sử dụng, mạnh mẽ và được nhiều nhà khoa học và kỹ sư hàng đầu thế giới lựa chọn:

- SciPy chứa nhiều loại gói phụ giúp giải quyết vấn đề phổ biến nhất liên quan đến tính toán khoa học.
- SciPy là thư viện Khoa học được sử dụng nhiều nhất chỉ sau Thư viện Khoa học GNU cho C/C ++ hoặc Matlab.
- Dễ sử dụng và hiểu cũng như sức mạnh tính toán nhanh.
- Nó có thể hoạt động trên mảng (array) của thư viện NumPy.

9.1 Cài đặt thư viện

Bạn có thể cài đặt thông qua các phân phối: Anaconda, Miniconda, WinPython, Pyzo.

Sử dụng pip:

```
python-m pip install --user scipy
```

Bạn có thể cài đặt cùng lúc nhiều thư viện với pip:

```
python-m pip install --user numpy scipy matplotlib ipython
jupyter pandas sympy nos
```

> Lưu ý: Numpy(chương 7) phải được cài đặt trước. Bạn cũng nên cài đặt Matplotlib(chương 8) khi sử dụng Scipy

9.2 Các hàm cơ bản

Scipy được xây dựng trên Numpy nên bạn có thể sử dụng các hàm của Numpy để thực hiện tất cả các thao tác mảng cơ bản. Chi tiết về các hàm này các bạn có thể đọc chương 7 hoặc sử dụng các lệnh help, info và source. Bên dưới là ví dụ về một số thao tác mảng cơ bản.

```
>>> # khai báo numpy
>>> import numpy as np
>>> # Hàm mrid()
>>> np.mgrid[0:5,0:4]
array([[[0, 0, 0, 0],
        [1, 1, 1, 1],
        [2, 2, 2, 2],
        [3, 3, 3, 3],
        [4, 4, 4, 4]],

       [[0, 1, 2, 3],
        [0, 1, 2, 3],
        [0, 1, 2, 3],
        [0, 1, 2, 3],
        [0, 1, 2, 3]]])
>>> # Hàm mrid() số ảo
>>> np.mgrid[0:5:4j,0:4:3j]
array([[[0.        , 0.        , 0.        ],
        [1.66666667, 1.66666667, 1.66666667],
        [3.33333333, 3.33333333, 3.33333333],
        [5.        , 5.        , 5.        ]],

       [[0.        , 2.        , 4.        ],
        [0.        , 2.        , 4.        ],
        [0.        , 2.        , 4.        ],
        [0.        , 2.        , 4.        ]]])
>>> # Hàm đa thức
>>> from numpy import poly1d
>>> p = poly1d([2,6])
>>> print(p)
```

```
2 x + 6
>>> # Bình phương hàm p
>>> print(p*p)
   2
4 x + 24 x + 36
>>> # Đạo hàm của p
>>> print(p.deriv())

2
>>> # Giá trị p tại x = 1, 2, 10
>>> p([1,2,10])
array([ 8, 10, 26])
>>> # Hàm select() để lấy ra array theo điều kiện
>>> x = np.arange(15)
>>> dieu_kien = [x< 10, x>=5]
>>> chon_ra = [x, x**3]
>>> np.select(dieu_kien, chon_ra)
array([ 0,  1,  8, 27, 64,  5,  6,  7,  8,  9, 10, 11, 12, 13,
14])
>>> dieu_kien = [x>12, x<3]
>>> np.select(dieu_kien, chon_ra)
array([ 0,  1,  8,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 13,
14])
>>>
```

9.3 Tổng quan về các gói con của Scipy

Việc sử dụng thư viện SciPy yêu cầu một số thư viện khác để hoạt động, phụ thuộc chính là NumPy. Nó đòi hỏi một các thư viện và công cụ để xây dựng thư viện hoặc xây dựng tài liệu. Các công cụ và thư viện luôn thay đổi. SciPy cần tương thích với một số phiên bản phát hành của các thư viện và công cụ phụ thuộc. Do đó người dùng cần chú ý sự tương thích giữa phiên bản của các thư viện khác có liên quan với phiên bản của Scipy.

Scipy có nhiều gói con và liên tục được phát triển, bổ sung tính năng mới. Bảng tổng hợp các gói con của Scipy theo thứ tự alphabet:

gói con	Miêu tả
`cluster`	Thuật toán phân cụm (Clustering Algorithms)
`constants`	Các hằng số toán học và vật lý
`fftpack`	Hàm biến đổi Fourier nhanh (Fast Fourier Transform)
`integrate`	Giải phương trình vi phân và tích phân
`interpolate`	Nội suy và làm mịn spline
`io`	Đầu vào và đầu ra
`linalg`	Đại số tuyến tính

gói con	Miêu tả
`ndimage`	Xử lý ảnh N chiều
`odr`	Hồi quy khoảng cách trực giao
`optimize`	Tối ưu hóa và chương trình root-finding
`signal`	Xử lý tín hiệu
`sparse`	Ma trận sparse và các đoạn chương trình liên quan
`spatial`	Các cấu trúc dữ liệu không gian và thuật toán
`special`	Các hàm toán học đặc biệt
`stats`	Các hàm và phân phối thống kê

9.4 Tính tích phân

Python dùng trong khoa học kỹ thuật cung cấp nhiều phương pháp để tính tích phân. Mục đích chung của các phương pháp dùng tính tích phân I có dạng như sau: $I = \integrate$ Scipy cung cấp hàm quad() trong module `scipy.integrate` dùng để tính tích phân. Hàm này nhận tham số đầu vào là hàm `f(x)` dùng để tính tích phân và cận trên (a), cận dưới (b). Hàm này trả về 2 giá trị trong tuple: giá trị thứ nhất tính kết quả giá trị tích phân và giá trị thứ 2 là ước lượng giá trị độ lỗi khi tính tích phân theo phương pháp gần đúng.

Sau đây là một ví dụ:

```
import math
from math import cos , exp , pi
from scipy.integrate import quad
#Hàm dùng để tính tích phân.
def f(x):
  return exp(math.cos(-2 * x * pi)) + 3.2

#Gọi hàm quad để tính tích phân hàm f với cận từ -2 đến 2
res, err = quad(f, -2, 2)
print("Kết quả tính tích phân là {:f} (+-{:g})" .format(res, err))
```

```
Kết quả tính tích phân là 17.864264 (+-1.55117e-11)
```

> Hàm quad() nhận thêm tham số tuỳ chọn là `epsabs` và `epsrel` để tăng hoặc giảm độ chính xác của kết quả tính được. Giá trị mặc định của tham số là `epsabs=1.5e-8` và `epsrel=1.5e-8`

9.5 Giải phương trình vi phân

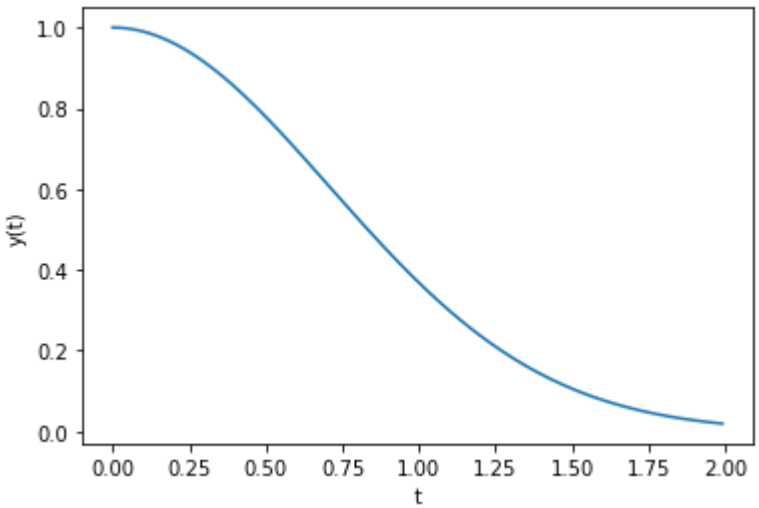
Để giải một phương trình vi phân có dạng:

$$\frac{dy}{dt}(t) = f(y,t)$$

Với giá trị $y(t_0) = y_0$ chúng ta có thể sử dụng hàm `odeint` trong scipy. Sau đây là ví dụ (kèm giải thích trong code) với đề bài tìm $y(t)$ với $t \in [0,2]$ với phương trình vi phân:

$$\frac{dy}{dt}(t) = -2yt \quad với \quad y(0) = 1$$

```
from scipy.integrate import odeint
import numpy as N

def f(y, t):
  """this is the rhs of the ODE to integrate, i.e. dy/dt=f(y,t)"""
  return -2 * y * t
y0 = 1 # Giá trị khởi tạo
a = 0 # Giới hạn của t
b = 2
t = N.arange(a, b, 0.01) # Chọn các giá trị t để tính y(t)

y = odeint(f, y0, t) # Tính giá trị y(t)
import pylab # Vẽ kết quả
pylab.plot(t, y)
pylab.xlabel('t'); pylab.ylabel('y(t)')
pylab.show()
```



> Hàm `odeint` nhận các tham số tuỳ chọn để thay đổi độ lỗi. Các bạn có thể xem thêm tại:
>
> `>> help(scipy.integrate.odeint)`

9.6 Tìm nghiệm phương trình

Tìm nghiệm của phương trình là tìm các giá trị `x` sao cho f(x) = 0. Có nhiều phương pháp để tìm nghiệm trong module `optimize` của scipy.

9.6.1 Giải phương trình bằng phương pháp chia đôi - BISECTION

Ý tưởng Thu hẹp dần khoảng phân li nghiệm bằng cách chia đôi

Cho phương trình $f(x) = 0$, $f(x)$ liên tục và trái dấu tại 2 đầu $[a, b]$. Giả sử $f(a) < 0, f(b) < 0$ (nếu ngược lại thì xét $-f(x) = 0$). Theo định lý 1, trên $[a, b]$ phương trình có ít nhất 1 nghiệm μ. Để tìm nghiệm gần đúng c, ta thực hiện một số hữu hạn lần quá trình lặp các bước sau đây:

- Bước 1: Ta chọn c là điểm chính giữa của đoạn $[a, b]$, $c = (a + b)/2$
- Bước 2: Nếu $f(c) = 0$ thì ta khẳng định ngay c là nghiệm cần tìm và chuyển sang bước 4, ngược lại chuyển sang bước 3.
- Bước 3: Nếu $f(a)f(c) < 0$ thì ta đặt lại b = c và quay về bước 1. Còn nếu $f(a)f(c) > 0$ thì đặt lại $a = c$ rồi cũng quay về bước 1.
- Bước 4: Thông báo nghiệm c tìm được và kết thúc công việc tìm nghiệm của phương trình $f(x) = 0$.

Quá trình trên gọi là phương pháp chia đôi bởi vì cứ mỗi một lần lặp lại từ đầu thì khoảng $[a, b]$ cần xem xét được thu gọn lại chỉ còn một nửa so với lần trước bởi điểm chính giữa c. Quá trình lặp trên cũng dừng lại khi đoạn $[a, b]$ quá ngắn (nhỏ hơn một số dương rất nhỏ nào đó, gọi là sai số).

Giả sử chúng cần tìm nghiệm của phương trình $f(x) = x^3 - 2x^2$. Hàm này có 2 nghiệm tại $x = 0$ và một nghiệm khác nằm giữa $x = 1.5$ (ta có $f(1.5) = -1.125$) và $x = 3$ (ta có $f(3) = 9$. Có thể dễ dàng đoán được nghiệm còn lại nằm tại $x = 2$.

Sau đây là code thực hiện tìm nghiệm theo phương pháp chia đôi:

```
from scipy.optimize import fsolve
def f(x):
  return x ** 3 - 2 * x ** 2
x = fsolve(f, 3) # one root is at x=2.0
print("Nghiệm xấp xỉ x=%21.19g" % x)
print("Độ lỗi %g." % (2 - x))
```

```
Nghiệm xấp xỉ x= 2.000000000000006661
Độ lỗi -6.66134e-15.
```

9.6.2 Tìm nghiệm bằng hàm fsolve

Hàm `fsolve` tìm nghiệm hiệu quả hơn bisection được sử dụng phổ biến hơn. Hàm này thường được dùng để tìm nghiệm cho phương trình nhiều biến. Ví dụ về việc tìm nghiệm:

```
from scipy.optimize import fsolve
def f(x):
  return x ** 3 - 2 * x ** 2
x = fsolve(f, 3) # one root is at x=2.0
```

```
print("Nghiệm xấp xỉ của phương trình x=%21.19g" % x )
print("Độ lỗi của nghiệm %g." % (2 - x))
```

```
Nghiệm xấp xỉ của phương trình x= 2.000000000000006661
Độ lỗi của nghiệm -6.66134e-15.
```

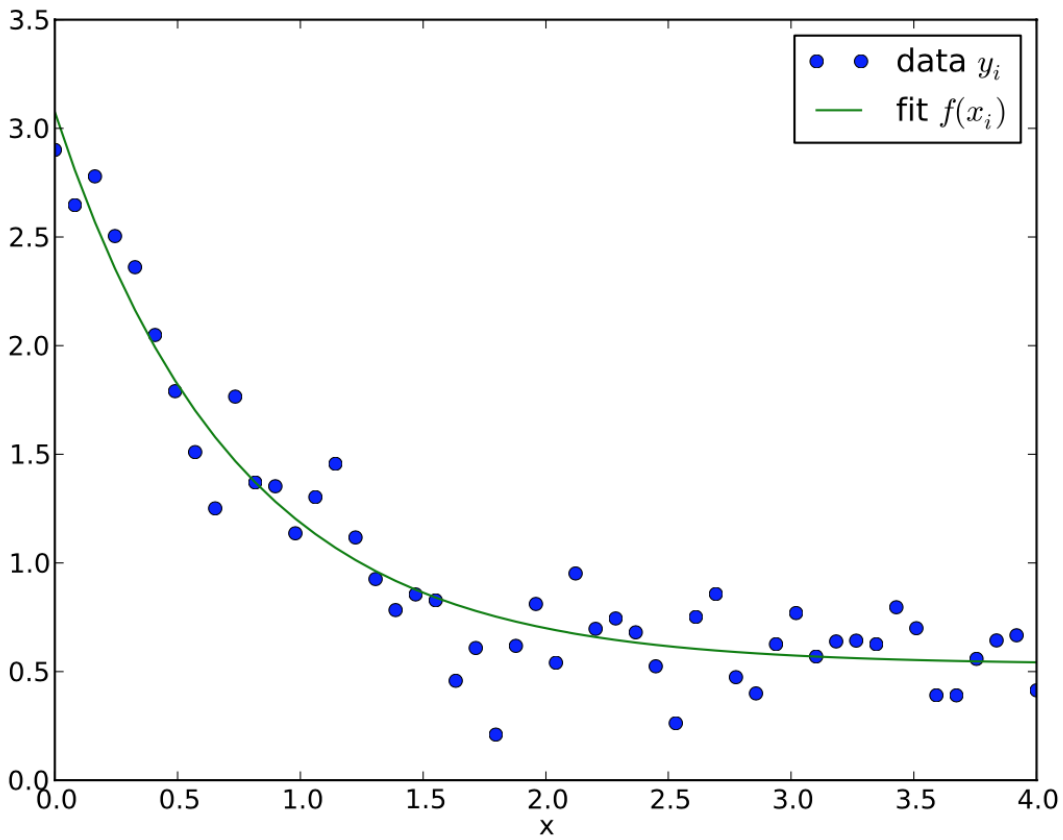
9.7 Nội suy

Nội suy là phương pháp ước tính giá trị của các điểm dữ liệu chưa biết trong phạm vi của một tập hợp rời rạc chứa một số điểm dữ liệu đã biết.

Trong khoa học kỹ thuật, người ta thường có một số điểm dữ liệu đã biết giá trị bằng cách lấy mẫu thực nghiệm. Những điểm này là giá trị đại diện của một hàm số của một biến số độc lập có một lượng giới hạn các giá trị. Thường chúng ta phải nội suy (hoặc ước tính) giá trị của hàm số này cho một giá trị trung gian của một biến độc lập

Cho một tập N điểm (x_i, y_i) với $i = 1, 2, \ldots N$. Chúng ta cần tìm ra hàm f(x) thoả mãn $y_i = f(x_i)$ với $x == x_i$

Hàm `y0 = scipy.interpolate.interp1d (x, y, kind = 'near')` thực hiện nội suy này dựa trên các đường cong liên tục qua các điểm dữ liệu. Lưu ý rằng hàm `interp1d` trả về một hàm `y0`, sau đó sẽ nội suy dữ liệu y cho bất kỳ x truyền vào y0(x). Ví dụ sau đây thực hiện nội suy hàm như đồ thị dưới.




```
# Tạo dữ liệu test
import numpy as np
```

```
import scipy.interpolate
import pylab
def create_data(n):
  """Given an integer n, returns n data points x and values y as a numpy.a
  xmax = 5.
  x = np.linspace(0, xmax, n)
  y = - x**2
  #make x-data somewhat irregular
  y += 1.5 * np.random.normal(size=len(x))
  return x, y
```

```
#Phần thực hiện
n = 10
x, y = create_data(n)
#use finer and regular mesh for plot
xfine = np.linspace(0.1, 4.9, n * 100)
#interpolate with piecewise constant function (p=0)
y0 = scipy.interpolate.interp1d(x, y, kind='nearest')
#interpolate with piecewise linear func (p=1)
y1 = scipy.interpolate.interp1d(x, y, kind='linear')
#interpolate with piecewise constant func (p=2)
y2 = scipy.interpolate.interp1d(x, y, kind='quadratic')
pylab.plot(x, y, 'o', label='Các điểm dữ liệu')
pylab.plot(xfine, y0(xfine), label='nearest')
pylab.plot(xfine, y1(xfine), label='linear')
pylab.plot(xfine, y2(xfine), label='cubic')
pylab.legend()
pylab.xlabel('x')
pylab.show()
```

Tài liệu tham khảo

1. Python for Data Analysis, Wes McKinney
2. Introduction to Python for Computational Science and Engineering, Hans Fangohr, Faculty of Engineering and the Environment University of Southampton
3. https://phamdinhkhanh.github.io/2019/09/16/VisualizationPython.html
4. Một số blog trên internet.