上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Introduction to Computation (CS1602) · Lecture 2

Characters and Numbers

Instructor: Tao Huang

Part I: Foundations and Python Basics · Fall 2026

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。

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。

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

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。

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.

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

PART 1 OF 6

A Complete Program

L1 left one question open: a bit pattern means nothing on its own.

First, though: **what does a whole program even look like?**

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
```

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。

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。

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。

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。

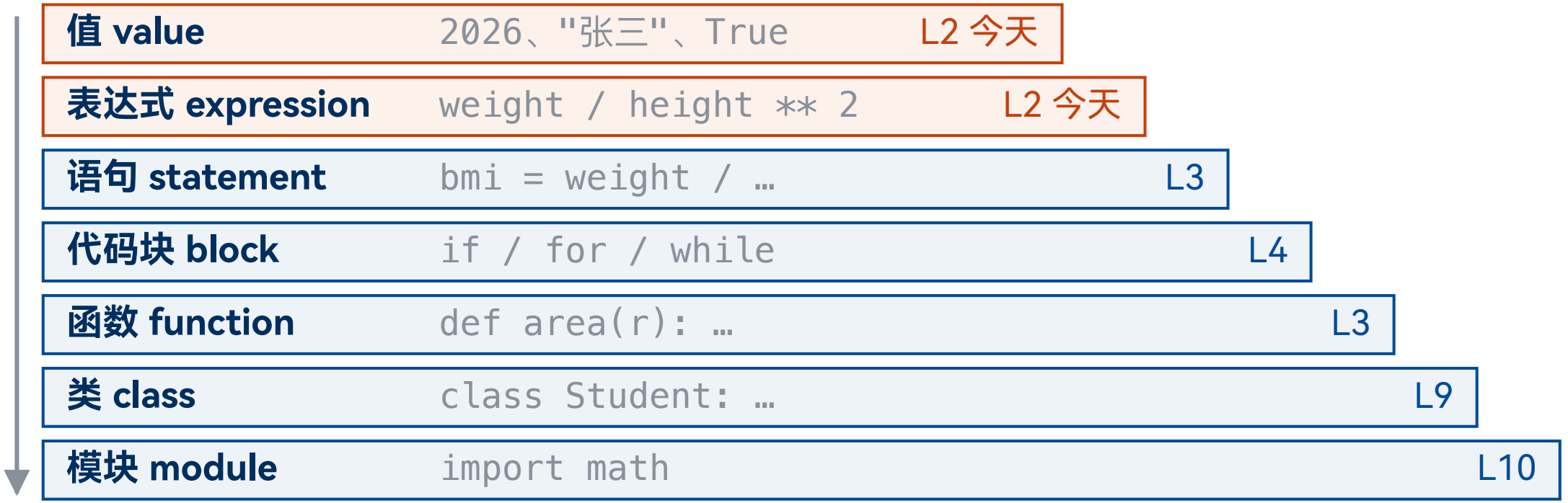
The Three Parts of a Program

get data · compute · report

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

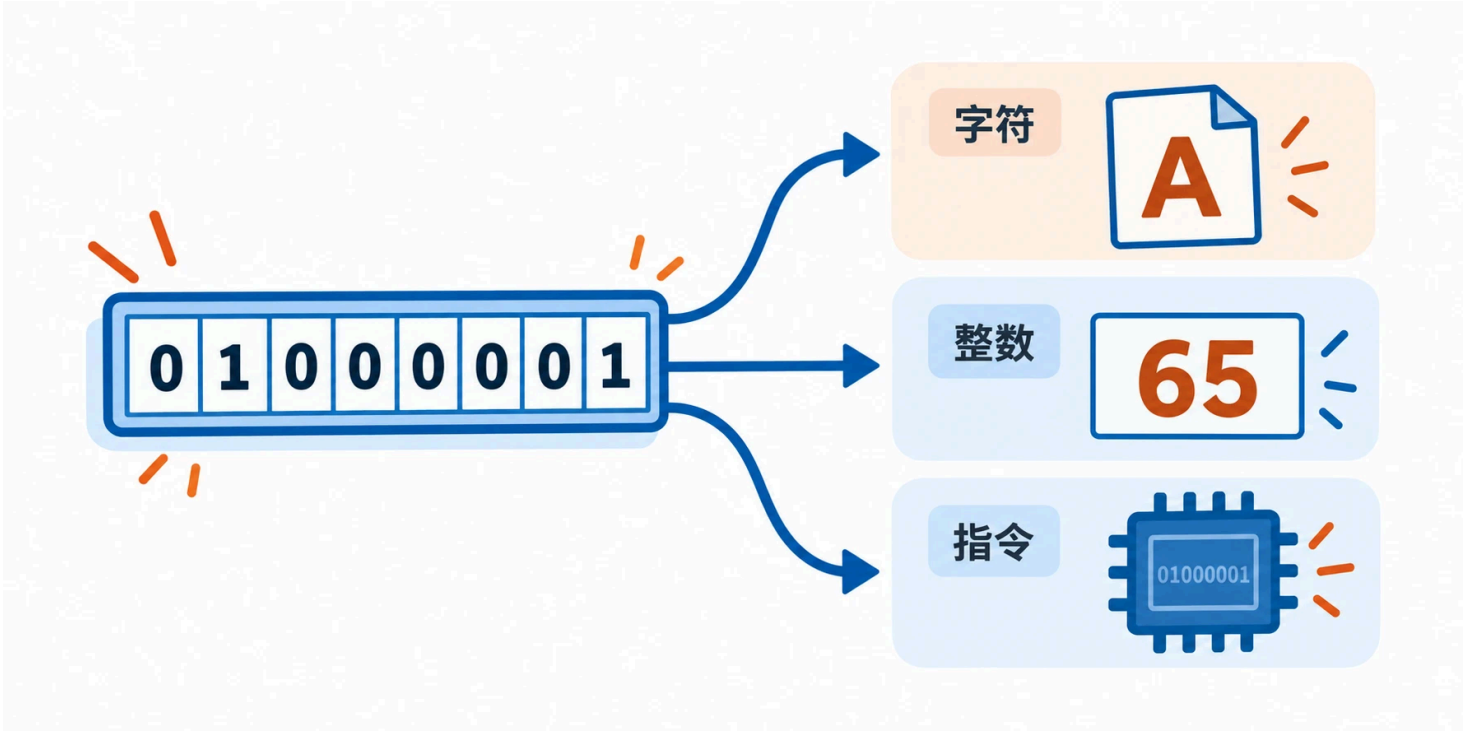
From Values to Modules

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。

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？

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。**

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))
```

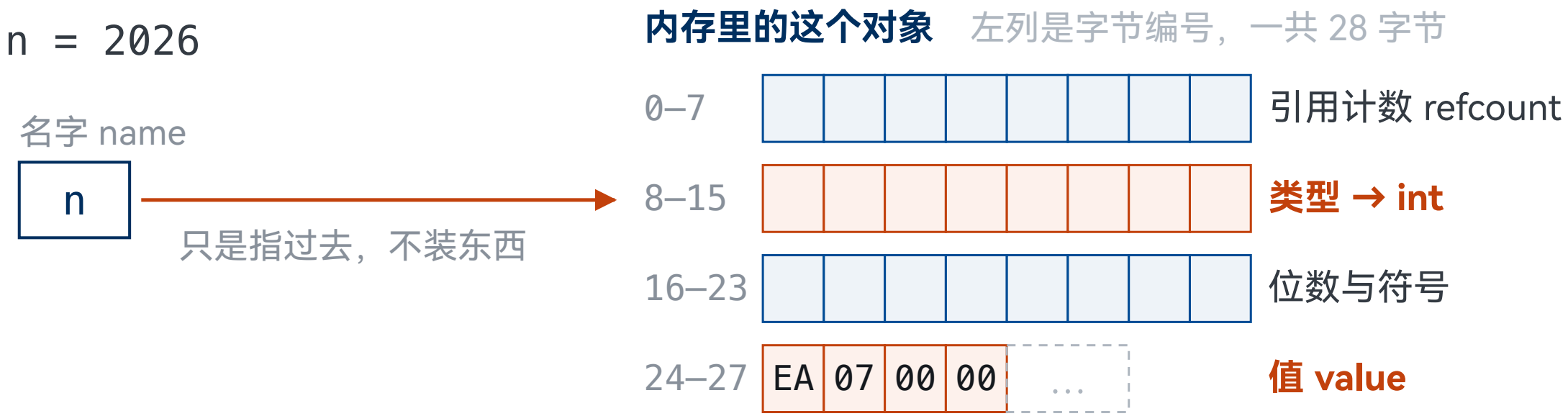
output

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

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

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。

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**.

PART 2 OF 6

Bits and Bases

Settled: every value has a **type**, and the type decides what an operator means.

Now go one level down: **underneath, all of it is bits.**

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.

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。

Base Conversion in Python

base conversion

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

output

```
13
0b1101
13
```

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.

The Four Literals, Checked

four notations, one number

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

output

```
96 96 96 96
True
```

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。

PART 3 OF 6

Characters

Settled: a base is only **notation** — the bits underneath never change.

So a machine that holds numbers: **how do letters get in?**

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.

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.

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。

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

ord / chr

```
1  print(ord('A'), ord('a'), ord('0'), ord(' '))
2  print(chr(65), chr(97), chr(48))
3
4  # the cases are 32 apart, so this works
5  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.

'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。

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.

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

```
1  # 一次 print，两行输出。怎么写?
2  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。**

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**.

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"
```

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。**

Multi-line strings 三引号

triple quotes

```
1  poem = """江南好，风景旧曾谙。
2  日出江花红胜火，
3  春来江水绿如蓝。"""
4
5  print(poem)
6  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。

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。

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(): 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**。

PART 4 OF 6

Numbers

Settled: letters get in by **convention** — ASCII, then Unicode.

Numbers need conventions too, and **this time they cost you accuracy.**

Integers Without a Ceiling

big integers

```
1  big = 2 ** 200
2  print(big)
3  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。**

Back to the Opening Question

print more digits

```
1  print(0.1 + 0.2)
2  print(f"{0.1:.20f}")
3  print(f"{0.2:.20f}")
4  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。

Why Binary Cannot Hold 0.1

- In decimal, 1/3 = 0.3333… never terminates
- In binary, **0.1 never terminates either**: 0.0001100110011001100… (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。**

Where 0.1 Gets Cut Off

decimal　十进制

`1/3 = 0.333333333333…`　never terminates —— nobody blames decimal

---

binary　二进制

`0.1` written in base 2: `0.0001100110011001` `1001 1001…`

cut off at 53 bits

「0011」 repeats for ever —— **it never terminates either**

---

you write `0.1`, the machine actually stores

`0.1000000000000000055511151231257827…`

误差不是「算错了」，是「存不下」。 The `0.1` you wrote and the number in memory stopped being the same number at the moment of assignment.

When the Error Cancels

four lines that look alike

```
1  print(0.1 + 0.2 == 0.3)
2  print(0.1 + 0.3 == 0.4)
3  print(0.2 + 0.4 == 0.6)
4  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。

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.

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 ==。

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**.

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。

PART 5 OF 6

Arithmetic

Settled: integers are exact and unbounded; floats are neither.

So the question for every expression you write: **which of the two comes back?**

Seven Arithmetic Operators

arithmetic

```
1  a, b = 7, 2
2
3  print(a + b, a - b, a * b)
4  print(a / b)      # true division
5  print(a // b)     # floor division 整除
6  print(a % b)      # remainder 取余
7  print(a ** b)     # power
```

output

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

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`

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'>
```

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。

Floor Division and Remainder on Negatives

negatives

```
1  print(7 // 2)      # 3
2  print(-7 // 2)     # guess: -3 or -4?
3  print(7 % 2)
4  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。

Precedence（优先级）

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

precedence

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

PART 6 OF 6

Logic

Arithmetic hands back a number. **Comparison hands back one of two values.**

And those two turn out to be numbers as well.

What True and False Are For

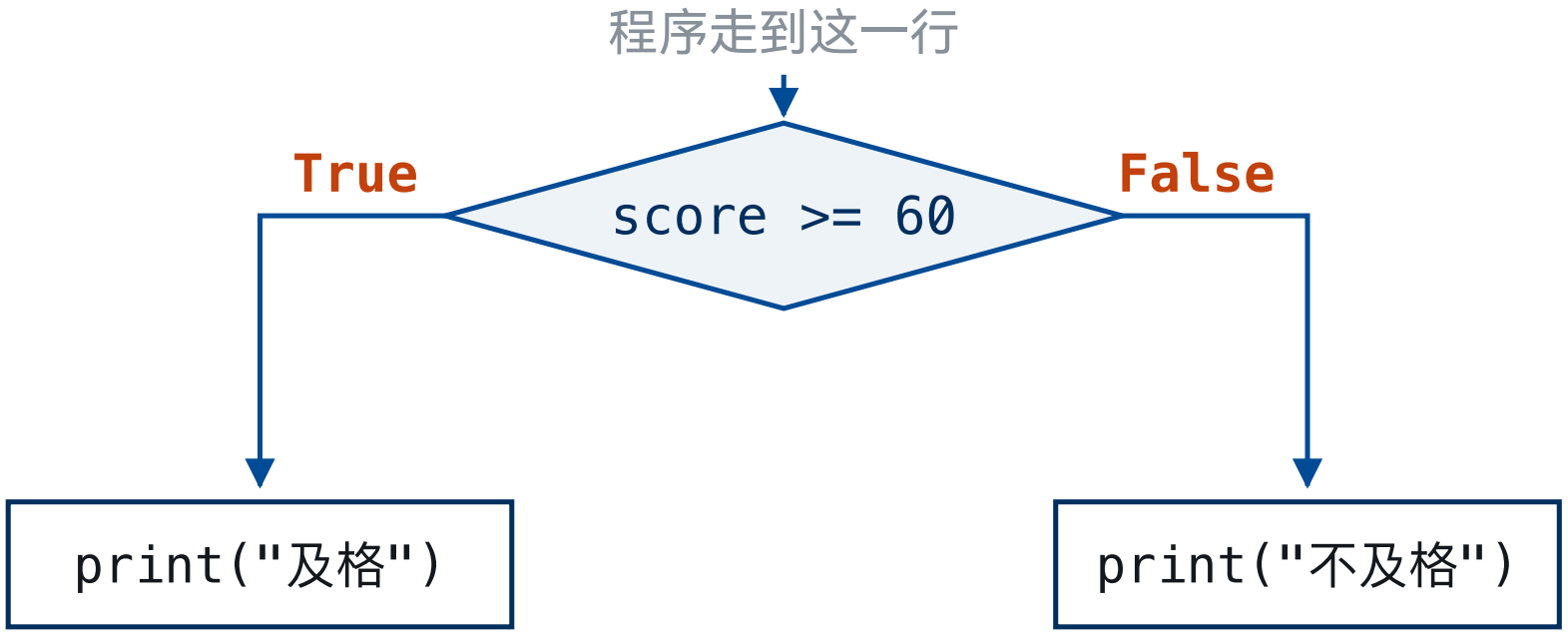
Every program so far has run straight through, top to bottom. **Deciding and repeating both start here.**

- Arithmetic answers **how much**. Comparison answers **yes or no**
- `if score >= 60:` — the program **takes one branch or the other** L4
- `while guess != answer:` — the program **knows when to stop** L4
- `if name in roster:` — **is it in there?** L6
- `sum([True, False, True])` — **count how many are true** today

程序能选路、能停下来、能筛选，靠的都是这两个值。

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。

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。

= 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。

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 && || !。

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.

随堂小测 · 第 2 轮

扫码作答 · 这一轮三题

1. 0b1010 + 10 等于多少？
2. len("a\nb") 是多少？
3. -7 // 2 和 -7 % 2 分别是多少？

今天三段各收一题。一题一题放，答完一题公布一题。

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

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。**

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

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。

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.