

1.1. Giới thiệu tổng quan về Python

Ngôn ngữ lập trình Python đã xuất hiện từ rất lâu, được phát triển bởi lập trình viên Guido van Rossum vào năm 1989. Sau đó Guido tiếp tục tham gia và phát triển Python cho tới ngày nay. Trong 5 năm gần đây, Python trở thành một trong các ngôn ngữ lập trình phổ biến nhất thế giới do tính dễ sử dụng và cộng đồng hỗ trợ lớn mạnh.

Hiện nay, Python có hai dòng phiên bản là Python 2.7.x và Python 3.x. Người dùng có thể cài đặt nhiều phiên bản Python trên máy cùng một lúc, tuy nhiên đối với mỗi câu lệnh đường dẫn thì chỉ có một phiên bản được sử dụng, ví dụ như:

- Lệnh `python2` để chạy phiên bản cài đặt mới nhất của Python 2.7.x trên máy tính.
- Lệnh `python3` để chạy phiên bản cài đặt mới nhất của Python 3.x trên máy tính.

Phiên bản mới sẽ được kế thừa các câu lệnh và đối tượng của phiên bản cũ và đồng thời cập nhật những câu lệnh mới và đối tượng mới. Vì vậy, nếu người dùng cài đặt Python phiên bản 2.7.15 thì sẽ chạy các tập kịch bản được viết bởi phiên bản 2.7.15 hoặc các phiên bản trước đó, nếu người dùng cài đặt Python phiên bản 2.7.1, thì có thể sẽ không chạy được các tập kịch bản được viết trong phiên bản 2.7.15.

Thông thường, các mã kịch bản của Python sẽ được lưu với đuôi mở rộng là `.py`, các tập tin đuôi `.py` là các mã nguồn Python dưới dạng không nén (uncompressed), người dùng có thể dễ dàng đọc và chỉnh sửa mã nguồn của các tập tin này với các trình soạn thảo. Python còn có đuôi mở rộng như `.pyc`, các tập Python byte code, đây là kết quả của quá trình: trình thông dịch biên dịch tập lệnh Python thành một tập tin trung gian, chúng được tối ưu hóa cho trình thông dịch đọc và thực thi. Python thường tự động tạo các tập tin đuôi mở rộng `.pyc` khi tập tin cùng tên đuôi `.py` được import, người dùng có thể thủ công tạo các tập tin `.pyc` với cú pháp như sau

```
1. import py_compile  
2. py_compile.compile("script")
```

Ngoài ra, các tập tin kịch bản Python cũng có thể được chuyển thành dạng thực thi ở các hệ điều hành khác nhau nhờ sự hỗ trợ của một số công cụ như: py2exe và pyinstaller cho hệ điều hành Windows, Freeze cho hệ điều hành Linux, py2app cho hệ điều hành MacOS.

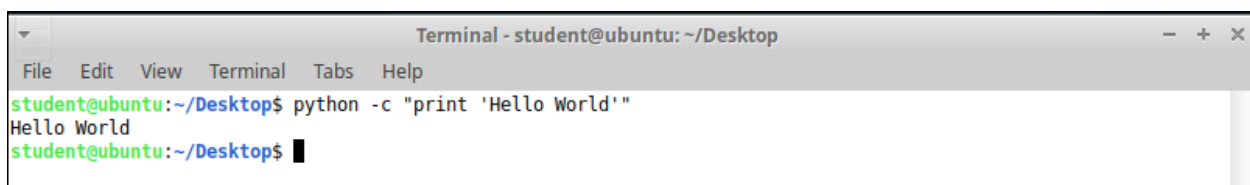
Python là ngôn ngữ thông dịch nên có đôi chút khác biệt với các ngôn ngữ biên dịch như C. Ngôn ngữ biên dịch sử dụng trình biên dịch. Trình biên dịch sẽ lấy mã nguồn, đọc mã nguồn và cung cấp một tập tin thực thi nhị phân phù hợp với hệ điều hành. Chương trình được sinh ra có thể thực thi với hệ điều hành mà không cần sự trợ giúp thêm từ trình biên dịch và mã nguồn. Đối với ngôn ngữ thông dịch, trình thông dịch sẽ đọc và xử lý đoạn kịch bản trong khi thực thi đoạn kịch bản đó, nghĩa là, trình thông dịch phải được cài đặt trên máy mà người dùng chạy chương trình. Vì vậy muốn chạy được các mã kịch bản của Python, cần phải có trình thông dịch của Python. Trình thông dịch của Python thực thi bắt đầu từ dòng lệnh đầu tiên bắt gặp, tiếp đến là dòng lệnh thứ hai, và tiếp tục, tiến trình sẽ được thực thi theo kiểu Top-Down - từ trên xuống dưới. Vì vậy ta có thêm lưu ý trước khi lập trình đó là người dùng cần định nghĩa, khai báo các hàm trước khi gọi và sử dụng lại các hàm đó.

Tương tự với các ngôn ngữ lập trình khác, để khởi động với Python, ta có một chương trình đầu tiên, đó là helloworld.py. Tại trình soạn thảo, sử dụng câu lệnh print và nội dung “Hello world”, ta được tập tin với nội dung như sau:

```
1. # Hello World
2. print "Hello World"
```

Để thực thi chương trình helloworld.py, ta có ba cách:

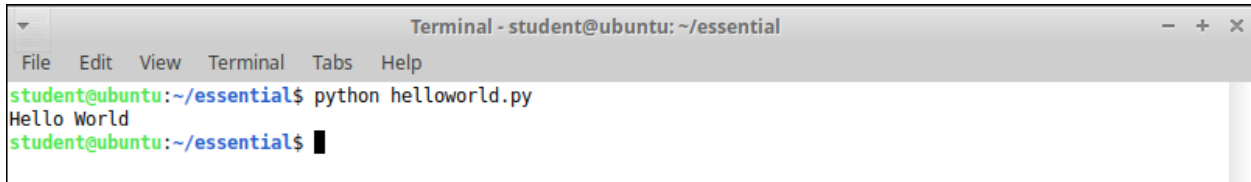
Cách 1, chạy giao diện dòng lệnh và truyền trực tiếp mã nguồn vào trình thông dịch thông qua nó như hình 3.1.

A screenshot of a terminal window titled "Terminal - student@ubuntu: ~/Desktop". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The terminal shows a prompt "student@ubuntu:~/Desktop\$" followed by the command "python -c 'print 'Hello World''". The output "Hello World" is displayed on the next line. The prompt is then followed by a cursor "█".

```
Terminal - student@ubuntu: ~/Desktop
File Edit View Terminal Tabs Help
student@ubuntu:~/Desktop$ python -c "print 'Hello World'"
Hello World
student@ubuntu:~/Desktop$ █
```

Hình 3.1. Thực thi chương trình với giao diện dòng lệnh

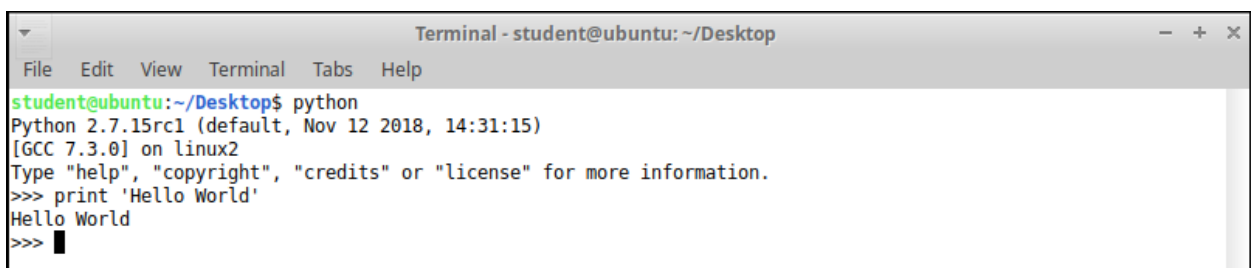
Cách 2, truyền trực tiếp tập tin helloworld.py vào trình thông dịch, hình 3.2.



```
Terminal - student@ubuntu: ~/essential
File Edit View Terminal Tabs Help
student@ubuntu:~/essential$ python helloworld.py
Hello World
student@ubuntu:~/essential$
```

Hình 3.2. Thực thi chương trình bằng trình thông dịch

Cách 3, sử dụng Python Shell và thực thi nội dung của chương trình, hình 3.3.



```
Terminal - student@ubuntu: ~/Desktop
File Edit View Terminal Tabs Help
student@ubuntu:~/Desktop$ python
Python 2.7.15rc1 (default, Nov 12 2018, 14:31:15)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello World'
Hello World
>>>
```

Hình 3.3. Thực thi nội dung chương trình với Python shell

```
1. #!/usr/bin/python
2. # User can comment a single line with a pound sign
3. """
4. The first string in the program is the DocString and is used by
5. help function to describe the program
6. """
7. import sys
8. def main():
9.     'A "Docstring" for the main function here'
10.    print 'You passed the argument: ' + sys.argv[1]
11.
12. if __name__ == '__main__':
13.    main()
```

Trên đây là đoạn mã mô tả cấu trúc cơ bản của một chương trình python, các thành phần chính như sau. Dòng đầu tiên đánh dấu vị trí của trình thông dịch Python trong hệ điều hành Linux. Dòng thứ hai là ghi chú mã nguồn, ta có thể sử dụng dấu thăng #, để bắt đầu bất kỳ dòng ghi chú nào. Dòng tiếp theo sau dòng ghi chú được gọi là Docstring, tùy chọn có hoặc không. Để định nghĩa Docstring, ta sử dụng ba dấu nháy đơn. Docstring sẽ được hiển thị khi một người dùng sử dụng lệnh help() tới đoạn mã nguồn. Tiếp theo, import thư viện bất kỳ mà người dùng cảm thấy cần thiết cho chương trình. Sau khi import, người dùng cần định nghĩa hàm main(),

mặc dù không nhất thiết phải có, vì python sẽ tự động thực thi bất kỳ các dòng lệnh python được mô tả trong tập tin kịch bản. Tuy nhiên, với thực hành tốt nhất, ta nên cho đoạn mã nguồn chính vào hàm main(), với cách này khi người dùng muốn chuyển chương trình hiện tại của bản thân thành một module nhỏ hơn cho chương trình khác, thì không cần phải thay đổi quá nhiều trong mã nguồn. Cuối cùng, ta có câu lệnh điều kiện if, so sánh biến `__name__` với `"__main__"`. Dòng này để kiểm tra trường hợp đoạn kịch bản này có được thực thi như một hàm main của chương trình hay không? Hay nó được import bởi chương trình khác? Nếu đoạn kịch bản được import bởi một chương trình khác thì hàm main() trong đoạn code sẽ không được gọi. Và ngược lại, nếu đây là chương trình chính, hàm main() sẽ được gọi ngay lập tức khi trình thông dịch bắt đầu thực thi.

Khi trình thông dịch Python phân tích một dòng code để xác định những hành động cần thực hiện, trình thông dịch sẽ tìm các từ khóa, toán tử, dấu phân cách, ghi chú và biến. Các ghi chú bắt đầu bằng dấu thăng và kết thúc bằng cách xuống một dòng mới. Các ghi chú sẽ được trình thông dịch bỏ qua. Python phiên bản 2.7 có 31 từ khóa (hình 3.4), người dùng có thể liệt kê các từ khóa thông qua Python Shell.

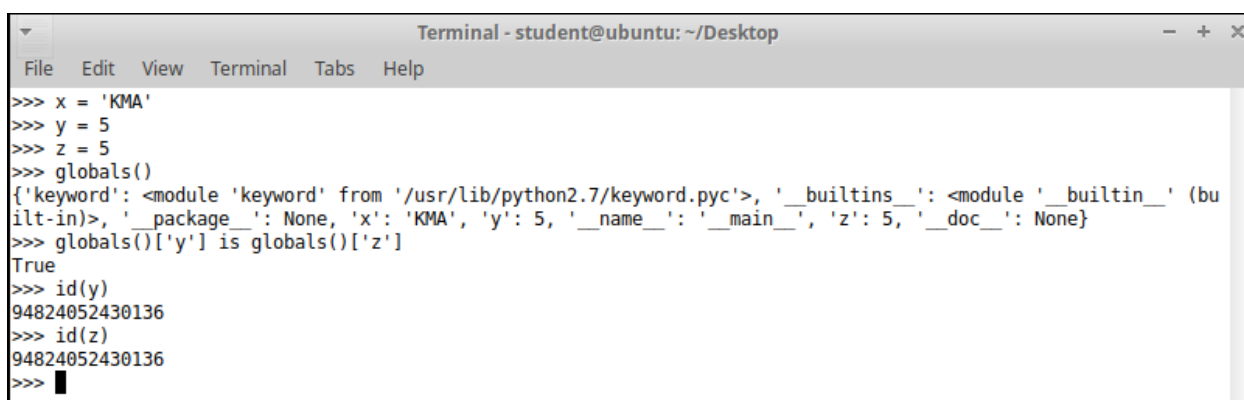
```
>>> import keyword
>>> print keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'exec', 'finally',
 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return',
 'try', 'while', 'with', 'yield']
>>> █
```

Hình 3.4. Từ khóa trong Python 2.7

Python phiên bản 3 có 33 từ khóa, 'print' và 'exec' được loại bỏ, thêm vào đó là 'False', 'None', 'True' và 'nonlocal'. Các toán tử được sử dụng trong công việc tính toán, bao gồm các toán tử toán học thông thường như dấu cộng và dấu trừ và các toán tử nhị phân như dấu mũ đại diện cho phép XOR, dấu sỏ đứng đại diện cho phép OR, dấu và đại diện cho phép AND. Các dấu phân cách được sử dụng để phân tách các tham số, xác định cấu trúc dữ liệu và xác định thứ tự các toán tử. Khi trình thông dịch tìm thấy một đối tượng không khớp với các đối tượng trên, trình thông dịch sẽ coi đây là một biến, với điều kiện tên biến được đặt đúng với tiêu chuẩn đặt tên các biến của Python, tên biến phải bắt đầu bằng dấu gạch chân hoặc một ký tự chữ từ a tới z và không phân biệt chữ hoa chữ thường.

1.2. Biến

Ta có thể hiểu đơn giản, một biến là một nhãn văn bản được tạo cho một địa chỉ bộ nhớ, nơi lưu trữ một đối tượng Python. Ví dụ, khi Python thực thi dòng lệnh “x = 5”, Python sẽ tạo một nhãn tên trong không gian tên hiện tại để chứa biến “x” và trả nó vào vị trí bộ nhớ chứa đối tượng số nguyên có giá trị 5. Mỗi khi chương trình tham chiếu tới biến “x”, chương trình sẽ biết rằng, người dùng đang muốn tham chiếu tới địa chỉ bộ nhớ có giá trị 5. Trong vai trò một lập trình viên, ta sử dụng biến để lưu trữ dữ liệu và thực thi các biến này. Tương tự như vậy, khi ta tiếp tục gán “x = 6”, Python sẽ tiến hành thay đổi địa chỉ mà x trỏ tới đến bộ nhớ mới, nơi chứa số nguyên 6.



```
Terminal - student@ubuntu: ~/Desktop
File Edit View Terminal Tabs Help
>>> x = 'KMA'
>>> y = 5
>>> z = 5
>>> globals()
{'keyword': <module 'keyword' from '/usr/lib/python2.7/keyword.pyc'>, '__builtins__': <module '__builtin__' (built-in)>, '__package__': None, 'x': 'KMA', 'y': 5, '__name__': '__main__', 'z': 5, '__doc__': None}
>>> globals()['y'] is globals()['z']
True
>>> id(y)
94824052430136
>>> id(z)
94824052430136
>>>
```

Hình 3.5. Ví dụ về biến và bộ nhớ lưu trữ biến

Các tên biến được lưu trữ trong miền không gian tên của Python, vì cả hai biến “y” và “z” trong ví dụ trên đều có giá trị nguyên 5, nên chúng cũng trỏ vào chung một vùng nhớ. Để kiểm tra điều này, ta có thể sử dụng từ khóa “is” để so sánh hai vùng nhớ mà hai biến trên trỏ tới. Ta có thể sử dụng thêm hàm id() với tham số đầu vào là các tên biến để lấy giá trị địa chỉ vùng nhớ các biến trỏ tới.

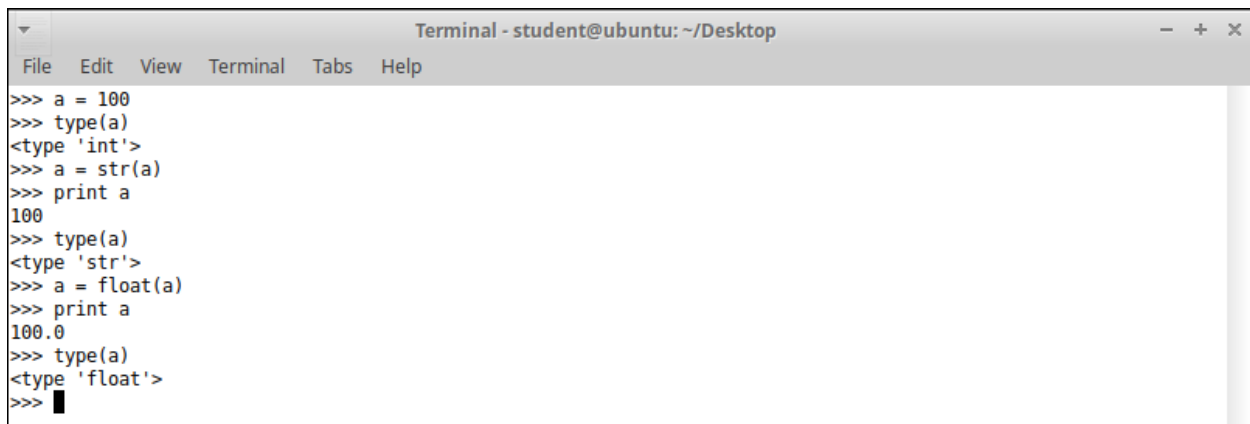
Trong Python, ta có các kiểu dữ liệu phổ biến như sau:

- Integer int(): Kiểu số nguyên.
- Float float(): Kiểu số thực.
- String str(): Kiểu ký tự.
- List list(): Kiểu danh sách.
- Tuple tuple(): Kiểu tuple.

- Dictionary dict(): Kiểu từ điển.

Để kiểm tra giá trị của một biến đang là kiểu dữ liệu nào, ta có thể sử dụng hàm type().

Để gán giá trị cho một biến, ta sử dụng toán tử gán, ví dụ như dấu bằng. Đoạn mã phía bên phải của toán tử gán sẽ được thực thi tính toán từ trái qua phải, sau đó kết quả của việc tính toán sẽ được lưu trữ vào biến nằm bên phía trái của toán tử gán. Không giống như C, với Python, người dùng không cần khai báo kiểu dữ liệu cho biến trước khi gán giá trị vào biến, Python sẽ tự động nhận dạng kiểu dữ liệu cho các biến khi được gán giá trị. Ta có thể gán lại loại dữ liệu cho giá trị của biến bằng cách sử dụng các hàm định dạng dữ liệu với tham số đầu vào là biến cần thay đổi kiểu dữ liệu.



```
Terminal - student@ubuntu: ~/Desktop
File Edit View Terminal Tabs Help
>>> a = 100
>>> type(a)
<type 'int'>
>>> a = str(a)
>>> print a
100
>>> type(a)
<type 'str'>
>>> a = float(a)
>>> print a
100.0
>>> type(a)
<type 'float'>
>>>
```

Hình 3.6. Gán biến và gán kiểu dữ liệu cho biến

1.3. Toán tử

Python hỗ trợ rất nhiều toán tử toán học. Người dùng có thể sử dụng dấu bằng để gán giá trị, dấu cộng để cộng các giá trị, dấu trừ để tiến hành trừ, dấu hoa thị để thực hiện phép nhân,... Bảng dưới đây liệt kê một số toán tử thường được sử dụng trong Python.

Bảng 3.1. Một số toán tử cơ bản

Toán tử	Ví dụ	Kết quả
Phép cộng	$X = 1 + 2$	3
Phép trừ	$X = 5 - 10$	-5

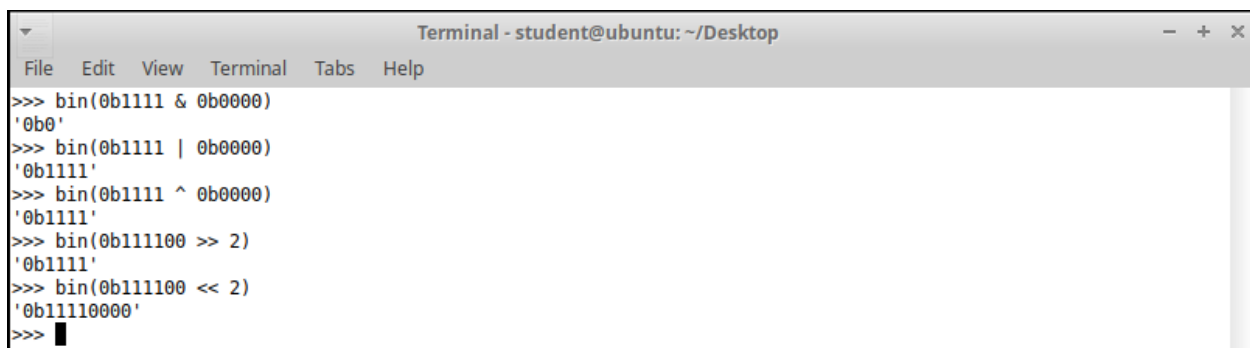
Phép nhân	$X = 2 * 3$	6
Phép chia	$X = 5 / 2$	2
Modulo	$X = 5 \% 2$	1
Phép mũ	$X = 5 ** 2$	25

Ta có thể viết gọn một số toán tử thành phép tính đơn giản hơn như bảng 3.2 sau:

Bảng 3.2. Các toán tử rút gọn

Phép toán thông thường	Rút gọn
$A = A + 1$	$A += 1$
$A = A - 1$	$A -= 1$
$A = A * 2$	$A *= 2$
$A = A / 5$	$A /= 5$
$A = A \% 6$	$A \% = 6$

Ngoài các phép toán thông thường với hệ thập phân, ta có thể sử dụng các phép toán Logic với hệ nhị phân hoặc lục phân, ví dụ như: AND, OR, XOR hay dịch bit... Hình 3.7 thể hiện ví dụ về các toán tử với hệ nhị phân.



```

Terminal - student@ubuntu: ~/Desktop
File Edit View Terminal Tabs Help
>>> bin(0b1111 & 0b0000)
'0b0'
>>> bin(0b1111 | 0b0000)
'0b1111'
>>> bin(0b1111 ^ 0b0000)
'0b1111'
>>> bin(0b111100 >> 2)
'0b1111'
>>> bin(0b111100 << 2)
'0b11110000'
>>>

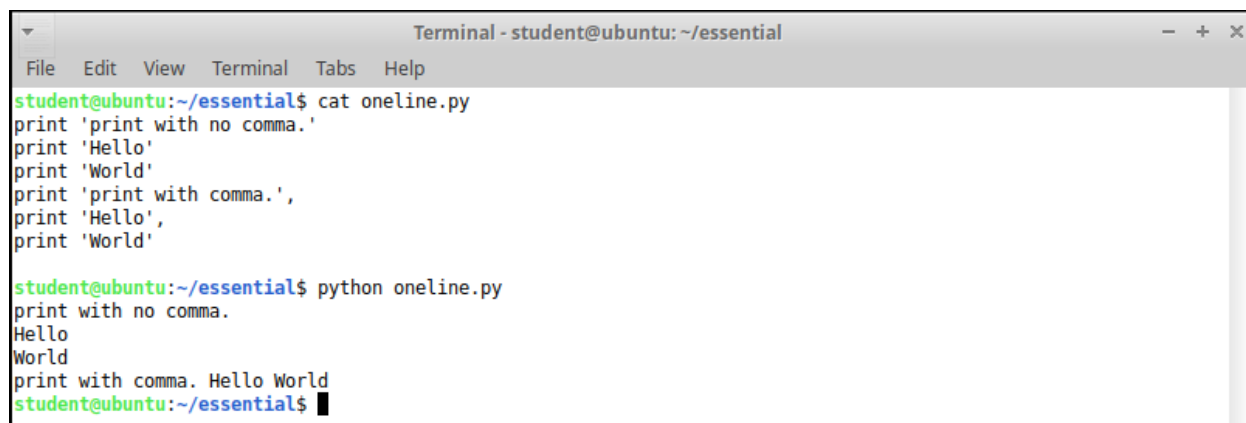
```

Hình 3.7. Các toán tử với hệ nhị phân

1.4. Câu lệnh print

Một trong số các chức năng cơ bản nhất mà người dùng có thể tương tác với Python là tiến hành xuất thông tin ra màn hình máy tính. Với Python, công việc này có thể được thực hiện bằng hai cách đó là sử dụng câu lệnh print hoặc sử dụng hàm print. Python với các phiên bản từ 2.7 trở về sau có thể sử dụng hàm print, Python

các phiên bản từ 2.7 tới các phiên bản cũ hơn có thể sử dụng câu lệnh print. Đối với đồ án này, ta sử dụng Python phiên bản 2.7 để lập trình, vì vậy ta có thể sử dụng cả hai cách trên để xuất thông tin ra màn hình máy tính. Thông thường trong lập trình, lập trình viên thường xuyên sử dụng câu lệnh print hơn hàm print, vì vậy ta sẽ ưu tiên phân tích cách tiếp cận được sử dụng nhiều hơn. Ví dụ về câu lệnh này được thể hiện trong hình 3.8.



```
Terminal - student@ubuntu: ~/essential
File Edit View Terminal Tabs Help
student@ubuntu:~/essential$ cat oneline.py
print 'print with no comma.'
print 'Hello'
print 'World'
print 'print with comma.',
print 'Hello',
print 'World'

student@ubuntu:~/essential$ python oneline.py
print with no comma.
Hello
World
print with comma. Hello World
student@ubuntu:~/essential$
```

Hình 3.8. Ví dụ về câu lệnh print

Câu lệnh print sẽ tự động nối thêm một dòng mới \n vào cuối đối số cuối cùng. Điều này có nghĩa là mọi thứ người dùng in ra sẽ xuất hiện trên mỗi dòng riêng biệt. Người dùng có thể thay đổi hành vi này bằng cách thêm dấu phẩy vào cuối câu lệnh print. Sự bổ sung này sẽ khiến câu lệnh print thêm một khoảng trắng vào cuối đối số cuối cùng của câu lệnh.

Đối với Python, ta có thể sử dụng chức năng định dạng chuỗi với câu lệnh print. Định dạng chuỗi được sử dụng để tối ưu về mặt nội dung cũng như thẩm mỹ, nó có thể được sử dụng trong câu lệnh print hoặc hành động gán giá trị cho các biến. Dưới đây là bảng và ảnh ví dụ về định dạng chuỗi thường dùng trong Python.

Bảng 3.3. Các định dạng chuỗi thường gặp

Giá trị	Định dạng
%d	Số nguyên thập phân
%s	Chuỗi ký tự
%x	Thập lục phân (Viết thường)

%f	Số thực dấu phẩy động
%%	In ra ký tự %%

```

Terminal - student@ubuntu: ~/Desktop
File Edit View Terminal Tabs Help
>>> print 'I would like %d %s' %(5, 'cats')
I would like 5 cats
>>> print '%%'
%%
>>> █

```

Hình 3.9. Ví dụ về định dạng chuỗi

1.5. Strings và Functions

1.5.1. Cơ sở lý thuyết

a) Kiểu dữ liệu Strings

Chuỗi là một nhóm gồm một hoặc nhiều ký tự. Chuỗi được tạo bằng cách đặt các ký tự vào trong dấu nháy kép. Bất kỳ chuỗi ký tự nào được đặt trong cùng một dấu nháy kép đều là một chuỗi, thậm chí tất cả các giá trị số kèm theo trong dấu nháy kép cũng là chuỗi. Một chuỗi có thể được đặt trong dấu nháy đơn (') hoặc dấu nháy kép ("). Vì các chuỗi cũng là các đối tượng, nên chúng có một số phương thức được xây dựng sẵn mà người dùng có thể sử dụng để thao tác.

Sử dụng hàm dir() với một biến có kiểu dữ liệu là chuỗi ký tự, ta thấy được các phương thức và thuộc tính có sẵn của biến, hình 3.10.

```

Terminal - student@ubuntu: ~/Desktop
File Edit View Terminal Tabs Help
>>> a = 'Give me all methods and attributes of String object'
>>> type(a)
<type 'str'>
>>> dir(a)
['_add_', '_class_', '_contains_', '_delattr_', '_doc_', '_eq_', '_format_', '_ge_', '_getattr_', '_getitem_', '_getnewargs_', '_getslice_', '_gt_', '_hash_', '_init_', '_le_', '_len_', '_lt_', '_mod_', '_mul_', '_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_', '_rmod_', '_rmul_', '_setattr_', '_sizeof_', '_str_', '_subclasshook_', '_formatter_field_name_split_', '_formatter_parser_', '_capitalize_', '_center_', '_count_', '_decode_', '_encode_', '_endswith_', '_expandtabs_', '_find_', '_format_', '_index_', '_isalnum_', '_isalpha_', '_isdigit_', '_islower_', '_isspace_', '_istitle_', '_isupper_', '_join_', '_ljust_', '_lower_', '_lstrip_', '_partition_', '_replace_', '_rfind_', '_rindex_', '_rjust_', '_rpartition_', '_rsplit_', '_rstrip_', '_split_', '_splitlines_', '_startswith_', '_strip_', '_swapcase_', '_title_', '_translate_', '_upper_', '_zfill_']
>>> █

```

Hình 3.10. Các phương thức dựng sẵn của Strings

Bảng 3.4. dưới đây chứa một số ví dụ về các phương thức thường dùng đối với chuỗi ký tự. Giả sử ta có biến x được gán giá trị 'pyGames KMA!'.

Bảng 3.4. Ví dụ về các phương thức của Strings

Chức năng	Phương thức	Kết quả
Viết hoa	x.upper()	PYGAMES KMA!
Viết thường	x.lower()	pygames kma!
Thay thế chuỗi con	x.replace('KMA','hvmm')	pyGames hvmm!
Tìm chuỗi con trong x	'KMA' in x	True
Tìm chuỗi con trong x	'ACT' in x	False
Chuyển đổi Strings qua kiểu dữ liệu Lists	x.split()	['pyGames','KMA!']
Đếm số chuỗi con	x.count('p')	1
Tính độ dài chuỗi	len(x)	12

Ngoài tất cả các phương thức được sử dụng để thao tác chuỗi, người dùng cũng có thể 'cắt' chuỗi dựa trên chỉ mục của các ký tự. Để cắt chuỗi, ta sử dụng cú pháp có cấu trúc như sau: `string[start:end:step]` , trong đó:

- String là chuỗi ký tự cần cắt chuỗi, ký tự đầu tiên của chuỗi sẽ nhận giá trị chỉ mục 0 và giá trị này sẽ tăng dần từ trái qua phải tương ứng với các giá trị của chuỗi cho đến ký tự cuối cùng. Ký tự cuối cùng của chuỗi ứng với giá trị -1, giá trị này sẽ giảm dần từ phải qua trái, tương ứng với các ký tự của chuỗi cho đến ký tự đầu tiên.

- Mỗi chỉ mục như start, end và step là tùy chọn và được phân cách với nhau bởi dấu hai chấm.

- Start là chỉ mục bắt đầu, kiểu dữ liệu số nguyên, start sẽ phải nhỏ hơn end trong trường hợp step dương, nếu không kết quả trả về sẽ là một chuỗi rỗng.

- End là chỉ mục kết thúc, kiểu dữ liệu số nguyên.

- Step là số bước trượt, kiểu dữ liệu số nguyên, nếu step có giá trị là số âm thì trượt lùi, lúc này giá trị start phải lớn hơn giá trị end.

Tham khảo một ví dụ về phép cắt chuỗi “I Love KMA” trong bảng 3.5.

Bảng 3.5. Ví dụ về các phép cắt chuỗi

X = “	I		L	o	v	e		K	M	A	”
	0	1	2	3	4	5	6	7	8	9	
	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	
Phép cắt chuỗi							Giá trị nhận được				
X[0]							I				
X[2]							L				
X[0:3] hoặc X[:3]							I L				
X[0:-1] hoặc X[:-1]							I Love KMA				
X[0::2] hoặc X [::2]							Ilv M				
X[::-1]							AMK evoL I				
X[:6][::-1]							evoL I				
X[5::-1]							evoL I				

Chuỗi ký tự cũng có các phương thức như encode() và decode() để mã hóa và giải mã dữ liệu, một số thuật toán mã hóa và giải mã thường thấy là: base64, bz2, rot13, utf-16, zip, hex và string_escape, hình 3.11 minh họa về hai hàm nói trên.

```
>>> 'aGVsbG8gd29ybGQ=' .decode('base64')
'hello world'
>>> 'hello world'.encode('base64')
'aGVsbG8gd29ybGQ=\n'
>>> █
```

Hình 3.11. Ví dụ về hàm encode() và decode()

Sau khi sử dụng một phương thức với chuỗi ký tự, ta được chuỗi ký tự mới có cùng bộ phương thức với chuỗi ký tự cũ. Vì vậy, người dùng có thể sử dụng nhiều phương thức khác nhau liên tiếp cho một chuỗi ký tự để ra một kết quả như ý muốn.

b) Tạo và sử dụng hàm

Hàm (Function), là một khối mã lệnh được tổ chức và có thể tái sử dụng một hoặc nhiều lần trong chương trình, để thực hiện một hành động nào đó. Trong các

phần trước, ta đã biết tới một số hàm được xây dựng sẵn trong Python, điển hình như hàm `print()`. Tuy nhiên người dùng cũng có thể tạo riêng cho mình các hàm, chúng được gọi là các hàm tự định nghĩa. Sử dụng hàm giúp cho chương trình tường minh, dễ hiểu và dễ đọc hơn.

Một hàm sẽ chấp nhận các tham số trong ngoặc đơn và sẽ trả về các giá trị thông qua lệnh `return`. Đồng thời, hàm không chia sẻ các biến của mình với phần còn lại của chương trình. Các biến trong hàm được gọi là các biến cục bộ. Vì vậy, các biến cục bộ chỉ tồn tại trong khi hàm đang thực thi. Chương trình gọi hàm không thể truy cập các biến cục bộ. Thay vào đó, hàm dựa vào lệnh `return` để gửi dữ liệu trở lại chương trình gọi nó.

Khi định nghĩa các hàm, người dùng cần lưu ý một số quy tắc sau:

- Từ khóa `def` được sử dụng để bắt đầu phân định nghĩa hàm.
- Sau từ khóa `def` là tên hàm, trong ví dụ phía dưới, ta có tên hàm là `function_name`, sau tên hàm là dấu ngoặc đơn.
- Nếu có tham số được truyền vào hàm, ta đưa các tham số vào bên trong các dấu ngoặc đơn. Các tham số được truyền vào trong ví dụ dưới đây là `argument1`, `argument2`.
- Sau ngoặc đơn là dấu hai chấm.
- Trình thông dịch Python đánh dấu sự khởi đầu của một khối mã lệnh bằng dấu hai chấm. Tất cả các dòng sau khi thụt lề ở cùng cấp đều là một phần của một khối mã lệnh (Block of code).
- Trong các khối mã lệnh có thể tồn tại các khối mã lệnh con, các khối con cũng có thể được xác định trong một khối mã bằng cách thụt sâu vào các dòng đó một hoặc nhiều mức nữa. Các khối mã lệnh con thường liên quan đến các câu lệnh điều kiện của một vòng lặp.
- Khối mã kết thúc khi thụt lề quay trở lại độ sâu thụt lề trước đó, đây là độ sâu thụt lề được sử dụng trước khi khối mã bắt đầu.
- Lệnh đầu tiên của hàm là tùy ý, thông thường là các Docstring.
- Sau đó là các lệnh để thực thi.
- Sử dụng lệnh `return` để trả về giá trị cho chương trình gọi hàm.

Để thực thi hàm, ta cần gọi lại hàm đó. Đối với ví dụ, ta sử dụng cú pháp như sau: `function_name(argument1,argument2)`.

```
1. def function_name(argument1,argument2):  
2.     'A Docstring for the help()' # Code Block 1  
3.     # Code beneath it is executed when function is called  
4.     # Arguments can be given default values by assigning them a value  
5.     # Code block 2  
6.     # Control statement of a loop, such as if/else or a for loop  
7.     # The code can return one or more values  
8.     return 'Return this string'
```

1.5.2. Bài tập thực hành

Để trau dồi kỹ năng của người học trong việc thao tác các chuỗi ký tự, pyGames cung cấp một danh sách các thử thách liên quan.

Chuẩn bị:

- Người học cần có kiến thức cơ bản về chuỗi ký tự và hàm.
- Người học cần có sẵn một phiên làm việc với pyGames, pygameserver và một biến chứa đối tượng pyGames.game().

Danh sách các bài thực hành: Bài 2, bài 3, bài 4, bài 5, bài 6, bài 7, bài 8, bài 9, bài 10 và bài 11.

1.6. Modules

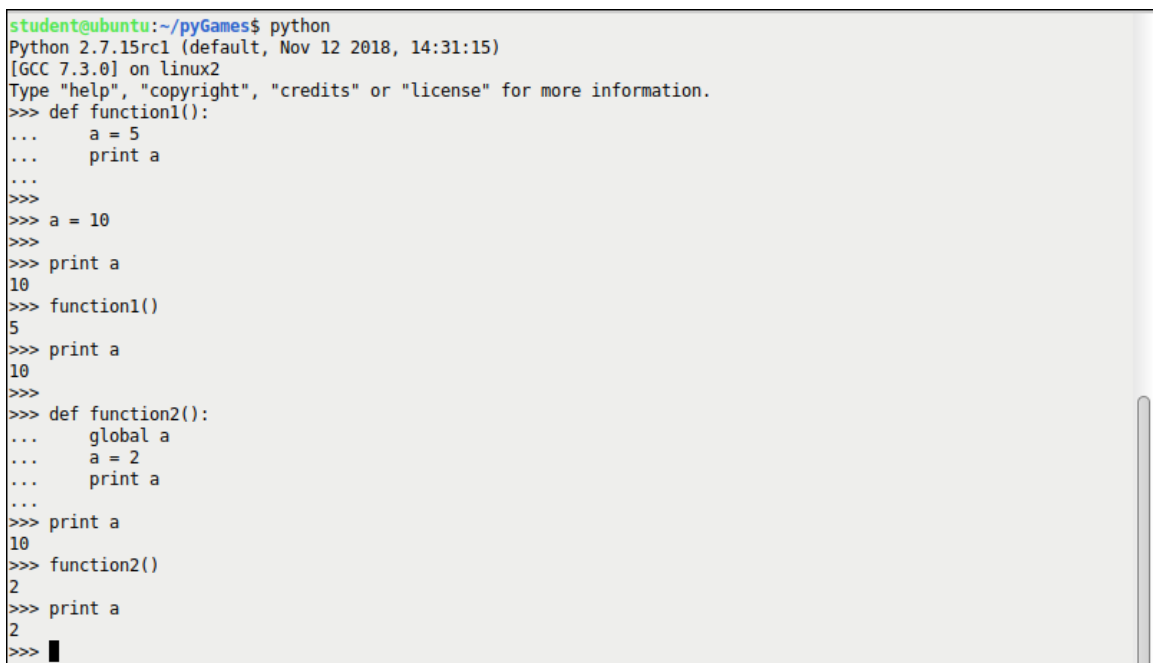
1.6.1. Cơ sở lý thuyết

Đối tượng và hàm được tạo bởi chương trình sẽ được lưu trữ trong không gian tên toàn cục. Người dùng có thể gọi hàm `globals()` để kiểm tra nội dung của không gian tên này, không gian tên được thiết kế dựa trên cấu trúc dữ liệu từ điển và lưu trữ các tên biến và đối tượng nó trở tới. Các dữ liệu này được tự động tạo ra bởi trình thông dịch để lưu trữ biến, đối tượng, các lớp, và các đối tượng khác. Không gian tên toàn cục không phải là không gian tên duy nhất. Một không gian tên khác sẽ được tạo khi hàm được gọi, khi các modules được sử dụng hoặc khi các lớp mới được định nghĩa. Mỗi trường hợp sẽ có không gian tên riêng. Trong các không gian tên cục bộ này, người dùng có thể sử dụng hàm `locals()` để xem nội dung trong đó. Khi Python phân giải một tên biến để tìm giá trị của biến, đầu tiên nó sẽ tìm trong không gian tên cục bộ. Nếu không tồn tại, Python tiếp tục tìm trong phạm vi của bất

kỳ hàm kèm theo nào. Nếu nó vẫn không tìm thấy, trình thông dịch sẽ tìm trong không gian tên toàn cục và cuối cùng là trong các thư viện dựng sẵn.

Khi người dùng nhập vào một thư viện bằng cú pháp “import module”, một tên mới trong không gian tên hiện tại sẽ được tạo cho module đó. Khi người dùng sử dụng module với cú pháp “from module import*”, các đối tượng định nghĩa trong modules sẽ được tạo tên trong không gian tên hiện tại. Không gian tên hiện tại có thể là globals() tại hàm chính, hoặc có thể là không gian tên liên quan tới một module khác hoặc một class, tùy thuộc vào nơi nó được nhập vào.

Khi hàm được thực thi, nó có thể đọc và ghi các biến nằm trong không gian tên cục bộ và có thể đọc các biến nằm trong không gian tên toàn cục. Tuy nhiên, hàm không thể cập nhật giá trị cho các biến nằm toàn cục. Một ví dụ về biến toàn cục và biến cục bộ được thể hiện trong hình 3.12.



```
student@ubuntu:~/pyGames$ python
Python 2.7.15rc1 (default, Nov 12 2018, 14:31:15)
[GCC 7.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def function1():
...     a = 5
...     print a
...
>>>
>>> a = 10
>>>
>>> print a
10
>>> function1()
5
>>> print a
10
>>>
>>> def function2():
...     global a
...     a = 2
...     print a
...
>>> print a
10
>>> function2()
2
>>> print a
2
>>>
```

Hình 3.12. Ví dụ về biến toàn cục và biến cục bộ

Modules là các đoạn mã nguồn có tác dụng mở rộng chức năng cho các hàm dựng sẵn, đối tượng và thư viện trong Python, ta có thể thêm module vào chương trình với hai cách được thể hiện trong đoạn mã phía dưới.

```
1. import Module_name
2. # Or
3. from Module_name import *
```

Sau khi cài đặt Python, người dùng có thể sử dụng các thư viện được mặc định cài đặt sẵn trong nó, dưới đây là danh sách một số thư viện dựng sẵn được đề cập trong đồ án:

- Module sys: Được sử dụng để lấy thông tin và làm việc với hệ thống.
- Module socket: Sử dụng để gửi thông tin qua mạng.
- Module re: Viết tắt của regular expression – biểu thức chính quy, cho phép người dùng trích xuất dữ liệu thông qua luật.
- Module subprocess: Được sử dụng để thực thi các chương trình trên hệ thống và lấy dữ liệu trả về của lệnh đó.
- Ngoài các thư viện dựng sẵn, ta có thể cài đặt thêm các thư viện do bên thứ ba phát triển, danh sách một số thư viện khác được sử dụng trong đồ án:
- Module Scapy: Là một module giúp người dùng đọc, nghe lén, và ghi các gói tin trong mạng.
- Module Requests: Với module này, người dùng dễ dàng tương tác với các ứng dụng web và các máy chủ web.

Để cài đặt các module bên thứ ba, ta cần các phần mềm quản lý gói cho Python (Python package manager). Hai phần mềm quản lý gói được biết đến nhiều nhất là: easy_install và pip. Tham khảo bảng 3.6 về cách cài đặt easy_install, pip trên Ubuntu và bảng 3.7 về cách sử dụng pip với cửa sổ dòng lệnh.

Bảng 3.6. Cách cài đặt easy_install và pip trên Ubuntu

Tên gói	Các bước cài đặt
Easy_install	Bước 1: apt-get install python-setuptools
PIP	Bước 1: tải xuống tập tin get-pip.py từ trang web: https://bootstrap.pypa.io/get-pip.py Bước 2: cài đặt bằng cách chạy câu lệnh: python get-pip.py

Bảng 3.7. Một số ví dụ về cách sử dụng pip

Câu lệnh	Ý nghĩa
\$ pip help install	In ra hướng dẫn sử dụng với câu lệnh install
\$ pip list	Liệt kê các gói đã cài
\$ pip show <installed package>	Lấy thông tin của các gói đã được cài đặt
\$ pip search <keyword>	Tìm kiếm gói giữa trên từ khóa liên quan
\$ pip install <package>	Cài đặt một gói
\$ pip install --upgrade <package>	Nâng cấp phiên bản một gói
\$ pip uninstall <package>	Gỡ cài đặt một gói

Vậy điều gì tạo nên sự khác biệt giữa thư viện với một chương trình bình thường. Ta có thể nhập bất kỳ chương trình nào vào chương trình khác bằng câu lệnh 'import'. Trong vai trò một người lập trình, ta cần phải suy nghĩ về việc đang phát triển một chương trình độc lập hay một chương trình sẽ được sử dụng như một module.

Khi một chương trình được nhập vào chương trình khác, Python sẽ thực thi đoạn kịch bản đó, tạo các hàm và chạy các mã lệnh nằm trong kịch bản. Giả sử ta có một chương trình đọc các đối số dòng lệnh. Nếu đoạn mã không thấy được các đối số cụ thể được truyền trên dòng lệnh và ta quyết định nhập nó ấy để sử dụng một số chức năng đã định nghĩa, thì chương trình chính sẽ không hoạt động như mong muốn. Vì vậy ta cần các chương trình thực thi chức năng chính của nó khi người dùng chạy chương trình và không tự chạy các chức năng khi người dùng import chương trình. Điều này được xử lý bằng cách sử dụng biến `__name__`. Bằng cách sử dụng nó, ta có thể xác định xem tập lệnh đang được nhập hay thực thi và để nó hoạt động khác nhau trong từng trường hợp.

```
1. # ! /usr/bin/python
2. # User can comment a single line with a pound sign
3. ....
4. The first string in the program is the DocString and is used by
5. help function to describe the program
6. '''
7. import sys
8.
9. def main():
10.     'A "Docstring" for the main function here'
11.     print 'You passed the argument: ' + sys.argv[1]
```



```
12.  
13. if __name__ == '__main__':  
14.     main()
```

Quay lại cấu trúc một chương trình cơ bản của Python, đã được nhắc tới trong phần 3.1. Ở đây, ta có một câu lệnh “if” để kiểm tra biến `__name__` với giá trị `'__main__'`. Câu lệnh điều khiển để kiểm tra chương trình này đang được thực thi hoặc được nhập bởi chương trình khác. Trong trường hợp chương trình được nhập bởi chương trình khác, biến `__name__` sẽ có giá trị khác với `'__main__'` và hàm `main()` sẽ không được chạy, và ngược lại, khi chương trình được thực thi, hàm `main()` sẽ được sử dụng, hình 3.13.

```
student@ubuntu:~/essentials$ cat module_or_not.py  
print __name__  
student@ubuntu:~/essentials$ python module_or_not.py  
__main__  
student@ubuntu:~/essentials$ python  
Python 2.7.15rc1 (default, Nov 12 2018, 14:31:15)  
[GCC 7.3.0] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import module_or_not  
module_or_not  
>>> █
```

Hình 3.13. Ví dụ về sử dụng biến `__name__`

Sau khi đã cài đặt các module và phân biệt được module với một chương trình thông thường, cuối cùng, người dùng cần các kỹ thuật để phân tích chi tiết các thông tin về một module cũng như cách dùng các phương thức, lớp đã được định nghĩa sẵn trong module đó. Với Python shell, ta có thể sử dụng các hàm như `dir()`, `help()` và `type()`. Hàm `help()` để kiểm tra các thông tin được ghi chú trong các phương thức, đối tượng, ví dụ: DocStrings, biến `__doc__`. Hàm `type()` để kiểm tra đối tượng là một thuộc tính hay phương thức.

Để minh họa cho việc phân tích module, đồ án sử dụng thư viện `hashlib` để tính toán giá trị băm của từ ‘KMA’ với thuật toán băm MD5. Với các công cụ tính toán có sẵn, ta biết trước giá trị băm MD5 của từ ‘KMA’ là: `b850d67b5a5cd84194d5fa8edcc0ab42`.

```

>>> import hashlib
>>> help(hashlib)

>>> dir(hashlib)
['_all_', '__builtins__', '__doc__', '__file__', '__get_builtin_constructor__', '__name__', '__package__', 'hashlib', 'algorithms', 'algorithms_available', 'algorithms_guaranteed', 'md5', 'new', 'pbkdf2_hmac', 'sha1', 'sha224', 'sha256', 'sha384', 'sha512']
>>> print hashlib.md5.__doc__
Returns a md5 hash object; optionally initialized with a string
>>> dir(hashlib.md5('KMA'))
['_class_', '__delattr__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'block_size', 'copy', 'digest', 'digest_size', 'digestsize', 'hexdigest', 'name', 'update']
>>> type(hashlib.md5('KMA'))
<type 'hashlib.HASH'>
>>> type(hashlib.md5('KMA').digest())
<type 'builtin function or method'>
>>> hashlib.md5('KMA').digest()
'\xb8P\xd6{Z\\\xd8A\x94\xd5\xfa\xe\xdc\xC0\xabB'
>>> hashlib.md5('KMA').digest().encode('hex')
'b850d67b5a5cd84194d5fa8edcc0ab42'
>>> hashlib.md5('KMA').hexdigest()
'b850d67b5a5cd84194d5fa8edcc0ab42'
>>> █

```

Hình 3.14. Phân tích và ứng dụng thư viện hashlib

Hình 3.14. minh họa các bước phân tích và sử dụng thư viện hashlib. Bước đầu tiên, ta tiến hành nhập module hashlib với từ khóa import. Tiếp theo, tiến hành đọc thông tin của module bằng help() và dir(), sử dụng ‘q’ để thoát khỏi hàm help(). Với help(), ta biết được hashlib có một phương thức để tính toán các giá trị băm của một chuỗi ký tự, đó là hàm md5(), hàm này nhận đối số là một chuỗi ký tự và trả về đối tượng băm. Với đối tượng băm của chuỗi ‘KMA’, ta có thể sử dụng thêm các hàm dựng sẵn trong module như digest() và hexdigest() để tính toán ra giá trị cần tìm.

1.6.2. Bài tập thực hành

Giúp rèn luyện kỹ năng của người học trong việc sử dụng các thư viện, pyGames cung cấp một danh sách các thử thách liên quan đến chúng.

Chuẩn bị:

- Người học cần có kiến thức cơ bản về thư viện, cách sử dụng pip.
- Kết nối mạng Internet.

Danh sách các bài thực hành: Bài 12 và bài 13.

1.7. Các câu lệnh điều khiển và kiểu dữ liệu Lists

1.7.1. Cơ sở lý thuyết

a) If elif và else

Giống với các ngôn ngữ lập trình phổ biến khác, trong Python, các câu lệnh điều khiển được sử dụng để thay đổi luồng logic chương trình của người dùng. Các câu lệnh điều khiển bao gồm: if/elif/else, vòng lặp for và vòng lặp while. Vòng lặp for và while thường được sử dụng trong việc thao tác với các dữ liệu dạng danh sách, từ điển,... Tuy nhiên, cho đến hiện tại, đề án chưa đề cập đến các kiểu cấu trúc dữ liệu nói trên, vì vậy trong phần này, ta sẽ tập trung vào các câu lệnh điều khiển if/elif/else và sẽ nhắc lại các kiến thức về while và for trong các phần sau.

Rất ít chương trình thực thi tuần tự từ trên xuống dưới mà không có bất kỳ nhánh nào trong mã lệnh. Khi chương trình gặp một câu lệnh điều khiển, ví dụ như “if” hoặc vòng lặp “for”, ngay lập tức chương trình nhảy đến các hàm và khối mã khác nhau trong bộ nhớ. Trong khi câu lệnh “if” sẽ chia chương trình thành hai hoặc nhiều nhánh. Vòng lặp “for” và “while” sẽ khiến chương trình nhảy ngược lại đến các khối mã được thực thi trước đó và thực thi chúng cho đến khi một số điều kiện được đáp ứng.

Để sử dụng câu lệnh điều khiển “if”, người dùng có thể sử dụng cú pháp:

```
1. if <logic expression>:  
2.     # Code Block
```

Trong đó:

- If là từ khóa.
- Sau đó là biểu thức logic, nó sẽ được đánh giá và tính toán bởi Python. Nếu biểu thức logic là đúng, thì khối mã theo sau câu lệnh “if” sẽ được thực thi.

```

>>> user1 = 'root'
>>> user2 = 'student'
>>>
>>> def compare_username(name):
...     print name
...     if name == 'root':
...         print 'Welcome to ROOT-Area'
...     if name != 'root':
...         print '403! Un-authorization'
...
>>> compare_username(user1)
root
Welcome to ROOT-Area
>>> compare_username(user2)
student
403! Un-authorization
>>> █

```

Hình 3.15. Ví dụ về câu lệnh điều khiển “if”

Hình 3.15 bên trên mang ví dụ về cách sử dụng “if”. Có thể tham khảo danh sách các toán tử logic mà người dùng có thể sử dụng trong biểu thức logic trong các bảng 3.8, 3.9 và 3.10.

Bảng 3.8. Các toán tử logic trong câu lệnh điều khiển “if”

Toán tử	Ý nghĩa	Ví dụ
<	Nhỏ hơn	$I < 100$
<=	Nhỏ hơn hoặc bằng với	$I \leq 100$
>	Lớn hơn	$I > 100$
>=	Lớn hơn hoặc bằng với	$I \geq 100$
==	Bằng	$I == 100$
!=	Khác	$I != 100$
not	Đúng khi biểu thức logic có kết quả sai	$\text{not } I = 1$
and	AND logic	$(I \leq 9) \text{ and } (X == \text{True})$
or	OR logic	$(I > 3) \text{ or } (F > 100.5)$

Bảng 3.9. Bảng chân lý - AND

Biến A	Biến B	A AND B
False (0)	False (0)	False (0)
False (0)	True (1)	False (0)
True (1)	False (0)	False (0)

True (1)	True (1)	True (1)
----------	----------	----------

Bảng 3.10. Bảng chân lý - OR

Biến A	Biến B	A AND B
False (0)	False (0)	False (0)
False (0)	True (1)	True (1)
True (1)	False (0)	True (1)
True (1)	True (1)	True (1)

Đối với cặp lệnh điều khiển “if/else” ta khai báo với cú pháp như sau:

```

1. if <logic expression>:
2.     # Code Block 1
3.     # Code Block 1
4. else:
5.     # Code Block 2
6.     # Code Block 2

```

Trong đó:

- If và else là hai từ khóa.
- Sau từ khóa “if” là biểu thức điều kiện. Nếu biểu thức sau từ khóa “if” đúng, khối lệnh sau đó sẽ được thực thi. Câu lệnh điều khiển ‘else’ chỉ thực thi nếu biểu thức điều kiện sai.

Có thể tham khảo hình 3.16 để hiểu được cách vận dụng của một cặp từ khóa “if/else”.

```

>>> user1 = 'root'
>>> user2 = 'student'
>>>
>>> def compare_username(name):
...     print name
...     if name == 'root':
...         print 'Welcome to ROOT-Area'
...     else:
...         print '403! Un-authorization'
...
>>> compare_username(user1)
root
Welcome to ROOT-Area
>>> compare_username(user2)
student
403! Un-authorization
>>> █

```

Hình 3.16. Ví dụ về câu lệnh điều khiển ‘if/else’

Câu lệnh điều khiển “elif”, viết tắt của else if, có thể được sử dụng để thêm các trường hợp rẽ nhánh cho câu lệnh điều khiển “if”. Người dùng có thể khai báo nhiều mệnh đề “elif” sau câu lệnh “if”, hình 3.17. Tiến hành khai báo câu lệnh ‘elif’ với cú pháp:

```
1. elif <logic expression>:  
2.     # Code Block
```

```
>>> def is_this_five(number):  
...     print number  
...     if number < 5:  
...         print '%d is less than 5' %number  
...     elif number == 5:  
...         print 'This is five'  
...     else:  
...         print '%d is more than 5' %number  
...  
>>> is_this_five(1)  
1  
1 is less than 5  
>>> is_this_five(5)  
5  
This is five  
>>> is_this_five(10)  
10  
10 is more than 5  
>>> █
```

Hình 3.17. Ví dụ về câu lệnh điều khiển “if/elif/else”

b) Kiểu dữ liệu Lists

Mảng là một phần cơ bản của nhiều ngôn ngữ lập trình, nó là tập hợp các phần tử của một kiểu dữ liệu duy nhất, ví dụ: mảng số nguyên, mảng chuỗi. Với những lập trình viên đã quen thuộc với mảng sẽ thấy rằng, kiểu dữ liệu Lists - danh sách - khá tương tự và chỉ có một số ngoại lệ nhất định.

Trong Python, ta có thể tạo một danh sách bằng cách gán một biến bằng với một danh sách nhiều đối tượng khác nhau được ngăn cách bằng dấu phẩy và được xếp trong cặp dấu ngoặc vuông. Ta cũng có thể tạo một danh sách rỗng với cú pháp: `empty_list = []`. Trong một danh sách, đối tượng đầu tiên của danh sách sẽ có chỉ mục là 0, giá trị của chỉ mục sẽ tăng dần từ 0,1,2... theo thứ tự từ trái qua phải. Danh sách có thể chứa tất cả các loại dữ liệu của đối tượng, ví dụ: người dùng có thể tạo một danh sách chứa cả số, chuỗi ký tự và các danh sách con,... Python sẽ không tạo một danh sách mới nếu ta sửa đổi một phần tử trong danh sách đã tạo. Hình 3.18 minh họa về kiểu dữ liệu này.

```

>>> list1 = ['Alice', 1, ['1', 2]]
>>> list1[0]
'Alice'
>>> type(list1[0])
<type 'str'>
>>> list1[1]
1
>>> type(list1[1])
<type 'int'>
>>> list1[2]
['1', 2]
>>> type(list1[2])
<type 'list'>
>>>

```

Hình 3.18. Ví dụ về kiểu dữ liệu Lists

Với một danh sách trống, người dùng sẽ không thể gán các giá trị cho các chỉ mục của nó, thay vào đó người dùng cần định nghĩa ra kích thước của danh sách – số phần tử mặc định mà danh sách có thể chứa – hoặc sử dụng phương thức `append()` để thêm các đối tượng vào danh sách loại này. Ta cũng không thể gán giá trị cho một chỉ mục, mà chỉ mục này có giá trị lớn hơn độ dài của danh sách trừ đi 1. Để định nghĩa kích thước của một danh sách ta tham khảo hình ảnh dưới đây.

```

>>> #newlist = [default value] * <size of the list>
...
>>> newlist = [None] * 10
>>> print newlist
[None, None, None, None, None, None, None, None, None, None]
>>> newlist[0] = 'Alice'
>>> newlist[10] = 'Error'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>>

```

Hình 3.19. Định nghĩa kích thước của một danh sách

Lists là các đối tượng với một số phương thức cho phép quản lý dữ liệu chứa trong nó. Bảng 3.11 mô tả một số phương thức thường được sử dụng khi tương tác với danh sách và hình 3.20 lấy ví dụ cho các phương thức này.

Bảng 3.11. Một số phương thức sử dụng với Lists

Các hàm/Phương thức	Ý nghĩa
<code>List[index] = value</code>	Thay đổi giá trị của một chỉ mục
<code>List.append(value)</code>	Thêm đối tượng vào cuối của danh sách

List.insert(position , value)	Thêm một giá trị vào vị trí đã đưa ra trong danh sách, vị trí nhận giá trị số nguyên âm hoặc số nguyên dương
List.remove(value)	Loại bỏ đối tượng đầu tiên có giá trị tương ứng
List.sort(key , direction)	Sắp xếp lại danh sách dựa trên giá trị ASCII của đối tượng
List.count(value)	Đếm số lượng các đối tượng có cùng giá trị
List.index(value)	Tìm kiếm giá trị của chỉ mục với giá trị cho trước
del List[index]	Xóa một đối tượng thông qua chỉ mục

```

>>> students = ['Alice', 'Bob', 'Eve']
>>> students.index('Eve')
2
>>> students.insert(1, 'Manh')
>>> students
['Alice', 'Manh', 'Bob', 'Eve']
>>> students.append('Cloud')
>>> students
['Alice', 'Manh', 'Bob', 'Eve', 'Cloud']
>>> students.remove('Cloud')
>>> students
['Alice', 'Manh', 'Bob', 'Eve']
>>> students.reverse()
>>> students
['Eve', 'Bob', 'Manh', 'Alice']
>>> del students[2]
>>> students
['Eve', 'Bob', 'Alice']
>>> █

```

Hình 3.20. Minh họa một số phương thức sử dụng với Lists

Khi người dùng gán một biến bằng với một danh sách đã tồn tại, người dùng đã tạo ra một con trỏ, trỏ cùng tới địa chỉ bộ nhớ với danh sách trước đó, nếu có bất kỳ thay đổi nào với danh sách đang tồn tại trong biến thì danh sách cũ cũng sẽ bị thay đổi theo. Để tạo một bản sao chép của một danh sách, ta sử dụng hàm list() như ví dụ sau, hình 3.21.


```

>>> list1 = ['demo', 'copy', 'a', 'list']
>>> a = list1
>>> print a
['demo', 'copy', 'a', 'list']
>>> a.append('variable "a" is a pointer')
>>> print list1
['demo', 'copy', 'a', 'list', 'variable "a" is a pointer']
>>> list2 = list(list1)
>>> list2.append('list2 is a copy of list1')
>>> print list1
['demo', 'copy', 'a', 'list', 'variable "a" is a pointer']
>>> print list2
['demo', 'copy', 'a', 'list', 'variable "a" is a pointer', 'list2 is a copy of list1']
>>> id(a)
140153781201592
>>> id(list1)
140153781201592
>>> id(list2)
140153781202456
>>> █

```

Hình 3.21. Tạo một bản sao của danh sách

Như trong phần 3.5. – lý thuyết về kiểu dữ liệu Strings, ta đã biết cách sử dụng hàm `split()` để chia một chuỗi thành một danh sách. Nếu ta không sử dụng đối số nào với hàm này, theo mặc định `split()` sẽ cắt chuỗi tạo bất kỳ ký tự khoảng trắng nào. Nếu sử dụng đối số, có thể là một ký tự hoặc một chuỗi ký tự, chuỗi sẽ được cắt tại các ký tự giống với đối số đã được truyền vào, kết quả ta được một danh sách mới không tồn tại đối số đã biết. Với chiều hướng ngược lại, ta có hàm `join()` để chuyển đổi một danh sách các chuỗi ký tự thành chuỗi ký tự duy nhất. Hình 3.22 thể hiện cách chuyển đổi giữa hai kiểu dữ liệu nói trên.

```

>>> ' '.join(['From', 'List', 'to', 'String'])
'From List to String'
>>> ''.join(['K', 'M', 'A'])
'KMA'
>>> '+'.join(['P', 'L', 'U', 'S'])
'P+L+U+S'
>>> print 'This is a fun stuff'.split()
['This', 'is', 'a', 'fun', 'stuff']
>>> print 'This is a fun stuff'.split('is')
['Th', ' ', ' ', 'a fun stuff']
>>> █

```

Hình 3.22. Chuyển đổi giữa Strings và Lists

Để tính tổng các số trong một danh sách số, ta sử dụng hàm `sum()`. Hàm `zip()` sẽ kết hợp hai hoặc nhiều danh sách với nhau, sau đó trả về một danh sách, hình 3.23. Danh sách này gồm nhiều tuple – kiểu dữ liệu Tuples – Mỗi tuple chứa các đối tượng từ các danh sách sao cho các đối tượng này có cùng một vị trí chỉ mục.

```
>>> sum([2, 3, 5])
10
>>> zip([0, 1, 2], ['a', 'b', 'c'])
[(0, 'a'), (1, 'b'), (2, 'c')]
>>> zip([0, 1], ['a', 'b'], ['x', 'y', 'z'])
[(0, 'a', 'x'), (1, 'b', 'y')]
>>> █
```

Hình 3.23. Hàm sum() và zip()

Hàm ‘map’ được sử dụng để áp dụng một chức năng cho mọi đối tượng trong danh sách hoặc cho iterable (iterable là bất kỳ cấu trúc dữ liệu nào mà người dùng có thể thực hiện với vòng lặp FOR). Đối số đầu tiên của hàm ‘map’ chính là tên hàm mà người dùng muốn áp dụng cho các đối tượng trong danh sách. Đối số thứ hai là danh sách cần tương tác. Nếu ta đưa vào hai hoặc nhiều hơn các danh sách, hàm ‘map’ sẽ có thêm chức năng gộp của hàm zip(), hình 3.24.

```
>>> def plus_1(a):
...     return a + 1
...
>>> map(plus_1, [0, 1, 100])
[1, 2, 101]
>>>
>>> def plus(x, y):
...     return x + y
...
>>> map(plus, [1, 2], [3, 4])
[4, 6]
>>> █
```

Hình 3.24. Ứng dụng hàm map() với Lists

Một cấu trúc dữ liệu gần giống với Lists đó là Tuples. Để khai báo một tuple, tham khảo hình 3.25. Mặc dù kiểu dữ liệu này có ít hàm hỗ trợ hơn so với kiểu dữ liệu Lists, nhưng ta có thể coi đây cũng là một điểm mạnh của nó, đó là sử dụng hiệu quả trong việc lưu trữ các bản ghi (tốn ít bộ nhớ hơn so với kiểu dữ liệu Lists) và sắp xếp các bản ghi.

```
>>> tuple1 = ('we', 'are', 'KMA')
>>> tuple2 = 'we', 'are', 'KMA'
>>>
>>> list1 = ['we', 'are', 'KMA']
>>>
>>> import sys
>>>
>>> sys.getsizeof(tuple(range(1000000)))
8000056
>>> sys.getsizeof(list(range(1000000)))
9000120
>>> █
```

Hình 3.25. Cách khai báo Tuples

c) Vòng lặp for và vòng lặp while

Vòng lặp for và vòng lặp while được sử dụng để thông qua các đối tượng nằm trong một danh sách, từ điển hay các cấu trúc dữ liệu lặp lại khác. Khi sử dụng vòng lặp for hay vòng lặp while, người dùng cần định nghĩa một khối mã lệnh để thực thi cho mỗi một đối tượng nằm trong cấu trúc dữ liệu.

Hầu hết các vòng lặp for được sử dụng để thông qua các đối tượng nằm trong một danh sách bằng cách sử dụng cú pháp: `for <iterator> in <list>`. Nó sẽ thực thi khối mã, số lần thực thi tương ứng với số lượng đối tượng nằm trong danh sách. Khi khối mã được thực thi, biến iterator sẽ được gán giá trị của một đối tượng. Vì vậy, khi vòng lặp được thực thi lần đầu, biến `<iterator>` sẽ có giá trị của đối tượng ở chỉ mục số 0, lần thứ hai nó sẽ chứa giá trị của đối tượng ở chỉ mục số 1, tiếp tục như vậy cho tới hết danh sách. Để hiểu rõ hơn, tham khảo ví dụ trong hình 3.26.

```
>>> list1 = 'I want to change the world!'.split()
>>> list1
['I', 'want', 'to', 'change', 'the', 'world!']
>>> for a_string in list1:
...     print a_string
...
I
want
to
change
the
world!
>>> █
```

Hình 3.26. Vòng lặp for với Lists

Đôi khi, ta cần đếm hoặc tính toán số lượng trong một khoảng các số, khi đó ta có thể sử dụng hàm `range()`. Hàm này trả về một danh sách chứa các số, các số trong danh sách phụ thuộc vào các đối số đầu vào. Hàm `range()` nhận ba tham số truyền vào: số bắt đầu chuỗi, số cần dừng lại trước nó, bước nhảy. Từ kết quả trả về của hàm `range()`, ta có thể dễ dàng sử dụng vòng lặp for để tính toán ra các kết quả mong muốn, ví dụ hình 3.27.

```

>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0, 15, 2)
[0, 2, 4, 6, 8, 10, 12, 14]
>>> for x in range(1, 11, 1):
...     print x
...
1
2
3
4
5
6
7
8
9
10
>>> █

```

Hình 3.27. Hàm range()

Trong một số trường hợp, ta cần giá trị của đối tượng và vị trí của đối tượng trong danh sách, nhưng vòng lặp for chỉ thông qua một tham số. Với trường hợp này, ta có thể sử dụng hàm enumerate() để trích xuất thông tin giá trị và vị trí của các đối tượng trong một danh sách. Vị trí được trích xuất bắt đầu từ 0 cho đối tượng đầu tiên, tiếp tục tăng cho tới giá trị độ dài của chuỗi trừ đi một. Hàm này trả về kết quả là một danh sách mới, danh sách chứa cấu trúc dữ liệu dạng Tuples, mỗi tuple chứa hai đối tượng đó là: vị trí và giá trị của các đối tượng trong danh sách cũ. Hình 3.28 minh họa cho việc sử dụng hàm enumerate().

```

>>> list1 = ['KMA', 'pyGames', 281196]
>>> for index, value in enumerate(list1):
...     print '%d + %s' %(index, value)
...
0 + KMA
1 + pyGames
2 + 281196
>>> list(enumerate(list1))
[(0, 'KMA'), (1, 'pyGames'), (2, 281196)]
>>> █

```

Hình 3.28. Hàm enumerate()

Trong khi vòng lặp for là hữu hạn lần, vòng lặp while có thể dẫn tới vô hạn lần lặp. Vòng lặp while thường được sử dụng khi người dùng cần tiếp tục lặp lại một công việc cho tới khi công việc hoàn tất. Vòng lặp while sẽ chỉ dừng lại khi biểu thức logic có giá trị đúng hoặc gặp phải câu lệnh break. Ta có thể khai báo một vòng lặp while với cú pháp: while <logic statement>: . Tham khảo hình ảnh phía dưới về cách sử dụng vòng lặp while.

```

>>> import random
>>> guess = ''
>>> answer = random.randrange(1, 5)
>>> while guess != answer:
...     guess = int(raw_input('Your guessing number > '))
...     if guess != answer:
...         print 'Try again!'
...     else:
...         print 'Correct!'
...
Your guessing number > 1
Try again!
Your guessing number > 2
Try again!
Your guessing number > 3
Correct!
>>> █

```

Hình 3.29. Ví dụ về vòng lặp while

Break và continue là hai câu lệnh được sử dụng trong vòng lặp for và vòng lặp while để điều khiển luồng thực thi của chương trình. Nếu Python gặp lệnh continue, tất cả các mã nguồn còn lại của vòng lặp hiện tại sẽ dừng lại, thay vào đó, luồng thực thi sẽ bắt đầu lại từ đầu vòng lặp. Nếu vòng lặp đó là vòng for, Python sẽ bắt đầu với đối tượng tiếp theo trong danh sách. Nếu đó là vòng lặp while, Python sẽ kiểm tra tính đúng sai của câu lệnh điều kiện, nếu đúng thì tiếp tục thực thi vòng lặp. Nếu Python gặp lệnh break, nó sẽ ngay lập tức thoát khỏi vòng lặp for hoặc vòng lặp while.

1.7.2. Bài tập thực hành

Ôn tập lại các kỹ năng của người học với kiểu dữ liệu Lists, giúp người học làm quen với vòng lặp for ở mức độ cơ bản.

Chuẩn bị:

- Người học cần có kiến thức về Lists và các dạng vòng lặp.
- Người học cần có sẵn một phiên làm việc với pyGames, pygamedevserver và một biến chứa đối tượng pyGames.game().

Danh sách các bài thực hành: Bài 14, bài 15, bài 16, bài 17, bài 18, bài 19, bài 20, bài 21, bài 22, bài 23, và bài 24.

1.8. Kiểu dữ liệu Dictionaries

1.8.1. Cơ sở lý thuyết

Giống với danh sách, kiểu dữ liệu từ điển có thể sử dụng để lưu trữ và trích xuất dữ liệu. Điểm khác biệt ở đây là thay vì sử dụng chỉ mục dưới dạng số nguyên,

Python cung cấp chỉ mục dưới dạng các giá trị. Các giá trị này được gọi là key (khóa) và có thể được lưu trữ dưới bất kỳ kiểu dữ liệu nào, ví dụ như: số nguyên, chuỗi ký tự... Một ví dụ dễ hiểu hơn, Python dictionaries giống như những bảng giá trị hashed trong các ngôn ngữ lập trình khác, cho phép lưu trữ và trích xuất dữ liệu một cách rất nhanh.

Ta có thể khai báo một từ điển bằng rỗng bằng cách gán giá trị của biến với một cặp ngoặc nhọn { }. Sau đó, giống như Lists, tiến hành gán từng value (giá trị) của từ điển với key đã biết và cũng từ các key, ta có thể trích xuất các value tương ứng. Tham khảo cách khai báo từ điển tại hình 3.30.

```
>>> dict = {}
>>> dict['a'] = 'Alpha'
>>> dict['b'] = 'Beta'
>>> dict['g'] = 'Gamma'
>>>
>>> dict
{'a': 'Alpha', 'b': 'Beta', 'g': 'Gamma'}
>>> dict['a']
'Alpha'
>>> dict['c']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
>>> █
```

Hình 3.30. Khai báo một từ điển

Với một cách khác, dùng hàm get() để trích xuất value từ một dictionary, hàm này lấy tối đa hai tham số, đối số đầu tiên là key mà ta cần xuất value, giá trị trả về sẽ là None nếu key không tồn tại. Trong trường hợp có đối số thứ hai, nếu key không tồn tại, giá trị trả về sẽ là chính đối số vừa truyền vào.

Để tạo một bản sao của một từ điển, sử dụng hàm dict() với tham số truyền vào là từ điển muốn sao chép, hoặc từ từ điển cần sao chép, sử dụng phương thức copy() như trong hình 3.31.

```

>>> dict1 = {'a': 'alpha', 'd': 'delta'}
>>>
>>> dict2 = dict(dict1)
>>>
>>> dict3 = dict1.copy()
>>>
>>> dict2
{'a': 'alpha', 'd': 'delta'}
>>> dict3
{'a': 'alpha', 'd': 'delta'}
>>>
>>> id(dict1)
140085002471976
>>> id(dict2)
140085002446560
>>> id(dict3)
140085002472256
>>> █

```

Hình 3.31. Sao chép một từ điển

Ngoài hàm `copy()`, các từ điển còn có các hàm dựng sẵn khác rất hữu ích, bảng 3.12 mô tả một số hàm nói trên.

Bảng 3.12. Các hàm dựng sẵn với Dictionaries

Hàm dựng sẵn	Chức năng
<code>items()</code>	Trả về một danh sách các tuple, tuple chứa các cặp (key,value)
<code>keys()</code>	Trả về một danh sách các keys
<code>values()</code>	Trả về một danh sách các values
<code>has_key('key')</code>	Trả về giá trị True nếu key tồn tại trong từ điển, trả về giá trị False nếu key không tồn tại trong từ điển

Với những hàm dựng sẵn, ta có thể dễ dàng kiểm tra một giá trị có tồn tại trong từ điển đã có hay không như hình dưới đây.

```

>>> print dict1
{'a': 'alpha', 'd': 'delta'}
>>> 'alpha' in dict1.values()
True
>>> 'gamma' in dict1.values()
False
>>> 'a' in dict1
True
>>> 'g' in dict1
False
>>> █

```

Hình 3.32. Kiểm tra dữ liệu trong một từ điển

Như đã đề cập tới trong phần lý thuyết của vòng lặp for và vòng lặp while, hai vòng lặp này có thể được sử dụng trong việc tương tác với Lists và Dictionaries. Trong kiểu dữ liệu từ điển, ta có 3 dữ liệu dạng iterable đó là: keys, values và items.

Hàm keys() trả về một danh sách các từ khóa có trong từ điển. Với danh sách các khóa này, ta có thể tương tác và trích xuất ra các giá trị tương ứng trong từ điển với vòng lặp. Để tối ưu bộ nhớ hơn, ta có thể sử dụng hàm iterkeys() để tương tác với vòng lặp, hình 3.33.

```
>>> print dict1
{'a': 'alpha', 'd': 'delta'}
>>> for k in dict1:
...     print k + ' ' + dict1[k]
...
a alpha
d delta
>>> for k in dict1.keys():
...     print k + ' ' + dict1[k]
...
a alpha
d delta
>>> for k in dict1.iterkeys():
...     print k + ' ' + dict1[k]
...
a alpha
d delta
>>>
```

Hình 3.33. Vòng lặp tương tác với từ khóa

Tương tự với values và items, ta sử dụng các hàm như itervalues() và iteritems() với các vòng lặp. Tham khảo hình 3.34.

```
>>> for v in dict1.itervalues():
...     print v
...
alpha
delta
>>> for i in dict1.iteritems():
...     print i
...
('a', 'alpha')
('d', 'delta')
>>>
```

Hình 3.34. itervalues() và iteritems() trong Dictionaries

1.8.2. Bài tập thực hành

Mục đích: Để người học hiểu và sử dụng thành thạo những cách tương tác với kiểu dữ liệu Dictionaries.

Chuẩn bị:

- Người học cần có kiến thức về kiểu dữ liệu từ điển.
- Người học cần có sẵn một phiên làm việc với pyGames, pygamesserver và một biến chứa đối tượng pyGames.game().

Danh sách các bài thực hành: Bài 25, bài 26, bài 27 và bài 28.