

条件、循环与函数

课件的每一页，配上讲这一页时说的话。幻灯片是网页原生渲染的，文字可以选中、可以搜索，也可以直接打印。



Introduction to Computation (CS1602) · Lecture 4

Conditions, Loops and Functions

Instructor: Tao Huang

Part I: Foundations and Python Basics · Fall 2026

1

第 1 页 · Conditions, Loops and Functions

前三讲你写的程序都是一条直线：第一行、第二行、第三行，跑完结束。这样的程序做不了什么——它对输入毫无反应，每次运行都干同样的事。

这一讲加两样东西：分岔和重复。加完之后，你手里就是一门图灵完备语言的全部控制结构了。

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……

要让程序说一百遍这句话，你现在的办法是复制粘贴一百次。

2

第 2 页 · 从前有座山

这个故事之所以讲不完，是因为它把自己套在了自己里面。

换成程序的说法：怎么让一段代码重复执行？

以你现在会的东西，唯一的办法是把那行 `print` 复制一百遍。一百遍还能忍，一万遍呢？要重复的次数得等程序跑起来才知道呢？

这一讲的后半段解决这个问题。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。

| Predict: how far apart are these two outputs?



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。先在心里定一个答案。

3

第 3 页 · Predict: how far apart are these two outputs?

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。

请先自己判断：它们的输出差多少？有没有一段会报错？

这一讲的最后一段专门讲缩进。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 上一页那两段，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

第 4 页 · 随堂小测 · 第 1 轮

开课先收一道题，不公布。等讲完缩进，再把这一轮的分布放出来对照。

缩进是 Python 最省事的设计，也是它最容易出低级错误的地方。

现在把答案收上来，但不公布。这一讲的最后一段讲缩进，讲完再回到这里，把开课时的分布放出来。

提示一句：**两段都不报错**。这正是它麻烦的地方。



| 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
- **`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. Inside a function, always `return`**
- And **the question L3 closed on**: what does `grade(95)` print?

最后两条今天一直要用：**今天的分支和循环，绝大多数写在函数里。那道题的答案在这一段末尾。**

5

第 5 页 · Where we left off

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最后两条。今天讲的条件和循环，在真实的程序里几乎总是写在函数体内——所以「定义不等于运行」和「函数内部一律 `return`」这两条今天会一直用着。

特别是 `return`，今天讲到嵌套循环时会看到它的一个新用途：`break` 只能跳一层，而 `return` 一次跳出所有层。

第一条的内存模型今天用得不多，但第 5 讲讲列表时是核心。

OVERVIEW



I Five parts

- **Conditions** — `if / elif / else`: take a different road depending
- **Loops** — `while` and `for`: do one thing many times
- **Bitwise operators** — work on the bits directly. L2 promised this
- **More on functions** — default arguments, `*args` / `**kwargs`
- **Type hints and indentation** — one good habit, one pit to walk around

五段。前两段是这一讲的主体，也是整门课后面所有内容的地基——第 5 讲之后的每一段代码里都会有循环。

第三段的位运算是第 2 讲那句「一切都是 bit」的兑现，内容不多，但优先级那个坑必须讲。

第四段接上第 3 讲的函数。第五段的类型提示是一个从这一讲开始要养成的习惯，讲义和参考解答从今天起都会带类型标注。

I The conclusion, up front



sequence + branch + repetition = any computable process

Not a figure of speech — a theorem (the structured program theorem). By the end of today you hold every control structure a Turing-complete language has.

8

第 8 页 · The conclusion, up front

这句话值得单独放一页。

你到今天为止只学了「顺序」这一样。加上这一讲的分岔和重复，三样凑齐——而这三样已经足以表达任何可计算的过程。

这是结构化程序定理的结论，1966 年由 Böhm 和 Jacopini 证明。它的意思是：后面你会学到的函数、递归、类、模块，没有一样在**能算什么**这个层面上扩展了这门语言的能力，它们扩展的是**你能把多复杂的程序写清楚**。

换句话说，这一讲之后，剩下的全是关于「怎么把话说明白」。

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.

第 1 到 7 周执行 AI 政策的第 0 级：不允许 AI 生成或补全代码，请关闭编辑器里的补全功能。

可以用 AI 解释概念、查文档、读报错，但不能让它替你写代码。

这一讲尤其如此。循环是需要在脑子里模拟执行的东西——变量每一轮变成什么、什么时候退出。这种能力只能靠自己写、自己跑错、自己盯着改出来。让 AI 补一个循环，你什么也不会得到。

Conditions



A boolean expression evaluates to True or False

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。它们是条件语句唯一认识的东西。

11

第 11 页 · A boolean expression evaluates to True or False

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 讲讲过它们，也讲过 = 是赋值、== 才是比较。这个区分在这一讲会变得更要紧，因为 if 后面写错一个等号，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。



| The order of and / or / not

precedence

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

output

```
True
False
```

not 最紧，然后 and，最后 or。不确定就加括号。

12

第 12 页 · The order 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：True or False 得 True，再 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，这里的建议永远成立：不确定就加括号。a or b and c 读者要停顿一下，a or (b and c) 一眼就明白。加括号不花钱。



Chained comparison: write it as you would in math

one thing Python gets right

```
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，但毫无意义——它先算 $1 < x$ 得 True，再拿 True 和 10 比。

13

第 13 页 · Chained comparison: write it as you would in math

Python 允许把比较串起来，写法和数学一样，含义也和数学一样。

这是 Python 相对 C、Java 的一个实实在在的优点。那些语言里 $1 < x < 10$ 是合法的，但它先算 $1 < x$ 得到一个布尔值，再把这个布尔值和 10 比较——结果毫无意义，而且不报错。

还有一个细节：链式比较里的中间值只求值一次。 $1 < f(x) < 10$ 只调用一次 f ，展开写成 and 的形式就调用两次了。函数开销大的时候，这个差别是实打实的。

| 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.

14

第 14 页 · if / elif / else

三件事要注意：条件后面的冒号不能少；分支体要缩进四个空格；从上往下第一个为真的分支被执行，其余全部跳过。

最后一条最重要，它解释了为什么第二个分支可以写成 `score >= 80`，而不必写成 `80 <= score < 90`——能走到那一行，就说明 `score >= 90` 已经是假的了。

这不只是少打几个字。条件写重复了，两处的边界就可能对不上，而这种错误极难发现。

| Get the order backwards and everything collapses



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写起。

15

第 15 页 · Get the order backwards and everything collapses

先把上一讲结尾那道题结掉：grade(95) 打印的是「及格」，不是「优秀」。

95 分先满足了 `score >= 60`，于是走进第一个分支返回「及格」，后面的 `elif` 根本不会被检查。

这是「第一个为真的分支被执行」这条规则的直接后果，也是初学阶段最常见的逻辑错误之一。它不报错，只有在你拿一个九十分的例子去试的时候才会暴露。

可操作的做法：写多路分支时，**把所有分支的条件按严格程度排个序**，从最严格的写起。分数、等级、税率这类问题都适用。



I The classic exercise: leap years

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

第 16 页 · The classic exercise: leap years

闰年规则有三层，正好用来练多路分支。

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，能就直接返回 True，函数结束；再看 100，再看 4。

这叫早返回。它的好处是每一层判断完就走人，读的人不需要在脑子里同时维护多个未决的分支。

1900 年不是闰年，2000 年是——这两个正好是规则里最特殊的两种情况，测试的时候要专门挑它们。



I The same logic, written two other ways

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。早返回那种最好读 —— 这是选它的唯一理由。

17

第 17 页 · The same logic, written two other ways

左边是嵌套写法。四层缩进，读的人要一直记着自己在哪一层，而且 else 分支离对应的 if 很远。

右边是一行写完。短，但要把三个条件的逻辑关系在脑子里同时拼出来，改起来也容易出错。

三种写法计算结果完全相同，机器不在乎你写哪种。选早返回那种的理由只有一个：人读得最省力。

一条经验：嵌套超过两层，通常意味着可以重构。先看看能不能改成早返回。



| The conditional expression: two short outcomes

A if cond else B

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

output

pass

别嵌套它。写到 `a if p else (b if q else c)` 就该换成 `if-elif-else` 了。

18

第 18 页 · The conditional expression: two short outcomes

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。

注意它是一个表达式，会算出一个值，所以可以直接赋给变量、写进 f-string、当作函数的参数。这是它和 if 语句的区别。

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，就说明该换成正经的 if-elif-else 了。



| match-case: instead of a long elif chain

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

第 19 页 · match-case: instead of a long elif chain

match-case 是 Python 3.10 引入的语法，用来替代一长串 elif。

写法上比 elif 整齐：要比的那个值只写一次，各个分支平铺在下面。

| 把多个值合成一个分支。_ 是通配符，放在最后接住所有没匹配上的情况，作用相当于 else。

但它真正强于 elif 的地方不在这里，在下一页。

| It matches structure, and binds as it goes



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

第 20 页 · It matches structure, and binds as it goes

这一页是 match 和 if 的本质区别。

case (0, y) 不只是在比较。它同时做两件事：检查这个值是不是一个两元素的结构、第一个元素是不是 0；如果是，就把第二个元素绑定到变量 y 上，分支体里可以直接用。

用 if 写同样的逻辑，要先判断类型、再判断长度、再取下标、再赋值，四步。

圆括号那个写法叫元组，第 6 讲讲。这里知道它是「两个值凑一对」就够了。

提醒一句：**别过度使用**。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，看得懂基本用法就够了。

Loops



| while: keep going while the condition holds

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 <= 5` 更新 `n += 1` —— 缺一个就出事。

22

第 22 页 · while: keep going while the condition holds

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走。

三个要素缺一不可：循环变量要先初始化；条件要能变成假；循环体里必须有能改变条件的语句。

这里用了第 3 讲讲的 end 参数，让五个数打在一行里。最后那个空的 print() 是为了补一个换行。



I Forget the update and it never stops

这一页没有输出面板 —— 这段代码永远不会停，真去跑会把这份课件的构建挂住。

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跑了死循环，按 **Ctrl+C** 中断。

23

第 23 页 · Forget the update and it never stops

漏掉更新语句，条件就永远为真，程序会一直打印下去。

这一页是这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时真跑出来的，而这一段跑不完。

死循环不报错，程序看起来在正常工作，只是永远不结束。在自己电脑上遇到了，按 Ctrl+C 中断。

下一页开始讲 for。能用 for 就用 for，很大一部分原因就是 for 不会漏掉更新。



| for: walk through a collection

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**.

24

第 24 页 · for: walk through a collection

for 的意思是「把后面那个东西里的元素逐个取出来，每取一个执行一遍循环体」。

字符串是可以逐个取出的——取出来的是一个一个字符。方括号括起来的那个叫列表，第 5 讲专门讲。

in 后面还可以跟元组、字典、集合，以及很多别的东西。它们背后的共同机制叫可迭代对象，第 13 讲会讲。

注意 for 不需要你写初始化和更新，取完自动结束。这就是它比 while 安全的原因。



I range(): when you need to go round n times

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 讲的切片同一个约定。

25

第 25 页 · range(): when you need to go round n times

range 生成一串整数。一个参数是从 0 到 n-1，两个参数是从 a 到 b-1，三个参数多一个步长。

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

步长可以是负数，倒着走。

注意每一行都套了一个 list()。原因在下一页。



I 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.

26

第 26 页 · 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。因为 `range` 并不真的把数存下来，它只记住起点、终点和步长，需要下一个的时候现算。

所以 `range(1000000)` 几乎不占内存，而一个一百万个数的列表要占几十兆。这个机制叫惰性求值，背后的东西叫生成器，第 13 讲讲。

想看到里面的内容，就用 `list()` 把它展开——这只是为了打印，实际写循环时不需要，直接 `for i in range(...)` 就好。

下面三行是 `range` 最常见的用法：从 1 加到 100。高斯当年心算出来的那个数，这里三行代码。

| for or 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。

27

第 27 页 · for or while

判断标准只有一条：开始循环之前，你知不知道要循环多少次。

知道就用 for。遍历一个集合也属于这一类——集合有多少个元素是确定的。

不知道就用 while。典型场景是读用户输入直到输入合法，或者迭代逼近某个精度——什么时候够了，得算出来才知道。

实践中大多数循环是 for。优先用 for 的理由是它把初始化和更新交给了语言，你少了两个可能出错的地方。



I Where while belongs: the $3n+1$ problem

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
```

至今没人能证明它对所有正整数都会回到 1。

28

第 28 页 · Where while belongs: the $3n+1$ problem

规则很简单：偶数除以 2，奇数乘 3 加 1，一直做下去。

这里必须用 while，因为要走几步事先算不出来——这正是 while 存在的理由。

从 27 出发要走 111 步，中间会冲到 9232 再掉下来。从 6 和 7 出发只要几步。规律看不出来。

这个问题叫考拉兹猜想，1937 年提出，至今没有证明。陶哲轩 2019 年证明了「几乎所有」正整数都会回到 1，但完整的证明还没有。

一个九行的程序，背后是一个未解的数学问题。



| 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

29

第 29 页 · break and continue

左边找 100 的第一个大于 1 的因数。找到就打印并 break，循环立刻结束，后面的数一个都不看。

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

两个都很常用。判断标准：这一轮不要了，用 **continue**；整个循环都不要了，用 **break**。



I In nested loops, break leaves only one level

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, the usual move is to **pull the logic into a function and return** — a return ends the whole function, however deep you are.

这也是「嵌套循环不超过两层」那条规范的来历之一。

30

第 30 页 · In nested loops, break leaves only one level

break 只对它所在的那一层循环有效。左边这段里，内层每次都在 j 等于 1 时跳出，但外层照样跑完三轮。

初学阶段经常有人以为 break 会跳出所有层，然后对着结果百思不得其解。

正确的做法在右边：把嵌套循环抽成一个函数，用 return 跳出。第 3 讲讲过 return 会立刻结束整个函数，不管它嵌在多深。

往年的课件里有一条编程规范：**for 嵌套不超过两层**，尤其是里面有 return 或 break 的时候。超过两层，第三层的逻辑就该抽成函数了。



| Loops have an else too

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.

31

第 31 页 · Loops have an else too

这是 Python 特有的语法，很多人写了几年也不知道。

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

这里 15 在 `i` 等于 3 时找到因数，`break` 出去，`else` 不执行；17 一直找到底也没找到因数，循环正常结束，`else` 执行，打印「是质数」。

它顶掉的是一个 `flag` 变量。不用 `for-else` 的写法要先设 `found = False`，找到就置 `True` 并 `break`，循环后再判断 `flag`。`for-else` 省掉了这个变量。

I But that else is badly named



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

It should have been called nobreak. Because it misleads, plenty of style guides ban it outright. Learn to read it — you will meet it in other people's code. In your own, a found flag is usually clearer.

32

第 32 页 · But that else is badly named

else 这个词在 if 语句里的意思是「条件不成立」，在循环里的意思完全不同，是「没被 break 打断」。同一个关键字两种含义，这是一个公认的设计失误。

Python 之父 Guido 后来也说过，如果重来一次不会这么设计。社区里有人建议改叫 nobreak，但改关键字会破坏兼容性，所以一直没改。

这门课的要求是：**看得懂就行**。读别人的代码会遇到它，读不懂会误判逻辑。但自己写的时候，多一个 found 变量换来读者不困惑，这笔交易是划算的。

Bitwise operators

ONE BIT AT A TIME



| Six bitwise operators

- **& 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**.

34

第 34 页 · Six bitwise operators

第 2 讲说过一切都是 bit。位运算符就是直接操作这些 bit 的工具。

前四个是逻辑运算，逐位进行：与、或、异或、取反。异或的规则是「不同为 1」，它有一个有用的性质——一个数异或同一个数两次会变回自己，加密和交换变量都用得上。

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，道理和十进制里小数点移位相当于乘除 10 一样。

这一段内容不多，但最后那个优先级的坑必须讲。

Work through it 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
```

这里用了第 3 讲的 {值:04b} —— 补零到 4 位，位与位才对得齐。

35

第 35 页 · Work through it bit by bit

把 a 和 b 按二进制对齐打出来，逐列比对就能看清楚每个运算符做了什么。

与：只有第一列两个都是 1，所以结果是 1000。或：有 1 就是 1，结果是 1010。异或：不同才是 1，第二列和第四列不同，结果是 0110。

这一页的关键其实是格式说明 04b。不补零到同样的位数，位和位就对不齐，这种表就没法看。第 3 讲讲过这个写法，这里是它的第一个实际用途。



| Shifting, and one trick worth keeping

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.

36

第 36 页 · Shifting, and one trick worth keeping

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。

下面那个是位运算最常见的实用技巧：n & 1 把除了最低位以外全部清零，只留最低位。二进制里最低位是 1 就是奇数。

它和 n % 2 == 1 完全等价。理论上位运算更快，但在 Python 里差别小到测不出来。

可读性优先，平时用 % 就好。讲它是因为你会在别人的代码里读到，尤其是从 C 移植过来的那些。

| Bitwise precedence trips people at both ends



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：+ - 和 << >> 紧于 & ^ |，而 & ^ | 又紧于比较运算符。

别去背这张表。位运算和别的运算符放在一起，一律加括号。

37

第 37 页 · Bitwise precedence trips people at both ends

算术运算符比位运算符结合得更紧，这一点非常反直觉，因为在多数人的心理模型里位运算是「更底层」的，应该更紧。

8 & 4 + 1 里，加法先算，变成 8 & 5，结果是 0。如果你以为是 (8 & 4) + 1，你会期待 1。

往比较那头看反而符合直觉，n & 1 == 1 确实是先算与。但这一点和 C 语言相反——C 里比较优先级更高，同一行代码在两种语言里含义不同。从 C 移植代码时这是个真实的坑。

结论很简单：不要背优先级表，加括号。

More on functions



| Default arguments: the caller may leave one out

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.**

39

第 39 页 · Default arguments: the caller may leave one out

在参数后面写等号和一个值，就给了它默认值。调用时不传这个参数，就用默认值；传了就用传的。

默认参数的价值在于：一个函数可以同时服务「大多数情况」和「需要定制的情况」，而调用方在大多数情况下不需要关心那些参数。

你已经用过默认参数了：第 3 讲的 `print(..., sep=" ", end="\n")`，那两个就是有默认值的参数。

I Defaulted parameters must 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，连定义都通不过。

40

第 40 页 · Defaulted parameters must come last

有默认值的参数必须排在没默认值的后面，否则连函数定义这一步都过不去，直接语法错误。

原因想一下就明白：如果允许 `def bad(greeting="你好", name)`，那么调用 `bad("张三")` 时，那个 "张三" 该给谁？按位置它该给 `greeting`，可 `name` 又没有默认值不能空着。规则冲突，所以语言直接禁止这种写法。

这是这一讲里少数几个在写的时候就报错的错误之一，改起来很快。

I Never default to a mutable object



错

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

that [] is created once,
and every call shares it

对

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

A default is evaluated once, at definition time. The full story is in L14.

左边那个函数第二次调用时会带着上一次的東西。

41

第 41 页 · Never default to a mutable object

这是 Python 最有名的几个坑之一，几乎每个用 Python 超过一年的人都踩过一次。

问题出在「默认值什么时候创建」。那个空列表是在 def 这一行执行的时候创建的，也就是整个程序里只创建一次。之后每次调用不传 box，用的都是同一个列表——上一次 append 进去的东西还在里面。

正确写法是用 None 当默认值，进函数后再判断、再创建。这样每次调用都是一个新列表。

列表是第 5 讲的内容，完整解释在第 14 讲。现在记住这条结论就够了：默认值只用不可变的東西——数、字符串、None、布尔值。

I Keyword arguments and positional ones



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
```

混用时位置参数必须在前。

42

第 42 页 · Keyword arguments and positional ones

三种调用方式结果完全一样。第一种按位置，第二种全部按名字，第三种混用。

混用时位置参数必须写在前面。这条规则的道理是：Python 得先把按位置对应的那些认领完，剩下的才能按名字分配。如果位置参数夹在关键字参数中间，它该对应第几个就说不清了。

什么时候用哪种，有个经验：参数只有一两个、含义很明显时用位置；参数多、或者有几个类型相同容易搞混时用名字。



| *args gathers the extra positional arguments into a tuple

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.

43

第 43 页 · *args gathers the extra positional arguments into a tuple

参数个数事先不确定的时候用它。一个星号加一个名字，调用时给多少个位置参数，都会被打包成一个元组交给它。

注意 type 打出来是 tuple，元组，第 6 讲讲。现在把它当成一个「不能改的列表」理解就够了。

一个也不传也可以，收到的是空元组，sum 得 0。

你已经用过接受任意多个参数的函数了：print、max、min 都是。



| **kwargs gathers the keyword arguments into a dict

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.

44

第 44 页 · **kwargs gathers the keyword arguments into a dict

两个星号把所有没被认领的关键字参数收成一个字典，键是参数名，值是传进来的值。字典是第 6 讲的内容。

三种参数可以一起用，顺序是固定的：普通参数、然后 *args、最后 **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 这两个名字没有语法意义，换成别的也能跑。但请沿用它们——所有人都这么写，换名字只会让读者多花时间确认你没有别的用意。



Python has no function overloading

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'
```

同名函数，后定义的直接盖掉前面的，不管参数长得一样不一样。

45

第 45 页 · Python has no function overloading

这和 C++、Java 不同。那些语言里同名不同参数的函数可以共存，叫重载。Python 没有这个机制。

在 Python 里，def 本质上是一个赋值：把函数体绑到那个名字上。第二个 def f 就是把 f 重新绑到新函数上，原来那个没人引用了，就消失了。这和 x = 1 之后 x = 2 是一回事。

所以 f(3, 4) 正常，f(3) 报 TypeError，说少了一个参数。

想「根据参数个数做不同的事」，用默认参数或者 *args。

实践中的影响：一个文件里不要有两个同名函数。复制粘贴改一改的时候特别容易犯，而且不报错，只是有一个函数悄悄失效了。

Type hints and indentation



| Annotate the parameters and the return value

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.

47

第 47 页 · Annotate the parameters and the return value

冒号后面写参数的类型，箭头后面写返回值的类型。这个语法从 Python 3.5 开始有。

读法是：这个函数接受一个浮点数，返回一个浮点数。不需要读函数体就知道它的接口长什么样。

下一页有一个重要的限制。



I Python does not enforce them

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'
```

标注不是检查。 类型传错照样跑，直到真正用到它的那一行才报错。

48

第 48 页 · Python does not enforce them

传一个字符串进去，Python 一声不吭地接受了，直到执行到乘法那一行才报错。

这一点必须说清楚：类型标注对 Python 解释器来说基本是注释，它不做任何检查。写了标注不等于类型安全。

报错信息是 can't multiply sequence by non-int of type 'float'——注意它说的是字符串乘法的问题，完全没提你标注了 float。

既然不强制，为什么还要写？下一页。



| So why bother writing them

- **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 silly mistakes 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。

49

第 49 页 · So why bother writing them

第一条，文档不会过期。写在注释里的「这个参数是个整数」，改代码时很容易忘记同步，于是注释开始骗人。类型标注就在参数旁边，改参数时不可能看不见。

第二条，工具能用上它。VS Code 装了 Python 扩展之后，有标注的函数会有更准的补全，传错类型当场标红，不用等运行。

第三条第 15 讲会展开。现在建立一个意识：当你让 AI 写一个函数，最有效的做法不是描述一段话，而是先把签名和类型写出来让它填实现。下一页有例子。



I Three lines beat a hundred words of prose

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会带类型标注。

50

第 50 页 · Three lines beat a hundred words of prose

这三行就是一份完整的规格：两个参数都是整数列表，返回值也是整数列表，那句文档说明了行为。

把这三行给 AI，它填出来的实现基本不会跑偏。换成一段自然语言描述——「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？返回新列表还是改原来的？输入没排序怎么办？

三个点是 Python 的一个合法语句，表示「这里先空着」，常用在写好签名还没写实现的时候。

课程约定：从这一讲开始，讲义和参考解答都会带类型标注。你的作业不强制要求，但强烈建议养成习惯。



I The shapes you will write most

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
```

没有返回值就标 -> None —— 呼应第 3 讲：没有 return 的函数返回 None。

51

第 51 页 · The shapes you will write most

四个例子覆盖了你现在会用到的绝大多数情况。

f2 里的 list[int] 表示「元素是整数的列表」。方括号里写元素类型，这个写法在 dict、tuple 上也一样。

f3 说明默认值和类型标注可以一起写，顺序是参数名、冒号、类型、等号、默认值。

f4 标了 -> 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「没有返回值」就不写，写出来才说明你是有意为之，而不是忘了。



I Back to the opening: four spaces apart

"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。一个打印一次，一个每轮都打印。

52

第 52 页 · Back to the opening: four spaces apart

这就是开场那道题。

左边的 print 顶格写在 for 外面，整个循环跑完才执行一次。右边缩进了四个空格，属于循环体，每一轮都执行，跑了三轮就打三遍。

注意右边输出的顺序：第一行就是「检查完毕」。因为 50 那一轮虽然不及格、不打印「及格」，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。你得自己看出来。

这是 Python 用缩进划分代码块所付出的代价：缩进错了，程序的含义就变了，而且往往不报错。

对应的习惯是：**写完循环，回头确认每一行的缩进层级是你想要的那一层**。尤其是循环体最后几行。



I Never mix 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。从网上抄代码最容易撞上这个。

53

第 53 页 · Never mix tabs and spaces

这是一个纯粹的低级错误，但它值得单独讲一页，因为它在屏幕上看不出来——两种缩进渲染出来可能一模一样。

解决办法只有一条：统一用 4 个空格。VS Code 默认会把 Tab 键转成空格，自己敲一般不会出问题。

出问题的场景基本都是复制粘贴：从网页、PDF 或者别人的代码里复制一段过来，里面混着 Tab。遇到 `TabError`，把那几行整个删掉重新敲一遍最快。

另外，VS Code 可以打开「显示空白字符」，Tab 和空格就能看出区别了。



扫码作答 —— 这一轮五题

- 1 `score = 95` 会打印什么
- 2 这段循环打印什么
- 3 `6 & 3 + 1` 的值
- 4 `*rest` 收到的是什么
- 5 回收开场那道题

今天五个落点各一题。一题一题放，答完一题公布一题。

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

第二轮小测，五道题，把今天几段各收一次。我一题一题地放，每题答完当场公布。

【第 1 题 · 下面这段代码，输入 95 会打印什么？】

考的是「第一个为真的分支被执行」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分支代入，遇到第一个为真的就停，后面的一律不看。

如果这道题的正确率不高，回去把闰年那三种写法再读一遍。

【第 2 题 · 下面这段循环打印什么？】

考的是 `break`、`continue` 和循环 `else` 三者的配合。

解法是老老实实按轮次模拟：每一轮 `i` 是多少、走到哪一句、是跳过本轮还是跳出整个循环。循环的题不能靠猜，要在纸上走一遍。

这个「在纸上模拟执行」的习惯，后面讲递归时会更加重要。

【第 3 题 · `6 & 3 + 1` 的值是多少？】

考的就是上一页那个优先级。加法比按位与结合得紧，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。

如果按 `(6 & 3) + 1` 算，`6 & 3` 是 110 与 011 得 010 也就是 2，加 1 得 3。两个答案都在选项里。

【第 4 题 · `show(1, 2, mode="fast")` 里，`rest` 是什么？】

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

1 给 `first`，2 进 `rest`，`mode` 进 `opts`。所以 `rest` 是只有一个元素的元组。

注意单元素元组的写法是 `(2,)` 带一个逗号，不带逗号的 `(2)` 只是加了括号的整数 2。第 6 讲会讲这个细节。

【第 5 题 · 回到开场那道题 —— 两段的输出差多少？】

回到开场那道题。先公布现在这一轮的结果，再把开课时那一轮放出来。

左边打印两行「及格」加一行「检查完毕」，共三行；右边打印两行「及格」和三行「检查完毕」，共五行。两段都不报错。

如果开课时有不少人认为其中一段会报错，现在没有了，这一段就讲到位了。

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, **and never default to a list or a dict**
- **Type hints are not enforced, and still worth writing.** Indent with 4 spaces

55

第 55 页 · What to take away

八条。第二条和第四条是这一讲最容易在实际写代码时出问题的，而且都不报错。

第三条会跟着你整个学期。从第 5 讲开始，几乎每一段代码里都会有 `for`。

第七条那个可变默认值的坑，第 5 讲讲完列表之后会更有体会，完整解释在第 14 讲。

第八条的类型提示从今天开始是课程约定，本周的实验要求所有函数都带类型标注。



Lab 4 — 七道题，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

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。

56

第 56 页 · Lab 4 — 七道题，条件与循环

A 段七道题。

4-1 考的是分支顺序：多路分支要从最严格的条件开始写，写反了前面那条会把后面的全吃掉，而且不报错。

4-2 不许转成字符串，就是逼你用今天讲的整除和取余把数位一位位剥出来。4-3 和 4-4 都是典型的循环题，4-4 那个式子还有个额外的好处：至今没人证明它对所有正整数都会停下来。

4-5 用到第 3 讲的格式说明，把宽度和对齐结合起来才排得整齐；注意末尾不要多一个换行，判题是按字符比的。

4-7 是下周个人作业 1 的预热——汉诺塔。这周只要求算出最少步数，先把规律找出来，下周再写完整的解。

从这次开始，所有函数都要求写类型提示，这是课程约定。

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。

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——把一批数据装在一起，当成一个东西来传递和处理。循环和列表是天生一对：循环负责逐个处理，列表负责装。

今天讲的 for 在下一讲会立刻变成主力。今天提到但没展开的元组、字典，第 6 讲讲。

另外，今天那个「默认值不要用可变对象」的坑，下一讲学完列表之后你就能真正理解它为什么危险了。