第 2 讲 · 讲义

字符与数

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。

第 1 页 · Characters and Numbers

> 上一讲说清楚了计算机里只有 0 和 1。这一讲讲的是：这些 0 和 1 怎么表示出字母、整数和小数。
>
> 重点有两个。一是同一串 0 和 1 可以有不同含义，这一点在代码里具体怎么体现；二是浮点数的计算结果为什么常常出乎意料。后一个问题几乎每个写程序的人都遇到过。

An Opening Prediction

A

```
print(0.1 + 0.2)
```

B

```
print(0.1 + 0.2 == 0.3)
```

先在心里定一个答案。 两行都很短，不需要动笔。

> 这两行代码很短，几乎没有出错的余地：把 0.1 和 0.2 加起来，再判断结果是不是 0.3。
>
> 公布结果之前，请先在心里给一个答案。先有判断再看结果，印象会比直接听讲深得多。

The Actual Output

run it

```
1  print(0.1 + 0.2)
2  print(0.1 + 0.2 == 0.3)
```

output

```
0.30000000000000004
False
```

第二行是 **False**。不是 Python 的缺陷，也不是这台机器的问题。 换任何一台机器、任何一种主流语言，结果都一样。

3

第 3 页 · The Actual Output

> 第一行的输出不是 0.3，而是 0.30000000000000004；第二行因此是 False。
>
> 先排除几种常见的误解：这不是 Python 的错误，不是这台机器的问题，也不只是「精度不够」这么简单。同样的运算在 C、Java、JavaScript 里结果完全一样。
>
> 原因在于二进制无法精确表示 0.1，就像十进制无法精确表示 1/3。第四段会详细解释。现在只要记住一条：**浮点数不能用 == 比较。**

随堂小测 · 第 1 轮

扫码作答 · 这一轮一题

1 那么 0.1 + 0.3 == 0.4 呢?

这一题现在不公布答案。讲完浮点数那一节，我们回来看开课时的分布。

https://taohuang.info/cs1602/zh/quiz/2

4

第 4 页 · 随堂小测 · 第 1 轮

> 现在收一道题，先不公布答案。第四段讲完浮点数，正好会讲到为什么这一道和开场那一道结果相反，那时候再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 1

Where Lecture 1 Ended

- A computer stores **nothing but 0s and 1s** — high voltage is 1, low is 0
- **The same bits can mean anything** — a number, a character, an instruction. The meaning comes from a **convention** 约定
- That is exactly why von Neumann could put code and data in one memory
- And the question L1 closed on: **so how does it hold -3, 3.14, and 「你好」?**

今天讲三套约定，以及每一套要付的代价。

第 5 页 · Where Lecture 1 Ended

先回顾上一讲。人工计算既慢又容易出错，所以要造机器来代替；机器用开关做计算，开关只有通和断两种状态，机器里因此只有 0 和 1。

和今天关系最直接的是最后这一点：一串 0 和 1 本身不表示任何东西，含义来自人为规定的约定。冯诺依曼体系把程序和数据放在同一块存储器里，正是因为这一点：在机器看来两者都是 0 和 1，区别只在于按哪种方式解释。

上一讲末尾留了一个问题：机器里只有 0 和 1，负数、小数和汉字要怎么表示。今天就回答这个问题。

一共三套约定：ASCII 和 Unicode 负责字符，IEEE 754 负责小数，类型规则负责运算。三套各有各的代价，代价也是今天要讲的。

Today's Question

How 0s and 1s become letters, integers and decimals

L1 said a bit pattern has no meaning of its own — the meaning comes from a convention. Today: which conventions, and what each one costs.

第 6 页 · Today's Question

> 这一讲把 L01 结尾那句「意义来自约定」展开来讲。ASCII 是一套约定，让 65 这个数代表字母 A；IEEE 754 是另一套约定，让一串 bit 代表一个小数。
>
> 约定都有代价。ASCII 只有 128 个位置，装不下中文；IEEE 754 用有限的位数去近似无限的小数，于是有了误差。这两套约定的代价，今天都会具体讲到。

ROADMAP

Outline

- **A complete program** — and the four basic types it already uses
- **Bits and bases** — a base is a notation, not a property of the number
- **Characters** — ASCII, Unicode, quoting, and what `print` really does
- **Numbers** — no ceiling on integers, no exactness on floats
- **Arithmetic** — seven operators, and what type each hands back
- **Logic** — comparisons, `and` / `or` / `not`, and why they are numbers too

> 四段的顺序是按依赖关系排的：先建立「一切都是 bit」这个前提，后面讲字符和数才有基础。
>
> 其中第四段的类型规则要多花点时间：它决定了一个表达式返回整数还是浮点数，这个差别会在取下标、比较、格式化输出时反复出现。

A Complete Program

A First Complete Program

bmi.py

```
1  # Body mass index
2  name = "Zhang San"
3  height = 1.75          # meters
4  weight = 68.5          # kilograms
5
6  bmi = weight / (height ** 2)
7
8  print(name, "has a BMI of", bmi)
9  print("rounded:", round(bmi, 1))
```

output

```
Zhang San has a BMI of 22.367346938775512
rounded: 22.4
```

9

第 9 页 · A First Complete Program

> 上一讲只写了一行 print。这是一个真正做事的程序，但它用到的东西这一讲全都会讲到。
>
> 先整体看。它做了三件事：准备数据（前四行）、计算（第五行）、输出结果（后两行）。绝大多数程序都是这个骨架。
>
> 注意第一行以 # 开头，那是注释，写给人看的，Python 会跳过。还要注意 bmi 的输出有一长串小数，那正是这一讲第四段要处理的问题。

The Program, Line by Line

Same code as the previous slide. Four things are happening.

bmi.py

```
# Body mass index
name = "Zhang San"
height = 1.75
weight = 68.5
bmi = weight / (height ** 2)
print(name, "has a BMI of", bmi)
```

注释：写给人看的，Python 整行跳过

赋值：把右边的值，存进左边这个名字

计算：算式的结果再存进一个新名字

输出：逗号隔开，中间自动补一个空格

准备数据、计算、输出。绝大多数程序都是这个骨架。

把上一页那段代码原样搬过来，逐行对照着看。

第一行以井号开头，是注释，写给人看的，Python 整行跳过。

中间三行是赋值：等号左边是名字，右边是值，把右边的值存进左边这个名字。三个名字分别拿到一个字符串和两个小数。

第五行先算右边那个式子，再把结果存进 bmi 这个新名字。所以等号右边可以是任意复杂的算式。

最后一行输出。print 里用逗号隔开多个值，Python 会在相邻两个之间自动补一个空格。

这一页右边四个框正好对应程序的三件事：准备数据、计算、输出。读任何一段陌生代码，都可以先这样切开。

第五行里的四个符号 —— 等号、除号、两个星号、括号 —— 下一页专门讲。

FROM THE LINE ABOVE

One Line, Four Symbols

- `=` — **put the right-hand value into the name on the left**
- `/` — divide
- `**` — power（幂）. `height ** 2` is height squared
- `( )` — do what is inside first

一个等号是把右边存进左边。数学里的相等，Python 里要写两个。

11

第 11 页 · One Line, Four Symbols

> 刚才那行 `bmi = weight / (height ** 2)` 里有四个符号，一个一个说清楚。
>
> 等号最要紧。它在这里的意思是「把右边算出来的值，存进左边这个名字」，从左往右读是「bmi 得到……」。数学里的相等在 Python 里写两个等号，今天最后一段会正式区分。
>
> 两个星号是幂运算，height **2 就是身高的平方。括号的作用和数学里一样，先算里面。这里其实不加也对，因为** 比 / 先算，但加上更好读。
>
> 接下来两页把今天会用到的运算符一次列出来，先有个印象。

Arithmetic Operators

Seven of them. Part 5 comes back to each one.

operator	meaning	example	result
`+`	add 加	`7 + 2`	9
`-`	subtract 减	`7 - 2`	5
`*`	multiply 乘	`7 * 2`	14
`/`	divide 除	`7 / 2`	3.5
`//`	floor divide 整除	`7 // 2`	3
`%`	remainder 取余	`7 % 2`	1
`**`	power 幂	`7 ** 2`	49

注意 `/` 这一行：除得尽，结果也是小数

七个算术运算符。现在只要认得，不用记。

七个算术运算符，这里一次列全。前三个和数学里一样，不用多说。

要留意的是第四个：一个斜杠的除法，**结果永远是小数**，7 除以 2 得到 3.5，就算除得尽也是小数，4 除以 2 得到的是 2.0，不是 2。这一点 Part 5 会讲为什么。

两个斜杠是整除，往下取整；百分号是取余。这两个在第 4 讲写循环时会经常用到。

表里的结果都是构建这份课件时实际运行得到的，可以自己逐条验证。

Comparison, Logic, Assignment

These answer **True** or **False**. Part 6 comes back to them.

Comparison 比较

结果只有两种：True 或 False

`>`	greater 大于	`7 > 2`	**True**
`<`	less 小于	`7 < 2`	**False**
`>=`	at least 大于等于	`7 >= 7`	**True**
`<=`	at most 小于等于	`2 <= 7`	**True**
`==`	equal 相等	`7 == 2`	**False**
`!=`	not equal 不等	`7 != 2`	**True**

Logic 逻辑

and 且　or 或　not 非

`and`	`True and False`	**False**
`or`	`True or False`	**True**
`not`	`not True`	**False**

一个等号和两个等号

`x = 7`　把 7 存进 x

`x == 7`　问 x 是不是 7

这些符号今天后半段会逐个讲透，现在认得就够了。

左边是比较，结果只有 True 和 False 两种，首字母都大写。

右上是逻辑运算，把几个判断接起来：and 要求都成立，or 要求至少一个成立，not 取反。这三个在第 4 讲写条件判断时是主力。

右下角那个框是今天最容易写错的地方：一个等号是赋值，把 7 存进 x；两个等号是提问，问 x 是不是 7。写错了轻则报语法错误，重则悄悄把变量的值改掉，后面这种最难查。

开场那道题用的就是表里的两个等号，现在应该知道它在问什么了。

The Three Parts of a Program

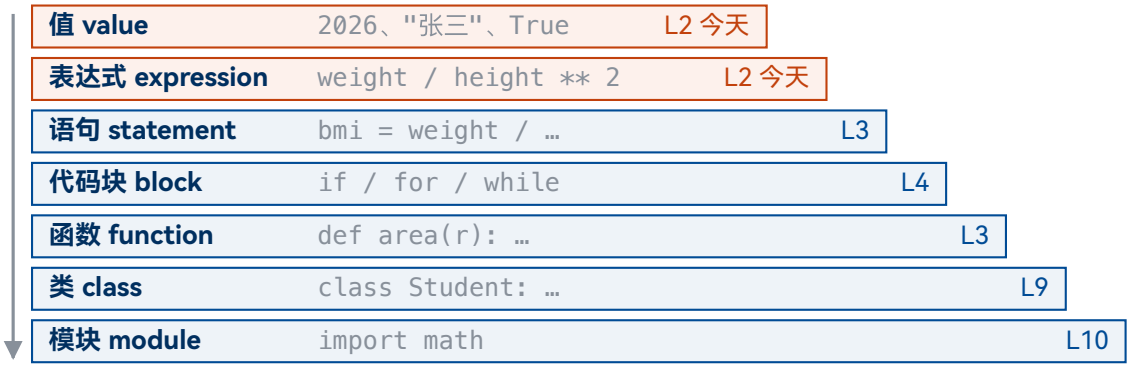
get data · compute · report

Today settles what data actually is. L3 turns data into text you can read; L4 adds deciding and repeating.

> 把这个骨架记住，后面读任何一段陌生代码都可以先按它切开：哪几行在准备数据、哪几行在计算、哪几行在输出。
>
> 这个习惯在代码变长之后特别有用。一段两百行的程序，多半也是这三块，只是每一块内部又有结构。

From Values to Modules

A program is built the way writing is（字词句段篇章）. Here are the Python names, and **each level is made of the one above**.



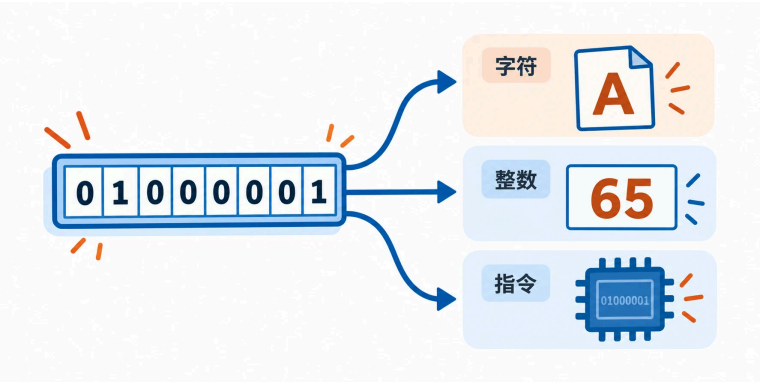
今天讲最基本的两级：值和表达式。往下每一级都由上一级搭起来。

15

第 15 页 · From Values to Modules

> 这个类比是这门课历任老师传下来的：程序的结构和写文章相似，字、词、句、段、篇、章。不过图上写的是 Python 自己的名词，比喻只是帮你记住顺序。
>
> 从上往下看，每一级由上一级组成：若干个值组成一个表达式，一个表达式加上赋值或者调用就成了一条语句，连续的语句用 if、for 围起来是一个代码块，代码块打包起来是函数，函数和数据打包起来是类，类和函数装进文件就是模块。
>
> 橙色的两条是今天的内容：值有哪些类型，表达式里的运算符是什么意思。
>
> 右边一列标着哪一讲讲。现在学的东西看起来零碎，但每一样都能在这张图上找到位置。

So How Does It Know?



*同一串 bit，可以是字母、可以是整数、可以是别的东西。**决定「按哪种方式读」的，就是类型。***

计算机里只有 0 和 1，那它怎么知道哪一块是字符、哪一块是数？

16

第 16 页 · So How Does It Know?

> 讲数据类型之前，先把问题问出来。
>
> 上一讲的结论是：计算机里只有 0 和 1，一串 bit 本身没有意义。那么问题来了 —— 同样是 01000001 这八位，凭什么有的时候它是字母 A，有的时候它是整数 65？
>
> 先让大家想十秒钟。答案不在这串 bit 里面，而在「谁来读它、按哪种约定读」。
>
> 程序里承担这件事的，就是下一页要讲的**类型**。

DATA TYPES

Data Types（数据类型）

- The BMI program already used three of them. With integers, that is the four you will use most of this term:
  - **string** `str` 字符串 — `"Zhang San"`. A piece of text
  - **float** `float` 浮点数 — `1.75`. A number with a fractional part
  - **boolean** `bool` 布尔值 — the result of `bmi > 24`. Only `True` and `False`
  - **integer** `int` 整数 — `2026`, `–3`, `0`. No fractional part
- **A value's type decides what you can do to it** — and what an operator means

物以类聚。**先分类，再谈能做什么。**

下面给这些不同性质的值一个正式的名字。

刚才那个 BMI 程序里，"张三" 是一段文字，1.75 是一个小数，bmi > 24 的结果是真或假。它们不只是「不同的值」，而是分属不同的**类型**。加上整数，这四种是这门课前期最常用的。

类型不只是个标签。它决定了这个值能参与哪些运算，也决定同一个运算符作用在它身上是什么意思。下面两页就是例子。

Python 里还有别的类型：列表、元组、字典、集合是 Part II 的内容，你自己定义的类是 Part IV 的内容。但规则是同一套。

type(): Asking a Value What It Is

the four basic types

```
1  print(type("Zhang San"))
2  print(type(1.75))
3  print(type(2026))
4  print(type(1.75 > 1))
```

output

```
<class 'str'>
<class 'float'>
<class 'int'>
<class 'bool'>
```

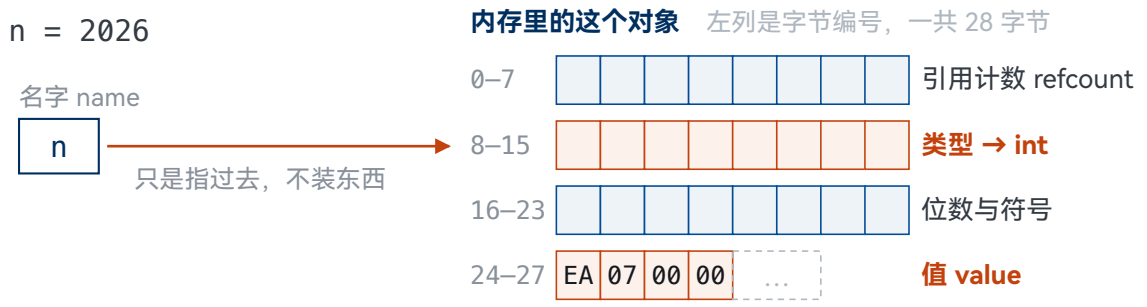
Ignore the word `class` for now — Part IV gets to it.

第 18 页 · type(): Asking a Value What It Is

> type() 是这一讲最有用的一个诊断工具。程序行为不符合预期时，第一件该做的事往往就是用 type() 看一下可疑的值。
>
> 输出里的 str、float、int、bool 就是这四种类型的名字，以后报错信息里也会用这些名字，比如「TypeError: can only concatenate str (not "int") to str」，读懂这句话需要先知道 str 和 int 是什么。
>
> 输出前面的 class 这个词，说明在 Python 里类型本身也是一种对象。这一点 Part IV 会展开，现在可以先忽略。

Where the Type Actually Lives

n = 2026. Here is what actually sits in memory.



每个小格是**一个字节**。type(n) 读的就是第 8–15 这八个字节。
2026 不大，**4 个字节就够**。数大了值这一段会变长，整数没有上限。

类型和值是**存在一起**的。所以运行时随时问得出来。

19

第 19 页 · Where the Type Actually Lives

上一页说「类型决定这串 bit 怎么读」。那类型本身存在哪里？把这个对象在内存里摊开看。

先看左边。n 只是一个名字，它自己不装东西，只是指向内存里的某个位置。

右边是被指向的那个对象，一共 28 个字节，每个小格就是一个字节。从上往下分四段：

开头八个字节是引用计数，记着「现在有几个名字指着我」，第 10 讲讲内存管理时会用到，今天不展开。

第 8 到 15 这八个字节是重点：它记着这个对象是什么类型，存的是 int 这个类型对象的地址。**type(n) 读的就是这八个字节。**上一页问的「计算机怎么知道哪块是字符哪块是数」，答案就在这里 —— 它不用猜，值自己带着答案。

再八个字节记位数和符号。最后四个字节才是值本身，2026 写成十六进制是 EA 07，小端序所以低位在前，后面两个字节是 0。

这里要特别说一句，免得记成「Python 的 int 是 32 位」。 2026 不大，四个字节装得下。数一大，值这一段就会变长：

2 的 60 次方，这个对象是 36 字节；10 的 100 次方，72 字节。所以图上值那一段后面画了虚线格 —— 它可以往后接。

这正是第四段「整数没有上限」的底层原因：C 的 int 定死 32 位，写满就溢出；Python 的整数会自己变长，只受内存限制。

（两个常见追问，备着。）

一、28 个字节为什么这么多？ 28 字节是 224 位，可真正是「数」的只有最后四个字节。前面 24 个字节是**每个 Python 对象都有的头**：引用计数（谁还在用我，回收内存要看它）、类型指针（type() 和运算符靠它）、位数与符号（因为长度不固定，得记着）。C 的 int 这三样都不需要 —— 类型编译完就不在内存里，长度定死，内存谁管由程序员自己写。这 24 个字节换来的是：类型能在运行时查、内存不用自己释放、整数没有上限。

顺带一提，小整数不会每次都新造一个：-5 到 256 在解释器启动时就建好了，全程序共用一份，所以循环里用小数字并不会真的一直分配 28 字节。

二、具体怎么存的？ CPython 按 30 位一组、每组占 4 字节。**不用主动讲**，绕这一层对这个阶段没有用。）

所以「类型跟着值走」这句话，在内存里是字面意义上的：类型和值装在同一个对象里，隔了十六个字节。

补一句严谨的：这是 64 位 CPython 的布局，我在 3.12 和 3.14 上都读过，课程用的 3.13 夹在中间。别的 Python 实现、别的版本可能不一样，**要记的是「类型和值存在一起」这件事，不是这些具体偏移量。**

Same operator, different meaning

the type decides

```
1  print(3 * 2)
2  print("ab" * 2)
3
4  print(1 + 2)
5  print("1" + "2")
```

output

```
6
abab
3
12
```

On numbers * **multiplies** and + **adds**. On strings * **repeats** and + **joins**.

20

第 20 页 · Same operator, different meaning

> 这四行是这一节的重点。同样的 * 和 +，作用在不同类型上，做的是完全不同的事。
>
> 3 * 2 是乘法，得到 6。"ab" * 2 是重复，得到 "abab"。1 + 2 是加法，得到 3。"1" + "2" 是拼接，得到 "12"。
>
> 所以「这行代码是什么意思」这个问题，不知道操作数的类型就回答不了。所以后面遇到奇怪的结果，先看类型往往最省时间。
>
> 顺带说一句，字符串的重复和拼接第 3 讲会正式讲，这里先提一下。

PART 2 OF 6

Bits and Bases

Bit and Byte（位与字节）

- A storage cell holds one of two states: high voltage and low, written **1** and **0**
- One 0 or 1 is a **bit** 位
- Eight bits are a **byte** 字节
- One bit distinguishes 2 cases, two bits 4, **n bits 2^n**

So one byte distinguishes **2^8 = 256** things. That number keeps coming back.

第 22 页 · Bit and Byte（位与字节）

这几个词后面会一直用到，先统一一下说法。

关键是最后一条：n 个 bit 能表示 2 的 n 次方种情况。这个式子解释了很多后面会遇到的数字：为什么一个字节能存 256 个值、为什么 ASCII 用 7 个 bit 正好覆盖 128 个字符、为什么 64 位整数的范围是那么大。

顺带说一句单位：1024 个 byte 是 1KB，1024KB 是 1MB。这里用 1024 而不是 1000，是因为 1024 正好是 2^{10}。

One Rule, Any Base

Same rule twice. Only the **base** changes.

十进制 decimal — 2026

		权重 weight		这一位的值
2	× 10^3	= 1000	→	2000
0	× 10^2	= 100	→	0
2	× 10^1	= 10	→	20
6	× 10^0	= 1	→	6
				2026

二进制 binary — 1101

		权重 weight		这一位的值
1	× 2^3	= 8	→	8
1	× 2^2	= 4	→	4
0	× 2^1	= 2	→	0
1	× 2^0	= 1	→	1
				13

每往左一位，权重乘一次底数。十进制乘 10，二进制乘 2。

23

第 23 页 · One Rule, Any Base

二进制并不是一套新规则，它和十进制用的是同一条规则。

上面一行是十进制的 2026。每个格子上面标的是这一位的权重：从右往左是 10 的 0 次方、1 次方、2 次方、3 次方。把数字和权重相乘，就是下面那行橙色的数，加起来正好是 2026。

下面一行是二进制的 1101，一模一样的做法，只是权重换成 2 的幂：8、4、2、1。把标着 1 的那几位加起来，得到 13。

所以手算二进制转十进制，方法就是从右往左写下 1、2、4、8、16，然后把对应位是 1 的加起来。这条规则对任何进制都成立，换个底数而已。

Base Conversion in Python

base conversion

```python
print(0b1101)          # a binary literal, prefix 0b
print(bin(13))         # the other direction
print(int("1101", 2))  # parse a string in base 2
```

output

```
13
0b1101
13
```


三个方向各有对应的写法。

`0b1101` 是字面量，在代码里直接写出一个二进制数。`bin(13)` 把整数转成二进制的字符串表示，注意它返回的是字符串，所以带着 `'0b'` 前缀。`int("1101", 2)` 从字符串解析，第二个参数说明按几进制读。

常见的错误是把 `bin()` 的结果当成数来用。它是字符串，`bin(13) + 1` 会报错。

ONE NUMBER

Four Notations for One Number

Binary is what the machine uses — and **unreadable for humans** past a few digits. These are shorter ways to write the same bits.

- decimal — no prefix — 96
- binary — `0b` — `0b1100000`
- octal — `0o` — `0o140`
- hexadecimal — `0x` — `0x60`

Different notations, identical bits once they are in memory.

25

第 25 页 · Four Notations for One Number

先说为什么要有不止一种写法。二进制是机器用的，可人读起来太长：一个字节写出来就是八位，一个内存地址三十二位，中间漏看一位都发现不了。所以需要更短的写法，而且要能一眼换算回二进制。

这四个写法指的是同一个数 96，就像「十二」「12」「XII」指的是同一个数量。

写法的差别只存在于源代码里。读进内存之后它们就是同一串 bit，没有办法区分这个数当初是怎么写的。

十六进制之所以常用，是因为**一位十六进制正好对应四位二进制**。0x60 拆开就是 0110 0000，一眼能看出二进制结构，而且比写八位数字短。以后读内存地址、颜色值、字节序列，见到的基本都是十六进制。

The Four Literals, Checked

four notations, one number

```
1  print(96, 0b1100000, 0o140, 0x60)
2  print(96 == 0x60)
```

output

```
96 96 96 96
True
```


> 第一行把四种写法打印出来，输出全是 96，因为 print 显示整数时一律用十进制。
>
> 第二行确认它们真的相等。这一步建议自己跑一次。「只是写法不同」这句话，看到 True 之后才会真的接受。

Why Binary（为什么偏偏是二进制）

- In a circuit, telling **current** from **no current** is the most reliable distinction there is
- Decimal would mean telling ten voltage levels apart
  - A little noise and 3 V reads as 3.3 V — wrong digit
- Two states leave the widest margin for error

这是**工程上的选择**，不是数学上的必然。

27

第 27 页 · Why Binary（为什么偏偏是二进制）

> 这个问题几乎每个初学者都会问：为什么不直接用十进制。
>
> 原因在电路那一层。表示两种状态，只需要区分「电压高于某个阈值」和「低于某个阈值」，中间留出很大的余量，元件老化、温度变化、电磁干扰都不容易让它认错。
>
> 如果要表示十种状态，就得把电压区间切成十份，每一份都很窄，稍有偏差就读成邻近的数字。历史上确实有人造过十进制计算机，但都没能在可靠性和成本上竞争过二进制。

PART 3 OF 6

Characters

Letters in a Number Machine

A computer stores numbers. What about letters?

There is no clever way out. Someone has to publish a table saying which number stands for which character.

29

第 29 页 · Letters in a Number Machine

> 这个问题的答案并不绕。既然存储单元里只能放数字，字母就只能用数字来表示。
>
> 剩下的问题只是：哪个字母对应哪个数字？这个对应关系不能各自为政，否则一台机器存的文件，换一台机器就读不出来。所以必须有一张公共的表，而且要有人来定。

ASCII

ASCII: the Agreed-on Table

- American Standard Code for Information Interchange
- **7 bits**, so **128** characters
  - Upper and lower case letters, digits, punctuation, some control characters
- `'0'` → **48** `'A'` → **65** `'a'` → **97** space → **32**

`'a'` is exactly **32** more than `'A'`. That gap gets used later.

30

第 30 页 · ASCII: the Agreed-on Table

> ASCII 定于 1963 年，用 7 个 bit，正好覆盖 128 个位置。为什么是 7 位不是 8 位？因为当时的电报设备按 7 位传，第 8 位常用来做校验。
>
> 表里的编号不是随便排的。数字 0 到 9 是连续的 48 到 57，大写字母 A 到 Z 是连续的 65 到 90，小写 a 到 z 是 97 到 122。连续这一点很有用：判断一个字符是不是数字，只要看它的编号在不在 48 到 57 之间。
>
> 大小写正好差 32，也就是第 6 个 bit 的权重。这不是巧合，是设计时特意安排的，这样只翻一个 bit 就能改变大小写。

The Whole ASCII Table

All **128** of them. A code is **the number at the start of its row, plus its column**: A is 64 + 1 = **65**.

行首 +	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32	SP	!	"	#	$	%	&	'	(	)	*	+	,	−	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

同一类字符的编号是连着的。四个起点：空格 32、数字 48、大写 65、小写 97。

31

第 31 页 · The Whole ASCII Table

这就是整张 ASCII 表，128 个编号一个不少。

先说怎么读：行首那个蓝色的数是这一行的起点，列号在最上面一排。编号等于两者相加。比如大写 A 在 64 那一行的第 1 列，编号就是 65。

看颜色分块。最上面两行灰的是控制字符，没有可见形状 —— HT 是 Tab，LF 是换行，CR 是回车，ESC 是退出键。待会儿讲转义字符时会回到这一段。

橙色那十个是数字 0 到 9，编号 48 到 57。两块蓝色分别是大写和小写，65 到 90、97 到 122。**每一块内部都是连续的**，所以判断一个字符是不是数字，只要看编号落没落在 48 到 57 之间。

最值得记的是：同一个字母的大写和小写正好差 32，图上就是竖着隔两行的同一列。32 是 2 的 5 次方，也就是只翻一个 bit 就能改变大小写 —— 这是设计时特意安排的。

表里的字符和控制字符缩写都是程序生成的，控制字符那组名字取自标准库的 curses.ascii，不是我手打的。

ord() and chr(): the Bridge Both Ways

ord / chr

```
print(ord('A'), ord('a'), ord('0'), ord(' '))
print(chr(65), chr(97), chr(48))

# the cases are 32 apart, so this works
print(chr(ord('h') - 32))
```

output

```
65 97 48 32
A a 0
H
```

`ord()` turns a character into its code, `chr()` turns it back.

这一节要用的是这两个函数。名字的来历：ord 是 ordinal（序号），chr 是 character（字符）。

最后那行演示了大小写转换：把 'h' 的编号减 32 得到 'H' 的编号，再转回字符。实际写代码时当然用 `.upper()`，但这个练习可以确认自己是不是真的理解了「字符就是数字」。

注意 `ord()` 只接受一个字符，`ord('ab')` 会报错。

'0' and 0（字符与数字）

错

```
'1' + '2'

# gives '12'
# string concatenation
```

对

```
1 + 2

# gives 3
# integer addition
```

`'0'` is a character, code 48. `0` is an integer. Different things in memory, and different rules when they meet an operator.

`input()` 拿到的**永远是字符串**。这个混淆在处理输入时最容易出事，第 3 讲会专门讲。

> 这是新手最常见的一种混淆，而且它不会报错：`'1' + '2'` 是合法的，只是结果是 '12' 而不是 3。
>
> 带引号的是字符或字符串，不带引号的是数。屏幕上看起来都是 1，但内存里一个是编号 49，一个是数值 1。
>
> 这个混淆最常在读取输入时出问题：`input()` 无论用户输入什么，拿到的都是字符串。要当数用就必须先 `int()` 转换。第 3 讲讲输入输出时会正面处理这个问题。

Unicode（统一码）

ASCII has **128** places. Enough for English — not for 中文, not even for café.

Unicode

```
1  print(ord('中'), ord('文'))
2  print(chr(20013), chr(25991))
```

output

```
20013 25991
中 文
```

Unicode numbers almost every script on earth. `ord()` and `chr()` work on all of it, unchanged.

34

第 34 页 · Unicode（统一码）

ASCII 只有 128 个位置，连西欧那些带变音符号的字母都装不下，更不用说汉字。

现代的方案是 Unicode，它给几乎所有文字系统的字符都分配了一个编号，目前收录了十几万个。汉字「中」的编号是 20013。

而 `ord()` 和 `chr()` 不需要为此改变：它们本来就是「字符和编号之间转换」，Unicode 只是把表变大了。

但事情没有这么简单：编号变大之后，一个字符要用几个字节存？不同的存法就是不同的编码，而编码不匹配就会出现乱码。这部分留到第 12 讲。

How Do You Print Two Lines?

Writing the **word** does not help: `"newline"` is seven characters, not the one key.

改改看，再点运行

```
# 一次 print，两行输出。怎么写?
print("第一行 第二行")
```

output

```
hit ▶ Run above
```

有些字符打不出来，有些字符和语法冲突。**下一页是解决办法。**

35

第 35 页 · How Do You Print Two Lines?

先别急着往下讲，问一个问题：想让一次 print 输出两行，怎么写?

让大家说。常见的答案有几种，都可以在这个框里当场试：

第一种，把「换行」两个字写进去 —— 打印出来就是这两个字本身。写英文 newline 也一样，那是七个字母，不是那一个键。

第二种，在引号中间直接按回车 —— Python 会报语法错误，因为一个普通字符串不能跨行。

问题出在哪？回车在 ASCII 表里确实有编号，是 10，刚才那张带子最左边那段就是它。但它**没有可见的形状**，你没办法把「换行」这个动作画在一对引号中间。

引号本身是另一种麻烦：它不是打不出来，而是**和语法冲突** —— 引号是字符串的边界，直接写进去，Python 会以为字符串在那里就结束了。

两类问题，一个解决办法：用两个普通字符的组合，去表示那一个打不出来的字符。下一页就是。

ESCAPE

Escape sequences 转义字符

- `\n` — newline
- `\t` — tab
- `\\` — one backslash
- `\'` — single quote　　`\"` — double quote

Once a backslash appears, it and the next character are read **as one**.

> 转义序列的规则很简单：反斜杠出现之后，它和紧跟着的字符合起来算一个。所以 `"a\nb"` 这个字符串的长度是 3，不是 4。
>
> 需要转义引号是因为引号本身被用来标记字符串的开始和结束。如果字符串里要出现同一种引号，就得转义，或者换一种引号：`"他说：'你好'"` 不需要转义。
>
> 反斜杠自己也要转义成 `\\`，因为单个反斜杠会被理解成转义的开始。

The Escapes at Work

escapes

```
1  print("line one\nline two")
2  print("Name\tStudent ID")
3  print("She said: \"hello\"")
```

output

```
line one
line two
Name    Student ID
She said: "hello"
```


> 三个例子分别演示换行、制表符和转义引号。
>
> 注意输出里看不到反斜杠。它只存在于源代码里，它是写给 Python 看的记号，说明「后面这个字符要特殊理解」。真正存进字符串的是那个特殊字符本身。
>
> 制表符的宽度不固定，它的作用是「跳到下一个制表位」，所以用它对齐输出并不可靠，列宽稍有变化就会错位。第 3 讲会讲更可靠的对齐办法。

Backslashes in a Windows Path

错

```
print("C:\name")

# \n became a newline
# two lines of output
```

对

```
print(r"C:\name")

# r = raw string 原始字符串
# no escaping at all
```

Windows separates path components with a backslash, and a backslash also starts an escape. They collide. An `r` before the quote turns escaping off.

也可以写成 `"C:\\name"`，把每个反斜杠都转义一次。**加 r 更清楚。**

这是转义机制最常见的一种「误伤」。Windows 的路径用反斜杠分隔，而 `\n`、`\t` 恰好都是合法的转义序列，于是路径里的一段就变了样。

`C:\name` 里的 `\n` 会变成换行，输出成两行；更糟的是 `C:\table`，`\t` 变成制表符，看起来只是多了几个空格，错得很隐蔽。

两种解法：每个反斜杠写两遍，或者在引号前加 r。写正则表达式时也会大量用到 r 字符串，第 15 讲会再遇到。

Multi-line strings 三引号

triple quotes

```
poem = """江南好，风景旧曾谙。
日出江花红胜火，
春来江水绿如蓝。"""

print(poem)
print("""He said "hi", I said 'bye'.""")
```

output

```
江南好，风景旧曾谙。
日出江花红胜火，
春来江水绿如蓝。
He said "hi", I said 'bye'.
```

代码里长什么样，打出来就是什么样。

39

三个引号括起来的字符串，里面的换行原样保留，不用写 \n。

一两个换行用 \n 更紧凑；整段文字、一幅图、一张表，用三引号。下次实验第一题要画一只熊，两种写法都试一遍就知道差别了。

第二行还顺带解决了引号的问题：三引号里面单引号和双引号都能直接写，一个都不用转义。

三引号还有第三种用途：写在函数开头当说明文档，那叫 docstring，第 3 讲讲函数时会再见到它。

The Console Echoes, a File Does Not

interactive console（控制台）

```
>>> 1 + 1
2
>>> x = 5
>>> x
5
>>> "SJTU"
'SJTU'
```

a .py file（脚本文件）

```
# quiet.py
1 + 1
x = 5
x
"SJTU"

$ python quiet.py
$
```

The console echoes the value of every expression. A `.py` file does not. 文件里想看到什么，就得自己 `print` 出来。第一周最常撞的就是这个。

第 40 页 · The Console Echoes, a File Does Not

这一页讲的是每年第一周都要解释一遍的问题。

左边是交互式控制台：在终端里运行 python 进入，提示符是三个大于号。在里面输入一个表达式，回车，它会**自动把值显示出来**。这是控制台特意做的，方便试语法。注意最后那行，字符串回显出来是带引号的。

右边是把同样几行写进 .py 文件再运行，屏幕上**什么都没有**。文件里的表达式照样求值了，只是没有人把结果显示出来。

所以：控制台里试语法很方便，但写程序的时候，想看到什么就得自己 print。很多人第一周在这里卡住，以为自己的程序没运行。

PRINT

A Mental Model for print()

- Think of a pen moving left to right across a page
- **Every print() ends with a newline** — the pen drops to the next line
- Arguments separated by commas get **one space** between them
- Don't want the newline? Pass end

> print 这个词的字面意思是「打印」，理解成一支笔在纸上移动很贴切。
>
> 两个默认行为要记住：参数之间自动加空格，结尾自动换行。这两个默认值都可以改：空格用 sep 参数，换行用 end 参数。
>
> 想让两次输出接在同一行，只要 end="" 就够了，不必绕别的办法。

print(): Comma, Plus, End

print arguments

```
1  print("a", "b", "c")       # spaces
2  print("a" + "b" + "c")     # no spaces
3  print("no newline", end="")
4  print(" <- same line")
```

output

```
a b c
abc
no newline <- same line
```

逗号和加号是两件不同的事：逗号**分隔参数**，加号**拼接字符串**。

42

第 42 页 · print(): Comma, Plus, End

> 第一行和第二行的区别要看清楚：逗号把三个值作为三个参数交给 print，print 在它们之间补空格；加号先把三个字符串拼成一个，再作为一个参数交给 print，中间没有空格。
>
> 还有一个区别更要紧：加号两边必须都是字符串。`print("年龄" + 18)` 会报错，而 `print("年龄", 18)` 没问题。所以在不确定类型的时候，用逗号更安全。
>
> 最后两行演示 end：把结尾的换行换成空字符串，下一次 print 就接在同一行。

Numbers

Integers Without a Ceiling

big integers

```
big = 2 ** 200
print(big)
print("that is", len(str(big)), "digits")
```

output

```
1606938044258990275541962092341162602522202993782792835301376
that is 61 digits
```

只受内存限制。**你不用操心溢出。**

44

第 44 页 · Integers Without a Ceiling

> 大多数语言里整数有固定长度，比如 64 位。超出范围就会溢出，数字突然变成负数，或者绕回去，而且通常不报错。这是 C 和 Java 里很难查的一类 bug。
>
> Python 没有这个问题。整数要多大有多大，解释器会自动用更多内存来存它。
>
> 代价是大整数运算会变慢：位数越多，一次乘法要做的工作越多。第 11 讲讲复杂度时会量一下这个代价具体是多少。对这门课的绝大多数题目来说，这个代价可以忽略。

Back to the Opening Question

print more digits

```
print(0.1 + 0.2)
print(f"{0.1:.20f}")
print(f"{0.2:.20f}")
print(f"{0.1 + 0.2:.20f}")
```

output

```
0.30000000000000004
0.10000000000000000555
0.20000000000000001110
0.30000000000000004441
```

`0.1` 存进去实际是 **0.10000000000000000555 11...**。问题不在加法，在**存**这一步。

45

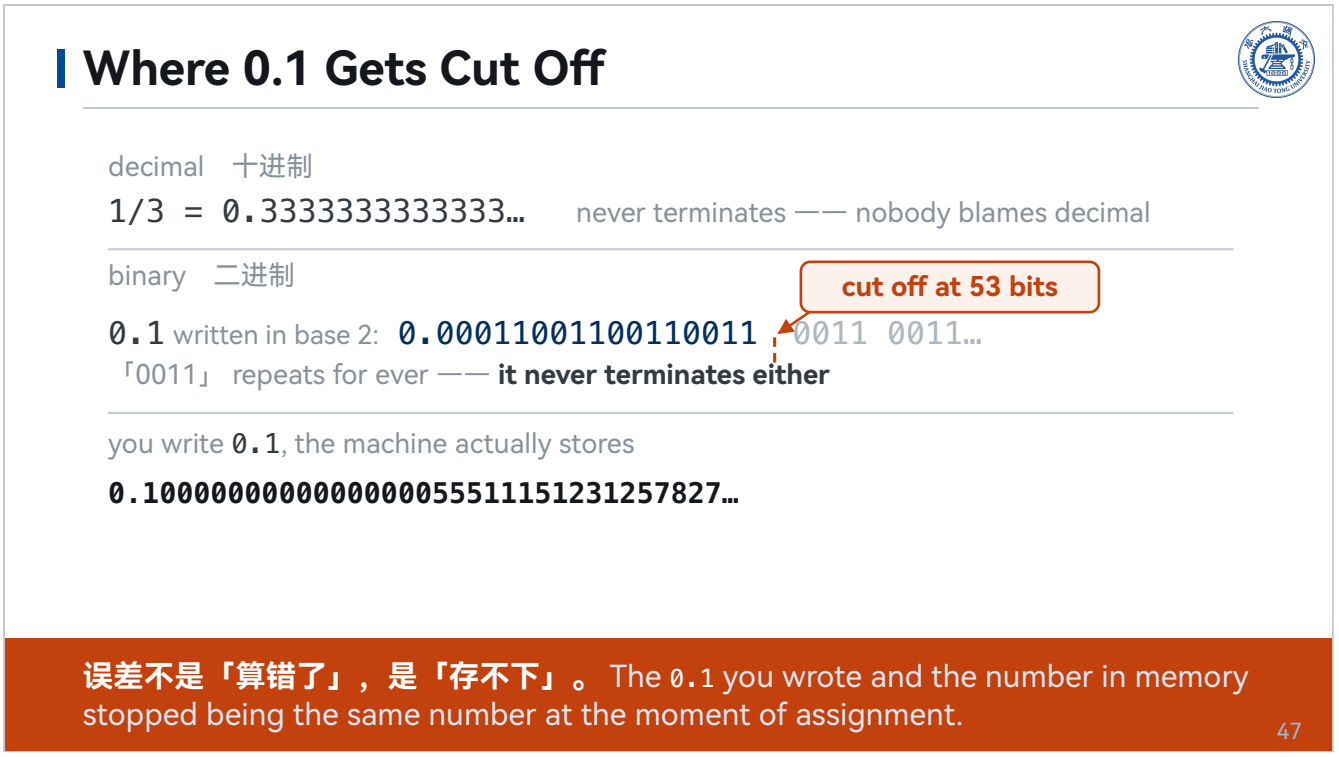
> 把小数点后 20 位都打出来，就能看到真相：0.1 存进去的时候已经不是 0.1 了，它是一个非常接近 0.1 的数。0.2 同理。
>
> 所以加法本身没有出错，它把两个近似值精确地加了起来，得到的当然也是一个近似值，而且误差正好大到能显示出来。
>
> `f"{x:.20f}"` 这个写法叫格式化字符串，第 3 讲会正式讲。现在只要知道 `.20f` 的意思是「按小数点后 20 位显示」。

Why Binary Cannot Hold 0.1

- In decimal, 1/3 = 0.3333… never terminates
- In binary, **0.1 never terminates either**: 0.000110011001100… (0011 forever)
- Storage is finite — 64 bits, following the **IEEE 754** standard — so it is cut off
- What gets stored is not 0.1 but a number very close to it

换任何一种主流语言都一样。**这是二进制本身的性质，不是 Python 的缺陷。**

> 二进制写不下 0.1，和十进制写不下 1/3 是同一个道理。
>
> 一个分数能不能在某个进制下写完，取决于它的分母。十进制的分母是 10 = 2×5，所以分母只含 2 和 5 的分数写得完，1/3 写不完。二进制的分母是 2，所以只有分母是 2 的幂的分数才写得完：0.5、0.25、0.125 可以，0.1（也就是 1/10，分母含 5）不行。
>
> 既然写不完，而存储空间有限，就只能在某一位截断。截断产生的那点误差，就是后面所有意外的来源。
>
> IEEE 754 是这套截断规则的国际标准，几乎所有硬件都实现它，所以换语言、换机器结果都一样。



第 47 页 · Where 0.1 Gets Cut Off

把上一页那句话画出来看。

最上面一行是类比：十进制写不完三分之一，没有人会因此说十进制有 bug。

中间那行是同一回事换到二进制：0.1 写成二进制是 0.0001100110011...，「0011」这四位一直重复下去，永远写不完。

而存储空间是有限的：double 的尾数一共 53 位，所以到那根橙色的虚线处就剪断了。剪断这个动作，就是误差的全部来源。

最下面一行是剪断之后实际存进去的数：0.1000000000000000055511151231257827...。注意它比 0.1 **大**了一点点。这个「一点点」下一页会派上用场：0.1 偏大、0.3 偏小，两个方向相反的误差凑在一起，就可能正好抵消。

所以请把说法改过来：不是浮点数「算错了」，是它**存不下**。你写的 0.1 和机器里那个数，从赋值那一刻起就不是同一个数了。

When the Error Cancels

four lines that look alike

```
print(0.1 + 0.2 == 0.3)
print(0.1 + 0.3 == 0.4)
print(0.2 + 0.4 == 0.6)
print(0.1 + 0.7 == 0.8)
```

output

```
False
True
False
False
```

第二个是 **True**。**碰巧相等确实存在，但你事先不知道是哪些。**

如果你自己去试别的组合，很快会撞上这一页的情况：有的式子居然是 True。

0.1 + 0.3 == 0.4 成立，0.1 + 0.2 == 0.3 不成立，0.2 + 0.4 == 0.6 又不成立。这四个式子从数学上看是同一类，结果却不一样。

这不是随机的，下一页会讲清楚机制。但要先说结论：**恰好相等确实存在，可是你没有办法在写代码的时候预判哪些会恰好相等。**一个只在部分输入上成立的判断，比从不成立的判断更危险：它会通过你的测试，然后在别的数据上出错。

Why 0.1 + 0.3 == 0.4 Is True

lay all four out

```
print(f"{0.1:.20f}")
print(f"{0.3:.20f}")
print(f"{0.1 + 0.3:.20f}")
print(f"{0.4:.20f}")
```

output

```
0.10000000000000000555
0.29999999999999998890
0.40000000000000002220
0.40000000000000002220
```

0.1 is stored a little **high**, 0.3 a little **low**. The two errors point opposite ways and cancel.

> 截断产生的误差有方向：可能偏大，也可能偏小，取决于那个数在二进制里被截断的位置。
>
> 0.1 存进去偏大（...00555），0.3 存进去偏小（...98890）。两者相加，一个多出来的部分正好被另一个少掉的部分吃掉，和落回到了 0.4 存进去的那个数上，于是 == 成立。
>
> 对比 0.1 + 0.2：0.1 偏大，0.2 也偏大，误差同向叠加，和是 0.30000000000000004441，而 0.3 本身存的是 0.29999999999999998890，两个数不同，== 就是 False。
>
> 所以这里没有玄学，每一步都是确定的。但要在写代码前算出两个误差会不会抵消，代价远大于直接改用容差比较。

Comparing Floats（浮点数的比较）

错

```
if x == 0.3:
    ...

# risky: x may be
# 0.30000000000000004
```

对

```
if abs(x - 0.3) < 1e-9:
    ...

# ask if they are close enough
```

1e−9 is a tolerance 容差; pick it from the precision you need. Asking whether two floats are equal really means asking how far apart they are.

碰巧相等不等于可以依赖。 只要是浮点数，就不要用 ==。

既然存进去的值本身就带误差，那么两个理论上相等的浮点数，实际存的可能是两个不同的数。用 == 比较，结果不可预测。

正确的做法是判断差值的绝对值是否小于一个很小的数。这个数叫容差，多小合适取决于你的场景：科学计算可能取 1e-12，算钱可能根本就不该用浮点数。

注意这条规则只针对浮点数。整数、字符串、布尔值用 == 比较是完全正确的。

Accumulated Error

add 0.1 ten times

```
total = 0.0
for _ in range(10):
    total += 0.1
print("after ten additions:", total)
print("is it 1.0?", total == 1.0)
```

output

```
after ten additions: 0.9999999999999999
is it 1.0? False
```

Ten invisible errors add up to **0.9999999999999999**.

51

> 单次误差小到看不见，但它会累积。这里加十次，误差已经大到肉眼可见了。
>
> 想象一个记账程序，每笔金额都有这么一点误差，一天几千笔，到月底对账的时候就会出现「差几分钱找不到」的情况。这在真实系统里是很常见的事故。
>
> 解决办法在下一页。

A RULE OF THUMB

Integers Wherever Possible

- **Money** is the classic case: `0.10 + 0.20` will bite you
- Count in cents instead: `10 + 20 = 30`, **exact, always**
- Divide by 100 only at the moment you display it
- If you truly need exact decimals, the standard library has `decimal` — much slower

Python 整数没有上限，所以「换成更小的单位」这条路基本上总能走通。

第 52 页 · Integers Wherever Possible

> 这是一条很实用的原则：把小数换算成整数来处理。
>
> 钱是最典型的例子。以「分」为单位，所有金额都是整数，加减乘全部精确，只在最后显示给人看的时候除以 100。真实的支付系统基本都是这么做的。
>
> 这条路在 Python 里尤其好走，因为整数没有上限，不用担心换成更小的单位之后数值超出范围。
>
> 如果确实需要十进制的精确小数运算，标准库里有 decimal 模块。这门课不深入，但你要知道有这个东西。

Arithmetic

Seven Arithmetic Operators

arithmetic

```python
a, b = 7, 2

print(a + b, a - b, a * b)
print(a / b)     # true division
print(a // b)    # floor division 整除
print(a % b)     # remainder 取余
print(a ** b)    # power
```

output

```
9 5 14
3.5
3
1
49
```

54

第 54 页 · Seven Arithmetic Operators

> 前三个和数学里一样。后四个需要留意。
>
> / 是数学意义上的除法，结果是 7/2 = 3.5。 // 是整除，结果向下取整，7 除以 2 商 3。 % 是取余，7 除以 2 余 1。 ** 是幂，7 的 2 次方是 49。
>
> 第一行的 a, b = 7, 2 叫同步赋值，一次给两个变量赋值。它比写两行更短，而且在交换两个变量时特别有用：a, b = b, a 一行就能交换，不需要中间变量。

TYPE RULES

Type Rules: the One Table to Memorize

- `/` — **always returns a float**, even when it divides evenly `4 / 2` → `2.0`
- `//` — int if both are int; float if either is float
- `%` — same as `//`
- `+ - *` — int if both are int; float if either is float
- `**` — same, except a **negative exponent gives a float** `2 ** -1` → `0.5`

> 这张表决定了你写的每一个表达式返回什么类型，而类型会在取下标、比较、格式化输出的时候暴露出来。
>
> 规律其实只有一条：**只要参与运算的有一个是浮点数，结果就是浮点数**。唯一的例外是 `/`，它无论如何都返回浮点数。
>
> `**` 的负指数那一条也好理解：2 的 -1 次方是 0.5，本来就不是整数，只能用浮点数表示。

The Type Rules, Checked

type() reports back

```
1  print(4 / 2,    type(4 / 2))
2  print(7 // 2,   type(7 // 2))
3  print(7.0 // 2, type(7.0 // 2))
4  print(2 ** -1,  type(2 ** -1))
```

output

```
2.0 <class 'float'>
3 <class 'int'>
3.0 <class 'float'>
0.5 <class 'float'>
```


> `type()` 返回一个值的类型，用它可以直接验证上一页的规则。
>
> 第一行最值得注意：4 除以 2 明明除得尽，结果却是 2.0 而不是 2，类型是 float。这是 Python 3 特意做的改动。Python 2 里 `4 / 2` 得到 2，结果很多人在写整数除法时踩坑。
>
> 第三行说明只要有一个操作数是浮点数，整除的结果也是浮点数，虽然它的值 3.0 看起来像整数。

The Float From /

错

```
a = [10, 20, 30]
mid = len(a) / 2
print(a[mid])

# TypeError
# an index must be an int
```

对

```
a = [10, 20, 30]
mid = len(a) // 2
print(a[mid])

# 20
```

`len(a) / 2` is `1.5`. And even for an even length, `4 / 2` is `2.0`, not 2 — still not usable as an index.

需要整数结果就用 `//`，或者用 `int()` 转一下。

> 这是 / 返回浮点数最常造成的实际问题：拿它算出来的值去取下标。
>
> 下标必须是整数，浮点数会直接报 TypeError。麻烦的是这个错误只在运行到那一行时才出现，如果那一行藏在某个分支里，可能测试了很久才碰到。
>
> 求中点、求一半、求平均之后取整，这些场合都应该用 //。第 5 讲讲列表切片时会大量用到。

Floor Division and Remainder on Negatives

negatives

```
print(7 // 2)      # 3
print(-7 // 2)     # guess: -3 or -4?
print(7 % 2)
print(-7 % 2)      # guess this one too
```

output

```
3
-4
1
1
```

// 是**向下取整**，不是「把小数部分截掉」。对负数来说这两者不同。

> −7 / 2 是 -3.5。向下取整意味着往更小的方向走，所以是 -4，不是 -3。如果理解成「把小数部分截掉」，就会答成 -3。
>
> 取余的结果跟着来：Python 保证 a == (a // b) * b + a % b 恒成立。代入 a=-7、b=2：(-4)×2 + 余数 = -7，所以余数是 1。
>
> 这一点和 C、Java 不同，那些语言里 −7 % 2 是 -1。Python 的做法保证余数的符号跟着除数走，在处理「循环下标」这类问题时更方便：i % n 的结果永远落在 0 到 n-1 之间。

Precedence（优先级）

** binds tightest, then * / // %, then + −. Equal ranks go left to right — except **, which goes **right to left**.

precedence

```
print(2 + 3 * 4)        # 14, not 20
print(2 ** 3 ** 2)      # 512, not 64
print((2 + 3) * 4)      # 20
```

output

```
14
512
20
```

拿不准优先级，就加括号把顺序写明白。多写不扣分，错了很难查。

59

第 59 页 · Precedence（优先级）

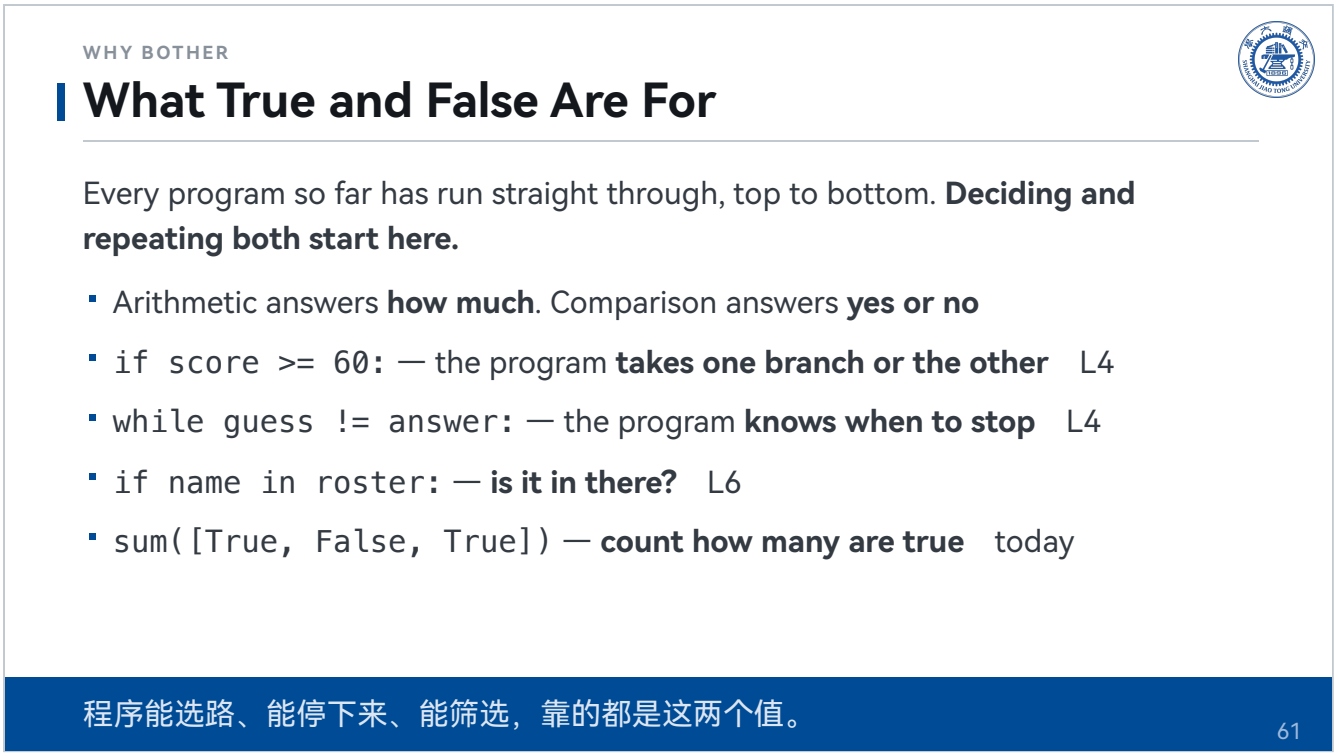
前两条和数学里一样，不容易出错。

第二行值得注意：2 ** 3 ** 2 是从右往左算的，先算 3 的 2 次方得 9，再算 2 的 9 次方得 512。如果从左往右算就是 $(2^3)^2 = 64$。这个方向和数学约定一致。

实际写代码的建议是：不确定就加括号。没有人会因为你多写了括号而扣分，但优先级搞错造成的 bug 很难发现，因为代码看起来完全正常。括号是写给读代码的人看的，包括三个月后的你自己。

PART 6 OF 6

Logic



第 61 页 · What True and False Are For

在讲 True 和 False 是什么之前，先说清楚要它干什么，不然这一段听起来只是又一种类型。

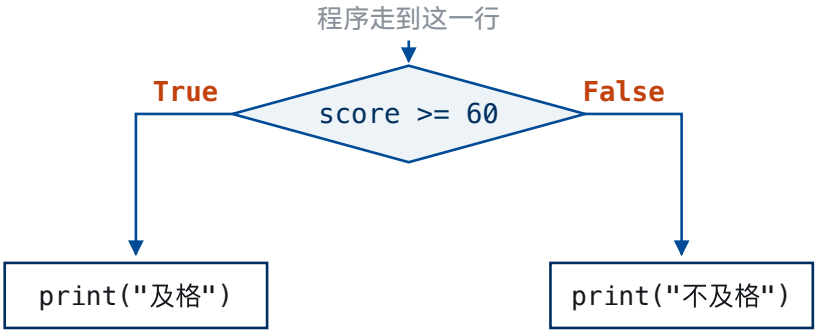
到今天为止，我们写的程序都是从第一行走到最后一行，一条路走到底，换什么输入都走同一条路。从第 4 讲开始，程序会**分支**、会**重复** —— 而每一个 if 后面的条件、每一个 while 后面的条件，**求值的结果都必须是 True 或 False**。

片子上四行就是最常见的四种用法：判断分数决定走哪个分支；判断猜没猜对决定循环停不停；判断名字在不在名单里；以及把一串判断加起来，数出「满足条件的有几个」——最后这个今天就会见到。

所以这一段是下一讲的地基。比较运算符本身很简单，值得花时间的是它的结果怎么被用起来。

One Condition, Two Paths

The condition is evaluated **first**. Its value — `True` or `False` — is what picks the path.



菱形里那句话**算出来的就是一个 True 或 False**。score 是 87 走左边，是 45 走右边。

先算出一个 True 或 False，再按它选路。第 4 讲写 if 就是这个图。

62

第 62 页 · One Condition, Two Paths

把上一页第一条用处画出来看。

程序走到这一行，先做菱形里那件事：把 score 和 60 比一下。**比出来的不是「及格」也不是「不及格」，是一个 True 或者 False。**

然后程序拿着这个值选路：True 走左边，False 走右边。score 是 87，比较的结果是 True，走左边打印「及格」；score 是 45，结果是 False，走右边。

所以这里是两步，不是一步：**先求值，再选路**。很多人一开始会把它想成「程序看了一眼分数就知道该走哪边」，中间那个布尔值被跳过去了。等到第 4 讲条件写复杂了 —— 比如 score >= 60 and attendance >= 0.8 —— 看得出中间那个值是什么，才查得出问题。

顺带说一下这两个形状：**菱形是判断，方框是动作**。这是流程图的通用画法，第 4 讲讲 if 和 while 时会一直用。

Comparison and Boolean（布尔值）

comparison

```
1  print(3 > 2)
2  print(3 == 2)      # two signs: "is it equal?"
3  print(3 != 2)      # not equal
4  print(3 >= 3)
```

output

```
True
False
True
True
```

Two values only: `True` and `False`. **首字母必须大写**，写成 true 会报 NameError。

63

第 63 页 · Comparison and Boolean（布尔值）

> 比较运算的结果是布尔值（bool），只有 True 和 False 两种。
>
> 注意首字母必须大写。写成 true 或 TRUE 都会报 NameError，因为 Python 会把它当成一个没定义过的变量名。
>
> 比较运算符有六个：> < >= <= == !=。其中 != 是「不等于」，感叹号在很多语言里都表示「非」。

= and ==（赋值与相等）

错

```
if x = 5:
    ...

# SyntaxError
# Python stops you here
```

对

```
if x == 5:
    ...

# asks whether x is 5
```

= **assigns**: it puts the value on the right into the name on the left. == **compares**: it asks whether the two sides are equal, and gives back True or False.

Python 在 if 里写 = 会直接报语法错误。**C 和 Java 不会**。在那些语言里，这个错误会悄悄改掉你的变量。

> 这是初学者最经典的错误之一。好消息是 Python 对它很严格：在 if 条件里写单个等号会直接报 SyntaxError，当场就发现了。
>
> C 和 Java 里 `if（x = 5）`是合法的：它先把 5 赋给 x，再判断 5 是不是真值，结果永远为真，而且你的 x 被悄悄改掉了。这类 bug 在那些语言里造成过很多事故。
>
> Python 把这个口子堵上了，这是它对初学者友好的地方之一。

and, or, not（与或非）

logical operators

```
1  age = 19
2  has_id = True
3
4  print(age >= 18 and has_id)
5  print(age < 18 or has_id)
6  print(not has_id)
```

output

```
True
True
False
```

Python 用英文单词，不是 && || !。

第 65 页 · and, or, not（与或非）

> 三个逻辑运算符用的是英文单词，这一点和 C、Java、JavaScript 不同，那些语言用 && || !。写 Python 时用错符号会报语法错误。
>
> 含义和日常语言一致：and 要求两边都为真，or 只要有一边为真，not 取反。
>
> 优先级是 not 最高，然后 and，最后 or。混用时建议加括号，理由和算术运算一样。

True and False Are 1 and 0

booleans in arithmetic

```
1  print(True + True)
2  print(sum([True, False, True, True]))
```

output

```
2
3
```

`sum()` counts the `True`s — the usual way to ask how many items satisfy a condition.

第 66 页 · True and False Are 1 and 0

> 在 Python 里 bool 是 int 的一个子类型，True 就是 1，False 就是 0。所以它们可以直接参与算术运算。
>
> `True + True` 得到 2 初看有点奇怪，但第二行是实用写法：把一串判断的结果放进列表，用 sum 一加，就得到「满足条件的有几个」。
>
> 第 5 讲讲列表推导时，这个写法会以更简洁的形式出现，比如统计一个班里及格的人数。

第 67 页 · 随堂小测 · 第 2 轮

第二轮小测，三道题。我一题一题地放，每题答完当场公布。

【第 1 题 · 0b1010 + 10 等于多少？】

这一题检验有没有真的接受「进制只是写法」。

0b1010 是二进制的 1010，也就是 8 + 2 = 10；后面那个 10 本来就是十进制的 10。两个都是整数 10，相加得 20。如果有人答 1020 或者 101010，说明还把前缀当成了数的一部分。

【第 2 题 · len("a\nb") 是多少？】

答案是 3。反斜杠加 n 合起来是一个字符（换行符），不是两个。

选 4 的人是把源代码里的写法和字符串里实际存的内容混为一谈了。反斜杠只存在于源代码里，是写给 Python 看的记号。

【第 3 题 · -7 // 2 和 -7 % 2 分别是多少？】

答案是 -4 和 1。

答错的多半是把 // 理解成了「去掉小数部分」。公布答案时用 a == (a // b) * b + a % b 这个恒等式验一遍，比单纯记结论好。

ANSWERING THE OPENING

Three Conventions, Three Costs

- **ASCII, then Unicode** · the convention that turns bits into letters
  - Cost: everyone has to agree on the same table, and 128 slots were never going to be enough
- **IEEE 754** · the convention that turns bits into decimals
  - Cost: **accuracy**. `0.1` is a repeating binary fraction, so it gets cut off
- **Types** · the convention that decides what an operator means
  - Cost: you have to know which type comes back. `/` is not `//`

L1 说一串 bit 本身没有意义，意义来自约定。**三套约定，三样代价。**

68

第 68 页 · Three Conventions, Three Costs

回到今天开头那个问题。L01 结尾说：一串 bit 本身没有意义，意义来自约定。今天讲的就是三套具体的约定，而每一套都是有代价的。

第一套是 ASCII，后来是 Unicode，把 bit 变成字母。代价有两层：所有人必须用同一张表，否则同一份文件在两台机器上显示出来是两回事；而且 128 个位置从一开始就不够用，中文一个字都放不进去。

第二套是 IEEE 754，把 bit 变成小数。代价是精度：0.1 在二进制下是无限循环，存的时候必然被剪断。今天开场那两行代码，全部原因就在这里。

第三套是类型，它决定同一个运算符是什么意思。代价是你得知道每个表达式返回什么类型，不然 / 和 // 的差别会在取下标的时候直接报错。

所以「一切都是约定」不是一句空话。约定由人来定，就必然有取舍，而取舍的代价迟早会在某个地方显出来。今天这三套，代价分别是兼容性、精度和类型。

RECAP

What to Take Away

- A **type** decides what a value can do: str, int, float, bool are the four you will meet most
- The four bases **are just notations** — once in memory they are the same bits
- `ord()` / `chr()` convert both ways. **`'0'` and `0` are different things**
- Python **integers have no ceiling** — overflow is not your problem
- **Floats are not exact.** Never compare them with ==. Prefer integers
- **`/` always gives a float**; `//` floors, and on negatives that surprises people

> 六条，其中第五条和第六条是这一讲最容易在实际写代码时出问题的。
>
> 第三条的 `'0'` 和 `0`，在第 3 讲讲 input 时会立刻用到：input 拿到的永远是字符串。
>
> 第六条的 `/` 返回浮点数，在第 5 讲取列表下标时会立刻用到。
>
> 其余几条更多是理解层面的：知道为什么，就不会在遇到怪现象时怀疑机器坏了。

LAB 2

Lab 2 — 六道题，逐条验今天讲的规则

- **2-1** 用转义字符画一只熊，反斜杠和引号都得转对
- **2-2** 大小写转换：**只能用 `ord()` 和 `chr()`**，不许 `.lower()`（平台判分）
- **2-3** 一个字符的一生：不用函数也不用循环，把它一路变下去
- **2-4** 海伦公式算三角形面积
- **2-5** `almost_equal(a, b, eps=1e-9)`：浮点数怎么算「相等」（平台判分）
- **2-6** 类型预测：**先写下结果和类型再运行**，记下猜错的

另有 **B 段**（一道大题分几问）和 **C 段**（命令行基本功，**期末要考**）。**2-6 先猜再跑**。猜错的那几道，往往记得最久。

A 段六道题，重点在 2-6 和 2-5。

2-6 要求先猜再验。这个顺序很重要：直接跑一遍看到结果，很容易「看懂了」就过去了；先猜错再看到正确答案，才会记住。把猜错的那几个单独记下来，那才是真正需要补的地方。

2-2 和 2-5 是平台判分的，写成函数交上去当场出分。2-2 限定只能用 ord() 和 chr()，就是要求用今天讲的字符编码，而不是调一个现成的方法。

2-5 对应今天浮点数那一段：既然不能用 == 比浮点数，那就得自己定义「差得足够小就算相等」。这个函数以后会反复用到。

A 段之外还有 B 段一道大题、C 段的命令行基本功。C 段每周一小段，期末要考，别跳过。三段同时开放，不必按顺序做完。

交到「小作业 1」，它覆盖前三周，10 月 10 日截止。

Next Lecture

You can compute now. But how does the program hear you?

L3: strings, variables, input and output. You will see that input() always hands back a string — and why that makes a beginner's first program print something very strange.

> 下一讲把这一讲的东西接到真实的输入上。
>
> 开场用的是这样一个现象：写一个加法程序，让用户输入两个数，输入 3 和 4，程序输出 34。原因就是 input 拿到的是字符串，而字符串的加号是拼接。
>
> 这一讲讲过的 `'0'` 和 `0` 的区别，下一讲会立刻派上用场。