

笔记本： 小雨成才自学系列笔记
创建时间： 2018/9/25 10:19
作者： 825365321@qq.com

更新时间： 2018/9/29 19:43

第一章 欢迎来到Python世界

第一节 什么是python

python：强大的，可以进行系统调用的解释型脚本语言。这意味着开发环节中没有了编译这个环节。

特点：

1. 高级：
2. 面向对象
3. 可升级
4. 可扩展
5. 可移植性
6. 易学、易读、易维护、健壮性

源文件的扩展名通常都是 .py

C语言是用来编写操作系统的贴近硬件的语言，所以，C语言适合开发那些追求运行速度、充分发挥硬件性能的程序。而Python是用来编写应用程序的高级编程语言。

缺点：

1. 第一个缺点就是运行速度慢，和C程序相比非常慢，因为Python是解释型语言，你的代码在执行时会一行一行地翻译成CPU能理解的机器码，这个翻译过程非常耗时，所以很慢。而C程序是运行前直接编译成CPU能执行的机器码，所以非常快。
2. 第二个缺点就是代码不能加密。如果要发布你的Python程序，实际上就是发布源代码，这一点跟C语言不同，C语言不用发布源代码，只需要把编译后的机器码（也就是你在Windows上常见的xxx.exe文件）发布出去。要从机器码反推出C代码是不可能的，所以，凡是编译型的语言，都没有这个问题，而解释型的语言，则必须把源码发布出去。

Python是跨平台的，它可以运行在Windows、Mac和各种Linux/Unix系统上。在Windows上写Python程序，放到Linux上也是能够运行的。

第二节 第一个python程序

输入和输出函数 (I/O) : input() print()

```
name = input('please input your name: ')
print('hello,', name)
```

运行python程序：



扩展章节：廖雪峰关于python的教程

第一节：Python基础知识

1.1 数据类型和变量：整数、浮点数、字符串（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。如果字符串内部既包含'又包含"怎么办？可以用转义字符\来标识，比如：

```
'I\'m \'OK\'!'
```

表示的是

```
I'm "OK"!
```

此外，Python允许用'''...'''的格式表示多行内容，如：

```
>>> print("line1
... line2
... line3")
line1
line2
line3
```

多行字符串'''...'''还可以在前面加上r使用

```
print(r"hello,\n
world")
hello,\n
world
```

布尔值：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
True
>>> False
False
>>> 3 > 2
True
>>> 3 > 5
False
```

布尔值可以用and、or和not运算
and

```
>>> True and True
True
>>> True and False
```

```
False  
>>> False and False  
False  
>>> 5 > 3 and 3 > 1  
True
```

or

```
>>> True or True  
True  
>>> True or False  
True  
>>> False or False  
False  
>>> 5 > 3 or 1 > 3  
True
```

not

```
>>> not True  
False  
>>> not False  
True  
>>> not 1 > 2  
True
```

布尔值经常用在条件判断中，比如：

```
if age >= 18:  
    print('adult')  
else:  
    print('teenager')
```

空值：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。

```
a = 123 # a是整数，此处注意在python中注释使用#表示  
print(a)  
a = 'ABC' # a变为字符串  
print(a)
```

```
123  
ABC
```

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。python是不需要定义变量的，直接用就可以了。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。例如Java是静态语言，赋值语句如下（// 表示注释）：

```
int a = 123; // a是整数类型变量  
a = "ABC"; // 错误：不能把字符串赋给整型变量
```

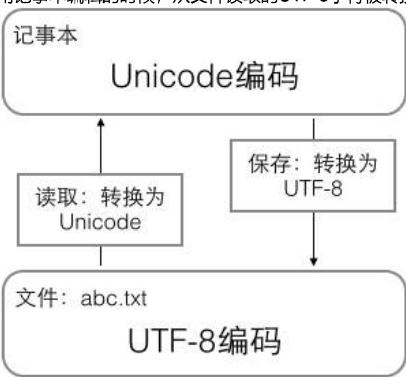
最后，关于除法计算，有：/除法计算结果是浮点数。还有一种除法是//，称为地板除，两个整数的除法仍然是整数。

```
>>> 10 / 3  
3.333333333333335  
>>> 9 / 3  
3.0  
>>> 10 // 3  
3  
>>> 10 % 3  
1
```

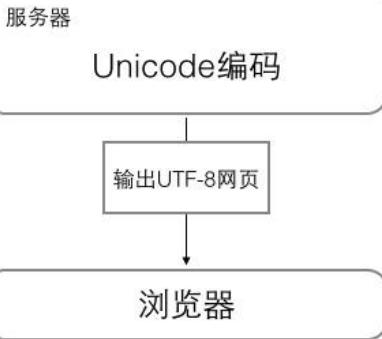
注意：Python的整数没有大小限制，而某些语言的整数根据其存储长度是有大小限制的，例如Java对32位整数的范围限制在-2147483648-2147483647。Python的浮点数也没有大小限制，但是超出一定范围就直接表示为inf（无限大）。

1.2字符串和编码： Unicode把所有语言都统一到一套编码里，这样就不会再有乱码问题了。ASCII编码是1个字节，而Unicode编码通常是2个字节。
总结一下现在计算机系统通用的字符编码工作方式：

用记事本编辑的时候，从文件读取的UTF-8字符被转换为Unicode字符到内存里，编辑完成后，保存的时候再把Unicode转换为UTF-8保存到文件：



浏览网页的时候，服务器会把动态生成的Unicode内容转换为UTF-8再传输到浏览器：



在最新的Python 3版本中，字符串是以Unicode编码的，也就是说，Python的字符串支持多语言，例如：

```
>>> print('包含中文的str')
包含中文的str
```

`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：order? character?

```
>>> ord('A')
65
>>> ord('中')
20013
>>> chr(66)
'B'
>>> chr(25991)
'文'
```

Python对bytes类型的数据用带b前缀的单引号或双引号表示。

```
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，例如：

```
>>> 'ABC'.encode('ascii')
b'ABC'
>>> '中文'.encode('utf-8')
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)
```

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。

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')
'ABC'
>>> b'\xe4\xb8\xad\xe6\x96\x87'.decode('utf-8')
'中文'
```

如果bytes中包含无法解码的字节，`decode()`方法会报错：

```
>>> 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
```

如果bytes中只有一小部分无效的字节，可以传入`errors='ignore'`忽略错误的字节：

```
>>> b'\xe4\xb8\xad\xff'.decode('utf-8', errors='ignore')
'中'
```

要计算str包含多少个字符，可以用`len()`函数：

```
>>> len('ABC')
3
>>> len('中文')
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')
3
>>> len(b'\xe4\xb8\xad\xe6\x96\x87')
6
>>> len('中文'.encode('utf-8'))
6
```

可见，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。

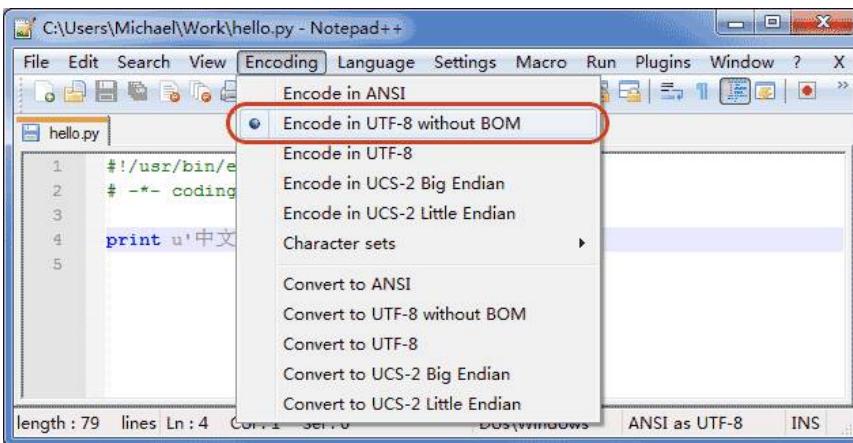
在操作字符串时，我们经常遇到str和bytes的互相转换。为了避免乱码问题，应当始终坚持使用UTF-8编码对str和bytes进行转换。

由于Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行：

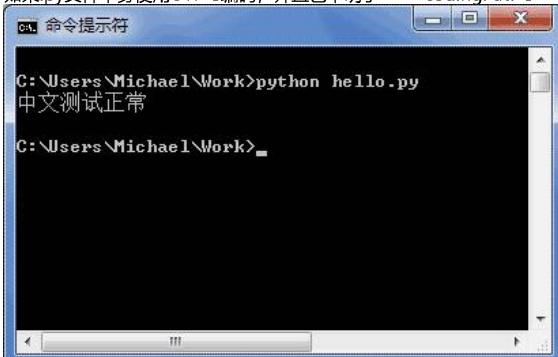
```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

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。

申明了UTF-8编码并不意味着你的.py文件就是UTF-8编码的，必须并且要确保文本编辑器正在使用UTF-8 without BOM编码：



如果.py文件本身使用UTF-8编码，并且也申明了# -*- coding: utf-8 -*-，打开命令提示符测试就可以正常显示中文：



格式化#下面的例子均需要加print()才能输出，原著略写了。

在Python中，采用的格式化方式和C语言是一致的，用%实现，举例如下：

```
>>> 'Hello, %s' % 'world'
'Hello, world'
>>> 'Hi, %s, you have $%d.' % ('Michael', 1000000)
'Hi, Michael, you have $1000000.'
```

在字符串内部，%s表示用字符串替换，%d表示用整数替换，有几个%?占位符，后面就跟几个变量或者值，顺序要对应好。如果只有一个%，括号可以省略。
常用的占位符有：

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

其中，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)
3-01
3.14
```

如果你不太确定应该用什么，%s永远起作用，它会把任何数据类型转换为字符串：

```
>>> 'Age: %s. Gender: %s' % (25, True)
'Age: 25. Gender: True'
```

有些时候，字符串里面的%是一个普通字符怎么办？这个时候就需要转义，用%%来表示一个%：

```
>>> 'growth rate: %d %%' % 7
'growth rate: 7 %'
```

format() 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('小明', 17.125)
'Hello, 小明, 成绩提升了 17.1%'
```

1.3 使用list和tuple (注意类比数组)

list：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。

比如，列出班里所有同学的名字，就可以用一个list表示：

```
>>> classmates = ['Michael', 'Bob', 'Tracy']
>>> classmates
['Michael', 'Bob', 'Tracy']

>>> len(classmates)
3

>>> classmates[0]
'Michael'
>>> classmates[1]
'Bob'
>>> classmates[2]
'Tracy'
>>> classmates[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

如果要取最后一个元素，除了计算索引位置外，还可以用-1做索引，直接获取最后一个元素，以此类推，可以获取倒数第2个、倒数第3个：

```
>>> classmates[-1]
'Tracy'
>>> classmates[-2]
'Bob'
>>> classmates[-3]
'Michael'
>>> classmates[-4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

list是一个可变的有序表，所以，可以往list中追加元素到末尾：

```
>>> classmates.append('Adam')
>>> classmates
['Michael', 'Bob', 'Tracy', 'Adam']
```

也可以把元素插入到指定的位置，比如索引为1的位置：

```
>>> classmates.insert(1, 'Jack')
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy', 'Adam']
```

要删除list末尾的元素，用pop()方法：

```
>>> classmates.pop()
'Adam'
>>> classmates
['Michael', 'Jack', 'Bob', 'Tracy']
```

要删除指定位置的元素，用pop(i)方法，其中i是索引位置：

```
>>> classmates.pop(1)
'Jack'
>>> classmates
['Michael', 'Bob', 'Tracy']
```

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[1] = 'Sarah'
>>> classmates
['Michael', 'Sarah', 'Tracy']
```

list里面的元素的数据类型也可以不同，比如：

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

list元素也可以是另一个list，比如：

```
>>> s = ['python', 'java', ['asp', 'php'], 'scheme']
>>> len(s)
4
```

要注意s只有4个元素，其中s[2]又是一个list，如果拆开写就更容易理解了：

```
>>> p = ['asp', 'php']
>>> s = ['python', 'java', p, 'scheme']
```

要拿到'php'可以写p[1]或者s[2][1]，因此s可以看成是一个二维数组，类似的还有三维、四维.....数组，不过很少用到。

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)
0
```

tuple: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，比如同样是列出同学的名字：

```
>>> classmates = ('Michael', 'Bob', 'Tracy')
```

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，你可以正常地使用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。如果可能，能用tuple代替list就尽量用tuple。

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，比如：

```
>>> t = (1, 2)
>>> t
(1, 2)
```

如果要定义一个空的tuple，可以写成():

```
>>> t = ()
>>> t
()
```

但是，要定义一个只有1个元素的tuple，如果你这么定义：

```
>>> t = (1)
>>> 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。

所以，只有1个元素的tuple定义时必须加一个逗号，来消除歧义：

```
>>> t = (1,)
>>> t
(1,)
```

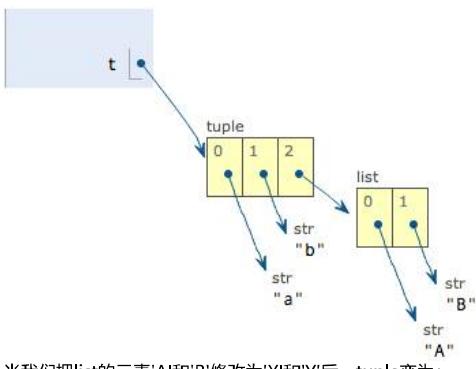
Python在显示只有1个元素的tuple时，也会加一个逗号，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”tuple：

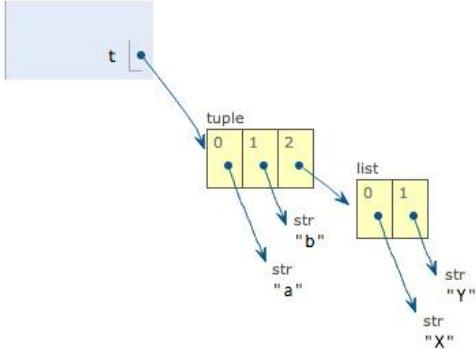
```
>>> t = ('a', 'b', ['A', 'B'])
>>> t[2][0] = 'X'
>>> t[2][1] = 'Y'
>>> t
('a', 'b', ['X', 'Y'])
```

这个tuple定义的时候有3个元素，分别是'a', 'b'和一个list。不是说tuple一旦定义后就不可变了吗？怎么后来又变了？

别急，我们先看看定义的时候tuple包含的3个元素：



当我们把list的元素'A'和'B'修改为'X'和'Y'后, tuple变为:



表面上看, tuple的元素确实变了, 但其实变的不是tuple的元素, 而是list的元素。tuple一开始指向的list并没有改成别的list, 所以, tuple所谓的“不变”是说, tuple的每个元素, 指向永远不变。即指向'a', 就不能改成指向'b', 指向一个list, 就不能改成指向其他对象, 但指向的这个list本身是可变的!

理解了“指向不变”后, 要创建一个内容也不变的tuple怎么做? 那就必须保证tuple的每一个元素本身也不能变。

1.4 条件判断

计算机之所以能做很多自动化的任务, 因为它可以自己做条件判断。

比如, 输入用户年龄, 根据年龄打印不同的内容, 在Python程序中, 用if语句实现:

```
age = 20
if age >= 18:
    print('your age is', age)
    print('adult')
```

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')
```

注意不要少写了冒号:。

当然上面的判断是很粗略的, 完全可以用elif做更细致的判断:

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

elif是else if的缩写, 完全可以有多个elif, 所以if语句的完整形式就是:

```
if <条件判断1>:
    <执行1>
elif <条件判断2>:
    <执行2>
elif <条件判断3>:
    <执行3>
else:
    <执行4>
```

if判断条件还可以简写, 比如写:

```
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, 否则为False。

再说 input

最后看一个有问题的条件判断。很多同学会用input()读取用户的输入, 这样可以自己输入, 程序运行得更有意思:

```
birth = input('birth: ')
if birth < 2000:
    print('00前')
else:
    print('00后')
```

输入1982, 结果报错:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
```

这是因为input()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。Python提供了int()函数来完成这件事情:

```
s = input('birth: ')
birth = int(s)
if birth < 2000:
    print('00前')
```

```
else:  
    print('00后')
```

再次运行，就可以得到正确地结果。但是，如果输入abc呢？又会得到一个错误信息：

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: invalid literal for int() with base 10: 'abc'
```

原来int()函数发现一个字符串并不是合法的数字时就会报错，程序就退出了。

如何检查并捕获程序运行期的错误呢？后面的错误和调试会讲到。

小知识：平方的表示方法： $2^*5=32$ ，表示2的5次方，同pow(2,5)

1.5 循环

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。

Python的循环有两种，一种是for...in循环，依次把list或tuple中的每个元素迭代出来，看例子：

```
names = ['Michael', 'Bob', 'Tracy']  
for name in names:  
    print(name)
```

执行这段代码，会依次打印names的每一个元素：

```
Michael  
Bob  
Tracy
```

所以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。

再比如我们想计算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)
```

如果要计算1-100的整数之和，从1写到100有点困难，幸好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。比如range(5)生成的序列是从0开始小于5的整数：

```
>>> list(range(5))  
[0, 1, 2, 3, 4]
```

range(101)就可以生成0-100的整数序列，计算如下：

```
sum = 0  
for x in range(101):  
    sum = sum + x  
print(sum)
```

5050

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。比如我们要计算100以内所有奇数之和，可以用while循环实现：

```
sum = 0  
n = 99  
while n > 0:  
    sum = sum + n  
    n = n - 2 # 此处可以看出python的特点，必须保持格式的对齐，缩进的格式不同，会运行出不同的结果  
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，循环退出。

break

在循环中，break语句可以提前退出循环。例如，本来要循环打印1~100的数字：

```
n = 1  
while n <= 100:  
    print(n)  
    n = n + 1  
print('END')
```

上面的代码可以打印出1~100。

如果要提前结束循环，可以用break语句：

```
n = 1  
while n <= 100:  
    if n > 10: # 当n = 11时，条件满足，执行break语句  
        break # break语句会结束当前循环  
    print(n)  
    n = n + 1  
print('END')
```

执行上面的代码可以看到，打印出1~10后，紧接着打印END，程序结束。

可见break的作用是提前结束循环。

continue

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  
    print(n)
```

上面的程序可以打印出1~10。但是，如果我们想只打印奇数，可以用continue语句跳过某些循环：

```
n = 0  
while n < 10:  
    n = n + 1  
    if n % 2 == 0: # 如果n是偶数，执行continue语句  
        continue # continue语句会直接继续下一轮循环，后续的print()语句不会执行  
    print(n)
```

执行上面的代码可以看到，打印的不再是1~10，而是1, 3, 5, 7, 9。

可见continue的作用是提前结束本轮循环，并直接开始下一轮循环。

循环是让计算机做重复任务的有效的方法。

break语句可以在循环过程中直接退出循环，而continue语句可以提前结束本轮循环，并直接开始下一轮循环。这两个语句通常都必须配合if语句使用。

要特别注意，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。

有些时候，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。

1.6 使用dict和set

dict
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。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']
95
```

为什么dict查找速度这么快？因为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。

你可以猜到，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['Adam'] = 67
>>> d['Adam']
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冲掉：

```
>>> d['Jack'] = 90
>>> d['Jack']
90
>>> d['Jack'] = 88
>>> d['Jack']
88
```

如果key不存在，dict就会报错：

```
>>> d['Thomas']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Thomas'
```

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：

```
>>> 'Thomas' in d
False
```

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.get('Thomas')
>>> d.get('Thomas', -1)
-1
```

注意：返回None的时候Python的交互环境不显示结果。

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

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

请务必注意，dict内部存放的顺序和key放入的顺序是没有关系的。

和list比较，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。

dict可以用在需要高速查找的很多地方，在Python代码中几乎无处不在，正确使用dict非常重要，需要牢记的第一条就是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

set和dict类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在set中，没有重复的key。

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
{1, 2, 3}
```

注意，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
{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.add(4)
>>> s
{1, 2, 3, 4}
>>> s.add(4)
>>> s
{1, 2, 3, 4}
```

通过remove(key)方法可以删除元素：

```
>>> s.remove(4)
>>> s
{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set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
{2, 3}
>>> s1 | s2
{1, 2, 3, 4}
```

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”。试试把list放入set，看看是否会报错。

第二节：函数及其调用

2.1 调用函数

Python内置了很多有用的函数，我们可以直接调用。要调用一个函数，需要知道函数的名称和参数。比如求绝对值的函数abs，只有一个参数。可以直接从Python的官方网站查看文档：<http://docs.python.org/3/library/functions.html#abs>

也可以在交互式命令行通过help(abs)查看abs函数的帮助信息。

调用abs函数：

```
>>> abs(100)
100
>>> abs(-20)
20
>>> abs(12.34)
12.34
```

而max函数max()可以接收任意多个参数，并返回最大的那个：

```
>>> max(1, 2)
2
>>> max(2, 3, 1, -5)
3
```

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
```

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 # 变量a指向abs函数
>>> a(-1) # 所以也可以通过a调用abs函数
1
```

2.2 定义函数

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。我们以自定义一个求绝对值的my_abs函数为例：

```
def my_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

请注意，函数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。

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。

在Python交互环境中定义函数时，注意Python会出现...的提示。函数定义结束后需要按两次回车重新回到>>>提示符下：

```
Command Prompt - python
>>> def my_abs(x):
...     if x >= 0:
...         return x
...     else:
...         return -x
...
>>> my_abs(-9)
9
>>> _
```

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）：

```
Command Prompt - python
>>> from abstest import my_abs
>>> my_abs(-9)
9
>>> _
```

空函数

如果想定义一个什么事也不做的空函数，可以用pass语句：

```
def nop():
    pass
```

pass语句什么都不做，那有什么用？实际上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，比如：

```
if age >= 18:  
    pass
```

缺少了pass，代码运行就会有语法错误。

参数检查

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：

```
>>> my_abs(1, 2)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: my_abs() takes 1 positional argument but 2 were given
```

但是如果参数类型不对，Python解释器就无法帮我们检查。试试my_abs和内置函数abs的差别：

```
>>> 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'
```

当传入了不恰当的参数时，内置函数abs会检查出参数错误，而我们定义的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。所以，这个函数定义不够完善。让我们修改一下my_abs的定义，对参数类型做检查，只允许整数和浮点数类型的参数。数据类型检查可以用内置函数 isinstance()实现：

```
def my_abs(x):  
    if not isinstance(x, (int, float)):  
        raise TypeError('bad operand type')  
    if x >= 0:  
        return x  
    else:  
        return -x
```

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
```

返回多个值

函数可以返回多个值吗？答案是肯定的。

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。

然后，我们就可以同时获得返回值：

```
>>> x, y = move(100, 100, 60, math.pi / 6)  
>>> print(x, y)  
151.96152422706632 70.0
```

但其实这只是一个假象，Python函数返回的仍然是单一值：

```
>>> r = move(100, 100, 60, math.pi / 6)  
>>> print(r)  
(151.96152422706632, 70.0)
```

原来返回值是一个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。

2.3 函数的参数

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

我们先写一个计算x²的函数：

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

对于power(x)函数，参数x就是一个位置参数。

当我们调用power函数时，必须传入有且仅有的一个参数x：

```
>>> power(5)  
25  
>>> power(15)  
225
```

现在，如果我们要计算x³怎么办？可以再定义一个power3函数，但是如果要计算x⁴、x⁵……怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把power(x)修改为power(x, n)，用来计算xⁿ，说干就干：

```
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)  
25  
>>> power(5, 3)  
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，旧的调用代码失败了，原因是我们增加了一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

```
>>> power(5)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'n'
```

Python的错误信息很明确：调用函数power()缺少了一个位置参数n。

这个时候，默认参数就派上用场了。由于我们经常计算 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)
25
>>> power(5, 2)
25
```

而对于 $n > 2$ 的其他情况，就必须明确地传入n，比如power(5, 3)。

从上面的例子可以看出，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。

使用默认参数有什么好处？最大的好处是能降低调用函数的难度。

举个例子，我们写个一年级小学生注册的函数，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('Sarah', 'F')
name: Sarah
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=6, city='Beijing'):
    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('Sarah', 'F')
name: Sarah
gender: F
age: 6
city: Beijing
```

只有与默认参数不符的学生才需要提供额外的信息：

```
enroll('Bob', 'M', 7)
enroll('Adam', 'M', city='Tianjin')
```

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。

默认参数很有用，但使用不当，也会掉坑里。默认参数有个最大的坑，演示如下：

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
```

当你正常调用时，结果似乎不错：

```
>>> add_end([1, 2, 3])
[1, 2, 3, 'END']
>>> add_end(['x', 'y', 'z'])
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

```
>>> add_end()
['END']
```

但是，再次调用add_end()时，结果就不对了：

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

很多初学者很疑惑，默认参数是[]，但是函数似乎每次都“记住了”上次添加了'END'后的list。

原因解释如下：

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，默认参数的內容就变了，不再是函数定义时的[]了。

定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用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
```

要修改上面的例子，我们可以用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()  
['END']  
>>> add_end()  
['END']
```

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。我们在编写程序时，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。

我们以数学题为例子，给定一组数字a, b, c.....，请计算a² + b² + c² +

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，我们首先想到可以把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])  
14  
>>> calc((1, 3, 5, 7))  
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)  
14  
>>> calc(1, 3, 5, 7)  
84
```

所以，我们把函数的参数改为可变参数：

```
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)  
5  
>>> calc()  
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])  
14
```

这种写法当然是可行的，问题是太繁琐，所以Python允许你在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)  
14
```

*nums表示把nums这个list的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

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。请看示例：

```
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，我们保证能接收到name和age这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你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。

命名关键字参数

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，我们希望检查是否有city和job参数：

```
def person(name, age, **kw):  
    if 'city' in kw:  
        # 有city参数  
        pass  
    if 'job' in kw:  
        # 有job参数  
        pass  
    print('name:', name, 'age:', age, 'other:', kw)
```

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')
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() takes 2 positional arguments but 4 were given
```

由于调用时缺少参数city和job，Python解释器把这4个参数均视为位置参数，但person()函数仅接受2个位置参数。

命名关键字参数可以有缺省值，从而简化调用：

```
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')
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):
    # 缺少 *, city和job被视为位置参数
    pass
```

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。

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。

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

最神奇的是通过一个tuple和dict，你也可以调用上述函数：

```
>>> args = (1, 2, 3, 4)
>>> kw = {'d': 99, 'x': '#'}
>>> f1(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}
>>> args = (1, 2, 3)
>>> kw = {'d': 88, 'x': '#'}
>>> f2(*args, **kw)
a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

所以，对于任意函数，都可以通过类似func(*args, **kw)的形式调用它，无论它的参数是如何定义的。

不同参数调用方法对比：

参数类型	调用方法	特征
位置参数	def power(x):	最简单，最基本的调用方法
默认参数	def enroll(name, gender, age=6, city='Beijing'):	name和gender位置参数，age和city是默认参数
可变参数	def calc(*numbers):	在参数前面加了一个*表示这个参数是可变参数
关键字参数	def person(name, age, **kw):	关键字参数kw，前面加**表示是关键字参数，调用时可以输入或者不输入这个参数信息，也可以随意增加想要增加的信息。格式是dict
命名关键字参数	def person(name, age, *, city, job):	在中间加一个*表示是命名关键字参数，表示如果想要传入那些附加信息，也只能传入指定的那几个参数（此处的city, job）

2.4递归函数

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，我们来计算阶乘 $n! = 1 \times 2 \times 3 \times \dots \times n$ ，用函数fact(n)表示，可以看出：

$\text{fact}(n) = n! = 1 \times 2 \times 3 \times \dots \times (n-1) \times n = (n-1)! \times n = \text{fact}(n-1) \times n$

所以，fact(n)可以表示为 $n \times \text{fact}(n-1)$ ，只有 $n=1$ 时需要特殊处理。

于是，fact(n)用递归的方式写出来就是：

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试：

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。由于栈的大小是有限的，所以，递归调用的深度过深，可能会导致栈溢出，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 -
```

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。
遗憾的是，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。

第三节：Python的高级特性

第十一章

取一个list或tuple的部分元素是非常常见的操作。比如，一个list如下：

```
>>> L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']
```

取前3个元素 应该怎么做?

>>> [L[0], L[1], L[2]]

```
['Michael', 'Sarah', 'Tracy']
```

之所以是笨办法是因为扩展一下，取前N个元素就没辙了。

```
>>> r = []
>>> n = 3
>>> for i in range(n):
...     r.append(L[i])
...
>>> r
['Michael', 'Sarah', 'Tracy']
```

对这种经常取指定索引范围的操作，用循环十分繁琐。因此，Python提供了切片（Slice）操作符，能大大简化这种操作。

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，对应上面的问题，取前3个元素，用一行代码就可以完成切片。

```
>>> L[0:3]
['Michael', 'Sarah', 'Tracy']
```

L[0:3]表示，从索引0开始取，直到索引3

```
>>> L[:3]
['Michael', 'Sarah']
```

也可以从索引1开始，取出2个元素出来：

```
>>> L[1:3]
['Sarah', 'Trac']
类似的，既然Pytho
```

```
[ 'Bob' ]
```

记住倒数第一个元素的索引是-1

```
>>> L = list(range(100))  
>>> L
```

```
[0, 1, 2, 3, ..., 99]
```

```
>>> L[:10]
```

12个数

```
>>> L[-10:]
```

前11-30个数：

>>> L[10:30]

```
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

前10个数，每两个取一个：

```
>>> L[::2]
[0, 2, 4, 6, 8]
```

所有数，每5个取一个：

```
>>> L[::5]
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]
```

甚至什么都不写，只写[:]就可以原样复制一个list：

```
>>> L[:]
[0, 1, 2, 3, ..., 99]
```

tuple也是一种list，唯一区别是tuple不可变。因此，tuple也可以用切片操作，只是操作的结果仍是tuple：

```
>>> (0, 1, 2, 3, 4, 5)[::3]
(0, 1, 2)
```

字符串'xxx'也可以看成是一种list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：

```
>>> 'ABCDEFG'[::3]
'ABC'
>>> 'ABCDEFG'[::2]
'ACEG'
```

在很多编程语言中，针对字符串提供了很多各种截取函数（例如，substring），其实目的就是对字符串切片。Python没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

3.2 迭代

如果给定一个list或tuple，我们可以通过for循环来遍历这个list或tuple，这种遍历我们称为迭代（Iteration）。

在Python中，迭代是通过for ... in来完成的，而很多语言比如C语言，迭代list是通过下标完成的，比如Java代码：

```
for (i=0; i<list.length; i++) {
    n = list[i];
}
```

可以看出，Python的for循环抽象程度要高于C的for循环，因为Python的for循环不仅可以用在list或tuple上，还可以作用在其他可迭代对象上。

list这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如dict就可以迭代：

```
>>> d = {'a': 1, 'b': 2, 'c': 3}
>>> for key in d:
...     print(key)
...
a
c
b
```

因为dict的存储不是按照list的方式顺序排列，所以，迭代出的结果顺序很可能不一样。

默认情况下，dict迭代的是key。如果要迭代value，可以用for value in d.values()，如果要同时迭代key和value，可以用for k, v in d.items()。

由于字符串也是可迭代对象，因此，也可以作用于for循环：

```
>>> for ch in 'ABC':
...     print(ch)
...
A
B
C
```

所以，当我们使用for循环时，只要作用于一个可迭代对象，for循环就可以正常运行，而我们不太关心该对象究竟是list还是其他数据类型。

那么，如何判断一个对象是可迭代对象呢？方法是通过collections模块的Iterable类型判断：

```
>>> from collections import Iterable
>>> isinstance('abc', Iterable) # str是否可迭代
True
>>> isinstance([1,2,3], Iterable) # list是否可迭代
True
>>> isinstance(123, Iterable) # 整数是否可迭代
False
```

最后一个小小问题，如果要对list实现类似Java那样的下标循环怎么办？Python内置的enumerate函数可以把一个list变成索引-元素对，这样就可以在for循环中同时迭代索引和元素本身：

```
>>> for i, value in enumerate(['A', 'B', 'C']):
...     print(i, value)
...
0 A
1 B
2 C
```

上面的for循环里，同时引用了两个变量，在Python里是很常见的，比如下面的代码：

```
>>> for x, y in [(1, 1), (2, 4), (3, 9)]:
...     print(x, y)
...
1 1
2 4
3 9
```

3.3 列表生成式

列表生成式即List Comprehensions，是Python内置的非常简单却强大的可以用来创建list的生成式。

举个例子，要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]可以用list(range(1, 11))：

```
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

但如果要生成[1x1, 2x2, 3x3, ..., 10x10]怎么做？方法一是循环：

```
>>> L = []
>>> for x in range(1, 11):
...     L.append(x * x)
...
>>> L
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

但是循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的list：

```
>>> [x * x for x in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

写列表生成式时，把要生成的元素x * x放到前面，后面跟for循环，就可以把list创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

for循环后面还可以加上if判断，这样我们就可以筛选出仅偶数的平方：

```
>>> [x * x for x in range(1, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

还可以使用两层循环，可以生成全排列：

```
>>> [m + n for m in 'ABC' for n in 'XYZ']
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

三层和三层以上的循环就很少用到了。

运用列表生成式，可以写出非常简洁的代码。例如，列出当前目录下的所有文件和目录名，可以通过一行代码实现：

```
>>> import os # 导入os模块，模块的概念后面讲到
>>> [d for d in os.listdir('.')] # os.listdir可以列出文件和目录
['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop', 'Documents', 'Downloads', 'Library', 'Movies', 'Music', 'Pictures', 'Public', 'VirtualBox VMs', 'Workspace', 'XCode']
```

for循环其实可以同时使用两个甚至多个变量，比如dict.items()可以同时迭代key和value：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C'}
>>> for k, v in d.items():
...     print(k, '=', v)
...
y = B
x = A
z = C
```

因此，列表生成式也可以使用两个变量来生成list：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C'}
>>> [k + '=' + v for k, v in d.items()] #这里的+表示字符串连接，k接上一个=然后再接上一个v
['y=B', 'x=A', 'z=C']
```

最后把一个list中所有的字符串变成小写：

```
>>> L = ['Hello', 'World', 'IBM', 'Apple']
>>> [s.lower() for s in L]
['hello', 'world', 'ibm', 'apple']
```

3.4生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。

要创建一个generator，有很多种方法。第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

```
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x * x for x in range(10))
>>> g
<generator object <genexpr> at 0x1022ef630>
```

创建L和g的区别仅在于最外层的[]和()，L是一个list，而g是一个generator。

我们可以直接打印出list的每一个元素，但我们怎么打印出generator的每一个元素呢？

如果要一个一个打印出来，可以通过next()函数获得generator的下一个返回值：

```
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)
9
>>> next(g)
16
>>> next(g)
25
>>> next(g)
36
>>> next(g)
49
>>> next(g)
64
>>> next(g)
81
>>> next(g)
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

我们讲过，generator保存的是算法，每次调用next(g)，就计算出g的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误。

当然，上面这种不断调用next(g)实在是太变态了，正确的方法是使用for循环，因为generator也是可迭代对象：

```
>>> g = (x * x for x in range(10))
>>> for n in g:
...     print(n)
...
0
1
4
9
16
25
36
49
64
81
```

所以，我们创建了一个generator后，基本上永远不会调用next()，而是通过for循环来迭代它，并且不需要关心StopIteration的错误。

generator非常强大。如果推算的算法比较复杂，用类似列表生成式的for循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
```

```
n = n + 1  
return 'done'
```

仔细观察，可以看出，fib函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似generator。也就是说，上面的函数和generator仅一步之遥。要把fib函数变成generator，只需要把print(b)改为yield b就可以了：

```
def fib(max):  
    n, a, b = 0, 0, 1  
    while n < max:  
        yield b  
        a, b = b, a + b  
        n = n + 1  
    return 'done'
```

这就是定义generator的另一种方法。如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator：

```
>>> f = fib(6)  
>>> f  
<generator object fib at 0x104feaaa0>
```

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

举个简单的例子，定义一个generator，依次返回数字1, 3, 5:

```
def odd():  
    print('step 1')  
    yield 1  
    print('step 2')  
    yield(3)  
    print('step 3')  
    yield(5)
```

调用该generator时，首先要生成一个generator对象，然后用next()函数不断获得下一个返回值：

```
>>> o = odd()  
>>> next(o)  
step 1  
1  
>>> next(o)  
step 2  
3  
>>> next(o)  
step 3  
5  
>>> next(o)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
StopIteration
```

可以看到，odd不是普通函数，而是generator，在执行过程中，遇到yield就中断，下次又继续执行。执行3次yield后，已经没有yield可以执行了，所以，第4次调用next(o)就报错。回到fib的例子，我们在循环过程中不断调用yield，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用next()来获取下一个返回值，而是直接使用for循环来迭代：

```
>>> for n in fib(6):  
...     print(n)  
...  
1  
1  
2  
3  
5  
8
```

但是用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

```
>>> g = fib(6)  
>>> while True:  
...     try:  
...         x = next(g)  
...         print('g:', x)  
...     except StopIteration as e:  
...         print('Generator return value:', e.value)  
...         break  
...  
g: 1  
g: 1  
g: 2  
g: 3  
g: 5  
g: 8  
Generator return value: done
```

3.5 迭代器

我们已经知道，可以直接作用于for循环的数据类型有以下几种：

一类是集合数据类型，如list、tuple、dict、set、str等；

一类是generator，包括生成器和带yield的generator function。

这些可以直接作用于for循环的对象统称为可迭代对象：Iterable。

可以使用isinstance()判断一个对象是否是Iterable对象：

```
>>> from collections import Iterable  
>>> isinstance([], Iterable)  
True  
>>> isinstance({}, Iterable)  
True  
>>> isinstance('abc', Iterable)  
True  
>>> isinstance((x for x in range(10)), Iterable)  
True  
>>> isinstance(100, Iterable)  
False
```

而生成器不但可以作用于for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。

可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator。

可以使用isinstance()判断一个对象是否是Iterator对象：

```
>>> from collections import Iterator  
>>> isinstance((x for x in range(10)), Iterator)  
True  
>>> isinstance([], Iterator)  
False  
>>> isinstance({}, Iterator)
```

```
False
>>> isinstance('abc', Iterator)
False
```

生成器都是Iterator对象，但list、dict、str虽然是Iterable，却不是Iterator。

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

```
>>> isinstance(iter([]), Iterator)
True
>>> isinstance(iter('abc'), Iterator)
True
```

这是因为Python的Iterator对象表示的是一个数据流，Iterator对象可以被next()函数调用并不断返回下一个数据，直到没有数据时抛出StopIteration错误。可以把这个数据流看做一个有序序列，但我们却不能提前知道序列的长度，只能不断通过next()函数实现按需计算下一个数据，所以Iterator的计算是惰性的，只有在需要返回下一个数据时它才会计算。

Iterator甚至可以表示一个无限大的数据流，例如全体自然数。而使用list是永远不可能存储全体自然数的。

凡是可作用于for循环的对象都是Iterable类型：

凡是可作用于next()函数的对象都是Iterator类型，它们表示一个惰性计算的序列；

集合数据类型list、dict、str等是Iterable但不是Iterator，不过可以通过iter()函数获得一个Iterator对象。

Python的for循环本质上就是通过不断调用next()函数实现的，例如：

```
# 首先获得Iterator对象:
it = iter([1, 2, 3, 4, 5])
# 循环:
while True:
    try:
        # 获得下一个值:
        x = next(it)
    except StopIteration:
        # 遇到StopIteration就退出循环
        break
```

第四节：函数式编程

4.1 高阶函数

变量可以指向函数

以Python内置的求绝对值的函数abs()为例，调用该函数用以下代码：

```
>>> abs(-10)
10
```

但是，如果只写abs呢？

```
>>> abs
<built-in function abs>
```

可见，abs(-10)是函数调用，而abs是函数本身。

要获得函数调用结果，我们可以把结果赋值给变量：

```
>>> x = abs(-10)
>>> x
10
```

但是，如果把函数本身赋值给变量呢？

```
>>> f = abs
>>> f
<built-in function abs>
```

结论：函数本身也可以赋值给变量，即：变量可以指向函数。

如果一个变量指向了一个函数，那么，可否通过该变量来调用这个函数？用代码验证一下：

```
>>> f = abs
>>> f(-10)
10
```

成功！说明变量f现在已经指向了abs函数本身。直接调用abs()函数和调用变量f()完全相同。

函数名也是变量

那么函数名是什么呢？函数名其实就是指向函数的变量！对于abs()这个函数，完全可以把函数名abs看成变量，它指向一个可以计算绝对值的函数！

如果把abs指向其他对象，会有什么情况发生？

```
>>> abs = 10
>>> abs(-10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

把abs指向10后，就无法通过abs(-10)调用该函数了！因为abs这个变量已经不指向求绝对值函数而是指向一个整数10！

当然实际代码绝对不能这么写，这里是是为了说明函数名也是变量。要恢复abs函数，请重启Python交互环境。

注：由于abs函数实际上是定义在import builtins模块中的，所以要让修改abs变量的指向在其它模块也生效，要用import builtins; builtins.abs = 10。

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

```
def add(x, y, f):
    return f(x) + f(y)
print(add(-5, 6, abs))
```

11

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

map/reduce(实际就是对一连串序列同时进行操作)

我们先看map。map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。

```
>>> def f(x):
...     return x * x
...
>>> r = map(f, [1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> list(r)
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

map()传入的第一个参数是f，即函数对象本身。由于结果r是一个Iterator，Iterator是惰性序列，因此通过list()函数让它把整个序列都计算出来并返回一个list。

你可能会想，不需要map()函数，写一个循环，也可以计算出结果：

```
L = []
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    L.append(f(n))
print(L)
```

的确可以，但是，从上面的循环代码，能一眼看明白“把f(x)作用在list的每一个元素并把结果生成一个新的list”吗？

所以，map()作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x)=x²，还可以计算任意复杂的函数，比如，把这个list所有数字转为字符串：

```
>>> list(map(str, [1, 2, 3, 4, 5, 6, 7, 8, 9]))  
['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

再看reduce的用法。reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算，其效果就是：

```
reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)
```

比方说对一个序列求和，就可以用reduce实现：

```
>>> from functools import reduce  
>>> def add(x, y):  
...     return x + y  
...  
>>> reduce(add, [1, 3, 5, 7, 9])  
25
```

当然求和运算可以直接用Python内建函数sum()，没必要动用reduce。
但是如果要把序列[1, 3, 5, 7, 9]变换成整数13579，reduce就可以派上用场：

```
>>> from functools import reduce  
>>> def fn(x, y):  
...     return x * 10 + y  
...  
>>> reduce(fn, [1, 3, 5, 7, 9])  
13579
```

这个例子本身没多大用处，但是，如果考虑到字符串str也是一个序列，对上面的例子稍加改动，配合map()，我们就可以写出把str转换为int的函数：

```
>>> from functools import reduce  
>>> def fn(x, y):  
...     return x * 10 + y  
...  
>>> def char2num(s):  
...     digits = {'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9}  
...     return digits[s]  
...  
>>> reduce(fn, map(char2num, '13579'))  
13579
```

filter

Python内建的filter()函数用于过滤序列。

和map()类似，filter()也接收一个函数和一个序列。和map()不同的是，filter()把传入的函数依次作用于每个元素，然后根据返回值是True还是False决定保留还是丢弃该元素。
例如，在一个list中，删掉偶数，只保留奇数，可以这么写：

```
def is_odd(n):  
    return n % 2 == 1  
  
list(filter(is_odd, [1, 2, 4, 5, 6, 9, 10, 15]))  
# 结果: [1, 5, 9, 15]
```

可见用filter()这个高阶函数，关键在于正确实现一个“筛选”函数。

注意到filter()函数返回的是一个Iterator，也就是一个惰性序列，所以要强迫filter()完成计算结果，需要用list()函数获得所有结果并返回list。

sorted

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个dict呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。

Python内置的sorted()函数就可以对list进行排序：

```
>> sorted([36, 5, -12, 9, -21])  
[-21, -12, 5, 9, 36]
```

此外，sorted()函数也是一个高阶函数，它还可以接收一个key函数来实现自定义的排序，例如按绝对值大小排序：

```
>> sorted([36, 5, -12, 9, -21], key=abs)  
[5, 9, -12, -21, 36]
```

key指定的函数将作用于list的每一个元素上，并根据key函数返回的结果进行排序。对比原始的list和经过key=abs处理过的list：

```
list = [36, 5, -12, 9, -21]  
keys = [36, 5, 12, 9, 21]
```

然后sorted()函数按照keys进行排序，并按照对应关系返回list相应的元素：

```
keys排序结果 => [5, 9, 12, 21, 36]  
                  | | | | |  
最终结果      => [5, 9, -12, -21, 36]
```

我们再看一个字符串排序的例子：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'])  
['Credit', 'Zoo', 'about', 'bob']
```

现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能用一个key函数把字符串映射为忽略大小写排序即可。忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

这样，我们给sorted传入key函数，即可实现忽略大小写的排序：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower)  
['about', 'bob', 'Credit', 'Zoo']
```

要进行反向排序，不必改动key函数，可以传入第三个参数reverse=True：

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower, reverse=True)  
['Zoo', 'Credit', 'bob', 'about']
```

4.2 函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

```
def calc_sum(*args):  
    ax = 0  
    for n in args:  
        ax = ax + n  
    return ax
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数：

```
def lazy_sum(*args):  
    def sum():  
        ax = 0  
        for n in args:  
            ax = ax + n  
        return ax  
    return sum
```

当我们调用lazy_sum()时，返回的并不是求和结果，而是求和函数：

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function lazy_sum.<locals>.sum at 0x101c6ed90>
```

调用函数f时，才真正计算求和的结果：

```
>>> f()
25
```

在这个例子中，我们在函数lazy_sum中又定义了函数sum，并且，内部函数sum可以引用外部函数lazy_sum的参数和局部变量，当lazy_sum返回函数sum时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用lazy_sum()时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
>>> f1 = lazy_sum(1, 3, 5, 7, 9)
>>> f2 = lazy_sum(1, 3, 5, 7, 9)
>>> f1==f2
False
```

f1()和f2()的调用结果互不影响。

返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

4.3 匿名函数

当我们在传入函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。

在Python中，对匿名函数提供了有限支持。还是以map()函数为例，计算 $f(x)=x^2$ 时，除了定义一个 $f(x)$ 的函数外，还可以直接传入匿名函数：

```
>>> list(map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

通过对比可以看出，匿名函数lambda x: x * x实际上就是：

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

关键字lambda表示匿名函数，冒号前面的x表示函数参数。

匿名函数有个限制，就是只能有一个表达式，不用写return，返回值就是该表达式的结果。

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

```
>>> f = lambda x: x * x
>>> f
<function <lambda> at 0x101c6ef28>
>>> f(5)
25
```

同样，也可以把匿名函数作为返回值返回，比如：

```
def build(x, y):
    return lambda : x * x + y * y #这一半表示的是一个函数！！！
t=build(1,2)
print(t())
5
```

4.4 偏函数

Python的functools模块提供了很多有用的功能，其中一个就是偏函数（Partial function）。要注意，这里的偏函数和数学意义上的偏函数不一样。

在介绍函数参数的时候，我们讲到，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。举例如下：

int()函数可以把字符串转换为整数，当仅传入字符串时，int()函数默认按十进制转换：

```
>>> int('12345')
12345
```

但int()函数还提供额外的base参数，默认值为10。如果传入base参数，就可以做N进制的转换：

```
>>> int('12345', base=8)
5349
>>> int('12345', 16)
74565
```

假设要转换大量的二进制字符串，每次都传入int(x, base=2)非常麻烦，于是，我们想到，可以定义一个int2()的函数，默认把base=2传进去：

```
def int2(x, base=2):
    return int(x, base)
```

这样，我们转换二进制就非常方便了：

```
>>> int2('1000000')
64
>>> int2('1010101')
85
```

functools.partial就是帮助我们创建一个偏函数的，不需要我们自己定义int2()，可以直接使用下面的代码创建一个新的函数int2：

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

所以，简单总结functools.partial的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新int2函数，仅仅是把base参数重新设定默认值为2，但也可以在函数调用时传入其他值：

```
>>> int2('1000000', base=10)
1000000
```

第五节：模块

5.1 模块简介

计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Python中，一个.py文件就称之为一个模块（Module）。

使用模块有什么好处？

最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。点这里查看Python的所有内置函数。

你也许还想到，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python又引入了按目录来组织模块的方法，称为包（Package）。

举个例子，一个abc.py的文件就是一个名字叫abc的模块，一个xyz.py的文件就是一个名字叫xyz的模块。

现在，假设我们的abc和xyz这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如mycompany，按照如下目录存放：

```
mycompany
├── __init__.py
├── abc.py
└── xyz.py
```

引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，abc.py模块的名字就变成了mycompany.abc，类似的，xyz.py的模块名变成了mycompany.xyz。

请注意，每一个包目录下面都会有一个__init__.py的文件，这个文件是必须存在的，否则，Python就把这个目录当成普通目录，而不是一个包。__init__.py可以是空文件，也可以有Python代码，因为__init__.py本身就是一个模块，而它的模块名就是mycompany。

类似的，可以有多级目录，组成多级层次的包结构。比如如下的目录结构：

```
mycompany
├── web
│   ├── __init__.py
│   ├── utils.py
│   └── www.py
├── __init__.py
└── abc.py
└── xyz.py
```

文件www.py的模块名就是mycompany.web.www，两个文件utils.py的模块名分别是mycompany.utils和mycompany.web.utils。

自己创建模块时要注意命名，不能和Python自带的模块名称冲突。例如，系统自带了sys模块，自己的模块就不可命名为sys.py，否则将无法导入系统自带的sys模块。

模块是一组Python代码的集合，可以使用其他模块，也可以被其他模块使用。

创建自己的模块时，要注意：

模块名要遵循Python变量命名规范，不要使用中文、特殊字符；

模块名不要和系统模块名冲突，最好先查看系统是否已存在该模块，检查方法是在Python交互环境执行import abc，若成功则说明系统存于此模块。

5.2 使用模块

Python本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。

我们以内建的sys模块为例，编写一个hello的模块：

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

' a test module '

__author__ = 'Michael Liao'

import sys

def test():
    args = sys.argv
    if len(args)==1:
        print('Hello, world!')
    elif len(args)==2:
        print('Hello, %s' % args[1])
    else:
        print('Too many arguments!')

if __name__=='__main__':
    test()
```

第1行和第2行是标准注释，第1行注释可以让这个hello.py文件直接在Unix/Linux/Mac上运行，第2行注释表示.py文件本身使用标准UTF-8编码；

第4行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

第6行使用__author__变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；

以上就是Python模块的标准文件模板，当然也可以全部删掉不写，但是，按标准办事肯定没错。

后面开始就是真正的代码部分。

你可能注意到了，使用sys模块的第一步，就是导入该模块：

```
import sys
```

导入sys模块后，我们就有了变量sys指向该模块，利用sys这个变量，就可以访问sys模块的所有功能。

sys模块有一个argv变量，用list存储了命令行的所有参数。argv至少有一个元素，因为第一个参数永远是该.py文件的名称，例如：

运行python3 hello.py获得的sys.argv就是['hello.py']；

运行python3 hello.py Michael获得的sys.argv就是['hello.py', 'Michael']。

最后，注意到这两行代码：

```
if __name__=='__main__':
    test()
```

当我们在命令行运行hello模块文件时，Python解释器把一个特殊变量__name__置为__main__，而如果在其他地方导入该hello模块时，if判断将失败，因此，这种if测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

我们可以用命令行运行hello.py看看效果：

```
$ python3 hello.py
Hello, world!
$ python hello.py Michael
Hello, Michael!
```

如果启动Python交互环境，再导入hello模块：

```
$ python3
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import hello
>>>
```

导入时，没有打印Hello, word!，因为没有执行test()函数。

调用hello.test()时，才能打印出Hello, word!：

```
>>> hello.test()
Hello, world!
```

作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过__前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如：abc, x123, PI等；

类似__xxx__这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的__author__, __name__就是特殊变量，hello模块定义的文档注释也可以用特殊变量__doc__访问，我们自己的变量一般不要用这种变量名；

类似__xxx和__xxx这样的函数或变量就是非公开的（private），不应该被直接引用，比如__abc, __abc等；

之所以我们说，private函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问private函数或变量，但是，从编程习惯上不应该引用private函数或变量。

private函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

```
def __private_1(name):
    return 'Hello, %s' % name
```

```
def _private_2(name):
    return 'Hi, %s' % name

def greeting(name):
    if len(name) > 3:
        return _private_1(name)
    else:
        return _private_2(name)
```

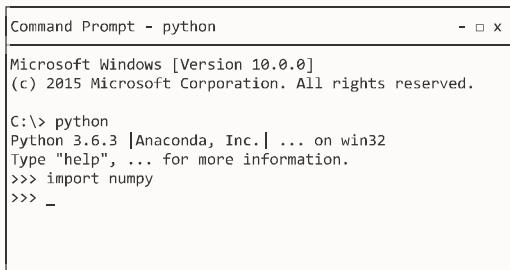
我们在模块里公开greeting()函数，而把内部逻辑用private函数隐藏起来了，这样，调用greeting()函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：
外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

5.3 安装第三方模块

安装常用模块

在使用Python时，我们经常需要用到很多第三方库，例如，上面提到的Pillow，以及MySQL驱动程序，Web框架Flask，科学计算NumPy等。用pip一个一个安装费时费力，还需要考虑兼容性。我们推荐直接使用Anaconda，这是一个基于Python的数据处理和科学计算平台，它已经内置了许多非常有用第三方库，我们装上Anaconda，就相当于把数十个第三方模块自动安装好了，非常简单易用。

可以从Anaconda官网下载GUI安装包，安装包有500~600M，所以需要耐心等待下载。网速慢的同学请移步国内镜像。下载后直接安装，Anaconda会把系统Path中的python指向自己自带的Python，并且，Anaconda安装的第三方模块会安装在Anaconda自己的路径下，不影响系统已安装的Python目录。
安装好Anaconda后，重新打开命令行窗口，输入python，可以看到Anaconda的信息：



Command Prompt - python

Microsoft Windows [Version 10.0.0]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\> python
Python 3.6.3 |Anaconda, Inc.| ... on win32
Type "help", ... for more information.
>>> import numpy
>>> -

可以尝试直接import numpy等已安装的第三方模块。

模块搜索路径

当我们试图加载一个模块时，Python会在指定的路径下搜索对应的.py文件，如果找不到，就会报错：

```
>>> import mymodule
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named mymodule
```

默认情况下，Python解释器会搜索当前目录、所有已安装的内置模块和第三方模块，搜索路径存放在sys模块的path变量中：

```
>>> import sys
>>> sys.path
[ '', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python36.zip', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6', ..., '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages' ]
```

如果我们要添加自己的搜索目录，有两种方法：

一是直接修改sys.path，添加要搜索的目录：

```
>>> import sys
>>> sys.path.append('/Users/michael/my_py_scripts')
```

这种方法是在运行时修改，运行结束后失效。

第二种方法是设置环境变量PYTHONPATH，该环境变量的内容会被自动添加到模块搜索路径中。设置方式与设置Path环境变量类似。注意只需要添加你自己的搜索路径，Python自己本身的搜索路径不受影响。

第六节：面向对象编程

数据封装、继承和多态是面向对象的三大特点

6.1 类和实例

面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，比如Student类，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

仍以Student为例，在Python中，定义类是通过class关键字：

```
class Student(object):
    pass
```

class后面紧接着是类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来的，继承的概念我们后面再讲，通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

定义好了Student类，就可以根据Student类创建出Student的实例，创建实例是通过类名+()实现的：

```
>>> bart = Student()
>>> bart
<__main__.Student object at 0x10a67a590>
>>> Student
<class '__main__.Student'>
```

可以看到，变量bart指向的就是一个Student的实例，后面的0x10a67a590是内存地址，每个object的地址都不一样，而Student本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例bart绑定一个name属性：

```
>>> bart.name = 'Bart Simpson'
>>> bart.name
'Bart Simpson'
```

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的_init_方法，在创建实例的时候，就把name，score等属性绑上去：

```
class Student(object):

    def __init__(self, name, score):    #注意：特殊方法“__init__”前后分别有两个下划线！！！
        self.name = name
        self.score = score
```

注意到_init_方法的第一个参数永远是self，表示创建的实例本身，因此，在_init_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。
有了_init_方法，在创建实例的时候，就不能传入空的参数了，必须传入与_init_方法匹配的参数，但self不需要传，Python解释器自己会把实例变量传进去：

```
>>> bart = Student('Bart Simpson', 59)
>>> bart.name
'Bart Simpson'
```

```
>>> bart.score  
59
```

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量self，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

数据封装

面向对象编程的一个重要特点就是数据封装。在上面的Student类中，每个实例就拥有各自的name和score这些数据。我们可以通过函数来访问这些数据，比如打印一个学生的成绩：

```
>>> def print_score(std):  
...     print('%s: %s' % (std.name, std.score))  
...  
>>> print_score(bart)  
Bart Simpson: 59
```

但是，既然Student实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在Student类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法：

```
class Student(object):  
  
    def __init__(self, name, score):  
        self.name = name  
        self.score = score  
  
    def print_score(self):  
        print('%s: %s' % (self.name, self.score))
```

要定义一个方法，除了第一个参数是self外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了self不用传递，其他参数正常传入：

```
>>> bart.print_score()  
Bart Simpson: 59
```

这样一来，我们从外部看Student类，就只需要知道，创建实例需要给出name和score，而如何打印，都是在Student类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

封装的另一个好处是可以给Student类增加新的方法，比如get_grade：

```
class Student(object):  
    ...  
  
    def get_grade(self):  
        if self.score >= 90:  
            return 'A'  
        elif self.score >= 60:  
            return 'B'  
        else:  
            return 'C'
```

6.2 访问限制

在Class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

但是，从前面Student类的定义来看，外部代码还是可以自由地修改一个实例的name、score属性：

```
>>> bart = Student('Bart Simpson', 59)  
>>> bart.score  
59  
>>> bart.score = 99  
>>> bart.score  
99
```

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线_，在Python中，实例的变量名如果以__开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们把Student类改一改：

```
class Student(object):  
  
    def __init__(self, name, score):  
        self.__name = name  
        self.__score = score  
  
    def print_score(self):  
        print('%s: %s' % (self.__name, self.__score))
```

改完后，对于外部代码来说，没什么变动，但是已经无法从外部访问实例变量__name和实例变量__score了：

```
>>> bart = Student('Bart Simpson', 59)  
>>> bart.__name  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'Student' object has no attribute '__name'
```

这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

但是如果外部代码要获取name和score怎么办？可以给Student类增加get_name和get_score这样的方法：

```
class Student(object):  
    ...  
  
    def get_name(self):  
        return self.__name  
  
    def get_score(self):  
        return self.__score
```

如果又要允许外部代码修改score怎么办？可以再给Student类增加set_score方法：

```
class Student(object):  
    ...  
  
    def set_score(self, score):  
        self.__score = score
```

你也许会问，原先那种直接通过bart.score = 99也可以修改啊，为什么要定义一个方法大费周折？因为在方法中，可以对参数做检查，避免传入无效的参数：

```
class Student(object):  
    ...  
  
    def set_score(self, score):  
        if 0 <= score <= 100:  
            self.__score = score  
        else:  
            raise ValueError('bad score')
```

需要注意的是，在Python中，变量名类似__xxx__的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用__name__、__score__这样的变量名。

有些时候，你会看到以一个下划线开头的实例变量名，比如`_name`，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。
双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问`_name`是因为Python解释器对外把`_name`变量改成了`_Student__name`，所以，仍然可以通过`_Student__name`来访问`_name`变量：

```
>>> bart._Student__name  
'Bart Simpson'
```

但是强烈建议你不要这么干，因为不同版本的Python解释器可能会把`_name`改成不同的变量名。
总的来说就是，Python本身没有任何机制阻止你干坏事，一切全靠自觉。

最后注意下面的这种错误写法：

```
>>> bart = Student('Bart Simpson', 59)  
>>> bart.get_name()  
'Bart Simpson'  
>>> bart.__name = 'New Name' # 设置__name变量!  
>>> bart.__name  
'New Name'
```

表面上看，外部代码“成功”地设置了`__name`变量，但实际上这个`__name`变量和class内部的`__name`变量不是一个变量！内部的`__name`变量已经被Python解释器自动改成了`_Student__name`，而外部代码给`bart`新增了一个`__name`变量。不信试试：

```
>>> bart.get_name() # get_name()内部返回self.__name  
'Bart Simpson'
```

6.3 继承和多态

在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

比如，我们已经编写了一个名为`Animal`的class，有一个`run()`方法可以直接打印：

```
class Animal(object):  
    def run(self):  
        print('Animal is running...')
```

当我们需要编写`Dog`和`Cat`类时，就可以直接从`Animal`类继承：

```
class Dog(Animal):  
    pass  
  
class Cat(Animal):  
    pass
```

对于`Dog`来说，`Animal`就是它的父类，对于`Animal`来说，`Dog`就是它的子类。`Cat`和`Dog`类似。

继承有什么好处？最大的好处是子类获得了父类的全部功能。由于`Animal`实现了`run()`方法，因此，`Dog`和`Cat`作为它的子类，什么事也没干，就自动拥有了`run()`方法：

```
dog = Dog()  
dog.run()  
  
cat = Cat()  
cat.run()
```

当然，也可以对子类增加一些方法，比如`Dog`类：

```
class Dog(Animal):  
  
    def run(self):  
        print('Dog is running...')  
  
    def eat(self):  
        print('Eating meat...')
```

继承的第二个好处需要我们对代码做一点改进。你看到了，无论是`Dog`还是`Cat`，它们`run()`的时候，显示的都是`Animal is running...`，符合逻辑的做法是分别显示`Dog is running...`和`Cat is running...`，因此，对`Dog`和`Cat`类改进如下：

```
class Dog(Animal):  
  
    def run(self):  
        print('Dog is running...')  
  
class Cat(Animal):  
  
    def run(self):  
        print('Cat is running...')
```

再次运行，结果如下：

```
Dog is running...  
Cat is running...
```

当子类和父类都存在相同的`run()`方法时，我们说，子类的`run()`覆盖了父类的`run()`，在代码运行的时候，总是会调用子类的`run()`。这样，我们就获得了继承的另一个好处：多态。要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个class的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如`str`、`list`、`dict`没什么两样：

```
a = list() # a是list类型  
b = Animal() # b是Animal类型  
c = Dog() # c是Dog类型
```

判断一个变量是否是某个类型可以用`isinstance()`判断：

```
>>> isinstance(a, list)  
True  
>>> isinstance(b, Animal)  
True  
>>> isinstance(c, Dog)  
True
```

来`a`、`b`、`c`确实对应着`list`、`Animal`、`Dog`这3种类型。

但是等等，试试：

```
>>> isinstance(c, Animal)  
True
```

不过仔细想想，这是有道理的，因为`Dog`是从`Animal`继承下来的，当我们创建了一个`Dog`的实例`c`时，我们认为`c`的数据类型是`Dog`没错，但`c`同时也是`Animal`也没错，`Dog`本来就是`Animal`的一种！

所以在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

```
>>> b = Animal()  
>>> isinstance(b, Dog)  
False
```

`Dog`可以看成`Animal`，但`Animal`不可以看成`Dog`。

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个`Animal`类型的变量：

```
def run_twice(animal):
    animal.run()
    animal.run()
```

当我们传入Animal的实例时，run_twice()就打印出：

```
>>> run_twice(Animal())
Animal is running...
Animal is running...
```

当我们传入Dog的实例时，run_twice()就打印出：

```
>>> run_twice(Dog())
Dog is running...
Dog is running...
```

当我们传入Cat的实例时，run_twice()就打印出：

```
>>> run_twice(Cat())
Cat is running...
Cat is running...
```

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个Tortoise类型，也从Animal派生：

```
class Tortoise(Animal):
    def run(self):
        print('Tortoise is running slowly...')
```

当我们调用run_twice()时，传入Tortoise的实例：

```
>>> run_twice(Tortoise())
Tortoise is running slowly...
Tortoise is running slowly...
```

你会发现，新增一个Animal的子类，不必对run_twice()做任何修改，实际上，任何依赖Animal作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

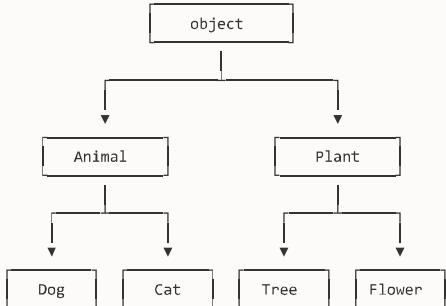
多态的好处就是，当我们需要传入Dog、Cat、Tortoise.....时，我们只需要接收Animal类型就可以了，因为Dog、Cat、Tortoise.....都是Animal类型，然后，按照Animal类型进行操作即可。由于Animal类型有run()方法，因此，传入的任意类型，只要是Animal类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思：

对于一个变量，我们只需要知道它是Animal类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Animal、Dog、Cat还是Tortoise对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种Animal的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增Animal子类；

对修改封闭：不需要修改依赖Animal类型的run_twice()等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：



静态语言 vs 动态语言

对于静态语言（例如Java）来说，如果需要传入Animal类型，则传入的对象必须是Animal类型或者它的子类，否则，将无法调用run()方法。

对于Python这样的动态语言来说，则不一定需要传入Animal类型。我们只需要保证传入的对象有一个run()方法就可以了：

```
class Timer(object):
    def run(self):
        print('Start...')
```

这就是动态语言的“鸭子类型”，它不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。

Python的“file-like object”就是一种鸭子类型。对真正的文件对象，它有一个read()方法，返回其内容。但是，许多对象，只要有read()方法，都被视为“file-like object”。许多函数接收的参数就是“file-like object”，你不一定非要传入真正的文件对象，完全可以传入任何实现了read()方法的对象。

小结

继承可以把父类的所有功能都直接拿过来，这样就不必重零做起，子类只需要新增自己特有的方法，也可以把父类不适合的方法覆盖重写。

动态语言的鸭子类型特点决定了继承不像静态语言那样是必须的。

6.4 获取对象信息

当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

type()

首先，我们来判断对象类型，使用type()函数：

基本类型都可以用type()判断：

```
>>> type(123)
<class 'int'>
>>> type('str')
<class 'str'>
>>> type(None)
<type('None') 'NoneType'>
```

如果一个变量指向函数或者类，也可以用type()判断：

```
>>> type(abs)
<class 'builtin_function_or_method'>
>>> type(a)
<class '__main__.Animal'>
```

但是type()函数返回的是什么类型呢？它返回对应的Class类型。如果我们要在if语句中判断，就需要比较两个变量的type类型是否相同：

```
>>> type(123)==type(456)
True
>>> type(123)==int
True
>>> type('abc')==type('123')
True
>>> type('abc')==str
True
>>> type('abc')==type(123)
False
```

判断基本数据类型可以直接写int, str等，但如果要判断一个对象是否是函数怎么办？可以使用types模块中定义的常量：

```
>>> import types
>>> def fn():
...     pass
...
>>> type(fn)==types.FunctionType
True
>>> type(abs)==types.BuiltinFunctionType
True
>>> type(lambda x: x)==types.LambdaType
True
>>> type((x for x in range(10)))==types.GeneratorType
True
```

isinstance()

对于class的继承关系来说，使用type()就很不方便。我们要判断class的类型，可以使用isinstance()函数。
我们回顾上次的例子，如果继承关系是：

```
object -> Animal -> Dog -> Husky
```

那么，isinstance()就可以告诉我们，一个对象是否是某种类型。先创建3种类型的对象：

```
>>> a = Animal()
>>> d = Dog()
>>> h = Husky()
```

然后，判断：

```
>>> isinstance(h, Husky)
True
```

没有问题，因为h变量指向的就是Husky对象。

再判断：

```
>>> isinstance(h, Dog)
True
```

h虽然自身是Husky类型，但由于Husky是从Dog继承下来的，所以，h也还是Dog类型。换句话说，isinstance()判断的是一个对象是否是该类型本身，或者位于该类型的父继承链上。

因此，我们可以确信，h还是Animal类型：

```
>>> isinstance(h, Animal)
True
```

能用type()判断的基本类型也可以用isinstance()判断：

```
>>> isinstance('a', str)
True
>>> isinstance(123, int)
True
>>> isinstance(b'a', bytes)
True
```

并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是list或者tuple：

```
>>> isinstance([1, 2, 3], (list, tuple))
True
>>> isinstance((1, 2, 3), (list, tuple))
True
```

总是优先使用isinstance()判断类型，可以将指定类型及其子类“一网打尽”。

dir()

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：

```
>>> dir('ABC')
['_add__', '__class__', ..., '_subclasshook__', 'capitalize', 'casifold', ..., 'zfill']
```

类似__xxx__的属性和方法在Python中都是有特殊用途的，比如__len__方法返回长度。在Python中，如果你调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的__len__()方法，所以，下面的代码是等价的：

```
>>> len('ABC')
3
>>> 'ABC'.__len__()
3
```

我们自己写的类，如果也想用len(myObj)的话，就自己写一个__len__()方法：

```
>>> class MyDog(object):
...     def __len__(self):
...         return 100
...
>>> dog = MyDog()
>>> len(dog)
100
```

剩下的都是普通属性或方法，比如lower()返回小写的字符串：

```
>>> 'ABC'.lower()
'abc'
```

仅仅把属性和方法列出来是不够的，配合getattr()、setattr()以及hasattr()，我们可以直接操作一个对象的状态：

```
>>> class MyObject(object):
...     def __init__(self):
...         self.x = 9
...     def power(self):
...         return self.x * self.x
...
>>> obj = MyObject()
```

紧接着，可以测试该对象的属性：

```
>>> hasattr(obj, 'x') # 有属性'x'吗?
True
>>> obj.x
9
>>> hasattr(obj, 'y') # 有属性'y'吗?
False
>>> setattr(obj, 'y', 19) # 设置一个属性'y'
>>> hasattr(obj, 'y') # 有属性'y'吗?
True
>>> getattr(obj, 'y') # 获取属性'y'
19
>>> obj.y # 获取属性'y'
```

如果试图获取不存在的属性，会抛出AttributeError的错误：

```
>>> getattr(obj, 'z') # 获取属性'z'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyObject' object has no attribute 'z'
```

可以传入一个default参数，如果属性不存在，就返回默认值：

```
>>> getattr(obj, 'z', 404) # 获取属性'z'，如果不存在，返回默认值404
404
```

也可以获得对象的方法：

```
>>> hasattr(obj, 'power') # 有属性'power'吗？
True
>>> getattr(obj, 'power') # 获取属性'power'
<bound method MyObject.power of <__main__.MyObject object at 0x10077a6a0>>
>>> fn = getattr(obj, 'power') # 获取属性'power'并赋值到变量fn
>>> fn # fn指向obj.power
<bound method MyObject.power of <__main__.MyObject object at 0x10077a6a0>>
>>> fn() # 调用fn()与调用obj.power()是一样的
81
```

一个正确的用法的例子如下：

```
def readImage(fp):
    if hasattr(fp, 'read'):
        return readData(fp)
    return None
```

假设我们希望从文件流fp中读取图像，我们首先要判断该fp对象是否存在read方法，如果存在，则该对象是一个流，如果不存在，则无法读取。hasattr()就派上了用场。

请注意，在Python这类动态语言中，根据鸭子类型，有read()方法，不代表该fp对象就是一个文件流，它也可能是网络流，也可能是内存中的一个字节流，但只要read()方法返回的是有效的图像数据，就不影响读取图像的功能。

6.5 实例属性和类属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过self变量：

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

s = Student('Bob')
s.score = 90
```

但是，如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归Student类所有：

```
class Student(object):
    name = 'Student'
```

当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。来测试一下：

```
>>> class Student(object):
...     name = 'Student'
...
>>> s = Student() # 创建实例s
>>> print(s.name) # 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性
Student
>>> print(Student.name) # 打印类的name属性
Student
>>> s.name = 'Michael' # 给实例绑定name属性
>>> print(s.name) # 山于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性
Michael
>>> print(Student.name) # 但是类属性并未消失，用Student.name仍然可以访问
Student
>>> del s.name # 如果删除实例的name属性
>>> print(s.name) # 再次调用s.name，山于实例的name属性没有找到，类的name属性就显示出来了
Student
```

实例属性属于各个实例所有，互不干扰；

类属性属于类所有，所有实例共享一个属性；

不要对实例属性和类属性使用相同的名字，否则将产生难以发现的错误。

第七节：面向对象高级编程

7.1 使用slots

正常情况下，当我们定义了一个class，创建了一个class的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。先定义class：

```
class Student(object):
    pass
```

然后，尝试给实例绑定一个属性：

```
>>> s = Student()
>>> s.name = 'Michael' # 动态给实例绑定一个属性
>>> print(s.name)
Michael
```

还可以尝试给实例绑定一个方法：

```
>>> def set_age(self, age): # 定义一个函数作为实例方法
...     self.age = age
...
>>> from types import MethodType
>>> s.set_age = MethodType(set_age, s) # 给实例绑定一个方法
>>> s.set_age(25) # 调用实例方法
>>> s.age # 测试结果
25
```

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

```
>>> s2 = Student() # 创建新的实例
>>> s2.set_age(25) # 尝试调用方法
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'set_age'
```

为了给所有实例都绑定方法，可以给class绑定方法：

```
>>> def set_score(self, score):
...     self.score = score
...
```

```
>>> Student.set_score = set_score
```

给class绑定方法后，所有实例均可调用：

```
>>> s.set_score(100)
>>> s.score
100
>>> s2.set_score(99)
>>> s2.score
99
```

通常情况下，上面的set_score方法可以直接定义在class中，但动态绑定允许我们在程序运行的过程中动态给class加上功能，这在静态语言中很难实现。

使用_slots_

但是，如果我们想要限制实例的属性怎么办？比如，只允许对Student实例添加name和age属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的_slots_变量，来限制该class实例能添加的属性：

```
class Student(object):
    __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称
```

然后，我们试试：

```
>>> s = Student() # 创建新的实例
>>> s.name = 'Michael' # 绑定属性'name'
>>> s.age = 25 # 绑定属性'age'
>>> s.score = 99 # 绑定属性'score'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'score'
```

由于'score'没有被放到_slots_中，所以不能绑定score属性，试图绑定score将得到AttributeError的错误。

使用_slots_要注意，_slots_定义的属性仅对当前类实例起作用，对继承的子类是不起作用的：

```
>>> class GraduateStudent(Student):
...     pass
...
>>> g = GraduateStudent()
>>> g.score = 9999
```

除非在子类中也定义_slots_，这样，子类实例允许定义的属性就是自身的_slots_加上父类的_slots_。

7.2 使用@property

在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参数，导致可以把成绩随便改：

```
s = Student()
s.score = 9999
```

这显然不合逻辑。为了限制score的范围，可以通过一个set_score()方法来设置成绩，再通过一个get_score()来获取成绩，这样，在set_score()方法里，就可以检查参数：

```
class Student(object):

    def get_score(self):
        return self._score

    def set_score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

现在，对任意的Student实例进行操作，就不能随心所欲地设置score了：

```
>>> s = Student()
>>> s.set_score(60) # ok!
>>> s.get_score()
60
>>> s.set_score(9999)
Traceback (most recent call last):
...
ValueError: score must between 0 ~ 100!
```

但是，上面的调用方法又略显复杂，没有直接用属性这么直接简单。

有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的Python程序员来说，这是必须要做到的！

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python内置的@property装饰器就是负责把一个方法变成属性调用的

```
class Student(object):

    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

@property的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上@property就可以了，此时，@property本身又创建了另一个装饰器@score.setter，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

```
>>> s = Student()
>>> s.score = 60 # OK, 实际转化为s.set_score(60)
>>> s.score # OK, 实际转化为s.get_score()
60
>>> s.score = 9999
Traceback (most recent call last):
...
ValueError: score must between 0 ~ 100!
```

注意到这个神奇的@property，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：

```
class Student(object):

    @property
    def birth(self):
        return self._birth

    @birth.setter
    def birth(self, value):
        self._birth = value
```

```

@property
def age(self):
    return 2015 - self._birth

```

上面的birth是可读写属性，而age就是一个只读属性，因为age可以根据birth和当前时间计算出来。

7.3多重继承

继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。

回忆一下Animal类层次的设计，假设我们要实现以下4种动物：

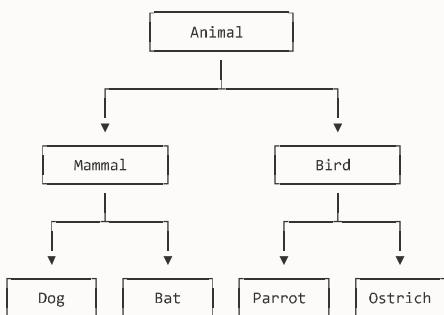
Dog - 狗狗；

Bat - 蝙蝠；

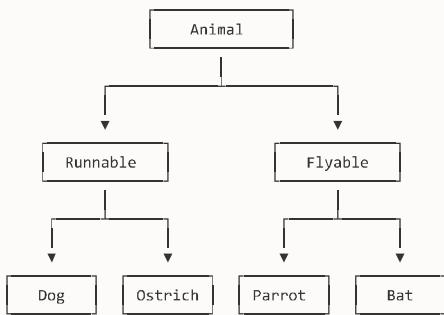
Parrot - 鹦鹉；

Ostrich - 鸵鸟。

如果按照哺乳动物和鸟类归类，我们可以设计出这样的类的层次：



但是如果按照“能跑”和“能飞”来归类，我们就应该设计出这样的类的层次：

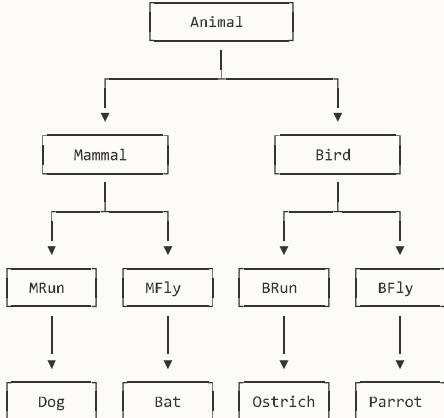


如果要把上面的两种分类都包含进来，我们就得设计更多的层次：

哺乳类：能跑的哺乳类，能飞的哺乳类；

鸟类：能跑的鸟类，能飞的鸟类。

这么一来，类的层次就复杂了：



果要再增加“宠物类”和“非宠物类”，这么搞下去，类的数量会呈指数增长，很明显这样设计是不行的。

正确的做法是采用多重继承。首先，主要的类层次仍按照哺乳类和鸟类设计：

```

class Animal(object):
    pass

# 大类:
class Mammal(Animal):
    pass

class Bird(Animal):
    pass

# 各种动物:
class Dog(Mammal):
    pass

class Bat(Mammal):
    pass

class Parrot(Bird):
    pass

class Ostrich(Bird):
    pass
  
```

现在，我们要给动物再加上Runnable和Flyable的功能，只需要先定义好Runnable和Flyable的类：

```

class Runnable(object):
    pass
  
```

```
def run(self):
    print('Running...')

class Flyable(object):
    def fly(self):
        print('Flying...')
```

对于需要Runnable功能的动物，就多继承一个Runnable，例如Dog：

```
class Dog(Mammal, Runnable):
    pass
```

对于需要Flyable功能的动物，就多继承一个Flyable，例如Bat：

```
class Bat(Mammal, Flyable):
    pass
```

通过多重继承，一个子类就可以同时获得多个父类的所有功能。

MixIn

在设计类的继承关系时，通常，主线都是单一继承下来的，例如，Ostrich继承自Bird。但是，如果需要“混入”额外的功能，通过多重继承就可以实现，比如，让Ostrich除了继承自Bird外，再同时继承Runnable。这种设计通常称之为MixIn。

为了更好地看出继承关系，我们把Runnable和Flyable改为RunnableMixIn和FlyableMixIn。类似的，你还可以定义出肉食动物CarnivorousMixIn和植食动物HerbivoresMixIn，让某个动物同时拥有好几个MixIn：

```
class Dog(Mammal, RunnableMixIn, CarnivorousMixIn):
    pass
```

MixIn的目的就是给一个类增加多个功能，这样，在设计类的时候，我们优先考虑通过多重继承来组合多个MixIn的功能，而不是设计多层次的复杂的继承关系。

由于Python允许使用多重继承，因此，MixIn就是一种常见的设计。

只允许单一继承的语言（如Java）不能使用MixIn的设计。

7.4 定制类

看到类似_slots_这种形如_xxx_的变量或者函数名就要注意，这些在Python中是有特殊用途的。

_slots_我们已经知道怎么用了，_len_()方法我们也知道是为了能让class作用于len()函数。

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

__str__

我们先定义一个Student类，打印一个实例：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...
>>> print(Student('Michael'))
<__main__.Student object at 0x109afb190>
```

打印出一堆<__main__.Student object at 0x109afb190>，不好看。

怎么才能打印得好看呢？只需要定义好__str__()方法，返回一个好看的字符串就可以了：

```
>>> class Student(object):
...     def __init__(self, name):
...         self.name = name
...     def __str__(self):
...         return 'Student object (name: %s)' % self.name
...
>>> print(Student('Michael'))
Student object (name: Michael)
```

这样打印出来的实例，不但好看，而且容易看出实例内部重要的数据。

但是细心的朋友会发现直接敲变量不用print，打印出来的实例还是不好看：

```
>>> s = Student('Michael')
>>> s
<__main__.Student object at 0x109afb310>
```

这是因为直接显示变量调用的不是__str__(), 而是__repr__(), 两者的区别是__str__()返回用户看到的字符串，而__repr__()返回程序开发者看到的字符串，也就是说，__repr__()是为调试服务的。

解决办法是再定义一个__repr__(). 但是通常__str__()和__repr__()代码都是一样的，所以，有个偷懒的写法：

```
class Student(object):
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return 'Student object (name=%s)' % self.name
    __repr__ = __str__
```

__iter__
如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个__iter__()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的__next__()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

```
class Fib(object):
    def __init__(self):
        self.a, self.b = 0, 1 # 初始化两个计数器a, b

    def __iter__(self):
        return self # 实例本身就是迭代对象，故返回自己

    def __next__(self):
        self.a, self.b = self.b, self.a + self.b # 计算下一个值
        if self.a > 100000: # 退出循环的条件
            raise StopIteration()
        return self.a # 返回下一个值
```

现在，试试把Fib实例作用于for循环：

```
>>> for n in Fib():
...     print(n)
...
1
1
2
3
5
...
46368
75025
```

7.5枚举类

当我们需要定义常量时，一个办法是用大写变量通过整数来定义，例如月份：

```
JAN = 1
FEB = 2
MAR = 3
...
NOV = 11
DEC = 12
```

好处是简单，缺点是类型是int，并且仍然是变量。

更好的方法是为这样的枚举类型定义一个class类型，然后，每个常量都是class的一个唯一实例。Python提供了Enum类来实现这个功

```
from enum import Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))
```

这样我们就获得了Month类型的枚举类，可以直接使用Month.Jan来引用一个常量，或者枚举它的所有成员：

```
for name, member in Month.__members__.items():
    print(name, '=>', member, ',', member.value)
```

value属性则是自动赋给成员的int常量，默认从1开始计数。

如果需要更精确地控制枚举类型，可以从Enum派生出自定义类：

```
from enum import Enum, unique

@unique
class Weekday(Enum):
    Sun = 0 # Sun的value被设定为0
    Mon = 1
    Tue = 2
    Wed = 3
    Thu = 4
    Fri = 5
    Sat = 6
```

@unique装饰器可以帮助我们检查保证没有重复值。

访问这些枚举类型可以有若干种方法：

```
>>> day1 = Weekday.Mon
>>> print(day1)
Weekday.Mon
>>> print(Weekday.Tue)
Weekday.Tue
>>> print(Weekday['Tue'])
Weekday.Tue
>>> print(Weekday.Tue.value)
2
>>> print(day1 == Weekday.Mon)
True
>>> print(day1 == Weekday.Tue)
False
>>> print(Weekday(1))
Weekday.Mon
>>> print(day1 == Weekday(1))
True
>>> Weekday(7)
Traceback (most recent call last):
...
ValueError: 7 is not a valid Weekday
>>> for name, member in Weekday.__members__.items():
...     print(name, '=>', member)
...
Sun => Weekday.Sun
Mon => Weekday.Mon
Tue => Weekday.Tue
Wed => Weekday.Wed
Thu => Weekday.Thu
Fri => Weekday.Fri
Sat => Weekday.Sat
```

7.6 使用元类

略

第八节：错误、调试和测试

8.1 错误处理

高级语言通常都内置了一套try...except...finally...的错误处理机制

try

让我们用一个例子来看看try的机制：

```
try:
    print('try...')
    r = 10 / 0
    print('result:', r)
except ZeroDivisionError as e:
    print('except:', e)
finally:
    print('finally...')
print('END')
```

当我们认为某些代码可能会出错时，就可以用try来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即except语句块，执行完except后，如果有finally语句块，则执行finally语句块，至此，执行完毕。

上面的代码在计算 $10 / 0$ 时会产生一个除法运算错误：

```
try...
except: division by zero
finally...
END
```

从输出可以看到，当错误发生时，后续语句print('result', r)不会被执行，except由于捕获到ZeroDivisionError，因此被执行。最后，finally语句被执行。然后，程序继续按照流程往下走。

如果把除数0改成2，则执行结果如下：

```
try...
result: 5
finally...
END
```

由于没有错误发生，所以except语句块不会被执行，但是finally如果有，则一定会被执行（可以没有finally语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的except语句块处理。没错，可以有多个except来捕获不同类型的错误：

```
try:
```

```
print('try...')
r = 10 / int('a')
print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
finally:
    print('finally...')
print('END')
```

int()函数可能会抛出ValueError，所以我们用一个except捕获ValueError，用另一个except捕获ZeroDivisionError。此外，如果没有错误发生，可以在except语句块后面加一个else，当没有错误发生时，会自动执行else语句：

```
try:
    print('try...')
    r = 10 / int('2')
    print('result:', r)
except ValueError as e:
    print('ValueError:', e)
except ZeroDivisionError as e:
    print('ZeroDivisionError:', e)
else:
    print('no error!')
finally:
    print('finally...')
print('END')
```

Python的错误其实也是class，所有的错误类型都继承自BaseException，所以在使用except时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:
    foo()
except ValueError as e:
    print('ValueError')
except UnicodeError as e:
    print('UnicodeError')
```

第二个except永远也捕获不到UnicodeError，因为UnicodeError是ValueError的子类，如果有，也被第一个except给捕获了。

Python所有的错误都是从BaseException类派生的，常见的错误类型和继承关系看这里：

<https://docs.python.org/3/library/exceptions.html#exception-hierarchy>

使用try...except捕获错误还有一个巨大的好处，就是可以跨多层调用，比如函数main()调用foo()，foo()调用bar()，结果bar()出错了，这时，只要main()捕获到了，就可以处理：

```
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        print('Error:', e)
    finally:
        print('finally...')
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写try...except...finally的麻烦。

调用栈

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。来看看err.py：

```
# err.py:
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    bar('0')

main()
```

执行，结果如下：

```
$ python3 err.py
Traceback (most recent call last):
  File "err.py", line 11, in <module>
    main()
  File "err.py", line 9, in main
    bar('0')
  File "err.py", line 6, in bar
    return foo(s) * 2
  File "err.py", line 3, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
```

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：
错误信息第1行：

```
Traceback (most recent call last):
```

告诉我们这是错误的跟踪信息。

第2~3行：

```
  File "err.py", line 11, in <module>
    main()
```

调用main()出错了，在代码文件err.py的第11行代码，但原因是第9行：

```
  File "err.py", line 9, in main
    bar('0')
```

调用bar('0')出错了，在代码文件err.py的第9行代码，但原因是第6行：

```
  File "err.py", line 6, in bar
    return foo(s) * 2
```

原因是return foo(s) * 2这个语句出错了，但这还不是最终原因，继续往下看：

```
  File "err.py", line 3, in foo
    return 10 / int(s)
```

原因是return 10 / int(s)这个语句出错了，这是错误产生的源头，因为下面打印了：

```
ZeroDivisionError: integer division or modulo by zero
```

根据错误类型ZeroDivisionError，我们判断，int(s)本身并没有出错，但是int(s)返回0，在计算10 / 0时出错，至此，找到错误源头。

出错的时候，一定要分析错误的调用栈信息，才能定位错误的位置。

记录错误

如果不捕获错误，自然可以让Python解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python内置的logging模块可以非常容易地记录错误信息：

```
# err_logging.py

import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        logging.exception(e)

main()
print('END')
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python3 err_logging.py
ERROR:root:division by zero
Traceback (most recent call last):
  File "err_logging.py", line 13, in main
    bar('0')
  File "err_logging.py", line 9, in bar
    return foo(s) * 2
  File "err_logging.py", line 6, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
END
```

通过配置，logging还可以把错误记录到日志文件里，方便事后排查。

抛出错误

因为错误是class，捕获一个错误就是捕获到该class的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。

如果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用raise语句抛出一个错误的实例：

```
# err_raise.py
class FooError(ValueError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n

foo('0')
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python3 err_raise.py
Traceback (most recent call last):
  File "err_throw.py", line 11, in <module>
    foo('0')
  File "err_throw.py", line 8, in foo
    raise FooError('invalid value: %s' % s)
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择Python已有的内置的错误类型（比如ValueError，TypeError），尽量使用Python内置的错误类型。

最后，我们来看另一种错误处理的方式：

```
# err_reraise.py

def foo(s):
    n = int(s)
    if n==0:
        raise ValueError('invalid value: %s' % s)
    return 10 / n

def bar():
    try:
        foo('0')
    except ValueError as e:
        print('ValueError!')
        raise

bar()
```

在bar()函数中，我们明明已经捕获了错误，但是，打印一个ValueError!后，又把错误通过raise语句抛出去了，这不有病么？

其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。好比一个员工处理不了一个问题时，就把问题抛给他的老板，如果他的老板也处理不了，就一直往上抛，最终会抛给CEO去处理。

raise语句如果不带参数，就会把当前错误原样抛出。此外，在except中raise一个Error，还可以把一种类型的错误转化成另一种类型：

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个IOError转换成毫不相干的ValueError。

8.2 调试

程序能一次写完并正常运行的概率很小，基本不超过1%。总会有各种各样的bug需要修正。有的bug很简单，看看错误信息就知道，有的bug很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复bug。

第一种方法简单直接粗暴有效，就是用print()把可能有问题的变量打印出来看看：

```
def foo(s):
```

```
n = int(s)
print('>>> n = %d' % n)
return 10 / n

def main():
    foo('0')

main()
```

执行后在输出中查找打印的变量值：

```
$ python err.py
>>> n = 0
Traceback (most recent call last):
...
ZeroDivisionError: integer division or modulo by zero
```

用print()最大的坏处是将来还得删掉它，想想程序里到处都是print()，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。

断言

凡是用print()来辅助查看的地方，都可以用断言（assert）来替代：

```
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

assert的意思是，表达式n != 0应该是True，否则，根据程序运行的逻辑，后面的代码肯定会出错。

如果断言失败，assert语句本身就会抛出AssertionError：

```
$ python err.py
Traceback (most recent call last):
...
AssertionError: n is zero!
```

程序中如果到处充斥着assert，和print()相比也好不到哪去。不过，启动Python解释器时可以用-O参数来关闭assert：

```
$ python -O err.py
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

关闭后，你可以把所有的assert语句当成pass来看。

logging

把print()替换为logging是第3种方式，和assert比，logging不会抛出错误，而且可以输出到文件：

```
import logging

s = '0'
n = int(s)
logging.info('n = %d' % n)
print(10 / n)
```

logging.info()就可以输出一段文本。运行，发现除了ZeroDivisionError，没有任何信息。怎么回事？

别急，在import logging之后添加一行配置再试试：

```
import logging
logging.basicConfig(level=logging.INFO)
```

看到输出了：

```
$ python err.py
INFO:root:: 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这就是logging的好处，它允许你指定记录信息的级别，有debug, info, warning, error等几个级别，当我们指定level=INFO时，logging.debug就不起作用了。同理，指定level=WARNING后，debug和info就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

logging的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如console和文件。

pdb

第4种方式是启动Python的调试器pdb，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

```
# err.py
s = '0'
n = int(s)
print(10 / n)
```

然后启动：

```
$ python -m pdb err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(2)<module>()
-> s = '0'
```

以参数-m pdb启动后，pdb定位到下一步要执行的代码-> s = '0'。输入命令来查看代码：

```
(Pdb) l
 1      # err.py
 2  -> s = '0'
 3      n = int(s)
 4      print(10 / n)
```

输入命令n可以单步执行代码：

```
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(3)<module>()
-> n = int(s)
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(4)<module>()
-> print(10 / n)
```

任何时候都可以输入命令p 变量名来查看变量：

```
(Pdb) p s
'0'
(Pdb) p n
0
```

输入命令q结束调试，退出程序：

```
(Pdb) q
```

这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有一千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

pdb.set_trace()

这个方法也是用pdb，但是不需要单步执行，我们只需要import pdb，然后，在可能出错的地方放一个pdb.set_trace()，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print(10 / n)
```

运行代码，程序会自动在pdb.set_trace()暂停并进入pdb调试环境，可以用命令p查看变量，或者用命令c继续运行：

```
$ python err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(7)<module>()
-> print(10 / n)
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这种方式比直接启动pdb单步调试效率要高很多，但也高不到哪去。

IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。目前比较好的Python IDE有：

Visual Studio Code: <https://code.visualstudio.com/>, 需要安装Python插件。

PyCharm: <http://www.jetbrains.com/pycharm/>

另外，Eclipse加上pydev插件也可以调试Python程序。