数据结构

Python中有四种最基本的数据类型，可以使用`type()`获得数据类型

---

int & float & bool

python中int没有大小限制，可以表示任意大的整数

并且python没有独立的double类型，float就是双精度浮点数，使用64位来储存，提供15-17位十进制有效数字的精度

尽管python中float精度很高，但由于内部是二进制储存的问题，某些十进制小数（0.1，0.2）无法被精确表示，导致著名的精度问题

```
result = 0.1 + 0.2
print(result)
# 输出: 0.30000000000000004
```

如果需要更精确的计算，需要使用`decimal`模块

数值类型自身的方法很少，一般与一些内置函数一起使用

- `abs(number)`：返回绝对值
- `round(number,ndigits)`：对浮点数进行四舍五入
- `float()`: 转为浮点数
- `int()`: 转为整数

---

str

在python中没有独立的char类型，单个字符视为长度为1的字符串类型

字符串一旦创建，就不能被修改。所有对字符串的操作（如替换、拼接）都会生成一个新的字符串对象

并且字符串是有序的，可以通过索引访问字符串中的每个字符

```
my_string = "Hello"
first_char = my_string[0]

print(f"first_char 的值是: {first_char}")
print(f"first_char 的类型是: {type(first_char)}")
print(f"first_char 的长度是: {len(first_char)}")
```

常用方法：

- `str.lower()/str.upper()`：转小写/转大写
- `str.strip()/str.lstrip()`：移除两断空白/移除左端空白

- `str.split(separator)`：按分隔符切片，返回list
- `str.join(list)`：用字符串连接列表元素，返回str
- `str.find(sub)`：查找子串位置
- `str.replace(old,new)`：替换子串

```
text = "Hello, world!"
index = text.find("world")
replaced_text = text.replace("world", "Python")
print(f"'world'的位置: {index}") # 输出: 'world'的位置: 7
print(f"替换后的文本: {replaced_text}") # 输出: 替换后的文本: Hello, Python!

words = text.split(" ")
joined_words = "-".join(words)
print(f"分割成列表: {words}") # 输出: 分割成列表: ['Hello,', 'world!']
print(f"连接成字符串: {joined_words}") # 输出: 连接成字符串: Hello,-world!
```

值得注意的是，split( )方法默认是根据任何空白字符（如空格，制表符，换行符等）进行分隔，而不是分隔成单个字符

想要分隔成单个字符，需要传入一个空字符串`''`作为分隔符

```
name = "Hello"
characters = name.split('')
print(characters)
# 输出: ['H', 'e', 'l', 'l', 'o']
```

此外，对与字符串的拼接，如果在循环中频繁使用`+=`会产生大量临时字符串对象，导致性能下降，推荐使用`"".join(list)`

---

list

list是python中最常用和重要的数据结构之一，列表中元素有固定的顺序，可以通过索引（从0开始）访问任何元素，且列表可以包含不同数据类型的元素

基本用法：

利用`[]`创建列表

```
# 创建一个空列表
empty_list = []
fruits = ["apple","banana","cherry"]
# 列表可以包含不同的数据类型
mixed_list = [1,"two",3.0,True,fruits]
```

列表的访问主要通过索引和切片两种方式实现

1. 索引：访问单个元素
    - 正向索引：从列表的开头开始，0是第一个元素，1是第二个元素
    - 负向索引：从列表的末尾开始，-1是倒数第一个元素，-2是倒数第二个元素

如果访问索引超出列表范围，会抛出IndexError的错误

2. 切片：访问子列表`list[start:stop:step]`
    - `start`：起始索引（包含）。如果省略默认为开头（0）
    - `stop`：结束索引（不包含）。如果省略，默认为结尾
    - `step`：步长。如果省略，默认为1

```
fruits = ["apple","banana","cherry","orange","grape"]
print(f"{fruits[1:4]}")
# 输出: ['banana','cherry','orange']
print(f"{fruits[::-1]}")
# 输出: ['grape','orange','cherry','banana','apple']
```

值得注意的是，将一个列表赋值给另一个变量时，它们实际上指向了内存中的同一个列表对象，修改一个，另一个也会跟着改变

```
original_list = [1, 2, 3]
new_list = original_list  # 这不是拷贝，而是引用
new_list.append(4)
print(f"原始列表: {original_list}")
# 输出: 原始列表: [1, 2, 3, 4]
```

解决方法是使用切片`[:]`或`copy()`方法创建浅拷贝，得到一个独立的新列表

常用方法：

方法	描述	实例
`list.append(item)`	在列表末尾添加元素	`fruits.append("kiwi")`
`list.insert(index,item)`	在指定索引处插入元素	`fruits.insert(1,"grape")`
`list.extend(list)`	将一个可迭代对象中所有元素添加到当前列表末尾	`fruits.extend(["mango","fig"])`
`list.remove(item)`	删除列表中第一个匹配的元素	`fruits.remove("banana")`
`list.pop(index)`	删除指定索引的元素并返回该元素，如果省略索引，默认删除最后一个	`last_fruit = fruits.pop()`
`list.sort()`	在原地对列表进行升序排列	`numbers.sort()`
`list.reverse()`	在原地反转列表元素顺序	`fruits.reverse()`
`list.index(item)`	返回第一个匹配元素的索引	`idx=fruits.index("cherry")`

方法	描述	实例
`list.count(item)`	返回元素在列表中出现的次数	`num_apples=fruits.count("apple")`
`list.copy()`	返回列表的一个浅拷贝	`new_list=my_list.copy()`

dict

字典以键值对的形式储存数据，一个字典不能有两个相同的键，字典的键必须是不可变的数据类型

基本用法：

使用`{}`来创建字典，或者使用`dict()`函数

```
# 创建一个空字典
empty_dict = {}
person = {"name":"Bob","age":30,"city":"New York"}
```

可以通过键来访问、修改或添加元素

常用方法：

方法	描述	实例
`dict.keys()`	返回一个由所有键组成的视图	`person.keys()`
`dict.values()`	返回一个由所有值组成的视图	`person.values()`
`dict.items()`	返回一个由所有键值对（元组）组成的视图	`person.items()`
`dict.get(key,default)`	获取指定键的值，若键不存在则返回default	`person.get('age',0)`
`dict.update(other_dict)`	使用另一个字典的键值对来更新当前字典，若键不存在会添加到字典中，若键已存在则会将新值覆盖	`person.update({'job':'Engineer'})`
`dict.pop(key,default)`	移除指定键的键值对，并返回其值	`age=person.pop('age')`
`dict.copy()`	返回字典的一个浅拷贝	`new_dict=person.copy()`

tuple

元组可以视作常列表，被创建后元素不能被修改、添加或删除

基本用法：

创建元组用`()`

```
# 创建一个空元组
empty_tuple = ()
coordinates = (10,"hellp",3.14)
# 创建单元素元组时，必须在元素后加一个逗号
single_item_tuple = (5,)
```

尤其要注意单元组的语法!

元组的访问和列表完全相同

任何看似修改元组的操作，实际上都是创建了一个新的元组

并且由于元组的不可变性，元组可以作为字典的键

常用方法:

方法	描述	实例
`tuple.count(item)`	返回元素在元组中的出现次数	`my-tuple.count(2)`
`tuple.index(item)`	返回元素第一次出现的索引	`my_tuple.index('b')`

---

set

set是python中非常独特的数据结构，擅长处理唯一且无序的数据，因为集合中的元素没有固定的顺序，所以不能像列表那样通过索引来访问或修改元素，且集合会自动移除所有重复的元素

基本用法:

创建集合有两种方式

```
# 从一个可迭代对象（如列表）创建集合，会自动去重
my_list = [1,2,3,2,4,1]
my_set = set(my_list)
# 使用花括号{}创建
my_set2 = {5,6,7}
# 注意：创建空集合必须使用set()，使用{}创建的是空字典
empty_set = set()
```

可以使用add( )或update( )方法来添加元素，使用remove( )或discard( )来删除元素 ， 使用discard( )删除不存在的元素不会报错

集合运算

集合最强大的地方在于它的数学运算，可以高效的处理数据

方法	运算符	描述	实例
`s.union(t)`	`s|t`	返回两个集合的并集	{1,2}|{2,3}->{1,2,3}

方法	运算符	描述	实例
`s.intersection(t)`	`s&t`	返回两个集合的交集	{1,2}&{2,3}->{2}
`s.difference(t)`	`s-t`	返回两个集合的差集	{1,2}-{2,3}->{1}
`s.issubset(t)`	`s<=t`	检查s是否是t的子集	{1,2}<={1,2,3}->TRUE
`s.issuperset(t)`	`s>=t`	检查s是否是t的超集	{1,2,3}>={1,2}->TRUE

基本语法

python中用缩进来表示代码的层次结构，并且每个语句不需要分号，这是python的美观哲学，因此在写python过程中不能胡乱缩进

定义变量

- 在python中不需要提前声明数据类型，只需要给变量一个名字并直接赋值，python会自动识别数据类型

条件语句

- `if`
- `elif`
- `else`

三元运算符：对于简单的 `if-else`表达式，可以用更加紧凑的三元运算符

```
status = "成年" if age >= 18 else "未成年"
```

此外，条件语句常用`in`运算符来检查某个元素是否在序列中

```
fruits = ["apple","banana"]
if "apple" in fruits:
    print("小苹果! ")
```

循环

python中主要有 `for`和 `while`循环，并没有 `do...while`

for:

for可以遍历任何可迭代对象（列表，字符串，元组等）中的元素

`for item in iterable:`

```
my_string = "Pyhon"
for char in my_string:
    print(char)
```

需要注意的是，在for char中相当于同时给char进行了声明和赋值，而无需提前声明

此外对于整数序列的遍历，常用 `range()`函数：

- `range(stop)`：从0到stop-1
- `range(start,stop)`：从start到stop-1
- `range(start,stop,step)`：从start到stop-1，步长为step

```
for i in range(3):
    print(i,end="")
# 输出: 0, 1, 2
```

为了写出简洁美观的代码，可以灵活运用推导式

`[expression for item in iterable if condition]`:

- `expression`：想对item执行的操作
- `item`：可迭代对象中的每个元素
- `iterable`：想遍历的序列（列表，字符串，range( )等）
- `if condition`：可选部分，用于过滤元素

```
# 列表推导式
squares_comp = [i*i for i in range(1,11)]
```

此外，在有序容器中，如果需要在遍历时同时获取索引和元素，常用 `enumerate()`函数

```
fruits = ["apple","banana","cherry"]
for index, fruit in enumerate(fruits):
    print(f"索引{index}: {fruit}")
```

进一步，如果需要同时遍历两个或多个列表时，可以使用 `zip()`函数

```
names = ["Alice","Bob"]
ages = [25,30]
for name,age in zip(names,ages):
    print(f"{name}今年{age}岁。")
```

while:

- `break`：立即跳出循环
- `continue`：跳过当前循环的剩余部分，直接进入下一次循环

迭代

`iter()`函数是python迭代机制的基石，可以手动从一个可迭代对象（iterable）中获取迭代器（iterator），实现更精细的控制

但不同于C++，python中的迭代器是单向的，且只能使用next( )来获取下一个元素，其底层是一个简单对象，实现了 `__iter__`和`__next__`方法

```
my_list = [1,2,3]
iterator = iter(my_list)
print(next(iterator))  # 输出: 1
print(next(iterator))  # 输出: 2
# 当迭代器耗尽后便无法恢复
```

此外iter( )还有一个更高级的用法：哨兵值迭代，iter( )接收两个参数，一个可调用的对象（例如一个函数），和一个哨兵值

`iter(callable,sentinel)`会创建一个迭代器，不断调用 `callable()`，直到返回值等于 `sentinel`时停止

```
with open('large_file.txt','r') as file:
    # iter()会不断调用f.read(1024)
    # 直到返回值等于''时停止，即为空时
    for chunk in iter(lambda: f.read(1024),''):
        print(f"读取到{len(chunk)}个文字。")
```

当一个函数包含 `yield`语句时，它就变成了一个生成器函数，调用这个函数时，它不会立即执行函数体内的代码，而是返回一个生成器对象

生成器是一种特殊的迭代器，它不会一次性将所有数据加载到内存中，而是在每次迭代时按需生成一个值，在内存首受限的情况下十分高效

理解 `yield`的最好方式是将其视作一个特殊的 `return`，但它返回一个生成器对象，而不是立即返回一个值并结束函数

```
# 一个生成器函数，按需生成斐波那契数列
def fibonacci():
    a,b = 0,1
    while 1:
        yield a
        a,b = b,a+b
# 可以只取出前5个数，而不会生成无限个
fib_gen = fibonacci()
for i in range(5):
```

```
    print(next(fib_gen),end=" ")
# 输出: 0 1 1 2 3
```

函数

python中函数不需要写返回值类型，只需加上 `def`关键字即可创建函数

如果在函数内部没有使用 `return`语句，会隐式的返回 `None`

同样，python中函数也是局部作用域，内部定义的变量只在内部可见

在更加良好习惯下的定义函数，可以使用三引号编写文档字符串，来解释函数的作用、参数和返回值

```
def get_full_name(first_name,last_name):
    """
    拼接姓和名，返回完整的名字
    Args:
        first_name: 名
        last_name: 姓
    Return:
        返回完整的名字字符串
    """
    return f"{first_name}{last_name}"
```

参数类型:

python中参数类型非常灵活，可分为四种

- 位置参数：最基本的参数，在调用函数时，传入参数顺序必须与函数定义时的顺序一样

```
def describe_pet(animal,name):
    print(f"我的宠物是一只{animal}，它的名字是{name}")

describe_pet("dog","papa")
# 必须按顺序传入
```

- 关键字参数：在调用函数时，可以通过参数的名称来赋值。好处是参数顺序更不重要，代码意图更加明显

```
def create_user(name,age,city):
    print(f"创建用户：{name}，年龄{age}，城市{city}")

create_user(age=25,name="kekdou",city="Peking")
# 使用关键字参数，顺序可以颠倒
```

- 默认参数：定义函数时设置缺省值

python中参数默认值规则与C++类似，所有带有默认值的参数都必须放在没有默认值的参数之后

但需要注意可变参数的陷阱，在python中，函数的默认参数是在函数被定义时（即加载到内存时）初始化的，而不是在每次函数被调用时

如果默认参数是一个可变对象，那么所有对该函数的调用都会共享一个对象

```
def add_item(item,list=[1,2]):
    list.append(item)
    return list

list1 = add_item(3)
print(f"第一次调用：{list1}")
list2 = add_item(4)
print(f"第二次调用：{list2}")
# 第一次调用：[1,2,3]
# 第二次调用：[1,2,3,4]
```

尽管好像不太常用(

- 可变参数：
    - *args：用于接收任意数量的位置参数，这些参数会打包成一个元组
    - **kwargs：用于接收任意数量的关键字参数，这些参数会打包成一个字典

```
def print_info(name,*subjects,**grades):
    print(f"学生姓名：{name}")
    print(f"所选课程：{subjects}")
    print("各科成绩：")
    for subject,grade in grades.items():
        print(f"{subject}:{grade}")

print_info("kekdou","math","English",english=95,chemistry=88)
```

千万不要认为* 是指针，在python中没有指针的概念

函数名与重定义：

python中不支持函数重载，即不能有多个同名函数但参数列表不同。后定义的函数会覆盖先定义的函数

参数传递：

python的参数传递称为传对象引用

- 对于不可变对象：int、str、tuple等，在函数内部修改它们时，实际是创建了一个新的对象，并让局部变量指向这个新对象。原对象不受影响，类似传值
- 对于可变对象：list、dict、set等，由于函数内部和外部的变量指向同一个对象，因此在函数内部对该对象的任何 **原地修改**都会反映到外部变量上

因此如果要修改外部变量的值，应该使用return语句，如果想让函数返回多个值，可以return一个元组，再使用元组拆包的方式将这些值重新赋给外部变量

```
def process_data(num,text):
    """
    返回一个新的数字和新的字符串
    """
    new_num = num + 10
    new_text = "Modified:" + text
    return new_num,new_text

x = 5
y = "Original"
x,y = process_data(x,y)
```

输入输出

---

f-string

f-string是Python3.6引入的新方式，也是目前最常用的方式，具有简洁，可读性高，执行速度快的特点

- 基本使用：

```
name = "张三"
age = 30
print(f"你好，我的名字是{name}，我今年{age}岁。”)
```

- 也可以在{ }内进行表达式计算：

```
price = 19.9
quantity = 5
print(f"总价是：{price * quantity:.2f}元")  # .2f表示保留两位小数
# 输出：总价是99.95元
```

- 格式化对齐：
  - :<左对齐
  - :^居中对齐
  - :>右对齐
  - width指定宽度(跟在后面的数字)

```
print(f"|{'姓名':<10}|{'年龄':>5}|")
print(f"|{'李四':<10}|{25:>5}|")
# 输出：
```

```
# |姓名       |   年龄|
# |李四       |   25|
```

注意此处姓名等要加上''，否则会被编译为变量名导致错误

---

print( )

print( )会在每次调用后自动换行，可以传入一个或多个值，用空格隔开

- 基本用法：

```
age = 30
print("我的年龄是",age,“岁”)
# 输出：我的年龄是30岁
```

- 常用参数：

1. sep(separator)：用于指定多个值之间分隔符，默认是一个空格

```
print("apple","banana","orange")
# 输出：apple banana orange
print("2025","09","15",sep="-")
# 输出：2025-09-15
```

2. end(ending)：用于指定输出的结尾符，默认是换行符 \n，如果想让输出在同一行，可以使用 end=""或其他字符

```
print("第一部分",end=" ")
print("第二部分")
# 输出：第一部分 第二部分
```

---

input( )

input( )函数用于获取用户在命令行输入的数据，会暂停程序直到用户输入内容并按下回车

- 基本用法： input( )接收一个可选的字符串参数，作为提示信息显示给用户，并且**input( )返回的所有数据类型都是字符串（str）**

```
name = input("请输入你的名字：")
print("你好，" + name)    # str可以用 + 连接
```

- 类型转换：
  由于input( )返回的是字符串，如需处理数字，必须使用`int()`或`float()`函数进行类型转换

```
age_str = input("请输入你的年龄：")
# 将字符串转换为整数
age = int(age_str)
print("你明年就",age+1,"岁了。")
```

---

open（ ）

所有文件操作的第一步都是使用内置的open( )函数来创建一个文件对象

基本语法：`open(file,mode='r',encoding=None)`

- `file`：必需参数，文件的路径和名称
- `mode`：可选参数，指定打开文件的目的
- `encoding`：可选参数，指定文件编码格式，通常使用`utf-8`来处理中文等多种语言字符

常见的文件打开模式（mode）：

模式	描述	文件不存在时	文件存在时
`'r'`	读取（read），默认模式	报错	正常打开
`'w'`	写入（write）	创建新文件	清空文件内容，然后打开
`'a'`	追加（append）	创建新文件	在文件末尾追加新内容
`'t'`	文本模式（text），默认模式	N/A	N/A
`'b'`	二进制模式（binary），用于处理图片，音频等非文本文件	N/A	N/A
`+`	更新（update），与`'r'`、`'w'`、`'a'`结合使用，表示可读可写	N/A	N/A

以`r+`为例：

1. 文件必须存在，否则抛出FileNotFoundError错误
2. 文件打开后指针默认位于开头
3. 读写行为
   - 读取：可以从文件开头开始读取内容
   - 写入：从当前光标开始覆盖原有内容

文件操作中，非常重要的概念是资源管理，打开一格文件后，程序需要显式的关闭它来释放系统资源，如果忘记关闭或者程序在操作过程中报错，会导致资源泄露

`with`语句是解决这个问题的最佳方案，能确保文件在使用完毕后，无论是否发生异常，都会被**自动关闭**

相当于在with的作用域内进行安全操作

```
# 推荐写法：使用with语句
with open('example.txt','r') as file:
    content = file.read()
    print(content)
```

读写方法

一旦使用open( )创建了文件对象，就可以使用其方法来读写数据

读取文件

- `file.read(size)`：读取文件的所有内容或指定字节数，并作为一个**字符串**返回。这对于小文件很方便
- `file.readline()`：读取一行内容，返回一个**字符串**，每次调用会从当前位置读取一行，包括换行符
- `file.readlines()`：读取所有行，返回一个包含所有行的**列表**，但会一次性加载所有内容，容易导致内存溢出
- `for line in file:`：最推荐的迭代方式，懒加载，每次只加载一行到内存，效率高且节省内存

```
with open('data.txt','r',encoding='utf-8') as file
    for line in file:
        processed_line = line.strip()
        print(processed_line)
```

写入文件

- `file.write(string)`：将指定的字符串写入文件，但不会自动添加换行符，需手动添加`\n`
- `file.writelines(list)`：将字符串列表写入文件，同样不会自动添加换行符

```
# 写入字符串
with open('output.txt','w',encoding='utf-8') as file:
    file.write('这是第一行。\n')
    file.write('这是第二行。')

# 写入字符串列表
lines = ["第一行","第二行","第三行"]
with open("output.txt",'w'''encoding='utf-8') as file
    file.writelines(line + '\n' for line in lines)
```

读写模式

对于 `r+`，`w+` ，`a+`等模式，只需注意写入为覆写

如果有一个文件 `greeting.txt`，其内容如下：

```
Hello World!
```

利用 r+模式打开，并进行读写操作：

```
with open('greeting.txt','r+') as file:
    file.write("Hi")
    file.seek(0)
    content = file.read()
    print(content)
```

此时输出为：

```
Hillo World!
```

光标操作

Python提供tell( )和seek( )两个方法来精确调控光标

`file.tell()`：查看光标位置

- 返回一个整数，表示从文件开头算起的字节数

```
with open('example.txt,'w') as file:
    file.write("Hello")
    file.write("World")
    print(f"当前光标位置：{f.tell()}") # 输出：当前光标位置：10
```

`file.seek(offset,whence=0)`：移动光标位置

- `offset`：一个整数，代表要移动的字节数
- `whence`：一个整数，代表移动的参照点。最重要的参数

whence的三个值

whence值	符号常量	参照点	描述
0	os.SEEK_SET	文件开头	默认值，offset必须是正数或0
1	os.SEEK_CUR	当前位置	offset可以是正数（向后），也可以是负数（向前）
2	os.SEEK_END	文件结尾	offset通常为负数或0，从末尾向前移动

值得注意的是，在文本模式下，seek( )的offset参数只能是0或者由tell( )返回的值。这是因为不同编码中的字符所占字节数不同，python无法准确计算非零offset对应的字节位置

如果要精确的字节级别控制，须使用二进制模式（'b'）

```
with open('example.txt','rb') as file:
    file.read(5)   # 读取“Hello”，光标移动到第5个字节
    file.seek(3,1)    # 从当前位置向后移动3个字节
    content = file.read()
    print(content.decode())   # 输出：rld
```

decode( )方法的作用是将字节数据转换为可读的字符串，在二进制模式相下，read( )等方法返回的是字节对象，而非字符串，这时想看文件内容则需要使用decode( )

```
with open('greeting.txt','rb') as file:
    byte_data = file.read()
    # 此时，byte_data 的值是：b'\xe4\xbd\xa0\xe5\xa5\xbd'
    print(f"原始字节数据：{byte_data}")
    # 使用decode()解码
    string_data = byte_data.decode('utf-8')
    # 此时，string_data 的值是："你好"
    print(f"解码后的字符串：{string_data}")
```

encode( )作用与decode( )相反，将字符串转换为字节，成为编码

面向对象的编程

在python中，所有东西都是对象

- 类（class）：定义对象的属性（数据）和方法（功能）
- 对象（object）：类的实例

封装

类的基本结构：

- `class`：关键字，用于定义新类
- `__init__`：类的初始化方法，在创建类的实例时自动被调用，用于为对象的属性赋初始值
- `self`：类方法的第一个参数，代表实例本身，通过self可以在方法内部访问或修改实例的属性
- 实例属性：通过 `self.attribute_name`定义的属性，每个属性都有自己独立的属性副本

python的封装哲学与其他语言不同

- 没有 `private`等关键字，所有属性和方法都是默认公开的
- python的封装基于一种约定，通过命名来表明一个属性或方法是用于内部使用

1. 公有（public）：默认的，没有前缀
2. 保护（protected）：在属性或方法前加一个单下滑线 _，这是一个约定，告诉其他开发者不要在类的外部直接访问或修改它

3. 私有（private）：在属性或方法名前加双下划线 __这是一种特殊的机制，并不是真正的私有化，而是python对属性名进行的名称修饰，python解释器会自动将 __name重命名为 _ClassName__name，使得在外部无法直接访问

```
class Account:
    def __init__(self, name, sex):
        self._name = name
        self.__sex = sex
    def get_name(self):
        return self._name

account = Account("kekdou", "man")
print(account._name)  # 可以访问，但约定上不推荐
print(account.__sex)  # 这段代码会引发AttributeError
print(account._Account__sex) # 理论上可行，但不应该这样做
```

此外，类中还有许多特殊方法，在python中又称魔术方法（magic methods）或者Dunder方法（Double Underscore，双下划线的缩写），这是python中独特和强大的功能

能够让自定义对象像内置类型（如字符串、列表）一样工作，从而与python的内置函数和运算符无缝协作

- 对象表示
    - __str__(self)：返回对象非正式，可读的字符串表示，print( )函数会优先调用它

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __str__(self):
        return f"坐标点({self.x},{self.y})"

p = Point(1,2)
print(p)
```

- 运算符重载
    - __eq__(self,other)：定义想等性
    - __lt__(self,other)：定义小于
    - __add__(self,other)：定义加法
    - __len__(self)：定义len( )函数的行为

在python中，我们一般约定other代表另一个对象

```
class Vector:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __add__(self,other):
```

```
        return Vector(selx.x + other.x,self.y + self.y)
    def __eq__(self,other):
        return self.x == other.x and self.y ==other.y
    def __str__(self):
        return f"({self.x},{self.y})"

v1 = Vector(1,2)
v2 = Vector(3,4)
print(f"向量加法: {v1+v2}")
if v1==v2:
    print(f"v1 = v2")
```

- 容器行为
  - `__getitem__(self,key)`：定义获取元素时的行为（obj[key]）
  - `__setitem__(self,kry,value)`：定义设置元素时的行为（obj[key]=value）
  - `__iter__(self)`：定义迭代器的获取方式（for循环会自动调用）

继承

继承允许一个类（称为子类或派生类）从另一个类（称为父类、基类或者超类）继承属性和方法

在定义子类时，只需将父类的名字放在圆括号里

```
class Animal:
    def __init__(self,name):
        self.name = name
    def speak(self):
        print("动物发出声音。")

class Dog(Animal):
    def __init__(self,name,breed):
        # 调用父类的__init__方法，确保父类属性初始化
        super().__init__(name)
        self.breed = breed  # 子类特有属性
    # 方法重载
    # 子类可以提供自己的speak方法，覆盖父类的方法
    def speak(self):
        print(f"{self.name}说: 汪汪! ")

my_dog = Dog("papa","金毛")
my_dog.speak()  # 调用的是子类重载的speak方法
print(my_dog.name()) # 继承自父类的name属性
```

但格外需要注意的是，如果在子类中定义了`__init__`方法，它会覆盖父类的`__init__`，这意味着父类的初始化逻辑将不会执行，因此必须在子类的`__init__`方法中显式的调用父类的`__init__`

`super()`函数是一个非常重要函数，它能够动态的调用MRO（方法解析顺序，Method Resolution Order）链中下一类的方法

在python中没有虚继承的概念，因为MRO机制从根本上解决了这个问题

在初始化问题上，如果不使用`super()`，而是使用`ParentClass.__init__()`，容易导致父类的 `__init__`重复调用，从而引发逻辑错误或不必要的开销

并且在多重继承的情况下，如果不同父类含有相同的方法，使用 `super()`会按照MRO顺序分别调用一次

```
class A:
    def greet(self):
        print("Hello from A")
        super().greet()

class B(A):
    def greet(self):
        print("Hello from B")
        super().greet()

class C(A):
    def greet(self):
        print("Hello from C")
        super().greet()

class D(B, C):
    def greet(self):
        print("Hello from D")
        super().greet()

# 查看 D 的方法解析顺序
print("D 的方法解析顺序 (MRO):", D.mro())
print("-" * 30)

# 调用 D 实例的 greet 方法
d = D()
d.greet()
```

输出结果：

```
D 的方法解析顺序 (MRO): [<class '__main__.D'>, <class '__main__.B'>, <class
'__main__.C'>, <class '__main__.A'>, <class 'object'>]
------------------------------
Hello from D      # D.greet()
Hello from B      # super() 在 D 中调用 B.greet()
Hello from C      # super() 在 B 中调用 C.greet()
Hello from A      # super() 在 C 中调用 A.greet()
```

多态

多态指的是不同类的对象可以对同一个方法调用做出不同的响应

```python
class Animal:
    def speak(self):
        raise NotImplementedError("子类必须实现speak方法")

class Dog(Animal):
    def speak(self):
        return "汪汪! "

class Cat(Animal):
    def speak(self):
        return "喵喵~"

def animal_sound(animal):
    """
    这个函数能够处理任何实现了speak()方法的对象
    它不关心对象的具体类型
    """
    print(animal.speak())
# 创建不同类型的对象
dog = Dog()
cat = Cat()
# 同一个函数，传入不同类型的对象，得到不同的结果
print("------")
animal_sound(dog)  # 输出：汪汪!
animal_sound(cat)  # 输出：喵喵~
```