上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Introduction to Computation (CS1602) · Lecture 3

Strings, Variables, Input and Output

Instructor: Tao Huang

Part I: Foundations and Python Basics · Fall 2026

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。

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。**

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

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。**

OVERVIEW

Today

Let the program talk to a person. Then learn to organize the code.

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

PART 1 OF 7

Variables

L2 settled what a **value** is: a type, and the operators that go with it.

Now values need **names**, and = is not the equals sign from mathematics.

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
```

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。

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()`.

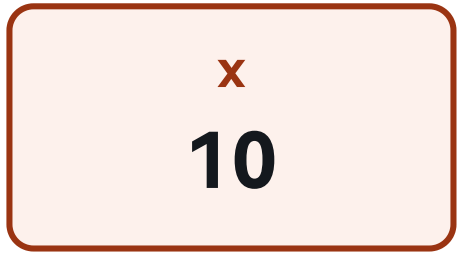
What a Variable Really Is

- 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。

Not a box — a label

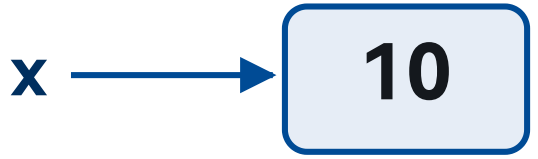
✗ a box: x holds 10 inside



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。

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。

== 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。

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.

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。**

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.

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。**

Augmented Assignment（复合赋值）

+= -= *= //=

```python
x = 10
x += 3        # same as x = x + 3
print(x)

x -= 5
x *= 2
x //= 3
print(x)
```

output

```
13
5
```

+= −= ∗= /= //= %= ∗∗= all work this way.

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。**

PART 2 OF 7

Strings

Settled: a name is a **label stuck on a value**, not a box.

Of all the types, one carries every word between program and person: **the string.**

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.

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.**

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。**

Length, index, slice

picking characters

```
1  s = "Python"
2  #     P  y  t  h  o  n
3  #     0  1  2  3  4  5
4
5  print(len(s))        # how many characters
6  print(s[0])          # the first; indices start at 0
7  print(s[-1])         # the last
8  print(s[2:5])        # 2, 3, 4 — 5 not included
```

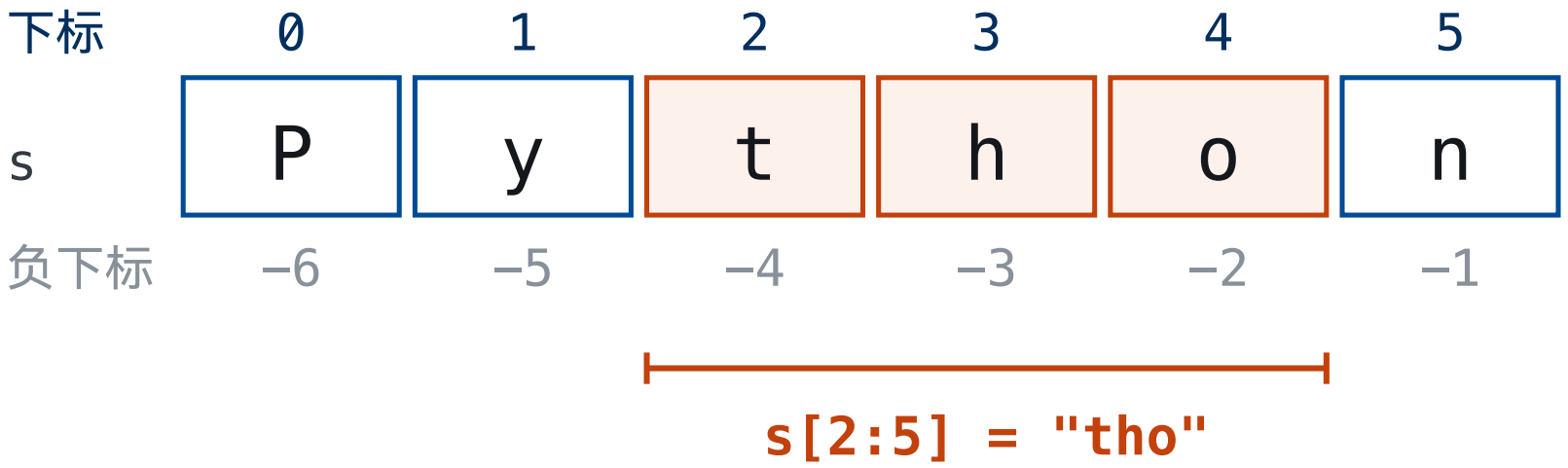
output

```
6
P
n
tho
```

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

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。

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.

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。

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.**

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.

PART 3 OF 7

Conversion and Input

Settled: a string is a sequence of characters, and it is **immutable**.

Now let the person type one in。**第一个重点：`input()` 拿回来的永远是字符串。**

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.

int() Truncates, It Does Not Round

truncate vs round

```
1  print(int(3.99))
2  print(int(-3.99))
3
4  print(round(3.99))
5  print(round(-3.99))
```

output

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

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

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」。

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。

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.

Why This Does Not Triple the Number

wanted: triple

```
1  # input("a number: ") returns a string.
2  # Writing it out directly changes nothing:
3  n = "5"
4
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。

Converting at the Right Moment

int(input(...))

```python
# 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。**

eval(): a String as an Expression

eval

```
print(eval("1 + 2 * 3"))
print(eval("2 ** 10"))

a, b = 1, 2
print(eval("a + b"))
```

output

```
7
1024
3
```

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

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。

PART 4 OF 7

Formatted Output

Settled: data comes in as a string, and you convert it on the way in.

Going the other way now: **put a value inside a sentence, and control how it looks.**

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.**

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.**

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.**

Decimal Places

{value:.Nf}

```
1  pi = 3.14159265
2
3  print(f"{pi:.2f}")          # two decimals
4  print(f"{pi:.4f}")
5  print(f"{pi:.0f}")          # none
6
7  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".

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.**

A Table That Lines Up

width + alignment

```
print(f"{'Name':<8}{'Score':>8}")
print("-" * 16)
print(f"{'Alice':<8}{87.5:>8.1f}")
print(f"{'Bob':<8}{92.0:>8.1f}")
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.

Printing in another base

{value:b/o/x}

```
1  n = 255
2
3  print(f"{n:b}")          # binary
4  print(f"{n:o}")          # octal
5  print(f"{n:x}")          # hexadecimal
6
7  m = 5
8  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**.

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.

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。**

print(): sep and end

sep / end

```
1  print("a", "b", "c")           # space
2  print("a", "b", "c", sep="")  # none
3  print("2026", "09", "14", sep="-")
4
5  print("no newline", end="")
6  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.

PART 5 OF 7

Functions

Settled: a program can now read from a person and write back to one.

But it is still one long file. **The first tool for organising it is the function.**

Why Functions Exist

- Real software is not measured in the dozens of lines you have written so far:
  - one file runs past a few thousand lines; a system runs into the millions
  - a team goes from 3 people to 300; some systems outlive the people who started them · 最早那批开发者可能已经退休了
- So the problem stops being「how do I write this」and becomes **「how does anyone still understand this next year」**
- Three principles, and Python gives you a mechanism for each:
  - **reuse** 复用 · write it once ｜ **isolation** 隔离 · keep the inside from colliding with the outside · **structure** 结构 · name it so it reads as one line

function → class → module，三件工具，分别是第 3、9、10 讲。**今天是第一件。**

The Three Parts of a Function

your first function

```
1  # def, name, parameter, return value
2  def area_of_circle(r):
3      return 3.14159 * r * r
4
5  print(area_of_circle(1))
6  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）。

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.

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.

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。

How a Call Jumps（调用的控制流）

```
1 def add(a, b):
2     s = a + b
3     return s
```

①

③

```
5 v = add(2, 3)
6 print(v)
```

at definition time

not one line runs

① jump to line 1； a = 2, b = 3

② the body runs top to bottom

③ return hands 5 back； v = 5

函数**内外隔离（空间）、一次性（时间）**
返回之后，函数里的 a、b、s 全部消失，外面碰不到；下次调用重新来过

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。

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.**

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。

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。

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.**

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.

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.

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.

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.

PART 6 OF 7 · SECOND KEY POINT

print() and return

Settled: a function takes parameters in and hands a value back.

第二个重点：`print` 和 `return` 没有任何关系。这是最难自己发现的错误，因为屏幕上看起来是对的。

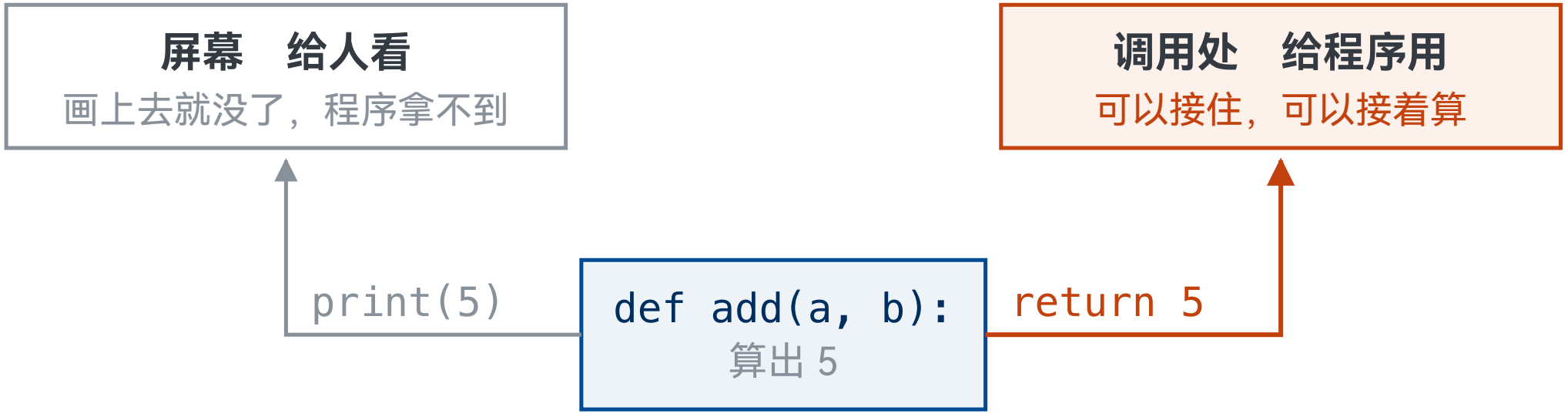
print() Is Not return

`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.

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。

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。

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**。

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。

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。

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。

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.

PART 7 OF 7

Modules

Settled: you can write a function, call it, and use what it returns.

You do not have to write them all. The standard library is already full of them.

import（导入模块）

the math module

```python
import math

print(math.sqrt(16))          # square root
print(math.pi)                # pi
print(math.floor(3.7))        # round down
print(math.ceil(3.2))         # round up
print(math.factorial(5))      # 5!
```

output

```
4.0
3.141592653589793
3
4
120
```

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

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。

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.

随堂小测 · 第 2 轮

扫码作答 · 这一轮四题

1. 连续赋值之后 x 是多少
2. 四个比较里哪个是 False
3. input 读进来的两个数
4. 格式化输出这一行的结果

今天四个落点各一题。一题一题放，答完一题公布一题。

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

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`**

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。**

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。

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.