上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Introduction to Computation (CS1602) · Lecture 4

Conditions, Loops and Functions

Instructor: Tao Huang

Part I: Foundations and Python Basics · Fall 2026

从前有座山

从前有座山，山上有座庙，庙里有个老和尚，

老和尚在给小和尚讲故事，故事讲的是——

从前有座山，山上有座庙，庙里有个老和尚，

老和尚在给小和尚讲故事，故事讲的是……

要让程序说一百遍这句话，你现在的办法是复制粘贴一百次。

Predict: how far apart are these two outputs?

A

```
for s in [50, 70, 90]:
    if s >= 60:
        print(s, "pass")
print("done")
```

B

```
for s in [50, 70, 90]:
    if s >= 60:
        print(s, "pass")
    print("done")
```

两段只差四个空格。 先在心里定一个答案。

随堂小测 · 第 1 轮

扫码作答 —— 这一轮一题

1 上一页那两段，输出差多少？

这一题现在不公布答案。讲到缩进那一节再回来。

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

Where we left off

- A variable is **a name pointing at a value**, not a box. `id()` gives the address, `is` asks whether two names point at the same thing
- **`input()` always returns a string** — convert it yourself
- Format output with an **f-string**
- A function is a name, parameters, a return value. **Defining is not running**
- **`print` is for people, `return` is for the program. Inside a function, always `return`**
- And **the question L3 closed on**: what does `grade(95)` print?

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

OVERVIEW

Today

Give the program a fork in the road, and a way to go round again.

Five parts

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

The conclusion, up front

sequence + branch + repetition = any computable process

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

AI policy — Level 0: No AI-generated code

- Weeks 1–7
  - Do not let an AI write or autocomplete code for you. Turn off AI completion in your editor.
  - You may ask an AI to explain a concept, look something up in the docs, or make sense of an error message.

PART 1 OF 5

Conditions

Let the program take a different road depending on what it finds.

A boolean expression evaluates to True or False

comparison

```
1  print(3 > 2)
2  print(3 == 2)
3  print("abc" != "abd")
4  print(3 >= 3)
```

output

```
True
False
True
True
```

第 2 讲讲过这些。**它们是条件语句唯一认识的东西。**

The order of and / or / not

precedence

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

output

```
True
False
```

not 最紧，然后 and，最后 or。**不确定就加括号。**

Chained comparison: write it as you would in math

one thing Python gets right

```
1  x = 5
2
3  print(1 < x < 10)            # same as 1 < x and x < 10
4  print(0 <= x <= 100)
```

output

```
True
True
```

C 和 Java 里 `1 < x < 10` **不报错，但毫无意义** —— 它先算 `1 < x` 得 `True`，再拿 `True` 和 `10` 比。

if / elif / else

a multi-way branch

```
1  score = 85
2
3  if score >= 90:
4      print("excellent")
5  elif score >= 80:
6      print("good")
7  elif score >= 60:
8      print("pass")
9  else:
10     print("fail")
```

output

```
good
```

Top to bottom: the first true branch runs, the rest are skipped.

Get the order backwards and everything collapses

错

```
if score >= 60:
    print("pass")
elif score >= 90:
    print("excellent")

# a 95 is graded "pass"
```

对

```
if score >= 90:
    print("excellent")
elif score >= 60:
    print("pass")

# strictest condition first
```

左边这段**不报错**，只是九十分以上的人全被判成了及格。

上一讲那道题：`grade(95)` 打印 pass。 多路分支从最严格的写起。

The classic exercise: leap years

early return

```
1  def is_leap(year):
2      if year % 400 == 0:
3          return True
4      if year % 100 == 0:
5          return False
6      return year % 4 == 0
7
8  for y in [1900, 2000, 2024, 2026]:
9      print(y, is_leap(y))
```

output

```
1900 False
2000 True
2024 True
2026 False
```

The rule: divisible by 4 is a leap year, unless divisible by 100 — unless also divisible by 400.

The same logic, written two other ways

NESTED — works, but tiring to read

```
def leap(y):
    if y % 4 == 0:
        if y % 100 == 0:
            if y % 400 == 0:
                return True
            else:
                return False
        else:
            return True
    else:
        return False
```

ONE EXPRESSION — harder to read

```
def leap(y):
    return (y % 4 == 0
            and (y % 100 != 0
                 or y % 400 == 0))
```

三种写法结果完全一样。**早返回那种最好读 —— 这是选它的唯一理由。**

The conditional expression: two short outcomes

A if cond else B

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

output

```
pass
```

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

match-case: instead of a long elif chain

Python 3.10 and later

```
def http_message(code):
    match code:
        case 200:
            return "OK"
        case 301 | 302:      # | means "or"
            return "Redirect"
        case 404:
            return "Not Found"
        case _:              # _ is the wildcard
            return "Unknown"

for c in [200, 302, 404, 418]:
    print(c, http_message(c))
```

output

```
200 OK
302 Redirect
404 Not Found
418 Unknown
```

| means **or**; _ is the wildcard, the equivalent of `else`.

It matches structure, and binds as it goes

pattern matching

```
def describe(point):
    match point:
        case (0, 0):
            return "the origin"
        case (0, y):
            return f"on y axis: {y}"
        case (x, y):
            return f"point ({x}, {y})"
        case _:
            return "not a point"

for p in [(0, 0), (0, 5), (2, 7), "hello"]:
    print(p, "->", describe(p))
```

output

```
(0, 0) -> the origin
(0, 5) -> on y axis: 5
(2, 7) -> point (2, 7)
hello -> not a point
```

`case (0, y)` does two things at once: it **checks** that the first is 0 and **binds** the second to `y`.

PART 2 OF 5

Loops

Back to the monk and his story.

while: keep going while the condition holds

three moving parts

```
1  n = 1
2  while n <= 5:
3      print(n, end=" ")
4      n += 1
5  print()
```

output

```
1 2 3 4 5
```

初始化 `n = 1`　**条件** `n <= 5`　**更新** `n += 1` —— 缺一个就出事。

Forget the update and it never stops

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

this page does not run

```
1  n = 1
2  while n <= 5:
3      print(n)
4      # forgot n += 1
5      # n stays 1, the condition stays true
```

在自己电脑上不小心跑了死循环，按 **Ctrl+C** 中断。

for: walk through a collection

one item at a time

```
1  for ch in "Python":
2      print(ch, end=" ")
3  print()
4
5  for x in [10, 20, 30]:
6      print(x, end=" ")
7  print()
```

output

```
P y t h o n
10 20 30
```

Anything you can **take items out of one by one** can follow `in`.

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

range

```
print(list(range(5)))           # 0 to 4
print(list(range(2, 8)))        # 2 to 7
print(list(range(0, 10, 3)))    # step 3
print(list(range(10, 0, -2)))   # backwards
```

output

```
[0, 1, 2, 3, 4]
[2, 3, 4, 5, 6, 7]
[0, 3, 6, 9]
[10, 8, 6, 4, 2]
```

`range(a, b)` **含头不含尾** —— 和第 3 讲的切片同一个约定。

range is not a list

why wrap it in list()

```
1  print(range(5))
2  print(list(range(5)))
3
4  total = 0
5  for i in range(1, 101):
6      total += i
7  print("1 to 100 =", total)
```

output

```
range(0, 5)
[0, 1, 2, 3, 4]
1 to 100 = 5050
```

`range(1000000)` uses almost no memory — it **does not build** a million numbers; it computes the next one when asked.

for or while

USE for

you know how many times

you have a collection to walk

most of the time

the count **is** fixed, so you
cannot forget the update,
and cannot loop forever

USE while

you do **not** know how many

you loop until something holds

reading input until it **is** valid
iterating toward a precision

you own the init, the
condition **and** the update

能用 for 就用 for。

Where while belongs: the 3n+1 problem

the Collatz conjecture

```python
def collatz_steps(n):
    steps = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        steps += 1
    return steps

for start in [6, 7, 27]:
    k = collatz_steps(start)
    print(f"from {start}: {k} steps to 1")
```

output

```
from 6: 8 steps to 1
from 7: 16 steps to 1
from 27: 111 steps to 1
```

至今没人能证明它对所有正整数都会回到 1。

break and continue

break: leave the whole loop now

```
for i in range(2, 50):
    if 100 % i == 0:
        print("first divisor:", i)
        break
```

output

```
first divisor: 2
```

continue: skip the rest of this round

```
for i in range(1, 11):
    if i % 3 == 0:
        continue
    print(i, end=" ")
```

output

```
1 2 4 5 7 8 10
```

break 结束**整个循环** · continue 只结束**这一轮**

In nested loops, break leaves only one level

错

```
for i in range(3):
    for j in range(3):
        if j == 1:
            break   # inner only
    # the outer loop goes on
```

对

```
def find(...):
    for i in range(3):
        for j in range(3):
            if ...:
                return i, j
    # return leaves every level
```

To leave two levels at once, the usual move is to **pull the logic into a function and `return`** — a return ends the whole function, however deep you are.

这也是「嵌套循环不超过两层」那条规范的来历之一。

Loops have an else too

for-else

```
 1  def find_divisor(n):
 2      for i in range(2, n):
 3          if n % i == 0:
 4              print(f"{n} is divisible by {i}")
 5              break
 6      else:
 7          print(f"{n} is prime")
 8
 9  find_divisor(15)
10  find_divisor(17)
```

output

```
15 is divisible by 3
17 is prime
```

The `else` runs when the loop **finishes normally**; a `break` skips it.

But that else is badly named

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

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

PART 3 OF 5

Bitwise operators

L2 said everything is bits. This is the payoff.

Six bitwise operators

- & **AND** 按位与 — 1 only where both bits are 1
- | **OR** 按位或 — 1 where either bit is 1
- ^ **XOR** 按位异或 — 1 where the two bits differ
- ~ **NOT** 按位取反 — every 0 becomes 1, every 1 becomes 0
- << **left shift** — shift left, pad with 0. One place left is ×2
- >> **right shift** — shift right. One place right is floor−÷2

They turn the integer into binary and work **bit by bit**.

Work through it bit by bit

line the bits up

```
a, b = 0b1100, 0b1010      # 12 and 10

print(f"a     = {a:04b}")
print(f"b     = {b:04b}")
print(f"a & b = {a & b:04b}  = {a & b}")
print(f"a | b = {a | b:04b}  = {a | b}")
print(f"a ^ b = {a ^ b:04b}  = {a ^ b}")
```

output

```
a     = 1100
b     = 1010
a & b = 1000  = 8
a | b = 1110  = 14
a ^ b = 0110  = 6
```

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

Shifting, and one trick worth keeping

n & 1 tests parity

```
1  n = 5
2  print(n, "<< 1 =", n << 1)      # times 2
3  print(n, ">> 1 =", n >> 1)      # floor-divide by 2
4
5  for k in [7, 8, 9, 10]:
6      print(k, "odd" if k & 1 else "even")
```

output

```
5 << 1 = 10
5 >> 1 = 2
7 odd
8 even
9 odd
10 even
```

`n & 1` takes the **lowest bit**: if it is 1, the number is odd.

Bitwise precedence trips people at both ends

错

```
8 & 4 + 1
# not (8 & 4) + 1 = 1
# but 8 & 5 = 0

1 << 2 + 3
# not (1 << 2) + 3 = 7
# but 1 << 5 = 32
```

对

```
(8 & 4) + 1

(1 << 2) + 3

# always parenthesize
```

结合顺序是：`+` `-` 和 `<<` `>>` **紧于** `&` `^` `|`，而 `&` `^` `|` 又紧于比较运算符。

别去背这张表。位运算和别的运算符放在一起，一律加括号。

PART 4 OF 5

More on functions

L3 gave you the shape of a function. Here is what parameters can do.

Default arguments: the caller may leave one out

default values

```
1  def greet(name, greeting="Hello"):
2      return f"{greeting}, {name}"
3
4  print(greet("Alice"))
5  print(greet("Alice", "Good morning"))
```

output

```
Hello, Alice
Good morning, Alice
```

Give a parameter a default and **the caller can omit it**.

Defaulted parameters must come last

wrong order

```
1  def bad(greeting="Hello", name):
2      return f"{greeting}, {name}"
```

traceback

```
File "example.py", line 1
    def bad(greeting="Hello", name):
                              ^^^^
SyntaxError: parameter without a default follows parameter with a default
```

报错是 **SyntaxError** —— 这是语法规定，连定义都通不过。

Never default to a mutable object

错

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

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

对

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

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

左边那个函数**第二次调用时会带着上一次的东西。**

Keyword arguments and positional ones

three ways to call it

```python
def describe(name, age, city):
    return f"{name}, {age}, from {city}"

print(describe("Alice", 18, "Shanghai"))
print(describe(name="Alice", age=18, city="Shanghai"))
print(describe("Alice", city="Shanghai", age=18))
```

output

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

混用时**位置参数必须在前**。

*args gathers the extra positional arguments into a tuple

when you do not know how many

```
1  def total(*numbers):
2      print("got:", numbers)
3      return sum(numbers)
4
5  print(total(1, 2, 3))
6  print(total(1, 2, 3, 4, 5))
7  print(total())
```

output

```
got: (1, 2, 3)
6
got: (1, 2, 3, 4, 5)
15
got: ()
0
```

One star collects every positional argument into a **tuple**. Zero of them is fine.

**kwargs gathers the keyword arguments into a dict

the order is fixed: plain -> *args -> **kwargs

```
1  def show(first, *rest, **opts):
2      print("first =", first)
3      print("rest  =", rest)
4      print("opts  =", opts)
5
6  show(1, 2, 3, mode="fast", retry=2)
```

output

```
first = 1
rest  = (2, 3)
opts  = {'mode': 'fast', 'retry': 2}
```

Two stars collect the keyword arguments into a **dict**. `args` and `kwargs` are only customary names — `*` and `**` are the syntax.

Python has no function overloading

the later def wins

```
1  def f(x):
2      return x * 2
3  def f(x, y):
4      return x + y
5  print(f(3))   # the one-argument f is gone
```

traceback

```
Traceback (most recent call last):
  File "example.py", line 5, in <module>
    print(f(3))   # the one-argument f is gone
          ^^^^
TypeError: f() missing 1 required positional argument: 'y'
```

同名函数，后定义的直接盖掉前面的，不管参数长得一样不一样。

PART 5 OF 5

Type hints and indentation

One habit to start today, one pit to walk around.

Annotate the parameters and the return value

Python 3.5 and later

```
1  def c_to_f(c: float) -> float:
2      return c * 9 / 5 + 32
3
4  print(c_to_f(37))
```

output

```
98.6
```

`c: float` says the argument should be a float; `-> float` says a float comes back.

Python does not enforce them

try passing a string

```python
def c_to_f(c: float) -> float:
    return c * 9 / 5 + 32
print(c_to_f("37"))
```

traceback

```
Traceback (most recent call last):
  File "example.py", line 3, in <module>
    print(c_to_f("37"))
          ^^^^^^^^^^^^
  File "example.py", line 2, in c_to_f
    return c * 9 / 5 + 32
           ~~~~~~^~~
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

标注不是检查。 类型传错照样跑，直到真正用到它的那一行才报错。

So why bother writing them

- **One: they are documentation that cannot go stale.**
  - A comment drifts away from the code. An annotation sits on the parameter
- **Two: your editor can use them.**
  - Completion gets sharper, and silly mistakes are underlined as you type
- **Three: they are the spec you hand an AI.**
  - Write the signature and the types first, let the AI fill the body — far more precise than a paragraph of prose

三条理由，**一条比一条重要。**

Three lines beat a hundred words of prose

What goes in, what comes out, what it does — **all three lines say it**.

a spec an AI can read

```python
def merge_sorted(a: list[int], b: list[int]) -> list[int]:
    """Merge two sorted integer lists into one sorted list."""
    ...
```

从这一讲起，讲义里的函数示例和参考解答都会带类型标注。

The shapes you will write most

four common signatures

```python
def f1(x: int) -> str:
    return str(x)

def f2(items: list[int]) -> int:
    return sum(items)

def f3(name: str, age: int = 18) -> str:  # default
    return f"{name} is {age}"

def f4(x: float) -> None:                 # no return
    print(x)

print(f1(42), f2([1, 2, 3]), f3("Ada"))
```

output

```
42 6 Ada is 18
```

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

Back to the opening: four spaces apart

"done" outside the for

```
for s in [50, 70, 90]:
    if s >= 60:
        print(s, "pass")
print("done")
```

output

```
70 pass
90 pass
done
```

"done" inside the for

```
for s in [50, 70, 90]:
    if s >= 60:
        print(s, "pass")
    print("done")
```

output

```
done
70 pass
done
90 pass
done
```

两段都不报错。 一个打印一次，一个每轮都打印。

Never mix tabs and spaces

错

```
def f():
    print(1)       # 4 spaces
        print(2)       # one tab

# TabError
```

对

```
def f():
    print(1)
    print(2)

# 4 spaces throughout
```

Tab 和四个空格在屏幕上可能一样宽，Python 却认为它们不同，会报 `TabError: inconsistent use of tabs and spaces in indentation`。

VS Code 默认把 Tab 转成空格。**从网上抄代码最容易撞上这个。**

随堂小测 · 第 2 轮

扫码作答 —— 这一轮五题

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

今天五个落点各一题。一题一题放，答完一题公布一题。

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

What to take away

- sequence + branch + repetition **expresses any computable process**
- `if–elif–else` takes the **first true** branch. Write the **strictest condition first**
- **Use `for` whenever you can.** `while` is for "I do not know how many times"
- `break` leaves the loop, `continue` skips the round. **Nested, `break` leaves one level**
- A loop's `else` means "**never broken out of**" — read it, rarely write it
- Mix bitwise with anything else and **always parenthesize**
- Defaulted parameters go last, **and never default to a list or a dict**
- **Type hints are not enforced, and still worth writing.** Indent with 4 spaces

Lab 4 — 七道题，条件与循环

- **4-1** `grade(score)` 成绩分级 —— **注意分支顺序**
- **4-2** `digit_sum(n)` 各位数字之和，**不许转成字符串**
- **4-3** `is_prime(n)` 判断质数
- **4-4** `collatz_steps(n)` 考拉兹步数：偶数除 2，奇数乘 3 加 1
- **4-5** `triangle(n)` 打印星号三角形，**末尾不要换行**
- **4-6** 猜数游戏 —— 写一个**程序**，不是函数
- **4-7** `hanoi_steps(n)` 汉诺塔最少步数 · **下周个人作业 1 的预热**

另有 **B 段**和 **C 段**（命令行，**期末要考**）。**从这次开始，所有函数都要求写类型提示。**

Next time

You can repeat now. But the data is still scattered

L5: lists. Until today you handled one value at a time. With a list, a whole batch becomes one thing you can pass, compute with and store.