

Python

☑ Trắc nghiệm 6 ⏴ PDF bài

Từ bài sau trở đi, mọi code trong khóa học đều viết bằng Python: đọc dữ liệu bằng pandas, tính toán bằng NumPy, huấn luyện mô hình bằng scikit-learn và PyTorch. Vòng lặp, hàm, thuật toán đã học ở C++ vẫn dùng như cũ. Bài này chỉ tập trung vào những chỗ Python khác C++ dễ làm ra kết quả sai: chia và lấy dư với số âm, gán list mà hai biến dùng chung một list, và notebook vẫn nhớ biến cũ sau khi đã sửa code.

MỤC TIÊU

- Tính đúng phép chia `//`, `%` với số âm; so sánh số thực.
- Biết khi nào hai biến dùng chung một list, và cách sao chép list.
- Chọn list, set hay dict theo độ phức tạp; đo thời gian chạy.
- Đọc, ghi file CSV; đọc thông báo lỗi để sửa code.

Notebook

Code trong khóa học chạy trong notebook (Jupyter, Colab hoặc Kaggle). Notebook chia code thành nhiều ô (cell). Mỗi ô chạy riêng được, theo thứ tự nào cũng được, và mọi ô dùng chung biến. Các biến này nằm trong một tiến trình Python chạy ngầm gọi là kernel; khởi động lại kernel là xóa hết biến.

Vì vậy notebook nhớ kết quả của lần chạy gần nhất, không phải code đang hiện trên màn hình. Ví dụ ô trên có `n = 3`, ô dưới có `total = sum(range(n + 1))`, chạy cả hai được `total = 6`. Sửa ô trên thành `n = 5` và chỉ chạy lại ô trên: `n` đã là 5, nhưng `total` vẫn là 6, vì ô dưới chưa chạy lại.

1. Chạy cả hai ô

```
n = 3
```

```
total = sum(range(n + 1))
```

total = 6

2. Sửa n, chỉ chạy ô trên

```
n = 5
```

```
total = sum(range(n + 1))
```

total = 6 cũ

3. Chạy lại ô dưới

```
n = 5
```

```
total = sum(range(n + 1))
```

total = 15

Sửa n mà chưa chạy lại ô dưới thì total vẫn là 6.

Đoạn code sau làm lại đúng các bước đó trong một ô:

Python

```
n = 3
total = sum(range(n + 1))
n = 5                                # giống như chạy lại riêng ô đầu
print(n, total)
total = sum(range(n + 1))           # chạy lại bước phụ thuộc
print(n, total)
```

KẾT QUẢ

```
5 6
5 15
```

Cùng `n = 5`, nhưng `total` chỉ thành 15 sau khi dòng tính tổng chạy lại. Vì vậy trước khi nộp hay gửi notebook cho người khác, hãy khởi động lại kernel rồi chạy tất cả ô từ trên xuống (*Restart & Run all*). Kết quả của lượt chạy đó mới là kết quả thật của notebook.

HỎI NHANH

Một notebook có hai ô: ô trên là `n = 3`, ô dưới là `total = sum(range(n + 1))`. Hôm qua đã chạy cả hai ô. Hôm nay mở lại notebook (kernel mới), ta bấm chạy ô dưới trước, chưa chạy ô trên. Ô dưới cho ra gì?

- A `total` bằng 0, vì `n` chưa gán thì Python coi là 0
- B `total` bằng 6, vì Python đọc hết notebook rồi mới chạy
- C Lỗi `NameError`, vì chưa có biến `n`
- D `total` bằng 6, vì dùng lại `n` của lần chạy hôm qua

Chạy notebook trên Colab và Kaggle

Notebook của mọi bài nằm trong thư mục `docs/giao_trinh/notebook` của [repo giáo trình](#). Colab mở được thẳng từ GitHub (*File* → *Open notebook* → *GitHub*); Kaggle nhập qua *File* → *Import Notebook*. Chương 0 đến 4 chạy được trên CPU; từ chương 5 nên bật GPU.

- Colab: *Runtime* → *Change runtime type* → T4 GPU, rồi *Runtime* → *Run all*. Tệp tạo trong phiên mất khi phiên đóng.
- Kaggle: trong *Session options*, chọn *Accelerator* là GPU T4 x2 hoặc GPU P100; bật *Internet* nếu cần tải dữ liệu. Hai mục này chỉ mở khi tài khoản đã xác minh số điện thoại ở *Settings* → *Phone verification*.

Ô đầu tiên nên kiểm GPU:

Python

```
import torch

device = 'cuda' if torch.cuda.is_available() else 'cpu'
print('device:', device)
```

KẾT QUẢ

device: cpu

Máy dựng giáo trình không có GPU nên in cpu; bật GPU đúng thì in cuda. Trên Kaggle, *Save Version* → *Save & Run All* chạy lại notebook từ đầu và lưu output.

HỎI NHANH

Trên Kaggle, mục *Accelerator* chỉ chọn được *None*. Lý do thường gặp nhất là gì?

- A Notebook có lỗi cú pháp
- B Tài khoản chưa xác minh số điện thoại
- C Kaggle chỉ mở GPU vào cuối tuần
- D Chưa cài torch bằng pip

Cú pháp cơ bản

Ai đã dùng Python quen thì có thể làm phần Kiểm tra cuối bài trước; đúng hết thì sang luôn bài 0.2.

Python dùng thụt lề để chia khối, không dùng {}. Dấu : mở thân hàm, vòng lặp hay câu điều kiện; các dòng trong cùng khối thụt vào bằng nhau, thường là bốn dấu cách. Hàm sau lọc ra các điểm từ 5 trở lên:

Python

```
def passed_scores(values, threshold=5.0):  
    result = []  
    for value in values:  
        if value >= threshold:  
            result.append(value)  
    return result
```

```
print(passed_scores([4.5, 7.0, 5.0]))
print(list(range(1, 5)))          # đầu được lấy, cuối không được lấy
```

KẾT QUẢ

```
[7.0, 5.0]
[1, 2, 3, 4]
```

`return` thực tế ngang với `for`, nên nó chạy sau khi vòng lặp xong; nếu thực thêm một mức vào trong vòng lặp, hàm sẽ trả về ngay ở lần lặp đầu. `range(1, 5)` lấy từ 1 đến 4, không lấy 5; cách lấy đầu, bỏ cuối này dùng chung cho slicing của list và NumPy.

Số và phép toán

`int` của Python có độ chính xác tùy ý: số lớn được cấp thêm bộ nhớ, không quay vòng tại một mốc 32 hay 64 bit. Giới hạn thực tế vẫn là bộ nhớ và thời gian; nhân hai số hàng triệu chữ số không còn là phép toán hằng thời gian.

Python

```
print(2 ** 100)
print(2 ** 63 - 1 + 1)          # vượt LLONG_MAX của C++ vẫn đúng
```

KẾT QUẢ

```
1267650600228229401496703205376
9223372036854775808
```

Phép chia thì có hai dạng. `/` luôn cho số thực, kể cả khi chia hết. `//` là chia lấy phần nguyên, nhưng làm tròn **xuống** (về phía âm vô cùng), không phải làm tròn về 0 như C++. Phép `%` đi theo `//`, nên với số chia dương thì kết quả luôn không âm:

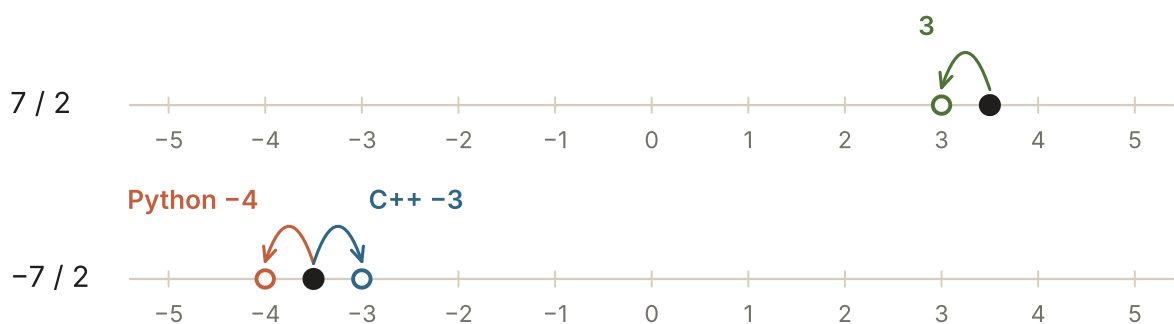
Python

```
print(7 / 2, 6 / 2)
print(7 // 2, -7 // 2)
print(7 % 3, -7 % 3)
print(int(-3.7))                # int() cắt phần thập phân, về phía 0
```

KẾT QUẢ

```
3.5 3.0
3 -4
1 2
-3
```

Trong C++, $-7 / 2$ bằng -3 và $-7 \% 3$ bằng -1 . Trong Python là -4 và 2 . Khi số chia âm thì ngược lại, $\%$ cho kết quả không dương: $10 // -3$ là -4 vì -3.33 làm tròn xuống, nên $10 \% -3$ là $10 - (-4)(-3) = -2$. Quy tắc của Python giữ đúng đẳng thức $a == (a // b) * b + a \% b$ và làm $\%$ tiện hơn khi tính chỉ số vòng tròn: $(i - 1) \% n$ luôn nằm trong $[0, n)$, không cần cộng thêm n .



// làm tròn xuống; phép chia nguyên của C++ cắt về 0.

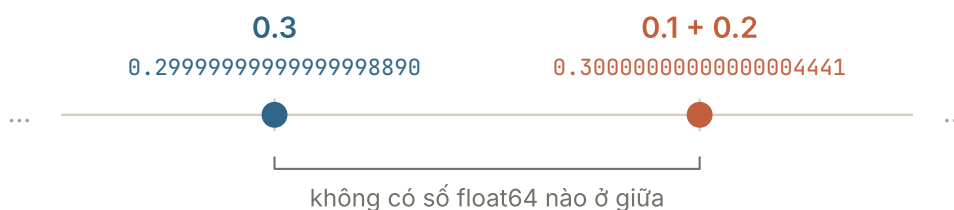
float của Python là số thực nhị phân 64 bit, giống double của C++. Phần lớn số thập phân như 0.1 không viết được chính xác bằng hữu hạn chữ số nhị phân:

Python

```
import math
print(0.1 + 0.2, 0.1 + 0.2 == 0.3)
print(math.isclose(0.1 + 0.2, 0.3, rel_tol=1e-9, abs_tol=1e-12))
print(round(2.5), round(3.5), round(0.125, 2))
```

KẾT QUẢ

```
0.30000000000000004 False
True
2 4 0.12
```



0.3 và 0.1 + 0.2 rơi vào hai số float64 liền kề.

Vì vậy không so hai số thực bằng `==`. Dùng `math.isclose`, hoặc `np.isclose` với mảng: hai số được coi là bằng nhau khi chênh lệch nhỏ hơn mức cho phép. `rel_tol` là mức cho phép tính theo độ lớn của số; `abs_tol` là mức cho phép cố định, cần khi so với 0.

Dòng cuối cho thấy `round` không làm tròn kiểu “5 thì lên”: khi số nằm đúng giữa hai mốc, nó chọn mốc chẵn. Vì thế `round(2.5)` là 2, `round(3.5)` là 4, `round(0.125, 2)` là 0.12. Nếu đề yêu cầu cách làm tròn khác thì phải tự viết theo đúng yêu cầu đó.

HỎI NHANH

`print(-7 // 2, -7 % 2)` in ra gì?

A -3 -1

B -4 1

C -3 1

D -3.5 1

Biến là cái tên, không phải cái hộp

Trong C++, `vector<int> b = a;` tạo ra một bản sao: hai cái hộp, mỗi hộp chứa dữ liệu riêng. Trong Python, phép gán không bao giờ sao chép dữ liệu. Một biến chỉ là một cái tên dán lên một đối tượng. `b = a` dán thêm tên `b` lên **cùng** đối tượng mà `a` đang chỉ tới.

Python

```
a = [1, 2, 3]
b = a
b.append(4)
print(a, b, a is b)
```

KẾT QUẢ

```
[1, 2, 3, 4] [1, 2, 3, 4] True
```

Sửa qua b thì a thấy ngay, vì chỉ có một list. Toán tử `is` hỏi “hai tên có chỉ cùng một đối tượng không”, khác với `==` hỏi “hai đối tượng có bằng nhau không”.

Vậy tại sao với số nguyên lại không thấy hiện tượng này?

Python

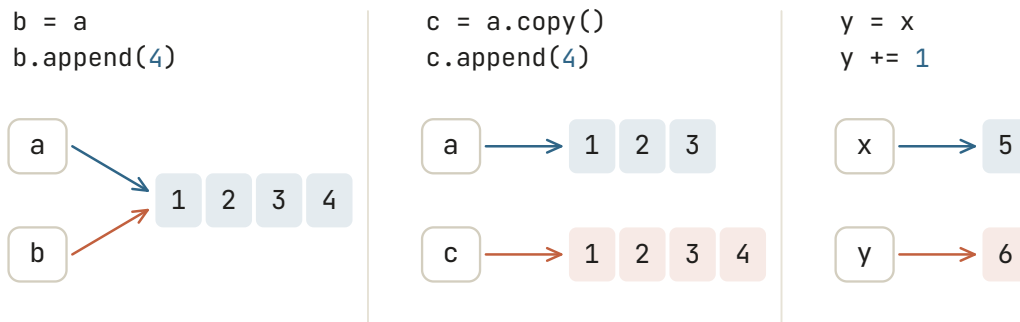
```
x = 5
y = x
y += 1
print(x, y)
```

KẾT QUẢ

```
5 6
```

Vì `int` là bất biến (immutable): không có cách nào sửa đối tượng 5. Lệnh `y += 1` tạo đối tượng mới 6 rồi dán tên `y` sang đó; `x` vẫn chỉ vào 5. Các kiểu bất biến gồm `int`, `float`, `str`, `tuple`. Các kiểu sửa được (mutable) gồm `list`, `dict`, `set`, và mảng NumPy. Hiện tượng “sửa chỗ này, đổi chỗ kia” chỉ xảy ra với nhóm thứ hai.

Với `list`, `+=` và `+` cũng khác nhau. Sau `a = [1, 2]` và `b = a`, lệnh `a += [3]` sửa thẳng list chung, nên `b` thấy `[1, 2, 3]`. Còn `a = a + [3]` tạo một list mới rồi dán tên `a` sang đó; `b` vẫn chỉ vào list cũ `[1, 2]`.



Gán chỉ dẫn thêm tên; `copy()` tạo list mới; int bất biến.

Cách hiểu này giải thích luôn chuyện truyền tham số. Tham số của hàm là tên cục bộ trở tới đối tượng được truyền vào. Sửa nội dung list qua tên đó thì người gọi thấy thay đổi; gán lại riêng tên tham số thì không gán lại tên của người gọi.

HỎI NHANH

```
def reset(scores):
    scores.append(0)
    scores = []

marks = [7, 8]
reset(marks)
print(marks)
```

Đoạn code trên in ra gì?

A []

B [7, 8]

C [0]

D [7, 8, 0]

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 94

Muốn một list độc lập thì phải nói rõ:

Python

```
a = [1, 2, 3]
c = a.copy()          # hoặc a[:], list(a)
c.append(4)
print(a, c)
```

KẾT QUẢ

```
[1, 2, 3] [1, 2, 3, 4]
```

`copy()` chỉ sao chép một tầng (shallow copy). Với list lồng nhau, các list con vẫn dùng chung. Đây là cái bẫy kinh điển khi tạo bảng hai chiều:

Python

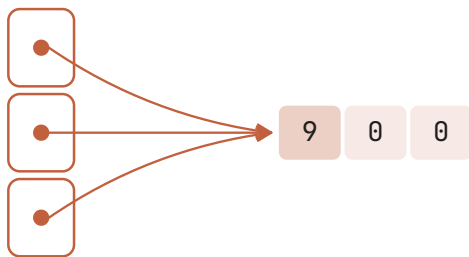
```
grid = [[0] * 3] * 3          # SAI: ba tên cho cùng một hàng
grid[0][0] = 9
print(grid)

grid = [[0] * 3 for _ in range(3)] # ĐÚNG: mỗi lần lặp tạo một hàng mới
grid[0][0] = 9
print(grid)
```

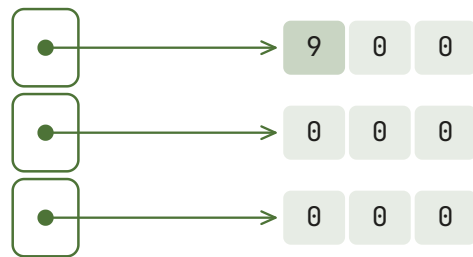
KẾT QUẢ

```
[[9, 0, 0], [9, 0, 0], [9, 0, 0]]
[[9, 0, 0], [0, 0, 0], [0, 0, 0]]
```

`[[0] * 3] * 3`



`[[0] * 3 for _ in range(3)]`



Gán 9 vào ô đầu: bên trái cả ba hàng đổi, bên phải chỉ một hàng đổi.

Phép `*` 3 bên ngoài nhân bản các tên, không nhân bản các hàng. Khi cần sao chép sâu mọi tầng, dùng `copy.deepcopy`. Trong thực tế, bảng số nên dùng mảng NumPy (bài 0.2) thay cho list lồng nhau.

HỎI NHANH

Chạy lần lượt `a = [[1, 2], [3, 4]]`, `b = a.copy()`, `b[0].append(9)`. Lúc này `a` là gì?

A `[[1, 2], [3, 4]]`

B `[[1, 2, 9], [3, 4]]`

C `[[1, 2], [3, 4], 9]`

D Báo lỗi

list, tuple, dict, set

Bốn cấu trúc dữ liệu dựng sẵn này thay cho phần lớn STL:

C++	Python	Ghi chú
<code>vector<T></code>	<code>list</code>	Chứa được nhiều kiểu; <code>append</code> có chi phí amortized $O(1)$
<code>pair, tuple</code>	<code>tuple</code>	Không đổi các vị trí; làm khóa nếu mọi phần tử hashable
<code>unordered_map<K, V></code>	<code>dict</code>	Giữ thứ tự chèn; tra cứu trung bình $O(1)$
<code>unordered_set<T></code>	<code>set</code>	Kiểm tra in trung bình $O(1)$
<code>map, set có thứ tự</code>	không có sẵn	Dùng <code>sorted()</code> khi cần, hoặc thư viện ngoài

List đánh số từ 0: list có `len(a)` phần tử nằm ở vị trí 0 tới `len(a) - 1`, `a[-1]` là phần tử cuối, còn `a[len(a)]` báo `IndexError`. Dấu `+` nối hai list chứ không cộng từng phần tử: `[1, 2] + [3, 4]` là `[1, 2, 3, 4]`. Muốn cộng từng phần tử thì dùng mảng NumPy ở bài 0.2.

Set bỏ phần tử trùng, nên câu “con mèo đuổi con chuột” có 5 từ nhưng `len(set("con mèo đuổi con chuột".split()))` bằng 4. Với hai set `A, B`, `A & B` là phần chung, `A | B` là hợp. Tỉ số `len(A & B) / len(A | B)` thường dùng để đo hai câu giống nhau đến đâu theo các từ chúng có.

Khóa của dict và phần tử của set phải hashable, tức có giá trị băm không đổi suốt đời đối tượng. Số, chuỗi và tuple chứa toàn phần tử hashable đều dùng được. List thì không: `d[[1, 2]] = "y"` báo `TypeError: unhashable type: 'list'`, còn `d[(1, 2)] = "x"` chạy bình thường. Tuple chứa một list cũng không được: bản thân tuple không đổi được, nhưng list bên trong vẫn đổi được, nên giá trị băm không thể ổn định.

Hai cú pháp sẽ gặp ở khắp nơi. Slicing `a[start:stop:step]` lấy một đoạn, với chỉ số âm đếm từ cuối. List comprehension thay cho vòng lặp tạo list:

Python

```
a = [5, 3, 8, 1, 9, 2]
print(a[1:4], a[-1], a[-2:], a[::-1])
print([x * x for x in a if x % 2 == 1])

scores = {"An": 8.5, "Bình": 6.0}
scores["Chi"] = 9.0
print(scores.get("Dũng", 0.0), len(scores), "An" in scores)
```

KẾT QUẢ

```
[3, 8, 1] 2 [9, 2] [2, 9, 1, 8, 3, 5]
[25, 9, 1, 81]
0.0 3 True
```

`dict.get(key, mặc_định)` tránh lỗi khi khóa chưa có. Để đếm tần suất, `collections.Counter` làm sẵn việc mà trong C++ ta thường tự viết bằng `map<string, int>`:

Python

```
from collections import Counter
words = "mèo chó mèo gà mèo chó".split()
print(Counter(words).most_common(2))
print(sorted(scores.items(), key=lambda kv: -kv[1]))
```

KẾT QUẢ

```
[('mèo', 3), ('chó', 2)]
[('Chi', 9.0), ('An', 8.5), ('Bình', 6.0)]
```

mèo chó mèo gà mèo chó

"mèo" 1

từ "mèo": khóa mới, đặt bằng 1

mèo chó mèo gà mèo chó

"mèo" 3

"chó" 1

"gà" 1

từ "mèo": đã có khóa, tăng 2 → 3

mèo chó mèo gà mèo chó

"mèo" 2

"chó" 1

từ "mèo": đã có khóa, tăng 1 → 2

mèo chó mèo gà mèo chó

"mèo" 3

"chó" 2

"gà" 1

most_common(2) sắp theo số lần giảm dần: mèo 3, chó 2

Đếm "mèo chó mèo gà mèo chó" bằng dict: mỗi từ chỉ chạm đúng một khóa, khóa mới bắt đầu từ 1.

sorted nhận tham số key thay cho hàm so sánh của C++. Ở đây sắp xếp các cặp (tên, điểm) theo điểm giảm dần.

Độ phức tạp vẫn quan trọng như trong lập trình thi đấu. Kiểm tra x in L với list có thể phải duyệt cả list, $O(n)$; với set, tra cứu có chi phí trung bình $O(1)$ khi việc băm hoạt động tốt. Đổi sang set một lần tốn thời gian và bộ nhớ, nhưng có ích khi phải hỏi nhiều lần:

Python

```
L = list(range(10**5))
S = set(L)
queries = range(-500, 500)

r1 = [query in L for query in queries]
r2 = [query in S for query in queries]
print(r1 == r2, sum(r2))
```

KẾT QUẢ

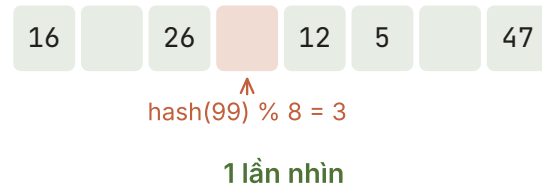
True 500

Hai cách cho cùng kết quả: 500 truy vấn không âm có trong tập. Nhưng với mỗi truy vấn âm, list phải so hết 100 000 phần tử mới biết là không có, còn set tra bảng băm và trả lời gần như ngay. Nếu chỉ hỏi một lần thì không đáng đổi sang set, vì dựng set cũng tốn một lượt duyệt cả list.

99 in list



99 in set



List so từng phần tử; set nhảy thẳng tới một ngăn của bảng băm.

HỎI NHANH

Có 50 000 mã học sinh đã nộp bài. Cần kiểm 10 000 mã khác xem từng mã đã nộp hay chưa. Nên làm thế nào?

- A Đổi danh sách 50 000 mã thành set rồi dùng in
- B Cứ để list và dùng in, 50 000 phần tử chưa nhiều
- C Sắp xếp list trước để in chạy nhanh hơn
- D Viết hai vòng for lồng nhau cho dễ kiểm

Hàm, vòng lặp và tốc độ

Vòng for của Python duyệt thẳng các phần tử, không cần chỉ số. Khi cần cả chỉ số thì dùng enumerate, khi duyệt song song hai dãy thì dùng zip:

Python

```
names = ["An", "Bình", "Chi"]
marks = [8.5, 6.0, 9.0]
for i, (name, mark) in enumerate(zip(names, marks), start=1):
    print(f"{i}. {name}: {mark}")
```

KẾT QUẢ

1. An: 8.5
2. Bình: 6.0

Hàm có tham số mặc định, gọi được bằng tên tham số, và trả về nhiều giá trị (thật ra là một tuple):

Python

```
def stats(xs, ddof=0):
    n = len(xs)
    if not 0 <= ddof < n:
        raise ValueError("cần 0 <= ddof < số phần tử")
    mean = sum(xs) / n
    var = sum((x - mean) ** 2 for x in xs) / (n - ddof)
    return mean, var

m, v = stats(marks)
print(round(m, 3), round(v, 3), round(stats(marks, ddof=1)[1], 3))
```

KẾT QUẢ

7.833 1.722 2.583

Trung bình là 7.833; phương sai chia cho n là 1.722. Nếu chia cho $n - 1$, kết quả là 2.583. `ddof` là số bị trừ khỏi n ở mẫu số; bài 0.7 sẽ nói khi nào dùng cách nào. Dòng `raise` ở đầu hàm báo lỗi khi danh sách rỗng hoặc quá ít phần tử, thay vì để hàm chia cho 0.

Có một cái bẫy với tham số mặc định: giá trị mặc định được tạo **một lần** khi định nghĩa hàm, không phải mỗi lần gọi. Nếu nó là một list, mọi lần gọi dùng chung list đó:

Python

```
def add_score(x, history=[]):          # SAI
    history.append(x)
    return history

print(add_score(1), add_score(2))

def add_score(x, history=None):        # ĐÚNG
    if history is None:
        history = []
    history.append(x)
    return history
```

```
print(add_score(1), add_score(2))
```

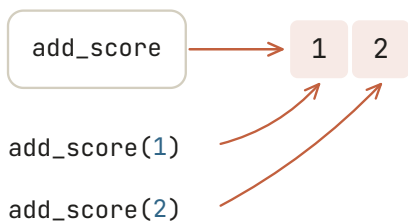
KẾT QUẢ

```
[1, 2] [1, 2]
```

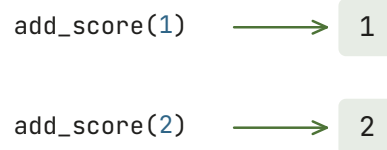
```
[1] [2]
```

Ở bản sai, hai lần gọi trả về **cùng một** list, nên khi print chạy thì cả hai đều đã chứa [1, 2]. Lần gọi thứ hai không bắt đầu từ list rỗng như ta tưởng. Bản đúng dùng None làm giá trị mặc định và tạo list mới bên trong hàm.

history=[]



history=None



List mặc định tạo một lần và dùng chung; None cho mỗi lần gọi một list mới.

HỎI NHANH

Hàm `add_score(score, history=[])` có lỗi: gọi lần sau vẫn thấy điểm của các lần trước trong `history`. Sửa thế nào để mỗi lần gọi mà không truyền `history` thì bắt đầu từ danh sách rỗng?

- A Giữ `history=[]`, gọi `history.clear()` ở cuối hàm
- B Giữ `history=[]`, vì mỗi lần gọi Python tạo lại list
- C Đặt `history=None`, rồi tạo list mới trong hàm
- D Dùng một list toàn cục, gán lại `[]` trước mỗi lần gọi

Giờ đến tốc độ. Vòng lặp Python chậm hơn C++ nhiều, vì ở mỗi bước Python phải đọc lại câu lệnh và xử lý từng số như một đối tượng. Ta so ba cách tính tổng $0 + 1 + \dots + (n - 1)$ với $n = 500\,000$: vòng for tự viết, hàm `sum` có sẵn, và NumPy. Trước khi đo, kiểm cả ba

ra đúng đáp án bằng công thức $n(n - 1)/2$, giống dùng code trâu để kiểm trong lập trình thi đấu:

Python

```
import numpy as np

def loop_sum(n):
    s = 0
    for i in range(n):
        s += i
    return s

n = 500_000
expected = n * (n - 1) // 2
print(loop_sum(n), expected)
print(loop_sum(n) == sum(range(n)) == int(np.arange(n, dtype=np.int64).sum()))
```

KẾT QUẢ

```
124999750000 124999750000
True
```

Đáp án khớp rồi mới đo. `timeit.repeat` chạy mỗi cách năm lần, mỗi lần gộp ba lượt (`number=3`) rồi chia cho 3; lấy lần nhanh nhất để bớt nhiễu do các tiến trình khác trên máy.

Python

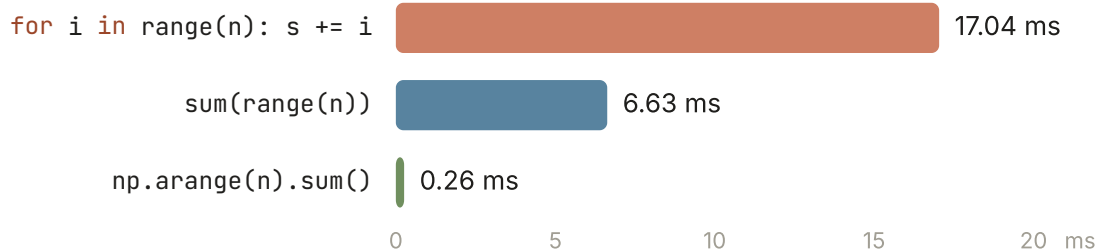
```
from timeit import repeat

jobs = [
    ("for", lambda: loop_sum(n)),
    ("sum", lambda: sum(range(n))),
    ("numpy", lambda: int(np.arange(n, dtype=np.int64).sum())),
]

timings = {}
for label, job in jobs:
    assert job() == expected
    timings[label] = min(repeat(job, repeat=5, number=3)) / 3
print("nhanh -> chậm:", sorted(timings, key=timings.get))
```

KẾT QUẢ

```
nhanh -> chậm: ['numpy', 'sum', 'for']
```



Cùng một tổng của 500 000 số, đo trên một máy CPU.

NumPy nhanh nhất dù phải tạo mảng, rồi đến `sum`, vòng `for` tự viết chậm nhất. `sum` và NumPy nhanh hơn vì vòng lặp của chúng chạy bên trong code C đã biên dịch. Số giây trên máy đang chạy nằm trong biến `timings`; hình trên vẽ từ một lần đo như vậy. Dù vậy, công thức $n(n-1)/2$ vẫn nhanh hơn cả ba vì không cần vòng lặp: thuật toán tốt vẫn quan trọng hơn thư viện nhanh. Bài 0.2 sẽ học cách tính trên mảng bằng NumPy.

HỎI NHANH

Hai cách cài đều tính đúng cùng một tổng. Cách A chạy 8 ms nhưng phải mất thêm 5 ms chuẩn bị dữ liệu. Cách B chạy 11 ms và dùng luôn dữ liệu có sẵn. Nếu chỉ chạy một lần, cách nào nhanh hơn?

- A B nhanh hơn: 11 ms so với 13 ms
- B Như nhau, vì cùng độ phức tạp $O(n)$
- C Chưa so được, vì mỗi lần đo một khác
- D A nhanh hơn: 8 ms so với 11 ms

Chuỗi và file

Chuỗi `str` là bất biến và chứa Unicode, nên tiếng Việt có dấu không cần xử lý gì đặc biệt. Cách định dạng nên dùng là f-string: biểu thức trong `{}`, định dạng sau dấu `:`.

```

name, acc = "logistic", 0.85432
print(f"{name:>10} | accuracy = {acc:.3f} | {acc:.1%}")
line = " 12, 7.5 ,An \n"
print(line.strip().split(","), [s.strip() for s in line.split(",")])
print(" - ".join(["a", "b", "c"]))

```

KẾT QUẢ

```

logistic | accuracy = 0.854 | 85.4%
['12', ' 7.5 ', 'An'] ['12', '7.5', 'An']
a - b - c

```

strip bỏ khoảng trắng hai đầu, split cắt theo dấu phân cách, join ghép ngược lại. Đọc dữ liệu thô từ file thường chỉ cần bộ ba này.

Đọc và ghi file dùng open trong khối with, khối này đóng file kể cả khi có lỗi. Ghi rõ encoding="utf-8" để không phụ thuộc encoding mặc định của máy. Module csv hiểu dấu phẩy nằm trong ô được đặt ngoặc kép; split(",") đơn thuần không đủ cho mọi CSV.

Ví dụ sau tạo thư mục tạm **bên trong thư mục làm việc**, ghi file rồi đọc lại và tự dọn thư mục tạm khi kết thúc. Khi làm đề thật, dùng đúng đường dẫn bài yêu cầu và giữ file cần nộp:

Python

```

import csv
from pathlib import Path
from tempfile import TemporaryDirectory

preds = {1: 0.91, 2: 0.15, 3: 0.67}
with TemporaryDirectory(dir=".") as directory:
    path = Path(directory) / "submission.csv"
    with path.open("w", newline="", encoding="utf-8") as file:
        writer = csv.writer(file)
        writer.writerow(["id", "prediction"])
        for student_id, prediction in preds.items():
            writer.writerow([student_id, prediction])
    with path.open(encoding="utf-8", newline="") as file:
        rows = list(csv.reader(file))
print(rows)
print(type(rows[1][1]), float(rows[1][1]) + 1)

```

KẾT QUẢ

```
[[ 'id', 'prediction'], [ '1', '0.91'], [ '2', '0.15'], [ '3', '0.67']]  
<class 'str'> 1.9100000000000001
```

Mọi thứ đọc từ CSV đều là chuỗi, kể cả "0.91". Phải đổi sang số trước khi tính hay sắp xếp: sắp xếp chuỗi thì "10" đứng trước "9". Bài 0.3 sẽ dùng pandas để đọc CSV, tự đoán kiểu cho từng cột. Module csv vẫn đáng biết khi file quá lạ để pandas đoán đúng.

HỎI NHANH

Đọc cột điểm bằng module csv, ta được ["9", "10", "2"]. sorted trả về gì?

A ['2', '9', '10']

B ['10', '2', '9']

C ['9', '10', '2']

D TypeError, vì không sắp xếp được chuỗi

Đọc lỗi

Khi code lỗi, Python in một traceback: danh sách các lời gọi hàm dẫn đến chỗ lỗi, dòng cuối cùng là loại lỗi và thông báo. Đọc từ **dưới lên**: dòng cuối cho biết chuyện gì xảy ra, các dòng ngay trên cho biết ở đâu.

```
Traceback (most recent call last):
  File ".../diem.py", line 7, in <module>
    print(mean_of("Dũng"))
    ~~~~~^~~~~~
  File ".../diem.py", line 4, in mean_of
    marks = scores[name]
    ~~~~~^~~~~~
KeyError: 'Dũng'
```



Traceback thật của Python 3.13: đọc từ dòng cuối lên.

Bốn loại lỗi gặp nhiều nhất trong khóa học:

Python

```
tests = [lambda: "Điểm: " + 9, lambda: scores["Dũng"], lambda: marks[3], lambda: int("7.5")]
for f in tests:
    try:
        f()
    except Exception as e:
        print(type(e).__name__ + ":", e)
```

KẾT QUẢ

```
TypeError: can only concatenate str (not "int") to str
KeyError: 'Dũng'
IndexError: list index out of range
ValueError: invalid literal for int() with base 10: '7.5'
```

- **TypeError**: dùng sai kiểu. Cộng chuỗi với số phải viết `f"Điểm: {9}"` hoặc `"Điểm: " + str(9)`.
- **KeyError**: tra khóa không có trong dict, hoặc tên cột không có trong bảng pandas.
- **IndexError**: chỉ số vượt quá độ dài list.
- **ValueError**: đúng kiểu nhưng giá trị không đổi được, như `int("7.5")`. Điểm có phần thập phân thì đọc bằng `float("7.5")`.

Ví dụ trên dùng `try/except` để in cả bốn lỗi mà chương trình không dừng. Khi tự sửa code thì không nên bắt lỗi như vậy, để traceback chỉ ra đúng dòng hỏng.

Khi kết quả sai mà không có lỗi, cách nhanh nhất là in ra kiểu, độ dài, vài phần tử đầu ở từng bước, hoặc đặt `assert` cho giả định cần kiểm. `assert len(preds) == 3, len(preds)` dừng chương trình ngay khi giả định sai và in ra giá trị thật. Các bài sau dùng cách này rất nhiều với shape của mảng.

HỎI NHANH

Quen đếm từ 1 như nhiều bài C++, một bạn viết `for i in range(1, len(marks) + 1): print(marks[i])`, với `marks` có 3 phần tử. Vòng lặp chạy ra sao?

- A In đủ ba phần tử, như vòng 1 đến n bên C++
- B Báo `IndexError` ngay ở vòng đầu
- C In `marks[1]`, `marks[2]` rồi dừng bình thường
- D In `marks[1]`, `marks[2]`, rồi `IndexError` ở `i = 3`

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 137

Class

Từ chương 3, model được viết thành `class Net(nn.Module)` và dữ liệu đi qua một `Dataset`. Class gom dữ liệu cùng các hàm dùng dữ liệu ấy. `__init__` chạy khi tạo đối tượng; `self` là chính đối tượng đó, giống `this` của C++ nhưng phải ghi ra.

Python

```
class ScoreBook:
    def __init__(self, names, marks):
        assert len(names) == len(marks), "mỗi tên một điểm"
        self.names = list(names)
        self.marks = list(marks)

    def __len__(self):
        return len(self.marks)
```

```

def __getitem__(self, i):
    return self.names[i], self.marks[i]

def mean(self):
    return sum(self.marks) / len(self)

book = ScoreBook(["An", "Bình", "Chi"], [8.5, 6.0, 9.0])
print(len(book), book[1], round(book.mean(), 3))
for name, mark in book:
    print(name, mark)

```

KẾT QUẢ

```

3 ('Bình', 6.0) 7.833
An 8.5
Bình 6.0
Chi 9.0

```

`len(book)` gọi `book.__len__()`, `book[1]` gọi `book.__getitem__(1)`. Vòng `for` gọi `__getitem__` với 0, 1, 2... tới khi gặp `IndexError`. Dataset ở bài 3.4 cũng chỉ cần hai hàm này.

Class con ghi tên class cha trong ngoặc; `super().__init__(...)` chạy phần khởi tạo của cha. `__call__` cho phép gọi đối tượng như một hàm:

Python

```

class Scaler:
    def __init__(self, factor):
        self.factor = factor

    def __call__(self, values):
        return self.forward(values)

    def forward(self, values):
        return [v * self.factor for v in values]

class ShiftScaler(Scaler):
    def __init__(self, factor, shift):
        super().__init__(factor)
        self.shift = shift

    def forward(self, values):
        return [v + self.shift for v in super().forward(values)]

```

```
f = ShiftScaler(2, 1)
print(f([1, 2, 3]), f.factor, isinstance(f, Scaler))
```

KẾT QUẢ

[3, 5, 7] 2 True

`f([1, 2, 3])` gọi `Scaler.__call__`, hàm này gọi `self.forward`; `f` là `ShiftScaler` nên `forward` của class con chạy: nhân 2 rồi cộng 1. `nn.Module` làm giống vậy, nên ta viết `forward` nhưng gọi `model(x)`.

viết

Python gọi

`len(ds)`



`ds.__len__()`

Dataset

`ds[i]`



`ds.__getitem__(i)`

Dataset

`model(x)`



`model.__call__(x)` → `forward(x)`

nn.Module

Ba cú pháp và hàm đặc biệt mà chúng gọi.

HỎI NHANH

Xóa dòng `super().__init__(factor)` trong `ShiftScaler.__init__`, rồi gọi `ShiftScaler(2, 1)([1, 2, 3])`. Điều gì xảy ra?

- A Vẫn ra [3, 5, 7], vì Python tự gọi `__init__` của class cha
- B Ra [2, 3, 4], vì thiếu `factor` thì Python coi là 1
- C Báo `AttributeError`, vì đối tượng chưa có thuộc tính `factor`
- D Báo lỗi ngay ở dòng tạo đối tượng, trước khi gọi

Khi đi thi

- *Restart & Run all* trước khi nộp.

- Tính trên nhiều số thì dùng `sum` hoặc `NumPy`, không tự viết vòng `for`.
- Ghi file với `encoding="utf-8"`, rồi đọc lại file nập: đủ dòng, đúng tên cột, cột số không bị đọc thành chuỗi.

Tóm tắt

- Vòng lặp Python chậm; tính trên nhiều số thì dùng `sum`, `NumPy`, và đo trước khi tối ưu.
- `int` có độ chính xác tùy ý. `//` làm tròn xuống, `%` đi theo nó, khác C++ với số âm. So hai số thực bằng `math.isclose`, không dùng `==`.
- Biến là tên dán lên đối tượng; phép gán không sao chép. Kiểu sửa được (list, dict, set, mảng `NumPy`) có thể bị sửa qua tên khác. Sao chép bằng `copy()`, sâu hơn thì `deepcopy`.
- list, tuple, dict, set thay cho STL; chọn theo độ phức tạp, đặc biệt là in trên set thay cho list.
- Ghi file với `encoding="utf-8"`; dữ liệu đọc từ CSV là chuỗi. Đọc traceback từ dưới lên.

Kiểm tra cuối bài

Mười câu dưới đây bao quát cả bài. Làm hết rồi xem điểm; câu nào sai thì đọc lại mục tương ứng.

CÂU 1

Sau `a = [1, 2]`, `b = [1, 2]`, `c = a`, lệnh `print(a == b, a is b, a is c)` in ra gì?

A True True True

B True False True

C False False True

D True False False

PHÒNG THEO NOAI 2026 Singapore · Vòng sơ loại · Câu 95

CÂU 2

Có n phần tử xếp thành vòng tròn, đánh số từ 0 đến $n - 1$. Biểu thức nào cho số thứ tự của phần tử đứng ngay trước i , kể cả khi $i = 0$?

A $(i + 1) \% n$

B $\text{abs}(i - 1) \% n$

C $(i - 1) // n$

D $(i - 1) \% n$

CÂU 3

`0.1 * 3 == 0.3` cho kết quả gì? Muốn so hai số thực như vậy thì nên viết thế nào?

A `False`; dùng `math.isclose(0.1 * 3, 0.3)`

B `True`; dùng `==` là được

C Báo lỗi, vì không so được số thực

D `False`; đổi cả hai sang `int` rồi so

CÂU 4

Sau bốn lệnh `a = [[1, 2], [3, 4]]`, `b = a.copy()`, `b[0] = [9, 9]`, `b[1].append(5)`, `a` là gì?

A `[[1,2],[3,4]]`

B `[[1,2],[3,4,5]]`

C `[[9,9],[3,4]]`

D `[[9,9],[3,4,5]]`

CÂU 5

Chạy `grid = [[0] * 3] * 2` rồi `grid[1][2] = 7`. Lúc này `grid` là gì, và phải sửa dòng tạo `grid` thế nào để hai hàng không dính nhau?

A `[[0,0,0],[0,0,7]]`; thêm `grid = grid.copy()` sau khi tạo

B `[[0,0,0],[0,0,7]]`; không cần sửa, hai hàng đã riêng

C `[[0,0,7],[0,0,7]]`; dùng `grid=[[0]*3 for _ in range(2)]`

D `[[0,0,7],[0,0,7]]`; dùng `grid=([0]*3).copy()*2`

CÂU 6

Hàm `def f(x, acc=[]): acc.append(x); return len(acc)` được gọi ba lần liên: `f(1), f(2), f(3)`. Lần thứ ba trả về bao nhiêu?

A 1

B 2

C 3

D Báo lỗi

CÂU 7

Có list từ `words = ["mèo", "chó", "mèo"]`. Cần gán cho mỗi từ khác nhau một số 0, 1, 2,... để sau này đổi từ thành số. Yêu cầu: chạy lại lần nào cũng ra đúng một bảng, kể cả khi các từ trong `words` đổi thứ tự. Viết thế nào?

A `vocab = {w: i for i, w in enumerate(sorted(set(words)))}`

B `vocab = {w: i for i, w in enumerate(list(set(words)))}`

C `vocab = {w: i for i, w in enumerate(sorted(words))}`

D `vocab = {w: words.index(w) for w in words}`

PHÒNG THEO [USA-NA-AIO 2026 · Vòng 1 · Part 5.1–5.2](#)

CÂU 8

Quen C++, một bạn muốn cộng từng phần tử của hai dãy nên viết `[1, 2] + [3, 4]`. Python cho kết quả gì?

A `[4, 6]`

B `[[1, 2], [3, 4]]`

C Báo lỗi

D `[1, 2, 3, 4]`

ĐỀ THI NOAI 2026 Singapore · Vòng sơ loại · Câu 91

CÂU 9

Đọc file CSV bằng module `csv`, ô `row[1]` hiện là `8.5`. Biểu thức `row[1] + 1` cho kết quả gì?

A `9.5`

B `"8.51"`

C `TypeError`

D `ValueError`

CÂU 10

Trong notebook, một ô phía trên lỡ đặt biến `max = 10`. Từ đó tên `max` chỉ số 10 chứ không còn chỉ hàm `max` có sẵn, nên `max([4, 9])` báo `TypeError: 'int' object is not callable`. Làm sao để dùng lại được hàm `max` mà vẫn giữ số 10 dưới tên khác?

A Đặt `best = 10`; `max = best`, rồi gọi `max([4, 9])`

B Đổi lời gọi thành `max((4, 9))` cho khớp kiểu

C Đặt `max = None`, rồi gọi lại `max([4, 9])`

D Đặt `best = max`, `del max`, rồi gọi `max([4, 9])`

Đọc thêm

- Python Software Foundation — [The Python Tutorial](#). Tài liệu chính thức, đọc nhanh các chương 3–5 và 7 để có toàn bộ cú pháp cơ bản.
- Ned Batchelder — [Facts and myths about Python names and values](#). Bài viết kinh điển về mô hình “tên và đối tượng” của mục thứ ba, có hình minh họa.
- Python Software Foundation — [Floating Point Arithmetic: Issues and Limitations](#). Vì sao $0.1 + 0.2 \neq 0.3$ và vì sao `round(2.675, 2)` cho 2.67.
- Python Wiki — [TimeComplexity](#). Bảng độ phức tạp của mọi thao tác trên list, dict, set; dùng như bảng tra STL.
- Downey — [Think Python, 3rd edition](#). Sách miễn phí, viết bằng notebook; hợp để đọc lại những phần còn lấn cấn.
- Python Software Foundation — [Programming FAQ](#), các mục về tham số mặc định và cách sao chép đối tượng.
- Python Software Foundation — [copy](#) và [Hashable](#). Đọc để phân biệt sao chép nông, sao chép sâu và điều kiện làm khóa của dict.
- Python Software Foundation — [timeit](#) và [csv](#). Đọc cách đo các đoạn code ngắn và đọc CSV có dấu phân cách nằm trong ô.