Python

1 Python入门

1.1 Python基本语法

1. 严格区分大小写。
2. 每一行就是一条语句，每条语句以换行符结束。
3. 一条语句可以分多行编写，行尾用`\`表示换行。
4. Python 是缩进严格的语言，不能随便添加缩进。
5. 使用`#`表示行注释，`#`后面的内容都属于注释，注释的内容会被解释器忽略。

1.2 变量和标识符

Python 中使用变量不需要声明，直接为变量赋值即可。

不能使用未赋值的变量，否则会报错。变量必须先赋值后使用。

Python 是动态类型语言，可以为变量赋任意类型的值，也可以任意修改变量的值。

标识符命名规则：

1. 标识符中可以含有字母、数字、下划线。
2. 只能以字母或下划线开头，不能以数字开头。
3. 不能是 Python 中的关键字和保留字。
4. 不建议使用内置函数名作为标识符，因为这样会导致函数被覆盖，无法再被调用。

标识符命名规范：

1. 下划线命名法：所有字母小写，单词之间用下划线分割。
2. 驼峰命名法：每个单词首字母大写，其余字母小写。

1.3 数据类型

1.3.1 数值

Python 中的数值分为 3 类：整数、浮点数、复数。

1.3.1.1 整数

Python 中的整数为`int`类型。

Python 中整数的大小没有限制，整数可以无限大。如果数字的长度过大，可以使用下划线作为分隔符。

十进制整数不能以 0 开头，否则会报错。

二进制整数以`0b`开头，八进制整数以`0o`开头，十六进制整数以`0x`开头。

1.3.1.2 浮点数

Python 中浮点数为`float`类型。

对浮点数进行运算时，可能得到不精确的结果。

1.3.2 字符串

Python 中字符串为`str`类型。Python 不支持单字符类型，单字符在 Python 中也是作为字符串使用。

Python 中字符串要用引号括起来，引号可以是双引号、单引号、三双引号或三单引号。

引号必须成对使用，单双引号不能混用。

相同的引号之间不能嵌套，双引号内部不能直接嵌套双引号。如果需要引号嵌套，应交替使用单双引号。

单引号和双引号不能跨行使用。如果字符串跨行，需要在每一行末尾添加`\`，而且不保留换行格式。

三引号可以跨行使用，并且会保留字符串中的格式（如换行符等）。例如：

```
'''
锄禾日当午，
汗滴禾下土。
谁知盘中餐，
粒粒皆辛苦。
'''
```

转义字符可以用来表示特殊字符。常见的转义字符如下表所示。

转义字符	含义
`\'`	单引号
`\"`	双引号
`\t`	制表符
`\n`	换行符
`\\`	反斜杠
`\u****`	Unicode 编码

在字符串前面添加字母`r`或`R`，可以使该字符串成为原始字符串。在原始字符串中转义字符无效，`\`是一个普通字符，不具有转义功能。原始字符串常用于 Windows 目录路径。

在创建字符串时，可以在字符串中添加占位符，然后用其他值填充占位符。占位符用`%`引导，添加在字符串中。填充值也用`%`引导，跟在字符串后面。填充值多于一个时要加括号，多个填充值之间用逗号隔开，只有一个填充值时可以不加括号。

占位符	含义
`%m.ns`	字符串，最小长度为`m`，最大长度为`n`。长度不足时在左边补空格， 长度过大时截取前`n`个字符。
`%.nf`	浮点数，默认 6 位小数。`n`指定小数位数，小数位数不足时补零， 位数过大时四舍五入取`n`位小数。
`%d`	整数。如果填充值为浮点数，直接去掉小数部分，不进行四舍五入。

例如：

```
'hello %s' % 'world'  # hello world
'%s = %f' % ('a', 12.3)  # a = 12.300000
'x = %d' % 12.9  # x = 12
```

从 3.6 版本开始，可以在字符串前面添加`f`或`F`来创建格式化字符串，在格式化字符串中可以用大括号直接嵌入变量。例如：

```
a = 123
b = 'abc'
print(f'hello {a} and {b}')  # hello 123 and abc
```

1.3.3 布尔值

布尔值为`bool`类型，只有两种可能的取值：`True`表示真，`False`表示假。

布尔值属于整型，`True`相当于 1，`False`相当于 0。

1.3.4 空值

`None`为空值，表示不存在，其类型为`NoneType`。

1.3.5 类型检查

`type()`函数用于检查变量和值的类型，返回检查结果。例如：

```
type(1)  # <class 'int'>
type(1.2)  # <class 'float'>
type('abc')  # <class 'str'>
type(True)  # <class 'bool'>
type(None)  # <class 'NoneType'>
```

1.4 对象

1.4.1 对象的概念

Python 是面向对象的语言，在 Python 中一切皆对象。

对象就是内存中专门用来存储指定数据的一块区域。

数值、字符串、布尔值、`None`都是对象。

1.4.2 对象的结构

每个对象中都要保存 3 种数据：

1. id：用来标识对象。
   - 每个对象都有唯一的 id。
   - 可以通过`id()`函数查看对象的 id。
   - id 是由解析器生成的，在 CPython 中，id 就是对象的内存地址。
   - 对象一旦创建，它的 id 就永远不能改变。
2. type：标识当前对象所属的类型。
   - 类型决定了对象具有哪些功能。
   - Python 是强类型语言，对象一旦创建，其类型就永远不能改变。
   - 可以通过`type()`函数查看对象的类型。
3. value：对象中存储的具体的数据。
   - 对象分成两类：可变对象、不可变对象。可变对象的值可以改变，不可变对象的值不可改变。
   - 数值、字符串、布尔值、`None`都是不可变对象。

1.4.3 变量与对象

对象并没有直接存储到变量中。变量中存储的不是对象的值，而是对象的 id。使用变量时，实际上是在通过 id 查找对象。

对变量重新赋值时，改变的是变量中保存的对象 id，也就是使变量指向另一个对象。这种指向关系只有在为对象重新赋值时才会改变。

给一个变量重新赋值时，不会影响其他变量。

当修改可变对象的值时，不会改变变量所指向的对象。

如果有多个变量指向同一个对象，通过其中一个变量修改对象的值时，其他变量也会受到影响。

1.5 类型转换

类型转换就是将一个类型的对象转换成其他类型的对象。

类型转换不是改变对象本身的类型，而是根据当前对象的值创建新的对象。

类型转换函数：

1. `int()`：将其他对象转换成`int`型。
   - 布尔值：`True`转换成 1，`False`转换成 0。
   - 浮点数：取整，舍去小数部分。
   - 字符串：合法的整数字符串直接转换成对应的数字。如果不是合法的整数字符串，则抛出`ValueError`异常。
   - 对于其他不可转换为整型的对象，抛出`TypeError`异常。
2. `float()`：将其他对象转换成`float`型。
3. `str()`：将其他对象转换成字符串。
   - 布尔值：`True`转换成`'True'`，`False`转换成`'False'`。
   - 数值：转换成对应的字符串。
4. `bool()`：将其他对象转换成布尔值。
   - 表示空性的对象转换为`False`，其他对象转换为`True`。
   - 转换为`False`的情况：整数 0、空字符串`''`、`None`

1.6 运算符

1.6.1 算术运算符

1. `+`：加法运算符
   - 两个数值做加法运算，运算规则与数学运算相同。
   - 两个字符串做加法运算，相当于将两个字符串拼接成一个字符串。
2. `-`：减法运算符
   - 两个数值做减法运算，运算规则与数学运算相同。
   - 字符串不能做减法运算。
3. `*`：乘法运算符
   - 两个数值做乘法运算，运算规则与数学运算相同。
   - 整数与字符串做乘法运算，相当于将字符串重复若干次。
4. `/`：除法运算符
   - 两个数值做除法运算，返回值总是浮点类型。
   - 除数不能为 0，否则将抛出`ZeroDivisionError`。
5. `//`：取整除运算符
   - 两个数值做除法运算，将运算结果向下取整。

    - 如果操作数中有浮点数，返回值也是浮点数。
6. `**`：幂运算符。
7. `%`：取余运算符。

1.6.2 赋值运算符

1. `=`：将右侧的值赋给左侧的变量
2. `+=`
3. `-=`
4. `*=`
5. `/=`
6. `**=`
7. `//=`
8. `%=`

1.6.3 关系运算符

关系运算符用来比较两个对象之间的关系，返回一个布尔值。关系成立则返回`True`，否则返回`False`。

1. `>`
2. `>=`
3. `<`
4. `<=`
5. `==`
6. `!=`

如果两个操作数都是数值，则比较它们的大小关系。

如果两个操作数都是字符串，则逐位比较每个字符的 Unicode 编码值。如果第一位相同，则继续比较第二位；如果第二位也相同，则继续比较第三位；以此类推，直至某一位能够区分大小，则直接返回结果，不再继续比较。

数值和字符串不能直接比较，会抛出`TypeError`异常。

`==`和`!=`比较的是对象的值，对象的 id 和 type 不参与比较。

`is`关键字用于比较两个对象的 id，判断左右两侧是否为同一个对象。如果两个对象的 id 相同则返回`True`，不同则返回`False`。

`is not`与`is`相反，如果两个对象的 id 不同则返回`True`，相同则返回`False`。

Python 中关系运算符可以连用，先计算每个关系运算符及其左右操作数的结果，再将所有结果做与运算。例如：

```
print(1 < 2 < 3)  # True，相当于 1 < 2 and 2 < 3
print(1 < 2 > 3 < 1)  # False，相当于 1 < 2 and 2 > 3 and 3 < 1
```

1.6.4 逻辑运算符

1. `not`：逻辑非
2. `and`：逻辑与
3. `or`：逻辑或

如果操作数是布尔值，直接运算；如果操作数中有非布尔值，则先将其自动转换为布尔值，再进行运算。

对非布尔值进行`not`运算时，返回值为布尔值。

`and`和`or`运算具有短路特性：

- 在`and`运算中，如果左操作数为`False`，则直接返回左操作数，不再计算右操作数；如果左操作数为`True`，则直接返回右操作数。
- 在`or`运算中，如果左操作数为`True`，则直接返回左操作数，不再计算右操作数；如果左操作数为`False`，则直接返回右操作数。

1.6.5 条件运算符

语法：`语句1 if 条件表达式 else 语句2`

含义：对条件表达式的值进行判断，如果为`True`，则执行语句 1，并返回语句 1 的结果；如果为`False`，则执行语句 2，并返回语句 2 的结果。

1.7 控制台输入输出

`print()`函数用于向控制台输出，把要输出的内容作为参数。可以提供多个参数，同时输出多项内容，输出的多个内容之间会用空格分隔。默认情况下，每次输出以换行符结尾，可以用`end`参数指定结束符。

`input()`函数用于接收控制台输入，将输入的内容以字符串形式返回。可以提供一个字符串参数，作为提示信息。

2 流程控制语句

2.1 `if` 语句

语法：

```
if 条件表达式：语句

if 条件表达式:
    代码块

if 条件表达式:
    代码块
else:
    代码块

if 条件表达式:
    代码块
elif 条件表达式:
    代码块
elif 条件表达式:
    代码块
......
elif 条件表达式:
    代码块
else:
    代码块
```

代码块以缩进开始，直到代码恢复到之前的缩进级别时结束。代码块中可以包含多行代码，同一个代码块中多行代码缩进级别相同。

缩进有两种方式，一种是用 Tab，另一种是用 4 个空格。Python 官方文档中推荐使用空格。无论使用哪一种，代码中的缩进方式必须统一，不能混用。

代码块可以嵌套，每增加一个缩进级别，代码块就低一级。

2.2 循环语句

`while` 语句：

```
while 条件表达式:
    代码块

while 条件表达式:
    代码块
else:
    代码块  # 当条件为 False 时，执行一次
```

2.3 `break` 和 `continue`

`break` 只能在循环语句中使用，用于退出循环语句。如果 `while` 语句后面有 `else` ，当使用 `break` 退出循环时， `else` 部分的代码块也不会执行。

`continue` 用于跳过当次循环，直接进入下一次循环。

`break` 和 `continue` 都只对离它最近的循环起作用。

2.4 `pass`

`pass` 用于在分支或循环语句中占位。 `if` 语句和循环语句的执行体不能为空，当不能确定执行体的内容时，可以使用 `pass` 占位，执行时跳过该 `if` 语句或循环语句。

3 序列

序列（sequence）是 Python 中最基本的数据结构。

序列用于有序地保存一组对象，序列中的每个对象都有一个唯一的位置值，称为索引。索引从 0 开始，依次递增。序列中的对象会按照添加的顺序来分配索引。

序列的分类：

- 可变序列：序列中的元素可以改变
  - 列表（list）
- 不可变序列：序列中的元素不能改变
  - 字符串
  - 元组（tuple）

3.1 列表

列表（list）是 Python 中的一种对象，用于有序地保存多个数据。

3.1.1 列表的创建

使用 `[]` 创建列表。例如：

```
list1 = []  # 创建空列表
```

列表中存储的数据称为元素。一个列表中可以存储多个元素。元素的类型没有限制，可以将类型不同的元素保存在同一个列表中。

创建列表时可以指定列表中的元素，多个元素之间使用逗号分隔。例如：

```
list2 = [1, 'hello', True, None, [1, 2, 3], print]
```

可以使用`list()`将其他序列转换成列表，规则如下：

1. 数值不能转换成列表。
2. 字符串转换为列表时，会把字符串中的每个字符作为列表元素。
3. 元组转换为列表时，会把元组中的每个元素作为列表元素。
4. 字典转换为列表时，只保留字典中的键，不包括字典中的值。
5. 集合转换为列表时，结果是无序的。

3.1.2 列表元素的访问

列表中的元素按照插入顺序依次存储。索引用于标识元素在列表中的位置，列表中的每个元素都有一个索引。索引是从 0 开始的整数。

可以通过索引访问列表中的元素，语法为：`列表名[索引值]`。

索引可以是负数，此时将从后向前访问元素。例如，-1 表示倒数第一个元素，-2 表示倒数第二个元素，以此类推。

如果使用的索引值超过了列表的最大索引，则会抛出`IndexError`异常，表示索引越界。

索引操作对所有的序列都适用。

3.1.3 列表的切片

切片是指从列表中截取一个子列表。

语法：`列表名[起始索引:结束索引:步长]`
含义：从起始位置开始，到结束位置的前一个元素为止，每次对索引加上一个步长，将选取到的元素组成一个新的列表。

注意：

1. 切片时，包括起始位置的元素，而不包括结束位置的元素。
2. 切片操作总会返回一个新的列表，不会影响原来的列表。

起始索引和结束索引都可以省略，但二者之间的冒号不可省略。如果省略起始索引，则从第一个元素开始截取，相当于起始索引为 0；如果省略结束索引，则截取到最后一个元素为止，相当于结束

索引为列表长度；如果起始位置和结束位置都省略，则相当于将列表复制一遍。

起始索引和结束索引可以越界。

步长表示获取下一个元素的间隔，每次获取完一个元素之后，用当前索引加上步长，即可得到下一个元素的索引。

步长可以省略，此时步长取默认值 1。当省略步长时，步长之前的冒号可以省略，也可以不省略。

步长不能为 0，但可以为负数。当步长为负数时，表示从后向前选取元素。

例如：

```python
names = ['张三', '李四', '王五', '小明', '小红', '小刚']

# 从 0 号元素开始，到 2 号元素为止
print(names[0:3])  # ['张三', '李四', '王五']

# 从 1 号元素开始，到 3 号元素为止
print(names[1:4])  # ['李四', '王五', '小明']

# 从第一个元素开始，到 3 号元素为止
print(names[:4])  # ['张三', '李四', '王五', '小明']

# 从 1 号元素开始，到最后一个元素为止
print(names[1:])  # ['李四', '王五', '小明', '小红', '小刚']

# 从 1 号元素开始，到 5 号元素为止
print(names[1:6])  # ['李四', '王五', '小明', '小红', '小刚']

# 起始索引和结束索引都省略，相当于复制列表，新列表与原列表是不同的对象
print(names[:])  # ['张三', '李四', '王五', '小明', '小红', '小刚']
print(id(names) == id(names[:]))  # False

# 从倒数第四个元素开始，到倒数第一个元素的前一个元素（即倒数第二个元素）为止
print(names[-4:-1])  # ['王五', '小明', '小红']

# 从倒数第四个元素开始，到 1 号元素为止
print(names[-4:2])  # []

# 结束索引可以越界
print(names[1:8])  # ['李四', '王五', '小明', '小红', '小刚']

# 起始索引和结束索引都越界，不存在索引为 7 的元素，所以结果为空列表
print(names[7:8])  # []

# 从 2 号元素开始，到 4 号元素为止，步长为 1
print(names[2:5:1])  # ['王五', '小明', '小红']

# 从 2 号元素开始，到 4 号元素为止，省略步长但不省略步长前面的冒号
print(names[2:5:])  # ['王五', '小明', '小红']

# 从 1 号元素开始，到 4 号元素为止，步长为 2
print(names[1:5:2])  # ['李四', '小明']

# 从 4 号元素开始，到 1 号元素为止（不包括 1 号元素），步长为 -1
print(names[4:1:-1])  # ['小红', '小明', '王五']

# 从 4 号元素开始，到 1 号元素为止（不包括 1 号元素），步长为 -2
print(names[4:1:-2])  # ['小红', '王五']

# 起始索引和结束索引都省略，步长为 -1，相当于将列表倒序
print(names[::-1])  # ['小刚', '小红', '小明', '王五', '李四', '张三']
```

可以使用`slice()`来获取`slice`对象，它可以用于列表的切片。例如：

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']

s = slice(2)  # 相当于 [:2]
print(names[s])  # ['张三', '李四']

s = slice(1, 5)  # 相当于 [1:5]
print(names[s])  # ['李四', '王五', '小明', '小红']

s = slice(1, 5, 2)  # 相当于 [1:5:2]
print(names[s])  # ['李四', '小明']
```

切片操作对所有的序列都适用。

3.1.4 序列的通用操作

序列之间可以做`+`运算，将两个序列拼接为一个序列。例如：

```
print([1, 2] + [3, 4, 5])  # [1, 2, 3, 4, 5]
```

序列可以与整数做`*`运算，将序列重复指定的次数。例如：

```
print([1, 2] * 3)  # [1, 2, 1, 2, 1, 2]
```

`in`用于检查对象是否存在于序列中，如果存在则返回`True`，否则返回`False`。`not in`用于检查对象是否不在序列中，如果不在则返回`True`，否则返回`False`。

`len()`函数用于获取序列的长度，即序列中元素的个数。

`min()`函数获取序列中的最小值，`max()`函数获取序列中的最大值。

序列对象的`index()`方法获取指定元素在序列中第一次出现时的索引，参数为元素。例如：

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
print(names.index('张三'))  # 0
```

`index()`方法的第二个和第三个参数可选，第二个参数指定查找的起始位置，第三个参数指定查找的结束位置（查找时不包括结束位置）。例如：

```
names = ['张三', '李四', '张三', '小明', '小红', '张三']
print(names.index('张三'))  # 0
print(names.index('张三', 1))  # 2
print(names.index('张三', 0, 3))  # 0
print(names.index('张三', 3, 6))  # 5
```

调用`index()`方法时如果找不到元素，则会抛出`ValueError`异常。例如：

```
names = ['张三', '李四', '张三', '小明', '小红', '张三']
print(names.index('王五'))  # ValueError: '王五' is not in list
print(names.index('张三', 3, 5))  # ValueError: '张三' is not in list
```

序列对象的`count()`方法用于统计指定元素在序列中出现的次数。例如：

```
names = ['张三', '李四', '张三', '小明', '小红', '张三']
print(names.count('张三'))  # 3
print(names.count('李四'))  # 1
print(names.count('王五'))  # 0
```

3.1.5 列表的修改

可以直接通过索引来修改列表元素。例如：

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
names[0] = 'Jack'
print(names)  # ['Jack', '李四', '王五', '小明', '小红', '小刚']
```

可以通过切片来修改列表中的子列表。将列表的切片放在赋值号左侧，将另一个序列放在赋值号右侧，即可用指定的序列替换切片部分的子列表。新序列的长度可以不等于切片的长度。例如：

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
names[0:2] = ['Jack', '孙悟空']
print(names)  # ['Jack', '孙悟空', '王五', '小明', '小红', '小刚']
```

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
names[0:2] = ['Jack', '孙悟空', '猪八戒']
print(names)  # ['Jack', '孙悟空', '猪八戒', '王五', '小明', '小红', '小刚']
```

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
names[0:2] = ['Jack']
print(names)  # ['Jack', '王五', '小明', '小红', '小刚']
```

当切片的起始索引和结束索引相同时，切片为空，此时可以将指定的序列插入到列表的指定位置。例如：

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
names[0:0] = ['Jack']  # 将序列插入到 0 号位置
print(names)  # ['Jack', '张三', '李四', '王五', '小明', '小红', '小刚']
```

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
names[1:1] = ['Jack', '李明']  # 将序列插入到 1 号位置
print(names)  # ['张三', 'Jack', '李明', '李四', '王五', '小明', '小红', '小刚']
```

当切片的步长不为 1 时，新序列的长度必须和切片的长度相同，否则抛出`ValueError`。例如：

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
names[0:6:2] = ['Jack', 'Mike', 'ABC']
print(names)  # ['Jack', '李四', 'Mike', '小明', 'ABC', '小刚']
```

`del`关键字用于删除列表中的元素。例如：

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
del names[2]
print(names)  # ['张三', '李四', '小明', '小红', '小刚']
```

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
del names[0:2]
print(names)  # ['王五', '小明', '小红', '小刚']
```

```
names = ['张三', '李四', '王五', '小明', '小红', '小刚']
del names[::2]
print(names)  # ['李四', '小明', '小刚']
```

以上修改操作只适用于可变序列，不可变序列无法通过索引和切片来修改。

3.1.6 列表的方法

`append()`方法用于向列表的最后添加一个元素，参数为要添加的元素。例如：

```
names = ['张三', '李四', '王五']
names.append('小明')
print(names)  # ['张三', '李四', '王五', '小明']
```

`insert()`方法用于向列表的指定位置插入一个元素，有 2 个参数，第一个参数为插入位置，第二个参数为要插入的元素。例如：

```
names = ['张三', '李四', '王五']
names.insert(2, '小明')
print(names)  # ['张三', '李四', '小明', '王五']
```

`extend()`方法用于将指定的序列添加到当前列表尾部，参数为序列。例如：

```
names = ['张三', '李四', '王五']
names.extend(['小明', '小红'])
print(names)  # ['张三', '李四', '王五', '小明', '小红']
```

`clear()`方法用于清空列表。

`pop()`方法用于删除指定位置的元素，并将被删除的元素返回。参数为要删除元素的位置，如果参数为空，默认删除最后一个元素。例如：

```
names = ['张三', '李四', '王五']
element = names.pop(1)
print(names)  # ['张三', '王五']
print(element)  # 李四
```

`remove()`方法用于删除指定值的元素，没有返回值。如果相同值的元素有多个，只删除第一个。例如：

```
names = ['张三', '李四', '王五', '李四']
names.remove('李四')
print(names)  # ['张三', '王五', '李四']
```

`reverse()`方法用于翻转列表。例如：

```
names = ['张三', '李四', '王五', '小明']
names.reverse()
print(names)  # ['小明', '王五', '李四', '张三']
```

`sort()`方法用于对列表中的元素排序，默认为升序排列。例如：

```
nums = [20, 3, 5, -1, 98, 36]
nums.sort()
print(nums)  # [-1, 3, 5, 20, 36, 98]
```

如果希望`sort()`方法降序排序，可以将参数`reverse`设置为`True`。例如：

```
nums = [20, 3, 5, -1, 98, 36]
nums.sort(reverse=True)
print(nums)  # [98, 36, 20, 5, 3, -1]
```

`copy()`方法用于创建列表的副本。

3.1.7 列表的遍历

用`while`循环来遍历列表：

```
names = ['张三', '李四', '王五', '小明']

i = 0
while i < len(names):
    print(names[i])
    i += 1
```

`for`循环语法：

```
for 变量 in 序列:
    代码块
```

在`for`循环中，每次执行代码块时，都会将序列中的一个元素赋值给变量，所以可以通过变量来获取序列中的元素。

`else`、`break`和`continue`都可以在`for`循环中使用。

使用`for`循环遍历列表会更加方便，例如：

```
names = ['张三', '李四', '王五', '小明']

for name in names:
    print(name)
```

3.1.8 `range()`函数

`range()`函数可以用来生成整数序列，它有 3 个参数：

1. 起始位置（可选，默认为 0）
2. 结束位置
3. 步长（可选，默认为 1）

`range()`函数的返回值类型为`<class 'range'>`，可以使用`list()`将其转换成列表。例如：

```
r1 = range(5)
print(r1)  # range(0, 5)
print(list(r1))  # [0, 1, 2, 3, 4]

r2 = range(3, 6)
print(list(r2))  # [3, 4, 5]

r3 = range(0, 10, 2)
print(list(r3))  # [0, 2, 4, 6, 8]

r4 = range(10, 0, -1)
print(list(r4))  # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

`for`循环可以遍历`range()`函数得到的序列。可以用`range()`函数创建执行指定次数的`for`循环，例如：

```
# 创建一个执行 10 次的 for 循环
for i in range(10):
    print(i)
```

3.2 元组

元组（tuple）是不可变序列。除了修改操作不能使用外，元组的操作方式与列表基本相同。

使用`()`来创建元组。例如：

```
tuple1 = ()  # 创建空元组
tuple2 = (1, 2, 3)
```

如果元组不为空，在创建元组时可以省略括号。例如：

```
tuple3 = 1, 2, 3
print(tuple3)  # (1, 2, 3)
print(type(tuple3))  # <class 'tuple'>
```

如果在创建元组时省略了括号，则至少要有一个逗号。如果元组中只有一个元素，创建时要在该元素后面加逗号。例如：

```
tuple4 = 1,
print(tuple4)  # (1,)
print(type(tuple4))  # <class 'tuple'>
```

`tuple()`将其他序列转换成元组，转换规则与`list()`相同。

可以用索引获取元组中的元素，但不能对元组中的元素重新赋值。

`sum()`函数可以对序列中的所有元素求和。

元组的解包（解构）：将元组中的每个元素都赋值给一个变量。例如：

```
my_tuple = (1, 2, 3, 4)

a, b, c, d = my_tuple
print(f'a={a}')  # a=1
print(f'b={b}')  # b=2
print(f'c={c}')  # c=3
print(f'd={d}')  # d=4
```

元组的解包可以用于交换两个变量的值，例如：

```
a = 1
b = 2

a, b = b, a  # 交换
print(f'a={a}')  # a=2
print(f'b={b}')  # b=1
```

元组解包时，变量的个数必须和元组中元素的个数一致，否则抛出`ValueError`。也可以在变量前面添加一个`*`，这样会将元组中所有剩余的元素以列表的形式赋值给该变量。例如：

```
my_tuple = (1, 2, 3, 4)

a, b, *c = my_tuple  # a 和 b 为前两个元素，c 为剩下的元素
print(f'a={a}')  # a=1
print(f'b={b}')  # b=2
print(f'c={c}')  # c=[3, 4]

a, *b, c = my_tuple  # a 为第一个元素，c 为最后一个元素，b 为剩下的元素
print(f'a={a}')  # a=1
print(f'b={b}')  # b=[2, 3]
print(f'c={c}')  # c=4

*a, b, c = my_tuple  # b 和 c 为最后两个元素，a 为剩下的元素
print(f'a={a}')  # a=[1, 2]
print(f'b={b}')  # b=3
print(f'c={c}')  # c=4
```

不能同时出现两个及以上的`*`变量，否则抛出`SyntaxError`。

解包适用于所有序列。

3.3 字符串

字符串也是序列，支持索引、切片等序列操作。字符串中的每个字符都是一个元素。

由于 Python 不支持字符类型，对字符串做索引操作将产生一个长度为 1 的字符串。

字符串是不可变序列，不能用索引和切片对字符串赋值。

字符串之间可以相加，相当于拼接字符串。字符串只能和字符串相加，不能和其他类型值相加。例如：

```
'abc' + '123'  # abc123
```

字符串可以和整数做乘法，如果是正整数，相当于将字符串重复若干次后得到新的字符串。例如：

```
'abc' * 3  # abcabcabc
```

如果整数是 0 或负整数，与字符串相乘后得到的是空字符串。

`join()`方法以当前字符串为分隔符，将若干字符串拼接在一起。参数为一个可迭代对象（列表或元组），其中的元素只能是字符串，`join()`方法将这些字符串拼接在一起，中间用当前字符串分隔。例如：

```
t = ('你好', '我叫不紧张', '今年18岁')
s = ','.join(t)
print(s)  # 你好,我叫不紧张,今年18岁
```

`find()`方法用于查找子串，如果找到则返回子串第一次出现的位置，找不到则返回 -1。它有 3 个参数：

1. 待查找的子串。
2. 查找的起始位置。可选，默认值为 0。
3. 查找的结束位置（不包含）。可选，默认值为 -1。

`count()`方法返回子串在当前字符串中出现的次数，参数与`find()`方法相同。

`replace()`方法用于替换指定的子串，参数为：

1. 待替换的子串。
2. 用于替换的字符串。
3. 替换的次数，从左到右进行指定次数的替换。可选，默认替换所有的。

`upper()`方法将小写字母转换为大写，`lower()`方法将大写字母转换为小写。

`split()`方法用指定的子串分割字符串，返回一个列表，包含分割成的各个片段。参数为：

1. 用于分割的子串。
2. 分割的次数，从左到右进行指定次数的分割。可选，默认找到所有的分割点进行分割。

```
s = '我叫不紧张,今年18岁,性别男,喜欢唱跳rap'
print(s.split(','))  # ['我叫不紧张', '今年18岁', '性别男', '喜欢唱跳rap']
print(s.split(',', 2))  # ['我叫不紧张', '今年18岁', '性别男,喜欢唱跳rap']
```

`strip()`方法用于去除前导和末尾字符。参数为要去除的字符组成的字符串，如果省略或为`None`，默认去除空白符。对于前导字符，从前到后查看每个字符，如果该字符包含于参数字符串，则去除该字符，并继续向后查看；如果该字符不包含于参数字符串，则停止去除过程。末尾字符的操作方式相同。例如：

```
s1 = '\t\t\t+abcd      '
print(s1.strip())  # +abcd

s2 = '0000000000abcd1230000000000000000'
print(s2.strip('0'))  # abcd123

s3 = '1121222abcd121a212'
print(s3.strip('12'))  # abcd121a
```

`format()`方法用于格式化字符串。在字符串中使用`{}`占位，然后对这个字符串调用`format()`方法，将参数按顺序传递给占位符，返回格式化之后的字符串。例如：

```
s = '我叫{}，今年{}岁，性别{}'
print(s.format('不紧张', 18, '男'))  # 我叫不紧张，今年18岁，性别男
```

`format()`方法的位置参数会被包装到一个元组中，然后将元组中的元素填充到占位符的位置。可以在占位符中指定索引，向该位置填充相应索引的参数。例如：

```
s = '我叫{2}，今年{1}岁，性别{0}'
print(s.format('不紧张', 18, '男'))  # 我叫男，今年18岁，性别不紧张

s = '我叫{1}，今年{1}岁，性别{1}'
print(s.format('不紧张', 18, '男'))  # 我叫18，今年18岁，性别18
```

要么都指定索引，要么都不指定索引，不能一部分指定而一部分不指定，否则会抛出`ValueError`异常。如果不指定索引，默认按顺序依次将参数填充到占位符处，此时参数的个数不能少于占位符的个数，否则抛出`IndexError`异常。

可以在占位符中指定关键字参数的名称，并传递相应的关键字参数。例如：

```
s = '我叫{name}，今年{age}岁，性别{gender}'
print(s.format(name='不紧张', age=18, gender='男'))  # 我叫不紧张，今年18岁，性别男
```

位置参数和关键字参数可以混用。例如：

```
s = '我叫{name}，今年{1}岁，性别{0}'
print(s.format('男', 18, name='不紧张'))  # 我叫不紧张，今年18岁，性别男
```

可以在索引或关键字参数名之后指定参数的格式，以`:`开头，后面跟具体的格式规格。例如，`.mf`表示定点小数，小数点后保留 m 位；`.m%`表示百分数，小数点后保留 m 位；等等。

4 字典

字典（dictionary）属于一种新的数据结构，称为映射（mapping）。

字典中的每个元素都有一个唯一的名字，通过这个名字可以快速查找指定的元素。这个唯一的名字称为**键**（key），对应的元素称为**值**（value）。因此字典也称为键值对（key-value）结构。

字典中可以有多个键值对，每个键值对称为一项（item）。

字典中的值可以是任意对象，键可以是任意的不可变对象（如`int`、`str`、`bool`、`tuple`等）。

4.1 字典的创建

使用`{}`创建字典。例如：

```
d = {}  # 创建空字典
print(d)  # {}
print(type(d))  # <class 'dict'>
```

字典中的每个键值对用`key:value`表示，多个键值对之间用逗号隔开。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}
```

字典中的键不能重复，如果有重复，后面的值会替换前面的值。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男', 'name': 'Jack'}
print(d)  # {'name': 'Jack', 'age': 18, 'gender': '男'}
```

可以使用`dict()`来创建字典，每个关键字参数都是一个键值对，参数名为键，参数值为值。用这种方式创建的字典，键都是字符串。例如：

```
d = dict(name='小明', age=18, gender='男')
print(d)  # {'name': '小明', 'age': 18, 'gender': '男'}
print(type(d))  # <class 'dict'>
```

`dict()`也可以将一个包含双值子序列的序列转换为字典。序列中的每个元素都是双值子序列，每个双值子序列转换为一个键值对，其中第一个元素为键，第二个元素为值。例如：

```
d = dict([('name', '小明'), ('age', 18)])
print(d)  # {'name': '小明', 'age': 18}
```

4.2 字典的使用

可以根据键获取值，语法为：`字典名[键]`。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}
print(d['name'])  # 小明
print(d['age'])  # 18
print(d['gender'])  # 男
```

如果使用了字典中不存在的键，会抛出`KeyError`。

`len()`函数获取字典中键值对的个数。

`in`检查字典中是否包含指定的键，`not in`检查字典中是否不包含指定的键。

字典对象的`get()`方法也可以根据键来获取值，如果使用的键在字典中不存在，则返回`None`。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}
print(d.get('name'))  # 小明
print(d.get('abc'))  # None
```

`get()`方法有一个可选参数，用于指定键不存在时的返回值。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}
print(d.get('abc', '不存在'))  # 不存在
```

可以直接对`字典名[键]`赋值来修改键值对，如果键存在则修改其值，如果键不存在则添加键值对。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}

# 修改
d['name'] = 'Jack'
print(d)  # {'name': 'Jack', 'age': 18, 'gender': '男'}

# 添加
d['id'] = '123'
print(d)  # {'name': 'Jack', 'age': 18, 'gender': '男', 'id': '123'}
```

字典对象的`setdefault()`方法用于向字典中添加键值对。如果键存在，则返回其值，并且不改变其值；如果键不存在，则向字典中添加这个键，并设置其值，返回设置的值。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}

result = d.setdefault('name', 'Jack')
print(d)  # {'name': '小明', 'age': 18, 'gender': '男'}
print(result)  # 小明

result = d.setdefault('id', '123')
print(d)  # {'name': '小明', 'age': 18, 'gender': '男', 'id': '123'}
print(result)  # 123
```

字典对象的`update()`方法用于将其他字典中的键值对添加到当前字典中，如果有重复的键，则用新的值替换当前值。例如：

```
d = {'a': 1, 'b': 2}
d2 = {'b': 3, 'c': 4, 'd': 5}

d.update(d2)
print(d)  # {'a': 1, 'b': 3, 'c': 4, 'd': 5}
```

可以使用`del`删除字典中的键值对。如果使用的键不存在，则抛出`KeyError`。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}

del d['age']
print(d)  # {'name': '小明', 'gender': '男'}
```

字典对象的`popitem()`方法可以随机删除字典中的一个键值对，一般都会删除最后一个。返回一个元组，元组中有 2 个元素，第一个元素为被删除的键，第二个元素为被删除的值。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}

result = d.popitem()
print(d)  # {'name': '小明', 'age': 18}
print(result)  # ('gender', '男')
```

如果字典为空，调用`popitem()`方法会抛出`KeyError`。

字典对象的`pop()`方法删除字典中指定键的项，参数为键，返回被删除的值。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}

result = d.pop('age')
print(d)  # {'name': '小明', 'gender': '男'}
print(result)  # 18
```

如果指定的键不存在，则`pop()`方法会抛出`KeyError`。`pop()`方法的第二个参数可选，用于指定键不存在时的返回值。如果指定了第二个参数，当键不存在时，不会抛出异常，而是返回指定的值。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}

result = d.pop('id', '不存在')
print(d)  # {'name': '小明', 'age': 18, 'gender': '男'}
print(result)  # 不存在
```

字典对象的`clear()`方法用于清空字典。

字典对象的`copy()`方法用于对字典进行浅复制，复制后的对象和原对象是独立的。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}
d2 = d.copy()

print(d2)  # {'name': '小明', 'age': 18, 'gender': '男'}
print(id(d) == id(d2))  # False
```

“浅复制”意味着只复制对象中的值，不会复制对象本身。例如：

```
d = {'a': {'name': '小明'}, 'b': 1, 'c': 2}
d2 = d.copy()

print(id(d) == id(d2))  # False, d 和 d2 是不同的对象
print(id(d['a']) == id(d2['a']))  # True, d['a'] 和 d2['a'] 指向同一个对象
```

`fromkeys()`方法用于创建一个新的字典，参数为：

1. 一个序列（列表、元组、集合），作为字典的键。
2. 可以是任何数据类型，作为字典的值，为所有键设置相同的值。可选，默认值为`None`。

```
d = {}

d1 = d.fromkeys(['name', 'age'])
print(d1)  # {'name': None, 'age': None}

d2 = d.fromkeys(['name', 'age'], 20)
print(d2)  # {'name': 20, 'age': 20}

d3 = d.fromkeys(['name', 'age'], ['小明', 18])
print(d3)  # {'name': ['小明', 18], 'age': ['小明', 18]}
```

4.3 字典的遍历

字典对象的`keys()`方法返回一个序列，包含字典中所有的键。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}

print(d.keys())  # dict_keys(['name', 'age', 'gender'])

# 通过遍历这个序列可以获取所有的键，进而获取所有的值
for key in d.keys():
    print(key, d[key])
```

字典对象的`values()`方法返回一个序列，包含字典中所有的值。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}
print(d.values())  # dict_values(['小明', 18, '男'])
```

字典对象的`items()`方法返回一个序列，序列中包含双值子序列，第一个元素是键，第二个元素是值。例如：

```
d = {'name': '小明', 'age': 18, 'gender': '男'}

print(d.items())  # dict_items([('name', '小明'), ('age', 18), ('gender', '男')])

# 解包，分别获取键和值
for k, v in d.items():
    print(k, v)
```

5 集合

集合与列表的不同：

1. 集合中只能存储不可变对象。
2. 集合中存储的对象是无序的（不按照元素的插入顺序保存）。
3. 集合中不能出现重复的元素。

5.1 集合的创建

使用`{}`创建集合。例如：

```
s = {1, 3, 2, 4}
print(s)  # {1, 2, 3, 4}
print(type(s))  # <class 'set'>
```

使用空的大括号`{}`创建的是空字典，而不是空集合。要创建空集合，需要使用`set()`，参数为空。例如：

```
s = set()
print(s)  # set()
print(type(s))  # <class 'set'>
```

`set()`可以将序列或字典转换成集合，转换规则与`list()`相似，但是结果会去重并且无序。例如：

```
s1 = set([1, 3, 5, 2, 1])
print(s1)  # {1, 2, 3, 5}

s2 = set('hello')
print(s2)  # {'h', 'e', 'o', 'l'}

s3 = set({'a': 1, 'b': 2, 'c': 3})
print(s3)  # {'a', 'c', 'b'}
```

5.2 集合的操作

集合不能使用索引操作。

可以用`in`判断对象是否在集合中，用`not in`判断对象是否不在集合中。

`len()`函数获取集合中的元素数量。

集合对象的`add()`方法可以向集合中添加元素，没有返回值。

集合对象的`update()`方法可以将其他集合中的元素添加到当前集合中，参数可以是集合、序列或字典。

集合对象的`pop()`方法随机删除集合中的一个元素，返回被删除的元素。

集合对象的`remove()`方法用于删除集合中的指定元素，没有返回值。

集合对象的`clear()`方法用于清空集合。

集合对象的`copy()`方法对集合进行浅复制。

5.3 集合的运算

1. `&`：交运算
2. `|`：并运算
3. `-`：差运算

4. `^`：异或运算，获取只在一个集合中出现的元素
5. `<=`：检查一个集合是否为另一个集合的子集
6. `<`：检查一个集合是否为另一个集合的真子集
7. `>=`：检查一个集合是否为另一个集合的超集
8. `>`：检查一个集合是否为另一个集合的真超集

例如：

```
s1 = {1, 2, 3, 4, 5}
s2 = {3, 4, 5, 6, 7}
s3 = {1, 2, 3}

print(s1 & s2)  # {3, 4, 5}
print(s1 | s2)  # {1, 2, 3, 4, 5, 6, 7}
print(s1 - s2)  # {1, 2}
print(s1 ^ s2)  # {1, 2, 6, 7}
print(s3 <= s1)  # True
print(s3 <= s2)  # False
print(s3 < {1, 2, 3})  # False
print(s3 < s1)  # True
```

6 函数

6.1 函数简介

函数也是对象，可以用来保存一些可执行代码，并且可以在需要时对这些代码进行调用。

函数的创建：

```
def 函数名([形参1, 形参2, ... , 形参n]):
    代码块
```

函数名必须符合标识符的命名规则。

函数中保存的代码不会立即执行，只有在调用函数时才会执行。

函数的调用：`函数名(实参1, 实参2, ... , 实参n)`

6.2 函数的参数

6.2.1 形参与实参

在定义函数时，可以在括号中定义形参。可以没有形参，也可以有一个或多个形参，多个形参之间用逗号隔开。

定义形参就相当于在函数内部声明了变量，但并不赋值。

如果函数定义时指定了形参，则调用函数时必须传递实参，实参将会赋值给对应的形参。实参与形参一一对应，实参的数量要与形参数量相同。

实参可以是任意类型的对象。

传递实参相当于为形参赋值。如果实参是一个变量，则传参后，实参变量与形参变量指向同一个对象。在函数中对形参重新赋值，不会影响其他变量。如果形参指向一个可变对象，通过形参修改对象时，会影响到所有指向该对象的变量。

6.2.2 形参的默认值

定义形参时，可以为形参指定默认值。如果在调用时传递了实参，则默认值没有作用；如果没有传递实参，默认值就会生效。例如：

```
def fn(a, b, c=20):
    print(a, b, c)


fn(1, 2, 3)  # 1 2 3
fn(1, 2)  # 1 2 20
```

6.2.3 参数的传递方式

参数的传递方式：

1. 位置参数：实参与形参按位置一一对应，第一个实参对应第一个形参，第二个实参对应第二个形参，以此类推。
2. 关键字参数：传参时指定参数名，将实参传递给指定的形参。此时可以不按照形参定义的顺序传参。

例如：

```
def fn(a, b, c):
    print(a, b, c)


# 位置参数
fn(1, 2, 3)  # 1 2 3

# 关键字参数
fn(b=1, c=2, a=3)  # 3 1 2
```

位置参数和关键字参数可以混用，此时必须将位置参数写在前面。如果一个形参已经通过位置参数赋值，就不能再用关键字参数再次赋值。例如：

```
def fn(a, b=10, c=20):
    print(a, b, c)


fn(1, c=2, b=3)  # 1 3 2
fn(a=1, 2, c=3)  # SyntaxError: positional argument follows keyword argument
fn(1, a=2)  # TypeError: fn() got multiple values for argument 'a'
```

6.2.4 不定长参数

在定义函数时，可以在形参前面加`*`，该形参将会把所有实参保存到一个元组中。例如：

```
def fn(*a):
    print(a, type(a))


fn()  # () <class 'tuple'>
fn(1, 2, 3)  # (1, 2, 3) <class 'tuple'>
```

带`*`的形参只能有一个。

带`*`的形参只能接收位置参数，不能接收关键字参数。

带`*`的形参可以和普通形参配合使用。例如：

```
def fn(a, b, *c):
    print(a, b, c)


fn(1, 2, 3, 4, 5)  # 1 2 (3, 4, 5)
```

带`*`的形参可以写在任意位置，但是对于带`*`的形参之后的所有形参，必须使用关键字参数来传参。例如：

```
def fn(a, *b, c):
    print(a, b, c)


fn(1, 2, 3, 4, c=5)  # 1 (2, 3, 4) 5
```

可以直接使用`*`作为参数，`*`之后的参数都必须用关键字传入。例如：

```
def fn(a, b, *, c):
    print(a, b, c)

fn(1, 2, 3)  # TypeError
fn(1, 2, c=3)  # 1 2 3
```

关键字参数的参数名必须是已定义的形参名，否则抛出`TypeError`。在形参前面加`**`，可以让该形参接收未定义的关键字参数，这些参数被保存在一个字典中，每个参数是一个键值对，参数名为键，参数值为值。例如：

```
def fn(**a):
    print(a, type(a))


fn(a=0, b=1, c=2, d=3)  # {'a': 0, 'b': 1, 'c': 2, 'd': 3} <class 'dict'>
```

带`**`的形参只能有一个，并且必须写在所有形参的最后。

带`**`的形参只接收未定义的关键字参数，如果参数名有定义，则会传递给对应的参数。例如：

```
def fn(b, **c):
    print(b, c)


fn(a=0, b=1, c=2, d=3)  # 1 {'a': 0, 'c': 2, 'd': 3}
```

6.2.5 参数的解包

在传递实参时，可以在序列类型的参数前面添加`*`，将序列中的元素作为位置参数依次传递。例如：

```
def fn(a, b, c, d):
    print(a, b, c, d)


t = (2, 3)
fn(1, *t, 4)  # 1 2 3 4
```

也可以在字典类型的实参前面添加`**`，将字典中的键值对作为关键字参数，字典中的键必须和形参名对应。例如：

```
def fn(a, b, c):
    print(a, b, c)


d = {'a': 1, 'b': 2, 'c': 3}
fn(**d)  # 1 2 3
```

带`**`的实参必须写在位置参数之后。

6.3 函数的返回值

可以通过`return`关键字来指定函数的返回值，将要返回的内容跟在`return`后面即可。

函数的返回值可以是任意类型的对象。

如果没有`return`，或者`return`后面什么都不写，则返回值为`None`。

`return`一旦执行，函数直接结束，不再执行函数中的其他代码。

6.4 文档字符串

`help()`是 Python 的内置函数，用来查询 Python 中函数的用法，参数为函数对象。例如：

```
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

在定义函数时，可以在函数内部编写文档字符串，作为函数的说明。编写文档字符串后，就可以使用`help()`函数来查看函数的说明。

在函数的第一行写一个字符串，这就是文档字符串。例如：

```
def fn(a, b, c):
    '''
    文档字符串示例

    参数:
        a: ...
        b: ...
        c: ...
    返回值: ...
    '''
    return


help(fn)
```

```
fn(a, b, c)
    文档字符串示例

    参数:
        a: ...
        b: ...
        c: ...
    返回值: ...
```

在形参列表中，可以在形参后面用冒号说明该参数期望传参的类型。这只是说明性的标记，只会在`help()`函数的输出内容中显示，不会对传入的实参产生限制。例如：

```
def fn(a: int, b: str, c: bool):
    '''
    文档字符串
    '''
    return


help(fn)
```

```
fn(a: int, b: str, c: bool)
    文档字符串
```

在函数定义中，可以在形参的括号后面用`->`说明函数的返回类型。例如：

```python
def fn(a: int, b: str, c: bool) -> int:
    '''
    文档字符串
    '''
    return 10


help(fn)
```

```
fn(a: int, b: str, c: bool) -> int
    文档字符串
```

6.5 作用域

作用域（scope）是指变量生效的区域。Python 中有 2 种作用域：

1. 全局作用域
    - 全局作用域在程序运行时创建，在程序结束时销毁。
    - 所有函数以外的区域都是全局作用域。
    - 在全局作用域中定义的变量都属于全局变量，全局变量可以在程序的任意位置访问。
2. 函数作用域
    - 函数作用域在函数调用时创建，在调用结束时销毁。
    - 函数每调用一次就会产生一个新的函数作用域。
    - 在函数作用域中定义的变量都是局部变量，只能在函数内部访问。

访问变量时，会优先在当前作用域中寻找，如果找到则直接使用，如果找不到则到上一级作用域中寻找。如果直到全局作用域都找不到，则抛出`NameError`。

在函数中为变量赋值时，默认都是为局部变量赋值。如果希望在函数内部修改全局变量，需要使用`global`关键字来声明变量。例如：

```python
a = 10

def fn():
    global a  # 声明全局变量
    a = 20
    print(f'函数内部: a={a}')


print(f'函数外部: a={a}')  # 函数外部: a=10
fn()                       # 函数内部: a=20
print(f'函数外部: a={a}')  # 函数外部: a=20
```

6.6 命名空间

命名空间（namespace）是变量存储的位置，任何变量都要存储到命名空间中。

每一个作用域都有对应的命名空间。全局命名空间用于保存全局变量，函数命名空间用于保存函数中的局部变量。

`locals()`函数用来获取当前作用域的命名空间，在全局作用域中调用`locals()`则获取全局命名空间，在函数作用域中调用`locals()`则获取函数命名空间，返回值是一个字典。例如：

```
scope = locals()
print(type(scope))  # <class 'dict'>

def fn():
    a = 10
    print(locals())


fn()  # {'a': 10}
```

由于命名空间是字典，可以通过字典操作从命名空间中获取变量。例如：

```
a = 10
scope = locals()
print(scope['a'])  # 10
```

向命名空间字典中添加键值对，就相当于创建了一个变量。（不建议这样做）例如：

```
scope = locals()
scope['a'] = 1
print(a)  # 1
```

`globals()`函数可以在任意位置获取全局命名空间。例如：

```
scope = locals()

def fn():
    global_scope = globals()
    print(scope == global_scope)


fn()  # True
```

6.7 高阶函数

在 Python 中，函数是一等对象。一等对象具有以下特点：

1. 对象是在运行时创建的。
2. 能赋值给变量或作为数据结构中的元素。
3. 能作为参数传递。
4. 能作为返回值返回。

Python 中任何对象都是一等对象。

Python 支持函数式编程，但 Python 不是函数式编程语言。

作为函数式编程的重要特性，Python 支持高阶函数。高阶函数至少要符合以下两个特点中的一个：

1. 接收一个或多个函数作为参数。
2. 将函数作为返回值返回。

`filter()` 函数可以从可迭代对象中过滤出符合条件的元素，并保存到新的可迭代对象中。它有 2 个参数：

1. 函数，返回值为 `bool` 类型。此函数会作用到序列中的每个元素，只有返回值为 `True` 的元素会被选择。
2. 待过滤的可迭代对象。

`filter()` 函数需要一个函数作为参数，它就是一个高阶函数。根据传入函数的不同，可以得到不同的筛选策略，具有很强的灵活性。例如：

```
def is_odd(n):
    '''
    判断给定的整数是否为奇数
    '''
    if n % 2 != 0:
        return True
    return False


my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# 选择列表中所有的奇数
result = filter(is_odd, my_list)
print(result)  # <filter object at 0x0000024E060FB4F0>
print(list(result))  # [1, 3, 5, 7, 9]
```

列表对象的 `sort()` 方法也是一个高阶函数。默认情况下，`sort()` 方法直接使用 `<` 比较列表中的元素。`sort()` 方法可以接收一个关键字参数 `key`，参数类型为函数，该函数将会作用到列表中的所有元素，用该函数的返回值进行比较，以此对元素进行排序。例如：

```python
my_list = ['aa', 'c', 'bbb', 'ddddddd']

# 按字符串长度排序
my_list.sort(key=len)
print(my_list)  # ['c', 'aa', 'bbb', 'ddddddd']
```

```python
my_list = [1, 3, '12', '5']

# 将元素转换成整数，再比较
my_list.sort(key=int)
print(my_list)  # [1, 3, '5', '12']

# 将元素转换成字符串，再比较
my_list.sort(key=str)
print(my_list)  # [1, '12', 3, '5']
```

`sorted()` 函数和 `sort()` 方法类似，但是 `sorted()` 函数可以对任意序列进行排序，并且不会影响原序列，而是返回一个新的序列。例如：

```python
my_list = [1, 3, '12', '5']

result = sorted(my_list, key=int)
print(result)  # [1, 3, '5', '12']
print(my_list)  # [1, 3, '12', '5']
```

6.8 匿名函数

匿名函数又称为 lambda 表达式，用来创建一些简单的函数。

语法：`lambda 参数列表：返回值`

例如：

```python
# 定义一个匿名函数，用于计算两个数的和
print(lambda a, b: a + b)  # <function <lambda> at 0x000002017D127280>

# 可以将匿名函数赋值给变量，但一般不会这样做
fn = lambda a, b: a + b

# 通过变量调用该函数
result = fn(10, 20)
print(result)  # 30

# 直接调用匿名函数
result = (lambda a, b: a + b)(10, 20)
print(result)  # 30
```

匿名函数可以用于向高阶函数传递参数，例如：

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

# 选择所有偶数
result = filter(lambda n: n % 2 == 0, my_list)
print(list(result))  # [2, 4, 6, 8, 10]

# 选择所有奇数
result = filter(lambda n: n % 2 != 0, my_list)
print(list(result))  # [1, 3, 5, 7, 9]
```

`map()`函数可以对可迭代对象中的所有元素做指定的操作，将操作结果保存到一个新的可迭代对象中返回。例如：

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

# 将所有元素加 1
result = map(lambda e: e + 1, my_list)
print(list(result))  # [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

6.9 闭包

在外层函数中定义一个内层函数，在内层函数中访问外层函数中的局部变量，外层函数将内层函数返回，此时外层函数就是一个闭包。例如：

```
def outer():
    a = 10

    # 定义内层函数
    def inner():
        print(a)  # 在内层函数中访问外层函数中的局部变量

    return inner  # 外层函数将内层函数返回


fn = outer()
print(fn)  # <function outer.<locals>.inner at 0x000001C1CA7303A0>
fn()  # 10
```

正常情况下，在外部作用域中无法访问内部作用域中的变量。但是内层函数可以访问外层函数中的变量，只要将内层函数返回，就可以在外层函数的外部使用内层函数来访问外层函数中的局部变量。

可以将私有数据藏在闭包中，这些数据只能通过返回的内层函数来访问，其他方式访问不到，可以保证数据安全。

6.10 装饰器

装饰器可以在不修改函数的情况下对函数进行扩展。

装饰器是一种高阶函数，参数为要扩展的函数，在装饰器内部创建一个新的函数实现扩展功能，并返回这个新的函数。例如：

```
def begin_end(fn):
    '''
    装饰器，在函数调用前打印 "开始执行"，函数调用后打印 "执行结束"
    '''

    def new_fn(*args, **kwargs):
        '''
        装饰后的新函数
        args: 接收所有位置参数
        kwargs: 接收所有关键字参数
        '''
        print("开始执行")
        result = fn(*args, **kwargs)
        print("执行结束")
        return result

    return new_fn  # 返回装饰后的新函数


def add(a, b):
    return a + b


f = begin_end(add)  # 对 add() 函数进行装饰，返回装饰后的函数
r = f(111, 222)  # 调用装饰后的函数
print(r)
```

```
开始执行
执行结束
333
```

在定义函数时，可以使用`@`指定装饰器，用指定的装饰器来扩展函数。可以为函数指定多个装饰器，此时将按照从内到外的顺序进行装饰。

```
def begin_end(fn):
    '''
    装饰器，在函数调用前打印 "开始执行"，函数调用后打印 "执行结束"
    '''

    def new_fn(*args, **kwargs):
        print("开始执行")
        result = fn(*args, **kwargs)
        print("执行结束")
        return result

    return new_fn


def decorator(fn):
    '''
    装饰器，在函数调用前后打印分割线
    '''

    def new_fn(*args, **kwargs):
        print('------------------')
        result = fn(*args, **kwargs)
        print('------------------')
        return result

    return new_fn


@decorator
@begin_end
def say_hello():
    print('hello')


say_hello()
```

```
------------------
开始执行
hello
执行结束
------------------
```

7 面向对象

Python 是一种面向对象的编程语言。

7.1 类

使用`class`关键字定义类，语法：

```
class 类名:
    代码块
```

类名使用大驼峰命名法。

创建类对象：`类名()`。例如：

```
class MyClass:
    pass


print(MyClass)  # <class '__main__.MyClass'>

obj = MyClass()  # 创建 MyClass 类的对象
print(obj)  # <__main__.MyClass object at 0x0000016BB5C43D30>
print(type(obj))  # <class '__main__.MyClass'>
```

`int()`、`float()`、`bool()`、`str()`、`list()`实际上不是函数，而是在创建相应类的实例。例如：

```
a = int(10)  # 创建 int 类的实例
b = str('hello')  # 创建 str 类的实例
```

`isinstance()`函数用来检查一个对象是否是指定类的实例。例如：

```
a = int(10)
print(isinstance(a, int))  # True
print(isinstance(a, str))  # False
```

类也是对象，类就是一个用来创建对象的对象。例如：

```
class MyClass:
    pass


print(id(MyClass))  # 1947031180304
print(type(MyClass))  # <class 'type'>
```

类是`type`类的对象，定义类就是创建了一个`type`类的对象。

7.2 属性和方法

7.2.1 属性

在类中定义的变量称为**类属性**，所有该类实例都可以访问这些属性。可以使用`类名.属性名`或`对象名.属性名`来访问类属性。

类属性会保存到类对象中，而不会保存到实例对象中。

也可以直接向对象中添加变量，对象中的变量称为**实例属性**。实例属性会保存在对象中。实例属性只能通过`对象.属性名`来访问。例如：

```
class MyClass:
    pass


obj = MyClass()
obj.name = '小明'
print(obj.name)  # 小明
```

如果使用`对象.类属性 = 值`赋值语句，只会改变对象的实例属性，而不会影响到类属性的值。例如：

```
class MyClass:
    a = 10  # 类属性 a


obj = MyClass()
print(obj.a)  # 10

obj2 = MyClass()
print(obj2.a)  # 10

obj2.a = 20  # 为对象 obj2 创建实例属性，不影响类属性
print(obj2.a)  # 20
print(obj.a)  # 10
```

属性的查找流程：先在对象中查找是否含有该属性，如果有，则直接访问；如果没有，则到当前对象的类对象中去寻找，如果找到则访问；如果都没有找到，则报错。

类对象和实例对象中都可以保存属性。如果属性是所有实例共享的，则应该将其保存到类对象中；如果属性是某个实例独有的，则应该将其保存到实例对象中。

一般情况下，将属性保存到实例对象中，将方法保存到类对象中。

7.2.2 实例方法

在类中定义的函数称为**方法**（method）。所有方法都会保存到类对象中。

在类中定义、第一个形参为`self`的方法称为**实例方法**。`self`参数用于接收调用方法的对象，在方法中可以使用`self`来访问调用者的属性和方法。

可以使用`类名.方法名()`或`对象名.方法名()`来调用实例方法。

- 通过实例对象调用实例方法时，会自动将调用者作为`self`参数传入。
- 通过类名调用实例方法时，不会自动传递`self`参数，需要手动传递。

7.2.3 类方法

在类中定义、使用`@classmethod`修饰的方法称为**类方法**。

类方法的第一个形参通常定义为`cls`，用于接收类对象。

类方法可以通过类名调用，也可以通过实例对象调用，二者没有区别。

7.2.4 静态方法

在类中定义、使用`@staticmethod`修饰的方法称为**静态方法**。

静态方法没有默认参数。

静态方法可以通过类名调用，也可以通过实例对象调用。

静态方法与类无关，它只是一个保存到类中的函数。静态方法一般都是工具方法。

7.3 对象的初始化

在类中可以定义一些特殊方法，也称为魔术方法。特殊方法以`__`开头，以`__`结尾。特殊方法会在特殊的时刻自动调用，不需要手动调用，一般不会直接调用特殊方法。

`__init__()`方法在创建对象时执行，用于对新对象的实例属性进行初始化。使用`类名()`创建对象时可以传递参数，这些参数都会传递给`__init__()`方法，在`__init__()`方法中完成初始化操作。例如：

```
class MyClass:
    def __init__(self, name):
        # 使用 self 设置新对象的属性
        self.name = name


obj = MyClass('小明')
print(obj.name)  # 小明
```

创建对象的流程：

1. 在内存中为对象分配空间。
2. 执行 `__init__()` 方法，为对象初始化。

7.4 封装

在属性名之前添加双下划线 `__`，就可以将该属性隐藏起来，只能在类的内部访问，类外访问不到。例如：

```
class Person:
    def __init__(self, name):
        self.__name = name  # 将 name 属性隐藏起来

    def get_name(self):
        return self.__name  # 在类的内部访问 __name 属性

    def set_name(self, name):
        self.__name = name


obj = Person('小明')
print(obj.get_name())  # 小明
print(obj.__name)  # AttributeError: 'Person' object has no attribute '__name'
```

属性的隐藏是通过自动修改属性名来实现的，Python 解释器会在属性名的前面添加 `_类名`，因此仍然可以通过修改后的属性名访问到该属性。例如：

```
class Person:
    def __init__(self, name):
        self.__name = name  # 实际上是将属性名改成了 _Person__name

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name


obj = Person('小明')
print(obj._Person__name)  # 小明
```

类似地，在方法名前面添加双下划线 `__` 也可以将方法隐藏起来。

在属性名和方法名前面添加 `__` 并不能实现彻底的封装，所以一般不用这种方式。

一般情况下，以单下划线 `_` 开头表示该属性或方法是私有的，希望不要从类外直接访问。但这只是一种标记，希望其他人遵守，实际上并不能阻止对私有属性和方法的访问。

7.5 `@property` 装饰器

如果在方法名之前使用 `@property` 装饰器，调用该方法时就可以省略括号。可以用这种方式改造 getter 方法，改造后就可以像访问属性一样调用 getter 方法。例如：

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name


obj = Person('小明')
print(obj.name)  # 小明
```

可以用 `@属性名.setter` 装饰 setter 方法，要求必须存在用 `@property` 装饰的 getter 方法，并且 setter 方法和 getter 方法的名字必须与 `@属性名.setter` 中的属性名相同。装饰之后，就可以通过赋值来修改属性。例如：

```
class Person:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        self._name = name


obj = Person('小明')
print(obj.name)  # 小明

obj.name = '小刚'
print(obj.name)  # 小刚
```

7.6 继承

7.6.1 子类

在定义类时，可以在类名之后用小括号指定父类。子类会直接继承父类中所有的属性和方法。例如：

```
class Animal:
    def eat(self):
        print('动物吃东西')

    def sleep(self):
        print('动物睡觉')


# 子类，继承 Animal 类
class Dog(Animal):
    def bark(self):
        print('汪汪汪')


dog = Dog()
dog.eat()  # 动物吃东西
dog.sleep()  # 动物睡觉
dog.bark()  # 汪汪汪
```

如果在定义类时不指定父类，则默认以`object`类为父类。`object`类是所有类的父类，所有的类都继承自`object`类。

`issubclass(A, B)`函数用来检查`A`是否为`B`的子类，如果是则返回`True`，否则返回`False`。

使用`isinstance()`函数时，如果指定的对象是指定类的子类的实例，也会返回`True`。任何对象都是`object`类的实例，因此`isinstance(xxx, object)`总会返回`True`。

7.6.2 方法的重写

如果在子类中定义父类中的同名方法，就构成了方法的重写（override），此时子类的方法会覆盖父类的同名方法。当子类对象调用该方法时，调用的是子类的方法。例如：

```python
class Animal:
    def eat(self):
        print('动物吃东西')

    def sleep(self):
        print('动物睡觉')


class Dog(Animal):
    # 重写 Animal 类中的 eat() 方法
    def eat(self):
        print('狗吃东西')

    def bark(self):
        print('汪汪汪')


dog = Dog()
dog.eat()  # 狗吃东西
dog.sleep()  # 动物睡觉
dog.bark()  # 汪汪汪
```

调用一个对象的方法时，先去当前类中寻找是否存在该方法，如果有则直接调用，如果没有则去父类中寻找；如果父类中有，则调用父类中的方法，如果没有则去父类的父类中寻找。以此类推，直到`object`类，如果依然没有，就会报错。

7.6.3 `super()`函数

`super()`函数可以获取当前类的父类，通过这种方式可以在子类中访问父类的属性和方法。例如：

```python
class Animal:
    def __init__(self, name):
        self._name = name


class Dog(Animal):
    def __init__(self, name, age):
        super().__init__(name)  # 调用父类的 __init__() 方法
        self._age = age
```

7.6.4 多继承

Python 支持多继承，可以为一个类指定多个父类。定义类时，多个父类用逗号隔开。

`类名.__bases__`可以获取当前类的所有父类，返回值为元组。例如：

```
class A:
    pass


class B:
    pass


class C(A, B):
    pass


print(C.__bases__)  # (<class '__main__.A'>, <class '__main__.B'>)
```

如果在多个父类中有同名方法，则子类调用该方法时需要指定调用哪个父类中的方法。如果子类调用时没有指定，则会按照从左到右的顺序查找父类，一旦找到就不再继续查找。因此写在前面的父类会覆盖后面类的同名方法。

应尽量避免使用多继承。

7.7 多态

鸭子类型：看起来像鸭子，叫起来也像鸭子，那么肯定就是鸭子。

指一种编程风格，它并不依靠查找对象类型来确定其是否具有正确的接口，而是直接调用或使用其方法或属性。由于强调接口而非特定类型，设计良好的代码可通过允许多态替代来提升灵活性。

鸭子类型避免使用`type()`或`isinstance()`检测，而往往会采用`hasattr()`检测或是 EAFP 编程。

EAFP：Easier to Ask for Forgiveness than Permission（求原谅比求许可更容易）。这种 Python 常用代码编写风格会假定所需的键或属性存在，并在假定错误时捕获异常。这种简洁快速风格的特点就是大量运用`try`和`except`语句。

与 EAFP 相对的则是 LBYL（Look Before You Leap，先查看后跳跃）风格，常见于 C 等许多其他语言。这种代码编写风格会在进行调用或查找之前显式地检查前提条件。此风格与 EAFP 方式恰成对比，其特点是大量使用`if`语句。

在多线程环境中，LBYL 方式会导致“查看”和“跳跃”之间发生条件竞争风险。这种问题可通过加锁或使用 EAFP 方式来解决。

例如，只要一个类定义了`__len__()`特殊方法，就可以使用`len()`函数来获得该类对象的长度。`len()`函数并不检查对象的类型，只要有`__len__()`方法就可以。序列对象都可以通过`len()`函数得到长度，是因为序列类都定义了`__len__()`方法。

7.8 垃圾回收

程序运行过程中会产生垃圾，这些垃圾会影响程序的性能，因此必须及时清理。

没有被引用的对象就是垃圾。垃圾回收就是将垃圾对象从内存中删除。

Python 有自动垃圾回收机制，它会自动将垃圾对象删除。不用手动处理垃圾回收。

`__del__(self)`是一个特殊方法，它会在对象被回收前调用。

7.9 特殊方法

`__str__(self)`特殊方法会在使用`str()`将对象转换为字符串的时候调用，返回转换后的字符串。例如：

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

    def __str__(self):
        return 'Person [name=%s, age=%d]' % (self.name, self.age)


person = Person('小明', 18)
print(str(person))  # Person [name=小明, age=18]
```

`__bool__(self)`特殊方法会在使用`bool()`将对象转换为布尔值的时候调用，用于指定对象转换为布尔值的规则。

`__repr__(self)`特殊方法会在对当前对象使用`repr()`函数时调用，设置`repr()`函数的返回值。`repr()`函数用来在交互模式中直接输出对象。例如：

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

    def __repr__(self):
        return 'Person [name=%s, age=%d]' % (self.name, self.age)


person = Person('小明', 18)
print(repr(person))  # Person [name=小明, age=18]
```

用于对象间比较的特殊方法：

1. `__lt__(self, other)`：小于，在用`<`比较对象时调用，返回值作为比较结果。
2. `__le__(self, other)`：小于等于，在用`<=`比较对象时调用，返回值作为比较结果。
3. `__eq__(self, other)`：等于，在用`==`比较对象时调用，返回值作为比较结果。
4. `__ne__(self, other)`：不等于，在用`!=`比较对象时调用，返回值作为比较结果。
5. `__gt__(self, other)`：大于，在用`>`比较对象时调用，返回值作为比较结果。
6. `__ge__(self, other)`：大于等于，在用`>=`比较对象时调用，返回值作为比较结果。

例如：

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

    def __gt__(self, other):
        return self.age > other.age


p1 = Person('小明', 18)
p2 = Person('小刚', 20)
print(p1 > p2)  # False
```

用于对象间运算的特殊方法：

1. `__add__(self, other)`：加法
2. `__sub__(self, other)`：减法
3. `__mul__(self, other)`：乘法
4. `__truediv__(self, other)`：除法
5. `__floordiv__(self, other)`：地板除
6. `__mod__(self, other[, modulo])`：取余
7. `__pow__(self, other)`：幂运算
8. `__lshift__(self, other)`：左移
9. `__rshift__(self, other)`：右移
10. `__and__(self, other)`：逻辑与
11. `__xor__(self, other)`：逻辑异或
12. `__or__(self, other)`：逻辑或

8 模块化

8.1 模块

模块化是指将一个完整的程序分解为多个模块。

模块化的优点：

1. 便于协同开发
2. 便于维护
3. 模块可以复用

在 Python 中，一个 .py 文件就是一个模块（module），创建模块就是创建一个 .py 文件。

模块名（即 .py 文件名）要符合标识符的命名规则。

引入外部模块的方式：

1. `import 模块名`
2. `import 模块名 as 别名`：引入模块，并为模块设置别名。

引入一个模块时，会自动执行该模块内部的代码。

模块也是一个对象。同一个模块可以引入多次，但模块的实例只会创建一个。

`import`可以写在程序的任意位置，但是一般情况下，都会将`import`写在程序开头。

每一个模块内部都有一个`__name__`属性，通过这个属性可以获取模块的名字。

`__name__`属性值为`__main__`的模块是主模块，一个程序中只有一个主模块。当前执行的 .py 文件就是主模块。

可以用`模块名.变量名`访问外部模块中的变量，用`模块名.函数名`访问外部模块中的函数，用`模块名.类名`访问外部模块中的类。

可以只引入模块中的部分内容，语法为：`from 模块名 import 名字`。引入的内容可以是变量、函数或类，多个内容之间用逗号隔开。按这种方式引入后，可以直接用名字来访问相应的内容，不用在前面写模块名。

可以为引入的内容设置别名，语法为：`from 模块名 import 名字 as 别名`

`from 模块名 import *`可以引入模块中的所有内容。一般不会使用这种方式。

可以在变量、函数或类的名字之前添加`_`，在其他模块中通过`from 模块名 import *`引入该模块时，不会引入以`_`开头的内容。

可以在模块中编写测试代码，这部分代码只有在当前模块作为主模块时才执行，而当前模块被其他模块引入时不执行。为此需要检查当前模块是否为主模块，方法如下：

```python
if __name__ == '__main__':
    # 测试代码
```

8.2 包

包（package）也是一个模块。当模块中代码过多时，或者需要将一个模块分解为多个模块时，就需要使用包。

一个包就是一个文件夹。包中必须要有一个`__init__.py`文件，在这个文件中编写包中的主要内容。

包中可以有多个模块，通过`import 包名`引入整个包，通过`from 包名 import 模块名`引入包中的模块。

为了提高程序运行的效率，Python 会将编译生成的机器语言代码保存到缓存文件中。当下一次加载模块（包）时，就可以不再编译，而是直接加载缓存文件。缓存文件扩展名为 .pyc，一个包的缓存文件统一保存在这个包中的`__pycache__`文件夹中。

8.3 Python标准库

8.3.1 `sys`模块

`sys`模块提供了一些变量和函数，可以用来获取 Python 解析器的信息，或者通过函数来操作 Python 解析器。

`sys.argv`属性是一个列表，保存了执行代码时的命令行参数。

`sys.modules`属性是一个字典，保存当前程序中引入的所有模块，键是模块的名字，值是模块对象。

`sys.path`属性是一个列表，保存模块的搜索路径。当引入模块时，就按照`sys.path`中的路径依次寻找相应的模块。

`sys.platform`属性表示当前 Python 运行的平台。例如：

```python
import sys

print(sys.platform)  # win32
```

`sys.exit()`函数用来退出程序。可以传递一个字符串参数作为错误信息。

8.3.2 `pprint` 模块

`pprint` 模块提供了一个 `pprint()` 函数，用来对打印的数据做简单的格式化。例如：

```
import pprint

d = {'1': 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa', '2': 123, '3': True, '4': None}
pprint.pprint(d)
```

```
{'1': 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa',
 '2': 123,
 '3': True,
 '4': None}
```

8.3.3 `os` 模块

`os` 模块提供了对操作系统进行访问的方法。

`os.environ` 属性用于获取系统的环境变量。

`os.system()` 函数可以用来执行操作系统的命令。

`os.listdir()` 函数获取指定目录的目录结构。返回一个列表，包括该目录中所有文件和文件夹的名字。参数为路径，默认参数为 `.`，表示当前路径。

`os.getcwd()` 函数获取当前所在的目录。返回一个字符串，表示当前所在目录的绝对路径。

`os.chdir()` 函数用于切换当前所在的目录。

`os.mkdir()` 函数用于在当前目录下创建一个目录，参数为目录的名字。

`os.rmdir()` 函数用于在当前目录下删除目录，参数为目录的名字。如果目录不存在，则抛出 `FileNotFoundError` 异常。如果目录不为空，则抛出 `OSError` 异常。

`os.remove()` 函数用于删除文件，参数为文件路径。如果文件不存在，则抛出 `FileNotFoundError` 异常。如果参数是一个目录而不是文件，则抛出 `IsADirectoryError` 异常。

`os.rename()` 函数用于重命名文件或移动文件，第一个参数为文件路径，第二个参数为新的名字或新的路径。

8.3.4 `time` 模块

`time.time()` 函数用来获取当前时间，返回从 1970 年 1 月 1 日 0 时 0 分 0 秒到现在经过的秒数。

9 异常

9.1 异常简介

程序在运行过程中，不可避免的会出现一些错误，这些错误称为异常。

一旦出现异常，程序会立即终止，后面的代码全都不会执行。

9.2 异常处理

`try`语句用于处理异常，语法为：

```
try:
    代码块  # 可能出现异常的代码
except:
    代码块  # 异常处理代码
else:
    代码块  # 没有出现异常时执行的代码
finally:
    代码块  # 无论是否出现异常都会执行
```

将可能出现异常的代码放到`try`语句中，如果没有异常则跳过`except`子句，执行`else`和`finally`子句中的代码；如果出现异常，则执行`except`子句中的代码，`else`子句不执行，`finally`子句执行。

`try`是必需的，`else`可选，`except`和`finally`至少有二者之一。

9.3 异常传播

在函数中出现异常时，如果在函数中对异常进行了处理，则异常不会继续传播；如果函数中没有对异常进行处理，则异常会向函数调用处传播。直到传递到全局作用域，如果依然没有处理，则程序终止，并显示异常信息。

当出现异常时，所有的异常信息会被保存到异常对象中。异常传播时，实际上是将异常对象抛给了调用处。

9.4 异常对象

Python 内置了多种异常类，例如，`ZeroDivisionError`类表示除 0 异常，`NameError`类表示变量错误的异常，等等。

可以在`except`子句中指定异常类型，语法为：`except 异常类型:`。此时该`except`子句只会捕获该类型的异常。

如果在`except`子句中不指定异常类型，则会捕获所有类型的异常。

`try`语句中可以有多个`except`子句，分别捕获不同类型的异常，对不同类型的异常做不同的处理。

`Exception`类是所有异常类的父类，使用`except Exception:`将会捕获所有类型的异常。

可以在`except`子句中接收指定类型的异常对象，语法为：`except 异常类型 as 变量名:`。会将捕获到的指定类型的异常对象赋值给变量。

可以使用`raise`语句来抛出异常，`raise`关键字之后要跟一个异常类或一个异常对象。在创建异常对象时可以传递一个字符串参数作为错误信息。

自定义异常类时，只需要创建一个类并继承`Exception`类即可。例如：

```
class MyError(Exception):
    pass
```

10 文件

10.1 打开和关闭文件

使用`open()`函数打开文件，第一个参数`file`表示要打开的文件的路径。

`open()`函数会返回一个对象，这个对象代表当前打开的文件。对文件的所有操作都要通过文件对象来进行。

完成对文件的操作之后，要关闭文件。文件对象的`close()`方法用于关闭文件。关闭文件后，就不能再对该文件对象进行文件操作。

使用`with-as`语句可以更方便地进行文件操作，语法为：

```
with 表达式 as 变量:
    代码块
```

将`open()`函数作为表达式，返回的文件对象赋值给变量，在代码块中执行文件操作。当代码块结束时，会自动调用`close()`方法，代码块外部无法通过变量来操作文件。

当文件路径不正确，无法找到文件时，`open()`函数会抛出`FileNotFoundError`，因此通常会将文件操作放在`try`语句中。

文件操作的标准格式：

```
try:
    with open(file_name) as file_obj:
        # 文件操作代码
except FileNotFoundError:
    # 异常处理代码
```

文件分为两种，一种是文本文件，另一种是二进制文件。使用`open()`函数打开文件时，默认按照文本文件的方式打开，并且编码为`None`。所以打开文本文件时，必须指定文件的编码。`open()`函数有一个`encoding`参数用来指定编码，可以将编码方式以字符串的形式传递给`encoding`参数。

10.2 文本文件

10.2.1 读取文本文件

文件对象的`read()`方法会将文件的所有内容以字符串形式返回。

`read()`方法会将文件的所有内容一次性加载到内存中，如果文件较大，容易导致内存溢出，所以对于较大的文件，不要直接调用`read()`方法。

`read()`方法的第一个参数为`size`，用于指定要读取的字符数量，默认值为 -1，表示读取所有字符。为`read()`方法指定`size`参数后，会读取指定数量的字符，并且每次读取都是从上次读取结束的位置开始的；如果剩余的字符数量比`size`小，则读取所有剩余字符；如果已经到达文件尾部，继续读取时会返回空串。

分次读取文件的方式：

```
try:
    with open(file_name, encoding='utf-8') as file_obj:
        file_content = ''  # 保存文件内容
        chunk = 100  # 指定每次读取的字符数量

        # 循环读取文件
        while True:
            content = file_obj.read(chunk)

            if not content:  # 如果读取到空串，说明读取完毕，退出循环
                break

            file_content += content
except FileNotFoundError:
    # 异常处理代码
```

`readline()`方法可以读取一行字符。

`readlies()`方法逐行读取整个文件，将读取到的内容包装为列表返回，列表中的每个元素是一行。

可以用`for`循环直接对文件对象进行遍历，每次遍历可以得到一行。例如：

```
for t in file_obj:
    print(t)  # t 是文件中的一行
```

10.2.2 写入文本文件

使用`open()`函数打开文件时，必须指定要做的操作（读取、写入、追加）。如果不指定操作类型，默认为读取文件，而不能写入文件。

`open()`函数的第二个参数为`mode`，用于指定操作类型，默认值为`'r'`，表示只读。`mode`参数可用的取值有：

1. `r`：只读。如果文件不存在会报错。
2. `w`：只可写，不可读。打开文件时，如果文件不存在，则创建空文件；如果文件存在，则清空文件。
3. `a`：只可追加，不可读。打开文件时，如果文件不存在，则创建空文件；如果文件存在，不做处理。写入文件时向文件最后添加内容。
4. `x`：新建，可写，不可读。如果文件不存在则创建空文件，如果文件存在则报错。
5. `r+`：可读可写。打开文件时，如果文件不存在会报错；如果文件存在，不做处理。写入文件时向文件最后添加内容。
6. `w+`：可读可写。就是在`w`类型的基础上增加了读权限。
7. `a+`：可追加，也可读。就是在`a`类型的基础上增加了读权限。
8. `x+`：新建，可读可写。就是在`x`类型的基础上增加了读权限。

`write()`方法用于向文件写入。对于文本文件，需要传入一个字符串作为参数。可以分多次写入。写入完成后，`write()`方法会返回写入的字符个数。

10.3 二进制文件

`open()`函数的`mode`参数还有 2 种取值，用于指定打开方式：

1. `t`：按文本文件打开，默认值。
2. `b`：按二进制文件打开。

2 种打开方式可以和操作类型一起使用。例如，`wb`是指同时使用`w`操作类型和`b`打开方式。

按二进制方式打开文件时，不需要指定`encoding`参数。

`read()`方法也可以读取二进制文件。读取文本文件时，`size`属性表示字符的个数；读取二进制文件时，`size`属性表示字节个数。

`write()`方法也可以向二进制文件写入。

`tell()`方法获取当前读取的位置，返回值为整数，单位为字节。

`seek(offset, whence)`方法设置当前读取的位置，并返回设置后的位置。2 个参数都是整型，`offset`表示偏移量，`whence`表示基准位置，从`whence`指定的位置加上`offset`得到结果。`whence`有 3 个可选值：

1. `SEEK_SET`或`0`，表示开头位置，默认值。
2. `SEEK_CUR`或`1`，表示当前位置。
3. `SEEK_END`或`2`，表示文件末尾。

`tell()`和`seek()`方法既可以用于二进制文件，也可以用于文本文件，但二进制文件更为常用。当文本文件中有中文时，每个字符的字节长度不相同，调用`seek()`方法可能会将一个字符截断，之后读取文件时会抛出`UnicodeDecodeError`异常，因此一般不会对文本文件使用`tell()`和`seek()`方法。