



上海交通大学
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

Part I: one value at a time, with branches and loops.

Part II: **a whole batch of values** as one thing, and how to pick the right container for it.



| Where Lecture 4 Ended

- A variable is a **reference**: `b = a` copies the address, not the object
- Integers and strings are **immutable**, so sharing one object is harmless
- `for` takes items one at a time; `while` runs until a condition fails
- Every function carries type hints: `def bmi(w: float, h: float) -> float:`

前两条是今天的关键。今天一大半内容都是它们的直接后果。

| A Shared Shopping List

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

两个人看的是同一份清单，各自手机上的只是一个入口。

| An Opening Prediction

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>

| Today's Question



Why did a change when I only touched b?

A shared shopping list, in code.

PART 1 OF 9

Why a Container

So far every variable holds **one** value.

Real data comes in batches, often of unknown size.

| Five Scores, Five Variables

```
1  score1 = 87
2  score2 = 92
3  score3 = 65
4  score4 = 78
5  score5 = 90
6
7  total = score1 + score2 + score3 + score4 + score5
8  print("average:", total / 5)
```



| When the Requirements Change

- Find the highest score
 - if $\text{score1} > \text{score2}$ and $\text{score1} > \text{score3}$ 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，并不知道会有多少个。



| The List (列表)

One name, any number of values.

```
1 scores = [87, 92, 65, 78, 90]
2
3 print("count: ", len(scores))
4 print("avg: ", sum(scores) / len(scores))
5 print("highest:", max(scores))
```

一个名字管一批数据，多少个都行。



| Data Structures (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 and Reading a List

How to write a list down, and how to get an item back out.

List Literals

```
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。

I type() and len() of a 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.

| Indexing from 0

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

```
1  a = ["red", "green", "blue", "yellow"]
2
3  print(a[0])           # red           (the first)
4  print(a[1])           # green
5  print(a[3])           # yellow        (the last, len(a) - 1)
```

下标从 0 起。最后一项是 $\text{len}(a) - 1$ ，写 $\text{len}(a)$ 就越界。

| Negative Indices

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.

| IndexError

an error you will see often

```
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。遇到它，先确认列表的长度。

I Membership: in and not in

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

L4's for loop, applied to a list.

| for x in a

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**.

| Two Ways to Write One Loop

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`.

| 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

What a List Really Is

A list can hold anything, and a `[i]` is still instant.

How is that possible?

| Arrays (数组)

- 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: $\text{start} + \text{size} \times i$
 - 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。

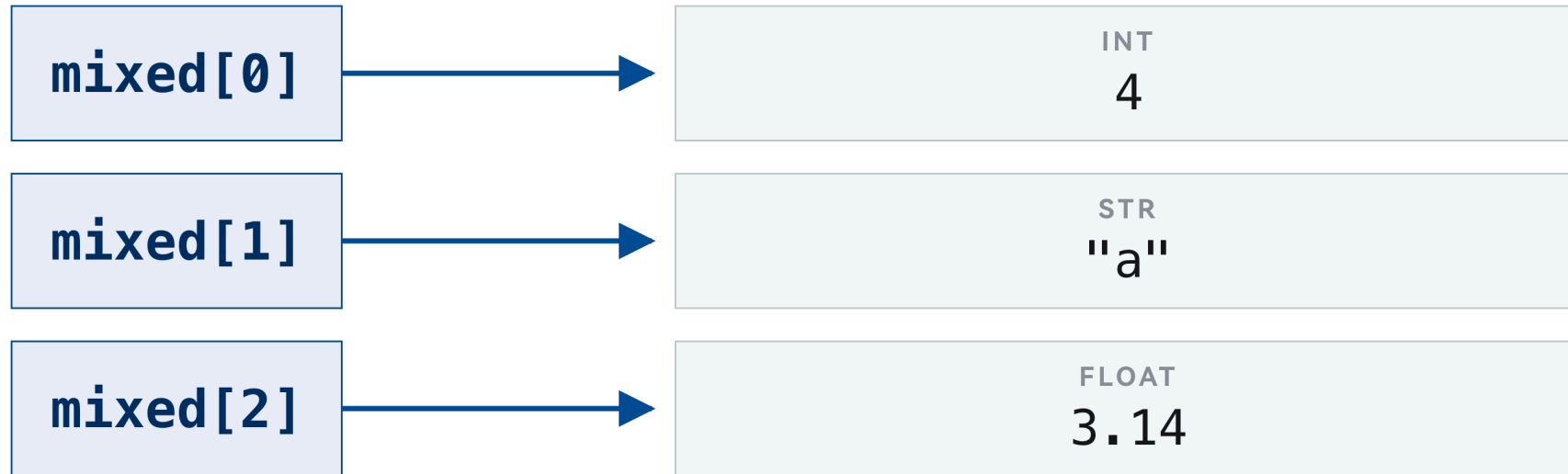
| A Python List Holds Anything

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

```
1 mixed = [4, "a", 3.14, True, [1, 2], None]
2
3 for x in mixed:
4     print(type(x).__name__, end=" ")
```

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

| A List Stores Addresses



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.

| Nesting Without Limit

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

```
1 deep = [[[[[1]]]]]]
2
3 print(deep[0][0][0][0][0][0])
4 print(len(deep))
```

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

| id() on a List

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: 4369797696
id of a[0]: 4409442808
id of b: 4369797696
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

Integers and strings cannot be changed; each "change" builds a new one.

A list **can be changed in place**.

| Lists Are Mutable (可变)

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] 4379284032
after:  [99, 2, 3] 4379284032
```

内容变了，地址没变。这就是「原地修改」。

| append()

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 +

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， $a + b$ 也逐项加，但另造一个新列表。

| In Place or a New List

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 4370141760
extend 4370141760
plus   4370461952
```

In-Place Methods Return 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。

| The `a = a.append(x)` 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)` · `a[::-1]` · `a + b` · `a[:]` build something new instead

改原列表的方法返回 `None`；要一个新列表，用 `sorted(a)` 这一类。



PART 6 OF 9

Slicing (切片)

L3 sliced strings with `s[i:j]`.

Lists slice the same way, and a list slice can also be **assigned to**.

| Half-Open Slices

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]
```

含头不含尾。 `a[1:4]` 取第 1、2、3 项，不含第 4 项，长度是 $4 - 1$ 。

| The Step

step

```
1 a = [10, 20, 30, 40, 50, 60]
2
3 print(a[::2])           # every other one
4 print(a[1::2])
5 print(a[::-1])         # backwards
```

output

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

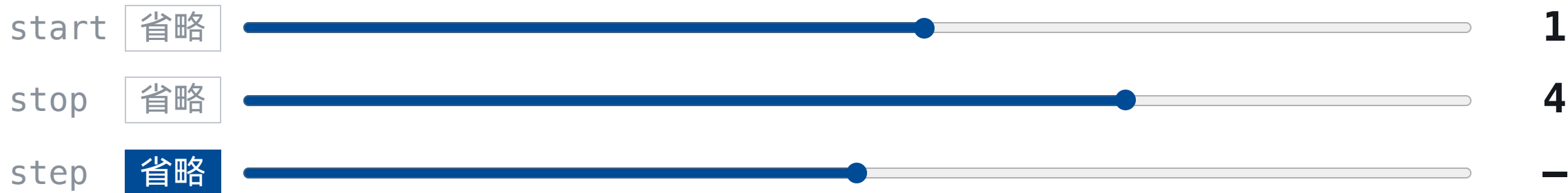
`a[::-1]` 得到一个倒序的新列表；`a.reverse()` 原地反转、返回 `None`。

| An Interactive Slicer

a[1:4]

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

→ **[20, 30, 40]**



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

| Slices Never Raise IndexError

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]
[]
[]
```

切片越界不报错，只会返回一个较短的列表或空列表。

| Slice Assignment

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 Is 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 比较是不是同一个对象。这两者不能混用。

| Copying by Slice Is Fast

copying 2,000,000 items

```
1 import time
2 f = list(range(2000000))
3
4 t = time.perf_counter()
5 g = []
6 for x in f:
7     g.append(x)
8 print("loop", round(time.perf_counter() - t, 4))
9
10 t = time.perf_counter()
11 h = f[:]
12 print("a[:]", round(time.perf_counter() - t, 4))
```

output

```
loop 0.0252
a[:] 0.0023
```

复制整个列表，`a[:]` 比自己写循环快得多。

PART 7 OF 9 · THE KEY IDEA

Two Names, One List

L3: `b = a` copies the address, so `a` and `b` share one object.

A list **can be changed**, so now the sharing shows.

| The Opening Question Revisited

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

The same thing as that shared shopping list.

| b = a Copies the Address

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 复制的是地址。a 和 b 是同一个列表的两个名字。

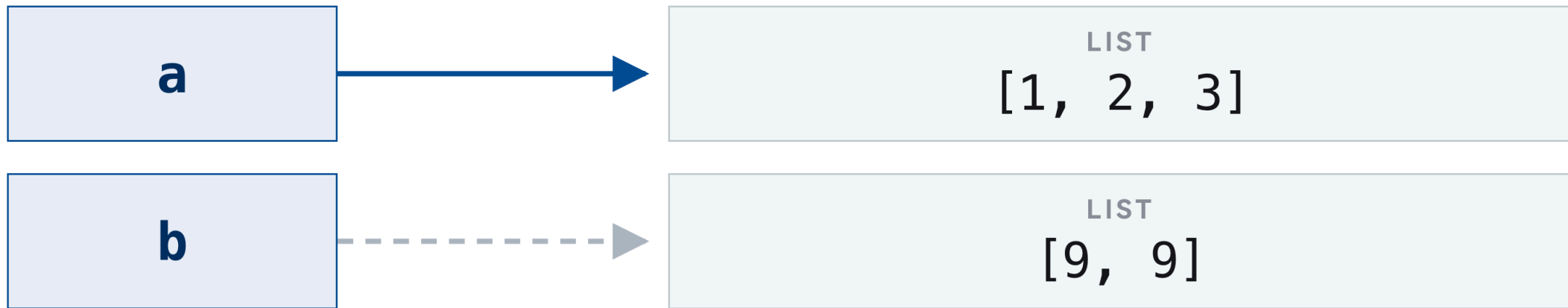
| One List, Two Names



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 vs 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 Your Own Copy

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)`.

| A Copy Goes Only One Level Deep

shallow copy

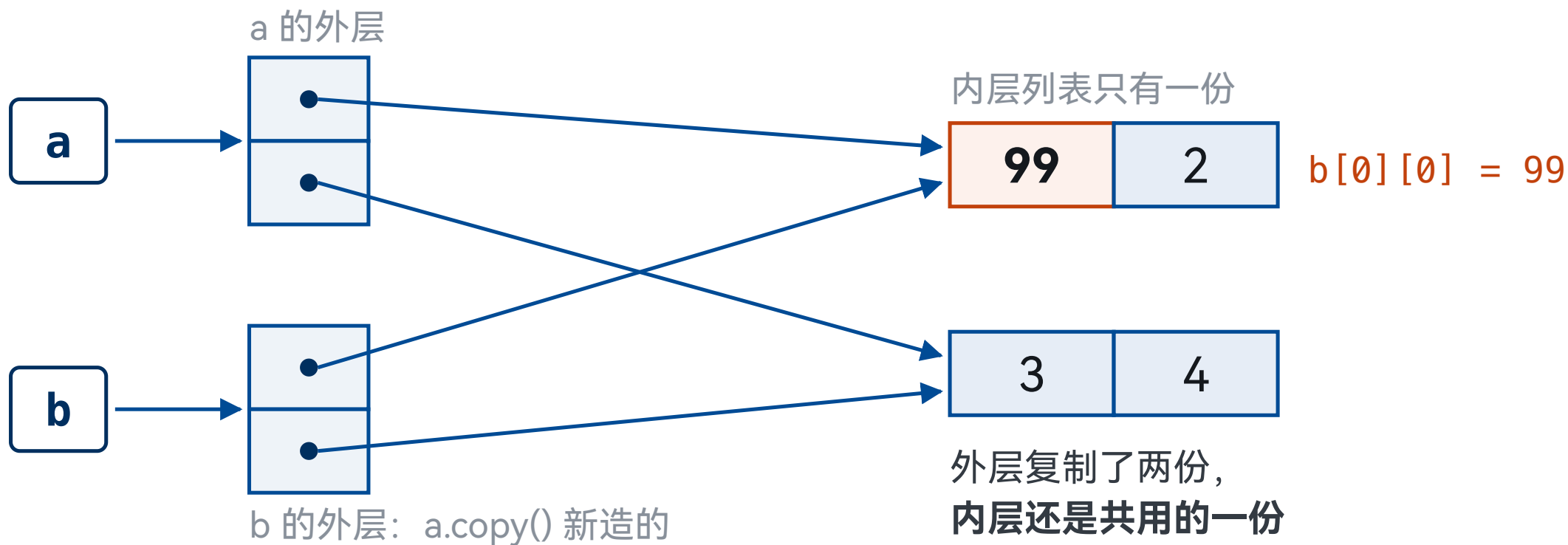
```
1  a = [[1, 2], [3, 4]]
2  b = a.copy()
3
4  b[0][0] = 99          # reaching INSIDE
5
6  print("a =", a)
7  print("b =", b)
8  print("a[0] is b[0]:", a[0] is b[0])
```

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.

| What a Shallow Copy Shares



`copy()` 复制的是外层的格子, 格子地址原样搬过去, 所以内层仍然共用。

| 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)`
- 浅拷贝复制外层的格子，格子内的地址原样搬过去，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.

| copy vs deepcopy

copy vs deepcopy

```
1 import copy
2 a = [[1, 2], [3, 4]]
3 shallow = a.copy()
4 deep = copy.deepcopy(a)
5
6 a[0][0] = 99
7 print("a      =", a)
8 print("shallow =", shallow)    # followed
9 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 靠得住。

| A Grid Built With *

[[0]*3]*3

```
1  grid = [[0] * 3] * 3      # looks fine
2  grid[0][0] = 1
3  print(grid)
4
5  good = []
6  for _ in range(3):
7      good.append([0] * 3)  # a new row each time
8  good[0][0] = 1
9  print(good)
```

output

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

* 3 复制的是三个地址，都指向同一行。改一行等于改三行。

| Never Default to a Mutable Object (可变默认值)

box=[]: one list, shared by every call

```
def add_item(item, box=[]):  
    box.append(item)  
    return box
```

```
print(add_item(1))  
print(add_item(2))
```

output

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

box=None: a new list each call

```
def add_item(item, box=None):  
    if box is None:  
        box = []  
    box.append(item)  
    return box
```

```
print(add_item(1))  
print(add_item(2))
```

output

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

默认值在 `def` 那一行只创建一次。默认值只用不可变的东西：数、字符串、None。

| Three Minutes with Your Neighbour

一个函数接收一个列表作为参数。要在函数内部排序，但不能改动调用者的那个列表。应该怎么写？

Two answers are correct here. Find both.



PART 8 OF 9

List Comprehension (列表推导)

"Loop over something, build a new list" comes up all the time.
Python has a one-line way to write it.

| 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.

| 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]
```

| When Not to Use 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

列表推导是为了更清楚，不是为了更短。写出来不清楚，就改回循环。



PART 9 OF 9

Deleting While Iterating

A loop and a change to the same list's length:
the one combination to avoid.

| A Loop That Skips Items

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 loop's position keeps climbing, **so the second 2 is skipped.**

| Correct Ways to Delete

three ways that work

```
1  a = [1, 2, 2, 3]
2  print([x for x in a if x != 2])    # 1. a new list
3
4  b = [1, 2, 2, 3]
5  for x in b[:]:                     # 2. loop over a copy
6      if x == 2:
7          b.remove(x)
8  print(b)
9
10 c = [1, 2, 2, 3]
11 while 2 in c:                       # 3. remove until gone
12     c.remove(2)
13 print(c)
```

output

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

不要一边遍历一边删除。用列表推导造一个新列表，最清楚，也最不容易出错。

随堂小测 · 第 2 轮



扫码作答 · 这一轮五题

- 1 下标和越界
- 3 用 * 造矩阵
- 5 边遍历边删

- 2 `a = a.append(4)` 之后
- 4 `b = a` 和 `c = a[:]` 的 id

下标、原地修改、矩阵、别名、边遍历边删各一题。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 address; `b = [...]` moves the arrow; `b.append()` changes the object
- In-place methods return `None`. `a = a.append(x)` loses your list
- `a[:]` and `copy()` copy one level. Nested data needs `copy.deepcopy()`
- Never use a list as a default value; never delete from a list you are looping over

| Lab 5 — 六道题，列表实战

- **5-1** 成绩统计（热身）
- **5-2** 手写 max：不许调用内置的那个
- **5-3** 有序去重：去掉重复元素，保持原来的先后顺序
- **5-4** 矩阵转置
- **5-5** 找出并修正 Bug
- **5-6** 拷贝实验：今天讲的三个常见错误

另有 **B 段**和 **C 段**（命令行，期末要考）。其中三道题是有意安排的陷阱，先自己写错，再自己找出原因。

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

Next time: tuple, dict and set, the common string methods, and the one question that decides which container to reach for.