CHƯƠNG 1. GIỚI THIỆU

1.1. Tạo và thực thi một chương trình Python

Mã Python có thể được viết bằng bất kỳ trình soạn thảo văn bản thuần túy nào có thể tải và lưu văn bản bằng cách sử dụng bảng mã ký tự ASCII hoặc UTF-8 Unicode. Mặc định, các tệp Python được sử dụng mã hóa ký tự UTF-8. Các tệp Python thường có phần mở rộng là `.py`, và các chương trình Python GUI (Giao diện Người dùng Đồ họa) thường có phần mở rộng là `.pyw`, đặc biệt là trên Windows và Mac OS X. Trong tài liệu này, tôi luôn sử dụng phần mở rộng của `.py` cho các chương trình console và module, và .pyw cho các chương trình GUI.

Chỉ để đảm bảo rằng mọi thứ được thiết lập chính xác và để hiển thị ví dụ cổ điển đầu tiên, hãy tạo một tệp có tên `hello.py` trong trình soạn văn bản thuần túy (Windows Notepad), với nội dung sau:

```
#!/usr/bin/env python3

print("Hello", "World!")
```

Dòng đầu tiên là chú thích, trong Python chú thích bắt đầu bằng dấu `#`. Dòng thứ hai là dòng trống, Python sẽ bỏ qua các dòng trống trong chương trình. Dòng thứ ba là mã lệnh Python, ở ví dụ trên là gọi hàm `print` với hai tham số, mỗi tham số là một kiểu chuỗi.

Mỗi câu lệnh xuất hiện trong file `.py` sẽ được thực hiện tuần tự, bắt đầu từ câu lệnh đầu tiên đến câu lệnh cuối cùng.

Giả sử lưu chương trình Python trong thư mục C:\Py3eg, với tên file `hello.py` và đóng chương trình soạn thảo. Để thực thi chương trình, ta có thể sử dụng chương trình trình thông dịch Python và thông thường điều này được thực hiện bên trong cửa sổ lệnh (Command Prompt). Khởi động cửa sổ lệnh và nhập phần:

```
C:\>cd c:\py3eg
C:\py3eg\>c:\python31\python.exe hello.py
```

Kết quả xuất hiện trên màn hình:

```
Hello World!
```

1.2. Các điểm nổi bật của Python

1.2.1. Kiểu dữ liệu

Một điều cơ bản mà bất kỳ ngôn ngữ lập trình nào cũng phải có đó là biểu diễn dữ liệu. Python cung cấp một số kiểu dữ liệu tích hợp sẵn, nhưng chúng ta sẽ tìm hiểu hai kiểu cơ bản sau. Python biểu diễn các số nguyên (số nguyên dương và âm) bằng cách sử dụng kiểu `int` và biểu diễn các chuỗi (chuỗi ký tự Unicode) sử dụng kiểu `str`. Ví dụ:

```
-973
21062458333711437339583605536734086463779019080109822250862195507 20
"Infinitely Demanding"
'Simon Critchley'
'positively αβγ ÷©'
''
```

Số thứ hai được hiển thị là 2217, kích thước số nguyên của Python chỉ bị giới hạn bởi bộ nhớ máy, không phải bởi một số byte cố định. Các chuỗi có thể được xác định bằng dấu ngoặc kép hoặc đơn, miễn là cùng một loại được sử dụng ở cả hai đầu và vì Python sử dụng Unicode, các chuỗi không bị giới hạn ở các ký tự ASCII, như chuỗi gần cuối thể hiện.

Python sử dụng dấu ngoặc vuông (`[]`) để truy cập một kí tự từ một chuỗi. Ví dụ: nếu chúng ta đang ở trong Python Shell (trong trình thông dịch tương tác hoặc trong IDLE), chúng ta có thể nhập như sau:

```
>>> "Hard Times"[5]
'T'
>>> "giraffe"[0]
'g'
```

Trong Python, cả `str` và các kiểu số cơ bản như `int` đều không thay đổi được, nghĩa là, sau khi được đặt, giá trị của chúng không thể thay đổi, chúng ta không thể sử dụng chúng để đặt một ký tự mới.

Để chuyển đổi dữ liệu từ kiểu này sang kiểu khác, chúng ta có thể sử dụng cú pháp `datatype(item)`. Ví dụ:

```
>>> int("45")
45
>>> str(912)
```

```
'912'
```

Chuyển đổi `int()` chấp nhận khoảng trắng ở đầu và cuối, vì vậy int (" 45 ") cũng sẽ hoạt động tốt. Chuyển đổi `str()` có thể được áp dụng cho hầu hết mọi mục dữ liệu. Nếu một chuyển đổi không thành công, Python sẽ phát sinh ngoại lệ.

1.2.2. Tham chiếu đối tượng

Khi có một số kiểu dữ liệu, thì cần có các biến để lưu trữ chúng. Python không có các biến như vậy, nhưng thay vào đó có các tham chiếu đối tượng.

Xét ví dụ đơn giản sau:

```
x = "blue"
y = "green"
z = x
```

Cú pháp đơn giản là `objectReference = value`. Không cần khai báo trước và không cần chỉ định loại của giá trị. Khi Python thực thi câu lệnh đầu tiên, nó tạo một đối tượng `str` với nội dung là `"blue"` và tạo một tham chiếu đối tượng được gọi là `x` tham chiếu đến đối tượng `str`. Chúng ta có thể nói rằng biến `x` đã được gán là chuỗi `"blue"`. Câu lệnh thứ hai cũng tương tự. Câu lệnh thứ ba tạo một tham chiếu đối tượng mới được gọi là `z` và đặt nó tham chiếu đến cùng một đối tượng mà tham chiếu đối tượng `x` tham chiếu đến (trong trường hợp này là str chứa văn bản `"blue"`).

Tiếp tục với ví dụ `x`, `y`, `z`:

```
print(x, y, z) # in ra: blue green blue
z = y
print(x, y, z) # in ra: blue green green
x = z
print(x, y, z) # in ra: green green green
```

Sau câu lệnh thứ tư (`x = z`), cả ba tham chiếu đối tượng đều tham chiếu đến cùng một `str`. Vì không còn tham chiếu đối tượng nào nữa đến chuỗi `"blue"`, Python có thể giải phóng bộ nhớ cho nó.

CHƯƠNG 2. KIỂU DỮ LIỆU VÀ THAO TÁC NHẬP/XUẤT

2.1. Định danh và từ khóa

Định danh trong Python là chuỗi kí tự khác rỗng có độ dài bất kì bắt đầu bởi một chữ cái sau đó có thể là chữ cái hoặc chữ số, dấu gạch dưới, … các kí tự trong bộ mã unicode. Định danh có phân biệt chữ hoa, chữ thường. Định danh không được trùng với các từ khóa sau:

```
and       continue  except    global    lambda    pass      while
as        def       False     if        None      raise     with
assert    del       finally   import    nonlocal  return    yield
break     elif      for       in        not       True
class     else      from      is        or        try
```

2.2. Số nguyên

2.2.1. Kiểu int

Kích thước của một số nguyên chỉ bị giới hạn bởi bộ nhớ của máy, vì vậy, có thể dễ dàng tạo và làm việc với các số nguyên dài hàng trăm chữ số, mặc dù chúng sẽ chậm sử dụng hơn so với các số nguyên truyền thống.

Bảng 2.1. *Các phép toán và hàm trên kiểu số.*

Cú pháp	Mô tả
`x + y`	Cộng số x và số y
`x - y`	Trừ số x cho y
`x * y`	x nhân y
`x / y`	Chia x cho y; sinh ra số thực hoặc số phức (nếu x hoặc y là số phức)
`x // y`	Chia x cho y, làm tròn phần thập phân vì vậy kết quả luôn là số nguyên
`x % y`	Chia lấy phần dư x cho y
`x ** y`	x mũ y
`-x`	Đổi dấu
`+x`	Không làm gì
`abs(x)`	Trị tuyệt đối của x

`divmod(x, y)`	Trả về thương số và phần dư của phép chia x cho y (một bộ 2 số nguyên)
`pow(x, y)`	Tính x mũ y (giống phép toán **)
`pow(x, y, z)`	Giống như (x ** y) % z
`round(x, n)`	Trả về số x đã làm tròn đến n chữ số. n âm làm tròn sau dấu thập phân, n dương làm tròn trước dấu thập phân. Giá trị trả về cùng kiểu với x

Bảng 2.2. *Hàm chuyển đổi cơ số.*

Cú pháp	Mô tả
`bin(i)`	Trả về biểu diễn nhị phân của số nguyên i (dạng chuỗi str)
`hex(i)`	Trả về biểu diễn thập lục phân của số nguyên i (dạng chuỗi)
`int(x)`	Chuyển một đối tượng x sang số nguyên.
`int(s, base)`	Chuyển một chuỗi s sang số nguyên với cơ số base (2-36)
`oct(i)`	Trả về biểu diễn bát phân của số nguyên i (dạng chuỗi)

Tất cả các phép toán hai ngôi (+, -, /, //, % và **) đều có phiên bản gán tăng cường (+=, -=, /=, //=, %= và **=) trong đó `x op= y` tương đương với `x = x op y`.

2.2.2. Kiểu bool

Python phiên bản mới có định nghĩa kiểu `bool` với hai giá trị `True` và `False`. Tuy nhiên những phiên bản trước đó coi giá trị 0 là `False` và khác 0 là `True`. Kiểu logic cũng có 3 phép toán cơ bản: `and`, `or`, `not`.

2.3. Số thực

Python cung cấp ba loại giá trị dấu phẩy động: loại `float`, `complex` và `Decimal`.

2.3.1. Kiểu số thực dấu phẩy động (float)

Tất cả các toán tử và hàm số trong Bảng 2.1 có thể được sử dụng với `float`, bao gồm cả các phiên bản phép gán tăng cường.

Bảng 2.3. *Các hàm toán học kiểu số trong module math.*

Cú pháp	Mô tả
`math.acos(x)`	Trả về arc cosine của x (radians)
`math.acosh(x)`	Trả về arc hyperbolic cosine của x (radians)
`math.asin(x)`	Trả arc sine của x (radians)
`math.asinh(x)`	Trả về arc hyperbolic sine của x (radians)
`math.atan(x)`	Trả về arc tangent của x (radians)
`math.atan2(y, x)`	Trả về arc tangent của y / x (radians)
`math.atanh(x)`	Trả về arc hyperbolic tangent của x (radians)
`math.ceil(x)`	Trả về số nguyên nhỏ nhất lớn hơn hoặc bằng x; ví dụ: math.ceil(5.4) == 6

Cú pháp	Mô tả
`math.copysign(x,y)`	Trả về số x với dấu của y
`math.cos(x)`	Trả về cosine của x (radians)
`math.cosh(x)`	Trả về hyperbolic cosine của x (radians)
`math.degrees(r)`	Đổi số thực r từ radians sang độ
`math.e`	Hằng số e; xấp xĩ 2.7182818284590451
`math.exp(x)`	Trả về ex, i.e., math.e ** x
`math.fabs(x)`	Trả về trị tuyệt đối x i.e. (float)
`math.factorial(x)`	Trả về x!
`math.floor(x)`	Trả về số nguyên lớn nhất nhỏ hơn hoặc bằng x; ví dụ: math.floor(5.4) == 5
`math.fmod(x, y)`	Trả về phần dư (số thực) của phép chia x/y
`math.frexp(x)`	Trả về hai giá trịReturns a 2-tuple with the mantissa (as a float) and the exponent (as an int) so, x = m × 2e; see math.ldexp()
`math.fsum(i)`	Trả về tổng các giá trị trong danh sách i (float)
`math.hypot(x, y)`	$\sqrt{x^2 + y^2}$
`math.isinf(x)`	Trả về True nếu số thực x là ± ∞
`math.isnan(x)`	Trả về True nếu số thực x là không phải số (“not a number”)
`math.ldexp(m, e)`	Trả về m × 2e; ngược với hàm math.frexp()
`math.log(x, b)`	Trả về logbx; b tùy chọn (ngầm định là số e)
`math.log10(x)`	Trả về log10x
`math.log1p(x)`	Trả về loge(1+ x)
`math.modf(x)`	Trả về phần thực và nguyên của số thực x
`math.pi`	Hằng số π; xấp xĩ 3.1415926535897931
`math.pow(x, y)`	Trả về xy (float)
`math.radians(d)`	Chuyển số thực d từ độ sang radians
`math.sin(x)`	Trả về sine của x (radians)
`math.sinh(x)`	Trả về hyperbolic sine của x (radians)
`math.sqrt(x)`	Trả về $\sqrt{x}$
`math.tan(x)`	Trả về tangent của x (radians)
`math.tanh(x)`	Trả về hyperbolic tangent của x (radians)
`math.trunc(x)`	Trả về phần nguyên của x

Để sử dụng các hàm này, yêu cầu phải: `import math`.

2.3.2. Kiểu số phức (complex)

Kiểu dữ liệu số phức là kiểu bất biến chứa một cặp số thực, một phần biểu diễn phần thực và phần còn lại là phần ảo của một số phức. Hằng số phức được viết với phần thực và phần ảo được nối bằng dấu + hoặc – và với phần ảo theo sau là kí hiệu `j`. Dưới đây là một số ví dụ: `3.5 + 2j`, `0.5j, 4 + 0j`, `–1–3.7j`. Lưu ý rằng nếu phần thực là

0, chúng ta có thể bỏ qua. Mỗi đối tượng số phức luôn có hai thuộc tính là `real` và `imag` để lưu phần thực và ảo.

Ngoại trừ phép toán `//`, `%`, `divmod()` và hàm ba đối số `pow()`, tất cả các phép toán và hàm số trong số thực `float` đều có thể được sử dụng với các số phức và các phiên bản gán tăng cường cũng vậy. Ngoài ra, số phức có một phương thức `conjugate()` đổi dấu phần ảo.

Để sử dụng các hàm trong kiểu số phức, ta sử dụng: `import cmath`.

2.3.3. Số thập phân (decimal.Decimal)

Trong thực tế, số thực `float` có độ chính xác kém (số chữ số sau dấu thập phân ít, khoảng 16 chữ số). Khi cần số thực có độ chính xác cao hơn, ta sử dụng module `decimal` để tạo ra đối tượng `Decimal`.

Để tạo đối tượng `Decimal`, ta thực hiện các câu lệnh sau:

```
>>> import decimal
>>> a = decimal.Decimal(9876)
>>> b = decimal.Decimal("54321.012345678987654321")
>>> a + b
Decimal('64197.012345678987654321')
```

Số thập phân được tạo bằng cách sử dụng hàm `decimal.Decimal()`. Hàm này có thể nhận một số nguyên hoặc một đối số chuỗi, nhưng không phải là một số `float`. Nếu một chuỗi được sử dụng, nó có thể sử dụng ký hiệu thập phân đơn giản hoặc ký hiệu hàm mũ. Ngoài việc cung cấp độ chính xác, sự biểu diễn chính xác của số thập phân. Các số thập phân có thể được so sánh chính xác hơn.

Tất cả các toán tử và hàm số được liệt kê trong Bảng 2.1, bao gồm cả các phiên bản gán tăng cường, có thể được sử dụng với đối tượng `decimal.Decimal`, nhưng có một số lưu ý. Nếu toán tử ** có toán hạng bên trái `decimal.Decimal` thì toán hạng bên phải của nó phải là số nguyên. Tương tự, nếu đối số đầu tiên của hàm `pow()` là `decimal.Decimal`, thì đối số thứ hai và thứ ba tùy chọn của nó phải là số nguyên.

2.4. Kiểu chuỗi (str)

Các chuỗi được biểu diễn bằng kiểu dữ liệu `str` chứa một chuỗi các ký tự Unicode. Kiểu dữ liệu `str` có thể được gọi như một hàm để tạo các đối tượng chuỗi, không có đối số, nó trả về một chuỗi rỗng, với đối số không phải chuỗi (`nonstring`), nó trả về dạng chuỗi của đối số và với đối số chuỗi, nó trả về một bản sao của chuỗi. Hàm `str()` cũng có thể được sử dụng như một hàm chuyển đổi, trong trường hợp đó, đối số đầu tiên phải

là một chuỗi hoặc một cái gì đó có thể chuyển đổi thành một chuỗi.

Các chuỗi kí tự được tạo bằng cách sử dụng dấu nháy kép hoặc dấu nháy đơn miễn là giống nhau ở cả hai đầu. Ngoài ra, chúng ta có thể sử dụng một chuỗi ba dấu nháy kép. Ví dụ:

```
text = """A triple quoted string like this can include 'quotes' and
"quotes" without formality. We can also escape newlines \
so this particular string is actually only two lines long."""
```

Bảng 2.4. *Các kí tự đặc biệt trong Python.*

Kí tự	Ý nghĩa
\newline	Tạo dòng mới
\\	Dấu Backslash (\)
\'	Dấu nháy đơn (’)
\"	Dấu nháy kép (")
\a	ASCII bell (BEL)
\b	ASCII backspace (BS)
\f	ASCII formfeed (FF)
\n	ASCII linefeed (LF)
\N{name}	Kí tự Unicode với tên
\ooo	Kí tự với giá trị số ở hệ 8
\r	ASCII carriage return (CR)
\t	ASCII tab (TAB)
\uhhhh	Kí tự Unicode với giá trị hệ 16 độ dài 16-bit
\Uhhhhhhhh	Kí tự Unicode với giá trị hệ 16 độ dài 32-bit
\v	ASCII vertical tab (VT)
\xhh	Kí tự với giá trị hệ 16 độ dài 8-bit

Nếu chúng ta muốn viết một chuỗi ký tự dài trải dài trên hai hoặc nhiều dòng nhưng không sử dụng chuỗi ba dấu ngoặc kép, chúng ta có thể thực hiện một số cách sau:

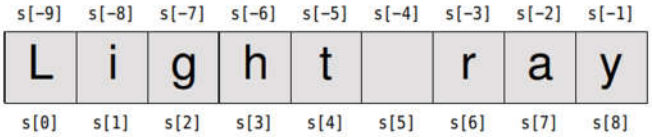
```
t = "This is not the best way to join two long strings " + \
"together since it relies on ugly newline escaping"
s = ("This is the nice way to join two long strings "
"together; it relies on string literal concatenation.")
```

Vì các tệp .py được mặc định sử dụng bảng mã UTF-8 Unicode, chúng ta có thể viết bất kỳ ký tự Unicode nào trong chuỗi ký tự bằng cách sử dụng kí tự \ như sau:

```
>>> euros = "€ \N{euro sign} \u20AC \U000020AC"
>>> print(euros)
>>> € € € €
```

2.4.2. Cắt chuỗi

Chuỗi trong Python được đánh số từ 0 đến hết chuỗi. Nhưng cũng có thể sử dụng chỉ số âm với kí tự chuỗi cùng (bên phải) có chỉ số là `-1` đến kí tự đầu tiên. Ví dụ `s="Light ray"`.



Để cắt chuỗi ta có 3 cách sau:

```
seq[start]
seq[start:end]
seq[start:end:step]
```

Trong đó `start` vị trí bắt đầu, `end` vị trí kết thúc, `step` bước nhảy. Nếu bước nhảy là số âm thì cắt chuỗi từ phải qua trái.

2.4.3. Phương thức và phép toán trên chuỗi

Trên chuỗi có các phép toán: `in` (kiểm tra xuất hiện), `+` (ghép chuỗi), `+=` (ghép vào cuối), `*` (sao chép) và `*=` (sao chép tăng cường).

Bảng 2.5. *Các phương thức trên chuỗi.*

Tên	Ý nghĩa
`s.capitalize()`	Trả về bản sao của chuỗi s với kí tự đầu là chữ hoa.
`s.center(width, char)`	Trả về bản sao của s được căn giữa trong một chuỗi có chiều dài chiều rộng được đệm bằng dấu cách hoặc tùy chọn bằng char (chuỗi có độ dài 1)
`s.count(t, start, end)`	Trả về số lần xuất hiện của chuỗi t trong chuỗi s (hoặc trong phạm vi start:end của s)
`s.encode(encoding, err)`	Trả về một đối tượng byte biểu diễn cho chuỗi bằng cách sử dụng mã hóa mặc định hoặc sử dụng mã hóa được chỉ định và xử lý lỗi theo đối số err tùy chọn
`s.endswith(x, start, end)`	Trả về True nếu s (hoặc đoạn start:end của s) kết thúc bằng str x, ngược lại trả về False.
`s.expandtabs(size)`	Trả về bản sao của s với các tab được thay thế bằng dấu cách theo bội số của 8 hoặc kích thước nếu được chỉ định
`s.find(t, start, end)`	Trả về vị trí tận cùng bên trái của t trong s (hoặc trong phần start: end của s) hoặc -1 nếu không tìm thấy. Sử dụng `str.rfind()` để tìm vị trí ngoài cùng bên phải.

Tên	Ý nghĩa
`s.format(...)`	Trả về bản sao của s được định dạng theo các đối số đã cho.
`s.index(t, start, end)`	Trả về vị trí tận cùng bên trái của t trong s (hoặc trong phần bắt đầu: kết thúc của s) hoặc phát sinh ValueError nếu không tìm thấy. Sử dụng str.rindex() để tìm từ bên phải
`s.isalnum()`	Trả về True nếu s không rỗng và mọi ký tự trong s đều là chữ và số
`s.isalpha()`	Trả về True nếu s không rỗng và mọi ký tự trong s đều là chữ cái
`s.isdecimal()`	Trả về True nếu s không rỗng và mọi ký tự trong s là chữ số hệ 10 (Unicode)
`s.isdigit()`	Trả về True nếu s không rỗng và mọi ký tự trong s là một chữ số (ASCII)
`s.isidentifier()`	Trả về True nếu s không rỗng và là một định danh hợp lệ
`s.islower()`	Trả về True nếu s có ít nhất một ký tự có thể viết thường và tất cả các ký tự có thể viết thường của nó đều là ký tự viết thường.
`s.isnumeric()`	Trả về True nếu s không rỗng và mọi ký tự trong s là ký tự (Unicode) số như: chữ số hoặc phân số
`s.isprintable()`	Trả về True nếu s rỗng hoặc nếu mọi ký tự trong s được coi là có thể in được, bao gồm khoảng trắng, nhưng không phải dòng mới.
`s.isspace()`	Trả về True nếu s không rỗng và mọi ký tự trong s là khoảng trắng
`s.istitle()`	Trả về True nếu s là một chuỗi tiêu đề không rỗng
`s.isupper()`	Trả về True nếu chuỗi s có ít nhất một ký tự có thể viết hoa và tất cả các ký tự có thể viết hoa của nó đều là chữ hoa;
`s.join(seq)`	Trả về phần nối mọi mục trong tập hợp seq.
`s.ljust(width, char)`	Trả về bản sao của s được căn trái trong chuỗi có độ dài width được đệm bằng dấu cách hoặc tùy chọn bằng char (chuỗi có độ dài 1).
`s.lower()`	Trả về bản sao viết thường của s;
`s.maketrans()`	Companion of str.translate(); see text for details
`s.partition(t)`	Trả về một bộ ba phần, phần của s trước t, phần chuỗi t, phần của s sau t.
`s.replace(t, u, n)`	Trả về bản sao của s với mọi (hoặc tối đa n nếu cho trước) lần xuất hiện của chuỗi t được thay thế bằng chuỗi u

Tên	Ý nghĩa
`s.split(t, n)`	Trả về danh sách các chuỗi phân tách nhiều nhất n lần dựa trên chuỗi t; nếu không được đưa ra chuỗi t, phân tách dựa trên khoảng trắng.
`s.splitlines(f)`	Trả về danh sách các dòng được tạo ra bằng cách tách s trên các đầu cuối dòng, loại bỏ các đầu cuối trừ khi f là True
`s.startswith(x, start, end)`	Trả về True nếu s (hoặc phần start:end của s) bắt đầu bằng chuỗi x hoặc với bất kỳ chuỗi nào trong bộ tuple x;
`s.strip(chars)`	Trả về bản sao của s với khoảng trắng ở đầu và cuối (hoặc các ký tự trong char) bị xóa; str.lstrip() chỉ xóa ở đầu và str.rstrip() ở cuối
`s.swapcase()`	Trả về bản sao của s với các ký tự viết hoa sẽ viết thường và các ký tự viết thường được viết hoa;
`s.title()`	Trả về bản sao của s trong đó chữ cái đầu tiên của mỗi từ là chữ hoa và tất cả các chữ cái khác là chữ thường.
`s.translate()`	Companion of str.maketrans(); see text for details
`s.upper()`	Trả về bản sao viết hoa của s;
`s.zfill(w)`	Trả về một bản sao của s, nếu ngắn hơn w sẽ được đệm bằng các số 0 ở đầu để làm cho nó dài w

2.4.4. Định dạng chuỗi bằng phương thức str.format()

Phương thức `str.format()` trả về một chuỗi mới với các trường thay thế trong chuỗi được thay thế bằng các đối số được định dạng phù hợp. Ví dụ:

```
>>> "The novel '{0}' was published in {1}".format("Hard Times",
1854)
"The novel 'Hard Times' was published in 1854"
```

Mỗi trường thay thế được xác định bằng một tên trường trong dấu ngoặc nhọn. Nếu tên trường là một số nguyên, nó được coi là vị trí chỉ mục của một trong các đối số được truyền đến `str.format()`.Vì vậy, trong trường hợp này, trường có tên 0 được thay thế bằng đối số đầu tiên và trường có tên 1 được thay thế bằng đối số thứ hai. Trường thay thế có cú pháp sau đây:

```
{field_name}
{field_name!conversion}
{field_name:format_specification}
{field_name!conversion:format_specification}
```

2.4.5. Định dạng chuỗi bằng kí hiệu %

Cú pháp: `"…..%c.....%d...." % (values)`. Trong đó `values` là giá trị cần định dạng, các kí tự định dạng kiểu dữ liệu sau dấu `%` như sau:

Kí tự	Ý nghĩa
`s`	Chuỗi kí tự
`c`	Kí tự hoặc int
`d`	Số thập phân
`i`	Số nguyên
`u`	Số nguyên không dấu
`o`	Số nguyên không dấu hệ 8
`x`	Số nguyên không dấu hệ 16
`X`	Giống x nhưng kí tự hiển thị chữ hoa
`e`	Số thực dấu phẩy động
`E`	Giống e, nhưng kí tự hoa
`f`	Số thực dấu phẩu động (decimal)
`F`	Giống f, nhưng kí tự chữ hoa
`%`	Hiện dấu %

2.5. Nhập xuất dữ liệu

2.5.1. Nhập dữ liệu từ bàn phím hàm input

Để nhập dữ liệu từ bàn phím, ta sử dụng hàm `input` với cú pháp như sau:

```
Name = input("Dòng thông báo")
```

Khi hàm thực hiện, nó hiện ra một dòng thông báo và chờ người dùng nhập vào một chuỗi dữ liệu và trả về chuỗi dữ liệu đó. Đề nhận được kiểu dữ liệu mong muốn trong lúc nhập, ta phải chuyển đổi kiểu dữ liệu bằng các hàm tương ứng ví dụ như: `int`, `float`, `complex`, `decimal.Decimal`, …

2.5.2. In dữ liệu ra màn hình

Ta có thể in dữ liệu ra màn hình bằng hàm `print` với cú pháp như sau:

```
print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout])
```

Trong đó `object` là đối tượng cần in, `sep` là kí tự ngăn cách giữa các đối tượng (ngầm định là dấu cách), `end` là kí tự xuất hiện ở cuối dòng in (ngầm định là dấu xuống

dòng), `file` chỉ ra đối tượng file để in giá trị đến (ngầm định là màn hình).

2.6. Ví dụ

Ví dụ 2.1. *Phương trình bậc hai là phương trình có dạng ax2 + bx + c = 0 trong đó a ≠ 0. Nghiệm của các phương trình được tính từ công thức* $x = \frac{-b^2 \pm \sqrt{b^2-4ac}}{2a}$. *Trong đó Δ=b2-4ac nếu dương có 2 nghiệm thực, nếu bằng 0 có 1 nghiệm thực và nếu nhỏ hơn không có 2 nghiệm phức. Viết chương trình nhập ba số thực a, b, c, tìm nghiệm của phương trình.*

```
import math
import cmath
a = float(input('Nhập hệ số a: '))
b = float(input('Nhập hệ số b: '))
c = float(input('Nhập hệ số c: '))
d = b**2 - 4*a*c
x1 = None
x2 = None
if d==0:
    x1 = -(b/(2*a))
else:
    if d>0:
        root = math.sqrt(d)
    else:
        root = cmath.sqrt(d)
    x1 = (-b-root)/(2*a)
    x2 = (-b+root)/(2*a)
equation = ("phương trình {0}x\N{SUPERSCRIPT TWO} + {1}x + {2} = 0
có nghiệm x = {3:.3}").format(a, b, c, x1)
if x2 is not None:
    equation += " hoặc x={0:.2}".format(x2)
print(equation)
```

2.7. Bài tập

Bài 2.1. Một quả bóng được ném thẳng đứng trong không khí từ độ cao h_0 so với mặt đất với vận tốc ban đầu v_0. Độ cao h và vận tốc v tiếp theo của nó được cho bởi các phương trình sau: $h = h_0 + v_0 t - \frac{1}{2} g t^2$, v = v_0 – gt, trong đó g=9.8m/s2 là gia tốc trọng trường. Viết đoạn chương trình tìm chiều cao và vận tốc v tại thời điểm t sau khi quả bóng được ném.

Bài 2.2. Viết câu lệnh tính giá trị các biểu thức sau: $a = \frac{2+e^{2.8}}{\sqrt{13}-2}$, $b = \frac{1-(1+ln2)^{-3.5}}{1+\sqrt{5}}$, $c =$

$sin\left(\frac{2-\sqrt{2}}{2+\sqrt{2}}\right)$

Bài 2.3. Viết chương trình nhập vào ba cạnh a, b, c của một tam giác. In ra màn hình chu vi, diện tích và các góc hợp bởi ba cạnh của tam giác đó.

Bài 2.4. Viết chương trình nhập vào số tiền, in ra màn hình số tờ của các mệnh giá 100000, 200000, 500000 tương ứng với số tiền đó.

Bài 2.5. Viết chương trình nhập vào số giây, in ra màn hình giờ:phút:giây tương ứng.

3.1. Tuple

Kiểu `tuple` là một dãy có thứ tự gồm không hoặc nhiều tham chiếu đối tượng. Kiểu `tuple` hỗ trợ cú pháp cắt và trích chọn phần tử giống như chuỗi. Giống như chuỗi, giá trị trong `tuple` là bất biến, vì vậy chúng ta không thể thay thế hoặc xóa bất kỳ mục nào của chúng.

Kiểu dữ liệu `tuple` có thể được gọi như một hàm, `tuple()`, không có đối số, nó trả về một bộ giá trị trống. Nếu có một đối số `tuple`, nó trả về một bản sao của đối số. Một giá trị `tuple` trống được tạo bằng cách sử dụng dấu ngoặc đơn trống, `()` và một giá trị `tuple` của một hoặc nhiều mục có thể được tạo bằng cách sử dụng dấu phẩy. Ví dụ:

```
t = "venus", -28, "green", "21", 19.74
```

t[-5]	t[-4]	t[-3]	t[-2]	t[-1]
'venus'	-28	'green'	'21'	19.74
t[0]	t[1]	t[2]	t[3]	t[4]

Kiểu `tuple` cung cấp hai phương thức: `t.count(x)` trả về số lần đối tượng `x` xuất hiện trong `t`, `t.index(x)` trả về chỉ số bên bên trái nhất đối tượng `x` trong `t`. Nếu `x` không xuất hiện trong `t`, phát sinh ngoại lệ `ValueError`.

Ngoài ra, các giá trị tuple có các phép toán + (nối), * (sao chép) và `[]` (trích chọn), và với `in` và `not` in kiểm tra thành viên. Các `tuple` có thể được so sánh bằng cách sử dụng các toán tử so sánh tiêu chuẩn (<, <=, ==, !=, >=, >)

```
>>> hair = "black", "brown", "blonde", "red"
>>> hair[2]
'blonde'
>>> hair[-3:] # same as: hair[1:]
('brown', 'blonde', 'red')
>>> hair[:2], "gray", hair[2:]
(('black', 'brown'), 'gray', ('blonde', 'red'))
```

Hoặc:

```
>>> hair[:2] + ("gray",) + hair[2:]
('black', 'brown', 'gray', 'blonde', 'red')
```

3.1.1. Tuple đặt tên

Một `tuple` được đặt tên hoạt động giống như một `tuple` đơn giản. Nó chỉ bổ sung thêm khả năng tham chiếu đến các mục trong `tuple` theo tên cũng như theo vị trí chỉ mục và điều này cho phép ta tạo tổng hợp các mục dữ liệu. Module `collections` cung cấp hàm `namestuple()`, hàm này được sử dụng để tạo các kiểu dữ liệu `tuple` tùy chỉnh. Ví du:

```
Sale = collections.namedtuple("Sale","productid customerid date
quantity price")
```

Đối số đầu tiên của `collection.nametuple()` là tên của kiểu dữ liệu `tuple` tùy chỉnh muốn tạo. Đối số thứ hai là một chuỗi các tên được phân biệt bằng dấu cách. Đối số đầu tiên và các tên trong đối số thứ hai phải là định danh hợp lệ. Hàm trả về một lớp (kiểu dữ liệu) có thể được sử dụng để tạo các giá trị `tuple` được đặt tên. Trong ví dụ trên, `Sale` là tên lớp mới và có thể dùng tên này để tạo các đối tượng kiểu `Sale`. Ví dụ:

```
sales = []
sales.append(Sale(432, 921, "2008-09-14", 3, 7.99))
sales.append(Sale(419, 874, "2008-09-15", 1, 18.49))
```

Ở đây, ta đã tạo một danh sách gồm hai bộ giá trị kiểu `sales`. Để truy cập đến các mặt hàng trong bộ giá trị ta sử dụng các chỉ số. Ví dụ: `price` của mục `sale` đầu tiên là `sales[0][-1]` (tức là 7,99), nhưng cũng có thể sử dụng tên, điều này làm cho mọi thứ rõ ràng hơn nhiều. Ví dụ:

```
total = 0
for sale in sales:
total += sale.quantity * sale.price
print("Total ${0:.2f}".format(total)) # prints: Total $42.46
```

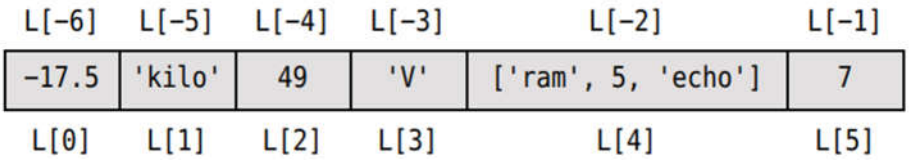
3.2. List

Kiểu `list` là một dãy có thứ tự không hoặc nhiều tham chiếu đối tượng. `list` hỗ trợ cú pháp cắt và trích như chuỗi và `tuple`.

Kiểu `list` có thể được gọi là một hàm, `list()` - không có đối số nó trả về một danh sách trống. Với danh sách đối số, nó trả về một bản sao `list`. Một đối tượng `list` trống được tạo bằng dấu `[]` và một đối tượng `list` có thể được tạo bằng cách sử dụng dãy các mục được phân tách bằng dấu phẩy bên trong dấu ngoặc `[]`. Ví dụ tạo danh sách

có tên L như sau:

```
L = [-17.5, "kilo", 49, "V", ["ram", 5, "echo"], 7]
```



Với danh sách L, ta có thể sử dụng toán tử cắt để truy cập các mục trong danh sách, chẳng hạn:

```
L[0] == L[-6] == -17.5
L[1] == L[-5] == 'kilo'
L[1][0] == L[-5][0] == 'k
L[4][2] == L[4][-1] == L[-2][2] == L[-2][-1] == 'echo'
L[4][2][1] == L[4][2][-3] == L[-2][-1][1] == L[-2][-1][-3] == 'c'
```

Các list có thể được lồng vào nhau, lặp lại và cắt nhỏ, giống như kiểu tuple. Trên thực tế, tất cả các ví dụ về tuple ở phần trước hoạt động hoàn toàn giống nhau nếu ta sử dụng list. list hỗ trợ phép toán thành viên in và not in, nối với +, mở rộng với += (tức là nối tất cả các mục trong toán hạng bên phải) và sao chép với * và *=. list có thể sử dụng với hàm len(), với lệnh del và các các phương thức ở bảng sau:

Bảng 3.1. *Các phương thức của lớp list.*

Cú pháp	Ý nghĩa
L.append(x)	Thêm đối tượng x vào cuối danh sách L
L.count(x)	Trả về số lần đối tượng x xuất hiện trong danh sách L
L.extend(m) L += m	Nối tất cả các mục có thể lặp lại của m vào cuối danh sách L, toán tử + = thực hiện điều tương tự
L.index(x, start, end)	Trả về chỉ số x xuất hiện ngoài cùng bên trái của trong danh sách L (hoặc trong phần start:end của L); ngoài ra phát sinh ngoại lệ ValueError
L.insert(i, x)	Chèn x vào danh sách L ở vị trí i
L.pop()	Trả về và xóa mục ngoài cùng bên phải của danh sách L
L.pop(i)	Trả về và xóa mục ở vị trí i trong L
L.remove(x)	Xóa x đầu tiên khỏi danh sách L; hoặc phát sinh ngoại lệ ValueError nếu không tìm thấy x
L.reverse()	Đảo ngược danh sách L
L.sort(...)	Sắp xếp danh sách L

3.2.2. Danh sách tổng quát

Danh sách nhỏ thường được tạo bằng cách sử dụng các hằng giá trị, nhưng danh sách dài hơn thường được tạo theo chương trình. Đối với danh sách các số nguyên, ta có thể sử dụng list(range(n)), hoặc nếu chỉ cần một trình lặp số nguyên, range() là

đủ, nhưng đối với các danh sách khác, sử dụng vòng lặp `for… in`. Ví dụ, ta muốn tạo ra một danh sách các năm nhuận trong một phạm vi nhất định. Đoạn chương trình như sau:

```
leaps = []
for year in range(1900, 1940):
     if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
          leaps.append(year)
```

Danh sách tổng quát là một biểu thức và một vòng lặp với điều kiện tùy chọn được đặt trong dấu ngoặc vuông, nơi vòng lặp được sử dụng để tạo các mục cho danh sách và nơi điều kiện có thể lọc ra các mục không mong muốn. Cú phát như sau:

```
[item for item in iterable]
```

Thao tác này sẽ trả về một danh sách có các phần tử trong `iterable`. Ta có thể sử dụng các biểu thức và có thể đính kèm một điều kiện, điều này dẫn đến hai cú pháp chung cho danh sách tổng quát:

```
[expression for item in iterable]
[expression for item in iterable if condition]
```

Ví dụ để tạo danh sách gồm những năm nhuận trong phạm vi từ 1900 đến năm 2020, ta có thể viết như sau:

```
leaps = [y for y in range(1900, 1940) if (y % 4 == 0 and y % 100 !=
0) or (y % 400 == 0)]
```

Ví dụ để tạo mã sản phẩm quần áo bao gồm ba thông tin: giới tính `(M, F)`, kích thước `(S, M, L, X)` và màu `(B, G, W)`. Chẳng hạng: `FLW` có nghĩa là trang phục cho nữ (`F`) với kích thước lớn (`L`) và màu trắng (`W`). Ta có thể tạo danh sách trên bằng đoạn chương trình sau:

```
codes = []
for s in "MF": # Male, Female
    for z in "SMLX": # Small, Medium, Large, eXtra large
        for c in "BGW": # Black, Gray, White
            codes.append(s + z + c)
```

Tuy nhiên, ta có thể tạo một danh sách tổng quát đơn giản như sau:

```
codes = [s+z+c for s in "MF" for z in "SMLX" for c in "BGW"]
```

Giả sử ta muốn loại trừ mã sản phẩm cho những loại quần áo là nữ (`F`) và kích cỡ là rất rộng (`X`), ta có viết đoạn chương trình sau:

```
codes = []
```

```
for s in "MF": # Male, Female
    for z in "SMLX": # Small, Medium, Large, eXtra large
        if s == "F" and z == "X":
            continue
        for color in "BGW": # Black, Gray, White
            codes.append(s + z + c)
```

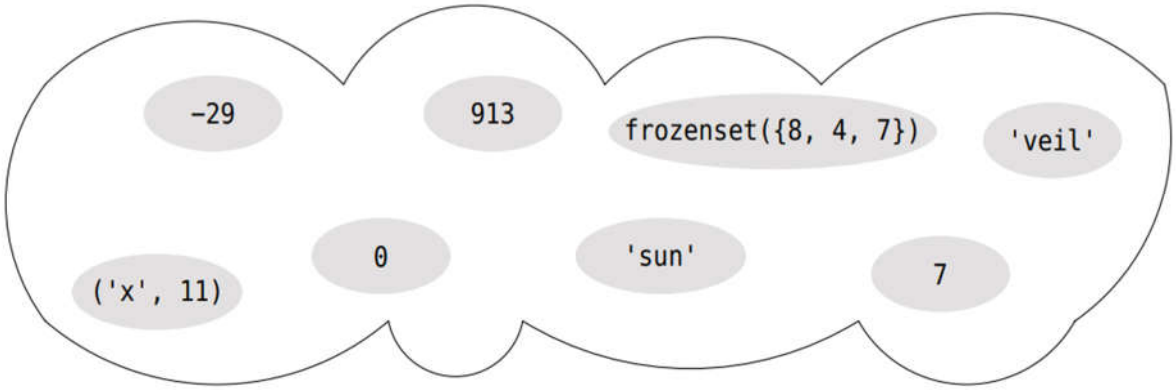
Tuy nhiên, ta có thể tạo một danh sách tổng quát đơn giản như sau:

```
codes = [s+z+c for s in "MF" for z in "SMLX" for c in "BGW" if not
(s=='F' and z=='X')]
```

3.3. Kiểu tập hợp

Một `set` là một tập hợp không có thứ tự gồm không hoặc nhiều tham chiếu đối tượng. Các `set` có thể thay đổi, vì vậy ta có thể thêm hoặc xóa các mục, nhưng vì chúng không có thứ tự nên không có khái niệm về chỉ số. Ví dụ:

```
S = {7, "veil", 0, -29, ("x", 11), "sun", frozenset({8, 4, 7}), 913}
```

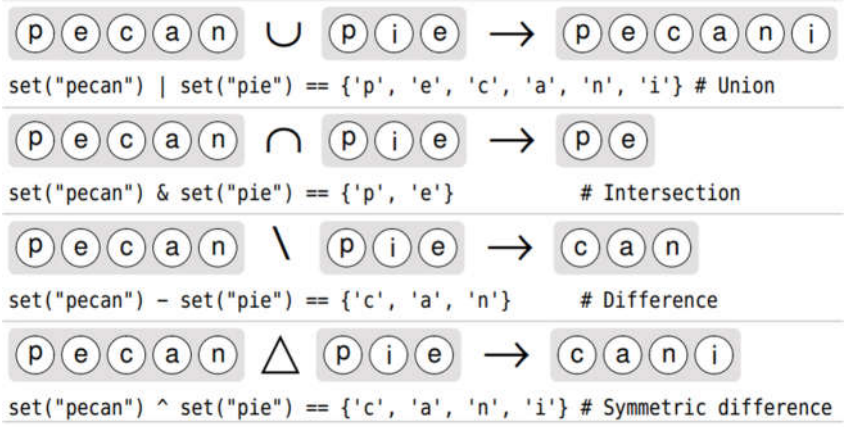


Kiểu dữ liệu `set` có thể được gọi như một hàm, `set()` (không có đối số) nó trả về một tập hợp trống, với một đối số tập hợp, nó trả về một bản sao của đối số và với bất kỳ đối số nào khác, nó sẽ cố gắng chuyển đổi đối tượng đã cho thành `set`. Có thể tạo một tập hợp một hoặc nhiều mục bằng cách sử dụng chuỗi các mục được phân tách bằng dấu phẩy bên trong dấu ngoặc nhọn `{}`.

Tập hợp luôn chứa các mục duy nhất, việc thêm các mục trùng lặp là được phép nhưng vô nghĩa. Ví dụ: ba tập hợp này giống nhau: `set("apple")`, `set("aple")` và `{'e', 'p', 'a', 'l'}`. Theo quan điểm này, các tập hợp thường được sử dụng để loại bỏ các bản sao. Ví dụ: nếu `x` là danh sách các chuỗi, sau khi thực thi `x = list(set(x))`, tất cả các chuỗi của `x` sẽ là duy nhất và theo thứ tự tùy ý.

Các bộ hỗ trợ chức năng `len()` tích hợp và kiểm tra tư cách thành viên nhanh chóng với `in` và `not in`. Chúng cũng cung cấp các toán tử tập hợp thông thường, như

hình sau:



Bảng 3.2. *Các phương thức của kiểu tập hợp.*

Cú pháp	Ý nghĩa
`s.add(x)`	Thêm mục x vào tập s nếu nó chưa có trong s.
`s.clear()`	Xóa các mục trong s
`s.copy()`	Trả về bản sao của tập hợp s (*)
`s.difference(t)` `s - t`	Trả về một tập hợp mới có mọi mục nằm trong tập hợp s không có trong tập hợp t (*)
`s.difference_update(t)` `s -= t`	Loại bỏ mọi mục trong tập t khỏi tập s
`s.discard(x)`	Loại bỏ các mục x khỏi tập hợp s nếu nó nằm trong s;
`s.intersection(t)` `s & t`	Trả về một tập hợp mới có mỗi mục nằm trong cả tập hợp s và tập hợp t (*)
`s.intersection_update(t)` `s &= t`	Làm cho tập s chứa giao của chính nó và tập t
`s.isdisjoint(t)`	Trả về True nếu tập s và t không có mục chung
`s.issubset(t)` `s <= t`	Trả về True nếu tập s bằng hoặc là tập con của tập t, sử dụng s<t để kiểm tra xem s có phải là tập con thực sự của t
`s.issuperset(t)` `s >= t`	Trả về True nếu tập s bằng hoặc là tập cha của tập t, sử dụng s>t để kiểm tra xem s có phải là tập hợp cha của t (*)
`s.pop()`	Trả về và xóa một mục ngẫu nhiên khỏi tập hợp s hoặc phát sinh ngoại lệ KeyError nếu s rỗng.
`s.remove(x)`	Loại bỏ mục x khỏi tập hợp s hoặc phát sinh ra ngoại lệ KeyError nếu x không nằm trong s; xem thêm set.discard()
`s.symmetric_difference(t)` `s ^ t`	Trả về một tập hợp mới có mọi mục trong tập hợp s và mọi mục trong tập hợp t, nhưng loại trừ các mục nằm trong cả hai tập hợp (*)
`s.symmetric_difference_update(t)` `s ^= t`	Làm cho tập s chứa sự hiệu đối xứng của chính nó và tập hợp t

Cú pháp	Ý nghĩa
`s.union(t)` `s | t`	Trả về một tập hợp mới có tất cả các mục trong tập hợp s và tất cả các mục trong tập hợp t không có trong tập hợp s (*)
`s.update(t)` `s |= t`	Thêm mọi mục trong tập hợp t không có trong tập hợp s, để đặt s

Tập hợp cũng được sử dụng để loại bỏ các mục không mong muốn. Ví dụ: nếu ta có một danh sách các tên tệp nhưng không muốn bao gồm bất kỳ tệp `makefile`, ta có thể viết như mã như sau:

```
filenames = set(filenames)
for makefile in {"MAKEFILE", "Makefile", "makefile"}:
     filenames.discard(makefile)
```

Tương tự ta có thể sử dụng toán tử hiệu tập hợp (–):

```
filenames = set(filenames) - {"MAKEFILE", "Makefile", "makefile"}
```

Ta cũng có thể sử dụng `set.remove()` để xóa các mục, mặc dù phương pháp này sinh ra ngoại lệ `KeyError` nếu mục mà nó được yêu cầu xóa không có trong bộ.

3.3.2. Tập tổng quát

Ngoài việc tạo tập hợp bằng cách gọi `set()` hoặc bằng cách sử dụng ký tự `{}`, ta cũng có thể tạo tập hợp bằng cách sử dụng tập hợp tổng quát như sau:

```
{expression for item in iterable}
{expression for item in iterable if condition}
```

Ta có thể sử dụng những cú pháp này để lọc dữ liệu một cách hiệu quả như sau:

```
html = {x for x in files if x.lower().endswith((".htm", ".html"))}
```

Cho trước danh sách các tên tệp trong `files`, ví dụ trên tạo ra tập hợp `html` chỉ giữ những tên tệp kết thúc bằng `.htm` hoặc `.html`, bất kể chữ hoa/thường.

3.3.3. Tập hợp cố định

Tập hợp cố định là tập hợp mà sau khi được tạo sẽ không thể thay đổi được. Tập cố định chỉ có thể được tạo bằng cách sử dụng kiểu dữ liệu `frozenset` như một hàm. Không có đối số, `frozenset()` trả về một tập hợp cố định rỗng, với một đối số `frozenset()`, nó trả về một bản sao của đối số và với bất kỳ đối số nào khác, nó cố gắng chuyển đổi đối tượng đã cho thành một tập hợp.

Vì các tập hợp cố định là không thay đổi, chúng chỉ hỗ trợ những phương thức và toán tử tạo ra kết quả mà không ảnh hưởng đến tập hợp cố định. Bảng 3.2 liệt kê tất cả các phương thức (đánh dấu *) hỗ trợ tập cố định như:

`frozenset.copy()`,

`frozenset.difference()` (–),

`frozenset.intersection()` (&),

`frozenset.isdisjoint()`, `frozenset.issubset()`, …

Nếu toán tử hai ngôi được sử dụng với một tập hợp và một tập hợp cố định, thì kiểu dữ liệu của kết quả giống với kiểu dữ liệu của toán hạng bên trái. Vì vậy, nếu `f` là một tập hợp cố định và `s` là một tập hợp, `f & s` sẽ tạo ra một tập hợp cố định và `s & f` sẽ tạo ra một tập hợp. Trong trường hợp của toán tử `==` và `!=`, thứ tự của các toán hạng không quan trọng và `f == s` sẽ tạo ra `True` nếu cả hai tập hợp chứa các mục giống nhau.

3.4. Kiểu ánh xạ

Kiểu ánh xạ là kiểu hỗ trợ toán tử thành viên (`in`) và hàm kích thước (`len()`) và có thể lặp lại. Ánh xạ là tập hợp các mục khóa-giá trị (key-value) và cung cấp các phương thức để truy cập các mục cũng như khóa và giá trị của chúng.

3.4.1. Kiểu từ điển

Kiểu `dict` là một tập hợp không có thứ tự gồm không hoặc nhiều cặp khóa-giá trị các khóa-giá trị này là tham chiếu đến các đối tượng.

Kiểu dữ liệu `dict` có thể được gọi là một hàm, `dict()`, không có các đối số nó trả về một từ điển rỗng và với một đối số ánh xạ, nó trả về một từ điển dựa trên đối số. Cũng có thể sử dụng dãy đối số với điều kiện mỗi mục trong dãy là một chuỗi của hai đối tượng, đối tượng đầu tiên được sử dụng làm khóa và đối tượng thứ hai được sử dụng làm giá trị. Đối với các từ điển, khóa là mã định danh Python hợp lệ. Từ điển cũng có thể được tạo bằng cách sử dụng dấu ngoặc nhọn `{}`, tạo một từ điển trống; Dấu ngoặc nhọn chứa một hoặc nhiều mục được phân cách bằng dấu phẩy, mỗi mục bao gồm một khóa, một dấu hai chấm và một giá trị.

Dưới đây là một số ví dụ minh họa các cú pháp khác nhau tạo ra cùng một từ điển:

```
d1 = dict({"id": 1948, "name": "Washer", "size": 3})
d2 = dict(id=1948, name="Washer", size=3)
d3 = dict([("id", 1948), ("name", "Washer"), ("size", 3)])
d4 = dict(zip(("id", "name", "size"), (1948, "Washer", 3)))
d5 = {"id": 1948, "name": "Washer", "size": 3}
```

Từ điển `d1` được tạo bằng cách sử dụng hàm `dict()`. Từ điển `d2` được tạo bằng cách sử dụng các đối số từ khóa. Từ điển `d3` và `d4` được tạo từ các chuỗi, và từ điển `d5` được tạo từ cặp dấu `{}`. Hàm `zip()` tích hợp sẵn được sử dụng để tạo từ điển `d4` trả về

danh sách các bộ giá trị, trong đó bộ đầu tiên là các "khóa", bộ thứ hai là các "giá trị".

Các khóa trong từ điển là duy nhất, vì vậy nếu ta thêm một mục khóa-giá trị có khóa trùng với khóa hiện có, tác dụng là thay thế giá trị của khóa đó bằng một giá trị mới. Dấu ngoặc vuông được sử dụng để truy cập các giá trị riêng lẻ, ví dụ: `d4["id"]` trả về 1948. Nếu truy cập một khóa không có trong từ điển sẽ phát sinh ngoại lệ `KeyError`.

Dấu ngoặc cũng có thể được sử dụng để thêm và xóa các mục từ điển. Để thêm một mục, ta sử dụng toán tử =, ví dụ: `d4["a"] = 1`. Và để xóa một mục, chúng tôi sử dụng câu lệnh `del`, ví dụ: `del d4["a"]` sẽ xóa mục có khóa là `"a"` từ từ điển hoặc sinh ra một ngoại lệ `KeyError` nếu không có mục nào. Các mục cũng có thể bị xóa (và trả về) khỏi từ điển bằng phương thức `dict.pop()`.

Bảng 3.3. *Các phương thức của từ điển dict.*

Cú pháp	Ý nghĩa
`d.clear()`	Xóa tất cả các mục khỏi dict d
`d.copy()`	Trả về một bản sao của dict d
`d.fromkeys(s, v)`	Trả về một dict có khóa là các mục trong chuỗi s và có giá trị là None hoặc v (nếu có)
`d.get(k)`	Trả về giá trị liên quan của khóa k hoặc None có nếu k không có trong dict d
`d.get(k, v)`	Trả về giá trị liên quan của khóa k hoặc v (nếu k không có trong dict d)
`d.items()`	Hiển thị tất cả các cặp (khóa, giá trị) trong dict d
`d.keys()`	Hiển thị các tất cả các khóa trong dict d
`d.pop(k)`	Trả về giá trị được liên kết của khóa k và xóa mục có khóa là k hoặc sinh ngoại lệ KeyError nếu k không nằm trong d
`d.pop(k, v)`	Trả về giá trị liên quan của khóa k và xóa mục có khóa là k hoặc trả về v nếu k không có trong dict d
`d.popitem()`	Trả về và xóa một cặp (khóa, giá trị) tùy ý khỏi dict d hoặc sinh ngoại lệ KeyError nếu d trống
`d.setdefault(k,v)`	Giống như phương thức dict.get(), ngoại trừ việc nếu khóa không nằm trong dict d, một mục mới sẽ được chèn với khóa k và với giá trị là Không hoặc của v nếu v được cho trước
`d.update(a)`	Thêm mọi cặp (khóa, giá trị) từ a không có trong dict d đến d và đối với mọi khóa nằm trong cả d và a, thay thế giá trị tương ứng trong d bằng giá trị trong a — a có thể là một từ điển, hoặc cặp (khóa, giá trị) hoặc các đối số từ khóa
`d.values()`	Hiển thị tất cả các giá trị trong dict d

Các phương thức `dict.items()`, `dict.keys()` và `dict.values()` đều trả về dạng xem từ điển. Chế độ xem từ điển thực sự là một đối tượng lặp thường để sử dụng trong vòng lặp `for…in`.

Từ điển thường được sử dụng để giữ số lượng các mục duy nhất. Ví dụ như đếm số lần xuất hiện của mỗi từ duy nhất trong một tệp. Đây là một chương trình hoàn chỉnh (uniquewords1.py) liệt kê mọi từ và số lần nó xuất hiện theo thứ tự bảng chữ cái cho tất cả các tệp được liệt kê trên dòng lệnh:

```
import string
import sys
words = {}
strip=string.whitespace + string.punctuation + string.digits + "\"'"
for filename in sys.argv[1:]:
     for line in open(filename):
          for word in line.lower().split():
               word = word.strip(strip)
               if len(word) > 2:
                    words[word] = words.get(word, 0) + 1
for word in sorted(words):
     print("'{0}' occurs {1} times".format(word, words[word]))
```

3.4.2. Từ điển ngầm định

Từ điển mặc định là từ điển, nó có tất cả các toán tử và phương thức mà từ điển cung cấp. Điểm khác biệt là cách chúng xử lý các khóa bị thiếu. Nếu ta sử dụng khóa không tồn tại thời điểm truy cập từ điển, sinh lỗi KeyError. Điều này rất hữu ích vì ta muốn biết khóa mà ta truy cập có bị thiếu hay không. Nhưng trong một số trường hợp, ta muốn mọi khóa sử dụng đều có mặt, điều đó có nghĩa là một mục có khóa được chèn vào từ điển tại thời điểm truy cập lần đầu tiên.

Ví dụ, nếu ta có một từ điển d không có khóa m, mã x = d[m] sẽ tạo ra một ngoại lệ KeyError. Nhưng nếu d là từ điển mặc định và nếu khóa m nằm trong từ điển mặc định, thì giá trị tương ứng được trả về, nhưng nếu m không phải là khóa trong từ điển mặc định, mục mới có khóa m được tạo với giá trị mặc định và giá trị của mục mới tạo được trả về. Để tạo một từ điển ngầm định, ta sử dụng cú pháp sau:

```
name = collections.defaultdict(type)
```

Trong đó name là tên tham chiếu, type kiểu dữ liệu ngầm định cho giá trị của khóa.

3.4.3. Kiểu từ điển có thứ tự

Loại từ điển có thứ tự - collection.OrderedDict - đã được xuất hiện trong Python 3.1. Từ điển có thứ tự có thể được sử dụng thay thế cho các từ không có thứ tự vì chúng cung cấp cùng một giao diện lập trình. Sự khác biệt là từ điển có thứ tự lưu trữ các mục của chúng theo thứ tự mà chúng được chèn vào. Ví dụ cách tạo từ điển có thứ

tự bằng danh sách các bộ hai giá trị:

```
d = collections.OrderedDict([('z', -4), ('e', 19), ('k', 7)])
```

Hoặc ta có thể tạo từ điển theo cách:

```
tasks = collections.OrderedDict()
tasks[8031] = "Backup"
tasks[4027] = "Scan Email"
tasks[5733] = "Build System"
```

Nếu ta tạo từ điển không có trật tự cách tương tự và yêu cầu các khóa của chúng, thứ tự của các khóa trả về sẽ là tùy ý. Nhưng đối với các từ điển có thứ tự, ta có thể nhận các khóa trả về theo đúng thứ tự mà chúng đã được chèn vào. Vì vậy, đối với những ví dụ này, nếu ta viết `list(d.keys())`, sẽ nhận được danh sách `['z', 'e', 'k']` và nếu viết `list(task.keys())`, nhận được danh sách `[8031, 4027, 5733]`.

Một tính năng khác của từ điển có thứ tự là nếu ta thay đổi giá trị của một mục, nghĩa là, nếu ta chèn một mặt hàng có cùng khóa với khóa hiện có - thì thứ tự sẽ không bị thay đổi. Vì vậy, nếu ta thực hiện `tasks[8031] = "Daily backup"`, và sau đó yêu cầu danh sách các khóa, ta sẽ nhận được danh sách theo đúng thứ tự như trước.

Nếu ta muốn di chuyển một mục đến cuối, ta phải xóa nó và sau đó chèn lại. Ta cũng có thể gọi `popitem()` để xóa và trả về mục (khóa-giá trị) cuối cùng trong từ điển có thứ tự; hoặc chúng ta có thể gọi `popitem(last = False)`, trong trường hợp này, mục đầu tiên sẽ được trả lại và loại bỏ.

3.5. Sao chép và duyệt trong kiểu kết hợp

3.5.1. Các hàm và phép toán hỗ trợ duyệt

Kiểu dữ liệu có thể duyệt là kiểu dữ liệu có thể trả về từng mục của nó. Bất kỳ đối tượng nào có phương thức `__iter__()` hoặc dãy bất kỳ (tức là đối tượng có phương thức `__getitem__()` nhận các đối số nguyên bắt đầu từ 0) là một đối tượng có thể duyệt và có thể cung cấp một trình để duyệt các phần tử. Trình duyệt là một đối tượng cung cấp phương thức `__next__()` trả về lần lượt từng mục kế tiếp và đưa ra ngoại lệ `StopIteration` khi không còn mục nào nữa. Bảng 3.4 liệt kê các toán tử và hàm có thể được sử dụng với các đối tượng lặp.

Bảng 3.4. *Các hàm và phép toán duyệt phần tử.*

Cú pháp	Ý nghĩa
`s + t`	Trả về một dãy là sự kết hợp của dãy s và t
`s * n`	Trả về một chuỗi là nối của chuỗi s n lần

Cú pháp	Ý nghĩa
`x in i`	Trả về True nếu mục x ở trong i; sử dụng not in để kiểm tra ngược lại
`all(i)`	Trả về True nếu mọi mục trong mục có thể duyệt
`any(i)`	Trả về True nếu bất kỳ mục nào trong có thể duyệt
`enumerate(i, start)`	Thường được sử dụng trong các vòng lặp for… in để cung cấp một dãy các bộ giá trị (chỉ mục, mục) với các chỉ mục bắt đầu từ 0 hoặc start.
`len(x)`	Trả về "độ dài" của x. Nếu x là tập hợp thì đó là số mục; nếu x là một chuỗi thì nó là số ký tự.
`max(i, key)`	Trả về mục lớn nhất trong i hoặc mục có giá trị key(mục) lớn nhất
`min(i, key)`	Trả về mục nhỏ nhất trong i hoặc mục có giá trị key(mục) lớn nhất
`range(start, stop, step)`	Trả về một trình duyệt các số nguyên. Với một đối số (stop), trình duyệt đi từ 0 đến stop - 1; với hai đối số (start, stop) trình lặp đi từ đầu đến stop - 1; với ba đối số, nó đi từ start đến stop - 1 với bước nhảy step.
`reversed(i)`	Trả về một trình duyệt các mục từ i theo thứ tự ngược lại
`sorted(i, key, reverse)`	Trả về danh sách các mục từ i theo thứ tự đã sắp xếp; Nếu reverse ngược là True, việc sắp xếp được thực hiện theo thứ tự ngược lại.
`sum(i, start)`	Trả về tổng của các mục trong i cộng với start (mặc định là 0); i không chứa chuỗi.
`zip(i1, ..., iN)`	Trả về một trình duyệt các giá trị i1 đến iN.

Thứ tự các mục lại phụ thuộc vào kiểu dữ liệu. Chẳng hạng, `list` và `tuple`, các mục thường được trả về theo thứ tự đầu tiên (vị trí chỉ mục 0) cho đến cuối, nhưng một số kiểu như `dict`, `set` trình duyệt trả về các mục theo thứ tự tùy ý.

3.5.2. Sao chép kiểu dữ liệu kết hợp

Vì Python sử dụng tham chiếu đối tượng nên khi ta sử dụng toán tử gán (=), không có quá trình sao chép nào diễn ra. Nếu toán hạng bên phải là một hằng như một chuỗi hoặc một số, thì toán hạng bên trái được đặt thành một tham chiếu đối tượng đến đối tượng trong bộ nhớ chứa giá trị của hằng đó. Nếu toán hạng bên phải là một tham chiếu đối tượng, thì toán hạng bên trái được đặt thành một tham chiếu đối tượng tham chiếu đến cùng một đối tượng như toán hạng bên phải. Do đó phép toán gán làm việc rất hiệu quả. Ví dụ:

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs
>>> beatles, songs
(['Because', 'Boys', 'Carol'], ['Because', 'Boys', 'Carol'])
```

Khi đó, một tham chiếu đối tượng mới (`beatles`) đã được tạo và cả hai đều tham chiếu đến cùng một danh sách, không có sự sao chép nào diễn ra. Vì danh sách có thể thay đổi, ta có thể thay đổi như sau:

```
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Cayenne'])
```

Ta đã thay đổi danh sách bằng cách sử dụng biến `beatles`, nhưng đây là tham chiếu đến danh sách giống như `songs`. Vì vậy, bất kỳ thay đổi nào được thực hiện thông qua một trong hai tham chiếu đối tượng sẽ tác động cho đối tượng kia.

Tuy nhiên, trong một số tình huống, ta thực sự muốn có một bản sao riêng của kiểu kết hợp (hoặc đối tượng có thể thay đổi khác). Đối với các dãy, khi chúng ta lấy một trích chọn, ví dụ: `songs[:2]` – phần này luôn là một bản sao độc lập của các mục được sao chép. Vì vậy, để sao chép toàn bộ một dãy chúng ta có thể làm như sau:

```
>>> songs = ["Because", "Boys", "Carol"]
>>> beatles = songs[:]
>>> beatles[2] = "Cayenne"
>>> beatles, songs
(['Because', 'Boys', 'Cayenne'], ['Because', 'Boys', 'Carol'])
```

Đối với từ điển và tập hợp, có thể sao chép bằng cách sử dụng hàm `dict.copy()` và `set.copy()`. Ngoài ra, module `copy` cung cấp hàm `copy.copy()` trả về một bản sao của đối tượng. Một cách khác để sao chép các kiểu kết hợp bên trong là sử dụng tên kiểu như một hàm với đối số là một đối tượng cần sao chép. Dưới đây là một số ví dụ:

```
copy_of_dict_d = dict(d)
copy_of_list_L = list(L)
copy_of_set_s = set(s)
```

Chú ý: đối với kiểu kết hợp lồng nhau (chẳng hạn như có 1 danh sách con bên trong danh sách), khi sao chép sẽ tạo ra hai tham chiếu đến cùng một danh sách con. Ví dụ:

```
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = x[:] # shallow copy
>>> x, y
([53, 68, ['A', 'B', 'C']], [53, 68, ['A', 'B', 'C']])
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['Q', 'B', 'C']])
```

Khi danh sách x được sao chép qua y, tham chiếu đến danh sách lồng nhau ["A", "B", "C"] sẽ được sao chép. Điều này có nghĩa là cả x và y đều có mục thứ ba là tham chiếu đến danh sách con này, vì vậy mọi thay đổi đối với danh sách con này đều tác động đến cả x và y. Nếu chúng ta thực sự cần các bản sao độc lập của các kiểu kết hợp tập lồng nhau tùy ý, chúng ta có thể dùng hàm sao chép sau:

```
>>> import copy
>>> x = [53, 68, ["A", "B", "C"]]
>>> y = copy.deepcopy(x)
>>> y[1] = 40
>>> x[2][0] = 'Q'
>>> x, y
([53, 68, ['Q', 'B', 'C']], [53, 40, ['A', 'B', 'C']])
```

3.6. Bài tập

Bài 3.1. Viết đoạn chương trình nhập vào một dãy số nguyên, in ra màn hình giá trị lớn nhất, nhỏ nhất, trung bình của dãy số.

Bài 3.2. Viết chương trình nhập vào tọa độ ba điểm trong mặt phẳng $A(x_a, y_a)$, $B(x_b, y_b)$, $C(x_c, y_c)$. Tính độ dài ba đoạn thẳng AB, BC, AC. Tính chi vi, diện tích, các góc hợp bởi ba đoạn thẳng vừa tính được.

Bài 3.3. Viết chương trình tạo ra một danh sách gồm các năm nhuận trong phạm vi từ 1900 đến 2099. Sau đó nhập vào một năm bất kì, thông báo năm đó có phải là năm nhuận hay không?

Bài 3.4. Viết chương trình tạo ra 3 tập hợp môn học của 3 ngành Khoa học máy tính, công nghệ phần mềm và mạng máy tính. Cho biết những môn học khác nhau, giống nhau giữa các ngành. Giả sử sinh viên học ngành này muốn học thêm ngành khác thì cần bổ sung những môn gì?

Bài 3.5. Viết chương trình tạo một 2 từ điển bao gồm: từ điển Mon bao gồm (mã môn:số đvht) và từ điển Diem bao gồm (mã môn:điểm). Hãy nhập tất cả các môn và số đvht, nhập điểm của những môn đã học. Tính điểm trung bình.

CHƯƠNG 4. CẤU TRÚC ĐIỀU KHIỂN VÀ HÀM

4.1. Cấu trúc điều khiển

4.1.1. Lệnh rẽ nhánh

Cú pháp:

```
if e1:
     s1
elif e2:
     s2
...
elif eN:
     sN
else:
     sM
```

Trong đó các mệnh đề `elif` và `else` là tùy chọn. Một số trường hợp ta có thể viết lệnh `if…else…` trên một dòng như sau:

```
expression1 if boolean_expression else expression2
```

Một mẫu lập trình phổ biến là đặt một biến thành giá trị mặc định, sau đó thay đổi giá trị nếu cần. Ví dụ:

```
offset = 20
if not sys.platform.startswith("win"):
     offset = 10
```

Ta có thể viết lại thành lệnh `if` một dòng như sau:

```
offset = 20 if sys.platform.startswith("win") else 10
```

4.1.2. Lệnh lặp

a. Lệnh while

Cú pháp:

```
while e:
     s1
```

```
else:
     s2
```

Mệnh đề `else` là tùy chọn. Nếu `e` là `True`, lệnh `s1` trong `while` được thực thi. Nếu `e` là `False`, vòng lặp sẽ kết thúc và nếu mệnh đề else có mặt, lệnh `s2` sẽ được thực thi. Bên trong khối `while`, nếu một câu lệnh `continue` được thực thi, quyền điều khiển ngay lập tức quay về đầu vòng lặp và biểu thức `e` kiểm tra lại. Nếu vòng lặp không kết thúc bình thường (ví dụ gặp câu lệnh `break`, `return` hoặc phát ngoại lệ), câu lệnh `s2` bị bỏ qua.

b. Vòng lặp for

Cú pháp:

```
for expression in iterable:
     s1
else:
     s2
```

`expression` thường là một biến đơn hoặc một dãy các biến, thường ở dạng một bộ giá trị. Nếu một câu lệnh `continue` được thực thi bên trong khối `for… in`, thì quyền điều khiển được chuyển đến đầu vòng lặp và lần lặp tiếp theo bắt đầu. Nếu vòng lặp kết thúc thì lệnh `s2` được thực thi. Nếu vòng lặp bị ngắt do câu lệnh `break` hoặc câu lệnh trả về (nếu vòng lặp nằm trong một hàm hoặc phương thức) hoặc nếu một ngoại lệ được đưa ra, lệnh `s2` không được thực thi.

4.2. Quản lý ngoại lệ

4.2.1. Bắt và phát sinh ngoại lệ

Các trường hợp ngoại lệ được phát hiện bằng cách sử dụng khối `try… except`, cú pháp chung như sau:

```
try:
     try_suite
except exception_group1 as variable1:
     except_suite1
…
except exception_groupN as variableN:
     except_suiteN
else:
     else_suite
finally:
     finally_suite
```

Phải có ít nhất một khối `except` nhưng cả khối `else` và `finally` đều là tùy chọn. Lệnh của khối `else` được thực thi khi lệnh trong try kết thúc bình thường, nhưng nó không được thực thi nếu xảy ra ngoại lệ. Nếu có khối `finally`, nó luôn được thực thi ở cuối.

Mỗi nhóm ngoại lệ của mệnh đề `except` có thể là một ngoại lệ duy nhất hoặc một loạt các ngoại lệ được đặt trong ngoặc đơn. Đối với mỗi nhóm, phần `'as variable'` là tùy chọn; nếu được sử dụng, biến này chứa ngoại lệ đã xảy ra và có thể được truy cập trong bộ của khối ngoại lệ.

Nếu một ngoại lệ xảy ra ở các lệnh trong khối `try`, thì từng mệnh đề `except` sẽ được thử lần lượt. Nếu ngoại lệ khớp với một nhóm ngoại lệ, lệnh tương ứng sẽ được thực thi. Để đối sánh với một nhóm ngoại lệ, ngoại lệ phải cùng loại với (hoặc một trong các) loại ngoại lệ được liệt kê trong nhóm.

4.3. Hàm

Hàm là một cách mà ta có thể đóng gói và tham số hóa chức năng. Bốn loại hàm có thể được tạo bằng Python: hàm toàn cục, hàm cục bộ, hàm lambda và phương thức. Các đối tượng toàn cục (bao gồm cả hàm) có thể truy cập vào bất kỳ mã nào trong cùng một module (tức là cùng một tệp `.py`) mà đối tượng được tạo. Các đối tượng toàn cục cũng có thể được truy cập từ các module khác. Các hàm cục bộ (còn được gọi là các hàm lồng nhau) là các hàm được định nghĩa bên trong các hàm khác. Các hàm này chỉ "hiển thị" đối với hàm mà chúng được định nghĩa.

Các hàm lambda là các biểu thức, vì vậy chúng có thể được tạo tại điểm sử dụng; tuy nhiên, chúng bị hạn chế hơn nhiều so với các hàm bình thường. Phương thức là các hàm được liên kết với một kiểu dữ liệu cụ thể và chỉ có thể được sử dụng cùng với kiểu dữ liệu đó.

Python cung cấp nhiều hàm tích hợp sẵn, thư viện chuẩn và thư viện của bên thứ ba thêm hàng trăm hàm nữa (hàng nghìn nếu chúng ta tính tất cả các phương thức), vì vậy trong nhiều trường hợp, hàm chúng ta muốn đã được viết sẵn. Vì lý do này, luôn nên kiểm tra tài liệu trực tuyến của Python để xem những gì đã có sẵn.

Cú pháp để viết một hàm:

```
def functionName(parameters):
     statement
```

Các tham số là tùy chọn và nếu có nhiều hơn một tham số, chúng được viết cách nhau bằng dấu phẩy hoặc dưới dạng chuỗi các cặp 'Định dang = giá trị' như chúng ta sẽ

thảo luận ngay sau đây. Ví dụ: đây là một hàm tính diện tích hình tam giác bằng công thức Heron:

```
def heron(a, b, c):
     s = (a + b + c) / 2
     return math.sqrt(s * (s - a) * (s - b) * (s - c))
```

Bên trong hàm, mỗi tham số, `a`, `b` và `c`, được khởi tạo với giá trị tương ứng được truyền dưới dạng đối số. Khi hàm được gọi, ta phải cung cấp tất cả các đối số, ví dụ, `heron(3, 4, 5)`. Nếu đưa ra quá ít hoặc quá nhiều đối số, phát sinh ngoại lệ `TypeError`. Khi chúng ta thực hiện một lời gọi hàm như trên, ta đang sử dụng các đối số vị trí, vì mỗi đối số được truyền vào ở vị trí tương ứng. Vì vậy, trong trường hợp này, `a` được đặt thành 3, `b` thành 4 và `c` thành 5, khi hàm được gọi.

Mọi hàm trong Python đều trả về một giá trị, mặc dù việc bỏ qua giá trị trả về là hoàn toàn có thể chấp nhận được (và phổ biến). Giá trị trả về là một giá trị đơn lẻ hoặc một bộ giá trị và các giá trị được trả về có thể là các kiểu kết hợp. Ta có thể thoát khỏi hàm bất kỳ lúc nào bằng cách sử dụng câu lệnh `return`. Nếu sử dụng `return` mà không có đối số hoặc nếu không có câu lệnh `return` nào, thì hàm sẽ trả về `None`.

Một số hàm có các tham số nhận giá trị mặc định. Ví dụ: đây là một hàm đếm các chữ cái trong một chuỗi, mặc định là các chữ cái ASCII:

```
def letter_count(text, letters=string.ascii_letters):
     letters = frozenset(letters)
     count = 0
     for char in text:
          if char in letters:
          count += 1
     return count
```

Ta đã chỉ định một giá trị mặc định cho tham số `letters` bằng cách sử dụng cú pháp tham số = giá trị ngầm định. Điều này cho phép ta gọi `letter_count()` chỉ với một đối số, ví dụ: `letter_count("Maggie và Hopey")`. Ở đây, bên trong hàm, các chữ cái sẽ là chuỗi được cung cấp làm giá trị mặc định. Nhưng chúng ta vẫn có thể thay đổi giá trị mặc định, ví dụ: sử dụng đối số vị trí bổ sung, `letter_count("Maggie và Hopey", "aeiouAEIOU")` hoặc sử dụng đối số từ khóa, `letter_count("Maggie và Hopey", letter = "aeiouAEIOU")`.

Cú pháp này không cho phép các tham số ngầm định đứng trước tham số không ngầm định, ví dụ hàm `def bad (a, b = 1, c):` sẽ không hoạt động. Mặt khác, đối số không cần phải tuân theo thứ tự chúng xuất hiện trong định nghĩa của hàm, thay vào

đó, chúng ta có thể sử dụng các đối số từ khóa, chuyển mỗi đối số ở dạng `name = value`.

Ví dụ dưới là một hàm nhỏ trả về chuỗi mà nó được cung cấp hoặc nếu nó dài hơn độ dài được chỉ định, nó sẽ trả về một phiên bản rút gọn với độ dài cho trước:

```
def shorten(text, length=25, indicator="..."):
     if len(text) > length:
          text = text[:length - len(indicator)] + indicator
     return text
```

Một số ví dụ khi gọi hàm:

```
shorten("The Silkie")                 # returns: 'The Silkie'
shorten(length=7, text="The Silkie")  # returns: 'The ...'
shorten("The Silkie", indicator="&", length=7) # returns: 'The Si&'
shorten("The Silkie", 7, "&")         # returns: 'The Si&
```

Bởi vì cả `length` và `indicator` đều có giá trị ngầm định, một trong hai hoặc cả hai đều có thể bị bỏ qua hoàn toàn, trong trường hợp đó giá trị mặc định được sử dụng, đây là điều xảy ra trong lần gọi đầu tiên. Trong lần gọi thứ hai, ta sử dụng các đối số với từ khóa cho cả hai tham số được chỉ định, vì vậy ta có thể sắp xếp chúng theo ý muốn. Lời gọi thứ ba kết hợp cả đối số vị trí và đối số với từ khóa. Ta phải dụng đối số vị trí trước đối số với từ khóa, và sau đó là hai đối số từ khóa. Lời gọi thứ tư chỉ đơn giản sử dụng các đối số vị trí.

4.3.1. Tên hàm và docstring

Việc sử dụng các tên hay cho một hàm và các tham số của nó sẽ giúp ích cho việc sử dụng hàm trở nên rõ ràng đối với các lập trình viên khác, và với chính chúng ta một thời gian sau khi chúng ta đã tạo hàm. Dưới đây là một số quy tắc ngón tay cái mà bạn có thể muốn xem xét:

Sử dụng cơ chế đặt tên và sử dụng nó một cách nhất quán. Trong tài liệu này, ta sử dụng `UPPERCASE` cho các hằng số, `TitleCase` cho các lớp (bao gồm cả ngoại lệ), các hàm và phương thức và chữ thường hoặc `lowercase_with_underscores` cho mọi thứ khác. Đối với tất cả các tên, tránh viết tắt, trừ khi chúng được chuẩn hóa và sử dụng rộng rãi.

Ta có thể thêm mô tả vào bất kỳ hàm nào bằng cách sử dụng `docstring` - đây chỉ đơn giản là một chuỗi nằm ngay sau dòng `def` và trước khi mã của hàm bắt đầu thích hợp. Ví dụ: đây là hàm `shorten()` mà chúng ta đã thấy trước đó, nhưng lần này được viết đầy đủ:

```
def shorten(text, length=25, indicator="..."):
```

```
    """Returns text or a truncated copy with the indicator added
    text is any string; length is the maximum length of the returned
    string (including any indicator); indicator is the string
    added at the end to indicate that the text has been shortened
    >>> shorten("Second Variety") 'Second Variety'
    >>> shorten("Voices from the Street", 17) 'Voices from th...'
    >>> shorten("Radio Free Albemuth", 10, "*") 'Radio Fre*' """
     if len(text) > length:
          text = text[:length - len(indicator)] + indicator
     return text
```

Để lấy chuỗi mô tả hàm `shorten()`, ta viết như sau: `shorten.__doc__`

4.3.2. Tham số và đối số

Ta có thể sử dụng dấu * để gởi bộ đối số. Ví dụ: nếu muốn tính diện tích của một tam giác với độ dài của các cạnh trong một danh sách, ta có thể thực hiện lệnh gọi như sau, `heron(side[0], side[1], side[2])`, hoặc đơn giản là sử dụng danh sách tham số và thực hiện lệnh gọi đơn giản hơn, `heron(* side)`. Ta cũng có thể sử dụng toán tử * trong danh sách tham số để tạo các hàm có thể nhận một số lượng thay đổi các đối số vị trí. Đây là một hàm `product()` tính tích các đối số mà nó được đưa ra:

```
def product(*args):
     result = 1
     for arg in args:
          result *= arg
     return result
```

Hàm này có một tham số là `args`. Có dấu * ở phía trước có nghĩa là bên trong hàm, tham số `args` sẽ là một bộ giá trị với các đối số được đưa ra. Dưới đây là một số lời gọi:

```
product(1, 2, 3, 4)   # args == (1, 2, 3, 4); returns: 24
product(5, 3, 8)      # args == (5, 3, 8); returns: 120
product(11)           # args == (11,); returns: 11
```

Chúng ta cũng có thể có các đối số với từ khóa theo sau các đối số thông thường. Ví dụ như hàm để tính tổng các đối số của nó, mỗi đối số được nâng lên thành lũy thừa với số mũ đã cho:

```
def sum_of_powers(*args, power=1):
     result = 0
     for arg in args:
          result += arg ** power
     return result
```

Hàm có thể được gọi với các đối số thông thường, ví dụ: `sum_of_powers(1, 3, 5)` hoặc với cả đối số với từ khóa, ví dụ: `sum_of_powers(1, 3, 5, power = 2)`.

Cũng có thể sử dụng * làm "tham số" theo đúng nghĩa của nó. Điều này có nghĩa là không thể có đối số thông thường sau dấu * (chỉ có đối số với từ khóa được phép). Đây là phiên bản sửa đổi của hàm `heron()`. Lần này hàm nhận chính xác ba đối số vị trí và có một đối số từ khóa tùy chọn:

```
def heron2(a, b, c, *, units="square meters"):
     s = (a + b + c) / 2
     area = math.sqrt(s * (s - a) * (s - b) * (s - c))
     return "{0} {1}".format(area, units)
```

Ví dụ các lời gọi sau:

```
heron2(25, 24, 7) # returns: '84.0 square meters'
heron2(41, 9, 40, units="sq. inches") # returns: '180.0 sq. inches'
heron2(25, 24, 7, "sq. inches") # WRONG! raises TypeError
```

Trong lời gọi thứ ba, ta sử dụng đối số thư tư thông thường, nhưng dấu * không cho phép điều này và gây ra lỗi `TypeError`.

Bằng cách đặt * làm tham số đầu tiên, chúng ta có thể bắt buộc người dùng phải sử dụng đối số với từ khóa. Ví dụ:

```
def f(*, a=1, b=2, c=3):
    return a+b+c
print(f())
print(f(a=4))
print(f(b=4, a=3))
print(3)   #lỗi
```

Ta có thể dùng một từ điển truyền các đối số với từ khóa của một hàm bằng cách sử dụng dấu (**). Ví dụ:

```
def f(a=1, b=2, c=3):
    return a+b+c
s = dict(a=4, b=5, c=7)
print(f(**s))
```

4.3.3. Hàm Lambda

Cú pháp:

```
lambda parameters: expression
```

Các tham số là tùy chọn và nếu được cung cấp, chúng thường chỉ là các tên biến

được phân tách bằng dấu phẩy. `expression` không được chứa các lệnh rẽ nhánh, lệnh lặp và không chứa câu lệnh `return`. Khi một hàm `lambda` được gọi, nó sẽ trả về kết quả của `expression`. Nếu `expression` là một bộ giá trị, nó phải được đặt trong dấu ngoặc đơn.

```
area = lambda b, h: 0.5 * b * h
area(5, 6)
```

4.4. Module

Trong khi các hàm cho phép chúng ta chia nhỏ các đoạn mã để chúng có thể được sử dụng lại trong một chương trình, các module cung cấp một tập hợp các hàm với nhau để chúng có thể được sử dụng bởi bất kỳ số lượng chương trình nào. Python cũng có các phương tiện để tạo các gói (package) là các module được nhóm lại với nhau, thường là do các module của chúng cung cấp chức năng liên quan hoặc vì chúng phụ thuộc vào nhau.

4.4.1. Module và gói

Một module Python đơn giản là một tệp `.py`. Module có thể chứa mã Python bất kì và do đó các module cũng như các chương trình. Sự khác biệt chính là các chương trình được thiết kế để chạy, trong khi các module được thiết kế để chèn vào các chương trình và sử dụng.

Không phải tất cả các module đều kết hợp trong tệp `.py`, ví dụ module `sys` được xây dựng trong Python và một số module được viết bằng các ngôn ngữ khác (phổ biến nhất là C). Tuy nhiên, phần lớn thư viện của Python được viết bằng Python, vì vậy, ví dụ: nếu viết `import collections`, ta có thể sử dụng câu lệnh `collection.nametuple()` và hàm ta đang truy cập nằm trong tệp `collections.py`. Không có gì khác biệt đối với các module được được viết bằng ngôn ngữ khác nhau.

Cú pháp chèn module vào chương trình:

```
import m
import m1, m2, …, mn
import m as name
```

Ở đây `m` thường là một module như collections, nhưng có thể là package hoặc module trong package, trong trường hợp này, mỗi phần được phân tách bằng dấu chấm (.), Ví dụ: `os.path`. Cách viết thứ ba là chọn một tên khác để sử dụng trong chương trình cho thuận tiện.

Các lệnh `import` nên đặt ở đầu mỗi file `.py` và sau dòng mô tả (`docstring`). Nên

chèn module chuẩn trước sau đó là các module của bên thứ ba và cuối cùng là module tự viết.

Cách thứ hai để chèn module theo cú pháp sau:

```
from m import o as name
from m import o1, o2, ..., oN
from m import (o1, o2, ..., oN)
from m import *
```

Chú ý khi sử dụng cú pháp này dễ dẫn đến tình huống xung đột tên.

Để biết các module và package được import ở đâu, ta sử dụng thuộc tính `sys.path` trong module `sys`.

Khi `import` một module, nếu nó không phải là module tích hợp sẵn, Python sẽ lần lượt tìm kiếm module trong từng đường dẫn được liệt kê trong `sys.path`. Hệ quả của việc này là nếu ta tạo một module hoặc chương trình có cùng tên với một trong các module thư viện của Python, thì module ta tạo được tìm thấy đầu tiên, chắc chắn sẽ gây ra sự cố. Để tránh điều này, đừng bao giờ tạo chương trình hoặc module có cùng tên với một trong các module ở thư mục cấp cao nhất của thư viện Python (module cấp cao nhất là module có tệp .py nằm trong một trong các thư mục trong đường dẫn của Python, chứ không phải trong một trong các thư mục con của thư mục đó.) Ví dụ, trên Windows, đường dẫn Python thường bao gồm một thư mục có tên C:\Python\Python38-32\Lib, vì vậy ta không được tạo module có tên `Lib.py`, cũng như module có cùng tên với bất kỳ module nào trong thư mục C:\Python\Python38-32\Lib.

Một cách nhanh chóng để kiểm tra xem tên module có đang được sử dụng hay không để `import`. Điều này có thể được thực hiện tại cửa sổ lệnh bằng cách gọi trình thông dịch với tùy chọn dòng lệnh `-c` ("mã thực thi") theo sau là câu lệnh import. Ví dụ: nếu chúng ta muốn xem liệu có module gọi là `Music.py` (hoặc một thư mục cấp cao nhất trong đường dẫn Python có tên là Music) hay không, chúng ta có thể nhập như sau vào cửa sổ lệnh:

```
python -c "import Music"
```

Nếu gặp ngoại lệ `ImportError`, ta biết rằng không có module hoặc thư mục cấp cao nhất nào của tên đó đang được sử dụng; bất kỳ đầu ra nào khác (hoặc không có) có nghĩa là tên được sử dụng.

Khi Python cần mã biên dịch byte-code của module, nó sẽ tự động tạo mã đó. Đầu tiên Python tìm kiếm một tệp có cùng tên với tệp .py của module nhưng có phần mở rộng .pyo, đây là phiên bản biên dịch mã byte được tối ưu hóa của module. Nếu không

có tệp .pyo (hoặc nếu nó cũ hơn tệp .py, nghĩa là, nếu nó đã lỗi thời), Python sẽ tìm kiếm tệp có phần mở rộng .pyc, đây là phiên bản biên dịch mã byte không được tối ưu hóa của module. Nếu Python tìm thấy phiên bản đã biên dịch mã byte cập nhật của module, nó sẽ tải nó; nếu không, Python tải tệp .py và biên dịch phiên bản đã biên dịch mã byte.

CHƯƠNG 5. MODULE NUMPY

Chương này sẽ trình bày chi tiết về NumPy. NumPy (viết tắt của Numerical Python) cung cấp một giao diện hiệu quả để lưu trữ và thao tác trên bộ đệm dữ liệu dày đặc. Theo một số cách, mảng NumPy giống như kiểu danh sách (`list`) tích hợp sẵn của Python, nhưng mảng NumPy cung cấp các thao tác lưu trữ và thao tác dữ liệu hiệu quả hơn nhiều khi các mảng lớn hơn về kích thước. Mảng NumPy tạo thành cốt lõi của gần như toàn bộ hệ sinh thái các công cụ khoa học dữ liệu bằng Python, vì vậy thời gian dành cho việc học cách sử dụng NumPy một cách hiệu quả sẽ rất có giá trị cho dù bạn quan tâm đến khía cạnh nào của khoa học dữ liệu. Để sử dụng Module NumPy trong chương trình, ta chèn bằng lệnh: `import numpy as py`. Ta thống nhất sử dụng bí danh `np` thay thế cho `numpy` trong suốt chương trình.

5.1. Kiểu dữ liệu trong NumPy

5.1.1. Tạo mảng trong NumPy

a. Tạo mảng từ danh sách của Python

Ta có thể tạo mảng bằng hàm array: `np.array(danh sách)`. Ví dụ:

```
>>> np.array([1, 4, 8])
array([1, 4, 8])
```

Chú ý, không giống như danh sách Python, mảng trong NumPy chứa các phần tử cùng một kiểu. Nếu các kiểu không khớp, NumPy sẽ ép kiểu nếu có thể.

Nếu muốn chỉ rõ kiểu dữ liệu trong lúc tạo mảng, ta có thể dùng từ khóa `dtype` như sau:

```
>>> np.array([1, 2, 3, 4], dtype='float32')
array([1., 2., 3., 4.], dtype=float32)
```

Mảng trong NumPy có thể nhiều chiều; đây là một cách khởi tạo mảng nhiều chiều bằng cách sử dụng danh sách trong danh sách:

```
>>> np.array([range(i, i + 3) for i in [2, 4, 6]])
array([[2, 3, 4],
```

```
       [4, 5, 6],
       [6, 7, 8]])
```

Các danh sách bên trong được coi là các hàng của mảng hai chiều.

b. Tạo mảng từ đầu

Đối với các mảng lớn hơn, việc tạo mảng từ đầu sẽ hiệu quả hơn bằng các đoạn chương trình được tích hợp sẵn trong NumPy. Dưới đây là một số ví dụ:

Tạo mảng có độ dài 10 với các giá trị ban đầu là số 0:

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

Tạo mảng hai chiều kích thước 3x5 với mỗi phần tử là số 1:

```
>>> np.ones((3, 5), dtype=float)
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

Tạo mảng hai chiều kích thước 3x5 với các giá trị là 3.14:

```
>>> np.full((3, 5), 3.14)
array([[3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

Tạo mảng một chiều các giá trị là 1 dãy bắt đầu từ 0 → 18, bước nhảy là 2:

```
>>> np.arange(0, 20, 2)
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Tạo mảng một chiều có 5 phần tử với các giá trị cách đều nhau trong phạm vi từ 0 đến 1:

```
>>> np.linspace(0, 1, 5)
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Tạo một mảng hai chiều 3x3 các giá trị ngẫu nhiên trong phạm vi từ 0 và 1 (hàm phân phối đều):

```
>>> np.random.random((3, 3))
array([[0.19503399, 0.81976534, 0.52584223],
       [0.2279521 , 0.34947431, 0.08257403],
       [0.54006605, 0.84780637, 0.99532485]])
```

Tạo một mảng hai chiều 3x3 các giá trị ngẫu nhiên theo phân phối chuẩn với giá

trị trung bình là 0 và độ lệch chuẩn 1:

```
>>> np.random.normal(0, 1, (3, 3))
array([[-0.68403104,  0.06443678, -1.01008137],
       [ 1.10911346, -0.68694069, -0.51366213],
       [-1.46013306,  0.60035975, -1.05455545]])
```

Tạo một mảng hai chiều 3x3 các số nguyên ngẫu nhiên trong khoảng [0, 10):

```
>>> np.random.randint(0, 10, (3, 3))
array([[9, 4, 4],
       [1, 9, 0],
       [3, 1, 7]])
```

Tạo ma trận đơn vị cấp 3x3:

```
>>> np.eye(3)
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Tạo mảng một chiều kích thước là 3 với các giá trị bất kì:

```
>>> np.empty(3)
array([1., 1., 1.])
```

c. Các kiểu dữ liệu trong NumPy sử dụng cho các hàm tạo mảng

Tên kiểu	Mô tả
bool_	Kiểu logic (True or False)
int_	Kiểu số nguyên ngầm định (32 hoặc 64 bit)
intc	Giống kiểu số nguyên trong C (32 hoặc 64 bit)
intp	Số nguyên làm chỉ số (giống kiểu ssize_t trong C, 32 hoặc 64 bit)
int8	Byte (–128 → 127)
int16	Integer (–32768 to 32767)
int32	Integer (–2147483648 to 2147483647)
int64	Integer (–9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Viết tắt của float64
float16	Kiểu float chính xác một nữa: 1 bit dấu, số mũ 5 bit, phần giá trị 10 bit
float32	Số thực chính xác đơn: 1 bit dấu, số mũ 8 bit, phần định trị 23 bit
float64	float chính xác kép: 1 bit dấu, số mũ 11 bit, phần định trị 52 bit
complex_	Viết tắt của complex128

complex64	Số phức, được biểu diễn bằng hai số thực 32 bit
complex128	Số phức, được biểu diễn bằng hai số nổi 64 bit

5.2. Các thao tác cơ bản

Thao tác dữ liệu trong Python gần như đồng nghĩa với thao tác mảng NumPy: ngay cả các công cụ mới hơn như Pandas cũng được xây dựng xung quanh mảng NumPy. Phần này sẽ trình bày một số ví dụ sử dụng thao tác mảng NumPy để truy cập dữ liệu và các mảng con, đồng thời để tách, định hình lại và nối các mảng.

5.2.1. Một số thao tác cơ bản

Thuộc tính của mảng: Xác định kích thước, hình dạng, mức tiêu thụ bộ nhớ và kiểu dữ liệu của mảng

Chỉ số mảng: Lấy và đặt giá trị của các phần tử mảng riêng lẻ

Cắt mảng: Lấy và thiết lập các mảng con nhỏ hơn trong một mảng lớn hơn

Định hình lại mảng: Thay đổi hình dạng của một mảng nhất định

Nối và tách mảng: Kết hợp nhiều mảng thành một và chia một mảng thành nhiều

5.2.2. Thuộc tính của mảng trong NumPy

Mỗi mảng có các thuộc tính `ndim` (số chiều), `shape` (kích thước của mỗi chiều) và `size` (tổng kích thước của mảng). Ngoài ra còn có thuộc tính `dtype` cho biết kiểu dữ liệu của từng phần tử, `itemsize` kích thước mỗi phần tử trong mảng, `nbytes` là tổng số byte của cả mảng. Ví dụ:

```
import numpy as np
x = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional
print(x)
print("x ndim: ", x.ndim)
print("x shape:", x.shape)
print("x size: ", x.size)
print("dtype:", x.dtype)
print("itemsize:", x.itemsize, "bytes")
print("nbytes:", x.nbytes, "bytes")
```

Kết quả xuất hiện như sau:

```
[[[0 8 1 2 5]
  [4 7 7 6 0]
  [7 3 8 6 3]
  [4 1 7 1 1]]
```

```
 [[1 2 8 6 0]
  [7 4 4 9 2]
  [1 4 5 2 9]
  [4 5 7 4 6]]

 [[7 0 3 8 5]
  [1 4 2 4 7]
  [9 9 3 6 0]
  [4 5 8 8 5]]]
x ndim:  3
x shape: (3, 4, 5)
x size:  60
dtype: int32
itemsize: 4 bytes
nbytes: 240 bytes
```

5.2.3. Truy cập từng phần tử của mảng

Mảng một chiều, trong NumPy có thể được truy cập giá trị thứ `i` (đếm từ 0) bằng cách chỉ định chỉ mục mong muốn trong dấu ngoặc vuông, giống như với danh sách Python, ví dụ:

```
>>> x = np.random.randint(10, size=8)
array([8, 7, 9, 0, 8, 2, 9, 0])
>>> x[1]
7
```

Đối với mảng hai chiều, truy cập theo bộ chỉ số cách nhau dấu phẩy, ví dụ:

```
x = np.random.randint(10, size=(3, 4))
array([[1, 5, 2, 0],
       [6, 3, 4, 6],
       [4, 4, 1, 6]])
>>> x[1,2]
4
```

Ta có thể sửa đổi giá trị của phần tử mảng thông qua chỉ số, ví dụ:

```
>>> x[1, 2] = 100
```

5.2.4. Trích chọn mảng

a. Trích mảng con một chiều

Ta có thể trích xuất mảng con một chiều bằng cú pháp như sau:

```
x[start:stop:step]
```

Trong đó `start` là chỉ số bắt đầu, `stop` là chỉ số kết thúc và `step` là bước nhảy. Ví dụ:

```
>>>x = np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[:5]
array([0, 1, 2, 3, 4])
>>> x[5:]
array([5, 6, 7, 8, 9])
>>> x[4:7]
array([4, 5, 6])
>>> x[::2]
array([0, 2, 4, 6, 8])
>>> x[1::2]
array([1, 3, 5, 7, 9])
>>> x[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
>>> x[5::-2]
array([5, 3, 1])
```

b. Trích mảng con nhiều chiều

Trích mảng con nhiều chiều giống như trích mảng con một chiều. Các chiều được ngăn cách nhau dấu phẩy.

```
>>>x = np.random.randint(10, size=(3, 4))
>>> x
array([[4, 1, 1, 8],
       [5, 5, 3, 3],
       [7, 6, 5, 9]])
>>> x[:2, :3]
array([[4, 1, 1],
       [5, 5, 3]])
>>> x[:3, ::2]
array([[4, 1],
       [5, 3],
       [7, 5]])
>>> x[::-1, ::-1]
array([[9, 5, 6, 7],
       [3, 3, 5, 5],
       [8, 1, 1, 4]])
```

c. Truy cập dòng, cột của mảng hai chiều

Ta có thể truy cập các hàng hoặc cột đơn của một mảng bằng cách kết hợp chỉ số và trích chọn mảng:

```
>>> x[:,0]
array([4, 5, 7]) #cột đầu tiên của mảng
>>> x[0,:]
array([4, 1, 1, 8]) #dòng đầu tiên của mảng
```

Đối với dòng, ta có thể bỏ qua dấu hai chấm:

```
>>> x[0]
array([4, 1, 1, 8])
```

d. Mảng con không sao chép

Một điều quan trọng và hữu ích cần biết về trích chọn mảng là trả về các dạng xem thay vì bản sao của dữ liệu mảng. Đây là một điểm khác biệt mảng NumPy khác với `list` trong sách Python: trong `list`, các trích chọn sẽ tạo bản sao. Ví dụ:

```
>>> x_copy = x[:2,:2]
>>> x_copy
array([[4, 1],
       [5, 5]])
>>> x_copy[0,0] = 100
>>> x_copy
array([[100,   1],
       [  5,   5]])
>>> x
array([[100,   1,   1,   8],
       [  5,   5,   3,   3],
       [  7,   6,   5,   9]])
```

e. Sao chép một mảng

Để sao chép một mảng ta sử dụng phương thức `copy()`.

```
>>> x_copy = x[:2,:2].copy()
>>> x_copy
array([[100,   1],
       [  5,   5]])
>>> x_copy[0,0]=1
>>> x_copy
array([[1, 1],
       [5, 5]])
>>> x
array([[100,   1,   1,   8],
```

```
       [  5,   5,   3,   3],
       [  7,   6,   5,   9]])
```

5.2.5. Định hình lại mảng

Một thao tác hữu ích khác là định hình lại các mảng. Cách linh hoạt nhất để làm điều này là với phương thức `reshape()`. Ví dụ: nếu bạn muốn đặt các số từ 1 đến 9 trong lưới 3 × 3, bạn có thể làm như sau:

```
>>> x = np.arange(1, 10)
>>> x
array([1, 2, 3, 4, 5, 6, 7, 8, 9]) #mảng một chiều
>>> x.reshape(3, 3) # mảng hai chiều 3x3
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Lưu ý rằng để điều này hoạt động, kích thước của mảng ban đầu phải khớp với kích thước của mảng được định hình lại. Nếu có thể, phương thức định hình lại sẽ sử dụng chế độ xem không sao chép của mảng ban đầu, nhưng với các bộ đệm bộ nhớ không liền nhau, điều này không phải lúc nào cũng đúng.

Một kiểu định hình lại phổ biến khác là chuyển đổi mảng một chiều thành ma trận hàng hoặc cột hai chiều. Bạn có thể thực hiện việc này bằng phương pháp định hình lại hoặc dễ dàng hơn bằng cách sử dụng từ khóa `newaxis` trong một thao tác trích mảng:

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

Hoặc:

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

5.3. Ghép mảng và chia mảng

5.3.1. Ghép nối các mảng

Việc nối, hoặc nối hai mảng trong NumPy, chủ yếu được thực hiện thông qua hàm `np.concatenate()`, `np.vstack()` và `np.hstack()`. Hàm `np.concatenate()` lấy một bộ (`tuple`) hoặc danh sách các mảng làm đối số đầu tiên của nó, ví dụ:

```
>>> x = np.array([1, 2, 3])
>>> y = np.array([3, 2, 1])
>>> np.concatenate((x, y))
array([1, 2, 3, 3, 2, 1])
```

Hoặc:

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

Tương tự với mảng hai chiều, tuy nhiên ta cần thêm một tham số là `axis = {0, 1, …}` để cho biết nối theo hàng hay cột.

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

Khi ghép mảng có số chiều không giống nhau, ta sử dụng hàm `np.vstack()`, `np.hstack()`, `np.dstack()`.

```
>>> x = np.array([1, 2, 3])
>>> grid = np.array([[9, 8, 7],[6, 5, 4]])
>>> np.vstack([x, grid])   #ghép mảng theo hàng.
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
>>> y = np.array([[99], [99]])
>>> np.hstack([grid, y])   #ghép mảng theo cột.
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

5.3.2. Tách mảng

Ngược lại với ghép mảng là tách, được thực hiện bởi các hàm `np.split()`, `np.hsplit()` và `np.vsplit()`. Đối với mỗi hàm này, ta có thể gởi danh sách các chỉ số cho điểm phân tách:

```
>>> x = np.array([1, 2, 3, 99, 99, 3, 2, 1])
>>> x1, x2, x3 = np.split(x, [3, 5])
>>> x1
array([1, 2, 3])
>>> x2
array([99, 99])
>>> x3
array([3, 2, 1])
```

Chú ý: nếu có n điểm chia thì sẽ tạo ra `n+1` mảng con. Hàm `np.hspit()`, `np.vspit()` cũng tương tự:

```
>>> grid = np.arange(16).reshape((4, 4))
>>> grid
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> upper, lower = np.vsplit(grid, [2])
>>> upper
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> lower
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Tương tự hàm `np.hspit()`:

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

5.4. Tính toán trên mảng trong NumPy

5.4.1. Hàm Ufunc

Đối với nhiều thao tác, NumPy cung cấp một giao diện thuận tiện. Đây là một thao tác vector hóa.

Ta có thể thực hiện điều này chỉ bằng cách thực hiện một thao tác trên mảng, thao tác này sau đó sẽ được áp dụng cho từng phần tử. Phương pháp vector hóa này được thiết kế để đẩy vòng lặp vào lớp đã biên dịch làm nền tảng cho NumPy, dẫn đến việc thực thi nhanh hơn nhiều.

Các thao tác được vector hóa trong NumPy được thực hiện thông qua các chàm gọi là Ufuncs, với mục đích chính là nhanh chóng thực hiện các thao tác lặp lại trên các giá trị trong mảng NumPy. Ufuncs cực kỳ linh hoạt, ta có thể thực hiện phép toán giữa vô hướng và mảng, hoặc chúng ta cũng có thể thoa tác giữa hai mảng:

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

Các phép tính sử dụng vectơ hóa thông qua Ufuncs gần như hiệu quả hơn so với thực hiện thông qua các vòng lặp Python, đặc biệt là khi các mảng kích thước lớn. Bất cứ khi nào ta thấy một vòng lặp như vậy trong tập lệnh Python, ta nên xem xét liệu nó có thể được thay thế bằng một biểu thức vector hóa hay không.

Các hàm Ufunc có hai loại, một loại phép toán một ngôi và phép toán hai ngôi.

Phép toán	Hàm	Ý nghĩa
+	np.add	Cộng (1 + 1 = 2)
-	np.subtract	Trừ (3 - 2 = 1)
-	np.negative	Trừ một ngôi (-2)

*	np.multiply	Nhân (2 * 3 = 6)
/	np.divide	Chia (3 / 2 = 1.5)
//	np.floor_divide	Chia lấy nguyên (3 // 2 = 1)
**	np.power	Lấy mũ (2 ** 3 = 8)
%	np.mod	Chia lấy dư (9 % 4 = 1

Các hàm số học:

Tên hàm	Ý nghĩa
np.abs	tính trị tuyệt đối
np.sin	sin
np.cos	cos
np.tan	tan
np.exp	ex
np.exp2	2x
np.power	x^y
np.log	ln
np.log2	logaric cơ số 2
np.log10	logaric cơ số 10
np.expm1	e^x-1
np.log1p	log(1+x)

5.4.2. Hàm Ufunc nâng cao

a. Thuộc tính out

Đối với các phép tính lớn, rất hữu ích khi có thể chỉ định mảng nơi lưu trữ kết quả của phép tính. Thay vì tạo một mảng tạm thời, bạn có thể sử dụng mảng này để ghi kết quả tính toán trực tiếp vào vị trí bộ nhớ mà bạn muốn. Đối với tất cả các hàm, bạn có thể thực hiện việc này bằng cách sử dụng đối số out của hàm:

```
>>> x = np.arange(5)
>>> y = np.empty(5)
>>> np.multiply(x, 10, out=y)
>>> print(y)
[ 0. 10. 20. 30. 40.]
```

```
y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)
[ 1. 0. 2. 0. 4. 0. 8. 0. 16. 0.]
```

b. Các hàm tổng hợp dữ liệu

Đối với hàm hai ngôi, có một số thao tác tổng hợp có thể được tính trực tiếp từ đối tượng. Ví dụ: nếu chúng ta muốn giảm một mảng trong một phép toán cụ thể, chúng ta có thể sử dụng phương thức `reduce()` của bất kỳ hàm ufunc nào.

```
np.add.reduce(x) # cộng các phần tử trong x
np.add.accumulate(x)  # cộng dồn, lưu kết quả mảng
array([ 0,  1,  3,  6, 10], dtype=int32)
```

c. Tích ngoài

Cuối cùng, bất kỳ hàm ufunc nào cũng có thể tính toán trên các cặp đầu vào khác nhau bằng phương pháp bên ngoài. Điều này cho phép bạn, trong một dòng, thực hiện những việc như tạo bảng cửu chương:

```
>>>x = np.arange(1, 10)
>>>print(np.multiply.outer(x, x))
[[ 1  2  3  4  5  6  7  8  9]
 [ 2  4  6  8 10 12 14 16 18]
 [ 3  6  9 12 15 18 21 24 27]
 [ 4  8 12 16 20 24 28 32 36]
 [ 5 10 15 20 25 30 35 40 45]
 [ 6 12 18 24 30 36 42 48 54]
 [ 7 14 21 28 35 42 49 56 63]
 [ 8 16 24 32 40 48 56 64 72]
 [ 9 18 27 36 45 54 63 72 81]]
```

5.5. Tổng hợp dữ liệu

Thông thường, khi gặp một lượng lớn dữ liệu, bước đầu tiên là tính thống kê tóm tắt cho dữ liệu được đề cập. Thống kê tóm tắt phổ biến nhất là giá trị trung bình và độ lệch chuẩn, cho phép bạn tóm tắt các giá trị "điển hình" trong tập dữ liệu, nhưng các tổng hợp khác cũng hữu ích (tổng, tích, trung bình, tối thiểu và tối đa, lượng tử, v.v.).

5.5.1. Các hàm cơ bản

Tên hàm	Ý nghĩa
np.sum	Tính tổng các phần tử
np.prod	Tính tích các phần tử

np.mean Tính trung bình các phần tử

np.std Tính độ lệch chuẩn

np.var Tính phương sai

np.min Giá trị nhỏ nhất

np.max Giá trị lớn nhất

np.argmin Tìm chỉ số giá trị nhỏ nhất

np.argmax Tìm chỉ số giá trị lớn nhất

np.median Tính trung vị của các phần tử

np.percentile Tính toán thống kê dựa trên thứ hạng của các phần tử

np.any Kiểm tra có hay không bất kì phần tử nào True

np.all Kiểm tra tất cả phần tử là True hay không

Ví dụ:

```
>>> L = np.random.random(100)
>>> np.sum(L)
45.39139629465517
>>> np.min(L), np.max(L)
(0.001194272267084573, 0.988071027436072)
>>> np.mean(L)
0.45391396294655173
>>> np.std(L)
0.2595001391662244
>>> np.argmin(L)
35
>>> np.argmax(L)
43
>>> np.var(L)
0.06734032222728982
>>> np.prod(L)
1.8776521239098422e-47
>>> np.median(L)
0.3883295033424118
```

5.5.2. Tổng hợp mảng nhiều chiều

Một loại tổng hợp phổ biến là tổng hợp dọc theo một hàng hoặc cột. Giả sử bạn có một số dữ liệu được lưu trữ trong mảng hai chiều:

```
>>> M = np.random.random((3, 4))
>>> print(M)
[[0.47866336 0.48494251 0.34255075 0.69707551]
 [0.68205182 0.76806745 0.99140357 0.86857063]
 [0.58486315 0.44441676 0.55308302 0.87073828]]
```

Theo mặc định, mỗi hàm tổng hợp NumPy thực hiện trên toàn bộ mảng:

```
>>> M.sum()
7.766426813578419
```

Các hàm tổng hợp lấy một đối số bổ sung axis chỉ định chiều cần tính toán. Ví dụ: chúng ta có thể tìm giá trị nhỏ nhất trong mỗi cột bằng cách chỉ `axis = 0`:

```
>>> M.min(axis=0)
array([0.47866336, 0.44441676, 0.34255075, 0.69707551])
```

Tương tự:

```
>>> M.max(axis=1)
array([0.69707551, 0.99140357, 0.87073828])
```

5.5.3. Ví dụ tổng hợp dữ liệu

Ví dụ đơn giản, hãy xem xét chiều cao của tất cả các tổng thống Hoa Kỳ. Dữ liệu này có sẵn trong tệp President_heights.csv trong thư mục data chứa danh sách nhãn và giá trị được phân tách bằng dấu phẩy đơn giản:

```
    order                    name  height(cm)
0       1       George Washington         189
1       2              John Adams         170
2       3        Thomas Jefferson         189
3       4           James Madison         163
...
```

Để đọc dữ liệu từ file này, ta sử dụng hàm `read_csv()` của gói pandas.

```
>>> import pandas as pd
>>> import numpy as np
>>> data = pd.read_csv('data/president_heights.csv')
>>> heights = np.array(data['height(cm)'])
>>> print(heights)
[189 170 189 163 183 171 185 168 173 183 173 173 175 178 183 193 178
173 174 183 183 168 170 178 182 180 183 178 182 188 175 179 183 193
182 183 177 185 188 188 182 185]
```

Bây giờ ta có thể thực hiện một vài thao tác thống kê tóm tắt như sau:

```
>>> print("Mean height: ", heights.mean())
Mean height:  179.73809523809524
>>> print("Standard deviation:", heights.std())
Standard deviation: 6.931843442745892
>>> print("Minimum height: ", heights.min())
Minimum height:  163
>>> print("Maximum height: ", heights.max())
Maximum height:  193
```

Hoặc ta có thể sử dụng các hàm tổng hợp dữ liệu theo cú pháp sau:

```
>>> print("25th percentile: ", np.percentile(heights, 25))
25th percentile:  174.25
>>> print("Median: ", np.median(heights))
Median:  182.0
>>> print("75th percentile: ", np.percentile(heights, 75))
75th percentile:  183.0
```

5.6. Tính toán trên mảng kích thước khác nhau (Broadcasting)

5.6.1. Giới thiệu Broadcasting

Broadcasting cho phép các loại phép toán hai ngôi được thực hiện trên các mảng có kích thước khác nhau. Ví dụ đơn giản:

```
>>> import numpy as np
>>> a = np.array([0, 1, 2])
>>> a+5
array([5, 6, 7])
```

Chúng ta có thể coi đây là một thao tác kéo dài hoặc nhân đôi giá trị 5 vào mảng [5, 5, 5] và thêm kết quả. Ta có thể áp điều này với mảng có chiều nhiều hơn.

```
>>> M = np.ones((3, 3))
array([[ 1., 1., 1.],
       [ 1., 1., 1.],
       [ 1., 1., 1.]])
>>> M + a
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

Ở ví dụ trên, mảng a tự động mở rộng số chiều cho bằng `M` và sau đó thực hiện phép toán cộng. Ví dụ khác:

```
>>> a = np.arange(3)
```

```
array([0, 1, 2])
>>> b = np.arange(3)[:, np.newaxis]
array([[0],
       [1],
       [2]])
>>> a + b
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

Trường hợp này mảng a mở rộng thêm số dòng, mảng b mở rộng thêm số cột cho bằng nhau rồi thực hiện thao tác cộng.

5.6.2. Các quy tắc broadcasting

a. Một bộ quy tắc để xác định sự tương tác giữa hai mảng:

• Quy tắc 1: Nếu hai mảng khác nhau về số chiều của chúng, thì mảng có chiều ít hơn sẽ được đệm bằng 1 ở phía đầu (bên trái) của nó.

• Quy tắc 2: Nếu mảng không trùng nhau về chiều nào thì mảng có chiều là 1 sẽ được kéo dài để trùng chiều với mảng còn lại.

• Quy tắc 3: Nếu trong bất kỳ chiều nào mà kích thước không bằng nhau và không bằng 1, lỗi sẽ được phát sinh.

b. Các ví dụ:

- Ví dụ 1:

```
>>> M = np.ones((2, 3))
>>> M
array([[1., 1., 1.],
       [1., 1., 1.]])
>>> a = np.arange(3)
>>> a
array([0, 1, 2])
```

Khi đó hình dạng của mảng M và a như sau:

```
>>> M.shape
(2, 3)
>>> a.shape
(3,)
```

Để thực hiện phép toán hai ngôi trên M và a, ta có các quy tắt sau:

Quy tắt 1: Mảng a có số chiều ít hơn nên sẽ chèn thêm số chiều ở phía trước (trái) số 1, tức là `a.shape = (1, 3)`.

Quy tắc 2: Số chiều đầu tiên của hai mảng M và a không giống nhau, do đó nó kéo dài chiều của mảng a (ít hơn) bằng mảng M, tức là `a.shape = (2, 3)`. Việc kéo dài mảng a thực chất là bổ sung thêm một dòng mới có giá trị giống dòng đầu.

Khi hình dạng hai mảng M và a giống nhau, ta có thể thực hiện phép toán hai ngôi bình thường. Ví dụ `M + a`:

```
>>> M+a
array([[1., 2., 3.],
       [1., 2., 3.]])
```

- Ví dụ 2:

```
>>> a = np.arange(3).reshape((3, 1))
>>> a
array([[0],
       [1],
       [2]])
>>> b = np.arange(3)
>>> b
array([0, 1, 2])
```

Hình dạng của mảng a và b như sau:

```
>>> a.shape
(3, 1)
>>> b.shape
(3,)
```

Để thực hiện phép toán hai ngôi trên mảng a và b, ta phải hiểu các quy tắt sau:

Quy tắt 1: Mảng b có số chiều ít hơn nên phải chèn thêm số chiều ở phía trước (trái) số 1. Khi đó `b.shape = (1, 3)`.

Quy tắt 2: kéo dài chiều ít hơn trong mỗi mảng a và b để chúng bằng nhau, tức là `a.shape = (3, 1)` → `a.shape=(3, 3)`, thực hiện bằng cách bổ sung thêm các cột bằng có giá trị giống cột đầu tiên. Tương tự, `b.shape = (1, 3)` → `b.shape = (3, 3)`, thực hiện bằng cách bổ sung thêm các dòng có giá trị giống dòng đầu tiên.

Lúc này mảng a và b có kích thước (hình dạng) giống nhau nên có thể thực hiện các phép toán hai ngôi, chẳng hạn a+b:

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

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

- Ví dụ 3:

```
>>> M = np.ones((3, 2))
>>> M
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
>>> a = np.arange(3)
>>> a
array([0, 1, 2])
```

Khi đó, hình dạng của `M` và `a` như sau:

```
>>> M.shape
(3, 2)
>>> a.shape
(3,)
```

Tiếp tục áp dụng quy tắt 1: Mảng `a` có số chiều ít hơn nên sẽ chèn thêm số chiều ở phía trước (trái) số 1. Khi đó `a.shape = (1, 3)`.

Áp dụng quy tắt 2: Số chiều `a` ít hơn `M` sẽ kéo dài để bằng `M`, tức là `a.shape=(1, 3)` → `a.shape = (3, 3)`. Số chiều thứ hai của `M` ít hơn số chiều thứ hai của `a`, tuy nhiên kích thước bằng 2 nên không thể kéo dài, vì vậy `M.shape` vẫn là (3, 2).

Lúc này, hình dạng của `M` và `a` khác nhau nên không thể thực hiện phép toán hai ngôi, chẳng hạn `M+a` sẽ phát sinh lỗi:

```
>>> M + a
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    M + a
ValueError: operands could not be broadcast together with shapes
(3,2) (3,)
```

5.6.3. Ứng dụng của broadcasting trong thực tế

a. Đưa mảng về tâm

Giả sử ta có một mảng gồm 10 quan sát, mỗi quan sát bao gồm 3 giá trị, ta sẽ lưu trữ dữ liệu này trong một mảng 10 × 3:

```
>>> X = np.random.random((10, 3))
```

Ta sẽ tính trung bình cho chiều đầu tiên (tính trung bình các giá trị trên mỗi dòng, tức là trên 1 cột) bằng hàm tổng hợp `mean()`.

```
Xmean = X.mean(0)
>>> Xmean
array([0.55141331, 0.41442454, 0.51202068])
```

Và bây giờ chúng ta có thể căn giữa mảng `X` bằng cách trừ giá trị trung bình (đây là thao tác *broadcasting*):

5.7. So sánh, Mặt nạ và Logic Boolean

5.7.1. Các phép toán so sánh

NumPy cũng cài đặt các toán tử so sánh như <(nhỏ hơn) và> (lớn hơn) dưới dạng các hàm trên cả mảng. Kết quả của các toán tử so sánh này luôn là một mảng có kiểu dữ liệu Boolean. Tất cả sáu phép toán so sánh tiêu chuẩn:

```
>>> x = np.array([1, 2, 3, 4, 5])
>>> x < 3 # less than
array([ True,  True, False, False, False])
>>> x > 3 # greater than
array([False, False, False,  True,  True])
>>> x <= 3
array([ True,  True,  True, False, False])
>>> x >= 3
array([False, False,  True,  True,  True])
>>> x == 3
array([False, False,  True, False, False])
>>> x != 3
array([ True,  True, False,  True,  True])
```

Giống như trong trường hợp các toán tử số học, các toán tử so sánh được thực hiện dưới dạng hàm Ufuncs trong NumPy; ví dụ: khi bạn viết `x<3`, bên trong NumPy sử dụng `np.less (x, 3)`. Tóm tắt các toán tử so sánh và Ufunc tương đương của chúng được hiển thị ở đây:

Phép toán	Hàm tương đương
`==`	`np.equal`
`!=`	`np.not_equal`
`<`	`np.less`
`<=`	`np.less_equal`

>	np.greater
>=	np.greater_equal

Ví dụ trên thực hiện trên mảng một chiều, mảng hai chiều cũng tương tự:

```
>>> x = np.random.randint(10, size=(3, 4))
>>> x
array([[1, 3, 8, 9],
       [1, 0, 9, 6],
       [0, 1, 4, 1]])
>>> x<6
array([[ True,  True, False, False],
       [ True,  True, False, False],
       [ True,  True,  True,  True]])
```

5.7.2. Các thao tác thực hiện trên mảng logic

Xét mảng x:

```
>>> x = np.random.randint(10, size=(3, 4))
>>> x
array([[7, 8, 7, 2],
       [3, 8, 3, 1],
       [1, 5, 7, 2]])
```

Hàm np.count_nonzero() đếm phần tử True:

```
>>> np.count_nonzero(x<6)
7
>>> np.count_nonzero(x<6, axis=0) #đếm trên từng cột
array([2, 1, 1, 3], dtype=int64)
>>> np.count_nonzero(x<6, axis=1) #đếm trên từng dòng
array([1, 3, 3], dtype=int64)
```

Có cách khác để đếm số lượng này đó là sử dụng hàm sum. Khi đó đó False là 0, True là 1.

```
>>> np.sum(x<6)
7
```

Để kiểm tra các phần tử có phải là True hay không, ta sử dụng hàm np.all():

```
>>> np.all(x<6)
False
>>> np.all(x<6, axis=0)
array([False, False, False,  True])
>>> np.all(x<6, axis=1)
```

```
array([False, False, False])
```

Để kiểm tra có bất kì phần tử nào là `True` hay không, ta sử dụng hàm `np.any()`:

```
>>> np.any(x<6)
True
>>> np.any(x<6, axis=0)
array([ True,  True,  True,  True])
>>> np.any(x<6, axis=1)
array([ True,  True,  True])
```

5.7.3. Toán tử logic

Ta có thể sử dụng toán tử logic để ghép các điều kiện: `&` (and), `|` (or), `~` (not), `^` (xor). Các phép toán này cũng có các hàm tương ứng:

Phép toán	Hàm
`&`	`np.bitwise_and`
`|`	`np.bitwise_or`
`^`	`np.bitwise_xor`
`~`	`np.bitwise_not`

```
>>> np.count_nonzero((x>3) & (x<6))
1
```

5.7.4. Mảng logic như mặt nạ

Sử dụng mảng logic như mặt nạ để chọn ra tập con dữ liệu. Giả sử ta lấy những phần tử trong mảng `x` có giá trị nhỏ hơn 5:

```
>>> x[x<5]
array([2, 3, 3, 1, 1, 2])
```

5.7.5. Ví dụ tổng hợp

Ta có một chuỗi dữ liệu đại diện cho lượng mưa mỗi ngày trong một năm ở thành phố Seattle vào năm 2014. Dữ liệu này lưu trong file Seattle2014.csv (file csv là file văn bản mà các giá trị cách nhau bởi dấu phẩy). Sử dụng hàm `read_csv()` của gói pandas (tìm hiểu sau) để đọc file này.

```
>>> data = pd.read_csv('data\seattle2014.csv')
STATION    STATION_NAME    DATE       PRCP       ...
GHCND      SEATTLE ...     20140101   0          ...
GHCND      SEATTLE ...     20140102   41         ...
```

Data là mảng hai chiều chứa các thông tin về dữ liệu các ngày mưa, để lấy cột chứa

thông tin về lượng mưa, ta sử dụng cú pháp `data['tên cột'].values`.

```
>>> rain = data['PRCP'].values
```

`rain` chứa thông tin lượng mưa từ ngày 1/1/2014 đến 31/12/2014, đánh chỉ số từ 0→ 364.

Dựa vào mảng `rain`, ta có thể trả lời một số câu hỏi, chẳng hạn như: "có bao nhiêu ngày không mưa trong năm 2014?"

```
>>> np.count_nonzero(rain==0)
215
```

Hoặc số ngày mưa:

```
>>> np.sum(rain>0)
150
```

Tiếp tục viết các câu lệnh trả lời các câu hỏi sau:

1. Số ngày mưa của quý I (ngày thứ 1/1/2014 đến ngày ngày 31/3/2014)?
2. Số ngày không mưa của quý I?
3. Lượng mưa trung bình của quý I bao nhiêu mm?
4. Lượng mưa lớn nhất trong quý I?
5. Lượng mưa nhỏ nhất trong quý I?
6. Tổng lượng mưa của quý I?
7. Số ngày mưa nhiều hơn lượng mưa trung bình?
8. Số ngày mưa ít hơn lượng mưa trung bình?
9. Ngày có lượng mưa nhiều nhất quý I?
10. Liệt kê những ngày có lượng mưa nhiều nhất.
11. Vẽ biểu đồ thể hiện lượng mưa của quý I.
12. Các câu hỏi tương tự cho các quý còn lại (II, III, IV).

CHƯƠNG 6. THAO TÁC DỮ LIỆU VỚI PANDAS

Trong chương trước, ta đã đi sâu vào chi tiết về NumPy và đối tượng `ndarray` của nó, cung cấp khả năng lưu trữ và thao tác hiệu quả với các mảng với các kiểu dày đặc trong Python. Ở đây, ta xem xét chi tiết cấu trúc dữ liệu do thư viện Pandas cung cấp. Pandas là một gói mới hơn được xây dựng trên NumPy và cung cấp các thao tác hiệu quả trên đối tượng `DataFrame`. `DataFrame` về cơ bản là mảng đa chiều kèm theo các nhãn hàng và cột, và thường có các kiểu không đồng nhất và/hoặc dữ liệu bị thiếu. Cũng như cung cấp giao diện thuận tiện để lưu trữ dữ liệu được gắn nhãn, Pandas cài đặt một số thao tác dữ liệu mạnh mẽ quen thuộc với người dùng trên nền tảng cơ sở dữ liệu và chương trình bảng tính.

6.1. Cài đặt Pandas

Nếu sử dụng Anacoda thì Pandas đã được cài đặt kèm với nó. Nếu dùng Python, ta có thể gõ lệnh: '`python -m pip install pandas`' từ cửa sổ lệnh (command prompt).

Khi đã cài đặt Pandas, ta có thể `import` và kiểm tra phiên bản bằng lệnh:

```
>>> import pandas
>>> pandas.__version__
'1.1.3'
```

Ta thường import pandas với bí danh là pd:

```
>>> import pandas as pd
```

Khi đã `import pandas` vào chương trình, ta có thể sử dụng thuộc tính `__doc__` để tìm hiểu thêm các thông tin từ nhà cung cấp.

6.2. Giới thiệu các đối tượng trong Pandas

Ở cấp độ cơ bản, các đối tượng Pandas có thể được coi là phiên bản nâng cao của mảng có cấu trúc NumPy, trong đó các hàng và cột được xác định bằng nhãn thay vì chỉ số nguyên đơn giản. Pandas cung cấp một loạt các công cụ, phương thức và hàm hữu ích trên các ba cấu trúc dữ liệu cơ bản: `Series`, `DataFrame` và `Index`.

6.2.1. Đối tượng Series

`Series` là một mảng một chiều với dữ liệu được đánh chỉ số. Nó có thể được tạo từ một danh sách hoặc mảng như sau:

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0])
>>> data
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

Như ta thấy trong đầu ra trước đó, `Series` bao gồm dãy cả giá trị và dãy chỉ số mà chúng ta có thể truy cập bằng thuộc tính `values` và `index`. Thuộc tính `values` chỉ đơn giản là một mảng NumPy quen thuộc:

```
>>> data.values
array([0.25, 0.5 , 0.75, 1.  ])
```

`Index` là một đối tượng giống như mảng có kiểu là `pd.Index`.

```
>>> data.index
RangeIndex(start=0, stop=4, step=1)
```

Giống như với mảng NumPy, dữ liệu có thể được truy cập bằng chỉ số thông qua ký hiệu dấu ngoặc vuông quen thuộc trong Python:

```
>>> data[1]
0.5
>>> data[1:3]
1    0.50
2    0.75
dtype: float64
```

a. Series dưới dạng là một mảng NumPy

Ta thấy đối tượng `Series` về cơ bản có thể hoán đổi cho nhau với mảng một chiều NumPy. Sự khác biệt cơ bản là sự xuất hiện của chỉ số: trong khi mảng NumPy có chỉ số nguyên được xác định ngầm để truy cập các giá trị, thì `Series` có chỉ mục được xác định rõ ràng được liên kết với các giá trị.

Định nghĩa chỉ số rõ ràng này cung cấp thêm cho đối tượng `Series` các khả năng. Ví dụ như: chỉ số không cần phải là một số nguyên, có thể bao gồm bất kỳ kiểu nào. Chẳng hạn, ta có thể sử dụng chuỗi làm chỉ số:

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0],index=['a', 'b', 'c',
'd'])
```

```
>>> data
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

Và các giá trị có thể truy cập:

```
>>> data['b']
0.5
```

Ta cũng có thể sử dụng chỉ số không liên tục:

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0],index=[2, 5, 3, 7])
>>> data
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
>>> data[5]
0.5
```

b. Series dưới dạng là một từ điển đặc biệt

Từ điển là một cấu trúc ánh xạ các khóa tùy ý đến một tập hợp các giá trị tùy ý và `Series` cũng là một cấu trúc ánh xạ các khóa vào một tập hợp các giá trị. Chúng ta có thể làm cho `Series` tương tự từ điển bằng cách xây dựng đối tượng `Series` trực tiếp từ từ điển Python:

```
>>> population_dict = {'California': 38332521,'Texas': 26448193,'New
York': 19651127,'Florida': 19552860,'Illinois': 12882135}
>>> population = pd.Series(population_dict)
>>> population
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

Mặc định, `Series` sẽ được tạo với chỉ số được lấy từ các khóa được sắp xếp. Từ đó, có thể thực hiện truy cập mục theo kiểu từ điển điển hình:

```
>>> population['California']
```

```
38332521
```

Tuy nhiên, không giống như từ điển, `Series` cũng hỗ trợ các phép toán kiểu mảng chẳng hạn như:

```
>>> population['California':'Illinois']
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

c. Tạo Series từ cấu tử

Cú pháp: `pd.Series(data, index=index)`. Trong đó, `index` là tham số tùy chọn, `data` có thể là nhiều loại đối tượng.

`Data` có thể là danh sách hoặc là một mảng NumPy, nếu `index` là ngầm định thì chỉ số là các số nguyên:

```
>>> pd.Series([2, 4, 6])
0    2
1    4
2    6
dtype: int64
```

Nếu `data` là một giá trị, nó sẽ được lặp lại để điền vào tập chỉ số:

```
>>> pd.Series(5, index=[100, 200, 300])
100    5
200    5
300    5
dtype: int64
```

`data` có thể là một từ điển, khi đó chỉ số là tập các khóa của từ điển đã sắp xếp:

```
>>> pd.Series({2:'a', 1:'b', 3:'c'})
2    a
1    b
3    c
dtype: object
```

Chú ý, nếu tạo `Series` bằng từ điển mà ta chỉ thêm tham số `index` thì `Series` sẽ ưu tiên chỉ số của tham số `index`:

```
>>> pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```

```
3    c
2    a
dtype: object
```

6.2.2. Đối tượng DataFrame

a. DataFrame như là mảng trong NumPy

Nếu `Series` là tương tự của mảng một chiều với các chỉ số linh hoạt, thì `DataFrame` là tương tự của mảng hai chiều với cả chỉ số hàng và tên cột linh hoạt. Cũng giống như bạn có thể nghĩ về mảng hai chiều là một dãy các cột, mỗi cột là một `Series`. Để minh họa điều này, xét ví dụ:

```
>>> population_dict = {'California': 38332521,'Texas': 26448193,'New
York': 19651127,'Florida': 19552860,'Illinois': 12882135}
>>> population = pd.Series(population_dict)
>>> population
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
>>> area_dict = {'California': 423967, 'Texas': 695662, 'New York':
141297,'Florida': 170312, 'Illinois': 149995}
>>> area = pd.Series(area_dict)
>>> area
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
dtype: int64
```

Thời điểm này, ta đã có hai đối tượng `Series`: `population` chứa thông tin dân số và `area` chứa thông tin về diện tích của các bang tương ứng. Bây giờ ta tạo một bảng chứa ba cột biểu diễn thông tin về dân số và diện tích các bang tương ứng:

```
>>> states = pd.DataFrame({'population': population,'area': area})
>>> states
            population    area
California    38332521  423967
Texas         26448193  695662
New York      19651127  141297
```

```
Florida       19552860  170312
Illinois      12882135  149995
```

Giống như đối tượng `Series`, `DataFrame` có thuộc tính `index` cho phép truy cập vào các giá trị với chỉ số là nhãn:

```
>>> states.index
Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'],
dtype='object')
```

Ngoài ra, `DataFrame` có thuộc tính `columns`, là đối tượng chứa các nhãn cột:

```
>>> states.columns
Index(['population', 'area'], dtype='object')
```

b. DataFrame như là một từ điển

Tương tự, chúng ta cũng có thể coi `DataFrame` là một từ điển. Từ điển ánh xạ một khóa tới một giá trị, `DataFrame` ánh xạ tên cột với `Series` dữ liệu cột. Ví dụ: yêu cầu thuộc tính '`area`' trả về đối tượng `Series` chứa diện tích trước đó:

```
>>> states['area']
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Lưu ý điểm nhầm lẫn tiềm ẩn ở đây: trong mảng NumPy hai chiều, `data[0]` sẽ trả về dòng đầu tiên. Đối với `DataFrame`, `data['col0']` sẽ trả về cột đầu tiên. Do đó, có lẽ tốt hơn nên nghĩ về `DataFrames` như từ điển tổng quát hơn là mảng tổng quát, mặc dù cả hai cách xem xét tình huống đều có thể hữu ích.

c. Tạo DataFrame từ cấu tử

`DataFrame` có thể được xây dựng theo nhiều cách khác nhau. Sau đây ta sẽ đưa ra một số cách:

- Cách 1: từ đối tượng `Series`:

```
>>> pd.DataFrame(population, columns=['population'])
            population
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
```

```
Illinois      12882135
```

- Cách 2: từ danh sách các từ điển:

```
>>> data = [{'a': i, 'b': 2 * i} for i in range(3)]
>>> pd.DataFrame(data)
   a  b
0  0  0
1  1  2
2  2  4
```

Ngay cả khi một số khóa trong từ điển bị thiếu, Pandas sẽ điền chúng bằng giá trị NaN (tức là "không phải số"):

```
>>> pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
     a  b    c
0  1.0  2  NaN
1  NaN  3  4.0
```

- Cách 3: từ từ điển với khóa là tên cột và các giá trị là đối tượng `Series`:

```
>>> pd.DataFrame({'population': population,'area': area})
            population    area
California    38332521  423967
Texas         26448193  695662
New York      19651127  141297
Florida       19552860  170312
Illinois      12882135  149995
```

- Cách 4: từ mảng hai chiều của lớp NumPy: Với một mảng dữ liệu hai chiều, chúng ta có thể tạo `DataFrame` với bất kỳ tên cột và chỉ số nào được chỉ định. Nếu bị bỏ qua, một chỉ số số nguyên sẽ được sử dụng:

```
>>> pd.DataFrame(np.random.rand(3, 2),columns=['foo', 'bar'],
index=['a', 'b', 'c'])
        foo       bar
a  0.784516  0.062297
b  0.934050  0.503866
c  0.991754  0.640982
```

Khi bỏ qua các nhãn cột hoặc chỉ số:

```
>>> pd.DataFrame(np.random.rand(3, 2))
          0         1
0  0.673353  0.206474
1  0.864113  0.899713
```

```
2  0.376230  0.535731
```

- Cách 5: từ mảng cấu trúc của NumPy:

```
>>> A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
>>> A
array([(0, 0.), (0, 0.), (0, 0.)], dtype=[('A', '<i8'), ('B',
'<f8')])
>>> pd.DataFrame(A)
   A    B
0  0  0.0
1  0  0.0
2  0  0.0
```

6.2.3. Đối tượng Index

Ta thấy rằng cả đối tượng `Series` và `DataFrame` đều chứa một chỉ mục rõ ràng cho phép bạn tham chiếu và sửa đổi dữ liệu. Đối tượng `Index` này tự nó là một cấu trúc và nó có thể được coi là một mảng bất biến hoặc như một tập hợp có thứ tự. Ví dụ đơn giản, hãy tạo `Index` từ danh sách các số nguyên:

```
>>> ind = pd.Index([2, 3, 5, 7, 11])
>>> ind
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

a. Index như là một mảng bất biến

Đối tượng `Index` theo nhiều cách hoạt động giống như một mảng. Ví dụ: chúng ta có thể sử dụng ký hiệu lập chỉ mục Python chuẩn để truy xuất các giá trị hoặc trích chọn:

```
>>> ind
Int64Index([2, 3, 5, 7, 11], dtype='int64')
>>> ind[1]
3
>>> ind[::2]
Int64Index([2, 5, 11], dtype='int64')
```

Các đối tượng chỉ mục cũng có nhiều thuộc tính quen thuộc từ mảng NumPy:

```
>>> print(ind.size, ind.shape, ind.ndim, ind.dtype)
5 (5,) 1 int64
```

Một điểm khác biệt giữa các đối tượng `Index` và mảng NumPy là các chỉ số là bất biến, nghĩa là chúng không thể được sửa đổi thông qua các lệnh thông thường:

```
>>> ind[0]=0
Traceback (most recent call last):
```

```
  File "<pyshell#33>", line 1, in <module>
    ind[0]=0
  File "C:\Python\lib\site-packages\pandas\core\indexes\base.py",
line 4081, in __setitem__
    raise TypeError("Index does not support mutable operations")
TypeError: Index does not support mutable operations
```

Tính bất biến này làm cho việc chia sẻ chỉ số giữa nhiều `DataFrame` và mảng trở nên an toàn hơn, mà không có khả năng xảy ra các tác dụng phụ do vô tình sửa đổi chỉ mục.

b. Index như là tập hợp có thứ tự

Các đối tượng Pandas được thiết kế để tạo các phép toán thuận tiện chẳng hạn như kết nối (joins) giữa các tập dữ liệu. Đối tượng `Index` tuân theo nhiều quy ước được sử dụng bởi cấu trúc dữ liệu tập hợp được tích hợp sẵn của Python, do đó, hợp, giao, hiệu và tổ hợp có thể được tính toán theo cách quen thuộc:

```
>>> indA = pd.Index([1, 3, 5, 7, 9])
>>> indB = pd.Index([2, 3, 5, 7, 11])
>>> indA & indB
Int64Index([3, 5, 7], dtype='int64')
>>> indA | indB
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
>>> indA ^ indB
Int64Index([1, 2, 9, 11], dtype='int64')
```

6.3. Trích chọn dữ liệu

6.3.1. Trích chọn dữ liệu trong Series

Như chúng ta đã thấy trong phần trước, một đối tượng `Series` hoạt động theo nhiều cách giống như mảng NumPy một chiều và theo nhiều cách giống như một từ điển Python chuẩn. Điều này sẽ giúp chúng ta hiểu các mô hình lập chỉ mục và lựa chọn dữ liệu trong các mảng này.

a. Series như là một từ điển

Giống như một từ điển, đối tượng `Series` cung cấp ánh xạ từ một tập hợp các khóa đến một tập hợp các giá trị:

```
>>> data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c',
'd'])
>>> data
a    0.25
```

```
b    0.50
c    0.75
d    1.00
dtype: float64
>>> data['b']
0.5
```

Chúng tôi cũng có thể sử dụng các biểu thức và phương thức Python giống như từ điển để kiểm tra các khóa/chỉ số và giá trị:

```
>>> data['b']
0.5
>>> 'a' in data
True
>>> data.keys()
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> list(data.items())
[('a', 0.25), ('b', 0.5), ('c', 0.75), ('d', 1.0)]
```

Các đối tượng `Series` thậm chí có thể được sửa đổi với cú pháp giống như từ điển. Cũng giống như bạn có thể mở rộng từ điển bằng cách gán cho một khóa mới, ta có thể mở rộng `Series` bằng cách gán giá trị cho một chỉ mục mới:

```
>>> data['e'] = 1.25
>>> data
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

b. Series như mảng một chiều

`Series` xây dựng trên giao diện giống như từ điển và cung cấp lựa chọn mục theo kiểu mảng thông qua các cơ chế cơ bản giống như mảng NumPy, nghĩa là trích chọn, tạo mặt nạ và lập chỉ mục. Ví dụ về những điều này như sau:

```
>>> data['a':'c']
a    0.25
b    0.50
c    0.75
dtype: float64
>>> data[0:2]
```

```
a    0.25
b    0.50
dtype: float64
>>> data[(data > 0.3) & (data < 0.8)]
b    0.50
c    0.75
dtype: float64
>>> data[['a', 'e']]
a    0.25
e    1.25
dtype: float64
```

Trong số này, trích chọn có thể gây nhầm lẫn nhiều nhất. Lưu ý rằng khi ta đang trích mảng với một chỉ mục rõ ràng (tức là `data['a':'c']`), sẽ bao gồm mục cuối cùng, trong khi trích chọn bằng một chỉ mục ngầm định (tức là, `data[0:2]`), chỉ mục cuối cùng bị loại trừ.

c. Thuộc tính chỉ số: loc, iloc và ix

Các cú pháp trích chọn và lập chỉ số này có thể là nguyên nhân gây nhầm lẫn. Ví dụ: nếu `Series` có chỉ số là số nguyên rõ ràng, thì thao tác `data[1]` sẽ sử dụng các chỉ số rõ ràng, trong khi thao tác trích như `data[1:3]` sẽ sử dụng chỉ số ẩn kiểu Python.

```
>>> data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
>>> data
1    a
3    b
5    c
dtype: object
>>> data[1] # truy cập bằng chỉ số rõ
'a'
>>> data[1:3] #truy cập bằng chỉ số ẩn
3    b
5    c
dtype: object
```

Do sự nhầm lẫn tiềm ẩn này trong trường hợp chỉ số nguyên, Pandas cung cấp một số thuộc tính đặc biệt để hiển thị rõ ràng các cơ chế lập chỉ mục nhất định.

Đầu tiên, thuộc tính `loc` cho phép lập chỉ mục và trích chọn luôn tham chiếu đến chỉ mục rõ ràng:

```
>>> data.loc[1]
'a'
```

```
>>> data.loc[1:3]
1    a
3    b
dtype: object
```

Thuộc tính `iloc` cho phép lập chỉ mục và trích chọn luôn tham chiếu đến chỉ mục ẩn kiểu Python:

```
>>> data.iloc[1]
'b'
>>> data.iloc[1:3]
3    b
5    c
dtype: object
```

Thuộc tính lập chỉ mục thứ ba, `ix`, là một kết hợp của hai thuộc tính và đối với các đối tượng `Series` tương đương với lập chỉ mục dựa trên `[]` tiêu chuẩn. Mục đích của trình chỉ mục `ix` sẽ trở nên rõ ràng hơn trong ngữ cảnh của các đối tượng `DataFrame`.

6.3.2. Trích chọn dữ liệu trong DataFrame

Hãy nhớ lại rằng `DataFrame` hoạt động theo nhiều cách giống như một mảng hai chiều hoặc mảng cấu trúc, và theo những cách khác giống như một từ điển của các cấu trúc `Series` chia sẻ cùng một chỉ mục.

a. DataFrame như một từ điển

`DataFrame` như một từ điển của các đối tượng `Series` có liên quan. Quay lại ví dụ về các diện tích và dân số các tiểu bang:

```
>>> area = pd.Series({'California': 423967, 'Texas': 695662,
'New York': 141297, 'Florida': 170312, 'Illinois': 149995})
>>> pop = pd.Series({'California': 38332521, 'Texas': 26448193,
'New York': 19651127, 'Florida': 19552860, 'Illinois': 12882135})
>>> data = pd.DataFrame({'area':area, 'pop':pop})
>>> data
              area       pop
California  423967  38332521
Texas       695662  26448193
New York    141297  19651127
Florida     170312  19552860
Illinois    149995  12882135
```

Các `Series` riêng lẻ tạo nên các cột của `DataFrame` có thể được truy cập thông qua lập chỉ số kiểu từ điển là của tên cột:

```
>>> data['area']
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Tương tự, chúng ta có thể sử dụng quyền truy cập kiểu thuộc tính với tên cột là chuỗi:

```
>>> data.area
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Giống như với các đối tượng `Series` đã thảo luận trước đó, cú pháp kiểu từ điển này cũng có thể được sử dụng để sửa đổi đối tượng, trong trường hợp này là thêm một cột mới:

```
>>> data['density'] = data['pop'] / data['area']
>>> data
              area       pop     density
California  423967  38332521   90.413926
Texas       695662  26448193   38.018740
New York    141297  19651127  139.076746
Florida     170312  19552860  114.806121
Illinois    149995  12882135   85.883763
```

b. DataFrame như mảng hai chiều

Như đã đề cập trước đây, ta cũng có thể xem `DataFrame` như một mảng hai chiều nâng cao. Ta có thể kiểm tra mảng dữ liệu thô bằng cách sử dụng thuộc tính `values`:

```
>>> data.values
array([[4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [6.95662000e+05, 2.64481930e+07, 3.80187404e+01],
       [1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [1.49995000e+05, 1.28821350e+07, 8.58837628e+01]])
```

Để lấy dữ liệu trên dòng của mảng này, ta dùng chỉ số kèm với thuộc tính `values`:

```
>>> data.values[0]
array([4.23967000e+05, 3.83325210e+07, 9.04139261e+01])
```

Ta có thể chuyển vị ma trận dữ liệu này bằng thuộc tính T:

```
>>> data.T
           California         Texas      New York       Florida
Illinois
area     4.239670e+05  6.956620e+05  1.412970e+05  1.703120e+05
1.499950e+05
pop      3.833252e+07  2.644819e+07  1.965113e+07  1.955286e+07
1.288214e+07
density  9.041393e+01  3.801874e+01  1.390767e+02  1.148061e+02
8.588376e+01
```

Ta có thể truy xuất từng cột dữ liệu bằng cách chỉ nhãn chỉ số:

```
>>> data['area']
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Ta có thể sử dụng thuộc tính loc, iloc để truy cập dòng cột của mảng:

```
>>> data.iloc[0] # dòng đầu tiên
area       4.239670e+05
pop        3.833252e+07
density    9.041393e+01
Name: California, dtype: float64
>>> data.iloc[:,0] #cột đầu tiên
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Ta có thể áp dụng các phép toán so sánh để lọc dữ liệu:

```
>>> data.loc[data.density > 100]
            area       pop     density
New York  141297  19651127  139.076746
Florida   170312  19552860  114.806121
```

Ta có thể sửa đổi các giá trị của mảng bằng các cú pháp đã nêu:

```
>>> data.iloc[0, 2] = 90
>>> data
              area       pop     density
California  423967  38332521   90.000000
Texas       695662  26448193   38.018740
New York    141297  19651127  139.076746
Florida     170312  19552860  114.806121
Illinois    149995  12882135   85.883763
```

6.4. Các phép toán trên dữ liệu trong Pandas

6.4.1. Bảo toàn chỉ số

Vì Pandas được thiết kế để hoạt động với NumPy nên mọi hàm của NumPy sẽ hoạt động trên các đối tượng Pandas: `Series` và `DataFrame`. Hãy bắt đầu bằng cách định nghĩa một `Series` và `DataFrame` đơn giản để chứng minh điều này:

```
>>> ser = pd.Series(np.random.randint(0, 10, 4))
>>> ser
0    9
1    7
2    8
3    6
dtype: int32
>>> df = pd.DataFrame(np.random.randint(0, 10, (3, 4)),
             columns=['A', 'B', 'C', 'D'])
>>> df
   A  B  C  D
0  2  1  6  3
1  5  6  3  1
2  6  2  3  6
```

Nếu áp dung các hàm trên NumPy vào các đối tượng `ser` hoặc `df`, sẽ sinh ra một đối tượng Pandas mới bảo toàn các chỉ số, ví dụ:

```
>>> np.sin(ser)
0    0.412118
1    0.656987
2    0.989358
3   -0.279415
dtype: float64
>>> np.sin(df)
          A         B         C         D
```

```
0  0.909297  0.841471 -0.279415  0.141120
1 -0.958924 -0.279415  0.141120  0.841471
2 -0.279415  0.909297  0.141120 -0.279415
```

6.4.2. Gióng hàng chỉ số

Đối với cac phép toán hai ngôi, Pandas sẽ gióng hàng các chỉ số trong quá trình tính toán, điều này cũng được thực hiện trong trường hợp dữ liệu bị thiếu, ví dụ:

```
>>> area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
'California': 423967}, name='area')
>>> population = pd.Series({'California': 38332521, 'Texas':
26448193,'New York': 19651127}, name='population')
>>> population / area
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

Mảng kết quả chứa sự kết hợp các chỉ số của hai mảng đầu vào. Ta có thể kiểm tra bằng cách sử dụng phép toán số học trên các chỉ số này:

```
>>> area.index | population.index
Index(['Alaska', 'California', 'New York', 'Texas'], dtype='object')
```

Các giá trị đánh dấu là `NaN` (*Not a Number*) cho biết dữ liệu bị thiếu. Ta có thể thay giá trị ngầm định `NaN` này bằng một giá trị khác bằng cách sử dụng phương thức của phép toán tương ứng và điều chỉnh thêm tham số `fill_value`:

```
>>> population.div(area, fill_value=1)
Alaska        5.802696e-07
California    9.041393e+01
New York      1.965113e+07
Texas         3.801874e+01
dtype: float64
```

Tương tự cho đối tượng `DataFrame`, sẽ gióng hàng theo cột và dòng:

```
>>> df1 = pd.DataFrame(np.random.randint(0, 20, (2, 2)),
columns=list('AB'))
>>> df1
    A   B
0   0  17
1   11  16
```

```
>>> df2 = pd.DataFrame(np.random.randint(0, 10, (3, 3)),
columns=list('BAC'))
>>> df2
   B  A  C
0  6  8  2
1  9  6  1
2  7  4  6
>>> df1+df2
      A     B   C
0   8.0  23.0 NaN
1  17.0  25.0 NaN
2   NaN   NaN NaN
```

Lưu ý rằng các chỉ số được căn chỉnh chính xác bất kể thứ tự của chúng trong hai đối tượng và các chỉ số trong kết quả được sắp xếp. Giống như trường hợp của `Series`, chúng ta có thể sử dụng phương thức số học của đối tượng được liên kết và gởi bất kỳ giá trị mong muốn vào `fill_value` thay cho các mục bị thiếu. Ở đây, ta sẽ điền vào giá trị trung bình của tất cả các giá trị trong `df1`.

```
>>> df1.add(df2,fill_value=1)
      A     B    C
0   8.0  23.0  3.0
1  17.0  25.0  2.0
2   5.0   8.0  7.0
```

Một số phép toán và phương thức tương ứng:

Phép toán	Hàm
+	add()
-	sub(), subtract()
*	mul(), multiply()
/	truediv(), div(), divide()
//	floordiv()
%	mod()
**	pow()

6.4.3. Phép toán giữa DataFrame và Series

Khi bạn thực hiện các thao tác giữa `DataFrame` và `Series`, việc căn chỉnh chỉ mục và cột được duy trì tương tự. Các phép toán giữa `DataFrame` và `Series` tương tự như

các phép toán giữa mảng NumPy hai chiều và một chiều. Hãy xét một phép toán phổ biến, để thấy sự khác biệt của mảng hai chiều và một trong các hàng của nó:

```
>>> A = np.random.randint(10, size=(3, 4))
>>> A-A[0]
array([[ 0,  0,  0,  0],
       [ 5, -4,  6, -6],
       [ 5, -2,  9,  1]])
```

Phép toán này ngầm định áp dụng trên hàng. Tương tự trong Pandas:

```
>>> df = pd.DataFrame(A, columns=list('QRST'))
>>> df
   Q  R  S  T
0  2  4  0  6
1  7  0  6  0
2  7  2  9  7
>>> df - df.iloc[0]
   Q  R  S  T
0  0  0  0  0
1  5 -4  6 -6
2  5 -2  9  1
```

Để áp dụng trên cột, ta gán thuộc tính `axis=0`:

```
>>> df.subtract(df['R'], axis=0)
   Q  R  S  T
0 -2  0 -4  2
1  7  0  6  0
2  5  0  7  5
```

6.5. Quản lý dữ liệu bị thiếu

Sự khác biệt giữa dữ liệu trong lý thuyết và trong thực tế là: dữ liệu trong thực tế hiếm khi rõ ràng và đồng nhất. Thông thường, bộ dữ liệu sẽ bị thiếu một số lượng dữ liệu. Dữ liệu bị thiếu thường là giá trị `null`, `NaN` hoặc `NA`.

6.5.1. Các quy ước cân bằng dữ liệu bị thiếu

a. None

Giá trị đầu tiên Pandas sử dụng cho dữ liệu bị thiếu là `None`. Vì `None` là đối tượng trong Python, nên giá trị nay chỉ sử dụng cho mảng đối tượng.

```
>>> val1 = np.array([1, None, 3, 4])
>>> val1
array([1, None, 3, 4], dtype=object)
```

Ta cũng không thể sử dụng các hàm tổng hợp dữ liệu như `sum()`, `min()`, …

```
>>> val1.sum()
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    val1.sum()
  File "C:\Python\lib\site-packages\numpy\core\_methods.py", line
47, in _sum
    return umr_sum(a, axis, dtype, out, keepdims, initial, where)
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

b. NaN

Biểu diễn dữ liệu bị thiếu khác, `NaN` (từ viết tắt của *Not a Number*), thì khác; nó là một giá trị dấu phẩy động đặc biệt được công nhận bởi tất cả các hệ thống sử dụng biểu diễn dấu phẩy động chuẩn IEEE:

```
>>> val2 = np.array([1, np.nan, 3, 4])
>>> val2
array([ 1., nan,  3.,  4.])
>>> val2.dtype
dtype('float64')
```

Lưu ý rằng NumPy đã chọn một kiểu dấu phẩy động riêng cho mảng này: điều này có nghĩa là không giống như mảng đối tượng trước đó, mảng này hỗ trợ các phép toán nhanh. Bạn nên biết rằng `NaN` hơi giống một 'vi rút' dữ liệu — nó lây nhiễm sang bất kỳ đối tượng nào khác tác động đến nó. Bất kể phép toán nào, kết quả của phép số học với `NaN` sẽ là một `NaN` khác:

```
>>> val2+1
array([ 2., nan,  4.,  5.])
>>> val2.sum()
Nan
```

Gói Nympy cung cấp một số hàm đặc biệt để bỏ qua các giá trị bị thiếu này:

```
>>> np.nansum(val2)
8.0
```

c. Nan và None trong Pandas

Trong Pandas, hai giá trị `None`, `nan` gần như giống nhau, chúng tự động chuyển đổi theo tình huống phù hợp:

```
>>> pd.Series([1, np.nan, 2, None])
0    1.0
```

```
1    NaN
2    2.0
3    NaN
dtype: float64
```

6.5.2. Các thao tác trên dữ liệu bị thiếu

a. Kiểm tra giá trị bị thiếu trong Pandas

Hàm `isnull()`, `notnulll()`, `dropna()`, `fillna()`.

```
>>> pd.isnull(pd1)
0    False
1     True
2    False
3     True
dtype: bool
>>> pd1.notnull()
0     True
1    False
2     True
3    False
dtype: bool
>>> pd1.dropna()
0    1.0
2    2.0
dtype: float64
>>> pd1.fillna(0)
0    1.0
1    0.0
2    2.0
3    0.0
```

Tương tự cho đối tượng `DataFrame`.

6.6. Nối dữ liệu

6.6.1. Concat

a. Concatenate trong NumPy

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

b. Concat trong Pandas

```
>>> ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
>>> ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
>>> pd.concat([ser1, ser2])
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

Đối với đối tượng `DataFrame`:

```
>>> data = [['A1', 'B1'],['A2', 'B2']]
>>> df1 = pd.DataFrame(data, columns=('A', 'B'),index=[1, 2])
>>> df1
    A   B
1  A1  B1
2  A2  B2
>>> df2 = pd.DataFrame(data, columns=('A', 'B'),index=[3, 4])
>>> df2
    A   B
3  A3  B3
4  A4  B4
>>> pd.concat([df1, df2])
    A   B
1  A1  B1
2  A2  B2
3  A3  B3
4  A4  B4
```

Tương tự như NumPy, ta cũng có thể ghép dữ liệu theo cột:

```
>>> data = [['A1', 'B1'], ['A2', 'B2']]
>>> df1 = pd.DataFrame(data, columns=['A', 'B'], index=[1, 2])
>>> df1
    A   B
1  A1  B1
2  A2  B2
>>> data = [['C1', 'D1'], ['C2', 'D2']]
>>> df2 = pd.DataFrame(data, columns=['C', 'D'], index=[1, 2])
>>> df2
```

```
    C   D
1  C1  D1
2  C2  D2
>>> pd.concat([df1, df2], axis=1)
    A   B   C   D
1  A1  B1  C1  D1
2  A2  B2  C2  D2
```

Một điểm khác biệt quan trọng giữa `np.concatenate()` và `pd.concat()` là phép nối Pandas bảo toàn các chỉ số, ngay cả khi kết quả sẽ có các chỉ số trùng lặp! Hãy xem xét ví dụ đơn giản:

```
>>>data = [['A1', 'B1'], ['A2', 'B2']]
>>> df1 = pd.DataFrame(data, columns=['A', 'B'], index=[1, 2])
>>> df1
    A   B
1  A1  B1
2  A2  B2
>>> data = [['A3', 'A3'], ['A4', 'A4']]
>>> df2 = pd.DataFrame(data, columns=['A', 'B'], index=[1, 2])
>>> df2
    A   B
1  A3  A3
2  A4  A4
>>> pd.concat([df1, df2])
    A   B
1  A1  B1
2  A2  B2
1  A3  A3
2  A4  A4
```

Nếu bạn chỉ muốn xác minh rằng các chỉ số trong kết quả của `pd.concat()` không trùng lặp, bạn có thể chỉ định cờ `verify_integrity`. Với giá trị này được đặt thành `True`, việc nối sẽ tạo ra một ngoại lệ nếu có các chỉ số trùng lặp. Dưới đây là một ví dụ bắt và in thông báo lỗi:

```
try:
    pd.concat([df1, df2], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
TypeError: cannot concatenate object of type '<class 'list'>'; only
Series and DataFrame objs are valid
```

Đôi khi bản thân chỉ số không quan trọng và ta muốn bỏ qua nó. Ta có thể chỉ định tùy chọn này bằng cách sử dụng cờ `ignore_index`. Với việc đặt này thành `True`, phép nối sẽ tạo ra một chỉ mục số nguyên mới cho `Series` kết quả:

```
>>> pd.concat([df1, df2],ignore_index=True)
    A   B
0  A1  B1
1  A2  B2
2  A3  A3
3  A4  A4
```

Một giải pháp thay thế khác là sử dụng tùy chọn `keys` để chỉ định nhãn cho các nguồn dữ liệu; kết quả sẽ là một chuỗi được lập chỉ mục phân cấp chứa dữ liệu:

```
>>> pd.concat([df1, df2],keys=['df1','df2'])
        A   B
df1 1  A1  B1
    2  A2  B2
df2 1  A3  A3
    2  A4  A4
```

c. Concat như là joins

```
>>> data = [['A1', 'B1', 'C1'], ['A2', 'B2', 'C2']]
>>> df1 = pd.DataFrame(data,columns=['A', 'B', 'C'], index=[1,2])
>>> df1
    A   B   C
1  A1  B1  C1
2  A2  B2  C2
>>> data = [['B3', 'C3','D3'], ['B4', 'C4','D4']]
>>> df2 = pd.DataFrame(data,columns=['B', 'C','D'], index=[3,4])
>>> df2
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
>>> pd.concat([df1, df2])
     A   B   C    D
1   A1  B1  C1  NaN
2   A2  B2  C2  NaN
3  NaN  B3  C3   D3
4  NaN  B4  C4   D4
```

Theo mặc định, các mục nhập không có sẵn dữ liệu được điền bằng các giá trị `NaN`. Để thay đổi điều này, chúng ta có thể chỉ định một trong một số tùy chọn cho các tham

số join và join_axes của hàm concat(). Theo mặc định, phép nối là sự kết hợp của các cột đầu vào (join='outer'), nhưng chúng ta có thể thay đổi điều này thành phần giao nhau của các cột bằng cách sử dụng join='inner':

```
>>> pd.concat([df1, df2], join='inner')
    B   C
1  B1  C1
2  B2  C2
3  B3  C3
4  B4  C4
```

6.6.2. Append

Bởi vì nối mảng trực tiếp rất phổ biến, nên các đối tượng Series và DataFrame có một phương thức append() có thể thực hiện điều tương tự với ít lần nhấn phím hơn. Ví dụ: thay vì gọi pd.concat([df1, df2]), bạn có thể chỉ cần gọi df1.append (df2):

```
>>> df1
    A   B   C
1  A1  B1  C1
2  A2  B2  C2
>>> df2
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
>>> df1.append(df2)
     A   B   C    D
1   A1  B1  C1  NaN
2   A2  B2  C2  NaN
3  NaN  B3  C3   D3
4  NaN  B4  C4   D4
```

Hãy nhớ rằng không giống như các phương thức append() và expand() của danh sách Python, phương thức append() trong Pandas không sửa đổi đối tượng gốc, thay vào đó, nó tạo một đối tượng mới với dữ liệu được kết hợp. Nó cũng không phải là một phương pháp quá hiệu quả, vì nó liên quan đến việc tạo ra một chỉ mục và bộ đệm dữ liệu mới. Do đó, nếu bạn dự định thực hiện nhiều thao tác nối thêm, nói chung tốt hơn là nên xây dựng một danh sách các DataFrame và chuyển tất cả chúng cùng một lúc vào hàm concat().

6.7. Trộn dữ liệu

Có 3 kiểu kết nối dữ liệu: *one-to-one*, *many-to-one*, *many-to-many* để thực hiện

cho hàm `merge()`.

6.7.1. One-to-one

Xét ví dụ:

```
>>> df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa',
'Sue'],'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
>>> df1
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
>>> df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
'hire_date': [2004, 2008, 2012, 2014]})
>>> df2
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
>>> df3 = pd.merge(df1, df2)
>>> df3
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

Hàm `pd.merge()` nhận biết rằng mỗi `DataFrame` có một cột `"employee"` và tự động kết nối bằng cách sử dụng cột này làm khóa. Kết quả của việc hợp nhất là một `DataFrame` mới kết hợp thông tin từ hai đầu vào. Lưu ý rằng thứ tự của các mục trong mỗi cột không nhất thiết phải được duy trì: trong trường hợp này, thứ tự của cột `"employee"` khác nhau giữa `df1` và `df2`.

6.7.2. Many-to-one

Các phép nối '*Many-to-one*' là các phép nối trong đó một trong hai cột chính chứa các mục trùng lặp. Đối với trường hợp này, kết quả `DataFrame` sẽ bảo toàn các mục nhập trùng lặp đó nếu thích hợp. Hãy xem xét ví dụ sau về phép nối *many-one*:

```
>>> df4 = pd.DataFrame({'group': ['Accounting', 'Engineering',
'HR'],'supervisor': ['Carly', 'Guido', 'Steve']})
```

```
>>>df4
         group supervisor
0   Accounting      Carly
1  Engineering      Guido
2           HR      Steve
>>> pd.merge(df3, df4)
  employee        group  hire_date supervisor
0      Bob   Accounting       2008      Carly
1     Jake  Engineering       2012      Guido
2     Lisa  Engineering       2004      Guido
3      Sue           HR       2014      Steve
```

Kết quả `DataFrame` có một cột bổ sung với thông tin "`supervisor`", nơi thông tin được lặp lại ở một hoặc nhiều vị trí theo yêu cầu của đầu vào.

6.7.3. Many-to-many

Các phép nối *many-many* có một chút khó hiểu về mặt khái niệm. Nếu cột khóa trong cả mảng bên trái và bên phải đều chứa các bản sao thì kết quả là một hợp nhất *many-many*. Xét ví dụ:

```
>>> df1
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
>>> df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
'Engineering', 'Engineering', 'HR', 'HR'],'skills': ['math',
'spreadsheets', 'coding', 'linux','spreadsheets', 'organization']})
>>> df5
         group        skills
0   Accounting          math
1   Accounting  spreadsheets
2  Engineering        coding
3  Engineering         linux
4           HR  spreadsheets
5           HR  organization
>>> pd.merge(df1, df5)
  employee        group        skills
0      Bob   Accounting          math
1      Bob   Accounting  spreadsheets
2     Jake  Engineering        coding
```

```
3     Jake  Engineering         linux
4     Lisa  Engineering        coding
5     Lisa  Engineering         linux
6      Sue           HR  spreadsheets
7      Sue           HR  organization
```

6.7.4. Cột khóa

Ta thấy hành động mặc định của `pd.merge()`: nó tìm kiếm một hoặc nhiều tên cột phù hợp giữa hai đầu vào và sử dụng cột này làm khóa. Tuy nhiên, thường thì các tên cột sẽ không khớp và `pd.merge()` cung cấp nhiều tùy chọn để xử lý điều này.

a. Từ khóa on

Đơn giản nhất, ta có thể chỉ định rõ ràng tên của cột khóa bằng cách sử dụng từ khóa `on`, lấy tên cột hoặc danh sách tên cột:

```
>>> pd.merge(df1, df2, on='employee')
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

Tùy chọn này chỉ hoạt động nếu cả `DataFrame` bên trái và bên phải có tên cột được chỉ định.

b. Từ khóa left_on và right_on

Đôi khi bạn có thể muốn hợp nhất hai tập dữ liệu với các tên cột khác nhau; ví dụ: chúng tôi có thể có một tập dữ liệu trong đó tên nhân viên được gắn nhãn là "`name`" thay vì "`employee`". Trong trường hợp này, chúng ta có thể sử dụng từ khóa `left_on` và `right_on` để chỉ định tên hai cột:

```
>>> df1
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
>>> df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
'salary': [70000, 80000, 120000, 90000]})
>>> df3
   name  salary
0   Bob   70000
```

```
1  Jake   80000
2  Lisa  120000
3   Sue   90000
>>> pd.merge(df1, df3,left_on='employee', right_on='name')
  employee        group  name  salary
0      Bob   Accounting   Bob   70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3      Sue           HR   Sue   90000
```

Kết quả có dư một cột mà chúng ta có thể loại bỏ nếu muốn, bằng cách sử dụng phương thức `drop()` của DataFrame:

```
>>> pd.merge(df1, df3,left_on='employee',
right_on='name').drop('name', axis=1)
  employee        group  salary
0      Bob   Accounting   70000
1     Jake  Engineering   80000
2     Lisa  Engineering  120000
3      Sue           HR   90000
```

c. Từ khóa left_index và right_index

Đôi khi, thay vì hợp nhất trên một cột, ta muốn hợp nhất trên một cột chỉ mục. Ví dụ, dữ liệu của ta có thể trông như thế này:

```
>>> df1a = df1.set_index('employee')
>>> df1a
                group
employee
Bob         Accounting
Jake       Engineering
Lisa       Engineering
Sue                 HR
>>> df2a = df2.set_index('employee')
>>> df2a
          hire_date
employee
Lisa           2004
Bob            2008
Jake           2012
Sue            2014
>>> pd.merge(df1a, df2a, left_index=True, right_index=True)
                group  hire_date
```

```
employee
Bob        Accounting       2008
Jake      Engineering       2012
Lisa      Engineering       2004
Sue               HR       2014
```

Để thuận tiện, `DataFrames` cài đặt phương thức `join()`, phương thức này thực hiện hợp nhất mặc định là kết hợp trên các chỉ số:

```
>>> df1a.join(df2a)
               group  hire_date
employee
Bob        Accounting       2008
Jake      Engineering       2012
Lisa      Engineering       2004
Sue               HR       2014
```

Nếu ta muốn kết hợp các chỉ mục và cột, bạn có thể kết hợp `left_index` với `right_on` hoặc `left_on` với `right_index` để có được kết quả mong muốn:

```
>>> df1a
               group
employee
Bob        Accounting
Jake      Engineering
Lisa      Engineering
Sue               HR
>>> df3
   name  salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000
>>> pd.merge(df1a, df3, left_index=True, right_on='name')
         group  name  salary
0   Accounting   Bob   70000
1  Engineering  Jake   80000
2  Engineering  Lisa  120000
3           HR   Sue   90000
```

6.7.5. Đặc tả phép toán tập hợp khi hợp nhất

Trong tất cả các ví dụ trước, ta đã đề cập đến một yếu tố quan trọng trong việc thực hiện phép nối: kiểu phép toán tập hợp được sử dụng trong phép nối. Điều này xuất hiện

khi một giá trị xuất hiện trong một cột chính nhưng không xuất hiện trong cột khác. Hãy xem xét ví dụ này:

```
>>> df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],'food':
['fish', 'beans', 'bread']},columns=['name', 'food'])
>>> df6
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
>>> df7 = pd.DataFrame({'name': ['Mary', 'Joseph'], 'drink':
['wine', 'beer']}, columns=['name', 'drink'])
>>> df7
     name drink
0    Mary  wine
1  Joseph  beer
>>> pd.merge(df6, df7)
   name   food drink
0  Mary  bread  wine
```

Ở đây ta đã hợp nhất hai tập dữ liệu chỉ có một mục chung duy nhất `'name': Mary`. Theo mặc định, kết quả chứa giao của hai tập đầu vào; đây được gọi là liên kết bên trong. Ta có thể chỉ định điều này một cách rõ ràng bằng cách sử dụng từ khóa `how`, mặc định là `'inner'`:

```
>>> pd.merge(df6, df7, how='inner')
   name   food drink
0  Mary  bread  wine
```

Các tùy chọn khác cho từ khóa `how` là: `'outer'`, `'left'` và `'right'`. Một phép nối ngoài trả về một phép nối trên hợp các cột đầu vào và điền vào tất cả các giá trị bị thiếu bằng `NaN`:

```
>>> pd.merge(df6, df7, how='outer')
     name   food drink
0   Peter   fish   NaN
1    Paul  beans   NaN
2    Mary  bread  wine
3  Joseph    NaN  beer
```

Phép nối bên trái và phép nối phải trả về toàn bộ các mục bên trái và các mục bên phải. Ví dụ:

```
>>> pd.merge(df6, df7, how='left')
```

```
    name   food drink
0  Peter   fish   NaN
1   Paul  beans   NaN
2   Mary  bread  wine
>>> pd.merge(df6, df7, how='right')
     name   food drink
0    Mary  bread  wine
1  Joseph    NaN  beer
```

6.7.6. Tên cột chồng lấp: từ khóa suffixes

Cuối cùng, ta có thể gặp trường hợp hai `DataFrame` có tên cột xung đột. Hãy xem xét ví dụ này:

```
>>> df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
'rank': [1, 2, 3, 4]})
>>> df8
   name  rank
0   Bob     1
1  Jake     2
2  Lisa     3
3   Sue     4
>>> df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
'rank': [3, 1, 4, 2]})
>>> df9
   name  rank
0   Bob     3
1  Jake     1
2  Lisa     4
3   Sue     2
>>> pd.merge(df8, df9,on='name')
   name  rank_x  rank_y
0   Bob       1       3
1  Jake       2       1
2  Lisa       3       4
3   Sue       4       2
```

Bởi vì đầu ra sẽ có hai tên cột xung đột, hàm hợp nhất tự động thêm hậu tố _x hoặc _y để làm cho các cột đầu ra là duy nhất. Nếu các giá trị mặc định này không phù hợp, có thể chỉ định một hậu tố tùy chỉnh bằng cách sử dụng từ khóa `suffixes`:

```
>>> pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
   name  rank_L  rank_R
0   Bob       1       3
```

```
1  Jake       2       1
2  Lisa       3       4
3   Sue       4       2
```

6.7.7. Ví dụ tổng hợp

Thao tác hợp nhất và kết hợp xuất hiện thường xuyên nhất khi kết hợp dữ liệu từ các nguồn khác nhau. Ở đây chúng ta sẽ xem xét một ví dụ về một số dữ liệu về các tiểu bang của Hoa Kỳ và dân số của họ. Dữ liệu được lưu trong 3 file: *state-population.csv* chứa dữ liệu dân số các bang, *state-area.csv* chứa dữ liệu diện tích các bang, *state-abbrevs.csv* chứa tên viết tắt của các bang.

```
>>> pop = pd.read_csv('data\state-population.csv')
>>> areas = pd.read_csv('data\state-areas.csv')
>>> abbrevs = pd.read_csv('data\state-abbrevs.csv')
>>> pop.head(4) #hiện 4 dòng dữ liệu đầu tiên
  state/region     ages  year  population
0           AL  under18  2012   1117489.0
1           AL    total  2012   4817528.0
2           AL  under18  2010   1130966.0
3           AL    total  2010   4785570.0
>>> areas.head() #hiện 5 dòng dữ liệu đầu tiên
        state  area (sq. mi)
0     Alabama          52423
1      Alaska         656425
2     Arizona         114006
3    Arkansas          53182
4  California         163707
>>> abbrevs.head()
        state abbreviation
0     Alabama           AL
1      Alaska           AK
2     Arizona           AZ
3    Arkansas           AR
4  California           CA
```

Với thông tin này, giả sử ta muốn tính toán một kết quả tương đối đơn giản: xếp hạng các tiểu bang và vùng lãnh thổ của Hoa Kỳ theo mật độ dân số năm 2010. Rõ ràng có dữ liệu để tìm kết quả này, nhưng ta sẽ phải kết hợp các tập dữ liệu để có được kết quả đó.

Ta sẽ bắt đầu với hợp nhất *many-one* để cung cấp tên tiểu bang đầy đủ trong `DataFrame` `pop`, hợp nhất dựa trên cột `state/region` của `pop` và cột `abbreviation`

của abbrevs. Sử dụng how='outer' để đảm bảo không có dữ liệu nào bị loại bỏ do các nhãn không khớp.

```
>>> merged = pd.merge(pop, abbrevs, how='outer',
left_on='state/region', right_on='abbreviation')
>>> merged.head()
  >>> merged.head()
  state/region     ages  year  population    state abbreviation
0           AL  under18  2012   1117489.0  Alabama           AL
1           AL    total  2012   4817528.0  Alabama           AL
2           AL  under18  2010   1130966.0  Alabama           AL
3           AL    total  2010   4785570.0  Alabama           AL
4           AL  under18  2011   1125763.0  Alabama           AL
>>> merged = merged.drop('abbreviation',axis=1)
>>> merged.head()
  state/region     ages  year  population    state
0           AL  under18  2012   1117489.0  Alabama
1           AL    total  2012   4817528.0  Alabama
2           AL  under18  2010   1130966.0  Alabama
3           AL    total  2010   4785570.0  Alabama
4           AL  under18  2011   1125763.0  Alabama
```

Hãy kiểm tra kỹ xem có bất kỳ dữ liệu nào không khớp ở đây hay không, chúng ta có thể thực hiện điều này bằng cách tìm kiếm các hàng có giá trị rỗng:

```
>>> merged.isnull().any()
state/region    False
ages            False
year            False
population       True
state            True
dtype: bool
```

Kết quả này cho ta thấy có một vài giá trị ở cột population và state không có dữ liệu, ta sẽ hiển thị kết quả ra như sau:

```
>>> merged[merged['population'].isnull()]
     state/region     ages  year  population state
2448           PR  under18  1990         NaN   NaN
2449           PR    total  1990         NaN   NaN
2450           PR    total  1991         NaN   NaN
2451           PR  under18  1991         NaN   NaN
2452           PR    total  1993         NaN   NaN
```

```
2453           PR  under18  1993         NaN   NaN
2454           PR  under18  1992         NaN   NaN
2455           PR    total  1992         NaN   NaN
2456           PR  under18  1994         NaN   NaN
2457           PR    total  1994         NaN   NaN
2458           PR    total  1995         NaN   NaN
2459           PR  under18  1995         NaN   NaN
2460           PR  under18  1996         NaN   NaN
2461           PR    total  1996         NaN   NaN
2462           PR  under18  1998         NaN   NaN
2463           PR    total  1998         NaN   NaN
2464           PR    total  1997         NaN   NaN
2465           PR  under18  1997         NaN   NaN
2466           PR    total  1999         NaN   NaN
2467           PR  under18  1999         NaN   NaN
```

Kết quả hiển thị trên cho ta biết như tất cả các giá trị dân số của *Puerto Rico* trước năm 2000 là rỗng (chưa có); điều này có thể do dữ liệu này không có sẵn từ nguồn ban đầu. Cụ thể là ta không thấy kết quả nào phù hợp ở khóa `abbreviation`.

```
>>> merged.loc[merged['state'].isnull(), 'state/region'].unique()
array(['PR', 'USA'], dtype=object)
```

Ta thấy: dữ liệu `population` bao gồm các mục của PR (*Puerto Rico*) và USA không xuất hiện trong khóa viết tắt của cột `state`. Ta có thể khắc phục những lỗi này nhanh chóng bằng cách điền vào các giá trị thích hợp:

```
>>> merged.loc[merged['state/region']=='PR','state'] = 'Puerto Rico'
>>> merged.loc[merged['state/region']=='USA','state']='United
States'
>>> merged.isnull().any()
state/region    False
ages            False
year            False
population       True
state           False
dtype: bool
```

Dữ liệu trên cho biết cột `state` đã đầy đủ.

Bây giờ ta có thể hợp nhất kết quả với dữ liệu `area` bằng quy trình tương tự. Kết quả trộn trên cột `state` và dữ liệu lấy nguyên vẹn từ bảng `merged`:

```
>>> final = pd.merge(merged, areas, on='state', how='left')
```

```
>>> final.head()
  state/region     ages  year  population    state  area (sq. mi)
0           AL  under18  2012   1117489.0  Alabama        52423.0
1           AL    total  2012   4817528.0  Alabama        52423.0
2           AL  under18  2010   1130966.0  Alabama        52423.0
3           AL    total  2010   4785570.0  Alabama        52423.0
4           AL  under18  2011   1125763.0  Alabama        52423.0
```

Tiếp tục ta kiểm tra có dữ liệu rỗng hay không?

```
>>> final.isnull().any()
state/region     False
ages             False
year             False
population        True
state            False
area (sq. mi)     True
dtype: bool
```

Có `null` trong cột `area`; ta có thể kiểm tra những vùng nào đã bị bỏ qua ở đây:

```
>>> final['state'][final['area (sq. mi)'].isnull()].unique()
array(['United States'], dtype=object)
```

Ta thấy rằng cột `'area (sq. mi)'` không chứa diện tích của *United States*. Ta có thể chèn giá trị thích hợp (ví dụ: bằng cách sử dụng tổng của tất cả các diện tích tiểu bang), nhưng trong trường hợp này, ta sẽ chỉ bỏ các giá trị rỗng.

```
>>> final.dropna(inplace=True)
>>> final.isnull().any()
state/region     False
ages             False
year             False
population       False
state            False
area (sq. mi)    False
dtype: bool
```

Bây giờ ta có tất cả dữ liệu cần thiết để trả lời câu hỏi quan tâm. Trước tiên ta chọn phần dữ liệu tương ứng với năm 2010 và tổng dân số. Ta sẽ sử dụng hàm `query()` (sẽ trình bày sau) để thực hiện việc này một cách nhanh chóng.

```
>>> data2010 = final.query("year == 2010 & ages == 'total'")
>>> data2010.head()
    state/region   ages  year  population       state  area (sq. mi)
```

```
3             AL  total  2010   4785570.0     Alabama        52423.0
91            AK  total  2010    713868.0      Alaska       656425.0
101           AZ  total  2010   6408790.0     Arizona       114006.0
189           AR  total  2010   2922280.0    Arkansas        53182.0
197           CA  total  2010  37333601.0  California       163707.0
```

Bây giờ, hãy tính toán mật độ dân số và hiển thị theo thứ tự. Ta sẽ bắt đầu bằng cách lập chỉ mục lại dữ liệu trên cột `state`, sau đó tính toán kết quả:

```
>>> data2010.set_index('state', inplace=True)
>>> density = data2010['population'] / data2010['area (sq. mi)']
>>> density.sort_values(ascending=False, inplace=True)
>>> density.head()
state
District of Columbia    8898.897059
Puerto Rico             1058.665149
New Jersey              1009.253268
Rhode Island             681.339159
Connecticut              645.600649
dtype: float64
```

Kết quả là bảng xếp hạng các tiểu bang của Hoa Kỳ cộng với Washington, DC và Puerto Rico theo thứ tự mật độ dân số năm 2010, tính theo số cư dân trên một dặm vuông. Chúng ta có thể thấy rằng cho đến nay khu vực dày đặc nhất trong tập dữ liệu này là Washington, DC (tức là Đặc khu Columbia); trong số các tiểu bang, dày đặc nhất là New Jersey.

Ta có thể xem dữ liệu cuối danh sách:

```
>>> density.tail()
state
South Dakota    10.583512
North Dakota     9.537565
Montana          6.736171
Wyoming          5.768079
Alaska           1.087509
dtype: float64
```

6.8. Tổng hợp và nhóm dữ liệu

Một phần thiết yếu của phân tích dữ liệu lớn là tóm tắt hiệu quả: tính toán các tổng hợp như `sum()`, `mean()`, `median()`, `min()` và `max()`, trong đó một số duy nhất cung cấp thông tin chi tiết về bản chất của một tập dữ liệu. Trong phần này, chúng ta sẽ khám phá các tổng hợp trong Pandas, từ các phép toán đơn giản giống với những gì chúng ta

đã thấy trên mảng NumPy, đến các phép toán phức tạp hơn dựa trên khái niệm nhóm.

6.8.1. Các hàm tổng hợp đơn giản trong Pandas

Tương tự như NumPy, Pandas có các hàm tổng hợp đơn giản `sum()`, `min()`, `max()`, …

Tên hàm	Ý nghĩa
count()	Sô lượng các phần tử
first(), last()	Tham chiếu phần tử đầu, cuối
mean(), median()	Trung bình và trung vị
min(), max()	Giá trị lớn nhất, nhỏ nhất
std(), var()	Độ lệch chuẩn và phương sai
mad()	Độ lệch trung bình
prod()	Tích các phần tử
sum()	Tổng các phân tử

Ví dụ:

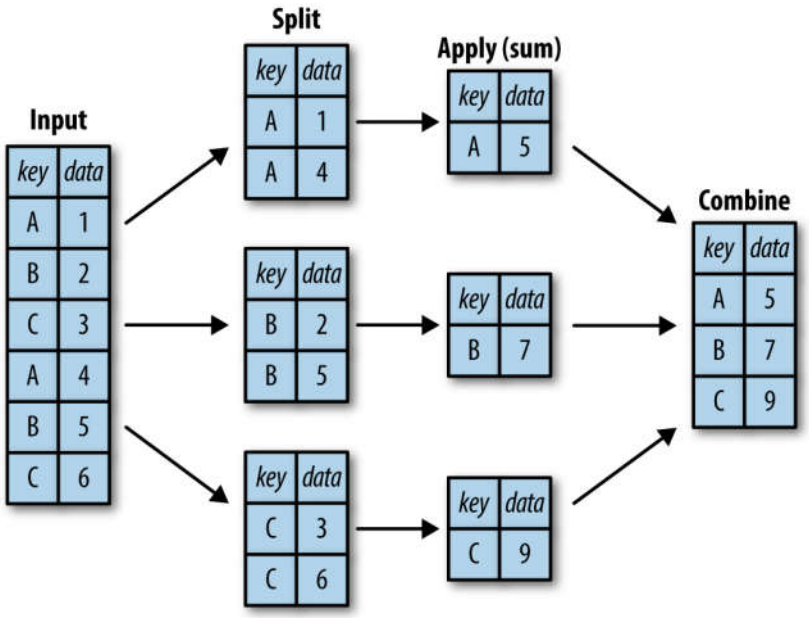
```
>>> ser = pd.Series(np.random.randint(0, 10, 5))
>>> ser
0    8
1    2
2    6
3    2
4    2
dtype: int32
>>> ser.sum()
20
>>> ser.mean()
4.0
>>> ser.median()
2.0
```

6.8.2. Nhóm dữ liệu (groupby)

Các tổng hợp dữ liệu ở trên là đơn giản, nhưng thường thì ta muốn tổng hợp có điều kiện trên một số nhãn hoặc cột chỉ mục: điều này được cài đặt trong các phép toán `'group by'`.

a. Split, apply, combine

Quá trình thực hiện tính toán theo nhóm thường thực hiện qua ba bước: tách dữ liệu theo nhóm (split), áp dụng tính toán trên từng nhóm (apply) và tổng hợp kết quả của từng nhóm (combine).



Xét ví dụ:

```
>>> df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],'data':
range(6)}, columns=['key', 'data'])
>>> df
  key  data
0   A     0
1   B     1
2   C     2
3   A     3
4   B     4
5   C     5
>>> df.groupby('key')
<pandas.core.groupby.generic.DataFrameGroupBy object at
0x000002311DC8F490>
```

Lưu ý rằng những gì được trả về không phải là một `DataFrame`, mà là một đối tượng `DataFrameGroupBy`. Đối tượng này là dạng đặc biệt của `DataFrame`, dùng để thực hiện tính toán theo các nhóm.

```
>>> df.groupby('key').sum()
     data
key
```

```
A       3
B       5
C       7
>>> df.groupby('key').max()
     data
key
A       3
B       4
C       5
>>> df.groupby('key').mean()
     data
key
A     1.5
B     2.5
C     3.5
```

b. Cột chỉ mục

Đối tượng `GroupBy` hỗ trợ lập chỉ mục cột theo cách giống như `DataFrame` và trả về một đối tượng `GroupBy` đã sửa đổi. Ví dụ:

```
>>> hanghoa = pd.DataFrame({'mh': ['A', 'B', 'C', 'A', 'B', 'C'],
'xuat': np.random.randint(5, size=6), 'nhap': np.random.randint(10,
size=6)}, columns=['mh', 'nhap', 'xuat'])
>>> hanghoa
  mh  nhap  xuat
0  A     4     1
1  B     1     3
2  C     6     1
3  A     8     4
4  B     0     2
5  C     5     0
>>> hanghoa.groupby('mh')['nhap'].sum()
mh
A    12
B     1
C    11
Name: nhap, dtype: int32
>>> hanghoa.groupby('mh')['xuat'].max()
mh
A    4
B    3
C    1
```

```
Name: xuat, dtype: int32
```

c. Lặp trên groupby

Để lặp trên đối tượng groupby, ta dùng bộ hai biến để lặp. Biến thứ nhất chứa giá trị của nhóm, biến thứ hai chứa dữ liệu của nhóm

```
for (ma, dulieu) in hanghoa.groupby('mh'):
     print('Mặt hàng {0} có số lượng xuất:{1}'.format(ma,
dulieu['xuat'].sum()))

Mặt hàng A có số lượng xuất: 5
Mặt hàng B có số lượng xuất: 5
Mặt hàng C có số lượng xuất: 1
```

Ví dụ trên ma chứa giá trị là nhóm các mặt hàng, dulieu chứa bảng giá trị của nhóm ma tương ứng.

d. Hàm aggregate

Ta có cách khác tổng hợp dữ liệu, sử dụng hàm arrgregate(). Hàm truyền tham số là tên các hàm tổng hợp dữ liệu như max, min, ...

```
hanghoa = pd.DataFrame({'mh': ['A', 'B', 'C', 'A', 'B', 'C'],
'xuat': np.random.randint(5, size=6), 'nhap': np.random.randint(10,
size=6)}, columns=['mh', 'nhap', 'xuat'])
>>> hanghoa
  mh  nhap  xuat
0  A     7     2
1  B     3     1
2  C     9     1
3  A     1     2
4  B     8     1
5  C     1     3
>>> hanghoa.groupby('mh').aggregate(['min', 'mean','max'])
   nhap          xuat
    min mean max  min mean max
mh
A     1  4.0   7    2    2   2
B     3  5.5   8    1    1   1
C     1  5.0   9    1    2   3
```

Hoặc có thể gọi theo cách khác:

```
>>> hanghoa.groupby('mh').aggregate({'nhap':'min', 'xuat':'max'})
    nhap  xuat
```

```
mh
A       1       2
B       3       1
C       1       3
```

CHƯƠNG 7. TRỰC QUAN HÓA VỚI MATPLOTLIB

Matplotlib là một thư viện trực quan hóa dữ liệu đa nền tảng được xây dựng trên mảng NumPy và được thiết kế để hoạt động với ngăn xếp SciPy rộng lớn hơn. Nó được John Hunter hình thành vào năm 2002, ban đầu là một bản vá cho IPython để cho phép vẽ biểu đồ kiểu MATLAB tương tác thông qua gnuplot từ dòng lệnh IPython. Người sáng tạo của IPython, Fernando Perez, vào thời điểm đó đang cố gắng hoàn thành chương trình Tiến sĩ của mình và cho John biết rằng anh ấy sẽ không có thời gian để xem xét bản vá trong vài tháng. John coi đây là một gợi ý để tự mình bắt đầu, và gói Matplotlib ra đời, với phiên bản 0.1 được phát hành vào năm 2003.

7.1. Một số thao tác cơ bản

7.1.1. Chèn matplotlib vào chương trình

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Giao diện plt thường được sử dụng nhiều hơn.

Ta sử dụng hàm `plt.style()` để chọn kiểu vẽ phù hợp. Ở đây ta thường sẽ thiết lập kiểu `classic`:

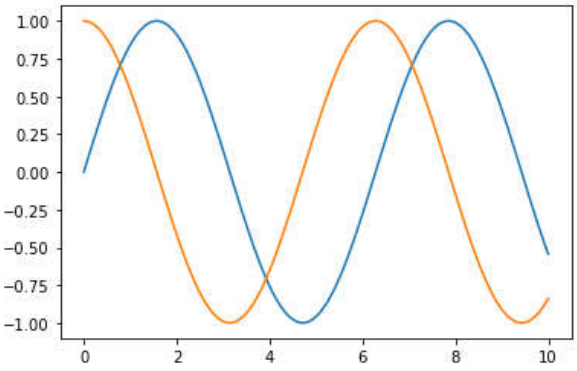
```
plt.style.use('classic')
```

7.1.2. Tình huống thường sử dụng thư viện Matplotlib

a. Trong tập lệnh:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.show()
```

Kết quả:



Lệnh `plt.show()` làm rất nhiều việc vì nó phải tương tác với phần phụ trợ đồ họa của hệ thống. Chi tiết của hoạt động này có thể khác nhau rất nhiều giữa các hệ thống, nhưng Matplotlib cố gắng hết sức để ẩn tất cả các chi tiết.

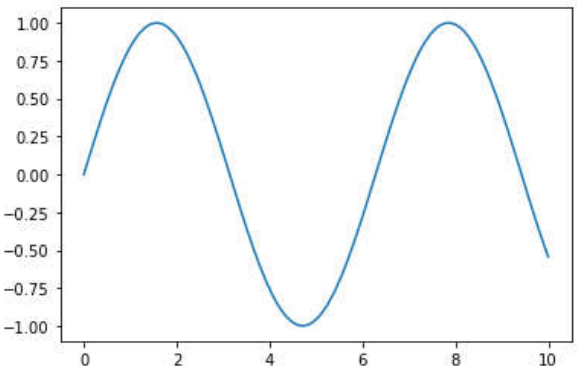
Chú ý: lệnh `plt.show()` chỉ nên được sử dụng một lần cho mỗi phiên Python và thường được viết ở cuối tập lệnh.

b. Vẽ trong Ipython shell

Nếu thực hiện các câu lệnh trong Ipython shell (chế độ gõ từng lệnh), thì bất kỳ lệnh `plt.plot()` nào cũng sẽ vẽ hình ra cửa sổ và các lệnh `plt.plot()` khác nếu có sẽ vẽ hình khác thay thể hình đã có. Sử dụng `plt.show()` ở chế độ này là không cần thiết. Ví dụ:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))
```

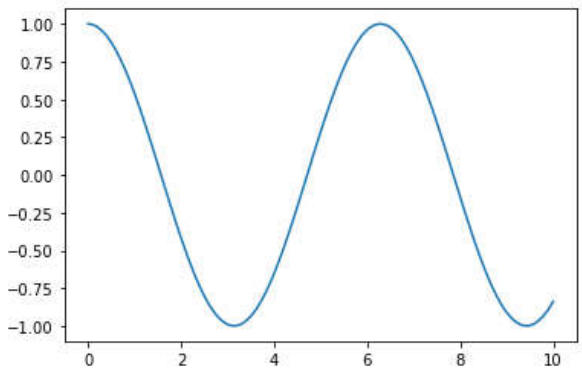
Khi đó cửa sổ sẽ hiển thị:



Nếu tiếp tục gõ lệnh:

```
plt.plot(x, np.cos(x))
```

Thì cửa sổ sẽ thay bằng hình mới:



c. Vẽ từ Jupyter Notebook

IPython notebook là một công cụ phân tích dữ liệu tương tác dựa trên trình duyệt có thể kết hợp mã, đồ họa, các phần tử HTML và hơn thế nữa vào một tài liệu thực thi duy nhất.

Vẽ đồ thị trong một IPython notebook có thể được thực hiện bằng lệnh %matplotlib và hoạt động theo cách tương tự như IPython shell. Trong IPython notebook, bạn cũng có tùy chọn nhúng đồ họa trực tiếp vào sổ ghi chép, với hai tùy chọn khả thi:

% matplotlib, notebook sẽ dẫn đến các hình vẽ được nhúng trong notebook

% matplotlib inline, sẽ dẫn đến *hình ảnh tĩnh* của hình vẽ được nhúng trong #notebook.

7.1.3. Lưu hình vẽ đến file

Một tính năng nữa của Matplotlib là khả năng lưu các hình vẽ ở nhiều định dạng khác nhau. Bạn có thể lưu một hình bằng lệnh `savefig()`. Ví dụ: để lưu hình trước đó dưới dạng tệp PNG, bạn có thể chạy như sau:

```
plt.savefig('d:\hinh1.png')
```

Để xem lại ảnh này ta sử dụng đối tượng Ipython.Image để hiển thị nội dung của nó:

```
from IPython.display import Image
Image('d:\hinh1.png')
```

Trong hàm `savefig()`, định dạng tệp được suy ra từ phần mở rộng của tên tệp đã cho. Tùy thuộc vào phần phụ trợ đã cài đặt, nhiều định dạng tệp khác nhau có sẵn. Ta có thể tìm thấy danh sách các loại tệp được hỗ trợ cho hệ thống của mình bằng cách sử dụng phương pháp sau của đối tượng `canvas`:

```
plt.figure().canvas.get_supported_filetypes()
```

Kết quả:

```
Out[54]:
{'ps': 'Postscript',
 'eps': 'Encapsulated Postscript',
 'pdf': 'Portable Document Format',
 'pgf': 'PGF code for LaTeX',
 'png': 'Portable Network Graphics',
 'raw': 'Raw RGBA bitmap',
 'rgba': 'Raw RGBA bitmap',
 'svg': 'Scalable Vector Graphics',
 'svgz': 'Scalable Vector Graphics',
 'jpg': 'Joint Photographic Experts Group',
 'jpeg': 'Joint Photographic Experts Group',
 'tif': 'Tagged Image File Format',
 'tiff': 'Tagged Image File Format'}<Figure size 432x288 with 0
}
```

7.2. Hai giao diện vẽ

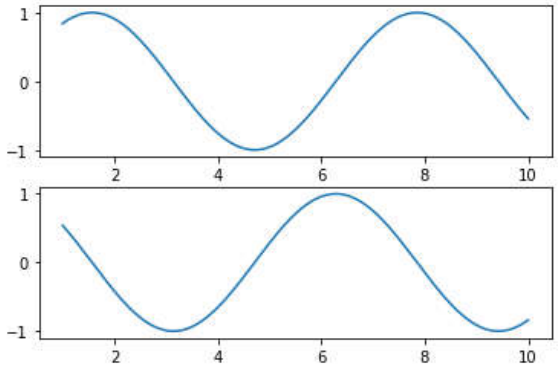
Một tính năng có thể gây nhầm lẫn của Matplotlib là giao diện kép của nó: giao diện dựa trên trạng thái kiểu MATLAB và giao diện hướng đối tượng mạnh mẽ hơn. Ta sẽ tìm hiểu sự khác biệt sau đây.

7.2.1. Giao diện giống MathLab

Matplotlib ban đầu được viết như một sự thay thế Python cho người dùng MATLAB và phần lớn cú pháp của nó phản ánh điều đó. Các công cụ kiểu MATLAB được chứa trong giao diện `pyplot` (plt). Ví dụ, đoạn mã sau có thể trông khá quen thuộc với người dùng MATLAB:

```
import matplotlib.pyplot as plt
import numpy as np
plt.figure()
x = np.linspace(1, 10, 1000)
plt.subplot(2, 1, 1)
plt.plot(x, np.sin(x))
plt.subplot(2, 1, 2)
plt.plot(x, np.cos(x))
```

Kết quả:



Điều quan trọng cần lưu ý trong giao diện này là trạng thái: nó theo dõi hình và trục "hiện tại", là nơi áp dụng tất cả các lệnh `plt`. Ta có thể tham chiếu đến những đối tượng này bằng cách sử dụng hàm `plt.gcf()` (lấy hình hiện tại) và hàm `plt.gca()` (lấy trục hiện tại).

Mặc dù giao diện này nhanh chóng và thuận tiện cho các hình đơn giản, nhưng nó rất dễ gặp sự cố. Ví dụ: khi bảng thứ hai được tạo, làm thế nào chúng ta có thể quay lại và thêm một cái gì đó vào bảng thứ nhất? Điều này có thể thực hiện được trong giao diện kiểu MATLAB, nhưng hơi rắc rối. Có một cách tốt hơn.

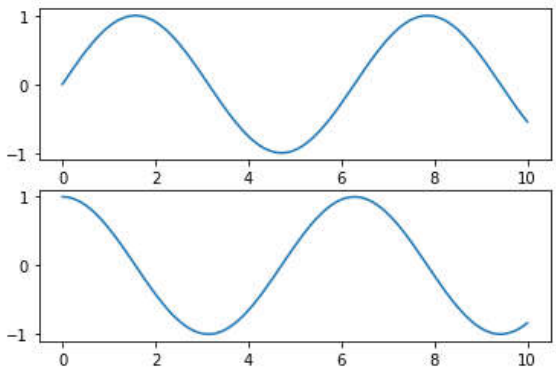
7.2.2. Giao diện hướng đối tượng

Giao diện hướng đối tượng có sẵn cho những trường hợp phức tạp hơn và khi muốn kiểm soát nhiều hơn trong hình của mình. Thay vì phụ thuộc vào một số khái niệm về hình hoặc trục "hoạt động", trong giao diện hướng đối tượng, các hàm vẽ đồ thị là các phương thức của các đối tượng `Figure` và `Axes` rõ ràng. Để tạo lại hình trước bằng cách sử dụng kiểu vẽ này, bạn có thể làm như sau:

Với tất cả các hình vẽ trong Matplotlib, ta bắt đầu bằng cách tạo một đối tượng `figure` và một đối tượng `axes`. Ở dạng đơn giản nhất, một `figure` và `axes` có thể được tạo ra như sau:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 100)
# đầu tiên tạo lưới các hình
# ax là mảng chứa hai đối tượng Axes để vẽ, mỗi đối tượng sẽ vẽ trên
một subplot
fig, ax = plt.subplots(2)
# gọi hàm plot() trên từng đối tượng
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x))
```

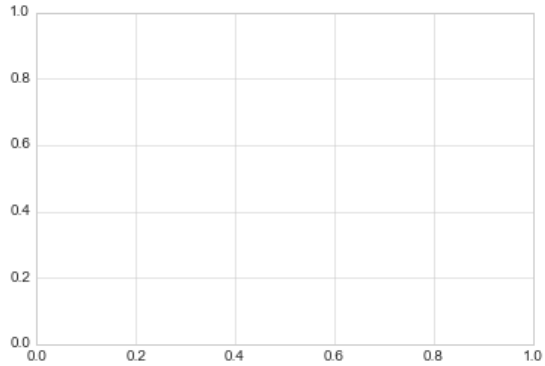
Kết quả:



Đối với những hình đơn giản hơn, việc lựa chọn sử dụng phong cách nào phần lớn là vấn đề sở thích, nhưng cách tiếp cận hướng đối tượng có thể trở nên cần thiết khi các hình trở nên phức tạp hơn. Trong suốt chương này, ta sẽ chuyển đổi giữa giao diện kiểu MATLAB và giao diện hướng đối tượng, tùy thuộc vào điều gì thuận tiện nhất. Trong hầu hết các trường hợp, sự khác biệt là nhỏ như chuyển `plt.plot()` sang `ax.plot()`, nhưng có một vài lỗi mà chúng tôi sẽ nêu bật khi chúng xuất hiện trong các phần sau.

7.3. Vẽ đường đơn giản

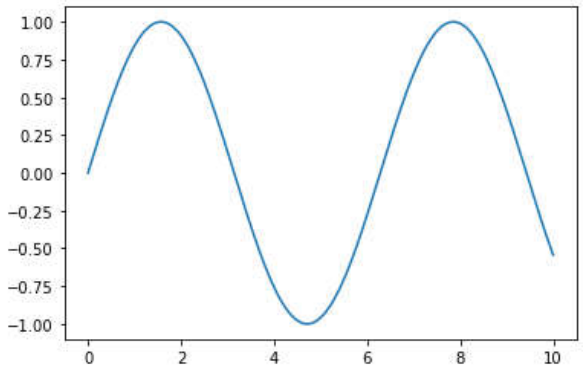
Đối với tất cả các đồ thị Matplotlib, chúng ta bắt đầu bằng cách tạo một hình và một trục. Ở dạng đơn giản nhất, một hình và các trục có thể được tạo như sau:



Trong Matplotlib, `figure` (một thể hiện của lớp `plt.Figure`) có thể được coi như một vùng chứa duy nhất chứa tất cả các đối tượng đại diện cho các `axes`, `graphic`, `text` và `label`. Các `axes` (một thể hiện của lớp `plt.Axes`) là những gì chúng ta thấy ở trên: một hộp giới hạn với lưới và nhãn, mà sẽ chứa các hình vẽ tạo nên ảnh. Ta sẽ sử dụng hàm plot của đối tượng `axes` để vẽ, cụ thể như sau:

```
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```

kết quả như hình sau:



7.3.1. Màu và kiểu đường vẽ

Ta có thể tùy chỉnh màu sắc và kiểu đường trên hàm `plt.plot()` bằng tham số `color` và `linestyle`.

```
plt.plot(x, np.sin(x - 0), color='blue') # tên màu
plt.plot(x, np.sin(x - 1), color='g') # viết tắt tên màu (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75') # mức xám (0 → 1)
plt.plot(x, np.sin(x - 3), color='#FFDD44') # RRGGBB từ 00 đến FF
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) # bộ RGB, (0 → 1)
plt.plot(x, np.sin(x - 5), color='chartreuse'); # tên màu HTML
```

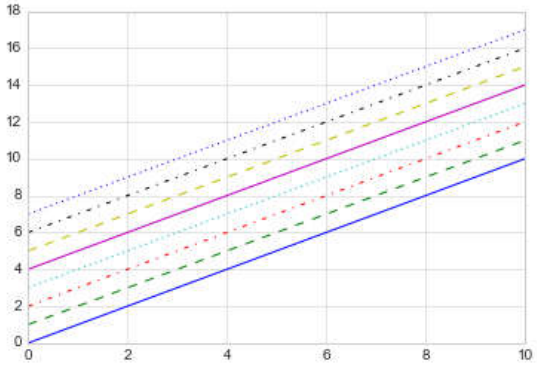
Nếu không chỉ thuộc tính `color`, Matplotlib lấy màu vẽ ngầm định. Tương tự, kiểu đường vẽ:

```
plt.plot(x, x + 0, linestyle='solid')
plt.plot(x, x + 1, linestyle='dashed')
plt.plot(x, x + 2, linestyle='dashdot')
plt.plot(x, x + 3, linestyle='dotted')
```

Hoặc:

```
plt.plot(x, x + 4, linestyle='-') # solid
plt.plot(x, x + 5, linestyle='--') # dashed
plt.plot(x, x + 6, linestyle='-.') # dashdot
plt.plot(x, x + 7, linestyle=':'); # dotted
```

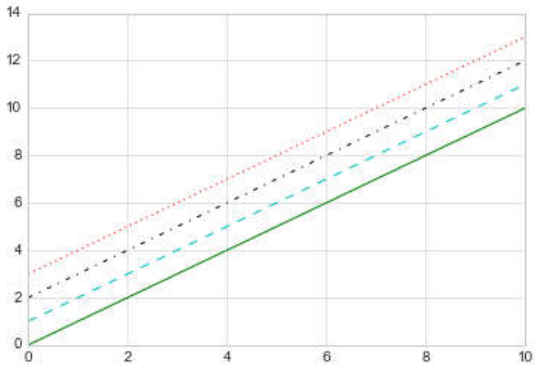
Kết quả:



Nếu muốn ngắn gọn, các mã màu và kiểu đường kẻ này có thể được kết hợp thành một đối số duy nhất cho hàm `plt.plot()`.

```
plt.plot(x, x + 0, '-g') # solid green
plt.plot(x, x + 1, '--c') # dashed cyan
plt.plot(x, x + 2, '-.k') # dashdot black
plt.plot(x, x + 3, ':r'); # dotted red
```

Kết quả:

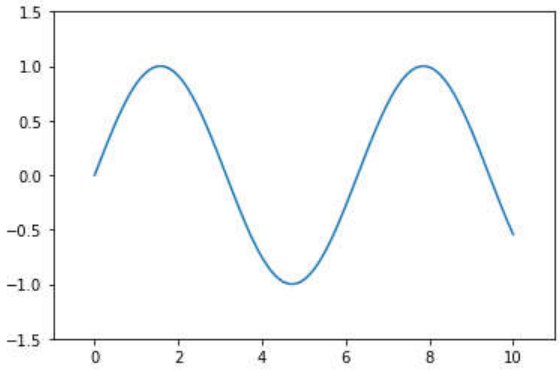


7.3.2. Giới hạn trục

Matplotlib thường thực hiện chọn các giới hạn trục mặc định cho hình vẽ, nhưng đôi khi ta có thể kiểm soát tốt hơn. Cách cơ bản nhất để điều chỉnh giới hạn trục là sử dụng phương thức `plt.xlim()` và `plt.ylim()`:

```
plt.plot(x, np.sin(x))
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```

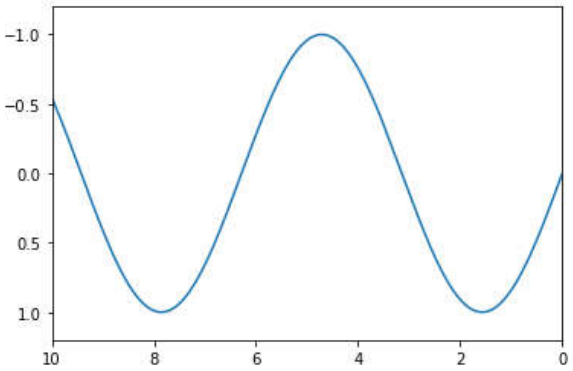
kết quả:



Nếu vì lý do nào đó ta muốn một trong hai trục được hiển thị ngược lại, ta chỉ cần đảo ngược thứ tự của các đối số:

```
plt.plot(x, np.sin(x))
plt.xlim(10, 0)
plt.ylim(1.2, -1.2);
```

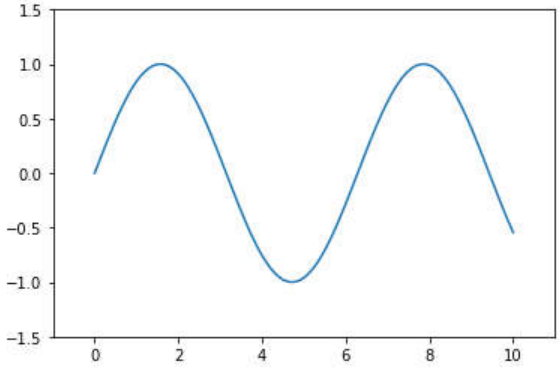
Kết quả:



Một phương thức hữu ích có liên quan là `plt.axis()`. Phương thức `plt.axis()` cho phép bạn đặt giới hạn `x` và `y` bằng một lệnh gọi, bằng cách chuyển danh sách chỉ định `[xmin, xmax, ymin, ymax]`:

```
plt.plot(x, np.sin(x))
plt.axis([-1, 11, -1.5, 1.5]);
```

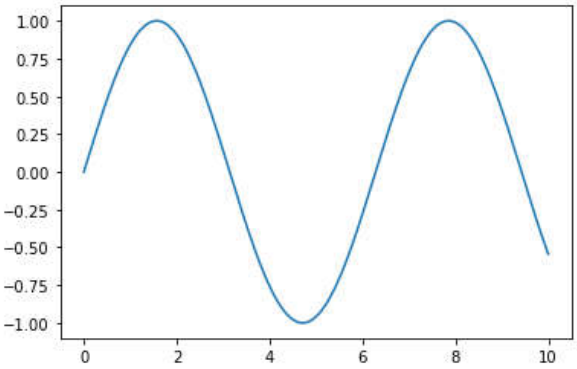
Kết quả:



Phương thức `plt.axis()` còn cho phép ta làm những việc như tự động thu hẹp các đường viền vừa với hình.

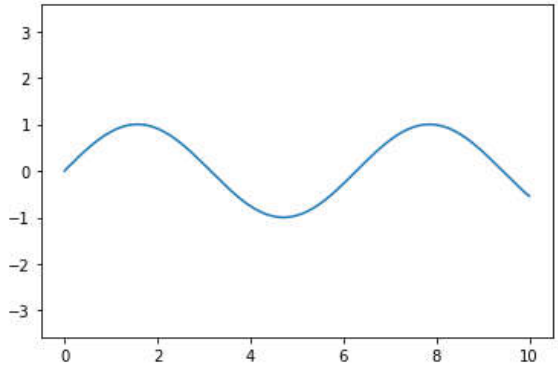
```
plt.plot(x, np.sin(x))
plt.axis('tight');
```

Kết quả:



Nó cho phép các thông số kỹ thuật cấp cao hơn, chẳng hạn như đảm bảo tỷ lệ khung hình bằng nhau để trên màn hình của bạn, một đơn vị `x` bằng một đơn vị `y`:

```
plt.plot(x, np.sin(x))
plt.axis('equal');
```

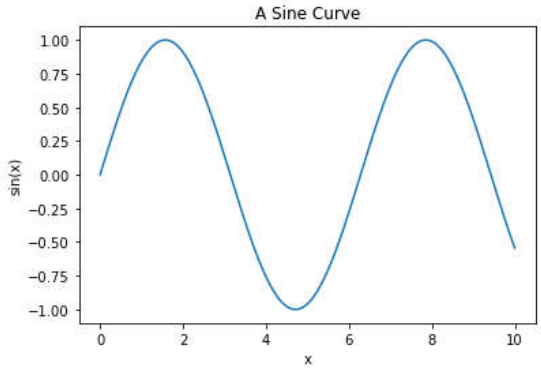


7.3.3. Gán nhãn hình ảnh

Ta sử dụng 3 phương thức để gán nhãn tiêu đề, trục `x` (ngang) và trục `y` (dọc): `plt.title()`, `plt.xlabel()`, `plt.ylabel()`:

```
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x))
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)");
```

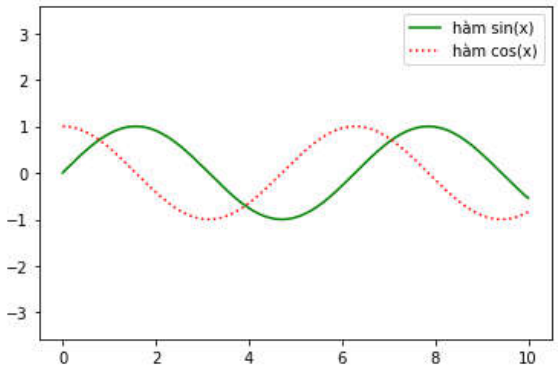
Kết quả:



Khi nhiều đường được hiển thị trong một hệ trục, ta thể tạo chú giải cho mỗi đường vẽ bằng cách gắn nhãn cho thuộc tính `label` của hàm `plot()` và sau đó gọi phương thức `legend()`.

```
x = np.linspace(0, 10, 1000)
plt.plot(x, np.sin(x),'-g', label='hàm sin(x)')
plt.plot(x, np.cos(x),':r', label='hàm cos(x)')
plt.axis('equal')
plt.legend();
```

kết quả hiển thị:



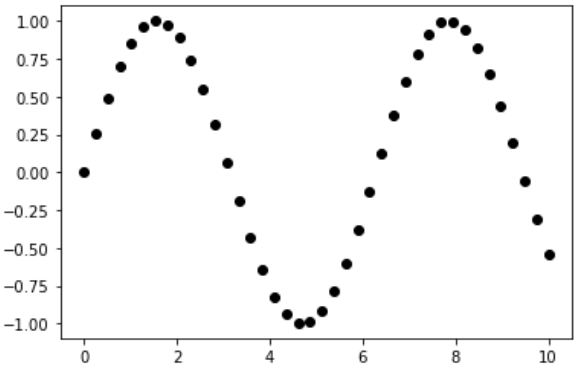
7.4. Vẽ biểu đồ phân tán

Một kiểu hình vẽ thường được sử dụng khác là biểu đồ phân tán đơn giản. Thay vì các điểm được nối bằng các đoạn thẳng, ở đây các điểm được biểu diễn riêng lẻ bằng một dấu chấm, hình tròn hoặc hình dạng khác.

7.4.1. Vẽ biểu đồ phân tán bằng hàm plt.plot()

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 40)
y = np.sin(x)
plt.plot(x, y, 'o', color='black')
```

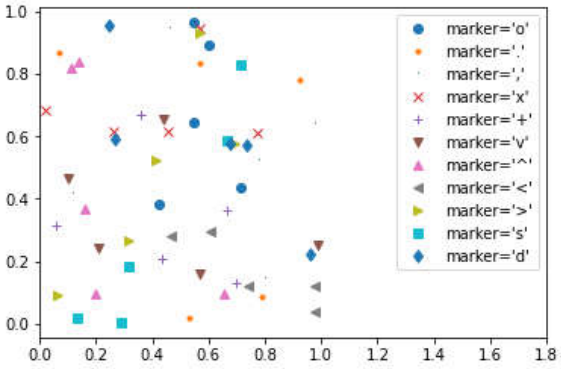
Kết quả:



Đối số thứ ba trong lệnh gọi hàm là một ký tự đại diện cho loại ký hiệu được sử dụng cho biểu đồ. Cũng giống như ta có thể chỉ định các tùy chọn như `'-'` và `'--'` để chọn kiểu đường, kiểu đánh dấu có bộ mã chuỗi ngắn của riêng nó. Ta có thể xem danh sách đầy đủ các ký hiệu có sẵn trong tài liệu của plt.plot hoặc trong tài liệu trực tuyến của Matplotlib. Hầu hết các khả năng đều khá trực quan và ta sẽ chỉ ra một số khả năng phổ biến hơn sau đây:

```
import matplotlib.pyplot as plt
import numpy as np
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's',
'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker, label =
"marker='{0}'".format(marker))
plt.legend(numpoints=1)
plt.xlim(0, 1.8);
```

Kết quả như hình sau:

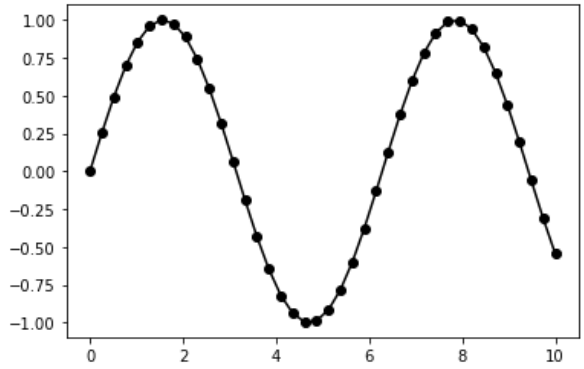


Để có nhiều khả năng hơn nữa, các mã ký tự này có thể được sử dụng cùng với mã đường và mã màu để vẽ các điểm cùng với một đường nối chúng:

```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 40)
```

```
y = np.sin(x)
plt.plot(x, y, '-ok'); # line (-), circle marker (o), black (k);
```

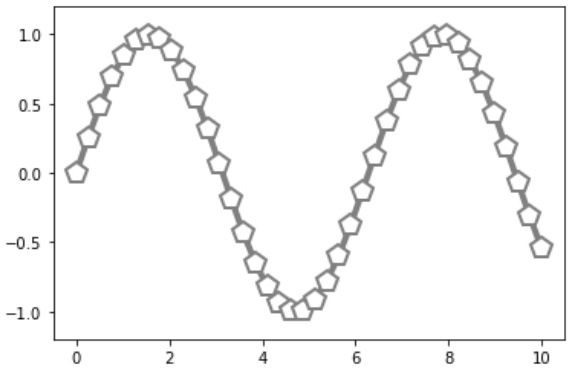
Kết quả:



Các đối số từ khóa bổ sung cho `plt.plot()` chỉ định một loạt các thuộc tính của các đường và điểm đánh dấu:

```
x = np.linspace(0, 10, 40)
y = np.sin(x)
plt.plot(x, y, '-p', color='gray', markersize=15, linewidth=4,
markerfacecolor='white', markeredgecolor='gray', markeredgewidth=2)
plt.ylim(-1.2, 1.2);
```

Kết quả:

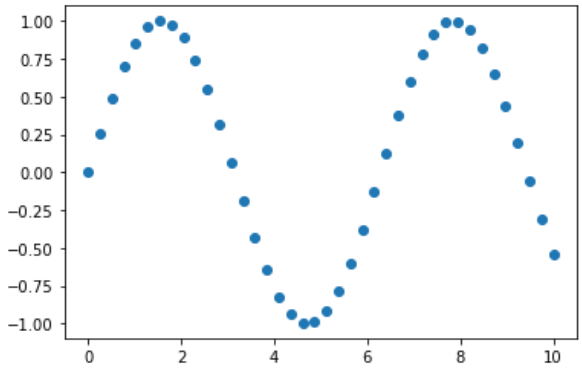


7.4.2. Vẽ biểu đồ bằng plt.scatter()

Một phương pháp thứ hai, mạnh mẽ hơn để tạo các biểu đồ phân tán là hàm `plt.scatter()`, có thể được sử dụng rất giống với hàm `plt.plot()`:

```
x = np.linspace(0, 10, 40)
y = np.sin(x)
plt.scatter(x, y, marker='o')
```

Kết quả:

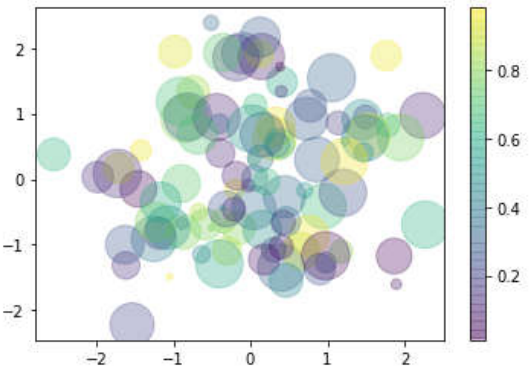


Sự khác biệt cơ bản của `plt.scatter()` so với `plt.plot()` là nó có thể được sử dụng để tạo các biểu đồ phân tán trong đó các thuộc tính của từng điểm riêng lẻ (kích thước, màu mặt, màu cạnh, v.v.) có thể được điều khiển riêng hoặc ánh xạ tới dữ liệu.

Ví dụ sau tạo một biểu đồ phân tán ngẫu nhiên với các điểm có nhiều màu sắc và kích thước. Để thấy rõ hơn các kết quả chồng chéo, ta cũng sẽ sử dụng từ khóa `alpha` để điều chỉnh mức độ trong suốt:

```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)
plt.scatter(x, y, c=colors, s=sizes, alpha=0.3, cmap='viridis')
plt.colorbar(); # show color scale
```

Kết quả:



Lưu ý rằng đối số màu được ánh xạ tự động thành thang màu (được hiển thị ở đây bằng lệnh `colorbar()`) và đối số kích thước được tính bằng pixel. Bằng cách này, màu sắc và kích thước của các điểm có thể được sử dụng để truyền tải thông tin trực quan, nhằm minh họa dữ liệu đa chiều.

7.4.3. Plt.plot() so với plt.scatter()

Ngoài các tính năng khác nhau có sẵn trong `plt.plot()` và `plt.scatter()`, tại sao bạn có thể chọn sử dụng cái này hơn cái kia? Mặc dù điều đó không quan trọng lắm đối với lượng dữ liệu nhỏ, vì tập dữ liệu lớn hơn vài nghìn điểm, `plt.plot()` có thể hiệu quả hơn đáng kể so với `plt.scatter()`. Lý do là `plt.scatter()` có khả năng hiển thị kích thước và/hoặc màu sắc khác nhau cho mỗi điểm, vì vậy trình kết xuất phải thực hiện thêm công việc xây dựng từng điểm riêng lẻ. Mặt khác, trong `plt.plot()`, các điểm luôn là bản sao của nhau, do đó công việc xác định diện mạo của các điểm chỉ được thực hiện một lần cho toàn bộ tập dữ liệu. Đối với các tập dữ liệu lớn, sự khác biệt giữa hai thứ này có thể dẫn đến hiệu suất rất khác nhau và vì lý do này, `plt.plot()`được ưu tiên hơn `plt.scatter()` cho các tập dữ liệu lớn.

7.5. Vẽ biểu đồ lỗi

Đối với bất kỳ phép đo khoa học nào, việc tính toán chính xác các sai sót gần như quan trọng, nếu không muốn nói là quan trọng hơn việc báo cáo chính xác con số. Ví dụ: hãy tưởng tượng rằng bạn đang sử dụng một số quan sát vật lý thiên văn để ước tính Hằng số Hubble, phép đo cục bộ về tốc độ giãn nở của vũ trụ. Tôi biết rằng tài liệu hiện tại đề xuất giá trị khoảng 71 (km/s)/Mpc và tôi đo giá trị 74 (km/s)/Mpc bằng phương pháp của mình. Các giá trị có nhất quán không? Câu trả lời đúng duy nhất, với thông tin này, là: không có cách nào để biết.

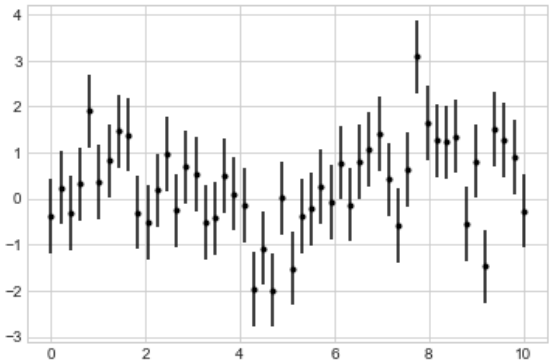
Giả sử ta bổ sung thông tin này với các độ không đảm bảo được báo cáo: các tài liệu hiện tại cho thấy giá trị khoảng 71 ± 2,5 (km/s)/Mpc và phương pháp đã đo được giá trị là 74 ± 5 (km/s)/Mpc. Bây giờ các giá trị có nhất quán không? Đó là một câu hỏi có thể được trả lời một cách định lượng.

Trong hình dung dữ liệu và kết quả, việc hiển thị những lỗi này một cách hiệu quả có thể làm cho một hình truyền tải thông tin đầy đủ hơn nhiều.

7.5.1. Hàm ErrorBar

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
x = np.linspace(0, 10, 50)
dy = 0.8
y = np.sin(x) + dy * np.random.randn(50)
plt.errorbar(x, y, yerr=dy, fmt='.k')
```

Kết quả:

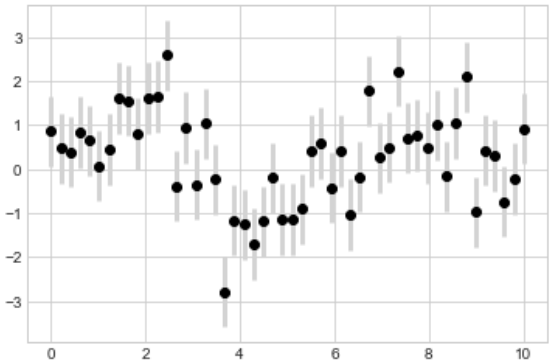


Trong đó `fmt` là mã định dạng điều khiển sự xuất hiện của đường và điểm và có cùng cú pháp với hàm `plt.plot()`.

Ngoài các tùy chọn cơ bản này, hàm `errorbar()` có nhiều tùy chọn để tinh chỉnh kết quả đầu ra. Sử dụng các tùy chọn bổ sung này, bạn có thể dễ dàng tùy chỉnh tính thẩm mỹ của biểu đồ lỗi của mình.

```
plt.errorbar(x, y, yerr=dy, fmt='o', color='black',
ecolor='lightgray', elinewidth=3, capsize=0);
```

Kết quả:



7.6. Đồ thị mật độ và đồ thị đường viền

Đôi khi, rất hữu ích khi hiển thị dữ liệu ba chiều trong hai chiều bằng cách sử dụng các đường viền hoặc các vùng được mã hóa màu. Có ba hàm Matplotlib có thể hữu ích cho nhiệm vụ này: `plt.contour()` cho các ô đường đồng mức, `plt.contourf()` cho các ô đường đồng mức đã điền và `plt.imshow()` để hiển thị hình ảnh. Phần này xem xét một số ví dụ về việc sử dụng chúng.

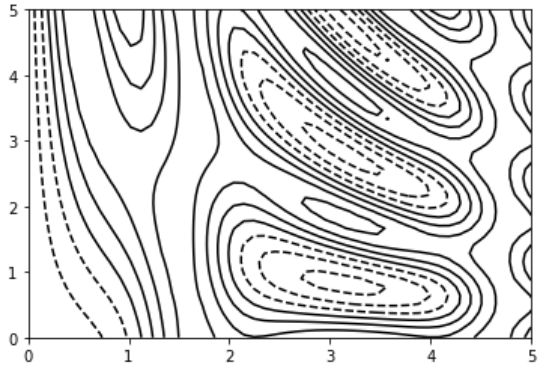
7.6.1. Hiển thị hàm ba chiều

Chúng ta sẽ bắt đầu bằng cách biểu diễn một biểu đồ đường bao bằng hàm z = f(x, y), sử dụng lựa chọn cụ thể sau cho f:

```
def f(x, y):
```

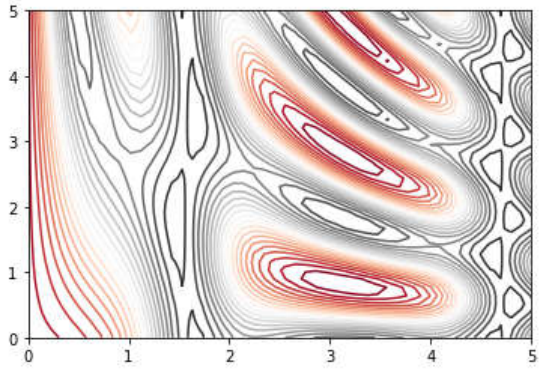
```
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
plt.contour(X, Y, Z, colors='black')
```

Kết quả:



Lưu ý rằng theo mặc định khi sử dụng một màu, các giá trị âm được biểu thị bằng các đường đứt nét và các giá trị dương bằng các đường liền. Ngoài ra, bạn có thể mã màu cho các dòng bằng cách chỉ định một bản đồ màu với đối số là `cmap`. Ở đây, ta cũng sẽ chỉ định rằng vẽ nhiều dòng hơn với 20 khoảng cách đều nhau trong phạm vi dữ liệu:

```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```



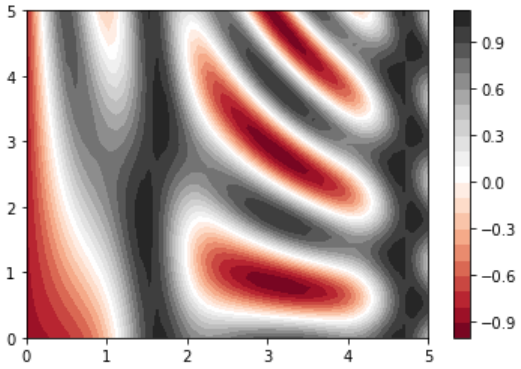
Ở đây, ta chọn bản đồ màu `RdGy` (viết tắt của Red-Grey), đây là một lựa chọn tốt cho dữ liệu. Matplotlib có sẵn một loạt các bản đồ màu, bạn có thể dễ dàng duyệt qua trong IPython bằng cách hoàn thành tab trên mô-đun `plt.cm<Tab>`.

Hình vẽ trông đẹp hơn, nhưng khoảng cách giữa các dòng có thể hơi phân tán. Chúng ta có thể thay đổi điều này bằng cách chuyển sang một biểu đồ đường bao đã được tô bằng cách sử dụng hàm `plt.contourf()` (chú ý đến f ở cuối), sử dụng phần lớn cú pháp giống như hàm `plt.contour()`.

```
plt.contourf(X, Y, Z, 20, cmap='RdGy')
```

```
plt.colorbar();
```

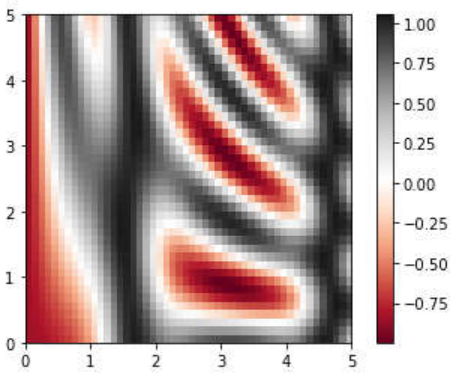
Kết quả:



Một vấn đề có thể xảy ra với hình vẽ đó là, các bước màu diễn ra rời rạc thay vì liên tục, điều này không phải lúc nào cũng như mong muốn. Ta có thể khắc phục điều này bằng cách đặt số lượng đường viền thành một con số rất cao, nhưng điều này dẫn đến một hình vẽ kém hiệu quả. Một cách tốt hơn để xử lý điều này là sử dụng hàm `plt.imshow()`, hàm này diễn giải một lưới dữ liệu hai chiều dưới dạng hình ảnh:

```
plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower', cmap='RdGy')
plt.colorbar()
plt.axis(aspect='image');
```

Kết quả:



Chú ý một số lỗi có thể xuất hiện:

- `plt.imshow()` không chấp nhận lưới `x` và `y`, vì vậy bạn phải chỉ định thủ công phạm vi `[xmin, xmax, ymin, ymax]` của hình ảnh trên biểu đồ.

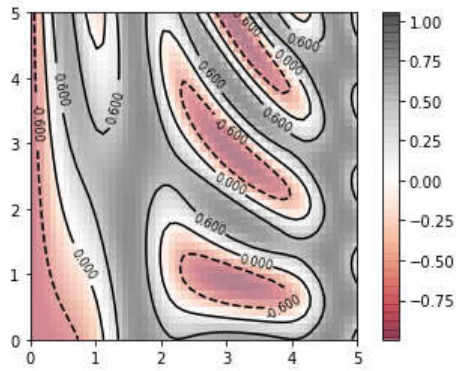
- `plt.imshow()` theo mặc định tuân theo định nghĩa mảng hình ảnh tiêu chuẩn trong đó điểm gốc nằm ở phía trên bên trái, không phải ở phía dưới bên trái như trong hầu hết các biểu đồ đường viền. Điều này phải được thay đổi khi hiển thị dữ liệu dạng lưới.

- `plt.imshow()` sẽ tự động điều chỉnh tỷ lệ khung hình trục để khớp với dữ liệu đầu vào; bạn có thể thay đổi điều này bằng cách cài đặt, ví dụ: `plt.axis(aspect = 'image')` để làm cho các đơn vị `x` và `y` khớp với nhau.

Cuối cùng, đôi khi có thể hữu ích khi kết hợp các ô đường đồng mức và các ô hình ảnh. Ví dụ: để tạo hiệu ứng được hiển thị trong hình sau, ta sẽ sử dụng hình nền trong suốt một phần (với độ trong suốt được thiết lập thông qua tham số `alpha`) và các đường viền trên hình vẽ với các nhãn trên chính đường viền (sử dụng `plt.clabel()`).

```
ontours = plt.contour(X, Y, Z, 3, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',
cmap='RdGy', alpha=0.5)
plt.colorbar();
```

Kết quả:

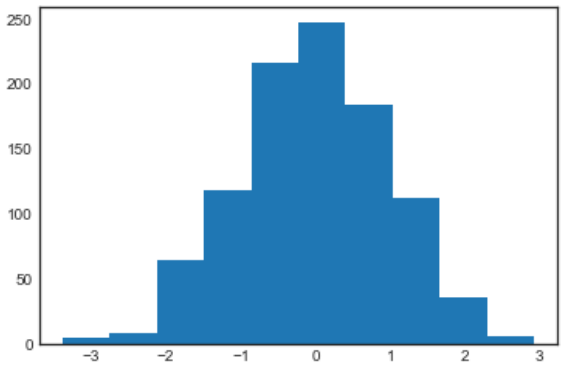


7.7. Histograms, Binnings và Density

Để vẽ biểu đồ đơn giản ta dùng hàm `hist()` như sau:

```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
data = np.random.randn(1000)
plt.hist(data);
```

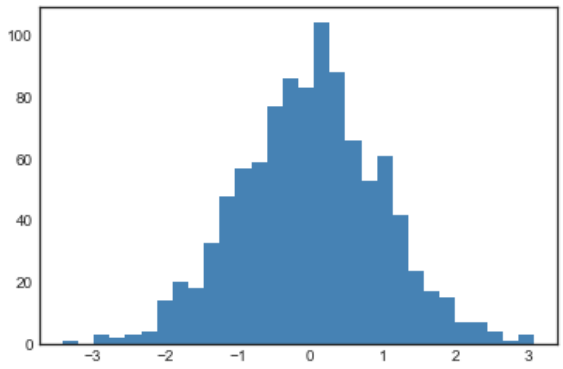
Kết quả:



Hàm `hist()` có nhiều tùy chọn để điều chỉnh cả tính toán và hiển thị; đây là một ví dụ về biểu đồ tùy chỉnh:

```
plt.hist(data, bins=30, normed=True, alpha=0.5,
histtype='stepfilled', color='steelblue', edgecolor='none')
```

Kết quả:

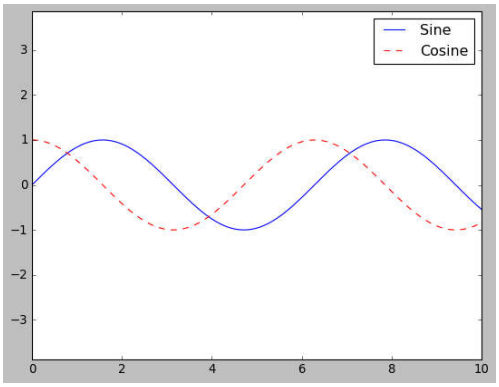


7.8. Tùy chỉnh chú thích cho hình vẽ

Chú thích hình vẽ mang lại ý nghĩa cho quan sát dữ liệu, gán nhãn cho các thành phần khác nhau của hình vẽ. Chú thích đơn giản nhất có thể được tạo bằng lệnh `plt.legend()`, lệnh này sẽ tự động tạo chú thích cho bất kỳ hình vẽ nào được gắn nhãn.

```
import matplotlib.pyplot as plt
plt.style.use('classic')
import numpy as np
x = np.linspace(0, 10, 1000)
fig, ax = plt.subplots()
ax.plot(x, np.sin(x), '-b', label='Sine')
ax.plot(x, np.cos(x), '--r', label='Cosine')
ax.axis('equal')
leg = ax.legend();
```

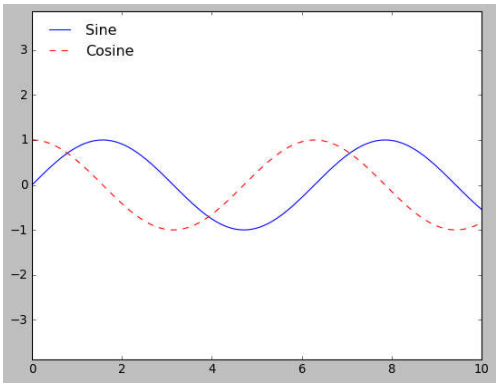
Kết quả:



Nhưng có nhiều cách ta có thể tùy chỉnh một chú thích như vậy. Ví dụ, ta có thể chỉ định vị trí và tắt khung hình:

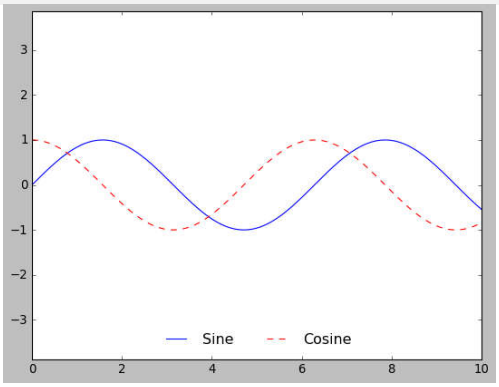
```
ax.legend(loc='upper left', frameon=False)
```

Kết quả:



Ta có thể sử dụng lệnh `ncol` để chỉ số cột của chú thích:

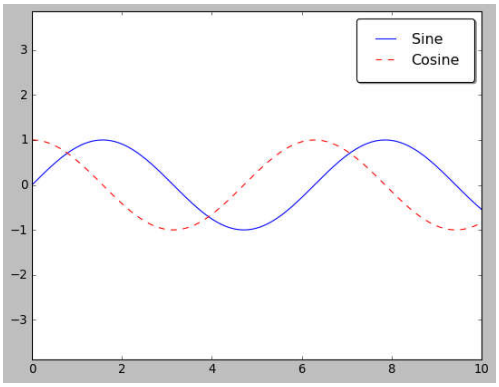
```
ax.legend(frameon=False, loc='lower center', ncol=2)
```



Chúng ta có thể sử dụng hộp làm tròn (`fancybox`) hoặc thêm `shadow`, thay đổi độ trong suốt (giá trị `alpha`) của khung hoặc thay đổi phần đệm xung quanh văn bản:

```
ax.legend(fancybox=True, framealpha=1, shadow=True, borderpad=1)
```

Kết quả:



7.9. Biểu đồ con

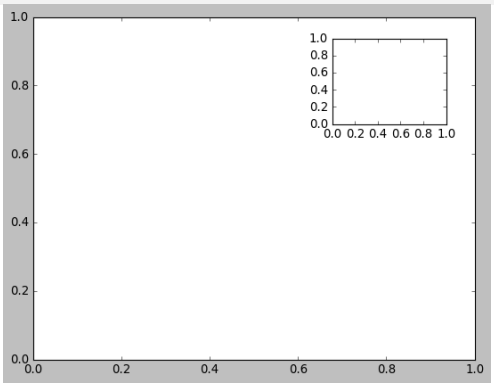
Đôi khi, việc so sánh các chế độ xem dữ liệu cạnh nhau sẽ rất hữu ích. Matplotlib có khái niệm về các biểu đồ con: nhóm các trục nhỏ hơn có thể tồn tại cùng nhau trong một hình duy nhất. Các ô con này có thể là các ô trong, các ô lưới hoặc các bố cục phức tạp hơn khác. Trong phần này, chúng ta sẽ khám phá bốn quy trình để tạo các biểu đồ con trong Matplotlib.

7.9.1. Plt.axes()

Phương pháp cơ bản nhất để tạo trục là sử dụng hàm `plt.axes()`. Mặc định, hàm tạo ra một đối tượng trục tiêu chuẩn lấp đầy toàn bộ hình. `plt.axes()` cũng nhận đối số tùy chọn là danh sách bốn số trong hệ tọa độ hình. Những con số này đại diện cho `[bottom, left, width, height]` trong hệ tọa độ hình, nằm trong khoảng từ 0 (phía dưới bên trái của hình) đến 1 (phía trên bên phải của hình).

Ví dụ: ta có thể tạo các trục lồng vào ở góc trên cùng bên phải của các trục khác bằng cách đặt vị trí `x` và `y` thành 0,65 (nghĩa là bắt đầu từ 65% chiều rộng và 65% chiều cao của hình) và x và y mở rộng đến 0,2 (nghĩa là, kích thước của các trục là 20% chiều rộng và 20% chiều cao của hình). Hình sau cho thấy kết quả của mã này:

```
ax1 = plt.axes() # standard axes
ax2 = plt.axes([0.65, 0.65, 0.2, 0.2])
```

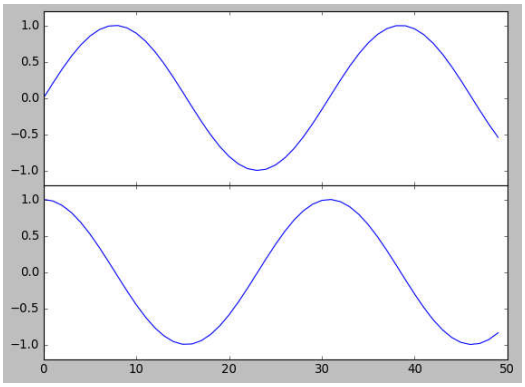


Câu lệnh khác để thêm trục là: `add_axes()` của đối tượng `figure`. Ví dụ sau vẽ

hai đường trên hai hệ trục khác nhau:

```
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.5, 0.8, 0.4],
xticklabels=[], ylim=(-1.2, 1.2))
ax2 = fig.add_axes([0.1, 0.1, 0.8, 0.4],
ylim=(-1.2, 1.2))
x = np.linspace(0, 10)
ax1.plot(np.sin(x))
ax2.plot(np.cos(x));
```

Kết quả:

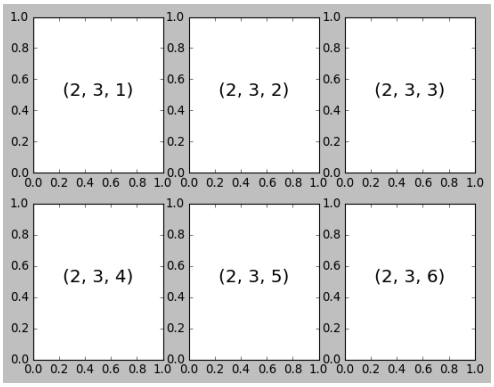


7.9.2. Hàm plt.subplot()

Hàm này nhận ba đối số nguyên: số hàng, số cột và chỉ mục của hình vẽ sẽ được tạo trong lược đồ này, chạy từ phía trên bên trái đến phía dưới bên phải, ví dụ tạo ra 6 biểu đồ con trong biểu đồ chính:

```
import matplotlib.pyplot as plt
for i in range(1, 7):
    plt.subplot(2, 3, i)
    plt.text(0.5, 0.5, str((2, 3, i)), fontsize=18, ha='center')
```

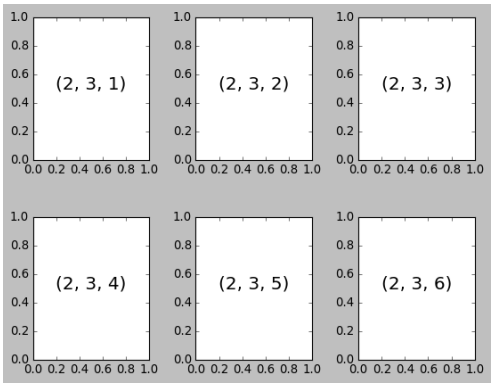
Kết quả:



Lệnh `plt.subplots_adjust()` có thể được sử dụng để điều chỉnh khoảng cách giữa các ô này:

```
import matplotlib.pyplot as plt
fig = plt.figure()
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(1, 7):
    ax = fig.add_subplot(2, 3, i)
    ax.text(0.5, 0.5, str((2, 3, i)),fontsize=18, ha='center')
```

Kết quả:

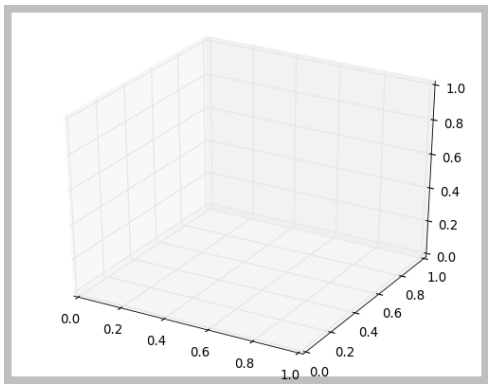


7.10. Vẽ hình ba chiều

Ban đầu Matplotlib được thiết kế chỉ với mục đích vẽ biểu đồ hai chiều. Vào khoảng thời gian phát hành phiên bản 1.0, một số tiện ích vẽ biểu đồ ba chiều đã được xây dựng trên màn hình hai chiều của Matplotlib và kết quả là một bộ công cụ thuận tiện để trực quan hóa dữ liệu ba chiều. Ta kích hoạt các ô ba chiều bằng cách `import` bộ công cụ `mplot3d`, đi kèm với cài đặt Matplotlib chính. Trước tiên ta hiển thị tọa độ ba chiều bằng lệnh:

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.axes(projection='3d')
```

Kết quả:

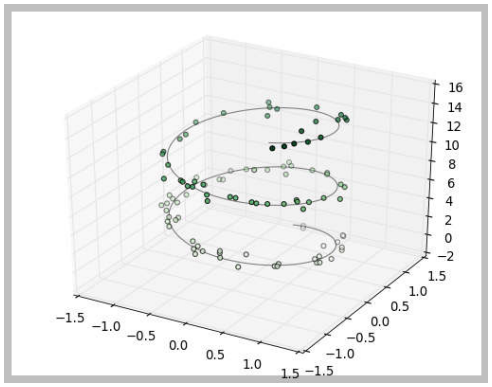


7.10.1. Vẽ điểm và đường 3D

Biểu đồ ba chiều cơ bản nhất là một đường thẳng hoặc biểu đồ phân tán được tạo từ các bộ ba `(x, y, z)`. Tương tự với các biểu đồ hai chiều thảo luận trước đó, ta có thể tạo chúng bằng cách sử dụng các hàm `ax.plot3D()` và `ax.scatter3D()`. Ví dụ, ta vẽ một hình xoắn ốc lượng giác, cùng với một số điểm được vẽ ngẫu nhiên gần đường thẳng:

```
import matplotlib.pyplot as plt
import numpy as np
ax = plt.axes(projection='3d')
# Data for a three-dimensional line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline, 'gray')
# Data for three-dimensional scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='Greens');
```

Kết quả:

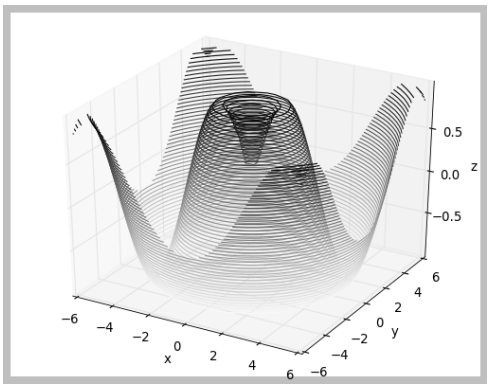


7.10.2. Đường đồng mức (contour) ba chiều

Tương tự như các ô đường đồng mức, mplot3d chứa các công cụ để tạo các ô ba chiều bằng cách sử dụng các đầu vào giống nhau. Giống như biểu đồ `ax.contour()` hai chiều, `ax.contour3D()` yêu cầu tất cả dữ liệu đầu vào phải ở dạng lưới thông thường hai chiều, với dữ liệu `Z` được đánh giá tại mỗi điểm. Ví dụ, ta sẽ hiển thị một biểu đồ đường bao ba chiều của một hàm hình sin ba chiều:

```
import matplotlib.pyplot as plt
import numpy as np
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```

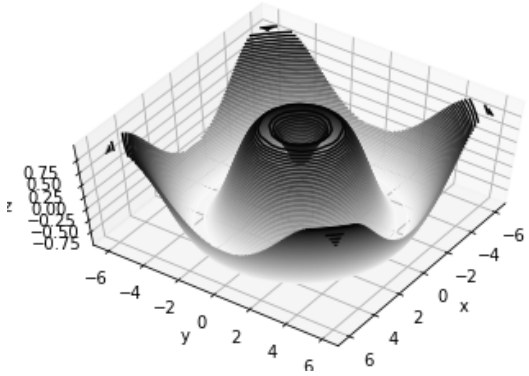
Kết quả:



Đôi khi góc nhìn mặc định không phải là tối ưu, trong trường hợp này chúng ta có thể sử dụng phương thức `view_init()` để thiết lập độ cao và góc phương vị. Trong ví dụ sau, ta sử dụng độ cao 60 độ (nghĩa là 60 độ so với mặt phẳng `x-y`) và góc phương vị là 35 độ (nghĩa là xoay 35 độ ngược chiều kim đồng hồ về trục `z`):

```
ax.view_init(60, 35)
```

Kết quả:

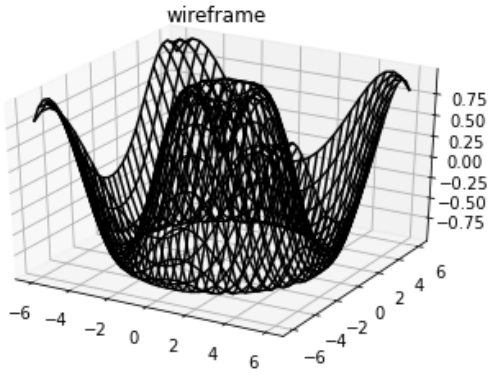


7.10.3. Khung dây và mặt ngoài

Hai loại biểu đồ ba chiều khác hoạt động trên dữ liệu lưới là khung dây và biểu đồ bề mặt. Chúng lấy một lưới các giá trị và chiếu nó lên bề mặt thời gian cụ thể và có thể tạo ra các dạng ba chiều khá dễ hình dung. Dưới đây là một ví dụ sử dụng `wireframe`:

```
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_title('wireframe');
```

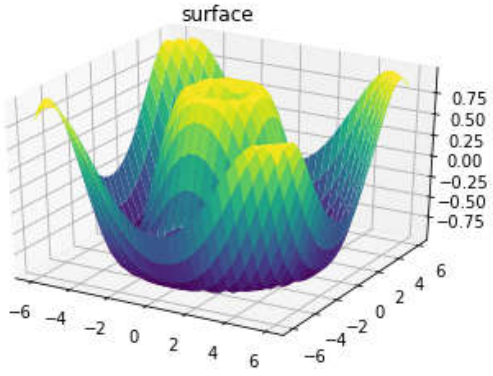
Kết quả:



Biểu đồ bề mặt giống như biểu đồ khung dây, nhưng mỗi mặt của khung dây là một đa giác được tô màu. Thêm một bản đồ màu vào các đa giác đã lấp đầy có thể hỗ trợ nhận thức về cấu trúc liên kết của bề mặt đang được hiển thị:

```
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap='viridis',
edgecolor='none')
ax.set_title('surface');
```

Kết quả:


surface
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−6
−4
−2
0
2
4
6
−6
−4
−2
0
2
4
6

CHƯƠNG 8. HỌC MÁY

8.1. Giới thiệu

Học máy là cách trích xuất tri thức từ dữ liệu. Nó là một lĩnh vực nghiên cứu ở giao điểm của thống kê, trí tuệ nhân tạo và khoa học máy tính và còn được gọi là phân tích dự đoán hoặc học thống kê. Việc áp dụng các phương pháp học máy trong những năm gần đây đã trở nên phổ biến trong cuộc sống hàng ngày. Từ các đề xuất tự động về những bộ phim nên xem, món ăn nên đặt hoặc mua sản phẩm nào, đến đài phát thanh trực tuyến được cá nhân hóa và nhận ra bạn bè của bạn trong ảnh của bạn, nhiều trang web và thiết bị hiện đại có các thuật toán học máy. Khi bạn xem một trang web phức tạp như Facebook, Amazon hoặc Netflix, rất có thể mọi phần của trang web đều chứa nhiều mô hình học máy.

8.1.1. Tại sao phải học máy

Trong những ngày đầu của các ứng dụng "thông minh", nhiều hệ thống đã sử dụng các quy tắc được mã hóa thủ công các quyết định "if" và "else" để xử lý dữ liệu hoặc điều chỉnh theo đầu vào của người dùng. Xét bộ lọc thư rác với việc chuyển các email đến vào một thư mục thư rác. Ta có thể tạo ra một danh sách đen các từ dẫn đến một email bị đánh dấu là thư rác. Đây sẽ là một ví dụ về việc sử dụng hệ thống quy tắc để thiết kế một ứng dụng thông minh. Việc xây dựng các quy tắc quyết định theo cách thủ công là khả thi đối với một số ứng dụng, đặc biệt là những ứng dụng trong đó con người hiểu rõ về quy trình để lập mô hình. Tuy nhiên, việc sử dụng các quy tắc được mã hóa thủ công để đưa ra quyết định có hai nhược điểm lớn:

• Yêu cầu để đưa ra quyết định là cụ thể cho một lĩnh vực và nhiệm vụ. Thay đổi nhiệm vụ thậm chí một chút có thể yêu cầu viết lại toàn bộ hệ thống.

• Việc thiết kế các quy tắc đòi hỏi sự hiểu biết sâu sắc về cách một chuyên gia đưa ra quyết định.

Một ví dụ về trường hợp phương pháp mã hóa thủ công sẽ không thành công là phát hiện khuôn mặt trong hình ảnh. Vấn đề chính là cách mà các pixel (tạo nên hình ảnh trong máy tính) được máy tính "cảm nhận" rất khác với cách con người cảm nhận

một khuôn mặt. Sự khác biệt trong cách biểu diễn này khiến con người về cơ bản không thể đưa ra một bộ quy tắc tốt để mô tả những gì tạo nên một khuôn mặt trong một hình ảnh kỹ thuật số.

Tuy nhiên, sử dụng học máy, chỉ cần một chương trình với một bộ sưu tập lớn hình ảnh các khuôn mặt là đủ để thuật toán xác định một khuôn mặt.

a. Các bài toán học máy có thể giải quyết

Các loại thuật toán học máy thành công nhất là những thuật toán tự động hóa quy trình ra quyết định bằng cách tổng quát hóa từ các ví dụ đã biết. Trong trường hợp này, được gọi là học có giám sát, người dùng cung cấp cho thuật toán các cặp đầu vào và đầu ra mong muốn, đồng thời thuật toán tìm cách tạo ra đầu ra mong muốn khi cung cấp một đầu vào. Đặc biệt, thuật toán có thể tạo đầu ra cho một đầu vào mà nó chưa từng thấy trước đây mà không cần bất kỳ sự trợ giúp nào từ con người. Quay trở lại ví dụ phân loại thư rác, sử dụng học máy, người dùng cung cấp thuật toán với một số lượng lớn email (là đầu vào), cùng với thông tin về việc liệu bất kỳ email nào trong số này có phải là thư rác hay không (là đầu ra mong muốn). Với một email mới, thuật toán sau đó sẽ đưa ra dự đoán liệu email mới có phải là thư rác hay không.

Ví dụ về các tác vụ học máy được giám sát bao gồm:

- Nhận dạng mã zip từ các chữ số viết tay trên phong bì: Ở đây đầu vào là bản quét chữ viết tay và đầu ra mong muốn là các chữ số thực trong mã zip. Để tạo tập dữ liệu xây dựng mô hình học máy, bạn cần thu thập nhiều phong bì. Sau đó, bạn có thể tự đọc mã zip và lưu trữ các chữ số như kết quả mong muốn của bạn.

- Xác định khối u có lành tính hay không dựa trên hình ảnh y tế: Ở đây đầu vào là hình ảnh và đầu ra là khối u có lành tính hay không. Để tạo tập dữ liệu xây dựng mô hình, bạn cần có cơ sở dữ liệu về hình ảnh y tế. Bạn cũng cần có ý kiến của chuyên gia, vì vậy bác sĩ cần xem xét tất cả các hình ảnh và quyết định khối u nào là lành tính và khối u nào không. Thậm chí có thể cần thực hiện chẩn đoán bổ sung ngoài nội dung của hình ảnh để xác định xem khối u trong hình ảnh có phải là ung thư hay không.

- Phát hiện hoạt động gian lận trong giao dịch thẻ tín dụng: Ở đây đầu vào là bản ghi của giao dịch thẻ tín dụng, và đầu ra là liệu nó có khả năng bị gian lận hay không. Giả sử rằng bạn là tổ chức phân phối thẻ tín dụng, việc thu thập tập dữ liệu có nghĩa là lưu trữ tất cả các giao dịch và ghi lại nếu người dùng báo cáo bất kỳ giao dịch nào là gian lận.

Học không giám sát là thuật toán chỉ biết dữ liệu đầu vào và không biết thông tin

dữ liệu đầu ra. Mặc dù có nhiều ứng dụng thành công của các phương pháp này, nhưng chúng thường khó hiểu và khó đánh giá hơn.

Ví dụ về học tập không có giám sát bao gồm:

Xác định chủ đề trong một tập hợp các bài đăng trên blog: Nếu bạn có một bộ sưu tập lớn dữ liệu văn bản, bạn có thể muốn tóm tắt nó và tìm các chủ đề thịnh hành trong đó. Bạn có thể không biết trước những chủ đề này là gì, hoặc có thể có bao nhiêu chủ đề. Do đó, không có kết quả đầu ra nào được biết đến.

Phân khúc khách hàng thành các nhóm có sở thích giống nhau: Với một bộ hồ sơ khách hàng, bạn có thể muốn xác định khách hàng nào giống nhau và liệu có nhóm khách hàng nào có sở thích giống nhau hay không. Đối với một trang web mua sắm, chủ đề có thể là "cha mẹ", "mọt sách" hoặc "game thủ". Vì không biết trước những nhóm này có thể là gì, hoặc thậm chí có bao nhiêu nhóm, bạn không có kết quả đầu ra nào.

Đối với cả học có giám sát và không giám sát, điều quan trọng là phải biểu diễn dữ liệu đầu vào mà máy tính có thể hiểu được. Thông thường, coi dữ liệu dưới dạng một bảng. Mỗi điểm dữ liệu (mỗi email, mỗi khách hàng, mỗi giao dịch) là một hàng và mỗi thuộc tính mô tả điểm dữ liệu đó (giả sử tuổi của khách hàng hoặc số tiền hoặc vị trí của giao dịch) là một cột. Ta có thể mô tả người dùng theo độ tuổi, giới tính, thời điểm tạo tài khoản và tần suất mua hàng từ cửa hàng trực tuyến. Bạn có thể mô tả hình ảnh của khối u bằng các giá trị thang độ xám của mỗi pixel hoặc có thể bằng cách sử dụng kích thước, hình dạng và màu sắc của khối u. Mỗi hàng được gọi là một mẫu (hoặc điểm dữ liệu), các cột được gọi là đặc trưng.

b. Biết nhiệm vụ và hiểu dữ liệu

Phần quan trọng nhất trong quá trình học máy là hiểu dữ liệu ta đang làm việc và nó liên quan như thế nào đến nhiệm vụ cần giải quyết. Sẽ không hiệu quả nếu chọn ngẫu nhiên một thuật toán và đưa dữ liệu của bạn vào nó. Cần phải hiểu những gì đang diễn ra trong tập dữ liệu trước khi bắt đầu xây dựng một mô hình. Mỗi thuật toán khác nhau về loại dữ liệu và cách đặt vấn đề phù hợp nhất với nó. Khi xây dựng một giải pháp học máy, ta nên trả lời hoặc ít nhất hãy ghi nhớ những câu hỏi sau:

• Tôi đang cố trả lời (những) câu hỏi nào? Tôi có nghĩ rằng dữ liệu thu thập được có thể trả lời câu hỏi đó không?

• Cách tốt nhất để diễn đạt (các) câu hỏi của tôi là vấn đề học máy là gì?

• Tôi đã thu thập đủ dữ liệu để biểu diễn vấn đề tôi muốn giải quyết chưa?

• Tôi đã trích xuất những đặc trưng nào của dữ liệu, và những đặc trưng này có

cho phép những dự đoán đúng không?

• Tôi sẽ đo lường thành công trong ứng dụng của mình như thế nào?

• Giải pháp máy học sẽ tương tác với các phần khác trong sản phẩm nghiên cứu hoặc kinh doanh của tôi như thế nào?

8.1.2. Tại sao Python

Python đã trở thành ngôn ngữ phổ biến cho nhiều ứng dụng khoa học dữ liệu. Nó kết hợp sức mạnh của các ngôn ngữ lập trình có mục đích chung, dễ sử dụng như MATLAB hoặc R. Python có các thư viện để tải dữ liệu, trực quan hóa, thống kê, xử lý ngôn ngữ tự nhiên, xử lý hình ảnh và hơn thế nữa. Hộp công cụ rộng lớn này cung cấp cho các nhà khoa học dữ liệu một loạt các chức năng cho mục đích chung và mục đích đặc biệt. Một trong những lợi thế chính của việc sử dụng Python là khả năng tương tác trực tiếp với mã, sử dụng thiết bị đầu cuối hoặc các công cụ khác như Jupyter Notebook mà chúng ta sẽ xem xét ngay sau đây. Học máy và phân tích dữ liệu về cơ bản là các quá trình lặp đi lặp lại, trong đó dữ liệu thúc đẩy quá trình phân tích. Điều cần thiết cho các quy trình này là phải có các công cụ cho phép lặp lại nhanh chóng và tương tác dễ dàng.

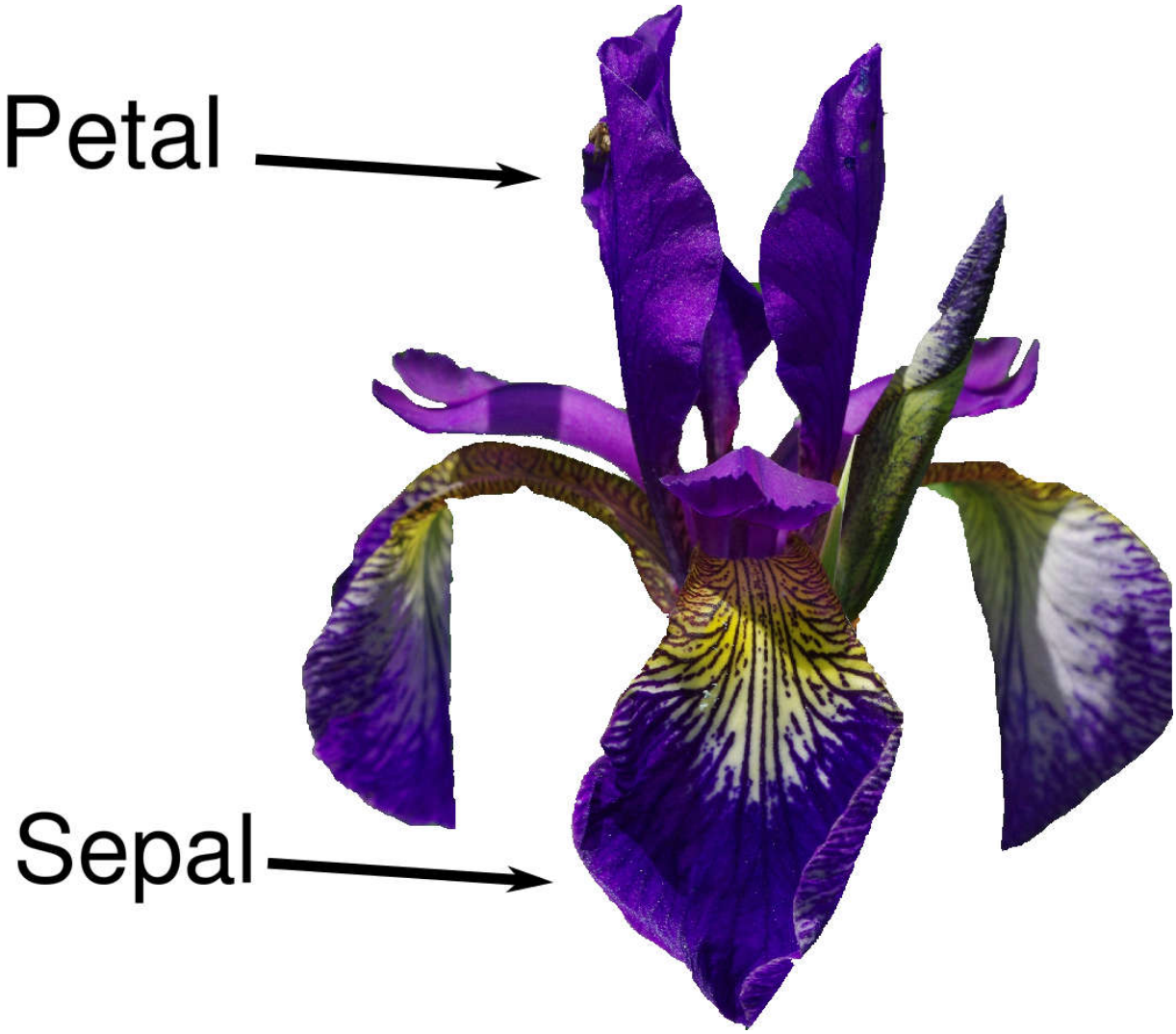
Là một ngôn ngữ lập trình có mục đích chung, Python cũng cho phép tạo ra các giao diện người dùng đồ họa phức tạp (GUI) và các dịch vụ web, cũng như để tích hợp vào các hệ thống hiện có.

8.1.3. Scikit-learn

scikit-learning là một dự án mã nguồn mở và không ngừng được phát triển và cải tiến, đồng thời nó có một cộng đồng người dùng rất tích cực. Nó chứa một số thuật toán học máy hiện đại, cũng như tài liệu toàn diện về từng thuật toán. scikit-learning là một công cụ rất phổ biến và là thư viện Python nổi bật nhất dành cho học máy. Nó được sử dụng rộng rãi trong ngành công nghiệp và học thuật, và rất nhiều hướng dẫn và đoạn mã có sẵn trực tuyến.

8.1.4. Ví dụ đầu tiên: phân loại các loài cây Iris

Trong phần này, chúng ta sẽ tìm hiểu một ứng dụng học máy đơn giản và tạo mô hình đầu tiên. Giả sử rằng một nhà thực vật học quan tâm đến việc phân biệt các loài của một số loài hoa iris. Ta có một số phép đo liên quan đến mỗi loài: chiều dài và chiều rộng của cánh hoa và chiều dài và chiều rộng của các đài hoa, tất cả được đo bằng cm (xem Hình 1-2).



Ta cũng có các phép đo của một số loài iris đã được một nhà thực vật học chuyên nghiệp xác định trước đây là thuộc loài setosa, versicolor, hoặc virginica. Đối với những phép đo này, ta có thể chắc chắn rằng mỗi iris thuộc về loài nào. Giả sử rằng đây là những loài duy nhất mà ta gặp trong tự nhiên.

Mục tiêu là xây dựng một mô hình học máy có thể học hỏi từ các phép đo của những iris có loài đã biết này, để thể dự đoán loài cho một iris mới.

Bởi vì ta có các phép đo loài iris chính xác, đây là một bài toán học có giám sát. Trong bài toán này, ta muốn dự đoán một trong số loài hoa iris. Đây là một ví dụ về bài toán phân loại. Các đầu ra có thể là các loài iris khác nhau được gọi là các lớp. Mỗi loài iris trong tập dữ liệu thuộc một trong ba lớp, vì vậy bài toán này là bài toán phân loại ba lớp.

Đầu ra mong muốn cho một điểm dữ liệu duy nhất (iris) là loài của hoa này. Đối với một điểm dữ liệu cụ thể, loài mà nó thuộc về được gọi là nhãn của nó.

a. Tập dữ liệu

Dữ liệu ta sử dụng cho ví dụ này là tập dữ liệu Iris, một tập dữ liệu cổ điển trong học máy và thống kê. Nó được tích hợp trong gói scikit-learning trong mô-đun datasets. Ta có thể tải nó bằng cách gọi hàm load_iris:

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

Đối tượng iris được load_iris trả về là đối tượng Bunch, rất giống với từ điển. Nó chứa các khóa và giá trị:

```
In[11]:
     print("Keys of iris_dataset: \n{}".format(iris_dataset.keys()))
Out[11]:
     Keys of iris_dataset:
     dict_keys(['target_names', 'feature_names', 'DESCR', 'data',
'target'])
```

Giá trị của khóa DESCR là một mô tả ngắn gọn về tập dữ liệu. Ta hiển thị phần đầu của mô tả ở đây:

```
In[12]:
     print(iris_dataset['DESCR'][:193] + "\n...")
Out[12]:
     Iris Plants Database
     ====================
     Notes
     ----
     Data Set Characteristics:
     :Number of Instances: 150 (50 in each of three classes)
     :Number of Attributes: 4 numeric, predictive att
     ...
     ----
```

Giá trị của key target_names là một mảng các chuỗi, chứa các loài hoa mà chúng ta muốn dự đoán:

```
In[13]:
     print("Target names: {}".format(iris_dataset['target_names']))
Out[13]:
     Target names: ['setosa' 'versicolor' 'virginica']
```

Giá trị của feature_names là một danh sách các chuỗi, đưa ra mô tả của từng đặc trưng:

```
In[14]:
     print("Feature
names:\n{}".format(iris_dataset['feature_names']))
Out[14]:
     Feature names:
     ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
'petal width (cm)']
```

Bản thân dữ liệu được chứa trong các trường target và data. data chứa các phép đo bằng số của chiều dài đài hoa, chiều rộng đài hoa, chiều dài cánh hoa và chiều rộng cánh hoa trong mảng NumPy:

```
In[15]:
     print("Type of data: {}".format(type(iris_dataset['data'])))
Out[15]:
     Type of data: <class 'numpy.ndarray'>
```

Các hàng trong mảng dữ liệu tương ứng với hoa, trong khi các cột đại diện cho bốn phép đo đã được thực hiện cho mỗi bông hoa:

```
In[16]:
     print("Shape of data: {}".format(iris_dataset['data'].shape))
Out[16]:
     Shape of data: (150, 4)
```

Ta thấy rằng mảng chứa các số đo cho 150 bông hoa khác nhau. Nhớ rằng các mục riêng lẻ được gọi là mẫu trong học máy và thuộc tính của chúng được gọi là đặc trưng. Hình dạng của mảng data là số lượng mẫu nhân với số lượng đặc trưng. Đây là một quy ước trong scikit-learning và dữ liệu sẽ luôn được giả định ở dạng này. Dưới đây là các giá trị đặc trưng cho năm mẫu đầu tiên:

```
In[17]:
     print("First five columns of data:\n{}"
     .format(iris_dataset['data'][:5]))
Out[17]:
     First five columns of data:
     [[ 5.1 3.5 1.4 0.2]
     [ 4.9 3. 1.4 0.2]
     [ 4.7 3.2 1.3 0.2]
     [ 4.6 3.1 1.5 0.2]
     [ 5. 3.6 1.4 0.2]]
```

Từ dữ liệu này, chúng ta có thể thấy rằng tất cả năm bông hoa đầu tiên có chiều rộng cánh hoa là 0,2 cm và bông hoa đầu tiên có lá đài dài nhất, 5,1 cm.

Mảng target chứa các loài hoa đã được đo, cũng như mảng NumPy:

```
In[18]:
     print("Type of target: {}"
     .format(type(iris_dataset['target'])))
Out[18]:
     Type of target: <class 'numpy.ndarray'>
```

target là mảng một chiều, với giá trị là nhãn bông hoa:

```
In[19]:
     print("Shape of target: {}"
     .format(iris_dataset['target'].shape))
Out[19]:
     Shape of target: (150,)
```

Các loài được mã hóa dưới dạng số nguyên từ 0 đến 2. Ý nghĩa của các con số được đưa ra bởi mảng iris['target_names']: 0 nghĩa là setosa, 1 nghĩa là versicolor và 2 nghĩa là virginica.

```
In[20]:
     print("Target:\n{}".format(iris_dataset['target']))
Out[20]:
Target:
     [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
      2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
```

b. Đo lường thành công: Dữ liệu đào tạo và kiểm tra

Ta muốn xây dựng một mô hình học máy từ dữ liệu này có thể dự đoán loài iris cho các phép đo mới. Nhưng trước khi có thể áp dụng mô hình của mình cho các phép đo mới, chúng ta cần biết liệu nó có thực sự hoạt động hay không - nghĩa là liệu chúng ta có nên tin vào các dự đoán của nó hay không.

Thật không may, ta không thể sử dụng dữ liệu đã sử dụng để xây dựng mô hình để đánh giá nó. Bởi vì mô hình của ta luôn nhớ toàn bộ tập huấn luyện và do đó sẽ luôn dự đoán nhãn chính xác cho bất kỳ điểm nào trong tập huấn luyện. Việc "ghi nhớ" này không cho biết liệu mô hình của chúng ta có khái quát hóa tốt hay không (nói cách khác, liệu nó có hoạt động tốt trên dữ liệu mới hay không).

Để đánh giá hiệu suất của mô hình, ta tách dữ liệu được gắn nhãn (150 phép đo)

thành hai phần. Một phần của dữ liệu được sử dụng để xây dựng mô hình học máy và được gọi là dữ liệu huấn luyện hoặc tập huấn luyện. Phần dữ liệu còn lại sẽ được sử dụng để đánh giá mức độ hoạt động của mô hình; đây được gọi là dữ liệu thử nghiệm, tập hợp thử nghiệm.

scikit-learning chứa một hàm xáo trộn tập dữ liệu và phân chia nó cho bạn: hàm train_test_split. Hàm này trích xuất 75% các hàng trong dữ liệu dưới dạng tập huấn luyện, cùng với các nhãn tương ứng cho dữ liệu này. 25% dữ liệu còn lại, cùng với các nhãn còn lại, được khai báo là tập kiểm tra. Quyết định lượng dữ liệu bạn muốn đưa vào tập huấn luyện và tập thử nghiệm tương ứng là tùy ý, nhưng sử dụng tập thử nghiệm chứa 25% dữ liệu là một nguyên tắc chung.

Hãy gọi train_test_split trên dữ liệu và gán kết quả đầu ra bằng cách sử dụng danh pháp này:

```
In[21]:
     from sklearn.model_selection import train_test_split
     X_train, X_test, y_train, y_test = train_test_split(
     iris_dataset['data'], iris_dataset['target'], random_state=0)
```

Trước khi thực hiện phân tách, hàm train_test_split xáo trộn tập dữ liệu bằng cách sử dụng bộ tạo số giả ngẫu nhiên. Nếu chỉ lấy 25% dữ liệu cuối cùng làm tập kiểm tra, tất cả các điểm dữ liệu sẽ có nhãn 2, vì các điểm dữ liệu được sắp xếp theo nhãn (xem kết quả đầu ra cho iris ['target'] được hiển thị trước đó). Việc sử dụng tập hợp thử nghiệm chỉ chứa một trong ba lớp sẽ không cho biết nhiều về mức độ tổng quát của mô hình, vì vậy ta xáo trộn dữ liệu của mình để đảm bảo dữ liệu thử nghiệm chứa dữ liệu từ tất cả các lớp.

Để đảm bảo rằng ta nhận được cùng một đầu ra nếu chạy cùng một hàm nhiều lần, ta cung cấp trình tạo số giả ngẫu nhiên với một hạt giống cố định bằng cách sử dụng tham số random_state. Điều này sẽ làm cho kết quả cố định, vì vậy dòng này sẽ luôn có cùng một kết quả.

Đầu ra của hàm train_test_split là X_train, X_test, y_train và y_test, tất cả đều là mảng NumPy. X_train chứa 75% số hàng của tập dữ liệu và X_test chứa 25% còn lại:

```
In[22]:
     print("X_train shape: {}".format(X_train.shape))
     print("y_train shape: {}".format(y_train.shape))
Out[22]:
     X_train shape: (112, 4)
     y_train shape: (112,)
```

```
In[23]:
     print("X_test shape: {}".format(X_test.shape))
     print("y_test shape: {}".format(y_test.shape))
Out[23]:
     X_test shape: (38, 4)
     y_test shape: (38,)
```

c. Quan sát dữ liệu

Trước khi xây dựng một mô hình học máy, ta thường nên kiểm tra dữ liệu, để xem liệu nhiệm vụ có thể giải quyết dễ dàng mà không cần học máy hay không hoặc liệu thông tin mong muốn có thể không được chứa trong dữ liệu hay không.

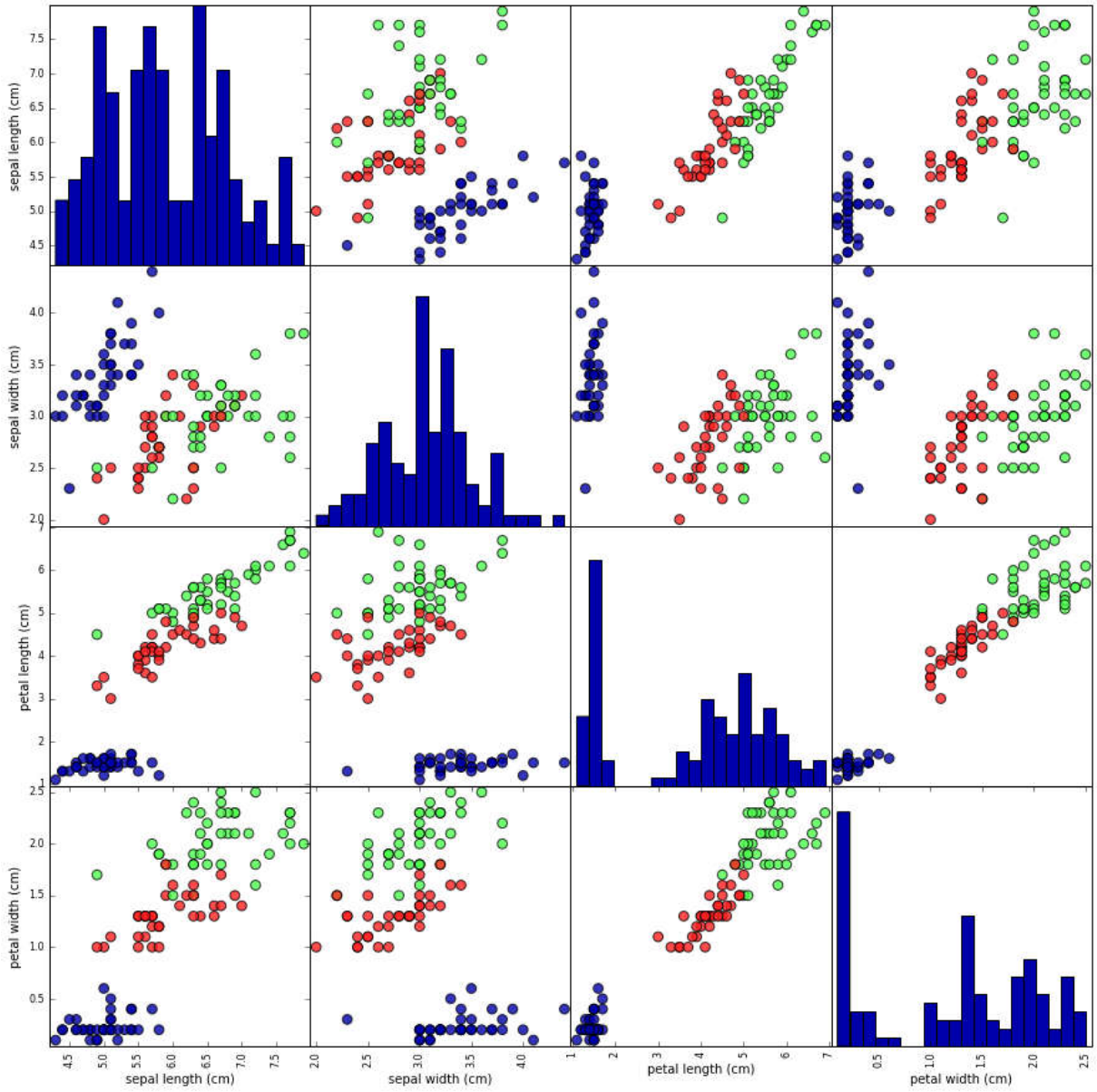
Ngoài ra, kiểm tra dữ liệu của bạn là một cách tốt để tìm ra những điểm bất thường và đặc biệt. Ví dụ, có thể một số iris được đo bằng inch chứ không phải cm.

Một trong những cách tốt nhất để kiểm tra dữ liệu là trực quan hóa nó. Một cách để làm điều này là sử dụng biểu đồ phân tán. Biểu đồ phân tán của dữ liệu đặt một đặc trưng dọc theo trục x và đặc trưng khác theo trục y và vẽ một dấu chấm cho mỗi điểm dữ liệu. Thật không may, màn hình máy tính chỉ có hai chiều, cho phép chúng ta vẽ chỉ hai (hoặc có thể ba) đối tượng cùng một lúc. Rất khó để vẽ các tập dữ liệu có nhiều hơn ba đặc trưng theo cách này.

Một cách để giải quyết vấn đề này là thực hiện một biểu đồ theo cặp, biểu đồ này xem xét tất cả các cặp đặc điểm có thể có. Nếu có một số lượng nhỏ các đặc trưng, chẳng hạn như bốn đặc trưng, điều này khá hợp lý. Tuy nhiên, biểu đồ cặp không thể hiện sự tương tác của tất cả các đặc trưng cùng một lúc.

Hình 1-3 là một cặp biểu đồ của các tính năng trong tập huấn luyện. Các điểm dữ liệu được tô màu theo loài iris. Để tạo biểu đồ, trước tiên ta chuyển đổi mảng NumPy thành DataFrame. pandas có một hàm để tạo ra các cặp biểu đồ được gọi là scatter_matrix.

```
In[24]:
# create dataframe from data in X_train
# label the columns using the strings in iris_dataset.feature_names
     iris_dataframe =
     pd.DataFrame(X_train,columns=iris_dataset.feature_names)
# create a scatter matrix from the dataframe, color by y_train
     grr = pd.scatter_matrix(iris_dataframe, c=y_train,
          figsize=(15, 15), marker='o', hist_kwds={'bins': 20},
          s=60, alpha=.8, cmap=mglearn.cm3)
```



d. Mô hình học máy đầu tiên

Có nhiều thuật toán phân loại trong scikit-learning có thể sử dụng. Ở đây ta sẽ sử dụng bộ phân loại k-láng giềng gần nhất. Việc xây dựng mô hình này chỉ bao gồm việc lưu trữ tập huấn luyện. Để đưa ra dự đoán cho một điểm dữ liệu mới, thuật toán tìm điểm trong tập huấn luyện gần nhất với điểm mới. Sau đó, nó gán nhãn của điểm huấn luyện cho điểm dữ liệu mới.

K trong k-láng giềng gần nhất biểu thị rằng thay vì chỉ sử dụng 1 láng giềng gần nhất cho điểm dữ liệu mới, ta có thể lấy bất kỳ số k cố định nào của láng giềng trong quá trình huấn luyện (ví dụ: ba hoặc năm láng giềng gần nhất). Sau đó, ta có thể đưa ra dự đoán bằng cách sử dụng lớp đa số này.

Tất cả các mô hình học máy trong scikit-learning đều được cài đặt trong các lớp

riêng, được gọi là các lớp Estimator. Thuật toán phân loại k-láng giềng gần nhất được cài đặt trong lớp KNeighborsClassifier trong mô-đun neighbors. Trước khi có thể sử dụng mô hình, chúng ta cần khởi tạo lớp thành một đối tượng. Đây là lúc chúng ta sẽ thiết lập bất kỳ thông số nào của mô hình. Tham số quan trọng nhất của KNeighborsClassifier là số lượng hàng xóm, ta đặt thành 1:

```
In[25]:
     from sklearn.neighbors import KNeighborsClassifier
     knn = KNeighborsClassifier(n_neighbors=1)
```

Đối tượng knn chứa thuật toán được sử dụng để xây dựng mô hình từ dữ liệu huấn luyện, cũng như chứa thuật toán để đưa ra dự đoán về các điểm dữ liệu mới. Nó cũng giữ thông tin mà thuật toán đã trích xuất từ dữ liệu huấn luyện. Trong trường hợp của KNeighborsClassifier, nó sẽ chỉ lưu trữ tập huấn luyện.

Để xây dựng mô hình trên tập huấn luyện, ta gọi phương thức fit của đối tượng knn, phương thức này nhận các đối số là mảng NumPy X_train chứa dữ liệu huấn luyện và mảng NumPy y_train của các nhãn huấn luyện tương ứng:

```
In[26]:
     knn.fit(X_train, y_train)
Out[26]:
     KNeighborsClassifier(algorithm='auto', leaf_size=30,
     metric='minkowski', metric_params=None, n_jobs=1,
     n_neighbors=1, p=2, weights='uniform')
```

e. Ra quyết định

Bây giờ, ta có thể đưa ra dự đoán bằng cách sử dụng mô hình này trên dữ liệu mới mà có thể không biết nhãn chính xác. Hãy tưởng tượng ta tìm thấy một hoa iris trong tự nhiên với chiều dài đài hoa là 5 cm, chiều rộng đài hoa là 2,9 cm, chiều dài cánh hoa là 1 cm và chiều rộng cánh hoa là 0,2 cm. Loài iris này sẽ là gì? ta có thể đưa dữ liệu này vào một mảng NumPy, số lượng mẫu (1) với số lượng đặc trưng (4):

```
In[27]:
     X_new = np.array([[5, 2.9, 1, 0.2]])
     print("X_new.shape: {}".format(X_new.shape))
Out[27]:
     X_new.shape: (1, 4)
```

Lưu ý rằng ta lưu các phép đo của bông hoa đơn lẻ này thành một hàng trong một mảng NumPy hai chiều, vì scikit-learning yêu cầu các mảng hai chiều cho dữ liệu.

Để đưa ra dự đoán, ta gọi phương thức predict của đối tượng knn:

```
In[28]:
     prediction = knn.predict(X_new)
     print("Prediction: {}".format(prediction))
     print("Predicted target name: {}"
     .format(iris_dataset['target_names'][prediction]))
Out[28]:
     Prediction: [0]
     Predicted target name: ['setosa']
```

f. Đánh giá mô hình

Sử dụng tập dữ liệu X_test đã tạo trước đó để đánh giá mô hình. Ta có thể đưa ra dự đoán cho từng iris trong dữ liệu thử nghiệm và so sánh với nhãn của nó (loài đã biết). Ta có thể tính toán độ chính xác bằng cách so sánh giá trị dự đoán với giá trị đã biết:

```
In[29]:
     y_pred = knn.predict(X_test)
     print("Test set predictions:\n {}".format(y_pred))
Out[29]:
     Test set predictions:
     [2 1 0 2 0 2 0 1 1 1 2 1 1 1 1 0 1 1 0 0 2 1 0 0 2 0 0 1 1 0 2
      1 0 2 2 1 0 2]
In[30]:
     print("Test set score: {:.2f}".format(np.mean(
     y_pred == y_test)))
Out[30]:
     Test set score: 0.97
```

Chúng ta cũng có thể sử dụng phương thức score của đối tượng knn, phương thức này sẽ tính toán độ chính xác của bộ thử nghiệm:

```
In[31]:
     print("Test set score: {:.2f}".format(knn.score(
     X_test, y_test)))
Out[31]:
     Test set score: 0.97
```

Đối với mô hình này, độ chính xác của bộ thử nghiệm là khoảng 0,97, có nghĩa là ta đã dự đoán đúng cho 97% iris trong bộ thử nghiệm.

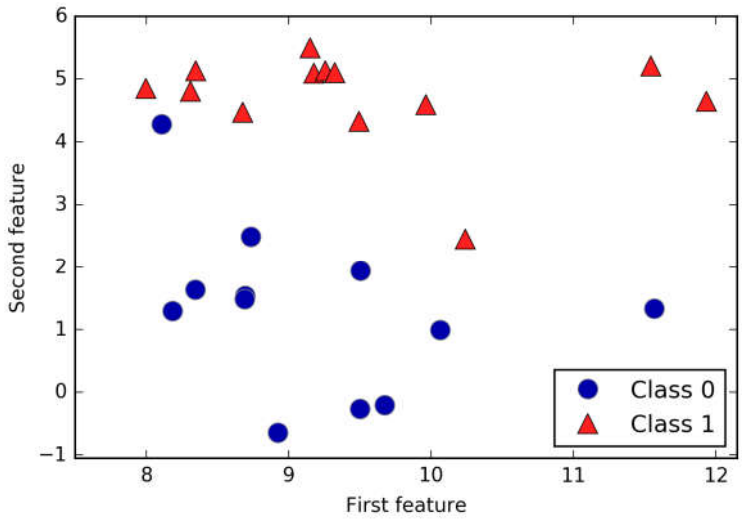
8.2. Học giám sát

8.2.1. Một số tập dữ liệu mẫu

Ta sử dụng một số bộ dữ liệu để minh họa các thuật toán khác nhau. Một số bộ dữ liệu sẽ nhỏ và tổng hợp (có nghĩa là được tạo ra), được thiết kế để làm nổi bật các khía cạnh cụ thể của các thuật toán. Các bộ dữ liệu khác sẽ là các ví dụ lớn, trong thế giới thực.

Một ví dụ về tập dữ liệu phân loại hai lớp là tập dữ liệu forge, có hai đặc trưng. Đoạn mã sau tạo một biểu đồ phân tán (Hình 2-2) hiển thị tất cả các điểm dữ liệu trong tập dữ liệu này. Đồ thị có đặc trưng đầu tiên trên trục x và đặc trưng thứ hai trên trục y. Mỗi điểm dữ liệu được biểu diễn dưới dạng một dấu chấm. Màu sắc và hình dạng của dấu chấm cho biết lớp của nó:

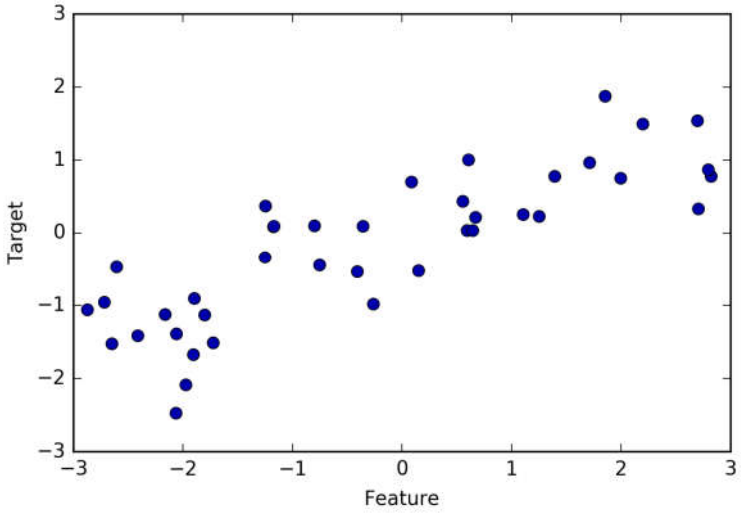
```
In[2]:
      # generate dataset
      X, y = mglearn.datasets.make_forge()
      # plot dataset
      mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
      plt.legend(["Class 0", "Class 1"], loc=4)
      plt.xlabel("First feature")
      plt.ylabel("Second feature")
      print("X.shape: {}".format(X.shape))
Out[2]:
      X.shape: (26, 2)
```



Để minh họa các thuật toán hồi quy, ta sử dụng tập dữ liệu wave. Tập dữ liệu @wave có một đặc trưng đầu vào duy nhất và một biến mục tiêu liên tục (hoặc phản hồi) mà ta muốn lập mô hình. Biểu đồ được tạo ở Hình 2-3 cho thấy đặc trưng trên trục x và mục tiêu hồi quy (đầu ra) trên trục y:

```
In[3]:
      X, y = mglearn.datasets.make_wave(n_samples=40)
```

```
    plt.plot(X, y, 'o')
    plt.ylim(-3, 3)
    plt.xlabel("Feature")
    plt.ylabel("Target")
```



Ta bổ sung các tập dữ trong thế giới thực được đưa vào scikit-learning. Một là bộ dữ liệu về Ung thư vú Wisconsin (viết tắt là cancer), ghi lại các phép đo lâm sàng của các khối u ung thư vú. Mỗi khối u được dán nhãn là "benign" (đối với khối u vô hại) hoặc "malignant" (đối với khối u ung thư) và nhiệm vụ là tìm hiểu để dự đoán liệu khối u có ác tính hay không dựa trên các phép đo của mô.

Dữ liệu có thể được tải bằng cách sử dụng hàm load_breast_cancer từ scikit-learning:

```
In[4]:
    from sklearn.datasets import load_breast_cancer
    cancer = load_breast_cancer()
    print("cancer.keys(): \n{}".format(cancer.keys()))
Out[4]:
    cancer.keys():
    dict_keys(['feature_names', 'data', 'DESCR', 'target',
    'target_names'])
```

Tập dữ liệu bao gồm 569 điểm dữ liệu, mỗi điểm có 30 đặc trưng:

```
In[5]:
    print("Shape of cancer data: {}".format(cancer.data.shape))
Out[5]:
    Shape of cancer data: (569, 30)
```

Trong số 569 điểm dữ liệu này, 212 điểm được dán nhãn là malignant và 357 điểm là benign:

```
In[6]:
     print("Sample counts per class:\n{}".format({n: v for n, v in
     zip(cancer.target_names, np.bincount(cancer.target))}))
Out[6]:
     Sample counts per class:
     {'benign': 357, 'malignant': 212}
```

Để xem mô tả về ý nghĩa của từng đặc trưng, ta có thể xem thuộc tính feature_names:

```
In[7]:
     print("Feature names:\n{}".format(cancer.feature_names))
Out[7]:
     Feature names:
     ['mean radius' 'mean texture' 'mean perimeter' 'mean area'
     'mean smoothness' 'mean compactness' 'mean concavity'
     'mean concave points' 'mean symmetry' 'mean fractal dimension'
     'radius error' 'texture error' 'perimeter error' 'area error'
     'smoothness error' 'compactness error' 'concavity error'
     'concave points error' 'symmetry error' 'fractal dimension
     error''worst radius' 'worst texture' 'worst perimeter'
     'worst area' 'worst smoothness' 'worst compactness'
     'worst concavity' 'worst concave points' 'worst symmetry'
     'worst fractal dimension']
```

Có thể tìm hiểu thêm về dữ liệu bằng cách đọc cancer.DESCR.

Ta sử dụng tập dữ liệu hồi quy trong thế giới thực, tập dữ liệu Nhà ở Boston. Nhiệm vụ liên quan đến tập dữ liệu này là dự đoán giá trị trung bình của các ngôi nhà ở một số khu vực lân cận Boston vào những năm 1970, sử dụng thông tin như tỷ lệ tội phạm, mức độ gần sông Charles, khả năng tiếp cận đường cao tốc, v.v. Tập dữ liệu chứa 506 điểm dữ liệu, được mô tả bởi 13 đặc trưng:

```
In[8]:
     from sklearn.datasets import load_boston
     boston = load_boston()
     print("Data shape: {}".format(boston.data.shape))
Out[8]:
     Data shape: (506, 13)
```

8.2.2. Thuật toán K láng giềng gần nhất (k-NN)

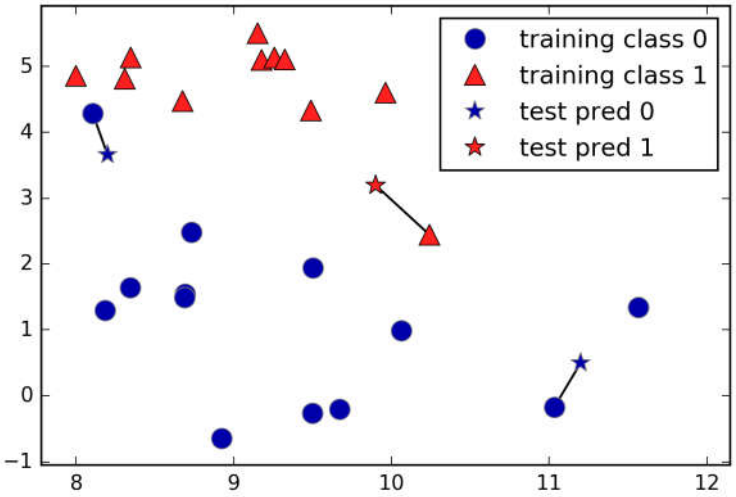
Thuật toán k-NN được cho là thuật toán học máy đơn giản nhất. Việc xây dựng mô hình chỉ bao gồm việc lưu trữ tập dữ liệu đào tạo. Để đưa ra dự đoán cho một điểm

dữ liệu mới, thuật toán tìm các điểm dữ liệu gần nhất trong tập dữ liệu huấn luyện - "những người hàng xóm gần nhất" của nó.

a. Phân loại k láng giềng

Trong phiên bản đơn giản nhất của nó, thuật toán k-NN chỉ xem xét chính xác một láng giềng gần nhất, đó là điểm dữ liệu huấn luyện gần nhất với điểm mà ta muốn đưa ra dự đoán. Dự đoán sau đó chỉ đơn giản là đầu ra đã biết cho điểm đào tạo này. Hình 2-4 minh họa điều này cho trường hợp phân loại trên tập dữ liệu forge:

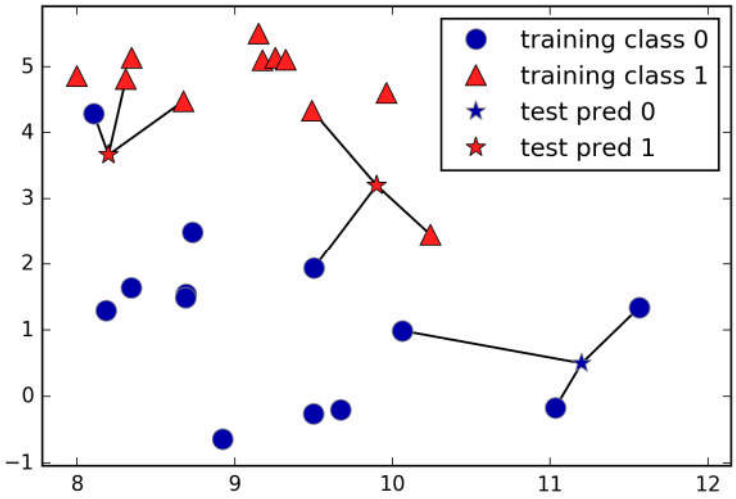
```
In[10]:
     mglearn.plots.plot_knn_classification(n_neighbors=1)
```



Ở đây, ta đã thêm ba điểm dữ liệu mới, được hiển thị dưới dạng các ngôi sao. Đối với mỗi ngôi sao, ta đánh dấu điểm gần nhất trong tập huấn luyện. Dự đoán của thuật toán 1-NN là nhãn của điểm đó (được thể hiện bằng màu).

Thay vì xét 1 hàng xóm gần nhất, ta cũng có thể xét một số k tùy ý, của các hàng xóm. Khi xét nhiều hơn một hàng xóm, ta sử dụng "biểu quyết" để gán nhãn. Điều này có nghĩa là đối với mỗi điểm kiểm tra, ta đếm có bao nhiêu hàng xóm thuộc lớp 0 và bao nhiêu hàng xóm thuộc lớp 1. Sau đó, ta gán lớp xuất hiện nhiều hơn: tức là, lớp đa số trong số k láng giềng gần nhất. Ví dụ sau (Hình 2-5) sử dụng ba láng giềng gần nhất:

```
In[11]:
     mglearn.plots.plot_knn_classification(n_neighbors=3)
```



Trong khi minh họa này dành cho bài toán phân loại nhị phân, phương pháp này có thể được áp dụng cho các tập dữ liệu với số lớp bất kì. Đối với nhiều hơn hai lớp, ta đếm xem có bao nhiêu hàng xóm thuộc mỗi lớp và chọn dự đoán là lớp xuất hiện nhiều nhất.

Bây giờ, hãy xem cách áp dụng thuật toán k-NN bằng cách sử dụng scikit-learning. Đầu tiên, ta chia dữ liệu thành một tập huấn luyện và một tập kiểm tra để đánh giá hiệu suất tổng quát hóa:

```
In[12]:
     from sklearn.model_selection import train_test_split
     X, y = mglearn.datasets.make_forge()
     X_train, X_test, y_train, y_test = train_test_split(X, y,
     random_state=0)
```

Tiếp theo, ta nhập và khởi tạo lớp. Tại đây ta thiết lập các tham số, chẳng hạn như số lượng hàng xóm sẽ sử dụng (k=3):

```
In[13]:
     from sklearn.neighbors import KNeighborsClassifier
     clf = KNeighborsClassifier(n_neighbors=3)
```

Bây giờ, ta đưa dữ liệu cho bộ phân loại bằng cách sử dụng tập huấn luyện. Đối với KNeighborsClassifier, điều này có nghĩa là lưu trữ tập dữ liệu:

```
In[14]:
     clf.fit(X_train, y_train)
```

Để đưa ra dự đoán trên dữ liệu thử nghiệm, ta gọi là phương pháp dự đoán. Đối với mỗi điểm dữ liệu trong tập kiểm tra, ta tính toán các hàng xóm gần nhất của nó trong tập huấn luyện và tìm ra lớp phổ biến nhất trong số này:

```
In[15]:
     print("Test set predictions: {}".format(clf.predict(X_test)))
Out[15]:
     Test set predictions: [1 0 1 0 1 0 0]
```

Để đánh giá mô hình tổng quát như thế nào, ta gọi phương thức score với dữ liệu kiểm tra cùng với các nhãn kiểm tra:

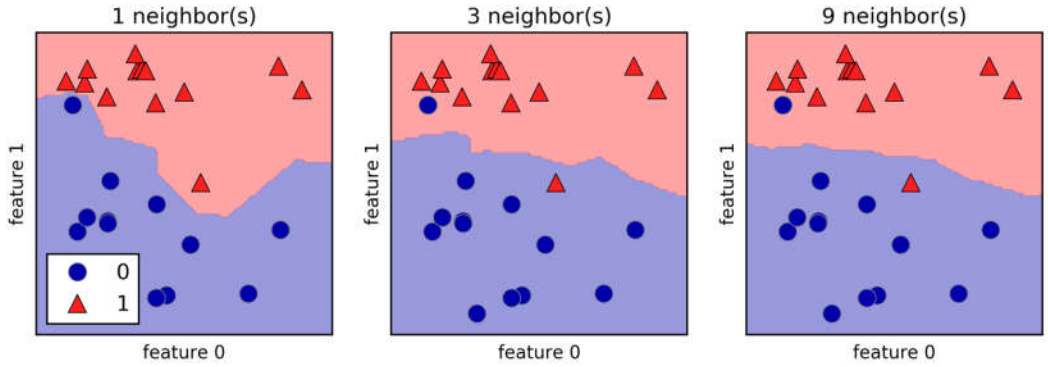
```
In[16]:
     print("Test set accuracy: {:.2f}".format(clf.score(
     X_test, y_test)))
Out[16]:
     Test set accuracy: 0.86
```

Ta thấy rằng mô hình chính xác khoảng 86%, có nghĩa là mô hình đã dự đoán phân loại chính xác cho 86% số mẫu trong tập dữ liệu thử nghiệm.

b. Phân tích KneighborsClassifier

Đối với tập dữ liệu hai chiều, ta minh họa dự đoán cho tất cả các điểm kiểm tra trong mặt phẳng xy. Ta tô màu mặt phẳng theo màu của lớp được gán cho một điểm trong vùng này. Điều này cho phép ta xem ranh giới quyết định giữa lớp 0 và lớp 1. Đoạn mã sau tạo ra các hình ảnh trực quan về ranh giới quyết định cho k = 1, 3, 9 được hiển thị trong Hình 2-6:

```
In[17]:
     fig, axes = plt.subplots(1, 3, figsize=(10, 3))
     for n_neighbors, ax in zip([1, 3, 9], axes):
     # the fit method returns the object self, so we can instantiate
     # and fit in one line
     clf = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X, y)
     mglearn.plots.plot_2d_separator(clf, X, fill=True, eps=0.5,
                                     ax=ax, alpha=.4)
     mglearn.discrete_scatter(X[:, 0], X[:, 1], y, ax=ax)
     ax.set_title("{} neighbor(s)".format(n_neighbors))
     ax.set_xlabel("feature 0")
     ax.set_ylabel("feature 1")
     axes[0].legend(loc=3)
```



c. Hồi quy k láng giềng