第 3 讲 · 讲义

字符串、变量与输入输出

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。

第 1 页 · Strings, Variables, Input and Output

> 前两讲的程序都是封闭的：数据写死在代码里，算完打印出来。这一讲让程序开始和人打交道：从外面读进来的东西怎么处理，算出来的结果怎么呈现。
>
> 后半段引入函数。函数是这门课第一件真正意义上的工具：在此之前写的是一条条指令，从函数开始，才谈得上组织代码。

An Opening Prediction

add.py

```
a = input("first number: ")
b = input("second number: ")

print(a + b)
```

at run time

```
first number: 3
second number: 4

output?
```

先在心里定一个答案。 用户输入 3 和 4。

> 这是上一讲结尾预告过的那个程序。四行，读两个数，把它们加起来。
>
> 用户输入 3 和 4。在看答案之前，请先自己定一个。
>
> 这个程序几乎每一届学生都会写，也几乎每一届都在这里卡住。

The Actual Output

the same thing, spelled out

```
1  # input() hands back a string. Writing the strings out
2  # directly here changes nothing:
3  a = "3"
4  b = "4"
5
6  print(a + b)
```

output

```
34
```

输出是 **34**，不是 7。**加号没有算错，是两个操作数不是数。**

3

第 3 页 · The Actual Output

> 输出是 34。
>
> 原因上一讲已经讲过：加号作用在字符串上是拼接，不是加法。而 input() 无论输入什么，交回来的都是字符串。输入 3，得到的是 "3" 这个字符串，不是数字 3。
>
> 所以这段代码做的是把 "3" 和 "4" 拼成 "34"。
>
> 课件上这段把 input() 换成了等价的字符串赋值，为的是让每一个输出都来自实际运行。讲义上的同一段代码可以真的弹出输入框，可以自己试。
>
> 这一讲第三段会详细讲 input。现在先看下一页那道题。

随堂小测 · 第 1 轮

扫码作答 · 这一轮一题

1 下面这段代码会输出什么？

这一题现在不公布答案。讲到 print 和 return 那一节再回来。

https://taohuang.info/cs1602/zh/quiz/3

4

第 4 页 · 随堂小测 · 第 1 轮

> 开课先收一道题，不公布答案。
>
> 这一道错了，后果比开场那个加法程序严重得多：那个只是结果奇怪，这个会直接报错，或者在实验平台上一分不得。
>
> 第六段专门讲 print 和 return 的区别，讲到那里会揭晓，顺便把这一轮的分布放出来。
>
> 先自己判断，再听解释，记得会更牢。

RECAP · LECTURE 2

Where Lecture 2 Ended

- Every value has a **type**: `str`, `int`, `float`, `bool` are the four you use most
- **The type decides what an operator means** — + adds numbers and joins strings
- `'0'` (a character) and `0` (an integer) are **two different things**
- `/` always returns a `float`; floats are inexact, so never compare them with ==

第二条和第三条今天就会用到。**开场那个程序输出 34，就是它们的直接后果。**

5

第 5 页 · Where Lecture 2 Ended

> 上一讲的四段：数据类型、进制、字符、数与运算。
>
> 和今天直接相关的是前两条。类型决定运算符的含义，这条规则今天会在两个地方出问题：开场那个加法程序，以及第三段讲 input 的时候。
>
> 第三条的「'0' 和 0 不是一回事」，今天会变成一个更具体的说法：input 拿到的永远是字符串，哪怕用户输入的看起来是个数。
>
> 最后一条今天用不到，但下一讲讲条件判断时会用到。

OVERVIEW

Today

ROADMAP

Outline

- **Variables** — what = actually does, and how to name things
- **Strings** — quotes, joining, indexing, comparing, and what **immutable** means
- **Conversion and input** — `input()` always returns a string. First key point
- **Formatted output** — f-strings: a value inside a sentence, laid out the way you want
- **Functions** — name, parameters, return value
- **print() and return** — the second key point, and the hardest to spot alone
- **Modules** — functions other people already wrote

7

第 7 页 · Outline

> 七段。前四段是把程序和人连起来所需要的语法：变量、字符串、输入转换、格式化输出。后三段是组织代码的第一件工具：函数怎么写、print 和 return 的区别、以及怎么用别人写好的函数。
>
> 两个重点分别在前后两半：第三段的 input 返回字符串，是新生第一个月最高频的错误；第六段的 print 与 return，是最难自己发现的错误，因为屏幕上看起来是对的。
>
> 这两件事在这一讲之后会跟着你整个学期，本周的实验课也各有一道题专门针对它们。

PART 1 OF 7

Variables

Assignment（赋值）

assignment

```
1  x = 10
2  print(x)
3
4  x = 20        # same name, now pointing at a new value
5  print(x)
```

output

```
10
20
```

9

第 9 页 · Assignment（赋值）

> 等号在 Python 里叫赋值，读作「把右边的值绑到左边这个名字上」。
>
> 第一行之后，x 这个名字指向 10。第四行之后，同一个名字改为指向 20，原来那个 10 就不再有名字指向它。
>
> 注意执行顺序是**先算右边，再绑给左边**。这一点在下一页会变得很重要。

Why x = x + 1 Makes Sense

increment

```
1  count = 5
2  count = count + 1     # right side first, then bind
3  print(count)
```

output

```
6
```

数学里 x = x + 1 是个假命题；程序里它完全合法，因为**两边不是同一件事**：右边是求值，左边是绑定。

10

第 10 页 · Why x = x + 1 Makes Sense

> 数学里写 x = x + 1，这个方程无解。程序里它却是最常见的语句。
>
> 区别在于等号的含义不同。数学的等号断言两边相等；赋值的等号是一个动作：先把右边算出来，再把结果绑到左边的名字上。
>
> 按这个顺序读第二行：右边 count 现在是 5，加 1 得 6；然后把 6 绑给 count。于是 count 变成 6。

How to Read the Equals Sign

Read `a = b` as "a becomes b"

Not "a equals b". The habit saves a lot of confusion later, especially on lines like `x = x + 1` and `s = s.upper()`.

11

第 11 页 · How to Read the Equals Sign

> 这是一个很小、但很有效的习惯。
>
> 把赋值语句读成「变成」，方向就清楚了：右边是原料，左边是结果，箭头是从右指向左的。
>
> 后面遇到 s = s.upper() 这种语句时，这个读法直接就是对的：s 变成 s 的大写形式。而如果读成「s 等于 s 的大写」，就会觉得别扭。

MEMORY MODEL

What a Variable Really Is

- Memory is a long row of cells. **Each cell is numbered**, and that number is its **address** 地址
- `x = 10` does two separate things:
  - put 10 somewhere in memory
  - make the name x **point at** that place
- So a variable is **not a box**. It is **a label stuck onto a value**
- `x = 20` does not swap the contents of a box — it **moves the label elsewhere**

记成一个箭头：**名字 → 值**。赋值改的是箭头指向哪里。

第 12 页 · What a Variable Really Is

这一页要建立的模型，后面几讲会反复用到，请多花点时间。

内存可以想象成一长条格子，每个格子有一个编号，这个编号叫地址。数据存在格子里，程序靠地址找到它。

x = 10 做了两件事：把 10 放进某个格子，然后让 x 这个名字指向那个格子。

要紧的是：**变量不是盒子**。很多人把变量想象成一个盒子，里面装着值，赋值就是换掉盒子里的东西。这个比喻在只有数和字符串的时候不会出错，但到了第 5 讲的列表就会把人带偏。

正确的模型是箭头：名字指向值。x = 20 不是把盒子里的 10 换成 20，而是把 x 这个箭头改指到另一个格子。原来那个 10 还在，只是没人指着它了。

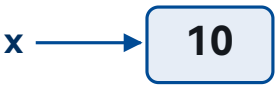
Not a box — a label

✗ a box: x holds 10 inside

after x = 20: swap what is in the box

is 10 **overwritten**?

✓ a label: x points at that 10



one cell in memory

after x = 20: **the arrow moves**, the 10 does not

the 10 is still there, just unreferenced

左边那种画法在数和字符串上不会出错，所以很多人一直用着它，到第 5 讲的列表就会把人带偏。

13

第 13 页 · Not a box — a label

> 把两种模型并排画出来看。
>
> 左边是多数人脑子里那个：x 是一个盒子，里面装着 10；x = 20 就是把盒子里的东西换掉，10 被覆盖了。
>
> 右边是对的：x 是一张贴在值上的标签。x = 10 做了两件事：把 10 放进内存里的某个格子，然后让 x 指向那个格子。x = 20 不是把格子里的 10 改成 20，而是把 x 这个箭头改指到另一个格子；原来那个 10 还在，只是没人指着它了。
>
> 请注意左边那种画法**在数和字符串上不会出错**：两种模型推出来的结论一样。所以很多人一直用着盒子模型，一路用到第 5 讲的列表，然后突然发现改了一个名字另一个也跟着变，那时候盒子模型就解释不了了。
>
> 现在换过来，成本最低。

id(): That Address

id() and is

```
1  x = 10
2  print(id(x))
3
4  y = x            # point y at the same place
5  print(id(y))
6  print(x is y)    # is asks: the same one?
```

output

```
4342711064
4342711064
True
```

`id()` gives the address; `is` asks whether two names point at the **same** thing. **每次运行地址都不一样**，那只是内存分配的结果。

14

第 14 页 · id(): That Address

id() 返回一个名字当前指向的那个值在内存里的地址。这个数每次运行都不同，它是操作系统和解释器分配的结果，所以**不要去记具体的数值，要看的是两个 id 相不相等**。

y = x 之后，两个 id 一样。这说明赋值复制的是箭头，不是值本身。内存里只有一个 10，两个名字都指着它。

is 是一个运算符，问的是「这两个名字是不是指向同一个对象」。它和 == 不同：== 问的是「值相不相等」，is 问的是「是不是同一个」。下一页说明这个区别为什么要紧。

现在这个模型看起来是多余的：数和字符串上，两种理解得到的结论一样。但第 5 讲讲列表时，很多现象都要靠它才能解释。

== and is（相等与同一）

equal in value

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

print(a == b)
```

output

```
True
```

the same object?

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

print(a is b)
```

output

```
False
```

== 比值，is 比身份。 两个内容一样的东西，可以是两个不同的东西。

第 15 页 · == and is（相等与同一）

方括号那个写法叫列表，第 5 讲讲，这里只借它来说明问题，因为数和短字符串上看不出这个区别。

两个列表内容完全一样，所以 == 是 True。但它们是分别造出来的两个东西，在内存里占两块地方，所以 is 是 False。

就像两张一模一样的复印件：内容相同，但不是同一张纸。

实践中的规矩：**比较值就用 ==，只有和 None 比较时用 is。**写 if x is None 是标准写法，因为整个程序里只有一个 None，问「是不是那一个」比问「值相不相等」更准确，也更快。

这一页现在留个印象就够了，第 5 讲会详细讲。

Simultaneous Assignment（同步赋值）

simultaneous assignment

```
a, b = 7, 3
print(a, b)

# the whole right side is evaluated first,
# so swapping needs no temporary variable
a, b = b, a
print(a, b)
```

output

```
7 3
3 7
```

The whole right side is evaluated first, then bound across. So swapping needs no temporary.

16

第 16 页 · Simultaneous Assignment（同步赋值）

> 等号左右都可以是用逗号隔开的多项，数量对上就行。这叫同步赋值。
>
> 关键是执行顺序：**右边先整个算完，然后才一一绑给左边的名字。**所以第六行的 a, b = b, a 里，右边先取到当前的 b 和 a 的值（3 和 7），然后再绑回去，交换就完成了。
>
> 在 C、Java 里交换两个变量需要一个临时变量，Python 这个写法往往是很多人第一次觉得它顺手的地方。
>
> 这个写法后面会反复出现：这一讲后面函数返回多个值时用它接住，第 4 讲的循环里也会见到。

Naming：Rules and Conventions

RULES — break one and it fails

```
start with a letter or _
then letters, digits, _

case matters:
age and Age are two names

no keywords:
if for class return def ...
```

CONVENTIONS — no error, follow them anyway

```
say what it holds:
student_count > n > x

join words with _:
max_score  user_name
this is snake_case

never l or O alone:
too close to 1 and 0
```

规则由 Python 强制，习惯由人强制。**后者对你的成绩影响更大。**

17

第 17 页 · Naming：Rules and Conventions

> 左边这些不遵守，程序跑不起来，当场就知道。右边这些不遵守，程序照常运行，但代码会越来越难改，包括你自己三个月后来改。
>
> snake_case 是 Python 社区的约定，写在 PEP 8 里。别的语言有别的约定，比如 Java 用 camelCase。在一个语言里就用那个语言的约定。
>
> 最后一条是实践经验：小写 l、大写 O 和数字 1、0 在很多等宽字体里几乎分不出来，读代码的人会为此浪费时间。

The 35 Keywords（关键字）

how to check

```
1  import keyword
2
3  print(len(keyword.kwlist))
4  print(keyword.iskeyword("class"))
5  print(keyword.iskeyword("klass"))
```

output

```
35
True
False
```

No need to memorize them — **your editor colors keywords differently**, so a collision is visible at a glance.

> 关键字是 Python 语法自己占用的词，不能拿来当变量名。一共 35 个。
>
> 这一页不要求记住它们，只要求知道有这么一类词，以及知道 keyword 模块可以查。
>
> 实际写代码时，编辑器的语法高亮会把关键字标成特殊颜色。如果给变量起的名字突然变了颜色，那就是撞上关键字了，换一个即可。

Shadowing a Built-in Name

错

```
list = [1, 2, 3]        # no error here

print(list(range(3)))   # TypeError
```

对

```
numbers = [1, 2, 3]     # just pick another name

print(list(range(3)))   # [0, 1, 2]
```

`list`, `str`, `sum`, `max`, `type`, `id` and `input` are built-in functions. Using one as a variable name is **not a syntax error** — Python will not stop you.

报错发生在很后面，而且**错误信息完全指不到问题所在**。

> 关键字撞了会立刻报错，内置函数名撞了不会，后者因此更麻烦。
>
> 左边那段代码，第一行合法，Python 允许你用 list 这个名字。但从那一行开始，list 指向的是这个列表，不再是内置函数。后面某处再调用 list(...)，就成了「拿一个列表当函数用」，报 TypeError。
>
> 问题在于报错的位置可能离出事的位置很远，而且报错信息只会说「list 对象不可调用」，不会提示是在哪一行把它覆盖掉的。
>
> 记住那几个最常撞的名字就够了：list、str、sum、max、min、type、id、input。

Augmented Assignment（复合赋值）

+= -= *= //=

```
1  x = 10
2  x += 3       # same as x = x + 3
3  print(x)
4
5  x -= 5
6  x *= 2
7  x //= 3
8  print(x)
```

output

```
13
5
```

+= -= *= /= //= %= **= all work this way.

> 复合赋值是一种简写。x += 3 完全等价于 x = x + 3。
>
> 第二段跟着算一遍：13 减 5 得 8，乘 2 得 16，整除 3 得 5。
>
> 推荐用复合赋值。理由不在于少打几个字符，而在于**左边的名字只出现一次**。变量改名的时候，写成 total = total + x 需要改两处，漏掉一处就是一个不报错的逻辑错误；写成 total += x 就没有这个风险。

Comments（注释）： Why, Not What

错

```
# add one to x
x += 1
```

对

```
# skip the header row
x += 1
```

Python skips everything after # to the end of the line. Its only reader is a person.

代码已经说清楚「做了什么」。**注释要补的是代码说不出来的那部分。**

> 左边那条注释是复述。代码写着 x += 1，任何人都看得出是加一，这条注释没有提供任何新信息，只是占地方，而且改代码时容易忘记同步，变成误导。
>
> 右边那条解释的是意图：为什么要跳过一行。这是从代码本身读不出来的。
>
> 一个可操作的判断标准：如果把注释和代码对照，注释里出现的词基本都能在代码里找到，那它多半是复述，可以删掉。

Strings

Three kinds of quote

quotes

```
1  a = 'single'
2  b = "double"
3  c = """triple quotes
4  can span lines"""
5
6  print(a, b)
7  print(c)
8
9  # use the other kind on the outside
10 print("he said 'hi'")
11 print('she said "hello"')
```

output

```
single double
triple quotes
can span lines
he said 'hi'
she said "hello"
```

Single and double quotes are **exactly equivalent**. So **use the other kind on the outside** and skip escaping entirely.

23

第 23 页 · Three kinds of quote

Python 里单引号和双引号是完全等价的，选哪一个纯粹看方便。

三引号的字符串可以跨行，里面的换行会原样保留。它常用来写多行文本，也常用来写函数的说明文档，那个用法第 10 讲会讲。

有些语言里单引号表示单个字符、双引号表示字符串，Python 没有这个区分，Python 里没有「字符」这个类型，单个字符就是长度为 1 的字符串。

既然两种引号等价，就可以拿这个自由度来省事：字符串内容里要出现单引号，外层就用双引号，反过来也一样，这样就不用写反斜杠转义。

上一讲讲过转义，写成带反斜杠的形式也对，但实际写代码时优先选不需要转义的那种。

Joining and Repeating

+ and *

```
1  first = "Shanghai"
2  second = "Jiao Tong"
3
4  print(first + " " + second)  # join
5  print("–" * 24)             # repeat
6  print(first * 2)
```

output

```
Shanghai Jiao Tong
------------------------
ShanghaiShanghai
```

From last time: **the same operator means different things on different types.**

> 加号作用在字符串上是拼接，星号是重复。这是上一讲那个结论的直接应用。
>
> 第二行的 "-" * 30 是一个很常用的写法，用来打一条分隔线，比手敲三十个减号可靠。
>
> 注意拼接不会自动加空格，第一行里那个 " " 是自己补上去的。这一点和 print 的多参数不同：print(a, b) 会自动补一个空格，而 a + b 不会。

String Plus Number

错

```
print("Age: " + 18)

# TypeError:
# can only concatenate str
# (not "int") to str
```

对

```
print("Age: " + str(18))

print("Age:", 18)

print(f"Age: {18}")
# the f-string comes in part 4
```

Both sides of + must be the same kind. All three fixes work; the f-string is the one this course uses.

这条报错信息值得读一遍：**它把两边的类型都告诉你了。**

> 加号要求两边类型一致，字符串加整数没有意义，所以报 TypeError。
>
> 这条报错信息写得很清楚：只能把 str 拼到 str 上，不能是 int。养成读报错信息的习惯，绝大多数报错都直接写着原因。
>
> 右边给了三种改法。第一种手工转换，第二种交给 print 处理，第三种是 f-string，这一讲第四段会讲，也是以后最常用的写法。

Length, index, slice

picking characters

```
s = "Python"
#    P  y  t  h  o  n
#    0  1  2  3  4  5

print(len(s))       # how many characters
print(s[0])         # the first; indices start at 0
print(s[-1])        # the last
print(s[2:5])       # 2, 3, 4 — 5 not included
```

output

```
6
P
n
tho
```

Indices start at 0; negatives count from the right. A slice **includes the start and excludes the end**.

26

第 26 页 · Length, index, slice

这四个是字符串上最常用的操作。

下标从 0 开始，所以 s[0] 是第一个字符 P。负数下标从右边数，s[-1] 是最后一个字符 n。

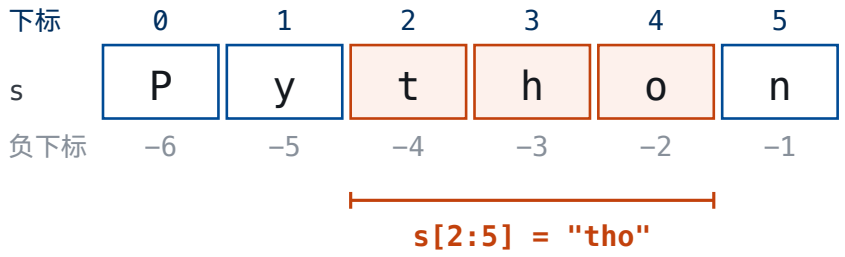
s[2:5] 取下标 2、3、4，得到 t、h、o 三个字符，**不包含下标 5**。这一位最容易数错。

含头不含尾看起来别扭，但有两个很实用的好处：一是切片的长度正好等于两个下标之差，5 减 2 等于 3；二是 s[:2] 和 s[2:] 正好把字符串切成不重不漏的两段。如果改成两端都含，这两条性质都不成立。

第 5 讲讲列表切片时，会用一个可以拖动的游标再演示一遍。

The Index Ruler

Two sets of numbers for the same six characters — and a slice that stops **before** where it says.



切到 **5 为止，不含 5** 本身。所以取到 3 个字符，正好是两个下标之差。

切片的长度正好是两个下标之差。这条比背规则好使。

27

第 27 页 · The Index Ruler

把上一页那行代码画出来。

格子里是六个字符。上面一排是正下标，从 0 开始数；下面一排是负下标，从 -1 开始倒着数。**同一个字符有两个编号**，比如最后那个 n 既是 5 也是 -1。

橙色那一段是 s[2:5]。注意右边的括号停在下标 5 的**左边** —— 取的是 2、3、4 三个字符，5 那一格没有被圈进去。

「含头不含尾」听起来别扭，但它换来两条很实用的性质：**一是切片的长度正好等于两个下标之差**，5 减 2 等于 3，不用数；二是 s[:2] 和 s[2:] 正好把字符串切成不重不漏的两段。如果改成两端都含，这两条都不成立。

第 5 讲讲列表切片时会用一个可以拖动的游标再演示一遍，规则和这里完全一样。

Dictionary Order（字典序）

comparing

```
1  print("apple" < "banana")     # 'a' < 'b'
2  print("abc" < "abd")          # third char decides
3  print("abc" < "abcd")         # shorter prefix first
4  print("Zebra" < "apple")      # look at this one
```

output

```
True
True
True
True
```

Character by character, comparing **ASCII codes**. The first difference decides it.

28

第 28 页 · Dictionary Order（字典序）

> 字符串比较的规则是字典序：从左往右逐个字符比较，遇到第一个不同的字符就决定结果，后面的不看了。
>
> 前三行符合直觉。第四行需要解释：Zebra 小于 apple，也就是说大写的 Z 排在小写的 a 前面。下一页说明为什么。
>
> 如果一个字符串是另一个的前缀，短的排前面，就像字典里abc 排在 abcd 前面一样。

Capitals Sort Before Lowercase

why

```
1  print(ord("Z"), ord("a"))
2
3  print("Zebra" < "apple")
4
5  # to ignore case, fold both sides first
6  print("Zebra".lower() < "apple".lower())
```

output

```
90 97
True
False
```

`'Z'` 是 **90**，`'a'` 是 **97**。比的是编号，不是字母表位置。

29

第 29 页 · Capitals Sort Before Lowercase

> 上一讲讲过 ord()：它给出一个字符的 ASCII 编号。大写字母占 65 到 90，小写字母占 97 到 122。
>
> 所以任何大写字母的编号都小于任何小写字母，字符串比较又是按编号比的，于是 Zebra 小于 apple。
>
> 这一点在排序名单时会直接出问题：一份大小写混排的名单按默认规则排序，所有大写开头的会聚到前面。解决办法是排序前先统一转换大小写，也就是最后一行那种写法。

Strings Are Immutable（不可变）

try to change one character

```
1  s = "hello"
2  s[0] = "H"
```

traceback

```
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    s[0] = "H"
    ~^^^
TypeError: 'str' object does not support item assignment
```

The message says it plainly: **a str does not support item assignment.**

> 字符串创建之后就不能改了，这叫不可变。想改只能造一个新的。
>
> 这不是 Python 的限制，很多语言都这么设计。好处是字符串可以放心地到处共享：既然谁也改不了它，就不必担心它在别处被改掉。
>
> 下一页给出正确的写法。

Changing One Means Building a New One

upper()

```
1  s = "hello"
2  s = s.upper()        # new string, then rebind s
3  print(s)
4
5  t = "hello"
6  t.upper()            # computed, but nobody caught it
7  print(t)
```

output

```
HELLO
hello
```

`s.upper()` **does not change s**. It returns a new string, and if you do not catch it, nothing happened.

31

第 31 页 · Changing One Means Building a New One

> upper() 返回一个全新的大写字符串，原来那个 s 一个字符也没变。所以必须把返回值接住，也就是第二行那个赋值。
>
> 下面三行演示不接住的后果：t.upper() 确实算出了 HELLO，但没有赋给任何名字，算完就丢了，t 还是原来的 hello。
>
> 这是初学阶段很常见的一个错误，而且不报错。凡是「返回新值」的操作都有这个问题，见到就要想一下有没有接住。
>
> 字符串的完整方法在第 6 讲集中讲，这里够用了。

PART 3 OF 7

Conversion and Input

Three conversion functions

int() float() str()

```
1  print(int("42") + 1)        # string -> integer
2  print(float("3.14") * 2)    # string -> float
3  print(str(42) + "!")        # number -> string
```

output

```
43
6.28
42!
```

With `type()` from last time, these four are your entire toolkit for type problems.

33

第 33 页 · Three conversion functions

> 上一讲讲了 type()，它给出一个值的类型。这一页的三个函数负责在类型之间转换。
>
> int() 把字符串或浮点数变成整数，float() 变成浮点数，str() 把数变成字符串。
>
> 第三行是解决上一段那个 TypeError 的手工办法：既然加号两边要同类，就把数先转成字符串。

int() Truncates, It Does Not Round

truncate vs round

```
print(int(3.99))
print(int(-3.99))

print(round(3.99))
print(round(-3.99))
```

output

```
3
-3
4
-4
```

`int()` always cuts **toward zero**. For rounding, use `round()`.

int(3.99) 是 3，不是 4。int() 做的是截断：直接把小数部分砍掉。

负数要特别注意方向：int(-3.99) 是 -3，不是 -4。截断是朝零的方向，不是朝小的方向。这一点和上一讲讲的 // 正好相反，// 是向下取整，-3.99 // 1 会得到 -4。

需要四舍五入就用 round()。但 round 有一个反直觉的细节，看下一页。

round()：Banker's Rounding（银行家舍入）

exactly halfway

```
1  print(round(0.5))
2  print(round(1.5))
3  print(round(2.5))
4  print(round(3.5))
```

output

```
0
2
2
4
```

正好一半时往偶数靠。 不是「四舍五入」，是「四舍六入五取偶」。

35

第 35 页 · round()：Banker's Rounding（银行家舍入）

> round(0.5) 是 0，round(2.5) 是 2。看起来像 bug，其实是有意设计的。
>
> 如果一律「逢五进一」，那么大批数据四舍五入之后，总和会系统性地偏大，因为进位的机会比舍去多一个。往偶数靠可以让进位和舍去的机会各占一半，累计偏差趋近于零。
>
> 这个规则叫银行家舍入，是 IEEE 754 规定的默认舍入方式，在金融和统计场景里是标准做法。
>
> 记住它主要是为了不在看到 round(2.5) 得 2 时怀疑机器坏了。

A Failed Conversion

int("3.14")

```
1  print(int("3.14"))
```

traceback

```
Traceback (most recent call last):
  File "example.py", line 1, in <module>
    print(int("3.14"))
          ^^^^^^^^^^^
ValueError: invalid literal for int() with base 10: '3.14'
```

`int()` **只认整数形式的字符串**，小数点它不处理。

> int("abc") 会失败，这个好理解，abc 不是一个数。
>
> 但 int("3.14") 也失败，这个经常出乎意料，3.14 明明是个数。原因是 int() 只接受整数形式的字符串。
>
> 要把 "3.14" 变成整数，得分两步：先 float("3.14") 得到 3.14，再 int(3.14) 得到 3。写成一行是 int(float("3.14"))。
>
> 两种情况报的都是 ValueError，并且把那个不合法的字符串原样打出来了，调试时很有用。

input() Always Returns a String

Whatever you type, what comes back is a str

Type 18 and you get "18", not 18. This is the single most common mistake of a first-year's first month — and why the program we opened with printed 34.

> 这是这一讲的第一个重点，请单独记住。
>
> input() 的工作是把用户输入的一行字原样交回来。它不做任何猜测，也不做任何转换。输入 18，你拿到的是两个字符 '1' 和 '8' 组成的字符串。
>
> 上一讲讲过 '0' 和 0 不是一回事，这里就是它的具体表现。

Why This Does Not Triple the Number

wanted: triple

```
# input("a number: ") returns a string.
# Writing it out directly changes nothing:
n = "5"

print(n * 3)
```

output

```
555
```

字符串乘以整数是**重复**。这就是第二段那条规则。

38

> 想要 15，得到 555。
>
> 原因还是那条：星号作用在字符串上是重复，不是乘法。n 是字符串 "5"，重复三遍就是 "555"。
>
> 这个错误的麻烦之处在于它**不报错**。程序照常跑完，给你一个看起来像数字的结果，你可能要过很久才发现不对。
>
> 开场那个加法程序输出 34，和这里是同一个原因。

Converting at the Right Moment

int(input(...))

```
# n = int(input("a number: "))
# is the same as:
n = int("5")

print(n * 3)
print(type(n))
```

output

```
15
<class 'int'>
```

`int(input(...))` 是标准写法：**读进来，立刻转成你要的类型。**

> 把 input 的结果立刻用 int 包一层，之后 n 就是一个真正的整数了。
>
> 推荐养成的习惯是：**在读进来的那一行就转换完**，不要先存成字符串、隔十几行再想起来转。后者的代码里会同时存在同一个值的字符串版本和数字版本，很容易用错。
>
> 要小数就用 float(input(...))。
>
> 这个写法有一个缺陷：如果用户输入的不是数字，程序会直接崩。现在先这么写，第 11 讲学了异常处理之后，你会知道怎么优雅地应对。

eval(): a String as an Expression

eval

```
1  print(eval("1 + 2 * 3"))
2  print(eval("2 ** 10"))
3
4  a, b = 1, 2
5  print(eval("a + b"))
```

output

```
7
1024
3
```

It even resolves variables. Very convenient — the next slide says why to avoid it.

> eval 把一个字符串当作 Python 表达式来求值，连当前的变量都能用上。
>
> 它看起来很方便：eval(input()) 一句话就能读进一个数，而且用户输入 3+4 也能直接算出来，不用管类型。
>
> 往年的课件里 eval 用得比较多。这门课要求你知道它存在、看得懂别人的代码，但下一页会说明为什么不推荐用它。

eval() on Untrusted Input

错

```
x = eval(input())

# if the user types code
# instead of a number,
# that code runs
```

对

```
x = int(input())

# bad input raises an error
# instead of running
```

`eval()` executes **any** Python code in that string. When the input comes from a user, a file or the network, whoever wrote it can delete files or read your secrets. The attack has a name: **code injection** 代码注入.

需要一个数，就用 `int()` / `float()`。 这条规则没有例外。

> eval 的危险在于它的能力远远超出「读一个数」这个需求。
>
> int() 只能得到一个整数，输入不合法就报错，能造成的最坏后果是程序崩溃。eval() 能执行任何代码，最坏后果没有上限。
>
> 安全领域有一条基本原则：**给一段代码的权限，应该正好等于它完成任务所需，不多给。**用 int() 而不用 eval()，就是这条原则的一次具体应用。
>
> 在自己的小程序里用 eval 不会有什么后果，但习惯要现在养成，因为以后处理的输入不一定来自你自己。

Formatted Output

Concatenation, and Why It Is Clumsy

written with +

```
1  name = "Alice"
2  score = 87.5
3
4  print(name + " scored "
5        + str(score) + " points")
```

output

```
Alice scored 87.5 points
```

Every value needs its own `str()`, and you count the spaces yourself. **The sentence is in pieces.**

43

第 43 页 · Concatenation, and Why It Is Clumsy

> 这段代码是对的，但有三个问题。
>
> 第一，score 是浮点数，必须手工 str() 一次，忘了就报 TypeError。第二，空格要自己算准放在哪个引号里面。第三，也是最要紧的：你要输出的那句话被加号切成了四段，读代码的人得在脑子里把它拼回去才知道最终长什么样。
>
> 变量一多，这种写法就没法维护了。下一页是正确的做法。

f-string（格式化字符串）

f-string

```
1  name = "Alice"
2  score = 87.5
3
4  print(f"{name} scored {score}")
```

output

```
Alice scored 87.5
```

An **f** before the quote, values inside {}. **The sentence stays whole.**

44

第 44 页 · f-string（格式化字符串）

在引号前面加一个字母 f，这个字符串就变成了 f-string，里面的花括号会被替换成对应变量的值。

和上一页对比：不需要 str()，不需要数空格，而且那句话在代码里是完整的一句，所见即所得。

f 是 formatted 的意思，这个语法是 Python 3.6 引入的。**这门课统一用 f-string**，请从现在开始就用它。

Any Expression Inside the Braces

not only variables

```
1  a, b = 7, 3
2
3  print(f"{a} + {b} = {a + b}")
4  print(f"{a} / {b} = {a / b}")
5  print(f"mean: {(87 + 92 + 65) / 3}")
```

output

```
7 + 3 = 10
7 / 3 = 2.3333333333333335
mean: 81.33333333333333
```

But those last two decimals run on. **Next: controlling the format.**

45

第 45 页 · Any Expression Inside the Braces

> 花括号里不限于变量名，任何表达式都可以，Python 会先算出来再填进去。
>
> 第一行的 a、b 和 a + b 三个花括号，正好把一个算式和它的结果排在一起，这是调试时很常用的写法。
>
> 但后两行暴露了一个问题：除法的结果有十几位小数，直接打出来很难看，也不是给人读的样子。下一页讲怎么控制。

Decimal Places

{value:.Nf}

```
pi = 3.14159265

print(f"{pi:.2f}")        # two decimals
print(f"{pi:.4f}")
print(f"{pi:.0f}")        # none

print(f"{1/3:.2%}")       # as a percentage
```

output

```
3.14
3.1416
3
33.33%
```

After the colon comes the **format spec**: `.2f` means "fixed point, two decimals".

> 花括号的完整形式是 {值:格式}，冒号后面写格式说明。
>
> .2f 里的 f 表示 fixed-point，定点小数；点后面的数字是保留几位。注意它做的是四舍五入，不是截断。
>
> 最后一行的 .2% 会把值乘以 100 再加百分号，算及格率、正确率的时候很方便，不用自己乘。
>
> 格式说明的完整语法很长，这门课要求掌握的就是这一页和下两页的内容。

Width and Alignment

{value:width}

```
pi = 3.14159

print(f"[{pi:10.2f}]")     # width 10, right
print(f"[{pi:<10.2f}]")    # left
print(f"[{pi:^10.2f}]")    # centered
print(f"[{pi:010.2f}]")    # zero-padded
```

output

```
[      3.14]
[3.14      ]
[   3.14   ]
[0000003.14]
```

< left > right ^ center. **Numbers default to right, text to left.**

47

第 47 页 · Width and Alignment

> 冒号后面的数字是总宽度，不足的部分用空格补齐。两边加了方括号，这样能看清楚空格补在哪一侧。
>
> 三个符号分别是左对齐、右对齐、居中。数字默认右对齐，字符串默认左对齐，这个默认值符合大多数场景。
>
> 宽度前面加一个 0 表示用零补齐而不是空格，打印时间、编号这类固定位数的东西时会用到。

A Table That Lines Up

width + alignment

```
1  print(f"{'Name':<8}{'Score':>8}")
2  print("-" * 16)
3  print(f"{'Alice':<8}{87.5:>8.1f}")
4  print(f"{'Bob':<8}{92.0:>8.1f}")
5  print(f"{'Carol':<8}{65.25:>8.1f}")
```

output

```
Name         Score
----------------
Alice         87.5
Bob           92.0
Carol         65.2
```

Text column left, number column right — that is all a table is.

> 姓名靠左、数字靠右，是表格排版的通行做法：数字靠右能让同样位数的部分对齐，一眼看得出大小。
>
> 宽度要留够。名字那一列宽 8，如果有人的名字超过 8 个字符，格式说明不会截断它，那一行就会被撑开、和别人对不齐。
>
> 还有一个坑：宽度算的是**字符个数**，不是显示宽度。中文字在等宽字体里占两格宽，用同样的宽度排中文名会看起来偏窄。处理中文表格时要自己按显示宽度算，或者干脆用制表符分隔。

Printing in another base

```{value:b/o/x}
n = 255

print(f"{n:b}")          # binary
print(f"{n:o}")          # octal
print(f"{n:x}")          # hexadecimal

m = 5
print(f"{m:08b}")        # zero-padded to 8 digits
```

output

```
11111111
377
ff
00000101
```

`bin()` / `oct()` / `hex()` from last time keep the `0b`, `0o`, `0x` prefix. These **do not**.

> 格式说明里写 b、o、x 就按对应进制输出。
>
> 和上一讲的 bin()、oct()、hex() 相比，区别是这里不带前缀，而且可以配合宽度和补零。
>
> 最后一行的 08b 是很实用的写法：补零到 8 位，一个字节的每一位都对齐，多行排在一起就能看出位模式。第 12 讲讲编码时会用到。

The {expr=} Form, for Debugging

prints its own label

```
1  x = 42
2  name = "Alice"
3
4  print(f"{x=}")
5  print(f"{name=}")
6  print(f"{x * 2=}")
```

output

```
x=42
name='Alice'
x * 2=84
```

The expression is echoed **verbatim**, then an equals sign, then its value.

> 在花括号里的表达式后面加一个等号，Python 会把表达式本身和它的值一起打出来。
>
> 这是调试时最省事的写法。以前要写 print("x =", x)，变量一改名就得改两处；现在写 print(f"{x=}") 就行了。
>
> 注意 name 的值打出来是带引号的 '张三'，因为这个写法用的是 repr 而不是 str，这样能看清楚值的类型，字符串的 42 和整数的 42 打出来不一样。这正是调试时想要的。

Two Older Spellings

three generations

```python
name, score = "Alice", 87.5

# first generation: C-style %
print("%s scored %.1f" % (name, score))

# second generation: str.format()
print("{} scored {:.1f}".format(name, score))

# third generation: f-string -- use this one
print(f"{name} scored {score:.1f}")
```

output

```
Alice scored 87.5
Alice scored 87.5
Alice scored 87.5
```

三行输出完全一样。**前两代要求你看得懂，自己写的时候用第三代。**

Python 有过三代格式化语法，f-string 是最新的一代。前两代在旧代码和教科书里还大量存在，你需要读得懂。

第一代来自 C 语言，%s 表示字符串、%d 表示整数、%f 表示浮点数。它的问题是占位符和实际类型不一致时行为很怪，历史上这类格式串引发过大量安全漏洞。

第二代用花括号占位，类型自动推导，比第一代安全，但变量都堆在末尾的括号里，占位符多了就要数第几个对应第几个。

第三代把变量写回它出现的位置，这是它取代前两代的原因。

print(): sep and end

sep / end

```
print("a", "b", "c")          # space
print("a", "b", "c", sep="")  # none
print("2026", "09", "14", sep="-")

print("no newline", end="")
print(" <- joined on")
```

output

```
a b c
abc
2026-09-14
no newline <- joined on
```

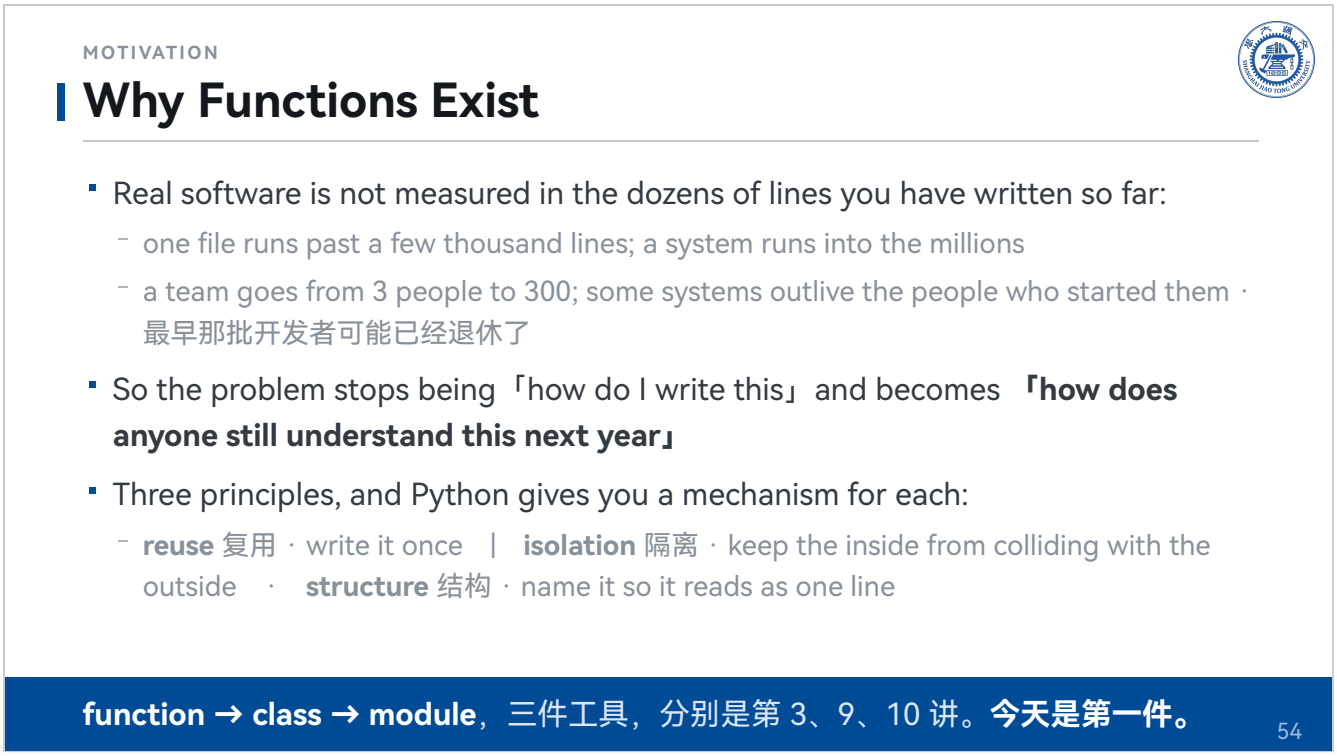
sep goes **between values**, end goes **at the end of the line**. A space and a newline by default.

52

第 52 页 · print(): sep and end

> print 可以接受任意多个值，默认用一个空格隔开，最后自动补一个换行。这两个默认值都可以改。
>
> sep 改的是值之间的分隔符，第三行用它拼日期。
>
> end 改的是行尾。设成空字符串就不换行，下一次 print 会接着往后打。第 4 讲讲循环时，这个写法可以在同一行里打出一串结果。
>
> 上一讲讲过 print 的行为模型，这一页是它的两个参数。

Functions



第 54 页 · Why Functions Exist

到目前为止写的程序都只有几十行，还感觉不到问题。但真实的软件动辄几万到几千万行，没有组织手段是写不出来的。

而且规模不只是行数，还有**人和时间**。一个项目从三个人做到三百个人，中途有人离职、有新人进来；有些系统跨度二十年，最早那批开发者已经退休了，代码还在跑。于是问题从「怎么写出来」变成了「明年还有没有人看得懂」。

Python 给了三件工具：函数、类、模块，分别是第 3、9、10 讲。今天是第一件。

函数解决的是三件事：复用、隔离、命名。复用是同一段逻辑只写一次；隔离是函数内部的事情不会影响外面；命名是给一段逻辑起个名字，读代码的人看到名字就知道它干什么，不必读进去。

第三件常被低估：一个好名字能让读者不必读进去就知道那几十行在做什么。

其实已经在用别人写好的函数了：print、len、int、input，以及上一讲的 ord 和 chr。现在开始写自己的。

The Three Parts of a Function

your first function

```python
# def, name, parameter, return value
def area_of_circle(r):
    return 3.14159 * r * r

print(area_of_circle(1))
print(area_of_circle(2.5))
```

output

```
3.14159
19.6349375
```

`def` 定义　`r` 是**参数** parameter（输入）　`return` 后面是**返回值** return value（输出）。 55

第 55 页 · The Three Parts of a Function

def 是定义函数的关键字，来自 define。

area_of_circle 是函数名，命名规则和变量一样，习惯上用动词或者能说清楚它算什么的名词短语。

括号里的 r 是参数，是这个函数的输入。return 后面是返回值，是这个函数的输出。

第一行末尾的冒号不能少，函数体必须缩进。下一页专门讲缩进。

定义好之后，写函数名加括号就能调用，括号里给出参数的实际值。同一个函数可以调用任意多次，每次给不同的参数。

Indentation（缩进）Decides What Is Inside

indentation

```
1  def greet(name):
2      print("greeting")  # inside
3      print(f"Hi, {name}")
4
5  print("no indent, outside")
6  greet("Alice")
```

output

```
no indent, outside
greeting
Hi, Alice
```

Consecutive lines at the same indent form one block. Four spaces per level, by convention.

56

Python 用缩进划分代码块，这一点和大多数语言不同，它们用花括号，缩进只是排版习惯。Python 里缩进是语法的一部分。

注意输出的顺序：没缩进的那一行先打出来。因为函数定义只是登记，不会执行；程序真正从上往下执行的第一条语句是那个 print，然后才是 greet 的调用。这就是下一页的内容。

缩进用 4 个空格，这是 PEP 8 的约定。不要混用 Tab 和空格，看起来一样，Python 却会报错。VS Code 默认把 Tab 转成空格，一般不用操心。

Defining Is Not Running

define vs call

```
1  def say_hi():
2      print("hi")
3
4  print("program starts")
5  # defined above, but not run even once
6  say_hi()          # now it runs
7  say_hi()          # as often as you like
```

output

```
program starts
hi
hi
```

A def registers the code; it does not run it. The name plus parentheses is what runs it.

57

第 57 页 · Defining Is Not Running

这一页要建立的是执行顺序的概念。

Python 从上往下读这个文件。读到 def 时，它把函数体记下来，绑到 say_hi 这个名字上，然后**跳过整个函数体**继续往下。

所以最先打出来的是「程序开始」。之后两次调用，各执行一遍函数体。

这个区分很重要：函数体里的代码在定义的那一刻一行也没跑。如果函数体里有语法之外的错误，要等到第一次调用才会暴露。

The Definition Must Come First

wrong order

```
1  say_bye()
2
3  def say_bye():
4      print("bye")
```

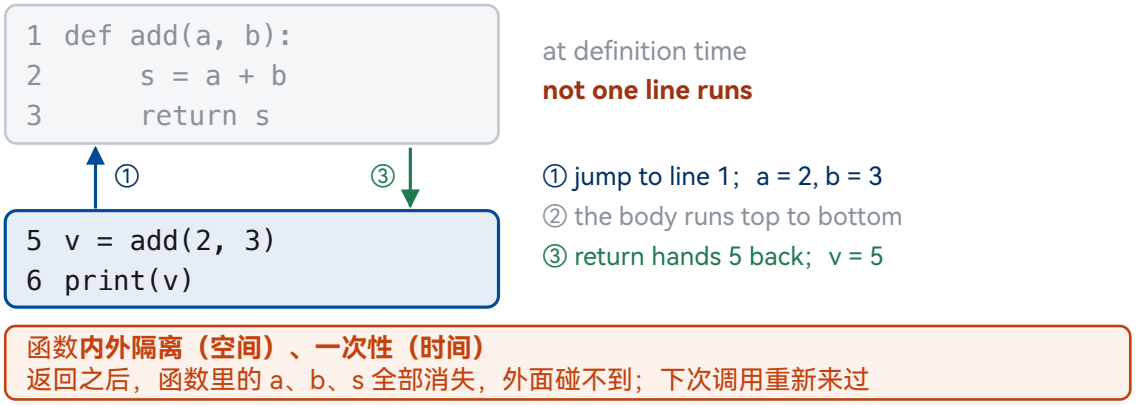
traceback

```
Traceback (most recent call last):
  File "example.py", line 1, in <module>
    say_bye()
    ^^^^^^^
NameError: name 'say_bye' is not defined
```

报错是 **NameError**：执行到第一行时，`say_bye` 这个名字还不存在。

> 既然 Python 是从上往下执行的，那么调用写在定义前面就会出问题。
>
> 执行到第一行时，def 那几行还没被读到，say_bye 这个名字还没有绑定，于是报 NameError，说这个名字没定义。
>
> 实践中的习惯是：**函数定义集中放在文件靠前的位置**，主流程放在后面。这样既满足了顺序要求，也方便别人打开文件先看到有哪些函数。

How a Call Jumps（调用的控制流）



定义只是把它记下来，调用才真的跑。 跳过去、跑一遍、带着返回值跳回来。这三步要能在脑子里过一遍。

59

第 59 页 · How a Call Jumps（调用的控制流）

> 把前面两页合起来画一遍，这是这一讲最该在脑子里形成图像的一页。
>
> 上面那块是定义。程序执行到这里的时候，**一行都不会跑**，它只是把「add 这个名字对应这么一段代码」记下来。很多人第一次写函数，在定义里写了 print 却看不到输出，就是这个原因。
>
> 下面那块才是调用。执行到第 5 行，控制流跳到第 1 行，参数按顺序对应：a 拿到 2，b 拿到 3。然后函数体从上到下跑，遇到 return 就带着那个值跳回第 5 行，v 拿到 5。
>
> 最底下那条是往年课件里的说法：函数**内外隔离（空间）、一次性（时间）**。空间上，函数里的 a、b、s 外面碰不到；时间上，函数一返回它们就全消失了，下次调用重新来过。
>
> 这两句话合起来，正好解释了下一段要讲的隔离。

Matching by Position

positional arguments

```
1  def rectangle_area(width, height):
2      return width * height
3
4  print(rectangle_area(3, 4))      # width=3, height=4
5  print(rectangle_area(4, 3))      # width=4, height=3
```

output

```
12
12
```

Names in the def are **parameters** 形参; values at the call are **arguments** 实参. **They pair up in order.**

> 调用时括号里的值会按顺序绑给定义时的参数名：第一个给 width，第二个给 height。
>
> 这一页两次调用结果一样，因为乘法可交换。但绝大多数函数不是这样：顺序传错了就是另一个意思，而且往往不报错。
>
> 参数的个数必须对得上。少给一个或多给一个都会报 TypeError，报错信息会告诉你期望几个、实际给了几个。

Matching by Name（关键字实参）

keyword arguments

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

output

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

参数一多，**按名字给更不容易出错**，读代码的人也不用回去数顺序。

> 调用时写成 参数名=值 的形式，就不用管顺序了。这叫关键字参数。
>
> 两次调用结果完全一样。
>
> 什么时候用哪种：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。
>
> 其实已经用过关键字参数了：上一段的 print(..., sep="-") 就是。

Inside and Outside Are Sealed Off

local variables

```
1  def compute():
2      result = 42          # a local variable
3      return result
4
5  print(compute())
6  print(result)            # there is no result out here
```

traceback

```
Traceback (most recent call last):
  File "example.py", line 6, in <module>
    print(result)            # there is no result out here
          ^^^^^^
NameError: name 'result' is not defined
```

函数里定义的变量叫**局部变量** local variable，函数一结束就消失。

> 第五行正常输出 42，第六行报 NameError。
>
> result 是在函数体里创建的，它只在这次调用期间存在，函数返回之后就销毁了。外面从来看不见它。
>
> 这个隔离是有意设计的，而且是好事。下一页说明为什么。

Why Isolation Helps

same name, no interference

```
1  def f():
2      x = "the x inside"
3      return x
4
5  x = "the x outside"
6  print(f())
7  print(x)        # untouched by the call
```

output

```
the x inside
the x outside
```

The two xs are two different variables. **You never have to check what names the outside already uses.**

63

第 63 页 · Why Isolation Helps

> 函数里的 x 和外面的 x 同名，但互不相干。
>
> 这种隔离在写大程序时才看得出好处：如果没有隔离，写一个函数之前得先确认自己用的每个变量名在整个程序里都没被占用过，这是不可能做到的。
>
> 有了隔离，函数就成了一个封闭的小世界，只通过参数接收输入、通过返回值交出输出，别的一概不来往。函数能被放心复用，靠的也是这一点。
>
> 完整的作用域规则叫 LEGB，在第 14 讲讲。现在记住「函数内外互不干扰」就够了。

What return Does

hand back a value, and stop

```
1  def check(n):
2      if n > 0:
3          return "positive"
4      # only if the if above did not return
5      return "not positive"
6
7  print(check(5))
8  print(check(-3))
```

output

```
positive
not positive
```

It **hands the value back** and **ends the function**, both at once.

return 的第一个作用是把值交回调用它的地方，第二个作用是立刻结束这个函数，后面的语句不再执行。

这段代码里，如果 n 大于 0，第三行的 return 就结束了函数，第四行根本轮不到。只有 n 不大于 0 时才会走到第四行。

if 是下一讲的内容，这里只要读懂意思就行。

利用「立即结束」这个性质，可以写出很清爽的分支结构：满足某个条件就直接返回，不满足才继续往下。

Nothing After return Ever Runs

unreachable code

```
1  def demo():
2      print("this line runs")
3      return 1
4      print("this line never runs")
5
6  print(demo())
```

output

```
this line runs
1
```

Line four is **unreachable code** 不可达代码. Editors usually gray it out for you.

> return 一执行，函数立刻结束，它后面的语句是死代码。
>
> 这种情况一般是写错了，比如本来想在返回前做点收尾工作，结果顺序放反了。
>
> 好的编辑器会把不可达的代码显示成灰色，或者给一个警告。看到代码莫名其妙变灰，先检查上面是不是有 return。

Returning Several Values

returning two values

```
1  def min_max(a, b, c):
2      return min(a, b, c), max(a, b, c)
3
4  low, high = min_max(3, 9, 5)
5  print(low, high)
6
7  print(min_max(3, 9, 5))     # really one tuple
```

output

```
3 9
(3, 9)
```

The counts on both sides must match. Comma-separated values are really packed into a tuple.

66

第 66 页 · Returning Several Values

return 后面用逗号隔开多个值，左边也用逗号隔开多个名字接住，就实现了一次返回多个结果。

最后一行说明了实情：Python 并没有真的返回两个值，而是把它们打包成了一个元组，只是接的时候又自动拆开了。元组是第 6 讲的内容。

两边数量必须一致，多一个少一个都会报错。

min 和 max 是内置函数，可以接受任意多个参数，返回最小和最大的那个。

No return Means None

None

```
1  def no_return():
2      print("I print, I do not return")
3
4  r = no_return()
5  print("returned:", r)
6  print("its type:", type(r))
```

output

```
I print, I do not return
returned: None
its type: <class 'NoneType'>
```

None means **no value**. It is **not** 0, **not** False, **not** the empty string.

67

第 67 页 · No return Means None

> 函数如果没有 return，或者 return 后面不跟任何东西，它的返回值就是 None。
>
> None 是 Python 里表示「没有值」的那个值，它自成一个类型 NoneType，而且整个程序里只有这一个 None。
>
> 要区分清楚：None 不是 0，不是 False，也不是空字符串。它表示的是「这里根本没有一个值」，而 0 和空字符串都是实实在在的值。
>
> 这一页是下一段的铺垫。下一段讲的那个错误，根源就是一个函数返回了 None 而作者以为它返回了结果。

PART 6 OF 7 · SECOND KEY POINT

print() and return

print() Is Not return

`print` is for a person · `return` is for the program

print puts text on the screen and that is the end of it — no part of the program can get it back. return hands the value to whoever called, and it can go on being computed with.

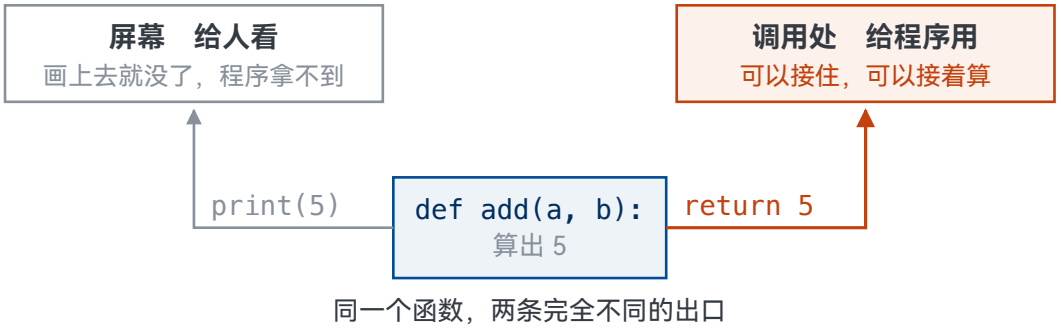
69

第 69 页 · print() Is Not return

> 这两者经常被混为一谈，因为它们看起来都「给出了结果」。实际上做的是完全不同的事。
>
> print 的输出对象是人。它把字符画到屏幕上，程序本身拿不到那些字符。
>
> return 的输出对象是程序。它把一个值交回调用点，那个值可以被赋给变量、参与运算、传给别的函数。屏幕上什么也不会出现。
>
> 接下来三页用两个几乎一样的函数把这个区别讲清楚。

Two Outlets, One Function

`print` and `return` send the value to **different places**. That is the whole difference.



同一个函数，两条完全不同的出口

`x = add(2, 3)`　　用 print 的那个：x 拿到 **None**　　用 return 的那个：x 拿到 **5**

print 通向屏幕，return 通向调用处。两条路互不相通。

70

第 70 页 · Two Outlets, One Function

在看代码之前，先把这张图看明白，后面四页就是它的验证。

中间是同一个函数，它算出了 5。往左那条路是 print：5 被画到屏幕上，人看得见，**但程序拿不到** —— 画上去就没了，没有任何办法把它捡回来。

往右那条路是 return：5 被交回给调用它的那一行，屏幕上什么都不会出现，**但程序拿得到**，可以赋给变量、可以接着算。

所以底下这一行最要紧：同样写 x = add(2, 3)，用 print 的那个函数让 x 拿到 None，用 return 的那个让 x 拿到 5。**屏幕上有没有东西，和程序里有没有拿到值，是两件独立的事。**

判题平台站在右边那条路上 —— 它只看 return 交回来的东西，左边那条路上打了什么它一个字都收不到。

Two Almost Identical Functions

add_print

```
def add_print(a, b):
    print(a + b)

add_print(2, 3)
```

output

```
5
```

add_return

```
def add_return(a, b):
    return a + b

add_return(2, 3)
```

output

（没有输出）

右边**算对了**，但屏幕上什么都没有：它把值交回来了，你没接住。

> 两个函数只差一个词。左边用 print，右边用 return。
>
> 左边屏幕上出现 5。右边屏幕上什么都没有。
>
> 初学者往往会误以为右边写错了：看起来没反应，好像什么都没发生。实际上它算得完全正确，只是把 5 交回给了调用它的那一行，而那一行没有把它接住，5 就被丢掉了。
>
> 下一页把它接住看看。

Catch Both and Look

the difference is whether you can keep going

```
1  def add_print(a, b):
2      print(a + b)
3
4  def add_return(a, b):
5      return a + b
6
7  x = add_print(2, 3)       # 5 on screen, but x is None
8  y = add_return(2, 3)      # nothing on screen, but y is 5
9
10 print("x =", x)
11 print("y =", y)
```

output

```
5
x = None
y = 5
```

`add_print` 没有 `return`，所以它的返回值是 **None**。

> 第七行执行时，屏幕上出现 5，那是函数内部的 print 打的。但 add_print 没有 return，所以它交回来的是 None，x 拿到的是 None。
>
> 第八行执行时，屏幕上什么都没有，但 y 拿到了 5。
>
> 所以最后两行打出 x = None 和 y = 5。
>
> 看清楚这个差别：屏幕上有没有东西，和程序里有没有拿到值，是两件独立的事。

Computing With the Result

None + None

```
1  def add_print(a, b):
2      print(a + b)
3
4  total = add_print(2, 3) + add_print(4, 5)
```

traceback

```
Traceback (most recent call last):
  File "example.py", line 4, in <module>
    total = add_print(2, 3) + add_print(4, 5)
            ~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'
```

屏幕上 5 和 9 都出现了，**看起来一切正常**，然后报 TypeError。

73

第 73 页 · Computing With the Result

这一页是这个错误最典型的样子。

两次调用都执行了，5 和 9 都打在屏幕上了，从输出看程序似乎在正常工作。然后加法这一步报错，因为两个返回值都是 None，而 None 不能相加。

报错信息是「unsupported operand type(s) for +: 'NoneType' and 'NoneType'」。看到 NoneType 出现在报错里，第一反应就应该是：**哪个函数忘了 return。**

这个线索很可靠，记住能省下大量调试时间。

Why the Two Get Confused

in the console　控制台

```
>>> add_return(2, 3)
5
>>> add_print(2, 3)
5

# looks identical
```

in a .py file　脚本文件

```
add_return(2, 3)    # nothing
add_print(2, 3)     # 5

# one prints, one does not
```

控制台会自动回显每个表达式的值，那是它为了方便你试语法额外做的事，不是 Python 语法的一部分。写进文件，两者立刻分家。

74

第 74 页 · Why the Two Get Confused

这个错误之所以难以自己发现，原因就在这一页。

L02 讲过：交互式控制台会把每个表达式的值自动回显出来。在控制台里调用 add_return，会看到 5，那是控制台替你显示的返回值；调用 add_print，你也看到 5，那是 print 打出来的。**两者看起来一模一样。**

于是很多人在控制台里试通了，以为函数写对了。可一旦写进 .py 文件，或者提交到判题平台，差别立刻出来：判题平台只看返回值，print 出来的东西它一个字都收不到。

请记住：控制台的回显是**解释器额外定义的行为**，不属于 Python 语法。规范的 IDE 里不会有这个效果。

The Rule

错

```
def f(x):
    print(x * 2)

# good for showing, useless
# for anything else
```

对

```
def f(x):
    return x * 2

print(f(3))          # to show it
print(f(3) + f(4))   # to keep computing
```

函数里一律 `return`，不要 `print`，除非题目明确要求输出。要显示的时候，在函数**外面**打印。

这样函数就是**可复用的**：想显示就显示，想接着算就接着算。

75

第 75 页 · The Rule

> 把 print 写死在函数里，就等于替所有使用者做了决定：这个函数只能用来显示。别人想拿结果做别的事就没办法了。
>
> 改成 return，选择权就交回给使用者：想显示，就在外面 print；想接着算，直接拿去算。函数本身不需要改。
>
> 这是一个更一般的设计原则的实例：**一个函数只做一件事，不要顺手替调用者决定别的事。**
>
> 调试过程中在函数里插几个 print 看中间值完全没问题，但记得交作业前删掉。

What the Rule Costs You

The judge grades by calling your function and checking what it returns

A function that prints instead of returning hands the judge None and scores zero — even when your algorithm is perfect and the screen looks exactly right.

第 76 页 · What the Rule Costs You

> 这不是抽象的编程风格问题，它直接影响成绩。
>
> 实验平台的判分方式是：导入提交的代码，调用题目要求的那个函数，把返回值和标准答案比对。它不看屏幕上打了什么。
>
> 所以一个用 print 不用 return 的函数，在自己电脑上运行看起来完全正确，提交上去是零分。每一届都有人在这里丢分，而且往往要丢两三次才反应过来。
>
> 本周的实验课有一道题专门用来暴露这个混淆，请认真做。

Modules

import（导入模块）

the math module

```
1  import math
2
3  print(math.sqrt(16))          # square root
4  print(math.pi)                # pi
5  print(math.floor(3.7))        # round down
6  print(math.ceil(3.2))         # round up
7  print(math.factorial(5))      # 5!
```

output

```
4.0
3.141592653589793
3
4
120
```

The shape is `import name`, then `name.function()`.

78

第 78 页 · import（导入模块）

> Python 自带大量现成的函数，按模块组织。math 是数学函数模块。
>
> import 一般写在文件最前面几行，一个模块 import 一次就够了。
>
> 注意 math.sqrt(16) 返回的是 4.0 不是 4，开方的结果一律是浮点数，这符合上一讲讲的类型规则。
>
> math.floor 和 math.ceil 是向下和向上取整，和上一讲的 // 以及这一讲的 int() 各有分工，不要混。
>
> 模块的完整机制在第 10 讲。现在会用就够了。

math.pi in the Circle Program

half of today, in six lines

```python
import math

def area_of_circle(r):
    return math.pi * r ** 2

print(f"{area_of_circle(1):.6f}")
print(f"{area_of_circle(2.5):.6f}")
```

output

```
3.141593
19.634954
```

这六行里有**函数、模块、幂运算、f-string、格式说明**，今天的一半。

79

第 79 页 · math.pi in the Circle Program

> 这是这一讲的一个小结：六行代码用上了今天讲的大部分东西。
>
> 和前面那个手写 3.14159 的版本比，用 math.pi 更准，而且意图更清楚，读代码的人不用去猜那串数字是什么。
>
> 这是使用标准库的一般理由：更准、更清楚，而且已经被大量使用验证过。**写代码之前先想一下标准库里有没有现成的**，这个习惯能省下很多时间。

From Functions to Bigger Structures

a function wraps logic · a class wraps data and logic · a module wraps functions and classes

The same idea repeated at three scales: divide and conquer. Classes are L9, modules are L10.

> 函数是这门课里第一个组织手段，但不是最后一个。
>
> 程序再大一些就会发现，有些数据和操作它们的函数总是一起出现，那时候需要把它们打包在一起，这就是类，第 9 讲讲。
>
> 再大一些，一组类和函数需要放进单独的文件，这就是模块，第 10 讲讲。
>
> 三者是同一个思路的三个尺度：**把大问题切成小问题，让每一块内部紧凑、块与块之间联系尽量少。**这个思路本身比任何具体语法都重要，换任何一门语言它都成立。

第 81 页 · 随堂小测 · 第 2 轮

第二轮小测，四道题，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 · 执行完下面三行，x 是多少？】

这一题考的是复合赋值和运算顺序，都是刚讲过的。

容易错的地方在于把 //= 当成 /=，或者忘了先乘后除的顺序。复合赋值的规则很简单：a op= b 就是 a = a op b，把整个右边先算完。

【第 2 题 · 下面四个比较，哪一个是 False？】

考的是字典序和大小写编号。四个选项里有三个符合直觉，剩下那个需要用到「大写字母编号更小」这条。

如果这道题答错了，回去把 ord('A')、ord('Z')、ord('a')、ord('z') 这四个数记一下，它们在后面处理文本时会反复用到。

【第 3 题 · 用户输入 3 和 4，下面这段输出什么？】

这是开场那个程序的变体，把加号换成了别的运算。

解题的关键是先确定每个操作数的类型：input 的结果一定是字符串，int() 包过的一定是整数。类型确定了，运算符的含义就确定了。

这一题如果全班正确率很高，说明第三段讲到位了。

【第 4 题 · 下面这行输出什么？】

考的是格式说明的读法：宽度、对齐、小数位三样都在里面。

读格式串的顺序是：冒号后面先看有没有对齐符号，再看宽度，最后看小数位。把这三样分开看，再长的格式串也能读明白。

RECAP

What to Take Away

- = **binds**. Read it as "becomes", and never name a variable after a built-in
- Strings compare in **dictionary order**, capitals first. They are **immutable**
- `int()` **truncates**; `round()` goes to even on an exact half
- **`input()` always returns a string.** `eval()` is dangerous — never read a number with it
- Format with an **f-string**: `{v:.2f}` for decimals, `{v:>10}` to align
- 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`**

82

第 82 页 · What to Take Away

七条，其中第四条和第七条是这一讲的两个重点，也是本周实验的重点。

第四条会在每一个有输入的程序里出现。第七条会在你提交的每一份作业里出现：判分平台只看返回值。

第二条的字符串不可变，在第 5 讲讲列表时会作为对照出现：列表是可变的，两者的差别会带来一系列后果。

第六条的「定义不等于运行」，在第 7 讲讲递归时会变得关键。

LAB 3

Lab 3 — 六道题，前三题练输出，后三题练函数

- **3-1** `c_to_f(c)` 摄氏转华氏，返回**保留一位小数的浮点数**
- **3-2** `mask_id(sid)` 学号脱敏：留前四后四，中间换成 `*`
- **3-3** `format_row(name, score)` 对齐的成绩表：姓名左对齐占 10 格
- **3-4** 一段想算平均值的代码报 `TypeError`，找出并修正 print/return 的误用
- **3-5** `read_int(prompt)` 安全的数字输入
- **3-6** `stats(a, b, c)` 一次返回最小值、最大值、平均值

另有 **B 段**和 **C 段**（命令行，**期末要考**）。**3-1 和 3-4 合起来就是今天的第二个重点：平台按返回值判分。**

83

第 83 页 · Lab 3 — 六道题，前三题练输出，后三题练函数

A 段六道题，前三题练这一讲的输出部分，后三题练函数。

3-1 看起来很简单，但每一届都有人在这里丢分。写成 print 之后自己运行看起来完全正确，提交上去零分。提交前请自己确认一次：这个函数有没有 return。

3-4 要求写出「原来错在哪」，不只是改对。能把错误说清楚，才算真的懂了。

3-3 用到今天讲的格式说明，宽度和对齐结合起来才能排整齐。3-6 是第一次遇到「一个函数返回多个值」，它返回的其实是一个元组，第 6 讲会正式讲。

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 1」，它覆盖前三周，10 月 10 日截止。

One Question Before You Go

`if` and `elif` are not formally until L4, but you can already read this — `if` turned up today when we did `return`.

grade.py

```
1  def grade(score):
2      if score >= 60:
3          return "pass"
4      elif score >= 90:
5          return "excellent"
6      return "fail"
7
8  print(grade(95))
```

95 分，它打印什么？ 不许运行。答案在下一讲的第一段。

84

第 84 页 · One Question Before You Go

留一道题回去想。

这段代码给一个分数，返回等级。规则看起来没问题：60 分以上及格，90 分以上优秀，其余不及格。

问题是：95 分传进去，它打印什么？

请不要运行它，先自己判断。这段代码**不报错**，跑起来也很正常，值得想一想的正是这一点。

if 和 elif 下一讲才正式讲，但今天讲 return 的时候你已经见过 if 了，读懂这段没有障碍。答案在下一讲的第一段。

Next Lecture

So far your programs only run straight down, once

L4: conditions and loops. Letting a program decide and repeat is where it stops being a list of instructions and becomes an algorithm — and where that last question gets answered.

第 85 页 · Next Lecture

> 到今天为止，程序都是从第一行顺序执行到最后一行，中间没有分支。
>
> 下一讲加两样东西：判断和重复。有了它们，同一段代码在不同输入下可以走不同的路、可以执行不同的次数，程序才谈得上根据不同情况做不同的事。
>
> 这一讲讲的函数会立刻派上用场：判断和循环写在函数里，才能被反复调用。今天讲的 return「立即结束函数」那个性质，下一讲写分支时会用得很频繁。