Python数据类型

Python支持的数据类型包括：`int`（整数）、`float`（浮点数）、`string`（字符串）、`complex`（复数）、`bool`（布尔值）等。

注释写法

注释用于解释代码，不参与程序执行，分为以下两种：

1. **单行注释**：使用 `#` 开头，规范写法为“# 后隔一个空格再写注释内容”。
   示例：`# 这是一行单行注释`
2. **多行注释**：使用三引号 `"""""""` 包裹，常用于解释代码文件、类或方法，需写在对应对象的开头。
   示例：

```
"""
这是一段多行注释
用于说明整个代码文件的功能
或类、方法的作用
"""
```

变量定义

- 无需预先声明数据类型，直接赋值即可定义变量。
  示例：`a = 10`（自动识别为`int`类型）、`b = "hello"`（自动识别为`string`类型）
- 变量存储的数据有类型，但变量本身无类型（可通过重新赋值改变存储数据的类型）。

print语句与type语句

print语句

- 直接输出变量，支持用逗号隔开实现多个输出。
  示例：`print(a, b)`（输出变量`a`和`b`的值，默认用空格分隔）
- 默认换行输出，不换行需指定 `end` 参数，语法：`print("内容", end=' ')`（`end` 后引号内为分隔符，可自定义）。

type语句

- 功能：返回变量存储数据的类型。
  语法：`type(变量名)`
  示例：`type(a)`（若`a=10`，返回 `<class 'int'>`）

数据类型转换

转换后原变量 `x` 不变，返回转换后的新值，支持以下三种核心转换：

1. `int(x)`：将 `x` 转换为整数类型，返回整数值。
   示例：`int(3.8)` → `3`、`int("5")` → `5`

2. `float(x)`：将 `x` 转换为浮点数类型，返回浮点数。
   示例：`float(3)` → `3.0`、`float("4.2")` → `4.2`
3. `str(x)`：将 `x` 转换为字符串类型，返回字符串。
   示例：`str(100)` → `"100"`、`str(3.14)` → `"3.14"`

变量命名规则

- 核心原则：简洁明了，见名知义。
- 命名风格：采用**下划线命名法**（如 `user_name`、`total_score`）。
- 特殊规则：全英文字母大写的变量常表示常量（如 `MAX_NUM = 100`）。

Python独特运算符

与C++对比，Python有以下独特运算符，且无 `++` 和 `--` 运算：

运算符	功能说明	示例	与C++对比
`/`	除法，默认返回浮点数	`5/2` → `2.5`	C++中`/`对整数返回整数，Python无此区别
`//`	整除，返回商的整数部分	`5//2` → `2`	类似C++整数除法 `5/2`
`**`	幂运算，返回指数值	`2**3` → `8`	C++需用`pow(2,3)`，Python直接支持运算符
`and`	逻辑与	`3>2 and 5<6` → `True`	等同于C++的 `&&`
`or`	逻辑或	`3>4 or 5<6` → `True`	等同于C++的 `||`
`not`	逻辑非	`not (3>2)` → `False`	等同于C++的 `!`

字符串的多重定义方法

字符串支持三种定义方式，其中三引号与多行注释语法相同，需通过“是否有变量接收”区分：

- **单引号**：`str1 = 'hello python'`（适合内容不含单引号的场景）
- **双引号**：`str2 = "hello python"`（适合内容不含双引号的场景）
- **三引号**：`str3 = """hello\npython"""`（支持多行字符串，若无变量接收则为注释）

字符串引号嵌套

为避免引号冲突，支持以下三种嵌套方式：

1. 单引号内写双引号：`'He said "hello"'`
2. 双引号内写单引号：`"He said 'hello'"`
3. 转义字符 `\`：解除引号效用，示例：`"He said \"hello\""`（双引号内嵌套双引号）、`'He said \'hello\''`（单引号内嵌套单引号）

字符串格式化

用于将变量或表达式的值嵌入字符串，支持三种方法：

方法一：% 占位符（与C++ printf 类似）

- 语法："字符串内容 %s %d %f" % (变量1, 变量2, 变量3)
  - %s：字符串占位符（支持任意数据类型）
  - %d：整数占位符
  - %f：浮点数占位符（支持数字宽度和精度控制，如 %.2f 表示保留2位小数，四舍五入）
- 示例："姓名：%s，年龄：%d，成绩：%.2f" % ("张三", 18, 95.5) → "姓名：张三，年龄：18，成绩：95.50"

方法二：f-string（推荐）

- 语法：f"字符串内容 {变量名/表达式}"
- 优势：不限制数据类型，无需指定占位符，直接嵌入变量或表达式。
- 示例：name = "张三"; age = 18; print(f"姓名：{name}，年龄：{age+1}") → "姓名：张三，年龄：19"

方法三：表达式格式化

- 规则：上述两种方法中"放变量名的位置"均可替换为表达式，输出表达式的计算结果。
- 示例（方法一）："1+2=%d" % (1+2) → "1+2=3"；示例（方法二）：f"1+2={1+2}" → "1+2=3"

数据输入语句 input()

- 功能：读取键盘输入的值，默认返回字符串类型（需手动用数据类型转换函数转为其他类型）。
- 语法：变量名 = input("提示内容")（括号内的提示内容会在输入前显示）。
- 示例：age = int(input("请输入年龄："))（读取输入并转为整数类型存入age）

布尔类型

- 字面量：True（等价于整数1）和 False（等价于整数0）。
- 用途：用于判断条件（如分支、循环的条件表达式），返回布尔值表示"真"或"假"。

if-elif-else 分支语句

语法规则

- elif 是 else if 的简写，elif 和 else 均可省略。
- 用**缩进**（通常4个空格）表示代码的从属关系（缩进的代码属于上一级条件）。
- 支持嵌套。

```
if 条件1:
    语句1  # 条件1为True时执行
    语句2
elif 条件2:
    语句1  # 条件1为False、条件2为True时执行
    语句2
...
else:
```

```
    语句1  # 所有条件均为False时执行
    语句2
```

优化技巧

- 将 `input()` 直接放入条件中，减少代码量，示例：`if int(input("请输入分数：")) >= 60: print("及格")`

while 循环语句

语法规则

- 先判断条件，条件为 `True` 时执行循环体（无C++的 `do-while` 结构）。
- 用缩进表示循环体，支持嵌套。

```
while 条件:
    语句1  # 条件为True时重复执行
    语句2
    ...
```

- 示例（打印1-5）：

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

for 循环语句（遍历循环）

语法规则

- 用于遍历“待处理数据集”（如字符串、列表等序列类型），依次取出元素赋值给“临时变量”。
- 临时变量可在循环外访问，但规范上不建议（建议预先定义临时变量）。
- 支持嵌套。

```
for 临时变量 in 待处理数据集:
    语句1  # 每次取出一个元素执行一次
    语句2
    ...
```

- 示例（遍历字符串）：

```
for char in "hello":
    print(char)  # 依次输出 'h'、'e'、'l'、'l'、'o'
```

range 语句（生成数字序列）

用于生成连续或指定步长的数字序列，常与 `for` 循环搭配使用，支持三种语法：

1. **语法一：range(num)**
   - 功能：生成从 `0` 开始、不包含 `num` 的整数序列。
   - 示例：`range(5)` → `0,1,2,3,4`
2. **语法二：range(num1, num2)**
   - 功能：生成从 `num1` 开始、不包含 `num2` 的整数序列。
   - 示例：`range(2,5)` → `2,3,4`
3. **语法三：range(num1, num2, step)**
   - 功能：生成从 `num1` 开始、不包含 `num2`、步长为 `step` 的整数序列（`step` 默认为1，可正可负）。
   - 示例：`range(1,10,2)` → `1,3,5,7,9`；`range(10,0,-2)` → `10,8,6,4,2`

Python函数（定义与调用）

函数定义

- 语法：

```
def 函数名(传入参数):  # 传入参数可省略（无参函数）
    函数体  # 函数核心逻辑
    return 返回值  # 返回值可省略（无返回值函数）
```

- 说明：
  - 无传入参数时，括号内留空（如 `def func():`）。
  - 无返回值时，默认返回 `None`（类型为 `NoneType`，真值等同于 `False`，可用于变量预赋值）。

函数调用

- 语法：`函数名(参数)`（参数数量与定义时的“传入参数”一致）。
- 示例：

```
# 定义函数
def add(a, b):
    return a + b
# 调用函数
result = add(3, 5)
print(result)  # 输出 8
```

函数的说明文档

用于解释函数功能、参数含义和返回值，鼠标悬停函数名时可见，需写在“函数名”与“函数体”之间，用三引号包裹。

格式

```
def 函数名(传入参数):
    """
    函数说明：描述函数的核心功能
    :param 参数1：解释参数1的含义和类型
    :param 参数2：解释参数2的含义和类型
    ...
    :return：解释返回值的含义和类型
    """
    函数体
    return 返回值
```

- 示例：

```
def add(a, b):
    """
    计算两个数的和
    :param a：第一个加数（int/float）
    :param b：第二个加数（int/float）
    :return：两个数的和（int/float）
    """
    return a + b
```

变量的定义域

变量的作用范围分为“局部变量”和“全局变量”：

1. **局部变量**：在函数内部定义的变量，仅在函数内部有效，函数执行结束后销毁。
2. **全局变量**：在函数外部定义的变量，在整个代码文件中有效（函数内部可读取，但默认不可修改）。

函数内修改全局变量

需先用 `global` 声明变量为全局变量，语法：

```
global 变量名   # 写在函数体开头
变量名 = 新值   # 后续可修改
```

- 示例：

```python
num = 10  # 全局变量
def change_num():
    global num  # 声明num为全局变量
    num = 20   # 修改全局变量
change_num()
print(num)  # 输出 20
```

Python数据容器

Python支持五种核心数据容器：列表（list）、元组（tuple）、字符串（string）、集合（set）、字典（dict），各自有不同的语法、操作和适用场景。

列表（list）：可修改的有序容器

基本语法

- 定义：`变量名 = [元素1, 元素2, 元素3, ...]`（元素类型不限，支持嵌套）。
- 定义空列表：`变量名 = []` 或 `变量名 = list()`。
- 示例：`my_list = [1, "hello", 3.14, [4,5]]`（包含整数、字符串、浮点数和嵌套列表）。

下标索引

- 正向索引：从 `0` 开始（第一个元素为 `0`，第二个为 `1`，...）。
- 反向索引：从 `-1` 开始（最后一个元素为 `-1`，倒数第二个为 `-2`，...）。
- 访问元素：`列表名[索引]`；访问嵌套列表元素：`列表名[外层索引][内层索引]`。
- 示例：`my_list[0]` → `1`；`my_list[3][1]` → `5`。

列表操作（方法）

操作语法	功能说明	示例
`列表名.index(元素)`	返回元素的正向索引，无此元素报错 `ValueError`	`[1,2,3].index(2)` → `1`
`列表名[索引] = 值`	修改指定索引的元素值	`my_list[1] = "world"` → `[1, "world", 3.14, [4,5]]`
`列表名.insert(索引, 元素)`	在指定索引插入元素	`my_list.insert(2, "new")` → `[1, "world", "new", 3.14, [4,5]]`
`列表名.append(元素)`	在列表尾部追加元素	`my_list.append(6)` → `[1, "world", "new", 3.14, [4,5], 6]`
`列表名.extend(容器)`	将其他容器的元素依次追加到列表尾部	`my_list.extend([7,8])` → `[1, "world", "new", 3.14, [4,5], 6,7,8]`
`del 列表名[索引]`	删除指定索引的元素	`del my_list[2]` → 移除 `"new"`

操作语法	功能说明	示例
列表名.pop(索引)	删除指定索引的元素，并返回该元素	my_list.pop(3) → 返回 [4,5]，列表移除该元素
列表名.remove(元素)	删除第一个匹配的元素	[1,2,2,3].remove(2) → [1,2,3]
列表名.clear()	清空列表	my_list.clear() → []
列表名.count(元素)	返回元素在列表中的个数	[1,2,2,3].count(2) → 2
len(列表名)	返回列表的元素总数	len([1,2,3]) → 3

列表遍历

- **方式一：while循环（按索引遍历，更灵活）**：

```
my_list = [1,2,3]
i = 0
while i < len(my_list):
    print(my_list[i])
    i += 1
```

- **方式二：for循环（直接遍历元素，更简洁）**：

```
for elem in my_list:
    print(elem)
```

列表特点

- 可容纳多个、不同类型的元素；
- 数据有序（有下标），支持重复元素；
- 可修改，支持下标索引和for循环。

元组（tuple）：不可修改的有序容器

基本语法

- 定义：变量名 = (元素1, 元素2, ...)（元素类型不限，支持嵌套）。
- 注意：单个元素的元组需在元素后加逗号（如 (1,)，否则会被识别为普通数据类型）。
- 定义空元组：变量名 = () 或 变量名 = tuple()。
- 示例：my_tuple = (1, "hello", (2,3))（包含整数、字符串和嵌套元组）。

下标索引与列表一致

- 访问元素：`元组名[索引]`；访问嵌套元组：`元组名[外层索引][内层索引]`。
- 示例：`my_tuple[2][0]` → `2`。

元组操作（方法）

元组不可修改，仅支持以下只读操作：

操作语法	功能说明
`元组名.index(元素)`	返回元素的正向索引，无此元素报错 `ValueError`
`元组名.count(元素)`	返回元素在元组中的个数
`len(元组名)`	返回元组的元素总数

元组遍历

与列表一致（while按索引、for直接遍历元素）。

元组特点

- 可容纳多个、不同类型的元素；
- 数据有序（有下标），支持重复元素；
- **不可修改**（若元素为列表，列表内部可修改）；
- 支持下标索引和for循环。

字符串（string）：不可修改的字符序列

下标索引

与列表、元组一致（正向从`0`、反向从`-1`），访问语法：`字符串名[索引]`。

- 示例：`s = "hello"; s[1]` → `'e'`；`s[-1]` → `'o'`。

字符串操作（方法）

字符串不可修改，操作后返回新字符串：

操作语法	功能说明
`字符串名.index(子串)`	返回子串的起始正向索引，无此子串报错 `ValueError`
`字符串名.replace(子串1, 子串2)`	将所有子串1替换为子串2，返回新字符串（原字符串不变）
`字符串名.split(分隔符)`	按分隔符分割字符串，返回子串组成的列表
`字符串名.strip()`	去除前后空格和换行符，返回新字符串
`字符串名.strip(子串)`	去除前后指定字符（按单个字符匹配），返回新字符串
`字符串名.count(子串)`	返回子串在字符串中的个数
`len(字符串名)`	返回字符串的长度（字符个数）

字符串遍历

与列表、元组一致（while按索引、for直接遍历字符）。

字符串特点

- 仅容纳字符（串），长度任意；
- 数据有序（有下标），支持重复字符；
- **不可修改**；
- 支持下标索引和for循环。

序列的切片操作

“序列”指内容连续、有序、支持下标索引的容器（列表、元组、字符串），切片用于从序列中提取子序列。

语法

```
序列名[起始下标：结束下标：步长]
```

- 起始下标：留空表示从序列开头开始；
- 结束下标：不包含该索引（取到“结束下标-1”），留空表示取到序列结尾；
- 步长：默认1（连续取），负步长表示反向取（此时起始下标需大于结束下标）。

示例

序列	切片语法	结果
`[1,2,3,4,5]`	`[1:4]`	`[2,3,4]`（取索引1-3）
`[1,2,3,4,5]`	`[:3]`	`[1,2,3]`（从开头取到索引2）
`[1,2,3,4,5]`	`[2:]`	`[3,4,5]`（从索引2取到结尾）
`[1,2,3,4,5]`	`[::2]`	`[1,3,5]`（步长2，间隔1个元素取）
`[1,2,3,4,5]`	`[4:1:-1]`	`[5,4,3]`（反向取，从索引4到2）

集合（set）：不可重复的无序容器

基本语法

- 定义：`变量名 = {元素1, 元素2, ...}`（元素类型不限，不支持嵌套）。
- 定义空集合：**必须用** `变量名 = set()`（`{}` 表示空字典，非空集合）。
- 示例：`my_set = {1, "hello", 3.14}`（自动去重，如 `{1,1,2}` → `{1,2}`）。

集合操作（方法）

操作语法	功能说明
`集合名.add(元素)`	向集合中添加元素（重复元素不生效）
`集合名.remove(元素)`	移除指定元素，无此元素报错 `KeyError`

操作语法	功能说明
集合名.pop()	随机移除一个元素，并返回该元素
集合名.clear()	清空集合
集合1.difference(集合2)	返回集合1与集合2的差集（集合1有、集合2无的元素），原集合不变
集合1.difference_update(集合2)	在集合1中移除与集合2相同的元素，原集合1修改
集合1.union(集合2)	返回集合1与集合2的并集（所有不重复元素），原集合不变
len(集合名)	返回集合的元素总数

集合遍历

仅支持for循环直接遍历（无下标，无序）：

```
my_set = {1,2,3}
for elem in my_set:
    print(elem)  # 输出顺序不固定（如 2,1,3）
```

集合特点

- 可容纳多个、不同类型的元素；
- 数据无序（无下标），**不支持重复元素**；
- 可修改，支持for循环，不支持下标索引。

字典（dict）：键值对映射的无序容器

基本语法

- 定义：`变量名 = {key1: value1, key2: value2, ...}`（`key` 为键，`value` 为值，组成键值对）。
- 规则：`key` 不可重复（重复添加会覆盖原有 `value`），`key` 类型除字典外均可（如int、string），`value` 类型不限（支持嵌套）。
- 定义空字典：`变量名 = {}` 或 `变量名 = dict()`。
- 示例：`my_dict = {"name": "张三", "age": 18, "scores": [90, 85]}`（嵌套列表作为value）。

字典索引（按key访问）

- 访问value：`字典名[key]`；访问嵌套value：`字典名[外层key][内层key/索引]`。
- 示例：`my_dict["age"]` → `18`；`my_dict["scores"][0]` → `90`。

字典操作（方法）

操作语法	功能说明

操作语法	功能说明
`字典名[key] = value`	若key存在则更新value，若不存在则新增键值对
`字典名.pop(key)`	移除指定key的键值对，返回对应的value
`字典名.clear()`	清空字典
`字典名.keys()`	返回所有key组成的序列
`len(字典名)`	返回字典的键值对总数

字典遍历

- **方式一：遍历key（推荐）**：

```
for key in my_dict.keys():
    print(key, my_dict[key])  # 输出 key 和对应的 value
```

- **方式二：直接遍历字典（默认遍历key）**：

```
for key in my_dict:
    print(key, my_dict[key])
```

字典特点

- 以键值对存储数据，可容纳多个、不同类型的value；
- 数据无序（无下标），`key` 不可重复；
- 可修改，支持for循环，不支持下标索引（按key访问）。

数据容器特点对比

特性	列表（list）	元组（tuple）	字符串（string）	集合（set）	字典（dict）
元素数量	支持多个	支持多个	支持多个	支持多个	支持多个（键值对）
元素类型	任意类型	任意类型	仅字符	任意类型	key：除字典外任意；value：任意
下标索引	支持	支持	支持	不支持	不支持
重复元素	支持	支持	支持	不支持	key不支持，value支持
可修改性	支持	不支持	不支持	支持	支持

特性	列表（list）	元组（tuple）	字符串（string）	集合（set）	字典（dict）
数据有序	是	是	是	否	否（Python 3.7+有序）
使用场景	可修改、可重复的批量数据记录	不可修改、可重复的批量数据记录	字符序列存储	不可重复的唯一数据记录	按key检索value的映射场景

数据容器通用操作

所有数据容器均支持以下基础操作：

1. **for循环遍历**：依次取出容器元素（集合、字典无下标，列表、元组、字符串支持下标遍历）。
2. **len(容器名)**：返回容器的元素总数（字典返回键值对数量）。
3. **max(容器名)/min(容器名)**：返回容器中最大/最小元素（字典比较key的大小）。
4. **类型转换**：
   - `list(容器)`：转为列表（字符串转列表→单个字符；字典转列表→key）；
   - `str(容器)`：转为字符串（如 `str([1,2])` → `"[1,2]"`）；
   - `tuple(容器)`：转为元组（规则同列表转换）；
   - `set(容器)`：转为集合（自动去重，规则同列表转换）。
5. **sorted(容器)**：对容器元素排序，返回排序后的列表（默认升序）；降序需加参数 `sorted(容器, reverse=True)`（字典按key排序）。

函数进阶

返回多个返回值

- **语法**：函数中通过 `return 返回值1，返回值2……` 同时返回多个值；调用时通过 `变量名1，变量名2…… = 函数名(参数)` 接收。
- 示例：

```
def get_info():
    name = "张三"
    age = 18
    return name, age  # 返回多个值
# 接收多个返回值
user_name, user_age = get_info()
print(user_name, user_age)  # 输出：张三 18
```

多种传参方式

位置参数

- 定义：根据函数定义的参数位置顺序传递参数，实参顺序需与形参完全一致。
- 示例：

```
def add(a, b):
    return a + b
result = add(3, 5)  # 3对应a，5对应b，按位置传参
print(result)  # 输出：8
```

关键字参数

- 定义：通过 键=值 形式传递参数，无需遵循形参顺序，可明确参数对应关系。
- 注意：位置参数与关键字参数混用时，**位置参数必须在关键字参数之前。**
- 示例：

```
result = add(b=5, a=3)  # 关键字传参，顺序可调整
print(result)  # 输出：8
```

缺省参数

- 定义：函数定义时给末尾的形参赋默认值，调用时可省略该参数（使用默认值），若传入则覆盖默认值。
- 示例：

```
def add(a, b=2):  # b为缺省参数，默认值2
    return a + b
print(add(3))  # 省略b，用默认值2，输出：5
print(add(3, 5))  # 传入b=5，覆盖默认值，输出：8
```

不定长参数

- 用于接收不确定数量的参数，分为"位置传递不定长"和"关键字传递不定长"两种：
  1. **位置传递不定长**：形参前加 * （通常用 *args），接收的参数会合并为一个元组。 示例：

```
def sum_args(*args):
    total = 0
    for num in args:
        total += num
    return total
print(sum_args(1, 2, 3))  # 传入3个参数，输出：6
```

  2. **关键字传递不定长**：形参前加 ** （通常用 **kwargs），接收的键值对参数会合并为一个字典。示例：

```
def print_kwargs(**kwargs):
    for key, value in kwargs.items():
```

```
        print(f"{key}: {value}")
print_kwargs(name="张三", age=18)  # 传入2个键值对，输出：name：张三  age：18
```

匿名函数

- 定义：用 `lambda` 关键字定义的“一次性”函数，无函数名，仅包含一行代码，常用于作为参数传递。
- **语法**：`lambda 传入参数：函数体`（函数体无需写 `return`，直接返回计算结果）。
- 示例（作为函数参数传递）：

```
def calculate(func, a, b):
    return func(a, b)
# 用lambda定义匿名函数，作为参数传入
result_add = calculate(lambda x, y: x + y, 3, 5)
result_mul = calculate(lambda x, y: x * y, 3, 5)
print(result_add)  # 输出：8
print(result_mul)  # 输出：15
```

文件操作

文件操作核心流程为“打开文件→操作文件（读/写）→关闭文件”，具体语法如下：

打开文件

- **语法**：`文件对象名 = open(name, mode, encoding="编码格式")`
- 参数说明：
  - `name`：目标文件名（可包含路径，如 `"./test.txt"`）；
  - `mode`：打开模式（核心模式：`"r"` 只读、`"w"` 写入（覆盖原有内容，文件不存在则创建）、`"a"` 追加（在文件末尾添加，文件不存在则创建））；
  - `encoding`：编码格式（推荐 `UTF-8`，避免中文乱码）。
- 示例：`file = open("test.txt", "r", encoding="UTF-8")`（只读方式打开test.txt）。

读操作

用于读取文件内容，常见方式有4种：

1. **`文件对象.read(num)`**：读取 `num` 字节的内容，`num` 省略则读取全部内容，返回字符串。 示例：`content = file.read(10)`（读取前10字节）。
2. **`文件对象.readline()`**：读取一行内容，返回字符串（每次调用读一行，文件指针下移）。 示例：`line1 = file.readline()`（读第一行）、`line2 = file.readline()`（读第二行）。
3. **`文件对象.readlines()`**：一次性读取所有行，返回每行内容组成的列表。 示例：`lines = file.readlines()`（`lines[0]` 为第一行，`lines[1]` 为第二行）。
4. `for` **循环读取**：逐行读取，语法简洁，适合大文件。 示例：

```
for line in file:
    print(line)  # 逐行输出文件内容
```

- 注意：读取后**文件指针**会指向读取的最后一个字节的下一个位置，再次读取从指针位置开始。

写操作

用于向文件写入内容，需配合 `mode="w"` 或 `mode="a"`：

1. **`文件对象.write(内容)`**：将字符串"内容"写入文件（先存入内存缓冲区，未立即写入磁盘）。
2. **`文件对象.flush()`**：刷新缓冲区，将内存中的内容真正写入磁盘（确保内容保存）。

- 示例（写入内容）：

```
file = open("test.txt", "w", encoding="UTF-8")
file.write("Hello Python")  # 写入内容到缓冲区
file.flush()  # 刷新到磁盘
```

- 注意：`close()` 方法内置 `flush()` 功能，关闭文件时会自动刷新。

关闭文件

- **语法**：`文件对象.close()`，关闭后文件对象不可再使用，避免资源占用。
- 优化方案：用 `with open` 语法，操作完成后**自动关闭文件**，无需手动调用 `close()`：

```
with open("test.txt", "r", encoding="UTF-8") as file:
    content = file.read()  # 缩进内操作文件
    print(content)
# 缩进外文件已自动关闭，无需手动close()
```

捕获异常

用于处理程序运行中的错误（如文件不存在、除数为0等），避免程序崩溃，核心语法为 `try-except` 结构。

基本语法

```
try:
    可能发生错误的代码  # 监控该代码块的异常
except:
    如果出现异常执行的代码  # 异常发生时执行
else:
    如果没有异常执行的代码  # 无异常时执行（可选）
finally:
    无论有无异常都要执行的代码  # 必执行（可选，如关闭资源）
```

- 示例（捕获文件不存在异常）：

```
try:
    file = open("nonexistent.txt", "r")
except:
    print("文件不存在，无法读取")
else:
    content = file.read()
    print(content)
finally:
    print("操作结束")
```

捕获指定异常

仅捕获特定类型的异常，可通过 `as` 给异常起别名，查看异常详情：

```
try:
    可能发生错误的代码
except 异常名 as 异常别名:
    如果出现该异常执行的代码  # 仅当发生指定异常时执行
else:
    无异常执行的代码
finally:
    必执行的代码
```

- 说明：`Exception` 代表**所有异常类型**，可捕获任意异常。
- 示例（捕获除零异常）：

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"发生错误：{e}")  # 输出：发生错误：division by zero
```

捕获多个异常

同时捕获多种类型的异常，用元组包裹多个异常名：

```
try:
    可能发生错误的代码
except (异常名1, 异常名2, ...) as 异常别名:
    如果出现任一异常执行的代码
else:
    无异常执行的代码
finally:
    必执行的代码
```

- 示例（同时捕获文件不存在和除零异常）：

```
try:
    file = open("test.txt", "r")
    result = 10 / 0
except (FileNotFoundError, ZeroDivisionError) as e:
    print(f"发生错误：{e}")
```

异常的传递性

异常会在函数调用链中传递，而非直接报错。若外层函数未捕获异常，异常会继续向上传递，直到被捕获或导致程序崩溃。

- 示例：

```
def func1():
    10 / 0  # 产生除零异常
def func2():
    func1()  # 调用func1，异常传递到func2
try:
    func2()  # 调用func2，捕获传递的异常
except ZeroDivisionError as e:
    print(f"捕获到异常：{e}")
```

模块（Python工具包）

模块是包含Python代码（函数、类、变量）的 `.py` 文件，用于代码复用和模块化开发。

模块的导入

导入模块的核心语法为 `[from 模块名] import [模块/类/变量/函数/*] [as 别名]`，中括号内为可选项，常见组合如下：

导入方式	语法示例	使用方式
导入整个模块	`import module_name`	`module_name.功能名()`
导入模块中的指定功能	`from module_name import func1, cls1`	直接用 `func1()`、`cls1()`
导入模块中的所有功能	`from module_name import *`	直接用所有功能（不推荐，易冲突）
导入模块并指定别名	`import module_name as mn`	`mn.功能名()`
导入功能并指定别名	`from module_name import func1 as f1`	`f1()`

- 示例（导入Python内置 `math` 模块）：

```
import math as m
print(m.sqrt(16))  # 调用math模块的sqrt函数，输出：4.0
```

模块的使用

- 导入整个模块：需通过“模块名.功能名”调用，如 `math.pi`（访问math模块的pi变量）。
- 导入指定功能：可直接调用功能名，如 `from math import pi; print(pi)`。

自定义模块

在 `.py` 文件中定义函数/类，即可作为模块导入使用，需注意以下细节：

1. **同名函数覆盖**：若导入的模块与当前文件有同名函数，后导入的函数会覆盖先导入的。
2. **避免模块执行代码**：模块中若有函数调用代码，导入时会自动执行。可通过 `if __name__ == '__main__':` 包裹调用代码，仅在模块自身运行时执行，导入时不执行：

```
# 自定义模块 my_module.py
def add(a, b):
    return a + b
# 仅当直接运行my_module.py时执行，导入时不执行
if __name__ == '__main__':
    print(add(3, 5))
```

3. **`__all__` 变量**：在模块中定义 `__all__ = ['功能名1', '功能名2', ...]`，限制 `from 模块名 import *` 仅能导入列表中的功能：

```
# my_module.py
__all__ = ['add']  # 仅允许导入add函数
def add(a, b):
    return a + b
def sub(a, b):
    return a - b
# 导入时：from my_module import * 仅能获取add，无法获取sub
```

Python的包（模块集合）

包是包含多个模块和 `__init__.py` 文件的文件夹，用于组织多个相关模块，避免模块名冲突。

包的导入与使用

包的导入方式与模块类似，核心是通过“包名.模块名”定位模块，常见方式如下：

导入方式	语法示例	使用方式
导入包中的模块	`import package_name.module_name`	`package_name.module_name.功能名()`

导入方式	语法示例	使用方式
从包中导入模块	`from package_name import module_name`	`module_name.功能名()`
从包的模块中导入功能	`from package_name.module_name import func1`	直接用 `func1()`

- 示例（假设有包 `my_package`，包含模块 `my_module`）：

```
from my_package.my_module import add
print(add(3, 5))  # 输出：8
```

`__init__.py` 文件的作用

- 标识文件夹为Python包（即使文件为空）。
- 定义 `__all__` 变量：限制 `from 包名 import *` 仅能导入列表中的模块：

```
# my_package/__init__.py
__all__ = ['my_module']  # 仅允许导入my_module模块
```

安装第三方包

第三方包是开发者共享的工具包，需先安装再使用，常见安装方式：

1. **`pip` 命令安装**：在终端执行 `pip install 包名`，如 `pip install numpy`。
2. **PyCharm图形化安装**：通过“File → Settings → Project: XXX → Python Interpreter”，点击“+”搜索包名并安装。

- 常见第三方包：
  - `numpy`：科学计算（矩阵、数组操作）；
  - `pandas`：数据分析（表格数据处理）；
  - `pyspark`/`apache-flink`：大数据计算；
  - `matplotlib`/`pyecharts`：数据可视化（绘图）；
  - `tensorflow`：人工智能（深度学习）。

Python可视化图表案例

JSON数据交互

JSON是用于跨语言数据传递的“特定格式字符串”，Python中通过 `json` 模块实现JSON与Python数据的转换。

核心语法

- 导入模块：`import json`；
- `json.dumps(data)`：将Python数据（字典/嵌套字典的列表）转为JSON字符串，含中文字符需加 `ensure_ascii=False`；

- `json.loads(data)`：将JSON字符串转为Python数据（字典/列表）。
- 示例：

```
import json
# Python字典转JSON字符串
python_data = {"name": "张三", "age": 18}
json_str = json.dumps(python_data, ensure_ascii=False)
print(json_str)  # 输出：{"name": "张三", "age": 18}

# JSON字符串转Python字典
json_data = '{"name": "李四", "age": 20}'
python_dict = json.loads(json_data)
print(python_dict["name"])  # 输出：李四
```

Pyecharts初步（数据可视化库）

Pyecharts用于生成交互式HTML图表，支持折线图、地图、柱状图等，以下为核心图表的基础用法：

基础折线图

- **步骤**：导包 → 创建折线图对象 → 添加X/Y轴数据 → 配置全局选项 → 生成HTML文件。
- 示例：

```
# 1. 导包
from pyecharts.charts import Line
from pyecharts.options import TitleOpts, LegendOpts, ToolboxOpts

# 2. 创建折线图对象
line = Line()

# 3. 添加X/Y轴数据（X轴为列表，Y轴为“系列名+列表”）
line.add_xaxis(["1月", "2月", "3月", "4月"])
line.add_yaxis("销售额", [100, 150, 120, 180])

# 4. 配置全局选项（标题、图例、工具箱等）
line.set_global_opts(
    title_opts=TitleOpts(title="月度销售额", pos_left="center"),  # 标题居中
    legend_opts=LegendOpts(is_show=True),  # 显示图例
    toolbox_opts=ToolboxOpts(is_show=True)   # 显示工具箱（下载、刷新等）
)

# 5. 生成HTML图表文件
line.render("sales_line.html")
```

基础地图

- **步骤**：导包 → 创建地图对象 → 定义地图数据 → 添加数据（指定国家/地区） → 配置全局选项 → 生成HTML文件。
- 示例（中国省份数据地图）：

```
# 1. 导包
from pyecharts.charts import Map
from pyecharts.options import VisualMapOpts

# 2. 创建地图对象
map_chart = Map()

# 3. 定义地图数据（嵌套元组列表：[(省份名，数值)，...]）
data = [("北京市", 100), ("上海市", 150), ("广东省", 200)]

# 4. 添加数据（地图名：如"中国"，数据，地区名）
map_chart.add("数据指标", data, "中国")

# 5. 配置全局选项（视觉映射：手动调整数据范围）
map_chart.set_global_opts(
    visualmap_opts=VisualMapOpts(
        is_show=True,
        is_piecewise=True,  # 开启手动分段
        pieces=[
            {"min": 0, "max": 100, "label": "0-100", "color": "#FFE4B5"},
            {"min": 101, "max": 150, "label": "101-150", "color":
"#FFA500"},
            {"min": 151, "max": 200, "label": "151-200", "color": "#FF4500"}
        ]
    )
)

# 6. 生成HTML文件
map_chart.render("china_map.html")
```

基础柱状图

- **步骤**：导包 → 创建柱状图对象 → 添加X/Y轴数据 → 配置选项（如反转轴、调整标签位置） → 生成HTML文件。
- 示例（含Y轴标签位置调整和轴反转）：

```
# 1. 导包
from pyecharts.charts import Bar
from pyecharts.options import TitleOpts, LabelOpts

# 2. 创建柱状图对象
bar = Bar()

# 3. 添加X/Y轴数据（调整Y轴标签位置：position="top"）
bar.add_xaxis(["苹果", "香蕉", "橙子"])
```

```
bar.add_yaxis(
    "销量",
    [50, 30, 40],
    label_opts=LabelOpts(position="top")  # Y轴数值显示在柱子顶部
)

# 4. 反转X/Y轴（横向柱状图）
bar.reversal_axis()

# 5. 配置标题
bar.set_global_opts(title_opts=TitleOpts(title="水果销量",
pos_left="center"))

# 6. 生成HTML文件
bar.render("fruit_bar.html")
```

数据处理

利用JSON工具逐层解析复杂数据，提取所需的“数据列表”（如从JSON字符串中提取X轴和Y轴数据），再传入Pyecharts生成图表。

- 示例（解析JSON数据生成折线图）：

```
import json
from pyecharts.charts import Line

# 1. 解析JSON数据
json_data = '''
{"month": ["1月", "2月", "3月"], "sales": [120, 180, 150]}
'''
data_dict = json.loads(json_data)
x_data = data_dict["month"]  # 提取X轴数据
y_data = data_dict["sales"]  # 提取Y轴数据

# 2. 生成折线图
line = Line()
line.add_xaxis(x_data)
line.add_yaxis("销售额", y_data)
line.render("json_line.html")
```

面向对象——封装

封装是面向对象的核心特性之一，将数据（属性）和操作数据的方法封装在类中，隐藏内部细节，对外提供接口。

类的定义与使用

类的定义

- **语法**：

```
class 类名称:
    # 类的属性（成员变量）
    属性名 = 初始值

    # 类的方法（成员方法）
    def 方法名(self, 形参1, 形参2...):
        方法体  # 访问属性需用 self.属性名
```

- 说明：`self` 是成员方法的**必传参数**，代表类的实例对象，无需手动传入，Python自动传递。

创建类对象

- **语法**：`对象名 = 类名称()`，通过对象访问属性和方法：`对象名.属性名`、`对象名.方法名(参数)`。
- 示例：

```
# 定义类
class Student:
    # 属性
    name = ""
    age = 0

    # 方法
    def introduce(self):
        print(f"我叫{self.name}，今年{self.age}岁")

# 创建对象
stu1 = Student()
# 赋值属性
stu1.name = "张三"
stu1.age = 18
# 调用方法
stu1.introduce()  # 输出：我叫张三，今年18岁
```

魔术方法

魔术方法是Python类中以 __ 开头和结尾的特殊方法，自动触发执行，常见如下：

`__init__()` 方法（构造方法）

- 作用：创建对象时**自动执行**，用于初始化对象的属性（给属性赋值）。
- 语法：

```
class 类名称:
    def __init__(self, 形参1, 形参2...):
        # 初始化属性（self.属性名 = 形参）
```

```
        self.属性名1 = 形参1
        self.属性名2 = 形参2
```

- 示例：

```
class Student:
    # 构造方法：创建对象时传入name和age
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"我叫{self.name}，今年{self.age}岁")

# 创建对象时直接传参，自动调用__init__
stu1 = Student("张三", 18)
stu1.introduce()  # 输出：我叫张三，今年18岁
```

`__str__()` 方法（字符串方法）

- 作用：对象转换为字符串或直接打印时，**自动调用**，控制输出结果（默认输出对象地址，通过该方法自定义）。
- 语法：

```
def __str__(self):
    return "自定义的字符串内容"  # 必须返回字符串
```

- 示例：

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"Student(name={self.name}, age={self.age})"

stu1 = Student("张三", 18)
print(stu1)  # 自动调用__str__，输出：Student(name=张三, age=18)
```

`__lt__()` 方法（小于/大于比较）

- 作用：重载 `<` 运算符，实现类对象之间的“小于”比较（支持反向推导 `>`）。
- 语法：

```
def __lt__(self, other):
    # self：当前对象，other：另一个对象，返回布尔值
    return self.属性名 < other.属性名
```

- 示例（按年龄比较）：

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __lt__(self, other):
        return self.age < other.age

stu1 = Student("张三", 18)
stu2 = Student("李四", 20)
print(stu1 < stu2)  # 调用__lt__，输出：True
print(stu1 > stu2)  # 反向推导，输出：False
```

`__le__()` 方法（小于等于/大于等于比较）

- 作用：重载 `<=` 运算符，实现类对象之间的“小于等于”比较（支持反向推导 `>=`）。
- 语法：

```
def __le__(self, other):
    return self.属性名 <= other.属性名
```

- 示例：

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __le__(self, other):
        return self.age <= other.age

stu1 = Student("张三", 18)
stu2 = Student("李四", 18)
print(stu1 <= stu2)  # 输出：True
```

`__eq__()` 方法（等于比较）

- 作用：重载 `==` 运算符，自定义类对象的“等于”判断逻辑（默认比较对象地址，通过该方法比较属性）。

- 语法：

```
def __eq__(self, other):
    return self.属性名 == other.属性名
```

- 示例（按姓名和年龄比较是否相等）：

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __eq__(self, other):
        return self.name == other.name and self.age == other.age

stu1 = Student("张三", 18)
stu2 = Student("张三", 18)
stu3 = Student("李四", 18)
print(stu1 == stu2)  # 输出：True
print(stu1 == stu3)  # 输出：False
```

私有变量和私有方法

- 定义：在属性或方法名前加 __（双下划线），即为私有成员。
- 特性：
  - 不能通过对象直接访问（对象名.__私有成员 报错）；
  - 可在类内部直接访问（私有方法可调用私有变量，私有方法之间可互相调用）。
- 示例：

```
class Student:
    def __init__(self, name, score):
        self.name = name
        self.__score = score  # 私有变量：分数（外部不可直接访问）

    # 私有方法：内部计算等级
    def __get_grade(self):
        if self.__score >= 90:
            return "A"
        else:
            return "B"

    # 公有方法：对外提供接口，访问私有成员
    def show_info(self):
        grade = self.__get_grade()  # 内部调用私有方法
        print(f"姓名：{self.name}，等级：{grade}")

stu1 = Student("张三", 85)
```

```
stu1.show_info()  # 输出：姓名：张三，等级：B
# stu1.__score  # 报错：无法直接访问私有变量
# stu1._get_grade()  # 报错：无法直接调用私有方法
```

面向对象——继承

继承是面向对象的另一核心特性，子类可继承父类的所有属性和方法，实现代码复用，并可扩展新功能或修改父类功能。

基础语法

- **语法**：`class 子类名(父类名)：类内容体`
- 特性：子类自动继承父类的所有非私有属性和方法，可直接调用；子类可定义新属性/方法，扩展功能。
- 示例：

```
# 父类（基类）
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"我叫{self.name}，今年{self.age}岁")

# 子类（派生类）：继承Person
class Student(Person):
    # 子类新增属性
    def __init__(self, name, age, student_id):
        # 调用父类的__init__，初始化父类属性
        super().__init__(name, age)
        self.student_id = student_id  # 子类新增属性：学号

    # 子类新增方法
    def show_id(self):
        print(f"学号：{self.student_id}")

# 创建子类对象
stu1 = Student("张三", 18, "2025001")
stu1.introduce()  # 继承父类方法，输出：我叫张三，今年18岁
stu1.show_id()    # 子类新增方法，输出：学号：2025001
```

多继承

- 定义：子类同时继承多个父类，语法：`class 子类名(父类名1，父类名2，...)：类内容体`
- 特性：子类继承所有父类的非私有属性和方法；若多个父类有同名成员，**优先调用先传入的父类成员**（左优先原则）。
- 示例：

```
# 父类1
class A:
    def say(self):
        print("A的方法")

# 父类2
class B:
    def say(self):
        print("B的方法")

# 子类：继承A和B（A在前，B在后）
class C(A, B):
    pass

c = C()
c.say()  # 优先调用A的say方法，输出：A的方法
```

pass关键字

- 作用：当类或方法中暂时无内容时，用 `pass` 填充占位，避免语法错误（相当于“空实现”）。
- 示例：

```
class EmptyClass:
    pass  # 空类，无属性和方法，不报错

def empty_func():
    pass  # 空方法，无代码，不报错
```

复写（重写）

- 定义：子类定义与父类同名的方法/属性，覆盖父类的成员，调用时默认使用子类的成员（不修改父类本身）。
- 调用父类成员：复写后，若需在子类中调用父类的成员，有两种方式：

通过“父类名.成员”调用

- 语法：`父类名.属性名`、`父类名.方法名(self, 参数)`（需手动传递 `self`）。
- 示例：

```
class Person:
    def introduce(self):
        print("我是Person")

class Student(Person):
    def introduce(self):
        # 调用父类的introduce方法
        Person.introduce(self)
```

```
        # 子类自己的逻辑
        print("我是Student")

stu1 = Student()
stu1.introduce()  # 输出：我是Person  我是Student
```

通过`super()`调用

- 语法：`super().属性名`、`super().方法名(参数)`（无需传递 `self`，更简洁）。
- 示例：

```
class Student(Person):
    def introduce(self):
        # 调用父类的introduce方法
        super().introduce()
        # 子类自己的逻辑
        print("我是Student")

stu1 = Student()
stu1.introduce()  # 输出：我是Person  我是Student
```

类型注解

类型注解是给变量或函数参数/返回值打上“类型标记”，用于提示数据类型（非强制约束，不影响程序运行），提升代码可读性和IDE提示效果。

基本语法

变量的类型注解

- 一般写法：`变量名: 类型`（类型可为 `int`、`str`、类名等）；
- 注释写法：变量定义后加 `# type: 类型`（适用于隐式定义的变量）；
- 数据容器的详细注解：
  - 列表/集合：`变量名: list[元素类型]` / `变量名: set[元素类型]`；
  - 元组：`变量名: tuple[元素类型1, 元素类型2...]`（需指定每个元素的类型）；
  - 字典：`变量名: dict[键类型, 值类型]`。
- 示例：

```
# 一般写法
name: str = "张三"
age: int = 18
scores: list[int] = [90, 85, 95]
info: dict[str, str] = {"name": "张三", "gender": "男"}
```

```
# 注释写法
data = [1, 2, 3]  # type: list[int]
```

函数的类型注解

- 形参注解：`def 函数名(形参1: 类型1, 形参2: 类型2...):`；
- 返回值注解：`def 函数名(...) -> 返回值类型:`。
- 示例：

```
# 形参注解：a和b为int，返回值注解：返回int
def add(a: int, b: int) -> int:
    return a + b

# 调用时IDE会提示参数类型，若传入非int类型，IDE会警告（不影响运行）
result: int = add(3, 5)
```

Union类型（混合类型注解）

- 作用：描述变量或参数为"多种类型中的一种"（混合类型）。
- 语法：
  1. 导入模块：`from typing import Union`；
  2. 定义类型：`Union[类型1, 类型2, ...]`。
- 示例：

```
from typing import Union

# 变量可为int或str类型
var: Union[int, str] = 10
var = "hello"  # 合法

# 函数参数可为int或float，返回值可为int或float
def calculate(a: Union[int, float], b: Union[int, float]) -> Union[int,
float]:
    return a * b

calculate(3, 5.5)  # 合法，返回16.5
```

多态

多态是面向对象的第三大特性，基于继承实现：**同样的行为（调用同一方法），传入不同的对象，得到不同的结果。**

定义与示例

- 核心逻辑：定义函数时，通过类型注解声明“接收父类对象”；调用函数时，传入不同的子类对象，执行子类复写后的方法，得到不同结果。
- 示例：

```python
# 父类
class Animal:
    def speak(self):
        pass  # 父类方法空实现（抽象方法）

# 子类1：复写speak
class Dog(Animal):
    def speak(self):
        print("汪汪汪")

# 子类2：复写speak
class Cat(Animal):
    def speak(self):
        print("喵喵喵")

# 函数：接收Animal类型对象（父类类型）
def make_speak(animal: Animal):
    animal.speak()  # 调用speak方法，多态体现

# 传入不同子类对象，执行不同逻辑
make_speak(Dog())  # 输出：汪汪汪
make_speak(Cat())  # 输出：喵喵喵
```

抽象类（接口）

- 定义：含有“抽象方法”（空实现 `pass`）的类，作为“顶层设计”，规定子类必须实现的方法，自身不直接创建对象（仅用于被继承）。
- 特性：
  - 抽象方法：父类中定义，无具体实现（`pass`），子类必须复写该方法；
  - 抽象类不能直接实例化（`Animal()` 报错，需通过子类实例化）。
- 作用：统一子类的方法名称和参数，确保多态的一致性。

推导式

推导式是Python的简洁语法，用于从一个数据序列（列表、元组、集合、字典）快速构建另一个新的数据序列，减少代码量。

列表推导式

- 作用：从现有序列生成新列表，支持条件过滤和结果分支。
- 三种语法形式：
  1. 基础形式：`[表达式 for 变量 in 原序列]`（遍历原序列，用表达式处理变量，生成新列表）；
  2. 条件过滤：`[表达式 for 变量 in 原序列 if 条件]`（仅当条件为True时，生成元素）；
  3. 结果分支：`[结果1 if 条件 else 结果2 for 变量 in 原序列]`（根据条件生成不同结果）。

- 示例：

```
# 1. 基础形式：生成1-5的平方列表
squares = [x*x for x in range(1,6)]
print(squares)  # 输出：[1, 4, 9, 16, 25]

# 2. 条件过滤：生成1-10中的偶数列表
evens = [x for x in range(1,11) if x % 2 == 0]
print(evens)  # 输出：[2, 4, 6, 8, 10]

# 3. 结果分支：生成1-5的“奇数/偶数”标记列表
tags = ["奇数" if x % 2 == 1 else "偶数" for x in range(1,6)]
print(tags)  # 输出：["奇数", "偶数", "奇数", "偶数", "奇数"]
```

字典推导式

- 作用：从现有序列生成新字典，键和值均通过表达式定义。
- 两种语法形式：
  1. 基础形式：`{键表达式：值表达式 for 变量 in 原序列}`；
  2. 条件过滤：`{键表达式：值表达式 for 变量 in 原序列 if 条件}`。
- 示例：

```
# 1. 基础形式：生成“数字:平方”的字典
square_dict = {x: x*x for x in range(1,6)}
print(square_dict)  # 输出：{1:1, 2:4, 3:9, 4:16, 5:25}

# 2. 条件过滤：生成“偶数:平方”的字典
even_square_dict = {x: x*x for x in range(1,11) if x % 2 == 0}
print(even_square_dict)  # 输出：{2:4, 4:16, 6:36, 8:64, 10:100}
```

集合推导式

- 作用：从现有序列生成新集合（自动去重），语法与列表推导式类似，用 `{}` 包裹。
- 两种语法形式：
  1. 基础形式：`{表达式 for 变量 in 原序列}`；
  2. 条件过滤：`{表达式 for 变量 in 原序列 if 条件}`。
- 示例：

```
# 1. 基础形式：生成1-5的平方集合（自动去重，此处无重复）
square_set = {x*x for x in range(1,6)}
print(square_set)  # 输出：{1, 4, 9, 16, 25}
```

```
# 2. 条件过滤+去重：生成列表中大于3的元素集合（去重）
nums = [1,2,3,4,4,5]
filtered_set = {x for x in nums if x > 3}
print(filtered_set)  # 输出：{4, 5}
```

元组推导式（生成器表达式）

- 作用：从现有序列生成“生成器对象”（而非直接生成元组），语法与列表推导式类似，用 `()` 包裹；需通过 `tuple()` 转换为元组。
- 两种语法形式：
  1. 基础形式：`(表达式 for 变量 in 原序列)`（返回生成器）；
  2. 条件过滤：`(表达式 for 变量 in 原序列 if 条件)`（返回生成器）。
- 示例：

```
# 1. 基础形式：生成生成器，再转为元组
square_gen = (x*x for x in range(1,6))
square_tuple = tuple(square_gen)
print(square_tuple)  # 输出：(1, 4, 9, 16, 25)

# 2. 条件过滤：生成大于3的元素生成器，再转为元组
nums = [1,2,3,4,5]
filtered_gen = (x for x in nums if x > 3)
filtered_tuple = tuple(filtered_gen)
print(filtered_tuple)  # 输出：(4, 5)
```

- 说明：生成器是“惰性计算”的，仅在迭代时（如 `tuple()`、`for` 循环）才生成元素，节省内存。