上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Introduction to Computation (CS1602) · Lecture 5

Lists

Instructor: Tao Huang

Part II: Built-in Data Structures · Fall 2026

PART II

Built-in Data Structures

By the end of this part you can pick the right container for a problem, and handle a whole batch of data at once.

Where we left off

- A variable is **a name pointing at a value**, not a box. `id()` gives the address, `is` asks whether two names point at the same thing
- A function wraps logic. **`print` is for people, `return` is for the program**
- `for` takes items one at a time; `while` runs until a condition holds
- Since L4, every function carries type hints: `def bmi(w: float, h: float) -> float:`

第一条是今天的关键。 今天这一讲，一大半内容是它的直接后果。

Something from daily life first

You and your roommate share one shopping list.

They add soy sauce on their phone. You do nothing at all —

yet when you open yours, soy sauce is on it too.

两个人打开的是同一份，不是各自的副本

你们看的是同一份清单，各自手里的只是一个入口。

Now look at two snippets

A

```
x = 1
x2 = x
x2 = x2 + 1

print(x)
```

B

```
nums = [1]
nums2 = nums
nums2.append(1)

print(nums)
```

两段代码结构完全相同。**打印出来会一样吗?**

随堂小测 · 第 1 轮

扫码作答 —— 这一轮一题

1 B 打印出什么?

这一题现在不公布答案。讲到别名与拷贝那一节再回来。

https://taohuang.info/cs1602/zh/quiz/5

今天

Why did b change when I only touched a?

A shared shopping list, in code.

AI policy — Level 0: No AI-generated code

- Weeks 1–7
  - Do not let an AI write or autocomplete code for you. Turn off AI completion in your editor.
  - You may ask an AI to explain a concept, look something up in the docs, or make sense of an error message.

PART 1 OF 9

Why you need a container

Five scores, five variables — it works

```
score1 = 87
score2 = 92
score3 = 65
score4 = 78
score5 = 90

total = score1 + score2 + score3 + score4 + score5
print("average:", total / 5)
```

Five variables, five names.

Then the requirements change

- Find the highest score
  - if score1 > score2 and score1 > score3 and ... — 100 comparisons
- Drop the lowest one and re-average
  - which variable do you delete?
- Read the scores from a file, count unknown until you read it
  - you cannot even name the variables in advance

更麻烦的是：**写代码的时候，你并不知道会有多少个。**

The list 列表

One name, any number of values.

```
scores = [87, 92, 65, 78, 90]

print("count:  ", len(scores))
print("avg:  ", sum(scores) / len(scores))
print("highest:", max(scores))
```

One name for the whole batch, however many there are.

Data structure 数据结构

- A data structure = how data is represented + what you can do to it
  - representation — a list keeps its items in order, numbered from 0
  - operations — add, remove, look up, slice, sort
- This course covers four: list, tuple, dict, set
  - today is the first, and the one the other three are compared against

数据结构 = 怎么存 + 能对它做什么。

PART 2 OF 9

Building a list, and reading it back

Square brackets and commas make a list

```
1  list1 = ["Hello", "world"]          # two strings
2  list2 = [1, 3, 9, 7]                # four integers
3  list3 = [1.0, 2.0, 3.0, 1e-3]       # four floats
4  mixed = [1, "two", 3.0, True]       # types may be mixed
5  nested = ["I", "am", [1, 2, 3]]     # a list inside a list
```

方括号，逗号分隔。写成 {} 或者用分号都不行。

Whatever is inside, the type is list

type and len

```
1  list1 = ["Hello", "world"]
2  list3 = [1.0, 2.0, 3.0, 1e-3]
3  nested = ["I", "am", [1, 2, 3]]
4
5  print(type(list1), len(list1))
6  print(type(list3), len(list3))
7  print(type(nested), len(nested))
```

output

```
<class 'list'> 2
<class 'list'> 4
<class 'list'> 3
```

`nested` has length **3**. The inner list counts as one item.

The list() constructor 构造函数

list()

```
1  print(list("Python"))     # split up
2  print(list(range(5)))     # a range
3  print(list(range(2, 11, 3)))
4  print(list())              # empty
```

output

```
['P', 'y', 't', 'h', 'o', 'n']
[0, 1, 2, 3, 4]
[2, 5, 8]
[]
```

Anything you can take items out of one by one can go through `list()`. Such a thing is called an **iterable** 可迭代对象.

Items are numbered from 0

A list is ordered. Each item is found by its index.

```
a = ["red", "green", "blue", "yellow"]

print(a[0])      # red      — the first
print(a[1])      # green
print(a[3])      # yellow   — the last, len(a) - 1
```

下标从 **0** 起。最后一项是 `len(a) - 1`，写 `len(a)` 就越界。

A negative index counts back from the end

negative index

```
1  a = ["red", "green", "blue", "yellow"]
2
3  print(a[-1])      # the last item
4  print(a[-2])      # the one before it
5  print(a[-len(a)])  # same as a[0]
```

output

```
yellow
blue
red
```

`a[-1]` is the last item; negative indices count back from the end.

Out of range raises IndexError

what you will see, a lot

```
1  a = ["red", "green", "blue", "yellow"]
2  print(a[4])
```

traceback

```
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    print(a[4])
          ~^^^
IndexError: list index out of range
```

IndexError 是最常见的错误之一。 遇到它，先确认列表的长度。

in and not in ask whether it is there

membership

```
1  fruit = ["apple", "pear", "banana"]
2
3  print("apple" in fruit)
4  print("durian" in fruit)
5  print("durian" not in fruit)
6  print([1, 2] in [1, 2, 3])   # an ITEM?
```

output

```
True
False
True
False
```

最后一行是 `False`。**`in` 比较的是元素，不是片段。**

PART 3 OF 9

Walking through a list

for x in a gives you the item itself

for-in

```
1  fruit = ["apple", "orange", "pear"]
2
3  for x in fruit:
4      print(x, "has", len(x), "letters")
```

output

```
apple has 5 letters
orange has 6 letters
pear has 4 letters
```

Lists, strings, tuples and dicts can all be walked this way. Collectively: **iterables**.

One loop, two ways to write it

INDEX BY HAND

```
i = 0
while i < len(fruit):
    print(fruit[i])
    i += 1
```

PYTHONIC

```
for x in fruit:
    print(x)
```

Both do the same job. The right one removes three chances to get it wrong: the start, the end, and `i += 1`.

Need the index too? Use enumerate

enumerate

```
1  fruit = ["apple", "orange", "pear"]
2
3  for i, x in enumerate(fruit):
4      print(i, x)
```

output

```
0 apple
1 orange
2 pear
```

When you need the index, reach for `enumerate` — **not `for i in range(len(a))`**.

PART 4 OF 9 · THE KEY SECTION

So what is a list, really?

An array 数组: same type, laid out end to end

- Every item is the same type, so every item takes the same number of bytes
- The items sit in consecutive memory, one after another
- So the address of item i is: start + size x i
  - one multiplication and one addition — the computer jumps straight there

下标从 0 起的理由在这里：**第 0 项的地址就是起始地址。**

But a Python list holds anything at all

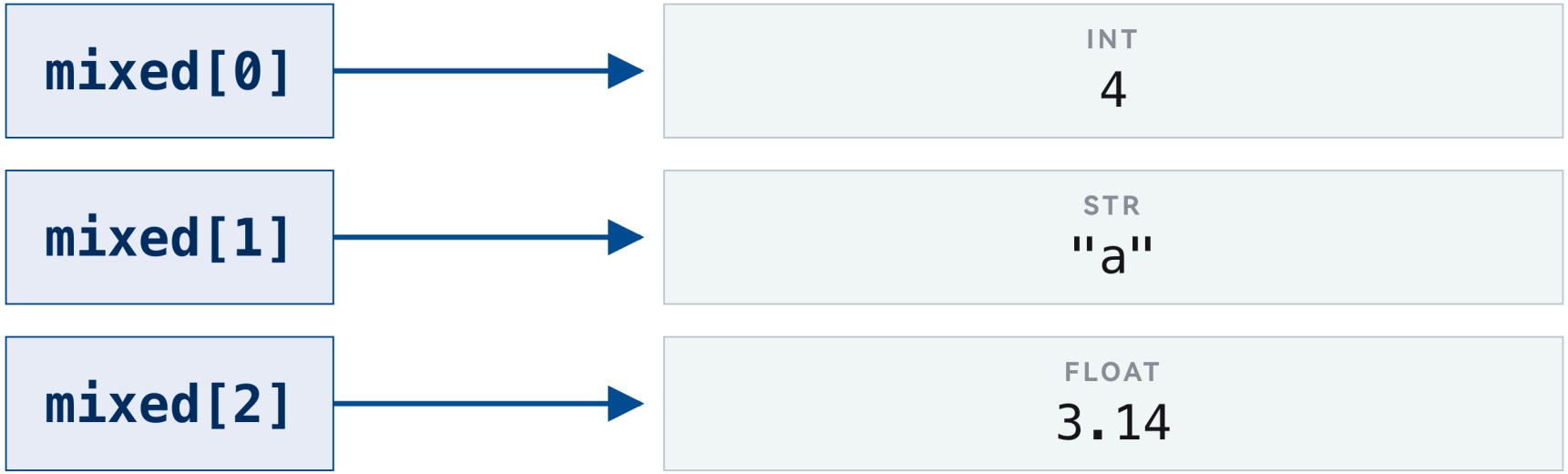
These have completely different sizes. The array trick cannot work.

```
mixed = [4, "a", 3.14, True, [1, 2], None]

for x in mixed:
    print(type(x).__name__, end="  ")
```

Different sizes means the address cannot be computed. So how does Python manage it?

A list stores addresses, not data



the slots are all the same size — they hold addresses, not values

An address is a fixed size. So the array holding the addresses is uniform after all.

Which makes this legal, if strange

Each slot holds an address; that address may point at another list.

```
deep = [[[[[[1]]]]]]

print(deep[0][0][0][0][0][0])
print(len(deep))
```

An address can point at a list of addresses, and so on down.

id() gives you that address

id()

```
1  a = [1, 2, 3]
2  print("id of a: ", id(a))
3  print("id of a[0]:", id(a[0]))
4
5  b = a
6  print("id of b: ", id(b))
7  print("a is b:  ", a is b)
```

output

```
id of a:  4372337216
id of a[0]: 4411933176
id of b:  4372337216
a is b:   True
```

`id(x)` is where `x` lives. **The number differs every run; which ones match does not.**

PART 5 OF 9

Changing a list

A list is mutable: you change it in place

mutable

```
1  a = [1, 2, 3]
2  print("before:", a, id(a))
3
4  a[0] = 99
5  print("after: ", a, id(a))
```

output

```
before: [1, 2, 3] 4353888832
after:  [99, 2, 3] 4353888832
```

内容变了，地址没变。**这就是「修改」。**

append adds one item at the end

append

```
1  a = [1, 2, 3]
2  a.append(4)
3  print(a)
4
5  a.append([5, 6])    # ONE item, a list
6  print(a)
7  print(len(a))
```

output

```
[1, 2, 3, 4]
[1, 2, 3, 4, [5, 6]]
5
```

append 一次只加一项。传进去的是列表，那也只算一项。

append, extend and + are three different things

append — ONE item

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

output

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

extend — EACH item

```
a = [1, 2]
a.extend([3, 4])
print(a)
print(len(a))
```

output

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

append 加一项，extend 逐项加，a + b 也逐项，**但另造一个新列表。**

a + b builds a new list; extend does not

in place or not

```
1  a = [1, 2]
2  print("a starts at", id(a))
3
4  a.extend([3, 4])
5  print("extend", id(a))   # same list
6
7  a = a + [5, 6]
8  print("plus  ", id(a))   # a new one
```

output

```
a starts at 4364309056
extend 4364309056
plus   4364629120
```

Changed in place, or newly built? Today keeps coming back to that.

Every in-place method returns None

what do they return?

```
1  a = [3, 1, 2]
2
3  print(a.append(4))
4  print(a.sort())
5  print(a.reverse())
6  print(a.insert(0, 9))
7  print(a)            # but a DID change
```

output

```
None
None
None
None
[9, 4, 3, 2, 1]
```

全是 None。 改的就是原列表，没有新对象可以返回。

Which makes a = a.append(x) a bug

错

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

# a is None; list lost
```

对

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

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

append added the 4, then returned None — and that None was bound back to a.

`sort`, `reverse`, `remove`, `insert`: all the same.

In-place methods return `None`. The ones that return a new object leave the original alone.

The in-place family

- append(x) · insert(i, x) · extend(it) · remove(x) · sort() · reverse() · clear()
  - all change the list itself, all return None
- pop(i) is the exception — it removes AND hands the item back
  - so x = a.pop() is correct, and useful
- sorted(a) · reversed(a) · a + b · a[:] build something new instead

写成 `a = a.append(x)`，`a` 就变成了 `None`，**原来的列表反而丢了**。

PART 6 OF 9

Slicing 切片

a[i:j] takes i up to j, but not j

slice

```
1  a = [10, 20, 30, 40, 50, 60]
2
3  print(a[1:4])
4  print(a[0:3])
5  print(a[:3])       # 0 can be omitted
6  print(a[2:])      # end omitted
7  print(a[:])        # both — the whole list
```

output

```
[20, 30, 40]
[10, 20, 30]
[10, 20, 30]
[30, 40, 50, 60]
[10, 20, 30, 40, 50, 60]
```

左闭右开。 `a[1:4]` 取的是第 1、2、3 项，不含第 4 项，长度是 4−1。

a[start:stop:step] adds a stride

step

```
a = [10, 20, 30, 40, 50, 60]

print(a[::2])       # every other one
print(a[1::2])
print(a[::-1])     # backwards
```

output

```
[10, 30, 50]
[20, 40, 60]
[60, 50, 40, 30, 20, 10]
```

`a[::-1]` gives a reversed copy. Worth committing to memory.

Drag the three cursors; watch the cells light up

a[1:4]

0	1	2	3	4	5
10	20	30	40	50	60
−6	−5	−4	−3	−2	−1

→ [20, 30, 40]

start 省略 — 1

stop 省略 — 4

step 省略 — –

The `stop` cell stays dark. **That is what half-open means.**

A slice never complains about going too far

no IndexError

```
1  a = [10, 20, 30]
2
3  print(a[1])          # fine
4  print(a[1:99])       # no error
5  print(a[99:])        # empty, no error
6  print(a[5:2])        # also empty
```

output

```
20
[20, 30]
[]
[]
```

切片不会因为越界而报错，它会安静地返回一个较短的列表或空列表。

You can assign to a slice

slice assignment

```
1  a = [1, 2, 3, 4, 5]
2  a[1:3] = [20, 30]          # same length
3  print(a)
4  a[1:3] = [7, 7, 7, 7]     # length may differ
5  print(a)
6  a[1:5] = []               # deletes
7  print(a)
```

output

```
[1, 20, 30, 4, 5]
[1, 7, 7, 7, 7, 4, 5]
[1, 4, 5]
```

The right side of a slice assignment must be iterable — `a[1:3] = 20` raises.

A slice always builds a new list

is vs ==

```
1  a = [1, 2, 3]
2  b = a           # the same list
3  c = a[:]        # a NEW list, same items
4
5  print("a is b:", a is b)
6  print("a is c:", a is c)
7  print("a == c:", a == c)
```

output

```
a is b: True
a is c: False
a == c: True
```

== 比较内容，is 比较是否为同一个对象。**这两者不能混用。**

And slicing is fast

copying 2,000,000 items

```
from time import perf_counter as now
f = list(range(2_000_000))

t = now(); g = [x for x in f]; loop = now() - t
t = now(); h = f[:];           sl = now() - t

print(f"loop {loop*1000:5.1f} ms")
print(f"a[:] {sl*1000:5.1f} ms")
```

output

```
loop   8.6 ms
a[:]   2.7 ms
```

To copy a whole list, `a[:]` is about three times faster than a loop you write yourself.

PART 7 OF 9 · TODAY IS RIGHT HERE

Two names, one list

Back to the opening question

x2 = x2 + 1 左边没变，nums2.append(1) 右边变了。

The same thing as that shared shopping list.

b = a copies the arrow, not the list

aliasing

```
1  a = [1, 2, 3]
2  b = a           # the SAME list
3
4  b.append(4)
5
6  print("a =", a)
7  print("b =", b)
8  print("same object?", a is b)
```

output

```
a = [1, 2, 3, 4]
b = [1, 2, 3, 4]
same object? True
```

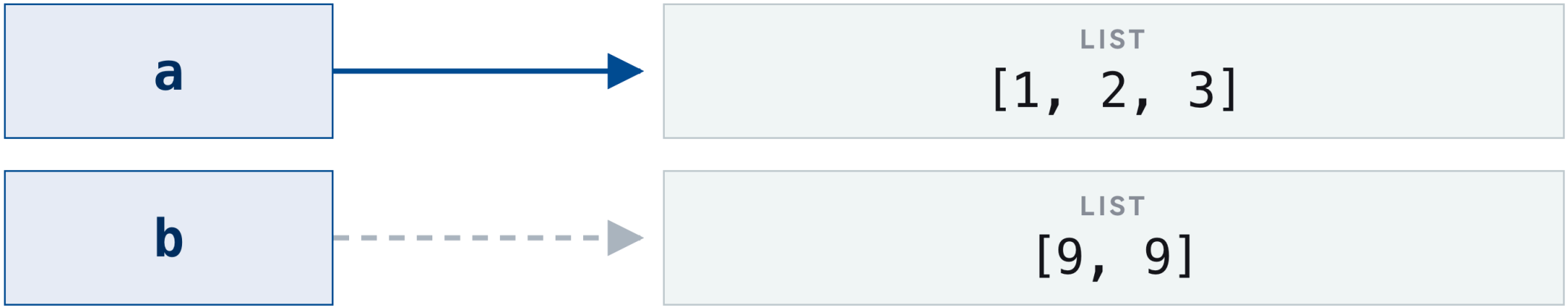
b = a 复制的是箭头。 a 和 b 是同一个列表的两个名字。

One list, two names

b = a made a second arrow to the same object

The reason it works this way: lists can be huge, and copying one on every assignment would cost far too much.

Mutating, or rebinding



b = [9, 9] pointed b at a NEW object; a never moved

`b.append(4)` 修改对象，`b = [9, 9]` 重新绑定。**前者影响 a，后者不会。**

Getting a copy of your own

copying

```
1  a = [1, 2, 3]
2  b = a[:]        # slice
3  c = a.copy()    # clearest
4  d = list(a)     # constructor
5  a.append(99)
6
7  print("a =", a)
8  print("b =", b)
9  print("c =", c)
```

output

```
a = [1, 2, 3, 99]
b = [1, 2, 3]
c = [1, 2, 3]
```

Three ways to get a copy of your own: `a[:]`, `a.copy()`, `list(a)`.

But a copy goes only one level deep

shallow copy

```
1  a = [[1, 2], [3, 4]]
2  b = a.copy()
3
4  b[0][0] = 99       # reaching INSIDE
5
6  print("a =", a)
7  print("b =", b)
8  print("a[0] is b[0]:", a[0] is b[0])
```

output

```
a = [[99, 2], [3, 4]]
b = [[99, 2], [3, 4]]
a[0] is b[0]: True
```

The outer level was copied; the inner level was not. **This is a shallow copy** 浅拷贝.

Deep copy 深拷贝

- copy() duplicates one level. The nested objects are still shared.
- copy.deepcopy() follows the whole chain of addresses and duplicates all of it.
  - import copy → d = copy.deepcopy(a)
- 浅拷贝复制的是外层那一排格子，格子里的地址原样搬过去，所以内层对象仍然是共用的。
- 深拷贝沿着每一个地址往下走，把每一层都复制一份，得到的结构和原来完全独立。

copy duplicates one level; deepcopy follows the chain of addresses all the way down.

deepcopy, checked

copy vs deepcopy

```python
import copy
a = [[1, 2], [3, 4]]
shallow = a.copy()
deep = copy.deepcopy(a)

a[0][0] = 99
print("a       =", a)
print("shallow =", shallow)   # followed
print("deep    =", deep)      # separate
```

output

```
a       = [[99, 2], [3, 4]]
shallow = [[99, 2], [3, 4]]
deep    = [[1, 2], [3, 4]]
```

嵌套结构要备份，**只有 deepcopy 是可靠的。**

The classic accident: a grid built with *

[[0]*3]*3

```python
grid = [[0] * 3] * 3      # looks fine
grid[0][0] = 1
print(grid)

good = [[0] * 3 for _ in range(3)]
good[0][0] = 1
print(good)
```

output

```
[[1, 0, 0], [1, 0, 0], [1, 0, 0]]
[[1, 0, 0], [0, 0, 0], [0, 0, 0]]
```

* 3 复制的是三个箭头，都指向同一行。**改一行等于改三行。**

Three minutes with the person next to you

一个函数接收一个列表作为参数。你想在函数内部排序，但不希望调用方的那个列表被改动。应该怎么写?

Two answers are correct here. Find both.

PART 8 OF 9

Deleting while you iterate

This looks right. It is not.

one 2 survives

```
1  a = [1, 2, 2, 3]
2
3  for x in a:
4      if x == 2:
5          a.remove(x)
6
7  print(a)
```

output

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

Remove an item and everything after it shifts left, while the index keeps climbing — **so the second 2 is skipped**.

Three ways that actually work

```python
# 1. build a new list — clearest
a = [x for x in a if x != 2]

# 2. loop over a copy, delete from the original
for x in a[:]:
    if x == 2:
        a.remove(x)

# 3. mark instead of delete, clean up afterwards
a = [None if x == 2 else x for x in a]
```

不要一边遍历一边删除。 新建一个列表最清楚，也最不容易出错。

PART 9 OF 9

List comprehension 列表推导

The same thing, four lines shorter

WITH A LOOP

```
squares = []
for x in range(10):
    squares.append(x**2)
print(squares)
```

WITH A COMPREHENSION

```
squares = [x**2 for x in range(10)]
print(squares)
```

Read it left to right: for every `x` in `range(10)`, take x^2.

The shape of a comprehension

- [expression for target in iterable if condition]
  - expression — what to put in the new list
  - for … in — where the values come from
  - if … — optional, keeps only the ones that pass
- The result is always a NEW list. The source is untouched.

Three parts: **what to take · where from · which ones**.

Adding a condition

filtering

```
1  nums = [3, -1, 4, -1, 5, -9, 2, 6]
2
3  print([x for x in nums if x > 0])
4  print([x * x for x in nums if x > 0])
5  print([abs(x) for x in nums])
```

output

```
[3, 4, 5, 2, 6]
[9, 16, 25, 4, 36]
[3, 1, 4, 1, 5, 9, 2, 6]
```

Filtering and transforming can share a line, but one job per line reads better.

When not to reach for one

- Two levels of nesting — write the loop instead
  - [[y for y in row] for row in grid] is already at the limit
- The expression needs more than one line to understand
- You are doing it for the side effect, not the result
  - a comprehension whose value you throw away should be a for loop

推导式的目的是更清楚，不是更短。 写出来不清楚就拆成循环。

随堂小测 · 第 2 轮

扫码作答 —— 这一轮六题

1. 下标和越界
2. append 和 extend 的区别
3. 就地修改的返回值是什么
4. 用 * 造矩阵的陷阱
5. b = a 和 c = a[:] 的 id
6. 边遍历边删

今天六个落点各一题。一题一题放，答完一题公布一题。

https://taohuang.info/cs1602/zh/quiz/5

What to take away

- A list stores addresses, not data — everything today follows from that
- b = a copies the arrow; b = [...] moves the arrow; b.append() changes the object
- In-place methods return None. a = a.append(x) destroys your list.
- a[:] copies one level. Nested data needs copy.deepcopy().
- Never delete from a list you are looping over.

只需要记住一条：**列表存放的是地址。**

Lab 5 — 六道题，列表实战

- **5-1** 成绩统计（热身）
- **5-2** 手写 `max` —— 不许调内置的那个
- **5-3** 有序去重：去掉重复元素但保持原来的先后
- **5-4** 矩阵转置
- **5-5** 找出并修正 Bug
- **5-6** 拷贝实验 —— 今天三个坑的正面交锋

另有 **B 段**和 **C 段**（命令行，**期末要考**）。**其中三道是刻意安排的陷阱题** —— 先自己踩进去，再自己爬出来。

Next time

A list can do almost everything. So why does Python have three more?

Next time: tuple, dict and set — and the one question that decides which of the four you should reach for.