

BÀI 1: GIỚI THIỆU NGÔN NGỮ PYTHON

Xem bài học trên website để ủng hộ Kteam: [Giới thiệu ngôn ngữ lập trình Python](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Tổng quan ngôn ngữ Python

NGÔN NGỮ PYTHON được Guido van Rossum tạo ra cuối năm 1990.

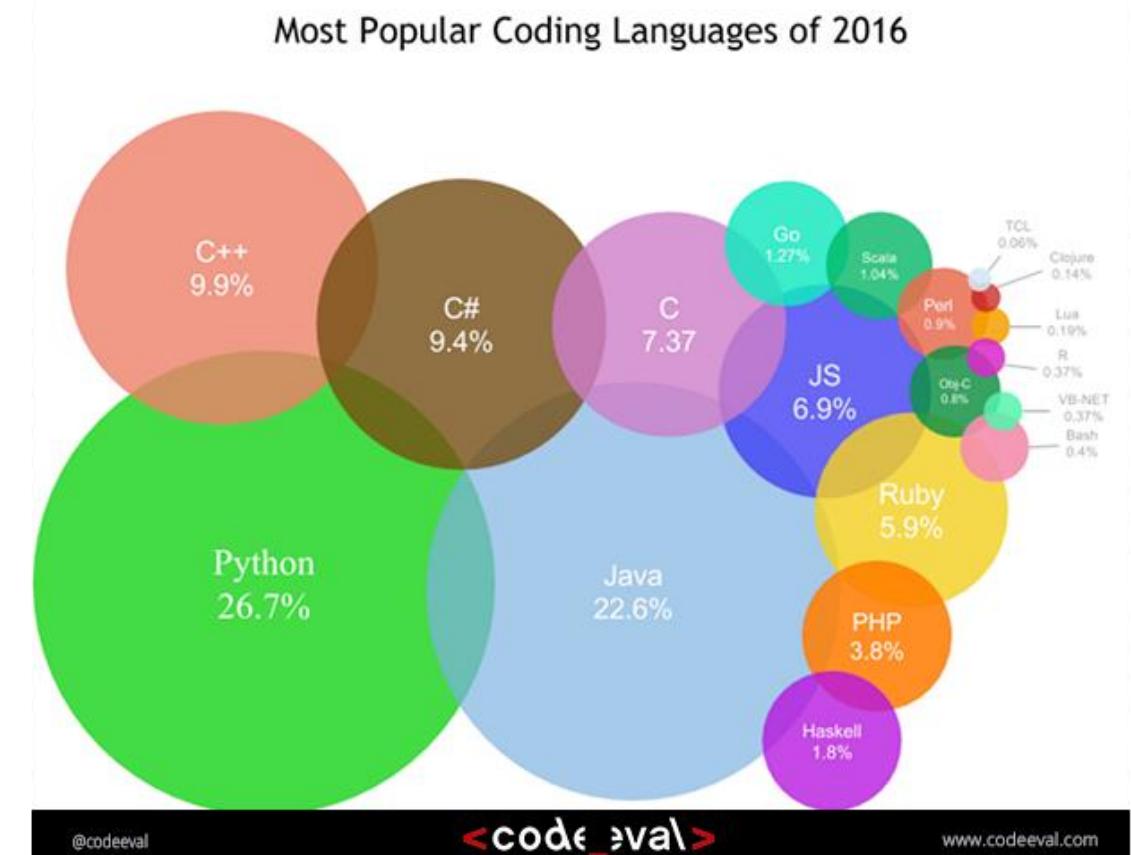
Python khá giống Perl, Ruby, Scheme, Smalltalk và Tcl.

Python được phát triển trong một dự án mã mở do một tổ chức phi lợi nhuận Python Software Foundation quản lý.

Python được phát triển để chạy trên nền Unix. Nhưng theo thời gian, nó đã "bành trướng" sang mọi hệ điều hành từ MS-DOS đến MAC OS, OS/2, Windows, Linux và một số điều hành khác thuộc họ Unix.

Python là ngôn ngữ **bậc cao (high-level)**. có hình thức sáng sủa, cấu trúc rõ ràng, thuận tiện cho người mới học lập trình. Cho phép người sử dụng viết mã với số lần gõ phím tối thiểu.

Python cũng là một trong những ngôn ngữ phổ biến nhất thế giới.



Tại sao nên học Python

Trước khi học ngôn ngữ này, ít ra bạn cũng nên có chút hiểu biết sơ lược về ưu điểm của ngôn ngữ này so với các ngôn ngữ khác. Tìm hiểu một số ưu điểm sau của **PYTHON** cùng **Kteam** nhé!

- Cú pháp dễ đọc, dễ học.
- Thư viện phong phú (<https://pypi.python.org>)
- Cộng đồng sử dụng lớn.
- Lương của lập trình viên Python cao

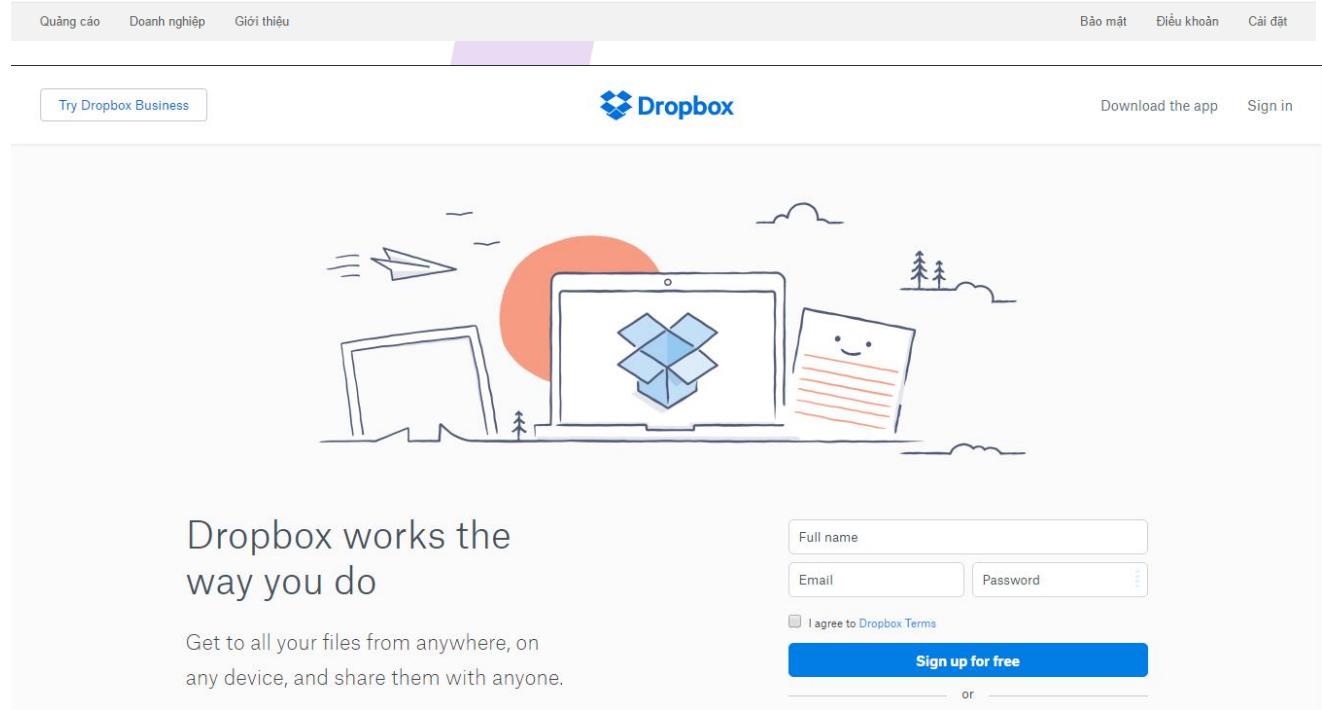
Những điều mà Python có thể làm được

Lập trình web

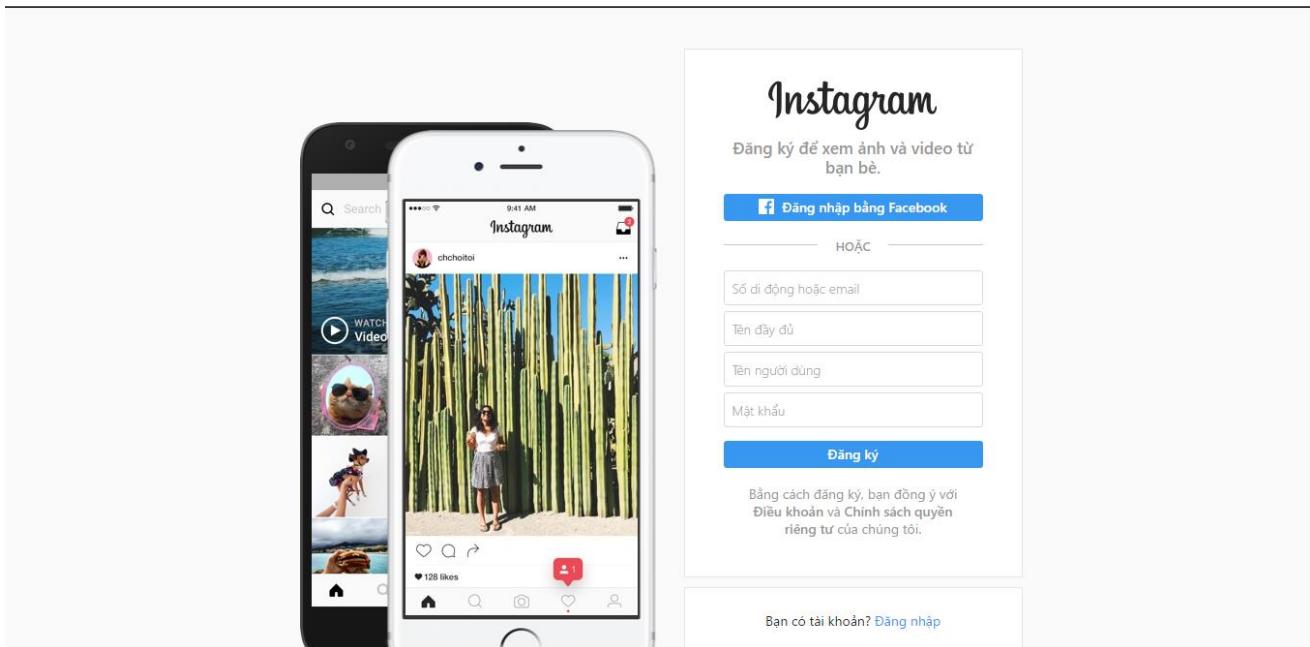
YouTube, Google, Dropbox, Quora, Reddit, Instagram, Nasa, Firefox, Yahoo Maps,...



The screenshot shows the Google search homepage in Vietnamese. At the top right are links for 'Gmail', 'Hình ảnh', and a grid icon. A blue button labeled 'Đăng nhập' (Log in) is also present. The main header features the 'Google' logo with 'Việt Nam' underneath. Below the logo is a search bar with a placeholder 'Tìm với Google'. To the right of the search bar are icons for a keyboard and microphone. Underneath the search bar are two buttons: 'Tim với Google' (Search with Google) and 'Xem trang đầu tiên tìm được' (View first search result). At the bottom of the page, there's a note about Google.com.vn supporting multiple languages: English, Français, 中文 (繁體). Navigation links at the bottom include 'Quảng cáo', 'Doanh nghiệp', 'Giới thiệu', 'Bảo mật', 'Điều khoản', and 'Cài đặt'.

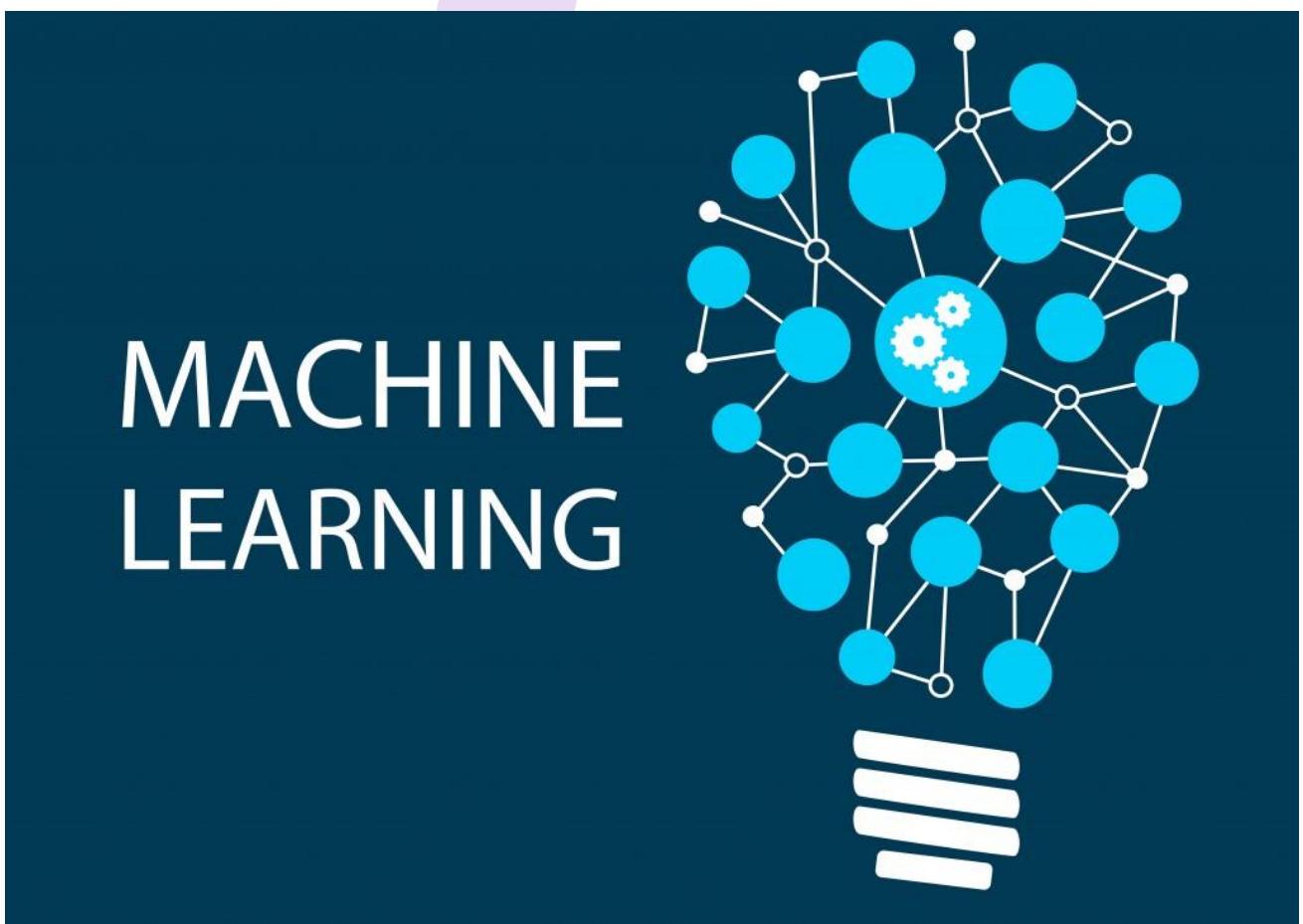


The screenshot shows the Dropbox sign-up page. At the top left is a link to 'Try Dropbox Business'. In the center is the 'Dropbox' logo. To the right are links for 'Download the app' and 'Sign in'. The main visual is a hand-drawn style illustration of a laptop displaying the Dropbox logo, with a smiling notepad character to its right. The background shows a simple landscape with a sun, clouds, and trees. Below the illustration, the text 'Dropbox works the way you do' is displayed, followed by the subtext 'Get to all your files from anywhere, on any device, and share them with anyone.' On the right side, there's a sign-up form with fields for 'Full name', 'Email', and 'Password'. A checkbox for 'I agree to Dropbox Terms' is present, along with a large blue 'Sign up for free' button. Below the sign-up form is a horizontal line with the word 'or' in the center.

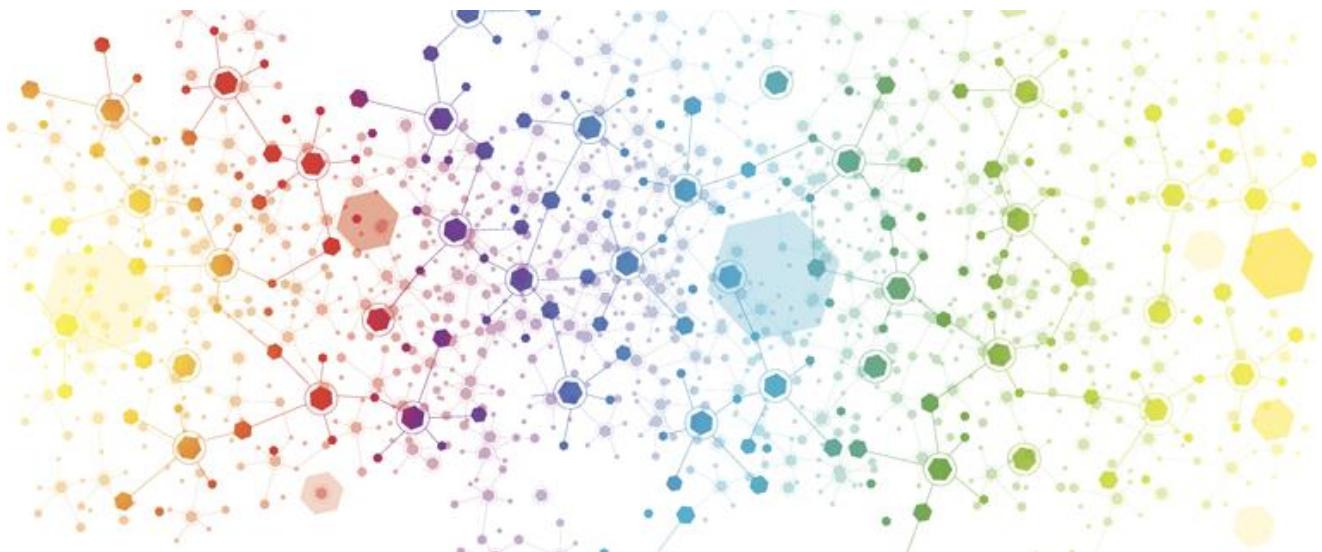


Data Analysis

- Machine Learning



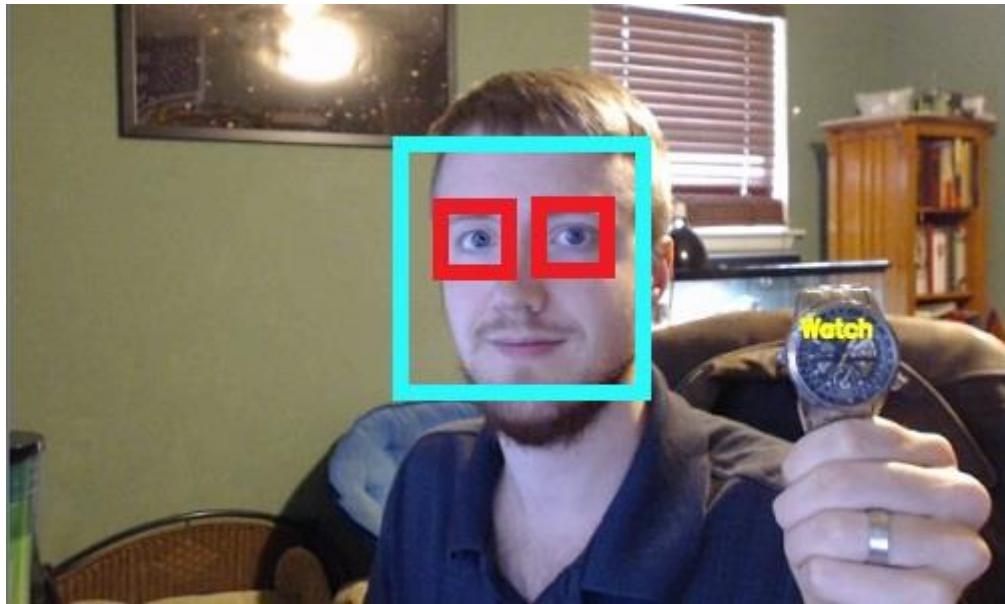
- Data Visualization



- Google Cloud



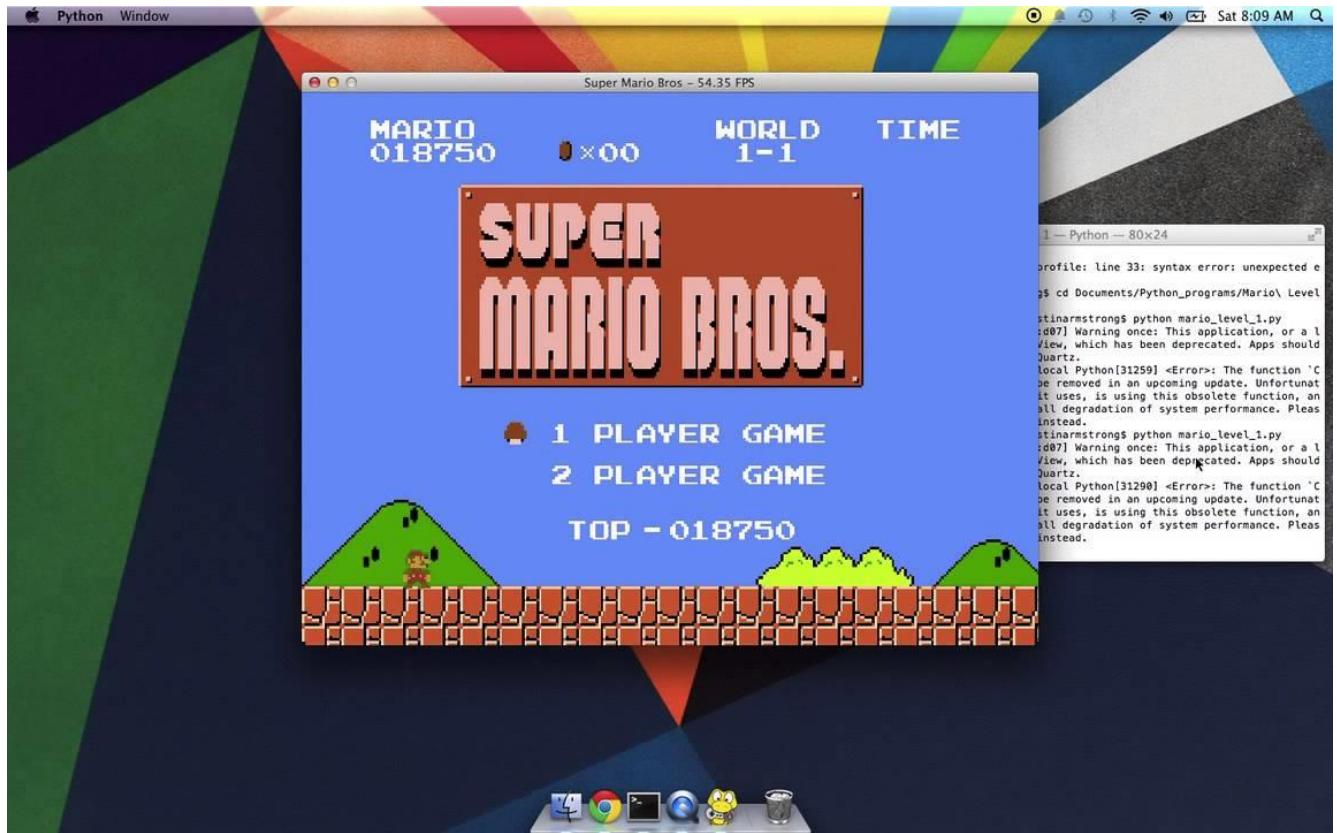
- Image and Video Analysis



Lập trình robot



Lập trình game



Lập trình ứng dụng

Ứng dụng lập trình bởi python có thể chạy trên nhiều hệ điều hành (Sublime Text, Blender)

A screenshot of a code editor interface, likely Sublime Text, displaying a file named `balinov.scss`. The left pane shows the main SCSS code, and the right pane shows a search results sidebar titled "balinov.scss — balinov". The sidebar lists several SCSS files and their locations:

- 217 .utility {
background: #000;
218 } // utility
- 219 .group {
margin-right:
220 }
221 ul {
a {
222 }
223 }
224 }
225 }
226 }
227 } // utility
- 228 .page-header {
background-image:
background-repeat:
color: rgba(fff,
@include box-sh:
229 h1 {
margin-bottom:
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 .home>Welcome {
p {
@include font-size(20,30);
240 }
241 }
242 }
243 }
244 }
245 .grid-container {
width: 100%;
margin: \$ gutter-width 0;
- 217 scss
220 balinov.scss
static/sass/balinov.scss
208 mixins.scss
static/sass/_mixins.scss
208 base.scss
static/sass/_base.scss
208 reset.scss
static/sass/_reset.scss
184 balinov.css
static/css/balinov.css
logo.svg
static/images/logo.svg
logo_light.svg
static/images/logo_light.svg
isotope-gutter.js
static/js/isotope-gutter.js
masonry.pkgd.js
static/js/masonry.pkgd.js

The sidebar also features a large orange letter "S" icon.

Bảo mật mạng và máy tính



Kết luận

Qua bài này, Kteam đã giới thiệu đến bạn TỔNG QUAN VỀ LẬP TRÌNH PYTHON

Trong bài tiếp theo, mình sẽ hướng dẫn các bạn cách [CÀI ĐẶT MÔI TRƯỜNG LÀM VIỆC VỚI PYTHON.](#)

Cảm ơn các bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thử thách – Không ngại khó**".

Bài 2: CÀI ĐẶT MÔI TRƯỜNG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Cài đặt môi trường Python](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Ở bài trước, **Kteam** đã giới thiệu với các bạn về **NGÔN NGỮ LẬP TRÌNH PYTHON** cũng như những ứng dụng mạnh mẽ của ngôn ngữ này.

Trong bài này, chúng ta sẽ cùng nhau tìm hiểu về cách **CÀI ĐẶT MÔI TRƯỜNG PHÁT TRIỂN** cho ngôn ngữ Python.

Nào! Chúng ta bắt đầu!

Nội Dung Chính

Trong bài học này chúng ta sẽ tìm hiểu các điều sau đây:

- Download và cài đặt môi trường Python
- Giới thiệu Sublime Text. Editor sử dụng trong khóa học này

Download và cài đặt môi trường Python

Ở khóa học này, **Kteam** sử dụng phiên bản Python 3.x. Cụ thể là phiên bản **3.6.1**, phiên bản mới nhất của phiên bản 3.x (trong thời điểm hiện tại 06/2017)

Tiến hành download Python 3.6.1

Bước 1: Truy cập vào **trang chủ** của Python

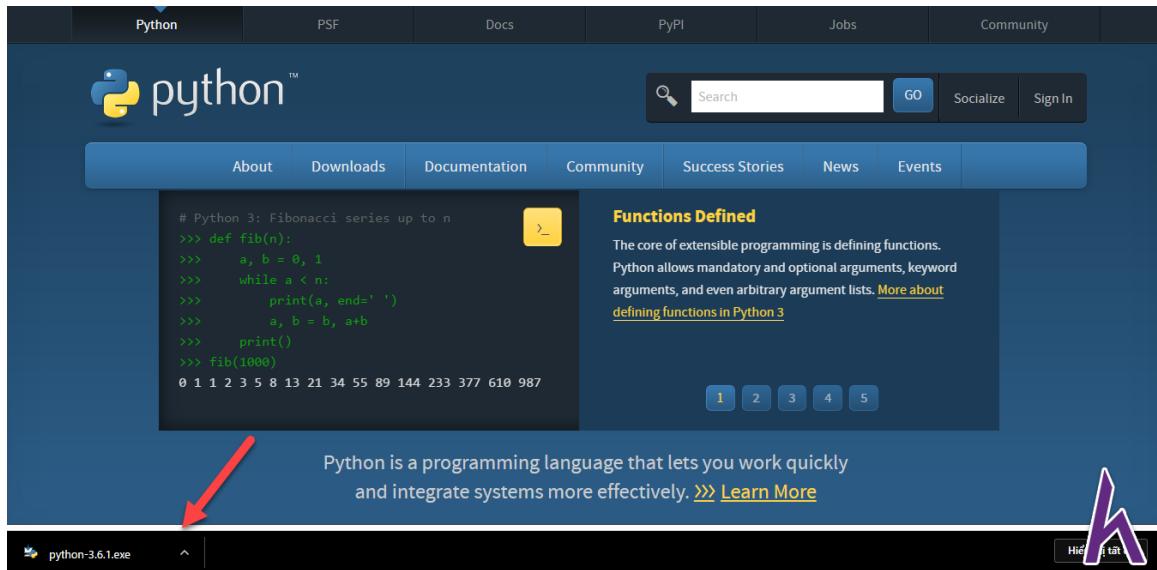
<https://www.python.org/>

The screenshot shows the Python.org homepage. The top navigation bar includes links for Python, PSF, Docs, PyPI, Jobs, and Community. Below the header is a search bar and a sign-in link. The main content area features a Python logo and a "Quick & Easy to Learn" section with a snippet of Python code demonstrating simple output and input assignment. A large call-to-action button at the bottom encourages users to learn more about Python's capabilities.

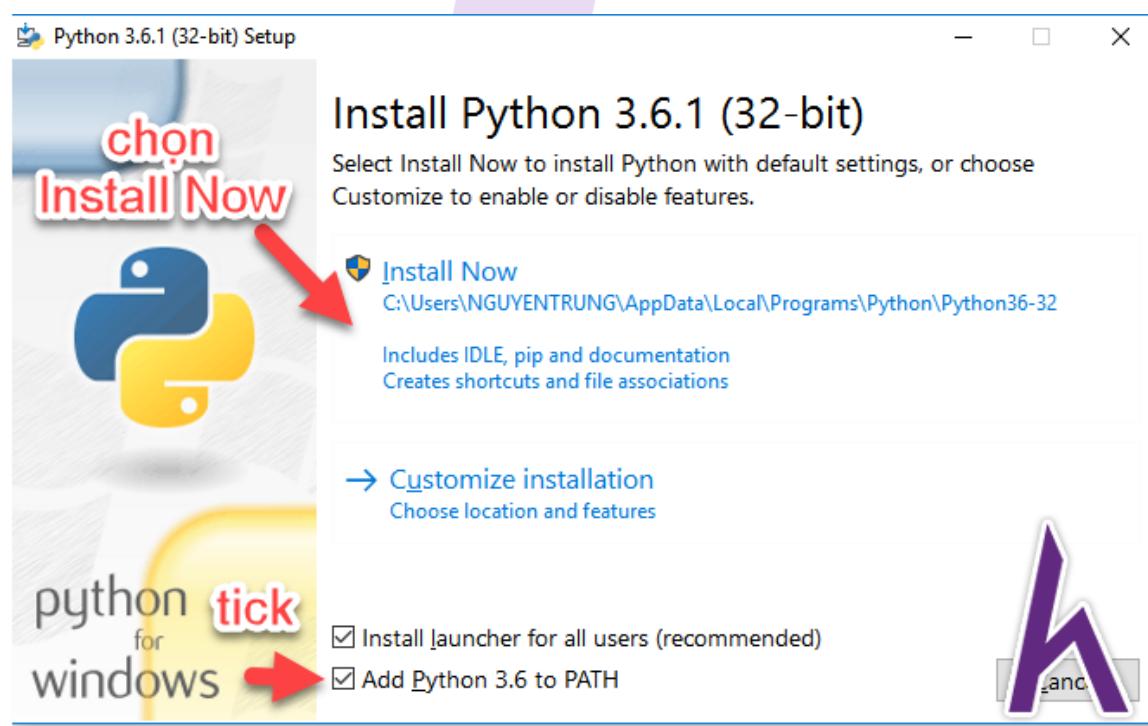
Bước 2: Chọn phiên bản ở mục **Downloads** và nhập chọn **Python 3.6.1** để download

This screenshot shows the 'Downloads' section of the Python.org website. A red arrow points to the 'Downloads' tab in the navigation bar. Another red arrow points to the 'Python 3.6.1' button within a modal window titled 'Download for Windows'. The modal also contains notes about compatibility with Windows XP and information about other platforms like Mac OS X and Other Platforms.

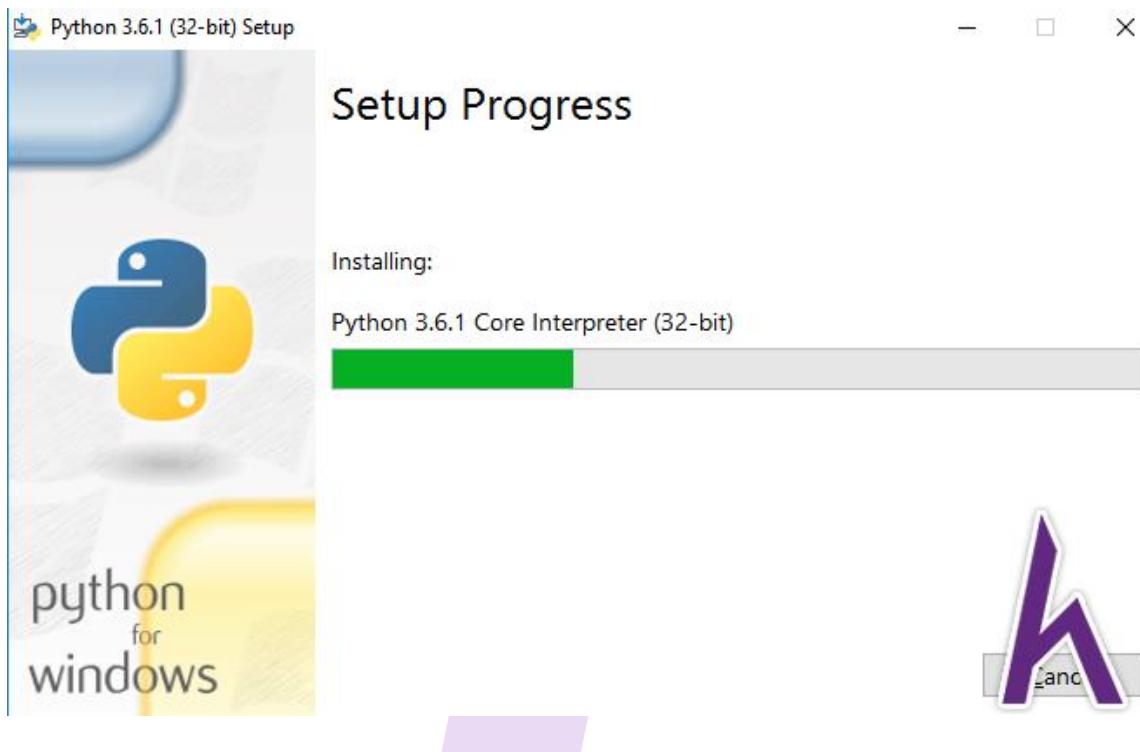
Bước 3: Sau download hoàn tất. Chúng ta nhấn chọn chạy file **python-3.6.1.exe** để bắt đầu tiến trình cài đặt.



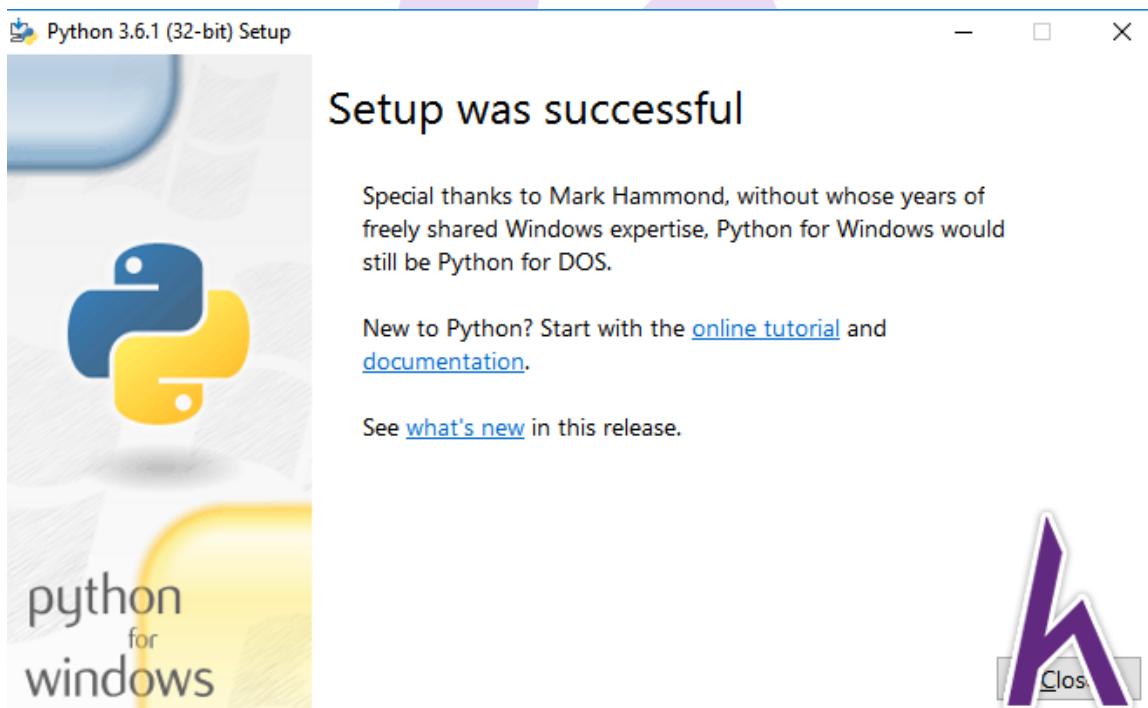
Bước 4: Tick vào ô **Add Python 3.6 to PATH** và chọn **Install Now**



Chờ đợi cho quá trình cài đặt hoàn thành

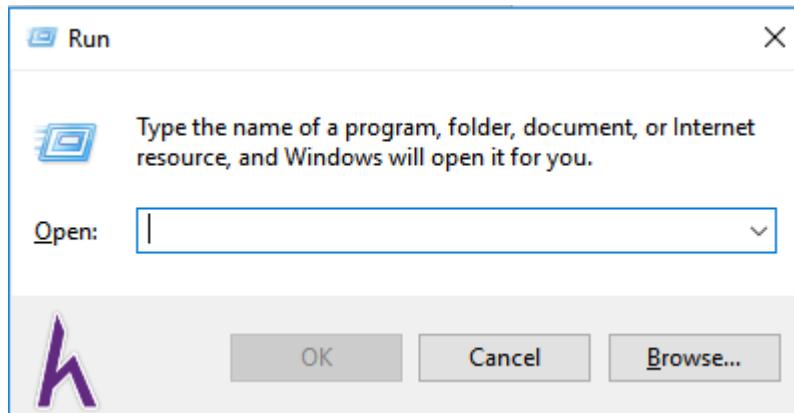


Bước 5: Khi cửa sổ hiển thị **Setup was successful** là ta đã cài đặt thành công môi trường Python > **Close**

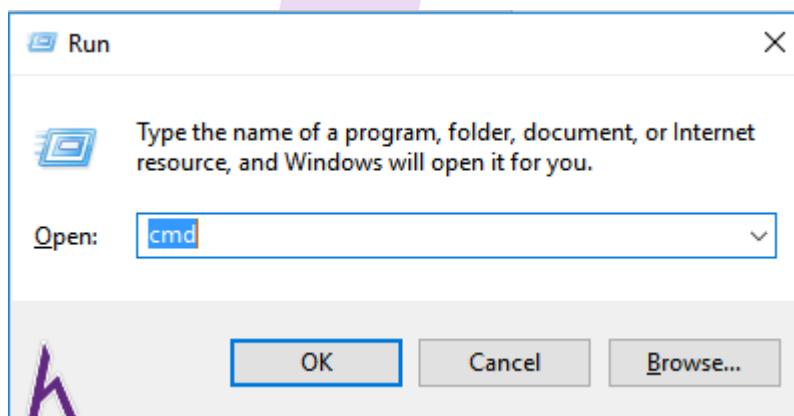


Bước 6: Tiếp đến, ta mở **Command Prompt (CMD)** để kiểm tra.

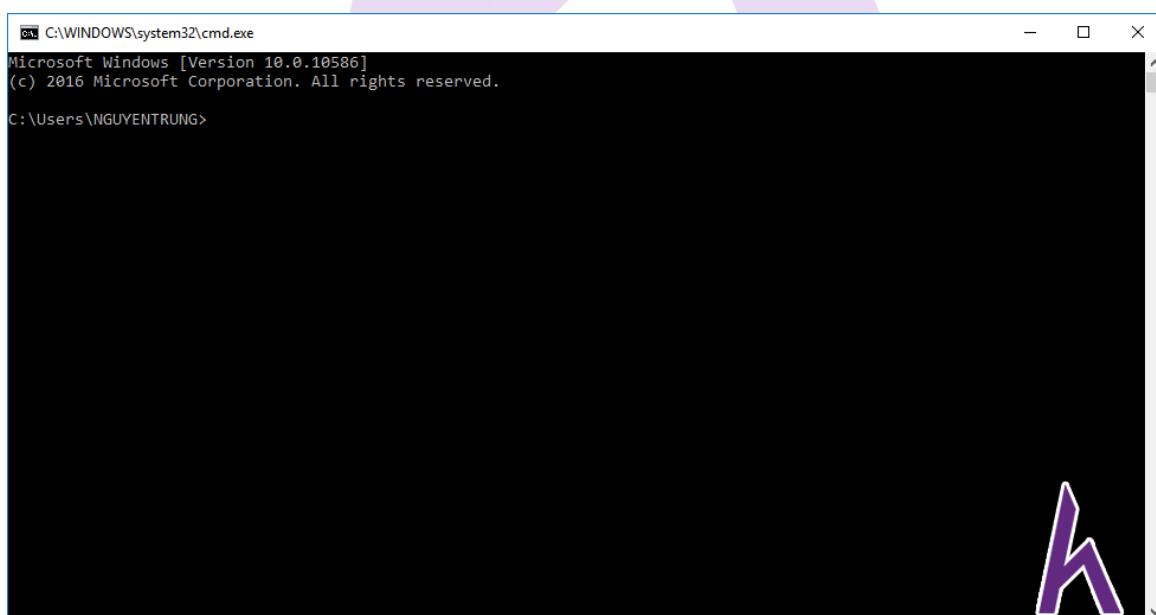
- Để mở **Command Prompt** bạn dùng tổ hợp phím **Windows + R** để mở hộp thoại **Run**



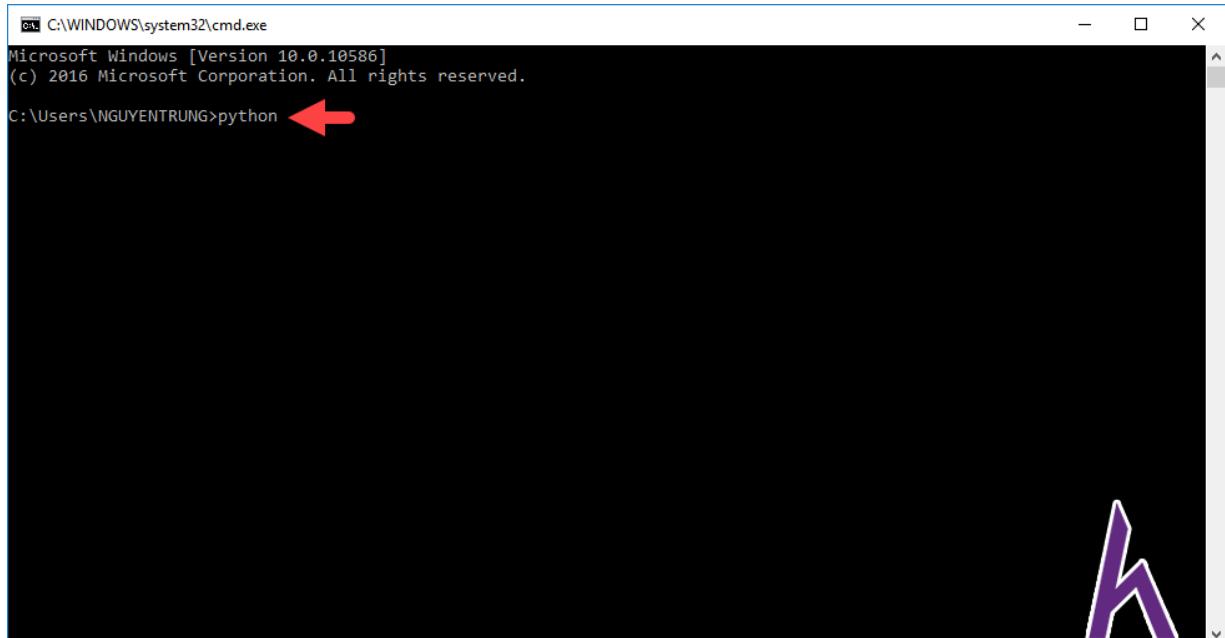
- Sau đó, gõ **cmd** > **Enter** để mở Command Prompt



- Cửa sổ Command Prompt hiển thị như bên dưới:

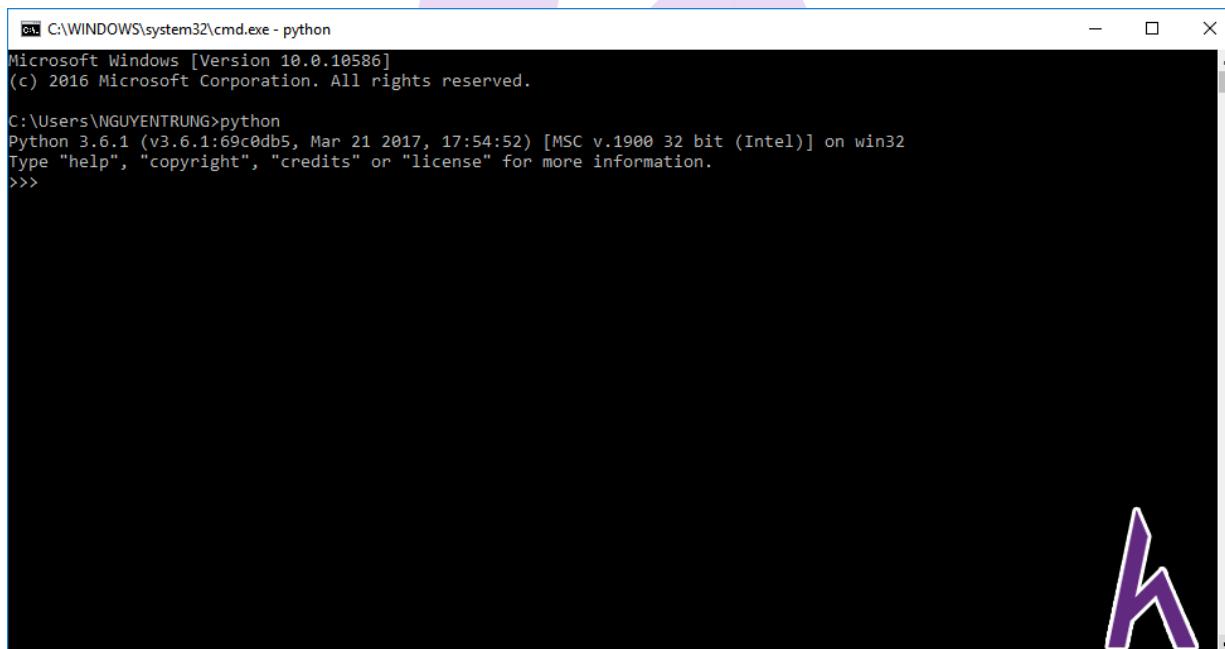


Bước 7: Trong cửa sổ **Command Prompt**, bạn gõ **python** > **Enter** để kiểm tra



A screenshot of a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The window shows the text: 'Microsoft Windows [Version 10.0.10586]' and '(c) 2016 Microsoft Corporation. All rights reserved.' Below this, the command 'C:\Users\NGUYENTRUNG>python' is typed, with a red arrow pointing to the 'python' part.

- Nếu bạn hiển thị ra được **shell** để tương tác với Python như hình ở dưới có nghĩa là phần cài đặt đã hoàn tất.



A screenshot of a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe - python'. The window shows the text: 'Microsoft Windows [Version 10.0.10586]' and '(c) 2016 Microsoft Corporation. All rights reserved.' Below this, the command 'C:\Users\NGUYENTRUNG>python' is typed, followed by the Python version information: 'Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32'. The prompt then changes to 'Type "help", "copyright", "credits" or "license" for more information.' and shows '>>>' indicating the start of the Python shell.

Đó là phần download và cài đặt Python trên **Windows**. Còn về các hệ điều hành thuộc Linux và hệ điều hành Mac OS thì tất cả những hệ điều hành đó đều đã được tích hợp Python 2.x lẫn Python 3.x. Các bạn muốn thao tác với nó

chỉ cần mở **Terminal** và gõ **python2** để chọn Python 2.x hoặc **python3** để chọn Python 3.x

Giới thiệu Sublime Text

Ở khóa học này, **Kteam** sẽ sử dụng editor soạn thảo code đó chính là **SUBLIME TEXT**. Phần download và cài đặt các bạn có thể xem chi tiết trong bài [HƯỚNG DẪN CÀI ĐẶT SUBLIME TEXT](#).



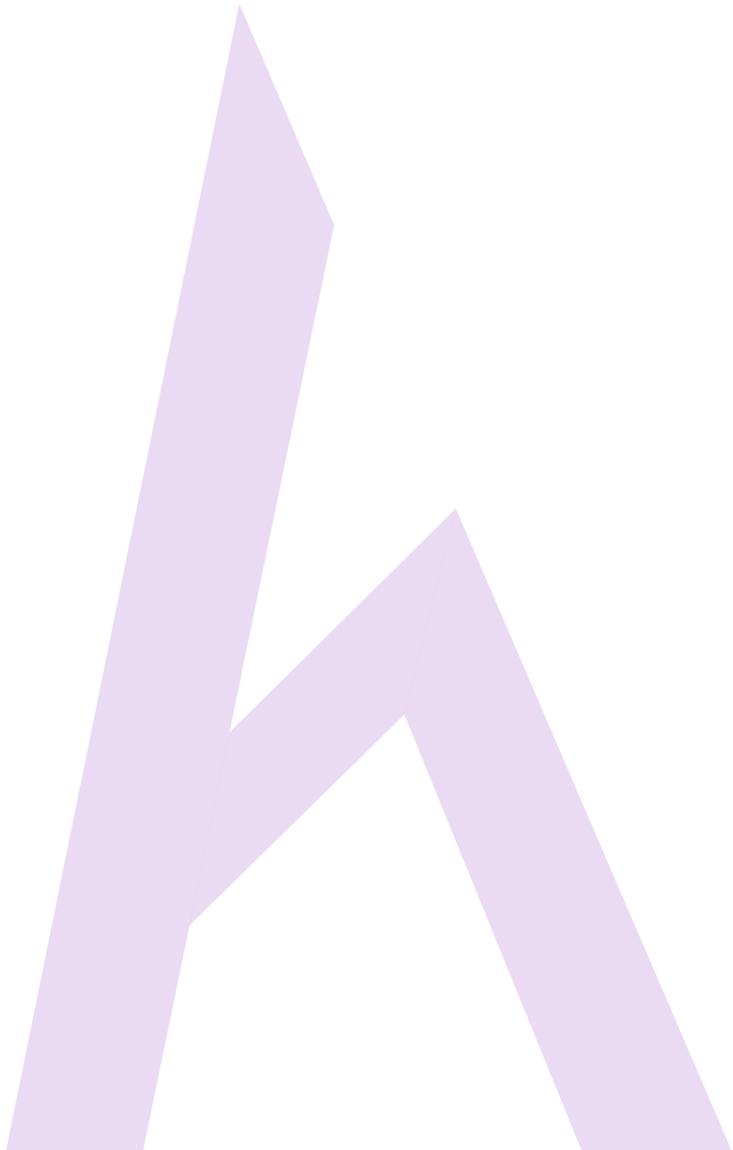
Mọi hướng dẫn cài đặt chi tiết về các phần mềm liên quan, bạn có thể tìm thấy ngay trong mục [HƯỚNG DẪN CÀI ĐẶT](#) tại [Howteam.com](#) hoặc có thể phản hồi trong **BÌNH LUẬN** bên dưới để được hỗ trợ.

Kết Luận

Trong bài này, chúng ta đã hoàn thiện việc chuẩn bị MÔI TRƯỜNG PHÁT TRIỂN CHO NGÔN NGỮ PYTHON.

Ở bài tiếp theo, mình sẽ hướng dẫn các bạn [CÁC CÁCH CHẠY CHƯƠNG TRÌNH PYTHON.](#)

Cảm ơn các bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".



Bài 3: CHẠY CHƯƠNG TRÌNH PYTHON

Xem bài học trên website để ủng hộ Kteam: [Chạy chương trình Python](#).

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong các bài trước chúng ta đã cùng nhau tìm hiểu [TỔNG QUAN NGÔN NGỮ PYTHON](#) cũng như [MÔI TRƯỜNG PHÁT TRIỂN](#) của nó.

Ở bài này, Kteam sẽ hướng dẫn các bạn **CÁCH CHẠY MỘT CHƯƠNG TRÌNH PYTHON** cơ bản nhé!

Nội dung chính

Để theo dõi bài này, bạn nên xem qua bài:

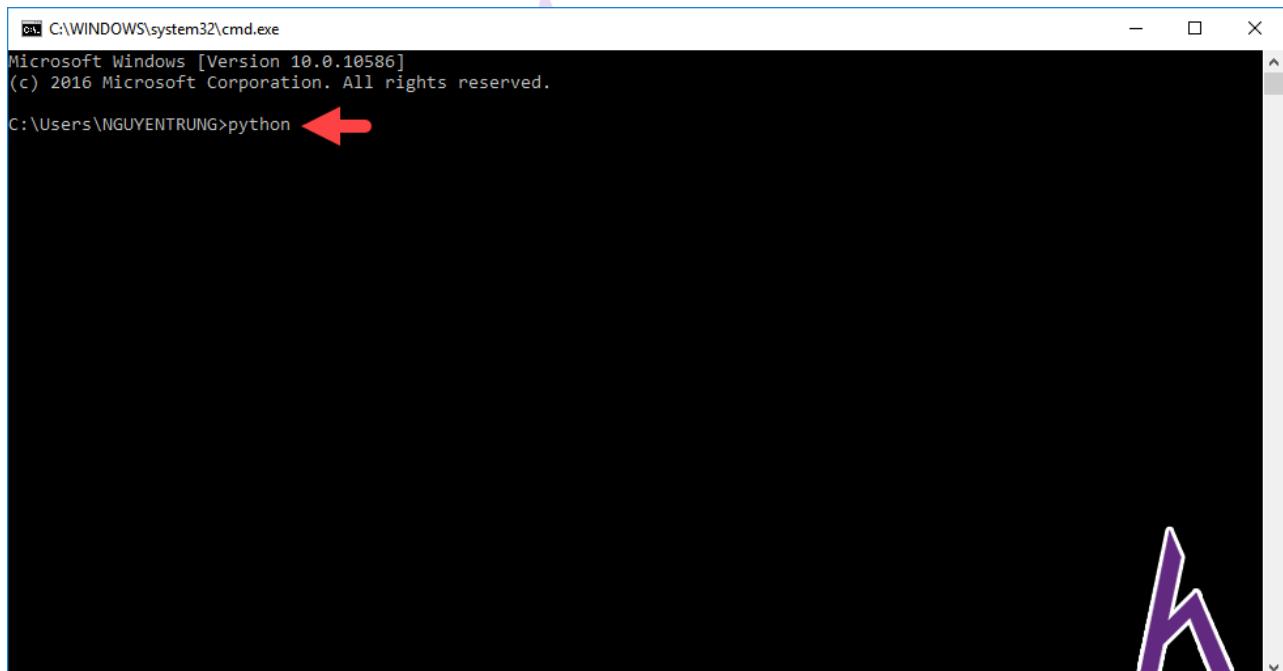
- [CÀI ĐẶT MÔI TRƯỜNG PHÁT TRIỂN CHO PYTHON.](#)

Trong bài này, chúng ta sẽ tìm hiểu:

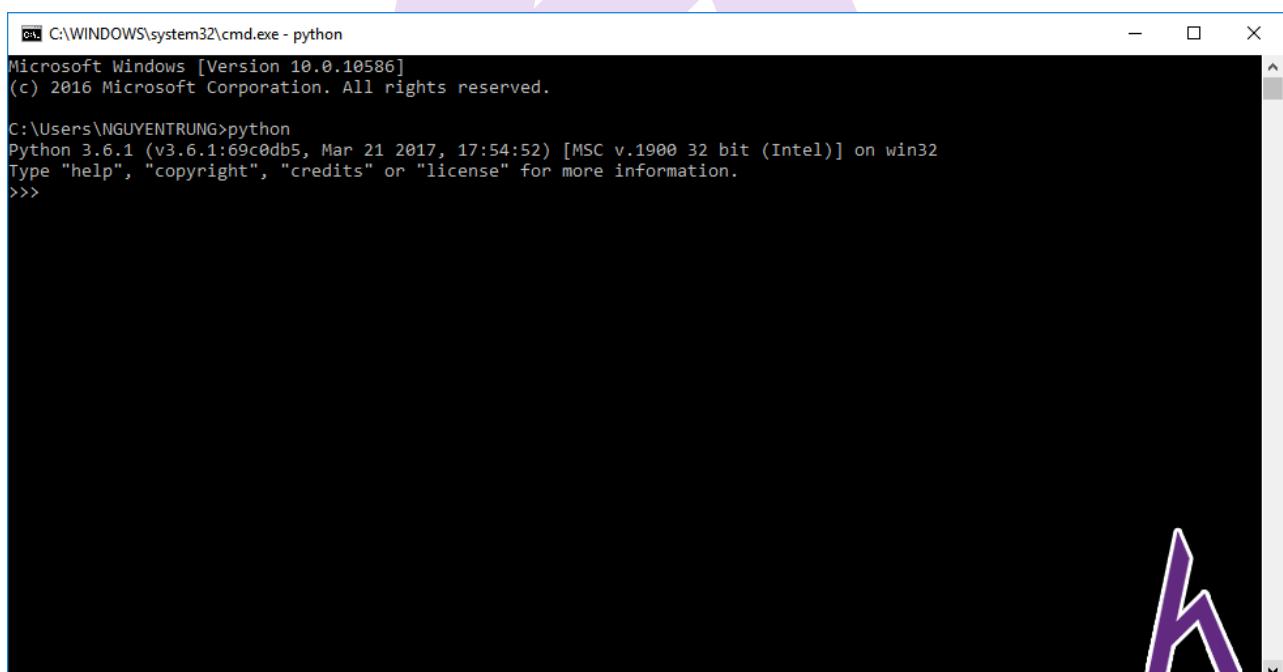
- Thao tác trực tiếp với Interactive Prompt
 - Chạy chương trình bằng command line
-

Thao tác trực tiếp với Interactive Prompt

Như ở [bài trước](#), các bạn sử dụng **Command Prompt** để kiểm tra cài đặt thành công hay chưa bằng cách gõ **python** lên Command Prompt.



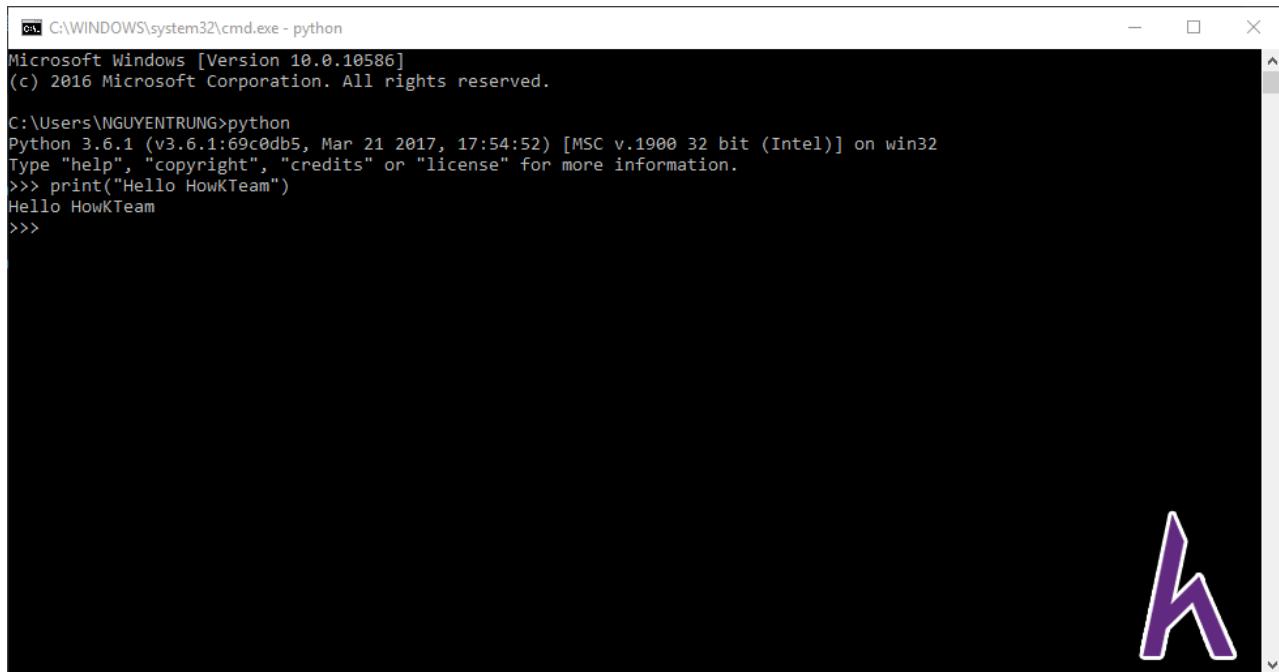
A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window shows the following text:
Microsoft Windows [Version 10.0.10586]
(c) 2016 Microsoft Corporation. All rights reserved.
C:\Users\NGUYENTRUNG>python **←**



A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe - python'. The window shows the following text:
Microsoft Windows [Version 10.0.10586]
(c) 2016 Microsoft Corporation. All rights reserved.
C:\Users\NGUYENTRUNG>python
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>

Như các bạn thấy ở hình trên đó chính là một **Interactive Prompt** để thao tác với Python.

Và chúng ta sẽ chạy chương trình huyền thoại “**Hello HowKTeam**”.



```
C:\WINDOWS\system32\cmd.exe - python
Microsoft Windows [Version 10.0.10586]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\NGUYENTRUNG>python
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello HowKTeam")
Hello HowKTeam
>>>
```

Tạm thời các bạn không cần quan tâm đến cách mà hàm **print("Hello HowKTeam")** hoạt động. Mình sẽ giới thiệu hàm này ở bài NHẬP XUẤT TRONG PYTHON.

Như các bạn có thể thấy mình vừa thao tác trực tiếp với interactive prompt.

Ưu điểm:

- Thao tác đơn giản, dễ dàng.
- Cho kết quả ngay lập tức khi kết thúc câu lệnh

Nhược điểm:

- Không thích hợp cho việc một dãy lệnh dài, có cấu trúc
- Khi code sai chính tả, sai logic không thể sửa

Chạy chương trình bằng command line

Muốn chạy một chương trình Python bằng **Command line** thì chúng ta phải tạo ra được một file Python. Một file Python là một file mà có phần đuôi (mở rộng) là **.py**



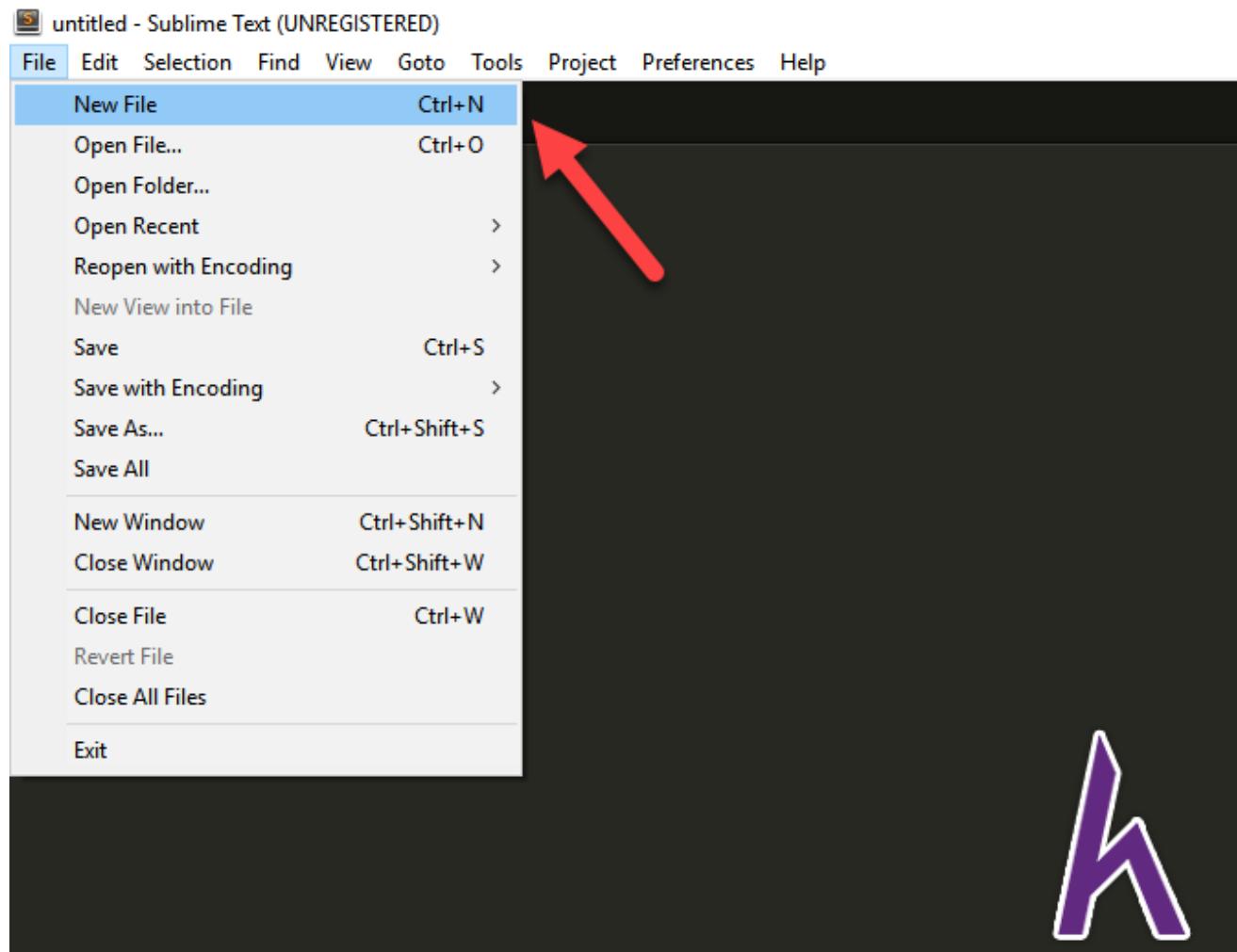
HelloHowKteam.py

Như đã giới thiệu ở bài [CÀI ĐẶT MÔI TRƯỜNG PHÁT TRIỂN PYTHON](#). Editor để soạn code sẽ là [SUBLIME TEXT](#). Và mình sẽ dùng **Sublime Text** tạo một file Python.

Khởi tạo file Python

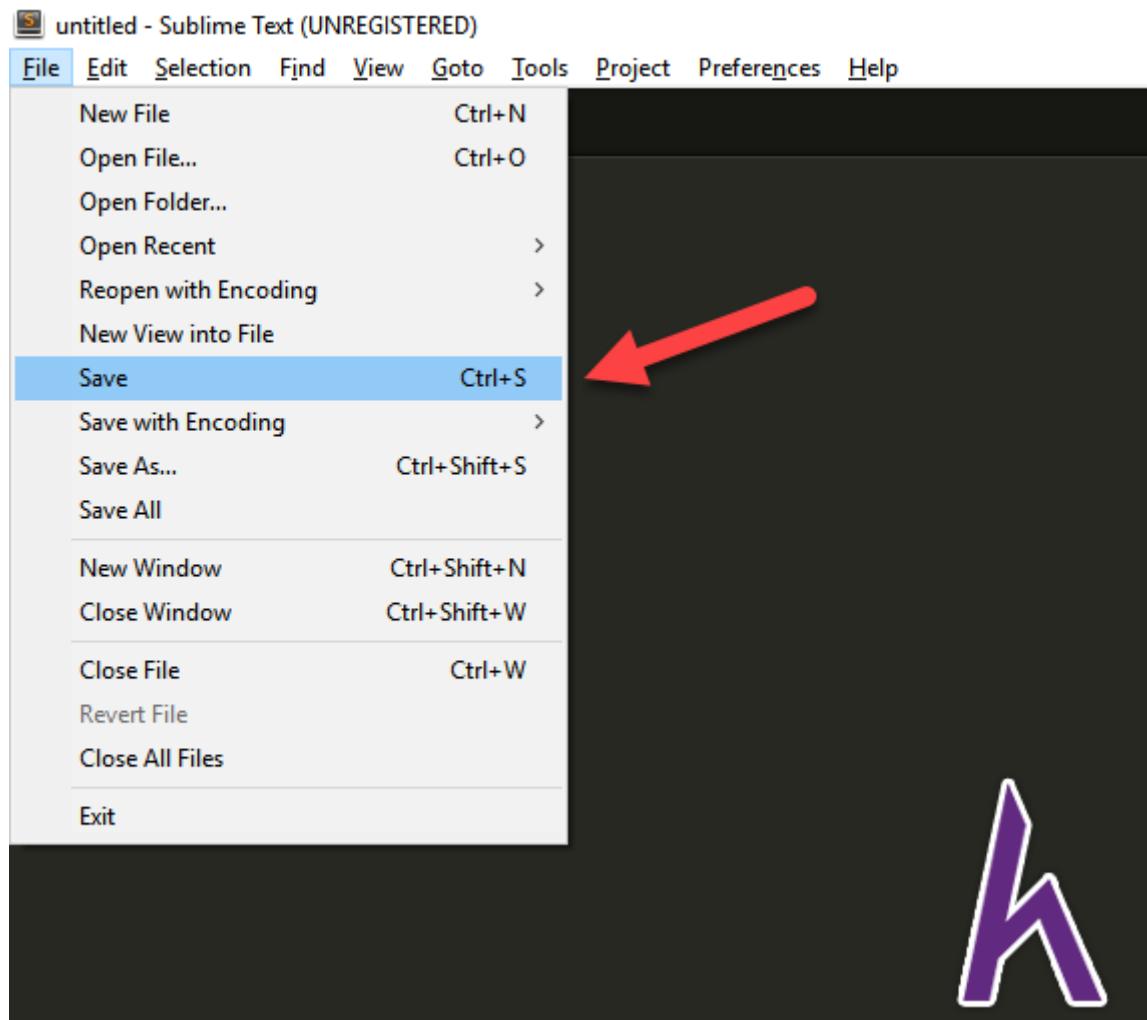
Bước 1: Đầu tiên, chúng ta mở Sublime Text lên và chọn **File > New File**

- Hoặc bạn cũng có thể dùng phím tắt **Ctrl + N**

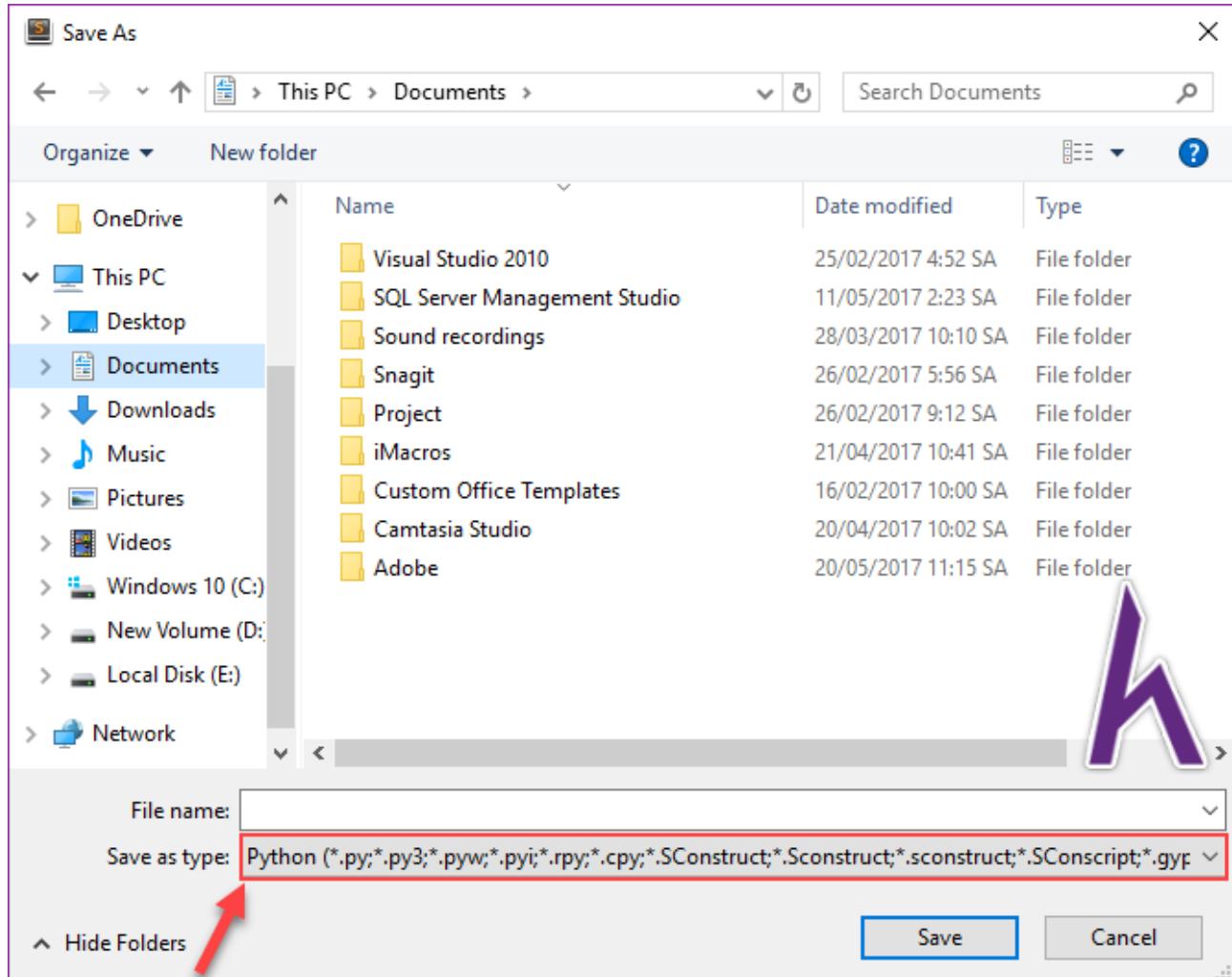


Bước 2: Tiếp tục chọn **File > Save**

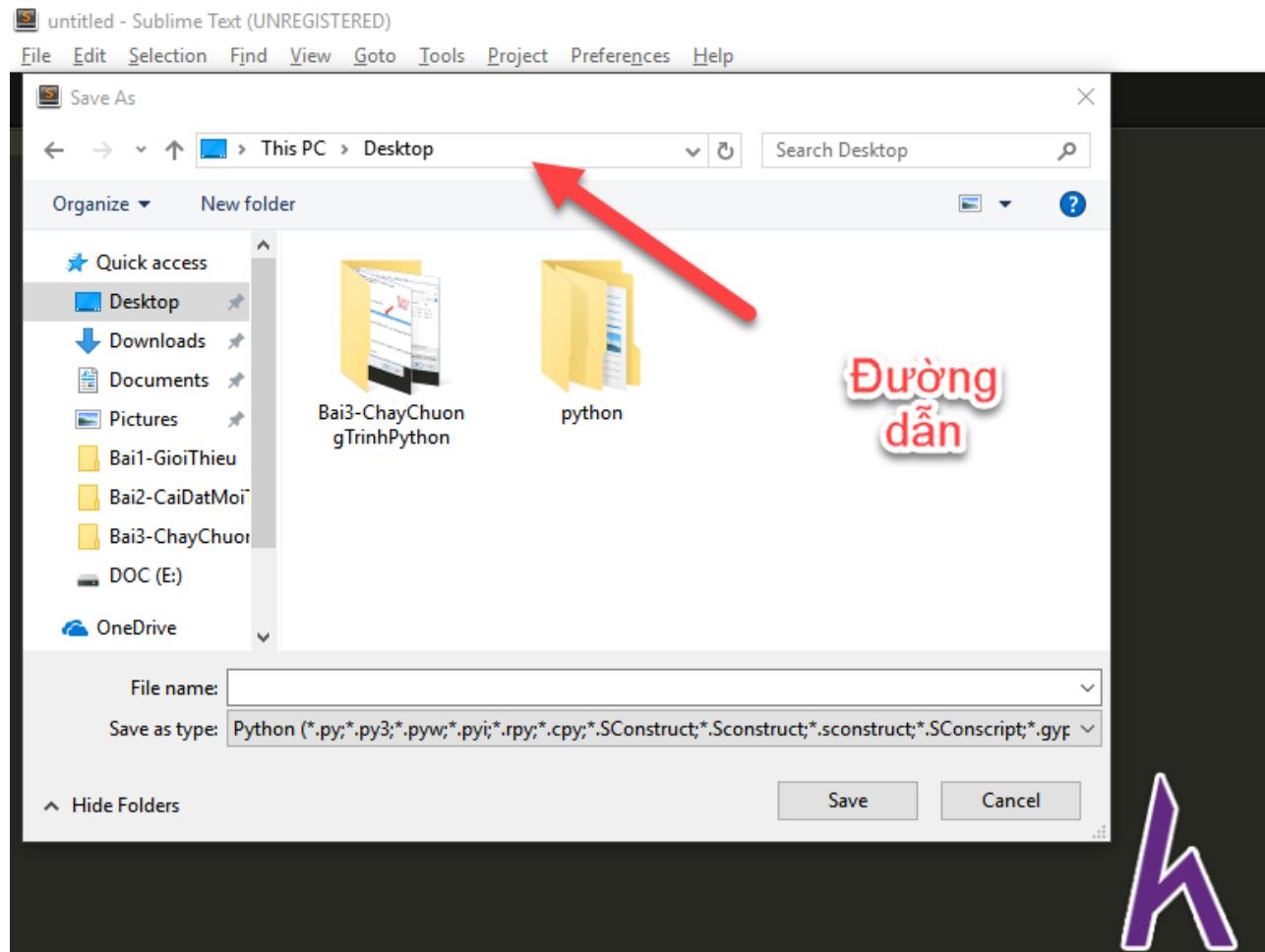
- Hoặc sử dụng phím tắt **Ctrl + S** để lưu lại file



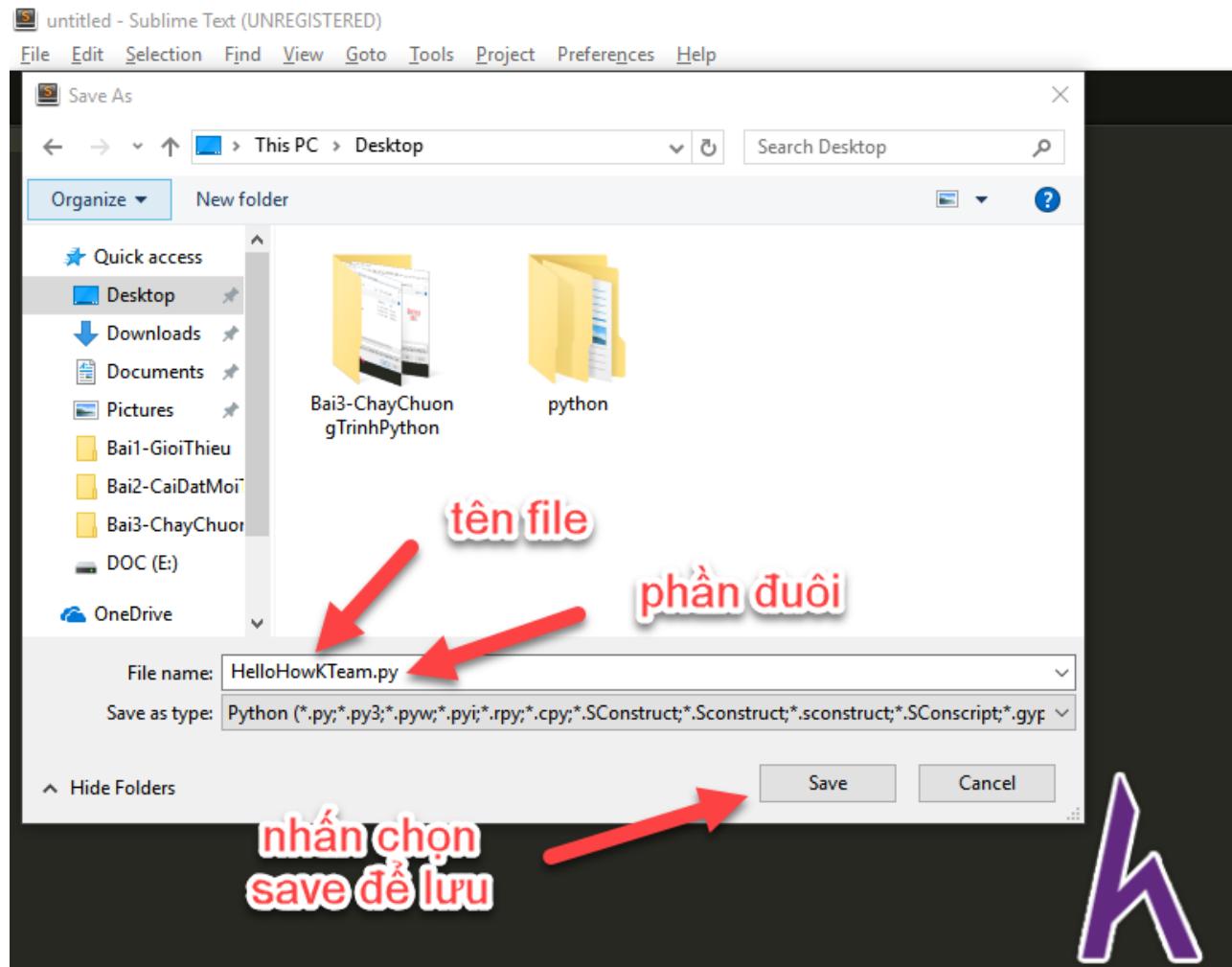
- Sau khi chọn Save thì sẽ có một hộp thoại yêu cầu tại **Save as Type** > chọn **Python** trong danh sách chọn.



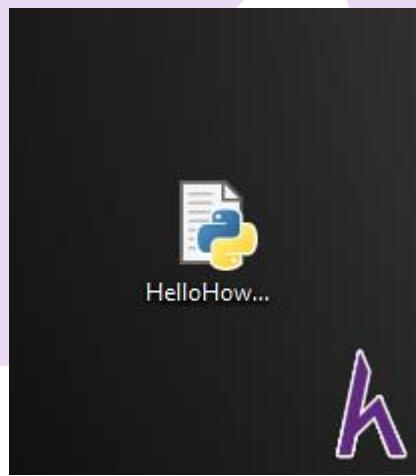
- Tiếp đến, các bạn tìm nơi để lưu lại file. Ở đây mình sẽ lưu ở ngoài Desktop



- Và cuối cùng, các bạn điền **tên file** vào ô **File name**. Các bạn nhớ là phải thêm **.py** vào cuối tên file nữa. Ở đây mình chọn tên file là **HelloHowKTeam** > **Save**

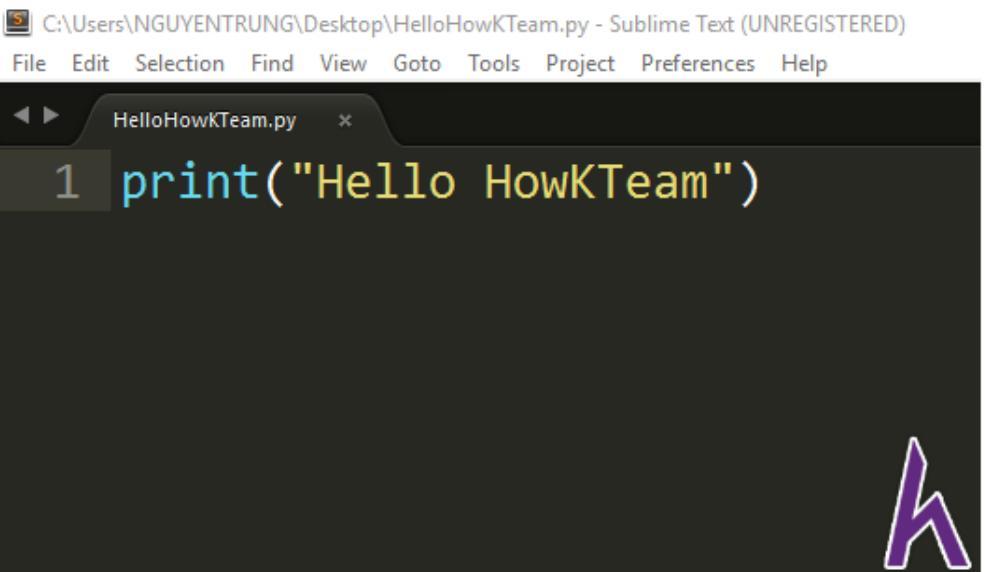


Sau cùng, đây chính là file Python mình vừa tạo ở ngoài Desktop



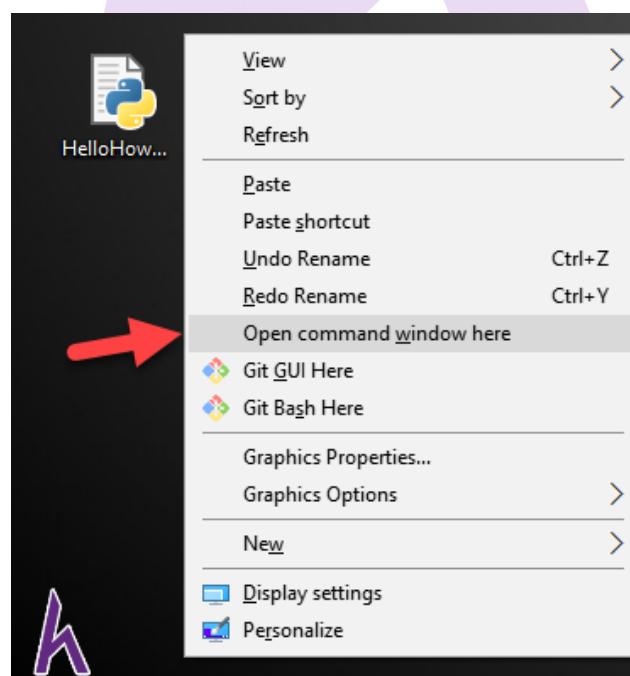
Thao tác trên file vừa khởi tạo

Công đoạn tiếp theo là mình nhập code vào trong file **HelloHowKTeam.py** vừa mới tạo ở phần trên, sau đó lưu lại bằng **Ctrl + S**



```
print ("Hello HowKTeam")
```

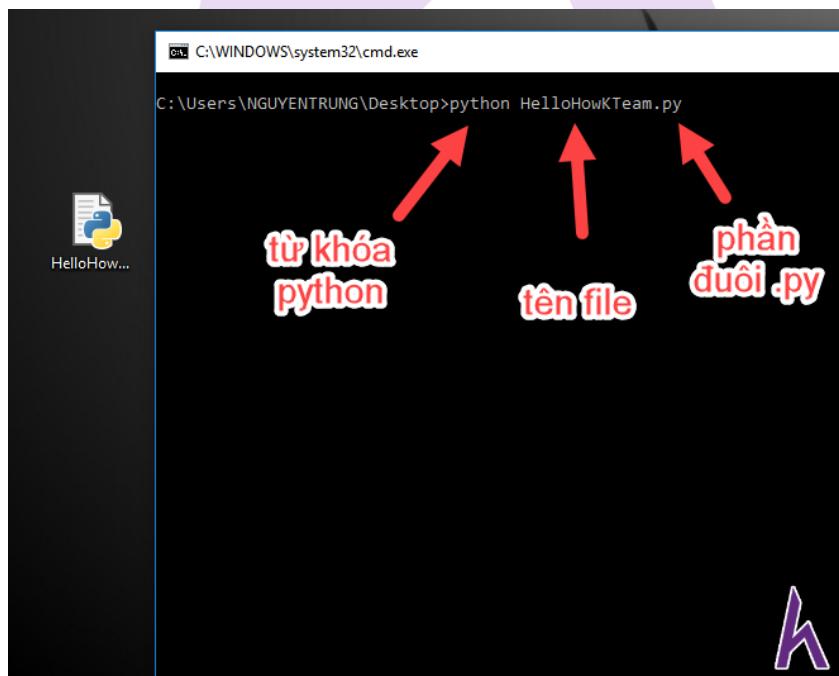
Kế đến mình ra ngoài Desktop – vị trí lưu file khởi tạo để bắt đầu thực hiện chạy chương trình. Các bạn giữ nút **Shift** và nhấn **chuột phải** vào một khoảng trống nào gần đấy và chọn **Open command window here** để mở Command Prompt





Và rồi sau đó, bạn gõ command line với cấu trúc sau > **Enter**

python <tên file.py>



Và như các bạn đã thấy chúng ta đã in ra thành công dòng chữ "Hello HowKTeam"

```
C:\WINDOWS\system32\cmd.exe
C:\Users\NGUYENTRUNG\Desktop>python HelloHowKTeam.py
Hello HowKTeam ←
C:\Users\NGUYENTRUNG\Desktop>
```

kết quả của
chương
trình

Kết Luận

Qua bài học này chúng ta đã hiểu được CÁCH CHẠY CHƯƠNG TRÌNH PYTHON qua interactive mode và bằng command line.

Bài sau chúng ta sẽ tìm hiểu [CÁCH GHI CHÚ TRONG PYTHON](#).

Cảm ơn các bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn nữa. Đừng quên "**Luyện tập – Thủ thắc – Không ngại khó**".

Bài 4: CÁCH GHI CHÚ TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Cách ghi chú trong Python](#).

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, chúng ta đã tìm hiểu [CÁCH TẠO & CHẠY MỘT CHƯƠNG TRÌNH PYTHON](#) cơ bản. Trong bài này, **Kteam** sẽ giới thiệu đến bạn **CÁCH GHI CHÚ TRONG PYTHON**.

Nội dung chính

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).

Trong bài học này, chúng ta sẽ cùng tìm hiểu các vấn đề:

- Cách ghi chú trong Python.
 - Chúng ta bắt đầu thôi!
-

Ghi chú trong Python

Khi làm một công việc nào đó muốn đạt hiểu quả cao thì một trong những cách hữu hiệu đó là làm việc nhóm. Và điều đó càng rõ ràng trong ngành công nghệ thông tin này.

Khi bạn tạo ra những dòng code nào đó và muốn những người khác đọc vào có thể kế thừa được ý tưởng của bạn một cách dễ dàng thì phải dùng đến chú thích. Đó là sự ra đời của ghi chú (comment)

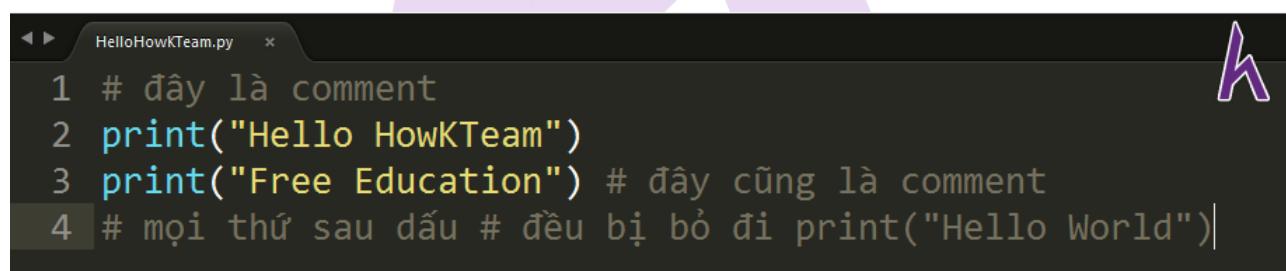
Cú pháp

Các dòng comment trong python có cú pháp

```
# <Nội dung ghi chú>
```

Trong đó

- Comment được bắt đầu với dấu **#**
- <Nội dung ghi chú>** được viết tính từ sau dấu **#** đến cuối hàng được hiểu là comment và chương trình sẽ bỏ qua trong quá trình chạy.



```
HelloHowKTeam.py
1 # đây là comment
2 print("Hello HowKTeam")
3 print("Free Education") # đây cũng là comment
4 # mọi thứ sau dấu # đều bị bỏ đi print("Hello World")
```

Lưu ý:

Thông thường, người ta không hay sử dụng comment **bên phải** dòng code. Vì nó rất không phù hợp trong những trường hợp dòng code dài. Vì vậy, **Kteam** khuyến khích các bạn nên đặt comment ở bên trên câu lệnh mà bạn cần giải thích.

Một vài trường hợp đặc biệt

Mặc dù mọi thứ ở bên phải dấu **#** trở đi đều tính là comment và được chương trình bỏ qua nhưng vẫn có một vài trường hợp ngoại lệ sau mà bạn cần lưu ý

Chuỗi có dạng là một comment

Ở trường hợp này, khi đặt **#** trong cặp dấu ngoặc kép (" "), chương trình sẽ hiểu là dấu **#** là ký tự mà chúng ta muốn in ra như một chuỗi bình thường. Do đó, nó cũng sẽ in ra dấu **#** và không bỏ qua những thứ bên phải dấu **#**. Lý do tại sao sẽ được giải thích chi tiết ở bài [CHUỖI TRONG PYTHON](#).

```
1 print("# Day la comment")
2 # trường hợp trên không phải là comment
```

Kết quả chương trình

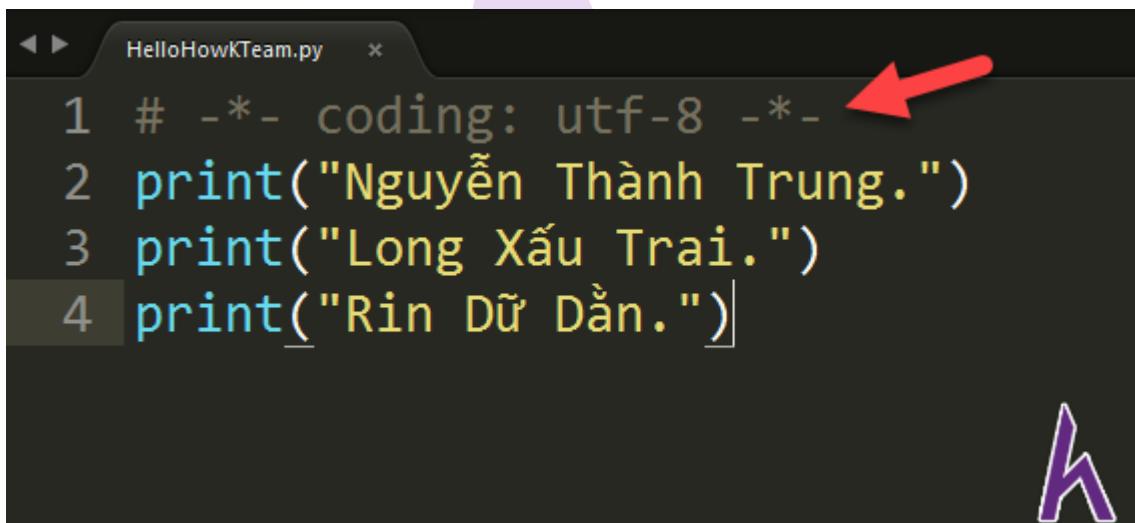
```
C:\WINDOWS\system32\cmd.exe
C:\Users\NGUYENTRUNG\Desktop>python HelloHowKteam.py
# Day la comment
```

Comment được tính là một định nghĩa:

Như trong trường hợp này là một định nghĩa về encoding UTF-8. Và đương nhiên bằng một cách nào đấy, chương trình đã “hack” và hiểu định nghĩa của chúng ta

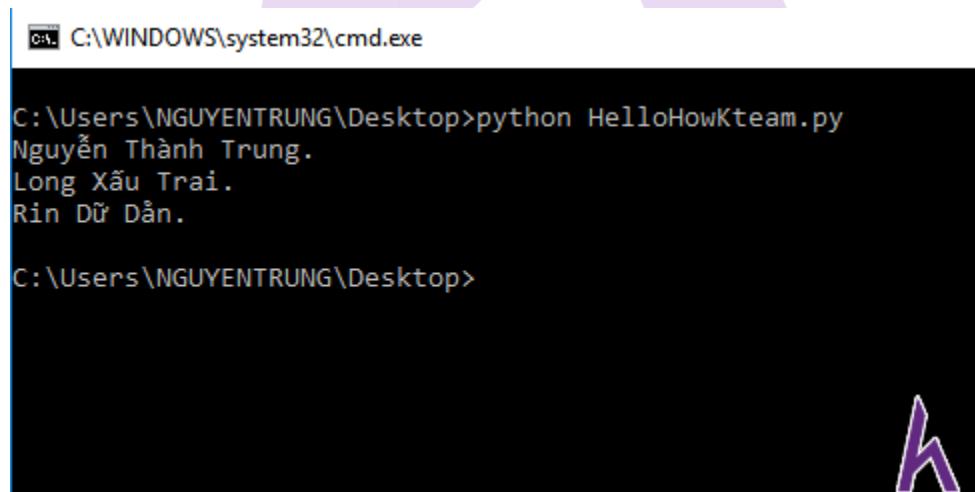
Ở ví dụ này, mình đã định nghĩa **encoding UTF-8** một encoding giúp chúng ta hiển thị tiếng Việt. Và nhờ đó chúng ta có thể hiển thị được tiếng việt trên Command Prompt

- Tuy nhiên, ở Sublime Text bạn đã được hỗ trợ để hiển thị tiếng việt.



```
>HelloHowKteam.py *  
1 # -*- coding: utf-8 -*- ←  
2 print("Nguyễn Thành Trung.")  
3 print("Long Xấu Trai.")  
4 print("Rin Dữ Dẫn.")
```

Kết quả của chương trình.



```
C:\WINDOWS\system32\cmd.exe  
C:\Users\NGUYENTRUNG\Desktop>python HelloHowKteam.py  
Nguyễn Thành Trung.  
Long Xấu Trai.  
Rin Dữ Dẫn.  
C:\Users\NGUYENTRUNG\Desktop>
```

Các bạn có thể tham khảo thêm ở đây về một số định nghĩa mã hóa khác của ngôn ngữ Python:

www.python.org/dev/peps/pep-0263/

Comment có quan trọng?

Câu trả lời là có!

Comment là một thứ cực kì quan trọng, nó giúp truyền đạt ý tưởng, nội dung, chức năng những dòng code của bạn. Khi người khác đọc vào họ có thể dễ dàng hiểu được những nội dung, ý tưởng đó.

Một mã nguồn mở, một dự án lớn đều luôn có các comment trong đó. Nhìn qua các dòng comment, chúng ta cũng có thể đánh giá được người lập trình viên viết đoạn mã nguồn này có kỹ năng code hay không.

Hãy luyện tập comment một cách có hiệu quả!

Kết Luận

Qua bài viết này, Kteam mong các bạn đã nắm được các kiến thức về GHI CHÚ TRONG PYTHON.

Ở bài viết tiếp theo, Kteam sẽ giới thiệu về các [BIẾN TRONG PYTHON](#) (Variables in Python)

Cảm ơn các bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết hơn nữa. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".

Bài 5: BIẾN TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Biến trong Python](#).

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong các bài trước, **Kteam** đã giới thiệu đến bạn cách [CHẠY MỘT CHƯƠNG TRÌNH PYTHON](#). Bên cạnh đó, chúng ta cũng đã biết [CÁCH GHI CHÚ TRONG PYTHON](#).

Ở bài này, Kteam sẽ giới thiệu với các bạn về **BIẾN TRONG PYTHON**. Một phần tuy cơ bản nhưng rất quan trọng trong lập trình và nó sẽ theo bạn mãi cho tới khi bạn còn code.

Nội dung chính

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ TRONG PYTHON](#).

Trong bài học này, chúng ta sẽ cùng tìm hiểu các vấn đề:

- Biến là gì?
- Tại sao lại cần biến?
- Khởi tạo biến trong Python.
- Cách kiểm tra kiểu dữ liệu của biến.

Biến là gì?

Nếu bạn từng làm các bài toán đại số thì các bạn luôn phải chạm mặt các biến như là biến x, biến y, biến a, biến b,... Và như bạn thấy nó chả có giá trị cụ thể.

Trong lập trình, biến (variable) là tên của một vùng trong bộ nhớ RAM, được sử dụng để lưu trữ thông tin. Bạn có thể gán thông tin cho một biến, và có thể lấy thông tin đó ra để sử dụng. Khi một biến được khai báo, một vùng trong bộ nhớ sẽ dành cho các biến.

Biến là một thứ cực kì quan trọng trong lập trình mà không thể thiếu trong bất cứ chương trình lớn, nhỏ nào.

Tại sao lại cần biến?

Biến giúp chúng ta lưu trữ các dữ liệu và cho phép chúng ta lấy các dữ liệu của chúng để tính toán được thuận tiện và chính xác hơn.

Hãy tưởng tượng như sau, bạn có một số dữ liệu là những con số với nhiều chữ số và các thao tác tính toán

```
# lưu giá trị 52348252408 cho biến a  
>>> a = 52348252408  
# lưu giá trị 523482034 cho biến b  
>>> b = 523482034  
# cộng giá trị hai biến a và b, sau đó lưu vào biến c  
>>> c = a + b  
# lưu giá trị 412312323 cho biến d  
>>> d = 412312323  
# cộng giá trị biến c với giá trị biến d  
>>> c + d  
53284046765
```

Một điều mà các bạn dễ dàng nhận ra đó là những con số với nhiều chữ số gây khó khăn trong việc sử dụng vì chúng có quá nhiều chữ số, đôi lúc chúng ta cũng có thể vô tình gây sai lệnh giá trị.

Ta hãy giải quyết bài toán trên khi nhờ tới sự giúp đỡ của các biến

```
# lưu giá trị 52348252408 cho biến a  
>>> a = 52348252408  
# lưu giá trị 523482034 cho biến b  
>>> b = 523482034  
# cộng giá trị hai biến a và b, sau đó lưu vào biến c  
>>> c = a + b  
# lưu giá trị 412312323 cho biến d  
>>> d = 412312323  
# cộng giá trị biến c với giá trị biến d  
>>> c + d  
53284046765
```

Dễ thấy, ta cũng được kết quả tương tự, nhưng lại dễ dàng tính toán, giảm thiểu tỉ lệ sai lệnh giá trị hơn khi không sử dụng tới biến.

Khởi tạo biến trong Python

Những thứ cần biết về tên của biến

- Tên của biến không được bắt đầu bằng số
- Tên biến không được trùng với các từ khóa của Python

Một số từ khóa của Python

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



- Tên của biến chỉ chứa các chữ cái, số và '_'
- Tên biến trong Python có phân biệt chữ in hoa và in thường. Ví dụ: PI, Pi, pl, pi là 4 tên biến khác nhau.

Khởi tạo một biến trong Python

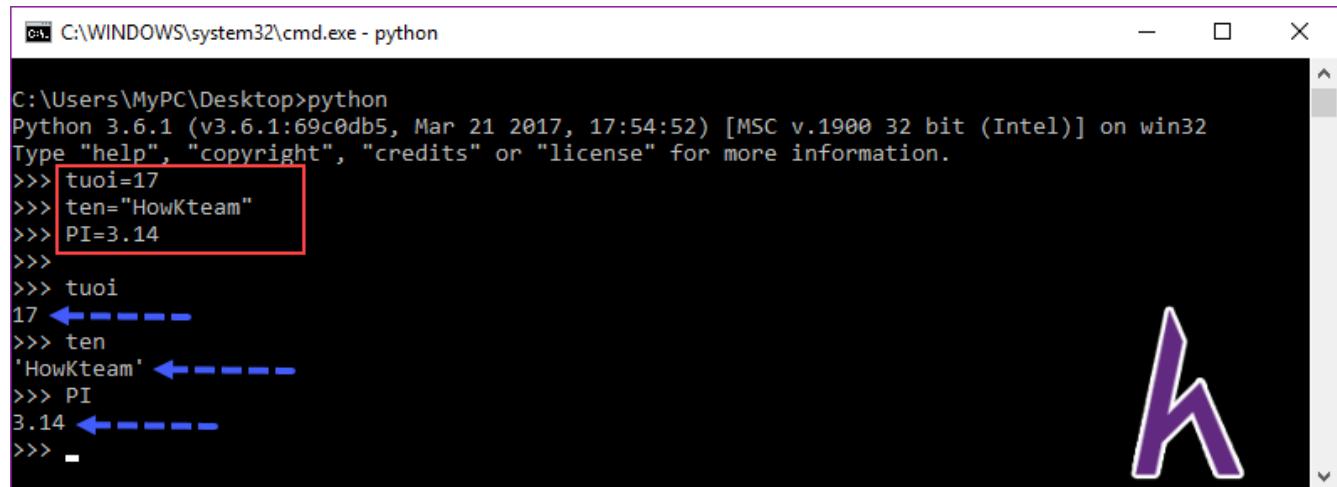
Cú pháp:

<đại lượng> = <giá trị của đại lượng>

Ví dụ:

```
# Đoạn code sau đây khai báo 3 biến tuoi, ten và PI và giá trị của chúng
>>>tuoi = 17
>>>ten = "How Kteam"
>>>PI = 3.14
```

Kết quả:



```
C:\WINDOWS\system32\cmd.exe - python
C:\Users\MyPC\Desktop>python
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> tuoi=17
>>> ten="HowKteam"
>>> PI=3.14
>>>
>>> tuoi
17
>>> ten
'HowKteam'
>>> PI
3.14
>>> -
```

Giải thích ví dụ:

Những dòng có **khung đỏ** đó chính là dòng lệnh dùng để **khai báo**. Còn dòng **mũi tên màu xanh** chính là **kết quả** của biến.

"ten", "tuoi", "PI" chính là những tên biến còn những thứ sau dấu bằng như là "17", "How Kteam", "3.14" đó chính là giá trị của biến.

Khởi tạo nhiều biến

Cú pháp:

```
<ten biến thứ nhất>, <ten biến thứ hai>, .., <ten tên biến thứ n> =  
<giá trị biến thứ nhất>, <giá trị biến thứ hai>, .., <giá trị biến thứ n>
```

Ví dụ:

```
# khai báo 3 biến tuoi, ten và PI và giá trị của chúng trên cùng một dòng  
>>> tuoi, ten, PI = 17, "How Kteam", 3.14
```

Kết quả:



```
C:\Windows\system32\cmd.exe - python
C:\Users\Admin\Desktop>python
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> tuoi, ten, PI = 17, "How KTeam", 3.14 ← khai báo
>>>
>>> tuoi ←
17 ←
>>> ten ←     kết quả
'How KTeam' ←
>>> PI ←
3.14 ←
```

Giải thích ví dụ:

Ở ví dụ trên ‘tuoi’ là biến thứ nhất, ‘ten’ là biến thứ hai và ‘PI’ là biến thứ ba. Và sau đó ‘17’ là giá trị biến thứ nhất, “How Kteam” là giá trị biến thứ hai và “3.14” là giá trị biến thứ ba. Do đó dòng code trên cũng tương tự như:

```
tuoi = 17 # biến thứ nhất lấy giá trị biến thứ nhất  
ten = "How Kteam" # biến thứ hai lấy giá trị biến thứ hai  
PI = 3.14 # biến thứ ba lấy giá trị biến thứ ba
```

Kiểm tra kiểu dữ liệu của biến

Không như đa số các ngôn ngữ lập trình khác, khi khai báo biến phải đi kèm với kiểu dữ liệu. Trong Python việc khai báo kiểu dữ liệu cho biến không cần thiết mà Python sẽ tự biết kiểu dữ liệu của giá trị gán cho biến.

Vậy để kiểm tra kiểu dữ liệu giá trị của một biến đã khởi tạo, ta sử dụng hàm **type()**

Cú pháp:

```
type(<tên biến>)
```

Ví dụ:

```
tuoi = 17  
  
ten = "How KTeam"  
  
PI = 3.14  
  
type(tuoi) # kiểm tra kiểu dữ liệu giá trị của biến tuoi  
  
type(ten) # kiểm tra kiểu dữ liệu giá trị của biến ten  
  
type(PI) # kiểm tra kiểu dữ liệu giá trị của biến PI
```

Kết quả:

```
C:\Windows\system32\cmd.exe - python
C:\Users\Admin\Desktop>python
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> tuoi = 17
>>> ten = "How KTeam"
>>> PI = 3.14
>>>
>>> type(tuoi)
<class 'int'>
>>>
>>> type(ten)
<class 'str'>
>>>
>>> type(PI)
<class 'float'>
>>>
```



Như bạn thấy ở ví dụ trên kết quả tra ra một số kiểu dữ liệu đó là 'int', 'str', 'float'. Đó là các kiểu dữ liệu phổ biến trong các ngôn ngữ lập trình hiện nay. Kteam sẽ giới thiệu những kiểu dữ liệu này ở những bài sau.

Kết luận

Qua bài viết này, bạn đã nắm được về những thứ cơ bản về BIẾN TRONG PYTHON (variable).

Các bài tiếp theo mình sẽ giới thiệu về [KIỂU DỮ LIỆU SỐ TRONG PYTHON](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".

Bài 6: KIỂU DỮ LIỆU SỐ TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu số trong Python](#).

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong các bài trước, bạn đã làm quen với khái niệm [BIẾN TRONG PYTHON](#).

Ở bài này **Kteam** sẽ đề cập đến các bạn **KIỂU DỮ LIỆU SỐ**. Một trong những kiểu dữ liệu cực kì quan trọng của Python!

Nội dung chính

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).

Trong bài học này, chúng ta sẽ cùng tìm hiểu các vấn đề:

- Số là gì?
- Một số kiểu dữ liệu số cơ bản trong Python.
- Các toán tử cơ bản với kiểu dữ liệu số trong Python.
- Thư viện math trong Python.

Số là gì?

Con số ở khắp nơi trong cuộc sống chúng ta. Bất cứ lúc nào bạn cũng có thể bắt gặp con số trong cuộc sống.

Tháng này có 30 hay 31 ngày? Mai đi chợ bó rau muốn 3000 đồng hay là 3500 đồng? Bài thi hôm bữa được 9,1 điểm hay là 1,9? Cái bánh này mình ăn $\frac{1}{2}$ hay là $\frac{3}{4}$. Có thể thấy, số không còn là điều gì xa lạ với bạn. Và đương nhiên điều này tương tự với "con trăn" Python.

Trong Python cũng hỗ trợ rất nhiều kiểu dữ liệu số. Một số kiểu dữ liệu cơ bản như số nguyên (integers), số thực (floating-point), phân số (fraction), số phức (complex). Và những kiểu dữ liệu này sẽ được Kteam giới thiệu cho các bạn ngay sau đây!

Một số kiểu dữ liệu số cơ bản trong Python

Số nguyên

Số nguyên bao gồm các số nguyên dương (1, 2, 3, ..), các số nguyên âm (-1, -2, -3) và số 0. Trong Python, kiểu dữ liệu số nguyên cũng không có gì khác biệt.

Ví dụ: Gán giá trị cho một biến a là 4 và xuất ra kiểu dữ liệu của a.

```
>>> a = 4 # gán giá giá trị của biến a là số 4, là một số nguyên
>>> a
4
>>> type(a) # số nguyên thuộc lớp 'int' trong Python
<class 'int'>
```

Một điểm đáng chú ý trong Python 3.X đó là kiểu dữ liệu số nguyên là vô hạn. Điều này cho phép bạn tính toán với những số cực kì lớn, điều mà đa số các ngôn ngữ lập trình khác **KHÔNG THỂ**.

Số thực

Về kiểu dữ liệu **số thực**, thì đây là tập hợp các số nguyên và số thập phân 1, 1.4, -123, 69.96,...

Ví dụ: Gán giá trị của biến f là 1.23 và xuất ra kiểu dữ liệu của f.

```
>>> f = 1.23 # gán giá trị của biến f là số 1.23, là một số thực
>>> f
1.23
>>> type(f) # số thực trong Python thuộc lớp 'float'
<class 'float'>
>>> q = 1.0 # đây là số thực, không phải số nguyên
>>> q
1.0
>>> type(q)
<class 'float'>
```

Lưu ý: Thường khi chúng ta viết số thực, phần nguyên và phần thập phân được tách nhau bởi dấu phẩy (,). Thế nhưng trong Python, dấu phẩy (,) này được thay thế thành dấu chấm (.)

Số thực trong Python có độ chính xác xấp xỉ 15 chữ số phần thập phân.

Ví dụ: Số thực 10/3

```
>>> 10 / 3 # đây là một số vô hạn tuần hoàn 3.3333333333333333..
3.333333333333335
```

Nếu bạn muốn có kết quả được chính xác cao hơn, bạn nên sử dụng **Decimal**

```
>>> from decimal import * # lấy toàn bộ nội dung của thư viện Decimal  
>>> getcontext().prec = 30 # lấy tối đa 30 chữ số phần nguyên và phần thập  
phân Decimal  
>>> Decimal(10) / Decimal(3)  
Decimal('3. 3333333333333333333333333333')  
>>> Decimal(100) / Decimal(3)  
Decimal('33.3333333333333333333333333333')  
>>> type(Decimal(5)) # các số Decimal thuộc lớp Decimal  
<class 'decimal.Decimal'>
```

Tuy **Decimal** có độ chính xác cao hơn so với **float** tuy nhiên nó lại khá rườm rà so với float. Do đó, hãy cân bằng sự tiện lợi và chính xác để chọn kiểu dữ liệu phù hợp.

Phân số

Chúng ta biết đến phân số qua sách giáo khoa toán lớp 3. Phân số gồm hai phần là **tử số** và **mẫu số**.

Tạo một phân số

Để tạo phân số trong python, ta sử dụng hàm **Fraction** với cú pháp sau

Fraction(<Tử_số>, <Mẫu_số>)

Ví dụ: Nhập phân số $\frac{1}{4}$, $\frac{3}{9}$, $\frac{3}{4}$,

```
>>> from fractions import * # lấy toàn bộ nội dung của thư viện decimal
>>> Fraction(1, 4) # phân số với tử số là 1, mẫu số là 4.
Fraction(1, 4)
>>> Fraction(3, 9) # phân số sẽ được tối giản nếu có thể
Fraction(1, 3)
>>> type(Fraction(3, 4)) # các phân số thuộc lớp Fraction
<class 'fractions.Fraction'>
```

Số phức

Nếu bạn chưa biết đến **Số Phức**, Kteam khuyên bạn nên bỏ qua phần này.

Số phức gồm 2 thành phần :

<Phần thực> + <Phần ảo> j

Trong đó

- **<Phần thực> <Phần ảo>** là số thực
- j là đơn vị ảo trong toán học với $j^2 = -1$

Tạo một số phức

Để tạo một số phức, bạn có thể sử dụng hàm **complex** với cú pháp sau:

complex(<Phần_thực>, <Phần_ảo>)

Gán giá trị số phức cho một biến

<tên_biến> = <Phần_thực> + <Phần_Ảo>j

Xuất ra từng phần của một biến số phức

Để xuất ra phần thực, ta sử dụng cú pháp:

```
<tên_bien>.real
```

Để xuất ra phần ảo của biến số phức, ta dùng cú pháp:

```
<tên_bien>.imag
```

Ví dụ: Nhập một số số phức sau

1. $1 + 3j$
2. Gán biến c có giá trị $2+1j$. Xuất ra phần thực và phần ảo của biến c.
3. $4 + j$ (sẽ có lỗi vì kiểu dữ liệu nhập vào không đúng).
4. Tạo số phức có phần thực là 3, phần ảo là 1.
5. Tạo số phức chỉ có phần thực là 2.
6. Xuất ra kiểu dữ liệu của số $3+1j$.

```
>>> 3j + 1 # phần thực là 1, phần ảo là 3
(1 + 3j)
>>> c = 2 + 1j # gán giá trị cho biến c là một số phức với phần thực là 2 còn
      phần ảo là 1
>>> c
(2 + 1j)
>>> 4 + j # phần ảo là 1, tuy vậy chúng ta không được phép bỏ số 1 như trong
      toán
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
>>> 4 + 1j
(4 + 1j)
>>> c.imag # lấy phần ảo của số phức  $2 + 1j$  mà ta đã gán cho biến c
1.0
>>> c.real # lấy phần thực
2.0
```

```
>>> complex(3, 1) # dùng hàm complex để tạo một số phức với phần thực là 3,  
ảo là 1  
(3 + 1j)  
>>> complex(2)    # chỉ có phần thực, phần ảo được mặc định là 0  
(2 + 0j)  
>>> type(3 + 1j) # các số phức thuộc lớp complex  
<class 'complex'>
```

Các toán tử cơ bản với kiểu dữ liệu số trong Python

Biểu thức chính là một **thực thể toán học**. Nói cách khác, nó là một sự kết hợp giữa 2 thành phần:

- **Toán hạng**: có thể là một hằng số, biến số (X , Y)
- **Toán tử**: xác định cách thức làm việc giữa các toán hạng (+,-,*,/)

Dưới đây là một số biểu thức toán học của kiểu dữ liệu số trong Python

BIỂU THỨC	MÔ TẢ
$X + Y$	Tổng của X với Y
$X - Y$	Hiệu của X với Y
$X * Y$	Tích của X với Y
X / Y	Thương của X với Y (kết quả luôn luôn là một số thực)
$X // Y$	Thương nguyên của X với Y (kết quả luôn luôn nhỏ hơn hoặc bằng X / Y)
$X \% Y$	Dư của thương X với Y
$X ** Y$	Lũy thừa mũ Y với cơ số X

Ví dụ: Cho 2 biến a,b lần lượt bằng 8 và 3. Thực hiện các biểu thức toán học với a,b.

```
>>> a = 8
>>> b = 3
>>> a + b # tương đương 8 cộng 3
11
>>> a - b # tương đương 8 trừ 3
5
>>> a * b # tương đương 8 nhân 3
24
>>> a / b # tương đương 8 chia 3
2.666666666666665
>>> a // b # tương đương với 8 chia nguyên 3
2
>>> a % b # tương đương với 8 chia dư 3
2
>>> a ** b # tương đương 8 mũ 3
512
```

Thư viện math trong Python

Thư viện math trong Python hỗ trợ rất nhiều hàm tính toán liên quan đến toán học.

Để sử dụng một thư viện nào đó, ta dùng lệnh:

```
import <tên_thư_viện>
```

Muốn sử dụng một hàm nào đó của thư viện, ta sử dụng cú pháp:

```
<tên_thư_viện>.<tên_hàm>
```

Dưới đây là một số hàm thường được dùng trong việc tính toán cơ bản.

TÊN HÀM	CÔNG DỤNG
.trunc(x)	Trả về một số nguyên là phần nguyên của số x
.floor(x)	Trả về một số nguyên được làm tròn số từ số x, kết quả luôn luôn nhỏ hơn hoặc bằng x
.ceil(x)	Trả về một số nguyên được làm tròn số từ số x, kết quả luôn luôn lớn hơn hoặc bằng x
.fabs(x)	Trả về một số thực là trị tuyệt đối của số x
.sqrt(x)	Trả về một số thực là căn bậc hai của số x
.gcd(x, y)	Trả về một số nguyên là ước chung lớn nhất của hai số x và y

Ví dụ:

```
>>> import math # lấy nội dung của thư viện math về sử dụng
>>> math.trunc(3.9)
3
>>> math.fabs(-3)
```

```
3.0  
>>> math.sqrt(16)  
4.0  
>>> math.gcd(6, 4)  
2
```

Câu hỏi củng cố

1. Kiểu dữ liệu số nguyên thuộc lớp nào?
2. Sự khác nhau giữa hai biến a và b dưới đây là gì?

```
>>> a = 0  
>>> b = 0.0
```

3. Tại sao lại có sự khác nhau khi sử dụng hàm `trunc` ở thư viện math so với toán tử `//`

```
>>> import math  
>>> math.trunc(15 / -4)  
-3  
>>> 15 // -4  
-4
```

Trong khi chúng lại có trùng kết quả ở phép tính này.

```
>>> import math  
>>> math.trunc(15 / 4)  
3  
>>> 15 // 4  
3
```

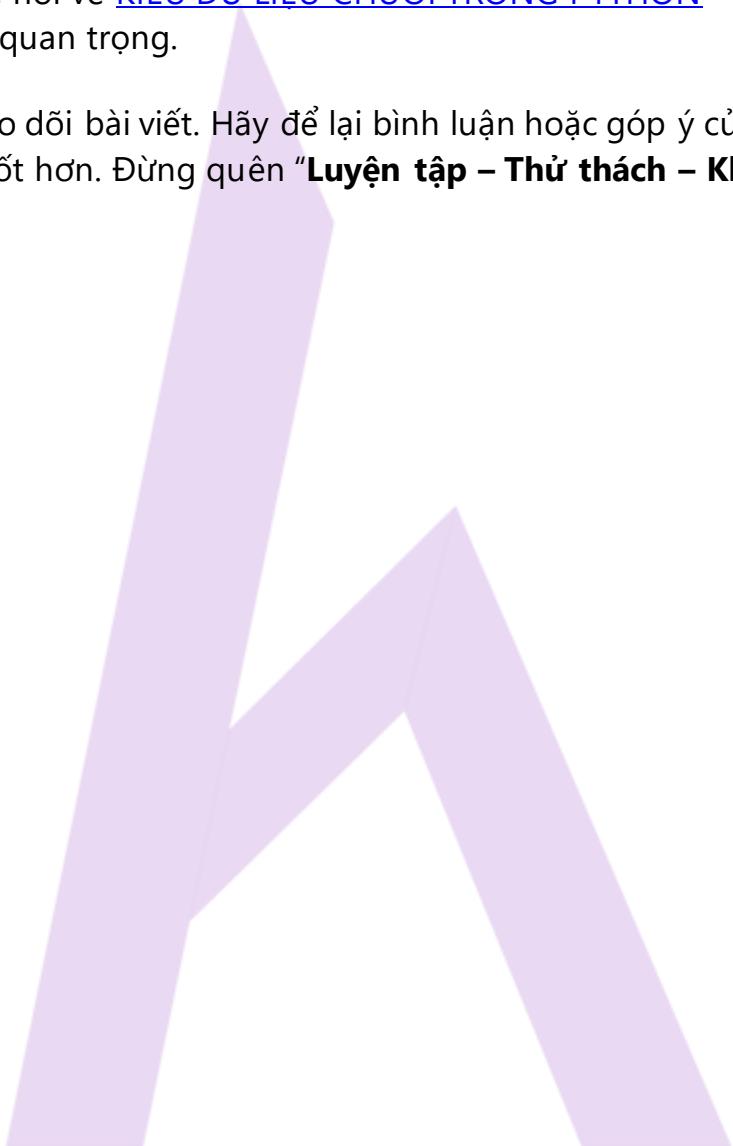
Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Bài viết này đã giới thiệu cho các bạn một số KIỂU DỮ LIỆU SỐ trong Python.

Ở bài sau, Kteam sẽ nói về [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON](#) - một kiểu dữ liệu cũng cực kì quan trọng.

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".



Bài 7: KIỂU DỮ LIỆU CHUỖI TRONG PYTHON (Phần 1)

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu chuỗi trong Python – Phần 1](#).

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong các bài trước, Kteam đã giới thiệu đến bạn **KIỂU DỮ LIỆU SỐ** trong Python

Ở bài này chúng ta sẽ bắt đầu tìm hiểu đến các **KIỂU DỮ LIỆU CHUỖI** trong Python. Một trong những kiểu dữ liệu cực kì quan trọng trong Python.

Nội dung chính

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).

Trong bài học này, chúng ta sẽ cùng tìm hiểu các vấn đề:

- Chuỗi là gì?
 - Sự khác nhau giữa '' và "".
 - Chuỗi nhiều dòng với "" và """". Khái niệm Docstring.
 - Escape Sequence là gì?
 - Câu hỏi củng cố
-

Chuỗi là gì?

Trong Python, chuỗi là những thứ được đặt trong cặp dấu '', hoặc "", có thể cũng là trong cặp """", """ """. Nhưng cơ bản và thường đường sử dụng nhất là cặp '' và """.

Ví dụ:

```
>>> 'How Kteam'  
'How Kteam'  
>>> s = 'Free Education' # gán cho biến s với giá trị là một chuỗi  
>>> s  
'Free Education'  
>>> player = "Kteam" # gán cho biến player với một chuỗi có giá trị là 'Kteam'  
>>> player  
'Kteam'  
>>> '# day la comment' # không đâu, nó là chuỗi. Đây mới là comment  
'# day la comment'  
>>> s = "String"  
>>> type(s) # và kiểu dữ liệu chuỗi sẽ thuộc lớp 'str'  
<class 'str'>
```

Sự khác nhau giữa " và ""

Nói về công dụng, thì hai cặp dấu nháy trên là tương đương. Những thứ nằm bên trong nó là một chuỗi.

Nhưng không có thứ gì sinh ra là để cho có. Hãy đặt vấn đề bạn muốn có chuỗi với nội dung sau đây và bạn muốn Python hiểu đó là một chuỗi

I'm Beginner

```
>>> 'I'm Beginner' # đặt nội dung vào trong cặp dấu ''  
File "<stdin>", line 1  
  'I'm Beginner'  
          ^  
SyntaxError: invalid syntax
```

Hãy nhìn lại và phân tích tại sao lại có lỗi xảy ra???

Khi bạn gõ 'I'm Beginner'. Python sẽ đọc từ trái qua phải và lấy từng kí tự trong chuỗi của bạn và việc này sẽ dừng lại khi nó gặp được dấu ' còn lại.

Có nghĩa là nó sẽ đọc được chuỗi 'I' sau đó kết thúc. Nhưng ta lại còn có một đoạn ở phía sau `m Beginner`. Thứ này với Python hoàn toàn vô nghĩa. Nó không hiểu được ý của bạn. Do đó một **SyntaxError** được thông báo lên.

Vậy, làm cách nào để ta có thể có được chuỗi với nội dung I'm Beginner???

Ta có 3 cách cơ bản để giải quyết vấn đề này. Và ngay sau đây, mình sẽ giới thiệu cách đơn giản nhất.

Ta sẽ lựa chọn cặp dấu ngoặc " " khi nội dung chuỗi của chúng ta có những kí tự ', và sẽ chọn cặp dấu ngoặc " nếu nội dung chuỗi của chúng ta có những kí tự ".

```
>>> "I'm Beginner" # nội dung có kí tự ', chọn cặp dấu ""  
"I'm Beginner"  
>>> s = "It's good"  
>>> s  
"It's good"  
>>> s2 = 'this is "special" word' # nội dung có kí tự ", chọn cặp "
```

```
>>> s
'this is "special" word
```

Bạn có thể có câu hỏi rằng: "Nếu nội dung trong chuỗi vừa có kí tự ", lại vừa có kí tự ', thì ta phải chọn cặp dấu ngoặc nào?".

Kteam sẽ nợ bạn câu hỏi này vào phần sau. Theo dõi phần tiếp theo sẽ có câu trả lời từ Kteam nhé!

Chuỗi nhiều dòng với "" và """. Khái niệm Docstring

Chuỗi nhiều dòng với "" và """

Thường khi nhắc đến chuỗi, ta hay nghĩ tới một dòng. Và khi đó, ta sử dụng cặp dấu '' hoặc ''. Nếu là nhiều dòng chuỗi kết nối với nhau, như những câu chữ bạn hay viết trong những cuốn vở thì đó cũng là một chuỗi, nhưng chuỗi đó sẽ được đặt trong cặp dấu "" và """.

```
>>> s = ""dong 1
...     dong 2
...     dong 3"""
>>> s
'dong 1\ndong2\ndong3'
>>> print(s) # kết quả mong muốn sẽ xuất hiện khi bạn sử dụng hàm print
dong 1
dong 2
dong 3
```

Hãy khoan nói về việc tại sao kết quả chúng ta mong muốn phải qua tay hàm **print**. Nếu để ý, những lần chúng ta nhấn phím **enter** để xuống dòng, nhập tiếp dòng tiếp theo. Ở đó sẽ được thêm vào 2 kí tự **\n** và **n**.

Sự thật, `\n` được coi là một kí tự. Và đây chính là một **escape sequence**. Để hiểu rõ nó ra sao, **Kteam** sẽ giới thiệu với các bạn ở phần tiếp theo.

Quay trở lại, chúng ta đã biết muốn có nhiều dòng chuỗi kết hợp với nhau, ta sử dụng cặp dấu `""` hoặc ``''. Và đương nhiên, những thứ đặt trong cặp dấu `''` hoặc `'''` cũng là một chuỗi. Do đó, ta cũng có thể tạo ra chuỗi chỉ một dòng và chứa những kí tự ' và " khác.

```
>>> """chuoi vua co ki tu ' va ki tu ", that vi dai"""
'chuoi vua co ki tu \' va ki tu ", that vi dai'
```

Có thể, một trong số các bạn sẽ bất ngờ với kết quả. Vì sao lại vậy nhỉ? Lại một lần nữa, vấn đề này liên quan tới escape sequence, thứ mà chúng ta sẽ tìm hiểu tiếp theo sau.

Khái niệm Docstring

Thêm một nội dung liên quan đến cặp dấu `''` và `'''` nữa Kteam muốn giới thiệu với các bạn. Hai cặp dấu này thường không được sử dụng để tạo ra một chuỗi nhiều dòng, mà dùng để làm **DOCSTRING**

Docstring là :

- Một dạng chú thích nhiều dòng.
- Hay xuất hiện ở đầu một file Python, sau một dòng định nghĩa lớp, hàm.
- Và đây cũng là một trong những chuẩn quy ước về định dạng, trình bày code Python.

```
"""
Đây là những dòng chú thích đầu file
Về việc import các thư viện, module
"""
```

```
import lungtung
import taolao
```

```
def ham_vo_dung():
    """
    Còn đây là docstring
```

```
Cho một function  
Đó là ham_vo_dung  
""  
pass
```

```
class vo_van:  
    ""  
    Class khong co gi dau  
    That day  
    ""  
    pass
```

Escape Sequence là gì?

Escape Sequence là một chuỗi (chính xác là kí tự) đặc biệt trong Python. Bắt đầu với một dấu \.

Python có rất nhiều các escape sequence. Tuy nhiên, Kteam sẽ giới thiệu một số escape sequence chúng ta hay sử dụng nhất đới với mức độ cơ bản.

Tên	Kí hiệu	Giải thích
Alert	\a	Phát ra một tiếng bíp
Backspace	\b	Đưa con trỏ về lại một khoảng trắng
Newline	\n	Đưa con trỏ xuống dòng tiếp theo
Horizontal tab	\t	In một horizontal tab
Single quote	\'	In ra kí tự '
Double quote	\"	In ra kí tự "
Blackslash	\\"	In ra kí tự \

Kteam sẽ giúp bạn hiểu hơn về những escape sequence này. Nhớ là, nó phải được qua tay hàm print và hàm này sẽ được Kteam đề cập ở bài Nhập xuất trong Python.

```
>>> print('\a') # bạn sẽ nghe thấy một tiếng bíp

>>> print('\a\a') # nhớ bật âm lượng cho máy tính của mình

>>> print('abcd\bbe') # lùi con trỏ về trước 1 space, đè lên chữ d
abce

>>> print('dong 1\ndong 2')
dong 1
dong 2
>>> print('xuong dong\ndong moi\nnthem mot dong moi')
xuong dong
dong moi
them mot dong moi
>>> print('abc\txyz')
abc      xyz
>>> print('\t\thello')
```

```
hello
>>> print('I'm Beginner')
I'm Beginner
>>> print("one thing \"special\"", that's it")
one thing "special", that's it
>>> print('Muon in dau \nay') # bạn cũng nên cẩn thận, kết quả sẽ không như
đợi đôi khi
Muon in dau
ay
>>> print('Muon in dau \\nay')
Muon in dau \nay
```

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIỂU DỮ LIỆU SỐ TRONG PYTHON](#).

1. Kiểu dữ liệu số thuộc lớp `int`
2. Biến a là số nguyên thuộc lớp `int`, còn biến b là số thực thuộc lớp `float`
3. Vì kết quả của hàm `trunc` sẽ trả về một số nguyên là phần nguyên của phép chia, còn toán tử // thì kết quả sẽ cũng là số nguyên nhưng luôn luôn nhỏ hơn hoặc bằng kết quả.
 - Ở trường hợp đầu tiên là

15 / - 4

Thì kết quả ta được

-3.75

Ta lấy phần nguyên bằng hàm `trunc` thì sẽ có kết quả là `-3`. Riêng với toán tử // sẽ làm tròn. Một là -3, hai là -4. Vì $-4 < -3.75$ do đó kết quả sẽ được là -4. Hai kết quả khác nhau

- Ở trường hợp thứ hai

15 / 4 thì kết quả sẽ là

3.75

Hàm `trunc` sẽ lấy phần nguyên là `3`. Toán tử // sẽ làm tròn. 3 hoặc là 4, mà $3 < 3.75$, do đó kết quả là 3. Hai kết quả giống nhau.

Câu hỏi củng cố

1. Những chuỗi nào sau đây là hợp lệ?

'nasdfiuqwnerp', "234a'adadf", """asd34'asdfjoaisdfadf""", "\\", "\\\\", \"

2. Sự khác nhau giữa hai biến a và b dưới đây là gì?

```
>>> a = 69  
>>> b = '69'
```

3. Chỉ ra các Escape Sequence trong những giá trị sau đây

Chuỗi 1: '35\53ni34'
Chuỗi 2: '\n'
Chuỗi 3: "VVVVVV"

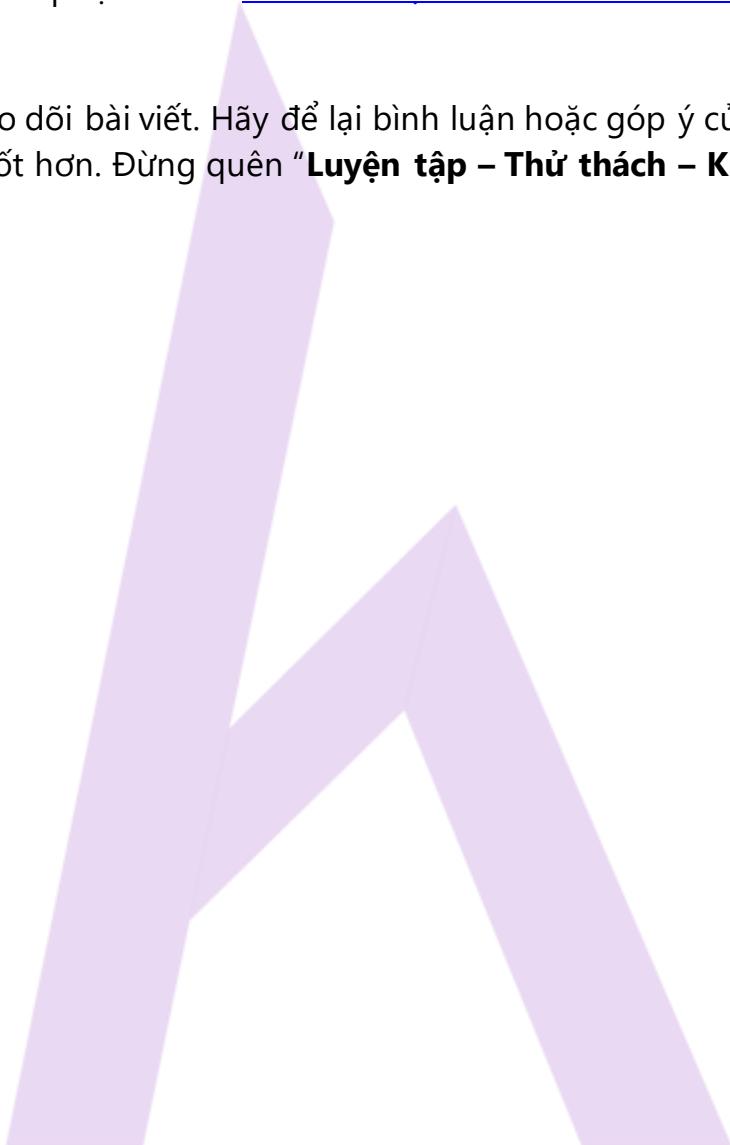
Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Bài viết này đã giới thiệu sơ cho các bạn KIỂU DỮ LIỆU CHUỖI TRONG PYTHON – Phần 1.

Ở bài sau, Kteam sẽ tiếp tục nói về [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON \(Phần 2\)](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".



Bài 8: KIỂU DỮ LIỆU CHUỖI TRONG PYTHON (Phần 2)

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu chuỗi trong Python – Phần 2](#).

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu sơ cho các bạn [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON – Phần 1](#)

Ở bài này chúng ta sẽ tiếp tục đề cập đến **KIỂU DỮ LIỆU CHUỖI** trong Python.

Nội dung chính

Để đọc hiểu bài này tốt nhất, bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU CHUỖI](#) (Phần 1)

Trong bài học này, chúng ta sẽ cùng tìm hiểu các vấn đề:

- Chuỗi trần là gì?
- Một số toán tử với chuỗi
- Indexing và cắt chuỗi
- Ép kiểu dữ liệu
- Thay đổi nội dung chuỗi

Chuỗi trần là gì?

Nếu bạn còn nhớ những ví dụ lần trước trong bài [KIẾU DỮ LIỆU CHUỖI \(Phần 1\)](#), Kteam đưa ra trong phần **Escape Sequence** là gì? Bạn dễ nhận thấy rằng, đôi khi bạn gặp trường hợp không mong muốn có escape sequen. Điển hình như ví dụ sau.

Bạn muốn in ra một dòng chuỗi với nội dung như sau:

Haizz, \neu mot nay nao do.

Và trong Python

```
>>> print('Haizz, \neu mot ngay nao do')
Haizz,
eu mot ngay nao do
```

Kết quả không mong muốn. May thay, bạn biết đó là do tác dụng của Escape Sequence. Và bạn cũng biết sử dụng Escape Sequence để có được kết quả như mình muốn

```
>>> print('Haizz, \\neu mot nay nao do')
Haizz,
\neu mot ngay nao do
```

Nhưng hãy đặt vấn đề, ví dụ như bạn thao tác với các đường dẫn file trên hệ điều hành Windows. Các đường dẫn file này sẽ có dạng

Đ\đĩa:\Thư_mục\Thư_mục

Sẽ rao sao nếu tên thư mục bắt đầu với các chữ cái t, n, a, v, b,... và kết hợp với kí tự \. Nó thành Escape Sequence, kết quả mà bạn không muốn. Và rất nhiều trường hợp khác mà việc bạn sửa Escape Sequence một cách thủ công là không chấp nhận được.

Vì lí do đó, Python cho phép bạn sử dụng một dạng chuỗi, gọi là **CHUỖI TRẦN**. Chuỗi trần này sẽ không quan tâm đến thứ gọi là Escape Sequence.

Cú pháp

r'nội dung chuỗi'

Ví dụ:

```
>>> a = r'\neu mot ngay' # chuỗi trần, bỏ qua Escape Sequence \n
>>> print(a)
'\neu mot ngay'
```

Sự thật thì, chuỗi trần không phải bỏ qua các Escape Sequence, mà nó sẽ giúp chúng ta sửa những Escape Sequence đó, như cách chúng ta làm

```
>>> a = r'\neu mot ngay'
>>> a # nội dung chuỗi trần gán vào biến a
'\\neu mot ngay'
```

Và bạn sẽ phải sử dụng chuỗi tràn này một cách thường xuyên, đặc biệt là khi bạn làm việc với BIỂU THỨC CHÍNH QUY (Regular Expression) sẽ được **Kteam** giới thiệu trong tương lai.

Một số toán tử với chuỗi

Toán tử +

Đây là một toán tử rất được hay sử dụng trong việc nối các chuỗi.

Cú pháp

A + B (với A và B là chuỗi)

Ví dụ:

```
>>> s = 'hello'  
>>> s += 'Python' # tương tự câu lệnh s = s + 'Python'  
>>> s  
'hello Python'  
>>> s2 = ', good bye'  
>>> s3 = s + s2  
>>> s3  
'hello Python, good bye'
```

Toán tử *

Không mấy ngôn ngữ lập trình hỗ trợ toán tử này, toán tử này giúp tạo ra một chuỗi nhờ lặp đi lặp lại chuỗi với số lần bạn muốn.

Cú pháp

A * N (Với A là một chuỗi, N là một số nguyên)

Ví dụ:

```
>>> 'a' * 3 # tạo ra chuỗi bằng cách lặp lại chuỗi 'a' 3 lần
'aaa'
>>> s = 'abc'
>>> s *= 2 # tương tự câu lệnh s = s * 2
>>> s
'abcabc'
>>> s * 0 # bất cứ chuỗi nào nhân với 0 cũng đều có kết quả là ""
"
>>> 'idoufhaionrewnrwnerlqwqr' * 0
"
>>> '8523nsalfnsdf' * -2 # đối với số âm cũng sẽ trả về một chuỗi ""
"
```

Toán tử in

Khi sử dụng toán tử này, bạn chỉ có thể nhận được một trong hai đáp án đó là True hoặc False.

Cú pháp:

s **in** A (Với s và A là chuỗi)

Kết quả sẽ là True nếu chuỗi s xuất hiện trong chuỗi A. Ngược lại sẽ là False.
(True và False là kiểu dữ liệu Boolean sẽ được Kteam giới thiệu trong bài [KIỂU DỮ LIỆU BOOLEAN](#) trong Python)

Ví dụ:

```
>>> 'a' in 'abc'  
True  
>>> 'ab' in 'abc'  
True  
>>> 'ac' in 'abc'  
False
```

Indexing và cắt chuỗi

Indexing

Trong một chuỗi của Python, các kí tự tạo nên chuỗi đó sẽ được đánh số từ 0 tới n – 1 từ trái qua phải với n là số kí tự có trong chuỗi.

Ví dụ: ta có một chuỗi với nội dung là 'abc xyz'. Ta sẽ mượn biến s giữ dùm ta giá trị này

```
>>> s = 'abc xyz'  
>>> s  
'abc xyz'
```

Và các kí tự trong chuỗi sẽ được đánh số như sau

a	b	c		x	y	z
0	1	2	3	4	5	6

Dựa vào đây, ta có thể lấy được bất cứ kí tự nào ta muốn trong chuỗi.

Cú pháp

<chuỗi>[vị trí]

Ví dụ:

```
>>> s = 'abc xyz'
>>> s[0]
'a'
>>> s[6]
'z'
>>> s[3]
' '
>>> s[7] # truy cập vị trí không có trong chuỗi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> s[1.2] # vị trí phải là một số nguyên, không phải số thực
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```

TypeError: string indices must be integers
>>> s['1'] # vị trí là số nguyên, không phải chuỗi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: string indices must be integers
>>> s[len(s) - 1] # truy cập phần tử cuối cùng trong trường hợp ta không biết vị
trí cuối
'z'

```

Không chỉ đánh số từ 0 tới $n - 1$ từ trái qua phải với n là độ dài chuỗi, các kí tự của chuỗi còn được đánh số từ phải qua trái từ -1 đến $-n$ với n là độ dài chuỗi.

a	b	c		x	y	z
<hr/>						
-7	-6	-5	-4	-3	-2	-1



Và cũng như trên, ta có thể truy cập bất cứ kí tự nào trong chuỗi bằng những vị trí này

```

>>> s = 'abc xyz'
>>> s[-5]
'c'
>>> s[-7]
'a'
>>> s[-1] # đơn giản hơn so với s[len(s) - 1]
'z'

```

Cắt chuỗi

Đây là một thứ lợi hại của Python. Dựa trên Indexing, Python cho phép chúng ta cắt chuỗi. Đương nhiên, các bạn cần nắm rõ được phương pháp Indexing.

Cú pháp

```
<chuỗi>[vị trí bắt đầu : vị trí dừng]
```

Khi ta sử dụng cú pháp này, ta sẽ nhận được một chuỗi. Chuỗi này chính là bản sao của chuỗi mà chúng ta muốn cắt. Và chúng ta sẽ cắt (lấy) từng các kí tự có vị trí từ **vị trí bắt đầu** đến **vị trí dừng** - 1 và từ trái sang phải.

Ví dụ:

```
>>> s = 'abc xyz'  
>>> s[1:5] # cắt từng kí tự có vị trí từ 1 đến 4  
'bc x'  
>>> s[0:3] # cắt từng kí tự có vị trí từ 0 đến 2  
'abc'
```

Như đã giới thiệu, mỗi kí tự được đánh 2 số vị trí, và lẽ dĩ nhiên, ta có thể sử dụng cả ví trí số âm.

```
>>> s = 'abc xyz'  
>>> s[-4: -1] # cắt từng kí tự có vị trí từ -4 đến -2  
' xy'  
>>> s[1: -1] # cắt từng kí tự có vị trí từ 1(-6) đến 5(-2) vì vị trí dừng là 6(-1)  
'bc xy'
```

Ở trường hợp ta sử dụng vị trí **vừa số âm vừa số dương**, bạn phải hiểu rằng, vị trí số âm hay số dương thì nó cũng sẽ chỉ ra một vị trí và vị trí đó nó sẽ xem xét để cắt. Thế nên, bạn phải nắm rõ được phần INDEXING.

Thêm một vấn đề nữa. Bạn sẽ thấy ta không có cách nào cắt mà lấy được giá trị cuối cùng của chuỗi. Lúc đó, ta sẽ sử dụng vị trí **None**. Một vị trí đặc biệt

```
>>> s = 'abc xyz'
>>> s[1:None] # lấy các kí tự có vị trí 1 đến hết chuỗi
'bc xyz'
>>> s[3:None] # lấy các kí tự có vị trí 3 đến hết chuỗi
' xyz'
>>> s[1:] # đặc biệt, ta chỉ cần bỏ trống, Python sẽ tự hiểu là None
'bc xyz'
>>> s[-3:]
'xyz'
```

Đó là bạn đặt None ở vị trí dừng. **Sẽ ra sao nếu bạn đặt None ở vị trí bắt đầu?**

Câu trả lời khi ta đặt None ở vị trí bắt đầu thì ta sẽ cắt chuỗi từ vị trí đầu tiên.

```
>>> s = 'abc xyz'
>>> s[None: 4] # lấy các kí tự có vị trí từ 0 đến 3
'abc '
>>> s[:-1] # ta cũng có thể để trống, Python sẽ tự hiểu là None
'abc xy'
>>> s[:] # một cách sao chép chuỗi
'abc xyz'
```

Lưu ý: Khi bạn đã đặt None ở vị trí bắt đầu, có nghĩa vị trí bắt đầu là 0, và khi đặt None ở vị trí kết thúc, thì có nghĩa là vị trí kết thúc sẽ là n với n là số kí tự trong chuỗi.

Như đã đề cập ở trên, việc cắt chuỗi này sẽ được cắt từ trái qua phải. **Vậy nếu muốn cắt từ phải qua trái thì sao?**

Vì phát sinh vấn đề đó, Python đã hỗ trợ chúng ta một cú pháp cắt khác.

Cú pháp

<chuỗi>[**vị trí bắt đầu : vị trí dừng : bước**]

Với cú pháp đầu tiên, thì bạn không cần phải nhập số bước và số bước này sẽ được đặt là 1. Có nghĩa là vị trí của kí tự tiếp theo hơn vị trí của kí tự kế tiếp 1 đơn vị (tính theo vị trí số dương).

```
>>> s = 'abc xyz'  
>>> s[2: 5: 1] # ta có bước bằng 1  
'c x'  
>>> s[2:5] # bước mặc định là 1  
'c x'
```

Hãy nhớ rằng, bước chính là thứ để tính được vị trí tiếp theo cách vị trí trước đó bao nhiêu đơn vị.

```
>>> s = 'abc xyz'  
>>> s[1: 7: 2] # bước là 2  
'b y'
```

Ở ví dụ trên, với bước là 2, ta sẽ có các vị trí trong khoảng 1 đến 6 đó là 1, 3, 5. Vì thế các kí tự ở vị trí này sẽ là kết quả của phép cắt trên.

Ta có thể điều chỉnh việc cắt từ trái sang phải thành phải sang trái bằng việc đặt bước là số âm.

```
>>> s = 'abc xyz'  
>>> s[1: 3] # bắt đầu ở 1 và dừng ở 3. Các vị trí lấy là 1 và 2  
'bc'  
>>> s[3:1:-1] # bắt đầu ở 3 và dừng ở 1. Các vị trí lấy là 3, 2  
' c'
```

Một lưu ý nhỏ khi các bạn đặt bước là một số âm, thì vị trí bắt đầu mà bạn để là None thì nó sẽ được đặt là $n - 1(-1)$ với n là độ dài chuỗi. Còn với vị trí dừng thì sẽ là cắt hết trọn vẹn tới đầu chuỗi có nghĩa là vị trí dừng ở trường hợp này không phải là 0.

```
>>> s = 'abc xyz'
>>> s[4::-1] # lấy các kí tự có vị trí từ 4 đến 0
'x cba'
>>> s[::-1] # một cách lấy chuỗi ngược nhờ có bước âm.
'zyx cba'
```

Lưu ý: bạn không được phép đặt bước bằng 0

```
>>> s = 'abc xyz'
>>> s[:0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: slice step cannot be zero
```

Ép kiểu dữ liệu

Như bạn đã biết. Hai biến `a` và `b` dưới đây là khác nhau

```
>>> a = '69'
>>> b = 69
>>> type(a) # biến a thuộc lớp 'str', kiểu dữ liệu chuỗi
<class 'str'>
>>> type(b) # biến b thuộc lớp 'int', kiểu dữ liệu số nguyên
<class 'int'>
```

Vì lí do đó, bạn sẽ nhận ra được vì sao có sự khác biệt trong hai biểu thức sau đây.

```
>>> '69' * 2 # một chuỗi nhân với một số
```

```
'6969'  
>>> 69 * 2 # một số nhân với một số  
138
```

Đôi lúc, bạn sẽ nhận được một số dưới dạng một chuỗi. Thế nên, trong trường hợp bạn muốn tính toán. Bạn phải đưa nó về từ kiểu dữ liệu chuỗi sang kiểu dữ liệu số. Ở trường hợp ví dụ ở đây là số nguyên.

Một hàm lí tưởng để làm việc đó chính là hàm **int()**

Cú pháp:

int(<giá trị>)

Ví dụ:

```
>>> a = '69'  
>>> int(a) # trả về giá trị được chuyển về số nguyên từ giá trị của biến a  
69  
>>> type(a) # biến a thuộc lớp 'str'  
<class 'str'>  
>>> b = int(a) # biến b giữ giá trị được chuyển sang số nguyên từ giá trị của biến a  
>>> type(b)  
<class 'int'>  
>>> c = '-3' # biến c giữ chuỗi với giá trị '-3'  
>>> type(c) # và dĩ nhiên giá trị biến c thuộc lớp 'str'  
<class 'str'>  
>>> d = int(c) # trả về giá trị được chuyển sang số nguyên từ giá trị của biến c  
>>> d  
-3  
>>> type(d) # số nguyên, thuộc lớp 'int'  
<class 'int'>
```

Đó là số nguyên, còn với số thực, xin bạn hãy lưu ý cho điều này. Khi sử dụng hàm int(). Ta có khả năng biến đổi một **số thực thành số nguyên** bằng cách bỏ đi phần thập phân.

Ví dụ:

```
>>> a = 3.1 # a là một biến giữ giá trị số thực
>>> type(a)
<class 'float'>
>>> b = int(a) # trả về một giá trị được chuyển đổi thành số nguyên từ giá trị của
biến a
>>> b
3
>>> type(b)
<class 'int'>
>>> int(3.9) # bỏ đi phần thập phân, không phải làm tròn, các bạn lưu ý
3
```

Lưu ý:

Bạn sẽ **không thể** chuyển đổi một số thực dưới dạng chuỗi bằng hàm int

Ví dụ:

```
>>> int('3.1')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.1'
```

Đương nhiên ta có giải pháp thay thế. Đó là hàm **float()**.

Cú pháp

float(<giá trị>)

Ví dụ:

```
>>> a = '3.1'  
>>> type(a)  
<class 'str'>  
>>> b = float(a)  
>>> b  
3.1  
>>> type(b)  
<class 'float'>
```

Thay đổi nội dung chuỗi

Trở về với khái niệm Indexing. Bạn có nghĩ tới việc thay đổi nội dung chuỗi nhờ Indexing không? Nếu như bạn đã từng học với các ngôn ngữ như Pascal, C, C++ thì có thể bạn sẽ sử dụng phương pháp Indexing.

Nhưng điều đáng buồn, Python **không** cho phép điều đó

```
>>> s = 'abc xyz'  
>>> s[0]  
'a'  
>>> s[0] = 'k'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'str' object does not support item assignment
```

Bạn chỉ có thể thay thế nó một cách gián tiếp giá trị chuỗi mà biến của bạn lưu giữ bằng cách sử dụng việc cắt chuỗi và toán tử + để tạo ra một chuỗi mới và gán lại vào biến của bạn.

Ví dụ:

```
>>> s = 'abc xyz'  
>>> s = 'k' + s[1:] # lấy các kí tự từ vị trí 1 đến hết chuỗi  
>>> s
```

```
'kbc xyz'
```

Vì ta không thể thay thế nội dung chuỗi, do đó kiểu dữ liệu chuỗi là một đối tượng có thể băm (hashable object).

Vì nó là một hashable object. Nên ta có thể sử dụng hàm **hash**.

```
>>> hash('abc')  
-720462249
```

Lưu ý: Khi chạy một chương trình Python, thì giá trị của hàm hash lên một giá trị nhất định không thay đổi. Nhưng giá trị đó sẽ thay đổi nếu như đó là lần chạy tiếp theo. Do đó, kết quả của bạn có thể sẽ khác so với Kteam. Và khi bạn chạy chương trình lần tiếp theo cũng sẽ nhận được kết quả khác so với kết quả ban đầu của bạn.

Bạn có thể hiểu nôm na hashable object là hằng, một giá trị không bao giờ thay đổi. Và có một vài trường hợp bắt buộc bạn phải sử dụng kiểu dữ liệu là hashable object điển hình như khóa (key) trong kiểu dữ liệu Dict của Python (một kiểu dữ liệu sẽ được Kteam giới thiệu ở bài [DICTIONARY TRONG PYTHON](#)).

Hashable object đôi lúc cũng có thể gọi là immutable object.

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON - Phần 1](#).

1. Các chuỗi hợp lệ là

```
'nasdfiuqwnerp', "234a'adadf", "
```

2.

```
>>> type(a) # biến a thuộc lớp int vì là một số  
<class 'int'>  
>>> type(b) # biến b thuộc lớp str vì là một chuỗi  
<class 'str'>
```

3. Các Escape Sequence là

Chuỗi 1: không có

Chuỗi 2: \\`

Chuỗi 3: \\`

Câu hỏi củng cố

1. Có bao nhiêu escape sequence trong giá trị của biến s dưới đây?

```
>>> s = r'\gte\teng\n\vz\adf\t'
```

2. Giá trị của biến s sau khi thực hiện toán tử + dưới đây là gì?

```
>>> s = " " + " " + " " + " " + " " + " " + " " + " "
```

3. Cho biến s với giá trị chuỗi

```
>>> s = 'abc xyz'
```

Phép cắt chuỗi nào dưới đây sẽ nhận được kết quả là một chuỗi rỗng

- a. s[:]
- b. s[len(s):]
- c. s[1:1]
- d. s[0::-1]
- e. s[0:0:-1]

Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Bài viết này đã giới thiệu thêm cho các bạn KIỂU DỮ LIỆU CHUỖI TRONG PYTHON.

Ở bài sau, Kteam sẽ tiếp tục nói về [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON \(Phần 3\)](#) về phần định dạng và một số phương thức của chuỗi

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”

Bài 9: KIỂU DỮ LIỆU CHUỖI TRONG PYTHON (Phần 3)

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu chuỗi trong Python – Phần 3](#).

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, **Kteam** đã giới thiệu thêm cho các bạn [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON – P2](#)

Ở bài này Kteam lại tiếp tục nói đến **KIỂU DỮ LIỆU CHUỖI** trong Python. Cụ thể là vấn đề **ĐỊNH DẠNG CHUỖI**

Nội dung chính

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU SỐ](#) và [KIỂU DỮ LIỆU CHUỖI](#) trong Python.

Trong bài này, chúng ta sẽ cùng tìm hiểu những nội dung sau đây:

- Giới thiệu về định dạng chuỗi trong Python
 - Định dạng bằng toán tử %
 - Định dạng bằng chuỗi f (f-string)
 - Định dạng bằng phương thức format
 - Câu hỏi củng cố
-

Giới thiệu về định dạng chuỗi trong Python

Với Python, có rất nhiều cách **Định dạng chuỗi**, và nó vô cùng tuyệt vời. Và ở phần này, Kteam xin được giới thiệu với các bạn **ba kiểu định dạng cơ bản** và được sử dụng nhiều nhất trong việc định dạng chuỗi.

Định dạng bằng toán tử %

Kiểu định dạng này sẽ là rất quen thuộc nếu bạn từng tiếp xúc với ngôn ngữ lập trình C. Hãy đến với một số ví dụ

```
>>> 'My name is %s.' %(‘Lucario’)
‘My name is Lucario’
>>> ‘%d. That is %s problem.’ %(1, ‘That’)
‘1. That is the problem.’
```

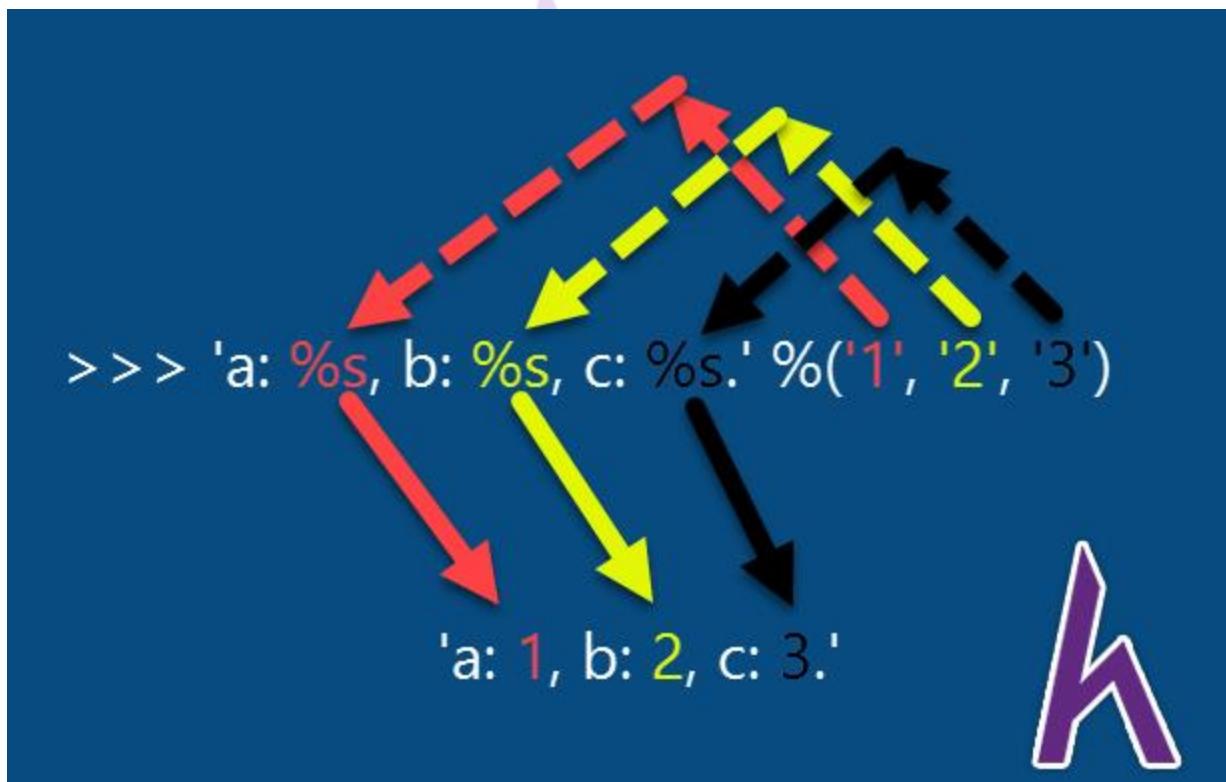
Cú pháp:

<chuỗi> **%** (giá trị thứ 1, giá trị thứ 2, .., giá trị thứ n – 1, giá trị thứ n)

Lưu ý:

Không hề có dấu ` ` tách phần chuỗi và phần giá trị cần định dạng

Để hiểu rõ hơn cách hoạt động của cách định dạng này, mời các bạn xem hình sau



Với hình vẽ trên, bạn có thể dễ dàng biết được cách mà nó hoạt động. Đó là từng phần kí hiệu **%s** sẽ lần lượt được **thay thế** lần lượt bởi các **giá trị nằm trong cặp dấu ngoặc đơn** (Đây là kiểu dữ liệu Tuple, sẽ được Kteam giới thiệu ở bài [KIẾU DỮ LIỆU TUPLE](#)).

Thêm một số ví dụ minh họa

```
>>> s = '%s %s'
>>> s %('one', 'two')
'one two'
>>> s %('a', 'b')
'a b'
>>> c = s %('c', 'cc')
>>> c
```

```
'c cc'
>>> s %('D') # không được, vì trong chuỗi của biến d có dư kí hiệu % để thay thế
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not enough arguments for format string
>>> d %('a', 'b') # không thể, vì trong chuỗi của biến d không có đủ kí hiệu % để
thay thế
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not all arguments converted during string formatting
```

Nếu các bạn để ý trong các ví dụ. Kteam không chỉ sử dụng mỗi kí hiệu **%s**, mà còn có **%d**. **Vậy sự khác nhau giữa %s và %d là gì? Liệu có còn kí hiệu % nào khác nữa không?**

Kteam sẽ giải đáp cho bạn ngay sau đây

Dưới đây là một số các toán tử % cơ bản trong Python

Toán tử	Giải thích
%s	Giá trị của phương thức <code>_str_</code> của đối tượng đó
%r	Giá trị của phương thức <code>_repr_</code> của đối tượng đó
%d	Giá trị của một số - Nếu là số thực thì sẽ chỉ lấy phần nguyên (chuyển sang số nguyên)
%<số chữ số phần thập phân>f	Giá trị của một số - Nếu là số sẽ được chuyển sang số thực

Có thể bạn sẽ cảm thấy khó hiểu ở hai toán tử **%s** và **%r**. Mọi thứ trong Python đều là các đối tượng của một lớp nào đó. Do đó nó đều có các phương thức, thuộc tính riêng. Các đối tượng trong Python luôn luôn có hai phương thức đó là `_str_` và `_repr_`.

Tuy các bạn chưa tiếp xúc với hướng đối tượng bao giờ để hiểu được khái niệm này. Nhưng Kteam sẽ viết một lớp đơn giản để giải thích cho bạn hiểu sự khác biệt giữa **%r** và **%s**.

```
>>> class SomeThing:  
...     def __repr__(self):  
...         return 'Đây là __repr__'  
...     def __str__(self):  
...         return 'Đây là __str__'  
...  
>>>
```

Vừa rồi, mình đã tạo một lớp với tên là SomeThing, giờ mình sẽ tạo một đối tượng thuộc lớp đó

```
>>> sthing = SomeThing()
```

Đừng vội bối rối! thật ra nó cũng là một giá trị bình thường thôi. Cũng giống như một chuỗi, một con số.

```
>>> type(sthing) # và nó thuộc lớp SomeThing  
<class '__main__.SomeThing'>
```

Và giờ, hãy xem giá trị của đối tượng sthing nhé.

```
>>> sthing  
Đây là __repr__  
>>> print(sthing)  
Đây là __str__
```

Nó có sự khác biệt. Và giờ, ta sẽ thấy **sự khác biệt giữa %s và %r**

```
>>> '%r' %(sthing)  
'Đây là __repr__'  
>>> '%s' %(sthing)  
'Đây là __str__'
```

Đó là sự khác biệt giữa `%s` và `%r`. Đây là một thứ mà nhiều bạn học Python nhầm lẫn.

Nếu bạn từng học **ngôn ngữ C** thì ngỡ `%s` là thay thế cho một chuỗi thì chưa đủ chính xác.

- `%s` thay thế cho giá trị của phương thức `_str_` tạo nên đối tượng đó.
- Còn về `%r` thì là phương thức `_repr_`.

Do đó, bạn có thể sử dụng `%s` hoặc `%r` với mọi đối tượng trong Python.

```
>>> '%s' %(1) # số
'1'
>>> '%r' %(1)
'1'
>>> '%s' %([1, 2, 3]) # kiểu dữ liệu list
'[1, 2, 3]'
>>> '%r' %([1, 2, 3])
'[1, 2, 3]'
>>> '%s' %((1, 2, 3)) # kiểu dữ liệu tuple
'(1, 2, 3)'
>>> '%r' %((1, 2, 3))
'(1, 2, 3)'
```

Ở kí hiệu `%d`, nó đơn giản dễ hiểu hơn với hai kí hiệu ta vừa biết qua ở trên. Kí hiệu này chỉ thay thế cho một số.

```
>>> '%d' %(3)
'3'
>>> '%d' %('3') # lỗi, vì '3' không phải 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %d format: a number is required, not str
>>> '%d' %(3.9) # chỉ lấy phần nguyên
'3'
>>> '%d' %(10/3)
'3'
```

Như bạn thấy, `%d` không phù hợp cho số thực, đó là lí do ta có `%f`

```
>>> '%f' %(3.9)
'3.900000'
>>> '%f' %('a') # %f cũng yêu cầu một số, ngoài ra đều là lỗi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be real number, not str
>>> '%f' %(3)
'3.000000'
>>> '%.2f' %(3.563545) # chỉ lấy 2 số ở phần thập phân
'3.56'
>>> '%.3f' %(3.9999) # %f cũng có khả năng làm tròn
'4.000'
```

Định dạng bằng chuỗi f (f-string)

Phương pháp định dạng này cho bạn khả năng thay thế một số chỗ ở trong một chuỗi bằng giá trị của các biến mà bạn đã khởi tạo và có. Và để có thể sử dụng cách này, bạn phải có một chuỗi f.

Một chuỗi f sẽ có cú pháp:

f 'giá trị trong chuỗi'

Ví dụ:

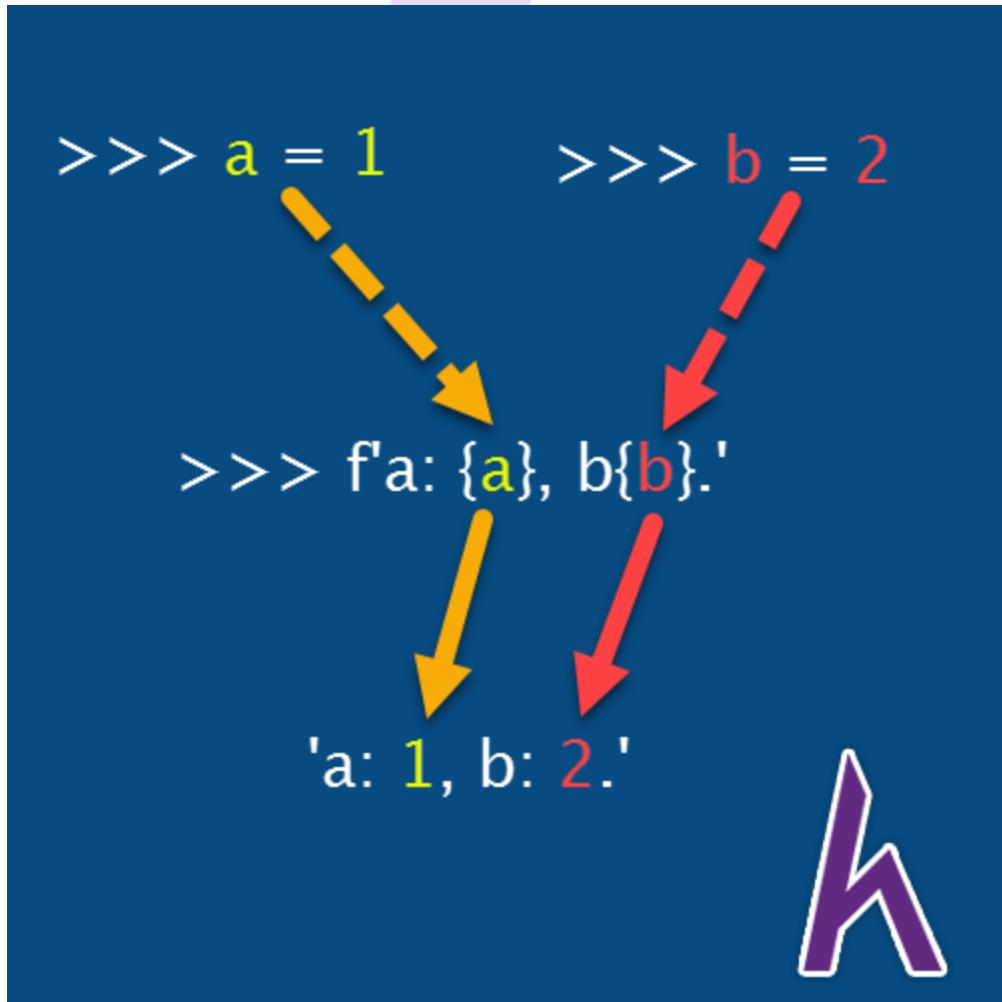
```
>>> f'abc' # đây là một f-string
'abc'
>>> s = f'xyz' # vẫn chưa có gì khác biệt so với chuỗi thông thường
>>> s
'xyz'
>>> print(f'a\tb')
a    4
```

Nhưng nó sẽ khác biệt, nếu bạn có một f-string theo kiểu này

```
>>> variable = 'string'  
>>> f'This is a {variable}.' # chú ý tới những thứ nằm trong cặp ngoặc nhọn  
'This is a string'
```

Đúng rồi đấy, giá trị của biến variable được thay thế trong cặp dấu ngoặc nhọn chứa tên của nó. Nếu bạn có biết qua PHP, bạn sẽ thấy cách này tương tự với việc bạn sử dụng cặp dấu "" để định dạng.

Mời các bạn xem hình ảnh minh họa sau đây



Vậy, **khi bạn sử dụng chuỗi f, đặt một giá trị biến chưa được khai báo, hoặc có trong chương trình thì sao?**

```
>>> f'{variable_2}' # chưa khởi tạo biến có tên variable_2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable_2' is not defined
```

Điều này đặt ra cho bạn một vấn đề, nếu như bạn muốn có chuỗi với nội dung như sau

1: {one}, 2: {two}, 3: {variable}

Và chỉ muốn định dạng mỗi chỗ **{variable}** thôi thì phải làm sao?

Cách giải quyết là hãy đặt thêm một dấu { kế bên {, còn với } là một dấu }. Tương tự như cách chúng ta muốn có một dấu \ mà để Python hiểu không phải là một kí tự bắt đầu kí tự escape sequence thì sẽ thêm một dấu \.

```
>>> variable = 'three'
>>> f'1: {{one}}, 2: {{two}}, 3: {variable}'
'1: {one}, 2: {two}, 3: three'
```

Định dạng bằng phương thức format

Cách định dạng này cho phép Python định dạng chuỗi một cách tuyệt vời, không chỉ tốt về mặt nội dung mà còn về thẩm mỹ. Định dạng bằng phương thức format

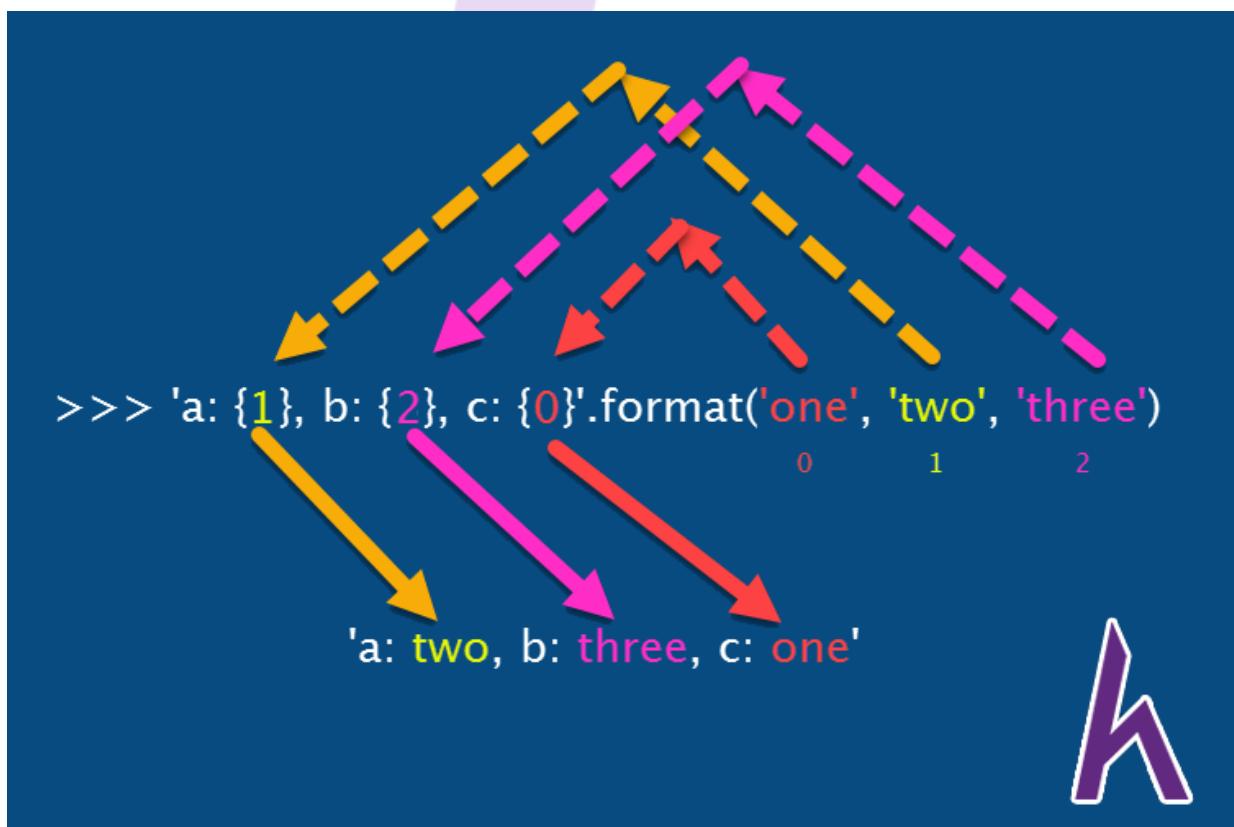
Đầu tiên là đơn giản nhất

```
>>> 'a: {}, b: {}, c: {}'.format(1, 2, 3)
'a: 1, b: 2, c: 3'
>>> 'a: %d, b: %d, c: %d' %(1, 2, 3) # tương tự như dùng phương thức format trên
'a: 1, b: 2, c: 3'
```

Nếu chỉ tương tự với toán tử **%**, phương thức này sẽ không có gì nổi bật. Vậy hãy đến với ví dụ tiếp theo

```
>>> 'a: {1}, b: {2}, c: {0}'.format('one', 'two', 'three')
'a: two, b: three, c: one'
```

Nếu vẫn còn mơ hồ, bạn hãy xem hình ảnh minh họa sau đây



Giá trị ở vị trí thứ nhất sẽ thay thế cho **vị trí thứ nhất** ở trong chuỗi, và cứ thế với các giá trị sau.

Và với phương thức này, cũng không quá khắt khe việc số các giá trị bằng số các nơi cần định dạng trong chuỗi. Ta có thể cho dư giá trị

```
>>> 'only one value: {0}'.format(1, 2)
'only one value: 1'
>>> 'only one value: {1}'.format(1, 2)
'only one value: 2'
>>> 'two same value: {0}, {0}'.format(1, 2) # và cũng có thể lặp lại
'two same value: 1, 1'
```

Vẫn chưa thỏa mãn, vì các vị trí đánh số còn chưa đủ rõ ràng, và bạn có khả năng bị nhầm lẫn. Phương thức format vẫn chiều lòng được bạn.

```
>>> '1: {one}, 2: {two}'.format(one=111, two=222)
'1: 111, 2: 222'
```

Như đã nói, không chỉ định dạng về nội dung, mà nó còn giúp tăng tính thẩm mỹ. Cụ thể là phương thức này giúp bạn định dạng căn lề một cách tuyệt vời. Cách này khá tương tự với việc sử dụng f-string đúng không nào?

Dưới đây là 3 cách căn lề cơ bản của phương thức format

Căn lề trái	<code>{:(c)<n}</code>
Căn lề phải	<code>{:(c)>n}</code>
Căn giữa	<code>{:(c)^n}</code>

Trong đó

- **c** là kí tự bạn muốn thay thế vào chỗ trống, nếu để trống thì sẽ là kí tự khoảng trắng
- **n** là số kí tự dùng để căn lề.

Để hiểu rõ hơn hãy đến với ví dụ:

```
>>> '{:^10}'.format('aaaa') # căn giữa
'    aaaa'
>>> '{:<10}'.format('aaaa') # căn lề trái
'aaaa    '
>>> '{:>10}'.format('aaaa') # căn lề phải
'    aaaa'
>>> '{*>10}'.format('aaaa') # căn lề trái, thay thế khoảng trắng bằng kí tự *
'*****aaaa'
>>> '{*<10}'.format('aaaa') # căn lề phải, thay thế khoảng trắng bằng kí tự *
'aaaa*****'
>>> '{*^10}'.format('aaaa') # căn giữa, thay thế khoảng trắng bằng kí tự *
'***aaaa***'
```

Nhờ việc căn lề bằng phương thức này, bạn sẽ dễ dàng hơn để có thể cho kết quả của bạn đẹp mắt.

Ví dụ*: Hãy tạo một file Python với nội dung sau.

```
# phần định dạng
row_1 = '+{:<6}+{:^15}+{:>10}+'.format(' ', ' ', '')
row_2 = '|{:<6}|{:^15}|{:>10}|'.format('ID', 'Ho va ten', 'Noi sinh')
row_3 = '|{:<6}|{:^15}|{:>10}|'.format('123', 'Yui Hatano', 'Japanese')
row_4 = '|{:<6}|{:^15}|{:>10}|'.format('6969', 'Sunny Leone', 'Canada')
row_5 = '+{:<6}+{:^15}+{:>10}+'.format(' ', ' ', '')

# phần xuất kết quả
print(row_1)
print(row_2)
print(row_3)
print(row_4)
print(row_5)
```

Khi chạy file Python đó, bạn sẽ có kết quả là

ID	Ho va ten	Noi sinh
123	Yui Hatano	Japanese
6969	Sunny Leone	Canada



Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON – Phần 2](#).

1. Có tổng cộng 6 escape sequence. Và 6 escape sequence này là `\\`

```
>>> s = r'\gte\teng\n\vz\adf\t'
>>> s
'\\gte\\teng\\n\\vz\\adf\\t'
```

2. Giá trị là chuỗi rỗng (````)
3. Các phép cắt có kết quả là một chuỗi rỗng là b, c, e

Câu hỏi củng cố

Bằng kiến thức về kiểu dữ liệu chuỗi của bạn. Hãy làm gọn code ở **ví dụ *** hết mức có thể.

Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Sau khi kết thúc bài viết này, bạn đã phần nào biết đến việc ĐỊNH DẠNG CHUỖI TRONG PYTHON. Và nhờ đó có thể tự định dạng nội dung của mình một cách đẹp nhất.

Ở bài viết sau, Kteam sẽ giới thiệu cho bạn về [CÁC PHƯƠNG THỨC CỦA KIỂU DỮ LIỆU CHUỖI](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.



Bài 10: KIỂU DỮ LIỆU CHUỖI TRONG PYTHON (Phần 4 – Các phương thức chuỗi)

Xem bài học trên website để ủng hộ Kteam: [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON \(Phần 4 – Các phương thức chuỗi\)](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu thêm cho các bạn về [ĐỊNH DẠNG CHUỖI TRONG PYTHON](#).

Ở bài này, chúng ta sẽ nói đến **KIỂU DỮ LIỆU CHUỖI** trong Python và nội dung chính là các phương thức của kiểu dữ liệu chuỗi.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU SỐ](#) và [KIỂU DỮ LIỆU CHUỖI](#) trong Python.

Trong bài học này, chúng ta sẽ cùng tìm hiểu các vấn đề:

- Giới thiệu về phương thức của kiểu dữ liệu chuỗi trong Python
 - Các phương thức biến đổi
 - Các phương thức định dạng
 - Các phương thức xử lí
-

Giới thiệu về phương thức của kiểu dữ liệu chuỗi trong Python

Kiểu dữ liệu của Python có khá nhiều các phương thức chuẩn (chưa tính đến các thư viện) để xử lí chuỗi.

Kteam sẽ giới thiệu với các bạn các phương thức cơ bản thường được sử dụng. Để có thể có được đầy đủ những phương thức chuẩn, hãy ghé thăm tài liệu của Python tại trang

[Python.org > String methods](https://www.python.org/doc/essays/tour/introduction.html#string-methods)

Các phương thức này đều có giá trị trả về và không ảnh hưởng gì tới giá trị ban đầu. Tương tự như một số hàm mà các bạn đã biết: int, float, str

```
>>> k = '12'  
>>> int(k)  
12  
>>> type(k) # k vẫn thuộc lớp str  
<class 'str'>
```

Các phương thức biến đổi

Phương thức capitalize

Cú pháp:

```
<chuỗi>.capitalize()
```

Công dụng: Trả về một chuỗi với kí tự đầu tiên được viết hoa và viết thường tất cả những kí tự còn lại.

```
>>> 'kteaM'.capitalize()  
'Kteam'  
>>> 'hello, Howkteam!'.capitalize()  
'Hello, howkteam!'  
>>> ' howKTEAM'.capitalize()  
' howkteam'
```

Phương thức upper

Cú pháp:

```
<chuỗi>.upper()
```

Công dụng: Trả về một chuỗi với tất cả các kí tự được chuyển thành các kí tự viết hoa

```
>>> 'kter'.upper()  
'KTER'  
>>> 'HOW kteam'.upper()  
'HOW KTEAM'  
>>> ' python'.upper()  
' PYTHON'
```

Phương thức lower

Cú pháp:

```
<chuỗi>.lower()
```

Công dụng: Trả về một chuỗi với tất cả các kí tự được chuyển thành các kí tự viết thường

```
>>> 'FREE education'.lower()  
'free education'  
>>> 'kteam'.lower()  
'kteam'  
>>> ' kTer'.lower()  
' kter'
```

Phương thức swapcase

Cú pháp:

```
<chuỗi>.swapcase()
```

Công dụng: Trả về một chuỗi với các kí tự viết hoa được chuyển thành viết thường, các kí tự viết thường được chuyển thành viết hoa

```
>>> 'free EDUCATION'.swapcase()  
'FREE education'  
>>> 'HoW kTeAm'.swapcase()  
'hOw kTeAm'
```

Phương thức title

Cú pháp:

```
<chuỗi>.title()
```

Công dụng: Trả về một chuỗi với định dạng tiêu đề, có nghĩa là các từ sẽ được viết hoa chữ cái đầu tiên, còn lại là viết thường

```
>>> 'share to be better'.title()  
'Share To Be Better'  
>>> 'FREE EDUCATION'.title()  
'Free Education'
```

Các phương thức định dạng

Phương thức center

Cú pháp:

```
<chuỗi>.center(width, [fillchar])
```

Công dụng: Trả về một chuỗi được căn giữa với chiều rộng **width**.

- Nếu **fillchar** là None (không được nhập vào) thì sẽ dùng kí tự khoảng trắng để căn, không thì sẽ căn bằng kí tự fillchar.
- Một điều nữa là kí tự fillchar là một chuỗi có độ dài là 1.

```
>>> 'abc'.center(12)  
'   abc   '  
>>> 'abc'.center(12, '*')  
*****abc*****  
>>> 'abc'.center(12, '*a')  
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: The fill character must be exactly one character long
>>> 'abc'.center(12, '')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: The fill character must be exactly one character long
```

Phương thức rjust

Cú pháp:

```
<chuỗi>.rjust(width, [fillchar])
```

Công dụng: Cách hoạt động tương tự như phương thức center, có điều là căn lề phải

```
>>> 'kteam'.rjust(12)
'      kteam'
>>> 'kteam'.rjust(12, '*')
'*****kteam'
```

Phương thức ljust

Cú pháp:

```
<chuỗi>.ljust(width, [fillchar])
```

Công dụng: Cách hoạt động tương tự phương thức center, nhưng căn lề trái.

```
>>> 'Kter'.ljust(12)
'Kter      '
>>> 'kter'.ljust(12, '*')
'kter*****'
```

Các phương thức xử lí

Phương thức encode

Cú pháp:

```
<chuỗi>.encode(encoding='utf-8', errors='strict')
```

Công dụng: Đây là phương thức dùng để encode một chuỗi với phương thức mã hóa mặc định là utf-8. Còn về errors mặc định sẽ là strict có nghĩa là sẽ có thông báo lỗi hiện lên nếu có vấn đề xuất hiện trong quá trình encode chuỗi. Một số giá trị ngoài strict là ignore, replace, xmlcharrefreplace. Vì phần này là phần nâng cao, Kteam xin phép không đi sâu.

```
>>> 'ő'.encode()
b'\xe1\xbb\x91 \xe1\xbb\x93'
```

Phương thức join

Cú pháp:

```
<kí tự nối>.join(<iterable>)
```

Công dụng: Trả về một chuỗi bằng cách nối các phần tử trong **iterable** bằng kí tự nối. Một iterable có thể là một tuple, list,... hoặc là một iterator (Kteam sẽ giải thích khái niệm này ở các bài sau).

- Một điểm lưu ý, các phần tử trong iterable buộc phải thuộc lớp str

```
>>> ''.join(['1', '2', '3']) # iterable ở đây là list ['1', '2', '3']
'1 2 3'
>>> ''.join(('1', '2', '3')) # iterable ở đây là tuple ('1', '2', '3')
'1 2 3'
>>> ''.join([1, 2, 3]) # phần tử trong list không phải là chuỗi
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

Phương thức replace

Cú pháp:

```
<chuỗi>.replace(old, new, [count])
```

Công dụng: Trả về một chuỗi với các chuỗi **old** nằm trong chuỗi ban đầu được thay thế bằng chuỗi **new**. Nếu **count** khác None (có nghĩa là ta cho thêm count) thì ta sẽ thay thế old bằng new với số lượng count từ trái qua phải.

- Nếu chuỗi old không nằm trong chuỗi ban đầu hoặc count là 0 thì sẽ trả về một chuỗi giống với chuỗi ban đầu

```
>>> 'abc how abc kteam'.replace('abc', 'aaa')
'aaa how aaa kteam'
>>> 'abc how abc kteam'.replace('a', 'AA')
'AAbc how AAAbc kteam'
>>> 'abc how abc kteam'.replace('abcd', 'AA')
'abc how abc kteam'
>>> 'abc how abc kteam'.replace('abc', 'AA', 1)
'AA how abc kteam'
>>> 'abc how abc kteam'.replace('abc', 'BB', 0)
'abc how abc kteam'
```

Phương thức strip

Phương thức này hơi rắc rối một tẹo nếu bạn chưa hiểu rõ cách nó hoạt động.

Cú pháp:

<chuỗi>.strip([chars])

Công dụng: Trả về một chuỗi với phần đầu và phần đuôi của chuỗi được bỏ đi các kí tự chars. Nếu chars bị bỏ trống thì mặc định các kí tự bị bỏ đi là dấu khoảng trắng và các escape sequence. Một số escape sequence ngoại lệ như \a sẽ được encode utf-8. Tuy vậy, không có ảnh hưởng gì tới nội dung.

```
>>> ' Kter '.strip()
'Kter'
>>> '%%%Kter%%%.strip('%')
'Kter'
>>> 'cababHowbaaaca'.strip('abc')
'How'
>>> '\t\n\aNter\aN\aN\n\v'.strip() # các \a biến thành \x07
'\x07Kter\x07\x07'
>>> print('\x07Kter\x07\x07') # nhưng khi dùng print vẫn có kết quả tương tự
```

Phương thức rstrip

Cú pháp:

<chuỗi>.rstrip()

Công dụng: Cách hoạt động hoàn toàn như phương thức strip, nhưng khác là chỉ bỏ đi ở phần đuôi (từ phải sang trái)

```
>>> ' Kter '.rstrip()
' Kter'
>>> '%%%Share%%%.rstrip('%')
'%%%Share'
>>> 'cababKterbaaaca'.rstrip('abc')
'cababKter'
```

Phương thức lstrip

Cú pháp:

```
<chuỗi>.lstrip()
```

Công dụng: Cách hoạt động tương tự phương thức rstrip, khác ở chỗ rstrip lo phần đuôi, còn lstrip lo phần đầu (từ trái sang phải)

```
>>> ' Kter '.lstrip()
'Kter '
>>> '%%%%%Kter%%%'.lstrip('%')
'Kter%%%'
>>> 'cababKterbaaaca'.lstrip('abc')
'Kterbaaaca'
```

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON – Phần 3](#).

Nếu bạn rút gọn được từ 5 dòng trở xuống thì bạn đã giải được câu hỏi trên. Còn đây là cách rút gọn ngắn nhất

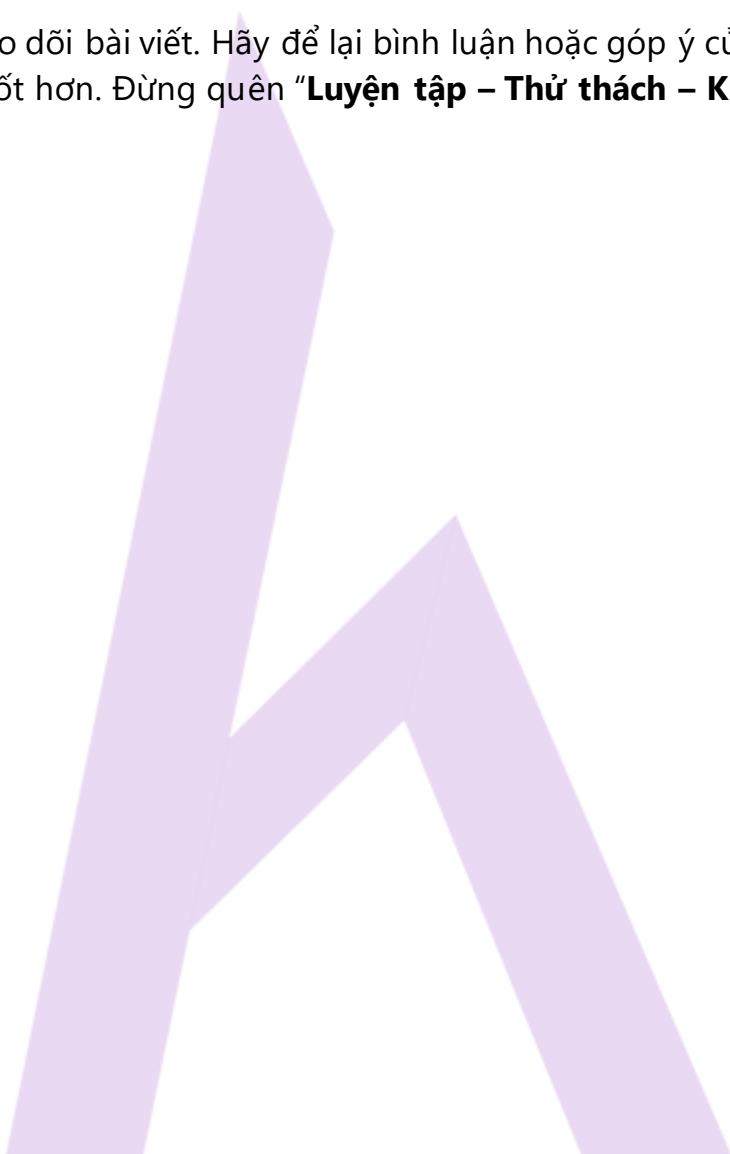
```
print('+ {-<6} + {-^15} + {:->10} +\n.format('', '', '') + '| {:<6} | {:^15} | {:>10}\n.format('ID', 'Ho va ten', 'Noi sinh') + '| {:<6} | {:^15} | {:>10} |\n.format('123', 'Yui\nHatano', 'Japanese') + '| {:<6} | {:^15} | {:>10} |\n.format('6969', 'Sunny Leone',\n'Canada') + '+ {-<6} + {-^15} + {:->10} +.format('', '', '')')
```

Kết luận

Qua bài viết này, bạn đã biết được một vài PHƯƠNG THỨC CHUỖI.

Ở bài viết sau, Kteam sẽ tiếp tục giới thiệu thêm một số phương thức của [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON \(Phần 5\)](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.



Bài 11: KIỂU DỮ LIỆU CHUỖI TRONG PYTHON (Phần 5 – Các phương thức chuỗi)

Xem bài học trên website để ủng hộ Kteam: [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON \(Phần 5 – Các phương thức chuỗi\)](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu một vài các [PHƯƠNG THỨC CHUỖI TRONG PYTHON](#).

Ở bài này, chúng ta sẽ tiếp tục tìm hiểu thêm một số phương thức của **KIỂU DỮ LIỆU CHUỖI** trong Python

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).

- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU SỐ](#) và [KIỂU DỮ LIỆU CHUỖI](#) trong Python.

Trong bài học này, chúng ta sẽ cùng tìm hiểu các vấn đề:

- Các phương thức tách chuỗi
- Các phương thức tiện ích
- Các phương thức xác thực

Các phương thức tách chuỗi

Phương thức split

Cú pháp:

```
<chuỗi>.split(sep=None, maxsplit=-1)
```

Công dụng: Trả về một list (kiểu dữ liệu sẽ được Kteam giới thiệu ở bài [KIỂU DỮ LIỆU LIST](#)) bằng cách chia các phần tử bằng kí tự **sep**.

- Nếu **sep** mặc định bằng None thì sẽ dùng kí tự khoảng trắng.
- Nếu **maxsplit** được mặc định bằng -1, Python sẽ không bị giới hạn việc tách, còn không, Python sẽ tách với số lần được cung cấp thông qua maxsplit.

```
>>> 'How Kteam K9'.split()  
['How', 'Kteam', 'K9']  
>>> 'How Kteam K9'.split(maxsplit=1)  
['How', 'Kteam K9']  
>>> 'How--Kteam--K9'.split('--')  
['How', 'Kteam', 'K9']  
>>> 'How&Kteam&K9'.split('&')  
['How', 'Kteam', 'K9']
```

Phương thức rsplit

Cú pháp:

```
<chuỗi>.split(sep=None, maxsplit=-1)
```

Công dụng: cũng hoàn toàn như phương thức split, có điều là việc tách từ bên **phải sang trái**

```
>>> 'How kteam EDUCATION'.rsplit()  
['How', 'kteam', 'EDUCATION']  
>>> 'How kteam EDUCATION'.rsplit(maxsplit=1)  
['How kteam', 'EDUCATION']
```

Phương thức partition

Cú pháp:

```
<chuỗi>.partition(sep)
```

Công dụng: Trả về một tuple với 3 phần tử. Các phần tử đó lần lượt là chuỗi **trước chuỗi sep, sep và chuỗi sau sep**.

- Trong trường hợp **không tìm thấy sep** trong chuỗi, mặc định trả về giá trị đầu tiên là chuỗi ban đầu và 2 giá trị kế tiếp là chuỗi rỗng.

```
>>> 'How kteam vs I hate python team vs Education'.partition('vs')  
('How kteam ', 'vs', ' I hate python team vs Education')  
>>> 'How kteam vs I hate python team vs Education'.partition('VS')  
('How kteam vs I hate python team vs Education', '', '')
```

Phương thức rpartition

Cú pháp:

```
<chuỗi>.rpartition(sep)
```

Công dụng: Cách phân chia giống như phương thức partition nhưng lại chia từ **phải qua trái**. Và với **sep** không có trong chuỗi thì sẽ trả về 2 giá trị đầu tiên là chuỗi rỗng và cuối cùng là chuỗi ban đầu

```
>>> 'How kteam vs I hate python team vs free Education'.rpartition('vs')
('How kteam vs I hate python team ', 'vs', ' free Education')
>>> 'How kteam vs I hate python team vs free Education'.rpartition('VS')
('', '', 'How kteam vs I hate python team vs free Education')
```

Các phương thức tiện ích

Phương thức count

Cú pháp:

```
<chuỗi>.count(sub, [start, [end]])
```

Công dụng: Trả về một số nguyên, chính là số lần xuất hiện của **sub** trong chuỗi. Còn **start** và **end** là số kĩ thuật slicing (lưu ý không hề có bước).

```
>>> 'kkkkk'.count('k')
5
>>> 'kkkkk'.count('kk')
2
>>> 'kkkkk'.count('k', 3)
2
>>> 'kkkkk'.count('k', 3, 4)
1
```

Phương thức startswith

Cú pháp:

```
<chuỗi>.startswith(prefix[, start[, end]])
```

Công dụng: Trả về giá trị **True** nếu chuỗi đó bắt đầu bằng chuỗi **prefix**. Ngược lại là **False**.

- Hai yếu tố **start**, **end** tương trưng cho việc slicing (không có bước) để kiểm tra với chuỗi slicing đó.

```
>>> 'how kteam free education'.startswith('ho')
True
>>> 'how kteam free education'.startswith('ha')
False
>>> 'how kteam free education'.startswith('ho', 4)
False
```

Phương thức endswith

Cú pháp:

```
<chuỗi>.endswith(prefix[, start[, end]])
```

Công dụng: Trả về giá trị **True** nếu chuỗi đó kết thúc bằng chuỗi **prefix**. Ngược lại là **False**.

- Hai yếu tố **start** **end** tương trưng cho việc slicing (không có bước) để kiểm tra với chuỗi slicing đó.

```
>>> 'how kteam free education'.endswith('n')
True
>>> 'how kteam free education'.endswith('ho')
```

```
False  
>>> 'how kteam free education'.endswith('n', 0, 9)  
False
```

Phương thức find

Cú pháp:

```
<chuỗi>.find(sub[, start[, end]])
```

Công dụng: Trả về một số nguyên, là vị trí đầu tiên của sub khi dò từ **trái sang phải** trong chuỗi. Nếu **sub** không có trong chuỗi, kết quả sẽ là **-1**. Vẫn như các phương thức khác, **start end** đại diện cho slicing và ta sẽ tìm trong chuỗi slicing này.

```
>>> 'howteam'.find('h')  
0  
>>> 'howteam'.find('k')  
3  
>>> 'howteam'.find('l')  
-1  
>>> 'howteam'.find('h', 2)  
-1
```

Phương thức rfind

Cú pháp:

```
<chuỗi>.rfind(sub[, start[, end]])
```

Công dụng: Tương tự phương thức find nhưng tìm từ **phải sang trái**

```
>>> 'howteamhow'.rfind('h')  
8
```

Phương thức index

Cú pháp:

```
<chuỗi>.index(sub[, start[, end]])
```

Công dụng: Tương tự phương thức find. Nhưng khác biệt là sẽ có lỗi **ValueError** nếu không tìm thấy chuỗi **sub** trong chuỗi ban đầu

```
>>> 'abcd'.index('z')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: substring not found
```

Phương thức rindex

Cú pháp:

```
<chuỗi>.rindex(sub[, start[, end]])
```

Công dụng: Tương tự phương thức rindex. Và cũng khác ở điểm là sẽ có **ValueError** nếu không tìm thấy chuỗi **sub** trong chuỗi ban đầu

```
>>> 'abcd'.rindex('z')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: substring not found
```

Các phương thức xác thực

Phương thức islower

Cú pháp:

```
<chuỗi>.islower()
```

Công dụng: Trả về **True** nếu tất cả các kí tự trong chuỗi đều là viết thường. Ngược lại là **False**

```
>>> 'python'.islower()  
True  
>>> 'pythoN'.islower()  
False
```

Phương thức isupper

Cú pháp:

```
<chuỗi>.isupper()
```

Công dụng: Trả về **True** nếu tất cả các kí tự trong chuỗi đều là viết hoa. Ngược lại là **False**

```
>>> 'HOWKTEAM'.isupper()  
True  
>>> 'HowKteam'.isupper()  
False
```

Phương thức istitle

Cú pháp:

```
<chuỗi>.istitle()
```

Công dụng: Trả về **True** nếu chuỗi đó là một dạng title. Ngược lại là **False**

```
>>> 'Free Education'.istitle()  
True  
>>> 'FrEe Education'.istitle()  
False
```

Phương thức isdigit

Cú pháp:

```
<chuỗi>.isdigit()
```

Công dụng: Trả về **True** nếu tất cả các kí tự trong chuỗi đều là những con số từ 0 đến 9

Lưu ý: Phương thức này gần giống với **isnumeric**. Nhưng vì liên quan nhiều đến toán nên Kteam sẽ không giới thiệu về phương thức isnumeric và cũng không so sánh sự khác nhau giữa hai phương thức.

```
>>> '0123'.isdigit()  
True  
>>> '123'.isdigit()  
True  
>>> '-123'.isdigit()  
False
```

Phương thức isspace

Cú pháp:

```
<chuỗi>.isspace()
```

Công dụng: Trả về **True** nếu tất cả các kí tự trong chuỗi đều là kí tự khoảng trắng

```
>>> ' '.isspace()  
True  
>>> 'd '.isspace()  
False
```

Câu hỏi củng cố

Với chuỗi s bên dưới

```
>>> s = 'aaaAAaaaooaaneu mot Ngay naO Doaaaaaaaa'
```

Hãy dùng các phương thức để có được chuỗi s sau đây

```
>>> s  
'Neu Mot Ngay Nao Do'
```

Hãy cố gắng làm càng ít dòng code càng tốt.

Đáp án của phần này sẽ được trình bày ở bài tiếp theo.

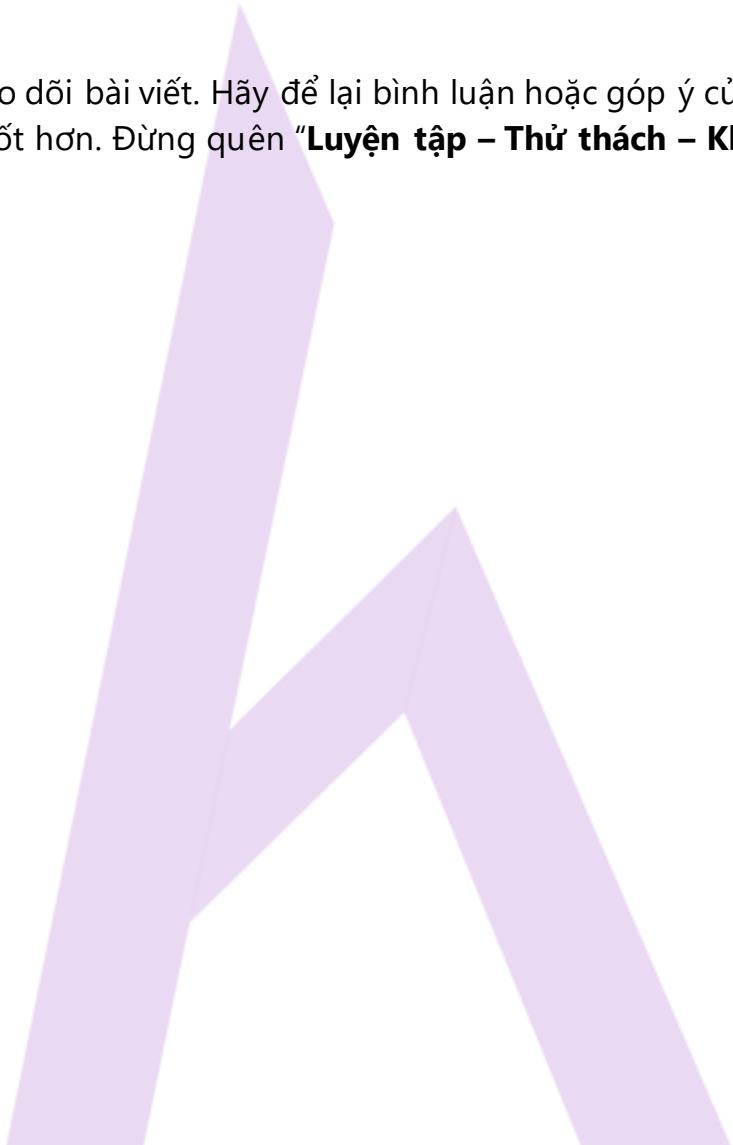
Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Qua bài viết này, bạn đã biết được các PHƯƠNG THỨC CHUỖI, và phần nào đó có các kiến thức tốt về KIỂU DỮ LIỆU CHUỖI.

Ở bài viết sau, Kteam sẽ giới thiệu với các bạn [KIỂU DỮ LIỆU LIST TRONG PYTHON](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".



Bài 12: KIỂU DỮ LIỆU LIST TRONG PYTHON

(Phần 1)

Xem bài học trên website để ủng hộ Kteam: [KIỂU DỮ LIỆU LIST TRONG PYTHON \(Phần 1\)](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong những bài trước, Kteam đã giới thiệu đến bạn loạt bài về [KIỂU DỮ LIỆU CHUỖI](#) trong Python gồm rất nhiều kiến thức chi tiết và dễ hiểu nhất có thể.

Sang bài này, chúng ta sẽ cùng nhau tìm hiểu một trong những kiểu dữ liệu cực kỳ quan trọng trong Python. Đó chính là **KIỂU DỮ LIỆU LIST**

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU SỐ](#) và [KIỂU DỮ LIỆU CHUỖI](#) trong Python.

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Container. Đặt vấn đề và cách giải quyết
 - Giới thiệu về List trong Python
 - Cách khởi tạo List
 - Một số toán tử với List trong Python
 - Indexing và cắt List trong Python
 - Thay đổi nội dung List trong Python
 - Ma trận
 - Vấn đề cần lưu tâm khi sử dụng List
 - Củng cố bài học
-

Container. Đặt vấn đề và cách giải quyết

Các bạn đã biết đến **BIẾN** (đã giới thiệu trong bài [BIẾN TRONG PYTHON](#)), đó là một **container** cho phép ta lưu trữ các dữ liệu và lấy ra khi cần, thay đổi khi ta cần cập nhật giá trị hoặc sửa chữa.

Nhưng, khả năng của biến vẫn bị giới hạn! Đơn giản với một ví dụ, ta cần biến **teo** lưu cho ta giá trị là chuỗi `“Teo”` là tên của Tèo, và tuổi của Tèo là số **17**.

```
>>> teo = "Teo"  
>>> teo  
'Teo'  
>>> teo = 17  
>>> teo  
17
```

Biến **teo** của chúng ta không thể lưu hai giá trị một lúc. Không chỉ tên và tuổi, Tèo còn rất nhiều thông tin muốn lưu vào biến **teo** nữa như ngày sinh của gấu, số lần fix bug trong một tháng, khóa học mới coi gần nhất, số lần tè dầm ở tuổi 17,...

Với năng lực của một người mới học lập trình, sáng kiến tối ưu nhất họ đưa ra là mỗi giá trị ta có một biến riêng biệt. Và đây được coi là một giải pháp hay!

Tuy nhiên vẫn ở tầm vi mô. Tèo nó tham, muốn lưu cả mấy thứ linh tinh về cô gấu dễ thương của hắn. Lúc đó, việc bạn kêu Tèo tạo ra số lượng biến để lưu trữ cũng là một điều gian nan rồi.

Đó là vì sao ta cần một thứ cũng như biến, nhưng nội công lại thâm hậu hơn biến, có khả năng lưu trữ nhiều giá trị cùng một lúc. Vì thế, Python có rất nhiều các container cho phép ta lưu trữ nhiều các giá trị, đối tượng cùng một lúc, hỗ trợ cho chúng ta trong việc truy xuất, tính toán, thay đổi (một số container trong Python không hỗ trợ việc thay đổi),...

Trong các ngôn ngữ lập trình khác, những container chứa được nhiều giá trị cùng một lúc thường được gọi là **ARRAY** (mảng).

Giới thiệu về List trong Python

LIST là một container được sử dụng rất nhiều trong các chương trình Python. Một List gồm các yếu tố sau:

- Được giới hạn bởi cặp ngoặc **[]**, tất cả những gì nằm trong đó là những phần tử của List.
- Các phần tử của List được phân cách nhau ra bởi dấu phẩy **(,)**.
- List có khả năng **chứa mọi giá trị, đối tượng** trong Python. Và bao gồm chứa chính nó! (một trường hợp hay ho Kteam sẽ giới thiệu ở phần khác).

Ví dụ:

```
>>> [1, 2, 3, 4, 5] # Một List chứa 5 số nguyên  
[1, 2, 3, 4, 5]  
>>> ['a', 'b', 'c', 'd'] # Một List chứa 4 chuỗi  
['a', 'b', 'c', 'd']  
>>> [[1, 2], [3, 4]] # Một List chứa 2 List là [1, 2] và [3, 4]  
[[1, 2], [3, 4]]  
>>> [1, 'one', [2, 'two']] # List chứa số nguyên, chuỗi, và List  
[1, 'one', [2, 'two']]
```

Cách khởi tạo List

Sử dụng cặp dấu ngoặc [] đặt giá trị bên trong

Cú pháp:

[<giá trị thứ nhất>, <giá trị thứ hai>, .., <giá trị thứ n – 1>, <giá trị thứ n>]

Ví dụ:

```
>>> lst = [1,2,5,"kteam"]
>>> lst
[1, 2, 5, 'kteam']
>>> empty_list = [] # khởi tạo list rỗng
>>> empty_list
[]
```

Sử dụng List Comprehension

Cú pháp

[Comprehension]

Ví dụ:

```
>>> a = [kteam for kteam in range(3)]
>>> a
[0, 1, 2]
>>> another_lst = [[n, n * 1, n * 2] for n in range(1, 4)]
>>> another_lst
[[1, 1, 2], [2, 2, 4], [3, 3, 6]]
```

List **comprehension** là một cách khởi tạo một List rất thú vị trong Python. Do đó, rất khó để có thể nói hết các trường hợp. Vì vậy, hãy tạm gác lại kiến thức này, bạn không cần phải cố gắng hiểu nó khi chúng ta chưa gặp gỡ các vòng lặp.

Sử dụng constructor List

Cú pháp:

list (iterable)

Lưu ý: **iterable** là một đối tượng nói chung của các container. Khái niệm này sẽ được Kteam giới thiệu ở bài sau. Đối với bạn khi theo dõi khóa học này của Kteam, bạn đã được biết hai iterable đó chính là chuỗi, và List.

Ví dụ:

```
>>> lst = list([1, 2, 3])
>>> lst
[1, 2, 3]
>>> str_lst = list('HOWKTEAM')
>>> str_lst
['H', 'O', 'W', 'K', 'T', 'E', 'A', 'M']
>>> list(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Một số toán tử với List trong Python

Các toán tử của List gần giống và tương tự với chuỗi (bạn có thể tham khảo toán tử của chuỗi ở bài [KIỂU DỮ LIỆU CHUỖI – phần 2](#)).

Toán tử +

```
>>> lst = [1, 2]
>>> lst += ['one', 'two']
>>> lst
[1, 2, 'one', 'two']
>>> lst += 'abc' # cộng List và chuỗi
>>> lst
[1, 2, 'one', 'two', 'a', 'b', 'c']
>>> 'abc' + [1, 2] # List cộng chuỗi cho phép, chuỗi cộng List thì không.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not list
```

Toán tử *

```
>>> lst = list('KTER') * 2
>>> lst
['K', 'T', 'E', 'R', 'K', 'T', 'E', 'R']
>>> [1, 2] * 3
[1, 2, 1, 2, 1, 2]
```

Toán tử in

```
>>> 'a' in [1, 2, 3]
False
>>> 'a' in ['a', 2, 3]
True
>>> 'a' in [['a'], 'b', 'c'] # chỉ có ['a'] thôi, không có 'a'
False
```

Indexing và cắt List trong Python

Như đã đề cập, List với chuỗi giống nhau rất nhiều điểm, và phần Indexing và cắt List này hoàn toàn giống với Indexing và cắt chuỗi. (Nếu chưa biết về chuỗi bạn có thể tham khảo qua các bài về [KIẾU DỮ LIỆU CHUỖI TRONG PYTHON – Phần 1](#))

```
>>> lst = [1, 2, 'a', 'b', [3, 4]]
```

```
>>> lst[0]
1
>>> lst[-1]
[3, 4]
>>> lst[3]
'b'
>>> lst[1:3]
[2, 'a']
>>> lst[:2]
[1, 2]
>>> lst[2:]
['a', 'b', [3, 4]]
>>> lst[::-1]
[[3, 4], 'b', 'a', 2, 1]
```

Thay đổi nội dung List trong Python

Như bạn đã biết, ta không thể thay đổi nội dung của chuỗi như ví dụ bên dưới

```
>>> s = 'math'
>>> s[1]
'a'
>>> s[1] = 'i'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Còn về phần List, ta có thể thay đổi nội dung của nó

```
>>> lst = [1, 'two', 3]
>>> lst[1]
'two'
>>> lst[1] = 2
>>> lst
[1, 2, 3]
```

Ma trận

Nghe ma trận hoàn tráng thế thôi, bạn đã thấy nó rồi. Ví dụ một List chứa một List khác đây.

```
>>> lst = [[1, 2, 3], [4, 5, 6]]  
>>> lst  
[[1, 2, 3], [4, 5, 6]]
```

Ta dễ dàng truy cập hai phần tử của List vừa mới khởi tạo

```
>>> lst[0]  
[1, 2, 3]  
>>> lst[-1]  
[4, 5, 6]
```

Hai giá trị đó cũng là một List. Và lẽ dĩ nhiên, bạn có quyền truy cập đến các phần tử con của phần tử nằm trong List bạn vừa khởi tạo. Thậm chí là cắt List!

```
>>> lst[0][0]  
1  
>>> lst[0][-1]  
3  
>>> lst[1][1]  
5  
>>> lst[0][:2]  
[1, 2]  
>>> lst[1][:]  
[4, 5, 6]
```

Vấn đề cần lưu tâm khi sử dụng List

Những lưu ý này nếu bạn không biết, chương trình của bạn có thể có output khác với bạn mong muốn.

Không được phép gán List này qua List kia nếu không có chủ đích

Hãy xem xét đoạn code sau đây

```
>>> lst = [1, 2, 3]
>>> another_lst = lst
>>> lst
[1, 2, 3]
>>> lst = [1, 2, 3]
>>> lst
[1, 2, 3]
>>> another_lst = lst
>>> another_lst
[1, 2, 3]
>>> lst
[1, 2, 3]
```

Mọi thứ ổn, không có gì xảy ra, cho tới khi bạn thay đổi giá trị bất kì của một trong hai List đó.

```
>>> another_lst[1]
2
>>> another_lst[1] = 'Two'
>>> another_lst
[1, 'Two', 3]
>>> lst
[1, 'Two', 3]
```

Chỉnh một, nhưng đổi tới hai. Lí do là vì khi bạn gán giá trị List trực tiếp như thế, bạn đang đưa hai List đó trở cùng vào một nơi.

Hãy tưởng tượng Tèo có 50 nghìn. Sau đó bạn sử dụng phép thuật của mình gán số tiền cô gấu của Tèo bằng số tiền của Tèo. Khi đó, cô gấu dễ thương của Tèo không tự nhiên mà có 50 nghìn, mà thay vào đó, bạn đã gián tiếp cho phép gấu của Tèo sử dụng số tiền của Tèo nhịn ăn mì tôm bấy lâu nay. Và vào một ngày trời mưa không rời, cô ấy chạy đi mua một gói Snack mất 5 nghìn và sử dụng số tiền 50 nghìn bạn vừa mới gán cho cô ấy. Hậu quả là Tèo về thấy mất đâu 5 nghìn.

Do đó, trước khi gán, bạn phải copy giá trị của List

```
>>> lst = [1, 2, 3]
>>> lst
```

```
[1, 2, 3]
>>> another_list = list(lst) # cách 1
>>> another_list
[1, 2, 3]
>>> last_list = lst[:] # cách 2
>>> last_list
[1, 2, 3]
>>> another_list[1] = 'Two'
>>> last_list[1] = 'Chu'
>>> lst
[1, 2, 3]
>>> another_list
[1, 'Two', 3]
>>> last_list
[1, 'Chu', 3]
```

Thêm một trường hợp nữa bạn cần phải lưu ý, đó là lúc bạn cần copy giá trị của một ma trận

```
>>> lst = [[1, 2, 3], [4, 5, 6]]
>>> lst
[[1, 2, 3], [4, 5, 6]]
>>> another_lst = lst[:]
>>> another_lst[0] = 'ok'
>>> lst
[[1, 2, 3], [4, 5, 6]]
>>> another_lst
['ok', [4, 5, 6]]
```

Đúng như chúng ta mong đợi. Thế nhưng...

```
>>> another_lst[1]
[4, 5, 6]
>>> another_lst[1][0] = 'changed'
>>> another_lst
['ok', ['changed', 5, 6]]
>>> lst
[[1, 2, 3], ['changed', 5, 6]]
```

Lưu ý: nó chỉ sao chép các phần tử của List. Không hề sao chép các phần tử con của các phần tử nằm trong List. Do đó, nếu bạn thay đổi các phần tử trong List thì không sao, tuy nhiên nếu thay đổi phần tử con của các phần tử trong List, thì vấn đề lại xuất hiện.

Đương nhiên, bạn vẫn giải quyết được, nhưng vì rườm rà khi không có vòng lặp. Do đó, chúng ta tạm dừng ở việc nhận biết. Còn phần giải quyết sẽ đợi nay mai. Khi võ công Xà Ngữ của chúng ta được nâng cao và tiếp cận với tuyệt kĩ vòng lặp.

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIẾU DỮ LIỆU CHUỖI TRONG PYTHON – Phần 5](#)

- Cách đơn giản

```
>>> s = s.lower()  
>>> s = s.strip('a')  
>>> s = s.lstrip('ao')  
>>> s = s.title()  
>>> s  
'Neu Mot Ngay Nao Do'
```

- Cách ngắn

```
>>> s = s.lower().strip('a').lstrip('ao').title()  
>>> s  
'Neu Mot Ngay Nao Do'
```

Câu hỏi củng cố

1. Tìm các cách khởi tạo List hợp lệ dưới đây
 - a. list(list(list('abc')))
 - b. [1, 2, 3] + list(4)
 - c. list()
 - d. [0] * 3
2. List có phải là một hashable object (immutable object)?
3. Với chuỗi s dưới đây

```
s = 'aaaaaaaaAAAAAaaa//123123//000000//&&TTT%%abcxyznontqfadf'
```

Hãy lấy mật mã trong chuỗi s, biết mật mã nằm giữa && và %. Cố gắng tối thiểu dòng code

Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Bài viết này đã sơ lược cho các bạn KIỂU DỮ LIỆU LIST TRONG PYTHON.

Ở bài sau, Kteam sẽ tiếp tục nói về [KIỂU DỮ LIỆU LIST TRONG PYTHON - Phần 2](#). Cụ thể là một số phương thức của List trong Python

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.

Bài 13: KIỂU DỮ LIỆU LIST TRONG PYTHON

(Phần 2)

Xem bài học trên website để ủng hộ Kteam: [KIỂU DỮ LIỆU LIST TRONG PYTHON \(Phần 2\)](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu cho các bạn [KIỂU DỮ LIỆU LIST TRONG PYTHON – Phần 1](#).

Ở bài này Kteam sẽ tiếp tục đề cập đến **KIỂU DỮ LIỆU LIST TRONG PYTHON** – Các phương thức List.

Nội dung

Các kiến thức để có thể hiểu tốt được bài viết này

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU SỐ](#) và [KIỂU DỮ LIỆU CHUỖI](#) trong Python.
- [KIỂU DỮ LIỆU LIST](#) trong Python

Trong bài học này, chúng ta sẽ cùng tìm hiểu các vấn đề

- Giới thiệu về phương thức của kiểu dữ liệu List trong Python
 - Các phương thức tiện ích
 - Các phương thức cập nhật
 - Các phương thức xử lí
-

Giới thiệu về phương thức của kiểu dữ liệu List trong Python

Kiểu dữ liệu List của Python có một số phương thức giúp chúng ta xử lí các vấn đề liên quan đến nó. Kteam sẽ giúp bạn tìm hiểu các phương thức đó.

Một số phương thức Kteam sẽ không nói rõ về nó vì có một số kiến thức bạn chưa nắm được. Điển hình đó là hàm.

Bên cạnh đó có một số phương thức có dạng biến thể là một hàm sẽ được Kteam đề cập ở một bài trong tương lai

Các phương thức tiện ích

Phương thức count

Cú pháp:

```
<List>.count(sub, [start, end])
```

Công dụng: Giống với phương thức count của kiểu dữ liệu chuỗi.

- Trả về một số nguyên, chính là số lần xuất hiện của **sub** trong chuỗi.
- Còn **start** và **end** là số kĩ thuật slicing (lưu ý không hề có bước).

```
>>> Kteam = [1, 5, 1, 6, 2, 7]
>>> Kteam.count(1)
2
>>> Kteam.count(3)
0
```

Phương thức index

Cú pháp:

```
<List>.index(sub[, start[, end]])
```

Công dụng: Tương tự phương thức index của kiểu dữ liệu chuỗi.

```
>>> Kteam = [1, 2, 3]
>>> Kteam.index(2)
1
>>> Kteam.index(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 4 is not in list
```

Phương thức copy

Cú pháp:

```
<List>.copy()
```

Công dụng: Trả về một List tương tự với `List[:]`

```
>>> lst = [1, 2, 3]
>>> another_lst = lst.copy() # tương tự lst[:]
>>> another_lst[0] = 4
>>> another_lst
[4, 2, 3]
>>> lst
[1, 2, 3]
```

Phương thức clear

Cú pháp:

```
<List>.clear()
```

Công dụng: Xóa mọi phần tử có trong List.

Lưu ý: Các phiên bản Python 2.X hoặc dưới Python 3.2 sẽ không có phương thức này

```
>>> Kteam = [1, 2, 3]
>>> Kteam.clear()
>>> Kteam
[]
```

Phương thức trên bản chất không như những cách gán với một List rỗng.
Giống như dưới đây:

```
>>> lst = []
>>> lst = list()
```

Phương thức clear sẽ xóa đi các phần tử ở trong List. Các bạn sẽ biết thêm khi biết tới **câu lệnh del** sẽ được Kteam giới thiệu trong các bài sau.

Để thể hiện rõ sự khác biệt giữa hai trường hợp trên. Kteam sẽ lấy ví dụ để minh họa:

- Bạn còn nhớ ví dụ về việc Tèo và gấu của Tèo dùng chung số tiền chứ?

```
>>> tien_teo = [50]
>>> tien_teo
[50]
>>> tien_gau_cua_teo = tien_teo # Tèo và gấu của Tèo đang dùng chung 50
nghìn
>>> tien_gau_cua_teo
[50]
>>> tien_gau_cua_teo = [] # ta gán lại số tiền cô gấu của Tèo là một List rỗng
>>> tien_gau_cua_teo
[]
>>> tien_teo # và đương nhiên, tiền của Tèo không bị ảnh hưởng
[50]
```

Tiếp đến, ta sẽ dùng **phương thức clear**.

```
>>> tien_teo = [50]
>>> tien_teo
[50]
>>> tien_gau_cua_teo = tien_teo # Tèo và gấu của Tèo đang dùng chung 50 nghìn
>>> tien_gau_cua_teo
[50]
>>> tien_gau_cua_teo.clear() # xử dụng phương thức clear
>>> tien_gau_cua_teo
[]
>>> tien_teo # tiền của Tèo đã bị xóa theo
[]
```

Các phương thức cập nhật

Phương thức append

Cú pháp:

```
<List>.append(x)
```

Công dụng: Thêm phần tử x vào cuối List

```
>>> howteam = [1, 2]
>>> howteam.append(3)
>>> howteam
[1, 2, 3]
>>> howteam.append([4, 5]) # chú ý trường hợp này
>>> howteam
[1, 2, 3, [4, 5]]
```

Phương thức extend

Cú pháp:

```
<List>.extend(iterable)
```

Công dụng: Thêm từng phần tử một của iterable vào cuối List.

```
>>> Kteam = [1, 2, 3]
>>> Kteam.extend([4, 5]) # xem sự khác biệt giữa append và extend
>>> Kteam
[1, 2, 3, 4, 5]
>>> Kteam.extend([[6, 7], 8])
```

```
>>> kteam  
[1, 2, 3, 4, 5, [6, 7], 8]
```

Phương thức insert

Cú pháp:

```
<List>.insert (i, x)
```

Công dụng: Thêm phần **x** vào vị trí **i** ở trong List.

```
>>> kteam = [1, 2, 3]  
>>> kteam.insert(1, 8) # thêm phần tử 8 vào trong List kteam ở vị trí 1  
>>> kteam  
[1, 8, 2, 3]
```

Nếu vị trí **i** lại lớn hơn hoặc bằng số phần tử ở trong List thì kết quả sẽ tương tự như phương thức append.

```
>>> kteam= [1, 2, 3]  
>>> kteam.insert(4, 20) # vị trí 4, nhưng trong List chỉ có 3 phần tử  
>>> kteam  
[1, 2, 3, 20]  
>>> kteam.insert(len(kteam), 5) # vị trí thứ 4, bằng số phần tử trong List  
>>> kteam  
[1, 2, 3, 20, 5]
```

Nếu vị trí **i** là một số âm, bạn cần lưu ý kỹ ví dụ sau. Bạn chắc vẫn còn nhớ về việc indexing với vị trí là một số âm? nếu không nhớ bạn có thể xem lại bài [KIỂU DỮ LIỆU CHUỖI TRONG PYTHON – Phần 2](#) trước khi vào ví dụ này.

```
>>> kter = [1, 2, 3]
>>> kter[-1]
3
>>> kter[-2]
2
>>> kter[-3]
1
```

Khi bạn insert mà lại dùng vị trí **i** là số âm, thì vị trí được insert sẽ là **i - 1**.

```
>>> kteam = [1, 2, 3]
>>> kteam[-1]
3
>>> kteam.insert(-1, 4) # thêm vào vị trí (-1 - 1) là -2
>>> kteam
[1, 2, 4, 3]
```

Nếu vị trí **i - 1** (đang xét indexing âm) không có trong List, mặc định, phần tử **x** sẽ được thêm vào đầu List

```
>>> kteam= [1, 2, 3]
>>> kteam[-20] # không có phần tử -20 trong List
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> kteam.insert(-20, 0)
>>> kteam
[0, 1, 2, 3]
```

Phương thức pop

Cú pháp:

<List>.pop([i]**)**

Công dụng: Bỏ đi phần tử thứ **i** trong List và trả về giá trị đó. Nếu vị trí **i** không được cung cấp, phương thức này sẽ tự bỏ đi phần tử cuối cùng của List và trả về giá trị đó.

```
>>> kter= [1, 2, 3, 4, 5, 6]
>>> kter.pop(3)
4
>>> kter
[1, 2, 3, 5, 6]
>>> kter.pop(-3)
>>> kter.pop(-3)
3
>>> kter
[1, 2, 5, 6]
>>> kter.pop() # mặc định sẽ pop phần tử cuối cùng nằm trong List
6
>>> kter
[1, 2, 5]
```

Phương thức remove

Cú pháp:

<List>.remove(**x**)

Công dụng: Bỏ đi phần tử đầu tiên trong List có giá trị **x**. Nếu trong List không có giá trị x sẽ có lỗi được thông báo

```
>>> kteam = [1, 5, 6, 2, 1, 7]
>>> kteam.remove(1)
>>> kteam
[5, 6, 2, 1, 7]
>>> kteam.remove(3)
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

Các phương thức xử lí

Phương thức reverse

Cú pháp:

```
<List>.reverse()
```

Công dụng: Đảo ngược các phần tử ở trong List.

```
>>> kteam= [1, 2, 3]
>>> kteam.reverse()
>>> kteam
[3, 2, 1]
```

Phương thức sort

Đây là phương thức mà Kteam sẽ chỉ giới thiệu sơ lược về nó. Kteam sẽ bỏ qua **key** trong phần giới thiệu cú pháp của phương thức bên dưới.

Cú pháp:

```
<List>.sort(key=None, reverse=False)
```

Công dụng: Sắp xếp các phần tử từ bé đến lớn bằng cách so sánh trực tiếp.

```
>>> howteam= [3, 6, 7, 1, 2, 4]
>>> howteam.sort()
>>> howteam
[1, 2, 3, 4, 6, 7]
```

Vì sao nói nó là so sánh trực tiếp. Bởi vì không chỉ số, nó còn so sánh cả chuỗi, cả List, và mọi thứ khác.

```
>>> lst = ['k', 'free', '9kteam', 'howteam']
>>> lst.sort()
>>> lst
['9kteam', 'free', 'howteam', 'k']
```

Ghi nhớ rằng, các phần tử phải có thể so sánh với nhau. Trường hợp dưới đây bạn không thể so sánh chuỗi với số được, do đó sẽ có lỗi hiện lên.

```
Ist = ['kteam', 69]
>>> Ist.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

Chúng ta sẽ nói đến từ khóa **reverse**. Từ khóa này bạn chỉ có thể cho 2 giá trị, một là **True**, hai là **False**.

- Nếu là **False**, các phần tử được sắp xếp từ bé đến lớn, còn ngược lại là từ lớn đến bé.

```
>>> kteam = [6, 8, 2, 5, 1, 10, 4]
>>> true_reverse = kteam.copy() #tạo một bản sao của kteam và không ảnh hưởng đến kteam
>>> kteam.sort() # không đưa giá trị cho reverse thì mặc định là False
>>> true_reverse.sort(reverse=True)
>>> kteam
[1, 2, 4, 5, 6, 8, 10]
>>> true_reverse
[10, 8, 6, 5, 4, 2, 1]
```

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIẾU DỮ LIỆU LIST TRONG PYTHON – Phần 1](#).

1. Những đáp án c, d là các cách khởi tạo đúng
2. Không, vì ta có thể thay đổi nội dung của List
3. Đáp án

```
>>> code = s.split('&&')[-1].split('%%')[0]
```

Câu hỏi củng cố

1. Chuyện gì xảy ra khi ta dùng phương thức pop lên một List rỗng

```
>>> lst = list()  
>>> lst  
[]  
>>> lst.pop
```

2. Ta có thể sắp xếp được List dưới đây bằng phương thức sort hay không?

```
>>> lst = [[1, 2], ['abc', 'def']]
```

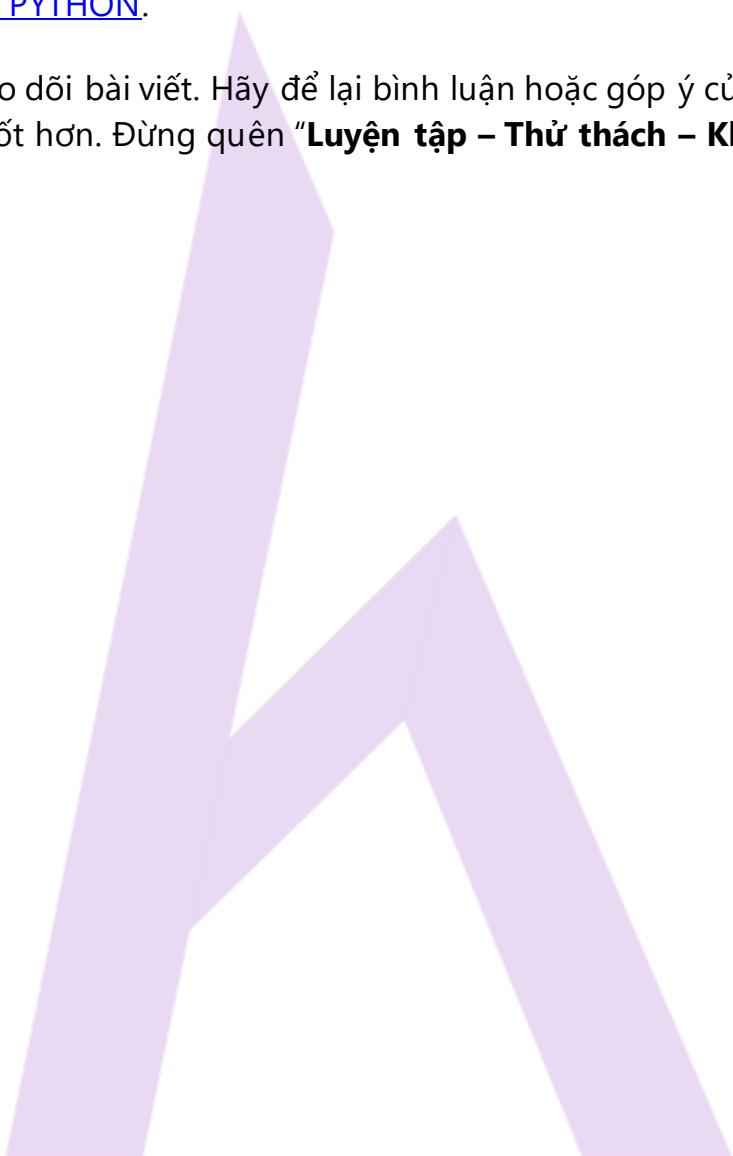
Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Qua bài viết này, Bạn đã hiểu thêm về các phương thức của KIỂU DỮ LIỆU LIST TRONG PYTHON

Ở bài sau. Kteam sẽ giới thiệu tới bạn một container nữa đó chính là [KIẾU DỮ LIỆU TUPLE TRONG PYTHON](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.



Bài 14: KIỂU DỮ LIỆU

TUPLE TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [KIỂU DỮ LIỆU TUPLE TRONG PYTHON](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong các bài trước, Kteam đã giới thiệu đến bạn [KIỂU DỮ LIỆU LIST](#), một container tuyệt vời trong Python

Ở bài này Kteam sẽ giới thiệu tới bạn một container khác đó chính **KIỂU DỮ LIỆU TUPLE** trong Python

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU SỐ](#) và [KIỂU DỮ LIỆU CHUỖI](#) trong Python.
- [KIỂU DỮ LIỆU LIST](#) trong Python

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Giới thiệu về Tuple trong Python.
- Cách khởi tạo Tuple.
- Một số toán tử với Tuple trong Python.

- Indexing và cắt Tuple trong Python.
 - Thay đổi nội dung Tuple trong Python.
 - Ma trận.
 - Tuple có phải luôn luôn là một Hash object?
 - Các phương thức của Tuple.
 - Khi nào thì chọn Tuple thay cho List?
-

Giới thiệu về Tuple trong Python

Tuple là một container cũng được sử dụng rất nhiều trong các chương trình Python không thua kém gì List. (List đã được giới thiệu trong bài [KIỂU DỮ LIỆU LIST TRONG PYTHON](#))

Một Tuple gồm các yếu tố sau:

- Được giới hạn bởi cặp ngoặc **()**, tất cả những gì nằm trong đó là những phần tử của Tuple.
- Các phần tử của Tuple được phân cách nhau ra bởi dấu phẩy **,**.
- Tuple có khả năng chứa mọi giá trị, đối tượng trong Python.

Ví dụ:

```
>>> (1, 2, 3, 4, 5) # Một Tuple chứa 5 số nguyên  
(1, 2, 3, 4, 5)  
>>> ('k', 't', 'e', 'r') # Một Tuple chứa 4 chuỗi  
('k', 't', 'e', 'r')  
>>> ([1, 2], (3, 4)) # Một Tuple chứa 1 List là [1, 2] và 1 Tuple là (3, 4)  
([1, 2], (3, 4))  
>>> (1, 'kteam', [2, 'k9']) # Tuple chứa số nguyên, chuỗi, và List  
(1, 'kteam', [2, 'k9'])
```

Cách khởi tạo Tuple

Sử dụng cặp dấu ngoặc () và đặt giá trị bên trong

Cú pháp:

(<giá trị thứ nhất>, <giá trị thứ hai>, ..., <giá trị thứ n – 1>, <giá trị thứ n>)

Ví dụ:

```
>>> tup = (1, 2, 3, 4)
>>> tup
(1, 2, 3, 4)
>>> empty_tup = () # khởi tạo tuple rỗng
>>> empty_tup
()
>>> type(tup) # kiểu dữ liệu Tuple thuộc lớp tuple
<class 'tuple'>
```

Bạn hãy chú ý khi khởi tạo tuple với một giá trị.

```
>>> tup = (9) # Tuple có một giá trị là số 9
>>> tup # có kết quả là
9
>>> type(tup) # không thuộc lớp Tuple
<class 'int'>
>>> str_tup = ('howteam') # thử một trường hợp khác
>>> str_tup
'howteam'
>>> type(str_tup)
<class 'str'>
```

Vì sao khi khởi tạo một Tuple với một phần tử thì kiểu dữ liệu của Tuple đó lại là kiểu dữ liệu của phần tử duy nhất đó?

- Đó là do khi bạn viết một giá trị nào đó đặt trong cặp dấu ngoặc đơn thì nó được xem là một giá trị.

Vì sao lại phải xem là một giá trị?

- Vì khi ta tính toán, hay sử dụng cặp ngoặc () để được ưu tiên.

```
>>> 1 + 3 * 2 # 3 * 2 sau đó + 1 vì nhân trước cộng sau theo như toán học
7
>>> (1 + 3) * 3 # giờ thi ta sẽ làm phép tính trong ngoặc trước
12
```

Thế nên, trường hợp đó không thể tính là một Tuple. Do đó, khi muốn khởi tạo một Tuple chỉ duy nhất một phần tử, ta phải thêm dấu ` ` vào sau giá trị đó, để báo cho Python biết, đây là Tuple.

```
>>> tup = (9,)
>>> tup
(9,)
>>> type(tup) # kết quả đã như mong đợi
<class 'tuple'>
```

Sử dụng Tuple Comprehension

Với Tuple thì khái niệm Comprehension này không được áp dụng

```
>>> tup = (value for value in range(3))
>>> tup
<generator object <genexpr> at 0x039F5D20>
```

Mà đó được coi là **Generator Expression** (Kteam sẽ giới thiệu trong tương lai).

Đối tượng được tạo từ Generator Expression cũng là một dạng iterable.

Sử dụng constructor Tuple

Cú pháp:

tuple(iterable)

Công dụng: Giống hoàn toàn với việc bạn sử dụng constructor List. Khác biệt duy nhất là constructor Tuple sẽ tạo ra một Tuple.

```
>>> tup = tuple([1, 2, 3])
>>> tup
(1, 2, 3)
>>> str_tup = tuple('KTEAM')
>>> str_tup
('K', 'T', 'E', 'A', 'M')
>>> generator = (value for value in range(10) if value % 2 == 0)
>>> generator # bạn không cần phải cố gắng hiểu khi chưa rõ comprehension
<generator object <genexpr> at 0x039F5D20>
>>> tuple(generator)
(0, 2, 4, 6, 8)
>>> tuple(123)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

Một số toán tử với Tuple trong Python

Các toán tử của Tuple giống với toán tử của chuỗi. Nếu bạn đọc kĩ phần này ở bài List thì bạn sẽ thấy Kteam đề cập là toán tử của List chỉ là gần giống với toán tử của chuỗi. Lý do vì sao sẽ được giải thích trong bài sự khác biệt các toán tử của **hash object** (immutable như chuỗi, Tuple) và **unhash object** (mutable như List)

Toán tử +

```
>>> tup = [1, 2]
>>> tup += ('how', 'kteam')
>>> tup
[1, 2, 'how', 'kteam']
```

Toán tử *

```
>>> tup = tuple('kter') * 3
>>> tup
('k', 't', 'e', 'r', 'k', 't', 'e', 'r', 'k', 't', 'e', 'r')
>>> (1,) * 0
()
>>> (1,) * 3
(1, 1, 1)
```

Toán tử in

```
>>> 1 in (1, 2, 3)
True
>>> 4 in ('k', 'kteam', 9)
False
```

Indexing và cắt Tuple trong Python

Indexing và cắt Tuple hoàn toàn tương tự như với kiểu dữ liệu List. (Nếu chưa biết về List bạn có thể tham khảo qua các bài về [KIỂU DỮ LIỆU LIST TRONG PYTHON](#))

```
>>> tup = (1, 2, 'a', 'b', [3, 4])
>>> len(tup) # lấy số phần tử có trong tuple
5
>>> tup[0]
1
```

```
>>> tup[-1]
[3, 4]
>>> tup[3]
'b'
>>> tup[1:3]
(2, 'a')
>>> tup[:2]
(1, 2)
>>> tup[2:]
('a', 'b', [3, 4])
>>> tup[::-1]
([3, 4], 'b', 'a', 2, 1)
```

Thay đổi nội dung Tuple trong Python

Tuple và chuỗi đều là một dạng **hash object** (immutable). Do đó việc bạn muốn thay đổi nội dung của nó trên lí thuyết là không.

```
>>> tup = ('kter', 'howkteam', 69)
>>> tup[1]
'howkteam'
>>> tup[1] = 'changed'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Vì sao lại nói là trên lí thuyết? Bạn sẽ biết được ngay ở phần sau.

Ma trận

Nếu bạn nắm vững khái niệm này ở List. Thì xin chúc mừng bạn vì không phải đau đầu. Nó hoàn toàn tương tự.

```
>>> tup = ((1, 2, 3), [4, 5])
>>> tup[0]
(1, 2, 3)
>>> tup[0][2]
3
>>> tup[1][-2]
4
```

Tuple có phải luôn luôn là một hash object?

Như đã định nghĩa ở bài chuỗi, một **hash object** là một đối tượng bạn **không thể thay đổi nội dung** của nó. Và trong phần thay đổi nội dung Tuple, bạn cũng thấy ta không thể thay đổi giá trị ở bên trong Tuple. Tuy nhiên, không phải lúc nào cũng vậy.

```
>>> tup = ([1, 2],)
>>> tup[0]
[1, 2]
>>> tup[0][0]
1
>>> tup[0][1]
2
```

Giá trị bên trong tuple đó là một List. Và, List là một **unhash object**. Suy ra, ta có thể thay đổi nội dung của nó.

```
>>> tup[0][0]
1
>>> tup[0][0] = 'howteam'
>>> tup
(['howteam', 2],)
```

Ta đã thay đổi nội dung của Tuple bằng một cách đó là thay đổi nội dung của một phần tử trong Tuple.

Vì thế, một Tuple sẽ được coi là một hash object khi nó chứa các phần tử đều là hash object.

Các phương thức của Tuple

Phương thức count

Cú pháp:

```
<Tuple>.count(value)
```

Công dụng: Trả về một số nguyên, chính là số lần xuất hiện của value trong Tuple.

```
>>> tup = (1, 5, 3, 5, 6, 1, 1)
>>> tup.count(1)
3
>>> tup.count(4)
0
```

Phương thức index

Cú pháp:

```
<Tuple>.index(sub[, start[, end]])
```

Công dụng: Tương tự phương thức index của kiểu dữ liệu chuỗi.

```
>>> tup = (1, 2, 3)
>>> tup.index(2)
1
>>> tup.index(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 4 is not in list
```

Khi nào thì chọn Tuple thay cho List?

Tuple khác List ở chỗ Tuple không cho phép bạn sửa chữa nội dung, còn List thì có. Vì đặc điểm đó, Tuple mạnh hơn List ở những điểm sau:

- **Tốc độ truy xuất** của Tuple **nhanh hơn** so với List
- **Dung lượng** chiếm trong bộ nhớ của Tuple **nhỏ hơn** so với List
- Bảo vệ dữ liệu của bạn sẽ không bị thay đổi
- Có thể dùng làm key của Dictionary (một kiểu dữ liệu sẽ được giới thiệu). Điều mà List không thể vì List là unhash object.

Những điểm trên là những điều giúp bạn có thể cân nhắc việc chọn Tuple hay List để lưu trữ dữ liệu dưới một mảng.

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIẾU DỮ LIỆU LIST TRONG PYTHON – Phần 2](#).

1. Sẽ có lỗi **IndexError**

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop from empty list
```

2. Không. Vì khi đó, ta phải so sánh hai List [1, 2] và ['abc', 'def']. Mà khi so sánh hai List này một cách trực tiếp. Python sẽ phải so sánh từng phần tử của mỗi hai List đó với nhau. Nhưng một bên là số, một bên là chuỗi, nên việc so sánh trực tiếp là không được.

```
>>> lst.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

Câu hỏi củng cố

1. Tìm các cách khởi tạo List hợp lệ dưới đây
 - a. tup = tuple((1,2, 3) + [3, 4])
 - b. tup = (1)
 - c. tup = 1
 - d. tup = 1, 2
2. Dự đoán kết quả của chương đoạn code dưới đây

```
>>> tup = (1, 2, [3, 4])
>>> tup[2] += [50, 60]
```

Lựa chọn phương án đúng

- a. tup = (1, 2, [3, 4, 50, 60])
- b. TypeError: 'tuple' object does not support item assignment
- c. a và b đúng

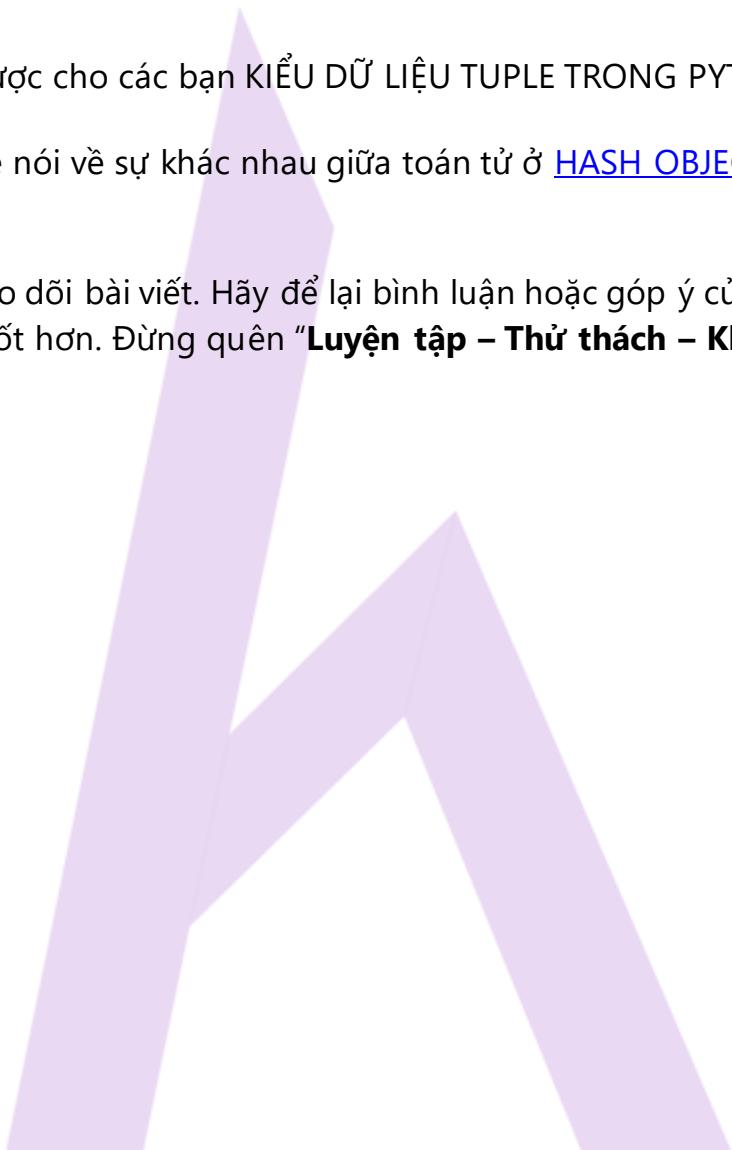
Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Bài viết này đã sơ lược cho các bạn KIỂU DỮ LIỆU TUPLE TRONG PYTHON.

Ở bài sau, Kteam sẽ nói về sự khác nhau giữa toán tử ở [HASH OBJECT VÀ UNHASH OBJECT](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.



Bài 15: SỰ KHÁC NHAU VỀ TOÁN TỬ CỦA HASHABLE OBJECT & UNHASHABLE OBJECT TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Sự khác nhau về toán tử của Hashable object và Unhashable object trong Python](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong các bài trước, Kteam đã giới thiệu đến bạn [KIẾU DỮ LIỆU TUPLE](#), một container thuộc thể loại **hashable object** trong Python

Ở bài này Kteam sẽ nói về **sự khác nhau** của toán tử giữa hai loại kiểu dữ liệu **Hashable Object** (immutable) và **Unhashable Object** (mutable) trong Python.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU SỐ](#) và [KIỂU DỮ LIỆU CHUỖI](#) trong Python.
- [KIỂU DỮ LIỆU LIST](#), [KIỂU DỮ LIỆU TUPLE](#) trong Python

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Giới thiệu cơ bản về hàm id
- Toán tử là một phương thức
- Khác biệt về toán tử Hash Object và Unhash Object
- Tại sao có List lại còn sinh ra Tuple? Hoặc là sử dụng Tuple thôi, cần gì tới List?
- -----

Giới thiệu cơ bản về hàm id

Cú pháp:

id(<giá trị>)

Như Kteam đã từng đề cập ở các bài trước đây, mọi thứ trong Python xoay quanh các đối tượng, và các giá trị ở đây chính là một đối tượng. Tuy vậy vẫn để là **<giá trị>** để tránh gây khó hiểu.

Công dụng: Theo định nghĩa về hàm **id** trong tài liệu của Python thì hàm này sẽ trả về một số nguyên ([int](#) hoặc [longint](#)).

- Giá trị này là một giá trị duy nhất và là hằng số không thay đổi suốt chương trình.
- Trong chi tiết bổ sung của CPython có nói giá trị trả về của hàm id là địa chỉ của giá trị (đối tượng) đó trong bộ nhớ.

Cao siêu là thế, nhưng bạn hoàn toàn có thể nghĩ đơn giản, con số trả về đó như cái số nhà của bạn. Bạn ở đâu, thì số nhà của bạn cũng sẽ tương ứng.

```
>>> n = 69
>>> s = 'How KTeam'
>>> lst = [1, 2]
>>> tup = (3, 4)
>>> id(n)
1446271792
>>> id(s)
53865712
>>> id(lst)
53838352
>>> id(tup)
53865768
>>>
>>> id(123)
1446272656
>>> id('Free Education')
53865832
```

Kteam sẽ tiếp tục giới thiệu hàm id khi nói tới các toán tử so sánh trong Python ở một bài khác.

Toán tử là một phương thức

Lặp lại thêm một lần nữa, mọi thứ xoay quanh Python toàn là hướng đối tượng. Cả các toán tử cũng thế!

```
>>> n = 69
>>> n + 1
70
>>> n
69
>>> n.__add__(1) # tương tự khi bạn n + 1
70
>>> n
69
```

```
>>> n.__sub__(9) # tương tự n - 9  
60  
>>> n.__mul__(2) # tương tự n * 2  
138  
>>> n.__radd__(1) # tương tự 1 + n  
70  
>>> n.__rsub__(9) # tương tự 9 - n  
-60  
>>> n.__neg__() # tương tự -n  
-69
```

Mỗi toán tử của mỗi đối tượng sẽ có toán tử đi kèm.

Khác biệt về toán tử Hash Object và Unhash Object

Vấn đề chính của bài này, là chỉ ra sự khác biệt giữa toán tử ở **hash object** và **unhash object**. Kteam sẽ lấy ví dụ so sánh đơn giản đó chính là sự khác biệt giữa việc **s = s + i** với lại **s += i**

Hãy xem xét đoạn code dưới đây, Kteam sẽ xét một **hash object** là chuỗi:

```
>>> s_1 = 'HowKteam'  
>>> s_2 = 'Free Education'  
>>> id(s_1)  
53866032  
>>> id(s_2)  
53865712  
>>> s_1 = s_1 + ' Python'  
>>> s_2 += ' Python'  
>>> id(s_1) # đã có sự thay đổi  
53866152  
>>> id(s_2) # cũng có sự thay đổi  
23088304  
>>> s_1  
'HowKteam Python'
```

```
>>> s_2  
'Free Education Python'
```

Ta cũng thấy, 2 toán tử `=` + cũng không có gì khác biệt lắm so với `+=`.

Giờ ta xét tới một **unhash object**

```
>>> lst_1 = [1, 2]  
>>> lst_2 = [3, 4]  
>>> id(lst_1)  
53839752  
>>> id(lst_2)  
53864048  
>>> lst_1 = lst_1 + [0]  
>>> lst_2 += [0]  
>>> id(lst_1) # có sự thay đổi  
53864088  
>>> id(lst_2) # không hề có sự thay đổi  
53864048  
>>> lst_1  
[1, 2, 0]  
>>> lst_2  
[3, 4, 0]
```

Đã có khác biệt, khi thử với unhash object. Tại sao lại như vậy?

Đó là vì khi bạn làm như cách dưới đây. Tức có nghĩa bạn vừa mới gán lại giá trị cho biến **lst**. Nói cách khác, bạn đã đưa **lst** tới một địa chỉ khác.

```
>>> lst = [1, 2]  
>>> lst = lst + [3]  
>>> lst  
[1, 2, 3]
```

Còn khi bạn làm như thế này

```
>>> lst = [1, 2]
>>> lst += [3]
>>> lst
[1, 2, 3]
```

Thì không như vậy, bạn đã gián tiếp gọi một phương thức

```
>>> lst = [1, 2]
>>> id(lst)
53839752
>>> lst.__iadd__([3])
[1, 2, 3]
>>> id(lst)
53839752
>>> lst
[1, 2, 3]
```

Vậy vì sao, các hash object lại không như vậy?

Là bởi vì các hash object không hề có phương thức **iadd**, hay **imul** như các **unhash object**. Thế nên, khi bạn dùng toán tử **+=**, Python sẽ làm tương tự như bạn dùng cách gán giá trị.

Vì sao các hash object lại không có phương thức iadd, imul?

Khi bạn khởi tạo một giá trị, nó sẽ được lưu trong bộ nhớ máy tính.

- Với **hash object**, bạn không thể thay đổi nội dung của nó. Do đó, Python sẽ xin đủ khoảng trống để lưu trữ dữ liệu của bạn, không nhiều hơn và cũng không ít hơn. Giúp không hoang phí bộ nhớ của bạn. Thế nên, khi bạn cộng thêm một thứ gì đó, Python không biết nhét cái thứ bạn muốn cộng vào chỗ nào. Nên nó đành cuốn gói đi ra chỗ đó, tìm chỗ mới thoáng có đủ khoảng trống.
- Còn với **unhash object**. Là một đối tượng bạn thay đổi được nội dung, vì thế, Python luôn xin dư bộ nhớ để chứa chỗ cho các giá trị tiếp theo bạn có thể thêm vào. Trong bài trước, Kteam đã đề cập đến việc Tuple

chiếm ít dung lượng hơn List vì Tuple là **hash object**. (Bạn có thể tham khảo chi tiết tại bài [KIẾU DỮ LIỆU TUPLE](#))

Tại sao có List lại còn sinh ra Tuple? Hoặc là sử dụng Tuple thôi, cần gì tới List?

Đáng lẽ, Kteam sẽ nói vấn đề này ở bài trước, nhưng vì muốn bạn hiểu hơn về các **hash object** với **unhash object** nên đã để tới bài này.

Bạn dễ dàng nhận thấy, việc ta thay đổi giá trị của Tuple, không nhất thiết là phải trực tiếp như List.

```
>>> lst = [1, 2]
>>> lst.append(3)
>>> lst
[1, 2, 3]
>>> tup = (1, 2)
>>> tup += (3,)
>>> tup
(1, 2, 3)
```

Các bạn cũng thấy, nó không khác nhau là mấy. Ta cũng có thể tạo ra các hàm thay đổi nội dung của Tuple bằng cách **slicing**. Đã thế List lại còn nặng về việc chiếm nhiều dung lượng hơn Tuple, truy xuất chậm hơn Tuple. Việc gì khiến nó còn được trọng dụng?

Vì khi bạn thay đổi Tuple như cách trên, Python phải đi vòng vòng trong bộ nhớ của bạn tìm chỗ nào trống, phù hợp để chứa cái Tuple của bạn không, trong khi với List thì không. Do đó, bạn phải biết được dữ liệu của bạn là dạng dữ liệu như thế nào, có cần phải thay đổi không. Dựa vào đó, để chọn ra một kiểu dữ liệu phù hợp cho mình, tối ưu hóa dung lượng sử dụng, thời gian truy xuất.

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIỂU DỮ LIỆU TUPLE TRONG PYTHON](#).

1. Chỉ có **d** là cách khởi tạo đúng. Bạn sẽ hiểu được khái niệm này khi biết tới **Unpacking** và **Packing argument** sẽ được Kteam giới thiệu trong tương lai.
2. c đúng

Nếu bạn thắc mắc vì sao có lỗi. Trong khi ví dụ ở phần “Có phải Tuple luôn là một hash object?” thì lại không có lỗi?

Lí là do vì trong ví dụ phần “Có phải Tuple luôn là một hash object?”. Việc thay đổi nội dung List trong Tuple như thế thì Python chỉ làm việc duy nhất với List đó. Không liên quan gì đến Tuple chứa nó.

Riêng ở câu hỏi này, Python đã làm như thế này

- Đưa phần tử tup[2] lên TOS (Top of stack)
- Gán TOS (chính là List [3, 4]) bằng việc cộng thêm cho List đó một List nữa là [50, 60]. Suy ra, List bây giờ đang là [3, 4, 50, 60]
- Sau đó, Python gán lại tup[2] = TOS. Và dĩ nhiên bạn cũng biết, Tuple không thể làm như vậy.

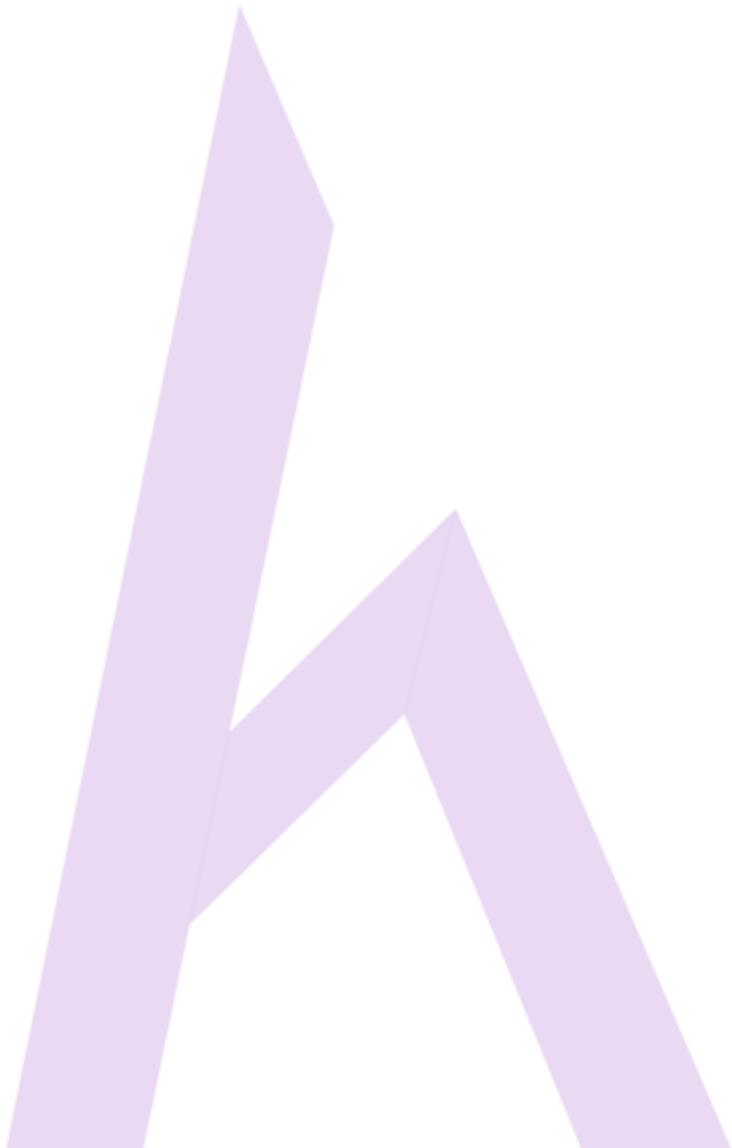
Có thể bạn chưa nắm được kiến thức này, nhưng bạn sẽ thấy nó không hề khó khi đã theo dõi phần hàm id ở đầu bài này.

Kết luận

Bài viết này đã cho bạn biết được cách hoạt động của các toán tử trong Python và một vài sự khác biệt

Ở bài sau, Kteam sẽ nói về một kiểu dữ liệu nữa, đó chính là [KIẾU DỮ LIỆU SET trong Python](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".



Bài 16: KIỂU DỮ LIỆU SET TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Set trong Python](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong các bài trước, Kteam đã giới thiệu đến bạn một số container của Python.

Ở bài này Kteam sẽ giới thiệu tới bạn một container khác đó chính **KIỂU DỮ LIỆU SET** trong Python.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU SỐ](#), [KIỂU DỮ LIỆU CHUỖI](#) trong Python
- [KIỂU DỮ LIỆU LIST](#), [KIỂU DỮ LIỆU TUPLE](#) trong Python.

Trong bài này, chúng ta sẽ cùng tìm hiểu các vấn đề

- Giới thiệu về Set trong Python
- Cách khởi tạo Set
- Một số toán tử với Tuple trong Python

- Indexing và cắt Set trong Python
 - Các phương thức của Set
 - Set không phải là một hash object
-

Giới thiệu về Set trong Python

Set là một container, tuy nhiên không được sử dụng nhiều bằng [LIST](#) hay [TUPLE](#).

Một Set gồm các yếu tố sau:

- Được giới hạn bởi cặp ngoặc {}, tất cả những gì nằm trong đó là những phần tử của Set.
- Các phần tử của Set được phân cách nhau ra bởi dấu phẩy (,).
- Set không chứa nhiều hơn 1 phần tử trùng lặp

Set chỉ có thể chứa các **hashable object** nhưng chính nó **không phải** là một **hashable object**. Do đó, bạn không thể chứa một set trong một set.

Ví dụ:

```
>>> set_1 = {69, 96}
>>> set_1
{96, 69}
>>> type(set_1) # kiểu set thuộc lớp set
<class 'set'>
>>> set_2 = {'How Kteam'}
>>> set_2
{'How Kteam'}
>>> set_3 = {(69, 'Free Education'), (1, 2, 3)}
>>> set_3
{(69, 'Free Education'), (1, 2, 3)}
>>> set_4 = {[1, 2], [3, 4]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> set_5 = {(1, 2, ['How Kteam'])}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: unhashable type: 'list'
>>> set_6 = {1, 2, {'HowKteam'}}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Cách khởi tạo Set

Sử dụng cặp dấu ngoặc {} và đặt giá trị bên trong

Cú pháp:

{<giá trị thứ nhất>, <giá trị thứ hai>, ..., <giá trị thứ n – 1>, <giá trị thứ n>}

Lưu ý: Khi khởi tạo bằng cách này, ít nhất phải có một giá trị.

Ví dụ:

```
>>> set_ = {1, 2, 3, 4}
>>> set_
{1, 2, 3, 4}
>>> set_1 = {1, 1, 1} # các giá trị trùng lặp bị loại bỏ
>>> set_1
{1}
>>> empty_set = {} # thử khởi tạo set rỗng
>>> empty_set
{}
>>> type(empty_set) # không phải là set
<class 'dict'>
```

Sử dụng Set Comprehension

```
>>> set_1 = {value for value in range(3)}  
>>> set_1  
{0, 1, 2}
```

Sử dụng constructor Set

Cú pháp:

set(iterable)

Công dụng: Giống hoàn toàn với việc bạn sử dụng constructor **List**. Khác biệt duy nhất là constructor Set sẽ tạo ra một Set.

Ví dụ:

```
>>> set_1 = set((1, 2, 3))  
>>> set_1  
{1, 2, 3}  
>>> set_2 = set('How Kteam')  
>>> set_2 # set không quan tâm đến vị trí của các phần tử  
{'o', ' ', 'a', 'm', 'H', 'K', 't', 'w', 'e'}  
>>> set_3 = set('aaaaaaaaaa')  
>>> set_3  
{'a'}  
>>> set_4 = set([1, 6, 8, 3, 1, 1, 3, 6])  
{8, 1, 3, 6}  
>>> empty_set = set() # cách bạn tạo được empty set  
>>> empty_set  
set()
```

Một số toán tử với Set trong Python

Nhằm giúp các bạn dễ hiểu hơn về các toán tử với Set trong Python, Kteam minh họa các set dưới dạng biểu đồ Venn, với S1, S2 tương ứng các Set1, Set2 chứa các phần tử.

Toán tử in

Cú pháp:

```
value in <Set>
```

Công dụng: Kết quả trả về là True nếu value xuất hiện trong Set. Ngược lại sẽ là False

Ví dụ:

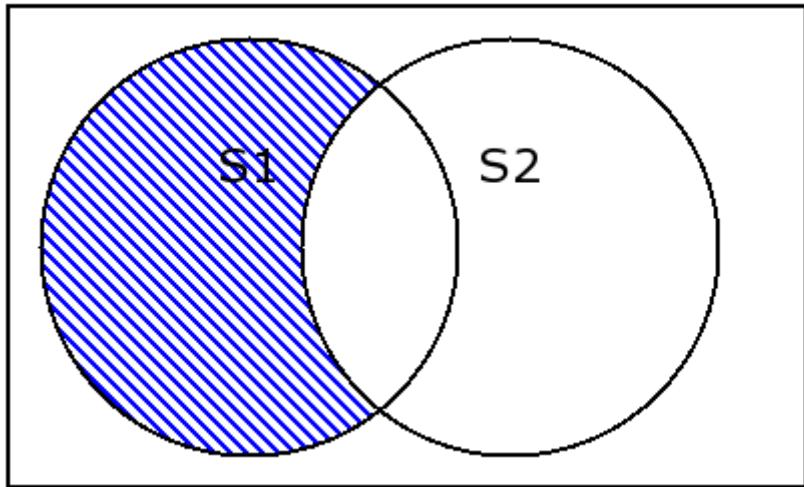
```
>>> 1 in {1, 2, 3}  
True  
>>> 4 in {'a', 'How Kteam', 5}  
False
```

Toán tử -

Cú pháp:

```
<Set1> - <Set2>
```

Công dụng: Kết quả trả về là một Set gồm các phần tử chỉ tồn tại trong Set1 mà không tồn tại trong Set2



Ví dụ:

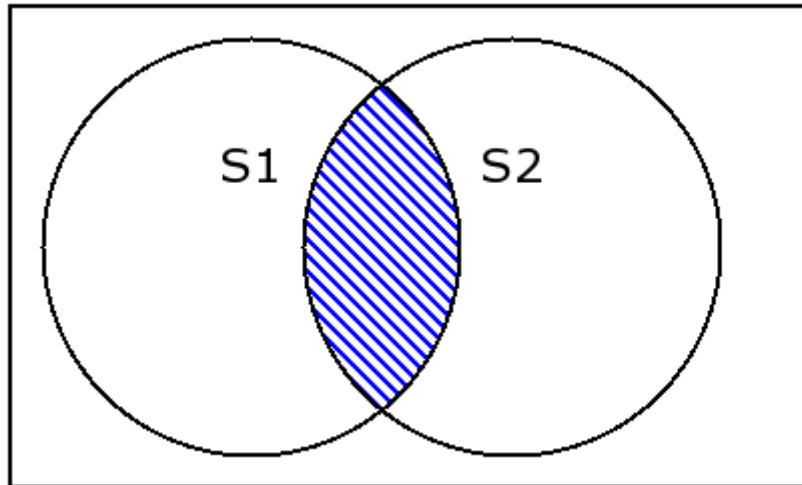
```
>>> {1, 2, 3} - {2, 3}  
{1}  
>>> {1, 2, 3} - {4}  
{1, 2, 3}  
>>> {1, 2, 3} - {1, 2, 3}  
set()  
>>> {1, 2, 3} - {1, 2, 3, 4}  
set()
```

Toán tử &

Cú pháp:

<Set1> <Set2>

Công dụng: Kết quả trả về là một Set chứa các phần tử vừa tồn tại trong Set1 và vừa tồn tại trong Set2



Ví dụ:

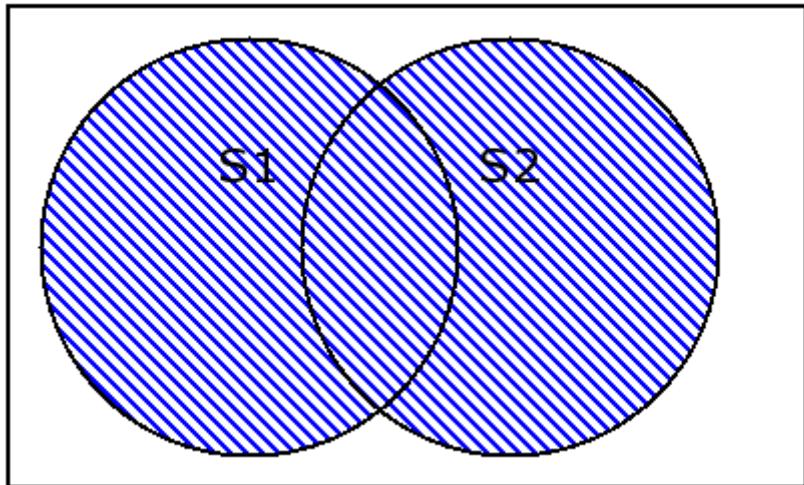
```
>>> {1, 2, 3} & {4, 5}  
set()  
>>> {1, 2, 3} & {1, 4, 5}  
{1}  
>>> {1, 2, 3} & {1, 2, 3}  
{1, 2, 3}
```

Toán tử |

Cú pháp:

```
<Set1> | <Set2>
```

Công dụng: Kết quả trả về là một Set chứa tất cả các phần tử tồn tại trong hai Set



Ví dụ:

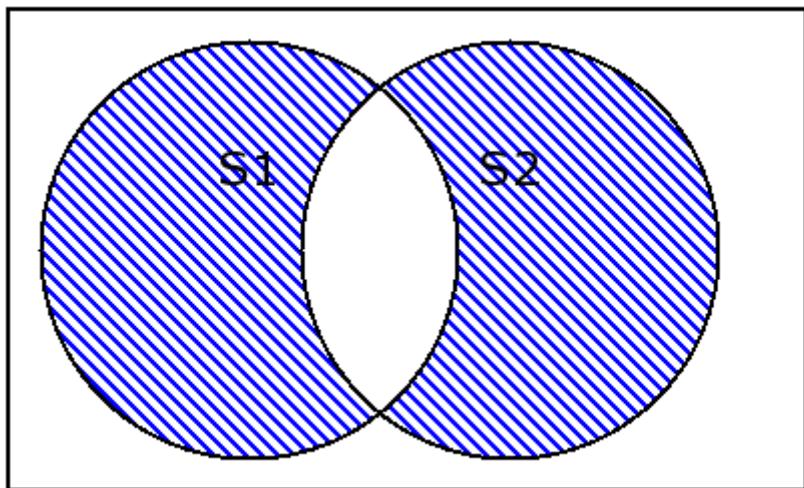
```
>>> {1, 2, 3} | {1, 2, 3}  
{1, 2, 3}  
>>> {1, 2, 3} | {4, 5}  
{1, 2, 3, 4, 5}
```

Toán tử ^

Cú pháp:

```
<Set1> ^ <Set2>
```

Công dụng: Kết quả trả về là một Set chứa tất cả các phần tử chỉ tồn tại ở một trong hai Set



Ví dụ:

```
>>> {1, 2, 3} ^ {4, 5}  
{1, 2, 3, 4, 5}  
>>> {1, 2, 3} ^ {1, 2, 3}  
set()  
>>> {1, 2, 3} ^ {1, 4}  
{2, 3, 4}
```

Indexing và cắt Set trong Python

Ở trên Kteam đã đề cập về việc set không quan tâm đến vị trí của phần tử nằm trong set. Nên, việc indexing và cắt set trong Python không được hỗ trợ.

Các phương thức của Set

Set cũng có khá nhiều phương thức. Nhưng Kteam chỉ giới thiệu một số phương thức cơ bản.

Phương thức clear

Cú pháp:

```
<Set>.clear()
```

Công dụng: Loại bỏ hết tất cả các phần tử có trong Set

Ví dụ:

```
>>> set_1= {1, 2}  
>>> set_1.clear()  
>>> set_1  
set()
```

Phương thức pop

Cú pháp:

```
<Set>.pop()
```

Công dụng: Kết quả trả về một giá trị được lấy ra từ Set, đồng thời loại bỏ giá trị đã lấy ra khỏi Set ban đầu

- Nếu là set rỗng, sẽ có lỗi

Ví dụ:

```
>>> set_1 = {1, 2}  
>>> set_1.pop()  
1  
>>> set_1
```

```
{2}
>>> set_1.pop()
2
>>> set_1
set()
>>> set_1.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

Lưu ý: trong một số trường hợp, bạn sẽ pop được các giá trị từ set ra từ bé đến lớn. Nhưng đó không phải bản chất của nó, việc pop này liên quan đến các giá trị của hàm hash trong của các phần tử. Đó là lí do set chỉ chứa các phần tử là các hashable object. Vì kiến thức này không quan trọng ở mức cơ bản nên Kteam xin phép được bỏ qua.

Phương thức remove

Cú pháp:

```
<Set>.remove(value)
```

Công dụng: Loại bỏ giá trị value ở trong Set. Nếu như value không ở trong Set, thông báo lỗi **KeyError**.

Ví dụ:

```
>>> a = {1, 2}
>>> a.remove(1)
>>> a
{2}
>>> a.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
KeyError: 3
```

Phương thức discard

Cú pháp:

```
<Set>.discard(value)
```

Công dụng: Loại bỏ giá trị `value` ở trong Set. Nếu như `value` không ở trong Set, thì sẽ bỏ qua.

Ví dụ:

```
>>> a = {1, 2}
>>> a.discard(1)
>>> a
{2}
>>> a.discard(4)
>>> a
{2}
```

Phương thức copy

Cú pháp:

```
<Set>.copy()
```

Công dụng: Trả về một bản sao của Set

Ví dụ:

```
>>> a = {1, 2}  
>>> b = a.copy()  
>>> b  
{1, 2}  
>>> a  
{1, 2}
```

Phương thức add

Cú pháp:

```
<Set>.add(value)
```

Công dụng: Thêm value vào trong set. Nếu như **value** đã có trong Set thì bỏ qua.

Ví dụ:

```
>>> a = {1, 2}  
>>> a.add(3)  
>>> a  
{1, 2, 3}  
>>> a.add(2)  
>>> a  
{1, 2, 3}
```

Set không phải là một hash object

Đúng như vậy! Điều đó có thể chứng minh theo hai cách:

Ở ví dụ dưới, bạn cũng thấy, ta đã thay đổi nội dung của set nhưng id của set vẫn là id ban đầu

Ví dụ:

```
>>> a = {1, 2}  
>>> id(a)  
52255360  
>>> a.add(3)  
>>> id(a)  
52255360
```

Thêm nữa, set không thể chứa một set khác

```
>>> a = {1, 2}  
>>> b = {a}  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unhashable type: 'set'
```

Củng cố bài học

Câu hỏi củng cố

Giải thích lí do tại sao lại có sự thay đổi ở set a? Cho giải pháp khắc phục?

```
>>> a = {1, 2}  
>>> b = a  
>>> b.clear()  
>>> a # tại sao lại trở thành set rỗng?  
set()
```

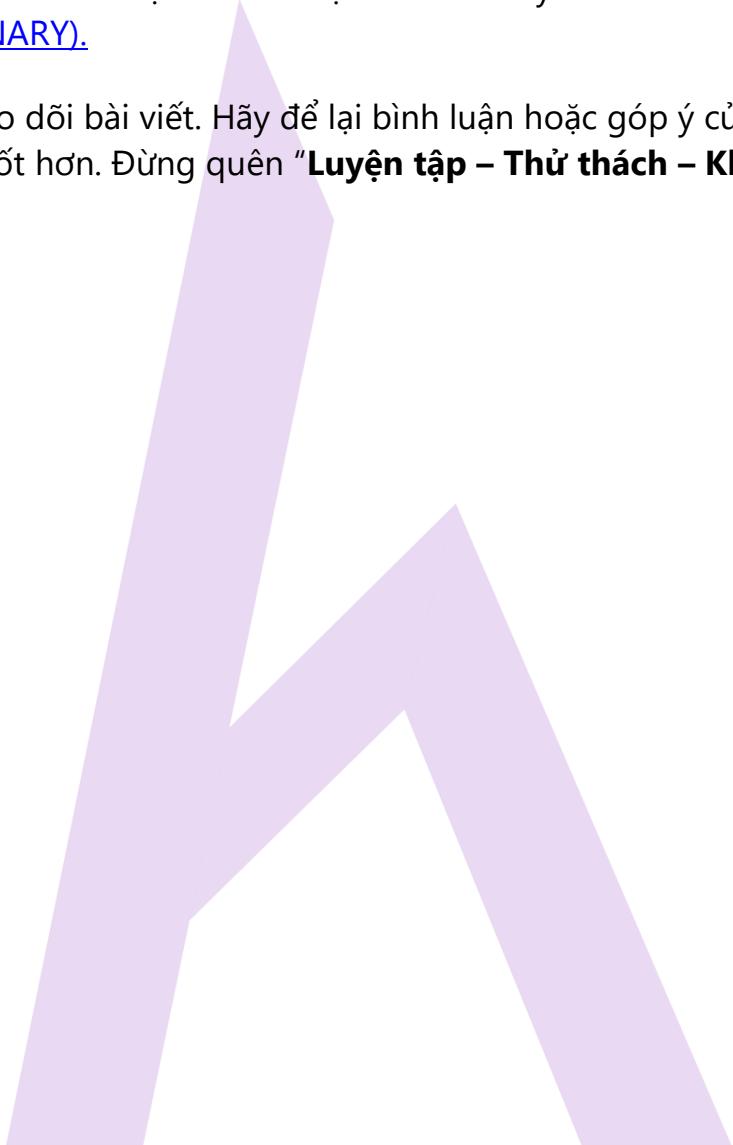
Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Bài viết này đã giới thiệu cho các bạn KIỂU DỮ LIỆU SET TRONG PYTHON.

Ở bài sau, Kteam sẽ nói về một kiểu dữ liệu khác của Python chính là [KIỂU DỮ LIỆU DICT \(DICTIONARY\)](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thử thách – Không ngại khó**”.



Bài 17: KIỂU DỮ LIỆU DICT TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Dict trong Python](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong các bài trước, Kteam đã giới thiệu đến bạn [KIỂU DỮ LIỆU SET](#) trong Python

Ở bài này Kteam sẽ đề cập đến các bạn **KIỂU DỮ LIỆU DICT** trong Python. Một trong những kiểu dữ liệu cực kì quan trọng trong Python.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU SỐ](#), [KIỂU DỮ LIỆU CHUỖI](#) trong Python
- [KIỂU DỮ LIỆU LIST](#), [KIỂU DỮ LIỆU TUPLE](#), [KIỂU DỮ LIỆU SET](#) trong Python.

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Giới thiệu về Dict trong Python
 - Cách khởi tạo Dict
 - Lấy value trong Dict bằng key
 - Thay đổi nội dung Dict trong Python
 - Thêm thủ công một phần tử vào Dict
-

Giới thiệu về Dict trong Python

Dict(Dictionary) cũng là một container như [LIST](#), [TUPLE](#). Có điều khác biệt là những container như List, Tuple **có các index** để phân biệt các phần tử thì Dict dùng các **key** để phân biệt.

Chắc bạn cũng dùng từ điển tiếng Anh để tra từ vựng rồi nhỉ? Có rất nhiều từ vựng trong đó nhưng mà không từ vựng nào giống nhau. Có chăng chúng chỉ giống nhau về nghĩa? Và đó cũng như Dict(Dictionary) hoạt động trong Python

Một Dict gồm các yếu tố sau:

- Được giới hạn bởi cặp ngoặc nhọn {}, tất cả những gì nằm trong đó là những phần tử của Dict.
 - Các phần tử của Dict được phân cách nhau ra bởi dấu phẩy ,.
 - Các phần tử của Dict phải là một cặp **key-value**
 - Cặp **key-value** của phần tử trong Dict được phân cách bởi dấu hai chấm :
 - Các key buộc phải là một **hash object**
-

Cách khởi tạo Dict

Sử dụng cặp dấu ngoặc {} và đặt giá trị bên trong

Cú pháp:

```
{<key_1: value_1>, <key_2: value_2>, .., <key_n: value_n>}
```

Ví dụ:

```
>>> dic = {'name': 'Kteam', 'member': 69}
>>> dic
{'name': 'Kteam', 'member': 69}
>>> empty_dict = {} # khởi tạo dict rỗng
>>> empty_dict
{}
>>> type(dic) # kiểu dữ liệu dict thuộc lớp 'dict'
<class 'dict'>
```

Sử dụng Dict Comprehension

Ví dụ:

```
>>> dic = {key: value for key, value in [('name', 'Kteam'), ('member', 69)]}
>>> dic
{'name': 'Kteam', 'member': 69}
```

Sử dụng constructor Dict

Với dict, ta có 4 cách để khởi tạo một Dict bằng constructor:

Khởi tạo một Dict rỗng

Cú pháp:

dict()

Ví dụ:

```
>>> dic = dict()  
>>> dic  
{}
```

Khởi tạo một dict từ một mapping object

Cú pháp:

dict(mapping)

Trong đó:

Mapping object cũng gần giống so với dict object.

- Một object là **Mapping object** khi có đủ hai phương thức **keys** và **__getitem__**.
- **Dict object** cũng là một **mapping object**. Tuy nhiên, không phải mapping object nào cũng là dict object vì dict object không chỉ có hai phương thức keys và __getitem__ và còn nhiều phương thức khác.

Bạn có thể bỏ qua ví dụ bên dưới hoặc xem để tham khảo vì phần này có thể khá khó hiểu.

Ví dụ:

```
>>> class Map_Class:  
...     def keys(self):  
...         return [1, 2, 3]  
...     def __getitem__(self, key):  
...         return key * 2  
...  
>>> map_obj = Map_Class()  
>>> dic = dict(map_obj)  
>>> dic  
{1: 2, 2: 4, 3: 6}
```

Khởi tạo bằng iterable

Cú pháp:

dict(iterable**)**

Trong đó:

iterable này đặc biệt hơn các **iterable** mà bạn dùng để khởi tạo List hay Tuple, đó là các phần tử trong iterable phải có 2 value đó chính là **key-value**.

Bạn có thể dùng List, Tuple hoặc bất cứ container nào (trừ mapping object) để chứa cặp **key-value**.

```
>>> iter_ = [('name', 'Kteam'), ('member', 69)]  
>>> dic = dict(iter_)  
>>> dic  
{'name': 'Kteam', 'member': 69}
```

Khởi tạo bằng keyword arguments

Cú pháp:

dict(**kwargs)

Trong đó:

Bạn chưa tìm hiểu đến hàm, nên khái niệm **keyword arguments** vẫn còn rất xa lạ!

- Cú hiểu đơn giản là giống như việc bạn khởi tạo biến và giá trị rồi đưa cho dict đó giữ giùm.
- Một lưu ý là những biến này không bị ảnh hưởng hoặc ảnh hưởng gì đến các biến bên ngoài

Ví dụ:

```
>>> name = 'SpaceX'  
>>> member = 696969  
>>> dic = dict(name='Kteam', member=69)  
>>> dic  
{'name': 'Kteam', 'member': 69}  
>>> name  
'SpaceX'  
>>> member  
696969
```

Sử dụng Phương thức fromkeys

Cú pháp:

dict.fromkeys(iterable, value)

Công dụng: Cách này cho phép ta khởi tạo một dict với các **keys** nằm trong một **iterable**. Các giá trị này đều sẽ nhận được một giá trị với mặc định là **None**

Ví dụ:

```
>>> iter_ = ('name', 'number')
>>> dic_none = dict.fromkeys(iter_)
>>> dic_none
{'name': None, 'number': None}
>>> dic = dict.fromkeys(iter_, 'non None value')
>>> dic
{'name': 'non None value', 'number': 'non None value'}
```

Lấy value trong Dict bằng key

Cú pháp:

Your_dict[key]

Ví dụ:

```
>>> dic # ta có một dict như sau
{'name': 'Kteam', 'member': 69}
>>> dic['name']
'Kteam'
>>> dic['member']
69
>>> dic['non_exist'] # hãy chắc chắn rằng key bạn dùng có trong dict
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'non_exist'
```

Thay đổi nội dung Dict trong Python

Dict là một **unhashable object**. Do đó, chắc bạn cũng biết ta có thể thay đổi được nội dung nó hay không. Nếu bạn nào nhanh trí, chắc cũng đã biết được cách thay đổi rồi. Tương tự như List thôi!

Ví dụ:

```
>>> dic # ta có một dict như sau  
{'name': 'Kteam', 'member': 69}  
>>> dic['name'] = 'How Kteam'  
>>> dic  
{'name': 'How Kteam', 'member': 69}  
>>> dic['member'] = dic['member'] + 1  
>>> dic  
{'name': 'How Kteam', 'member': 70}
```

Thêm thủ công một phần tử vào dict

Cách này khá giống với cách bạn thay đổi nội dung của Dict. Khác ở chỗ, bây giờ bạn sẽ sử dụng một **key** chưa hề có trong dict.

Ví dụ:

```
>>> dic # ta có một dict như sau  
{'name': 'Kteam', 'member': 69}  
>>> dic['slogan'] = 'Free Education'  
>>> dic  
{'name': 'Kteam', 'member': 69, 'slogan': 'Free Education'}
```

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIỂU DỮ LIỆU SET TRONG PYTHON](#).

Vì khi:

```
>>> a = {1, 2}  
>>> b = a
```

Ta đã cho a và b cùng trỏ vào một chỗ. Do đó thay đổi b thì a cũng sẽ bị tác động.

Muốn giải quyết chuyện này ta nên sử dụng phương thức copy

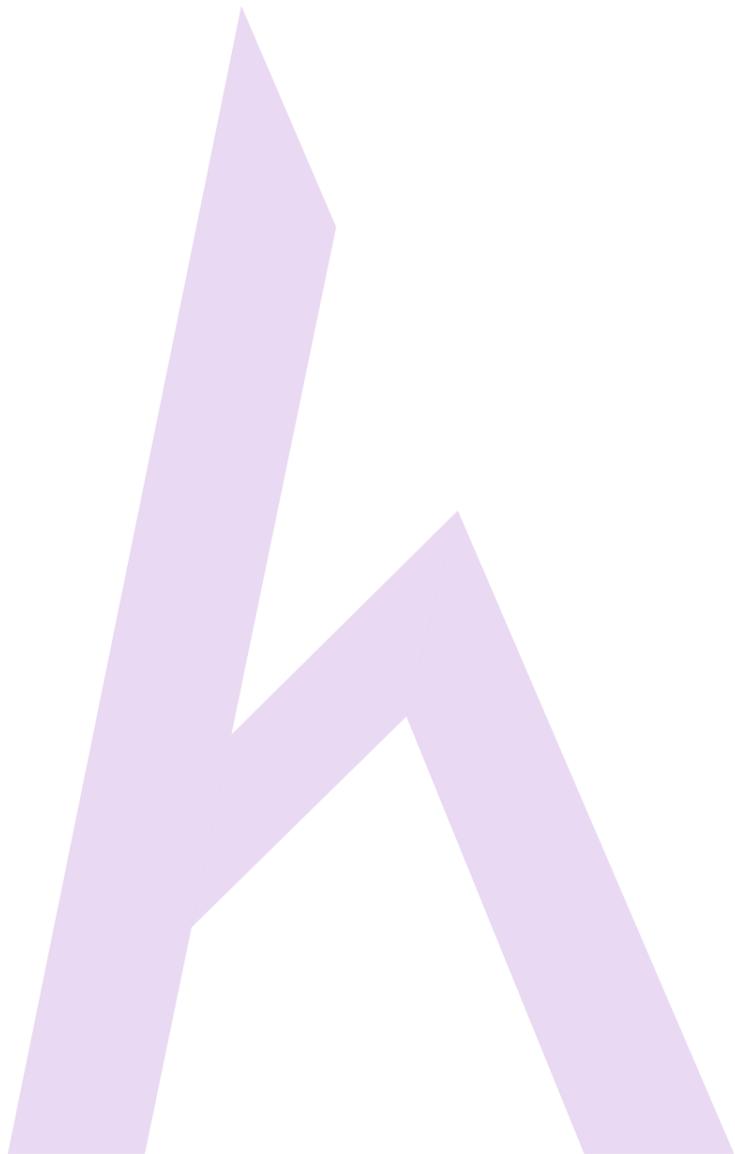
```
>>> a = {1, 2}  
>>> b = a.copy()  
>>> b.clear()  
>>> b  
set()  
>>> a  
{1, 2}
```

Kết luận

Bài viết này đã giới thiệu cho các bạn cơ bản về KIỂU DỮ LIỆU DICT TRONG PYTHON.

Ở bài sau, Kteam sẽ đề cập về các phương thức của [KIỂU DỮ LIỆU DICT - Phần 2](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".



Bài 18: KIỂU DỮ LIỆU DICT TRONG PYTHON -

Phần 2

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Dict trong Python - Phần 2](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong các bài trước, Kteam đã giới thiệu đến bạn [KIỂU DỮ LIỆU DICT](#) trong Python

Ở bài này Kteam sẽ nói về **CÁC PHƯƠNG THỨC CỦA KIỂU DỮ LIỆU DICT** trong Python.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU SỐ](#), [KIỂU DỮ LIỆU CHUỖI](#) trong Python

- [KIỂU DỮ LIỆU LIST](#), [KIỂU DỮ LIỆU TUPLE](#), [KIỂU DỮ LIỆU SET](#) trong Python.
- [KIỂU DỮ LIỆU DICT](#) trong Python

Trong bài này, chúng ta sẽ cùng tìm hiểu những nội dung sau đây

- Giới thiệu về phương thức của kiểu dữ liệu Dict trong Python
- Các phức thức tiện ích
- Các phương thức xử lí

Giới thiệu về phương thức của kiểu dữ liệu Dict trong Python

Kiểu dữ liệu Dict có hỗ trợ một số phương thức và đa số là xử lí các dữ liệu có trong Dict.

Mình mong các bạn sẽ hiểu rõ được các phương thức để sau này áp dụng vào giải quyết các vấn đề với việc viết ít code nhất, hạn chế lỗi nhất.

Các phương thức tiện ích

Phương thức copy

Cú pháp:

<Dict>.copy()

Công dụng: Giống với phương thức copy trong [LIST](#). Để làm gì thì chắc các bạn cũng có thể suy nghĩ ra.

Ví dụ:

```
>>> d = {'team': 'Kteam', (1, 2): 69}  
>>> d_2 = d.copy()  
>>> d_2  
{'team': 'Kteam', (1, 2): 69}  
>>> d  
{'team': 'Kteam', (1, 2): 69}
```

Phương thức clear

Cú pháp:

<Dict>.clear()

Công dụng: Loại bỏ tất cả những phần tử có trong Dict

Ví dụ:

```
>>> d = {'team': 'Kteam', (1, 2): 69}  
>>> d.clear()  
>>> d  
{}
```

Các phương thức xử lí

Phương thức get

Cú pháp:

```
<Dict>.get(key [,default])
```

Công dụng: Trả về giá trị của khóa **key**. Nếu **key** không có trong Dict thì trả về giá trị **default**. **Default** có giá trị mặc định là **None** nếu chúng ta không truyền vào.

Ví dụ:

```
>>> d = {'team': 'Kteam', (1, 2): 69}
>>> d.get('team')
'Kteam'
>>> d.get('a')
>>> d.get('a', 'haha')
'haha'
```

Phương thức items

Cú pháp:

```
<Dict>.items()
```

Công dụng: Trả về một giá trị thuộc lớp **dict_items**. Các giá trị của **dict_items** sẽ là một tuple với giá trị thứ nhất là **key**, giá trị thứ hai là **value**.

- **Dict_items** là một **iterable**.

Ví dụ:

```
>>> d = {'team': 'Kteam', (1, 2): 69}
>>> items = d.items()
>>> items
dict_items([('team', 'Kteam'), ((1, 2), 69)])
>>> type(items)
<class 'dict_items'>
>>> list_items = list(items)
>>> list_items
[('team', 'Kteam'), ((1, 2), 69)]
>>> list_items[0]
('team', 'Kteam')
>>> list_items[0][1]
'Kteam'
```

Phương thức keys

Cú pháp:**<Dict>.keys()**

Công dụng: Trả về một giá trị thuộc lớp `dict_keys`. Các giá trị của `dict_keys` sẽ là các **key** trong Dict.

- `Dict_keys` là một **iterable**.

Ví dụ:

```
>>> d = {'team': 'Kteam', (1, 2): 69}
>>> keys = d.keys()
>>> keys
dict_keys(['team', (1, 2)])
>>> type(keys)
<class 'dict_keys'>
>>> list_keys = list(keys)
```

```
>>> list_keys  
['team', (1, 2)]  
>>> list_keys[-2]  
'team'
```

Phương thức values

Cú pháp:

```
<Dict>.values()
```

Công dụng: Trả về một giá trị thuộc lớp `dict_values`. Các giá trị của `dict_values` sẽ là các **value** trong Dict.

- `Dict_values` là một iterable.

Ví dụ:

```
>>> d = {'team': 'Kteam', (1, 2): 69}  
>>> values = d.values()  
>>> values  
dict_values(['Kteam', 69])  
>>> type(values)  
<class 'dict_values'>  
>>> list_values = list(values)  
>>> list_values  
['Kteam', 69]
```

Phương thức pop

Cú pháp:

```
<Dict>.pop(key [,default])
```

Công dụng: Bỏ đi phần tử có **key** và trả về **value** của **key** đó. Trường hợp **key** không có trong dict.

- Báo lỗi **KeyError** nếu **default** là **None** (ta không thêm vào).
- Trả về **default** nếu ta thêm default vào.

Ví dụ:

```
>>> d = {'team': 'Kteam', (1, 2): 69}
>>> d.pop('team')
'Kteam'
>>> d
{(1, 2): 69}
>>> d.pop('non-exist')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'non-exist'
>>> d.pop('non-exist', 'defaul_value')
'defaul_value'
```

Phương thức popitem()

Cú pháp:

```
<Dict>.popitem()
```

Công dụng: Trả về một 2-tuple với **key** và **value** tương ứng bất kì (vấn đề này liên quan đến giá trị của hash của key. Do đó bạn cũng hiểu vì sao key buộc phải là một **hash object**) trong Dict. Và cặp **key-value** sẽ bị loại bỏ ra khỏi Dict.

- Nếu Dict là một empty Dict. Sẽ có lỗi **KeyError**

Ví dụ:

```
>>> d = {'team': 'Kteam', (1, 2): 69}
>>> d.popitem()
((1, 2), 69)
>>> d
{'team': 'Kteam'}
>>> d.popitem()
('team', 'Kteam')
>>> d.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

Phương thức setdefault

Cú pháp:

<Dict>.setdefault(key [,default])

Công dụng: Trả về giá trị của **key** trong Dict. Trường hợp **key** không có trong Dict thì sẽ trả về giá trị **default**. Thêm nữa, một cặp **key-value** mới sẽ được thêm vào Dict với **key** bằng **key** và **value** bằng **default**.

- Default mặc định là **None**

Ví dụ:

```
>>> d = {'team': 'Kteam', (1, 2): 69}
>>> d.setdefault('team')
'Kteam'
>>> d.setdefault('non-exist_1')
>>> d
{'team': 'Kteam', (1, 2): 69, 'non-exist_1': None}
>>> d.setdefault('non-exists_2', 'default_value')
'default_value'
>>> d
{'team': 'Kteam', (1, 2): 69, 'non-exist_1': None, 'non-exists_2': 'default_value'}
```

Phương thức update

Cú pháp:

<D>.update([E,]F)**

Công dụng: Phương thức giúp bạn cập nhật nội dung cho Dict.

- **F** là một Dict được tạo thành bởi **packing arguments** (khái niệm sẽ được Kteam giải thích ở một bài trong tương lai). Và sẽ thêm vào Dict bằng cách:

```
for k in F: D[k] = F[k]
```

- Nếu **E** được truyền vào và đối tượng **E** có phương thức **keys()**, thì sẽ cập nhật Dict bằng cách:

```
for k in E: D[k] = E[k]
```

- Nếu **E** được truyền vào và đối tượng **E**, đối tượng này có các giá trị là một container chứa hai giá trị thì sẽ cập nhật Dict bằng cách.

```
for k, v in E: D[k] = v
```

Nếu bạn đọc xong và không hiểu gì, thì cũng đừng thất vọng. Kteam sẽ cho bạn vài ví dụ minh họa. Nó rất đơn giản.

Đây là update theo kiểu sử dụng **packing arguments**.

```
>>> d = {'a': 1}
>>> d
{'a': 1}
>>> d.update(b=2,c=3)
>>> d
{'a': 1, 'b': 2, 'c': 3}
```

Đây là cách bạn truyền E với E là một đối tượng có phương thức **keys**

```
>>> d = {'a': 1}
>>> E = {'b': 2, 'c': 3}
>>> d.update(E)
>>> d
{'a': 1, 'b': 2, 'c': 3}
```

Đây là truyền vào một E với E có các giá chứa hai giá trị

```
>>> d = {'a': 1}
>>> E = [('b', 2), ('c', 3)]
>>> d.update(E)
>>> d
{'a': 1, 'b': 2, 'c': 3}
>>> E_f = [('d', 69), ('e', 96)]
>>> d.update(E_f)
>>> d
{'a': 1, 'b': 2, 'c': 3, 'd': 69, 'e': 96}
```

Củng cố bài học

Câu hỏi巩固

- Tại sao thay đổi dict2 mà dict1 lại cũng bị thay đổi theo? Hãy cho giải pháp khắc phục

```
>>> dict1 = {'key': 6969}  
>>> dict1  
{'key': 6969}  
>>> dict2 = dict1  
>>> dict2  
{'key': 6969}  
>>> dict2['key'] = 'changed'  
>>> dict2  
{'key': 'changed'}  
>>> dict1  
{'key': 'changed'}
```

- Nêu sự khác nhau giữa

```
>>> d = {}  
>>> d.update({'a': 3})  
  
và  
  
>>> d = {}  
>>> d.update(3)
```

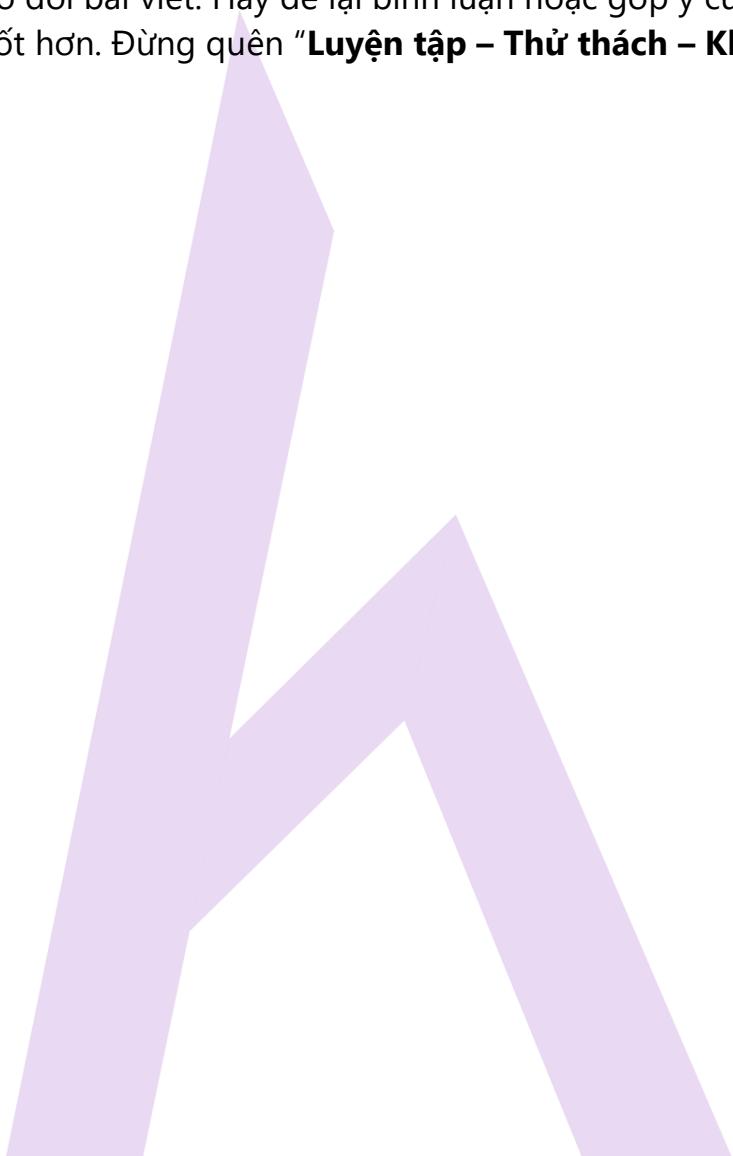
Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để巩固 kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Qua bài viết này, Bạn đã hiểu thêm về DICT qua các phương thức của nó có.

Ở bài viết sau. Kteam hướng dẫn các bạn [XỬ LÝ FILE TRONG PYTHON](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".



Bài 19: XỬ LÝ FILE TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Xử lý file trong Python](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn các phương thức của [KIỂU DỮ LIỆU DICT](#) trong Python

Ở bài này Kteam sẽ giới thiệu với các bạn **CÁCH XỬ LÝ FILE** trong Python. Một trong những điều thiết yếu mà bất cứ ngôn ngữ lập trình nào bạn cũng đều phải tìm hiểu.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU LIST](#), [KIỂU DỮ LIỆU TUPLE](#), [KIỂU DỮ LIỆU SET](#) trong Python.

Trong bài này, chúng ta sẽ cùng tìm hiểu những nội dung sau đây

- Khái quát về File trong Python
- Mở File trong Python

- Đóng File trong Python
 - Đọc File trong Python
 - Ghi File trong Python
 - Kiểm soát con trỏ File
 - Câu lệnh with
-

Khái quát về File trong Python

File là một thứ rất quen thuộc đối với những người sử dụng máy tính. Bạn thao tác, tạo lập file hàng ngày. Nó có thể là một bức hình, một văn bản tài liệu, một file thực thi và nhiều thứ khác nữa.

Trong Python, file có 2 loại:

Text File

- Được cấu trúc như một dãy các dòng, mỗi dòng bao gồm một dãy các kí tự và một dòng tối thiểu là một kí tự dù cho dòng đó là dòng trống.
- Các dòng trong text file được ngăn cách bởi một kí tự newline và mặc định trong Python chính là kí tự **escape sequence newline \n**.

Binary File

- Các file này chỉ có thể được xử lý bởi một ứng dụng biết và có thể hiểu được cấu trúc của file này.
 - Và chúng ta ở đây với mức độ cơ bản chỉ xử lý text file.
-

Mở File trong Python

Khởi phải bàn, muốn thao tác với file, ta phải mở file. Mà muốn mở file, ta cũng cần phải có file.

Ở đây, Kteam sẽ tạo một file, và sau đó mở **CMD** ở ngay trong thư mục chứa file đó để không gặp nhiều khó khăn trong việc xử lý đường dẫn (Việc xử lý đường dẫn, Kteam sẽ giới thiệu cách xử lý bằng thư viện os trong tương lai).

Tên file sẽ là: [kteam.txt](#)

Nội dung file:

How Kteam

Free Education

Share to better

print('hello world!')

```
1 How Kteam
2 Free Education
3
4 Share to better
5
6 print('hello world!')
7 |
```

Hàm open

Được rồi, bây giờ chúng ta sẽ mở file bằng cách sử dụng hàm open

Cú pháp:

```
open(file, mode='r', buffering=-1, encoding=None, errors=None,
newline=None, closefd=True, opener=None)
```

Công dụng: Ở mức độ cơ bản, chúng ta sẽ chỉ quan tâm đến 2 parameter: **file** và **mode**.

Nếu các bạn muốn tìm hiểu rõ hơn về các parameter khác. Hãy dùng lệnh:

```
>>> help(open)
```

Ta sẽ bắt đầu bắc cách thử mở một file.

Lưu ý: Kteam xin được khuyến khích các bạn không sử dụng **interactive prompt** ở bài này.

```
>>> file_object = open('kteam.txt')
>>> file_object
<_io.TextIOWrapper name='kteam.txt' mode='r' encoding='cp1258'>
>>> type(file_object) # không cần quan tâm lắm
<class '_io.TextIOWrapper'>
```

Lưu ý: hàm open trả về một **file object**. Đây cũng là một **iterable**.

Tiếp đến là các mode mở file. Và cũng với mức độ cơ bản, Kteam sẽ cung cấp một số mode cơ bản liên quan đến text file.

MODE	USAGE
r	Mở để đọc. Đây là mode mặc định
r+	Mở để đọc và ghi
w	Mở để ghi. Trước đó, nó sẽ xóa hết nội dung của file hiện có. Nếu file không tồn tại, sẽ tạo ra một file với tên là tên file chúng ta truyền vào
w+	Mở để ghi và đọc. Trước đó cũng đã xóa hết nội dung của file hiện có. Nếu file không tồn tại, sẽ tạo ra một file với tên là tên file chúng ta truyền vào
a	Mở để ghi. Nếu file không tồn tại, sẽ tạo ra một file với tên là tên file chúng ta truyền vào
a+	Mở để ghi và đọc Nếu file không tồn tại, sẽ tạo ra một file với tên là tên file chúng ta truyền vào

Đóng File trong Python

Đây là việc chúng ta nên làm sau khi thao tác xong với file. Đó là đóng file.

Cú pháp:

```
<File>.close()
```

Tại sao chúng ta nên đóng file sau khi hoàn tất công việc với file?

- Giới hạn hệ điều hành. Chẳng hạn một hệ điều hành chỉ cho mở một số file nhất định cùng lúc thì nếu quên đóng file sẽ gây hao tốn. Đặc biệt là các file với dung lượng bự.
- Khi một file được mở, hệ điều hành sẽ khóa file đó lại, không cho các chương trình khác có thể xử lý trên file đó nữa nhằm đảm bảo tính nhất quán của dữ liệu.

Do đó hãy close file khi xong việc!

Dẫu vậy, nếu chương trình kết thúc. Tất cả các file đang mở cũng sẽ được đóng lại. Tuy nhiên việc đóng file vẫn là trách nhiệm nằm ở chúng ta.

```
>>> fobj = open('kteam.txt')
>>> fobj
<_io.TextIOWrapper name='kteam.txt' mode='r' encoding='cp1258'
>>> fobj.close()
>>> fobj # sau khi đóng file, các phương thức xử lí sẽ không thể sử dụng được
<_io.TextIOWrapper name='kteam.txt' mode='r' encoding='cp1258'>
```

Đọc File trong Python

Ta có một số phương thức có thể lấy được nội dung của file

Phương thức read

Cú pháp:

```
<File>.read(size=-1)
```

Công dụng: Nếu **size** bị bỏ trống hoặc là một số âm. Nó sẽ đọc hết nội dung của file đồng thời đưa con trỏ file tới cuối file. Nếu không nó sẽ đọc tới n kí tự (với $n = size$) hoặc cho tới khi nội dung của file đã đọc xong.

- Sau khi đọc được nội dung, nó sẽ trả về dưới một dạng chuỗi.

- Nếu không đọc được gì, phương thức sẽ trả về một chuỗi có độ dài bằng 0

Ví dụ:

```
>>> fobj = open('kteam.txt')
>>> data = fobj.read()
>>> data
"How Kteam\nFree Education\n\nShare to better\n\nprint('hello world!')\n"
>>> print(data)
How Kteam
Free Education

Share to better

print('hello world!')

>>> fobj.read() # con trỏ file ở vị trí cuối cùng, bạn không thể đọc được gì nữa
"
>>> fobj.close() # nhớ đóng file
```

Dưới đây là một ví dụ về đọc từng số kí tự một

```
>>> fobj = open('kteam.txt')
>>> fobj.read(2)
'Ho'
>>> fobj.read(10)
'w Kteam\nFr'
>>> fobj.read(20)
'ee Education\n\nShare '
>>> fobj.read()
"to better\n\nprint('hello world!')\n"
>>> fobj.close()
```

Phương thức readline

Cú pháp:

```
<File>.readline(size=-1)
```

Công dụng: Với parameter `size` thì hoàn toàn tương tự như phương thức `read`.

- Khác biệt ở chỗ, phương thức readline chỉ đọc một dòng có nghĩa là đọc tới khi nào gặp `newline` hoặc hết file thì ngừng.
- Con trỏ file cũng sẽ đi từ dòng này qua dòng khác.
- Kết quả đọc được trả về dưới dạng một chuỗi.
- Nếu không đọc được gì, phương thức sẽ trả về một chuỗi có độ dài bằng

Ví dụ:

```
>>> fobj = open('kteam.txt')
>>> fobj.readline()
'How Kteam\n'
>>> fobj.readline(10)
'Free Educa'
>>> fobj.readline()
'tion\n'
>>> fobj.readline()
'\n'
>>> fobj.readline()
'Share to better\n'
>>> fobj.close()
```

Phương thức readlines

Cú pháp:

```
<File>.readlines(hint=-1)
```

Ở mức độ cơ bản, ta không phải quan tâm đến parameter **hint**.

Công dụng: Phương thức này sẽ đọc toàn bộ file, sau đó cho chúng vào một list. Với các phần tử trong list là mỗi dòng của file.

- Con trỏ file sẽ được đưa tới cuối file. Khi đó, nếu bạn tiếp tục dùng `readlines`. Bạn sẽ nhận được một list rỗng.

Ví dụ:

```
>>> fobj = open('kteam.txt')
>>> list_content = fobj.readlines()
>>> list_content
['How Kteam\n', 'Free Education\n', '\n', 'Share to better\n', '\n', "print('hello
world!')\n"]
>>> list_content[2]
'\n'
>>> list_content[-1]
"print('hello world!')\n"
>>> fobj.close()
```

Đọc file bằng constructor nhận iterable

Như đã nói, file object nhận được từ hàm `open` cũng là một **iterable**.

Thế nên, ta có thể sử dụng constructor list

```
>>> fobj = open('kteam.txt')
>>> list_content = list(fobj)
```

```
>>> list_content  
['How Kteam\n', 'Free Education\n', '\n', 'Share to better\n', '\n', "print('hello  
world!')\n"]  
>>> fobj.close()
```

Và cũng có thể là Tuple.

```
>>> fobj = open('kteam.txt')  
>>> tup_content = tuple(fobj)  
>>> tup_content  
('How Kteam\n', 'Free Education\n', '\n', 'Share to better\n', '\n', "print('hello  
world!')\n")  
>>> fobj.close()
```

Các constructor này cũng sẽ đưa con trỏ file xuống cuối file.

Ghi File trong Python

Chúng ta có sự giúp đỡ của phương thức write để ghi nội dung vào file.

Và chúng ta cũng không cần phải tạo file. Vì các mode ghi sẽ giúp chúng ta tạo file.

Phương thức write

Cú pháp:

```
<File>.write(text)
```

Công dụng: Phương thức này sẽ trả về số kí tự mà chúng ta ghi vào.

Ví dụ:

```
>>> fobj = open('kteam_2.txt', 'w')
>>> fobj.write('The first line\n') # thêm \n để kết thúc 1 dòng
15
>>> fobj.write('And last line too')
17
>>> fobj.close()
```

Mỗi lần sử dụng write. Con trỏ file sẽ được đặt ngay sau kí tự cuối cùng được ghi. Hãy lưu ý điều này, nó rất quan trọng đấy. Đặc biệt là khi bạn sử dụng các mode vừa đọc vừa ghi.

Nhưng, bạn sẽ gặp vấn đề như thế này khi sử dụng mode w. Ta hãy mở lại file kia ta mới ghi một vài dòng vào nhé.

```
>>> fobj = open('kteam_2.txt')
>>> fobj.read()
'The first line\nAnd last line too'
>>> fobj.close()
>>> fobj = open('kteam_2.txt', 'w')
>>> fobj.write('\none more line')
14
>>> fobj.close()
>>> fobj = open('kteam_2.txt')
>>> fobj.read()
'\none more line'
>>> fobj.close()
```

Đó là nội dung file ban đầu của bạn sẽ bị mất đi. Đó là lí do chúng ta cần mode a.

Ta hãy mở lại file ta mới viết thêm một lần nữa.

```
>>> fobj = open('kteam_2.txt', 'a')
>>> fobj.write('\nthe second line')
16
>>> fobj.close()
>>> fobj = open('kteam_2.txt')
>>> fobj.read()
'\none more line\nthe second line'
>>> fobj.close()
```

Kiểm soát con trỏ file

Bạn có thể thấy, con trỏ file rất quan trọng, nó dẫn đường cho việc đọc file, viết file. Và bạn cũng cần phải kiểm soát được nó.

Việc đó, ta sẽ nhờ tới phương thức seek

Phương thức seek

Cú pháp:

```
<File>.seek(offset, whence=0)
```

Với Python 3.X. Một text file sẽ chỉ được sử dụng `whence = 0`. `whence = 1` hoặc `whence = 2` chỉ sử dụng với binary file.

Với Python 2.X thì bạn không phải quan tâm vấn đề này.

Do đó, ta cũng không cần quan tâm tới parameter `whence`.

Công dụng: Phương thức này giúp ta di chuyển con trỏ từ vị trí đầu file qua `offset` kí tự. Và parameter `offset` phải là một số tự nhiên.

- Nhờ phương thức này, ta có thể ghi nội dung từ bất cứ đâu trong file.
- Và từ đó ta có thể đọc lại file sau khi ta đưa con trỏ file xuống cuối file.

Ví dụ:

```
>>> fobj = open('kteam.txt')
>>> fobj.read()
"How Kteam\nFree Education\n\nShare to better\n\nprint('hello world!')\n"
>>> fobj.read()
"
>>> fobj.seek(0)
```

```

0
>>> fobj.read()
"How Kteam\nFree Education\n\nShare to better\n\nprint('hello world!')\n"
>>> fobj.seek(10)
10
>>> fobj.read()
"\nFree Education\n\nShare to better\n\nprint('hello world!')\n"
>>> fobj.close()

```

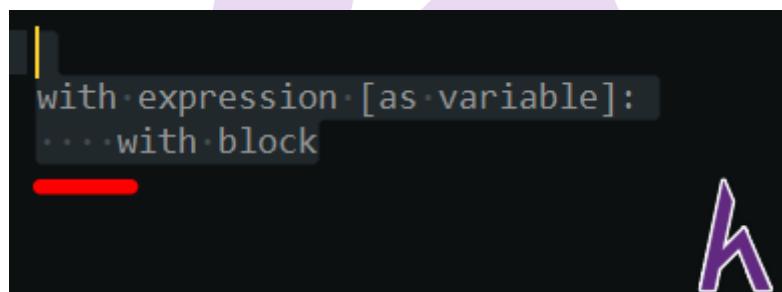
Câu lệnh with

Cấu trúc cơ bản của câu lệnh with là

```

with expression [as variable]:
    with-block

```



Nhớ rằng **with-block** nằm thut vào so với dòng **with expression** (theo chuẩn PEP8 là 4 space và là dùng space không dùng tab)

Câu lệnh này liên quan đến phương thức **enter** và **exit** của đối tượng. Do đó, ở đây Kteam sẽ nói cơ bản khi sử dụng file.

Đặc điểm của câu lệnh with khi sử dụng với file là. Khi kết thúc **with-block**. File sẽ được đóng.

```

>>> with open('kteam.txt') as fobj:
...     data = fobj.read()
...

```

```
>>> data
"How Kteam\nFree Education\n\nShare to better\n\nprint('hello world!')\n"
>>> fobj.read() # không thể đọc file, vì file đã đóng
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIẾU DỮ LIỆU DICT TRONG PYTHON – Phần 2](#).

1. Vì hai dict trỏ cùng vào một nơi. Cách khắc phúc là ta dùng phương thức copy để có bản sao dict1.
2. Sẽ có lỗi ở

```
>>> d = {}
>>> d.update(3)
```

Câu hỏi củng cố

1. Nêu sự khác nhau giữa mode r+ và w+
2. Tèo mở file dưới mode vừa đọc và ghi. Tèo đang thắc mắc là vì sao sau khi ghi xong rồi, mà Tèo vẫn không đọc được gì cả. Hãy giải đáp giúp Tèo.

```
>>> teo_file = open('teo.txt', 'w+')
>>> teo_file.write('Teo dep trai\n')
13
>>> teo_file.read()
```

"

Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Qua bài viết này, Bạn đã hiểu cơ bản về FILE TRONG PYTHON.

Ở bài viết sau. Kteam sẽ nói về [ITERATION & MỘT SỐ HÀM CƠ BẢN](#) hay được sử dụng.

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".

Bài 20: ITERATION VÀ MỘT SỐ HÀM HỖ TRỢ CHO ITERATION OBJECT TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Iteration và một số hàm hỗ trợ cho iteration object trong Python](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn cách [XỬ LÝ FILE](#) trong Python

Ở bài này Kteam sẽ giới thiệu với các bạn **MỘT SỐ HÀM HỖ TRỢ CHO ITERABLE OBJECT** trong Python. Một trong những điều thiết yếu mà bất cứ ngôn ngữ lập trình nào bạn cũng đều phải tìm hiểu.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).

- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU LIST](#), [KIỂU DỮ LIỆU TUPLE](#), [KIỂU DỮ LIỆU SET](#), [KIỂU DỮ LIỆU DICT](#) trong Python.

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Khái niệm iteration trong Python
- Giới thiệu iterable object trong Python
- Giới thiệu iterator object trong Python
- Một số hàm hỗ trợ cho iterable object trong Python

Khái niệm iteration trong Python

Iteration là một khái niệm chung cho việc lấy từng phần tử một của một đối tượng nào đó, bất cứ khi nào bạn sử dụng vòng lặp hay kĩ thuật nào đó để có được giá trị một nhóm phần tử thì đó chính là Iteration.

Ví dụ: như bạn ăn một snack, bạn sẽ lấy từng miếng trong bọc snack ra ăn cho tới khi hết thì thôi. Bạn có thể coi việc lấy bánh là một vòng lặp. đương nhiên bạn cũng có thể chọn không lấy hết số bánh ra.

Giới thiệu iterable object trong Python

Iterable object là một object có phương thức `__iter__` trả về một `iterator`, hoặc là một object có phương thức `__getitem__` cho phép bạn lấy bất cứ phần tử nào của nó bằng `indexing` ví dụ như Chuỗi, List, Tuple.

Giới thiệu iterator object trong Python

Iterator object đơn giản chỉ là một đối tượng mà cho phép ta lấy từng giá trị một của nó. Có nghĩa là bạn không thể lấy bất kì giá trị nào như ta hay làm với **List** hay **Chuỗi**.

Iterator không có khả năng tái sử dụng trừ một số iterator có phương thức hỗ trợ như file object sẽ có phương thức **seek**.

Iterator sử dụng hàm **next** để lấy từng giá trị một. Và sẽ có lỗi **StopIteration** khi bạn sử dụng hàm next lên đối tượng đó trong khi nó hết giá trị đưa ra cho bạn.

Các **iterable object** chưa phải là iterator. Khi sử dụng hàm **iter** sẽ trả về một iterator. Đây cũng chính là cách các vòng lặp hoạt động.

Ví dụ minh họa:

```
>>> [x for x in range(3)] # thuộc lòng 3 giá trị của comprehension này
[0, 1, 2]
>>> itor = (x for x in range(3)) # sử dụng () cho ra một generator expression – một
iterator
>>> itor
<generator object <genexpr> at 0x03374CC0>
>>> next(itor)
0
>>> next(itor)
1
>>> next(itor)
2
>>> next(itor) # chỉ có 3 giá trị, và ta đã lấy hết
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

File object cũng là một iterator. Bạn cũng có thể sử dụng cách này để đọc file.

```
>>> lst = [6, 3, 7, 'kteam', 3.9, [0, 2, 3]]
>>> iter_list = iter(lst) # iter_list là một iterator tạo từ list
>>> iter_list
<list_iterator object at 0x03647730>
>>> iter_list[0] # đương nhiên, iterator không hỗ trợ indexing
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list_iterator' object is not subscriptable
>>> next(iter_list)
6
>>> next(iter_list)
3
>>> next(iter_list)
7
>>> next(iter_list)
'kteam'
>>> next(iter_list)
3.9
>>> next(iter_list)[-2]
2
>>> next(iter_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Bạn cũng lưu ý, iterator này cũng dính một vấn đề như List, Dict đó chính là **chỉnh một, thay đổi hai**.

```
>>> it_1 = iter('kteam')
>>> it_1
<str_iterator object at 0x03647770>
>>> it_2 = it_1
>>> next(it_2)
'k'
>>> next(it_2)
't'
>>> next(it_2)
'e'
>>> next(it_1)
'a'
>>> next(it_1)
'm'
```

```
>>> next(it_2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(it_1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Một số hàm hỗ trợ cho iterable object trong Python

Một điều lưu ý: Các hàm này buộc phải lấy các giá trị của iterable để xử lí, do đó nếu bạn đưa vào một iterator. Thì bạn sẽ không sử dụng iterator đó được nữa.

Hàm tính tổng – sum

Cú pháp:

```
sum(iterable, start=0)
```

Công dụng: Trả về tổng các giá trị của iterable và iterable này chỉ chứa các giá trị là số. Còn **start** chính là giá trị ban đầu. Có nghĩa là sẽ cộng từ **start** lên. Mặc định là **0**

Ví dụ:

```
>>> sum([1, 6, 3])
10
>>> sum([1, 6, 3], 10)
20
>>> sum(iter([6, 3, 9]))
```

```

18
>>> it = (x for x in range(3))
>>> sum(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

Hàm tìm giá trị lớn nhất – max

Cú pháp:

```
max(iterable, *[, default=obj, key=func])
```

Công dụng: Nhận vào một iterable.Tìm giá trị lớn nhất bằng **key** (mặc định là sử dụng operator **>**). Default là giá trị muốn nhận về trong trường hợp không lấy được bất kì giá trị nào trong iterable.

- Dấu ***** chính là kí hiệu yêu cầu **keyword-only argument**. Bạn sẽ hiểu thêm khi Kteam giới thiệu **parameter** trong function.

```

•   >>> max([1, 2, 3])
•   3
•   >>> max([1, 2, 3], default='default value')
•   3
•   >>> max([], default='default value')
•   'default value'

```

Hoặc

```
max(arg1, arg2, *args, *[, key=func])
```

Trong đó:

- ***args** là **packing arguments** (bạn sẽ hiểu thêm khi Kteam giới thiệu với bạn **packing arguments**). Ở đây không có parameter **default**, vì khi theo cách này, bạn luôn luôn có ít nhất 2 giá trị so sánh

```
>>> max(1, 2, 3)  
3  
>>> max(1, 2)  
2
```

Hàm tìm giá trị nhỏ nhất – min

Cú pháp:

```
min(iterable, *, default=obj, key=func)
```

hoặc

```
min(arg1, arg2, *args, *, key=func)
```

Ý nghĩa: giống như hàm max. Khác ở chỗ đây là tìm **giá trị nhỏ nhất**

```
>>> min([1, 2, 3])  
1  
>>> min([], default='kteam')  
'kteam'
```

Hàm sắp xếp – sorted

Cú pháp:

```
sorted(iterable, /, *, key=None, reverse=False)
```

Công dụng: Giống với phương thức sort của List object.

Ví dụ:

```
>>> sorted([1, 6, 7, 2])
[1, 2, 6, 7]
>>> sorted([1, 6, 7, 2], reverse=True)
[7, 6, 2, 1]
```

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [XỬ LÝ FILE TRONG PYTHON](#).

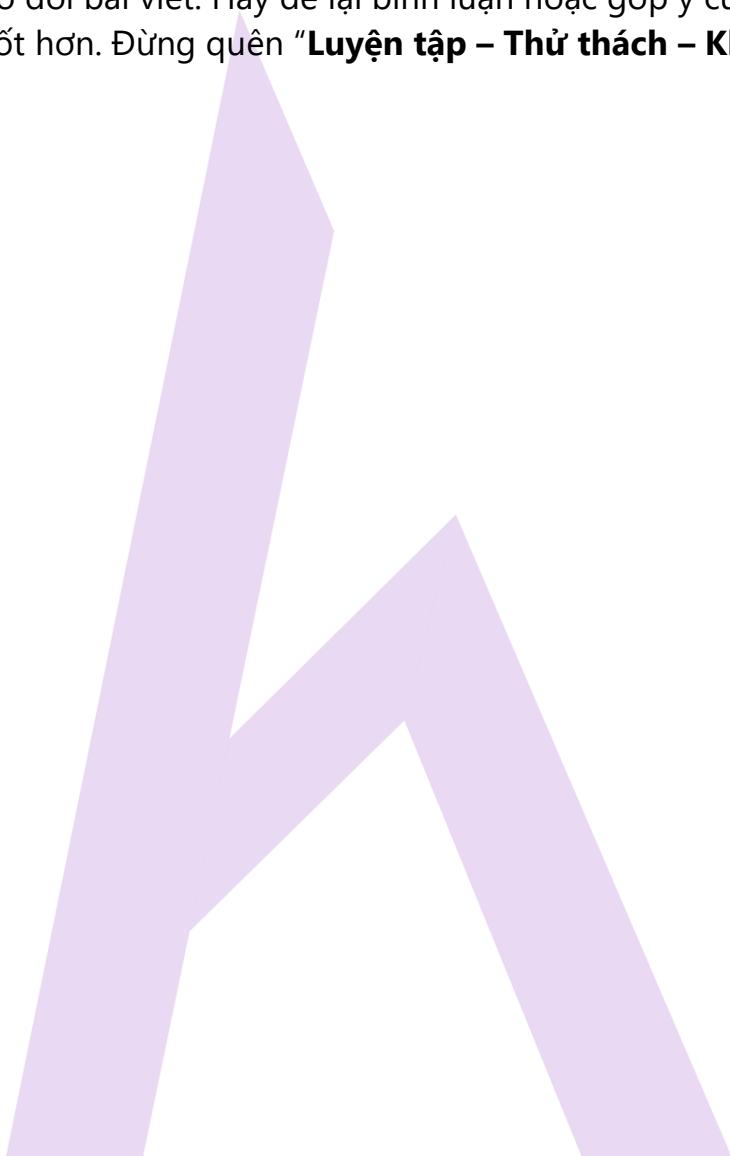
1. w+ tạo ra một file nếu file đó hiện chưa có.
 2. Vì khi Tèo ghi xong, con trỏ file nằm ở cuối file > Tèo không đọc được gì. Trường hợp đó, ta sử dụng phương thức seek.
-

Kết luận

Qua bài viết này, Bạn đã hiểu hơn về ITERABLE OBJECT trong Python.

Ở bài viết sau. Kteam sẽ nói về [NHẬP XUẤT TRONG PYTHON](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.



Bài 21: NHẬP XUẤT TRONG PYTHON - HÀM XUẤT

Xem bài học trên website để ủng hộ Kteam: [Nhập xuất trong Python – Hàm xuất](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [ITERATION & MỘT SỐ HÀM HỖ TRỢ CHO ITERABLE OBJECT](#) trong Python

Ở bài này Kteam sẽ giới thiệu với các bạn việc **Nhập xuất trong Python**. Một điều rất cần thiết!

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU LIST](#), [KIỂU DỮ LIỆU TUPLE](#), [KIỂU DỮ LIỆU SET](#), [KIỂU DỮ LIỆU DICT](#) trong Python.
- Biết cách [XỬ LÍ FILE TRONG PYTHON](#)

Trong bài này, bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Vì sao cần hàm print?
 - Tìm hiểu cách sử dụng hàm print thông qua các parameter.
 - Print Python 3.X và Python 2.X có gì khác nhau?
-

Vì sao cần hàm print

Nếu bạn hay dùng **interactive prompt** thì bạn nhận ra rằng, kết quả luôn xuất hiện sau mỗi dòng code của bạn. Tuy nhiên, nó sẽ không như vậy khi bạn viết những dòng code vào trong một file Python và chạy chương trình đó.

Bạn cần một hàm giúp bạn xuất các nội dung mà bạn muốn cụ thể ở đây là xuất ra Shell (terminal, command prompt, powershell,...). Đó là lí do hàm print ra đời!

Tìm hiểu cách sử dụng hàm print through qua các parameter

Hàm print có cú pháp như sau

Cú pháp:

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

Chúng ta sẽ tìm hiểu parameter đầu tiên

*objects

* chính là **packing argument**. Ở đây hiểu nôm na sẽ là nó sẽ gom lại các argument của bạn lại thành một **Tuple**.

```
>>> packing = 1, 2, 3, 4 # giống như gọi hàm function(1, 2, 3, 4)
>>> packing
(1, 2, 3, 4)
```

Khi bạn truyền các argument vào hàm (giá trị 1, giá trị 2, giá trị 3,...) thì nó sẽ gói lại thành một Tuple giống như trên.

```
>>> print('Kteam')
Kteam
>>> print('Kteam', 'Free Education')
Kteam Free Education
>>> print('Kteam', 'Free Education', 'one more argument')
Kteam Free Education one more argument
```

Nhờ như vậy, bạn có thể truyền argument vào hàm **print** với số lượng bất kì. Điều này giúp bạn **không phải** ép kiểu dữ liệu, để rồi nối chúng lại với nhau thành một giá trị rồi mới truyền cho hàm print.

```
>>> print('Kteam' + 69)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
>>> print('Kteam' + str(69))
Kteam69
>>> print('Kteam', 69)
Kteam 69
>>> print(123, [1, 2, 3], 'Kteam')
123 [1, 2, 3] Kteam
```

Chắc bạn cũng nhận ra một chút khác biệt ở hai trường hợp bên dưới.

```
>>> print('Kteam' + 'Python')
```

```
KteamPython  
>>> print('Kteam', 'Python')  
Kteam Python
```

Để hiểu điều đó, chúng ta tới với parameter tiếp theo

sep (separate – chia ra, phân ra)

Giá trị mặc định của parameter này là **một khoảng trắng**. Khi các argument bạn ném vào cho hàm print để hàm print in ra nội dung, như đã biết là nó sẽ được gói vào một **Tuple**. Các giá trị trong Tuple sẽ được nối với nhau bằng parameter **sep**.

Lưu ý: Khi truyền giá trị vào cho parameter theo cách **keyword argument** thì sẽ không bị **packing**. Nghĩa là sẽ không bị gói vào trong giá trị của **parameter object**.

```
>>> print('Kteam', 'Python', 'Course') # sep mặc định là 1 khoảng trắng  
Kteam Python Course  
>>> print('Kteam', 'Python', 'Course', sep='---')  
Kteam---Python---Course  
>>> print('Kteam', 'Python', 'Course', sep='|||')  
Kteam|||Python|||Course  
>>> print('Kteam', 'Python', 'Course', sep='\n')  
Kteam  
Python  
Course  
>>> print('Kteam', 'Python', 'Course', sep='')  
KteamPythonCourse
```

Tiếp theo là một parameter khá rắc rối

end (kết thúc bằng)

Đầu tiên, hãy chạy một file Python với nội dung sau đây.

```
print('line 1')
print('line 2')
print('line 3')
```

Kết quả bạn nhận được chắc chắn sẽ là

```
line 1
line 2
line 3
```

Nếu bạn từng học qua ngôn ngữ **lập trình C** hoặc **C++** hay là **Java** cũng có thể là **C#**. Bạn sẽ nhận thấy, mỗi lần print, chúng sẽ tự xuống dòng.

Đó là nhờ parameter `end`. Nó sẽ tự thêm một kí tự `newline (\n)` vào cuối để có thể đưa con trỏ xuống dòng mới thay vì bạn phải tự thêm `\n` như một số ngôn ngữ lập trình khác (một số ngôn ngữ lập trình có hỗ trợ thêm phương thức giúp xuất nội dung và tự động xuống dòng)

Và đương nhiên, chúng ta cũng có thể thay đổi giá trị của parameter này.

```
>>> print('a line without newline', end="")
a line without newline>>> print('a line without newline', end='|||')
a line without newline|||>>> print()

>>>
```

Bạn cũng thấy nếu không có `end` bằng một kí tự `newline` thì **interactive prompt** lộn xộn thế nào.

Nhưng đó không phải vấn đề. Hãy cẩn thận khi sử dụng `print` mà không có `newline`.

Hãy tạo một file Python có nội dung như sau:

```
from time import sleep # nhập hàm sleep từ thư viện time  
  
print('start....')  
sleep(3) # dừng chương trình 3 giây  
print('end...')
```

Khi chạy chương trình, bạn sẽ thấy xuất hiện dòng **'start....'** sau đó 3 giây sau sẽ xuất hiện tới dòng **'end...'**.

Kết quả này hoàn toàn bình thường và đúng như những gì dự đoán. Nhưng hãy thử thay đổi một tí:

```
from time import sleep # nhập hàm sleep từ thư viện time  
  
print('start....', end='') # in ra nội dung và kết thúc bởi một chuỗi rỗng  
sleep(3) # dừng chương trình 3 giây  
print('end...')
```

Lần này đã có khác biệt. Bạn sẽ không thấy gì xuất hiện ban đầu, mãi đến 3 giây sau bạn mới thấy dòng **'start....end...'**. Kết quả thì đúng, nhưng cách kết quả được xuất ra thì không giống như bạn nghĩ.

Vì sao lại vậy? Đó là do mỗi lần hàm print nhận được các giá trị bạn muốn in. Các giá trị đó được gói trong một **Tuple**. Tiếp đến, hàm print nạp từng giá trị trong **Tuple** vào bộ nhớ đệm. Nếu giá trị đó là một chuỗi và có kí tự **newline** (ở vị trí bất kỳ) thì hàm print sẽ yêu cầu bộ nhớ đệm xuất những gì có trong bộ nhớ đệm từ nãy nạp đến giờ.

Hoặc khi kết thúc chương trình, những gì còn trong bộ đệm cũng sẽ được xuất ra.

Một số ví dụ

Ví dụ 1: Hãy thử một vài ví dụ khác để hiểu thêm

```
from time import sleep # nhập hàm sleep từ thư viện time
```

```
print('line 1\n', 'line2', end='')
sleep(3) # dừng chương trình 3 giây
print('end...')
```

Kết quả xuất hiện sẽ là `line1` > đợi 3 giây > xuất hiện các nội dung còn lại. Vì chuỗi 'line 1\n' có kí tự newline nên chuỗi đó được xuất ra. Còn chuỗi 'line 2' thì không nên vẫn nằm trong bộ nhớ đệm. **Ví dụ 2:**

```
from time import sleep # nhập hàm sleep từ thư viện time

print('line 1', 'lin\nne2', end='')
sleep(3) # dừng chương trình 3 giây
print('end...')
```

Kết quả sẽ là xuất in hai chuỗi 'line 1' và 'line 2' > đợi 3 giây > xuất nội dung còn lại.

Quy trình sẽ là nạp chuỗi line 1 vào bộ nhớ đệm, nạp tiếp chuỗi line 2 vào bộ nhớ đệm, thấy chuỗi line 2 có kí tự newline, xuất những gì có trong bộ nhớ đệm ra. Sau đó đợi 3 giây và rồi xuất nội dung còn lại.

file

Mặc định hàm print sẽ ghi nội dung vào file `sys.stdout`. Cũng nhờ vậy, bạn mới thấy được nội dung trên shell. Đương nhiên, dựa vào đây, ta cũng có thể sử dụng hàm print như là `phương thức write` trong việc ghi file.

```
>>> with open('printtext.txt', 'w') as f:
...     print('printed by print function', file=f)
...
>>> with open('printtext.txt') as f:
...     f.read()
...
'printed by print function\n'
```

flush

Parameter cuối cùng - **flush**. Giá trị mặc định giá trị là `False`. Liên quan khá nhiều đến parameter `end` lúc nay thế nên ta hãy quay lại ví dụ lúc nay.

```
from time import sleep # nhập hàm sleep từ thư viện time  
  
print('start...', end='')  
sleep(3) # dừng chương trình 3 giây  
print('end...')
```

Sau 3 giây chương trình mới có kết quả. Bạn cũng đã biết vì sao rồi, đúng chứ?

Nào, hãy để cho parameter `flush` giá trị `True`

```
from time import sleep # nhập hàm sleep từ thư viện time  
  
print('start...', end='', flush=True)  
sleep(3) # dừng chương trình 3 giây  
print('end...')
```

Kết quả bây giờ vẫn vậy, nhưng quá trình xuất kết quả có chút khác biệt. Bạn ngay lập tức nhìn thấy nội dung dòng print đầu tiên. Đó là nhờ parameter `flush`. Nếu là `True`, nó sẽ yêu cầu bộ đệm xuất những gì có trong bộ đệm ra.

Print trong Python 3.X và Python 2.X có gì khác nhau?

Print trong Python 3.X là một hàm, như đã giới thiệu. Còn với Python 2.X nó là một câu lệnh.

```
# print trong Python 2.X
```

```
print 'Kteam'  
print 'Kteam', 'Free Education'  
# tương tự với trong Python 3.X sẽ là  
print('Kteam')  
print('Kteam', 'Free Education')
```

Một số bạn nhầm lần rằng Print Python 2.X cũng có thể sử dụng như Python 3.X

```
# print trong Python 2.X  
print('Kteam')  
# và nhận được kết quả giống như Python 3.X  
print('Kteam')
```

Nhưng bản chất là khác nhau

```
# print trong Python 2.X  
print('Kteam')  
# tương đương với Python 3.X là  
print(('Kteam'))
```

Đây là **interactive prompt** của Python 2.X. Ta sẽ thử một ví dụ để làm rõ điều này

```
>>> print('Kteam')  
Kteam  
>>> print('Kteam', 'Free Education')  
('Kteam', 'Free Education')
```

Bạn cũng thấy, cặp dấu () không phải là một cặp dấu ngoặc như cách gọi hàm. Đó giống như việc bạn đặt một giá trị trong cặp dấu ngoặc đơn mà thôi. Vì nó có một giá trị nên không có sự khác biệt

Còn khi bạn đặt hai giá trị trở lên, Python hiểu đó là một Tuple.

Một đoạn code nhỏ dành cho bạn tự nhiên cứu:

```
from time import sleep

your_name = "Henry"
your_great = "Hello! My name is "

for c in your_great + your_name:
    print(c, end=",", flush=True)
    sleep(0.1)
print()
```

Kết luận

Qua bài viết này, Bạn đã biết về việc xuất nội dung trong Python.

Ở bài viết sau. Kteam sẽ nói về [NHẬP XUẤT TRONG PYTHON – HÀM NHẬP](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".

Bài 22: NHẬP XUẤT TRONG PYTHON - HÀM NHẬP

Xem bài học trên website để ủng hộ Kteam: [Nhập xuất trong Python – Hàm nhập](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [HÀM PRINT](#) – một hàm giúp bạn xuất kết quả ra màn hình (Shell)

Ở bài này Kteam sẽ tiếp tục giới thiệu với các bạn việc **Nhập xuất trong Python**. Cụ thể là việc nhập!

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [KIỂU DỮ LIỆU LIST](#), [KIỂU DỮ LIỆU TUPLE](#), [KIỂU DỮ LIỆU SET](#), [KIỂU DỮ LIỆU DICT](#) trong Python.
- Biết cách [XỬ LÍ FILE TRONG PYTHON](#)
- Biết [CÁCH SỬ DỤNG HÀM PRINT TRONG PYTHON](#)

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Vì sao cần hàm input?
- Tìm hiểu cách sử dụng hàm input
- Hàm input Python 3.X và raw_input Python 2.X
- Lưu ý giành cho hàm input Python 2.X

Vì sao cần hàm input

Tèo là một Kter “bờ rào” của Kteam. Hôm trước, Tèo có làm một chương trình đơn giản. Đó chính là in ra dòng chữ “Xin chào Tiên”. Và đây là code của Tèo.

```
name = 'Tien'  
print('Xin chao', name)
```

Mọi chuyện diễn ra tốt đẹp, chương trình chạy đúng theo ý Tèo và cũng nhờ đó Tèo đã tạo được cảm tình với Tiên.

Thu thấy thế, cũng muốn Tèo viết cho một chương trình như Tiên và phải in ra dòng chữ “Xin chào Thu”.

Tèo lại mở code lên mà sửa lại:

```
name = 'Thu'  
print('Xin chao', name)
```

Sau đó, một số bạn nữ khác cũng muốn Tèo viết cho mình một chương trình như Thu và Tiên bao gồm Quỳnh, Nhi, Giao, Như, Uyên, Hương, Loan, Trung, Nam,... Kể không xuể. Và bạn thấy vấn đề đã nảy sinh. Tèo phải sửa code hết lần này đến lần khác.

Có thể việc này không mất quá nhiều thời gian, vì Tèo vẫn có thể viết cho mỗi bạn một cái chương trình riêng hoặc là mỗi lần viết là Tèo viết cho một bạn và thay đổi mã nguồn.

Nhưng nếu dung lượng máy của Tèo có hạn, không thể chứa nhiều chương trình hoặc Tèo không có đủ thời gian để chỉnh sửa code hết lần này tới lần khác thì sao? Tèo muốn viết một mà lại có thể cho nhiều người.

Điều này đưa ra cho Tèo một yêu cầu, đó chính là biến **name** phải là một biến có dữ liệu được nhập mỗi khi chạy chương trình thay vì được đưa sẵn cho một giá trị.

Và nhờ một hàm có tên là **input**. Tèo đã giải quyết được vấn đề nan giải sau ba ngày ba đêm tìm kiếm trên **GOOGLE**.

Tìm hiểu cách sử dụng hàm input

Theo như Tèo tìm kiếm trong tài liệu trên trang chủ của Python, hàm input có cú pháp như sau

input(prompt=None)

Lưu ý: Có lúc bạn sẽ nhìn thấy cú pháp của nó là **input(prompt=None, /)**. Cái phần thêm vào là kí tự **/** chỉ là một kí tự cho biết parameter **prompt** chỉ nhận giá trị dưới dạng **positional argument**. Nghĩa là khi bạn truyền vào cho hàm, bạn không được phép điền thêm chữ **prompt**.

```
>>> input('string') # hợp lệ
>>> input(prompt='string') # không hợp lệ
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: input() takes no keyword arguments
```

Parameter prompt là một parameter tùy chọn. Bạn có thể nhập hoặc không vì nó đã có giá trị mặc định là **None**.

Công dụng: Hàm này giúp chúng ta đọc một chuỗi từ **standard input** (hiểu nôm na là việc bạn nhập dữ liệu lên trên **Shell**) sau đó trả về cho chúng ta. Và vì nó là đọc một chuỗi, nên dù bạn có nhập cái gì đi chăng nữa thì nó vẫn là một chuỗi dù là số, list, tuple, set, dictionary,...

Việc nhập sẽ kết thúc sau khi bạn nhấn phím **enter**. Ở đây, khi bạn nhấn phím **enter** (phím return) thì cũng đồng nghĩa với việc bạn gửi vào một kí tự **newline**. Nhưng kí tự newline này sẽ bị bỏ đi.

Nếu trong lúc nhập bạn nhấn **EOF**

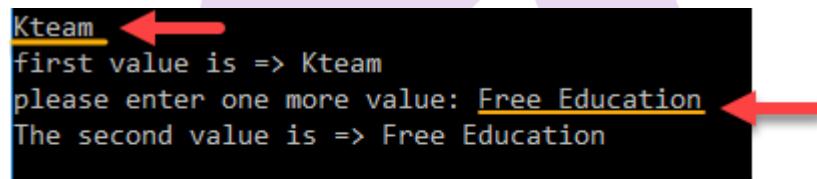
*nix: Ctrl + D, Windows: Ctrl + Z > Return (Enter) hoặc Ctrl + C

thì sẽ sinh lỗi **EOFError**.

Nếu **prompt** khác **None**, có nghĩa là bạn gửi cho prompt một giá trị. Thì giá trị này sẽ được in ra mà không có kí tự newline đi kèm trước khi đọc giá trị nhập vào.

Chúng ta đến với ví dụ. Hãy tạo một file có nội dung như sau

```
value = input() # prompt để None
print('first value is =>', value)
next_value = input('please enter one more value: ')
print('The second value is =>', next_value)
```



Đây là hình ảnh khi chạy chương trình trên. Trong đó:

- Những dòng có **mũi tên màu đỏ** là những dòng thực hiện hàm **input**.
- Những chữ **gạch chân màu vàng** chính là giá trị nhập vào.

Đầu tiên, ta sẽ được yêu cầu nhập dữ liệu vào cho biến **value**. Ở đây, Kteam nhập vào giá trị là **Kteam**.

Và điều đó được kiểm chứng bằng việc ở dòng tiếp theo, giá trị Kteam được in ra màn hình.

Tiếp đế, chúng ta tiếp tục được yêu cầu nhập dữ liệu. Bạn có thể thấy khác so với lần chúng ta sử dụng hàm `input` khi không truyền giá trị vào cho parameter `prompt`. Giờ đây, chúng ta có một dòng ghi chú yêu cầu nhập dữ liệu. Và với giá trị nhập vào là **Free Education**, giá trị đó đã được in ra ở dòng cuối cùng.

Kteam xin được lưu ý thêm một lần nữa đó là bạn nhập cái gì thì giá trị trả về **LUÔN LUÔN LÀ CHUỖI**.

Hãy thử đoạn code sau:

```
# reading input
int_num = input('Enter an integer: ')
float_num = input('Enter a float: ')
lst = input('Enter a list: ')
tup = input('Enter a tuple: ')
set_ = input('Enter a set: ')
dict_ = input('Enter a dict: ')

# print out output
print('Type of int_num', type(int_num))
print('Type of float_num', type(float_num))
print('Type of lst', type(lst))
print('Type of tup', type(tup))
print('Type of set_', type(set_))
print('Type of dict_', type(dict_))
```

```
Enter an integer: 69
Enter a float: 3.14
Enter a list: [1, 2, 'Kteam']
Enter a tuple: ('Kteam', 'Free Education', [1, 2, 101.00])
Enter a set: {(1, 2), 'abc'}
Enter a dict: {'team': 'Kteam', (90, 4): ['list', 'tuple']}
Type of int_num <class 'str'>
Type of float_num <class 'str'>
Type of lst <class 'str'>
Type of tup <class 'str'>
Type of set_ <class 'str'>
Type of dict_ <class 'str'>
```

Như bạn thấy, tất cả đều thuộc lớp chuỗi. Kteam sẽ tiếp tục thêm một số ví dụ với hàm input.

```
value = input('Enter something => ')
print('You just entered', value)
print('__repr__ method: %r' %value)
```

Lần này, Kteam sẽ chỉ nhấn phím **Enter**.

```
Enter something =>
You just entered
__repr__ method: ''
```



Khi bạn không nhập thứ gì và nhấn phím **Enter**. Chuỗi bạn nhận được từ hàm **input** là một chuỗi rỗng (số kí tự trong chuỗi bằng 0).

Tiếp tục với đoạn code trên, lần này Kteam sẽ nhấn **EOF**.

```
Enter something => ^Z
Traceback (most recent call last):
  File "a.py", line 1, in <module>
    value = input('Enter something => ')
EOFError
```



- Lỗi **EOFError** hiện lên. Chương trình kết thúc ngay lập tức.

Hàm input Python 3.X và raw_input Python 2.X

Hàm **raw_input** **không** tồn tại trong Python 3.X, nó đã được đổi tên thành **input** ở phiên bản Python 3.X.

Lưu ý: giàngh cho hàm input Python 2.X

Trong **Python 2.X**, còn một hàm nữa cũng gần giống với hàm **raw_input** (chính là hàm **input** ở **Python 3.X**) là hàm **input**.

Cú pháp của hàm này hoàn toàn tương tự với hàm **input** trong **Python 3.X**. Nó cũng sẽ nhận vào một chuỗi như hàm **input** **Python 3.X** (**raw_input Python 2.X**). Tuy nhiên, chuỗi đó sẽ được truyền vào hàm **eval**.

Do đó **input** Python 2.X có cú pháp

```
input(prompt=None)
```

Sẽ tương tự

```
eval(raw_input(prompt=None))
```

Và tương đương ở Python 3.X sẽ

```
eval(input(prompt=None))
```

Hàm `eval` có khả năng thực thi một `expression` với `expression` đưa vào dưới dạng chuỗi.

Một `expression` là một giá trị nào đó như một con số, một chuỗi, một list.

Sau đây là một vài ví dụ về hàm `eval`:

```
>>> eval('123')
123
>>> eval('[1, 2, 3]')
[1, 2, 3]
>>> x = 1
>>> eval('x + 2')
3
>>> eval('print("This is exec by eval fucntion")') # hàm print là một expression với
giá trị là None
This is exec by eval fucntion
>>> eval('a = 3') # đây là một statement. Không phải expression.
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
  File "<string>", line 1
    a = 3
    ^
SyntaxError: invalid syntax
```

Lưu ý: Ở đây, Kteam có một lưu ý với các bạn đó là **không nên** sử dụng hàm `eval` trừ khi thực sự rất cần thiết.

Có một số lí do để bạn nên tránh sử dụng hàm `eval`:

- Khiến việc debug khó khăn
- Làm chậm chương trình
- Luôn có cách tốt hơn thay thế
- Rất nguy hiểm và không an toàn.

Nếu bạn thắc mắc tại sao lại nguy hiểm. Thì Kteam có thể đưa ra một số ví dụ đơn giản.

Ví dụ: bạn cho phép người dùng sử dụng chương trình của bạn. Bạn yêu cầu họ nhập một số thứ nhưng lại sử dụng hàm eval bọc lên hàm input. Thế nên, họ có thể sử dụng nó để phá chương trình của bạn.

Giả sử bạn có một ứng dụng web. Nếu một kẻ xấu nào đó nhập vào với nội dung dạng thế này thì coi như ứng dụng của bạn toí.

```
Enter something: __import__('shutil').rmtree('/root')
```

Câu lệnh dưới, có thể xóa sạch cây thư mục của bạn. Đó là một dạng của **command injection**. Điều này rất nguy hiểm cho hệ thống của bạn.

```
>>> __import__('shutil').rmtree('/root')
```

Do đó, việc sử dụng **eval** phải được cân nhắc. Đương nhiên sẽ có trường hợp eval không nguy hiểm như trên, hoặc là bạn phải dùng tới nó. Nhưng hãy hạn chế!

Kết luận

Qua bài viết này, Bạn đã biết về việc yêu cầu người dùng NHẬP NỘI DUNG từ bàn phím trong Python.

Ở bài viết sau. Kteam sẽ nói về [KIỂU DỮ LIỆU BOOLEAN TRONG PYTHON](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".

Bài 23: KIỂU DỮ LIỆU BOOLEAN TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Boolean trong Python](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [HÀM INPUT](#) - một hàm giúp bạn yêu cầu nhập dữ liệu từ bàn phím

Ở bài này Kteam sẽ giới thiệu với các bạn **Kiểu dữ liệu Boolean trong Python**. Một kiểu dữ liệu cực kì cần thiết trong các phần sử dụng cấu trúc rẽ nhánh, vòng lặp.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON](#)

Trong bài này, chúng ta sẽ cùng tìm hiểu những nội dung sau đây

- Giới thiệu về Boolean trong Python
 - Boolean trong các toán tử so sánh
 - NOT, AND và OR
 - Các giá trị cũng là các Boolean
 - Syntaxic sugar cho việc so sánh trong Python
-

Giới thiệu về Boolean trong Python

Boolean là một kiểu dữ liệu mà các ngôn ngữ lập trình ngày nay đều thường xuyên sử dụng. Python cũng không ngoại lệ.

Kiểu dữ liệu này chỉ có hai giá trị:

- Một là **True** – có nghĩa là đúng
- Nếu không thì là **False** – có nghĩa là sai.

Bạn cũng đã thấy nó rồi khi sử dụng toán tử in trong các bài kiểm dữ liệu chuỗi, list,...

Boolean trong các toán tử so sánh

So sánh giữa số với số

Bạn chắc biết so sánh là gì nhờ các tiết học toán ở trường. Ví dụ như

- **3 > 1** là đúng
- **69 < 10** là sai
- **241 = 141 + 100** là đúng
- **(5 x 0) ≠ 0** là sai.

Trong Python cũng có các toán tử như vậy. Tuy nhiên kí hiệu của chúng thì có khác đôi chút.

Bảng sau đây sẽ cho các bạn thông tin về những toán tử so sánh trong Python

Toán học	Python
>	>
<	<
=	==
≥	>=
≤	<=
≠	!=

Hãy xem ví dụ minh họa trong Python:

```
>>> 3 > 1 # 3 > 1 là đúng => True
True
>>> 69 < 10 # 69 < 10 là sai => False
False
>>> 241 == 141 + 100 # 241 = 141 + 100 là đúng => True
True
>>> (5 * 0) != 0 # 5 x 0 ≠ 0 là sai => False
False
```

Bạn cũng có thể so sánh nhiều kiểu dữ liệu khác nữa, không chỉ là giữa số với số.

So sánh giữa hai iterable cùng loại

Khi so sánh hai iterable cùng loại. Python sẽ lấy lần lượt từng phần tử trong iterable ra so sánh. Kteam sẽ lấy ví dụ về kiểu chuỗi:

```
>>> 'Kteam' == "Kteam"
True
>>> 'Free' == 'Education'
False
```

Lưu ý: Python so sánh các kí tự với nhau bằng cách đưa chúng về dưới dạng số bằng hàm **ord**. Bạn có thể tham khảo giá trị của nó trong **ASCII Table**.

```
>>> ord('A')
65
>>> ord('a')
97
```

Khi bạn so sánh bằng các toán tử **==**, **>=**, **<=** thì Python sẽ so sánh hết các phần tử.

Còn nếu bạn dùng các toán tử như **>**, **<**, **!=** thì nhiều lúc Python sẽ không cần phải đi hết các giá trị iterable. Nếu như ở vị trí **i** nào đó mà đã hai giá trị không bằng nhau thì nó sẽ dừng lại.

```
>>> 'a' > 'ABC'
# ord('a') không bằng ord('A'), không cần phải so sánh tiếp và ord('a') > ord('A') là
đúng => True
True
>>> 'aaa' < 'aaAcv'
# ord('a') không bằng ord('A') ở vị trí thứ 2, không cần phải so sánh tiếp và ord('a')
< ord('A') là sai => False
False
>>> 'aaa' < 'aaaAcv'
# 3 phần tử đầu tiên bằng nhau. Ở phần tử thứ tư, ta sẽ so sánh 0 và ord('A') và dĩ
nhiên ord('A') > 0 => True
True
```

Toán tử **is**

Đây là một toán tử dễ nhầm lẫn với toán tử **==**. Nhưng thật sự thì nó rất đơn giản!

Ở đây, Kteam sẽ nói tới một phần kiến thức ở tiếng Anh để bạn có thể dễ phân biệt 2 toán tử trên. Từ **is** trong tiếng Việt (ở ngữ cảnh này – ngôn ngữ lập trình Python) có nghĩa là “**là**”. Còn toán tử **==** có nghĩa là **bằng**.

Kteam sẽ đưa ra một ví dụ. Bạn cũng không nên khắt khe việc đúng sai trong ví dụ này, nó chỉ giúp bạn hiểu sự khác nhau giữa toán tử `==` và `is` thôi.

Thế nào là **bằng** (`==`)?

- **Bằng** là toán tử so sánh khi nói về mặt giá trị.
- **Ví dụ:** Chiều cao của Tèo **bằng** chiều cao của Tí

Thế nào là **là** (`is`)?

- **Là** (`is`) trong trường hợp này là liên từ diễn giải định nghĩa, tính chất của một sự vật/sự việc/con người.
- **Ví dụ:** Ta không thể nói "Chiều cao của Tèo là chiều cao của Tí" vì của Tèo là của Tèo, đâu phải của Tí. Nên nói là "Chiều cao của Tèo là chiều cao của Tèo" hoặc "Chiều cao của Tí là chiều cao của Tí"

Ta hãy trở lại với Python bằng việc khởi tạo hai List

```
>>> lst = [1, 2, 3]
>>> lst_ = [1, 2, 3]
```

Chúng đều có giá trị là một List gồm ba phần tử 1, 2 và 3. Vậy chúng có **bằng** nhau? đương nhiên là **có**. Thử luôn là biết.

```
>>> lst == lst_
True
```

Nhưng **lst** có phải là **lst_**? đương nhiên là **không**. Vì đó là hai List khác nhau không liên quan đến nhau.

```
>>> lst is lst_
False
```

Vậy nếu ta có một List khác

```
>>> _lst = lst
>>> _lst
[1, 2, 3]
```

Thì `_lst` có phải là `lst` không? Nếu bạn còn nhớ một số điều lưu ý khi sử dụng List trong bài [KIẾU DỮ LIỆU LIST TRONG PYTHON – PHẦN 1](#) thì chắc chắn là bạn còn nhớ, 2 List này đang trỏ chung vào một địa chỉ. Do đó, chúng là một, chỉ khác nhau cái nhãn thôi.

```
>>> _lst is lst  
True
```

Từ đây, ta có thể suy ra một kết luận. Khi so sánh hai giá trị (đối tượng) bằng toán tử `==` thì Python sẽ **so sánh bằng giá trị** của chúng. Còn nếu so sánh bằng toán tử `is` thì Python sẽ lấy **giá trị của hàm id để so sánh**.

Lưu ý toán tử `is`

Bạn không nên so sánh 2 số như thế này

```
>>> 699 is 699  
True
```

Kết quả luôn là **True**. Bạn sẽ chỉ thấy khác biệt khi:

```
>>> a = 699  
>>> b = 699  
>>> a is b  
False
```

Nhưng, có một số trường hợp bạn cần biết:

```
>>> a = -5  
>>> b = -5  
>>> a is b  
True
```

```
>>> c = 256
>>> d = 256
>>> c is d
True
>>> a = 'abc'
>>> b = 'abc'
True
```

Các số từ -5 đến 256 hoặc là một số chuỗi có số kí tự dưới 20 thì các biến có cùng một giá trị sẽ có cùng một giá trị trả về từ **hàm id**.

NOT, AND và OR

Not là **phủ định**.

Đây là cách bạn có thể đổi giá trị Boolean. Trong một số trường hợp đặc biệt. Việc kiểm tra giá trị Boolean đó là **False** hay là **True** hơi phức tạp, rườm ra trong khi đó việc kiểm tra giá trị ngược lại thì dễ dàng, đơn giản hơn.

Value	Not
True	False
False	True

And là **và**.

Or là **hoặc**.

Bạn cần nắm lòng bảng sau để có thể kết hợp những điều kiện một cách nhuần nhuyễn. Từ đó, bạn có thể sử dụng linh hoạt các câu lệnh điều kiện, đặt expression cho các vòng lặp một cách hiệu quả.

Bạn hãy xem bảng sau đây:

Left-Value	Right-Value	And	Or
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Ví dụ: để rõ hơn nhé. Đầu tiên là **and**

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

Tiếp đến là **or**

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

Cuối cùng là **not**

```
>>> not True
False
>>> not False
True
```

Các giá trị cũng là các Boolean

Thật vậy, các giá trị đều là các boolean. Và đương nhiên, bạn có thể chuyển đổi chúng thành các Boolean bằng hàm **bool**.

Mọi giá trị khi chuyển về Boolean đều là **True** trừ một số trường hợp sau

- Số 0
- None
- Rỗng

Ví dụ: để hiểu hơn

```
>>> bool(0)
False
>>> bool(None)
False
>>> bool('')
False
>>> bool([])
False
>>> bool(())
False
>>> bool(set())
False
>>> bool({})
False
```

Thêm một số trường hợp **True**

```
>>> bool(1)
True
>>> bool('abc')
True
>>> bool([1, 2, 3])
True
```

1 là True, 0 là False

Không quá quan trọng, nhưng cũng nên biết

```
>>> True + 1  
2  
>>> False + 1  
1  
>>> int(True)  
1  
>>> int(False)  
0
```

Syntaxnic sugar cho việc so sánh trong Python

Nếu bạn từng học một số ngôn ngữ lập trình khác. Bạn đôi lúc phải kiểm tra những trường hợp như kiểm tra một số **n** có nằm trong khoảng **(a; b)**, đoạn **[a; b]**, nửa khoảng **(a; b]**, nửa khoảng **[a; b)** hay không? hoặc là kiểm tra xem một số **k** có bằng một trong những số như x, y hoặc z hay không. Đương nhiên, những lần làm như vậy cũng làm bạn hơi cực

```
>>> n = 5  
>>> n > 1 and n < 6 # kiểm tra xem n có nằm trong khoảng (1; 6) hay không  
True  
>>> n > 1 and n < 4 # kiểm tra xem n có nằm trong khoảng (1; 4) hay không  
False
```

Nhưng với Python, bạn có thể làm thế này.

```
>>> 1 < a < 6  
True  
>>> b = -4  
>>> b < -3 < -1 < 0 < a < 6 # thậm chí là dài như thế này  
True
```

Với trường hợp nếu bạn muốn kiểm tra xem một số **k** có bằng **x** hoặc **y** hoặc là **z** hay không thì thường bạn phải viết khá dài.

```
>>> k = 4  
>>> k == 3 or k == 4 or k == 5  
True
```

Tuy nhiên, bạn cũng có thể

```
>>> k in (3, 4, 5) # nên dùng () hơn là [] hoặc thứ gì khác  
True
```

Kết luận

Bài viết này đã giới thiệu sơ cho các bạn KIỂU DỮ LIỆU BOOLEAN TRONG PYTHON.

Ở bài sau, Kteam sẽ giới thiệu đến bạn [CẤU TRÚC RẼ NHÁNH TRONG PYTHON.](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.

Bài 24: CẤU TRÚC RẼ NHÁNH TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Cấu trúc rẽ nhánh trong Python](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [KIỂU DỮ LIỆU BOOLEAN TRONG PYTHON](#).

Ở bài này Kteam sẽ giới thiệu với các bạn **Cấu trúc rẽ nhánh trong Python – Câu lệnh IF**. Một câu lệnh thường xuyên được sử dụng trong các chương trình.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON](#)

Trong bài này, chúng ta sẽ cùng tìm hiểu những nội dung sau đây

- If là gì? Có ăn được không
- If
- If – else if

- If – else
 - If – else if – else
 - Block trong Python
-

If là gì? Có ăn được không?

If là một từ tiếng Anh thường gặp, khi dịch nó ra tiếng Việt ta sẽ được nghĩa là “Nếu” hoặc là “Giá mà”, “Miễn là”,... Dĩ nhiên là “Nếu” là một từ chẳng mấy xa lạ với các bạn. Chúng ta sử dụng nó cả trăm, ngàn lần một ngày.

- Nếu hôm nay chủ nhật, Tèo sẽ đi chơi.
- Nếu ủng hộ đủ 5000 điểm thì Kteam sẽ xuất bản khóa Kỹ Thuật Import/Export Cookie Selenium.
- Nếu được vote up câu hỏi thì bạn được cộng điểm, còn nếu bị vote down bạn sẽ bị trừ điểm, không có vote thì số điểm không thay đổi.

Python cũng biết nếu, có điều nếu khác chúng ta một tẹo. Để biết khác thế nào, chúng ta hãy cùng tìm hiểu!

If

Đây là ví dụ về câu lệnh if cơ bản nhất. **Nếu ... thì ...**

- Nếu **m – 1 < 0** thì **m < 1**

Từ đó, Python đã xây dựng một cấu trúc nếu tương tự như trên:

```
if expression:  
    # If-block
```

Lưu ý: Tất cả các câu lệnh nằm trong **if-block** là các câu lệnh có lề thụt vào trong so với câu lệnh if. Chi tiết Kteam sẽ trình bày ở phần tiếp theo

Ở đây, nếu **expression** là một giá trị khi đưa về kiểu dữ liệu Boolean là **True** thì Python sẽ nhảy vào thực hiện các câu lệnh trong **if-block**. Còn nếu không thì không thì sẽ bỏ qua **if-block** đó.

```
>>> a = 0
>>> b = 3
>>>
>>> if a - 1 < 0: # (a - 1 < 0) có giá trị là True
...     print('a nhỏ hơn 1')
...
a nhỏ hơn 1
>>>
>>> if b - 1 < 0: # (b - 1 < 0) có giá trị là False
...     print('b nhỏ hơn 1')
...
>>>
```

If – else if

Đây là bản nâng cấp của cấu trúc **if** vừa rồi chúng ta tìm hiểu. Nó có cấu trúc như sau:

```
if expression:
    # If-block

elif 2-expression:
    # 2-if-block

elif 3-expression:
    # 3-if-block

...
elif n-expression:
```

n-if-block

Ở đây, bạn có thể đặt bao nhiêu lần **nếu** cũng được. Và từ câu lệnh **if** đến lần **elif** lần thứ **n - 1** (câu lệnh với **n-expression**) là một khối, ta sẽ đặt cho nó một cái tên là khối **BIG** để dễ hiểu. Nó sẽ hoạt động như sau:

Bước 1: Kiểm tra xem **expression** có phải là một giá trị Boolean **True** hay không?

Bước 2: Nếu có, thực hiện **if-block** sau đó kết thúc khối **BIG**. Không thì chuyển sang Bước 3.

Bước 3: Kiểm tra xem **2-expression** có phải là một giá trị Boolean **True** hay không?

Bước 4: Nếu có, thực hiện **2-if-block** sau đó kết thúc khối **BIG**. Không thì chuyển sang Bước 5.

Bước 5: Kiểm tra xem **3-expression** có phải là một giá trị Boolean **True** hay không?

Bước 6: Nếu có, thực hiện **3-if-block** sau đó kết thúc khối **BIG**. Không thì chuyển sang Bước 7

...

Bước (n - 1) x 2: Kiểm tra xem **n-expression** có phải là một giá trị Boolean **True** hay không?

Bước (n - 1) x 2 + 1: Nếu có, thực hiện **n-if-block**.

Bước (n - 1) x 2 + 2: Kết thúc khối **BIG**.

Ví dụ để các bạn dễ hiểu hơn

```
>>> a = 3
>>>
>>> if a - 1 < 0: # False, tiếp tục
...     print('a nhỏ hơn 1')
... elif a - 2 < 0: # False, tiếp tục
```

```
...     print('a nhỏ hơn 2')
... elif a - 3 < 0: # False, tiếp tục
...     print('a nhỏ hơn 3')
... elif a - 4 < 0: # True, kết thúc
...     print('a nhỏ hơn 4')
... elif a - 5 < 0: # Khối BIG đã kết thúc, dù đây là True nhưng không ý nghĩa
...     print('a nhỏ hơn 5')
...
a nhỏ hơn 4
```

If - else

Cấu trúc vừa rồi không biết có làm bạn đau đầu hay không. Nếu có, hãy thư giãn vì cấu trúc sau đây đơn giản hơn nhiều.

```
if expression:
    # If-block
else:
    # else-block
```

Nếu **expression** là một giá trị Boolean **True**, thực hiện **if-block** và kết thúc. Không quan tâm đến **else-block**. Còn nếu không sẽ thực hiện **else-block** và kết thúc.

Ví dụ:

```
>>> a = 0
>>> b = 3
>>>
>>> if a - 1 < 0:
...     print('a nhỏ hơn 1')
... else:
...     print('a lớn hơn hoặc bằng 1')
...
```

```
a nhỏ hơn 1
>>>
>>> if b - 1 < 0: # False, nên sẽ thực hiện else-block
...   print('b nhỏ hơn 1')
... else:
...   print('b lớn hơn hoặc bằng 1')
...
b lớn hơn hoặc bằng 1
```

If – else if – else

Nó không có gì mới mẻ nếu bạn nắm rõ 3 cấu trúc trên. Sau đây là cấu trúc của **if – else if – else**

if expression:

If-block

elif 2-expression:

2-if-block

...

elif n-expression:

n-if-block

else:

else-block

Bạn có thể đặt bao nhiêu lần **elif** cũng được nhưng **else** thì chỉ một. Và từ câu lệnh **if** đến câu lệnh **else** là một khối, ta cũng sẽ đặt cho nó một cái tên là khối **BIG** để dễ hiểu. Nó sẽ hoạt động như sau:

Bước 1: Kiểm tra xem **expression** có phải là một giá trị Boolean **True** hay không?

Bước 2: Nếu có, thực hiện **if-block** sau đó kết thúc khối **BIG**. Không thì chuyển sang Bước 3.

Bước 3: Kiểm tra xem **2-expression** có phải là một giá trị Boolean **True** hay không?

Bước 4: Nếu có, thực hiện **2-if-block** sau đó kết thúc khối **BIG**. Không thì chuyển sang Bước 5

...

Bước (n - 1) x 2: Kiểm tra xem **n-expression** có phải là một giá trị Boolean **True** hay không?

Bước (n - 1) x 2 + 1: Nếu có, thực hiện **n-if-block** sau đó kết thúc khối **BIG**.

Bước (n - 1) x 2 + 2: Nếu không thì thực hiện **else-block** và kết thúc khối **BIG**.

Ví dụ:

```
>>> a = 0
>>> if a - 1 < 0:
...     print('a nhỏ hơn 1')
... elif a - 1 > 0:
...     print('a lớn hơn 1')
... else:
...     print('a bằng 1')
...
a nhỏ hơn 1
>>>
>>> b = 2
>>> if b - 1 < 0:
...     print('b nhỏ hơn 1')
... elif b - 1 > 0:
...     print('b lớn hơn 1')
... else:
...     print('b bằng 1')
...
b lớn hơn 1
>>>
```

```
>>> c = 1
>>> if c - 1 < 0:
...     print('c nhỏ hơn 1')
... elif c - 1 > 0:
...     print('c lớn hơn 1')
... else:
...     print('c bằng 1')
...
c bằng 1
```

Block trong Python

Với đa số ngôn ngữ lập trình hiện nay, thường dùng cặp dấu ngoặc {} để phân chia các block.

Riêng đối với Python lại sử dụng việc **định dạng code để suy ra các block**. Đây là điều giúp code Python luôn luôn phải đẹp mắt.

Một số điều lưu ý về việc định dạng code block trong Python:

- Câu lệnh mở block kết thúc bằng dấu hai chấm (:), sau khi sử dụng câu lệnh có dấu hai chấm(:) **buộc phải xuống dòng và lùi lề vào trong** và có tối thiểu một câu lệnh để không bỏ trống block.
- Những dòng code cùng lề thì là cùng một block.
- Một block có thể có nhiều block khác.
- Khi căn lề block không sử dụng cả tab lẫn space.
- Nên sử dụng **4 space** để căn lề một block

Sau đây là một hình minh họa của Kteam.

Các câu lệnh nằm trong một khung màu là một block, và block đó được mở bởi câu lệnh nằm ngay bên trên khung màu.

```

1  a = 3
2
3  if a - 1 < 0:
4      print('a nhỏ hơn 1')
5      if a < 0:
6          print('a nhỏ hơn 0')
7      else:
8          print('a không nhỏ hơn 0')
9  elif a - 1 > 0:
10     print('a lớn hơn 1')
11     if a - 2 > 0:
12         print('a lớn hơn 2')
13         if a - 3 > 0:
14             print('a lớn hơn 3')
15         elif a - 2 == 0:
16             print('a bằng 2')
17         else:
18             print('1 < a < 2')
19     else:
20         print('a bằng 1')
21

```

Lưu ý: Kteam có đề cập đến việc sau khi sử dụng câu lệnh có dấu hai chấm (:) buộc phải xuống dòng và lùi lề vào trong. Tuy nhiên, Bạn vẫn có thể đi ngược lại điều này trong một vài trường hợp

Ví dụ:

- >>> a = 3
- >>> if a - 1 > 0: print('a lớn hơn 1')
- ...
 - a lớn hơn 1
 - >>> if a - 1 > 0: print('a lớn hơn 1'); print('có thể a lớn hơn 2')
 - ...
 - a lớn hơn 1
 - có thể a lớn hơn 2

Tuy nhiên, việc sử dụng như vậy không được khuyến khích vì chỉ tiết kiệm được một vài dòng code mà lại gây khó đọc thì không đáng để tiết kiệm.

Và bạn cũng đã biết thêm một điều Python không hề cấm dấu chấm phẩy (;). Nó vẫn là một cú pháp hợp lệ. Nếu bạn quen tay có thể dùng dấu chấm phẩy (;) thoải mái.

Củng cố bài học

Câu hỏi củng cố

Nhập từ bàn phím 3 số, in ra số lớn nhất (cố gắng ít dòng code nhất có thể - ở đây không tính việc nhập dữ liệu)

Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Qua bài viết này, Bạn đã biết về CÂU ĐIỀU KIỆN IF TRONG PYTHON.

Ở bài viết sau. Kteam sẽ nói về khái niệm vòng lặp và biết tới [CẤU TRÚC VÒNG LẶP WHILE TRONG PYTHON.](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".

Bài 25: VÒNG LẶP WHILE TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Vòng lặp While trong Python](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [CÂU ĐIỀU KIÊN IF](#) - một dạng cấu trúc rẽ nhánh rất quan trọng trong mọi ngôn ngữ lập trình không chỉ riêng Python

Ở bài này Kteam sẽ giới thiệu với các bạn **Vòng lặp While trong Python**.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON](#)
- [CÂU ĐIỀU KIÊN IF TRONG PYTHON](#)

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Đặt vấn đề
- Cấu trúc vòng lặp while và cách hoạt động
- Sử dụng vòng lặp để xử lý chuỗi, list, tuple

- Câu lệnh break và continue
 - Cấu trúc vòng lặp while-else và cách hoạt động
-

Đặt vấn đề

Lại là câu chuyện về Tèo – Kter “bờ rào” của Kteam. Sắp tới là sinh nhật Tèo, Tèo tham vọng mời tất cả thành viên trong group lập trình của Kteam. Thế nên, Tèo mua một xấp giấy về ghi thiệp mời các bạn tham dự buổi tiệc.

Một bạn, hai bạn, rồi ba bạn và tới bạn thứ năm thì Tèo đã thấm mệt. Dòng chữ cũng không được nắn nót như ban đầu. Nhớ lại là còn hơn 9999 người cần phải mời nữa. Nên Tèo mệt quá, không muốn mời ai nữa và ăn sinh nhật một mình luôn.

Nếu bạn là Tèo, bạn sẽ viết được bao nhiêu tấm thiệp với dòng chữ nắn nót và đẹp như tấm thiệp ban đầu? Liệu bạn có đủ kiên nhẫn viết hết 1000 tấm thiệp thậm chí là 100000?

Hiển nhiên là “Không!”. Mà trường hợp của Tèo cũng chả phải hiếm. Vì vậy, con người đã tạo ra máy tính để giúp họ làm những việc tương tự. Máy tính có khả năng lặp đi lặp lại một tiến trình với số lần rất lớn. Hiệu suất của lần cuối cùng cũng như lần đầu tiên. Thêm một điều nữa là công việc đó được làm với một tốc độ chóng mặt

Làm sao chúng làm được như vậy? Đó là nhờ tuyệt kĩ vòng lặp. Và chúng ta sẽ bắt đầu đi tìm hiểu chiêu thức vòng lặp đầu tiên trong Python chính là **While**.

Cấu trúc vòng lặp while và cách hoạt động

Nào! Cùng ngó sơ cấu trúc, sau đó Kteam sẽ giải thích cho bạn cách mà nó hoạt động

```
while expression:
```

```
    # while-block
```

Lưu ý: Việc chia **block** như thế này cũng giống như khi bạn sử dụng **câu lệnh if** và đã được Kteam giới thiệu ở bài trước [CẤU TRÚC RẼ NHÁNH](#).

Nó sẽ hoạt động ra sao?

Rất đơn giản! Việc đầu tiên, Python sẽ kiểm tra giá trị boolean của **expression**. Nếu là **False**, thì bỏ qua **while-block** và đến với câu lệnh tiếp theo. Ngược lại, sẽ thực hiện toàn bộ câu lệnh trong **while-block**. Sau khi thực hiện xong, quay ngược lại kiểm tra giá trị boolean của **expression** một lần nữa. Nếu **False** thì bỏ qua **while-block**, còn **True** thì tiếp tục thực hiện **while-block**. Và sau khi thực hiện xong **while-block** lại quay về kiểm tra giá trị boolean **expression** như những lần trước.

Ví dụ:

```
>>> k = 5
>>>
>>> while k > 0:
...     print('k =', k)
...     k -= 1
...
k = 5
k = 4
k = 3
k = 2
k = 1
>>> k # k bằng 0 nên > 0 là một boolean False, do đó vòng lặp đã kết thúc
0
```

Sử dụng vòng lặp để xử lí chuỗi, list, tuple

Đây là những **iterable** cho phép ta truy xuất một giá trị bất kí trong nó bằng phương pháp **indexing**. Thế nên, ta có thể nhờ điều này kết hợp với vòng lặp để xử lí chúng.

```
>>> s = 'How Kteam'  
>>> idx = 0 # vị trí bắt đầu bạn muốn xử lí của chuỗi  
>>> length = len(s) # lấy độ dài chuỗi làm mốc kết thúc  
>>>  
>>> while idx < length:  
...     print(idx, 'stands for', s[idx])  
...     idx += 1 # di chuyển index tới vị trí tiếp theo  
  
0 stands for H  
1 stands for o  
2 stands for w  
3 stands for  
4 stands for K  
5 stands for t  
6 stands for e  
7 stands for a  
8 stands for m
```

Đơn giản phải không nào. **List** và **Tuple** hoàn toàn tương tự.

Câu lệnh break và continue

Lưu ý: Hai câu lệnh này chỉ có thể dùng trong các vòng lặp

Câu lệnh break

Câu lệnh **break** dùng để kết thúc vòng lặp. Cứ nó nằm trong block của vòng lặp nào thì vòng lặp đó sẽ kết thúc khi chạy câu lệnh này.

Trong trường hợp vòng lặp a chứa vòng lặp b. Trong **vòng lặp b** chạy câu lệnh **break** thì chỉ **vòng lặp b** kết thúc, còn **vòng lặp a** thì không.

Ví dụ *:

```
>>> five_even_numbers = []
>>> k_number = 1
>>>
>>> while True: # vòng lặp vô hạn vì giá trị này là hằng nên ta không thể tác động
    được
...     if k_number % 2 == 0: # nếu k_number là một số chẵn
...         five_even_numbers.append(k_number) # thêm giá trị của k_number vào list
...     if len(five_even_numbers) == 5: # nếu list này đủ 5 phần tử
...         break # thì kết thúc vòng lặp
...     k_number += 1
...
>>> five_even_numbers
[2, 4, 6, 8, 10]
>>> k_number
10
```

Câu lệnh continue

Câu lệnh này dùng để chạy tiếp vòng lặp. Giả sử một vòng lặp có cấu trúc như sau:

while expression:

#while-block-1

continue

#while-block-2

Khi thực hiện xong **while-block-1**, câu lệnh **continue** sẽ tiếp tục vòng lặp, không quan tâm những câu lệnh ở dưới continue và như vậy nó đã bỏ qua **while-block-2**.

Ví dụ:

```
>>> k_number = 1
>>> while k_number < 10:
...     if k_number % 2 == 0: # nếu k_number là số chẵn
...         k_number += 1 # thì tăng một đơn vị cho k_number và tiếp tục vòng lặp
...         continue
...     print(k_number, 'is odd number')
...     k_number += 1
...
1 is odd number
3 is odd number
5 is odd number
7 is odd number
9 is odd number
```

Cấu trúc vòng lặp while-else và cách hoạt động

Ta sẽ xem cấu trúc trước:

while expression:

while-block

else:

else-block

Cấu trúc này gần tương tự như **while** bình thường. Thêm một điều, khi vòng lặp **while** kết thúc thì khối lệnh **else-block** sẽ được thực hiện.

Ví dụ:

```
>>> while k < 3:  
...     print('value of k is', k)  
...     k += 1  
... else:  
...     print('k is not less than 3 anymore')  
...  
value of k is 0  
value of k is 1  
value of k is 2  
k is not less than 3 anymore
```

Trong trường hợp trong **while-block** chạy câu lệnh **break** thì vòng lặp **while** sẽ kết thúc và phần **else-block** cũng sẽ không được thực hiện.

```
>>> k = 0  
>>> while k < 5:  
...     print('value of k is', k)  
...     k += 1  
...     if k > 3:  
...         print('k is greater than 3')  
...         break  
... else:  
...     print('k is not less than 5 anymore')  
...  
value of k is 0  
value of k is 1  
value of k is 2  
value of k is 3  
k is greater than 3
```

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [CẤU TRÚC RẼ NHÁNH TRONG PYTHON](#)

Cách 1:

```
k1 = int(input('Nhập số thứ nhất\n=> '))
k2 = int(input('Nhập số thứ hai\n=> '))
k3 = int(input('Nhập số thứ ba\n=> '))

if k1 > k2 and k1 > k3:
    print('số lớn nhất là', k1)
elif k2 > k1 and k2 > k3:
    print('số lớn nhất là', k2)
else:
    print('số lớn nhất là', k3)
```

Cách 2:

```
k1 = int(input('Nhập số thứ nhất\n=> '))
k2 = int(input('Nhập số thứ hai\n=> '))
k3 = int(input('Nhập số thứ ba\n=> '))

if k1 > k2 and k1 > k3: print('số lớn nhất là', k1)
elif k2 > k1 and k2 > k3: print('số lớn nhất là', k2)
else: print('số lớn nhất là', k3)
```

Câu hỏi củng cố

- Viết lại một vòng lặp có chức năng tương tự **ví dụ *** nhưng không dùng câu lệnh **break**
- Cho một file text tên **draft.txt** như sau:

```
an so dfn Kteam odsa in fasfna Kteam mlfjier
as dfasod nf ofn asdfer fsan dfoans ldnfad Kteam asdfna
asdofn sdf pzcvqp Kteam dfaojf kteam dfna Kteam dfaodf
afdna Kteam adfoasdf ncxvo aern Kteam dfad
```

Trong file này có một số chữ **Kteam** (**Kteam** sẽ không xuất hiện ở đầu dòng), và trước nó là một chữ ngẫu nhiên nào đó và nhiệm vụ của bạn là đổi chữ đó thành **How**. Nhớ là sử dụng vòng lặp.

Sau khi đổi thành công, bạn lưu nội dung đó vào file tên **kteam.txt**.

Đây là mẫu của **kteam.txt**:

```
an so How Kteam odsa in How Kteam mlfjier
as dfasod nf ofn asdfer fsan dfoans How Kteam asdfna
asdofn sdf How Kteam dfaojf kteam How Kteam dfaodf
How Kteam adfoasdf ncxvo How Kteam dfad
```

- Sắp xếp một mảng số nguyên có dạng như sau:

```
[56, 14, 11, 756, 34, 90, 11, 11, 65, 0, 11, 35]
```

Lưu ý: là các số 11 là những số cố định không được thay đổi vị trí của nó.

Sau khi sắp xếp lại mảng trên sẽ là:

```
[0, 14, 11, 34, 35, 56, 11, 11, 65, 90, 11, 756]
```

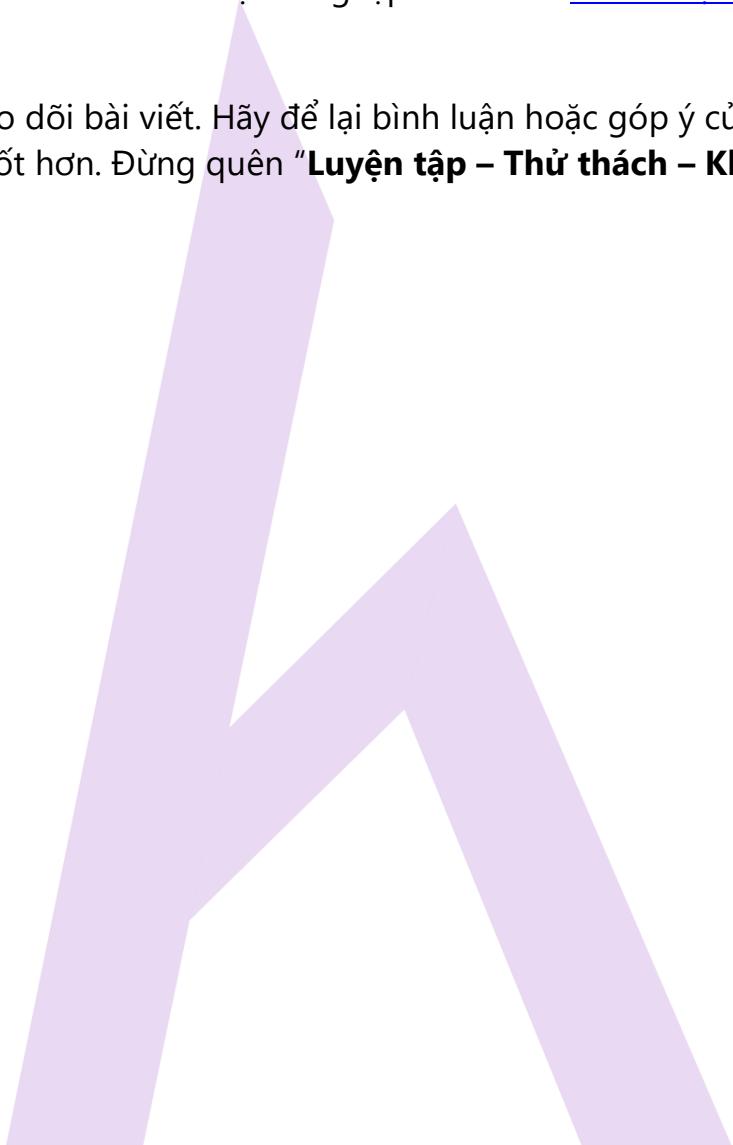
Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Qua bài viết này, Bạn đã biết về VÒNG LẶP WHILE TRONG PYTHON.

Ở bài viết sau, Kteam sẽ nói đến một vòng lặp nữa đó là [VÒNG LẶP FOR TRONG PYTHON](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.



Bài 26: VÒNG LẶP FOR TRONG PYTHON

Xem bài học trên website để ủng hộ Kteam: [Vòng lặp For trong Python](#).

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn một cấu trúc vòng lặp, đó chính là [VÒNG LẶP WHILE TRONG PYTHON](#).

Ở bài này Kteam sẽ giới thiệu với các bạn một công phu của vòng lặp nữa là **Vòng lặp For trong Python**.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- [CÂU ĐIỀU KIÊN IF TRONG PYTHON](#)
- [VÒNG LẶP WHILE TRONG PYTHON](#)

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Hạn chế của vòng lặp while
- Cấu trúc vòng lặp for và cách hoạt động

- Sử dụng vòng lặp để xử lí các iterator và Dict
 - Câu lệnh break và continue
 - Cấu trúc vòng lặp for-else và cách hoạt động
-

Hạn chế của vòng lặp while

Bạn có thể sử dụng vòng lặp while để có thể duyệt một List, chuỗi hoặc là một Tuple. Và thậm chí là một **iterator** (một **object** không hỗ trợ indexing) khi biết được số phần tử mà **iterator** đó chứa.

Ví dụ:

```
>>> length = 3
>>> iter_ = (x for x in range(length))
>>> c = 0
>>> while c < length:
...     print(next(iter_))
...     c += 1
...
0
1
2
```

Nếu bạn không biết trước được số phần tử mà **iterator** đó có thì cũng không sao. Python vẫn cho phép bạn làm được điều đó bằng **try-block** (Kteam sẽ giới thiệu ở một bài khác)

Ví dụ:

```
>>> iter_ = (x for x in range(3)) # giả sử ta không biết có 3 phần tử
>>> while 1: # 1 là một expression True
...     try:
...         print(next(iter_))
...     except StopIteration:
...         break
...
0
1
```

2

Nhưng “con trăn” Python không thích sự rườm rà. Xưa nay vốn được biết đến với danh hiệu **one-liner*** nên điều này không chấp nhận được.

Vậy nên Python có một vòng lặp khác giúp làm chuyện này đơn giản và ngắn gọn hơn chính là **vòng lặp for**.

Chú thích **One-liner**: Nhiều thuật toán dài hàng chục dòng có thể được viết ngắn gọn trong Python chỉ bằng một dòng. Điều này khá phổ biến với nhiều ngôn ngữ scripting đặc biệt trong số đó là Python.

Cấu trúc vòng lặp for và cách hoạt động

Chúng ta sẽ cùng tìm hiểu phần cấu trúc trước:

```
for variable_1, variable_2, .. variable_n in sequence:  
    # for-block
```

Sequence ở đây là một **iterable object** (có thể là **iterator** hoặc là một dạng **object** cho phép sử dụng **indexing** hoặc thậm chí không phải hai kiểu trên).

Lưu ý: Nếu **sequence** là một **iterator object** thì việc dùng vòng lặp duyệt qua cũng sẽ tương tự như bạn sử dụng hàm **next**.

Ở cấu trúc vòng lặp này, bạn có thể **for** bao nhiêu biến theo sau cũng được. Nhưng phải đảm bảo một điều rằng, nếu bạn **for** với **n** biến thì mỗi phần tử trong **sequence** cũng phải bao gồm **n** (không lớn hơn hoặc nhỏ hơn) giá trị để **unpacking** (gỡ) đưa cho **n** biến của bạn.

Giả sử bạn có một **sequence** gồm 2 phần tử. Mỗi phần tử gồm 3 giá trị.

Bạn đưa vào vòng **for** gồm 3 biến h, k, t.

Bây giờ là nói về cách hoạt động của vòng lặp for này.

Bước 1: Vòng for sẽ bắt đầu bằng cách lấy **giá trị đầu tiên** của sequence.

Bước 2: Giá trị đầu tiên này có 3 giá trị. Bạn đưa vào 3 biến. Kiểm tra hợp lệ.

Bước 3: **unpacking** 3 giá trị này và lần lượt gán giá trị này cho ba biến h, k, t.

Dưới đây là một ví dụ **unpacking**:

```
>>> h = (1, 2, 3) # khởi tạo Tuple bình thường
>>> type(h)
<class 'tuple'>
>>>
>>> h, k, t = (1, 2, 3) # unpacking.
>>> h
1
>>> k
2
>>> t
3
```

Bước 4: Thực hiện nội dung **for-block**.

Bước 5: Lấy giá trị tiếp theo của **sequence** sau đó làm tương tự như Bước 2, 3, 4.

Bước 6: Lúc này, **sequence** đã hết giá trị. Kết thúc vòng lặp.

Sử dụng vòng lặp để xử lí các iterator và Dict

Lí thuyết là thế! Giờ chúng ta sẽ làm một vài ví dụ bằng cách bắt đầu với vấn đề lúc đầu:

```
>>> iter_ = (x for x in range(3))
>>> iter_ = (x for x in range(3))
>>> for value in iter_:
...     print('->', value)
...
-> 0
-> 1
-> 2
>>> value # biến value giàn tiếp được khai báo
2
>>> next(iter_) # hãy học cách tiếp kiệp. Đây là object chỉ dùng một lần.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Tiếp đến chúng ta sẽ dùng vòng lặp này để duyệt một Dict. Nếu như một số ngôn ngữ khác phải có một vòng lặp mới **for-reach** thì với Python lại không cần.

Trước tiên hãy ôn lại bài cũ. Bạn còn chờ **phương thức items** của lớp Dict chứ? (nếu không, bạn có thể tham khảo lại trong bài [KIẾU DỮ LIỆU DICT TRONG PYTHON](#))

```
>>> howteam = {'name': 'Kteam', 'kter': 69}
>>> howteam.items()
dict_items([('name', 'Kteam'), ('kter', 69)])
```

Dict-items không phải là một **iterator object**. Cũng không phải là một **object** cho phép bạn **indexing**. Nhưng nó vẫn là một **iterable**, nên ta có thể dùng

một constructor nào đó để biến đổi nó về một thứ dễ xem xét hơn. Chẳng hạn thế này.

```
>>> list_values = list(team.items())
>>> list_values
[('name', 'Kteam'), ('kter', 69)]
>>> list_values[0]
('name', 'Kteam')
>>> list_values[-1]
('kter', 69)
```

Từ đó, ta có thể dễ dàng suy ra cách để có thể có được một vòng lặp duyệt một Dict. Và đây là ví dụ:

```
>>> for key, value in team.items():
...     print(key, '=>', value)
...
name => Kteam
kter => 69
```

Câu lệnh break, continue

Những câu lệnh này có chức năng hoàn toàn tương tự như trong **vòng lặp while**.

Ví dụ về câu lệnh **break** trong vòng lặp **for**:

```
>>> s = 'How Kteam'
>>> for ch in s:
...     if ch == ' ':
...         break
...     else:
...         print(ch)
...
H
o
w
```

Ví dụ về câu lệnh **continue** trong vòng lặp for

```
>>> s = 'H o w K t e a m'  
>>> for ch in s:  
...     if ch == ' ':  
...         continue  
...     else:  
...         print(ch)  
  
H  
o  
w  
K  
t  
e  
a  
m
```

Cấu trúc vòng lặp for-else và cách hoạt động

Cấu trúc:

```
for variable_1, variable_2, .. variable_n in sequence:  
    # for-block  
  
else:  
    # else-block
```

Nếu bạn nắm rõ cách vòng lặp **while-else** hoạt động thì bạn cũng có thể tự đoán được cách mà **for-else** làm việc.

Cũng sẽ tương tự như **while-else**, vòng lặp hoạt động bình thường. Khi vòng lặp kết thúc, khối **else-block** sẽ được thực hiện. Và đương nhiên nếu trong quá trình thực hiện **for-block** mà xuất hiện câu lệnh **break** thì vòng lặp sẽ kết thúc mà bỏ qua **else-block**.

- **For-else** bình thường:

```
>>> for k in (1, 2, 3):
...     print(k)
... else:
...     print('Done!')
...
1
2
3
Done!
```

- **For-else** có **break**:

```
>>> for k in (1, 2, 3):
...     print(k)
...     if k % 2 == 0:
...         break
... else:
...     print('Done!')
...
1
2
```

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [VÒNG LẶP WHILE TRONG PYTHON](#).

1.

```
five_even_numbers = []
k_number = 1

while len(five_even_numbers) < 5:
    if k_number % 2 == 0:
        five_even_numbers.append(k_number)
    k_number += 1
```

2.

```
with open('draft.txt') as f:
    # lấy nội dung của file dưới dạng một list
    data = f.readlines()

idx = 0 # mốc bắt đầu
length = len(data) # mốc kết thúc
new_content = "" # nội dung mới sẽ ghi vào file mới

while idx < length:
    # tách một dòng thành một list
    line_list = data[idx].split()
    idx_line = 0
    length_line = len(line_list)
    while idx_line < length_line:
        if line_list[idx_line] == 'Kteam':
            # thay thế chữ trước Kteam là How
            line_list[idx_line - 1] = 'How'
        idx_line += 1
    # nối lại thành một dòng chuỗi
    new_content += ''.join(line_list) + '\n'
    idx += 1
```

```
with open('kteam.txt', 'w') as new_f:  
    # ghi nội dung mới vào file kteam.txt  
    new_f.write(new_content)
```

3.

```
lst = [56, 14, 11, 756, 34, 90, 11, 11, 65, 0, 11, 35]
```

```
idx = 0  
max_idx = len(lst) - 1  
  
max_jdx = len(lst)  
  
while idx < max_idx:  
    if lst[idx] == 11:  
        idx += 1  
        continue  
    jdx = idx + 1  
    while jdx < max_jdx:  
        if lst[jdx] == 11:  
            jdx += 1  
            continue  
        if lst[idx] > lst[jdx]:  
            lst[idx], lst[jdx] = lst[jdx], lst[idx]  
            jdx += 1  
    idx += 1
```

Câu hỏi củng cố

1. Hãy dự đoán kết quả của hàm `next` dưới đây. Giải thích tại sao?

```
>>> iter_ = (x for x in range(3))  
>>> for value in iter_:  
...     print(non_exist_variable)  
...  
Traceback (most recent call last):
```

```
File "<stdin>", line 2, in <module>
NameError: name 'non_exist_variable' is not defined
>>>
>>> next(iter_) # kết quả là gì?
```

2. Sử dụng vòng lặp để tính tổng các số trong set sau đây

```
>>> set_ = {5, 8, 1, 9, 4}
```

Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Qua bài viết này, Bạn đã biết sơ lược về VÒNG LẶP FOR TRONG PYTHON.

Ở bài viết sau. Kteam sẽ tiếp tục đề cập đến [VÒNG LẶP FOR TRONG PYTHON.](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".

Bài 27: VÒNG LẶP FOR TRONG PYTHON – PHẦN

2

Xem bài học trên website để ủng hộ Kteam: [Vòng lặp For trong Python – Phần 2](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [VÒNG LẶP FOR TRONG PYTHON](#).

Và ở bài này Kteam sẽ tiếp tục tìm hiểu với các bạn **Vòng lặp For trong Python**.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- [CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON](#)
- [CÂU ĐIỀU KIỆN IF TRONG PYTHON](#)
- [VÒNG LẶP WHILE](#) và [VÒNG LẶP FOR TRONG PYTHON](#)

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Kiểu dữ liệu range (dãy số)
- Sự khác nhau giữa sequence scan và indexing scan
- Comprehension
- Giới thiệu hàm enumerate

Kiểu dữ liệu range (dãy số)

Bạn gặp kiểu dữ liệu này suốt các phần liên quan đến **comprehension** hoặc là liên quan đến **iterator object**.

Đây là một kiểu dữ liệu rất đặc biệt vì ta có thể lấy nhiều giá trị từ nó nhưng bản chất thì nó không lưu giữ những giá trị mà chúng ta lấy. Trước khi đến với điều thú vị này, chúng ta cùng ngó tổng quát về kiểu dữ liệu này.

Chúng ta có hai cách khởi tạo.

Cách khởi tạo thứ nhất

Cú pháp:

range(stop)

Với cách này, ta sẽ tạo một dãy số bắt đầu bằng số **0** và kết thúc là **stop - 1**. Dãy số này là một cấp số cộng với công sai là **1**.

```
>>> k = range(3)
>>> type(k)
<class 'range'>
>>> k[0] # range có hỗ trợ indexing
0
>>> k[1]
1
>>> k[-1]
```

```

2
>>> k[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: range object index out of range
>>> list(k)
[0, 1, 2]
>>> k[0] = 10 # range object là hasable object
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'range' object does not support item assignment

```

Cách khởi tạo thứ hai

Cú pháp:

range(start, stop[, step])

Với cú pháp này, ta sẽ tạo một dãy số bắt đầu bằng **start** và kết thúc là **stop** – **1.** Dãy số này là một cấp số cộng với công sai là **1**.

Trong trường hợp **step** (buộc phải khác **0**) được đưa vào thì công sai sẽ là **step**.

```

>>> list(range(2, 5))
[2, 3, 4]
>>> list(range(4, 1, -1))
[4, 3, 2]
>>> list(range(2, -3, -1))
[2, 1, 0, -1, -2]

```

Và đây là điều thú vị của hàm **range**. Hãy tạo một List chứa một dãy số cộng từ **0** tới một số kha khá lớn. đương nhiên là cũng sẽ có một Range có một dãy số tương tự.

```
>>> k = range(99999999) # nếu máy bạn có khỏe thì hãy cho số lớn hơn tí nữa để  
thấy rõ sự khác biệt  
>>> lst = list(k)
```

Tiếp đến, hãy dùng toán tử **in**

```
>>> 9999999999 in k  
False  
>>> 9999999999 in lst  
False
```

Nếu bạn chưa thấy gì thì hãy thử số nào то hơn chút. Còn nếu thấy rồi, thì đó chính là tốc độ. Chênh nhau vài mili giây. Đối với máy tính hiện đại, một vài mili giây là đủ để làm rất nhiều thứ. Vậy điều gì làm nên khác biệt đó?

Range là một lớp được thiết kế riêng để lưu giữ những dãy số. Vậy nên nó đã được những kĩ sư Python sử dụng các thuật toán để có thể có được sự linh hoạt này.

Mỗi lần bạn lấy một giá trị trong một đối tượng thuộc hàm range thì đối tượng này sẽ lấy các giá trị của **start**, **stop**, **step** và một vài thứ khác để tính toán và sinh ra một con số.

Để hiểu rõ hơn bạn tham khảo câu hỏi này trên **Stack Overflow**

[Why is "1000000000000000 in range\(1000000000000001\)" so fast in Python 3?](#)

Sử dụng range để duyệt một List, Tuple, Chuỗi

Chúng ta sử dụng một dãy số để dùng **indexing** lấy các giá trị trong một List, Tuple hoặc Chuỗi.

Chúng ta có hàm **range** sinh ra một dãy số.

Kết hợp chúng lại, ta có thể duyệt một List, Tuple hoặc Chuỗi:

```
>>> lst = [s, (1, 2, 3), {'abc', 'xyz'}]
>>> for i in range(len(lst)):
...     print(lst[i])
...
How Kteam
(1, 2, 3)
{'abc', 'xyz'}
```

Sự khác nhau giữa sequence scan và indexing scan

Trong bài trước, bạn thấy rằng ta không cần dùng tới hàm **range** vẫn có thể duyệt hết các phần tử của một List. Vậy điều gì khiến chúng ta đôi lúc phải dùng tới hàm **range** để xử lí một List?

Đó là khi ta **cần update (cập nhật) List**. Hãy xem hai ví dụ sau đây:

Đầu tiên là **sequence scan**

```
>>> lst = [1, 2, 3]
>>> for value in lst:
...     value += 1
...
>>> lst
[1, 2, 3]
```

Biến variable là một biến riêng lẻ, nên không thể cập nhật được List ban đầu.

Còn đối với **indexing scan**

```
>>> lst = [1, 2, 3]
>>> for i in range(len(lst)):
...     lst[i] += 1
```

```
...  
>>> lst  
[2, 3, 4]
```

Hãy lựa chọn cách sử dụng vòng lặp một cách thông minh phù hợp với mục đích của mình.

Comprehension

Có lẽ bây giờ những **comprehension** không còn phức tạp với các bạn nữa.

Comprehension là một công cụ rất hiệu quả của Python để xử lý rất nhiều việc mà chỉ cần một dòng.

Bên cạnh đó. Người ta còn so sánh những **comprehension** và những đoạn code với chức năng tương tự thì comprehension **có tốc độ nhanh hơn**.

Lời tác giả:

- Mọi người sẽ phải Ồ lên khi thấy bạn có một comprehension **chỉ tốn một dòng** và **thời gian thực thi nhanh hơn**. Thế nên bạn nên luyện tập sử dụng comprehension thường xuyên.
- Sau này khi kết hợp với **anonymous function** là lambda bạn sẽ tạo ra được những thứ mang đậm thương hiệu **one-liner**.
- Python không khó. Quan trọng là bạn phải nắm lòng các **API** của Python (các chức năng mà ngôn ngữ hỗ trợ) là một trong những thứ đó

Ta có thể tổng quát đơn giản cú pháp của một **comprehension** như sau

Cú pháp:

```
[ output-expression for-statement optional-predicate ]
```

Ở đây Kteam sử dụng **[]** cho List, các bạn có thể sử dụng các cặp ngoặc khác nhưng phải để **output-expression** phù hợp với kiểu dữ liệu. Như dict thì bạn phải để **output-expression** là một cặp **key-value**.

Một số ví dụ

```
>>> ['--'.join((a.capitalize(), b.upper() + c.lower())) for a, b, c in [('how', 'kteam', 'EDUCATION'), ('chia', 'sẻ', 'FREE')]] # bỏ trống optional-predicate
['How--KTEAMeducation', 'Chia--SẺfree']
```

Nếu không sử dụng **comprehension** thì sẽ như sau:

```
>>> lst = []
>>> for a, b, c in [('how', 'kteam', 'EDUCATION'), ('chia', 'sẻ', 'FREE')]:
...     a = a.capitalize()
...     b = b.upper()
...     c = c.lower()
...     lst.append('--'.join((a, b + c)))
...
>>> lst
['How--KTEAMeducation', 'Chia--SẺfree']

>>> {key:value + 1 for key, value in (('Kteam', 69), ('Tèo', 50), ('Tün', 14), ('Free Education', 93)) if value % 2 != 0}
{'Kteam': 70, 'Free Education': 94}
```

Khi không sử dụng **comprehension**

```
>>> dic = {}
>>> for key, value in (('Kteam', 69), ('Tèo', 50), ('Tün', 14), ('Free Education', 93)):
...     if value % 2 != 0:
...         dic[key] = value + 1
...
>>> dic
{'Kteam': 70, 'Free Education': 94}
```

Giới thiệu hàm enumerate

Giả sử bạn có một danh sách học sinh.

```
>>> student_list = ['Long', 'Trung', 'Giàu', 'Thành']
```

Việc in ra danh sách này thì rất đơn giản.

```
>>> for student in student_list:  
...     print(student)  
...  
Long  
Trung  
Giàu  
Thành
```

Nhưng như vậy thì không rõ ràng cho lắm vì danh sách này không hề có số thứ tự. Bạn nghĩ đến việc sử dụng hàm **range**.

Đó cũng là một cách, nhưng Python có hỗ trợ cho bạn một hàm hay hơn đó chính là **enumerate**. Hàm có cú pháp như sau:

Cú pháp:

```
enumerate(iterable[, start])
```

Nếu **start** không được gửi vào thì mặc định là **0**

Hàm này là một **generator** nhờ câu lệnh **yield** trong hàm. Nó sẽ tạo ra mỗi giá trị là một cặp gồm số thứ tự và giá trị có cấu trúc như sau

```
(start + 0, seq[0]), (start + 1, seq[1]), (start + 2, seq[2]), ...
```

Ví dụ:

```
>>> gen = enumerate(student_list)
>>> gen
<enumerate object at 0x02D6D850>
>>> list(gen)
[(0, 'Long'), (1, 'Trung'), (2, 'Giàu'), (3, 'Thành')]
```

Và khi đó, ta có thể sử dụng vòng for như sau

```
>>> for idx, student in enumerate(student_list):
...     print(idx, '=>', student)
...
0 => Long
1 => Trung
2 => Giàu
3 => Thành
```

Nếu bạn không thích bắt đầu từ số 0 thì ta cũng có thể thay đổi

```
>>> for idx, student in enumerate(student_list, 1):
...     print(idx, '=>', student)
...
1 => Long
2 => Trung
3 => Giàu
4 => Thành
```

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [VÒNG LẶP FOR TRONG PYTHON](#)

- Kết quả là **1**. Chính xác là giá trị thứ hai của biến `iter_`

```
>>> next(iter_)
1
```

Python là ngôn ngữ thông dịch. Vậy nên nó sẽ đọc từng câu lệnh. Và như đã đề cập trong cách làm việc của vòng lặp này. Nó sẽ lấy giá trị từ **sequence** gán cho biến rồi mới vào trong **for-block**. Vậy nên sau khi có giá trị, vòng trong **for-block** mới có lỗi phát sinh. Khi đó, chúng ta đã vừa lấy mất đi một giá trị của biến `iter_`. Vậy nên khi dùng hàm `next` thì kết quả sẽ là kết quả thứ hai.

- 2.

```
>>> set_ = {5, 8, 1, 9, 4}
>>> sum_of_set = 0
>>> for value in set_:
...     sum_of_set += value
...
>>> sum_of_set
27
```

Câu hỏi củng cố

- Sử dụng **sequence scan** để thay đổi phần tử đầu tiên của mỗi phần tử trong List dưới đây thành **None**

```
>>> lst = [[1, 2, 3], [4, 5, 6]]
Sau khi thay đổi
>>> lst
[[None, 2, 3], [None, 5, 6]]
```

- Một spiral matrix là một ma trận vuông **$n \times n$** (n cột, n hàng) gồm N^2 số tự nhiên đầu tiên. Trong đó số tăng tuần tự đi xung quanh các mép của mảng xoắn bên trong nó.

Ví dụ với một spiral matrix 5x5 thì sẽ như sau:

0	1	2	3	4
15	16	17	18	5
14	23	24	19	6
13	22	21	20	7
12	11	10	9	8

Viết một đoạn script yêu cầu nhập số n (chính là số cột - hàng) của một spiral matrix. Sau đó dùng vòng lặp tạo một spiral matrix in ra shell (Nếu in ra số có một chữ số như 0, 1, 2,... thì thêm trước đó là chữ số 0 -> 00, 01, 02,...)

Với spiral matrix như trên sẽ được in ra như sau:

00	01	02	03	04
15	16	17	18	05
14	23	24	19	06
13	22	21	20	07
12	11	10	09	08

Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Qua bài viết này, Bạn đã biết nhiều hơn về VÒNG LẶP FOR TRONG PYTHON.

Ở bài viết sau. Kteam sẽ giới thiệu với các bạn [HÀM TRONG PYTHON](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên "**Luyện tập – Thủ thách – Không ngại khó**".

Bài 28: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON - SƠ LƯỢNG VỀ HÀM

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Function trong Python - Sơ lược về hàm](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [VÒNG LĂP FOR TRONG PYTHON](#).

Và ở bài này Kteam sẽ lại tìm hiểu với các bạn **KIỂU DỮ LIỆU FUNCTION TRONG PYTHON - Sơ lược về hàm.**

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).

- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- [CÂU ĐIỀU KIÊN IF TRONG PYTHON](#)
- [VÒNG LẶP WHILE](#) và [VÒNG LẶP FOR TRONG PYTHON](#)
- [NHẬP XUẤT TRONG PYTHON](#)

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Vấn đề.
- Khai báo hàm. (Create function)
- Gọi hàm. (Call function)
- Đừng viết lại code! (DRY – Don't Repeat Yourself)
- Parameter và Argument.
- Giá trị mặc định của parameter. (default argument)

Vấn đề

Bạn đã được giới thiệu qua [HÀM PRINT](#). Vậy bạn có biết người ta đã tạo nên hàm đó như thế nào không? Kteam nghĩ bạn cũng không nên tìm hiểu làm gì vì vốn nó cũng rất phức tạp!

Vậy nếu bây giờ không có hàm print thì có phải mỗi lần bạn muốn in ra thứ gì đó trên **Shell** thì bạn phải viết một dãy lệnh dài để có thể làm điều đó đúng chứ?

Ta thử tính đơn giản thôi!

Ví dụ: hàm print chỉ tốn 10 dòng để có thể in ra một chuỗi (thật sự là nhiều hơn vậy rất nhiều), vậy nếu bạn dùng 10 lần print thì nó đã tới 100 dòng.

Mà với một chương trình, liệu bạn chỉ có sử dụng mỗi hàm **print**? Nếu không nhờ những kĩ sư đã viết sẵn cho chúng ta rất nhiều hàm để cho chúng ta sử dụng liệu chúng ta tốn mất bao lâu và bao nhiêu dòng code cho một **script** in ra dòng chữ "[Hello Kteam!](#)" ở trên Shell?

Thời gian xưa, con người ta khi viết các dòng code thì sẽ viết từ trên xuống, lệnh nào làm trước thì viết trước và cứ thế hoàn thành đoạn **script**. Ta gọi đó là **Lập trình tuyến tính** (linear programming).

Và khi nhiều vấn đề phát sinh từ **linear programming** như việc sửa đổi, cập nhật, rất khó khăn và nhiều nguyên nhân khác đã đưa ra một thời kì lập trình mới, đó chính là **Lập trình thủ tục** (procedural programming)

Để có thể có một chương trình theo hướng **procedural programming**, thì ta phải biết khái niệm hàm và cụ thể trong bài này, Kteam sẽ giới thiệu với các bạn về nó.

Khai báo hàm (create function)

Ở đây, Kteam sử dụng từ "**khai báo**", và với tựa bài có cụm từ "**kiểu dữ liệu**" để muốn nói với bạn rằng trong Python, những hàm mà ta tạo là những biến đặc biệt mà ta khai báo.

Bạn nên nắm rõ điều này để sau này có khi bạn sẽ tiếp cận tới khái niệm **meta class** (siêu lớp) sẽ hiểu rõ hơn.

Để khai báo một hàm, ta sử dụng từ khóa "**def**" với cú pháp như sau

Cú pháp:

```
def <function_name>(parameter_1, parameter_2, .., parameter_n):  
    function-block
```

Trong cú pháp đó, bạn không được bỏ sót bất kì thứ nào ngoại trừ bạn có thể bỏ trống các **parameter**.

Ví dụ:

```
>>> def kteam():  
...     pass  
...  
>>> kteam  
<function kteam at 0x014EC5D0>
```

```
>>> type(kteam)
<class 'function'>
```

Lưu ý:

Lệnh `pass` ở trên là một lệnh “**giữ chỗ**” (placeholder statement) để giúp cho các **block** của Python không bị thiếu câu lệnh trong trường hợp bạn chưa biết viết gì cho phù hợp.

Bạn có thể thấy, khi in ra hàm `kteam`, bạn sẽ nhận được một dòng khá tương một một **generator expression**.

Gọi hàm (call function)

Việc gọi hàm, ta có cú pháp sau đây

Cú pháp:

```
<function>()
```

Khi gọi hàm, các câu lệnh có trong hàm sẽ được thực thi

Ví dụ:

```
>>> def kteam():
...     print('Hello Kteam!')
...
>>> kteam
<function kteam at 0x0323FD68>
>>> kteam()
Hello Kteam!
```

Ta gọi hàm `kteam`, vậy nên hàm `kteam` sẽ thực thi các lệnh mà nó có. Cụ thể ở đây là nó dùng hàm `print` in ra màn hình một dòng chuỗi.

Đừng viết lại code (DRY - Don't Repeat Yourself)

Giả sử, bạn có một **script** với nhiệm vụ in ra 8 dòng in ra "Hello Kteam!" và "Free Education"

```
print('Hello Kteam!')  
  
print("Free Education")  
  
print('Hello Kteam!')  
  
print("Free Education")  
  
print('Hello Kteam!')  
  
print("Free Education")  
  
print('Hello Kteam!')  
  
print("Free Education")
```

Lưu ý:

Việc sử dụng vòng lặp để làm chuyện này là khả thi, nhưng có nhiều trường hợp các câu lệnh không nằm liền kề nhau như thế này, bạn không thể dùng vòng lặp rút gọn.

Bây giờ, bạn muốn thay đổi dòng "Hello Kteam!" thành "Hi Kteam!", vậy là bạn phải chỉnh sửa lại 4 dòng lệnh.

Giờ ta đưa vấn đề xa hơn một tí nữa. Nếu nhiệm vụ của bạn không chỉ là **print** ra tám dòng chữ mà còn phải làm nhiều thứ khác, thì có phải bạn đang viết lại rất nhiều code không? Và khi chỉnh sửa mà nếu chỉnh sửa nhiều thì bạn sẽ phải mất rất nhiều công sức.

Để có thể tránh được việc đó, ta hãy sử dụng hàm

```
def kteam():  
    print('Hello Kteam!')
```

```
print("Free Education")
kteam()
kteam()
kteam()
kteam()
```

Và khi muốn chỉnh sửa, ta chỉ cần chỉnh sửa bên trong hàm, thì ta sẽ thay đổi được tất cả.

Parameter và Argument

Đầu tiên, ta khởi tạo một hàm có các **parameter**

```
>>> def kteam(text):
...     print(text)
```

Và khi gọi hàm có **parameter**, bạn phải truyền vào **argument** tương ứng.

```
>>> kteam('Hello Kteam!')
Hello Kteam!
```

Ở đây, **argument** chúng ta đưa vào là một chuỗi. Chuỗi này sẽ được đưa vào gán cho **parameter** tương ứng là text. Và rồi hàm thực hiện việc in text ra.

Đương nhiên là chúng ta có thể biến hóa nhiều ra nữa

```
>>> def kteam(greeting, name):
...     print(greeting, name + '!')
...
>>> kteam('Hi', 'Kteam')
Hi Kteam!
>>> kteam('Hello', 'SpaceX')
Hello SpaceX!
```

Giá trị mặc định của parameter (Default argument)

Hãy coi ví dụ sau:

```
>>> def kteam(greeting, name):
...     print(greeting, name + '!')
...
>>> kteam('Hi', 'Kteam')
Hi Kteam!
>>> kteam('Hello', 'SpaceX')
Hello SpaceX!
>>> kteam('Hi', 'Tesla')
Hi Tesla!
>>> kteam('Hi', 'Python')
Hi Python!
>>> kteam('Hi', 'Jack')
Hi Jack!
```

Ta thấy, tần suất xuất hiện chuỗi “**Hi**” cho parameter `greeting` rất cao. Giờ ta cần một **parameter** giữ giá trị là chuỗi “**Hi**” nhưng vẫn cho ta thay đổi khi cần. Bấy giờ, ta nên sử dụng **default argument**.

```
>>> def kteam(name, greeting='Hi'):
...     print(greeting, name + '!')
...
>>> kteam('Kteam')
Hi Kteam!
>>> kteam('SpaceX')
Hi SpaceX!
>>> kteam('SpaceX', 'Hello')
Hello SpaceX!
```

Lưu ý:

Khi bạn đưa **default argument** cho các **parameter**, phải để các **parameter** có **default argument** ở sau cùng.

Default argument là một **unhashable container**

Như các bạn đã biết, **unhashable container** phổ biến mà ta đã từng biết như [LIST](#), [DICT](#), [SET](#). Ở đây có một cảnh báo cho bạn việc bạn sử dụng **default argument** cho **parameter** là một **unhashable container** đó là giá trị của nó không được làm mới (refresh) sau mỗi lần gọi hàm mà không **pass argument** mới cho **parameter** đó. Đương nhiên là nếu bạn có **pass** cho nó một **argument** mới thì **container** đó vẫn không hề mất giá trị nếu lần sau bạn gọi nó.

```
>>> def f(kteam=[]):
...     kteam.append('F')
...     print(kteam)
...
>>> f()
['F']
>>> f()
['F', 'F']
>>> f()
['F', 'F', 'F']
>>> f([1, 2, 3])
[1, 2, 3, 'F']
>>> f()
['F', 'F', 'F', 'F']
```

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [VÒNG LĂP FOR TRONG PYTHON - Phần 2](#)

1.

```
>>> for l in lst:
...     l[0] = None
```

2.

```
n = int(input('Enter size of matrix: '))
dx, dy = 1, 0
x, y = 0, 0
spiral_matrix = [[None] * n for j in range(n)]

for i in range(n ** 2):
    spiral_matrix[x][y] = i
    nx, ny = x + dx, y + dy
    if 0 <= nx < n and 0 <= ny < n and spiral_matrix[nx][ny] == None:
        x, y = nx, ny
    else:
        dx, dy = -dy, dx
        x, y = x + dx, y + dy

for y in range(n):
    for x in range(n):
        print("%02i" % spiral_matrix[x][y], end=' ')
    print()

print()
```

Kết luận

Qua bài viết này, Bạn đã biết một chút về Kiểu dữ liệu Function trong Python.

Ở bài viết sau. Kteam sẽ tiếp tục giới thiệu thêm với các bạn về [KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – Phần 2](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.

Bài 29: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON - POSITIONAL VÀ KEYWORD ARGUMENT

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Function trong Python - Positional và keyword argument](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn KIỂU DỮ LIỆU FUNCTION TRONG PYTHON.

Và ở bài này Kteam sẽ lại tìm hiểu với các bạn **Kiểu dữ liệu Function trong Python - Positional và keyword argument**.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- [CÂU ĐIỀU KIÊN IF TRONG PYTHON](#)
- [VÒNG LẶP WHILE](#) và [VÒNG LẶP FOR TRONG PYTHON](#)
- [NHẬP XUẤT TRONG PYTHON](#)

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Positional argument và keyword argument
- Bắt buộc (force) positional argument và keyword argument

Positional argument và keyword argument

Với một hàm thông thường như sau

```
>>> def kteam(a, b):
...     pass # lệnh giữ chỗ.
```

Thì ta có thể **pass argument** vào cho hàm như sau

```
>>> kteam(3, 'Free Education')
```

Trong ví dụ trên, hai giá trị là số 3 và chuỗi 'Free Education' gọi là positional argument.

Còn với trường hợp dưới đây

```
>>> kteam(a=3, b='Free Education')
```

Thì hai giá trị trên (chính là số 3 và chuỗi 'Free Education') là những keyword argument.

Sau đây là những điều tuy nhỏ nhưng bạn cần phải biết. Khi **pass argument** theo **positional argument**. Thì các argument sẽ được gán **LẦN LƯỢT** cho các parameter. Riêng đối với **keyword argument**. Bạn đã tự mình gán giá trị cho các parameter. Vậy nên:

```
>>> def kteam(a, b):
...     print('a', a)
...     print('b', b)
...
>>> kteam(a=3,b=4)
a 3
b 4
>>> kteam(b=3,a=4)
a 4
b 3
```

Hai cách gọi hàm trên đều tương tự như nhau.

Một điều nữa là bạn không được phép để **positional** theo sau (follow) **keyword**.

Có nghĩa là bạn có thể **pass argument** vừa **positional** và **keyword** cùng một lúc được, nhưng những positional buộc phải đứng trước **keyword**.

Trường hợp ngớ ngẩn của Tèo sau đây sẽ cho bạn biết điều đó:

```
>>> def teo_with_sone(name, verb):
...     print('Teo', verb + 's', name)
...
>>> teo_with_sone('Python', 'love')
Teo loves Python
>>> teo_with_sone('HTML', verb='like')
Teo likes HTML
>>> teo_with_sone(verb='like', 'CSharp')
File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
>>> teo_with_sone(name='Java', 'hate')
File "<stdin>", line 1
```

```
SyntaxError: positional argument follows keyword argument
```

Bắt buộc (force) Positional argument và keyword argument

Keyword argument

Trong Python, có một số hàm bắt chúng ta buộc phải **pass argument** một cách rõ ràng rành mạch như hàm **sorted**.

```
>>> sorted([3, 4, 1], reverse=True)
[4, 3, 1]

>>> sorted([3, 4, 1], True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must use keyword argument for key function
```

Bạn thấy đấy, ta không thể **pass argument** cho parameter reverse theo positional argument.

Việc thiết kế này cũng rất tiện lợi vì nhiều trường hợp nhiều parameter cùng một lúc đều có default argument value. Hãy xem vấn đề sau đây mà Tèo mắc phải.

Tèo có một hàm

```
>>> def Teo(a, b=2, c=3, d=4):
...     f = (a + d) * (b + c)
...     print(f)
...
>>> Teo(1)
25
>>> Teo(1, 2, 3, 5)
```

30

Tèo gọi hàm thì thấy kết quả theo đúng ý mình. Giờ Tèo muốn đổi giá trị của parameter **d** thành **5**. Nên Tèo phải truyền lại các giá trị **2** và **3** cho các parameter **b** và **c**.

Vậy, ta có cách nào không phải truyền lại hai giá trị cho parameter b và c không?

Có, chính là **keyword argument**.

```
>>> Teo(1, d=5)  
30
```

Đôi lúc, chúng ta nên sử dụng **keyword argument** để tiện lợi và rõ ràng.

Python cho phép chúng ta tạo ra các parameter chỉ nhận giá trị bằng việc pass argument theo kiểu keyword argument.

Cú pháp

```
def function (*, key_arg1, key_arg2):  
    # function-block
```

Khi tạo một hàm mà có một parameter *****. Thì Python sẽ hiểu đó không phải là parameter mà chính là **syntax** để rồi nó biến các parameter sau ***** thành các **keyword only argument** (chỉ nhận giá trị theo kiểu keyword argument)

Ví dụ là dễ hiểu nhất!

```
>>> def kteam(pos_or_key_arg, *, key_arg1, key_arg2):  
...     print(pos_or_key_arg)  
...     print(key_arg1)  
...     print(key_arg2)  
...
```

```
>>> kteam(1, key_arg1=2, key_arg2='Kteam')
1
2
Kteam
>>> kteam(1, 2, key_arg2='Kteam')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kteam() takes 1 positional argument but 2 positional arguments (and 1
keyword-only argument) were given

>>> kteam(1, 2, 'Kteam')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kteam() takes 1 positional argument but 3 were given
```

Lưu ý: ta có thể thay thế dấu * bằng *identifier. Tuy nhiên phổ biến vẫn là *.

Positional argument

Bạn còn nhớ hàm input chứ? Kteam từng nói với bạn rằng cú pháp của hàm input có thể được viết như thế này.



```
input(prompt=None, /)
```

Dấu / chính là một **syntax** để **force parameter prompt** trở thành **positional only argument**. Có nghĩa là bạn chỉ có thể pass argument cho **parameter prompt** theo kiểu **positional**. Chính xác thì dấu / sẽ biến các parameter đứng trước nó thành **positional only argument**

Tuy nhiên Kteam sẽ không đi sâu vào positional argument vì ở phiên bản Python 3.6.X trở đi không hỗ trợ positional only argument.

Lưu ý: 3.6.X trở đi không hỗ trợ không có nghĩa những bạn cũ hơn một chút xíu như 3.5, 3.4 có hỗ trợ.

Củng cố bài học

Câu hỏi củng cố

Câu hỏi: Dùng hàm help để xem cú pháp của hàm sorted? Sau đó cho biết parameter nào là positional only? Parameter nào là keyword only?

Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Qua bài viết này, Bạn đã biết một chút về hàm trong Python qua các khái niệm positional argument và keyword argument.

Ở bài viết sau. Kteam sẽ tiếp tục giới thiệu thêm với các bạn về [HÀM TRONG PYTHON \(FUNCTION\)](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.

Bài 30: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON - PACKING VÀ UNPACKING ARGUMENTS

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Function trong Python - Packing và unpacking arguments](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn Kiểu dữ liệu [Function trong Python - Positional và keyword argument](#)

Và ở bài này Kteam sẽ lại tìm hiểu với các bạn **Kiểu dữ liệu Function trong Python - Packing và unpacking arguments.**

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- [CÂU ĐIỀU KIÊN IF TRONG PYTHON](#)
- [VÒNG LẶP WHILE](#) và [VÒNG LẶP FOR TRONG PYTHON](#)
- [NHẬP XUẤT TRONG PYTHON](#)

Trong bài này, chúng ta sẽ cùng tìm hiểu những nội dung sau đây:

- Unpacking arguments với *
- Packing arguments với *
- Unpacking arguments với **
- Packing arguments với **

Unpacking arguments với *

Giả sử, bạn có một hàm thế này

```
>>> def kteam(k, t, e, r):
...     print(k)
...     print(t, e)
...     print('end', r)
...
```

Bạn thấy đấy! Hàm này gồm 4 **parameter** và không có **default argument**. Vậy nên khi gọi hàm này, bạn buộc phải đưa vào 4 **argument**.

Nhưng bạn có một vấn đề, 4 argument cần truyền vào khi gọi hàm này lại nằm trong một List.

```
>>> lst = ['123', 'Kteam', 69.96, 'Henry']
```

Chả sao cả, bạn có thể lấy từng phần tử (element) trong list ra một cách dễ dàng, sau đó bạn có thể gọi hàm kteam như thế này.

```
>>> kteam(lst[0], lst[1], lst[2], lst[3])
123
Kteam 69.96
end Henry
```

Phức tạp vấn đề lên một chút nào! Sẽ ra sao nếu bạn có 50 **parameter** và phải gõ hết 50 **indexing** để truyền vào cho hàm khi gọi nó?

Lập trình viên lười lắm, họ không làm chuyện đó đâu. Vậy nên, Python cho phép làm điều đó đơn giản chỉ bằng một dấu *

```
>>> kteam(*lst)
123
Kteam 69.96
end Henry
```

Khi bạn sử dụng *, bạn không chỉ có thể **unpack** được các List mà bên cạnh đó bạn có thể **unpack** các **container** khác như [Tuple](#), [Chuỗi](#), Generator, [Set](#), [Dict](#) (chỉ lấy được key).

Lưu ý:

Pass argument bằng cách **unpacking argument** như thế này là đang truyền vào dưới dạng **positional argument**. Hãy cẩn thận sử dụng kĩ thuật này với những hàm có parameter dạng **keyword-only argument**.

```
>>> def a(*, s, d):
...     print(s, d)
...
>>> a(*('a', 'b'))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: a() takes 0 positional arguments but 2 were given
```

Trong trường hợp container của bạn **unpack** các giá trị có trong container nhưng vẫn chưa đủ yêu cầu của hàm, thì bạn có thể truyền thêm:

```
>>> kteam(*('a', 'b', 'c'), 'd')
a
b c
end d
```

Packing arguments với *

Bạn còn nhớ hàm **print** chứ? Nó có khả năng nhận được bao nhiêu argument cũng được. Làm sao nó làm được như thế?

Đó chính là nhờ **packing argument**. Đôi lúc, bạn sẽ không thể biết trước được bạn sẽ pass vào bao nhiêu argument. Việc kiểm soát điều đó đôi lúc là **bất khả thi**.

Khi bạn sử dụng **packing argument**. Đồng nghĩa với việc bạn nhờ một biến đi gói tất cả các giá trị truyền vào cho hàm bằng **positional argument** thành một Tuple. Nếu không có gì để gói (bạn không truyền vào bất cứ argument nào) thì bạn sẽ nhận được một tuple rỗng. Để giao nhiệm vụ cho một biến làm công việc này, bạn đặt một dấu ***** trước nó.

```
>>> def kteam(*args):
...     print(args)
...     print(type(args))
...
>>> kteam('Kteam', 69.96, 'Henry')
('Kteam', 69.96, 'Henry')
<class 'tuple'>
>>> kteam(*(x for x in range(7))) # unpack sau đó là pack
(0, 1, 2, 3, 4, 5, 6)
<class 'tuple'>
```

Lưu ý:

Bạn không nên nhầm lẫn việc này với việc **force keyword-only argument**

Không được phép để 2 parameter cùng làm nhiệm vụ packing argument trong một hàm

Nếu sau một **packing parameter** còn có những parameter khác, khi gọi hàm muốn truyền giá trị vào cho các parameter sau packing parameter bạn cần phải sử dụng **keyword argument**.

```
>>> def f(*args, kter):
...     print(kter)
...
>>> f('1', '2')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() missing 1 required keyword-only argument: 'kter'
>>> f('1', '2', kter='3')
3
```

Bạn có thể sử dụng kỹ thuật này khi khai báo biến. Kteam sẽ nói về vấn đề này ở một bài khác.

Ở những ví dụ trên các bạn có thể thấy Kteam sử dụng biến packing có tên là **args**. Đó không phải là ngẫu nhiên mà là một quy ước nhỏ của các Pythonist với nhau. Thường người ta sẽ sử dụng biến có tên là **args** (viết gọn của arguments) để làm biến packing.

Trong Python, có rất nhiều quy ước là những luật bất thành văn như cách đặt tên biến, cách định dạng code, cách đặt tên file. Bạn sẽ biết thêm ở một bài khác của Kteam.

Unpacking arguments với **

Ta thử **unpacking** một Dict chỉ với một dấu *

```
>>> dic = {'name': 'Kteam', 'member': 69}
>>> def kteam(a, b):
...     print(a)
...     print(b)
...
>>> kteam(*dic)
name
member
```

Như bạn thấy, chúng ta chỉ lấy được key thôi.

Với Dict, thì nó phức tạp hơn một xíu khi mỗi phần tử là một cặp **key** và **value**. Vậy nên một dấu * là không đủ nội công để **unpack** hết được các giá trị. Do đó ta phải nhờ đến một cặp **.

Nếu bạn **unpacking** một Dictionary để truyền argument vào cho hàm khi gọi hàm thì đây chính là dạng **keyword argument**. Vậy nên bạn phải chắc chắn rằng **parameter** với **key** là giống nhau.

```
>>> dic = {'name': 'Kteam', 'member': 69}
>>> def kteam(a, b):
...     print(a)
...     print(b)
...
>>> kteam(**dic)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kteam() got an unexpected keyword argument 'name'
>>> def kteam(name, member):
...     print('name =>', name)
...     print('member =>', member)
...
>>> kteam(**dic)
name => Kteam
member => 69
```

Packing arguments với **

Đã có **unpacking** với ****** thì cũng phải có packing. Khác với dấu ***** là gói những **positional argument** thì ****** lại gói các **keyword argument**. Và đương nhiên, nó sẽ gói trong một Dict. Nếu không truyền gì vào sẽ là một dict rỗng.

```
>>> def kteam(**kwargs):
...     print(kwargs)
...     print(type(kwargs))
...
>>> kteam(name='Kteam', member=69)
{'name': 'Kteam', 'member': 69}
<class 'dict'>
```

Tên biến **kwargs** (viết gọn của **keyword arguments**) cũng là một quy ước đặt tên.

```
>>> def kteam(**kwargs):
...     for key, value in kwargs.items(): # phương thức items() của kiểu dữ liệu Dict
...         print(key, '=>', value)
...
>>> kteam(id=3424, name='Henry', age=18, lang='Python')
id => 3424
name => Henry
age => 18
lang => Python
```

Lưu ý:

bạn **không** được phép bỏ các **positional parameter** sau biến packing mà có ****** giống như với *****.

```
>>> def f(**a, b):
File "<stdin>", line 1
    def f(**a, b):
        ^
SyntaxError: invalid syntax
```

Nhờ những kiến thức trên, bạn có thể có một hàm cực kì linh hoạt như sau

```
>>> def best_function_ever(*args, **kwargs):
...     # việc còn lại của bạn là thỏa sức biến tấu
...     pass
...
```

Bạn hãy nắm chắc kĩ thuật này, tuy đơn giản nhưng lại được sử dụng rất nhiều.

Củng cố bài học

Đáp án bài trước

Bạn có thể tìm thấy câu hỏi của phần này tại CÂU HỎI CỦNG CỐ trong bài [KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – Phần 2](#).

Đáp án: Ta dùng lệnh `help`

```
>>> help(sorted)
```

Ta sẽ được cấu trúc của hàm **sorted** như sau:

```
sorted(iterable, /, *, key=None, reverse=False)
```

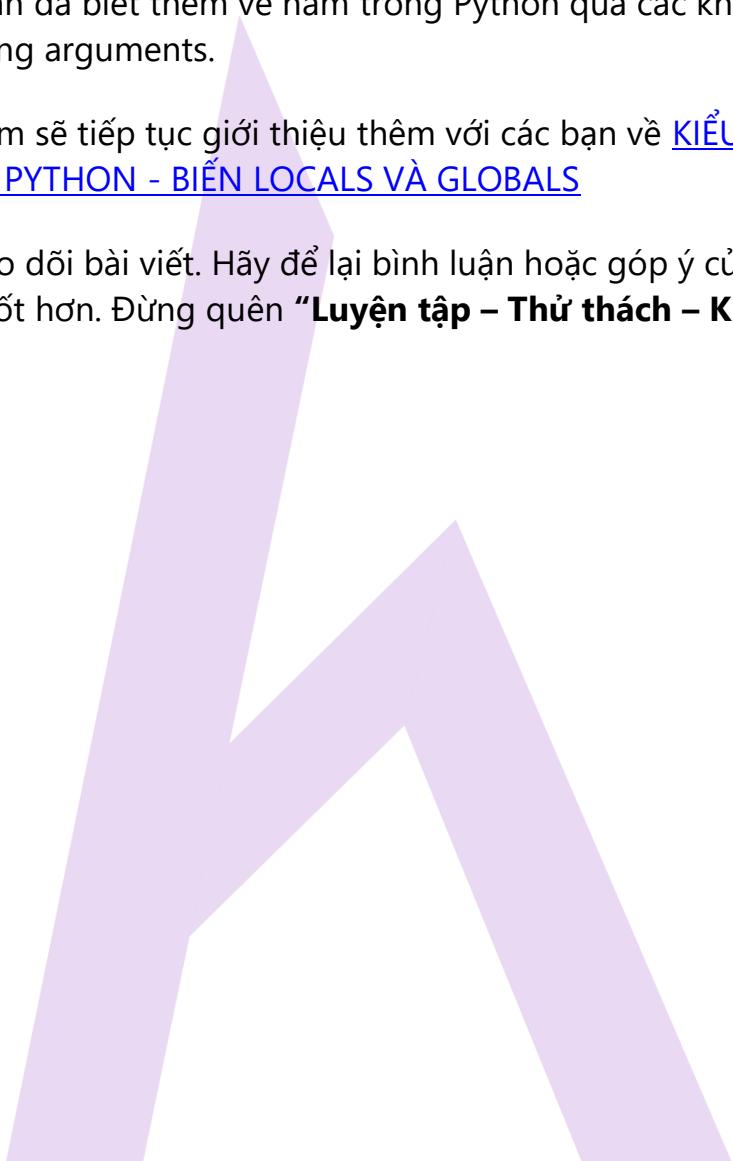
Vậy nên các **positional only** là **iterable**, còn **keyword only** là **key** và **reverse**.

Kết luận

Qua bài viết này, Bạn đã biết thêm về hàm trong Python qua các khái niệm packing và unpacking arguments.

Ở bài viết sau. Kteam sẽ tiếp tục giới thiệu thêm với các bạn về [KIẾU DỮ LIỆU FUNCTION TRONG PYTHON - BIẾN LOCALS VÀ GLOBALS](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.



Bài 31: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON - BIẾN LOCALS VÀ GLOBALS

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Function trong Python - Biến locals và globals](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn KIỂU DỮ LIỆU FUNCTION TRONG PYTHON.

Và ở bài này Kteam sẽ lại tìm hiểu với các bạn **Kiểu dữ liệu Function trong Python - Biến locals và globals**.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).

- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- [CÂU ĐIỀU KIÊN IF TRONG PYTHON](#)
- [VÒNG LẶP WHILE](#) và [VÒNG LẶP FOR TRONG PYTHON](#)
- [NHẬP XUẤT TRONG PYTHON](#)

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Sự khác biệt giữa khai báo biến ở trong hàm và ngoài hàm
- Thay đổi giá trị argument gián tiếp qua parameter
- Sử dụng lệnh global
- Giới thiệu hàm locals và globals

Khai báo biến ở trong hàm

Giả sử, bạn có một đoạn script như sau với việc khai báo một biến ngoài hàm và sử dụng ở trong hàm

```
kteam = 'How Kteam'

def say_slogan():
    print("We are", kteam)
say_slogan()
```

Ta dễ dàng biết được kết quả

```
1  kteam = 'How Kteam'
2
3  def say_slogan():
4      print("We are", kteam)
5
6  say_slogan()
7
We are How Kteam
[Finished in 0.3s]
```

Thử một trường hợp tiếp theo là khai báo biến ở trong hàm và sử dụng ở trong hàm

Kết quả cũng không có gì ngoài dự đoán

```

1 #kteam = 'How Kteam'
2
3 def say_slogan():
4     kteam = 'How Kteam'
5     print("We are", kteam)
6
7 say_slogan()
8
We are How Kteam
[Finished in 0.3s]

```

Giờ hãy thử khai báo một biến trong hàm mà sử dụng ngoài hàm xem nào

```

def make_slogan():
    kteam = 'How Kteam'

    print(kteam)

```

Có ai đoán trật kết quả không nhỉ?

```

1 def make_slogan():
2     kteam = 'How Kteam'
3
4 print(kteam)
5
File "D:\Stuff\a.py", line 4, in <module>  x
6
7 Traceback (most recent call last):
8   File "D:\Stuff\a.py", line 4, in <module>
9       print(kteam)
10      NameError: name 'kteam' is not defined

```

Cụ thể, Python nói với chúng ta rằng biến `kteam` chưa được khai báo. Nhưng rõ ràng là chúng ta đã khai báo nó ở trong hàm rồi mà?

À, là do chưa chạy hàm. Giờ ta sẽ thử lại. Lần này, ta để một dòng lệnh trong hàm luôn để chắc chắn rằng hàm đã chạy

```
def make_slogan():
    kteam = 'How Kteam'
    print('Run successfully!')  
  
make_slogan()
print(kteam)
```

Vẫn lỗi tương tự mặc dù hàm đã chạy

```
1 def make_slogan():
2     kteam = 'How Kteam'
3     print('Run successfully!')
4
5 make_slogan()
6 print(kteam)
7
8 |
Run successfully!
Traceback (most recent call last):
  File "D:\Stuff\1.py", line 6, in <module>
    print(kteam)
NameError: name 'kteam' is not defined
```

Từ đó, chúng ta có thể kết luận rằng việc khai báo biến ở trong hàm là có vấn đề.

Và quả thực là như vậy. Việc bạn khai báo một biến ở một hàm nào đó thì biến đó chỉ có thể sử dụng trong hàm đó. Còn nếu đi ra ngoài hàm khai báo nó thì biến đó hoàn toàn vô danh > thông báo chưa được khởi tạo.

Hãy tưởng tượng như thế này. Bạn sinh ra ở Việt Nam thế nên ở nước Việt Nam người ta có thể biết bạn là ai (có thể là qua CMND hoặc Căn Cước Công Dân). Nhưng nếu bạn đi qua Úc, Nhật hoặc thậm chí là Lào, Campuchia thì không ai biết được bạn là ai, tên gì cả.

Và như ví dụ trên. Bạn đã khai báo biến `kteam` trong hàm thôi thì chỉ trong hàm đó mới biết bạn. Còn nếu quăng ra ngoài thì chương trình chẳng biết nó là thằng nào cả.

Một lần nữa lặp lại đó là biến khai báo ở hàm nào thì chỉ hàm đó mới biết biến đó còn thoát ra ngoài hàm đó thì coi như không có. **Biến khai báo ở hàm cha có thể sử dụng trong hàm con nhưng biến ở hàm con không thể sử dụng ở hàm cha.**

Giống như ví dụ mà bạn thấy, nếu bạn khai báo biến ở ngoài hàm thì bạn hoàn toàn có thể sử dụng biến đó ở trong hàm.

Lưu ý: Biến là đối tượng nên bị ràng buộc bởi điều này. Do đó các HÀM (FUNCTION), LỚP (CLASS) cũng chịu sự ràng buộc này tương tự. Khai báo ở hàm nào thì chỉ dùng ở hàm đó.

Thay đổi giá trị argument gián tiếp qua parameter

Nói về vấn đề này, bạn nên biết 2 thuật ngữ là **pass by reference** và **pass by value**.

Đầu tiên, thế nào là **pass by reference**. Giả sử bạn có một chiếc laptop. Thằng bạn nó muốn mượn dùng một ngày. Thế là bạn mang máy của mình cho nó mượn. Về nhà thằng bạn nó down phim, down game, cài virus. Sau một ngày, hắn đem trả lại bạn. Coi như chiếc laptop của bạn tan nát. Việc bạn đưa laptop của mình cho thằng bạn cũng giống như việc **pass by reference**. Có nghĩa là bạn đưa bản gốc.

Sang tuần sau. Thằng bạn lại đến mượn đồ tiếp và bây giờ là chiếc xe đạp. Bạn biết tổng hắn kiểu gì cũng phá, liền lấy bảo bối là “Gương nhân đôi”, sau đó bạn nhân bản chiếc xe đạp của bạn ra. Bạn lấy bản nhân bản đưa hắn mượn. Và bạn thấy đó, dù cho hắn có đập nát chiếc xe kia thì xe của bạn cũng không sao. Đó gọi là bạn **pass by value**, đưa giá trị hoặc là “đưa bản sao”.

Trong Python, việc bạn **CHỈ CÓ THỂ pass by value**

Ta sẽ đến với ví dụ để hiểu hơn. Ta có đoạn script sau

```
num = 69
st = 'How Kteam'
lst = [1, 2, 3]
tup = tuple('Education')

def change(parameter):
    parameter = 'New value'
    print('Changed successfully!')

change(num)
change(st)
change(lst)
change(tup)
print('*' * 10)
print('{0}\n{1}\n{2}\n{3}'.format(num, st, lst, tup))
```

Và đây là kết quả

```

1 num = 69
2 st = 'How Kteam'
3 lst = [1, 2, 3]
4 tup = tuple('Education')
5
6 def change(parameter):
7     parameter = 'New value'
8     print('Changed successfully!')
9
10 change(num)
11 change(st)
12 change(lst)
13 change(tup)
14 print('*' * 10)
15 print('{0}\n{1}\n{2}\n{3}'.format(num, st, lst, tup))
16
Changed successfully!
Changed successfully!
Changed successfully!
Changed successfully!
*****
69
How Kteam
[1, 2, 3]
('E', 'd', 'u', 'c', 'a', 't', 'i', 'o', 'n')
[Finished in 0.5s]

```



Như bạn thấy, không một giá trị nào bị thay đổi vì đó là **pass by value**.

Tuy nhiên, chúng ta thử lật lại kiến thức một ít nào. Bạn nhớ phần "**Vấn đề cần lưu tâm khi sử dụng List**" Trong bài [KIẾU DỮ LIỆU LIST TRONG PYTHON – Phần 1](#) chứ?

Việc bạn truyền giá trị cho parameter giống hệt như lúc bạn khởi tạo biến parameter và đưa giá trị cho nó vậy.

Xem ví dụ để hiểu nhé!

```

lst = ['How Kteam', 1, 2]

def change(parameter):
    parameter[1] = 'New value'
    print('Changed successfully!')

```

```
print(lst)
change(lst)
print(lst)
```

Kết quả(*)

```
1  lst = ['How Kteam', 1, 2]
2
3  def change(parameter):
4      parameter[1] = 'New value'
5      print('Changed successfully!')
6
7  print(lst)
8  change(lst)
9  print(lst)
10
[ 'How Kteam', 1, 2]
Changed successfully!
[ 'How Kteam', 'New value', 2]
[Finished in 0.2s]
```

Bạn sẽ có một câu hỏi liên quan tới phần này ở cuối bài nhé!

Sử dụng lệnh global

Nếu như một biến nằm trong một hàm (như biến `kteam` trong ví dụ cuối ở phần đầu) thì người ta hay gọi đó là **local variable** (biến chỉ có hiệu lực trong một hàm nhỏ).

Đặt vấn đề là việc khai báo biến ở trong hàm trả nên cần thiết thì sao nhỉ?

Ta được Python hỗ trợ lệnh **global**.

Cú pháp:

global <variable>

Lệnh này như một phép màu mà bạn có thể tạo ra. Giống như bạn có thể biến một người thành tổng thống Mỹ vậy. Ai trên thế giới này cũng biết. Và như biến, ở nơi nào trong chương trình cũng dùng được.

Hãy đến với ví dụ giống như ví dụ cuối phần đầu ban nãy như có khác biệt một chút

```
def make_slogan():
    # khởi tạo với global không có giá trị nhé
    global kteam
    # sau khi khởi tạo xong, ta mới gán giá trị
    kteam = 'How Kteam'

# nhớ là phải chạy hàm nữa
make_slogan()

print(kteam)
```

Và đây là kết quả khi ta có global

```
1 def make_slogan():
2     # khởi tạo với global không có giá trị nhé
3     global kteam
4     # sau khi khởi tạo xong, ta mới gán giá trị
5     kteam = 'How Kteam'
6
7 # nhớ là phải chạy hàm nữa
8 make_slogan()
9
10 print(kteam)
11 [
```

How Kteam
[Finished in 0.2s]

Ở đây Kteam muốn bạn lưu ý một trường hợp là tên biến local **trùng** với tên biến global.

```
def make_global():
    global x
    x = 1

def local():
    x = 5
    print('x in local', x)

make_global()
print(x)
local()
print(x)
```

Kết quả

```
1 def make_global():
2     global x
3     x = 1
4
5 def local():
6     x = 5
7     print('x in local', x)
8
9 make_global()
10 print(x)
11 local()
12 print(x)
13 |
```



```
1
x in local 5
1
[Finished in 0.3s]
```



Như bạn thấy ở ví dụ trên, biến **x** trong hàm **local** đã trùng với biến **global x**. Tuy nhiên hai biến x này là hoàn toàn khác nhau. Biến **x** dùng trong hàm **local** thì có một địa chỉ riêng và một giá trị riêng, còn biến **x global** thì cũng có một

giá trị riêng và một địa chỉ riêng. Thêm nữa, nếu như ta sử dụng biến x ngoài hàm thì Python sẽ tìm tới biến x global chứ không phải là biến x local.

Lưu ý:

BẠN KHÔNG NÊN SỬ DỤNG GLOBAL trừ khi hết cách. Nó giống như **hàm eval** vậy. Việc sử dụng biến global làm cho chương trình rối, khó kiểm soát cho nên hạn hãi chế tối đa việc sử dụng.

Giới thiệu hàm locals và globals

Cái tên hàm nói lên tất cả. **Hàm locals** cho ta biết được những biến local (những biến được khai báo trong hàm) nằm trong chương trình của chúng ta. Còn **globals** là hàm giúp chúng ta biết được những biến global trong chương trình.

Kết quả trả ra của hai hàm này là một Dict. Với **key** là tên biến và **value** là giá trị của biến.

Lưu ý:

Với hàm **globals()** thì với biến globals có giá trị mới được trả về.

```
>>> locals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'__frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>}
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'__frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>}
>>>
```

```
>>> global x
>>>
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {},
'__builtins__': <module 'builtins' (built-in)>}
>>> x = 'How Kteam'

>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__': <class
'frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__': {},
'__builtins__': <module 'builtins' (built-in)>, 'x': 'How Kteam'}
>>> globals()['x']
'How Kteam'
```

Kết luận

Qua bài viết này, Bạn đã biết thêm về hàm trong Python qua các khái niệm local và global trong hàm.

Ở bài viết sau. Kteam sẽ tiếp tục giới thiệu thêm với các bạn về [KIẾU DỮ LIỆU FUNCTION TRONG PYTHON - RETURN](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.

Bài 32: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON - RETURN

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Function trong Python - Return](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – BIẾN LOCALS & GLOBALS](#).

Và ở bài này Kteam sẽ lại tìm hiểu với các bạn **KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – Return.**

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- [CÂU ĐIỀU KIỆN IF TRONG PYTHON](#)
- [VÒNG LẶP WHILE](#) và [VÒNG LẶP FOR TRONG PYTHON](#)

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Vấn đề
 - Giới thiệu lệnh return
 - Dùng return để trả về nhiều giá trị một lúc
-

Vấn đề

Giả sử bạn viết một hàm xử lí một công việc và bạn muốn sao lưu kết quả sau khi xử lí xong ra một biến. Nhưng bạn lại không thể làm điều đó. Vì nếu tạo ra một biến và lưu ngay trong hàm thì như ta đã biết, nó không thể sử dụng được ở mức toàn chương trình (**global**).

Giá mà bạn có thể ném cái dữ liệu sau khi xử lí xong ra ngoài nhỉ?

Có đấy, hầu như mọi ngôn ngữ lập trình bây giờ đều cho phép làm điều đó và đương nhiên Python không phải ngoại lệ.

Giới thiệu lệnh return

Đây là lệnh chỉ sử dụng được ở trong hàm (nếu sử dụng ở ngoài hàm sẽ có nhắc lỗi)

```
SyntaxError: 'return' outside function
```

Lệnh return có cú pháp như sau

```
return [object]
```

Ở đây, **object** là một đối tượng bất kì của một lớp nào đó, có thể là số (number), chuỗi (string), list, tuple, hàm (sẽ biết rõ hơn khi tìm hiểu decorator), lớp (class) hoặc thậm chí là bỏ trống – trường hợp bỏ trống thì **object** return về được tính là **None**.

Khi **return** được gọi, hàm được kết thúc và kết quả được trả ra ngoài. Kết quả trả ra ngoài nên được đưa cho một biến nào đó hứng, nếu không thì coi như bạn gọi hàm không để làm gì.

```
def cal_rec_per(width, height):
    per = (width + height) * 2
    return per

rec_1_width = 5
rec_1_height = 3

# khởi tạo một biến để hứng kết quả
rec_1_per = cal_rec_per(rec_1_width, rec_1_height)

print(rec_1_per)

# trường hợp này là khi bạn không cần tái sử dụng nó ở lần sau
print(cal_rec_per(7, 4))
```

```
1 def cal_rec_per(width, height):
2     per = (width + height) * 2
3     return per
4
5 rec_1_width = 5
6 rec_1_height = 3
7
8 # khởi tạo một biến để hứng kết quả
9 rec_1_per = cal_rec_per(rec_1_width, rec_1_height)
10
11 print(rec_1_per)
12
13 # trường hợp này là khi bạn không cần tái sử dụng nó ở lần sau
14 print(cal_rec_per(7, 4))|
```

```
def _return_ter_func():
    print('chúng ta sử dụng return để ngắt hàm')
    # dòng dưới đây tương tự như bạn viết return None
    return
    print('Hàm print này dĩ nhiên không được gọi')

none = _return_ter_func()
print(type(none))
```

```

1 def _return_ter_func():
2     print('chúng ta sử dụng return để ngắt hàm')
3     # dòng dưới đây tương tự như bạn viết return None
4     return
5     print('Hàm print này dĩ nhiên không được gọi')
6
7 none = _return_ter_func()
8 print(type(none))
9
chúng ta sử dụng return để ngắt hàm
<class 'NoneType'>
[Finished in 0.4s]

```



Dùng return để trả về nhiều giá trị một lúc

Với Python, việc bạn có thể return nhiều giá trị một lúc bản chất nó không nằm ở câu lệnh Python, mà là do Python thiết kế đặc biệt để có thể **unpack** các **object** trả về. Bạn hãy xem ví dụ về khai báo sau đây

```

>>> one, two, three = 'how', 'Kteam', 69
>>> one
'how'
>>> two
'Kteam'
>>> three
69
>>> h, o, w = ('k', 'team', 96) # Ở đây, cũng có thể sử dụng list hoặc một container
bất kì
>>> h, o, w
('k', 'team', 96)

```

Tận dụng điều trên, ta có thể "**return nhiều giá trị cùng một lúc**"

```

def cal_rec_area_per(width, height):
    perimeter = (width + height) * 2
    area = width * height
    return perimeter, area

```

```
rec_width = 3
rec_height = 9
rec_per, rec_area = cal_rec_area_per(rec_width, rec_height)

print(rec_per, rec_area)
```

```
1 def cal_rec_area_per(width, height):
2     perimeter = (width + height) * 2
3     area = width * height
4     return perimeter, area
5
6 rec_width = 3
7 rec_height = 9
8 rec_per, rec_area = cal_rec_area_per(rec_width, rec_height)
9
10 print(rec_per, rec_area)
11
24 27
[Finished in 0.2s]
```



Câu hỏi củng cố

- Như các bạn đã biết khái niệm hàm số, với hàm số $y = f(x)$ thì đồ thị hàm số $y = f(x)$ đi qua điểm $M(x_0, y_0)$ nếu như $y_0 = f(x_0)$.

Cho một list, mỗi phần tử là một tuple gồm hoành độ (x_0) và tung độ (y_0), kiểm tra xem đồ thị hàm số $y = f(x)$ có đi qua điểm đó hay không. Nếu có thì đưa sang list A, trường hợp không thì đưa phần tử đó sang list B.

Sau khi kết thúc, tính tổng các tung độ (y_0) của hai list A và B rồi in ra trị tuyệt đối của hiệu tổng tung độ hai list đó.

Ví dụ: với hàm $y = f(x)$ như sau

$$y = x^3 + 2x^2 - 4x + 1$$



Và một List các điểm

```
[(-5, -20), (-4, -15), (-3, 4), (-2, 9), (-1, 7), (0, 1)  
, (1, -7), (2, -9), (4, 81), (5, 130)]
```



Thì kết quả in ra là **21**

2. Cho 5 biến với giá trị mỗi biến là một số tự nhiên, gọi **m** là giá trị lớn nhất trong 5 số đó. In ra màn hình $2m - 1$

Ví dụ: Với 5 biến như sau, kết quả in ra sẽ là 117

```
a = 32  
b = 59  
c = 8  
d = 24  
e = 15
```



Lưu ý: Không dùng các hàm tìm **min max** hỗ trợ bởi thư viện, chương trình có sẵn, không sử dụng bất kì container nào. Và chương trình không quá 3 câu lệnh điều kiện (if hoặc elif hoặc else)

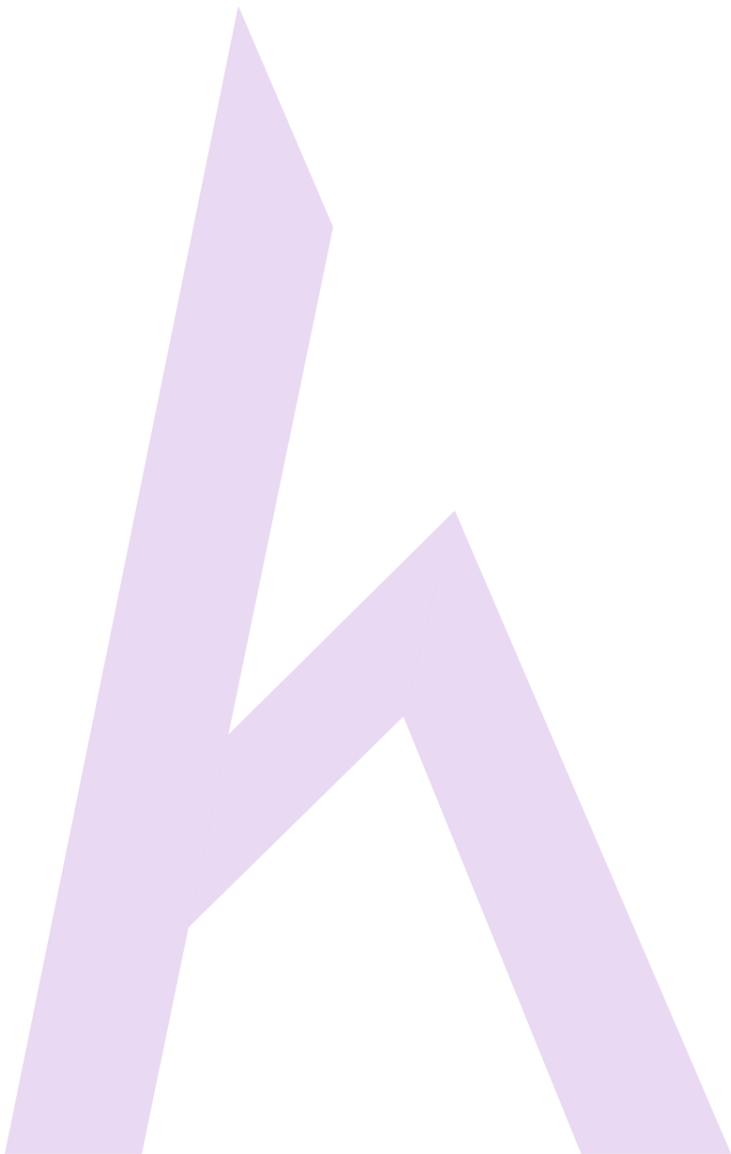
Đáp án của phần này sẽ được trình bày ở bài tiếp theo. Tuy nhiên, Kteam khuyến khích bạn tự trả lời các câu hỏi để củng cố kiến thức cũng như thực hành một cách tốt nhất!

Kết luận

Qua bài viết này, Bạn đã biết về lệnh return trong hàm.

Ở bài tiếp theo, Kteam sẽ nói đến một câu lệnh nữa có cách sử dụng rất giống return nhưng phức tạp rất nhiều - [KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – YIELD](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.



Bài 33: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – YIELD

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu function trong Python – Yield](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – RETURN](#).

Và ở bài này Kteam sẽ lại tìm hiểu với các **KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – YIELD**.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- [CÂU ĐIỀU KIỆN IF TRONG PYTHON](#)
- [VÒNG LẶP WHILE](#) và [VÒNG LẶP FOR](#) TRONG PYTHON

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Nhắc lại khái niệm iterables
- Giới thiệu generator
- Lệnh yield
- Phương thức send
- Vì sao nên dùng yield

Nhắc lại khái niệm iterables

Kteam đã từng giới thiệu với các bạn khái niệm này ở bài [ITERATION & MỘT SỐ HÀM HỖ TRỢ CHO ITERATION OBJECT TRONG PYTHON](#). Và ở bài này, chúng ta sẽ nhắc lại vài khái niệm trước khi đi đến lệnh yield

Khi bạn tạo ra một list, bạn có thể truy xuất lần lượt từng giá trị của list đó. Người ta gọi đó là **iteration**

```
>>> kteam_lst = [1, 'Kteam', 2]
>>> for value in kteam_lst:
...     print(value)
...
1
Kteam
2
```

“**kteam_lst**” ở đây được gọi là một **iterable**. Mọi thứ mà bạn có thể dùng cú pháp **“for ... in ...”** đều là một **iterable**. Ví dụ như chuỗi, list, tuple, file,..

Nhưng **iterable** này rất thuận tiện cho chúng ta lưu trữ và truy xuất thông tin. Và để được như vậy bạn phải lưu trữ những thông tin đó trong các vùng nhớ máy tính của bạn. Vì lẽ đó, sẽ có trường hợp bạn không cần thiết phải giữ tất cả thông tin cùng một lúc vì nó quá nhiều.

Giới thiệu generator

Generator là **iterator**, một dạng của **iterable** nhưng khác ở chỗ bạn không thể tái sử dụng. Vì sao lại như vậy? **Generator** không lưu trữ tất cả các giá trị của bạn ở bộ nhớ, mà nó sinh ra lần lượt

```
>>> kteam_gen = (value for value in range(3))
>>> for value in kteam_gen:
...     print(value)
...
0
1
2
```

Như đã nói, **generator** cũng là một **iterable**, nên nó cũng khá tương tự như khi bạn dùng **list** hoặc **tuple**. Nhưng, nếu bạn thử tái sử dụng generator đó

```
>>> for value in kteam_gen:
...     print(value)
...
>>>
```

Bạn thấy đấy, không có giá trị nào được in ra. Bởi vì khi nó sinh ra giá trị đầu tiên là 0, khi bạn kêu nó sinh tiếp giá trị 1, nó sẽ vứt bỏ giá trị 0 để nhường chỗ cho giá trị 1, và nếu bạn tiếp tục yêu cầu sinh thêm giá trị nó sẽ lại tiếp tục công việc như cũ cho tới khi kết thúc.

Lệnh yield

Các bạn chú ý: **y-i-e-l-d, yield**. Lệnh này khá là khó nhớ đặc biệt với những người chưa quen với tiếng Anh. Bạn cũng nên tra google để biết ý nghĩa của từ **yield**. Điều này sẽ giúp bạn biết rõ hơn lệnh này.

Lệnh này cách sử dụng gần giống với lệnh **return**, tuy nhiên nó khác **return** ở chỗ trả về một **object** thì **yield** sẽ trả về một **generator**.

Chúng ta hãy đến với một ví dụ với **return** sau đó ta sẽ so sánh nó với **yield**

```
>>> def square(lst):
...     sq_lst = []
...     for num in lst:
...         sq_lst.append(num**2)
...     return sq_lst

...
>>> kteam_ret = square([1, 2, 3])
>>> for value in kteam_ret:
...     print(value)

...
1
4
9
```

Và đây là khi sử dụng lệnh yield thay cho return

```
>>> def square(lst):
...     for num in lst:
...         yield num**2

...
>>> kteam_gen = square([1, 2, 3])
>>> for value in kteam_gen:
...     print(value)

...
1
4
9
```

Như bạn thấy, vì **return** sẽ quăng lại một list lưu trữ toàn bộ giá trị sau khi bình phương, thế nên bạn phải tạo một **list** để lưu hết những giá trị đó. Tuy nhiên, điều này là **không cần thiết** với **yield**. Nó sẽ lần lượt sinh ra từng giá trị bình phương một mà không cần một list để lưu trữ. Mỗi lần bạn gọi nó, nó sẽ chạy vào sinh ra cho bạn giá trị bạn cần như việc bạn sử dụng vòng lặp for để đọc từng giá trị trong một list.

Khi bạn dùng **yield** trong một hàm và khi gọi hàm đó, những dòng lệnh trong hàm sẽ không chạy ngay. Nó trả về một **generator**. Và mỗi khi bạn yêu cầu nó sinh thì nó mới bắt đầu chạy vào bên trong thực hiện những dòng lệnh trong hàm **CHO TỚI KHI GẶP LỆNH YIELD** và nó sẽ sinh ra giá trị bạn yêu cầu yield, hàm bây giờ được tạm dừng. Bạn cần lưu ý, là chỉ tạm dừng, có nghĩa là nếu lần sau gọi, hàm sẽ tiếp tục chạy ở phần đó không phải chạy lại từ đầu

Khi nào thì **yield** hết? Khi mà nó đi hết phần còn lại của hàm mà không gặp lệnh **yield**.

Bạn sẽ hiểu rõ hơn khi xem hai ví dụ sau đây.

```
>>> def gen():
...     for value in range(3):
...         print('yield', value + 1, 'times')
...         yield value
...
>>> for value in gen():
...     print(value)
...
yield 1 times
0
yield 2 times
1
yield 3 times
2
```

```
>>> def gen():
...     yield 'Kteam'
...     print('this is the second yield')
...     yield 'Free education'
...     print('this is the last yield')
...     yield 'Long đẽo trai'
...     print('Will not return anything')
...
>>> for value in test():
...     print(value)
Kteam
this is the second yield
Free education
this is the last yield
Long đẽo trai
Will not return anything
```

Bạn cũng cần lưu ý thêm, nếu không có giá trị **yield** khi được gọi tiếp thì sẽ **yield** sẽ không trả về bất cứ thứ gì, có nghĩa là **None object** cũng không được trả về.

Lưu ý: ngoài cách dùng **for** như bên trên để duyệt các generator, Kteam đã giới thiệu với các bạn hàm **next** ở bài [ITERATION & MỘT SỐ HÀM HỖ TRỢ CHO ITERATION OBJECT TRONG PYTHON](#) – một hàm để giúp bạn làm công việc tương tự.

Phương thức send

Lưu ý: Bạn đọc cần đọc và ngẫm thật kỹ **yield** ở phía trên trước khi đọc đến phần này.

Đây là phương thức giúp bạn gửi giá trị vào trong một generator.

Cú pháp:

generator.send(value)

Bạn cũng không cần phải lo lắng nếu không hiểu được đoạn code dưới đây

```
>>> def gen():
...     for i in range(4):
...         x = yield i
...         print('value sent from you', x)
...
>>> g = gen() # gán generator này cho biến g
>>> next(g) # gọi hàm next để chạy lệnh yield "x = yield i"
0
>>> g.send('Kteam') # x vừa nãy khi gán cho biến yield giờ sẽ được gửi giá trị
value sent from you Kteam
1
>>> g.send('Free education')
value sent from you Free education
2
```

```
>>> next(g) # lần này ta không dùng send, mặc định giá trị gửi vào là None  
value sent from you None  
3
```

Đây là một ví dụ khác nữa về phương thức **send**. Một lần nữa, hãy coi thật kĩ ví dụ **send** vừa trên trước khi đến với ví dụ tiếp sau đây

```
>>> def gen():  
...     while True:  
...         x = yield # ở đây ta đang yield None, vì ta không cần thiết sinh giá trị gì ở  
...         # đây  
...         yield x ** 2  
...  
>>> g = gen()  
>>> next(g) # chạy lệnh yield để ta gửi giá trị cho biến x lần sau  
>>> g.send(2)  
4  
>>> next(g) # tiếp tục chạy yield để có thể gửi giá trị  
>>> g.send(10)  
100
```

Vì sao nên dùng yield

Tốc độ, khi sử dụng **generator**, để duyệt các giá trị thì **generator** sẽ nhanh hơn khi bạn duyệt một **iterable** lưu trữ một lúc tất cả các giá trị

Bộ nhớ, bạn sẽ phải cân nhắc việc dùng **yield** khi bạn làm việc với những tập dữ liệu lớn. Lúc đó, bạn sẽ phải xem xét lại xem liệu bạn có cần giữ tất cả các giá trị một lúc không hay chỉ cần sinh ra từng giá trị một để tiết kiệm bộ nhớ.

Còn một số ưu điểm nữa của **yield**, bạn đọc có thể tham khảo câu trả lời sau trên Stack Overflow:

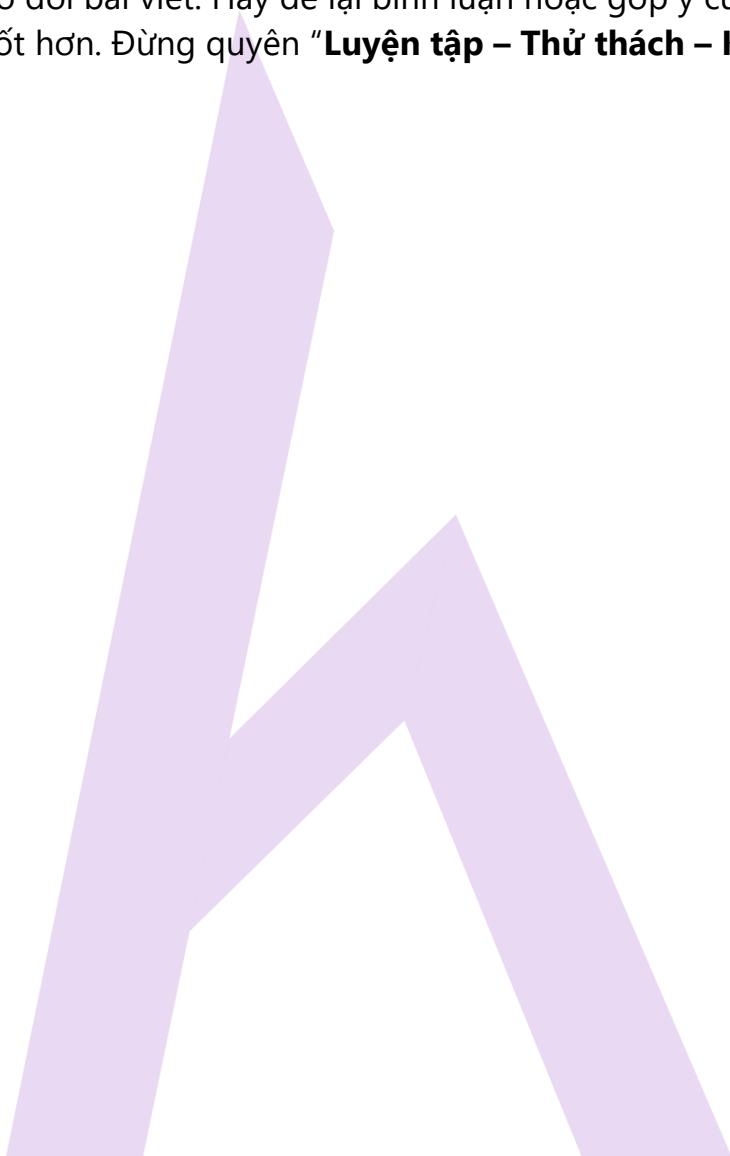
<https://stackoverflow.com/questions/102535/what-can-you-use-python-generator-functions-for>

Kết luận

Qua bài viết này, Bạn đã biết về lệnh yield trong hàm.

Ở bài tiếp theo, Kteam sẽ nói đến HÀM NẮC DANH TRONG PYTHON.

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quyên "**Luyện tập – Thủ thách – Không ngại khó**".



Bài 34: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – LAMBDA

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu function trong Python – Lambda](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [YIELD – KIỂU DỮ LIỆU FUNCTION TRONG PYTHON](#).

Và ở bài này Kteam sẽ lại tìm hiểu với **KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – LAMBDA**.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).
- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- [CÂU ĐIỀU KIỆN IF TRONG PYTHON](#)
- [VÒNG LẶP WHILE](#) và [VÒNG LẶP FOR](#) TRONG PYTHON

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Mở đầu
- Giới thiệu lambda
- Vì sao dùng lambda?
- Câu điều kiện cho lambda
- Lambda chồng lambda

Mở đầu

Ngoài từ khóa “**def**”, Python cũng hỗ trợ cho bạn một cách khác để có thể khai báo một function **object**, đó chính là **lambda**. Nó chỉ khác từ khóa “**def**” ở chỗ, thay vì **def** tạo một hàm với một cái tên xác định thì **lambda** trả về một hàm. Thế nên người ta hay gọi **lambda** là **hàm nặc danh (anonymous)**. Nó thường được sử dụng thường xuyên để có thể tạo ra một hàm chỉ với một dòng lệnh.

Giới thiệu lambda

Ta có cú pháp sau:

lambda argument_1, argument_2, ..., argument_n : expression

Như đã nói ở trên, **lambda** hoạt động như khi bạn dùng từ khóa “**def**” khai báo hàm. Tuy nhiên, vẫn có một vài ưu điểm nổi trội của **lambda** so với cách bình thường:

- **lambda** là một **expression**, không phải là một câu lệnh. (Khái niệm **expression** đã được Kteam giới thiệu). Do đó **lambda** có thể có ở một vài chỗ mà “**def**” không thể có (bạn đọc sẽ biết ở phần sau)
- **lambda** là một dòng **expression** duy nhất, không phải là một khối lệnh. Phần **expression** của **lambda** giống với phần khối lệnh của hàm với một lệnh **return** ở cuối hàm nhưng với **lambda** bạn chỉ cần ghi giá trị mà

không cần ghi return. Bạn đọc sẽ hiểu rõ hơn ở phần sau khi biết **lambda** có thể sử dụng các câu lệnh điều kiện mà không cần phải sử dụng tới lệnh "**if**". Nhờ được thiết kế như vậy, **lambda** được ưu tiên dùng cho việc tạo ra những hàm đơn giản, còn nếu phức tạp thì ta sẽ sử dụng đến từ khóa "**def**".

Để có thể hiểu hơn, mời bạn đọc xem qua các ví dụ sau đây

Đây là khi bạn sử dụng từ khóa "**def**"

```
>>> def ave(a, b, c):
...     return (a + b + c)/3
...
>>> ave(1, 3, 2)
2.0
```

Còn đây là khi sử dụng lambda

```
>>> ave = lambda a, b, c: (a + b+ c)/3
>>> ave(1, 3, 2)
2.0
```

Bạn còn nhớ **default argument** chứ?

```
>>> def x_power_a(x, a = 2):
...     return x ** a
...
>>> x_power_a(2)
4
>>> x_power_a(2, 3)
8
```

Điều đó cũng có thể làm được với lambda

```
>>> x_power_a = lambda x, a=2: x ** a
>>> x_power_a(2)
4
>>> x_power_a(2, 3)
8
```

Bạn cũng lưu ý thêm là lambda cũng phân biệt **global** và **local** nhé.

```
>>> def kteam():
...     mem = lambda x: x + ' is a member of Kteam'
...     return mem # trả về một hàm nặc danh
...
>>> call_mem = kteam() # lấy biến call_mem giữ hàm nặc danh
>>> call_mem('Long') # giá trị chuỗi được đưa vào cho biến x
'Long is a member of Kteam'
>>> call_mem('Giau')
'Giau is a member of Kteam'
>>> call_mem
<function kteam.<locals>.<lambda> at 0x03C2FDB0>
```

Vì sao dùng lambda?

Chung quy thì **lambda** là một công cụ nhanh gọn để bạn có thể tạo ra một hàm và sử dụng nó. Việc sử dụng nó thay cho "def" hay không là tùy ở bạn. Đương nhiên là bạn có thể chỉ sử dụng "**def**" thôi cũng được, hoàn toàn được, đặc biệt là những lúc mà hàm của bạn phức tạp, cần nhiều câu lệnh thì bạn không cần phải suy nghĩ nhiều nữa mà nên dùng "**def**" luôn. Nhưng giả sử bạn chỉ cần khởi tạo một hàm cấu trúc đơn giản và tái sử dụng nhiều lần thì sao? Lúc đó hãy nghĩ tới **lambda** nhé!

Chúng ta đến với một số ví dụ mà **lambda** hoàn toàn chiếm ưu thế so với "**def**".

```
>>> kteam_lst = [lambda x: x**2, lambda x: x**3, lambda x: x**4] # một list với các
phần tử là các hàm nặc danh
>>> kteam_lst[0]
<function <lambda> at 0x002CC618>
>>> kteam_lst[0](2) # 2**2
4
>>> kteam_lst[-1](4) # 4**4
256
>>> for func in kteam_lst:
...     func(3) # 3**2, 3**3, 3**4
...
9
27
81
```

Rất tiện lợi phải không nào, dĩ nhiên điều này “`def`” không thể có vì như đề cập ở phần trên, **lambda** là một **expression**, không phải một câu lệnh. Nên **lambda** có thể ở nhiều nơi mà “`def`” không thể.

Với ví dụ bên trên khi bạn muốn sử dụng “`def`”, bạn phải khởi tạo hàm ở ngoài rồi đưa vào **list**.

```
>>> def f1(x): return x**2
...
>>> def f2(x): return x**3
...
>>> def f3(x): return x**4
...
>>> kteam_lst = [f1, f2, f3]
>>> kteam_lst[0]
<function f1 at 0x00C8FE40>
>>> kteam_lst[-1](2) # 2**4
16
>>> for func in kteam_lst:
...     func(3)
...
9
27
81
```

Không chỉ mình **list**, bạn có thể sử dụng **lambda** với **dictionary**. Mời bạn đọc xem ví dụ sau đây:

```
>>> key = 'Kteam'
>>> {'Google': lambda: 'Gooooooooog',
... 'YouTube': lambda: 'Youuuuuuuuu',
... 'Kteam': lambda: 'Free Education'}[key]()
'Free Education'
```

Lưu ý: Bạn để ý ví dụ trên, phần **argument** của **lambda** ta để trống, điều này hoàn toàn đúng cú pháp vì phần **argument** là **optional** (không bắt buộc) nhưng phần **expression** bắt buộc phải có một **expression**.

Ta thử lại ví dụ trên nhưng không dùng **lambda** mà dùng “`def`”

```
>>> def f1(): return 'Gooooooooog'  
...  
>>> def f2(): return 'Youuuuuuuuu'  
...  
>>> def f3(): return 'Free Education'  
...  
>>> key = 'Kteam'  
>>> {'Google': f1, 'YouTube': f2, 'Kteam': f3}[key]()  
'Free Education'
```

Nó cũng như **lambda** thôi, nhưng rõ ràng ta thấy **lambda** tiện lợi hơn "**def**" dù cho chỉ vài dòng code. Điểm mạnh vượt trội của **lambda** so với "**def**" hoàn toàn được thể hiện với những hàm tính toán đơn giản nhanh chóng. Hơn thế nữa, khi dùng "**def**", bạn phải tạo ra một cái tên cho nó, và đôi khi việc bạn nghĩ ra một cái tên cho một cái hàm thực sự không hề đơn giản (việc này khá hiếm nhưng đã xảy ra).

Bạn sẽ còn thấy lambda còn tiện lợi hơn rất nhiều khi bạn tìm hiểu hàm **map** (Kteam sẽ giới thiệu ở bài khác).

Câu điều kiện cho lambda

Rõ ràng bạn đã thấy, **lambda** chỉ nhận một **expression**, do đó, bạn không thể chèn câu lệnh điều kiện như bình thường được mà phải theo một cách khác.

Giả sử với lệnh **if** như sau

```
if a:  
    b  
else:  
    c
```

Thì có thể viết dưới dạng **expression** với 2 cách như sau

Cách 1:

```
b if a else c
```

Cách 2:

```
(a and b) or c
```

Bạn có cần nhớ cả 2 cái không? Không cần thiết đâu, Kteam khuyến khích bạn đọc ghi nhớ và dùng cách 1 vì sự rõ ràng và dĩ nhiên cũng không nên bối rối khi thấy cách 2.

Hãy đến với ví dụ để hiểu thêm

```
>>> find_greater = lambda x, y: x if x > y else y
>>> find_greater(1, 3)
3
>>> find_greater(6, 2)
6
```

Ví dụ sau đây là kiểm tra xem số đó cùng có hai ước 2 và 3 hay không? Nếu có thì trả về 1, không thì là 0. Ví dụ này hoàn toàn có thể sử dụng **lambda** bằng cách sử dụng “**and**” nhưng ở đây Kteam muốn bạn biết chúng ta có thể lồng các **expression** lên nhau.

```
>>> cd_of_2_3 = lambda x: (1 if x % 3 == 0 else 0) if x % 2 == 0 else 0
>>> cd_of_2_3(6)
1
>>> cd_of_2_3(8)
0
>>> cd_of_2_3(9)
0
>>> cd_of_2_3(12)
1
```

Ở ví dụ trên, phần **if** bạn có thể thu gọn biểu thức đi một tẹo bằng cách dùng phủ định

```
>>> cd_of_2_3 = lambda x: (1 if not (x % 3) else 0) if not (x % 2) else 0
>>> cd_of_2_3(6)
1
>>> cd_of_2_3(9)
0
>>> cd_of_2_3(8)
0
```

```
>>> cd_of_2_3(12)
1
```

Lambda chồng lambda

Phần này sẽ hơi rắc rối nếu như bạn chưa thực sự hiểu **lambda**. Bạn có thể chồng 2 hoặc 3 **lambda** lên nhau cùng một lúc. Nhưng phải chú ý để biết được mình đang làm gì nhé.

```
>>> def kteam(first_string):
...     return lambda second_string: first_string + second_string # trả về một hàm, và
lưu biến first_string
...
>>> slogan = kteam('How Kteam ') # gửi giá trị cho biến first_string
>>> slogan
<function kteam.<locals>.<lambda> at 0x00CA4150>
>>> slogan('Free Education') # gửi nốt giá trị còn lại cho second_string
'How Kteam Free Education'
```

Ví dụ trên ta sử dụng “**def**”, và bạn để ý hàm sử dụng “**def**” trên ta hoàn toàn có thể sử dụng **lambda** thay thế.

```
>>> kteam = lambda first_string: (lambda second_string: first_string +
second_string)
>>> slogan = kteam('How Kteam ')
>>> slogan('Free Education')
'How Kteam Free Education'
>>> (lambda first_string: (lambda second_string: first_string +
second_string))('How Kteam ')('Free Education')
'How Kteam Free Education'
```

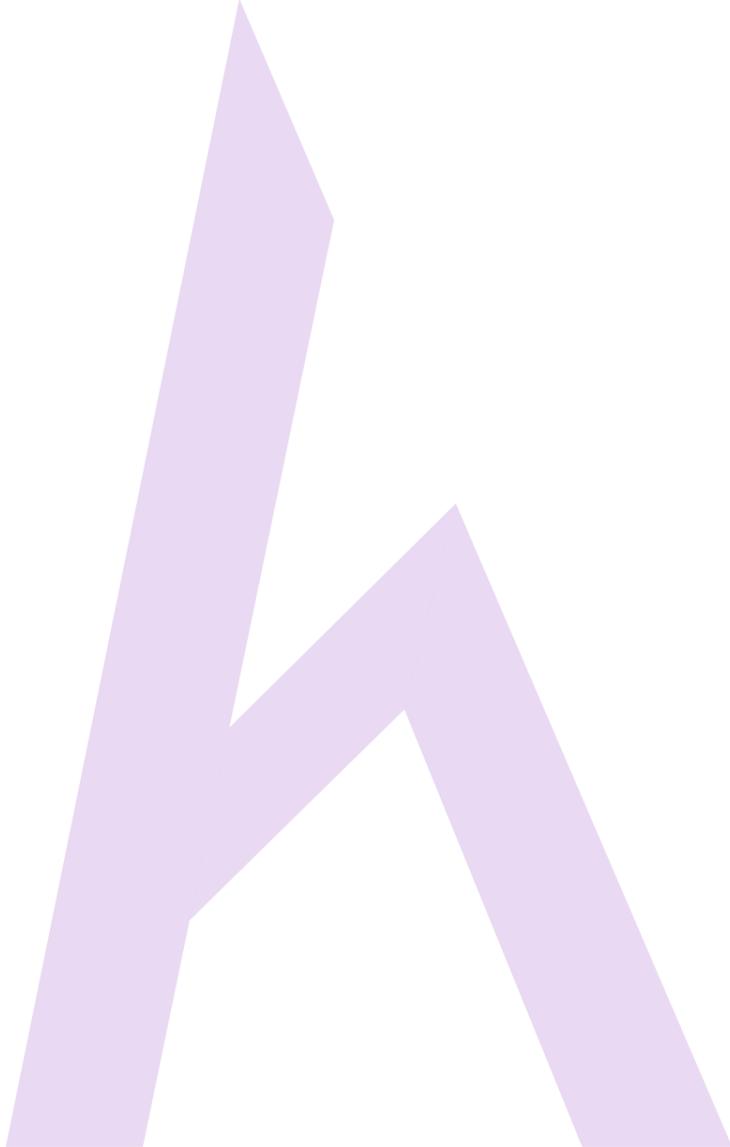
Thực tế thì những **lambda** chồng **lambda** này khá phức tạp. Python vốn không thích sự khó hiểu, phức tạp, sự thiếu thanh lịch thế nên thường thì việc chồng **lambda** như thế này rất **không được khuyến khích**.

Kết luận

Qua bài viết này, Bạn đã biết về hàm nặc danh lambda.

Ở bài tiếp theo, Kteam sẽ nói đến [MỘT SỐ HÀM HAY SỬ DUNG KẾT HỢP VỚI HÀM NẶC DANH TRONG PYTHON](#)

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quyên “**Luyện tập – Thủ thách – Không ngại khó**”.



Bài 35: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON - FUNCTIONAL TOOLS

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Function trong Python - Functional tools](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [KIỂU DỮ LIỆU FUNCTION – LAMBDA](#) trong Python.

Và ở bài này Kteam sẽ lại tìm hiểu với các **KIỂU DỮ LIỆU FUNCTION – FUNCTIONAL TOOLS** trong Python.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).

- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- [CÂU ĐIỀU KIÊN IF TRONG PYTHON](#)

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Hàm map
- Hàm filter
- Hàm reduce

Hàm map

Chúng ta thường hay phải xử lí các phần tử của một **list** hoặc một [container](#) nào đó bằng một phương thức.

Giả sử ta phải cập nhật một **list** bằng cách tăng mỗi giá trị trong **list** đó lên 1 đơn vị

```
>>> kteam = [1, 2, 3, 4]
>>> kteam_updated = []
>>> for value in kteam:
...     kteam_updated.append(value + 1)
...
>>> kteam_updated
[2, 3, 4, 5]
```

Chúng ta sẽ ngó sơ qua cú pháp của hàm **map** trước khi xem hàm map xử lí công việc bên trên. Đầu tiên là cái cơ bản nhất

map(func, iterable)

Hàm map này sẽ trả về một **map object** (một dạng [generator](#)).

Vậy hàm **map** hoạt động như thế nào? Nôm na là hàm map lấy từng phần tử của [iterable](#) sau đó dùng gọi hàm **func** với **argument** là giá trị mới lấy ra từ [iterable](#), kết quả trả về của hàm **func** sẽ được **yield**.

Nôm na hàm **map** nó sẽ trông như thế này:

```
>>> def mymap(func, iterable):
...     for x in iterable:
...         yield func(x)
```

Chúng ta đến với ví dụ để hiểu hơn

```
>>> def inc(x): return x + 1
...
>>> kteam = [1, 2, 3, 4]
>>> list(map(inc, kteam)) # dùng constructor list để ta dễ quan sát dữ liệu
[2, 3, 4, 5]
```

Bạn còn nhớ **lambda** chứ? (Nếu không có thể tham khảo bài [HÀM NẮC DANH LAMBDA](#))

```
>>> kteam = [1, 2, 3, 4]
>>> list(map(lambda x: x + 1, kteam))
[2, 3, 4, 5]
```

Đôi lúc, việc sử dụng hàm map còn nhanh hơn cả **list comprehension**

```
>>> inc = lambda x: x + 1
>>> kteam = [1, 2, 3, 4]
>>>
>>> [inc(x) for x in kteam]
[2, 3, 4, 5]
>>> list(map(inc, kteam))
[2, 3, 4, 5]
```

Lưu ý: Ở **list comprehension** trên, nếu bạn thay ngoặc vuông ([) bằng ngoặc tròn (() thì thời gian của ngoặc tròn có thể tương đương với hàm map và **tiết kiệm dữ liệu** hơn list comprehension vì nó cũng tạo ra một generator expression.

Ta mở rộng hàm map ra nhé. Vì đầy đủ cú pháp hàm map là

map(func, *iterable)

Bạn đọc lưu ý, khi bạn pass vào nhiều **container** để biến hàm map gom lại bằng cách **packing argument** thì các **container** phải cùng số lượng giá trị (cùng giá trị hàm len). Vì khi có nhiều container pass vào, thì hàm map sẽ cùng một lúc lấy lượt các giá trị của các **container**.

Bạn đọc sẽ hiểu kĩ khi xem ví dụ sau đây

```
>>> func = lambda x, y: x + y
>>>
>>> kteam_1 = [1, 2, 3, 4]
>>> kteam_2 = [5, 6, 7, 8]
>>>
>>> kteam = map(func, kteam_1, kteam_2)
>>> list(kteam)
[6, 8, 10, 12]
```

Như bạn thấy, hàm map sẽ lấy từng giá trị một của cả hai list rồi gửi nó vào hàm. À, bạn cũng phải lưu ý là bạn pass vào **n container** thì bạn cũng phải thiết kế cái hàm nào có thể nhận **n argument** luôn nhé.

```
>>> pow(2, 3) # 2^3
8
>>> pow(3, 4) # 3^4
81
>>> list(map(pow, [1, 2, 3], [2, 2, 2])) # 1^2, 2^2, 3^2
[1, 4, 9]
```

Hàm filter

Filter có nghĩa là bộ phận lọc. Nghe qua, chắc bạn cũng ít nhiều biết được nó sẽ làm gì rồi.

Cú pháp hàm này như sau:

filter(function or None, iterable)

Cũng như hàm map, hàm **filter** sẽ trả về một **filter object** (một dạng generator object)

Lưu ý: không như hàm map, iterable ở đây chỉ là 1 container, không hề có **packing argument**.

Hàm **filter** lấy từng giá trị trong **iterable**, sau đó gửi vào hàm, nếu như giá trị hàm trả ra là một giá trị mà khi chuyển sang kiểu dữ liệu **boolean** là **True** thì sẽ **yield** giá trị đó, nếu không thì bỏ qua.

Trường hợp bạn không gửi hàm vào mà là **None**, hàm **filter** lấy từng giá trị trong **iterable**, nếu giá trị đó chuyển sang giá trị **boolean** là **True** thì **yield**, nếu không thì bỏ qua.

Chúng ta đến với ví dụ để hiểu thêm. Đầu tiên sẽ làm có function bằng một ví dụ lọc lấy các số dương (lớn hơn 0)

```
>>> func = lambda x: x > 0
>>>
>>> kteam = [1, -3, 5, 0, 2, 6, -4, -9]
>>> list(filter(func, kteam))
[1, 5, 2, 6]
```

Hàm **func** nhận vào 1 giá trị, nếu giá trị đó lớn hơn 0 thì trả về **True**, còn không thì là **False** nhờ toán tử so sánh. Vậy nên, các giá trị gửi vào mà nhận giá trị **False** là không được **yield**.

Nếu bạn nào còn mông lung thì bạn hãy xem qua **list comprehension** tương đương với hàm filter trên

```
>>> kteam = [1, -3, 5, 0, 2, 6, -4, -9]
>>> [x for x in kteam if x > 0]
[1, 5, 2, 6]
```

Tiếp đến là một ví dụ khác khi ta gửi **None** thay vì một function

```
>>> kteam = [0, None, 1, 'Kteam', "", 'Free Education', 69, False]
>>> list(filter(None, kteam))
[1, 'Kteam', 'Free Education', 69]
```

Hàm reduce

Bất cứ giá trị nào khi chuyển qua giá trị **boolean** mà **False** thì sẽ không được **yield**. Đơn giản phải không nào? :D

Lưu ý: Với những bạn dùng Python 2.X, hàm reduce là hàm có sẵn, bạn chỉ việc dùng, còn với Python 3.X, nó đã được đưa vào trong thư viện `functools`. Vì thế nếu bạn muốn sử dụng nó, đừng quên import nhé.

```
from functools import reduce
```

Ở các ví dụ tiếp theo, thì sẽ không có dòng này vì lặp lại nên bạn đọc coi như chúng ta đã có dòng lệnh này ở đầu chương trình tức có nghĩa là chúng ta đã **import** hàm `reduce` từ thư viện `functools` rồi.

Hàm này khá phức tạp này, các bạn không cần nóng vội để hiểu nó. Ta hãy đến với cú pháp của nó.

```
reduce(function, sequence[, initial])
```

Lưu ý: Hàm reduce không giống như hai hàm trước là trả về một **generator expression** mà là một giá trị.

Để đơn giản nhất, chúng ta hãy tạm chưa xét tới **parameter initial**.

Đầu tiên, hàm reduce sẽ lần lượt lấy hai giá trị đầu tiên của **sequence (index 0, index 1)** và đưa vào hàm function

Lưu ý: đưa theo thứ tự (index 0, index 1)

Hàm function này sẽ trả ra một giá trị (ta kí hiệu là A). Sau đó lấy tiếp giá trị thứ ba của sequence (index 2), rồi gửi vào function cũng theo thứ tự (A, index 2), rồi lại lặp lại như thế cho tới khi hết sequence.

Hãy đến với các ví dụ để hiểu hơn

Ví dụ dùng **reduce** để tính tổng các số trong list

```
>>> kteam_add = lambda x, y: x + y
>>> kteam = [1, 2, 3, 4, 5]
>>> reduce(kteam_add, kteam) # (((1+2)+3)+4)+5
15
```

Ví dụ dùng **reduce** để tính tích các số trong list

```
>>> kteam_multi = lambda x, y: x * y
>>> kteam = [1, 2, 3, 4]
>>> reduce(kteam_multi, kteam) # (((1*2)*3)*4)
24
```

Nào, giờ chúng ta tới bước khi có **argument** cho **parameter initial**. Khi nãy, khi chưa có **initial**, hàm **reduce** lấy hai giá trị để quăng vào function đầu tiên. Nhưng khi bạn đưa argument vào cho parameter initial thì hàm **reduce** sẽ lấy giá trị **initial** và giá trị đầu tiên của **sequence (index 0)** đưa vào function và tiếp tục trả ra một giá trị, rồi giá trị đó lại tiếp tục với giá trị thứ hai của **sequence (index 1)**.

Ví dụ để hiểu thêm

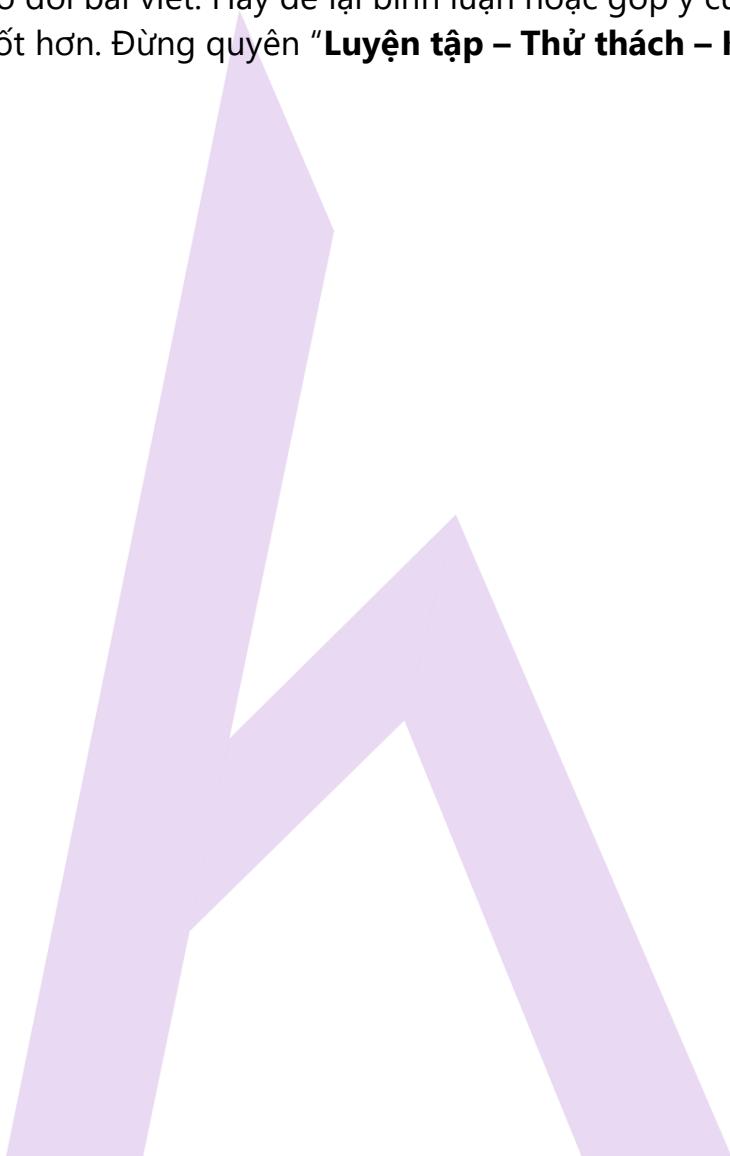
```
>>> kteam = [1, 2, 3, 4]
>>> kteam_add = lambda x, y: x + y
>>> kteam_multi = lambda x, y: x * y
>>>
>>> reduce(kteam_add, kteam, 10)
20
>>> reduce(kteam_multi, kteam, 10)
240
```

Kết luận

Qua bài viết này, Bạn đã biết về hàm nặc danh lambda.

Ở bài tiếp theo, Kteam sẽ nói đến [KỸ THUẬT ĐÊ QUY](#).

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.



Bài 36: KIỂU DỮ LIỆU FUNCTION TRONG PYTHON - ĐỆ QUY (RECURSION)

Xem bài học trên website để ủng hộ Kteam: [Kiểu dữ liệu Function trong Python - Đê quy \(recursion\)](#)

Mọi vấn đề về lỗi website làm ảnh hưởng đến bạn hoặc thắc mắc, mong muốn khóa học mới, nhằm hỗ trợ cải thiện Website. Các bạn vui lòng phản hồi đến Fanpage [How Kteam](#) nhé!

Dẫn nhập

Trong bài trước, Kteam đã giới thiệu đến bạn [KIỂU DỮ LIỆU FUNCTION TRONG PYTHON – FUNCTIONAL TOOLS](#).

Và ở bài này Kteam sẽ lại tìm hiểu với các **KIỂU DỮ LIỆU FUNCTION – ĐỆ QUY** trong Python.

Nội dung

Để đọc hiểu bài này tốt nhất bạn cần:

- Cài đặt sẵn [MÔI TRƯỜNG PHÁT TRIỂN CỦA PYTHON](#).
- Xem qua bài [CÁCH CHẠY CHƯƠNG TRÌNH PYTHON](#).

- Nắm [CÁCH GHI CHÚ](#) và [BIẾN TRONG PYTHON](#).
- CÁC KIỂU DỮ LIỆU ĐƯỢC GIỚI THIỆU TRONG PYTHON
- [CÂU ĐIỀU KIÊN IF TRONG PYTHON](#)

Bạn và Kteam sẽ cùng tìm hiểu những nội dung sau đây

- Giới thiệu về đệ quy
- Minh họa đệ quy bằng cách tính tổng
- Đệ quy theo phong cách Python
- Đệ quy và vòng lặp

Giới thiệu về đệ quy

Đệ quy là một mảng kiến thức nâng cao, ở Python thì nó không thường xuyên được dùng đến, do cách xử lí của Python có thể sử dụng những cấu trúc vòng lặp đơn giản mà không cần dùng tới đệ quy. Nhưng dù sao thì đây cũng là một kĩ thuật khá hữu dụng mà bạn đọc nên biết. Nó cũng chỉ đơn giản là việc chính nó gọi nó.

Minh họa đệ quy bằng cách tính tổng

Ta sẽ tính tổng các phần tử của một **list** (hoặc một sequence nào đó) bằng cách dùng **đệ quy** (Ví dụ này chỉ là minh họa, thực tế khi làm bạn nên sử dụng hàm **sum**)

```
>>> def cal_sum(lst):
...     if not lst: # tương đương if len(lst) == 0:
...         return 0
...     else:
...         return lst[0] + cal_sum(lst[1:])
...
>>> cal_sum([1, 2, 3, 4])
```

10

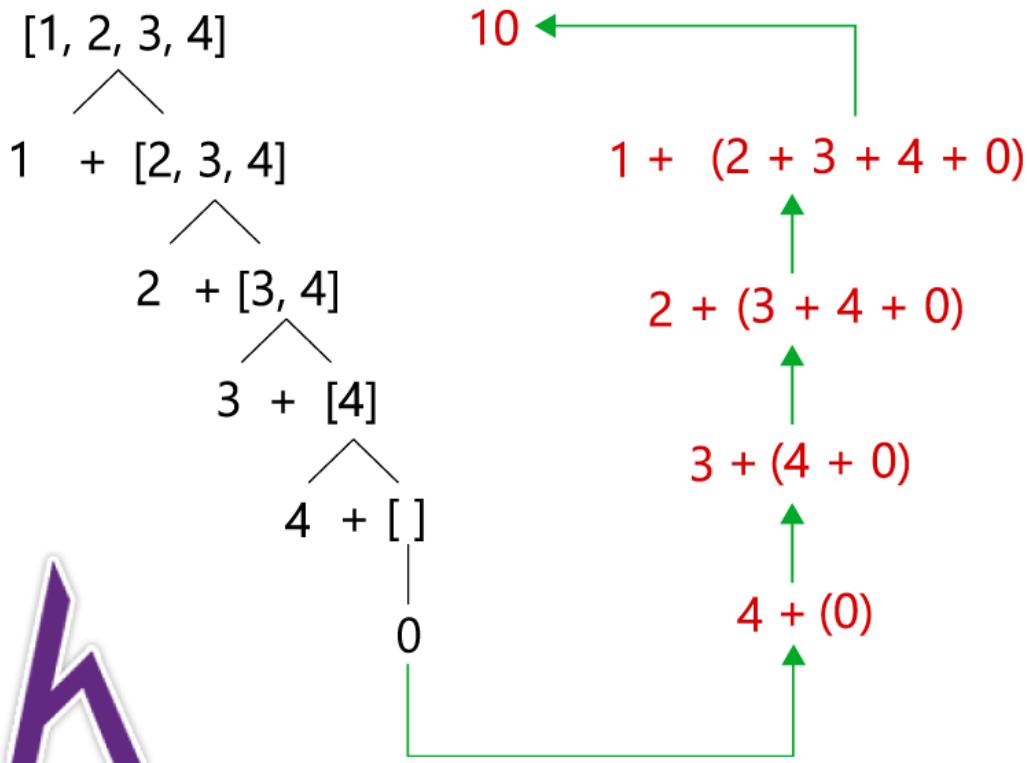
```
>>> cal_sum([1, 2, 3, 4, 5])  
15
```

Ở ví dụ trên, ta liên tục gọi lại hàm `cal_sum` với **argument** là phần còn lại của **List** tính từ **index 1**. Ở mỗi lần gọi hàm, ta để lại giá trị **index 0** ở **List** để khi trong **List** không còn phần tử nào ta sẽ trả về số 0 để kết thúc đệ quy.

Nếu bạn thấy vẫn còn chưa hiểu rõ thì cũng đừng lo lắng, ai cũng đều cảm thấy khó hiểu và điều này thường xuyên xảy ra với những bạn mới học. Những lúc như thế này, bạn nên để thêm một cái hàm `print` để xem cụ thể là chuyện gì xảy ra

```
>>> def cal_sum(lst):  
...     print(lst)  
...     if not lst:  
...         return 0  
...     else:  
...         return lst[0] + cal_sum(lst[1:])  
...  
>>> cal_sum([1, 2, 3, 4])  
[1, 2, 3, 4]  
[2, 3, 4]  
[3, 4]  
[4]  
[]  
10
```

Nếu bạn vẫn còn khó hiểu hãy vẽ ra giấy. Đâyq là cách mà mình đã làm



Đệ quy theo phong cách Python

Bạn còn nhớ cú pháp `if/else` trong [lambda](#) không? Nó còn có tên khác là **ternary expression**. Bạn có thể áp dụng để sử dụng nó để đệ quy:

```
>>> def cal_sum(lst):
...     return 0 if not lst else lst[0] + cal_sum(lst[1:])
...
>>> cal_sum([1, 2, 3])
6
```

Giả sử một list có **n** phần tử thì với đệ quy như trên cần phải có **n + 1** lần **return**, ta có thể giảm bớt xuống còn **n** lần **return** bằng cách:

```
>>> def cal_sum(lst):
...     return lst[0] if len(lst) == 1 else lst[0] + cal_sum(lst[1:])
...
>>> cal_sum([1, 2, 3])
6
```

Lưu ý: cách này **không** sử dụng được trong trường hợp **container rỗng**

Hoặc ta có thể sử dụng **packing argument**:

```
>>> def cal_sum(lst):
...     idx0, *r = lst # idx0, r = lst[0], lst[1:]
...     return idx0 if not r else idx0 + cal_sum(r)
...
>>> cal_sum([1, 2, 3])
6
```

Lưu ý: cách này cũng **không** sử dụng được khi **container là rỗng**. Tuy nhiên điểm lợi của nó cũng như cách vừa nãy là ta có thể cộng không chỉ số mà là chuỗi, hoặc list

```
>>> cal_sum(['a', 'b', 'c'])
'abc'
>>> cal_sum([[1, 2], [3, 4], [5, 6]])
[1, 2, 3, 4, 5, 6]
```

Đệ quy cũng có thể chuyển hướng. Hãy xem ví dụ sau. Một hàm gọi một hàm khác, sau đó lại gọi lại hàm đã gọi nó.

```
>>> def cal_sum(lst):
...     if not lst: return 0
...     return call_cal_sum(lst)
...
>>> def call_cal_sum(lst):
...     return lst[0] + cal_sum(lst[1:])
...
>>> cal_sum([1, 2, 3, 4, 5])
15
```

Đệ quy và vòng lặp

Ở những ví dụ trước, nếu phải chọn vòng lặp hay là đệ quy để xử lí thì Kteam khuyên bạn đọc nên chọn vòng lặp. Python chú trọng việc làm đơn giản hóa mọi việc như là vòng lặp vì nó theo một cách bình thường đơn giản. Vòng lặp cũng không yêu cầu bạn phải tạo ra một hàm mới có thể sử dụng được. Và thêm nữa, đệ quy còn thua vòng lặp ở mặt hiệu quả về bộ nhớ và thời gian thực hiện.

Kết luận

Qua bài viết này, bạn đã biết thêm về đệ quy. Ngoài ra, bạn cũng đã nắm được kha khá các kiến thức cơ bản của Python thông qua khóa [LẬP TRÌNH PYTHON CƠ BẢN](#). Hy vọng khóa học sẽ là nền tảng tốt để bạn có thể tiếp tục tự nghiên cứu hoặc tiến tới các khóa học Python khác.

Cảm ơn bạn đã theo dõi bài viết. Hãy để lại bình luận hoặc góp ý của mình để phát triển bài viết tốt hơn. Đừng quên “**Luyện tập – Thủ thách – Không ngại khó**”.