

Python 讲义

中文版

Contents

1	入门	3
1.1	你的第一个程序	3
1.2	注释与代码风格	3
1.3	输入、处理、输出	4
2	变量、类型与运算符	5
2.1	变量与赋值	5
2.2	数字: 整数与浮点数	5
2.3	表达式与类型转换	6
2.4	布尔值与比较	6
3	字符串	8
3.1	索引	8
3.2	切片	8
3.3	字符串方法与长度	9
3.4	f-字符串	10
4	选择	11
4.1	if / elif / else	11
4.2	组合条件	12
5	迭代	13
5.1	for 循环与 range	13
5.2	累加器模式	13
5.3	while 循环	14
5.4	嵌套循环	14
6	列表与二维列表	16
6.1	列表	16
6.2	遍历列表	17
6.3	二维列表 (网格)	17
6.4	列表推导式	17
6.5	元组与集合	18
7	字典	19
7.1	字典	19
8	函数与抽象	21
8.1	定义与调用函数	21
8.2	返回值	21
8.3	参数、实参与作用域	22
8.4	过程抽象	22

8.5	模块与导入	22
9	错误、异常与测试	24
9.1	错误与调试	24
9.2	try / except / raise	24
9.3	测试与健壮性	25
10	文件	26
10.1	读写文本文件	26
11	算法设计	28
11.1	算法与分解	28
11.2	伪代码与流程图	28
11.3	递归与调用栈	29
12	数据结构	30
12.1	抽象数据类型 (ADT)	30
12.2	栈	30
12.3	队列	31
12.4	链表	31
12.5	哈希表	32
12.6	二叉搜索树	33
12.7	图	34
13	查找、排序与效率	35
13.1	线性查找与二分查找	35
13.2	排序 (冒泡与插入)	36
13.3	算法效率	37
13.4	随机性与模拟	38
14	面向对象与编程范式	39
14.1	类与对象	39
14.2	继承、封装与多态	40
14.3	编程范式	41
15	数据表示	42
15.1	比特与二进制	42
15.2	数据压缩	43
16	计算概念	44
16.1	什么是计算与设计循环	44
16.2	互联网	44
16.3	并行与分布式计算	45
16.4	计算的影响	45

17 综合运用	46
17.1 端到端小项目	46

1 入门

1.1 你的第一个程序

Python 一次运行一行代码。每一行是一条语句 (statement)。一个程序 (program) 就是一串从上到下依次运行的语句。

`print()` 函数把文本显示在屏幕上, 这叫作输出 (output)。引号里的文本是一个字符串 (string)。

```
print("Hello, world!")  
print("I am learning Python")
```

- 每个 `print()` 都会另起一行。
- 引号可以是 " 双引号" 或 ' 单引号', 两种都能表示字符串。
- 程序只有在你运行它时才会工作。

Your first program: write → run → see



Exam tip: `print` shows a value; quotes mark a string; indentation is part of the syntax

Source code runs through the interpreter to produce output

1.2 注释与代码风格

注释 (comment) 以 `#` 开头。该行中 `#` 后面的内容会被 Python 忽略。注释是写给人看的, 用来解释代码; 它不会改变代码的行为。

```
# This line is a note for humans  
print("Hi")           # you can also comment at the end of a line
```

好的代码风格让代码更易读:

- 使用能说明含义的清晰名字。
- 每行只写一条语句。

- 普通行不要在开头加空格。在 Python 中, 行首的空格 (缩进, indentation) 有特殊含义, 所以多余的空格会导致错误 (error)。

1.3 输入、处理、输出

很多程序都遵循一个简单的流程: **输入** (input) → **处理** → **输出**。先得到一些数据, 对它做处理, 再显示结果。

`input()` 函数读取用户输入的文本。它总是返回一个字符串。

```
name = input("What is your name? ")
print("Hello, " + name)
```

- `input()` 会等待用户输入并按下回车。
- 把输入的文本存进一个变量 (variable), 这样以后就能使用它。
- 因为 `input()` 返回的是字符串, 如果你需要数字, 要先用 `int(...)` 转换。

常见错误

- 忘记加引号: `print(Hello)` 会去找一个名为 `Hello` 的变量, 并抛出 `NameError` (命名错误)。文本需要引号: `print("Hello")`。
- 行首多打一个空格: Python 把缩进当作结构, 会抛出 `IndentationError` (缩进错误)。
- 以为 `input()` 返回数字: 它总是返回字符串, 所以在做任何数学运算前, 要先用 `int(...)` 把它转换。

2 变量、类型与运算符

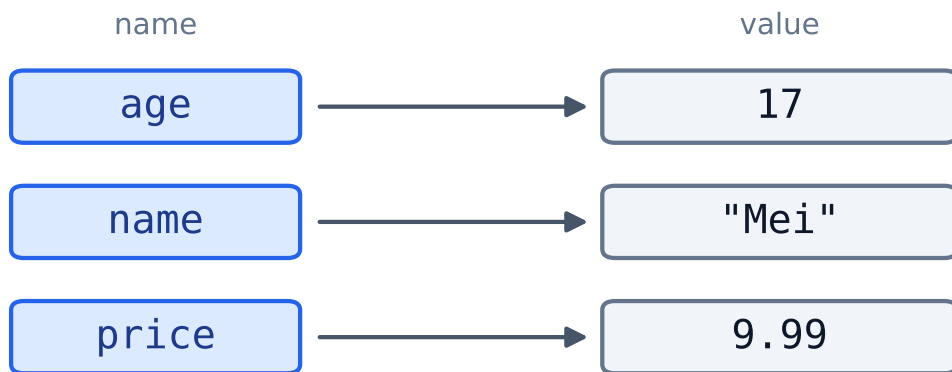
2.1 变量与赋值

变量 (variable) 是一个值 (value) 的名字。你用 = 来创建它, 这叫作赋值 (assignment)。名字写在左边, 值写在右边。

```
age = 17
name = "Mei"
price = 9.99
print(age, name, price)
```

现在 age 保存着 17。在任何需要这个值的地方都可以用这个名字, 以后也能改变它:

```
age = 17
age = age + 1    # age is now 18
print(age)
```



variable: name → value in memory

每个变量名都指向内存中的一个值

- = 不表示“相等”, 它的意思是“把右边的值存到左边这个名字下”。
- 要判断两个值是否相等, 用 == (见下文)。

2.2 数字: 整数与浮点数

Python 有两种主要的数字类型。整数 (integer, `int`) 是像 17 这样的整数。浮点数 (float) 带有小数点, 例如 9.99。

下面这些运算符 (operator) 用于数字:

运算符	含义	例子	结果
+	加	3 + 2	5
-	减	3 - 2	1
*	乘	3 * 2	6
/	除 (总是浮点数)	7 / 2	3.5
//	整除	7 // 2	3
%	取余 (取模)	7 % 2	1
**	幂	2 ** 3	8

- / 总是得到浮点数, 所以 4 / 2 是 2.0。
- // 和 % 常一起用: 17 // 5 是 3, 17 % 5 是 2。

2.3 表达式与类型转换

表达式 (expression) 是任何有值的东西, 例如 3 + 4 * 2。Python 按通常的数学顺序计算 (* 和 / 先于 + 和 -); 加括号能让顺序更清楚。

input() 返回字符串, 所以做数学前要先转换。把一个值从一种类型变成另一种, 叫作类型转换 (type conversion):

```
age = int("17")           # text "17" -> number 17
price = float("9.99")     # text -> 9.99
label = str(17)           # number -> text "17"
print(age, price, label)
```

- int("abc") 会失败, 所以只转换看起来像数字的文本。
- 混用类型也会失败: "age: " + 17 是错误的; 要写成 "age: " + str(17)。

2.4 布尔值与比较

布尔值 (Boolean) 只有两个取值之一: True 或 False。一次比较 (comparison) 会返回一个布尔值。

运算符	含义
==	等于
!=	不等于
< >	小于 / 大于
<= >=	小于等于 / 大于等于

```
print(7 > 2)           # True
print(3 == 3.0)        # True
age = 20
```



```
print(age >= 18)    # True
```

用 `and`、`or`、`not` 连接多个比较:

```
age = 20
print(age >= 13 and age <= 19)    # True only for a teenager
```

常见错误

- `/` 总是得到浮点数 (float), 连 `4 / 2` 也是 `2.0`。想要整数时用 `//`。
- 把 `=` 当成 `==` 是赋值, `==` 才是判断两个值是否相等。
- 把字符串和数字相加: `"age: " + 5` 会抛出 `TypeError`。先用 `str(5)` 转换。
- 浮点数并不精确, `0.1 + 0.2` 不正好等于 `0.3`—— 不要对浮点结果用 `==`。

3 字符串

3.1 索引

字符串 (string) 是引号里的文本。每个字符 (character) 都有一个位置, 叫作它的索引 (index)。第一个索引是 0, 不是 1。

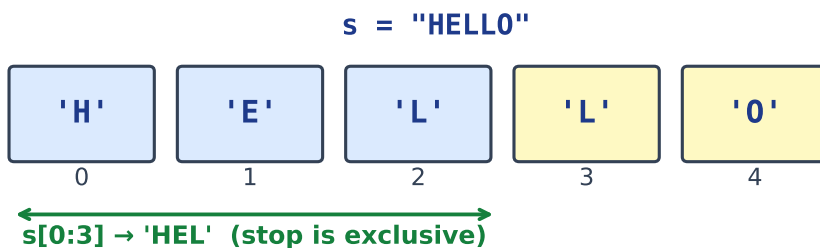
用方括号读取一个字符:

```
word = "Python"
print(word[0])    # P (the first character)
print(word[2])    # t
print(len(word))  # 6 (how many characters)
```

- 计数从 0 开始, 所以最后一个索引是 `len(word) - 1`。
- 负数索引从末尾往回数:`word[-1]` 是最后一个字符。

```
word = "Python"
print(word[-1])   # n
print(word[-2])   # o
```

- 索引太大会产生错误 (error)(一个 `IndexError`)。



Exam tip: slice is half-open [start, stop); omit ends for defaults; step is optional

`s[start:stop]` is a half-open window of characters

3.2 切片

切片 (slice) 取字符串的一部分。写成 `word[start:end]`。切片包含 `start`, 但在 `end` 之前停止。

```
word = "Python"
print(word[0:3])  # Pyt (positions 0, 1, 2)
```

```
print(word[2:5])    # tho
```

- 省略 start 表示从 0 开始; 省略 end 表示一直到末尾。

```
word = "Python"
print(word[:3])     # Pyt
print(word[3:])     # hon
```

- 第三个数字是步长 (step)。word[::-1] 会反转 (reverse) 字符串。

```
print("Python"[::-1]) # nohtyP
```

3.3 字符串方法与长度

方法 (method) 是属于某个值的函数。你用点号来调用它:

```
name = "mei chen"
print(name.upper())    # MEI CHEN
print(name.title())    # Mei Chen
print(len(name))       # 8
```

字符串是不可变的 (immutable): 方法会返回一个**新**字符串, 绝不改变原始 (original) 字符串。

```
name = "mei"
print(name.upper())    # MEI (the returned value)
print(name)            # mei (the original is unchanged)
```

常用方法 (每个都返回一个新值):

方法	含义	例子	结果
.upper() / .lower()	改变大小写	"Hi".lower()	hi
.strip()	去掉两端空格	" hi ".strip()	hi
.replace(a, b)	替换文本	"cat".replace("c", "b")	bat
.split(sep)	拆成列表	"a,b".split(",")	['a', 'b']

用 + 连接字符串, 这叫作拼接 (concatenation):

```
first = "Mei"
last = "Chen"
print(first + " " + last)    # Mei Chen
```

3.4 f-字符串

f-字符串 (f-string) 用各个值来拼出文本。在引号前加 `f`, 再用 `{...}` 把值括起来。

```
name = "Mei"
age = 17
print(f"{name} is {age} years old")    # Mei is 17 years old
```

- 大括号里可以放任何表达式 (expression)。
- `{value:.2f}` 会四舍五入到 2 位小数 (decimal places)。

```
price = 9.5
print(f"Two cost {price * 2}")        # Two cost 19.0
print(f"Pi is about {3.14159:.2f}")  # Pi is about 3.14
```

常见错误

- 字符串不能原地修改:`s[0] = "x"` 会报错。要新建一个字符串。
- 索引从 0 开始; 最后一个字符是 `s[-1]`, 而 `s[len(s)]` 会越界。
- 切片 `s[a:b]` 包含 `a`, 但在 `b` 之前停止。
- 字符串方法返回的是**新**字符串:`s.upper()` 不存下结果就等于没做。

4 选择

4.1 if / elif / else

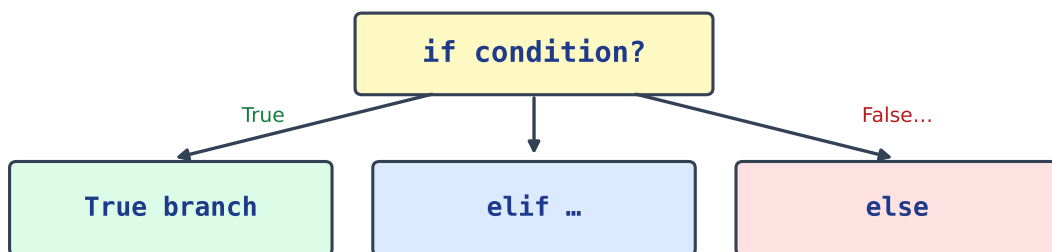
程序用 `if` 来选择做什么。只有当条件 (condition) 为真时, 它才运行缩进 (indent) 的代码块。`if` 这一行以冒号 (colon) 结尾。

```
score = 72
if score >= 60:
    print("pass")
# pass
```

用 `elif`(else-if) 增加更多情况, 用 `else` 表示“其它所有情况”。Python 只运行**第一个**为真的分支 (branch), 然后跳过其余的。

```
score = 72
if score >= 80:
    print("A")
elif score >= 60:
    print("B")
else:
    print("fail")
# B
```

- 用 `==(等于)`、`!=(不等于)`、`<`、`>`、`<=`、`>=` 比较值。
- 一次比较 (comparison) 会得到一个布尔值 (Boolean)——`True` 或 `False`。



Exam tip: only one branch runs; conditions are checked top to bottom; indent the body

if / elif / else: only one branch runs

4.2 组合条件

用 `and`、`or`、`not` 连接条件。`and` 需要**两边**都为真;`or` 需要**任意一边**为真;`not` 把布尔值取反。

```
age = 16
has_ticket = True
if age >= 18 and has_ticket:
    print("entry allowed")
else:
    print("entry refused")
# entry refused
```

- 用括号让顺序更清楚: `(a or b) and c`。

```
temp = 30
if temp > 25 and not temp > 35:
    print("warm but ok")
# warm but ok
```

常见错误

- 要写 `elif`, 不是 `else if`。
- 每个 `if` / `elif` / `else` 行都以冒号 `:` 结尾, 下面的语句块要缩进。
- `if x = 5:` 会报错 —— 比较要用 `==`。
- `else` 后面不跟条件, 只有 `if` 和 `elif` 才跟条件。

5 迭代

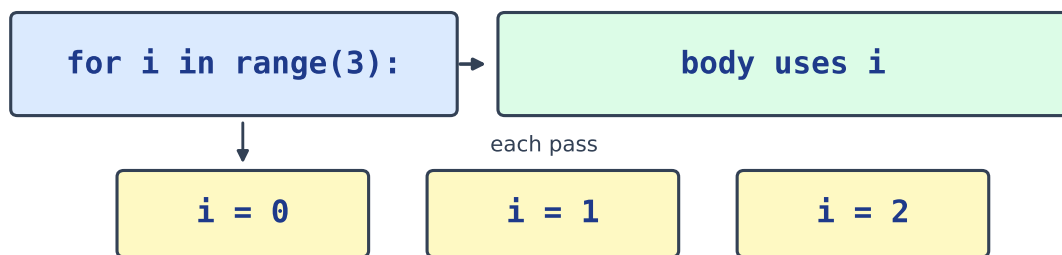
5.1 for 循环与 range

循环 (loop) 会重复运行代码。for 循环对序列 (sequence) 里的每个元素重复一次。range(n) 给出 0 到 $n - 1$ 的数字。

```
for i in range(5):  
    print(i)  
# 0, then 1, 2, 3, 4 (each on its own line)
```

- range(a, b) 从 a 到 b(不包含 b)。
- range(a, b, step) 每次增加一个步长 (step)。

```
for n in range(2, 11, 2):  
    print(n)          # 2 4 6 8 10
```



Exam tip: range(n) is 0 ... $n-1$; the loop variable takes one value per pass

for i in range(n): body runs with $i = 0 \dots n-1$

5.2 累加器模式

要在循环中逐步构建出一个结果, 就在循环之前先建一个变量, 然后每一轮更新 (update) 它。这就是累加器 (accumulator) 模式。

```
total = 0  
for n in range(1, 6):  
    total = total + n  
print(total)          # 15
```

- 同样的方法可以数出有多少个元素符合条件。

```
count = 0
for letter in "banana":
    if letter == "a":
        count = count + 1
print(count)          # 3
```

5.3 while 循环

while 循环**只要**条件保持为真就一直重复。要在里面改变某些东西, 否则它永不停止——这叫无限循环 (infinite loop)。

```
n = 1
while n <= 3:
    print(n)
    n = n + 1
# 1 2 3
```

- break 会立刻离开循环。

```
total = 0
while True:
    total = total + 10
    if total >= 30:
        break
print(total)          # 30
```

5.4 嵌套循环

一个循环放在另一个循环里面, 就是嵌套循环 (nested loop)。外层循环 (outer loop) 每转一轮, 内层循环 (inner loop) 都会完整运行一遍。

```
for row in range(3):
    line = ""
    for col in range(3):
        line = line + "*"
    print(line)
# ***
# ***
# ***
```

常见错误

- range(n) 从 0 到 n - 1, 不是 1 到 n——经典的差一 (off-by-one) 错误。
- 一边遍历列表一边删元素会漏掉元素; 要删就遍历一个副本。

- `while` 里忘了改循环变量, 就会无限循环。
- 缩进决定哪些语句在循环内; 缩进错的那行只会在循环结束后运行一次。

6 列表与二维列表

6.1 列表

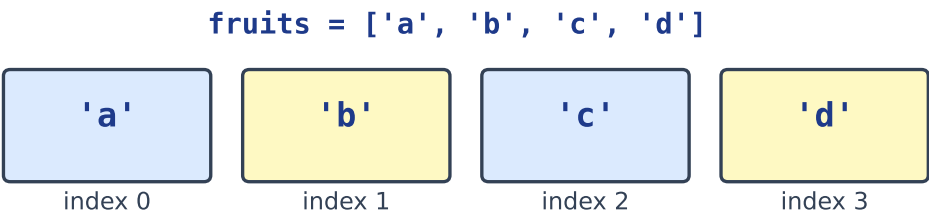
列表 (list) 在 `[]` 里按顺序保存许多值。每个元素 (item) 都有一个索引 (从 0 开始)。

```
scores = [88, 71, 95]
print(scores[0])      # 88
print(len(scores))    # 3
scores.append(60)     # add to the end
print(scores)         # [88, 71, 95, 60]
```

- 用索引改变某个元素: `scores[1] = 100`。
- 列表可以变长变短; 字符串不行。

天天要用的列表工具:

工具	作用
<code>a.append(x)</code>	在末尾加上 <code>x</code>
<code>a.insert(i, x)</code>	在位置 <code>i</code> 插入 <code>x</code>
<code>a.remove(x)</code>	删除第一个 <code>x</code>
<code>a.pop()</code> / <code>a.pop(i)</code>	删除并返回最后一个元素 / 第 <code>i</code> 个
<code>a.sort()</code>	原地排序
<code>sorted(a)</code>	返回一个 新的 已排序列表
<code>x in a</code>	<code>x</code> 在列表里吗?
<code>len(a)</code> 、 <code>sum(a)</code> 、 <code>max(a)</code> 、 <code>min(a)</code>	大小和快捷计算



Exam tip: first item is index 0; last is `len(list)-1`; negative indices count from the end

List indices start at 0

6.2 遍历列表

遍历 (traverse) 列表就是逐个访问每个元素。for 循环可以做到, 不需要索引。

```
scores = [88, 71, 95]
total = 0
for s in scores:
    total = total + s
print(total)           # 254
```

- 当你同时需要索引时, 用 enumerate。

```
for i, name in enumerate(["a", "b"]):
    print(i, name)      # 0 a / 1 b
```

6.3 二维列表 (网格)

二维列表 (2-D list) 是”列表的列表”—— 一个由行和列组成的网格 (grid)。用两个索引:grid[row][col]。

```
grid = [[1, 2, 3],
         [4, 5, 6]]
print(grid[0][2])      # 3
print(grid[1][0])      # 4
```

- 用嵌套循环 (nested loop) 访问每一个格子。

```
grid = [[1, 2], [3, 4]]
for row in grid:
    for value in row:
        print(value, end=" ")
print()                # 1 2 3 4
```

6.4 列表推导式

列表推导式 (list comprehension) 用一行就能构建一个新列表:[expression for item in sequence]。

```
squares = [x * x for x in range(5)]
print(squares)         # [0, 1, 4, 9, 16]
```

- 加上 if 只保留部分元素。

```
evens = [n for n in range(10) if n % 2 == 0]
print(evens)           # [0, 2, 4, 6, 8]
```

6.5 元组与集合

元组 (tuple) 是圆括号里的固定序列。创建之后不能修改——用它保存天生属于一起的值, 并把它解包 (unpack) 到多个名字里。

```
point = (3, 4)
x, y = point           # 解包
print(x, y)            # 3 4
```

需要一次交回两个结果的函数, 就返回一个元组:

```
def min_max(nums):
    return min(nums), max(nums)

lo, hi = min_max([5, 2, 9])
print(lo, hi)          # 2 9
```

集合 (set) 每个值只存一次, 而且没有顺序。它最适合去除重复, 以及做快速的成员测试 (membership test)。

```
votes = ["red", "blue", "red", "green", "red"]
colours = set(votes)
print(len(colours))     # 3 (重复已去除)
print("blue" in colours) # True
```

常见错误

- `b = a` **不会**复制列表: 两个名字指向同一个列表, 改一个另一个也变。用 `a.copy()` 或 `a[:]`。
- 最后一个元素是 `a[-1]`; `a[len(a)]` 会越界。
- `append` 只加一个元素; 要拼接另一个列表用 `extend` 或 `+`。
- 用 `[[0]*3]*3` 建网格会得到**同**一行的三个引用。要用循环逐行创建。
- 只有一个元素的元组要带逗号: `(5,)`, 不是 `(5)`。
- 集合没有顺序、没有重复, 所以不能用 `s[0]` 来索引。

7 字典

7.1 字典

字典 (dictionary, dict) 保存键 (key)→ 值 (value) 的成对数据。你通过键来查找值, 而不是通过数字索引。

```
student = {"name": "Mei", "score": 88}
print(student["name"])    # Mei
print(student["score"])   # 88
```

7.1.1 添加与更新

给一个键赋值, 就能添加它, 或改变已有的键。

```
student = {"name": "Mei"}
student["score"] = 88      # add a new key
student["score"] = 90      # update the value
print(student)             # {'name': 'Mei', 'score': 90}
```

7.1.2 检查与遍历

用 `in` 检查某个键是否存在。遍历所有键, 或遍历 `.items()` 同时得到键和值。

```
student = {"name": "Mei", "score": 90}
print("score" in student) # True
for key, value in student.items():
    print(key, "=", value)
# name = Mei
# score = 90
```

- 当键不存在时, `.get(key, default)` 会返回一个默认值 (default)—— 不会报错。

```
student = {"name": "Mei"}
print(student.get("age", 0)) # 0
```

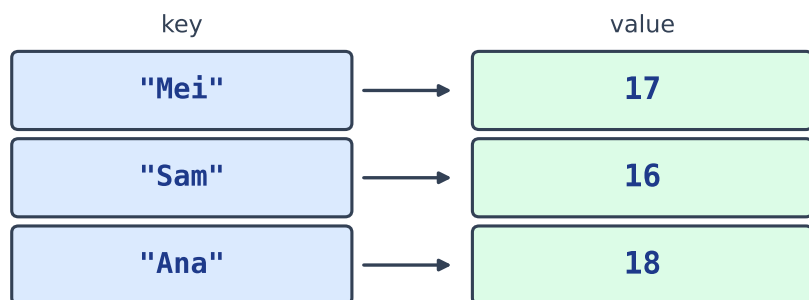
考试里的经典模式 —— 统计每个值出现多少次:

```
word = "banana"
counts = {}
for letter in word:
    counts[letter] = counts.get(letter, 0) + 1
print(counts)    # {'b': 1, 'a': 3, 'n': 2}
```

- 第一次遇到某个键时, `.get(letter, 0)` 会补上 0, 所以不会有 `KeyError`。

常见错误

- 用 `d[key]` 读取不存在的键会抛出 `KeyError`; 先用 `d.get(key)` 或判断 `if key in d`。
- 再次给 `d[key]` 赋值会覆盖旧值 —— 键是唯一的。
- 键必须是不可变的, 比如字符串或数字 —— 列表不能当键。



Exam tip: `ages["Mei"]` looks up the value; keys must be unique and immutable

A dictionary maps each key to one value

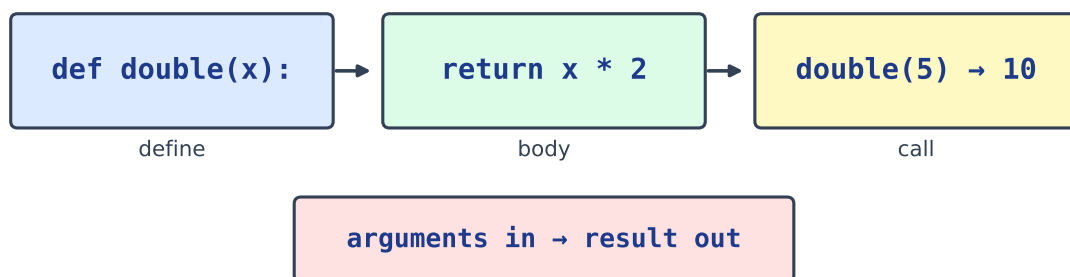
8 函数与抽象

8.1 定义与调用函数

函数 (function) 是一段有名字、可以重复使用的代码。用 `def` 定义 (define) 它, 再用名字调用 (call) 它。

```
def greet():  
    print("Hello!")  
  
greet()          # Hello!  
greet()          # Hello!
```

- 里面的代码只有在你调用函数时才运行。



Exam tip: `def` creates the function; `call` runs it; `return` sends a value back to the caller

def defines; call runs; return sends a value back

8.2 返回值

函数可以用 `return` 返回 (return) 一个值。调用处就代表那个值。

```
def square(n):  
    return n * n  
  
print(square(5))          # 25  
print(square(3) + 1)      # 10
```

- `return` 会立刻结束函数。没有 `return` 的函数返回 `None`。

8.3 参数、实参与作用域

形参 (parameter) 是 `def` 里的名字。实参 (argument) 是你传进去的值。

```
def power(base, exp):          # base, exp are parameters
    return base ** exp

print(power(2, 3))            # 8 (2 and 3 are arguments)
```

在函数内部建立的变量是局部 (local) 的 —— 它只在那里存在。这个范围就是它的作用域 (scope)。

```
def f():
    x = 10                    # local to f
    return x

print(f())                   # 10
# print(x) here would be an error: x is not defined outside f
```

形参可以有默认值 (default value), 调用者不传的时候就用它:

```
def greet(name, greeting="Hello"):
    return greeting + ", " + name

print(greet("Mei"))          # Hello, Mei
print(greet("Sam", "Welcome")) # Welcome, Sam
```

8.4 过程抽象

过程抽象 (procedural abstraction) 的意思是把细节藏在一个名字后面。你按函数的名字和它做的事来使用它, 而不是按它怎么实现来使用。

```
def area_of_rectangle(w, h):
    return w * h

print(area_of_rectangle(4, 5)) # 20
```

- 好的函数只做一件事, 有清楚的名字, 并避免重复代码。

8.5 模块与导入

模块 (module) 是一个装着现成函数的文件。用 `import` 导入 (import) 它。


```
import random
random.seed(0)           # makes the result repeatable
print(random.randint(1, 6)) # a dice roll

import math
print(math.sqrt(16))      # 4.0
```

常见错误

- 没有写 `return`, 函数就返回 `None`。打印 (`print`) 和返回 (`return`) 不是一回事。
- 不要用可变的默认值如 `def f(x=[])`—— 同一个列表会在所有调用之间共享。
- 函数内部创建的变量是局部的, 在外面看不到。
- 用 `f()` 才会运行函数; 单写 `f` 只是引用它。

9 错误、异常与测试

9.1 错误与调试

代码会以三种方式出错。语法错误 (syntax error) 违反了 Python 的规则, 所以根本无法运行。运行时错误 (runtime error) 在运行过程中发生, 比如除以零。逻辑错误 (logic error) 能运行, 但给出错误的答案。

```
# A runtime error, caught so this block still finishes:
try:
    print(10 / 0)
except ZeroDivisionError:
    print("cannot divide by zero")
# cannot divide by zero
```

- Python 会打印一段回溯 (traceback), 显示在哪里出错。从下往上读它。

SyntaxError code shape wrong	NameError name not defined	TypeError wrong type of value	IndexError index out of range
---	---	--	--

Exam tip: read the last line of the traceback first — it names the error and the line

Common Python errors: Syntax, Name, Type, Index

9.2 try / except / raise

把有风险的代码放进 try。如果它失败, except 会捕获异常 (exception) 并处理 (handle) 它, 而不是崩溃。

```
def to_int(text):
    try:
        return int(text)
    except ValueError:
        return 0

print(to_int("42"))    # 42
print(to_int("abc"))   # 0
```

- 捕获具体的类型 (ValueError、ZeroDivisionError 等)。
- raise 用来故意抛出你自己的错误。

```
def set_age(age):  
    if age < 0:  
        raise ValueError("age cannot be negative")  
    return age  
  
try:  
    set_age(-1)  
except ValueError as err:  
    print("error:", err)  
# error: age cannot be negative
```

9.3 测试与健壮性

测试 (test) 用来检查代码是否给出正确答案。既要试正常情况, 也要试边界情形 (edge case)——空输入、零、非常大的值。

```
def average(nums):  
    if len(nums) == 0:          # edge case: empty list  
        return 0  
    return sum(nums) / len(nums)  
  
print(average([2, 4, 6]))      # 4.0  
print(average([]))             # 0
```

- 健壮 (robust) 的代码不会因为奇怪的输入而崩溃; 它会优雅地处理它们。

常见错误

- 不要用光秃秃的 except:——要捕获具体错误, 例如 except ValueError:。
- 语法错误 (syntax error) 会在程序运行前就中止它, 所以先修这些。
- 要测试边界情况 (空输入、零、最大值), 不能只测最简单的那种。

10 文件

10.1 读写文本文件

文本文件 (text file) 把文本保存在磁盘上。用 `open(name, mode)` 打开它, 其中模式 (mode) 说明是读还是写。始终使用 `with`, 它会帮你关闭文件。

10.1.1 写入

模式 "w" 写一个新文件, 并抹掉任何旧内容。

```
with open("notes.txt", "w") as f:
    f.write("first line\n")
    f.write("second line\n")
print("saved")           # saved
```

10.1.2 读取

模式 "r"(默认) 用来读。`.read()` 把整个文件作为一个字符串返回。

```
with open("notes.txt", "w") as f:
    f.write("hello\nworld\n")
with open("notes.txt") as f:
    print(f.read().strip())  # hello / world
```

10.1.3 逐行读取

遍历文件, 一次得到一行。`.strip()` 会去除 (remove) 末尾的换行符 (newline)。

```
with open("data.txt", "w") as f:
    f.write("Mei,88\nSam,71\n")
with open("data.txt") as f:
    for line in f:
        name, score = line.strip().split(",")
        print(name, "scored", score)
# Mei scored 88
# Sam scored 71
```

10.1.4 追加

模式 "a" 追加 (append)—— 它在末尾添加, 而不抹掉原有内容。

```
with open("log.txt", "a") as f:
    f.write("line 1\n")
```

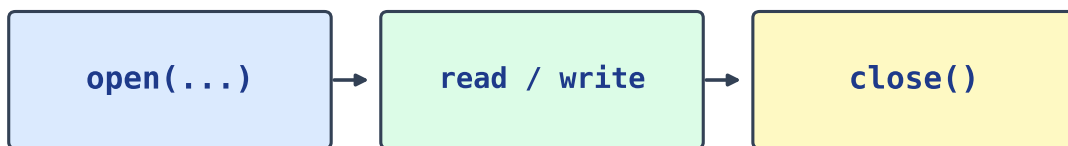
```
with open("log.txt", "a") as f:
    f.write("line 2\n")
with open("log.txt") as f:
    print(f.read().strip())    # line 1 / line 2
```

模式	含义
"r"	读 (默认)
"w"	写 (先抹掉)
"a"	追加 (添加到末尾)

常见错误

- 一定要关闭文件, 或用 `with open(...) as f:`, 它会替你关闭。
- `read()` 把整个文件当成一个字符串返回, 每行末尾仍带着 `\n`。
- 用 `"w"` 打开会先清空文件; 要追加到末尾用 `"a"`。

`with open(...) as f:` auto-closes



Exam tip: mode 'r' reads, 'w' writes (overwrites); prefer with so the file always closes

open → read/write → close (with auto-closes)

11 算法设计

11.1 算法与分解

算法 (algorithm) 是一份能解决问题的清晰步骤清单。分解 (decomposition) 的意思是把一个大问题拆成更小的、能逐个解决的部分。

```
# Algorithm: find the largest number in a list
def largest(nums):
    best = nums[0]
    for n in nums:
        if n > best:
            best = n
    return best

print(largest([3, 9, 2, 7])) # 9
```

- 抽象 (abstraction) 的意思是忽略细节: 你直接用 `largest(...)`, 不必再去读它怎么实现。

11.2 伪代码与流程图

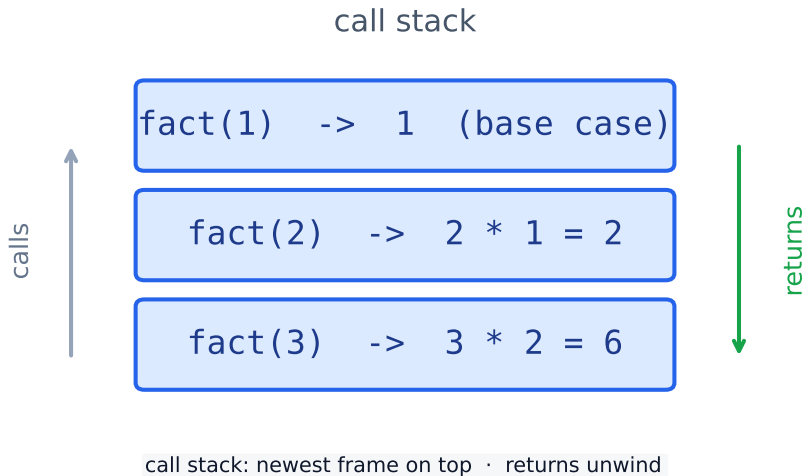
伪代码 (pseudocode) 是为算法写的、朴素而有结构的英文, 写在真正的代码之前。它不会被运行。

```
SET best TO first number
FOR each number n
    IF n > best THEN
        SET best TO n
OUTPUT best
```

流程图 (flowchart) 把同样的计划画出来: 每个步骤一个方框, 每个判断 (decision) 一个菱形, 用箭头表示顺序。

11.3 递归与调用栈

递归 (recursion) 是指一个函数调用它自己。它需要一个基准情形 (base case)(一个简单的输入, 立刻返回), 以及一个递归情形 (recursive case)(它在更小的输入上调用自己)。



factorial(3) 的调用栈: 每个调用先等待, 再按相反的顺序返回

```
def fact(n):
    return 1 if n <= 1 else n * fact(n - 1)

print(fact(5))    # 120
```

- 每个暂停的调用都停在调用栈 (call stack) 上, 直到它上面的调用返回。

常见错误

- 递归 (recursion) 必须有基准情形 (base case), 否则会无限自调用, 撑爆调用栈。
- 伪代码 (pseudocode) 用来规划 —— 不必能运行, 但每一步都要清楚无歧义。
- 先把大问题拆成一个个有名字的小步骤, 再动手写代码。

12 数据结构

12.1 抽象数据类型 (ADT)

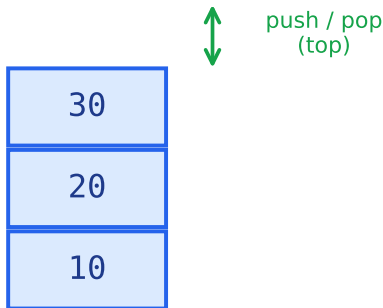
抽象数据类型 (abstract data type, ADT) 描述一些数据以及对它的操作, 与它如何实现分开。你通过它的操作来使用它, 而不是通过它内部的存储方式。

```
# A stack ADT, built on a list
s = []
s.append(1)      # add
s.append(2)
print(s.pop())   # 2 (remove the most recent)
```

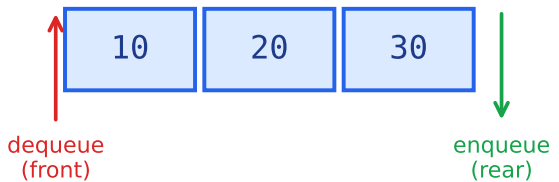
12.2 栈

栈 (stack) 是后进先出 (LIFO)。你把元素压入 (push) 栈顶, 也从栈顶弹出 (pop)。

Stack (LIFO)



Queue (FIFO)



stack: LIFO · queue: FIFO

栈从顶部移除 (后进先出); 队列从前端移除 (先进先出)

```
stack = []
stack.append("a")
stack.append("b")
print(stack.pop()) # b
print(stack.pop()) # a
```

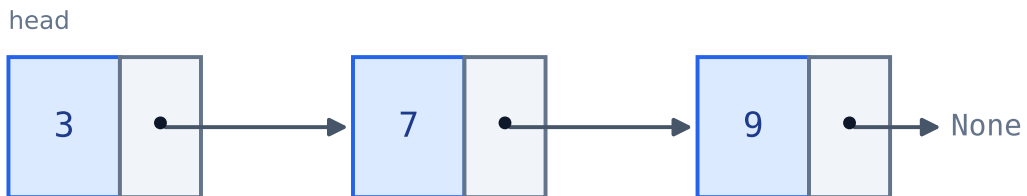

12.3 队列

队列 (queue) 是先进先出 (FIFO)。你在队尾入队 (enqueue), 从队首出队 (dequeue)。

```
queue = []
queue.append("a")      # enqueue
queue.append("b")
print(queue.pop(0))    # a  (dequeue the front)
print(queue.pop(0))    # b
```

12.4 链表

链表 (linked list) 是一串节点 (node)。每个节点保存数据和一个指向下一个节点的指针 (pointer); 最后一个指向 *None*。



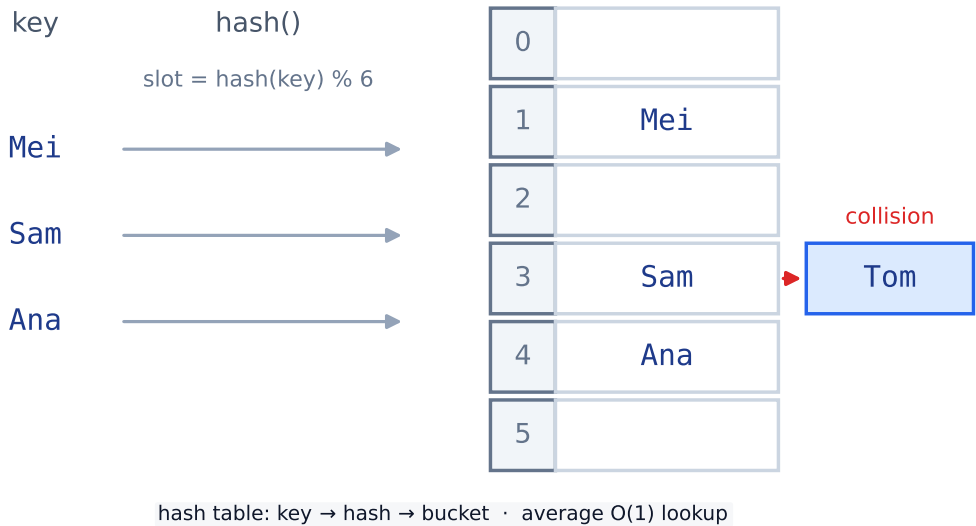
linked list: node → next · O(1) insert at head

链表: 每个节点保存数据和指向下一个节点的指针, 最后指向 *None*

```
n3 = {"data": 3, "next": None}
n2 = {"data": 2, "next": n3}
n1 = {"data": 1, "next": n2}
node = n1
while node is not None:      # traverse to the end
    print(node["data"])
    node = node["next"]
# 1 2 3
```

12.5 哈希表

哈希表 (hash table) 用哈希函数 (hash function) 把键映射到一个槽。两个键可能落入同一个槽 —— 这叫冲突 (collision)。Python 的 `dict` 就是哈希表, 所以查找很快。

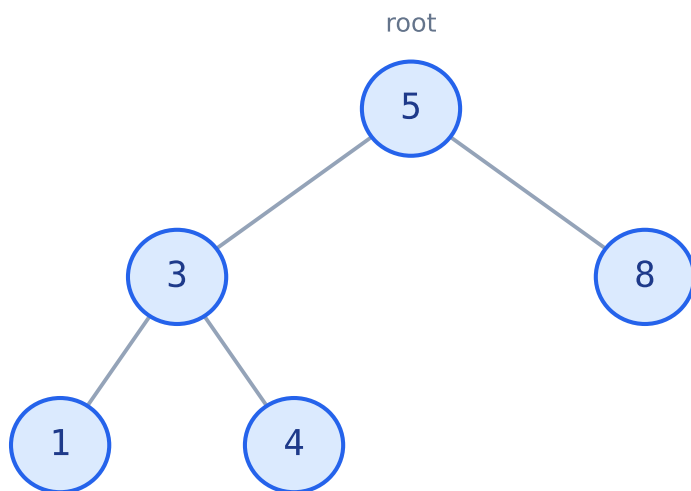


哈希函数把每个键映射到一个槽; 两个键可能落入同一个槽 (冲突)

```
table = {}
table["Mei"] = 88
table["Sam"] = 71
print(table["Mei"])    # 88 (fast lookup by key)
```

12.6 二叉搜索树

二叉搜索树 (binary search tree, BST) 保持有序: 每个左孩子都比它的节点小, 每个右孩子都比它大。查找一直很快。



in-order traversal: 1 3 4 5 8

BST: left < node < right · search $O(\log n)$ if balanced

二叉搜索树: 较小的值放在左边, 较大的值放在右边

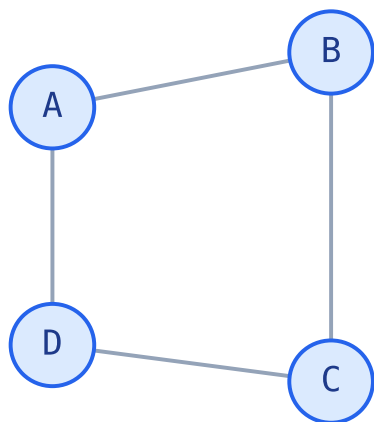
```
def insert(root, val):
    if root is None:
        return {"val": val, "left": None, "right": None}
    if val < root["val"]:
        root["left"] = insert(root["left"], val)
    else:
        root["right"] = insert(root["right"], val)
    return root

def inorder(root):
    if root is None:
        return []
    return inorder(root["left"]) + [root["val"]] + inorder(root["right"])

tree = None
for v in [5, 3, 8, 1, 4]:
    tree = insert(tree, v)
print(inorder(tree))  # [1, 3, 4, 5, 8]
```

12.7 图

图 (graph) 是一组由边 (edge) 连接的顶点 (vertex)。邻接表 (adjacency list)—— 一个 “邻居列表的字典”—— 是常见的存储方式。



```
graph = {  
    "A": ["B", "D"],  
    "B": ["A", "C"],  
    "C": ["B", "D"],  
    "D": ["A", "C"],  
}
```

graph: vertices + edges · BFS/DFS traverse

由顶点和边组成的图, 以及它的邻接表形式

```
graph = {"A": ["B", "D"], "B": ["A", "C"], "C": ["B", "D"], "D": ["A",  
    ↪ "C"]}  
for vertex in graph:  
    print(vertex, "->", graph[vertex])  
# A -> ['B', 'D'] (and so on for B, C, D)
```

常见错误

- 栈 (stack) 是后进先出, 队列 (queue) 是先进先出, 别搞混。
- 在 pop 或出队之前, 先检查结构是不是空的。
- 在链表 (linked list) 中, 弄丢 head 指针就等于弄丢了整条链表。

13 查找、排序与效率

13.1 线性查找与二分查找

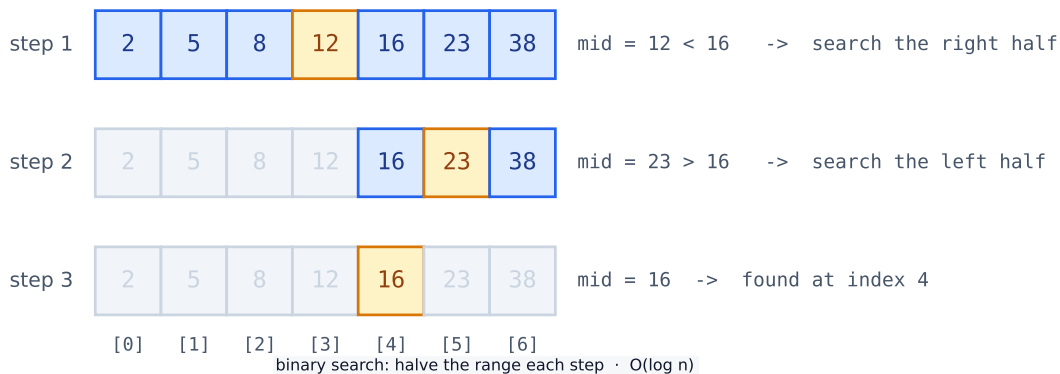
查找 (search) 用来找到某个值在哪里。线性查找 (linear search) 逐个检查每个元素, 所以它对任何列表都有效。

```
def linear_search(items, target):  
    for i in range(len(items)):  
        if items[i] == target:  
            return i  
    return -1      # not found  
  
print(linear_search([4, 8, 2, 9], 2))    # 2
```

二分查找 (binary search) 快得多, 但需要一个**已排序**的列表。它每一步把范围折半。

```
def binary_search(items, target):  
    lo, hi = 0, len(items) - 1  
    while lo <= hi:  
        mid = (lo + hi) // 2  
        if items[mid] == target:  
            return mid  
        elif items[mid] < target:  
            lo = mid + 1  
        else:  
            hi = mid - 1  
    return -1  
  
print(binary_search([1, 3, 5, 7, 9], 7))    # 3
```

binary search for 16 (the array must be sorted)



二分查找每一步把范围减半 —— 已排序列表上是 $O(\log n)$

13.2 排序 (冒泡与插入)

排序 (sort) 就是把元素按顺序排好。冒泡排序 (bubble sort) 反复交换 (swap) 顺序不对的相邻元素。

```
def bubble_sort(a):  
    a = a[:] # work on a copy  
    for i in range(len(a)):  
        for j in range(len(a) - 1 - i):  
            if a[j] > a[j + 1]:  
                a[j], a[j + 1] = a[j + 1], a[j]  
    return a  
  
print(bubble_sort([5, 2, 4, 1])) # [1, 2, 4, 5]
```

插入排序 (insertion sort) 一次把一个元素放进已排好的部分, 把每个新元素往回滑到属于它的位置:

```
def insertion_sort(a):  
    a = a[:] # 在副本上操作  
    for i in range(1, len(a)):  
        key = a[i]  
        j = i - 1  
        while j >= 0 and a[j] > key: # 把较大的值右移  
            a[j + 1] = a[j]  
            j -= 1  
        a[j + 1] = key # 把 key 放进空位  
    return a
```

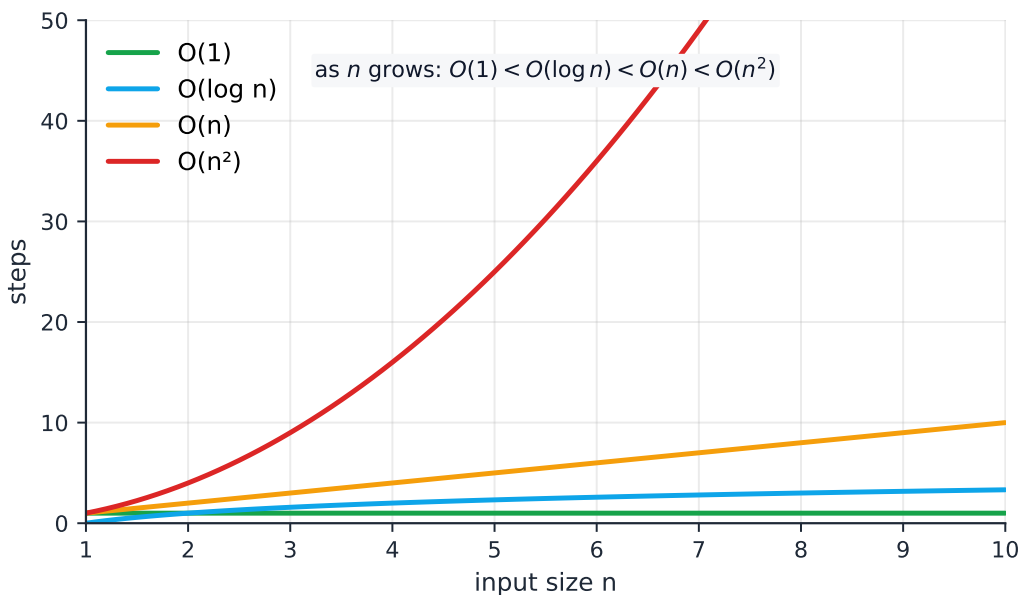
```
print(insertion_sort([5, 2, 4, 1])) # [1, 2, 4, 5]
```

- 在真实代码里, 用 Python 内置的 `sorted()`:

```
print(sorted([5, 2, 4, 1])) # [1, 2, 4, 5]
```

13.3 算法效率

效率 (efficiency) 问的是: 当输入变大时, 工作量怎样增长。我们用大 O 记号 (Big-O) 来描述它。



常见复杂度下, 步数随输入规模增长的情况

Big-O	名称	例子
$O(1)$	常数	查找字典的键
$O(\log n)$	对数	二分查找
$O(n)$	线性	线性查找
$O(n^2)$	平方	冒泡排序

```
def steps(n): # how many steps a linear scan takes
    count = 0
    for i in range(n):
        count = count + 1
    return count
```

```
print(steps(100))          # 100  -> O(n)
```

13.4 随机性与模拟

random 模块产生随机数。用一个种子 (seed) 让结果可重复。模拟 (simulation) 运行很多次随机试验来估计一个答案。

```
import random
random.seed(0)
rolls = [random.randint(1, 6) for _ in range(1000)]
print(rolls.count(6))      # about 1/6 of 1000
```

常见错误

- 二分查找 (binary search) 只对**已排序**的列表有效。
- 大 O(Big-O) 说的是时间如何**增长**, 不是确切时间; $O(n^2)$ 的方法只在极小输入时才可能快过 $O(n)$ 。
- 冒泡排序 (bubble sort) 是 $O(n^2)$ —— 学习可以, 但对大列表很慢。

14 面向对象与编程范式

14.1 类与对象

类 (class) 是一张蓝图。对象 (object) 是由它建造出来的一个具体东西 (一个实例 (instance))。__init__ 是构造方法 (constructor), 用来设置每个对象;self 就是对象本身。

```
class Dog:
    def __init__(self, name):
        self.name = name          # an attribute
    def speak(self):
        return self.name + " says woof"

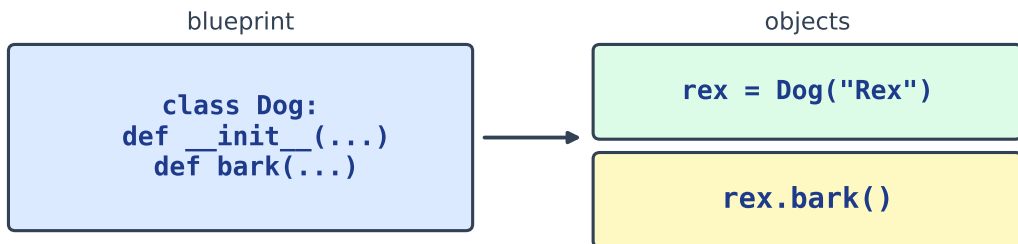
d = Dog("Rex")
print(d.speak())                # Rex says woof
```

- name 是属性 (attribute)(对象上的数据);speak 是方法 (method)(一个动作)。

加上 __str__, 就能控制 print(obj) 显示什么:

```
class Dog:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return f"Dog named {self.name}"

print(Dog("Rex"))               # Dog named Rex
```



Exam tip: class defines the type; calling ClassName(...) creates an object (instance)

A class is a blueprint; calling it creates an object

14.2 继承、封装与多态

继承 (inheritance) 让子类 (subclass) 复用父类 (superclass)。用 `super()` 调用父类; 重写 (override) 一个方法来改变它。

```
class Animal:
    def speak(self):
        return "some sound"

class Cat(Animal):
    def speak(self):                # override
        return "meow"

print(Cat().speak())    # meow
```

封装 (encapsulation) 把数据藏在方法后面; 名字前加一个下划线表示它是私有 (private) 的。

```
class Account:
    def __init__(self):
        self._balance = 0          # private
    def deposit(self, n):
        self._balance += n
    def balance(self):
        return self._balance

a = Account()
a.deposit(50)
print(a.balance())    # 50
```

多态 (polymorphism) 的意思是同一个名字、多种行为 —— 每个对象会运行各自正确的 `speak`。

```
class Cat:
    def speak(self):
        return "meow"

class Cow:
    def speak(self):
        return "moo"

for animal in [Cat(), Cow()]:
    print(animal.speak())    # meow, then moo
```

14.3 编程范式

范式 (paradigm) 是写程序的风格。过程式 (procedural) 代码是一连串步骤和函数。面向对象 (object-oriented) 代码把数据和方法组织进对象。声明式 (declarative) 代码说的是你**要什么**, 而不是**怎么做** (列表推导式或 SQL)。

```
def total(nums):                # procedural
    t = 0
    for n in nums:
        t += n
    return t
print(total([1, 2, 3]))        # 6

print(sum([1, 2, 3]))          # 6 (declarative: same result)
```

常见错误

- 每个方法的第一个参数都要是 `self`。
- `__init__` 用来初始化新对象, 创建对象时会自动运行。
- 同一个类的两个对象各有各的属性; 改一个不会影响另一个。

15 数据表示

15.1 比特与二进制

比特 (bit) 是单个的 0 或 1。二进制 (binary) 是以 2 为底的数制: 每一位的价值都是它右边一位的两倍 (1、2、4、8.....)。十进制 (denary, 以 10 为底) 是我们平常用的数。



$$8 + 4 + 1 = 13$$

binary: powers of 2 · bit k has weight 2^k

二进制位值: 1101 表示 $8 + 4 + 1 = 13$

```
print(bin(13))           # 0b1101
print(int("1101", 2))    # 13
```

- 8 个比特组成一个字节 (byte)。当数字太大时, 固定的位宽会溢出 (overflow)(绕回)。

```
x = 250
x = (x + 10) % 256      # one byte wraps at 256
print(x)                # 4
```

十六进制 (hexadecimal, base 16) 是阅读二进制的紧凑方式: 一个十六进制位恰好代表四个二进制位。Python 用 0x 表示十六进制:

```
print(hex(255))          # 0xff
print(0xFF)              # 255
print(int("ff", 16))     # 255
```

15.2 数据压缩

压缩 (compression) 让数据变小。无损 (lossless) 压缩保留每一个比特, 所以你能精确还原原始数据。有损 (lossy) 压缩丢掉一些细节 —— 更小但不精确 —— 用于照片和音乐。

游程编码 (run-length encoding) 是一种简单的无损方法: 把一段游程 (run)(重复) 存成 “个数 + 值”。

```
def rle(text):
    out = ""
    i = 0
    while i < len(text):
        run = 1
        while i + run < len(text) and text[i + run] == text[i]:
            run += 1
        out += str(run) + text[i]
        i += run
    return out

print(rle("AAAABBBCCD"))    # 4A3B2C1D
```

常见错误

- n 个比特 (bit) 能存 2^{**n} 种不同的值, 范围是 0 到 $2^{**n} - 1$ 。
- 有损压缩 (lossy) 会丢掉细节且无法还原; 无损压缩 (lossless) 可以精确还原。

16 计算概念

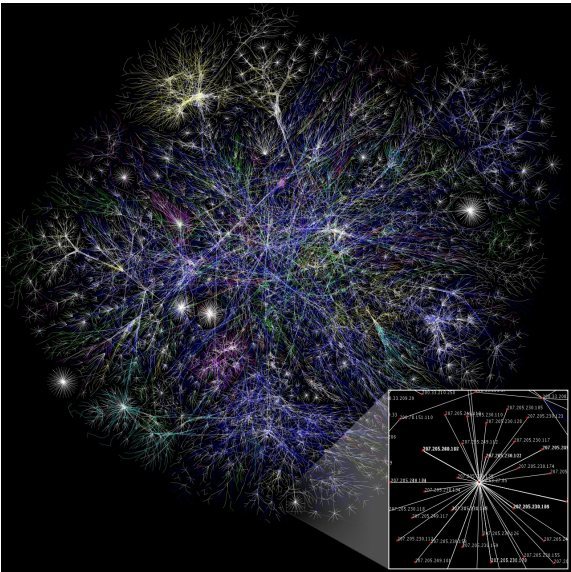
16.1 什么是计算与设计循环

计算 (computing) 就是用计算机解决问题: 输入、处理、输出。好的软件是在一个设计循环 (design cycle) 中做出来的 —— 计划、编写、测试、改进 —— 反复多次。

- 把问题拆开, 先做一小部分, 测试它, 再加更多。
- 程序员以团队方式工作, 并复用彼此的代码。

16.2 互联网

互联网 (Internet) 是 “ 网络的网络 ”。数据被切成一个个数据包 (packet), 它们分别传输, 在另一端再重新拼起来。叫作协议 (protocol) 的共同规则 (例如 TCP/IP) 让这一切运转。如果一条路径断了, 数据包就走另一条 —— 这叫冗余 (redundancy), 它带来容错 (fault tolerance)。



互联网地图: 每条线都是两个网络之间的一条路径

层	作用
HTTP	请求并发送网页
TCP	可靠、按顺序送达
IP	编址与路由

16.3 并行与分布式计算

顺序 (sequential) 代码一次做一步。并行 (parallel) 计算在多个核心 (core) 上同时做好几步, 可以带来加速 (speedup)。分布式 (distributed) 计算把工作分散到许多台计算机上, 例如云。

- 并非所有事都能并行: 有些步骤必须等待前一步的结果。

16.4 计算的影响

计算既带来好处, 也带来害处。数字鸿沟 (digital divide) 指的是并非每个人都能平等地用上它。软件可能从它学习的数据中带上偏见 (bias)。要尊重知识产权 (intellectual property)(许可), 并保护人们的个人数据 (personal data) 和隐私 (privacy)。

常见错误

- 互联网 (Internet) 和万维网 (World Wide Web) 不是一回事: 万维网只是运行在互联网之上的一个服务。
- 更多处理器核心 (core) 只有在任务能拆成可同时运行的部分时才有用。

17 综合运用

17.1 端到端小项目

一个小项目 (mini-project) 把前面学过的想法组合起来: 列表里的数据、一个在循环里带选择的函数, 以及打印出的输出。这也是 AP 创作型表现任务的形态。

17.1.1 项目: 平均分

```
def average(marks):  
    if len(marks) == 0:  
        return 0  
    return round(sum(marks) / len(marks), 1)  
  
print(average([88, 71, 95, 60]))    # 78.5
```

17.1.2 项目: 统计及格人数

```
def count_passes(marks, pass_mark=60):  
    passes = 0  
    for m in marks:                # iteration  
        if m >= pass_mark:        # selection  
            passes += 1  
    return passes  
  
print(count_passes([88, 50, 95, 60]))    # 3
```

17.1.3 项目: 筛选成新列表

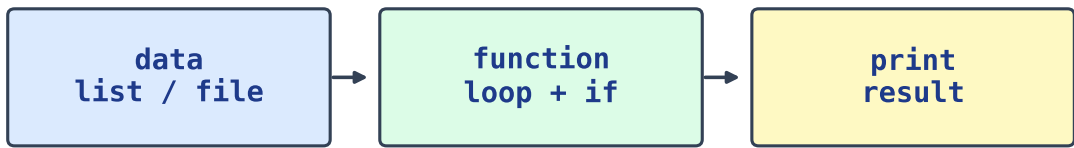
```
def merit(marks):  
    return [m for m in marks if m >= 80]  
  
print(merit([88, 71, 95, 60]))    # [88, 95]
```

AP 创作任务要求: 一个列表、一个用到选择 (selection) 和迭代 (iteration) 的带参过程 (procedure), 以及一些输入/输出。上面每个项目都正是这种形态 —— 先做小部件, 再把它们拼起来。

常见错误

- 小步构建, 每写好一部分就测试, 不要一次写完全部。
- 先读完整个题目, 再规划输入 → 处理 → 输出, 然后才开始写代码。

Mini-project pipeline



Exam tip: combine list + function + selection + loop + clear output— the Create task shape

Mini-project: data $\rightarrow$ function $\rightarrow$ printed result