杂七杂八的，我也不知道在写什么

Python程序是大小写敏感的，如果写错了大小写，程序会报错

在python程序中很重要的就是代码的缩进问题，所以一定要注意格式！(直接按`Tab`这个键就可以缩进了)

以`#`开头的语句是注释

代码每一行都是一个语句，当语句以冒号:结尾时，缩进的语句视为代码块（惯例是四个空格的缩进）

```
a = 100
if a >= 0:
    print(a)
else:
    print(-a)
```

用单引号或者双引号括起来的文本在程序中叫字符串，即`''`或`""`框起来的文本，同时值为字符串的变量叫字符串变量

**代表幂运算，/是除法，其他四则运算不变

```
#幂运算的使用：
print(2**10)
#结果为1024
```

> 如果遇到`SyntaxError`，表示输入的Python代码有语法错误，最常见的一种语法错误是使用了中文标点
>
> ```
> #例如：中文括号
> >>>print（'hello'）
> File "<stdin>", line 1
>  print（'hello'）
>       ^
> SyntaxError: invalid character '（' (U+FF08)
>
> #中文引号
> >>> print(“hello”)
> File "<stdin>", line 1
>  print(“hello”)
>        ^
> SyntaxError: invalid character '“' (U+201C)
> ```

print()函数也可以接受多个字符串，用逗号`,`隔开，就可以连成一串输出（遇到逗号`,`会输出一个空格，且可以写入其他指令）

```
print('The quick brown fox', 'jumps over', 'the lazy dog')
#输出：
#The quick brown fox jumps over the lazy dog
```

python中并不需要关心print()函数中是用的`''`还是`""`，换言之就是在其中选择这两个中任何 一个都可以，但不能一前一后地使用

```
#例如：
print('hello world!')
print("hello world!")
#两者在输出上并无差别
print("hello world!)
print('hello world!")
#这样就不对
```

python不用注意变量类型(暂时不需要关心数据类型)

`input()`用于获取用户的输入（可以在其中添加字符串）

```
name=input()
#比如说输入LiuJiaNi
print('Hello,',name)
#打印出来就是Hello LiuJiaNi
#我们可以尝试在input()中添加字符串
name=input('Please enter your name: ')
#依旧输入LiuJiaNi
print('Hello,',name)
#效果如下：
#Please enter your name: LiuJiaNi
#Hello, LiuJiaNi
```

`input` `output`分别为输入和输出，简写为IO

python虽然并不需要注重数据类型，但对于很大的数来说，例如`10000000000`，很难数清楚0的个数，即容易发生打错的情况，Python允许在数字中间以`_`分隔，因此，写成`10_000_000_000`和`10000000000`是完全一样的

浮点数运算可能会有四舍五入的误差（整数和浮点数在计算机内部存储的方式是不同的，整数运算永远是精确的）

Python的整数和浮点数没有大小限制，但是浮点数超出一定范围就直接表示为`inf`（无限大）

> 如果字符串内部既包含`'`又包含`"`，可以在前用转义字符`\`来标识
>
> ```
> #例如：'I\'m \"OK\"!'
> #猜猜看这应该是什么呢？
> #当然是I'm "OK"!
> ```
>
> 字符串是以单引号`'`或双引号`"`括起来的任意文本，比如`'abc'`，`"xyz"`等等。请注意，`''`或`""`本身只是一种表示方式，不是字符串的一部分，因此，字符串`'abc'`只有`a`，`b`，`c`这3个字符。如果`'`本身也是一个字符，那就可以用`""`括起来，比如`"I'm OK"`包含的字符是`I`，`'`，`m`，空格，`O`，`K`这6个字符。
>
> 转义字符`\`可以转义很多字符，比如`\n`表示换行，`\t`表示制表符（就是TAB那个东西，表示缩进），字符`\`本身也要转义，所以`\\`表示的字符就是`\`，如果字符串里面有很多字符都需要转义，就需要加很多`\`，为了简化，Python还允许用`r''`表示`''`内部的字符串默认不转义
>
> ```
> print('I\'m ok.')
> #输出：
> #I'm ok.
> print('I\'m learning\nPython.')
> ```

```
#输出:
#I'm learning
#Python.

print('\\\n\\')
#输出:
#\
#\

print('\\\t\\')
#输出:
#\       \

print(r'\\\t\\')
#输出:
#\\\t\\

#提示:是不是有点绕?没关系,不用管那么多,因为不重要,了解一下就行,谁jb没事写代码这么干啊
```

如果字符串内部有很多换行，用`\n`写在一行里不好阅读，为了简化，Python允许直接在print中表示多行内容

```
print('''line1
line2
line3''')
#输出:
#line1
#line2
#line3
```

布尔值和布尔代数的表示完全一致，一个布尔值只有`True`、`False`两种值，要么是`True`，要么是`False`，在Python中，可以直接用`True`、`False`表示布尔值（请注意大小写），也可以通过布尔运算计算出来

```
True
#输出:
#True

False
#输出:
#False

3 > 2
#输出:
#True

3 > 5
#输出:
#False
```

布尔值可以用`and`、`or`和`not`运算（这个就不用说了吧）

`and`运算是与运算，，只有所有都为`True`，`and`运算结果才是`True`

`or`运算是或运算，只要其中有一个为`True`，`or`运算结果就是`True`

`not`运算是非运算，它是一个单目运算符，把`True`变成`False`，`False`变成`True`

空值是Python里一个特殊的值，用`None`表示。`None`不能理解为`0`，因为`0`是有意义的，而`None`是一个特殊的空值

变量名必须是大小写英文、数字和_的组合，且不能用数字开头(注意一下就是了)

```
a = 1
t_007 = 'T007'
Answer = True
#分辨一下类型，分别是整数，字符串，布尔值
```

在Python中，等号`=`是赋值语句，可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量（不要当成数学上的`=`了）

```
a = 123
#a是整数
print(a)
a = 'ABC'
#a变为字符串
print(a)
```

变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言，这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错（python不用管）

常量就是不能变的变量，比如常用的数学常数π就是一个常量。在Python中，通常用全部大写的变量名表示常量

```
PI = 3.14159265359
#但事实上PI仍然是一个变量，Python根本没有任何机制保证PI不会被改变，所以，用全部大写的变量名表示常量只是一个习惯上的用法
```

在Python中，有两种除法，一种除法是`/`，一种除法是`//`，称为地板除，两个整数的除法仍然是整数

`/`除法计算结果是浮点数，即使是两个整数恰好整除，结果也是浮点数（可以认为是保留的小数位数不同）

还有一种除法是`//`，称为地板除，两个整数的除法仍然是整数

```
10 / 3
#输出：
#3.3333333333333335
#这时你就会发现有浮点数的运算就不那么精准了

9 / 3
#输出：
#3.0

10 // 3
#输出：
#3
```

```
13
14 #发现不同了吗？
```

`%`可以做除法的取余数，但是是保留整数

```
1 10 % 3
2 #输出：
3 #1
```

字符编码问题的话，看看ASCII表就可以了

ASCII 码表

ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符	ASCII 值	控制字符
0	NUT	32	(space)	64	@	96	、
1	SOH	33	!	65	A	97	a
2	STX	34	”	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	,	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DCI	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	X	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	TB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	/	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	—	127	DEL

对于单个字符的编码，Python提供了`ord()`函数获取字符的整数表示，`chr()`函数把编码转换为对应的字符

```
1 ord('A')
2 #输出：
3 #65
4
5 ord('中')
```

```
#输出：
#20013

chr(66)
#输出：
#'B'

chr(25991)
#输出：
#'文'

#具体的话看看ASCII表，里面存有对应的值
```

由于Python的字符串类型是`str`，在内存中以Unicode表示，一个字符对应若干个字节。如果要在网络上传输，或者保存到磁盘上，就需要把`str`变为以字节为单位的`bytes`，用带`b`前缀的单引号或双引号表示bytes类型的数据

> ***以下内容仅供了解***
>
> ```
> x = b'ABC'
> #要注意区分'ABC'和b'ABC'，前者是str，后者虽然内容显示得和前者一样，但bytes的每个字符都只占用一个字节
> ```
>
> 以Unicode表示的`str`通过`encode()`方法可以编码为指定的`bytes`
>
> ```
> 'ABC'.encode('ascii')
> #输出：
> #b'ABC'
>
> '中文'.encode('utf-8')
> #输出：
> #b'\xe4\xb8\xad\xe6\x96\x87'
>
> '中文'.encode('ascii')
> Traceback (most recent call last):
> File "<stdin>", line 1, in <module>
> UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1: ordinal not in range(128)
> #哦偶，错误
> #纯英文的str可以用ASCII编码为bytes，内容是一样的，含有中文的str可以用UTF-8编码为bytes。含有中文的str无法用ASCII编码，因为中文编码的范围超过了ASCII编码的范围，Python会报错
> ```
>
> 在`bytes`中，无法显示为ASCII字符的字节，用`\x##`显示
>
> 反过来，如果我们从网络或磁盘上读取了字节流，那么读到的数据就是`bytes`。要把`bytes`变为`str`，就需要用`decode()`方法

```
b'ABC'.decode('ascii')
#输出：
#'ABC'

b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
#输出：
#'中文'

b'\xe4\xb8\xad\xff'.decode('utf-8')
Traceback (most recent call last):
...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 3: invalid start byte
#反正就是报错了
```

如果要计算`str`包含多少个字符，可以用`len()`函数

```
len('ABC')
#输出：
#3

len('中文')
#输出：
#2
```

`len()`函数计算的是`str`的字符数，如果换成`bytes`，`len()`函数就计算字节数

```
len(b'ABC')
#输出：
#3

len(b'\xe4\xb8\xad\xe6\x96\x87')
#输出：
#6

len('中文'.encode('utf-8'))
#输出：
#6
```

1个中文字符经过`UTF-8`编码后通常会占用3个字节，而1个英文字符只占用1个字节

在操作字符串时，经常遇到`str`和`bytes`的互相转换。为了避免乱码问题，应当始终坚持使用`UTF-8`编码对`str`和`bytes`进行转换

> 为了让解释器按`UTF-8`编码读取，通常在文件开头写上这两行：

```
#!/usr/bin/env python3
\# -*- coding: utf-8 -*-

#第一行注释是为了告诉Linux/OS X系统，这是一个Python可执行程序，Windows系统会忽略这个注释
#第二行注释是为了告诉Python解释器，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码
#电脑能跑就不管这个
```

在Python中，采用的格式化方式和C语言是一致的（不管C语言是怎么搞的，不深究），用`%`实现

`%s`表示用字符串替换，`%d`表示用整数替换，如果只有一个`%`，括号可以省略（`%%`表示`%`）

常见的占位符有：

占位符	替换内容
%d	整数
%f	浮点数
%s	字符串
%x	十六进制整数

```
'Hello, %s' % 'world'
#输出：
#'Hello, world'

'Hi, %s, you have $%d.' % ('LiuJiaNi', 1000000)
#输出：
#'Hi, LiuJiaNi, you have $1000000.'
```

格式化整数和浮点数还可以指定是否补0和整数与小数的位数

```
print('%2d-%02d' % (3, 1))
print('%.2f' % 3.1415926)
#自己试试？
```

如果不太确定应该用什么，`%s`是最好的选择，它会把任何数据类型转换为字符串

```
'Age: %s. Gender: %s' % (19, True)
#输出：
#'Age: 19. Gender: True'

'growth rate: %d %%' % 7
#输出：
#'growth rate: 7 %'
#注意转义字符哦
```

另一种格式化字符串的方法是使用字符串的`format()`方法，它会用传入的参数依次替换字符串内的占位符`{0}`、`{1}`……，不过这种方式写起来比`%`要麻烦得多

```
'Hello, {0}, 成绩提升了 {1:.1f}%'.format('LiuJiaNi', 17.125)
#输出：
#'Hello, LiuJiaNi, 成绩提升了 17.1%'
```

这种`format()`方法为python3.6以前的版本常用的,也就是你们教科书所使用的版本,下面的`f-string`为3.6及以后版本的特性,你可以不用管

最后一种格式化字符串的方法是使用以`f`开头的字符串，称之为`f-string`，它和普通字符串不同之处在于，字符串如果包含`{xxx}`，就会以对应的变量替换

```
r = 2.5
s = 3.14 * r ** 2
print(f'The area of a circle with radius {r} is {s:.2f}')
#输出：
#The area of a circle with radius 2.5 is 19.62
#{r}被变量r的值替换，{s:.2f}被变量s的值替换，并且:后面的.2f指定了格式化参数（即保留两位小数），因此，{s:.2f}的替换结果是19.62
```

列表（好像不应该在这里讲的，无所谓了）

Python内置的一种数据类型是列表：`list`。`list`是一种有序的集合，可以随时添加和删除其中的元素

> 这里提一下,可以看到下面的变量名使用的是复数形式,这是业界的一种默认的命名规则,把列表的变量名写成复数形式,方便与普通变量作区分

```
classmates = ['LiuJiaNi', 'WangHan', 'WangShenZhi','TianWenJie']
#变量classmates就是一个list
#命名格式为：变量名=[]([]中写元素)
```

用`len()`函数可以获得`list`元素的个数

```
len(classmates)
#输出：
#4
```

如果想要得到其中的某一个元素，则用索引来访问`list`中每一个位置的元素（即索引可以理解为元素的顺序）

用索引来访问`list`中每一个位置的元素，但必须知道索引是从`0`开始的，而最后一个元素的索引是`len(classmates) - 1`（即`list`的元素始终比索引大`1`）

还可以用`-1`做索引，直接获取最后一个元素，以此类推

当索引超出了范围时，Python会报一个`IndexError`错误，所以，要确保索引不要越界

`list`是一个可变的有序表（待会就会有不可变的），所以，可以往`list`中追加元素

`list`.**append('元素')**（在`list`的末尾追加元素）

```
classmates = ['LiuJiaNi', 'WangHan', 'WangShenZhi','TianWenJie']
classmates.append('MiGaoFeng')
#此时classmates猜猜看变成什么了
```

`list`.**insert(number, '元素')**（在指定位置插入元素）

```
classmates = ['LiuJiaNi', 'WangHan', 'WangShenZhi','TianWenJie']
classmates.insert(1,'MiGaoFeng')
#此时classmates猜猜看变成什么了
```

`list`.**pop()**（删除`list`末尾元素）（括号中加入索引即可指定删除，不然就是删除最后一个）

```
classmates = ['LiuJiaNi', 'WangHan', 'WangShenZhi','TianWenJie']
classmates.pop()
#此时classmates猜猜看变成什么了

classmates = ['LiuJiaNi', 'WangHan', 'WangShenZhi','TianWenJie']
classmates.pop(1)
#此时classmates猜猜看变成什么了

```

要把某个元素替换成别的元素，可以直接赋值给对应的索引位置

```
classmates = ['LiuJiaNi', 'WangHan', 'WangShenZhi','TianWenJie']
classmates[1]='LiuQiuChen'
#此时classmates猜猜看变成什么了
```

`list`里面的元素的数据类型也可以不同（即可以是数字也可以是字符串等等等等）

```
L = ['Apple', 123, True]
```

`list`中的元素也可以是另一个`list`

```
s = ['python', 'java', ['asp', 'php'], 'scheme']
len(s)
#输出：
#4
#s这个list中仍然只有4个元素，不过却包含了另一个list，这就是嵌套

#多提一句，如果想要得到s里面的list中的元素，则应该这样访问
#s[2][0]或s[2][1]
#这样即为访问第一个元素和第二个元素

p = ['asp', 'php']
s = ['python', 'java', p, 'scheme']
#拆开写就是这样
```

如果一个`list`中一个元素也没有，就是一个空的`list`，它的长度为0

```
L = []
len(L)
#输出：
#0
```

另一种有序列表叫元组：`tuple`。`tuple`和`list`非常类似，但是`tuple`一旦初始化就不能修改

```
classmates = ('LiuJiaNi', 'WangHan', 'WangShenZhi','TianWenJie')
#发现和list的不同了吗？
```

现在，classmates这个`tuple`不能变了，它也没有`append()`，`insert()`这样的方法。其他获取元素的方法和`list`是一样的，可以正常地使用`classmates[0]`，`classmates[-1]`，但不能赋值成另外的元素（不可变的tuple有什么意义？因为tuple不可变，所以代码更安全）

`tuple`的陷阱：当你定义一个`tuple`时，在定义的时候，tuple的元素就必须被确定下来

```
t = (1, 2)
#比如这个

t = ()
#要定义一个空的tuple，可以这样写

#但是，要定义一个只有1个元素的tuple，如果这么定义：
t = (1)
#定义的不是tuple，是1这个数！这是因为括号()既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，计算结果自然是1

#Python在显示只有1个元素的tuple时，也会加一个逗号,，以免误解成数学计算意义上的括号
t = (1,)
```

其实`tuple`也不是不可以“变化”的

```
t = ('a', 'b', ['A', 'B'])
t[2][0] = 'X'
t[2][1] = 'Y'
#注意看，发现了什么吗？
#这个tuple定义的时候有3个元素，分别是'a', 'b'和一个list
#也就是说，tuple是不可变的，但我们可以在里面写入一个list来实现改变（额，感觉有种逼良为娼的感觉-_-，怪怪的）
```

```
#来个练习？
#打印Apple，打印Python，打印Lisa
L = [
    ['Apple', 'Google', 'Microsoft'],
    ['Java', 'Python', 'Ruby', 'PHP'],
    ['Adam', 'Bart', 'Lisa']
]
#试试？
#也要学习一下人家的格式是怎么写的哦
```

条件判断

在Python程序中，用`if`语句实现条件判断

```
age = 20
if age >= 18:
    print('your age is', age)
    print('adult')
#注意缩进，这样你就会发现if语句里面包含了两个打印语句
#根据Python的缩进规则，如果if语句判断是True，就把缩进的两行print语句执行了，否则，什么也不做
```

也可以给`if`添加一个`else`语句，意思是，如果`if`判断是`False`，不要执行`if`的内容，去把`else`执行了

```
age = 3
if age >= 18:
    print('your age is', age)
    print('adult')
else:
    print('your age is', age)
    print('teenager')
#冒号代表后面代码都属于同一个条件哦，如果要再写其他的则要回到前一个缩进位置（为什么是前一个呢，后面会讲，因为语句一多就会有很多缩进，所以要回到前一个缩进，不过有时候也不一定，注意代码逻辑即可
```

更细致一些：

```
age = 3
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
    print('kid')
#elif是else if的缩写

#标准格式
if <条件判断1>:
    <执行语句1>
elif <条件判断2>:
    <执行语句2>
elif <条件判断3>:
    <执行语句3>
......
else:
    <执行语句...>
```

`if`语句执行有个特点，它是从上往下判断，如果在某个判断上是`True`，把该判断对应的语句执行后，就忽略掉剩下的`elif`和`else`

`if`判断条件还可以简写

```python
if x:
    print('True')
#只要x是非零数值、非空字符串、非空list等，就判断为True即执行，否则为False即不执行
```

```python
#这是一段错误的代码
birth = input('birth: ')
if birth < 2000:
    print('00前')
else:
    print('00后')
#猜猜，错在哪？
#input()函数返回的数据类型是str，str不能直接和整数比较，必须先把str转换成整数

#修改
s = input('birth: ')
birth = int(s)
if birth < 2000:
    print('00前')
else:
    print('00后')
#这样就可以了
#Python提供了int()函数来对str进行转换成整数
#但int()函数发现一个字符串并不是合法的数字时就会报错
```

```python
#来个练习
#小明身高1.75，体重80.5kg。请根据BMI公式（体重除以身高的平方）帮小明计算他的BMI指数，并根据BMI指数：
#低于18.5：过轻
#18.5-25：正常
#25-28：过重
#28-32：肥胖
#高于32：严重肥胖
#用if-elif判断并打印结果：
```

当当当！现在就到了一般人最最头疼的地方来了—*循环*

循环语句

Python的循环有两种，一种是`for...in`循环，另一种循环是`while`循环

那么现在可以思考一下，循环这个东西究竟有什么作用呢？它能拿来干什么？

要计算1+2+3，我们可以直接写表达式：

```python
1 + 2 + 3
```

要计算1+2+3+...+10，也能写出来。但是，要计算1+2+3+...+10000，直接写表达式就不可能了。为了让计算机能计算成千上万次的重复运算，我们就需要循环语句

`for x in ...`循环就是把每个元素代入变量`x`，然后执行缩进块的语句

```python
#比如我们想计算1-10的整数之和，可以用一个sum变量做累加
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print(sum)
#最后是不是就可以得到1-10的和了，但是这样也比较麻烦
```

如果要计算1-100的整数之和，从1写到100有点困难，但Python提供一个`range()`函数，可以生成一个整数序列，再通过`list()`函数可以转换为`list`

```python
#这就是生成了0-4的整数，但是请注意，使用这个函数的时候是从0开始的，联系索引内容
list(range(5))
#输出：
#[0, 1, 2, 3, 4]

sum = 0
for x in range(101):
    sum = sum + x
print(sum)
#现在这个看起来是不是就简洁多了，上面那个如果用来写1-100的话，人都写麻了
```

`for x in ...`循环还有一个很重要的作用，就是对列表里面的元素进行迭代处理，什么意思呢？

```python
names = ['LiuJiaNi', 'WangHan', 'WangShenZhi','TianWenJie']
for name in names:
    print(name)
#执行这段代码，就会依次打印names的每一个元素
#是不是很简单，别急，我之所以说这是最最让人头疼的地方就是循环单看确实很简单，但是如果我掺杂了其他方面的内容的话，你脑壳还能转得过来吗？比如加上判断再加点列表和计算？
```

第二种循环是`while`循环，只要条件满足，就不断循环，条件不满足时退出循环

```python
sum = 0
n = 99
while n > 0:
    sum = sum + n
    n = n - 2
print(sum)
#在循环内部变量n不断自减，直到变为-1时，不再满足while条件
#什么叫做条件满足呢，其实仔细思考一下n>0是不是在每进行一次减2运算的时候都要判断一下？
#是不是要n>0这个式子的值为True时才继续啊，所以while后面的表达式的值要为True才能执行，否则跳过

#如果我这么写呢？
while True:
    sum = sum + n
#是不是永远都在里面循环了，那这个该怎么处理以及它有什么用呢，下面马上就会讲到

#来个练习？
#自己创建一个list类型的变量，往其中储存数据，使用循环对list类型中的数据打印出Hello,xxx!
#效果：
#Hello,LiuJiaNi!
#Hello,WangHan!
```

```
#Hello,WangShenZhi!
#Hello,TianWenJie
```

循环控制语句（猜猜看跟上面那个是什么关系？）

什么是循环控制语句呢？即`break`和`continue`，什么意思呢，顾名思义，break是用来打破循环的，也就是跳出循环，不过一个`break`对应一个循环语句哦（对的循环可以嵌套哦，所以它才麻烦啊，想想双重循环，甚至三重循环），而`continue`呢，则是将循环继续的，当然也是一一对应的哦

上面代码中的那个无限循环现在不就可以解决了吗

```
sum = 0
n = 99
while True:
    sum = sum + n
    n = n - 2
    if(n < 0):
        print(sum)
        break
print('over')
#看明白了吗？
#break的作用是提前结束循环,也就是说，如果在同一代码块中的break后面再写点代码会有用吗？
```

在循环过程中，也可以通过`continue`语句，跳过当前的这次循环，直接开始下一次循环

```
n = 0
while n < 10:
    n = n + 1
    if n % 2 == 0:
        continue
    print(n)
#看看这又是什么意思呢
#又问一句，如果在同一代码块中的continue后面再写点代码会有用吗？
```

不要滥用`break`和`continue`语句。`break`和`continue`会造成代码执行逻辑分叉过多，容易出错。大多数循环并不需要用到`break`和`continue`语句，上面的两个例子，都可以通过改写循环条件或者修改循环逻辑，去掉`break`和`continue`语句

如果代码写得有问题，会让程序陷入“死循环”，也就是永远循环下去。这时可以用`Ctrl+C`退出程序，或者强制结束Python进程

循环完了，是不是感觉还好或者是有点绕了，别急，多做几道题，你会更急的

字典

Python内置了字典：`dict`的支持，`dict`全称dictionary，在其他语言中也称为map，使用键-值（`key-value`）存储，具有极快的查找速度，需要注意的是，key是不能重复的

格式：`字典名={元素1,元素2,...}`

```
#举个例子，假设要根据同学的名字查找对应的成绩，如果用list实现，需要两个list：
```

```
names = ['Michael', 'Bob', 'Tracy']
scores = [95, 75, 85]

#给定一个名字，要查找对应的成绩，就先要在names中找到对应的位置，再从scores取出对应的成绩，list越长，耗时越长。
#如果用dict实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。

#用Python写一个dict如下：
d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
d['Michael']
#输出：
#95

#在这里提一句，虽然这种区域里我写的是代码，但是不一定代表它就能跑起来，emmm，也就是有的东西你看起来是能理解的，但是有的可能不能放在程序里面的，所以自己要多动手
```

`dict`的实现原理和查字典是一样的。假设字典包含了1万个汉字，我们要查某一个字，一个办法是把字典从第一页往后翻，直到找到我们想要的字为止，这种方法就是在`list`中查找元素的方法，`list`越大，查找越慢。第二种方法是先在字典的索引表里（比如部首表）查这个字对应的页码，然后直接翻到该页，找到这个字。无论找哪个字，这种查找速度都非常快，不会随着字典大小的增加而变慢。`dict`就是第二种实现方式，给定一个名字，比如`'Michael'`，`dict`在内部就可以直接计算出`Michael`对应的存放成绩的“页码”，也就是`95`这个数字存放的内存地址，直接取出来，所以速度非常快

但这种`key-value`存储方式，在放进去的时候，必须根据`key`算出`value`的存放位置，这样，取的时候才能根据`key`直接拿到`value`

把数据放入`dict`的方法，除了初始化时指定外，还可以通过`key`放入

```
d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
d['Adam'] = 67
d['Adam']
#输出：
#67

#由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值覆盖掉
d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
d['Jack'] = 90
d['Jack']
#输出：
#90
d['Jack'] = 88
d['Jack']
#输出：
#88
```

如果`key`不存在，`dict`就会报错，要避免`key`不存在的错误，有两种办法，一是通过`in`判断`key`是否存在，二是通过`dict`提供的`get()`方法，如果`key`不存在，可以返回`None`，或者自己指定的`value`

```
d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
'Thomas' in d
#输出：
#False

```

```
d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
d.get('Thomas')
#输出:
#
#是不是什么也没有？连空格都没有哦，这就是none

d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
d.get('Thomas', -1)
#输出:
#-1
#这就是指定值输出
```

要删除一个`key`，用`pop(key)`方法，对应的`value`也会从`dict`中删除

```
d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
d.pop('Bob')
#输出:
#75

d
#输出:
#{'Michael': 95, 'Tracy': 85}
```

`dict`有以下几个特点:

1. *查找和插入的速度极快，不会随着`key`的增加而变慢;*
2. *需要占用大量的内存，内存浪费多。*

而`list`相反:

1. *查找和插入的时间随着元素的增加而增加;*
2. *占用空间小，浪费内存很少。*

所以`dict`是用空间来换取时间的一种方法

需要记住的一点就是`dict`的`key`必须是**不可变对象**

这是因为`dict`根据`key`来计算`value`的存储位置，如果每次计算相同的`key`得出的结果不同，那`dict`内部就完全混乱了。这个通过`key`计算位置的算法称为**哈希算法（Hash）**。要保证hash的正确性，作为`key`的对象就不能变。在Python中，字符串、整数等都是不可变的，因此，可以放心地作为`key`。而`list`是可变的，就不能作为`key`

`set`和`dict`类似，也是一组`key`的集合，但不存储`value`。

要创建一个`set`，需要提供一个`list`作为输入集合

```
s = set([1, 2, 3])
s
#输出:
#{1, 2, 3}
#需要注意的是传入的参数[1, 2, 3]是一个list，而显示的{1, 2, 3}只是告诉你这个set内部有1，2，3这3个元素，显示的顺序也不表示set是有序的
#重复元素在set中自动被过滤
s = set([1, 1, 2, 2, 3, 3])
s
#输出:
#{1, 2, 3}
```

通过`add(key)`方法可以添加元素到`set`中，可以重复添加，但不会有效果

```
s = set([1, 2, 3])
s.add(4)
s
#输出:
#{1, 2, 3, 4}
s.add(4)
s
#输出:
#{1, 2, 3, 4}
```

通过`remove(key)`方法可以删除元素

```
s = set([1, 2, 3])
s.remove(4)
s
#输出;
#{1, 2, 3}
```

`set`可以看成数学意义上的无序和无重复元素的集合，因此，两个`se`t可以做数学意义上的交集、并集等操作

```
s1 = set([1, 2, 3])
s2 = set([2, 3, 4])
s1 & s2
#输出:
#{2, 3}

s1 | s2
#输出:
#{1, 2, 3, 4}

# &和|这俩玩意我讲过吗？算了，不重要，没讲问老王去，我累了
```

`set`和`dict`的唯一区别仅在于没有存储对应的`value`，但是，`set`的原理和`dict`一样，所以，同样不可以放入可变对象，因为无法判断两个可变对象是否相等，也就无法保证`set`内部“不会有重复元素”

> ## 再穿插一点别的，反正我都是在乱讲
>
> 前面有提到`str`是不变对象，而`list`是可变对象

对于可变对象，比如`list`，对`list`进行操作，`list`内部的内容是会变化的

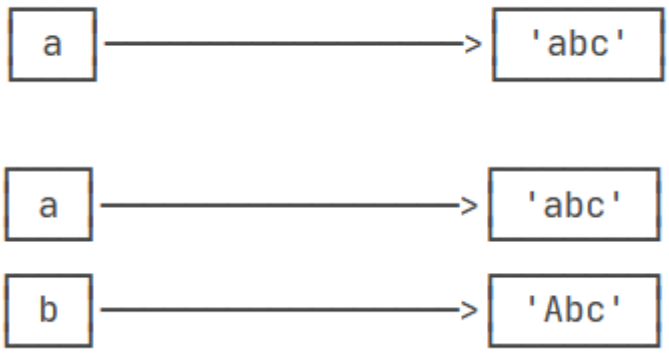
而对于不可变对象，比如`str`则是不行的

```
#不过下面这又是怎么一回事呢？
a = 'abc'
a.replace('a', 'A')
#输出：
#'Abc'

a
#输出：
#'abc'

#虽然字符串有个replace()方法，也确实变出了'Abc'，但变量a最后仍是'abc'

#修改一下
a = 'abc'
b = a.replace('a', 'A')
b
#输出：
#'Abc'
a
#输出：
#'abc'
#要始终牢记的是，a是变量，而'abc'才是字符串对象，我们经常说，对象a的内容是'abc'，但其实是指，a本身是一个变量，它指向的对象的内容才是'abc'，当我们调用a.replace('a', 'A')时，实际上调用方法replace是作用在字符串对象'abc'上的，而这个方法虽然名字叫replace，但却没有改变字符串'abc'的内容。相反，replace方法创建了一个新字符串'Abc'并返回，如果我们用变量b指向该新字符串，就容易理解了，变量a仍指向原有的字符串'abc'，但变量b却指向新字符串'Abc'了
```



所以，对于不变对象来说，调用对象自身的任意方法，也不会改变该对象自身的内容。相反，这些方法会创建新的对象并返回，这样，就保证了不可变对象本身永远是不可变的

函数

Python内置了很多有用的函数，可以直接调用，要调用一个函数，需要知道函数的名称和参数

```
#调用abs函数：
abs(100)
#输出：
#100
```

```
abs(-20)
#输出：
#20

abs(12.34)
#输出：
#12.34

#abs()函数其实就是求绝对值的

max(1, 2)
#输出：
#2

max(2, 3, 1, -5)
#输出：
#3

#max()可以接收任意多个参数，并返回最大的那个
```

调用函数的时候，如果传入的参数数量不对，会报`TypeError`的错误

```
abs(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: abs() takes exactly one argument (2 given)
#abs()有且仅有1个参数，但给予了两个
```

Python内置的常用函数还包括数据类型转换函数，比如`int()`函数可以把其他数据类型转换为整数：

```
int('123')
123
int(12.34)
12
float('12.34')
12.34
str(1.23)
'1.23'
str(100)
'100'
bool(1)
True
bool('')
False
#懒得打字了，自己琢磨
```

函数名其实就是指向一个函数对象的引用，完全可以把函数名赋给一个变量，相当于给这个函数起了一个“别名”

```
a = abs
a(-1)
#输出：
#1

# 变量a指向abs函数
# 所以也可以通过a调用abs函数
```

当然除了Python自带的函数，你也可以自己编写一个函数，毕竟不可能所有的情况都符合官方给的函数

在Python中，定义一个函数要使用`def`语句，依次写出函数名、括号、括号中的参数和冒号`:`，然后，在缩进块中编写函数体，函数的返回值用`return`语句返回

```
#比如自己写一个绝对值的函数
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

注意，函数体内部的语句在执行时，一旦执行到`return`时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑（跟循环那里一样）

如果没有`return`语句，函数执行完毕后也会返回结果，只是结果为`None`。`return None`可以简写为`return`

> 如果你已经把`my_abs()`的函数定义保存为`abstest.py`文件了，那么，可以在该文件的当前目录下启动Python解释器，用`from abstest import my_abs`来导入`my_abs()`函数，注意`abstest`是文件名（不含`.py`扩展名）
>
> 这里是我学的，你的不一定一样

```
def nop():
    pass
#如果想定义一个什么事也不做的空函数，可以用pass语句
#是不是挺纳闷这东西有啥用的，别急，后面会有用的
```

`pass`可以用来作为占位符，比如现在还没想好怎么写函数的代码，就可以先放一个`pass`，让代码能运行起来

`pass`还可以用在其他语句里（这就是它的作用，可以联系一下`break`和`continue`）

```
if age >= 18:
    pass
#缺少了pass，代码运行就会有语法错误。
```

调用函数时，如果参数个数不对，Python解释器会自动检查出来，并抛出`TypeError`错误，但是如果参数类型不对，Python解释器就无法帮我们检查（因为我开始用的是解释器）

当传入了不恰当的参数时，内置函数`abs`会检查出参数错误，而自己定义的`my_abs`没有参数检查，会导致`if`语句出错，出错信息和`abs`不一样

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in my_abs
TypeError: unorderable types: str() >= int()
>>> abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'
```

```
#修改一下，对参数类型做检查，只允许整数和浮点数类型的参数
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type')
    if x >= 0:
        return x
    else:
        return -x
#数据类型检查可以用内置函数isinstance()实现
```

添加了参数检查后，如果传入错误的参数类型，函数就可以抛出一个错误：

```
>>> my_abs('A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in my_abs
TypeError: bad operand type
```

函数运行结束后是会返回一个值的（当然一般来说），所以函数也可以返回多个值

比如在游戏中经常需要从一个点移动到另一个点，给出坐标、位移和角度，就可以计算出新的坐标：

```
import math
def move(x, y, step, angle=0):
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny
#import math语句表示导入math包，并允许后续代码引用math包里的sin、cos等函数
#调用它
x, y = move(100, 100, 60, math.pi / 6)
print(x, y)
#输出：
#151.96152422706632 70.0

#但其实这只是一种假象，Python函数返回的仍然是单一值，比如：
r = move(100, 100, 60, math.pi / 6)
print(r)
#输出：
#(151.96152422706632, 70.0)

#其实返回值是一个tuple，但是，在语法上，返回一个tuple可以省略括号，而多个变量可以同时接收一个tuple，按位置赋给对应的值，所以，Python的函数返回多值其实就是返回一个tuple，但写起来更方便
```

```
#来个练习？
#请定义一个函数quadratic(a, b, c)，接收3个参数，返回一元二次方程 ax^2+bx+c=0的两个解（这个很简单也很经典了吧）
#提示：计算平方根可以调用math.sqrt()函数
#例如：
import math
math.sqrt(2)
#输出：
#1.4142135623730951
```

定义函数的时候，我们把参数的名字和位置确定下来，函数的接口定义就完成了。对于函数的调用者来说，只需要知道如何传递正确的参数，以及函数将返回什么样的值就够了，函数内部的复杂逻辑被封装起来，调用者无需了解。Python的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

位置参数

```
def power(x):
    return x * x
```

对于`power(x)`函数，参数`x`就是一个位置参数。

当调用`power`函数时，必须传入有且仅有的一个参数`x`：

```
power(5)
#输出：
#25

power(15)
#输出：
#225
```

如果要计算x^3怎么办？可以再定义一个`power3`函数，但是如果要计算x^4、x^5……怎么办？

当然可以把`power(x)`修改为`power(x, n)`，用来计算x^n

```
def power(x, n):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
#对于这个修改后的power(x, n)函数，可以计算任意n次方

power(5, 2)
#输出：
#25
power(5, 3)
#输出：
#125

```

```
#修改后的power(x, n)函数有两个参数：x和n，这两个参数都是位置参数，调用函数时，传入的两个值按照位置顺序依次赋给参数x和n
```

默认参数

新的`power(x, n)`函数定义没有问题，但是，每次我都要输两个数字，感觉很烦，想偷懒，可不可以?

这个时候，默认参数就排上用场了。假设我经常需要计算x^2，所以，完全可以把第二个参数n的默认值设定为2

```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
#这样，当我们调用power(5)时，相当于调用power(5, 2)：
power(5)
#输出：
#25
power(5, 2)
#输出：
#25
#是不是没有区别，当然也可以在后面写其他的数字
```

默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数

使用默认参数最大的好处是能降低调用函数的难度

```
#举个例子，写个一年级大学生注册的函数，需要传入name和gender两个参数：
def enroll(name, gender):
    print('name:', name)
    print('gender:', gender)

#这样，调用enroll()函数只需要传入两个参数：
enroll('LiuJiaNi', 'F')
name: LiuJiaNi
gender: F

#如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加，我们可以把年龄和城市设为默认参数：
def enroll(name, gender, age=18, city='ChengDu'):
    print('name:', name)
    print('gender:', gender)
    print('age:', age)
    print('city:', city)

#这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：
```

```
enroll('LiuJiaNi', 'F')
name: LiuJiaNi
gender: F
age: 18
city: ChengDu

#只有与默认参数不符的学生才需要提供额外的信息：
enroll('WangHan', 'M', 19)
enroll('WangShenZhi', 'M', city='ZheJiang')

#可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个
```

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用`enroll('Bob', 'M', 7)`，意思是，除了`name`，`gender`这两个参数外，最后1个参数应用在参数`age`上，`city`参数由于没有提供，仍然使用默认值，也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用`enroll('Adam', 'M', city='Tianjin')`，意思是，`city`参数用传进去的值，其他默认参数继续使用默认值

定义默认参数要牢记一点：**默认参数必须指向不变对象**

```
#例如：先定义一个函数，传入一个list，添加一个END再返回：
def add_end(L=[]):
    L.append('END')
    return L
#正常调用时，好像没问题
add_end([1, 2, 3])
#输出：
#[1, 2, 3, 'END']
add_end(['x', 'y', 'z'])
#输出：
#['x', 'y', 'z', 'END']

#当使用默认参数调用时，一开始结果也是对的：
add_end()
#输出：
#['END']

#但是，再次调用add_end()时，结果就不对了：
add_end()
#输出：
#['END', 'END']
add_end()
#输出：
#['END', 'END', 'END']

#这是怎么回事呢？
#原因是Python函数在定义的时候，默认参数L的值就被计算出来了，即[]，因为默认参数L也是一个变量，它指向对象[]，每次调用该函数，如果改变了L的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的[]了
```

修改可以用`None`这个不变对象来实现：

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
#现在，无论调用多少次，都不会有问题：
add_end()
#输出：
#['END']
add_end()
#输出：
#['END']
#看，是不是就没问题了
```

为什么要设计`str`、`None`这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象

可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个

```
#举个例子
#给定一组数字a，b，c……，请计算a^2 + b^2 + c^2 + …… + n^2

#要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，首先可以把a，b，c……作为一个list或tuple传进来，这样，函数可以定义如下：
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
#但是调用的时候，需要先组装出一个list或tuple：
calc([1, 2, 3])
#输出：
#14
calc((1, 3, 5, 7))
#输出：
#84

#如果利用可变参数，调用函数的方式可以简化成这样：
calc(1, 2, 3)
#输出：
#14
calc(1, 3, 5, 7)
#输出：
#84

#所以，把函数的参数改为可变参数可以这样写：
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
```

```
    return sum
#定义可变参数和定义一个list或tuple参数相比，仅仅在参数前面加了一个*号。在函数内部，参数numbers接收到的是一个tuple，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：
calc(1, 2)
#输出：
#5
calc()
#输出：
#0

#如果已经有一个list或者tuple，要调用一个可变参数怎么办？可以这样做：
nums = [1, 2, 3]
calc(nums[0], nums[1], nums[2])
#输出：
#14

#这种写法当然是可行的，问题是太繁琐，所以Python允许在list或tuple前面加一个*号，把list或tuple的元素变成可变参数传进去：
nums = [1, 2, 3]
calc(*nums)
#输出：
#14

#*nums表示把nums这个list的所有元素作为可变参数传进去
#怎么样？学费了吗？是不是挺绕的？
```

关键字参数

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个`tuple`。而关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个`dict`

```
#例如这个：
def person(name, age, **kw):
    print('name:', name, 'age:', age, 'other:', kw)
#函数person除了必选参数name和age外，还接受关键字参数kw。在调用该函数时，可以只传入必选参数：
person('Michael', 30)
name: Michael age: 30 other: {}

#也可以传入任意个数的关键字参数：
person('Bob', 35, city='Beijing')
name: Bob age: 35 other: {'city': 'Beijing'}

person('Adam', 45, gender='M', job='Engineer')
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}
```

关键字参数有什么用？它可以扩展函数的功能。比如，在`person`函数里，保证能接收到`name`和`age`这两个参数，但是，如果调用者愿意提供更多的参数也能收到。试想正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。和可变参数类似，也可以先组装出一个`dict`，然后，把该`dict`转换为关键字参数传进去：

```
extra = {'city': 'Beijing', 'job': 'Engineer'}
person('Jack', 24, city=extra['city'], job=extra['job'])
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

#当然，上面复杂的调用可以用简化的写法：
extra = {'city': 'Beijing', 'job': 'Engineer'}
person('Jack', 24, **extra)
name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}
#**extra表示把extra这个dict的所有key-value用关键字参数传入到函数的**kw参数，kw将获得一个dict，注意kw获得的dict是extra的一份拷贝，对kw的改动不会影响到函数外的extra
#好像也没啥好讲的我感觉
```

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，就需要在函数内部通过`kw`检查。

```
#仍以person()函数为例，检查是否有city和job参数：
def person(name, age, **kw):
    if 'city' in kw:
        pass
    if 'job' in kw:
        pass
    print('name:', name, 'age:', age, 'other:', kw)
#这就是检查city和job参数

#但是调用者仍可以传入不受限制的关键字参数：
person('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123456)
#如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收city和job作为关键字参数。这种方式定义的函数如下：
def person(name, age, *, city, job):
    print(name, age, city, job)
#和关键字参数**kw不同，命名关键字参数需要一个特殊分隔符*，*后面的参数被视为命名关键字参数。
#调用方式如下：
person('Jack', 24, city='Beijing', job='Engineer')
#输出：
#Jack 24 Beijing Engineer

#如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符*了：
def person(name, age, *args, city, job):
    print(name, age, args, city, job)
```

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

```
>>> person('Jack', 24, 'Beijing', 'Engineer')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: person() missing 2 required keyword-only arguments: 'city' and 'job'
#由于调用时缺少参数名city和job，Python解释器把前两个参数视为位置参数，后两个参数传给*args，但缺少命名关键字参数导致报错
```

命名关键字参数可以有缺省值，从而简化调用：

```python
def person(name, age, *, city='Beijing', job):
    print(name, age, city, job)
#由于命名关键字参数city具有默认值，调用时，可不传入city参数：
person('Jack', 24, job='Engineer')
#输出：
#Jack 24 Beijing Engineer

#使用命名关键字参数时，要特别注意，如果没有可变参数，就必须加一个*作为特殊分隔符。如果缺少*，Python解释器将无法识别位置参数和命名关键字参数：
def person(name, age, city, job):
    pass
#缺少*，city和job被视为位置参数
```

参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这5种参数都可以组合使用。但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数

```python
#比如定义一个函数，包含上述若干种参数：
def f1(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)
def f2(a, b, c=0, *, d, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

```python
f1(1, 2)
#输出：
#a = 1 b = 2 c = 0 args = () kw = {}
f1(1, 2, c=3)
#输出：
#a = 1 b = 2 c = 3 args = () kw = {}
f1(1, 2, 3, 'a', 'b')
#输出：
#a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
f1(1, 2, 3, 'a', 'b', x=99)
#输出：
#a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
f2(1, 2, d=99, ext=None)
#输出：
#a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

最神奇的是通过一个`tuple`和`dict`，也可以调用上述函数：

```
args = (1, 2, 3, 4)
kw = {'d': 99, 'x': '#'}
f1(*args, **kw)
#输出:
#a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}

args = (1, 2, 3)
kw = {'d': 88, 'x': '#'}
f2(*args, **kw)
#输出:
#a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

递归函数（一个特殊的玩意）

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数

```
#举个例子，计算阶乘:
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
fact(1)
#输出:
#1
fact(5)
#输出:
#120
fact(100)
#输出:
#933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试`fact(1000)`

```
fact(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in fact
  ...
  File "<stdin>", line 4, in fact
RuntimeError: maximum recursion depth exceeded in comparison
```

解决递归调用栈溢出的方法是通过**尾递归**优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。尾递归是指，在函数返回的时候，调用自身本身，并且，return语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的 `fact(n)` 函数由于 `return n * fact(n - 1)` 引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

```
def fact(n):
    return fact_iter(n, 1)
def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
#可以看到，return fact_iter(num - 1, num * product)仅返回递归函数本身，num - 1和num * product在函数调用前就会被计算，不影响函数调用

#fact(5)对应的fact_iter(5, 1)的调用如下：
===> fact_iter(5, 1)
===> fact_iter(4, 5)
===> fact_iter(3, 20)
===> fact_iter(2, 60)
===> fact_iter(1, 120)
===> 120
```

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的 `fact(n)` 函数改成尾递归方式，也会导致栈溢出

```
#来个练习？
#懒得多写了，写个汉诺塔函数（自己百度去）
#请编写move(n, a, b, c)函数，它接收参数n，表示3个柱子A、B、C中第1个柱子A的盘子数量，然后打印出把所有盘子从A借助B移动到C的方法
#效果：
# A --> C
# A --> B
# C --> B
# A --> C
# B --> A
# B --> C
# A --> C
# move(3, 'A', 'B', 'C')
```

文件

读取文件

文本文件可存储的数据量多得难以置信：天气数据、交通数据、社会经济数据、文学作品等。每当需要分析或修改存储在文件中的信息时，读取文件都很有用，对数据分析应用程序来说尤其如此。例如，可以编写一个这样的程序：读取一个文本文件的内容，重新设置这些数据的格式并将其写入文件，让浏览器能够显示这些内容。要使用文本文件中的信息，首先需要将信息读取到内存中。为此，你可以一次性读取文件的全部内容，也可以以每次一行的方式逐步读取。
,首先我们介绍一次性读取文件的操作

```python
#此python程序的文件名为file_reader.py
with open('pi_digits.txt') as file_object:     #pi_digits.txt中的内容为
    contents  = file_object.read()             #3.1415926535
print(contents)                                #  8979323846
                                               #  2643383279
#上面的代码就是打开pi_digits.txt文件,然后将文件中的内容输出到屏幕上
#也就是说输出为3.1415926535
#               8979323846
#               2643383279
#
```

我们现在来一步一步的解释上述代码出现的语句

在这个程序中，第一行代码做了大量的工作。我们先来看看函数`open()`。要以任何方式使用文件，那怕仅仅是打印其内容，都得先打开文件，才能访问它。函数`open()`接受一个参数：要打开的文件的名称。

Python在**当前执行的文件所在的目录中查找指定的文件**。在本例中，当前运行的是file_reader.py，因此Python在file_reader.py所在的目录中查找pi_digits.txt。函数`open()`返回一个表示文件的对象。在这里，`open('pi_digits.txt')`返回一个表示文件pi_digits.txt的对象，Python将该对象赋给file_object供以后使用。关键字with 在不再需要访问文件后将其关闭。在这个程序中，注意到我们调用了`open()`，但没有调用`close()`。也可以调用open() 和`close()`来打开和关闭文件，但这样做时，如果程序存在bug导致方法`close()`未执行，文件将不会关闭。这看似微不足道，但未妥善关闭文件可能导致数据丢失或受损。如果在程序中过早调用`close()`，你会发现需要使用文件时它已关闭 （无法访问），这会导致更多的错误。并非在任何情况下都能轻松确定关闭文件的恰当时机，但通过使用前面所示的结构，可让Python去确定：你只管打开文件，并在需要时使用它，Python自会在合适的时候自动将其关闭。

有了表示pi_digits.txt的文件对象后，使用方法`read()`（前述程序的第二行）读取这个文件的全部内容，并将其作为一个长长的字符串赋给变量contents 。这样，通过打印contents 的值，就可将这个文本文件的全部内容显示出来

但是相比于原始文件，该输出唯一不同的地方是末尾多了一个空行。为何会多出这个空行呢？因为`read()`到达文件末尾时返回一个空字符串，而将这个空字符串显示出来时就是一个空行。要删除多出来的空行，可在函数调用`print()`中使用`rstrip()`方法,该方法就是去除字符串末尾的多余的空白的

```python
with open('pi_digits.txt') as file_object:
    contents = file_object.read()
print(contents.rstrip())
#输出为:
#3.1415926535
#  8979323846
#  2643383279
```

> 除了`rstrip()`方法外,还有其他方法去除不同位置的空白,下面一一介绍
> `lstrip()`方法是去除字符串开头的多余空白
> `strip()`方法是去除字符串头尾所有的多余空白

将类似于pi_digits.txt的简单文件名传递给函数open() 时，Python将在**当前执行的文件**（即.py程序文件）所在的目录中查找。

但是根据你组织文件的方式，有时可能要打开不在程序文件所属目录中的文件。例如，你可能将程序文件存储在了文件夹python_work中，而该文件夹中有一个名为text_files的文件夹用于存储程序文件操作的文本文件。虽然文件夹text_files包含在文件夹python_work中，但仅向`open()`传递位于前者中的文件名称也不可行，因为Python只在文件夹python_work中查找，而不会在其子文件夹text_files中查找。要让Python打开不与程序文件位于同一个目录中的文件，需要提供**文件路径**，让Python到系统的特定位置去查找。由于文件夹text_files位于文件夹python_work中，可以使用相对文件路径来打开其中的文件。相对文件路径让Python到指定的位置去查找，而该位置是相对于当前运行的程序所在目录的,基于上述说明,我们可以将程序改为如下所示:

```
with open('text_files/filename.txt') as file_object:
#这里只是改变了打开文件的语句,其他部分语句不改变
```

> 注意　显示文件路径时，Windows系统使用反斜杠（\）而不是斜杠（/），但在代码中依然可以使用斜杠。但是在MacOS系统里,就是使用的斜杠(/)

还可以将文件在计算机中的准确位置告诉Python，这样就不用关心当前运行的程序存储在什么地方了。这称为绝对文件路径 。在相对路径行不通时，可使用绝对路径。例如，如果text_files并不在文件夹python_work中，而在文件夹other_files中，则向`open()`传递路径`'text_files/filename .txt'`行不通，因为Python只在文件夹python_work中查找该位置。为明确指出希望Python到哪里去查找，需要提供完整的路径。绝对路径通常比相对路径长，因此将其赋给一个变量，再将该变量传递给`open()`会有所帮助:

```
file_path = '/home/ehmatthes/other_files/text_files/_filename_.txt'
with open(file_path) as file_object:
#这里同样只改写了打开文件的语句
```

通过使用绝对路径，可读取系统中任何地方的文件。就目前而言，最简单的做法是，要么将数据文件存储在程序文件所在的目录，要么将其存储在程序文件所在目录下的一个文件夹（如text_files）中。

> 如果在文件路径中直接使用反斜杠，将引发错误，因为反斜杠用于对字符串中的字符进行转义。例如，对于路径"C:\path\to\file.txt" ，其中的\t 将被解读为制表符。如果一定要使用反斜杠，可对路径中的每个反斜杠都进行转义，如"C:\path\to\file.txt" 。因为之前提到过,Windows系统的路径就是以反斜杠来写的,所以应当注意,不要使用反斜杠,但是Mac就没有这种烦恼

刚刚我们介绍了这题读取文件,但是在读取文件时，常常需要检查其中的每一行(而不是整个文件)可能要在文件中查找特定的信息，或者要以某种方式修改文件中的文本。例如，你可能要遍历一个包含天气数据的文件，并使用天气描述中包含sunny字样的行。在新闻报道中，你可能会查找包含标签 的行，并按特定的格式设置它。要以每次一行的方式检查文件，可对文件对象使用for 循环:

```
filename = 'pi_digits.txt'
with open(filename) as file_object:
    for line in file_object:
        print(line)
#输出为3.1415926535
#
#       8979323846
#
#       2643383279
#
```

首先，将要读取的文件的名称赋给变量filename 。这是使用文件时的一种常见做法。变量filename 表示的并非实际文件——它只是一个让Python知道到哪里去查找文件的字符串，因此可以轻松地将'pi_digits.txt' 替换为要使用的另一个文件的名称。调用 `open()` 后，将一个表示文件及其内容的对象赋给了变量file_object 。这里也使用了关键字with ，让Python负责妥善地打开和关闭文件。为查看文件的内容，通过对文件对象执行循环来遍历文件中的每一行。

打印每一行时，发现空白行更多了

为何会出现这些空白行呢？因为在这个文件中，每行的末尾都有一个看不见的换行符，而函数调用print()也会加上一个换行符，因此每行末尾都有两个换行符：一个来自文件，另一个来自函数调用print() 。要消除这些多余的空白行，可在函数调用print() 中使用 `rstrip()` ：

```
filename = 'pi_digits.txt'
with open(filename) as file_object:
    for line in file_object:
        print(line.rstrip())
#输出为3.1415926535
#       8979323846
#       2643383279
```

使用关键字with 时，open() 返回的文件对象只在with 代码块内可用。如果要在with 代码块外访问文件的内容，可在with 代码块内将文件的各行存储在一个列表中，并在with 代码块外使用该列表：可以立即处理文件的各个部分，也可以推迟到程序后面再处理。

下面的示例在with 代码块中将文件pi_digits.txt的各行存储在一个列表中，再在with 代码块外打印：

```
filename = 'pi_digits.txt'
with open(filename) as file_object:
    lines = file_object.readlines()
for line in lines:
    print(line.rstrip())
```

我们可以看到上面使用了一种新的方法,方法 `readlines()` 从文件中读取每一行，并将其存储在一个列表中。接下来，该列表被赋给变量lines 。在with 代码块外，依然可使用这个变量。使用一个简单的for循环来打印lines 中的各行。因为列表lines 的每个元素都对应于文件中的一行，所以输出与文件内容完全一致。
将内容存储在列表之后就可以通过列表来使用这些内容了,无需再次打开文件.

> 注意　读取文本文件时，Python将其中的所有文本都解读为字符串。如果读取的是数，并要将其作为数值使用，就必须使用函数 `int()` 将其转换为整数或使用函数 `float()` 将其转换为浮点数。

那么我们来简单利用一下目前为止所学的内容,来找找圆周率中是否存在自己的生日.
请先思考一下,思考过后再看下方的答案.

以确定某个人的生日是否包含在圆周率值的前1 000 000位中。为此，可将生日表示为一个由数字组成的字符串，再检查这个字符串是否包含在pi_string 中：

```
filename = 'pi_million_digits.txt'
with open(filename) as file_object:
    lines = file_object.readlines()
pi_string = ''
for line in lines:
    pi_string += line.strip()
print(f"{pi_string[:52]}...")
```

```
print(len(pi_string))
for line in lines:
    pi_string += line.strip()
#提醒用户输入生日
birthday = input("Enter your birthday, in the form mmddyy: ")
#检查这个字符串是否包含在pi_string中
if birthday in pi_string:
    print("Your birthday appears in the first million digits of pi!")
else:
    print("Your birthday does not appear in the first million digits of
pi.")
```

写入文件

既然有读取文件,那么自然有写入文件,接下来讲解写入文件相关操作
保存数据的最简单的方式之一是将其写入文件中。通过将输出写入文件，即便关闭包含程序输出的终端窗口，这些输出也依然存在：可以在程序结束运行后查看这些输出，可以与别人分享输出文件，还可以编写程序来将这些输出读取到内存中并进行处理。

首先写入空文件(往一个空文件中写入数据)
要将文本写入文件，你在调用`open()`时需要提供另一个实参，告诉Python你要写入打开的文件。为明白其中的工作原理，我们来将一条简单的消息存储到文件中，而不是将其打印到屏幕上：

```
#write_message.py
filename = 'programming.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

在本例中，调用open()时提供了两个实参。第一个实参也是要打开的文件的名称。第二个实参`('w')`告诉Python，要以写入模式打开这个文件。打开文件时，可指定读取模式`('r')`、写入模式`('w')`、附加模式`('a')`或读写模式`('r+')`。如果省略了模式实参，Python将以默认的只读模式打开文件。所以,在我们读取文件时,并未提供任何参数.

> 如果要写入的文件不存在，函数`open()`将自动创建它。然而，以写入模式（'w'）打开文件时千万要小心，因为如果指定的文件已经存在，Python将在返回文件对象前清空该文件的内容。

可以看到程序中使用文件对象的方法`write()`将一个字符串写入文件。这个程序没有终端输出，但如果打开文件programming.txt，将看到其中包含如下一行内容：

```
I love programming.
```

> 注意　Python只能将字符串写入文本文件。要将数值数据存储到文本文件中，必须先使用函数`str()`将其转换为字符串格式。

以上为写入单行的示例,接下来我们来看看写入多行,与写入单行区别不是很大,只是由于函数`write()`的特性,不会在写入的文本末尾添加换行符，因此如果写入多行时没有指定换行符(\n)，文件看起来可能不是你希望的那样

```
#若不手动添加换行符
filename = 'programming.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
    file_object.write("I love creating new games.")
```

```
#输出为
#I love programming.I love creating new games.
#所以一定要注意,在写入多行信息时,要是想要换行,一定不要忘记手动添加换行符(\n)

#所以,想让两句话在不同行的正确代码如下:
filename = 'programming.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.\n")
    file_object.write("I love creating new games.\n")
#输出如下:
#I love programming.
#I love creating new games.
```

> 当然除了换行符以外,你还可以使用空格、制表符和空行来设置这些输出的格式。

刚刚我们说到,若是以写入模式（'w'）打开文件时千万要小心，因为如果指定的文件已经存在，Python将在返回文件对象前清空该文件的内容。那么,接下来我们会讲到如何不覆盖原文件内容

如果要给文件添加内容，而不是覆盖原有的内容，可以以附加模式`('a')`打开文件。以附加模式打开文件时，Python不会在返回文件对象前清空文件的内容，而是将写入文件的行添加到文件末尾。当然,如果指定的文件不存在，Python将为你创建一个空文件。

下面来修改write_message.py，在既有文件programming.txt中再添加一些你酷爱编程的原因：

```
#程序执行之前programming.txt的内容
#I love programming.
#I love creating new games.

filename = 'programming.txt'
with open(filename, 'a') as file_object:
    file_object.write("I also love finding meaning in large datasets.\n")
    file_object.write("I love creating apps that can run in a browser.\n")

#程序执行完后programming.txt的内容
#I love programming.
#I love creating new games.
#I also love finding meaning in large datasets.
#I love creating apps that can run in a browser.
```

打开文件时指定了实参'a'，以便将内容附加到文件末尾，而不是覆盖文件原来的内容。又写入了两行，它们被添加到文件programming.txt末尾.

存储数据(选学)

接下来介绍一种文件格式json的相关使用方法

> (建议看看你们书上文件那一章是否讲到json文件,若没有,则可以跳过)

模块json 让你能够将简单的Python数据结构转储到文件中，并在程序再次运行时加载该文件中的数据。你还可以使用json 在Python程序之间分享数据。更重要的是，JSON数据格式并非Python专用的，这让你能够将以JSON格式存储的数据与使用其他编程语言的人分享。这是一种轻便而有用的格式，也易于学习。

> 注意　JSON（JavaScript Object Notation）格式最初是为JavaScript(一种脚本语言)开发的，但随后成了一种常见格式，被包括Python在内的众多语言采用。

我们来编写一个存储一组数的简短程序，再编写一个将这些数读取到内存中的程序。第一个程序将使用`json.dump()`来存储这组数，而第二个程序将使用`json.load()`。
函数`json.dump()`接受两个实参：要存储的数据，以及可用于存储数据的文件对象。下面演示了如何使用`json.dump()`来存储数字列表：

```
import json #导入模块json(模块是什么,会在下面的类当中了解到)

numbers = [2, 3, 5, 7, 11, 13]
filename = 'numbers.json'
with open(filename, 'w') as f:
    json.dump(numbers, f)
#文件numbers.json
#[2, 3, 5, 7, 11, 13]
```

先导入模块`json`，再创建一个数字列表。首先,指定了要将该数字列表存储到哪个文件中。通常使用文件扩展名`.json`来指出文件存储的数据为`JSON`格式。接下来，以写入模式打开这个文件，让`json`能够将数据写入其中。使用函数`json.dump()`将数字列表存储到文件`numbers.json`中。

下面再编写一个程序，使用`json.load()`将列表读取到内存中：

```
import json
filename = 'numbers.json'
with open(filename) as f:
    numbers = json.load(f)
print(numbers)
```

首先要确保读取的是前面写入的文件。这次以读取方式打开该文件，因为Python只需要读取它。然后,使用函数`json.load()`加载存储在`numbers.json`中的信息，并将其赋给变量numbers。最后，打印恢复的数字列表，看看是否与`number_writer.py`中创建的数字列表相同

类

类简介

面向对象编程是最有效的软件编写方法之一。在面向对象编程中，你编写表示现实世界中的事物和情景的类，并基于这些类来创建对象。编写类时，你定义一大类对象都有的通用行为。基于类创建对象 时，每个对象都自动具备这种通用行为，然后可根据需要赋予每个对象独特的个性。使用面向对象编程可模拟现实情景，其逼真程度达到了令人惊讶的地步。根据类来创建对象称为实例化，这让你能够使用类的实例.

创建类和使用类

使用类几乎可以模拟任何东西。下面来编写一个表示小狗的简单类Dog，它表示的不是特定的小狗，而是任何小狗。对于大多数宠物狗，我们都知道些什么呢？它们都有名字和年龄。我们还知道，大多数小狗还会蹲下和打滚。由于大多数小狗都具备上述两项信息（名字和年龄）和两种行为（蹲下和打滚），我们的Dog 类将包含它们。这个类让Python知道如何创建表示小狗的对象。编写这个类后，我们将使用它来创建表示特定小狗的实例。

根据Dog 类创建的每个实例都将存储名字和年龄，我们赋予了每条小狗蹲下(`sit()`)和打滚(`roll_over()`)的能力：

```
#dog.py
❶ class Dog:
❷   """一次模拟小狗的简单尝试。"""    #前面我们介绍过这种叫做文档注释
❸   def __init__(self, name, age):
        """初始化属性name和age。"""
❹       self.name = name
        self.age = age
❺   def sit(self):
        """模拟小狗收到命令时蹲下。"""
        print("{} is now sitting.".format(self.name))
    def roll_over(self):
        """模拟小狗收到命令时打滚。"""
        print("{} rolled over!".format(self.name))
```

看不懂?没关系我们会一一讲解上述代码.
这里需要注意的地方很多，但也不用担心，本节充斥着这样的结构，你有大把的机会熟悉它。❶处定义了一个名为Dog 的类。根据约定，在Python中，首字母大写的名称指的是类。这个类定义中没有圆括号，因为要从空白创建这个类。❷处编写了一个文档字符串，对这个类的功能做了描述。

方法`__init__()`

类中的函数称为方法 。你在前面学到的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。❸处的方法`__init__()`是一个特殊方法，每当你根据Dog 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，旨在避免Python默认方法与普通方法发生名称冲突。务必确保`__init__()`的两边都有两个下划线，否则当你使用类来创建实例时，将不会自动调用这个方法，进而引发难以发现的错误。我们将方法`__init__()`定义成包含三个形参：self 、name 和age 。在这个方法的定义中，形参self 必不可少，而且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调用这个方法来创建Dog 实例时，将自动传入实参self 。每个与实例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。创建Dog 实例时，Python将调用Dog 类的方法`__init__()`。我们将通过实参向Dog() 传递名字和年龄，self 会自动传递，因此不需要传递它。每当根据Dog 类创建实例时，都只需给最后两个形参（name 和age ）提供值。简单来说方法`__init__()`就是给你类里面的变量提供初值的.

❹处定义的两个变量都有前缀self 。以self 为前缀的变量可供类中的所有方法使用，可以通过类的任何实例来访问。`self.name = name`获取与形参name 相关联的值，并将其赋给变量name ，然后该变量被关联到当前创建的实例。`self.age= age`的作用与此类似。像这样可通过实例访问的变量称为属性 。

Dog 类还定义了另外两个方法：`sit()`和`roll_over()`（见❺）。这些方法执行时不需要额外的信息，因此它们只有一个形参self 。我们随后将创建的实例能够访问这些方法，换句话说，它们都会蹲下和打滚。当前，`sit()`和`roll_over()`所做的有限，只是打印一条消息，指出小狗正在蹲下或打滚。但可以扩展这些方法以模拟实际情况：如果这个类包含在一个计算机游戏中，这些方法将包含创建小狗蹲下和打滚动画效果的代码；如果这个类是用于控制机器狗的，这些方法将让机器狗做出蹲下和打滚的动作。

以上就是创建类的相关知识,那么我们下面来使用类,在使用类前,我们先要创建类实例.

```
class Dog:
"""一次模拟小狗的简单尝试。"""
    def __init__(self, name, age):
        """初始化属性name和age。"""
        self.name = name
        self.age = age
```

```
    def sit(self):
        """模拟小狗收到命令时蹲下。"""
        print("{} is now sitting.".format(self.name))
    def roll_over(self):
        """模拟小狗收到命令时打滚。"""
        print("{} rolled over!".format(self.name))

❶ my_dog = Dog('Willie', 6)
❷ print("My dog's name is {}.".format(self.name))
❸ print("My dog is {} years old.".format(self.age))
```

这这里使用的是前一个示例中编写的Dog 类。在❶处，让Python创建一条名字为'Willie' 、年龄为6 的小狗。遇到这行代码时，Python使用实参'Willie' 和6 调用Dog 类的方法 `__init__()` 。方法 `__init__()` 创建一个表示特定小狗的实例，并使用提供的值来设置属性name 和age 。接下来，Python返回一个表示这条小狗的实例，而我们将这个实例赋给了变量my_dog 。在这里，命名约定很有用：通常可认为首字母大写的名称（如Dog ）指的是类，而小写的名称（如my_dog ）指的是根据类创建的实例。

a. 访问属性

要访问实例的属性(属性,你可以把它理解为类里面的函数)，可使用句点表示法,这就是为什么之前使用方法时要使用句点的原因(之前的那些变量也可以当成一种特殊类,那些方法就是这些变量的属性)。❷处编写了如下代码来访问my_dog的属性name 的值： `my_dog.name`

句点表示法在Python中很常用，这种语法演示了Python如何获悉属性的值。在这里， Python先找到实例my_dog ， 再查找与该实例相关联的属性name 。在Dog 类中引用这个属性时，使用的是 `self.name` 。在❸处，使用同样的方法来获取属性age 的值。

```
#输出
#My dog's name is Willie.
#My dog is 6 years old.
```

b. 调用方法

根据Dog 类创建实例后，就能使用句点表示法来调用Dog 类中定义的任何方法了。下面来让小狗蹲下和打滚：

```
class Dog:
"""一次模拟小狗的简单尝试。"""
    def __init__(self, name, age):
        """初始化属性name和age。"""
        self.name = name
        self.age = age
    def sit(self):
        """模拟小狗收到命令时蹲下。"""
        print("{} is now sitting.".format(self.name))
    def roll_over(self):
        """模拟小狗收到命令时打滚。"""
        print("{} rolled over!".format(self.name))

my_dog = Dog('Willie', 6)
my_dog.sit()
my_dog.roll_over()
```

要调用方法，可指定实例的名称（这里是my_dog ）和要调用的方法，并用句点分隔。遇到代码`my_dog.sit()`时，Python在类Dog 中查找方法sit() 并运行其代码。Python以同样的方式解读代码`my_dog.roll_over()`。

Willie按我们的命令做了,所以会输出如下内容：

```
#Willie is now sitting.
#Willie rolled over!
```

这种语法很有用。如果给属性和方法指定了合适的描述性名称，如name 、age、`sit()`和`roll_over()`，即便是从未见过的代码块，我们也能够轻松地推断出它是做什么的。

c. 创建多个实例
可按需求根据类创建任意数量的实例。下面再创建一个名为your_dog 的小狗实例：

```
class Dog:
"""一次模拟小狗的简单尝试。"""
    def __init__(self, name, age):
        """初始化属性name和age。"""
        self.name = name
        self.age = age
    def sit(self):
        """模拟小狗收到命令时蹲下。"""
        print("{} is now sitting.".format(self.name))
    def roll_over(self):
        """模拟小狗收到命令时打滚。"""
        print("{} rolled over!".format(self.name))

my_dog = Dog('Willie', 6)
your_dog = Dog('Lucy', 3)
print("My dog's name is {}.".format(my_dog.age))
print("My dog is {} years old.".format(my_dog.age))
my_dog.sit()
print("\nYour dog's name is {}.".format(your_dog.name))
print("Your dog is {} years old.".format(your_dog.age))
your_dog.sit()
```

在本例中创建了两条小狗，分别名为Willie和Lucy。每条小狗都是一个独立的实例，有自己的一组属性，能够执行相同的操作：

```
#输出如下
#My dog's name is Willie.
#My dog is 6 years old.
#
#Willie is now sitting.
#Your dog's name is Lucy.
#Your dog is 3 years old.
#Lucy is now sitting.
```

即使给第二条小狗指定同样的名字和年龄，Python依然会根据Dog 类创建另一个实例。你可按需求根据一个类创建任意数量的实例，条件是将每个实例都存储在不同的变量中，或者占用列表或字典的不同位置。

可使用类来模拟现实世界中的很多情景。类编写好后，你的大部分时间将花在根据类创建的实例上。你需要执行的一个重要任务是修改实例的属性。可以直接修改实例的属性，也可以编写方法以特定的方式进行修改。

我们再来编写一个类,并使用它.下面来编写一个表示汽车的类。它存储了有关汽车的信息，还有一个汇总这些信息的方法：

```
class Car:
    """一次模拟汽车的简单尝试。"""
❶    def __init__(self, make, model, year):
        """初始化描述汽车的属性。"""
        self.make = make
        self.model = model
        self.year = year
❷   def get_descriptive_name(self):
        """返回整洁的描述性信息。"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()
❸ my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
```

在❶处，定义了方法`__init__()`。与前面的Dog 类中一样，这个方法的第一个形参为self 。该方法还包含另外三个形参：make 、model 和year 。方法`__init__()`接受这些形参的值，并将它们赋给根据这个类创建的实例的属性。创建新的Car 实例时，需要指定其制造商、型号和生产年份。

在❷处，定义了一个名为`get_descriptive_name()`的方法。它使用属性year、make 和model 创建一个对汽车进行描述的字符串，让我们无须分别打印每个属性的值。为在这个方法中访问属性的值，使用了`self.make`、`self.model`和`self.year`。在❸处，根据Car 类创建了一个实例，并将其赋给变量`my_new_car`。接下来，调用方法`get_descriptive_name()`，指出我们拥有一辆什么样的汽车：`2019 Audi A4`

为了让这个类更有趣，下面给它添加一个随时间变化的属性，用于存储汽车的总里程。

创建实例时，有些属性无须通过形参来定义，可在方法`__init__()`中为其指定默认值。
下面来添加一个名为`odometer_reading`的属性，其初始值总是为0。我们还添加了一个名为`read_odometer()`的方法，用于读取汽车的里程表：

```
class Car:
    def __init__(self, make, model, year):
        """初始化描述汽车的属性。"""
        self.make = make
        self.model = model
        self.year = year
❶       self.odometer_reading = 0
    def get_descriptive_name(self):
    """返回整洁的描述性信息。"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()
❷   def read_odometer(self):
        """打印一条指出汽车里程的消息。"""
        print(f"This car has {self.odometer_reading} miles on it.")
my_new_car = Car('audi', 'a4', 2019)
print(my_new_car.get_descriptive_name())
my_new_car.read_odometer()
```

现在，当Python调用方法`__init__()`来创建新实例时，将像前一个示例一样以属性的方式存储制造商、型号和生产年份。接下来，Python将创建一个名为`odometer_reading`的属性，并将其初始值设置为0（见❶）。在❷处，定义一个名为`read_odometer()`的方法，让你能够轻松地获悉汽车的里程 。

一开始汽车的里程为0：

```
#2019 Audi A4
#This car has 0 miles on it.
```

出售时里程表读数为0的汽车不多，因此需要一种方式来修改该属性的值。
我们能以三种方式修改属性的值：直接通过实例进行修改，通过方法进行设置，以及通过方法进行递增（增加特定的值）。下面依次介绍这些方式。

a. 直接修改属性的值

要修改属性的值，最简单的方式是通过实例直接访问它。下面的代码直接将里程表读数设置为23：

```
  class Car:              #这里的类定义和上面一样
      --snip--
  my_new_car = Car('audi', 'a4', 2019)
  print(my_new_car.get_descriptive_name())
❶ my_new_car.odometer_reading = 23
  my_new_car.read_odometer()
```

在❶处，使用句点表示法直接访问并设置汽车的属性`odometer_reading`。这行代码让Python在实例my_new_car 中找到属性`odometer_reading`，并将其值设置为23：

```
#输出
#2019 Audi A4
#This car has 23 miles on it.
```

有时候需要像这样直接访问属性，但其他时候需要编写对属性进行更新的方法。

b. 通过方法修改属性的值

如果有方法能替你更新属性，将大有裨益。这样就无须直接访问属性，而可将值传递给方法，由它在内部进行更新。下面的示例演示了一个名为`update_odometer()`的方法：

```
  class Car:
     --snip--
❶    def update_odometer(self, mileage):
         """将里程表读数设置为指定的值。"""
         self.odometer_reading = mileage
  my_new_car = Car('audi', 'a4', 2019)
  print(my_new_car.get_descriptive_name())
❷ my_new_car.update_odometer(23)
  my_new_car.read_odometer()
```

对Car 类所做的唯一修改是在❶处添加了方法`update_odometer()`。这个方法接受一个里程值，并将其赋给`self.odometer_reading`。在❷处，调用`update_odometer()`，并向它提供了实参23（该实参对应于方法定义中的形参mileage ）。它将里程表读数设置为23，而方法`read_odometer()`打印该读数：

```
#输出
#2019 Audi A4
#This car has 23 miles on it.
```

可对方法`update_odometer()`进行扩展，使其在修改里程表读数时做些额外的工作。下面来添加一些逻辑，禁止任何人将里程表读数往回调：

```
  class Car:
     --snip--
    def update_odometer(self, mileage):
      """
        将里程表读数设置为指定的值。
        禁止将里程表读数往回调。
        """
❶       if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
❷           print("You can't roll back an odometer!")
```

现在，`update_odometer()`在修改属性前检查指定的读数是否合理。如果新指定的里程（mileage）大于或等于原来的里程（`self.odometer_reading`），就将里程表读数改为新指定的里程（见❶）；否则发出警告，指出不能将里程表往回调（见❷）。

c. 通过方法对属性的值进行递增

有时候需要将属性值递增特定的量，而不是将其设置为全新的值。假设我们购买了一辆二手车，且从购买到登记期间增加了100英里的里程。下面的方法让我们能够传递这个增量，并相应地增大里程表读数：

```
  class Car:
        --snip--
        def update_odometer(self, mileage):
        --snip--
❶       def increment_odometer(self, miles):
            """将里程表读数增加指定的量。"""
            self.odometer_reading += miles
❷ my_used_car = Car('subaru', 'outback', 2015)
  print(my_used_car.get_descriptive_name())
❸ my_used_car.update_odometer(23_500)
  my_used_car.read_odometer()
❹ my_used_car.increment_odometer(100)
  my_used_car.read_odometer()
```

在❶处，新增的方法increment_odometer() 接受一个单位为英里的数，并将其加入`self.odometer_reading`中。在❷处，创建一辆二手车my_used_car 。在❸处，调用方法`update_odometer()`并传入23_500 ，将这辆二手车的里程表读数设置为23 500。在❹处，调用`increment_odometer()`并传入100 ，以增加从购买到登记期间行驶的100英里：

```
#2015 Subaru Outback
#This car has 23500 miles on it.
#This car has 23600 miles on it.
```

你可以轻松地修改这个方法，以禁止增量为负值，从而防止有人利用它来回调里程表。

> 注意　你可以使用类似于上面的方法来控制用户修改属性值（如里程表读数）的方式，但能够访问程序的人都可以通过直接访问属性来将里程表修改为任何值。要确保安全，除了进行类似于前面的基本检查外，还需特别注意细节。

继承

编写类时，并非总是要从空白开始。如果要编写的类是另一个现成类的特殊版本，可使用继承 。一个类继承 另一个类时，将自动获得另一个类的所有属性和方法。原有的类称为父类 ，而新类称为子类 。子类继承了父类的所有属性和方法，同时还可以定义自己的属性和方法。

子类的方法`__init__()`

在既有类的基础上编写新类时，通常要调用父类的方法`__init__()`。这将初始化在父类`__init__()`方法中定义的所有属性，从而让子类包含这些属性。

例如，下面来模拟电动汽车。电动汽车是一种特殊的汽车，因此可在前面创建的Car类的基础上创建新类ElectricCar 。这样就只需为电动汽车特有的属性和行为编写代码。下面来创建ElectricCar 类的一个简单版本，它具备Car 类的所有功能：

```
❶ class Car:
    """一次模拟汽车的简单尝试。"""
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0
    def get_descriptive_name(self):
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()
    def read_odometer(self):
        print(f"This car has {self.odometer_reading} miles on it.")
    def update_odometer(self, mileage):
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")
    def increment_odometer(self, miles):
        self.odometer_reading += miles
❷ class ElectricCar(Car):
      """电动汽车的独特之处。"""
❸     def __init__(self, make, model, year):
       """初始化父类的属性。"""
❹       super().__init__(make, model, year)
❺ my_tesla = ElectricCar('tesla', 'model s', 2019)
  print(my_tesla.get_descriptive_name())
```

首先是Car 类的代码（见❶）。创建子类时，父类必须包含在当前文件中，且位于子类前面。在❷处，定义了子类ElectricCar 。定义子类时，必须在圆括号内指定父类的名称。方法`__init__()`接受创建Car实例所需的信息（见❸）。

❹处的`super()`是一个特殊函数，让你能够调用父类的方法。这行代码让Python调用Car 类的方法`__init__()`，让ElectricCar 实例包含这个方法中定义的所有属性。父类也称为超类 （superclass），名称super 由此而来。

为测试继承能够正确地发挥作用，我们尝试创建一辆电动汽车，但提供的信息与创建普通汽车时相同。在❺处，创建ElectricCar 类的一个实例，并将其赋给变量my_tesla 。这行代码调用ElectricCar 类中定义的方法 `__init__()` ，后者让Python调用父类Car 中定义的方法 `__init__()` 。我们提供了实参'tesla'、'model s' 和2019 。

除方法 `__init__()` 外，电动汽车没有其他特有的属性和方法。当前，我们只想确认电动汽车具备普通汽车的行为： `#输出:2019 Tesla Model S`

ElectricCar 实例的行为与Car 实例一样，现在可以开始定义电动汽车特有的属性和方法了。

给子类定义属性和方法

让一个类继承另一个类后，就可以添加区分子类和父类所需的新属性和新方法了。下面来添加一个电动汽车特有的属性（电瓶），以及一个描述该属性的方法。我们将存储电瓶容量，并编写一个打印电瓶描述的方法：

```
  class Car:
    --snip--
  class ElectricCar(Car):
      """电动汽车的独特之处。"""
      def __init__(self, make, model, year):
      """
      初始化父类的属性。
      再初始化电动汽车特有的属性。
      """
      super().__init__(make, model, year)
❶     self.battery_size = 75
❷ def describe_battery(self):
      """打印一条描述电瓶容量的消息。"""
      print(f"This car has a {self.battery_size}-kWh battery.")
  my_tesla = ElectricCar('tesla', 'model s', 2019)
  print(my_tesla.get_descriptive_name())
  my_tesla.describe_battery()
```

在❶处，添加了新属性 `self.battery_size` ，并设置其初始值（75 ）。根据ElectricCar 类创建的所有实例都将包含该属性，但所有Car 实例都不包含它。
在❷处，还添加了一个名为 `describe_battery()` 的方法，打印有关电瓶的信息。调用这个方法时，将看到一条电动汽车特有的描述：

```
#输出
#2019 Tesla Model S
#This car has a 75-kWh battery.
```

对于ElectricCar 类的特殊程度没有任何限制。模拟电动汽车时，可根据所需的准确程度添加任意数量的属性和方法。如果一个属性或方法是任何汽车都有的，而不是电动汽车特有的，就应将其加入到Car 类而非ElectricCar 类中。这样，使用Car 类的人将获得相应的功能，而ElectricCar 类只包含处理电动汽车特有属性和行为的代码。

重写父类的方法

对于父类的方法，只要它不符合子类模拟的实物的行为，都可以进行重写。为此，可在子类中定义一个与要重写的父类方法同名的方法。这样，Python将不会考虑这个父类方法，而只关注你在子类中定义的相应方法。假设Car 类有一个名为`fill_gas_tank()`的方法，它对全电动汽车来说毫无意义，因此你可能想重写它。下面演示了一种重写方式：

```
class ElectricCar(Car):
    --snip--
def fill_gas_tank(self):
    """电动汽车没有油箱。"""
    print("This car doesn't need a gas tank!")
```

现在，如果有人对电动汽车调用方法`fill_gas_tank()`，Python将忽略Car 类中的方法`fill_gas_tank()`，转而运行上述代码。使用继承时，可让子类保留从父类那里继承而来的精华，并剔除不需要的糟粕。

将实例用作属性

使用代码模拟实物时，你可能会发现自己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提取出来，作为一个独立的类。可以将大型类拆分成多个协同工作的小类。

例如，不断给ElectricCar 类添加细节时，我们可能发现其中包含很多专门针对汽车电瓶的属性和方法。在这种情况下，可将这些属性和方法提取出来，放到一个名为Battery 的类中，并将一个Battery 实例作为ElectricCar 类的属性：

```
  class Car:
      --snip--
❶ class Battery:
      """一次模拟电动汽车电瓶的简单尝试。"""
❷ def __init__(self, battery_size=75):
      """初始化电瓶的属性。"""
      self.battery_size = battery_size
❸ def describe_battery(self):
      """打印一条描述电瓶容量的消息。"""
      print(f"This car has a {self.battery_size}-kWh battery.")
  class ElectricCar(Car):
      """电动汽车的独特之处。"""
      def __init__(self, make, model, year):
      """
      初始化父类的属性。
      再初始化电动汽车特有的属性。
      """
      super().__init__(make, model, year)
❹       self.battery = Battery()
my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
```

❶处定义一个名为Battery 的新类，它没有继承任何类。❷处的方法`__init__()`除self 外，还有另一个形参`battery_size`。这个形参是可选的：如果没有给它提供值，电瓶容量将被设置为75。方法`describe_battery()`也移到了这个类中（见❸）。

在ElectricCar 类中，添加了一个名为`self.battery`的属性（见❹）。这行代码让Python创建一个新的Battery 实例（因为没有指定容量，所以为默认值75），并将该实例赋给属性`self.battery`。每当方法`__init__()`被调用时，都将执行该操作，因此现在每个ElectricCar 实例都包含一个自动创建的Battery 实例。

我们创建一辆电动汽车，并将其赋给变量`my_tesla`。描述电瓶时，需要使用电动汽车的属性`battery`：

```
#输出
#my_tesla.battery.describe_battery()
```

这行代码让Python在实例`my_tesla`中查找属性`battery`，并对存储在该属性中的Battery 实例调用方法`describe_battery()`。
输出与你在前面看到的相同：

```
#2019 Tesla Model S
#This car has a 75-kWh battery.
```

这看似做了很多额外的工作，但是现在想多详细地描述电瓶都可以，且不会导致ElectricCar 类混乱不堪。下面再给Battery 类添加一个方法，它根据电瓶容量报告汽车的续航里程：

```
  class Car:
      --snip--
  class Battery:
      --snip--
❶ def get_range(self):
      """打印一条消息，指出电瓶的续航里程。"""
      if self.battery_size == 75:
          range = 260
      elif self.battery_size == 100:
          range = 315
          print(f"This car can go about {range} miles on a full charge.")
  class ElectricCar(Car):
      --snip--
  my_tesla = ElectricCar('tesla', 'model s', 2019)
  print(my_tesla.get_descriptive_name())
  my_tesla.battery.describe_battery()
❷ my_tesla.battery.get_range()
```

❶处新增的方法`get_range()`做了一些简单的分析：如果电瓶的容量为75 kWh，就将续航里程设置为260英里；如果容量为100 kWh，就将续航里程设置为315英里，然后报告这个值。为使用这个方法，也需要通过汽车的属性`battery`来调用（见❷）。

输出指出了汽车的续航里程（这取决于电瓶的容量）：

```
#2019 Tesla Model S
#This car has a 75-kWh battery.
#This car can go about 260 miles on a full charge.
```

导入类

随着不断给类添加功能，文件可能变得很长，即便妥善地使用了继承亦如此。为遵循Python的总体理念，应让文件尽可能整洁。Python在这方面提供了帮助，允许将类存储在模块中，然后在主程序中导入所需的模块。

导入单个类

下面来创建一个只包含Car 类的模块。这让我们面临一个微妙的命名问题：在本章中已经有一个名为`car.py`的文件，但这个模块也应命名为`car.py`，因为它包含表示汽车的代码。我们将这样解决这个命名问题：将Car 类存储在一个名为car.py的模块中，该模块将覆盖前面使用的文件`car.py`。从现在开始，使用该模块的程序都必须使用更具体的文件名，如`my_car.py`。下面是模块`car.py`，其中只包含Car 类的代码：

```
#car.py
❶ """一个可用于表示汽车的类。"""
class Car:
    """一次模拟汽车的简单尝试。"""
    def __init__(self, make, model, year):
        """初始化描述汽车的属性。"""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0
    def get_descriptive_name(self):
        """返回整洁的描述性名称。"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()
    def read_odometer(self):
        """打印一条消息，指出汽车的里程。"""
        print(f"This car has {self.odometer_reading} miles on it.")
    def update_odometer(self, mileage):
        """
        将里程表读数设置为指定的值。
        拒绝将里程表往回调。
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")
    def increment_odometer(self, miles):
        """将里程表读数增加指定的量。"""
        self.odometer_reading += miles
```

❶处包含一个模块级文档字符串，对该模块的内容做了简要的描述。你应为自己创建的每个模块编写文档字符串。
下面来创建另一个文件`my_car.py`，在其中导入Car 类并创建其实例：

```
#my_car.py
❶ from car import Car
  my_new_car = Car('audi', 'a4', 2019)
  print(my_new_car.get_descriptive_name())
  my_new_car.odometer_reading = 23
  my_new_car.read_odometer()
```

❶处的import 语句让Python打开模块car 并导入其中的Car 类。这样，我们就可以使用Car 类，就像它是在这个文件中定义的一样。输出与我们在前面看到的一样：

```
#2019 Audi A4
#This car has 23 miles on it.
```

导入类是一种有效的编程方式。如果这个程序包含整个Class 类，它该有多长啊！通过将这个类移到一个模块中并导入该模块，依然可以使用其所有功能，但主程序文件变得整洁而易于阅读了。这还让你能够将大部分逻辑存储在独立的文件中。确定类像你希望的那样工作后，就可以不管这些文件，而专注于主程序的高级逻辑了。

在一个模块中存储多个类

虽然同一个模块中的类之间应存在某种相关性，但可根据需要在一个模块中存储任意数量的类。Battery 类和ElectricCar 类都可帮助模拟汽车，下面将它们都加入模块`car.py`中：

```
#car.py
"""一组用于表示燃油汽车和电动汽车的类。"""
class Car:
    --snip--
class Battery:
    """一次模拟电动汽车电瓶的简单尝试。"""
    def __init__(self, battery_size=75):
        """初始化电瓶的属性。"""
        self.battery_size = battery_size
    def describe_battery(self):
        """打印一条描述电瓶容量的消息。"""
        print(f"This car has a {self.battery_size}-kWh battery.")
    def get_range(self):
        """打印一条描述电瓶续航里程的消息。"""
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315
        print(f"This car can go about {range} miles on a full charge.")
class ElectricCar(Car):
        """模拟电动汽车的独特之处。"""
    def __init__(self, make, model, year):
        """
        初始化父类的属性。
        再初始化电动汽车特有的属性。
        """
        super().__init__(make, model, year)
        self.battery = Battery()
```

现在，可以新建一个名为`my_electric_car.py`的文件，导入ElectricCar 类，并创建一辆电动汽车了：

```
#my_electric_car.py
from car import ElectricCar
my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

输出与我们在前面看到的相同，但大部分逻辑隐藏在一个模块中：

```
#2019 Tesla Model S
#This car has a 75-kWh battery.
#This car can go about 260 miles on a full charge.
```

从一个模块中导入多个类

可根据需要在程序文件中导入任意数量的类。如果要在同一个程序中创建普通汽车和电动汽车，就需要将Car 类和ElectricCar 类都导入：

```
#my_cars.py
❶ from car import Car, ElectricCar
❷ my_beetle = Car('volkswagen', 'beetle', 2019)
  print(my_beetle.get_descriptive_name())
❸ my_tesla = ElectricCar('tesla', 'roadster', 2019)
  print(my_tesla.get_descriptive_name())
```

在❶处从一个模块中导入多个类时，用逗号分隔了各个类。导入必要的类后，就可根据需要创建每个类的任意数量实例。
在本例中，在❷处创建了一辆大众甲壳虫普通汽车，并在❸处创建了一辆特斯拉Roadster电动汽车：

```
#2019 Volkswagen Beetle
#2019 Tesla Roadster
```

导入整个模块

还可以导入整个模块，再使用句点表示法访问需要的类。这种导入方式很简单，代码也易于阅读。因为创建类实例的代码都包含模块名，所以不会与当前文件使用的任何名称发生冲突。下面的代码导入整个car 模块，并创建一辆普通汽车和一辆电动汽车：

```
#my_cars.py
❶ import car
❷ my_beetle = car.Car('volkswagen', 'beetle', 2019)
  print(my_beetle.get_descriptive_name())
❸ my_tesla = car.ElectricCar('tesla', 'roadster', 2019)
  print(my_tesla.get_descriptive_name())
```

在❶处，导入了整个car 模块。接下来，使用语法 module_name.ClassName访问需要的类。像前面一样，在❷处创建一辆大众甲壳虫汽车，并在❸处创建一辆特斯拉Roadster汽车。

导入模块中的所有类

要导入模块中的每个类，可使用下面的语法：`from module_name import *`

不推荐使用这种导入方式，原因有二。第一，如果只看文件开头的import 语句，就能清楚地知道程序使用了哪些类，将大有裨益。然而这种导入方式没有明确地指出使用了模块中的哪些类。第二，这种方式还可能引发名称方面的迷惑。如果不小心导入了一个与程序文件中其他东西同名的类，将引发难以诊断的错误。这里之所以介绍这种导入方式，是因为虽然不推荐使用，但你可能在别人编写的代码中见到它。

需要从一个模块中导入很多类时，最好导入整个模块，并使用module_name.ClassName 语法来访问类。这样做时，虽然文件开头并没有列出用到的所有类，但你清楚地知道在程序的哪些地方使用了导入的模块。这也避免了导入模块中的每个类可能引发的名称冲突。

在一个模块中导入另一个模块

有时候，需要将类分散到多个模块中，以免模块太大或在同一个模块中存储不相关的类。将类存储在多个模块中时，你可能会发现一个模块中的类依赖于另一个模块中的类。在这种情况下，可在前一个模块中导入必要的类。下面将Car 类存储在一个模块中，并将ElectricCar 类和Battery 类存储在另一个模块中。将第二个模块命名为electric_car.py（这将覆盖前面创建的文件electric_car.py），并将Battery 类和ElectricCar 类复制到这个模块中：

```
#electric_car.py
  """一组可用于表示电动汽车的类。"""
❶ from car import Car
  class Battery:
      --snip--
  class ElectricCar(Car):
      --snip--
```

ElectricCar 类需要访问其父类Car ，因此在❶处直接将Car 类导入该模块中。如果忘记了这行代码，Python将在我们试图创建ElectricCar 实例时引发错误。还需要更新模块car ，使其只包含Car 类：

```
#car.py
"""一个可用于表示汽车的类。"""
class Car:
    --snip--
```

现在可以分别从每个模块中导入类，以根据需要创建任何类型的汽车了：

```
#my_cars.py
❶ from car import Car
  from electric_car import ElectricCar
  my_beetle = Car('volkswagen', 'beetle', 2019)
  print(my_beetle.get_descriptive_name())
  my_tesla = ElectricCar('tesla', 'roadster', 2019)
  print(my_tesla.get_descriptive_name())
```

在❶处，从模块car 中导入了Car 类，并从模块electric_car 中导入ElectricCar 类。接下来，创建了一辆普通汽车和一辆电动汽车。这两种汽车都被正确地创建出来了：

```
#2019 Volkswagen Beetle
#2019 Tesla Roadster
```

使用别名

使用模块来组织项目代码时，别名大有裨益。导入类时，也可为其指定别名。例如，要在程序中创建大量电动汽车实例，需要反复输入ElectricCar ，非常烦琐。为避免这种烦恼，可在import 语句中给ElectricCar 指定一个别名：`from electric_car import ElectricCar as EC` 现在每当需要创建电动汽车实例时，都可使用这个别名：`my_tesla = EC('tesla', 'roadster', 2019)`