Lập trình Python

Nội dung

1. Các khái niệm căn bản
2. Phát biểu if
3. Vòng lặp
4. Danh sách (List)
5. Bộ (Tuple)
6. Tập hợp (Set)
7. Dictionary
8. Hàm
9. Lớp
10. Numpy

1. Các khái niệm căn bản

Tạo file chương trình Python

- Có thể tạo ra một file chương trình Python như là một file văn bản chứa các phát biểu và dùng câu lệnh sau để thực thi các phát biểu trong file

  python filename.py

- File chương trình có thể được tạo ra bằng cách dùng một text editor, thí dụ như Notepad. File sẽ được đặt tên và có đuôi là .py.
- Chương trình Python sau đây sẽ hiển thị hai thông báo **Welcome to Python** và **Python is fun**.
- **LISTING 1.1** Welcome.py

```
# Display two messages
print("Welcome to Python")
print("Python is fun")
```

Định danh (Identifier)

- Định danh là tên mà chúng ta đặt cho một biến (variable) hoặc một hàm (function)
- Các định danh phải tuân theo các qui tắc sau identifiers must obey the following rules:
  - Một định danh là một chuỗi ký tự bao gồm ký tự chữ, ký tự số và dấu underscore (_).
  - Một định danh phải bắt đầu bằng một ký tự chữ hai dấu underscore. Nó không được bắt đầu bằng một ký tự số.
  - Một định danh không được trùng với một từ khóa (keyword).
  - Một định danh có chiều dài bất kỳ.
- Thí dụ: num1, num2, num3, area, radius

Biến, phát biểu gán và biểu thức

- Phát biểu mà gán một giá trị vào một biến (variable) được gọi là phát biểu gán (assignment statement) In
- Trong Python, dấu (**=**) được dùng làm toán tử gán (assignment operator). Ngữ pháp của phát biểu gán:

  variable = expression
- Biểu thức (expression) diễn tả một tính toán liên quan đến các biến, các trị và các toán tử mà giúp cho định trị được biểu thức.
- Thí dụ:

  x = x + **1**

  average = (number1 + number2 + number3) / **3**

  i = j = k = **1**

  Thí dụ thứ ba: một trị mà gán cho nhiều biến.

Kiểu dữ liệu số và các toán tử số học

- Python có hai loại dữ liệu số: số nguyên (integer) và số thực (floating-point number) để làm việc với các toán tử **+**, **-**, *****, ***/***, ***//***, ******, and **%**.

Bảng 2.1 Các toán tử số học

Tên	*Ý nghĩa*	*Thí dụ*	*Kết quả*
+	Cộng	**34 + 1**	**35**
-	Trừ	**34.0 - 0.1**	**33.9**
*****	Nhân	**300 * 30**	**9000**
/	Chia thực	**1 / 2**	**0.5**
//	Chia nguyên	**1 // 2**	**0**
******	Lũy thừa	**4 ** 0.5**	**2.0**
%	Phần dư	**20 % 3**	**2**

Thứ tự ưu tiên của các toán tử: 1) lũy thừa, 2) nhân chia, 3) cọng trừ

Hàm Python thông dụng

Bảng 2.2 Hàm Python có sẵn

Tên hàm	*Mô tả*	*Thí dụ*
abs(x)	Trả về trị tuyệt đối của **x**.	**abs(-2)** là **2**
max(x1, x2, ...)	Trả về trị lớn nhất trong số **x1**, **x2**, ...	**max(1, 5, 2)** là **5**
min(x1, x2, ...)	Trả về trị nhỏ nhất trong số **x1**, **x2**, ...	**min(1, 5, 2)** là **1**
pow(a, b)	Trả về $\mathbf{a^b}$. Tương tự như **a ** b**.	**pow(2, 3)** là **8**
round(x)	Tính tròn giá trị x thành một số nguyên. Nếu x là giá trị nằm giữa hai số nguyên, số nguyên chẵn sẽ được chọn.	**round(5.4)** is **5** **round(5.5)** là **6** **round(4.5)** là **4**
round(x, n)	Trả về trị số thực được làm tròn với **n** ký tự số sau dấu chấm thập phân.	**round(5.466, 2)** là **5.47** **round(5.463, 2)** là **5.46**

Dòng ký tự

- Một dòng ký tự (string) là một chuỗi gồm các ký tự. Python xem các ký tự và các dòng ký tự như nhau.

letter = **'A'**
numChar = **'4'**
message = **"Good morning"**

- Toán tử ghép kề (concatenation) dòng ký tự

Toán tử **+** có thể ghép kề hai dòng ký tự.

Vài thí dụ:

```
>>> message = "Welcome " + "to " + "Python"
>>> message
'Welcome to Python'
>>> chapterNo = 3
>>> s = "Chapter " + str(chapterNo)
>>> s
'Chapter 3'
```

Biểu thức logic và toán tử so sánh

- *Biểu thức logic là một biểu thức mà có giá trị được tính là giá trị logic* **True** *hay* **False***.*
- Python cung cấp 6 toán tử so sánh (còn được gọi là toán tử quan hệ (*relational operator*): <, **<= , >, >=, ==, !=**

Thí dụ:

```
radius = 1
print(radius > 0)            # sẽ in ra trị True
```

Thí dụ:

```
if radius < 0:
    print("Incorrect input")
else:
    area = radius * radius * 3.14159
    print("Area is", area)
```

2.Phát biểu rẽ nhánh

Phát biểu if

- Cú pháp của phát biểu **if** rẽ một nhánh như sau:

```
if boolean-expression:
    statement(s)
```

Phát biểu if rẽ hai nhánh

- Cú pháp của phát biểu **if –else** rẽ hai nhánh như sau :

```
if boolean-expression:
    statement(s)-for-the-true-case
else:
    statement(s)-for-the-false-case
```

Thí dụ:

```
if radius >= 0:
  area = radius * radius * 3.14159
  print("The area for the circle of radius", radius, "is", area)
else:
  print("Negative input")
```

3. Vòng lặp

Vòng lặp while

Cú pháp của vòng lặp **while** như sau:

while loop-continuation-condition:

  # Loop body

  Statement(s)

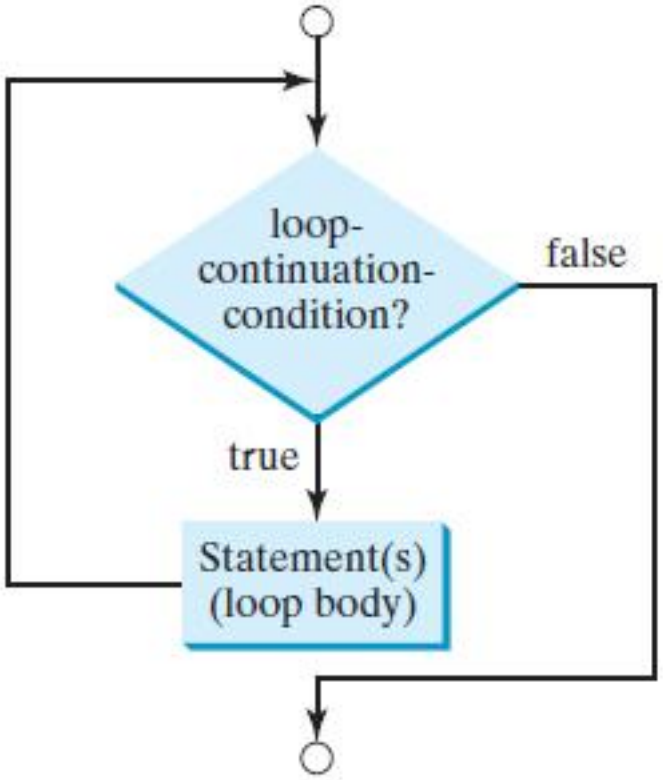
Thí dụ:

count = **0**

while count < **100**:

  print(**"Programming is fun!"**)

  count = count + **1**



Vòng lặp for

Vòng **for** của Python lặp qua một chuỗi (sequence) giá trị. Cú pháp tổng quát của vòng lặp **for** như sau:

```
for var in sequence:
    Loop body
```

Một sequence chứa nhiều phần tử dữ liệu, các phần tử được lưu kế tiếp nhau.

Hàm thư viện **range(a, b)** trả về một chuỗi (sequence) những số nguyên **a**, **a + 1**, ..., **b - 2**, và **b -1.**

Thí du: **for** v **in** range(4, 8):

```
        print(v)
4
5
6
7
```

4. Danh sách (List)

- Python cung cấp một kiểu dữ liệu được gọi là danh sách (list) để lưu một tập hợp tuần tự các phần tử.

Tạo list

- Lớp **list** dùng để định nghĩa các đối tượng list. Để tạo ra một list, ta có thể dùng hàm tạo (constructor) của lớp **list** như sau:

```
list1 = list()                  # tạo ra một list rỗng
list2 = list([2, 3, 4])         # tạo ra một list  gồm ba phần tử 2, 3, 4
list3 = list(["red", "green", "blue"])      # tạo ra một list gồm các dòng ký tự
list4 = list(range(3, 6))       # tạo ra một list  gồm ba phần tử  3, 4, 5
```

Ta cũng có thể tạp ra một list bằng cách dùng cú pháp sau:

- list1 = [] # tương tự như list()
- list2 = [2, 3, 4] # tương tự như list([2, 3, 4])
- list3 = ["red", "green"] # tương tự như list(["red", "green"])

List là kiểu dữ liệu chuỗi

Dòng ký tự và list là hai loại kiểu dữ liệu chuỗi (sequence) trong Python. Một string là một chuỗi các ký tự trong khi một list một chuỗi các phần tử bất kỳ.

Bảng 2.3 Các phép toán thông thường cho chuỗi *s*

Phép toán	*Mô tả*
• **x in s**	True nếu phần tử x có xuất hiện trong chuỗi s.
• **x not in s**	True nếu phần tử x không xuất hiện trong chuỗi s.
• **s1 + s2**	Ghép kề hai chuỗi s1 và s2.
• **s * n, n * s**	n bản sao của chuỗi s được ghép kề.
• **s[i]**	phần tử thứ i trong chuỗi s.
• **s[i : j]**	Phân đoạn của chuỗi s từ chỉ số i đến chỉ số j - 1.
• **len(s)**	Chiều dài của chuỗi s, tức là số phần tử của chuỗi s.
• **min(s)**	Phần tử nhỏ nhất trong chuỗi s.
• **max(s)**	Phần tử lớn nhất trong chuỗi s.
• **sum(s)**	Tổng các phần tử trong chuỗi s.

Thí dụ về các phép toán trên List

```
>>> list1 = [2, 3, 4, 1, 32]
>>> len(list1)
5
>>> max(list1)
32
>>> min(list1)
1
>>> sum(list1)
42
```

Một phần tử trong một list có thể được truy đạt đến nhờ chỉ số (index) với cú pháp như sau:

myList[index]

Các chỉ số của một list bắt đầu từ **0** b; tức là, chúng có trị trong tầm từ **0** đến **len(myList) - 1**.

Thí dụ:

myList = [**5.6**, **4.5**, **3.3**, **13.2**, **4.0**, **34.33**, **34.0**, **45.45**, **99.993**, **11123**]
for i **in** range(len(myList)):
    myList[i] = i

Vòng lặp gán **0** đến **myList[0]**, **1** đến **myList[1]**, ..., và **9** đến **myList[9]**.

List Slicing

- The index operator allows you to select an element at the specified index. The *slicing operator* returns a slice of the list using the syntax **list[start : end]**. The slice is a sublist from index **start** to index **end – 1**. Here are some examples:

```
>>> list1 = [2, 3, 5, 7, 9, 1]
>>> list1[2 : 4]
[5, 7]
```

- The starting index or ending index may be omitted. In this case, the starting index is **0** and the ending index is the last index. For example:

```
>>> list1 = [2, 3, 5, 2, 33, 21]
>>> list1[ : 2]
[2, 3]
```

List Slicing (tt.)

```
>>> list1[3 : ]
[2, 33, 21]
```

- You can use a negative index in slicing. For example:

```
 >>> list1 = [2, 3, 5, 2, 33, 21]
 >>> list1[1 : -3]
[3, 5]
4 >>> list1[-4 : -2]
[3, 5,2]
```

Các phương thức của list

append(x: object): None	Đính phần tử x vào cuối danh sách.
count(x: object): int	Trả về số lần mà phần tử x xuất hiện trong list.
extend(l: list): None	Đính tất cả các phần tử trong l vào list.
index(x: object): int	Trả về chỉ số của lần xuất hiện đầu tiên của phần tử x trong list.
insert(index: int, x: object):None	Chèn thêm phần tử x tại chỉ số được cho. Chú ý phần tử đầu tiên trong list có chỉ số 0.
pop(i): object	Gỡ bỏ phần tử tại vị trí được cho và trả giá trị này về. Tham số i có tính chất nhiệm ý. Nếu i không được nêu thì phần tử cuối cùng trong list sẽ được gỡ bỏ và trả về giá trị này.
remove(x: object): None	Gỡ bỏ sự xuất hiện đầu tiên của x trong list.
reverse(): None	Đảo ngược thứ tự của các phần tử trong list.
sort(): None	Sắp thứ tự các phần tử trong list

Thí dụ

```
>>> list1 = [2, 3, 4, 1, 32, 4]
>>> list1.append(19)
>>> list1
[2, 3, 4, 1, 32, 4, 19]
>>> list1.count(4)              # số lần xuất hiện của số 4
2
>>> list2 = [99, 54]
>>> list1.extend(list2)
>>> list1
[2, 3, 4, 1, 32, 4, 19, 99, 54]
>>> list1.index(4)       # trả về chỉ số của giá trị 4
2
>>> list1 = [2, 25, 3, 4, 1, 32, 4, 19, 99, 54]
>>> list1.pop()
54
>>> list1.remove(32)              # gỡ bỏ phần tử 32
```

List dựa vào biểu thức (List Comprehension)

- Tạo *list dựa vào biểu thức* là một cách gọn nhẹ để tạo ra một danh sách các phần tử. Một list dựa vào biểu thức gồm các dấu mở đóng ngoặc vuông chứa đựng một biểu thức rồi đến một mệnh đề **for** , sau đó có thể có thêm hoặc không có mệnh đề **for** hay mệnh đề **if** .
- Một list dựa vào biểu thức tạo ra một list với những phần tử được tính từ biểu thức. Sau đây là vài thí dụ:

```
>>> list1 = [x for x in range(5)]          # trả về một list gồm 0, 1, 2, 3, 4
>>> list1
[0, 1, 2, 3, 4]
>>>
>>> list2 = [0.5 * x for x in list1]
>>> list2
[0.0, 0.5, 1.0, 1.5, 2.0]
>>> list3 = [x for x in list2 if x < 1.5]
>>> list3
[0.0, 0.5, 1.0]
```

List hai chiều (Two-dimensional list)

- Dữ liệu trog một bảng hay một ma trận có thể được lưu trong một list hai chiều. Một list hai chiều là một list gồm những phần tử lại là những list một chiều.
- Một giá trị trong một ma trận có thể được truy đạt đến bằng cách dùng **matrix[i][j]**, với **i** và **j** là chỉ số hàng và chỉ số cột.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]          # matrix là một list được cho
for row in range(len(matrix)):
    for column in range(len(matrix[row])):
        print(matrix[row][column], end = " ")
    print()    # Print a new line

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    for value in row:
        print(value, end = " ")
    print() # Print a new line
```

5. Bộ (Tuple)

- Tuple cũng giống như list, nhưng những phần tử của tuple thì cố định; tức là, một khi tạo ra một tuple, ta không thể thêm phần tử mới vào, xóa hay thay thế các phần tử hoặc sắp xếp lại các phần tử trong tuple.
- Ta có thể tạp ra một tuple bằng cách liệt kê các phần tử của nó trong cặp dấu ngoặc. Các phần tử cách nhau bằng dấu phảy.

```
t1 = ()      # Tạo ra một tuple rỗng
t2 = (1, 3, 5)       # Tạo ra một tuple gồm 3 phần tử
# Tạo ra một tuple từ một list
t3 = tuple([2 * x for x in range(1, 5)])     # t3 là (2, 4, 6, 8)
# Tạo ra một tuple từ một dòng ký tự
t4 = tuple("abac")      # t4 is ('a', 'b', 'a', 'c‘)
```

- Tuple là chuỗi (sequence). Những phép toán làm việc trên chuỗi đều có thể làm việc với tuple. Ta có thể dùng các hàm **len**, **min**, **max**, và **sum** để làm việc với một tuple.

6. Tập hợp (Set)

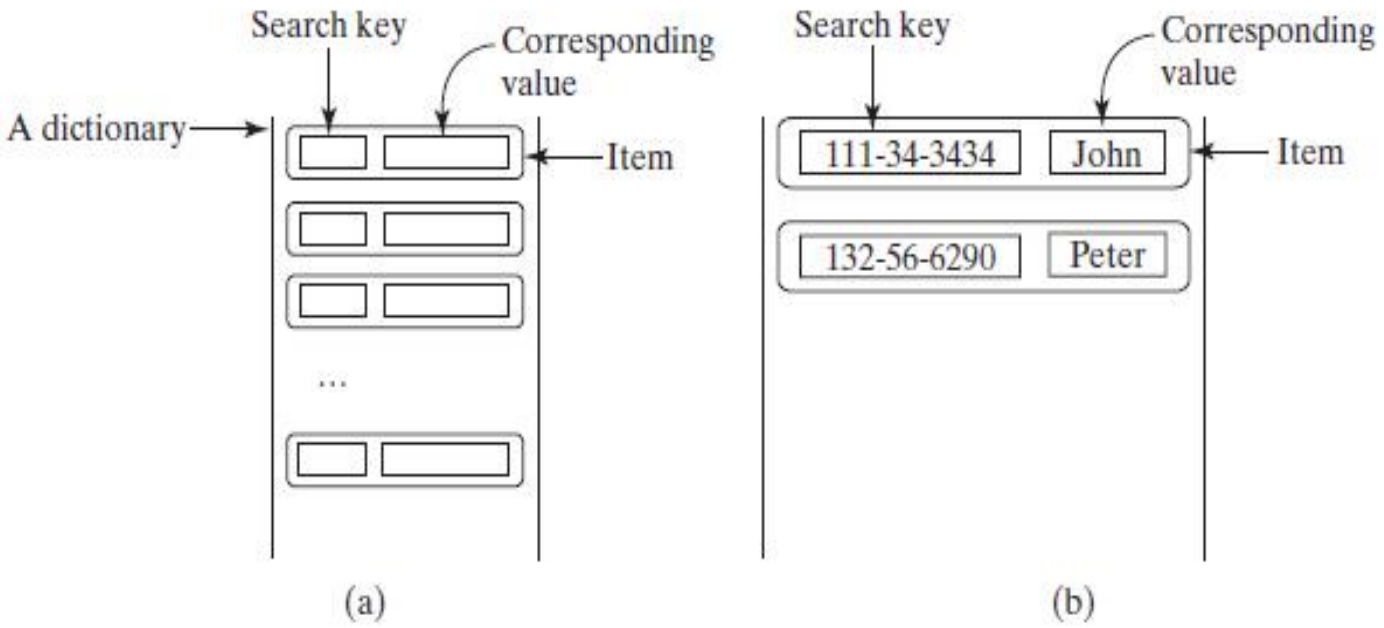
- Tập hợp cũng giống như list vì ta dùng tập hợp để chứa một tập phần tử. Tuy nhiên, khác với list, các phần tử trong một tập hợp không được trùng lặp và không được sắp xếp theo một thứ tự nào cả.
- Ta có thể tạo ra một tập hợp bằng cách đặt chúng vào trong cặp dấu ngoặc móc (**{}**). Các phần tử cách nhau bằng dấu phảy.

```
s1 = set()          # Tạo ra một tập hợp rỗng
s2 = {1, 3, 5}      # Tạo ra một tập hợp gồm 3 phần tử
s3 = set([1, 3, 5])         # Tạo ra một tập hợp từ một list
s4 = set([x * 2 for x in range(1, 10)])     # Tạo ra một tập hợp từ một list
s5 = set("abac")                    # s5 là {'a', 'b', 'c'}
```

- Ta có thể thêm vào tập hợp hoặc bớt một phần tử ra khỏi tập hợp bằng các phương thức **add(e)** hay **remove(e)** . Ta có thể dùng các hàm **len**, **min**, **max**, và **sum** với một tập hợp, và dùng vòng lặp **for** để duyệt qua tất cả các phần tử trong một tập hợp.
- Ta có thể dùng toán tử **in** hay **not in** o để xét xem một phần tử có hiện diện trong một tập hợp hay không.

7. Tự điển

- **Tự điển** (dictionary) là một đối tượng mà chứa một tập những cặp khóa/trị (key/value pair). Tự điển giúp truy xuất nhanh, loại bỏ hoặc cập nhật các giá trị nhờ vào các trị khóa.
- Các trị khóa đóng vai trò của chỉ số (index). Trong một tự điển, trị khóa phải là loại dữ liệu có thể áp dụng hàm băm được (hashable). Một tự điển không thể chứa các trị khóa trùng lặp. Một trị khóa có thể ánh xạ đến 1 giá trị.



Tạo ra một tự điển

- Ta có thể tạo ra một tự điển bằng cách đưa các phần tử là những cặp khóa/giá trị vào trong cặp dấu ngoặc móc (**{}**). Mỗi phần từ gồm một trị khóa, sau đó là một dấu :, rồi đến một giá trị thông tin. Các phần tử cách nhau bằng dấu phảy.

students = {**"111-34-3434"**:**"John"**, **"132-56-6290"**:**"Peter"**}

- Trị khóa phải thuộc loại dữ liệu cho thể áp dụng hàm băm, thí dụ số nguyên hay chuỗi ký tự (string). Giá trị thông tin thì có thể là bất kỳ loại dữ liệu nào.

```
students = {}                # Tạo ra một tự điển rỗng
```

- Để thêm một phần tử vào tự điển, ta dùng cú pháp:

```
dictionaryName[key] = value
```

Thí dụ:

students[**"234-56-9010"**] = **"Susan"**

- Nếu trị khóa đã có trong tự điển, thì phát biểu sẽ thay thế giá trị mới đối với trị khóa đó.

Biến đổi và truy xuất giá trị trong một tự điển

- Để truy xuất một giá trị thông tin, ta viết một biểu thức **dictionaryName[key]**. Nếu trị khóa nêu trong biểu thức đã có trong tự điển, giá trị thông tin tương ứng với trị khóa đó sẽ được trả về. Ngược lại, một lỗi **KeyError** sẽ được nêu ra.

```
>>> students = {"111-34-3434":"John", "132-56-6290":"Peter"}
>>> students["234-56-9010"] = "Susan"     # Thêm phần tử mới
>>> students["234-56-9010"]
"Susan"
>>> students["111-34-3434"] = "John Smith"
>>> students["111-34-3434"]
"John Smith"
>>> student["343-45-5455"]                # bị lỗi sai
```

- Để xóa một phần tử ra khỏi một tự điển, ta dùng câu lệnh del với cú pháp như sau:

  del dictionaryName[key]

```
>>> del students["234-56-9010"]
```

8. Hàm do người dùng định nghĩa

- Hàm là một tập các phát biểu mà có nhiệm vụ thực hiện một công tác nào đó.

Định nghĩa hàm

- Một định nghĩa hàm bao gồm tên hàm, các tham số và thân hàm.
- Cú pháp của một định nghĩa hàm như sau:

```
def functionName(list of parameters):
    Function body
```

- Một hàm bao gồm một tiêu đề hàm (header) và thân hàm (body). Tiêu đề hàm bắt đầu bằng từ khóa **def** keyword, sau đó là tên hàm, danh sách các tham số và kết thúc bằng dấu :.
- Các biến tại tiêu đề hàm được gọi à các *tham số hình thức* (formal parameters)
- Một số hàm trả về một giá trị, trong khi một số hàm khác thực hiện một công tác nào đó mà không cần trả về một giá trị. Nếu một hàm có trả về một giá trị, hàm ấy được gọi là *hàm trả về trị* (value-returning function)

Gọi hàm

```
def max(num1, num2):
    if num1 > num2:
        result = num1
    else:
        result = num2
return result
```

Gọi hàm

Gọi một hàm nhằm thực thi các câu lệnh trong hàm đó.

- Nếu một hàm trả về một giá trị, sự gọi hàm đó được xem như là một trị. Thí dụ,

  larger = max(**3**, **4**)

- Thí dụ:

  print(max(**3**, **4**))

Tầm vực của biến

A variable created inside a function is referred to as a *local variable*. Local variables can only be accessed within a function.

```
globalVar = 1
def f1():
   localVar = 2
   print(globalVar)
   print(localVar)

f1()
print(globalVar)
print(localVar)          # ngoài tầm vực, gây ra lỗi sai
```

Ta có thể làm cho một biến cục bộ trở thành biến toàn cục bằng cách dùng phát biểu **global** như sau.

```
x = 1
def increase():
    global x
    x = x + 1
    print(x)      # Displays 2

increase()
print(x)         # Displays 2
```

Truyền list sang hàm

Khi truyền một list sang một hàm, nội dung của list có thể bị thay đổi sau lần gọi hàm, vì list là loại đối tượng khả thay đổi (mutable object).

- Vì list là một đối tượng, việc truyền một list sang một hàm giống như việc truyền một đối tượng sang một hàm.
- Thí dụ: Hàm xuất ra các phần tử của một list:

```
def printList(lst):
    for element in lst:
        print(element)
```

- Ta có thể truyền một list khi gọi hàm này. Thí dụ:

```
lst1 = [3, 1, 2, 6, 4, 2]
printList(lst1)
```

Xử lý ngoại lệ

Xử lý ngoại lệ (exception handling) tạo điều kiện cho chương trình đối phó với trường hợp ngoại lệ và có thể tiếp tục công việc bình thường.

- Một lỗi sai xảy ra khi thực thi chương trình cũng được gọi là một *ngoại lệ*.

Cú pháp của xử lý ngoại lệ đưa vào chương trình đoạn mã mà có thể gây ra ngoại lệ trong mệnh đề **try** , như sau:

```
try:
  <body>
except <ExceptionType>:
    <handler>
```

Thí dụ:

```
while True:
    try:
       filename = input("Enter a filename: ").strip()
       infile = open(filename, "r")                # Open the file
       break
    except IOError:
       print("File " + filename + " does not exist. Try again")
```

9. Lớp

- Những đối tượng (object) thuộc về cùng một loại thường được định nghĩa bằng cách dùng một lớp (class) chung.
- Một lớp trong ngôn ngữ Python dùng các biến để lưu các thành phần dữ liệu của lớp và định nghĩa các phương thức (method) để thực hiện các tác vụ.
- Một đối tượng là một *thể hiện* (instance) của một lớp và chúng ta có thể tạo ra nhiều thể hiện của một lớp. Việc tạo ra một thể hiện của một lớp được gọi là sự *hiện thể hóa* (instantiation).

Định nghĩa lớp

- Python dùng cú pháp như sau để định nghĩa một lớp mới:

```
class ClassName:
    initializer
    methods
```

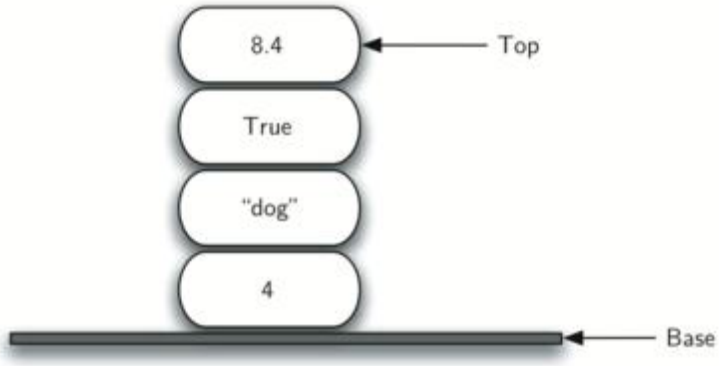
Thí dụ: Kiểu dữ liệu Stack

Kiểu dữ liệu stack được định nghĩa bằng cấu trúc và các tác vụ sau đây. Một stack được cấu trúc như là một tập phần tử có thứ tự mà khi các phần tử được thêm vào hay gỡ bỏ thì đang ở vị trí trên cùng (top) của stack. Stack được sắp thứ tự theo kiểu LIFO. Các tác vụ của lớp Stack như sau:

- **Stack()** tạo ra một stack mới và rỗng. Tác vụ này không đòi hỏi tham số và trả về một stack rỗng.
- **push(item)** thêm một phần tử mới vào vị trí top của stack. Tác vụ này cần một phần tử và không trả về giá trị nào cả.
- **pop()** gỡ bỏ phần tử ở vị trí top của stack. Tác vụ này không đòi hỏi tham số nào cả và trả về giá trị phần tử được gỡ bỏ. Stack bị thay đổi.
- **peek()** trả về phần tử ở vị trí top của stack nhưng không gỡ bỏ nó. Tác vụ này không đòi hỏi tham số nào cả . Stack không bị thay đổi.
- **is_empty()** kiểm tra xem stack có rỗng hay không. Tác vụ này không đòi hỏi tham số nào cả và trả về một giá trị bool.
- **size()** trả về số phần tử của stack. Tác vụ này không đòi hỏi tham số nào cả và trả về một giá trị nguyên.

Hiện thực Stack với ngôn ngữ Python

```
# Sự hiện thực đầy đủ của kiểu dữ liệu stack như là một lớp
class Stack:
    def __init__(self):    # phương thức initializer
        self.items = []     # items của một đối tượng stack là một list
    def is_empty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
```



Chú ý: Phương thức **init** cần có hai ký hiệu underscore đi trước và đi sau.

Tạo đối tượng

Sau khi một lớp đã được định nghĩa, ta có thể tạo ra những đối tượng từ lớp đó bằng cách gọi *hàm tạo* (constructor). Hàm tạo thực hiện 2 việc:

- tạo ra một đối tượng thuộc lớp đó.
- gọi phương thức **__init__** của lớp để gán trị khởi đầu cho đối tượng.

Mọi phương thức, kể cả phương thức khởi tạo, đều có tham số đầu tiên là **self**. Tham số này tham khảo đến đối tượng mà đang gọi phương thức hiện hành. Tham số **self** trong phương thức **__init__** tự động chỉ tới đối tượng mà vừa được tạo ra.

Phương thức initializer trong lớp Stack class sẽ dùng một list mặc định là []. Hàm tạo sau đây sẽ tạo ra một đối tượng**Stack** với một list mặc định rỗng **[]**:

```
Stack()
```

Sử dụng lớp Stack

Vài thí dụ sử dụng lớp Stack:

```
s1  = Stack()     # tạo ra một đối tượng Stack rỗng
s1 = s1.push(5)   # đưa 5 vào stack s1
s1 = s1.push(6)     # đưa 6 vào stack s1
s1= s1.push(8)     # đưa 8 vào stack s1
s1= s1.pop()        # gỡ phần tử cuối cùng ra khỏi s1
print(s1)
print(s1.peek())     # in ra phần tử 6
s1= s1.pop()         # gỡ phần tử 6 ra khỏi s1
s1= s1.pop()             # gỡ phần tử 5 ra khỏi s1
print(s1.isempty())     # in ra trị True
```

map và *lambda*

- Python có một cách đặc biệt để gọi hàm theo kiểu lặp mà không cần tạo vòng lặp. Nếu ta muốn áp dụng cùng một hàm lên mọi phần tử của một list, ta có thể dùng lệnh *map* của Python với cú pháp *map*(*function*, *list*). Lệnh này sẽ áp dụng hàm được nêu lên mọi phần tử của một list.

```
>>> items = [1,2,3,4,5]
>>> def inc(x): return x+1
...
>>> list(map(inc,items))
[2, 3, 4, 5, 6]
```

- Có một mánh lới khi hàm là một hàm vô danh là dùng lệnh **lambda** với cú pháp **lambda** *args* : *command*
- Hàm lambda chỉ thực thi trên một phát biểu, nhưng nó tạo điều kiện cho ta viết câu lệnh ngắn để thực hiện những việc phức tạp.
- Ví dụ, câu lệnh sau đây sẽ tiếp nhận một list và tính lập phương mỗi phần tử trong list và nhân với 7.

map(lambda x**:pow(**x,3**)+**7**,list)**

Lệnh filter

Một cách khác là lệnh **lambda** được dùng kết hợp với lệnh **filter**. Cách này trả về các phần tử của list mà thỏa điều kiện được nêu trong câu lệnh nằm trong phát biểu lambda.

Thí dụ

filter(lambda x**:**x**>=**2**,list)**

Trả về những phần tử trong list mà thỏa điều kiện lớn hơn hay bằng 2.

Tài liệu tham khảo:
Y. D. Liang, Introduction to Programming using Python, Pearson, 2013.

10. Thư viện NumPy

- Numpy là một trong những thư viện quan trọng nhất của Python dành cho tính toán số. Numpy hỗ trợ cho Python để làm việc với mảng nhiều-chiều (multi-dimensional array) và ma trận và thao tác nhanh trên những mảng như vậy.

ndarray của NumPy

- NumPy bao gồm hai phần quan trọng: *ndarray* and *Ufuncs* (các hàm phổ quát - universal function).
- ndarray là đối tượng mảng nhiều-chiều, là kiểu cấu trúc dữ liệu quan trọng cho nhiều tác vụ của NumPy.
- Các hàm phổ quát là những hàm làm việc trên các đối tượng ndarray theo kiểu xử lý từng phần tử một.
- Mảng hay ma trận là một trong những cách biểu diễn dữ liệu căn bản. Thông thường một mảng sẽ gồm những phần tử cùng kiểu dữ liệu và có thể có nhiều chiều. ndarray là sự tổng quát hóa loại dữ liệu như vậy.

ndarray

Thí dụ về cách tạo mảng

```
>>> import numpy as np           # import the numpy library
>>> arr = np.array([1,3,4,5,6])    # tạo ra một mảng một chiều
>>> arr
array([1, 3, 4, 5, 6])
>>> arr.shape
(5,)
>>> arr.dtype
dtype('int32')
```

Thuộc tính *shape* của một đối tượng mảng sẽ cho biết về kích thước của mảng.

Thuộc tính *dtype* của một đối tượng mảng sẽ cho biết về kiểu dữ liệu của các phần tử của mảng.

Lưu ý: tất cả mọi phần tử của mảng phải cùng một kiểu dữ liệu.

Tạo mảng

```
>>> arr = np.array([[1,2,3],[2,4,6],[8,8,8]])
>>> arr.shape
(3, 3)
>>> arr
array([[1, 2, 3],
       [2, 4, 6],
       [8, 8, 8]])
```

- Hàm *np.zeros*: tạo ra một ma trận với kích thước cho trước mà toàn chứa giá trị zero.

```
>>> arr = np.zeros((2,4))
>>> arr
array([[ 0., 0., 0., 0.],
       [ 0., 0., 0., 0.]])
```

- Hàm *np.ones*: tạo ra một ma trận với kích thước cho trước mà toàn chứa giá trị 1.

```
>>> arr = np.ones((2,4))
>>> arr
array([[ 1., 1., 1., 1.],
       [ 1., 1., 1., 1.]])
```

Tạo mảng (tt.)

Hàm np.identity tạo ra ma trận đơn vị với một kích thước được cho:

```
>>> arr = np.identity(3)
>>> arr
array([[ 1., 0., 0.],
       [ 0., 1., 0.],
        [ 0., 0., 1.]])
```

- Ta cũng có thể tạo ra một ma trận với kích thước cho trước mà chứa những trị ngẫu nhiên. Điều này được thực hiện bằng cách dùng hàm *randn* trong gói *numpy.random* :

```
>>> arr = np.random.randn(3,4)
>>> arr
array([[ 0.0102692 , -0.13489664, 1.03821719, -0.28564286],
       [-1.12651838, 1.41684764, 1.11657566, -0.1909584 ],
       [ 2.20532043, 0.14813109, 0.73521382, 1.1270668 ]])
```

The function np.arrange()

Hàm *np.arrange*() tạo ra một mảng chứa những giá trị được xác định, một phiên bản của hàm *range*() dành cho các đối tượng dữ liệu array.

Thí dụ,

```
>>> a = np.arrange(5)
>>> a
array([0, 1, 2, 3, 4]

>>> arr = np.arrange(3,7,2)
>>> arr
array([3, 5])

>>> b = np.arrange(12).reshape(3,4)        # a matrix with size 3 × 4
>>> b
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [8, 9, 10, 11])
```

Truy đạt đến phần tử của mảng

- Numpy cung cấp một số cách để truy đạt đến phần tử của mảng.

Đánh chỉ số căn bản

```
>>> arr = np.array([[1,2,3],[2,4,6],[8,8,8]])
>>> arr[0]
array([1, 2, 3])
>>> arr = np.arrange(12).reshape(2,2,3)     # hai ma trận có kích thước 2×3
>>> arr
array([[[ 0, 1, 2],
        [ 3, 4, 5]],
       [[ 6, 7, 8],
        [ 9, 10, 11]]])
>>> arr[0]
array([[0, 1, 2],
       [3, 4, 5]])
```

Đánh chỉ số bằng số nguyên

Đánh chỉ số bằng số nguyên (integer array indexing)

```
>>>: arr = np.arange(9).reshape(3,3)
>>>arr
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> arr[[0,1,2],[1,0,0]]
array([1, 3, 6])
```

Trong thí dụ này, chúng ta đã cung cấp một mảng trong đó phần thứ nhất xác định vị trí hàng mà ta muốn truy đạt và phần thứ hai xác định vị trí cột mà ta muốn truy đạt.

Các tác vụ trên mảng

Thí dụ:

```
>>> arr = np.arange(15).reshape(3,5)
>>> arr
array([[ 0, 1, 2, 3, 4],
       [ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14]])
>>> arr + 5
array([[ 5, 6, 7, 8, 9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> arr * 2
array([[ 0, 2, 4, 6, 8],
       [10, 12, 14, 16, 18],
       [20, 22, 24, 26, 28]])
```

Ta không cần dùng vòng **for** để thực hiện hai công tác trên

Các tác vụ trên mảng (tt.)

```
>>> arr1 = np.arange(15).reshape(5,3)
>>>arr1
array([[ 0, 1, 2],
       [ 3, 4, 5],
       [ 6, 7, 8],
       [ 9, 10, 11],
       [12, 13, 14]])
>>> arr2 = np.arange(5).reshape(5,1)
array([[0],
       [1],
       [2],
       [3],
       [4]])
>>> arr2 + arr1
array([[ 0, 1, 2],
       [ 4, 5, 6],
       [ 8, 9, 10],
       [12, 13, 14],
       [16, 17, 18]])
```

Đại số tuyến tính với numpy

Bây giờ, chúng ta học cách thực hiện các tính toán đại số tuyến tính trên các ndarray bằng cách dùng những hàm đã có sẵn trong gói numpy.

Hàm add

```
>>> a = np.arrange(4).reshape(2,2)
>>> a
array([[0, 1],
          [2, 3])
>>> b = np.arrange(3, 7).reshape(2,2)
>>>b
array([[3, 4],
          [5, 6])
>>> c= np.add(a, b)          # tương đương với c = a + b    cọng ma trận
>>> c
array([[3, 5],
         [7, 9])
```

Hàm *multiply* và tích ma trận (dot product)

Hàm *multiply* (nhân từng phần tử với nhau)

```
>>>a = np.array([[0, 1],[2,3])
>>>b = np.array([[3, 4],[5,6])
>>> d = np.multiply(a, b)
>>> d
array([[0, 4],
       [10, 18])
```

Tích ma trận (nhân matrận)

```
>>> A = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> B = np.array([[9,8,7],[6,5,4],[1,2,3]])
>>> A.dot(B)                    # tương đương với np.dot(A, B)
array([[ 24, 24, 24],
       [ 72, 69, 66],
       [120, 114, 108]])
```

Chuyển vị ma trận

- Hàm T

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

Giải hệ phương trình tuyến tính

Đại số tuyến tính thường được dùng để giải hệ phương trình tuyến tính. Dùng lối ký hiệu ma trận để diễn tả hệ phương trình và một hàm có sẵn của numpy, ta có thể giải được hệ phương trình một cách dễ dàng. Cho một hệ phương trình như sau:

$7x + 5y - 3z = 16$

$3x - 5y + 2z = -8$

$5x + 3y - 7z = 0$

Hệ phương trình có thể được biểu diễn bằng hai ma trận: một ma trận hệ số (gọi tên là a trong thí dụ này) và một vector hằng số (gọi tên là b).

```
>>> a = np.array([[7,5,-3], [3,-5,2],[5,3,-7]])
>>> b = np.array([16,-8,0])
>>> x = np.linalg.solve(a, b)
```

Giải hệ phương trình tuyến tính (tt.)

```
>>> x
array([ 1., 3., 2.])
```

- Ta có thể kiểm tra lời giải có đúng không, bằng cách dùng hàm *np.allclose*.

```
>>> np.allclose(np.dot(a, x), b)
True
```

- **Nghịch đảo ma trận vuông**

  np.linalg.inv(a)

- **Tính định thức (determinant) của một ma trận vuông**

  np.linalg.det(a)