变量

变量就是给程序中的数据，对象等取一个简单易懂的名字，一般用英文名和下划线组成，尽可能清晰表示变量的意义。

命名注意点：

1. 变量名只能包含字母、数字和下划线。
2. 变量名可以字母或下划线打头，但不能以数字打头，例如，可将变量命名为 message_1，但不能将其命名为 1_message。
3. 变量名不能包含空格，但可使用下划线来分隔其中的单词。例如，变量名 greeting_message 可行，但变量名 greeting message 会引发错误。
4. 不要将 Python 关键字和函数名用作变量名，即不要使用 Python 保留用于特殊用途的单词，如 print，class 等。

数据结构

程序的本质就是使用代码对数据进行操作，代码是对数据操作的逻辑，数据在高级代码语言里需要有一定的组织结构，Python 中最基础常用的几种就是：

字符串 str：

用引号（单双引号均可）括起来的部分，可以描述自然语言；str(参数)可以将参数如数字等转化为字符串。

列表 list：

用中括号括起来的部分，中间每一项用逗号（,）隔开，每一项的数据类型都可以不同，按特定顺序排列；

元组 tuple：

用小括号括起来的部分，中间每一项也是用逗号（,）隔开，

字典 dict：

用大括号括起来的部分，每一项包括 2 个子项，键 key 和值 value，键一般是字符串，键值对用冒号（：）分开，每一项也是用逗号（，）分开。

对数据结构进行的最基本的操作就是在创建之后的增删查改（不可变对象如字符串和元组的增删改都是返回新的对象），增就是添加，删就是删除，查就是获取对象，包括单个和多个元素的获取（切片），改就是修改，下面具体地介绍一下各个数据结构的基本操作。

对应数据结构的基本操作方法，直接使用名称.方法名（参数）即可。

字符串

注意字符串是不可变对象，增删改后返回的是新的字符串；即使没有实际改变，也会创建新对象。

创建：使用引号括起来的部分为字符串，可以为中文，英文，数字等符号。

str1='　　string'

join（）方法：在实际项目中创建字符串常用的一个方法是 join（），用于将序列（如列表、元组、字符串、字典、集合等）中的元素以指定的字符串连接起来生成一个新的字符串。

语法：str.join(sequence)，序列中的元素必须是字符串类型。如果序列中包含非字符串元素，则会抛出 TypeError。

与使用 + 操作符进行字符串拼接相比，join() 方法在性能上具有显著优势，特别是在处理大量字符串时。

由于字符串是不可变对象，每次使用 + 操作符都会创建一个新的字符串对象。而 join() 方法只需要在最后创建一次新字符串，大大减少了内存分配和复制操作。

```
import time
# 使用 + 操作符拼接（效率低）
start_time = time.time()
result = ""
for i in range(10000):
    result += str(i)
end_time = time.time()
print(f"使用 + 操作符耗时: {end_time - start_time:.4f} 秒")
# 使用 join() 方法拼接（效率高）
start_time = time.time()
parts = []for i in range(10000):
    parts.append(str(i))
result = "".join(parts)
end_time = time.time()
print(f"使用 join() 方法耗时: {end_time - start_time:.4f} 秒")
```

增（添加）：直接使用加号（+）来合并添加字符串，

```
str2=str1+'    abc    '
```

删（替换）：使用 replace() 方法删除特定字符或子串

```
text = "Hello, World!"# 删除逗号
new_text = text.replace(",", "")
print(new_text)  # 输出: Hello World!
```

rstrip()剔除字符串末尾的空格，lstrip()剔除字符串开头的空格，strip()同时剔除字符串两端的空格。也可以指定删除特定字符。

```
str5=str4.lstrip()
str6=str4.rstrip()
str7=str4.strip()
```

查（获取单个元素）：使用字符串名加下标 str[i]即可。

```
str_single=str1[5]
```

查（切片）：同时获取多个元素，名称加区间索引值 str[i:j]，实际获取的数据区间是左闭右开，要输出列表中的前三个元素，需要指定下标索引 0~3。

```
str_split=str1[0:5]
```

查（遍历）：

数据结构都是可迭代对象（后面详细介绍），可以使用 for 循环进行遍历取值 for str in str1:

print(str)

改：字符串不能直接使用下标更改特定值，只能使用切片拼接的方法。

```
text = "hello"
# 尝试修改字符串中的字符会引发错误
try:
    text[0] = "H"  # 这会引发 TypeError
except TypeError as e:
    print(f"错误: {e}")
# 正确的方法是创建新字符串#
方法 1: 使用切片和连接
new_text = "H" + text[1:]
print(f"方法 1 结果: {new_text}")
# 方法 2: 转换为列表再转回字符串
text_list = list(text)
text_list[0] = "H"
new_text = "".join(text_list)
print(f"方法 2 结果: {new_text}")
```

upper()将字符串改为全部大写;lower()全部小写，title()将每个单词的首字母都改为大写。

```
str3=str2.upper()
str4=str3.lower()
```

求长度：直接使用 len()函数

```
str_size=len(str1)
```

列表

列表中的元素不需要是同一种数据类型，可以创建包含数字，字符串，列表，元组等的列表；也可以将任何类实例化的对象加入列表中，其中的元素之间可以没有任何关系。

列表非常适合用于存储在程序运行期间可能变化的数据集。列表是可以修改的

创建：一般的创建方式有 2 种，一种是创建一个空列表，在实际操作中不断添加；另一种是直接把已知的数据内容创建出来，如下：

```
list1=[]
list2=[1,'a',list1]
```

列表解析创建

在实际项目中另一种经常使用的列表创建方法就是解析创建，一般方法生成列表 squares 的方式包含三行代码，

```
squares = []
for value in range(1,11):
    squares.append(value**2)
```

而列表解析只需编写一行代码就能生成这样的列表。列表解析将 for 循环和创建新元素的代码合并成一行，并自动附加新元素。

```
squares = [value**2 for value in range(1,11)]
```

要使用这种语法，首先指定一个描述性的列表名，如 squares；后面使用中括号括起来，里面先定义一个表达式（如 value**2 用于计算平方值），用于生成要存储到列表中的值。接下来，编写一个 for 循环（for value in range(1,11)），用于给表达式提供变量。请注意，这里的 for 语句末尾没有冒号。

转化列表：除了直接创建，实际操作中还经常进行数据类型间的转化，方法很简单，即用 list（var）

```
str1='abc'
list3=list(str1)
```

转换数值列表

range(a,b)来生成[a,b)区间的数字（左闭右开）；使用函数 list(range(a,b))将 range(a,b)的结果直接转换为列表。

使用函数 range()时，还可指定步长。list(range(2,11,2))，在这个示例中，函数 range()从 2 开始数，然后不断地加 2，直到达到或超过终值（11）。

对数字列表执行简单的统计计算

有几个专门用于处理数字列表的 Python 函数。例如，数字列表的最大值 max(lists)、 最小值 min(lists)和总和 sum(lists)：

添加：在列表末尾 ：append()将元素动态地附加到列表末尾。即先创建一个空列表，再使用一系列的 append()语句添加元素。

```
list1.append(1)
```

在列表中插入元素：insert()可在列表的任何位置添加新元素。只需要指定新元素的索引和值。索引后面的列表值都将右移一个单位。

获取单个元素:列表名称+[下标]

```
temp=list1[1]
```

列表切片：

同时获取多个元素，变量名后加区间索引值 list[i:j]，实际获取的数据区间是左闭右开，要输出列表中的前三个元素，需要指定索引 0~3，这将输出分别为 0、1 和 2 的元素；

```
list_split=list2[0:2]
```

一般这种区间的都是左闭右开，如 range(i,j)

（1）省略第一个索引 i，Python 将自动从列表开头开始;省略终止索引 j，切片终止于列表末尾;

（2）负数索引返回离列表末尾相应距离的元素，例如，如果要输出列表的最后三个元素，可使用切片 lists[-3:];

（3）要复制列表，可创建一个包含整个列表的切片，方法是同时省略起始索引和终止索引（[:]）。 这让 Python 创建一个始于第一个元素，终止于最后一个元素的切片，即复制整个列表，而且这两个列表是独立的。

遍历列表：

数据结构都是可迭代对象（后面详细介绍），可以使用 for 循环进行遍历取值，

for list in lists

如果要遍历部分元素，可在 for 循环中使用切片。使用单数和复数式名称，可帮助判断代码段处理的是单个元素还是整个数据结构。

修改列表元素：指定列表名和要修改的元素的索引，再指定该元素的新值。

list2[0]=10

删除列表元素

del 语句：可删除任何位置的列表元素，条件是知道其索引。del list[0]

pop()可删除列表末尾的元素，并可以用变量接收以接着使用，

list0=list2.pop()

术语弹出（pop）源自这样的类比：列表就像一个栈，而删除列表末尾的元素相当于弹出栈顶元素。实际上，你可以使用 pop()来删除列表中任何位置的元素，只需在括号中指定要删除的元素的索引即可。

remove()：根据值删除元素，若不知道要从列表中删除的值所处的位置。只知道要删除的元素的值，可使用方法 remove()，list.remove(value1)

求列表长度：

直接使用 len()函数，len(list)

列表排序

使用方法 sort()对列表进行永久性排序 ，使用完之后列表中元素的顺序永久改变。

使用函数 list.sorted()对列表进行临时排序，可以保留列表元素原来的排列顺序，同时以特定的顺序呈现它们。

逆序：方法 list.reverse()永久性地修改列表元素的排列顺序，但可随时恢复到原来的排列顺序，为此只需对列表再次调用 reverse()即可。

元组

Python 将不能修改的值称为不可变的，而不可改变单元素的列表被称为元组，使用小括号来标识。元组内的元素也可以是任何数据类型。因此，普通创建，切片，遍历方法都是相同的，但是添加，删除，更改等变动元组内部元素的情况是不可行的。元组在深度学习中经常用于定义数据张量的形状。

创建元组：直接在小括号内写上元素或空。

tuple1=()

tuple2 = (False,3+4j,“aaa”,456)

获取元素:元组名称+[下标]

temp=tuple1[1]

元组切片：

同时获取多个元素，名称后加区间索引值 tuplet[i:j]，实际获取的数据区间是左闭右开，要输出列表中的前三个元素，需要指定索引 0~3，这将输出分别为 0、1 和 2 的元素；

tuple_split=tuple1[:2]

重定义元组

虽然不能修改元组的元素，但可以给存储元组的变量赋值，即对同一个变量重新定义整个元组；如下，同一个变量名称重新赋值。

tuple3=(1,2,3)
tuple3=(4,5,6)

字典

字典是大括号括起来的一系列键值对，键值之间用冒号隔开，键值对之间用逗号隔开。

创建：直接使用大括号，键可以为字符串，数值，元组，但不可以为列表和字典（非哈希类型），值可以为任意数据类型。

```
dict1={'abc':'value1',1:[1,2],(1,2):{ }}
```

添加:只需要 dict[键 key]=value。

```
dict1['key']='value2'
```

删除：Python 中的字典（dict）提供了两种删除元素的方法：popitem() 和 pop()，两种方法都会从字典中移除指定的元素，并返回被删除的值。

```
del1=dict1.pop(1,None)
print('del1:',del1)
del2=dict1.popitem()
print(('del2:',del2))
## 键不存在时，报 KeyError: 2
# del3=dict1.pop(2)
```

二者的主要区别为：

popitem()	pop(key,default)
无需参数	必须指定键（key）参数
删除**最后插入**的元素（有序）	删除**指定键**对应的元素
若字典为空，抛出 KeyError	若键不存在：- 无默认值时抛出 KeyError - 有默认值时返回默认值

获取：名称后加 key 值 dict[key]，注意字典的 key 是什么数据类型，中括号内也要写什么数据类型（如字符串，元组等）

```
temp=dict1[(1,2)]
```

遍历：

数据结构都是可迭代对象（后面详细介绍），可以使用 for 循环进行遍历取值，for key,value in dict.itms()：注意这时字典是同时取得键值对的。

单独遍历键

for key in dic.keys() 可省略 keys（）

单独遍历值

for value in dic.values()

更新 update：

update()方法用于更新字典，使用方式为字典名.update(dict)，传入参数即为另外一个字典或键值对的可迭代对象。将它们的元素添加到当前字典中。 如果键存在，则更新其值；如果键不存在，则添加键值对。

```
# 基本使用，参数为另一个字典
dict1 = {'name': 'Alice', 'age': 25}
dict2 = {'age': 26, 'city': 'New York'}

dict1.update(dict2)
print("更新后的字典:", dict1)  # {'name': 'Alice', 'age': 26, 'city': 'New York'}
#键存在，更新值；不存在，添加键值对；

# 使用关键字参数
dict3 = {'a': 1, 'b': 2}
dict3.update(c=3, d=4)
print("使用关键字参数更新:", dict3)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# 使用可迭代对象
dict4 = {'x': 1}
dict4.update([('y', 2), ('z', 3)])
print("使用可迭代对象更新:", dict4)  # {'x': 1, 'y': 2, 'z': 3}
```

共性操作

1.元素获取：

列表，字符串，元组都是有序结构，因此可以直接使用名称后加下标索引 str[i]&&list[i]&&tuple[i]，str/list/tuple 为数据结构变量名，i 为下标；第一个元素的索引为 0，而不是 1。

访问最后一个元素可通过将索引指定为-1，倒数第 2 个索引为-2，以此类推。

字典则是使用名称后加 key 值 dict[key]。

实战小技巧：

下标索引和 key 值都是用中括号。这里一个值得注意的点：像这种获取元素，赋值等数据结构的操作一般使用中括号[]，而对于函数名后的传参操作等一般使用小括号()。

if 语句

if 语句是一个条件测试，满足则执行 if 下的语句，不满足则跳过，一般用一个等式来表示。若要检查多个条件，仅需把多个条件用 and 连接，则所有条件都需要满足才为 true。用 or 则是只需要一个条件满足即可。

检查特定值是否在列表里，可以用（not）in。

if else 结构

if elif else 结构

while 循环

while 循环 直到不满足条件时才退出。break 直接退出循环。continue 退出本次循环，回到循环开头。

函 数

函数是把完成特定功能的代码块封装起来，取个名字，以方便其它代码直接调用，就不用重复编写完成该任务的代码了。每个函数都应只负责一项具体的工作，这优于使用一个函数来完成两项工作。

函数有 2 个重要的知识点，首先学习的是参数的传递，第二个就是函数存储到模块里，尤其是在类里（改名为方法）。

函数定义

def function_name():

关键字 def 说明要定义一个函数。

function_name 为函数名，根据完成的任务取一个合适的名字；

括号内为需要传递的参数，指出函数为完成其任务需要什么样的信息。即使不需要任何信息，括号是空但也必不可少。

最后，定义以冒号结尾，紧跟在后面的所有缩进行构成了函数体。

函数调用与返回值

函数并非总是直接显示输出，或者只在函数体内完成一些操作，相反，它可以处理一些数据，并返回一个或一组值。函数返回的值被称为返回值，调用函数时用变量承接。在函数中，可使用 return 语句将值返回到调用函数的代码行。返回值让你能够将程序的大部分繁重工作移到函数中去完成，从而简化主程序。

参数传递

位置参数（Positional Arguments）和关键字参数（Keyword Arguments）

鉴于函数定义中可能包含多个形参，因此函数调用中也可能包含多个实参。向函数传递实参的方式很多，那么这些参数有什么区别吗？

是有的，我们想一下，事实上函数传递实参的目的是为了让形参接收到对应的实参，也就是说需要保证实参和形参是一一对应的，而为了保证这一点，函数设计了 2 种类型的参数，分为位置参数和关键字参数（可理解为列表的按下标和字典的键值对来确认对应关系）。

位置参数是在函数定义时按照顺序依次列出的参数，调用函数时，传递的参数必须按照定义时的顺序一一对应，参数的位置决定了它们会被赋值给哪个形参。位置参数有 2 个特性：

- **顺序性**：调用函数时，实参的顺序必须和形参的顺序一致。
- **数量匹配**：传递的实参数量必须和形参数量相同（除非有默认值）。

```
def add_numbers(a, b):
    return a + b
# 调用函数，按照位置传递参数
result = add_numbers(3, 5)
print(result) # 输出: 8
```

从上面的代码可以看出来，位置参数可以省略实参的变量名称，直接按照顺序和形参一一对应。

关键字参数是在调用函数时，通过指定参数名来传递参数的方式，参数名和值之间用等号连接。使用关键字参数时，参数的传递顺序可以和函数定义时的顺序不同。

- **顺序无关性**：可以不按照函数定义时的参数顺序传递参数，只要指定了参数名即可。
- **可选择性**：可以只传递部分参数，前提是其他参数有默认值。

```
def describe_person(name, age, city):
    print(f"{name} 今年 {age} 岁，住在 {city}。")
# 使用关键字参数调用函数
describe_person(age=25, city="New York", name="Alice")#调用顺序可以不同
```

结合使用位置参数和关键字参数

在**调用函数**时，可以同时使用位置参数和关键字参数，但位置参数必须放在关键字参数之前。

```
describe_person("Alice",age=25, city="New York" )#位置参数在前
```

默认参数

调用函数时候传递的实参数量是不是一定和形参一致呢？答案是否定的。看一些源码的时候，经常可以注意到，为了保证尽可能的全面，内部函数在定义时候会设置各种各样的情况，不同的情况对应不同的参数，但是在调用函数时候，往往并不需要传递相同数量的参数，这是因为内部函数定义时候给很多形参指定了默认值。

编写函数时，可给每个形参指定默认值。

```
def fun3(a,b=2):
```

在调用函数中给形参提供了实参时，Python 将使用指定的实参值；否则，将使用形参的默认值。因此，给形参指定默认值后，可在函数调用中省略相应的实参。使用默认值可简化函数调用，还可清楚地指出函数的典型用法。

提供的实参多于或少于函数完成其工作所需的信息（不一定是形参数，要看是否有默认值）时，将出现实参不匹配错误。这也是经常出现的错误。

***args 和**kwargs**

前面强调实参和形参需要一一对应，或者通过位置，或者通过关键字，那如果预先不知道函数需要接收多个实参怎么办呢？简单，Python 给函数提供了 2 个可装任意数量实参的大包，并分别起了个名字，接收位置参数的大包命名为 *args（其实就是空元组），接收关键字参数的大包命名为**kwargs（其实就是空字典）。args 是约定俗成的名字，你可以用其他名字（如 *numbers），但星号 * 是必须的。kwargs 同理。

那么函数在定义时就变成了下面这样：

def func2(a,b,*args,**kwargs):

问题又来了，这些参数没有名字在函数体内部怎么使用呢？

就是把*args 和**kwargs 当做元组和字典去遍历，得到所有的参数。位置实参使用形参*args，星号让 Python 创建一个名为 args 的空元组，并将收到的所有值都封装到这个元组中。关键字实参使用形参**kwargs，两个星号让 Python 创建一个名为 kwargs 的空字典，并将收到的所有名称—值对都封装到这个字典中。

*args 使用实例

```
def sum_numbers(*args):
    total = 0
    for num in args:
        total += num
    return total
# 调用时可以传递不同数量的位置实参
result1 = sum_numbers(1, 2, 3)
result2 = sum_numbers(10, 20, 30, 40)
result3 = sum_numbers(5)
```

**kwargs 的 2 种使用实例

```
def print_info(**kwargs):
    for key, value in kwargs.items():
    print(f"{key}: {value}")
# 调用函数并传入不同的关键字参数
print_info(name="Alice", age=25, city="New York")
print_info(product="Laptop", price=1000, brand="Dell")
```

还有一种常用的使用**kwargs 参数的方法是结合 setattr 给对象属性赋值

```
def setattr_with_kwargs(object,**kwargs):
    for k, v in kwargs.items(): #用于设置属性值，该属性不一定是存在的。
        setattr(object, k, v)  实例说明一下，输出结果
```

setattr(object, k, v) 是 Python 的内置函数，它的作用是给对象 object 设置一个属性，属性名是 k，属性值是 v。如果该属性原本不存在，就会创建这个新属性；如果原本存在，就会更新其值。

函数定义时的参数顺序规则

前面主要介绍的是调用函数时候的实参类型，为了配合调用函数时候传递参数的不确定性，在函数定义时，需要设定参数的顺序，一般为：位置参数、默认参数、*args（可变位置参数）、关键字参数（需指定参数名）、默认值、**kwargs（可变关键字参数）。注意，*args 必须放在 **kwargs 之前，且不能交叉传递。*args 之后的参数自动变为关键字参数。

在函数定义中，* 或 *args 之后的参数，无论是否有默认值，都是 “关键字参数”（强制用关键字传递）；其中没有默认值的，就是 “无默认值的关键字参数”。

如果要让函数接受不同类型的实参，必须在函数定义中将接纳任意数量实参的形参放在最后。Python 先匹配位置实参和关键字实参，再将余下的实参都收集到最后一个形参中。

```
def complex_function(a, b=2, *args, c, d=10, **kwargs):
    print(f"a = {a}")
    print(f"b = {b}")
    print(f"args = {args}")
    print(f"c = {c}")
    print(f"d = {d}")
    print(f"kwargs = {kwargs}")

# 调用示例
complex_function(1, 3, 4, 5, 6, d=20, e='extra', f=42)
```

形参与实参

def fun（arg）

定义函数 func(arg)时，括号里的变量 arg 是一个形参，没有真实的信息，只是作为一个符号用于函数体内；

那么真正的参数信息在哪里呢？就是在调用函数时候括号里的变量，函数调用时，直接函数名+括号内参数（若函数有返回值则前面加个变量进行接收）。

```
real_arg=1
func(real_arg)
```

变量‘real_arg’是一个实参，是真正存储信息的。

我们调用函数时，在 function(‘real_arg’)中，将实参‘real_arg’传递给了函数 function(arg)，这个值被存储在形参 arg 中。

二者的对比如下：

作用域：

- 形参：作用域在函数内部，是函数的局部变量。
- 实参：可以在函数外部定义，作用域根据定义的位置而定。

存在时间：

- 形参：在函数被调用时创建，函数执行结束后销毁。
- 实参：在函数调用前就已存在（除非是字面量），函数调用后通常仍然存在（除非在函数内部被修改且是可变对象，但实参变量本身不会销毁）。

绑定方式：

- 形参：在函数定义时只是占位符，没有具体的值。
- 实参：必须具有确定的值（可以是表达式的结果），该值会被传递给形参。

形参能否影响实参

在 Python 中，函数的参数传递是通过“对象引用”进行的。这意味着当你传递一个参数给函数时，实际上传递的是对象的引用（即内存地址），而不是对象本身的副本。因此，函数内部对该引用所指向对象的操作可能会影响原始对象，也可能不会，这取决于对象的类型（可变或不可变）以及你进行的操作。

1. 不可变对象：包括整数、浮点数、字符串、元组等。当传递不可变对象时，函数内部对形参的重新赋值不会影响实参，因为不可变对象不能被修改。如果尝试修改，实际上是创建了一个新的对象。

```
    #传递不可变对象（不会影响原值）
def unchange_obj(num,str1):
    num = 10 #  重新赋值，创建了一个新的整数对象
    str1='new-string'
    print("Inside function:", num,str1)
x = 5
orig_str='abc'
unchange_obj(x,orig_str)
print("Outside function:", x,orig_str)
#  输出: 5,abc
```

2. 可变对象：包括列表、字典、集合等。当传递可变对象时，函数内部对形参的修改（修改了传入的可变对象的内部状态，如修改列表元素、添加/删除元素）会直接影响实参，因为形参和实参引用的是同一个对象。但如果你在函数内部将形参重新绑定到一个新的对象（重新赋值），那么原始对象也不会被改变。因为此时形参指向了新的对象，而实参仍然指向原来的对象。

```
    #  传递可变对象（会影响原值）
    def change_list(my_list):
        my_list.append(4) #  修改列表，添加一个元素
        print("Inside function:", my_list)
    lst = [1, 2, 3]
    change_list(lst)
    print("Outside function:", lst)
    #  输出: [1, 2, 3, 4]


    #传递可变对象但重新赋值（不影响原值）
    def reassign_list(my_list):
        my_list = [4, 5, 6] #  重新赋值，指向新的列表
        print("Inside function:", my_list)
    lst = [1, 2, 3]
    reassign_list(lst)
    print("Outside function:", lst)
```

```
# 输出: [1, 2, 3]
```

函数传递列表很有用，这种列表包含的可能是名字、数字或更复杂的对象（如字典）。将列表传递给函数后，函数就能直接访问其内容，可对其进行修改。在函数中对这个列表所做的任何修改都是永久性的，这让你能够高效地处理大量的数据。

若要禁止函数修改列表，可向函数传递列表的副本而不是原件；这样函数所做的任何修改都只影响副本，而丝毫不影响原件。切片表示法[:]创建列表的副本。传递任意数量的实参。

默认参数的默认值

函数的默认参数在定义时只被求值一次，因此如果默认参数是可变对象，多次调用函数可能会共享同一个可变对象，导致意外的行为。因此，通常建议默认参数使用不可变对象，如果必须使用可变对象，可以在函数内部用 None 作为默认值，在函数内部增加一个判断条件，若为 none，则新建一个空对象，相当于每次调用都会创建一个新的对象。

```
    # 不好的做法
def bad_append_to(element, target=[]):
    target.append(element)
    return target
# 好的做法
def good_append_to(element, target=None):
    if target is None:
        target = []
    target.append(element)
    return target

bad_target1=bad_append_to(1)
print('bad_target1:',bad_target1) # [1]
bad_target2=bad_append_to(2)
print('bad_target2:',bad_target2) #[1,2]

good_target1=good_append_to(1)
print('good_target1:',good_target1) #[1]
good_target2=good_append_to(2)
print('good_target2:',good_target2) # [2]
```


将函数存储在模块中

函数的优点之一是，使用它们可将代码块与主程序分离。通过给函数指定描述性名称，可让主程序容易理解得多。还可以更进一步，将函数存储在被称为模块的独立文件中，再将模块导入到主程序中。import 语句允许在当前运行的程序文件中使用模块中的代码。

这点在后面的进阶版进一步讨论。

类和对象

类和对象是 Python 编程的核心构件，类有 2 个核心的组件，一是变量，类中称为属性，二是函数，类中称为方法，属性就是变量，方法就是函数，只是在类里重新起了个名字。

类可以理解为更高层次的函数，把某一类对象都有的通用行为抽象出来，比如猫科动物类；对象是类的实例化，比如实例化一个猫。一个类可以通过继承的方式在保留原类的不断扩展独有的一些特点。

在 Python 中，类在定义时，使用 class 关键字，类名首字母大写。类名后可以加括号，括号可以为空。不为空的话一般是写继承的基类名。

属性

基本概念

前面介绍类的属性就是变量，那么属性有哪些种类呢？又该如何赋值呢？

属性种类：

属性包括类属性和实例属性，类属性就是所有实例都有的，在类的一级结构下，可通过类名和实例名访问，实例属性在 init 初始化方法里，只能通过实例名访问。实例属性又包括私有属性，前面加双下划线，只能在类的方法中访问，实例名无法访问。

类的实例也可以作为属性，当给类添加的细节越来越多：属性和方法清单以及文件都越来越长。在这种情况下，可能需要将类的一部分作为一个独立的类提取出来。然后把这个独立的类作为大类的一个属性。

属性赋值：

类属性在类定义时会赋值（默认值），类的所有实例都会共享这个初始值；

实例属性在_init__方法里赋值，self.instance_attirbute=0。 值的来源有 2 种 ，一是直接给定初始值，可以为 0 或者空字符串等；二是使用_init__括号里的形参（也就意味着实例属性可以和括号里的形参数量不一致。）但我们知道，形参只是符号，没有具体值，真正的值来自调用函数传过来的实参。

那么怎么调用__init__方法呢？

答案就是在类的实例化时，__init__方法会自动调用，因此实例化时，类名括号里的参数应该和__init__方法的形参一致（有默认值的可不用）。如果有继承，那么也需要包括父类的__init__方法的形参。

类的属性和方法的参数的比较：

特征	属性（Attribute）	方法参数（Parameter）
定义位置	类内部直接定义或通过__init__初始化	方法定义时的括号内声明
生命周期	随对象存在，直到对象销毁	仅在方法调用期间存在
数据流向	存储对象状态数据	传递外部数据到方法内部
访问方式	通过 self.属性名访问	在方法内直接使用参数名访问
修改权限	通常通过方法间接修改	作为输入数据不可直接修改（除非是

特征　　属性（Attribute）　　方法参数（Parameter）

可变对象）

实例属性是在实例化时确定的，能够根据实例化时传递的参数给不同的实例赋予不同的属性值，灵活性较高；而类属性的初始值在类定义时就确定了，可以当做常量和默认值使用。

属性访问和修改：

这 2 种属性都可以通过实例名访问和动态修改，类属性还可以直接通过类名访问修改，使用句点表示法。实例名.属性；

类属性修改所有的实例对象访问得到的结果都会随着修改；而实例属性的话，每个实例都有属于自己的属性副本，一个实例对象修改不会影响其他实例对象的访问结果。

可以以三种不同的方式修改属性的值：

直接通过实例进行修改（不推荐，不安全）；

通过方法进行设置（推荐）；

通过方法进行递增（增加特定的值）。

代码实战

```
    # 类属性和实例属性
class BaseClass:
    # 类属性
    class_attribute='这是类属性'

    def __init__(self,instance_attribute):
        # 实例属性
        self.instance_attribute1=instance_attribute  # 通过形参赋值，类实例化时传入
        self.__instance_attribute2=0  # 可以直接初始化，不用通过形参赋值,且是私有属性，实例名和类名不能直接访问


# 类实例化，类名后的参数需要和__init__方法的形参一致
obj1=BaseClass(instance_attribute='instance_value1')
obj2=BaseClass(instance_attribute='instance_value2')

# 类属性访问,包括类名和实例名
print('类名访问类属性：',BaseClass.class_attribute)
print('实例名访问类属性：',obj1.class_attribute)
print('实例名访问类属性：',obj2.class_attribute)

# 实例属性访问
# print('类名访问实例属性：',BaseClass.instance_attribute1) # 报错，类名不能访问实例属性
print('实例名访问实例属性：',obj1.instance_attribute1)
```

```
print('实例名访问实例属性：',obj2.instance_attribute1)
# print('实例名访问实例属性：',obj2.instance_attribute2) # 报错，私有属性不能直接访问

# 类属性修改
BaseClass.class_attribute='类直接修改后的类属性'
print('实例名访问修改后的类属性：',obj1.class_attribute) # 输出：类直接修改后的类属性
obj1.class_attribute='实例直接修改后的类属性'
print('另一个实例名访问实例修改后的类属性：',obj2.class_attribute)  # 输出：类直接修改后的类属性

# 实例属性修改
obj1.instance_attribute1='实例修改后的实例属性'
print('实例名访问修改后的实例属性：',obj1.instance_attribute1)  # 输出：实例修改后的实例属性
print('另一个实例名访问修改后的实例属性：',obj2.instance_attribute1)  # 输出：instance_value2（实例属性没变）
```

方法

类中的函数称为方法；有关函数的一切都适用于方法，就目前而言，唯一重要的差别是调用方法的方式，即需要在调用的方法前面加实例/对象名，以表明是哪个实例对象调用的，因为同一个方法名不同的实例可能实现的不一样。

方法种类：

方法又包括基本方法，初始化方法__init__，类方法@classmethod，静态方法@staticmethod，基本方法和初始化方法有第一个隐参数 self，类方法有第一个隐参数 cls，静态方法无隐参数，不需要访问实例属性。

1. __init__() 初始化方法

__init__()是一个特殊的初始化方法，每当根据类创建新实例时，Python 都会自动运行它。在这个方法的名称中，开头和末尾各有两个下划线，这是一种约定，表示类私有。

方法__init__()定义中，形参 self 必不可少，还必须位于其他形参的前面。为何必须在方法定义中包含形参 self 呢？因为 Python 调用这个__init__()方法来创建实例时，将自动传入实参 self。每个与类相关联的方法调用都自动传递实参 self，它是一个指向实例本身的引用，让实例能够访问类中的属性和方法。

2.基本方法

类中的基本方法的第一个形参都是 self，可以有其它的形参，若有，则调用时需要按照普通函数那样传入对应的实参；类中调用基本方法：self.+名字。类外调用：实例名.方法名。

3.类方法

方法上面使用@classmethod 装饰器表示，第一个参数为 cls,使用类名.方法名进行调用，也可以通过**类的实例**来调用。更常见和推荐通过类名调用。

可以访问和修改**类属性**（cls.class_attribute）。**不能**直接访问或修改特定实例的属性（因为没有 self 指向实例）。

4.静态方法

方法上面使用@staticmethod 装饰器表示，参数只需要正常形参即可,使用类名.方法名进行调用。

定义**独立于类和实例状态**的辅助功能，**不能**直接访问或修改类属性或实例属性（因为没有 cls 或 self）。如果需要操作数据，必须通过参数显式传递。本质上就是一个放在类里面的普通函数，仅仅因为逻辑上属于这个类而放在这里。比如执行与类相关但不依赖于类或实例具体状态的辅助计算、格式化、验证等。

5.抽象方法

方法上面使用@abstractmethod，需要继承抽象类 ABC。具体见下面的抽象继承。

继承

如果要编写的类是另一个现成类的特殊版本，可使用继承。类名括号内写另一个类的名字，即为继承，括号内的类为父类/基类，新建的为子类。子类**自动获得父类的属性和方法**，并可通过扩展或修改实现新功能。

```
class Parent:                    # 父类/基类
    pass
class Child(Parent):          # 子类/派生类 (Parent 写在括号中)
    pass
```

继承的功能意义

继承就是子类默认拥有基类完整的属性和方法，然后又可以改动和新增自己的属性和方法，还可以重写父类的方法，只需要与要重写的父类方法同名。这样子类实例化对象后直接调用子类的方法而不是父类的方法，若没有重写，则仍调用父类的方法。这点在实际项目中经常遇到，子类继承父类之后，实例化对象调用的方法在子类的代码里没找到，大概率就是父类有这个方法，子类未重写。

注意一个特殊情况，即子类实例调用父类的其中一个方法 A，该方法调用父类的另一个方法 B，而子类重写了方法 B，那么子类实例调用方法 A 的时候最终也会调用子类的方法 B。这就是类的多态性。

```
class Parent:
    def methodA(self):
        print("Parent: methodA called")
        self.methodB()   # 此处实际调用子类重写的 methodB
    def methodB(self):
        print("Parent: methodB called")
class Child(Parent):
    def methodB(self):   # 重写父类的 methodB
        print("Child: methodB called")
child = Child()
child.methodA()  # 调用父类的 methodA 但输出 Child: methodB called  # 实际调用子类重写的 methodB
```

继承的核心功能如下：

功能	说明	示例代码片段

功能	说明	示例代码片段
代码复用	子类自动获得父类所有非私有属性和方法	child = Child() child.parent_method()
功能扩展	子类添加新属性和方法	class Child(Parent): def new_method(self):...
方法重写	子类定义同名方法覆盖父类实现	class Child(Parent): def method(self):...
多态支持	不同子类对同一方法有不同实现	obj.method() 根据实际对象类型调用对应实现
接口标准化	通过抽象基类强制子类实现特定方法	from abc import ABC, abstractmethod

继承的种类

单继承：子类只继承一个父类（最基础形式）

```
class Animal:
    def speak(self):
        print('动物会发出声音')
class Dog(Animal):
    def spark(self):
        print('小狗会汪汪叫')

dog=Dog()
dog.speak()  # 调用父类的方法，  输出：动物会发出声音
dog.spark()  # 调用自己的方法，输出：小狗会汪汪叫
```

多继承：子类同时继承多个父类（Python 特色）

```
class Flyer:
    def fly(self):
        print("在空中飞行")
class Swimmer:
    def swim(self):
        print("在水里游泳")
class Duck(Flyer, Swimmer):   # 多继承
    def quack(self):
        print("嘎嘎嘎!")

duck = Duck()
duck.fly()    # 来自 Flyer -> 在空中飞行
duck.swim()   # 来自 Swimmer -> 在水里游泳
duck.quack() # 子类自有方法 -> 嘎嘎嘎!
```

多层继承;形成继承链（祖父 → 父 → 子）

```
class Vehicle:
    def transport(self):
        print("运输工具")
class Car(Vehicle):
    def run(self):
        print("在公路上行驶")
class ElectricCar(Car):   # 多层继承
    def charge(self):
        print("电能驱动")

tesla = ElectricCar()
tesla.transport()   # 继承自 Vehicle -> 运输工具
tesla.run()         # 继承自 Car -> 在公路上行驶
tesla.charge()      # 自有方法 -> 电能驱动
```

菱形继承（钻石问题）:多继承中父类有共同祖先（通过 MRO 算法解决）

```
class A:
    def show(self):
        print('A')
class B(A):
    def show(self):
        print('B')
class C(A):
    def show(self):
        print('C')
class D(B,C):
    # def show(self):
    #     print('D')   # （1）如果子类重写方法，则直接调用自己的方法
    pass                # （2）如果子类不重写方法，则按照 MRO 顺序调用父类祖先类方法
d=D()
d.show()   # 重写输出 D；不重写：输出 B
print('D.mro:',D.mro())     # D.mro: [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

抽象基类继承:强制子类实现特定方法（接口约束）

```
from abc import ABC, abstractmethod
class Shape(ABC):   # 抽象基类
    @abstractmethod
    def area(self):   # 抽象方法
        pass
```

```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):   # 必须实现抽象方法
        return 3.14 * self.radius ** 2
# shape = Shape()   # 报错：不能实例化抽象类
circle = Circle(5)
print(circle.area())   # 78.5
```

方法解析顺序（MRO）

MRO（Method Resolution Order）是 Python 中用于在多继承中确定方法调用顺序的算法。

1. 什么是 MRO？ MRO 是一个顺序列表，它定义了在多继承中，当子类调用一个方法时，Python 解释器搜索该方法的顺序。 在 Python 中，每个类都有一个__mro__属性，它是一个列表，按照方法解析顺序列出了该类及其所有基类。

其核心功能如下：

解决菱形继承问题，保证子类调用方法时按照合适的顺序，**保证方法调用的确定性和一致性，**避免父类方法被多次调用或遗漏调用（在 super()的使用中体现）。

2. MRO 的原理（C3 线性化算法） Python 使用 C3 线性化算法来计算 MRO。该算法遵循以下三个原则：

- 子类优先于父类：例如，在 class C(A, B)中，A 和 B 是父类，则 C 的 MRO 中 C 本身在最前面。
- 继承图中保持基类的顺序：例如，class C(A, B)中，A 在 B 前面，那么在 MRO 中 A 的基类也会在 B 的基类前面（前提是不违反子类优先）。
- 单调性：若类 X 在类 Y 前，则 X 的所有子类也在 Y 前。

算法步骤（简述）：

- 对于每个类，它的 MRO 是它自身加上其父类的 MRO 的合并。
- 合并规则： 给定类定义 class D(B, C)：计算 L(D) = D + merge(L(B), L(C), [B, C])

merge 操作：

取第一个列表的头部元素
如果该元素不在其他列表的尾部（非第一个位置）
则添加到结果中并从所有列表中移除
否则，检查下一个列表的头
重复直到所有列表为空

3. 计算示例

```
class A:
Pass
```

```
class B(A):
Pass
class C(A):
Pass
class D(B, C):
pass
# 计算过程：
L(A) = [A]
L(B) = [B] + merge(L(A)) = [B, A]
L(C) = [C] + merge(L(A)) = [C, A]
L(D) = [D] + merge(L(B), L(C), [B, C])
        = [D] + merge([B,A], [C,A], [B,C])
# 第一步：取 B（在第一个列表头，且不在其他列表尾）
        = [D, B] + merge([A], [C,A], [C])
# 第二步：取 C（在第二个列表头，且不在其他列表尾）
        = [D, B, C] + merge([A], [A])
# 第三步：取 A
        = [D, B, C, A]
```

任何类都可以通过 类名.mro() 或 类名.__mro__ 查看 MRO 顺序：

```
print(D.mro())# 输出: [<class '__main__.D'>, <class '__main__.B'>, #
<class '__main__.C'>, <class '__main__.A'>, #          <class 'object'>]
```

当 MRO 无法计算时，Python 会抛出类型错误：TypeError: Cannot create a consistent method resolution order (MRO) for bases A, B

super() 函数

前面介绍MRO算法的时候讲到，它可以解决多继承中的方法调用顺序问题，事实上，MRO 算法只是提供了继承关系的解析顺序，而真正按照这个顺序完成方法调用的是 super()函数。

super()函数用于在子类中调用父类（超类）的方法，包括初始化方法。它会根据方法解析顺序（MRO）动态地确定要调用的父类方法，从而解决了多继承中的**方法调用顺序问题**。

super() 的核心原理

super() 基于 **MRO（Method Resolution Order，方法解析顺序）** 工作：具体来说：super() 在当前类的 MRO 中查找下一个类，返回一个代理对象，通过该对象调用父类方法。

1. 基本用法：调用父类方法

先看一个最基本的用法，在子类 init 方法中调用父类的 init 方法，

```
class Animal:
    def __init__(self, name):
        self.name = name
        print(f"Animal initialized: {self.name}")
```

```
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # 使用 super（）调用父类构造方法
        # Animal.__init__(self,name) # 硬编码即直接使用父类名称调用，需要加个 self
        self.breed = breed
        print(f"Dog initialized: {self.breed}")


dog = Dog("Buddy", "Golden Retriever")"""
输出:
Animal initialized: Buddy
Dog initialized: Golden Retriever
"""
```

同理可进行扩展，在子类的普通方法调用父类的普通方法，在子类的类方法里调用父类的类方法。但是静态方法中不能使用 super（）函数，因为无法绑定实例。

```
# 多继承
class Camera:
    def take_photo(self):
        print("Taking photo")
class Phone:
    def make_call(self):
        print("Making call")
class SmartPhone(Camera, Phone):
    def use_features(self):
        # super().take_photo()  # 使用 super 调用 Camera 的方法
        # super().make_call()  # 使用 super 调用 Phone 的方法
        Camera.take_photo(self)  # 直接使用父类名称调用 Camera 的方法
        Phone.make_call(self)  # 直接使用父类名称调用 Phone 的方法

phone = SmartPhone()
phone.use_features()
```

那么 super（）函数的优势或者说到底解决了什么问题呢？难度仅仅是不用写每个父类的名称而统一使用 super().调用吗？

前面讲到 super（）函数是解决了菱形继承的调用顺序问题，我们先看一下不使用 super（）函数会出现什么问题：

4. 菱形继承问题解决方案

```
class A:
    def process(self):
        print("Processing in A")
class B(A):
    def process(self):
        print("Processing in B")
```

```
            super().process()
            # A.process(self)
class C(A):
    def process(self):
            print("Processing in C")
            super().process()
            # A.process(self)
class D(B, C):
    def process(self):
            print("Processing in D")
            # super().process()
            B.process(self)
            C.process(self)
    def process2(self):
            print("Processing2 in D")
            super().process()
            # B.process(self)
            # C.process(self)
d = D()
d.process2()
```

结合上面的代码我们来分析一下使用 super（）函数的作用：

（1）第一种情况，所有的类都不使用 super（）函数，那么第一个问题就是当类 D 多继承 B 和 C 的时候，需要分别写出 B.process（self）和 C.process（self），顺序需要我们手动写；然后运行的时候会输出：D-B-A-C-A，也就是说是按照 B.process（self）和 C.process（self）依次运行的，也就导致 B 和 C 分别调用了一次 A；

（2）第二种情况，最后的子类 D 使用 super（）函数，其它类使用父类名称，则输出：D-B-A，可以看到，遗漏了类 C 的方法，这是因为 D 找到 B 之后直接使用父类名称调用了 A 就结束了。但这种情况适用于需要根据条件明确使用不同的父类，分条件调用父类方法。

（3）第三种情况，全部使用 super()函数：则输出 D-B-C-A,这说明是按照 MRO 提供的解析顺序调用父类方法的；

（4）第四种情况：最后的子类在不同名称的方法内部调用父类的方法，输出的顺序和上面一致。这说明什么？说明 super（）只是一个调用方法，可以在任意一个子类方法中使用，只需要标注要调用的父类方法。前面三种情况是为了完整介绍 super（）函数调用父类方法的顺序，但在实际项目中，子类继承的多个父类不可能都拥有同样名称的方法，因此并不是说 super（）函数只能在同名方法中使用。

（5）第五种情况，B 或 C 不使用 super（）函数，也不使用父类名称调用，则会输出 D-B，停止到不继续调用的类，这意味着在子类调用某个方法时，super（）函数只是按照 MRO 算法的顺序去寻找这个方法，找到之后会执行；但并不是所有同名方法都会被执行（那子类方法重写就没有意义了），而是只有这个方法内部又调用了 super()或父类名称，那么就会继续执行 MRO 链中下一个类的同名方法，以此类推，直到整个 MRO 链被遍历完或者某个方法没有调用 super()而终止。

深度学习中使用 super() 的原因分析

在深度学习框架（如 PyTorch）中，super() 被广泛用于网络构建尤其实是在初始化方法里，主要原因如下：

1. 确保父类初始化正确执行

深度学习模型通常继承自框架的基类（如 nn.Module），这些基类包含**关键初始化逻辑**：

```
import torch.nn as nn
class MyModel(nn.Module):
    def __init__(self):
        super().__init__()  # 必须调用父类初始化
        self.layer = nn.Linear(10, 5)

    def forward(self, x):
        return self.layer(x)
```

不调用 super().__init__() 的后果：

模型无法注册子模块（参数不被识别）

无法正确转移到 GPU

无法保存/加载模型状态

2. 支持模块化设计

深度学习网络常采用**分层结构**，super() 实现各层的协作：

```
class BaseBlock(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, in_channels, 3, padding=1)
    def forward(self, x):
        return self.conv(x)
class ResidualBlock(BaseBlock):
    def __init__(self, in_channels):
        super().__init__(in_channels)  # 初始化基础卷积
    def forward(self, x):
        return x + super().forward(x)  # 调用父类 forward 并添加残差连接
```

继承 vs 组合

	继承 (is-a 关系)	**组合 (has-a 关系)**
关系	"是一个" (狗是动物)	"有一个" (汽车有发动机)

代码	class Dog(Animal):	class Car:
		def __init__(self):
		self.engine = Engine()

优点	代码复用性强	降低耦合度

	继承 (is-a 关系)	组合 (has-a 关系)
缺点	可能产生深度耦合	需显式调用组件方法
原则	优先使用组合，继承用于真正"是"的关系	灵活构建复杂系统

总结：Python 继承要点

最佳实践：避免深度继承链（建议 ≤3 层）

多继承时使用 Mixin 类（单一功能的小类）

优先组合而非继承降低耦合度