

Conditions, Loops and Functions

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 4

Conditions, Loops and Functions

Instructor: Tao Huang

Part I: Foundations and Python Basics · Fall 2026

1

Slide 1 · Conditions, Loops and Functions

前三讲写的程序都是一条直线：第一行、第二行、第三行，执行完就结束。这样的程序对输入没有任何反应，每次运行做的都是同样的事。

这一讲加两样东西：分支和重复。有了这两样，程序就能根据情况走不同的路，也能把一段代码执行很多遍。

I 从前有座山



从前有座山，山上有座庙，庙里有个老和尚，

老和尚在给小和尚讲故事，故事讲的是——

从前有座山，山上有座庙，庙里有个老和尚，

老和尚在给小和尚讲故事，故事讲的是……

要让程序把这句话说一百遍，现在唯一的办法是把 `print` 复制一百次。

2

Slide 2 · 从前有座山

这个故事讲不完，因为故事里又套着同一个故事。

换成程序的问题：怎么让一段代码重复执行？

用前三讲学过的东西，只能把那行 `print` 复制一百遍。如果要一万遍呢？如果重复的次数要等程序运行起来才知道呢？复制粘贴就做不到了。

这一讲的后半段解决重复的问题。前半段先解决另一个更基本的问题：怎么让程序**根据情况走不同的路**。



| An Opening Prediction

A

```
for s in [50, 70, 90]:  
    if s >= 60:  
        print(s, "pass")  
print("done")
```

B

```
for s in [50, 70, 90]:  
    if s >= 60:  
        print(s, "pass")  
    print("done")
```

两段只差四个空格。请先在心里定一个答案。

3

Slide 3 · An Opening Prediction

两段代码逐字符对比，唯一的区别是最后一行前面多了四个空格。

for 和 if 今天才讲，但大致意思能猜出来：对 50、70、90 这三个分数逐个检查，60 分以上的打印 pass。

请先判断：两段的输出差多少？有没有一段会报错？

这一讲的最后一段讲缩进，讲完再回来看这道题。现在先把答案收上来。

I 随堂小测 · 第 1 轮



扫码作答 · 这一轮一题

1 两段的输出差多少

这一题现在不公布答案，讲到缩进那一节再回来。

<https://taohuang.info/cs1602/zh/quiz/4>

4

Slide 4 · 随堂小测 · 第 1 轮

开课先收一道题，不公布答案。

缩进是 Python 最省事的设计，也最容易出低级错误。这一讲最后一段讲缩进，讲完再回到这道题，把开课时的分布放出来对照。

可以先提示一句：**两段都不报错**。麻烦正在这里。



| Where Lecture 3 Ended

- A variable is a **reference**: the name holds an address, and the type lives on the object
- **input()** always returns a string. Convert it yourself
- Format output with an **f-string**
- A function is a name, parameters, a return value. **Defining is not running**
- **print is for people, return is for the program**
- And **the question L3 closed on**: what does `grade(95)` print?

今天的分支和循环，大多写在函数里。`grade(95)` 的答案在第一段末尾。

5

Slide 5 · Where Lecture 3 Ended

上一讲的五段：变量、字符串、类型转换与输入、格式化输出、函数。

和今天关系最大的是后两条。真实程序里的条件和循环几乎都写在函数体内，所以「定义不等于运行」和「函数内部用 `return`」今天会一直用到。

`return` 今天还有一个新用途：嵌套循环里，`break` 只能跳出一层，`return` 可以一次跳出所有层。

第一条的内存模型今天用得不多，第 5 讲讲列表时是核心。

上一讲结尾留了一道题：`grade(95)` 打印什么。答案在今天第一段的末尾。

OVERVIEW

Today

今天的内容用一句话概括：让程序会选路，也会绕回去重复。



I Outline

- **Conditions:** `if / elif / else`, choosing which lines run
- **Loops:** `while` and `for`, running the same lines many times
- **Bitwise operators:** working on the bits directly, as L2 promised
- **More on functions:** default arguments, `*args / **kwargs`
- **Type hints and indentation:** one habit to start, one mistake to avoid

五段。前两段是这一讲的主体，也是后面所有内容的基础。从第 5 讲开始，几乎每一段代码里都有循环。

第三段的位运算接第 2 讲「一切都是二进制位」，内容不多，但优先级那个问题必须讲。

第四段接第 3 讲的函数，讲参数的几种写法。第五段的类型提示是从这一讲开始要养成的习惯，讲义和参考解答从今天起都带类型标注。

| The Structured Program Theorem (结构化程序定理)



sequence + branch + repetition = any computable process

A theorem, proved by Böhm and Jacopini in 1966. After today you have all three.

8

Slide 8 · The Structured Program Theorem (结构化程序定理)

到上一讲为止，程序只有「顺序执行」这一种结构。加上今天的分支和重复，三种结构就齐了。

1966 年 Böhm 和 Jacopini 证明：顺序、分支、重复这三种结构，足以表达任何可计算的过程。这叫结构化程序定理。

也就是说，后面要学的递归、类、模块，都没有让 Python 能算的东西变多。它们解决的是另一个问题：程序越来越长的时候，怎么还能写清楚、读得懂。

PART 1 OF 5

Conditions

第 2 讲讲过，比较运算的结果是 True 或 False，当时还画过一张流程图：条件为真走左边，为假走右边。

这一段把那张图变成代码：用条件的真假决定执行哪几行。

I Boolean Expressions (布尔表达式)



comparison

```
1 print(3 > 2)
2 print(3 == 2)
3 print("abc" != "abd")
4 print(3 >= 3)
```

output

```
True
False
True
True
```

第 2 讲讲过这些。if 后面要的就是这样一个 True 或 False。

10

Slide 10 · Boolean Expressions (布尔表达式)

六个比较运算符：大于、小于、大于等于、小于等于、等于、不等于。结果一律是 True 或 False。

第 2 讲还讲过，一个等号是赋值，两个等号是比较。这个区分今天更要紧：if 后面少写一个等号，报错信息不一定指得清楚。

字符串也能比较，按第 3 讲讲的字典序。



| Precedence of and / or / not

precedence

```
1 print(True or False and False)
2 print((True or False) and False)
```

output

```
True
False
```

not 最先，其次 and，最后 or。拿不准就加括号。

11

Slide 11 · Precedence of and / or / not

第一行里 and 先算：False and False 得 False，然后 True or False 得 True。

第二行加了括号，顺序反过来：先算 True or False 得 True，再算 True and False 得 False。

两行只差一对括号，结果相反。

和算术优先级一样，建议始终是：拿不准就加括号。a or b and c 读的人要停下来想一想，a or (b and c) 一眼就能看懂。

Chained Comparison (链式比较)



written as in mathematics

```
1 x = 5
2
3 print(1 < x < 10)           # same as 1 < x and x < 10
4 print(0 <= x <= 100)
```

output

True
True

C 和 Java 里 $1 < x < 10$ 不报错，但意思完全不同：先算 $1 < x$ 得到一个布尔值，再拿它和 10 比。

12

Slide 12 · Chained Comparison (链式比较)

Python 允许把比较连起来写，写法和数学一样，意思也和数学一样。

这是 Python 比 C、Java 方便的地方。那些语言里 $1 < x < 10$ 也合法，但它先算 $1 < x$ 得到一个布尔值，再拿这个布尔值和 10 比较，结果和数学上的意思对不上，而且不报错。

还有一个细节：链式比较里中间那一项只求值一次。 $1 < f(x) < 10$ 只调用一次 f ，展开成 `and` 的写法就要调用两次。

| if / elif / else



a multi-way branch

```
1  score = 85
2
3  if score >= 90:
4      print("excellent")
5  elif score >= 80:
6      print("good")
7  elif score >= 60:
8      print("pass")
9  else:
10     print("fail")
```

output

good

Top to bottom: the first true branch runs, the rest are skipped.

13

Slide 13 · if / elif / else

elif 是 else if 的缩写。三点要注意：条件后面的冒号不能少；分支里的语句缩进四个空格；从上往下，第一个为真的分支执行，其余全部跳过。

最后一点最重要。它解释了为什么第二个分支可以只写 `score >= 80`，不必写成 `80 <= score < 90`：能走到这一行，说明 `score >= 90` 已经是假的了。

少写半个条件，不只是为了省字。同一个边界写两遍，两处就可能写得不一致，这种错误很难发现。

下一页把 `score = 85` 的执行过程画出来。



Slide 14 · How a Multi-Way Branch Runs (多路分支的执行)

所以一个 if-elif-else 结构里，最多只有一支会执行。

I Order of the Branches



错

```
if score >= 60:  
    print("pass")  
elif score >= 90:  
    print("excellent")  
  
# a 95 is graded "pass"
```

对

```
if score >= 90:  
    print("excellent")  
elif score >= 60:  
    print("pass")  
  
# strictest condition first
```

左边这段**不报错**，只是九十分以上的人全被判成了及格。

上一讲那道题：grade(95) 打印 pass。多路分支从最严格的条件写起。

15

Slide 15 · Order of the Branches

上一讲结尾那道题的答案：grade(95) 打印的是 pass，不是 excellent。

95 分先满足了 `score >= 60`，于是进入第一个分支，后面的 `elif` 根本不会被检查。上一页那张图里灰色的部分，就是这种情况。

这是初学阶段最常见的逻辑错误。它不报错，只有拿一个九十分以上的分数去试，才会暴露出来。

写多路分支之前，先把各个分支的条件按严格程度排好，从最严格的写起。分数分级、税率档位这类问题都是这样。



I Leap Years with Early Return

early return

```
1 def is_leap(year):
2     if year % 400 == 0:
3         return True
4     if year % 100 == 0:
5         return False
6     return year % 4 == 0
7
8 for y in [1900, 2000, 2024, 2026]:
9     print(y, is_leap(y))
```

output

```
1900 False
2000 True
2024 True
2026 False
```

The rule: divisible by 4 is a leap year, unless divisible by 100, unless also divisible by 400.

16

Slide 16 · Leap Years with Early Return

闰年规则有三层，正好用来练习多路分支。

这里的写法是从最特殊的条件开始，一路 **return** 出去：先看能不能被 400 整除，能就返回 True，函数到此结束；不能再看 100，最后看 4。

这种写法叫早返回。每判断完一层，能下结论就立即返回，读代码的人不需要同时记住好几个还没结束的分支。

1900 年不是闰年，2000 年是闰年。这两个年份正好对应规则里最特殊的两种情况，测试时要专门挑出来试。



I Two Other Ways to Write It

NESTED: works, but tiring to read

```
def leap(y):  
    if y % 4 == 0:  
        if y % 100 == 0:  
            if y % 400 == 0:  
                return True  
            else:  
                return False  
        else:  
            return True  
    else:  
        return False
```

ONE EXPRESSION: harder to read

```
def leap(y):  
    return (y % 4 == 0  
            and (y % 100 != 0  
                  or y % 400 == 0))
```

三种写法结果完全一样。选早返回，因为它最好读。

17

Slide 17 · Two Other Ways to Write It

左边是嵌套写法。四层缩进，读的时候要一直记着自己在哪一层，而且每个 else 离它对应的 if 很远。

右边一个表达式写完。很短，但要把三个条件的关系同时在脑子里拼出来，修改时也容易出错。

三种写法算出的结果完全相同，对计算机来说没有区别。选早返回那种，只因为人读起来最省力。

一条经验：嵌套超过两层，通常就该考虑重写。先看能不能改成早返回。

| The Conditional Expression (条件表达式)



A if cond else B

```
1 score = 75
2 result = "pass" if score >= 60 else "fail"
3 print(result)
```

output
pass

不要嵌套它。写成 `a if p else (b if q else c)` 时，就该换回 `if-elif-else` 了。

18

Slide 18 · The Conditional Expression (条件表达式)

格式是「值1 if 条件 else 值2」。条件为真取前一个值，否则取后一个。

它是一个表达式，会算出一个值，所以可以直接赋给变量、写进 f-string、作为函数的参数。if 语句做不到这些。

适用范围很窄：只有两种情况，而且两个值都很短。一旦想嵌套，就说明应该换回完整的 `if-elif-else`。



| match-case (模式匹配)

Python 3.10 and later

```
1 def http_message(code):
2     match code:
3         case 200:
4             return "OK"
5         case 301 | 302:      # | means "or"
6             return "Redirect"
7         case 404:
8             return "Not Found"
9         case _:             # _ is the wildcard
10            return "Unknown"
11
12 for c in [200, 302, 404, 418]:
13     print(c, http_message(c))
```

output

```
200 OK
302 Redirect
404 Not Found
418 Unknown
```

| means **or**; _ is the wildcard, the equivalent of else.

19

Slide 19 · match-case (模式匹配)

match-case 是 Python 3.10 引入的语法，可以代替一长串 elif。

它比 elif 整齐：要比较的那个值只写一次，各个分支平铺在下面。

竖线把几个值合成一个分支。下划线是通配符，放在最后，接住前面都没匹配上的情况，作用相当于 else。

match 比 elif 强的地方不在这里，在下一页。

Matching Structure and Binding Names



pattern matching

```
1 def describe(point):
2     match point:
3         case (0, 0):
4             return "the origin"
5         case (0, y):
6             return f"on y axis: {y}"
7         case (x, y):
8             return f"point ({x}, {y})"
9         case _:
10            return "not a point"
11
12 for p in [(0, 0), (0, 5), (2, 7), "hello"]:
13     print(p, "->", describe(p))
```

output

```
(0, 0) -> the origin
(0, 5) -> on y axis: 5
(2, 7) -> point (2, 7)
hello -> not a point
```

case (0, y) does two things at once: it **checks** that the first is 0 and **binds** the second to y.

20

Slide 20 · Matching Structure and Binding Names

这一页是 match 和 if 最大的区别。

case (0, y) 同时做了两件事：检查这个值是不是由两个值组成、第一个是不是 0；如果是，就把第二个值绑定到名字 y 上，分支里可以直接使用 y。

用 if 写同样的逻辑，要先判断类型，再判断长度，再取出第二个值，再赋给一个变量，一共四步。

圆括号的写法叫元组，第 6 讲讲。这里只需要知道它把两个值放在一起。

match 不要滥用。只是比较几个数值时，if-elif 更直白。这门课不考复杂的模式匹配，能读懂基本用法就够了。

PART 2 OF 5

Loops

分支解决了「走哪条路」。开场那个老和尚的故事还没有解决：怎么让同一段代码执行很多遍。

Python 有两种循环：while 和 for。



| while Loops

three moving parts

```
1 n = 1
2 while n <= 5:
3     print(n, end=" ")
4     n += 1
5 print()
```

output

1 2 3 4 5

初始化 $n = 1$ 条件 $n \leq 5$ 更新 $n += 1$, 缺一个都会出错。

22

Slide 22 · while Loops

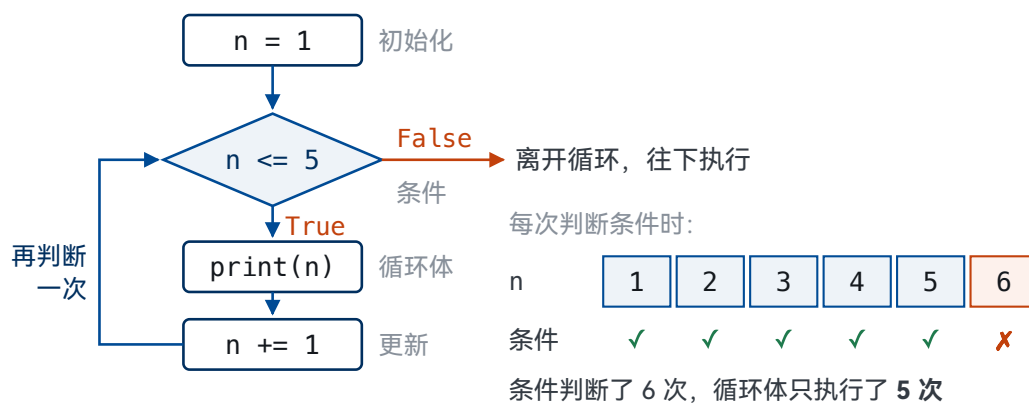
while 后面跟一个条件。条件为真，就执行一遍循环体，然后回到条件重新判断；直到条件为假，才往下执行。

三个要素缺一不可：循环变量要先初始化；条件要有可能变成假；循环体里要有改变条件的语句。

这里用了第 3 讲讲过的 end 参数，让五个数打印在同一行。最后那个空的 print() 用来补一个换行。

下一页把这段代码的执行过程画出来。

I How a while Loop Runs (while 的执行过程)



每执行完一遍循环体，都要回到条件重新判断。条件为 False，才离开循环。

23

Slide 23 · How a while Loop Runs (while 的执行过程)

左边是流程，右边是每次判断条件时 `n` 的值。

`n` 从 1 开始。判断 `n <= 5`，为真，执行循环体打印 `n`，再执行 `n += 1`。然后沿左边那根线回到条件，再判断一次。

这样重复下去。`n` 变成 6 时，条件第一次为假，程序从右边离开循环，接着执行循环后面的语句。

注意右边的表：条件一共判断了 6 次，循环体只执行了 5 次。最后一次判断的结果是 False，那一次循环体不执行。

如果去掉 `n += 1` 这一步，`n` 永远是 1，条件永远为真，程序就停不下来了。下一页就是这种情况。



| An Infinite Loop (死循环)

这一页没有输出面板：这段代码永远不会停，构建课件时真去运行，构建就会卡住。

this page does not run

```
1  n = 1
2  while n <= 5:
3      print(n)
4      # forgot n += 1
5      # n stays 1, the condition stays true
```

在自己电脑上不小心运行了死循环，按 **Ctrl+C** 中断。

24

Slide 24 · An Infinite Loop (死循环)

漏掉更新语句，条件就永远为真，程序会一直打印下去。

这一页是整份课件里唯一没有输出面板的代码页。其余每一页的输出都是构建课件时真正运行得到的，而这一段运行不完。

死循环不报错，程序看起来在正常工作，只是永远不结束。遇到这种情况，按 Ctrl+C 中断。

下一页讲 for。能用 **for** 的地方就用 **for**，一个重要原因是 for 不需要自己写更新语句，也就不会漏掉。

| for Loops



one item at a time

```
1 for ch in "Python":
2     print(ch, end=" ")
3 print()
4
5 for x in [10, 20, 30]:
6     print(x, end=" ")
7 print()
```

output

```
P y t h o n
10 20 30
```

Anything you can **take items out of one by one** can follow `in`.

25

Slide 25 · for Loops

`for` 的意思是：把 `in` 后面那个东西里的元素逐个取出来，每取出一个，执行一遍循环体。字符串可以逐个取出，取出来的是一个个字符。方括号括起来的叫列表，第 5 讲专门讲。`in` 后面还可以是元组、字典、集合等等。它们背后的共同机制叫可迭代对象，第 13 讲讲。`for` 不需要写初始化和更新，元素取完循环就自动结束。所以它比 `while` 更不容易写错。



I range()

range

```
1 print(list(range(5)))           # 0 to 4
2 print(list(range(2, 8)))        # 2 to 7
3 print(list(range(0, 10, 3)))    # step 3
4 print(list(range(10, 0, -2)))   # backwards
```

output

```
[0, 1, 2, 3, 4]
[2, 3, 4, 5, 6, 7]
[0, 3, 6, 9]
[10, 8, 6, 4, 2]
```

range(a, b) 含头不含尾，和第 3 讲的切片是同一个约定。

26

Slide 26 · range()

要把一段代码重复 n 次，for 后面需要一个有 n 个元素的东西。range 就是用来生成这样一串整数的。

一个参数是从 0 到 $n-1$ ，两个参数是从 a 到 $b-1$ ，第三个参数是步长。

含头不含尾，和切片是同一个约定。所以 `range(5)` 正好有 5 个数，`range(a, b)` 有 $b-a$ 个数，个数正好等于两个端点之差。

步长可以是负数，表示倒着数。

每一行都在 range 外面套了一个 `list()`，原因在下一页。



range Is Not a List

why wrap it in list()

```
1 print(range(5))
2 print(list(range(5)))
3
4 total = 0
5 for i in range(1, 101):
6     total += i
7 print("1 to 100 =", total)
```

output

```
range(0, 5)
[0, 1, 2, 3, 4]
1 to 100 = 5050
```

`range(1000000)` uses almost no memory: it **does not build** a million numbers; it computes the next one when asked.

27

Slide 27 · range Is Not a List

直接打印 `range(5)`，得到的是 `range(0, 5)`，而不是那五个数。`range` 并不把这些数存下来，它只记住起点、终点和步长，需要下一个数的时候再算出来。

所以 `range(1000000)` 几乎不占内存，而存一百万个数的列表要占几十兆。这种「用到时才计算」的做法叫惰性求值，第 13 讲还会讲到。

`list()` 把它展开成列表，只是为了打印出来看。写循环时不需要展开，直接写 `for i in range(...)`。

下面三行是 `range` 最常见的用法：从 1 加到 100。



I Choosing Between for and while

USE for

you know how many times

you have a collection to walk

most of the time

the count **is** fixed, so you
cannot forget the update,
and cannot loop forever

USE while

you do **not** know how many

you loop until something holds

reading **input** until it **is** valid
iterating toward a precision

you own the init, the
condition **and** the update

能用 for 就用 for。

28

Slide 28 · Choosing Between for and while

判断标准只有一条：进入循环之前，知不知道要循环多少次。

知道就用 for。遍历一组数据也属于这一类，数据有多少个是确定的。

不知道就用 while。典型的场景是反复读取输入，直到输入合法；或者反复迭代，直到达到某个精度。要循环几次，只有算的过程中才知道。

实际程序里大多数循环是 for。优先用 for，是因为它把初始化和更新交给了语言，少了两处可能出错的地方。



I The $3n+1$ Problem (考拉兹猜想)

the Collatz conjecture

```
1 def collatz_steps(n):
2     steps = 0
3     while n != 1:
4         if n % 2 == 0:
5             n = n // 2
6         else:
7             n = 3 * n + 1
8             steps += 1
9     return steps
10
11 for start in [6, 7, 27]:
12     k = collatz_steps(start)
13     print(f"from {start}: {k} steps to 1")
```

output

```
from 6: 8 steps to 1
from 7: 16 steps to 1
from 27: 111 steps to 1
```

至今没有人能证明，从任何正整数出发最后都会回到 1。

29

Slide 29 · The $3n+1$ Problem (考拉兹猜想)

规则很简单：偶数除以 2，奇数乘 3 加 1，一直做下去，直到得到 1。

这里只能用 while，因为要走多少步，事先算不出来。上一页说的「不知道要循环多少次」，就是这种情况。

从 27 出发要走 111 步，中间会升到 9232 再降下来；从 6 和 7 出发只要十几步。步数之间看不出规律。

这个问题叫考拉兹猜想，1937 年提出，至今没有被证明，也没有被推翻。计算机已经验证了非常大范围内的所有整数，但验证再多也不等于证明。

一个九行的程序，背后是一个悬而未决的数学问题。



| break and continue

break: leave the whole loop now

```
for i in range(2, 50):  
    if 100 % i == 0:  
        print("first divisor:", i)  
        break
```

output

first divisor: 2

continue: skip the rest of this round

```
for i in range(1, 11):  
    if i % 3 == 0:  
        continue  
    print(i, end=" ")
```

output

1 2 4 5 7 8 10

break 结束整个循环 · continue 只结束这一轮

30

Slide 30 · break and continue

有时循环不需要按部就班地走完。

左边找 100 的第一个大于 1 的因数。找到就打印，然后 break，循环立即结束，后面的数一个都不再看。

右边打印 1 到 10 里不能被 3 整除的数。遇到 3 的倍数就 continue，跳过这一轮剩下的 print，直接进入下一轮。

区分方法：只是这一轮不要了，用 continue；整个循环都不要了，用 break。



I break in Nested Loops

错

```
for i in range(3):
    for j in range(3):
        if j == 1:
            break # inner only
# the outer loop goes on
```

对

```
def find(...):
    for i in range(3):
        for j in range(3):
            if ...:
                return i, j
# return leaves every level
```

To leave two levels at once, **pull the loops into a function and return**. A return ends the whole function, however deep it is.

这也是「循环嵌套不超过两层」这条规范的一个来由。

31

Slide 31 · break in Nested Loops

break 只对它所在的那一层循环起作用。左边这段里，内层循环每次在 j 等于 1 时跳出，但外层循环照样跑完三轮。

初学时常有人以为 break 会跳出所有层，然后看不懂程序的输出。

右边是常用的做法：把嵌套循环放进一个函数，找到结果就 return。第 3 讲讲过，return 会立即结束整个函数，不管它在第几层循环里。

往年课件里有一条编程规范：循环嵌套不超过两层，里面有 return 或 break 时尤其如此。需要第三层时，就该把它抽成一个函数。

| for-else



for-else

```
1 def find_divisor(n):
2     for i in range(2, n):
3         if n % i == 0:
4             print(f"{n} is divisible by {i}")
5             break
6     else:
7         print(f"{n} is prime")
8
9 find_divisor(15)
10 find_divisor(17)
```

output

```
15 is divisible by 3
17 is prime
```

The `else` runs when the loop **finishes normally**; a `break` skips it.

32

Slide 32 · for-else

循环也可以带 `else`，这是 Python 特有的写法，不少人写了几年也不知道。

规则只有一条：循环正常结束，执行 `else`；被 `break` 打断，不执行 `else`。

15 在 `i` 等于 3 时找到因数，`break` 跳出，`else` 不执行。17 一直找到最后也没找到因数，循环正常结束，执行 `else`，打印 `prime`。

不用 `for-else`，就要多一个标记变量：先设 `found = False`，找到因数时改成 `True` 再 `break`，循环结束后再判断 `found`。`for-else` 省掉了这个变量。

I A Misleading Keyword



It means "the loop was never broken out of", not "the condition failed"

It should have been called nobreak. Learn to read it, since you will meet it in other people's code. In your own, a found flag is usually clearer.

33

Slide 33 · A Misleading Keyword

else 这个词在 if 语句里的意思是「条件不成立」，在循环里的意思却是「没有被 break 打断」。同一个关键字有两种含义，这是公认的设计缺陷。

社区里有人建议把它改名叫 nobreak，但改关键字会让大量已有代码失效，所以一直没有改。不少编程规范干脆禁止使用它。

这门课的要求是：**能读懂就行**。别人的代码里会遇到它，读不懂就会误解程序的逻辑。自己写的时候，多用一个 found 变量，读的人不会困惑，这样更好。

PART 3 OF 5

Bitwise Operators

第 2 讲讲过，整数在计算机里是一串二进制位。前面的运算符都把整数当作一个整体来算，这一段的运算符直接对每一位进行操作。

它们用在哪里，这一段的第四页会给出一个实际的例子。



| Six Bitwise Operators (位运算符)

- **& AND** 按位与: 1 only where both bits are 1
- **| OR** 按位或: 1 where either bit is 1
- **^ XOR** 按位异或: 1 where the two bits differ
- **~ NOT** 按位取反: every 0 becomes 1, every 1 becomes 0
- **<< left shift**: shift left, pad with 0. One place left is $\times 2$
- **>> right shift**: shift right. One place right is $\text{floor}(\div 2)$

They turn the integer into binary and work **bit by bit**.

35

Slide 35 · Six Bitwise Operators (位运算符)

前四个是逐位的逻辑运算：与、或、异或、取反。把两个数写成二进制、上下对齐，每一列单独算。

异或的规则是「两位不同得 1」。它有一个有用的性质：一个数和同一个数异或两次，会变回原来的值。

后两个是移位。左移一位相当于乘 2，右移一位相当于整除 2，和十进制里把数字整体左移一位相当于乘 10 是同一个道理。

取反涉及负数在计算机里的表示方法，这门课不展开。

Bit by Bit



line the bits up

```
1 a, b = 0b1100, 0b1010    # 12 and 10
2
3 print(f"a      = {a:04b}")
4 print(f"b      = {b:04b}")
5 print(f"a & b = {a & b:04b} = {a & b}")
6 print(f"a | b = {a | b:04b} = {a | b}")
7 print(f"a ^ b = {a ^ b:04b} = {a ^ b}")
```

output

```
a      = 1100
b      = 1010
a & b = 1000 = 8
a | b = 1110 = 14
a ^ b = 0110 = 6
```

0b 开头是二进制写法；{值:04b} 按二进制输出，**补零到 4 位**，各位才对得齐。

36

Slide 36 · Bit by Bit

0b 开头的数是二进制写法，0b1100 就是 12。

把 a 和 b 按二进制对齐打印出来，逐列对照，就能看清每个运算符做了什么。

与：只有第一列两个都是 1，结果是 1000。或：一列里有一个 1 就得 1，结果是 1110。异或：两位不同才得 1，第二列和第三列不同，结果是 0110。

格式说明 04b 的意思是按二进制输出、宽度 4、不足补 0。不补零，位数不同的两个数就对不齐，这张表就没法看了。



Shifts and the Parity Test

n & 1 tests parity

```
1 n = 5
2 print(n, "<< 1 =", n << 1)      # times 2
3 print(n, ">> 1 =", n >> 1)      # floor-divide by 2
4
5 for k in [7, 8, 9, 10]:
6     print(k, "odd" if k & 1 else "even")
```

output

```
5 << 1 = 10
5 >> 1 = 2
7 odd
8 even
9 odd
10 even
```

n & 1 takes the **lowest bit**: if it is 1, the number is odd.

37

Slide 37 · Shifts and the Parity Test

左移一位等于乘 2，右移一位等于整除 2。注意是整除，5 右移一位得 2，不是 2.5。

下面是位运算最常见的用法：n & 1 把最低位以外的位全部变成 0，只留下最低位。二进制最低位是 1 的数是奇数。

它和 n % 2 == 1 完全等价。在 Python 里两者的速度几乎没有差别。

平时写 % 就好，可读性更重要。讲它是因为别人的代码里经常会出现，尤其是从 C 语言改写过来的代码。



| Permission Bits (权限位)

three switches in one number

```
1 READ, WRITE, EXEC = 0b100, 0b010, 0b001
2
3 perm = READ | EXEC          # r-x
4 print(f"{perm:03b}")
5 print((perm & WRITE) != 0)  # may write?
6 print((perm & READ) != 0)   # may read?
```

output

```
101
False
True
```

一个 bit 当一个开关：| 打开，& 查看。ls -l 里的 rwxr-xr-x 就是这样存的。

38

Slide 38 · Permission Bits (权限位)

位运算有什么用？一个常见的用途是用一个整数存好几个开关。

lab02 里用 ls -l 看文件时，最左边有一串 rwxr-xr-x，当时说现在不用看懂。r、w、x 分别表示可读、可写、可执行，每一个字母就是一个二进制位：有是 1，没有是 0。

这里给三个开关各分配一位。READ | EXEC 把读和执行两位打开，得到 101，也就是 r-x。要查某一位是否打开，就和它做一次与运算：结果不是 0，说明这一位是开的。

注意 != 0 那两行都加了括号。原因在下一页。



Precedence of Bitwise Operators

错

```
8 & 4 + 1
# not (8 & 4) + 1 = 1
# but 8 & 5 = 0

1 << 2 + 3
# not (1 << 2) + 3 = 7
# but 1 << 5 = 32
```

对

```
(8 & 4) + 1
(1 << 2) + 3
# always parenthesize
```

结合顺序是：+ - 先于 << >>, << >> 先于 & ^ |, & ^ | 又先于比较运算符。

不要背这张表。位运算和别的运算符写在一起时，一律加括号。

39

Slide 39 · Precedence of Bitwise Operators

算术运算符比位运算符先算。这和很多人的直觉相反，因为位运算看上去更「底层」，好像应该先算。

8 & 4 + 1 里，加法先算，变成 8 & 5，结果是 0。按 (8 & 4) + 1 理解的人，会以为结果是 1。

和比较运算符放在一起时，Python 的规则符合直觉：n & 1 == 1 先算与，再比较。但 C 语言正好相反，比较先算。同一行代码在两种语言里意思不同，从 C 改写代码时容易出错。

结论很简单：不要去背优先级，加括号。上一页的写法就是这样。

PART 4 OF 5

More on Functions

第 3 讲讲了函数的三要素。其中参数还有几种更灵活的写法，print 其实已经在用了：它能接受任意多个值，还有 sep 和 end 两个可以不写的参数。

这一段讲这些写法是怎么定义出来的。



I Default Arguments (默认参数)

default values

```
1 def greet(name, greeting="Hello"):
2     return f"{greeting}, {name}"
3
4 print(greet("Alice"))
5 print(greet("Alice", "Good morning"))
```

output

```
Hello, Alice
Good morning, Alice
```

Give a parameter a default and **the caller can omit it.**

41

Slide 41 · Default Arguments (默认参数)

在参数后面写等号和一个值，就给了它一个默认值。调用时不传这个参数，就用默认值；传了就用传进来的值。

这样一个函数既能照顾最常见的情况，又能在需要时定制。大多数调用者不必关心那些有默认值的参数。

第 3 讲的 print 就是这样：sep 默认是一个空格，end 默认是换行，不写就用默认值。

I Defaults Come Last



wrong order

```
1 def bad(greeting="Hello", name):  
2     return f"{greeting}, {name}"
```

traceback

```
File "example.py", line 1  
    def bad(greeting="Hello", name):  
        ^^^  
SyntaxError: parameter without a default follows parameter with a default
```

报错是 **SyntaxError**：这是语法规则，函数定义本身就通不过。

42

Slide 42 · Defaults Come Last

有默认值的参数必须排在没有默认值的参数后面，否则连函数定义都通不过，直接报语法错误。

道理不难想：如果允许这样定义，调用 `bad("Alice")` 时，"Alice" 该给谁？按位置应该给 `greeting`，可 `name` 没有默认值，又不能空着。规则自相矛盾，所以语言直接禁止这种写法。

这是这一讲里为数不多的「写出来就报错」的错误，很容易改。

默认值还有一个要注意的地方，要等下一讲讲完列表才讲得清楚，到时候再说。



| Positional and Keyword Arguments

three ways to call it

```
1 def describe(name, age, city):  
2     return f"{name}, {age}, from {city}"  
3  
4 print(describe("Alice", 18, "Shanghai"))  
5 print(describe(name="Alice", age=18, city="Shanghai"))  
6 print(describe("Alice", city="Shanghai", age=18))
```

output

```
Alice, 18, from Shanghai  
Alice, 18, from Shanghai  
Alice, 18, from Shanghai
```

混用时，按位置传的参数必须写在前面。

43

Slide 43 · Positional and Keyword Arguments

第 3 讲讲过按位置和按名字两种传参方式。三种调用的结果完全一样：第一种全部按位置，第二种全部按名字，第三种混用。

混用时，按位置传的参数必须写在前面。Python 先按位置把前几个参数对应好，剩下的再按名字分配。如果按位置传的参数夹在中间，就说不清它该对应第几个参数了。

一条经验：参数只有一两个、意思很明显时，按位置传；参数多，或者有几个类型相同、容易弄混时，按名字传。

| *args



when you do not know how many

```
1 def total(*numbers):
2     print("got:", numbers)
3     return sum(numbers)
4
5 print(total(1, 2, 3))
6 print(total(1, 2, 3, 4, 5))
7 print(total())
```

output

```
got: (1, 2, 3)
6
got: (1, 2, 3, 4, 5)
15
got: ()
0
```

One star collects every positional argument into a **tuple**. Zero of them is fine.

44

Slide 44 · *args

参数个数事先不确定时，用一个星号加一个名字。调用时传多少个位置参数，都会被打包在一起交给它。

打印出来是用圆括号括起的一组值，这叫元组，第 6 讲讲。现在把它理解成一组不能修改的值就够了。sum 可以直接对它求和。

一个参数也不传也可以，收到的是空元组，求和得 0。

print、max、min 都能接受任意多个参数，用的就是这个机制。



I ****kwargs**

the order is fixed: plain -> *args -> **kwargs

```
1 def show(first, *rest, **opts):
2     print("first =", first)
3     print("rest  =", rest)
4     print("opts  =", opts)
5
6 show(1, 2, 3, mode="fast", retry=2)
```

output

```
first = 1
rest  = (2, 3)
opts  = {'mode': 'fast', 'retry': 2}
```

Two stars collect the keyword arguments into a **dict**. args and kwargs are only customary names; * and ** are the syntax.

45

Slide 45 · **kwargs

两个星号把所有没有对应上的关键字参数收集起来，放进一个字典。字典里记着每个参数的名字和值，第 6 讲讲字典。

三种参数可以一起用，顺序固定：普通参数、*args、**kwargs。

这里 1 给了 first，2 和 3 被 rest 收走，两个关键字参数被 opts 收走。

args 和 kwargs 这两个名字本身没有特殊含义，换成别的名字也能运行。但请沿用这两个名字，大家都这样写，换了名字，读代码的人要多花时间确认有没有别的用意。

No Function Overloading (没有函数重载)



the later def wins

```
1 def f(x):
2     return x * 2
3 def f(x, y):
4     return x + y
5 print(f(3))    # the one-argument f is gone
```

traceback

```
Traceback (most recent call last):
  File "example.py", line 5, in <module>
    print(f(3))    # the one-argument f is gone
        ^^^^^
TypeError: f() missing 1 required positional argument: 'y'
```

同名的函数，后定义会覆盖前面的，不管参数一样不一样。

46

Slide 46 · No Function Overloading (没有函数重载)

在 C++、Java 里，名字相同、参数不同的几个函数可以同时存在，这叫重载。Python 没有重载。

原因第 3 讲讲过：def 也是一次赋值，把函数对象绑到那个名字上。第二个 def f 把 f 改指到新的函数，原来那个函数就没有名字指向了。这和先写 x = 1 再写 x = 2 是同一回事。

所以 f(3, 4) 能正常运行，f(3) 报 TypeError，说缺少一个参数。

想根据参数个数做不同的事，可以用默认参数或者 *args。

实际写代码时，一个文件里不要有两个同名的函数。复制一个函数再改一改的时候最容易出这个问题，而且不报错，前一个函数就这样失效了。

PART 5 OF 5

Type Hints and Indentation

一个函数叫什么，大致说明了它做什么；但它接受什么类型的参数、返回什么类型的值，只看名字不知道。类型提示用来把这两件事写清楚。

这一段最后回到开场那道缩进题。

I Type Hints (类型提示)



Python 3.5 and later

```
1 def c_to_f(c: float) -> float:  
2     return c * 9 / 5 + 32  
3  
4 print(c_to_f(37))
```

output

98.6

c: float says the argument should be a float; -> float says a float comes back.

48

Slide 48 · Type Hints (类型提示)

参数名后面加冒号写参数的类型，括号后面加箭头写返回值的类型。这个语法从 Python 3.5 开始支持。

这一行的意思是：这个函数接受一个浮点数，返回一个浮点数。不用读函数体，就知道怎么调用它、会得到什么。

下一页有一个重要的限制。



I Hints Are Not Checked

try passing a string

```
1 def c_to_f(c: float) -> float:
2     return c * 9 / 5 + 32
3 print(c_to_f("37"))
```

traceback

```
Traceback (most recent call last):
  File "example.py", line 3, in <module>
    print(c_to_f("37"))
    ~~~~~^
  File "example.py", line 2, in c_to_f
    return c * 9 / 5 + 32
           ~~~~~^
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

类型提示不做检查。 类型传错照样运行，直到用到它的那一行才报错。

49

Slide 49 · Hints Are Not Checked

传一个字符串进去，Python 照样接受，直到执行乘法那一行才报错。

这一点要说清楚：类型提示对 Python 解释器来说基本等于注释，运行时不做任何检查。写了类型提示，不等于类型就一定对。

报错信息说的是字符串不能乘以浮点数，完全没有提到参数应该是 float。

既然不检查，为什么还要写？下一页回答。



| Why Write Type Hints

- **One: they are documentation that cannot go stale.**
 - A comment drifts away from the code. An annotation sits on the parameter
- **Two: your editor can use them.**
 - Completion gets sharper, and wrong types are underlined as you type
- **Three: they are the spec you hand an AI.**
 - Write the signature and the types first, let the AI fill the body. Far more precise than a paragraph of prose

三条评论，一条比一条重要。

50

Slide 50 · Why Write Type Hints

第一，文档不会过期。写在注释里的「这个参数是整数」，改代码时很容易忘了同步，注释就和代码对不上了。类型提示就写在参数旁边，改参数时一定看得见。

第二，编辑器能用上它。VS Code 装了 Python 扩展后，带类型提示的函数补全更准确，传错类型会当场标出来，不用等到运行。

第三条第 15 讲会展开。先说结论：让 AI 写一个函数时，与其用一段话描述，不如先写好函数签名和类型，让它补全函数体。下一页是一个例子。

I A Signature as a Specification



What goes in, what comes out, what it does: **all three lines say it.**

a spec an AI can read

```
1 def merge_sorted(a: list[int], b: list[int]) -> list[int]:  
2     """Merge two sorted integer lists into one sorted list."""  
3     ...
```

从这一讲起，讲义里的函数示例和参考解答都带类型提示。

51

Slide 51 · A Signature as a Specification

这三行是一份完整的规格说明：两个参数都是整数列表，返回值也是整数列表，文档字符串说明了函数的行为。

把这三行交给 AI，它写出的实现基本不会偏离要求。如果只说一句「写一个函数，把两个排好序的列表合起来」，它就要猜：元素是什么类型？返回新列表还是修改原来的？输入没排好序怎么办？

`list[int]` 表示元素都是整数的列表，列表下一讲讲。三个点是 Python 的一个合法语句，表示「这里暂时空着」，常用在写好了签名、还没写实现的时候。

课程约定：从这一讲开始，讲义和参考解答都带类型提示。本周实验要求所有函数都写类型提示。



| Common Signatures

four common signatures

```
1 def f1(x: int) -> str:
2     return str(x)
3
4 def f2(items: list[int]) -> int:
5     return sum(items)
6
7 def f3(name: str, age: int = 18) -> str: # default
8     return f"{name} is {age}"
9
10 def f4(x: float) -> None: # no return
11     print(x)
12
13 print(f1(42), f2([1, 2, 3]), f3("Ada"))
```

output

```
42 6 Ada is 18
```

没有返回值就写 -> None，对应第 3 讲：没有 return 的函数返回 None。

52

Slide 52 · Common Signatures

这四个例子覆盖了现阶段最常用的几种写法。

f2 里的 list[int] 表示元素是整数的列表。方括号里写元素的类型，这种写法以后用在字典、元组上也一样。

f3 说明默认值和类型提示可以一起写，顺序是参数名、冒号、类型、等号、默认值。

f4 写了 -> None。第 3 讲讲过，没有 return 的函数返回 None，所以它的返回类型就是 None。不要因为没返回值就不写，写出来，读的人才知道这是有意为之，不是忘了写。

I The Opening Question Revisited



"done" outside the for

```
for s in [50, 70, 90]:  
    if s >= 60:  
        print(s, "pass")  
print("done")
```

output

```
70 pass  
90 pass  
done
```

"done" inside the for

```
for s in [50, 70, 90]:  
    if s >= 60:  
        print(s, "pass")  
    print("done")
```

output

```
done  
70 pass  
done  
90 pass  
done
```

两段都不报错。一段只打印一次 done，一段每轮都打印。

53

Slide 53 · The Opening Question Revisited

回到开场那道题。可以先把开课时收上来的答案分布放出来。

左边的 print 顶格写，在 for 外面，整个循环结束后执行一次。右边缩进了四个空格，属于循环体，每一轮都执行，三轮打印三次。

注意右边输出的顺序：第一行就是 done。50 那一轮不及格，不打印 pass，但循环体里那句 print 照样执行。

关键在于**两段都不报错**。Python 认为两种缩进都是合法的程序，只是含义不同，哪一种是想要的，只能自己判断。

这是用缩进划分代码块的代价：缩进错了，程序的含义就变了，而且往往不报错。

相应的习惯是：**写完循环，回头检查每一行的缩进是不是在想要的那一层**，特别是循环体的最后几行。

I Tabs and Spaces



错

```
def f():  
    print(1)      # 4 spaces  
    print(2)      # one tab  
  
# TabError
```

对

```
def f():  
    print(1)  
    print(2)  
  
# 4 spaces throughout
```

Tab 和四个空格在屏幕上可能一样宽，Python 却认为它们不同，会报 `TabError: inconsistent use of tabs and spaces in indentation`。

VS Code 默认把 Tab 转成空格。从网上复制代码时最容易出这个问题。

54

Slide 54 · Tabs and Spaces

这是一个低级错误，但在屏幕上看不出来：两种缩进显示出来可能完全一样。

解决办法只有一条：统一用 4 个空格。VS Code 默认把 Tab 键转成空格，自己输入一般不会出问题。

出问题基本都是在复制粘贴时：从网页、PDF 或别人的代码里复制一段过来，里面混着 Tab。遇到 `TabError`，把那几行删掉重新输入最快。

VS Code 还可以打开「显示空白字符」，Tab 和空格就能区分开了。

I 随堂小测 · 第 2 轮



扫码作答 · 这一轮四题

- 1 `score = 95` 打印什么
- 2 这段循环打印什么
- 3 `6 & 3 + 1` 的值
- 4 `*rest` 收到什么

条件、循环、位运算、参数各一题。一题一题放，答完一题公布一题。

<https://taohuang.info/cs1602/zh/quiz/4>

55

Slide 55 · 随堂小测 · 第 2 轮

第二轮小测，四道题，前四段各一题。一题一题地放，每题答完当场公布。

【第 1 题 · `score = 95` 时打印什么？】

考的是「第一个为真的分支执行」这条规则，以及分支顺序写反的后果。解这类题的方法是从上往下逐个代入条件，遇到第一个为真的就停，后面的一律不看。

如果正确率不高，回到多路分支那张图再讲一遍。

【第 2 题 · 这段循环打印什么？】

考的是 `break`、`continue` 和循环的 `else` 三者的配合。解法是按轮次在纸上模拟：每一轮 `i` 是多少、执行到哪一句、是跳过这一轮还是跳出整个循环。循环的题不能靠猜。

在纸上模拟执行的习惯，后面讲递归时更加重要。

【第 3 题 · `6 & 3 + 1` 的值是多少？】

考的是位运算的优先级。加法比按位与先算，所以先算 `3 + 1` 得 4，再算 `6 & 4`。6 是二进制 110，4 是 100，按位与得 100，也就是 4。

按 `(6 & 3) + 1` 算会得到 3，这个错误答案也在选项里。

【第 4 题 · `show(5, 6, 7, 8, level=1)` 里，`rest` 是什么？】

考的是三种参数怎么分配：先按位置把普通参数填满，剩下的位置参数进 `*args`，关键字参数进 `**kwargs`。

5 给 first, 6、7、8 进 rest, level 进 opts。所以 rest 是 (6, 7, 8)。选 (5, 6, 7, 8) 的同学, 是忘了 first 先拿走了一个。

RECAP



I What to Take Away

- sequence + branch + repetition **expresses any computable process**
- if-elif-else takes the **first true** branch. Write the **strictest condition first**
- **Use for whenever you can.** while is for "I do not know how many times"
- break leaves the loop, continue skips the round. **Nested, break leaves one level**
- A loop's else means "**never broken out of**". Read it, rarely write it
- Mix bitwise with anything else and **always parenthesize**
- Defaulted parameters go last. *args is a tuple, **kwargs a dict
- **Type hints are not checked, and still worth writing.** Indent with 4 spaces

56

Slide 56 · What to Take Away

八条。第二条和第四条最容易在实际写代码时出问题, 而且都不报错。

第三条会一直用到学期结束。从第 5 讲开始, 几乎每一段代码里都有 for。

第八条的类型提示从今天起是课程约定, 本周实验要求所有函数都写类型提示。



Lab 4 — 十道题，条件与循环

- 4-1 `grade(score)` 成绩分级：注意分支顺序
- 4-2 `digit_sum(n)` 各位数字之和，不许转成字符串
- 4-3 `is_prime(n)` 判断质数
- 4-4 `collatz_steps(n)` 考拉兹步数：偶数除 2，奇数乘 3 加 1
- 4-5 `triangle(n)` 打印星号三角形，末尾不要换行
- 4-6 猜数游戏：写一个程序，不是函数
- 4-7 `hanoi_steps(n)` 汉诺塔最少步数 · 下周个人作业 1 的预热
- 4-8 ~ 4-10 姓名缩写、凯撒密码、手写进制转换：在字符串和数位上循环

另有 B 段和 C 段（命令行，期末要考）。从这次开始，所有函数都要求写类型提示。

57

Slide 57 · Lab 4 — 十道题，条件与循环

A 段十道题。

4-1 考分支顺序：多路分支要从最严格的条件写起。写反了，条件宽的分支会先被选中，严格的分支永远轮不到，而且不报错。

4-2 不许转成字符串，要用整除和取余把各位数字一位一位取出来。4-3 和 4-4 都是典型的循环题。4-4 就是今天讲的考拉兹猜想。

4-5 要用第 3 讲的格式说明，把宽度和对齐结合起来才能排整齐。注意末尾不要多一个换行，判题是逐字符比较的。

4-7 是下周个人作业 1 的预热，题目是汉诺塔。这周只要求算出最少步数，先找出规律，下周再写完整的解法。

4-8 到 4-10 是三道小题：姓名缩写要用不带参数的 `split()`；凯撒密码用第 2 讲的 `ord` 和 `chr`，注意 `k` 可以是负数；进制转换和 4-2 是同一个思路，注意 `n` 等于 0 的情况。

从这次开始，所有函数都要求写类型提示。

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。



You can repeat now. But the data is still scattered

L5: lists. Until today you handled one value at a time. With a list, a whole batch becomes one thing you can pass, compute with and store.

这一讲的循环解决了「重复做一件事」，但每次处理的还是单个的值。

下一讲讲列表：把一批数据放在一起，当作一个整体来传递和处理。循环负责逐个处理，列表负责存放，两者经常一起出现。

今天提到但没有展开的元组和字典，第 6 讲讲。

第 3 讲说过，几个名字共用一个可以修改的对象时，会出现意外的现象。列表就是第一个可以修改的对象，下一讲会看到具体的例子。