

Lists

Every slide from the lecture, with what was said over it. The slides are rendered natively — the text is selectable, searchable, and prints cleanly.



Introduction to Computation (CS1602) · Lecture 5

Lists

Instructor: Tao Huang

Part II: Built-in Data Structures · Fall 2026

1

Slide 1 · Lists

这一讲讲列表。它是 Python 里最常用的容器，也是这门课遇到的第一个可以原地修改的类型。

第 3 讲说过，几个名字可以共用同一个对象；对整数和字符串，这不会带来问题，因为它们不能修改。列表可以修改，于是会出现一些初看很意外的现象。这一讲的后半部分专门讲这些现象。

PART II

Built-in Data Structures

第一部分讲的是单个的值，以及怎么用分支和循环处理它们。

第二部分讲怎么把一批值放在一起，当作一个整体来处理，以及遇到具体问题时，该选哪一种容器。一共四讲，今天是第一讲。



| 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。

3

Slide 3 · Where Lecture 4 Ended

到上一讲为止，已经能写出包含输入、判断、循环和函数的完整程序，但处理的都是单个的值：一个整数、一个字符串、一个布尔值。

从这一讲起，处理的是一批值。要考虑的问题也随之变多：怎么存、怎么取、怎么遍历，以及修改一批数据时会发生什么。

第3讲讲过：Python的变量是引用，赋值复制的是地址；整数和字符串不可变，所以几个名字共用一个对象不会出问题。今天的列表是可变的，这两条放在一起，就是今天后半部分所有现象的来源。

I 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。

4

Slide 4 · A Shared Shopping List

先说一件和代码无关的事。

两个室友共享一份购物清单。一个人在自己手机上加了「酱油」，另一个人什么都没做，打开一看，清单上也多了酱油。

这一点也不奇怪：两个人看的是同一份清单，各自手机上的只是一个入口，不是各自的副本。

同样的事情在 Python 里发生时，很多人会觉得莫名其妙。下一页就是这样一个例子。

I 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？

5

Slide 5 · An Opening Prediction

两段代码的结构一模一样：定义一个东西，把它赋给第二个名字，通过第二个名字做修改，再打印第一个名字。

左边是整数 `x`，右边是列表 `nums`。`append` 的意思是在列表末尾加一项，今天会讲。

左边打印 1，这一点第 3 讲已经讲过。右边打印什么？这一页不给答案，请先定一个判断，讲完别名那一节再回来对照。

I 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>

开场先收一道题，不公布答案。讲完别名与拷贝，再把这一轮的分布放出来对照。

现在选错没有关系。这道题的作用是先形成一个判断，有了判断，后面的解释才有对照。

| Today's Question



Why did a change when I only touched b?

A shared shopping list, in code.

7

Slide 7 · Today's Question

这一讲从头到尾在回答一个问题：只动了 b，为什么 a 也变了？

答案和那份共享购物清单是同一回事，只是发生在代码里。在回答之前，要先把列表本身讲清楚：怎么建、怎么取、怎么改、怎么切。

PART 1 OF 9

Why a Container

到目前为止，每个变量都只存一个值。可实际的数据往往是一批：一个班的成绩、一个文件里的所有行。这一节看看只用单个变量处理一批数据时会遇到什么问题。

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)
```

Five values, five names.

9

Slide 9 · Five Scores, Five Variables

先看没有列表的时候怎么写。

五个成绩，五个变量：score1 到 score5。要求平均分，就把五个加起来除以五。代码能运行，逻辑也没错。

变量个数很少、而且固定不变的时候，这样写可以接受。问题在下一页。



I 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。

10

Slide 10 · When the Requirements Change

需求变了，而且是很自然的变化。

如果是五十个成绩，就要写五十个变量名，加法要写五十项。要求最高分，得写一长串比较；要按分数排序，用五十个独立的变量几乎做不到。

更关键的是，数量往往在写代码时还不知道。文件里有多少行、用户会输入多少个数，要等程序运行起来才清楚，而变量名必须在写代码时就定下来。

所以需要一种新的东西：能存放一批值，数量可以等到运行时才确定。

| 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。

11

Slide 11 · The List (列表)

这个东西就是列表。

一个名字管一批值，多少个都可以。len 求个数，sum 求和，max 求最大值，平均分就是 sum 除以 len。

和上一页对比：五十个成绩和五个成绩，这段代码一个字都不用改。



I 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。

12

Slide 12 · Data Structures (数据结构)

列表是一种数据结构。这个词在这门课里会反复出现，意思是：一种组织数据的方式，加上针对它的一组操作。

组织方式决定了哪些操作快、哪些操作慢。所以很多时候，选对数据结构比优化代码更能解决问题。第二部分的四讲讲四种常用的数据结构，以及它们各自适合场合。

PART 2 OF 9

Building and Reading a List

先讲最基本的两件事：怎么写出一个列表，怎么从里面取出某一项。

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。

14

Slide 14 · List Literals

列表的写法是方括号，各项之间用逗号隔开。

里面可以是整数、浮点数、字符串，也可以是另一个列表。空列表就是一对空方括号。

最后一项后面多写一个逗号，Python 也接受。列表分多行写的时候这很方便：加一项只需要加一行，不用改上一行。



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.

15

Slide 15 · type() and len() of a List

不管里面放的是什么，列表本身的类型都是 list。

type 给出类型，len 给出有几项。注意最后一个：nested 里有一个列表，但 len 是 3。里面那个列表整个算一项，不会被展开。

处理嵌套结构时，这一点经常被弄错。



I 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.

16

Slide 16 · The list() Constructor (构造函数)

除了直接写方括号，还可以用 `list()` 来构造列表。

`list()` 不带参数得到空列表。给它一个字符串，会把字符串拆成一个个字符，因为字符串可以逐个取出字符。给它一个 `range`，会把 `range` 展开成列表，第 4 讲打印 `range` 时用的就是这个办法。

能逐个取出元素的东西，统称可迭代对象。`for` 后面的 `in` 能接的，也正是这一类东西。



I 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。

17

Slide 17 · Indexing from 0

取第几项用方括号加下标，和第 3 讲取字符串里的字符一样。下标从 0 开始。

第一项是 $a[0]$ ，第二项是 $a[1]$ ，最后一项是 $a[\text{len}(a) - 1]$ 。下标表示的是「离开头有多远」：第一项离开头 0 个位置。这一节的第四部分会看到，这样编号在计算地址时最方便。

从 0 开始是初学时最容易出错的地方。凡是涉及「第几个」的循环，写完先用一个三四项的小列表试一遍边界。



| 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.

18

Slide 18 · Negative Indices

负数下标从末尾往前数。`a[-1]` 是最后一项，`a[-2]` 是倒数第二项。

这个写法很常用，因为取最后一项不需要先知道长度。`a[len(a) - 1]` 和 `a[-1]` 结果相同，后者更短，也更不容易写错。

负数下标从 -1 开始。没有 -0，因为 -0 就是 0，那是第一项。



| 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。

19

Slide 19 · IndexError

下标越界会报 IndexError。

一个有 4 项的列表，合法的下标是 0 到 3，以及 -1 到 -4。写 `a[4]` 就会出错，报错信息是 `list index out of range`。

报错其实是好事：程序在出错的地方立即停下，而不是拿一个错误的值继续往下算。记住这一点，因为后面的切片不是这样：切片越界不报错。



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。

20

Slide 20 · Membership: in and not in

in 判断一个值在不在列表里，not in 反过来。结果是 True 或 False。

它比自己写循环逐个比对简洁得多，意思也更清楚。

最后一行：[1, 2] 不是 [1, 2, 3] 里的某一项，所以是 False。in 问的是「有没有一项等于它」，不是「有没有一段等于它」。

对列表用 in，要从头到尾逐项比较，列表越长越慢。第 6 讲会讲集合，集合的 in 几乎不受长度影响。

PART 3 OF 9

Walking Through a List

第 4 讲的 for 循环，在列表上用得最多。这一节讲遍历列表的几种写法。



| 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**.

22

Slide 22 · for x in a

遍历列表最直接的写法是 for x in a。

每一轮循环，x 依次取到列表里的一项。x 拿到的是元素本身，不是下标。这和 C、Java 里习惯的「用下标循环」不一样。

只需要每一项的值时，这是最好的写法：短、清楚、不会越界。



I 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`.

23

Slide 23 · Two Ways to Write One Loop

同一个循环的两种写法。

左边用下标：自己初始化 `i`、自己判断边界、自己加一，再用 `fruit[i]` 取值。右边直接遍历元素。结果一样，右边更短，也不可能越界。

什么时候需要知道下标？需要知道「这是第几项」的时候。即使那样，也有更好的写法，见下一页。



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))`.

24

Slide 24 · enumerate()

既要元素又要下标时，用 `enumerate`。

它每一轮给出两个值：下标和元素。所以循环变量写成 `i, x`，这是第 3 讲讲过的同步赋值。

和 `range(len(a))` 再取 `a[i]` 的写法相比，`enumerate` 更短，也不会出现下标和元素对不上的问题。

想让编号从 1 开始，写 `enumerate(a, 1)`。打印给人看的清单时常用。

PART 4 OF 9 · THE KEY SECTION

What a List Really Is

列表里可以放任何类型的值，而且不管列表多长，取第几项都一样快。这两点放在一起其实并不简单。这一节讲列表在内存里是什么样子，它是后半节课所有现象的根源。



I 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。

26

Slide 26 · Arrays (数组)

先说 C、Java 里的数组。数组是一段连续的内存，每一项类型相同、占的字节数相同。

所以第 i 项的地址可以直接算出来：起始地址加上每项的大小乘以 i 。一次乘法一次加法，不管 i 是多少，都一样快。

这也解释了下标为什么从 0 开始：第 0 项的地址就是起始地址，不用再减 1。

但 Python 的列表可以混放不同类型的值，下一页看看这会带来什么问题。

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?

27

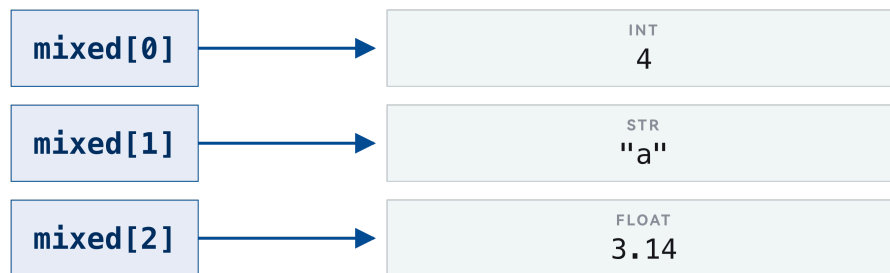
Slide 27 · A Python List Holds Anything

Python 的列表可以放任何东西，同一个列表里可以混着放整数、字符串、浮点数，甚至另一个列表。

这些值占的字节数各不相同。第 2 讲讲过，一个小整数对象就要 28 个字节，一个长字符串可以占几千个字节。大小不一样，上一页那个算地址的公式就用不上了。

那 Python 怎么还能按下标快速取到第几项？答案在下一页。

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.

28

Slide 28 · A List Stores Addresses

关键在于：列表里存的不是数据本身，而是数据的地址。

第 3 讲讲过，Python 的名字里存的是对象的地址。列表的每一格也一样：格子里放的是一个地址，指向真正的对象。

不管对象是一个小整数还是一个很长的字符串，地址的大小都是固定的。所以格子大小统一，按下标取值仍然很快；同时元素类型可以随意，因为真正的数据放在别处。

这个设计解释了三件事：列表为什么能混放各种类型；两个名字为什么能指向同一个列表；复制列表时，里面那一层为什么不会跟着复制。后两件马上就会遇到。



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.

29

Slide 29 · Nesting Without Limit

既然格子里存的是地址，这个地址也可以指向另一个列表，那个列表的格子又可以指向下一个列表，一层套一层。

deep 套了六层，最里面是 1。要取到它，得连写六次 [0]。而 len(deep) 是 1：最外层只有一项，就是指向第二层的那个地址。

实际写程序时很少嵌套这么深。两层的嵌套很常见，比如用列表的列表表示一张表格。



I 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.**

30

Slide 30 · id() on a List

第 3 讲用过 id(): 它返回一个对象的地址。

a 和 a[0] 的 id 不同: 列表是一个对象, 它的第 0 项是另一个对象, 列表的格子里存着第 0 项的地址。

b = a 之后, b 和 a 的 id 相同, a is b 是 True: 两个名字指向同一个列表。这一点后面会反复用到。

接下来几页都用 id 来验证「是不是同一个对象」, 而不是靠猜。

PART 5 OF 9

Changing a List

第 3 讲讲过, 整数和字符串不可变, 每次「修改」都是造一个新对象。列表不一样, 它可以在原地修改。这一节讲怎么修改, 以及修改时要注意什么。



I 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」。

32

Slide 32 · Lists Are Mutable (可变)

列表是可变的：对象还是那一个，内容可以修改。

看输出：修改前后，列表的内容变了，id 没变。还是同一个列表，只是第 0 项换了。

字符串正好相反。第 3 讲试过，`s[0] = "H"` 会直接报错，所有「修改」字符串的操作，实际上都在造一个新字符串。

I 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。

33

Slide 33 · append()

append 在末尾加一项。

它加的是「一项」：传进去一个列表，这个列表整个作为一项放进去，而不是把里面的元素逐个加进去。所以最后 len 是 5。

append 是在原来的列表上修改，不造新列表。这一点让它和后面的加号有本质区别。



I 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。

34

Slide 34 · append, extend and +

三种「加东西」的方式放在一起比较，这是这一节的重点。

append 把参数整个作为一项加进去。extend 把参数里的每一项分别加进去，所以传一个两项的列表会多出两项。加号造一个新列表，原来的两个都不变。

append 和 extend 修改原列表，加号不修改。选哪一个，取决于要不要保留原来的列表。

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
```

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

35

Slide 35 · In Place or a New List

这一页用 id 验证上一页的说法。

extend 之后，a 的 id 不变，说明是在原列表上加的。a = a + [5, 6] 之后，id 变了：加号造了一个新列表，a 改指到了新列表上。

这个区别在函数里特别要紧。把一个列表传进函数，函数里用 extend 或 append，调用者手里的那个列表也会跟着变，因为两边指向的是同一个列表；用加号再赋值则不会。



| 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。

36

Slide 36 · In-Place Methods Return None

原地修改列表的方法都返回 None。

append、extend、sort、reverse、insert、remove 都是如此。sort 是排序，reverse 是反转，insert 是在指定位置插入。它们的工作是修改原来的列表，没有新的东西要交回来，就统一返回 None。

这是有意的设计：返回 None 等于告诉调用者，这是一个原地操作。



I 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.

37

Slide 37 · The `a = a.append(x)` Bug

上一页那条规则，最常见的出错形式是 `a = a.append(x)`。

写的人以为是「把 `x` 加进去，再把结果赋回给 `a`」。实际发生的是：`append` 先把 `x` 加进列表，这一步是对的；然后返回 `None`，`None` 又被赋给了 `a`。于是 `a` 变成了 `None`，原来的列表反而找不到了。

这个错误不会当场报错，要等到下一次用 `a` 时才报错，报错信息还指向别的地方。出错的位置和报错的位置不一样，这类问题最难查。

正确写法就是 `a.append(x)`，不要赋值。



I 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。

38

Slide 38 · The In-Place Family

原地修改的方法可以归成一类，记住它们的共同点比逐个记更有用。

`append` 加一项，`extend` 加一批，`insert` 在指定位置插入，`remove` 按值删除第一个匹配项，`sort` 排序，`reverse` 反转，`clear` 清空。它们都修改原列表、都返回 `None`。

`pop` 是例外：它按下标取出一项并删除，然后把取出的那一项返回，因为这个值确实有用。

如果想要一个排好序的新列表、不动原来的，用 `sorted(a)`；想要一个反转的新列表，用 `a[::-1]`，下一节讲。

PART 6 OF 9

Slicing (切片)

第 3 讲讲过字符串的切片。列表的切片规则完全一样，另外列表是可变的，所以切片还可以放在等号左边，用来修改一段内容。



I 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。

40

Slide 40 · Half-Open Slices

a[i:j] 取从第 i 项到第 j 项，但不包含第 j 项。

这叫左闭右开，和第 3 讲字符串切片、第 4 讲的 range 是同一个约定。好处有两个：切出来的长度正好是 j 减 i；a[:i] 和 a[i:] 正好把列表分成两段，不重复也不遗漏。

开头的 0 可以省略，结尾省略表示到末尾，两个都省略就是整个列表。

| 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`。

41

Slide 41 · The Step

再加一个冒号可以指定步长：`a[start:stop:step]`。

`a[::2]` 每隔一项取一项，`a[1::2]` 从第二项开始每隔一项取一项。

`a[::-1]` 步长为负，得到倒过来的列表。这个写法很常见。它造的是一个新列表，和上一节的 `a.reverse()` 不同：后者原地反转，返回 `None`。

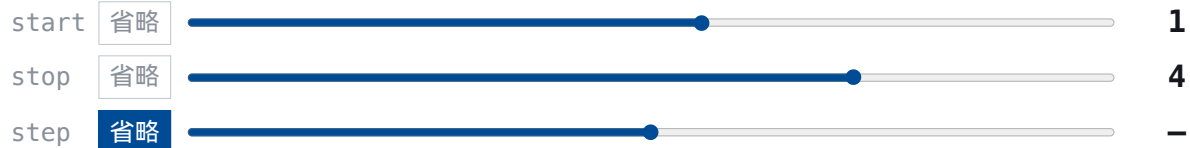
An Interactive Slicer



a[1:4]



→ [20, 30, 40]



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

42

Slide 42 · An Interactive Slicer

这一页可以动手操作。拖动下面三个游标，亮起来的格子就是切出来的部分，右边显示结果。

先把 stop 拖到 4，看第 4 格亮不亮。它不亮，这就是左闭右开。

再把 stop 拖到 9。这个列表只有 6 项，切到 9 却不报错，只给出到末尾为止的部分。下一页讲这一点。

最后把 step 拖到 -1，看看结果。

导出的 PDF 上是静态的一帧，要拖动请到课程网站的课件页。



| 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。

43

Slide 43 · Slices Never Raise IndexError

切片越界不报错。

前面讲过 `a[4]` 这样的下标越界会报 `IndexError`，但 `a[1:99]` 不会，它只返回从第 1 项到末尾的部分。`a[99:]` 也不报错，返回一个空列表。

这在很多场合很方便。比如取前十项，不足十项就有多少取多少，直接写 `a[:10]`，不用先判断长度。

但这也意味着切片写错了不会当场发现：拿到的是一个空列表或少几项的结果，要到后面才发现不对。不报错不等于正确。

| 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.

44

Slide 44 · Slice Assignment

列表的切片不但能读，还能赋值。字符串不可变，就不能这样做。

`a[1:3] = [20, 30]` 把第 1、2 项换成新的两项。两边的长度不必相等：`a[1:3] = [7, 7, 7, 7]` 用四项替换两项，列表变长。赋一个空列表，就是删掉那一段。

右边必须是可迭代对象。`a[1:3] = 20` 会报错，因为 20 不能逐个取出元素。



| 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。

45

Slide 45 · A Slice Is a New List

切片总是造一个新列表。

`b = a` 只是多了一个名字，`a is b` 是 `True`。`c = a[:]` 是切片，得到一个新列表，`a is c` 是 `False`；但内容相同，`a == c` 是 `True`。

所以 `a[:]` 是一个常见的写法，意思是把整个列表切一份，得到内容相同、但互相独立的新列表。后面讲拷贝时会用到。



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。

46

Slide 46 · Copying by Slice Is Fast

复制两百万项的列表，一种写法是自己写循环逐项 `append`，另一种是切片。`perf_counter` 返回当前时间，前后相减就是这段代码用了多少秒。

右边的数字是构建课件时实际测到的，每台机器不一样，但切片总是快很多。切片在 Python 内部用 C 代码一次完成；自己写的循环每一轮都要由解释器执行一次 `append`。

要注意的是，这只是常数倍的差别。两种写法处理 n 项，所花的时间都和 n 成正比。要让程序快上一个数量级，得换算法，而不是换写法。第 11 讲讲这个问题。

PART 7 OF 9 · THE KEY IDEA

Two Names, One List

第 3 讲讲过，`b = a` 复制的是地址，两个名字共用一个对象；整数和字符串不能修改，所以共用不会带来问题。

列表可以修改。两个名字共用一个列表时，通过一个名字修改，另一个名字就会看到。这一节回答开场的问题。

I 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.

48

Slide 48 · The Opening Question Revisited

回到开场那道题, 可以先把开课时收上来的分布放出来。

`nums2 = nums` 复制的是地址, 不是列表。赋值只是让第二个名字指向同一个列表, 没有产生新的列表。所以通过 `nums2` 做的 `append`, 从 `nums` 也能看到。

左边为什么不变? `x2 = x2 + 1` 并没有修改那个 1, 而是另造了一个 2, 再让 `x2` 改指过去。x 还指着原来的 1。

这就是那份共享购物清单: 两个入口, 一份清单。



I 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。

49

Slide 49 · b = a Copies the Address

用代码验证：b = a 之后，a is b 是 True，两个名字指向同一个列表。

所以 b.append(4) 之后，a 也变成了四项，因为自始至终只有一个列表。

同一个对象有两个名字，这种情况叫别名。

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.

50

Slide 50 · One List, Two Names

画成图更清楚：两个名字，两个箭头，指向同一个列表。

`b = a` 做的事情是「让 `b` 也指向 `a` 所指的對象」，而不是「把 `a` 的內容抄一份給 `b`」。

为什么这样设计？第 3 讲的取舍表讲过：赋值只复制一个地址，不管列表有一百万项还是一项，代价都一样小。

这个规则对后面所有可变类型都成立：字典、集合、以及第 9 讲自己定义的类。

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。

51

Slide 51 · Mutating vs Rebinding

这里要区分两种容易混淆的操作：修改和重新绑定。

`b.append(4)` 是修改：改的是列表本身。a 和 b 指的还是同一个对象，所以 a 能看到这个改动。

`b = [9, 9]` 是重新绑定：它没有动原来的列表，只是让 b 改指一个新列表。a 仍然指着原来那个，所以不受影响。

判断方法：看等号左边。左边只是一个名字，就是重新绑定；左边带方括号，或者调用的是方法，就是修改。

| 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)`.

52

Slide 52 · Getting Your Own Copy

想要一个独立的副本，有三种等价的写法：`a[:]`、`a.copy()`、`list(a)`。

三种都造一个新列表，之后修改其中一个不影响另一个。`a.copy()` 的意思最明显。

但下一页会看到，这种复制只解决了一半问题。



I 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.

53

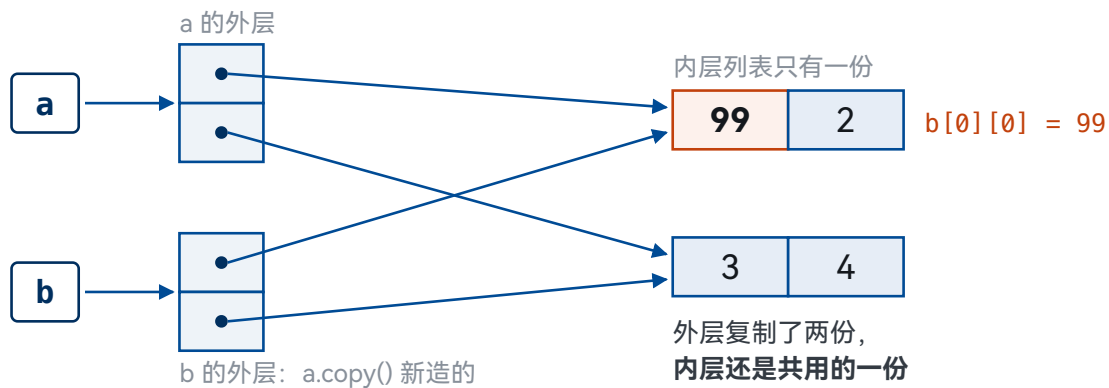
Slide 53 · A Copy Goes Only One Level Deep

a 是一个嵌套列表，里面有两个小列表。b = a.copy() 之后，修改 b[0][0]，a[0][0] 也变了。

最后一行 a[0] is b[0] 是 True：a 和 b 的第 0 项是同一个小列表。

这种只复制最外层的做法叫浅拷贝。为什么会这样，下一页画出来。

I What a Shallow Copy Shares



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

54

Slide 54 · What a Shallow Copy Shares

回忆这一讲第四部分：列表的格子里存的是地址。

`a.copy()` 造了一个新的外层列表，也就是图中 `b` 的外层。它把 `a` 外层格子里的地址原样抄了过去。抄过去的是地址，所以两个外层的格子，指向的是同一组内层列表。

`b[0][0] = 99` 先通过 `b` 的第 0 格找到第一个内层列表，再修改它的第 0 项。这个内层列表 `a` 也在用，所以从 `a` 也看得到 99。

这是这一讲最容易出错的地方，也最难自己发现：程序不报错，只是给出一个没有预料到的结果。



I 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.

55

Slide 55 · Deep Copy (深拷贝)

要连内层一起复制，用 `copy` 模块里的 `deepcopy`。

深拷贝沿着每一个地址往下，把每一层都复制一份，得到的新结构和原来完全独立，改哪一层都互不影响。

代价是慢，嵌套越深越慢。所以不要一律用 `deepcopy`：列表里只放数、字符串这类不可变的值时，浅拷贝就够了。



I 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。

56

Slide 56 · copy vs deepcopy

用代码验证：修改 a 里面那个小列表之后，shallow 跟着变了，deep 没有变。

三行输出放在一起，区别一眼就能看出来。遇到「明明只改了一个，却两个都变了」的情况，就回来看这一页。

I 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。

57

Slide 57 · A Grid Built With ``

这是嵌套列表最典型的一个错误：用乘法造二维表格。

`[[0] * 3] * 3` 看起来是三行三列的 0，打印出来也确实如此。但把 `grid[0][0]` 改成 1 之后，三行的第一项全变成了 1。

原因和浅拷贝一样：外层的乘法把同一个内层列表的地址重复了三次，三行其实是同一行。

正确的做法是在循环里每次新造一行，这样三行是三个不同的列表。循环变量写成下划线，表示这个变量用不到。学完列表推导后，这三行还可以写得更短。

内层的 `[0] * 3` 为什么没问题？因为 0 是整数，不可变，共用同一个 0 不会有任何影响。问题只出在可变对象上。



| 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。

58

Slide 58 · Never Default to a Mutable Object (可变默认值)

第 4 讲讲了默认参数，当时说还有一个问题要等讲完列表再讲，就是这一页。

左边第二次调用 `add_item(2)`，本来应该得到 `[2]`，结果却是 `[1, 2]`，带着上一次调用留下的 1。

原因在于默认值什么时候创建。那个空列表是在执行 `def` 那一行时创建的，整个程序里只创建一次。之后每次调用不传 `box`，`box` 指向的都是同一个列表，上一次 `append` 进去的东西还在。这又是两个名字共用一个可变对象。

右边是正确写法：用 `None` 当默认值，进入函数后判断，再新造一个空列表。这样每次调用得到的都是新列表。

结论：默认值只用不可变的东西，比如数、字符串、`None`。

I 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.

59

Slide 59 · Three Minutes with Your Neighbour

花三分钟和旁边的同学讨论这个问题。

把想法说出来，比自己默想更容易发现哪里没想清楚。

参考答案有两种：先用 `a.copy()` 复制一份再对副本调用 `sort`；或者直接用 `sorted(a)`，它返回一个排好序的新列表，不动原来的。

PART 8 OF 9

List Comprehension（列表推导）

前面好几次写过同一种循环：新建一个空列表，遍历另一组数据，逐个算出一个值 `append` 进去。这种写法太常见了，Python 为它提供了一种一行的写法。

| 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 .

61

Slide 61 · Four Lines Shorter

同一件事的两种写法：左边用循环，右边用列表推导。

右边这一行从左往右念：对 `range(10)` 里的每一个 x ，取 x 的平方，放进新列表。先学会读，再学着写。

列表推导不是必需的，任何列表推导都可以改写成循环。它的好处是把造新列表的过程写在一行里，读的人一眼就知道结果是一个列表。

| 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**.

62

Slide 62 · The Shape of a Comprehension

列表推导的结构是固定的，方括号里三部分：表达式、for ... in ... 、可选的 if 条件。

执行顺序和书写顺序不同：先执行 for，再判断 if，最后算表达式。所以读的时候先看中间那一段，就知道在遍历什么。

前面那个三行三列的表格，用列表推导可以写成 `[[0] * 3 for _ in range(3)]`。



I 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.

63

Slide 63 · Adding a Condition

加上 if 条件，只有满足条件的项才会进入结果。

注意 if 的位置。写在 for 后面是过滤，决定某一项要不要放进结果。写在最前面、配上 else，是第 4 讲的条件表达式，每一项都会放进结果，只是取值不同。这两种形式很容易弄混。

过滤的写法里没有 else，因为不满足条件的就不要了，不需要第二个值。



| 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。

64

Slide 64 · When Not to Use One

列表推导不是用得越多越好。有三种情况不该用。

第一，逻辑复杂，需要嵌套两三层 for 和好几个 if。这时写成普通循环更容易读懂。

第二，每一轮要做的事不止「算一个值」，还要打印、写文件、修改别的变量。列表推导的用途是造一个新列表，不是执行一串动作。

第三，结果根本用不到。只为了执行某些操作而写一个列表推导，会造出一个没人用的列表，白白占用内存。

判断方法：写完念一遍，念不顺就改回循环。

PART 9 OF 9

Deleting While Iterating

最后一个常见错误：一边遍历一个列表，一边删除它的元素。



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.**

66

Slide 66 · A Loop That Skips Items

这段代码想删掉所有的 2，结果还剩一个。

for 循环在内部记着现在走到了第几项。删掉第一个 2 之后，后面的元素整体往前移了一位，第二个 2 移到了刚删掉的那个位置上；循环却照常走向下一个位置，于是第二个 2 被跳过了。

这个错误有时候看起来是对的：要删的元素不相邻时，碰巧不会出问题；一旦相邻就会漏掉。时対时错的错误比每次都错的更难查。



| 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。

67

Slide 67 · Correct Ways to Delete

三种正确的写法，结果都是 [1, 3]。

第一种用列表推导，把要保留的挑出来，造一个新列表。这种写法最清楚，一般首选。

第二种遍历一份副本，在原列表上删除。b[:] 是副本，遍历副本的时候修改 b，不会影响循环。

第三种反复删除，直到列表里没有 2 为止。写起来简单，但每次 in 和 remove 都要从头找，列表很长时会慢。

记住这条规则：不要在遍历一个列表的同时改变它的长度。

I 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>

68

Slide 68 · 随堂小测 · 第 2 轮

第二轮小测，五道题。一题一题地放，每题答完当场公布。

【第 1 题 · 下标和越界】

在纸上把列表的每一项和它的下标写出来，正数下标写在上面，负数下标写在下面，然后按题目数一遍。凭感觉答很容易差一位。

【第 2 题 · `a = a.append(4)` 之后打印什么】

只有一个考点：**原地修改的方法返回 None**。`append` 已经把 4 加进去了，但它返回 None，None 又被赋回给 a，原来的列表就找不到了。

【第 3 题 · 用 * 造矩阵】

外层的乘法复制的是地址，三行是同一行。答错的同学，把那一页的两种写法各运行一遍，再对比每一行的 id。

【第 4 题 · `b = a` 和 `c = a[:]` 的 id】

`b = a` 复制的是地址，两个名字指向同一个列表，id 相同；`c = a[:]` 是切片，切片总是造一个新列表，id 不同。

这一题和开场那道题问的是同一件事。

【第 5 题 · 边遍历边删】

答错的同学，把那段代码在自己电脑上运行一遍，在循环里加一行 `print`，把每一轮的 x 和当前的列表都打印出来，就能看清第二个 2 是怎么被跳过的。



I 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

只需要记住一条：**列表存放的是地址。**

69

Slide 69 · What to Take Away

这一讲的要点都在这一页。

列表存的是地址，不是数据；`b = a` 复制的是地址；切片和 `copy` 只复制最外层，嵌套结构要用 `deepcopy`；原地修改的方法返回 `None`；默认值不用列表；不要边遍历边删除。

前两条是根本，其余都可以从它们推出来。如果只记一句话，就记住：赋值不复制对象。



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。

70

Slide 70 · Lab 5 — 六道题，列表实战

A 段六道题，覆盖列表的增删改查、切片、拷贝和嵌套结构。

其中三道是有意安排的陷阱题，对应今天讲的三个常见错误：原地修改的方法的返回值、浅拷贝、边遍历边删。自己写错一次再找出原因，比听讲印象深得多。

5-2 要求自己写 max，不许调用内置的 max。目的是练熟「遍历，同时记住当前最好的值」这个写法。找最小值、找第二大、找出现次数最多的，都是同一个结构。

5-4 矩阵转置要用嵌套列表，造新矩阵时注意今天讲的「用乘号造矩阵」那个问题。

A 段之外还有 B 段和 C 段，三段同时开放。C 段是命令行基本功，期末要考。

提交到「小作业 2」，它覆盖第 4 到 6 周，10 月 27 日截止。



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.

下一讲讲元组、字典和集合，并补齐字符串的常用方法。

列表几乎什么都能做，为什么 Python 还要另外三种容器？因为能做和适合做是两回事。下一讲会给出一个判断依据：遇到一个具体问题，该选哪一种。