print(4+6) #运算符:'+'。功能：加法。结果：10

print(3.5-2) #运算符:'-'，功能：减法。结果：1.5

print(9\*9) #运算符:'\*'，功能：乘法。结果：81

print(5/2) #运算符:'/'，功能：除法。结果：2.5

print(5\*\*3) #运算符:'\*\*'，功能：乘幂。结果：125

print(5//2) #运算符:'//'，功能：整除(舍去小数部分)。结果：2

print(7%3) #运算符:'%'，功能：求余数(模)。结果：1

print(10 == 10) #运算符：'=='。说明：等于。表达式值：True

print(10 != 10) #运算符：'!='。说明：不等于。表达式值：False

print(10 > 5.1) #运算符：'>'。说明：大于。表达式值：True

print(10.2 < 5) #运算符：'<'。说明：小于。表达式值：False

print(5 >= 10) #运算符：'>='。说明：大于等于。表达式值：False

print(5 <= 10) #运算符：'<='。说明：小于等于。表达式值：True

| **类型转换函数** | **描述** | **例子** |
| --- | --- | --- |
| int | 转换成整型数 | int('100') |
| float | 转换成浮点数 | float('100.123') |
| str | 转换成字符串 | str(100.1234) |

打印多个对象的时候，每个对象之间默认有一个空格，多个对象按位置顺序依次输出，。 对于本段示例，多个数字之间用一个空格打印出来分割并不太明显。 键入如下代码并观察执行结果。

print('多个参数', 5, 4078%211, 407.8//211)

print('多个参数', 5, 4078%211, 407.8//211, sep = ' ')

print('多个参数', 5, 4078%211, 407.8//211, sep = ',')

print('多个参数', 5, 4078%211, 407.8//211, sep = ' ')

print('多个参数', 5, 4078%211, 407.8//211, sep = ' ,')

print('多个参数', 5, 4078%211, 407.8//211, sep = '')

本段代码中，sep是参数关键字，是separator的缩写，意思是分隔符。sep = ','是一个赋值语句，,被赋值给sep，此时，打印多个对象时，对象间的分隔符就被指定为,。如果没有用sep参数指定，则print()函数的分隔符就是一个空格，与用sep = ' '来指定分隔符的效果相同。比较正式的说法是：sep参数的默认值是一个空格。 另外，带参数关键字的参数只能出现在没有关键字的参数(称为位置参数)的后面，键入如下代码并观察执行结果。

print('多个参数', 5, 4078%211, sep = ',', 407.8//211)

程序出现语法错误，提示Syntax Error: positional argument follows keyword argument，即语法错误：位置参数后有关键字参数。

4.md

函数定义的基本结构是：

**def** 函数名(参数表):

语句块

* 函数名(参数表)，一般称为**函数头**；
* 函数定义内部的语句块又称为**函数体**；
* 参数表就是一系列参数，可以有0，1或多个参数，多个参数之间用,分隔。

函数定义好之后，只是能使python解释器知道有一个自定义函数，知道这个函数的名字，知道函数参数表，但是并不执行，也并不知道函数参数表内参数的具体值。 因此，函数定义时的参数，一般称为**形式参数**，简称**形参**。  
python解释器知道很多已经定义好的函数，比如input()，比如print()，但是如果我们不在代码中写出来，不去调用使用，函数就不会被执行。  
这样，前面执行第一个自定义函数的代码，就只是让python解释器知道有一个自定义函数，名字是fake\_print，有一个参数content，这些都属于函数头信息。  
对自定义函数来说，仅让python解释器知道这个函数头的信息还不够，还需要在函数体实现函数的功能。这个功能是为以后函数被使用被调用时准备的。

* 键入如下代码并观察执行结果。

*# 函数定义*

**def** fake\_print(content):

**print**(content)

*# 主程序*

fake\_print('假装是print()函数。')

本段代码执行的大体流程是：

* 对第一行代码def fake\_print(content):进行处理，发现这是函数定义，于是保存此处地址，保存函数信息：函数名是fake\_print，一个函数形参是content；
* 跳过fake\_print()的函数体定义，直接从函数体外的第一行开始执行；
* 在执行到主程序[^1]第一行fake\_print('假装是print()函数。')时，发现这里调用了fake\_print()函数，有一个实参与函数定义的形参content相对应，其值为'假装是print()函数。'，于是将'假装是print()函数。'赋值给content；
* 保存调用函数的地址，然后根据fake\_print()函数的地址，跳转到def fake\_print(content):,准备运行函数体。此时，形参content已有实际值，已经可以作为有实际值的变量在函数体内使用了。执行函数时，其形参值是由实际参数在函数调用时由对应的实参传递过来的，一般把这个参数传递过程称为**参数传递**。注意参数传递时，实参会赋值给相**对应**的形参，最一般的就是位置对应，因此，**不需要**形参与对应实参的变量名相同；
* 运行函数体，调用print()函数打印'假装是print()函数。'；
* 函数体执行完毕，程序返回到调用fake\_print()函数地址的行末，继续执行；
* 执行到程序尾，整个程序执行完毕。

由自定义函数的执行过程可知：如果自定义函数在当前文件中定义，则函数定义的代码必须写在函数调用之前。有些“神秘”函数比如print()及input()，则不受此限制。实际上print()及input()等函数由于太过常用，函数已经被预先定义及实现好，并内置在Python语言中，称为**内置函数**。

*# 函数定义*

**def** compute\_sum(end):

i = 0

total\_n = 0

**while** i < end:

i = i + 1

total\_n = total\_n + i

**return** total\_n

主程序

**print**(compute\_sum(100))

终于得到了预期中的结果。  
本段代码中的return是python的关键字。由return将total\_n赋值给函数compute\_sum()，此时compute\_sum()的函数值即为total\_n当前的值。由于在得到函数的值以后，程序要返回到函数调用的位置，同时，函数的值也被顺利‘带回’，因此函数调用后得到的函数值，一般称为**函数返回值**。

* python程序可根据函数定义(0、1或多个)及主程序划分为多个区域，每个区域内都的变量(对象)的在各个区域的使用(作用)有一定规则限制，因此这种区域称作**作用域**，又分为**函数作用域**(对应函数体)与**全局作用域**(对应主程序)。
* 每个函数内部的变量只属于该函数的函数体，主程序无法使用。例：

*#函数作用域示例1*

**def** inner(n):

n = 10

**print**('inner', n)

*#主程序*

inner(n) *#报错*

将提示句法错误：name 'n' is not defined，即n没有定义。虽然在主程序前面的inner()函数定义中，变量n被赋值10，但主程序无法使用inner()函数中的变量n，因此报错。

* 每个函数内部的变量，其他函数也无法使用。例：

*#函数作用域示例2*

**def** inner1(n):

n = 10

**print**('inner', n)

**def** inner2():

**print**(n)

*#主程序*

inner1(10)

inner2() *#报错*

所提示的出错信息与前一段示例类似。  
因此函数内部变量一般称为**局部变量**。

* 自定义函数能够读取并使用主程序(同一个程序中的)中变量的值，因此主程序中的变量，也称为**全局变量**。例：

*#函数作用域示例3*

**def** inner():

**print**('inner', n)

*#主程序*

n=100

inner()

从示例3代码中可以看到，inner()函数内部并没有对n进行定义，但是能够从全局变量n中得到值100。但是自定义函数一般不能[^4]改变全局变量的值。例：

*#函数作用域示例4*

**def** inner():

n=10

**print**('inner', n)

*#主程序*

n=100

inner()

**print**('outer', n)

可知，虽然在自定义函数内部，将n赋值了10，但是在主程序中的全局变量n的值仍然是100。  
这个是怎么回事呢？是调用自定义函数inner()过程中，，全局变量n的值由100被赋值为10，然后从inner()函数返回的时候，n又被改回100了嘛？这似乎合理，但请看：

*#函数作用域示例5*

**def** inner():

**print**('inner', n)

n=10

*#主程序*

n=100

inner()

**print**('outer', n)

程序出现错误，local variable 'n' referenced before assignment，即局部变量n赋值前被使用/引用。示例4与5中，变量n被定义了(赋值就是一种定义)，因此其中的n虽然与全局变量n同名，但其实是局部变量。  
因此，示例4中使用的n是局部变量，其值为10，与全局变量n(值为100)无关；示例5中使用的n也是局部变量，在打印之前，并没有值，因此出错。只有示例3，由于函数内部没有对n进行定义，因此使用的是全局变量n的值。  
最后要说明的一点是，对在自定义函数中与全局变量同名的变量，只要在自定义函数内部对该同名变量进行了任何形式[^4]的赋值，无论在该函数体内的哪个位置，该同名变量即成为局部变量，与同名全局变量无关。以下是更多示例：

*#函数作用域示例6*

**def** inner():

**print**('inner', n)

n=n

*#主程序*

n=100

inner()

**print**('outer', n)

*#函数作用域示例7*

**def** inner():

n=n

**print**('inner', n)

*#主程序*

n=100

inner()

**print**('outer', n)

最后一个示例比较有趣，请读者自行分析。

* abs(x)

Return the absolute value of a number. The argument may be an integer or a floating point number. If the argument is a complex number, its magnitude is returned.

* divmod(a, b)

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using integer division. With mixed operand types, the rules for binary arithmetic operators apply. For integers, the result is the same as (a // b, a % b).

* round(number[, ndigits])

Return the floating point value number rounded to ndigits digits after the decimal point. If ndigits is omitted or is None, it returns the nearest integer to its input.

完整代码

*#求sigma1~m + sigma1~n + sigma1~k*

**def** compute\_sum(end):

i = 0

total\_n = 0

**while** i < end:

i = i + 1

total\_n = total\_n + i

**return** total\_n

n = int(input('请输入第1个整数，以回车结束。'))

m = int(input('请输入第2个整数，以回车结束。'))

k = int(input('请输入第3个整数，以回车结束。'))

**print**('最终的和是：', compute\_sum(m) + compute\_sum(n) + compute\_sum(k))

break及while...else  
首先对着函数的如下伪码，我们先一气呵成写完这个函数。

猜数函数流程：

请玩家输入一个整数n

在1-1000之间随机取一个整数number

取一个最多可猜测次数，玩家已猜测次数设为0，

一直做如下步骤直到已猜测次数大于最多可猜测次数：

让玩家猜测一次

玩家猜测次数加一

告诉玩家目前猜测次数

如果玩家猜中了：

恭喜猜中

显示数字及猜测次数

退出循环

如果玩家猜测数字大于number：

告诉玩家猜大了

否则：

告诉玩家猜小了

告诉玩家还能猜几次

否则：

显示超过次数，猜测失败。

* 请键入函数代码并观察运行结果。

import random, math

# 临时win(), lost()函数，测试用。

def win():

print('Win!')

def lose():

print('Lose!')

def guess\_game():

n = int(input('请输入一个大于0的整数，作为神秘整数的上界，回车结束。'))

number = random.randint(1, n)

max\_times = math.ceil(math.log2(n))

guess\_times = 0

while guess\_times < max\_times:

guess = int(input('请输入你猜测的整数，回车结束。'))

guess\_times += 1

print('一共可以猜', max\_times, '次')

print('你已经猜了', guess\_times, '次')

if guess == number:

win()

print('神秘数字是：', guess)

print('你比标准次数少', max\_times-guess\_times, '次')

break

elif guess > number:

print('抱歉，你猜大了')

else:

print('抱歉，你猜小了')

else:

print('神秘数字是：', number)

lose()

# 临时主程序，用于测试guess\_game()

guess\_game()

我们发现从根据伪码来写代码是一件较为轻松的事情，所有思路已经清楚，并记录下来，我们只要用已经掌握的python语法实现即可。  
ceil()是math模块中的函数，功能是由一个浮点数得到对应的向上取整的整数。  
break是python的关键字，用于退出break所在的当前层循环结构。

math.ceil(x)  
Return the ceiling of x, the smallest integer greater than or equal to x.  
math.floor(x)  
Return the floor of x, the largest integer less than or equal to x.  
math.exp(x)  
Return e\*\*x  
math.log(x[, base])  
With one argument, return the natural logarithm of x (to base e).  
With two arguments, return the logarithm of x to the given base.  
math.log2(x)  
Return the base-2 logarithm of x. This is usually more accurate than log(x, 2).  
math.log10(x)  
Return the base-10 logarithm of x. This is usually more accurate than log(x, 10).  
math.sqrt(x)  
Return the square root of x.  
math.sin(x)  
Return the sine of x radians.  
math.cos(x)  
Return the cosine of x radians.

序列之list

* 键入如下代码并观察运行结果。

# 序列之list示例1

numbers = [1, 2, 3, 4, 5]

print(numbers[1])

print(numbers[2])

print(numbers[3])

print(numbers[4])

print(numbers[0])

print(numbers[5]) # 错误，越界

* python语言用**序列**来存储一系列对象，其中每个对象对应一个从0开始的连续编号。
* list数据类型是一种序列，中文名称为**列表**可以存放数字、字符串等各类对象，存放的方式是：[对象1, 对象2, ... ,对象n]，即对象均放在[]之间，且用,分隔。
* 本段代码中，[1, 2, 3, 4, 5]就是一个list，这个list被赋值给了numbers，于是number也成为list类型，其值就是[1, 2, 3, 4, 5]。
* 序列中的对象根据其编号来进行定位，这个编号一般称为**索引**，取得序列某索引对象的方法是：序列名[索引]。
* 如果索引没在序列的范围之内，此时取该索引位置的对象，解释器就会提示出错。一般称为**索引越界**。
* 键入如下代码，运行多次，确保每个分支均被执行到，并观察运行结果。

# 序列之list示例2

numbers = [1, 2, 3, 4, 5]

n = int(input('请输入一个整数，回车结束'))

if n in numbers:

print('Yes,', n, 'is in numbers.')

else:

print('No,', n, 'is not in numbers.')

print(--------------分割线---------------)

if n+1 not in numbers:

print('No,', n+1, 'is not in numbers.')

else:

print('Yes,', n+1, 'is in numbers.')

* in是python关键字，可以用来判断某个对象是否在一个序列中，基本格式是：对象 in 序列，构成一个条件表达式，如果对象在序列中，则表达式的值为True，否则为False。
* 否定格式：对象 not in 序列，该条件表达式的值与基本格式的相反。其中not也是python关键字，意思是不。
* 与in相关，有两个内置函数any()和all()，但常用程度远不及in。
* 键入如下代码并观察运行结果。

# 序列之list示例3

print('any first:', any([0, 0, '', None, False]))

print('any second:', any([0, 5, '', None, False]))

print('all first:', all([5, 6, 'word', True]))

print('all second:', all([5, 10, 'fake', None, True]))

* any(...)是python内置函数，可以用来判断某个序列中是否存在真值，如果有，就返回True，如果没有（或者序列为空）则返回False。
* all(...)是python内置函数，可以用来判断某个序列中是否均为真值，如果均是，就返回True，否则返回False。
* 键入如下代码并观察运行结果。

# 序列之list示例4

words = ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']

i = 0

while i < 9: #目测words中有9个word

print(words[i])

i += 1

上述代码利用循环，取出list中的各个对象，这个过程称作**遍历**或**迭代**。

* 键入如下代码并观察运行结果。

# 序列之list示例5

numbers = [] # numbers = list()

print(numbers)

i = 0

while i < 9:

numbers.append(i)

print(numbers)

i += 1

i = 0

while i < len(numbers):

print('numbers[', i, ']=', numbers[i], sep = '')

i += 1

* []是空的list，可赋值给一个对象，该对象类型即成为list，值为一个空的list：[]，或者用numbers = list()，两者等价。例1-3中words与numbers分别通过赋值语句得到了list类型以及值，这个过程称作**初始化**。
* append()是python内置函数，但与前面模块内的函数和我们先前自定义的函数不同，append()是list类型对象的函数，任意一个对象被初始化成list类型以后，均可以使用append()函数，一般将类型的函数称为**方法**，如append()方法。  
  append()方法的功能是向一个list对象在尾部插入对象，用法是list对象.append(对象)，这个list对象可以是空list也可以非空，所加入对象的编号会立即得到对应位置的编号索引。
* len(...)是python内置函数，用法为len(iterable)，其中的参数iterable是一个序列，len函数可以返回该序列内所含元素的个数。

for循环

* 键入以下代码并观察执行结果

words = ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']

for word in words:

print(word)

* for是python关键字，与in共同组成for循环。for循环从序列中依次取对象(本例中就是从words中依次取出word)，每取一次对象执行一次语句块，直到取完序列中最后一个元素。基本用法如下：

for 对象 in 序列:

语句块

#蒙特卡洛方法求π的值

import random

def monte\_carlo\_method(n):

inside\_number = 0

for i in range(n):

x, y = random.random(),random.random()

if x\*x+y\*y < 1:

inside\_number += 1

return 4\*inside\_number/n

# 测试

number = int(input('input times please:'))

monte\_carlo\_method(number)

好，大功告成，但是感觉意犹未尽。我们期望能自己能初步实现个直观的可视化来展现效果。 这里介绍python可视化交互库Bokeh，最突出的特点是直接针web浏览器进行可视化展示。另一个使用较多的库是matplotlib，网上教程很多，请读者自行学习。

# 引入figure，output\_file及show函数

from bokeh.plotting import figure, output\_file, show

import random

# 设定绘图结果将输出到html文件'lines.html'

output\_file("lines.html")

经过以上几行代码后，就完成了绘图准备工作。

# 准备好画布，题目为'蒙特卡洛方法求π'，x轴坐标名称为'x'，y轴坐标名称为'y'

p = figure(title="蒙特卡洛方法求π", x\_axis\_label='x', y\_axis\_label='y')

至此，准备好了画布。

inside\_number = 0

for i in range(500):

x, y = random.random(),random.random()

if x\*x+y\*y < 1:

p.circle(x, y, size=3, color = 'red') # circle为画圆函数，x，y为坐标，size为大小，color为颜色

else:

p.circle(x, y, size=3, color = 'blue')

至此，500个圆点儿已经画在了画布上，但是还需要加一行代码让其显示出来。

# 显示结果

show(p)

我们还可以将画布直接嵌入到jupyter notebook之中，只需要略作修改。  
这里将计算与显示分开，先得到两类点的坐标，分别存放在list中，然后一并显示。

from bokeh.plotting import figure, output\_file, show

from bokeh.io import output\_notebook

import random

# 设定绘图结果将输出到notebook中'

output\_notebook()

# 准备好画布，题目为'蒙特卡洛方法求π'，x轴坐标名称为'x'，y轴坐标名称为'y'

p = figure(title="蒙特卡洛方法求π", x\_axis\_label='x', y\_axis\_label='y')

inside\_number = 0

inside\_xs = []

inside\_ys = []

outside\_xs = []

outside\_ys = []

for i in range(500):

x, y = random.random(),random.random()

if x\*x+y\*y < 1:

inside\_xs.append(x)

inside\_ys.append(y)

else:

outside\_xs.append(x)

outside\_ys.append(y)

# 画点

p.circle(inside\_xs, inside\_ys, size=3, color = 'red') # circle为画圆函数，x，y为坐标，size为大小，color为颜色

p.circle(outside\_xs, outside\_ys, size=3, color = 'blue')

# 显示结果

show(p)

* 任务2：打印一个乘法九九表。  
  我们先做如下尝试：

for i in range(10):

for j in range(10):

print('|', i, 'x', j, '=', i\*j, '|')

print('-----------------------------------------------------------------------------------------------------------')

好吧，我们打印出来的比九九表还厉害，但这些乘数被乘数中出现的0并不是我们想要的。我们可以通过给i及j加1，然后再控制循环的方式解决，请自行思考。但python为range()提供了更方便的用法。

* 键入以下代码并观察执行结果。

for i in range(1, 10):

for j in range(1, 10):

print('|', i, 'x', j, '=', i\*j, '|')

print('-----------------------------------------------------------------------------------------------------------')

可以为range()函数同时设置起始和终点参数，即range(start, end)，调用可以依次得到[stard, end)之间的整数，起始参数如果没有给出，则默认从0开始。完整的range()函数还有第三个参数，即间隔，请看示例：

for i in range(1, 100, 10):

print(i)

print('-----------------分隔线------------------')

for i in range(100, 10, -10):

print(i)

这么快就搞定了九九表的感觉很不错，但是我们还能做得更好。我们希望每个分割线之间的算式，是连成一排打印出来的。

* 键入如下代码观察执行结果。

for i in range(1, 10):

for j in range(1, 10):

print('|', i, 'x', j, '=', i\*j, '|', end='')

print()

print('---------------------------------------------------------------------------------------------------------------------------')

print()函数除了一些打印的位置参数以及sep关键字参数外，end是另一个关键字参数，用来控制当前print语句行尾打印字符的值，当不设置end关键字时，默认是打印换行。 已经完成的九九表基本已经可以使用了，但我们是希望尽量做到完美。目前的不完美主要是因为i\*j的结果可能是1位数，也可能是2位数，造成打印时候对不齐。因此我们需要用print()函数控制打印，使格式统一。

* 键入如下代码观察执行结果。

# 基本用法 1：

print('a={0},b={1},c={2}'.format(0,1,2))

print('a={1},b={0},c={2}'.format(0,1,2))

print('----------分隔符-------------')

# 基本用法 2：

n=20

print('{:3},{:3}'.format(n,10\*n))

print('{:3},{:3}'.format(10\*n,n))

format()是字符串类型的一个方法，可以利用format()方法来进行字符串的格式统一。  
基本用法1：'xxx{位置}xxx{位置}...xxx{位置}xxx'.format(变量, 变量..., 变量)。format中的多个变量是一个序列，从0开始自动连续编号，左侧字符串内的位置就是变量的索引(不要求一定顺序排列)，按照索引位置，由右侧各个变量的值来替换{位置}。  
基本用法2：'xxx{:整数}xxx{:整数}...xxx{:整数}xxx'.format(变量, 变量..., 变量)。左侧字符串内如果没有指定索引，则按照右侧序列变量顺序替换字符串内的{:整数}。{:整数}表示替换变量在字符串中的格式，其中的整数表示替换变量占据的宽度(字符个数)。  
好，就用到九九表中去吧！

* 键入如下代码观察执行结果。

print('------------------------------------------------------------------------------------------------------------')

for i in range(1, 10):

for j in range(1, 10):

print('|{} x {} = {:2}|'.format(i, j, i\*j), end = '')

print()

print('------------------------------------------------------------------------------------------------------------')

6.6 拓展与总结

* 字符转义  
  包含在单引号 ('...')或者双引号中的 ("...")均是字符串。 \ can be used to escape quotes. 实现字符转义：

print('doesn\'t') # use \' to escape the single quote.

print("doesn't") # or use double quotes instead.

s = 'First line.\nSecond line.' # \n means newline

s

# without print(), \n is included in the output

print(s) # with print(), \n produces a new line

如果想使在字符串中的\失效，则需要在字符串的首个引号前加一个字母r：

print('C:\some\name') # here \n means newline!

print(r'C:\some**\n**ame') # note the r before the quote

比较常用的python转义字符如下：

| **转义字符** | **转义后含义** |
| --- | --- |
| \\ | Backslash () |
| \' | Single quote (') |
| \" | Double quote (") |
| \a | ASCII Bell (BEL) |
| \b | ASCII Backspace (BS) |
| \f | ASCII Formfeed (FF) |
| \n | ASCII Linefeed (LF) |
| \r | ASCII Carriage Return (CR) |
| \t | ASCII Horizontal Tab (TAB) |

* 列表推导  
  对可迭代（遍历）对象如list，比较频繁的操作是：遍历list，取符合条件的元素子集并对其中每个元素进行某些操作，python提供了类似操作的简洁实现方式。

import math

numbers = [i for i in range(10)]

print(numbers, '\n')

numbers = [i+j for i in range(10) for j in range(10)]

print(numbers, '\n')

numbers = [math.sqrt(i) for i in range(10) if i%2==0]

print(numbers, '\n')

words = ['The', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']

selected\_words = [word for word in words if len(word) > 3]

print(selected\_words)

有关列表推导及相关使用，后续还将陆续介绍。

* sum(...), max(...), min(...)函数  
  即实现对序列求和、求最大值，最小值的函数。

numbers = [i+1 for i in range(10) if i%2==0]

print(sum(numbers), '\n')

numbers = [5, 7, 12, 100, 45]

print(max(numbers), min(numbers), '\n')

* pass, continue, break, return, exit()

以上关键字/函数的区别与练习可用以下表格概括：

| **关键字** | **功能** |
| --- | --- |
| pass | 什么也不做，纯占位用 |
| continue | 跳出本次循环 |
| break | 跳出本层循环 |
| return | 退出函数 |
| exit() | 退出程序 |

#pass, continue, break, return, exit()区别联系示例

for i in range(10):

pass

print(i)

print('-----------分割线----------------')

for i in range(10):

continue

print(i)

print('-----------分割线----------------')

for i in range(10):

for j in range(10):

print('i={},j={}'.format(i,j))

break

print('-----------分割线----------------')

def test():

for i in range(10):

for j in range(10):

print('i={},j={}'.format(i,j))

return

test()

print('-----------分割线----------------')

for i in range(10):

for j in range(10):

print('i={},j={}'.format(i,j))

exit()

print('-----------分割线----------------')

* str.format()

格式字符串的format()方法很多时候，配合print()一起使用，具体内容可参考： <https://docs.python.org/3/library/string.html#formatstrings>  
这里仅举一些实例进行一般的用法说明。

# str.format()示例1

print('{}, {}, {}'.format('a', 'b', 'c'))

print('{2}, {1}, {0}'.format('a', 'b', 'c'))

print('{0}{1}{0}'.format('abra', 'cad')) # arguments' indices can be repeated

# str.format()示例2

print('Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W'))

coord = [3, 5]

print('X: {0[0]}; Y: {0[1]}'.format(coord))

# str.format()示例3

print('{:<30}'.format('left aligned'))

print('{:>30}'.format('right aligned'))

print('{:^30}'.format('centered'))

print('{:\*^30}'.format('centered')) # use '\*' as a fill char

# str.format()示例4

print('{:+f}; {:+f}'.format(3.14, -3.14))

print('{: f}; {: f}'.format(3.14, -3.14))

print('{:,}'.format(1234567890))

6.7 完整代码

# 倒序打印用户输入的n个单词10次

words = []

n = int(input('请输入一个整数，表示将要输入的单词数，回车结束。'))

for i in range(n):

word = input('请输入一个单词，回车结束。')

words.append(word)

for i in range(10):

for j in range(n-1, -1,-1):

print(words[j])

7.2 序列之字符串(str)

从第一讲我们就接触到了字符串，无论是在print()函数中使用还是作为单行、多行注释来使用，以往均将字符串作为一个整体。事实上，字符串有一串字符组成，每个字符都有一个索引，索引从0开始，字符串是序列的一种，很多使用方法与列表(list)类似，字符串类型的关键字是str。

* 键入如下代码并观察执行结果。

# 示例代码 1

line = str()

print('用str()初始化：', line == '')

line = 'abcdefghijklmn'

print('a' in line)

print('a' not in line)

for ch in line:

print(ch)

print('--------------分隔线----------------')

for i in range(10):

print(i, line[i])

* 可以利用循环求得字符串的长度(含有的字符数)

# 示例代码 2

def my\_len(line):

len = 0

for ch in line:

len += 1

return len

# 主程序，测试函数用

my\_len('abcde')

实际上，由于求序列长度(内含对象个数)的需求十分频繁，python提供的内置方法len()不但可得到list对象所含元素个数，其实是可以得到**序列**对象所含元素个数，当然也包括str对象。

* 键入如下代码并观察执行结果。

# 示例代码 3

words = ['1','2','3','1','2','3']

print(len(words))

line = 'abcdefg'

print(len(line))

本段代码中形如序列[m:n:i]的操作称为**序列切片**，就是从序列中取出索引在[m,n)之间，以i为间隔的所有对象，默认i为1。

line = '北京语言大学信息科学学院'

print(line[2:6])

print(line[2:8:2])

print(line[8:1:-2])

print('-'\*15 + '分割线' + '-'\*15)

print(line[0:6])

print(line[:6])

print(line[6:12])

print(line[6:])

print(line[:])

print(line[::2])

print('-'\*15 + '分割线' + '-'\*15)

print(line[-5:-2])

print(line[0:4] + line[3::-1]) # 请对比示例代码 9

语言大学

语大信

科信大语

---------------分割线---------------

北京语言大学

北京语言大学

信息科学学院

信息科学学院

北京语言大学信息科学学院

北语大信科学

---------------分割线---------------

息科学

北京语言言语京北

In [ ]:

7.4 序列之元组(tuple)

* 键入如下代码并观察执行结果。

# 示例代码 12

line = ('赏', '花', '归', '去', '马', '如', '飞')

for i in range(1, len(line)\*2):

if i == 1:

print((line[0],))

elif i%2 == 1:

print(line[:i//2] + (line[i//2],) + line[i//2-1::-1])

else:

print(line[:i//2] + line[i//2-1::-1])

python内置类型中，与列表(list)及字符串(str)最相近的序列是元组(tuple)，tuple即为元组的关键字。

* 键入如下代码并观察执行结果。

# 示例代码 13

numbers = ()

print(numbers)

numbers = tuple()

print(numbers)

numbers = (1)   #数字而非元组

print(numbers)

numbers = (1,)

print(numbers)

numbers = 1,

print(numbers)

numbers = (1, 2, 3, 4, 5)

print(numbers)

numbers = 1, 2, 3, 4, 5

print(numbers)

可见元组(tuple)的初始化与列表(list)类似，元组用()，()也可以省略，但建议只在不降低程序可读性时省略。

* 键入如下代码并观察执行结果。

# 示例代码 14

numbers = ['102', 20]

x, y = numbers

print(x)

print(y)

a, b, c = x

print(a, b, c)

c, a, b = a, b, c

print(a, b, c)

序列内的对象，可以按照对象个数同时赋值给多个变量/对象，称为**解耦**。 本段中语句c, a, b= a, b, c，右侧实际上是利用三个变量形成了一个元组，然后解耦，在一行中同时完成了3个变量之间值的交换。

* 键入如下代码并观察执行结果。

# 示例代码 15

numbers = 1, 2, 3, 4, 5

for number in numbers:

print(number)

print('-'\*30)

numbers \*= 2

for i in range(10):

print(numbers[i])

print('-'\*30)

numbers = (1, 2, 3, 4, 5) \* 4

print(numbers)

可见元组的遍历、索引及乘法与列表及字符串类似。

* 键入如下代码并观察执行结果。

# 示例代码 16

def three\_return\_value():

return 1, 2, 3

print(three\_return\_value())

def three\_return\_word():

return 'one', 'two', 'three'

print(three\_return\_word())

def two\_return\_tuple():

numbers1 = 1,2,3

numbers2 = 4,5,6

return numbers1, numbers2

print(two\_return\_tuple())

def two\_return\_list():

numbers1 = [1,2,3]

numbers2 = [4,5,6]

return numbers1, numbers2

print(two\_return\_list())

可见，如函数返回多个值，则多个值将以元组的形式返回。同时，元组与列表内的元素还可以是元组或列表，一般称为**嵌套**。

* 键入如下代码并观察执行结果。

# 示例代码 17

numbers = [(1,2,3), (4,5,6), (7,8,9)]

print(numbers[0][0])

print(numbers[1][0])

print(numbers[2][2])

print('-'\*30)

for number in numbers:

print(number)

for num in number:

print(num)

虽然看似复杂，但仍然可以利用与先前类似的方法遍历以及利用索引取对象，索引顺序是从外层到内层。  
回到用非字符串解决回文图案的问题，如何利用元组或列表实现以及如何将图案扶正，留待读者自行思考解决。

7.5 可变与不可变数据类型

综上可见，元组与字符串一旦被初始化，其值就不能被改变，python语言称为**不可变数据类型**，而类似列表这样的数据类型，初始化后，其值可以改变，就称为**可变数据类型**。  
除了元组及字符串外，python中数值常量也是不可变数据类型，比如说，5就是5，不可能等于6，也不能等于5.1。  
好，数值常量的比较容易理解。那么可变与不可变的区分，比如列表与元组区别开有什么意义呢？元组不可变，因此对元组操作很多时候还还需要建立新元组，多麻烦。全部设计成可变的不是更好吗？实际上，元组单独作为一种数据类型，有其特别的考虑：一个原因是相同操作时，一般元组比列表速度更快；另外，利用元组的不可变性可以保护某些本来就不打算更改的序列；最后，后续一些数据类型如dict，只能用不可变类型作为键。

* 键入如下代码并观察执行结果。

# 示例代码 23

numbers\_1 = [1,2,3,4,5]

numbers\_2 = numbers\_1

print(numbers\_2)

numbers\_1[0] = 9

print(numbers\_2)

哎呦，好奇怪，明明没有对numbers\_2进行操作，只改变了numbers\_1啊，怎么会连带numbers\_2的值也给改了？这不是我的本意。  
让我们从赋值开始，详细解释下原因。  
什么是赋值？python语言中，赋值就是让一个变量/对象指向一个具体值，即常量，如：number = 10，就是先产生一个常量10，然后让number指向10。当使用变量/对象值运算时候，就取出其指向的常量具体值来参与运算。  
常量是不可变的，因此如果将number再次赋值，如number = 20，则是先产生一个常量20，然后让number指向20。为变量/对象再次赋值就是更改变量/对象的指向。  
一个变量/对象，在一个时刻，只能指向一个常量。此时，number仅指向20，不再指向10。  
一个常量，可以被多个变量/对象指向。

（此处画图说明） （此处画图说明） （此处画图说明）

* 键入如下代码并观察执行结果。

# 示例代码 24

number\_1 = 10

number\_2 = number\_1

print(id(number\_1), id(number\_2))

print('-'\*30)

number\_1 = 20

print(id(number\_1), id(number\_2))

本段代码中的id()是一个函数，可以返回一个变量/对象在计算机中的唯一标识，用法是id(对象)，这个唯一标识其实就是对象在计算机(内存)中存放的地址。  
可以看到，当将number\_1赋值给number\_2后，两者id相同，此时number\_1与number\_2都指向常量10，类似同一个对象起了两个名称，可以说是互为**别名**。  
当number\_1被再次赋值后，其id发生了改变，与number\_2没有关系了。

* 将先前‘非我们本意’的代码，加入id()函数仔细观察：

# 示例代码 25

numbers\_1 = [1,2,3,4,5]

numbers\_2 = numbers\_1

print('n1={}, n2={}'.format(id(numbers\_1), id(numbers\_2)))

numbers\_1[0] = 9

print(numbers\_1, numbers\_2)

print('n1={}, n2={}'.format(id(numbers\_1), id(numbers\_2)))

由本段代码可知，numbers\_1与numbers\_2的id相同，是指向同一个列表[1,2,3,4,5]的两个‘别名’，列表是可变的，因此这个列表变成[0,2,3,4,5]之后，指向这个列表的两个别名，值自然都变成了[0,2,3,4,5]。  
现在问题来了，如何能得到两个真正不同的列表，其值是相同但并非指向同一个对象？

# 示例代码 26

numbers\_1 = [1,2,3,4,5]

numbers\_2 = [1,2,3,4,5]

print(id(numbers\_1), id(numbers\_2))

上面这样是可以的，但是一个是比较重复，二来在多数程序中，可能列表的值是计算得到，是变量，难以这样显示的直接赋值。我们不考虑numbers\_1 = numbers\_2，因为已经确定行不通。

# 示例代码 27

numbers\_1 = [1,2,3,4,5]

numbers\_2 = numbers\_1.copy()

print(id(numbers\_1), id(numbers\_2))

numbers\_1[0] = 9

print(numbers\_1, numbers\_2)

python内置浅拷贝方法copy()，可以对可变序列进行浅拷贝复制，用法是：可变序列.copy。称为浅拷贝(shallow copy)的原因，是因为拷贝的确实不够深入。在实际使用中，浅拷贝多用切片来代替（也是浅拷贝）。最后，如果不注意，还是有产生‘非我们本意’情况的可能。

# 示例代码 28

numbers\_1 = [1,2,3,4,[9,8,7]]

numbers\_2 = numbers\_1.copy()

numbers\_3 = numbers\_1[:]

print(id(numbers\_1), id(numbers\_2), id(numbers\_3))

numbers\_1[4][0] = 5

print(numbers\_1, numbers\_2, numbers\_3)

我们非常遗憾的看到numbers\_1、numbers\_2及numbers\_3还是同时变化了。  
问题出在浅拷贝上，浅拷贝在拷贝这种嵌套结构(也称为复合结构)时，对复合结构内部的结构如序列，并非将其值拷贝，而是仅建立了指向。此时被拷贝对象和新对象虽然指向不同的复合结构，但是这两个复合结构内部，都存在指向相同结构的指向。（用图说明）。  
上例中，浅拷贝后，numbers\_1、numbers\_2及numbers\_3的索引为4的位置，同时指向列表[9,8,7]，将[9,8,7]改变成[5,8,7]以后，自然同时影响numbers\_1、numbers\_2及numbers\_3。  
与浅拷贝对应，就有深拷贝(deepcopy)。

# 示例代码 29

import copy

numbers\_1 = [1,2,3,4,[9,8,7]]

numbers\_2 = copy.deepcopy(numbers\_1)

print(id(numbers\_1), id(numbers\_2))

numbers\_1[4][0] = 5

print(numbers\_1, numbers\_2)

要使用deepcopy()方法，需要先导入copy模块，基本用法为：copy.deepcopy(对象)，实现深拷贝。

7.6 拓展与总结

* 序列通用操作 用下表来进行说明，令x = 1，i=2，n=3，j=7，k=2，s = [1,2,3,4,5,6,7]，t = [8,9,10]。

| **操作** | **结果或s的值** | **说明** |
| --- | --- | --- |
| x in s | True | 如s中有一个对象等于x，返回True否则返回False |
| x not in s | True | 如s中有一个对象等于x，返回False否则返回True |
| s+t | [1,2,3,4,5,6,7,8,9,10] | 将s与t连接 |
| n\*t或t\*n | [8,9,10,8,9,10,8,9,10] | 相当于将t自加n次 |
| s[i] | 3 | 序列s的第i个对象，起始对象为第0个 |
| s[i:j] | [3,4,5,6,7] | s从i到j之间的切片(不包含j) |
| s[i:j:k] | [3,5,7] | s从i到j之间间隔为k的切片(不包含j) |
| len(s) | 7 | s的长度(对象个数) |
| max(s) | 1 | s中最小的对象 |
| min(s) | 7 | s中最大的对象 |
| s.index(x) | 1 | s中x第一次出现时的索引位置 |
| s.count(x) | 1 | s中x出现的个数 |

* 可变序列操作 用下表来进行说明，令x = 1，i=2，n=3，j=7，k=2，s = [1,2,3,4,5,6,7,8]，t = [9,10,11]。

| **操作** | **结果或s的值** | **说明** |
| --- | --- | --- |
| s[i] = x | [1,2,1,4,5,6,7,8] | s中第i个对象替换成x |
| s[i:j] = t | [1,2,9,10,11,8] | 将s中i到j(不包括j位置对象)的切片替换为序列t中的对象。 |
| del s[i:j] | [1,2,8] | 等价于s[i:j] = [] |
| s[i:j:k] = t | [1,2,9,4,10,6,11,8] | 将s中i到j(不包括j位置对象)间隔为k的切片依次替换为序列t中的对象。 |
| del s[i:j:k] | [1,2,4,6,8] | 将s中i到j(不包括j位置对象)间隔为k位置的对象删除。 |
| s.append(x) | [1,2,3,4,5,6,7,8,1] | 将x加入到s的末端，等价于s[len(s):len(s)] = [x] |
| s.clear() | [] | 清空s中所有元素，等价于del s[:]，或s[:] = [] |
| s.copy() | [1,2,3,4,5,6,7,8] | s的浅拷贝，等价于s[:] |
| s += t 或者 s.extend(t) | [1,2,3,4,5,6,7,8,9,10,11] | s后面扩展t |
| t \*= n | [9,10,11,9,10,11,9,10,11] | t中内容重复n次后再复制给t |
| s.insert(i, x) | [1,2,1,3,4,5,6,7,8] | 在s的第i个位置插入元素x，等价于s[i:i] = [x]。 |
| s.pop([i]) | [1,2,4,5,6,7,8] 并返回3 | 从s中删除第i个元素，并将其作为返回值 |
| s.remove(x) | [2,3,4,5,6,7,8] | 从s中删除第一个x对象。 |
| s.reverse() | [8,7,6,5,4,3,2,1] | 将s进行就地反转。 |

* 切片索引图

-+---+---+---+---+---+---+

| P | y | t | h | o | n |

+---+---+---+---+---+---+

0 1 2 3 4 5 6

-6 -5 -4 -3 -2 -1

核心任务：点字成语/点字成诗游戏

8.1 任务简介与分析

* 任务简介

点字成诗游戏基本与中国诗词大会中的相同，这里仅考虑5言及7言诗词句，前者给出3X3共9个字，包含5言中的所有字，但不与其重复，随机打乱；后者与前者类似，但是给出3X4共12个字，随机打乱。 点字成语与点字成诗类似，成语只考虑4个字，将给出3X2共6个字随机打乱。 此外，还要考虑玩家积分，后续考虑设计等级提升等。

* 任务分析

成语库可以从网上下载搜集，然后放到文件中。可以将所有成语放到序列或者元组中，新增不在成语中的字则需要构建一个字表，，随机抽取及随机打乱则需要利用random模块。  
点字成诗与成语库类似。

8.2 文件读取与写入

请读者下载成语.txt，链接: <http://pan.baidu.com/s/1bpeUF87> 密码: wxni,并将文件保存到目录:d:\temp\下。  
首先，用记事本或者其他文本编辑器打开该文件，查看文件内容。

行尸走肉、

金蝉脱壳、

百里挑一、

金玉满堂、

背水一战、

霸王别姬、

天上人间、

不吐不快、

海阔天空、

情非得已、

满腹经纶、

...

我们要从文件中读取这些内容。

* 键入如下代码并观察执行结果。

# 示例代码8-1

fh = open(r'd:**\t**emp**\成**语.txt')

text = fh.read()

fh.close()

print(len(text))

print(text)

示例代码8-1中：

* open()是打开文件的内置函数，其以只读方式打开文件基本用法是：open(filename)，其中参数filename是一个字符串，内含一个文件全名(路径名/文件名.扩展名)，功能就是打开与filename对应的文件，返回一个文件对象。  
  本例中，文本文件成语.txt打开以后，文件对象被赋值给fh。
* read()是读取文件内容的方法，其基本用法是：文件对象.read()，该方法可以将文件中所有内容作为一个字符串读入。  
  本例中，成语.txt内的所有内容作为一个字符串读入后，被赋值给text。
* close()是关闭文件的方法，其基本用法是：文件对象.close()。注意，文件对象在程序退出前，必须关闭，否则有时会出现难以预知的错误。

下一步将在text中提取每一个成语，并将所有成语放入到一个列表中。一个思路是考虑是否可以利用字符串切片，按照四字成语的规律，循环将成语切出后放入列表。但仔细观察文件内容，发现首先有一些3字词，有一些多于四字，并且并非所有条目的末端都有符号、所以很难找到规律快速切出成语。 我们换一种思路，以每行的换行符\n定位成语末尾。

# 示例代码8-2

text = text.replace('、', '')

start = 0

end = 0

idioms = []

while end < len(text):

if text[end] == '\n':

idioms.append(text[start:end])

start = end + 1

end += 1

print(idioms)

示例代码8-2中：

* replace()是替换字符的方法，其基本用法是：字符串.replace(old\_str, new\_str)，old\_str·与·new\_str两个参数分别是字符串中将被替换的字符子串及新字符串，该方法将字符串中所有的old\_str替换成new\_str。  
  本例中将text中所有的、替换成''，即删去所有的、，这样，每个条目的末端就只剩下换行符：\n。  
  然后利用while循环，找到每次\n出现的位置，也就是每次切片的末端位置。  
  代码虽然并不复杂，考虑到通过某个分隔符将文本分开并放入到列表中这样的任务非常常见，当然可以将这段代码做成函数，以便复用。但是python提供了更好的解决办法。

# 示例代码8-3

text = text.replace('、', '')

idioms = text.split('\n')

print(idioms)

示例代码8-3中：

* split()是将字符按照分隔符分开的方法，其基本用法是：字符串.split(sep)，该方法将字符串按照分隔符sep分隔成多个子串，返回一个列表，列表每个对象为一个子串。  
  得到成语列表以后，我们应该将这个文件保存，因为以后多次运行游戏的时候，我们就可以直接用这个列表，而不用每次都进行上述处理。

# 示例代码8-4

fh = open(r'd:**\t**emp**\i**dioms\_correct.txt', 'w')

fh.writelines(idioms)

fh.close()

示例代码8-4中：

* 在open()函数中，多了一个参数'w'，该参数是英文write的缩写，参数指明文件以写入的方式打开。
* open()函数的基本用法是：open(filename, mode)，其中mode是一个字符串，默认为r，即只读(只能读取)方式，'w'是只写方式，且在写入前如果已有文件，则清空文件，否则新建文件。
* writelines(...)是像文件写入序列的方法，其基本用法是：文件对象.writelines(...)，该方法将序列中所有文件依次写入文件对象。

然后我们打开d:\temp\目录，发现确实新建立了文件idioms\_correct.txt，打开这个文件，会发现所有成语连到一起，这是因为writelines(...)方法是将序列所有对象连续写入文件。

# 示例代码8-5

fh = open(r'd:**\t**emp**\i**dioms\_correct.txt', 'w')

for idiom in idioms:

correct\_idioms.append(idiom+'\n')

fh.writelines(correct\_idioms)

fh.close()

示例代码8-5将每个idiom加入换行符，再利用writelines(...)写入就没有问题了。也可以利用**列表推导**，使代码更简洁。

# 示例代码8-6

fh = open(r'd:**\t**emp**\i**dioms\_correct.txt', 'w')

idioms = [idiom + '\n' for idiom in idioms]

fh.writelines(idioms)

fh.close()

示例代码8-6中：

* [idiom + '\n' for idiom in idioms]是列表推导，功能与示例代码8-5中for循环结构功能类似，都是将idioms中每一个idiom取出后加上\n，最后形成一个列表。这里让idioms指向用列表推导得到的新列表，没有必要再如示例代码8-5那样定义新的变量correct\_idioms。 也可以采用如下方式保存：

# 示例代码8-7

fh = open(r'd:**\t**emp**\i**dioms\_correct.txt', 'w')

for idiom in idioms:

fh.write(idiom + '\n')

fh.close()

示例代码8-7中：

* write()是向文件写入字符串的方法，其基本用法是：文件对象.write(str)，该方法将一个str写入文件对象。因此需要加入换行符，将每个成语独立成一行。

更为常用的方法是：

# 示例代码8-8

fh = open(r'd:**\t**emp**\i**dioms\_correct.txt', 'w')

fh.write('\n'.join(idioms))

fh.close()

示例代码8-8中：

* str.join(...)将序列内所有元素之间均用str进行连接的方法，其基本用法是：分隔字符串.join(...)。

前面只是完成了成语搜集和保存的工作，现在我们考虑游戏过程的实现。首先写出游戏main()函数的伪码：

# main()函数伪码

显示游戏菜单

一直做如下步骤：

等待用户选择

如果选择1：

则显示游戏玩法说明

显示游戏菜单

如果选择2：

则进入成语游戏

如果选择3：

则进入诗词游戏

如果选择4：

则显示GAME OVER

退出游戏

如果选择5：

则显示游戏制作团队

显示游戏菜单

main()函数部分与第5节的猜数游戏相关内容类似，请读者自行实现。  
然后写出成语游戏函数的伪码：

# 成语游戏函数伪码

设定初始玩家分数

设已猜测成语表为空

打开文件读入成语列表

形成成语用字表

取成语列表中第一个成语作为idiom

当玩家分数大于0时，则一直做如下步骤：

当idiom在已猜测成语表中，则一直做如下步骤：

在成语列表中随机取一个4字成语idiom

否则：

显示胜利界面

显示游戏者最后得分

退出成语游戏函数

在idiom中随机加入不在这个idiom中的2个字

将加字的idiom按照3X2形式显示在屏幕

玩家输入猜测成语字符串guess

如果guess==idiom：

玩家分数加10分

将此idiom加入到已猜测成语表

将此idiom从成语列表中删除

否则：

玩家分数减10分

否则：

显示失败页面

退出成语游戏函数

首先是读取文件，我们需要统计得到所有成语中都出现了哪些字。

8.3 字表统计  
首先给出伪码如下：

令字符表`ch\_talbe`为空列表

取字符串中每一个字:

如果该字不在字符表中则:

将其加入字符表

返回字符表

根据伪码写如下代码：

# 示例代码8-9

def get\_ch\_table(line):

ch\_table = []

for ch in line:

if ch not in ch\_table:

ch\_table.append(ch)

return ch\_table

# 主程序

fh = open(r'd:**\t**emp**\i**dioms\_correct.txt')

text = fh.read()

chs = get\_ch\_table(text.replace('\n', ''))

print(len(chs), chs)

示例代码8-9中：

* 我们将求一个字符串的字表做成了一个函数get\_ch\_table(str)，参数是字符串，该函数返回字表的list。
* 主程序中首先读入文本文件，将文本文件中的内容赋值给text中，然后利用replace()函数，将整个字符串中的成语去掉换行符连成一个字符串，并作为函数get\_ch\_table()的参数。
* 调用get\_ch\_table()，就得到了所有词条的字表。

目前的方法已经能够解决问题，但并非最佳方案，字表统计其实有更佳的解决办法，将在后续章节中详细介绍。

8.4 随机抽取序列元素

# 示例代码8-10

import random

idioms = text.split()

for i in range(15):

idiom = random.choice(idioms)

print(idiom)

示例代码8-10中：

* split()如果不带参数，则默认以空格(或连续空格，含换行、换页及回车等)作为分隔符。这样，text通过split()方法，就得到了成语列表idioms。
* choice()是在序列随机抽取对象的方法，其基本用法是：random.choice(seq)，该方法在参数seq指定的指定序列中随机取一个对象作为返回值。

可以看到，随机抽取了15次，抽取的idiom没有什么特殊规律。 下一步需要解决的问题就是，随机从字表中抽取不在成语中的2个字，与随机得到的成语idiom中的4个字，合并为6个字，并以3X2的形态输出。

# 示例代码8-11

import random

idiom = '千钧一发' #假设抽取到了这个'成语'

fh = open(r'd:**\t**emp**\i**dioms\_correct.txt')

text = fh.read()

chs = get\_ch\_table(text.replace('\n', ''))

guess\_ch\_table = [ch for ch in idiom]

while len(guess\_ch\_table) < 6:

ch = random.choice(chs)

if ch not in guess\_ch\_table:

guess\_ch\_table.append(ch)

random.shuffle(guess\_ch\_table)

for i in range(0,6,2):

print(guess\_ch\_table[i], guess\_ch\_table[i+1])

示例代码8-11中：

* 首先通过列表推导得到了‘千钧一发’的4字字表。
* 然后利用while循环，使guess\_ch\_table的大小控制在6个字内的条件下，从字表chs中随机抽取汉字，加入到guess\_ch\_table中。
* shuffle(...)是random模块中的随机打乱序列内部对象顺序的方法，基本用法为：random.shuffle(序列)。
* 最后将6个字的字表，按照3X2的形态打印出来。

有示例代码8-1到8-11，读者应该能够将这个游戏实现，别忘记限制所有成语是4个字。  
对点字成诗，与点字成语的实现类似，区别有三：原始数据文件、诗词长度、字的输出形态。请读者自行完成。

8.4 相关任务

* 诗词大会机器人设计任务。假设机器人能够读取保存所有诗词的文件，假设机器人已经能够把看到的字转换为文本，且已经得到含有9或12个字的列表，在这个前提下，我们来设计诗词大会答题机器人。  
  任务分析：  
  一个简单的思路就是，将9或12字的列表中所有的字，取其中任意5或7个字进行的全排列，然后看哪个排列在诗词列表中：

# 示例代码8-12

import itertools

text = ('清','行','人','断','肠','西','明','时','节','雨','纷','纷')

guesses = itertools.permutations(text, 7)

fh = open(r'd:**\t**emp**\诗**词大全.txt')

poems = fh.read().split()

fh.close()

for guess in guesses:

if ''.join(guess) in poems:

print('答案是：' guess)

示例代码8-11中：

* permutations(..., int)是得到序列全排列的函数，第一个参数是序列，第二个参数是取第一个序列中多少个元素来做全排列。该函数在itertools模块中，需要提前import。

要有一定的耐心，等待我们刚刚设计的‘机器人’输出答案，以这样的速度，毫无疑问，机器人将在诗词大会上出糗！  
我们来简单分析一下原因。itertools.permutations(text, 7)将产生P(12, 7) = 3991680 ，约400万个排列结果。诗词大全.txt中的诗词约有500句，则需要进行400万\*500次=20亿次字符串是否相等的比较。假设当前个人电脑每秒可以进行1000万次这种字符串比较，则完成这个任务需要200秒，即3分20秒。

我们可以换个更快的电脑，也可以换个思路，思路决定出路。 可以反过来，将每一个诗词的条目与题目相比较，如果条目中所有字均在题目之中，则该条目即为答案。  
那么需要多少运算次数和时间呢？对7言诗词，以7言为例进行估算，每个条目有7个字，每个字需要与题目中的12个字进行比较，一共有500个词条，则总共需要比较7x12x500=42000次，按照每秒1亿次字符串比较的计算速度，计算时间在0.001秒以内。

# 点字成诗机器人

def find\_poem\_sentence(poems, characters):

for poem in poems:

same\_character\_number = 0

for ch in poem:

if ch in characters:

same\_character\_number += 1

if same\_character\_number == 7:

return poem

return None

点字成诗机器人代码中：

* 函数需要两个参数，第一个参数poems是所有诗词的列表，其中每个元素为一句诗词；参数characters是给定的9或12字的列表。
* 如果存在正确诗句，则返回诗句；否则，返回None。

可以仿照点字成语游戏来进行诗词大会出题，并编写机器人进行答题。

8.5 拓展与总结

* random模块 random模块详解见：<https://docs.python.org/3/library/random.html> 这里列举最基本的使用：

| **操作** | **结果** | **说明** |
| --- | --- | --- |
| random() | 0.37444887175646646 | 返回[0.0, 1.0)之间的随机浮点数 |
| uniform(2.5, 10.0) | 3.1800146073117523 | 返回[2.5, 10.0)之间的随机浮点数 |
| randrange(10) | 6 | 返回[0,10)之间的随机整数 |
| randrange(0, 101, 2) | 28 | 返回[0,101)之间的随机偶数 |
| choice(['win', 'lose', 'draw']) | win | 从序列中随机选择一个元素 |
| shuffle(['win', 'lose', 'draw']) | ['win', 'draw', 'lose'] | 将序列随机打乱 |
| sample([10, 20, 30, 40, 50], k=4) | [10, 50, 30, 20] | 序列中随机取k个样本 |

* 字符串方法 作为一种重要的内置数据类型，python内部提供了字符串专用的很多函方法，具体可见：<https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str> 这里列出常用的部分如下：

| **操作** | **实例** | **结果** | **说明** |
| --- | --- | --- | --- |
| str.count(sub[, start[, end]]) | '北京语言大学在北京宇宙中心'.count('北京') | 2 | 返回子串sub在str中的出现次数(子串不重叠计算)，start与end为可选参数，表示在这个索引的切片内进行count()操作。 |
| str.endswith(suffix, [start, [, end]]) | '北京语言大学在北京宇宙中心'.endswith('中心') | True | 如果str以suffix结尾，则返回True，否则返回False，start与end为可选参数，表示在这个索引的切片内进行endswith()操作。 |
| str.startswith(prefix, [start, [, end]]) | '北京语言大学在北京宇宙中心'.startswith('北') | True | 如果str以prefix开头，则返回True，否则返回False，start与end为可选参数，表示在这个索引的切片内进行endswith()操作。 |
| str.find(sub, [start, [, end]]) | '北京语言大学在北京宇宙中心'.find('大学') | 4 | 如果sub在str中，则返回其起始索引，否则返回-1，start与end为可选参数，表示在这个索引的切片内进行find()操作。 |
| str.index(sub, [start, [, end]]) | '北京语言大学在北京宇宙中心'.index('大学') | 4 | 与find()类似，但当sub不在str时，会报错：ValueError。 |
| str.join(iterable) | '-'.join('北京语言大学') | '北-京-语-言-大-学' | 用str将iterable中的元素连接。 |
| str.lower() | 'Beijing Language and Culture University'.lower() | 'beijing language and culture university' | 将str中所有能够小写的字母小写并输出新的str。 |
| str.upper() | 'beijing language and culture university'.upper() | 'BEIJING LANGUAGE AND CULTURE UNIVERSITY' | 将str中所有能够大写的字母大写并输出新的str。 |
| str.replace(old, new[, count]) | '北京语言文化大学'.replace('语言文化', '语言') | '北京语言大学' | 将str中old子串用new子串替代并返回新的str，count参数可选，表示替换的个数(从左至右)。 |
| str.split(sep=None, maxsplit=-1) | '1,2,3'.split(',') | ['1', '2', '3'] | 将str中的对象用分隔符sep分割开并作为一个list返回。 |
| str.split(sep=None, maxsplit=-1) | '1,2,3'.split(',', maxsplit = 1) | ['1', '2,3'] | 参数maxsplit为分隔次数，默认为-1，表示分隔次数没有限制。 |
| str.split(sep=None, maxsplit=-1) | '1<>2<>3'.split('<>') | ['1', '2', '3'] | 分隔符sep可以是多个字符。 |
| str.split(sep=None, maxsplit=-1) | ' 1 2 3 '.split() | ['1', '2', '3'] | 分隔符sep没有指定时，则以空格(或连续空格，含换行、换页及回车等)作为分隔符。 |
| str.strip([chars]) | www.example.com'.strip('cmowz.') | 'example' | 在str的两端，删除chars所包含的字符(1或者多个)，并返回剩余的字符串。 |
| str.strip([chars]) | ' spacious '.strip() | 'spacious' | 如不指定chars，则默认删除两端的空白字符。 |

* open()

由于文件在使用完毕后如果忘记关闭，有时会产生严重后果如文件内容丢失等，因此，open()函数也常常被这样使用：

# 示例代码8-13

with open(r'd:**\t**emp**\'**成语.txt') as fh:

text = fh.read()

print(text)

示例代码8-13中：

* open()函数打开了一个文件，然后利用python内置的with与as关键字将其赋值给as后的变量fh。
* 利用read()函数读取文件内容后，with...as控制的代码块执行完毕，则文件被自动关闭。

打开文件有多种模式，open()函数的完整介绍详见：[https://docs.python.org/3/library/functions.html?highlight=open#open，常用的如下：](https://docs.python.org/3/library/functions.html?highlight=open#open%EF%BC%8C%E5%B8%B8%E7%94%A8%E7%9A%84%E5%A6%82%E4%B8%8B%EF%BC%9A)

| **模式** | **解释** |
| --- | --- |
| 'r' | 只读方式打开文件(默认模式) |
| 'w' | 只写方式打开文件(如无则创建，如有先清空文件) |
| 'x' | 只写方式打开文件（如无则创建，如有则报错) |
| 'a' | 只追加方式打开文件(如无则创建，如有则在原有内容后面写入) |
| 'r+' | 读写方式打开文件(如无则报错，如有则可读可写，读写文件从当前位置开始) |
| 'w+' | 写读方式打开文件(如无则创建，如有则可读可写，第一次写入时清空文件，) |
| 'a+' | 读追加方式打开文件(如无则创建，如有则可读可写，在原有内容后面写入) |

8.6 完整代码

# 点字成语游戏完整代码

import random

def win():

# 可将小节5中的同名函数代码复制

return

def lose():

# 可将小节5中的同名函数代码复制

return

def get\_ch\_table(line):

ch\_table = []

for ch in line:

if ch not in ch\_table:

ch\_table.append(ch)

return ch\_table

def idiom\_robot(file\_name):

with open(file\_name) as fh:

text = fh.read()

idioms = text.split()

idiom = random.choice(idioms)

chs = get\_ch\_table(text.replace('\n', ''))

guess\_ch\_table = [ch for ch in idiom]

while len(guess\_ch\_table) < 6:

ch = random.choice(chs)

if ch not in guess\_ch\_table:

guess\_ch\_table.append(ch)

random.shuffle(guess\_ch\_table)

for i in range(0,6,2):

print(guess\_ch\_table[i], guess\_ch\_table[i+1])

return idiom

def main():

filename = r'd:**\t**emp**\i**dioms\_correct.txt'

score = 10

while score >= 0:

real\_idiom = idiom\_robot(filename)

answer\_idiom = input('请输入猜测成语，回车结束，直接回车表示退出游戏：')

if answer\_idiom == real\_idiom:

print('答对了，加十分')

score += 10

print('你当前的分数是：', score)

if score == 100:

win()

return

elif answer\_idiom == '':

print('退出游戏。')

print('你最后的分数是：', score)

return

else:

score -= 10

print('答错了，减十分')

print('成语其实是：', real\_idiom)

print('你当前的分数是：', score)

else:

lose()

return

if \_\_name\_\_ == '\_\_main()\_\_':

main()

核心任务：统计一个文本文件的词表与词频

9.1 任务简介与分析

* 任务简介  
  给定一个文本文件，统计文件中出现的所有词的频次，对中文，假设中文已经分词并进行了词性标注。
* 分析  
  任务并不复杂，有多种方法可以解决，比如可以得到文件内文本的词表，然后遍历这个词表，得到每个词在文件文本中的频次。  
  单个文本一般不会太大，但也经常有几G的单个文本文件，因此效率问题仍然很重要。

9.1 抽取指定行文本作为实验语料

请读者下载文本文件：http://pan.baidu.com/s/1c1RukqW 密码: 3c32，并将文件解压，保存到目录:d:\temp\下，文件名为：语料.txt。解压时间较长，请耐心等待（好在只解压一次）。   (下载慢的话，可以临时下载：[http://yunpan.blcu.edu.cn:80/link/FC61CB2B791A1999439CEC52C1A30CE2，作为小的试验文件，只有100多k)。](http://yunpan.blcu.edu.cn/link/FC61CB2B791A1999439CEC52C1A30CE2%EF%BC%8C%E4%BD%9C%E4%B8%BA%E5%B0%8F%E7%9A%84%E8%AF%95%E9%AA%8C%E6%96%87%E4%BB%B6%EF%BC%8C%E5%8F%AA%E6%9C%89100%E5%A4%9Ak)%E3%80%82) 可用文本编辑器打开语料.txt来查看文件内数据格式，但是打开这个10G文本文件比较慢。我们可以进入到powershell下，键入：Get-Content d:\temp\语料.txt -totalcount 10，来查看文件的前10行。  
对单个文本文件，如果比较大(如10G bytes)，则可以考虑先取这个文件的前n(如n=5000)行另存为一个小文件，对这个小文件来进行统计，如果没有问题了，再对这个较大的文件进行处理。  
当然，在大多数时候，我们面临的统计任务可能会是很多文件，可能会是较大规模的语料(几十G或者更多)，则也最好不要直接对其进行编程操作，而是拷贝几个相同格式的语料到一个临时目录，对这个目录进行统计实验，这样不但执行起来快捷，而且如果程序有错误，也容易查找。

#coding: utf-8

#示例程序9-1

with open(r'd:**\t**emp**\t**est.txt', 'w') as fw, open(r'd:**\t**emp**\语**料.txt') as fr:

   line\_number = 0

for line in fr:

if line\_number < 5000:

fw.write(line)

line\_number += 1

else:

break

示例程序9-1中：

* 语句for line in fr:是从fr文件对象中读取一行，fr是用open()函数打开的文件对象，可用for语句来依次访问其每一行。  
  这样，抽取语料.txt的前5000行，保存到test.txt文件中，作为测试文件，我们先对这个小测试文件(约1M bytes)即可。

9.2 词频统计

首先取测试语料中的一行文本：非京/ns 籍/ng 购房/n 须/d 连缴/v 5/m 年/m 以上/f 社保/j 对/p 住房/n 公积金/n 缴存/v 年限/n 并/c 无/v 要求/v，整个文件可以视为一个大字符串，每行都类似这行的形式，分成行很多是因为有换行符\n的存在，我们要将其中每个词汇去掉词性标记(/ns等)，然后放到词表中，注意词表中的词不重复，于是可得到整个文件的词表。

#coding: utf-8

#示例程序9-2

def get\_word\_table(filename):

'''

给定文本文件，统计其词表

'''

word\_table = []

with open(filename) as f:

text = f.read()

words = [word.split('/')[0] for word in text.split()]

for word in words:

if word not in word\_table:

word\_table.append(word)

return word\_table

#测试用

filename = r'd:**\t**emp**\t**est.txt'

table = get\_word\_table(filename)

print(table[:100])

示例程序9-2中：

* get\_word\_table(filename)函数与第8节中的字表统计函数的方法类似，参数filename是要统计词频的文件。  
  得到test.txt文件的词表以后，我们统计词频。

#coding: utf-8

#示例程序9-3

def count\_words\_freq(filename, words):

word\_freq\_pairs = []

with open(filename) as f:

text = f.read()

for word in words:

number = text.count(word)

word\_freq\_pairs.append([word, number])

return word\_freq\_pairs

words\_freq = count\_words\_freq(filename, table)

#测试用print

print(words\_freq[:100])

示例程序9-3中：

* count\_words\_freq(filename, words)函数有两个参数，分别是要统计的文本文件filename及词表words。
* count()是统计字符串频次的方法，基本用法为：字符串.count(str)，参数也是字符串，功能是在字符串对象中统计参数字符串出现的频次，返回一个整型数。  
  我们应该会觉得，目前得到词表和统计词频需要打开两次同一个文件，并遍历两次，这个似乎可以改进，我们可以只打开一次文件，并仅遍历一次就同时得到词表与词频。

（图）

#coding: utf-8

#示例程序9-4

def count\_words\_freq(filename):

'''

给定文本文件，建立词表并同时统计词频

'''

word\_freq\_pairs = []

with open(filename) as f:

text = f.read()

words = [word.split('/')[0] for word in text.split()]

for word in words:

for item in word\_freq\_pairs:

if word == item[0]:

item[1] += 1

break

else:

word\_freq\_pairs.append([word, 1])

return word\_freq\_pairs

#测试用

filename = r'd:**\t**emp**\t**est.txt'

table = count\_words\_freq(filename)

print(table[:100])

(画图说明)

示例程序9-4中：

* word\_freq\_pairs是二维列表，初始为空。其中每一个元素也是列表，存放[词，词频]。
* 通过一个列表解析，得到words，存放filename文件中所有去掉词性标记后的词汇。
* 在for word in words:循环中,对每一个在words中的词汇，如果与word\_freq\_pairs中某一个item相同，则对应的item的频次加一，如果不在，则在word\_freq\_pairs中增加一个item，其初始频次为1。其中，for...else的用法与while...else的用法类似。
* 测试中，打印前100个词-词频进行显示。

好，程序既然相同格式的小文件test.txt已经取得成功，我们就在真正的目标文件上试一下吧！请读者自行将示例程序9-4中的filename赋值为r'd:\temp\语料.txt'，然后运行程序。

不出意外的话，会出现类似MemoryError这样的错误。原因何在呢？一般来说，当前(2017年4月)，个人电脑的内存标准配置为8G bytes，而我们要统计的文本文件语料.txt大小为10G，利用read()方法读入的时候，必然会导致内存不足。   幸运的是，python提供了优雅的解决办法，特别是对文本文件，这就是我们已经使用过的 for xxx in file:，可按行读入，避免一次性读入导致的内存溢出。

#coding: utf-8

#示例程序9-5

def count\_words\_freq(filename):

'''

给定文本文件，建立词表并同时统计词频

'''

word\_freq\_pairs = []

with open(filename) as f:

for line in f:

words = [word.split('/')[0] for word in line.split()]

for word in words:

for item in word\_freq\_pairs:

if word == item[0]:

item[1] += 1

break

else:

word\_freq\_pairs.append([word, 1])

return word\_freq\_pairs

#测试用

filename = r'd:**\t**emp**\t**est.txt'

table = count\_words\_freq(filename)

print(table[:100])

示例程序9-5中：

* 我们仍然用test.txt作为测试实验。
* for line in f:中，将每次从f中，也就是filename中读入一行赋值给line。
* 其余部分与示例程序9-4基本相同。

好，拿目标文本文件来跑一下。请读者自行将示例程序9-5中的filename赋值为r'd:\temp\语料.txt'，然后运行示例程序9-5。  
不用怀疑是否运行了这个程序，其实，考验每个人耐性的时候到了。  
当我们愉快的看会儿新闻/视频/小说等再切回来的时候，发现还在持续计算中。好吧，大家不用再等待了，因为可能等到明天还是一样。  
到底出了什么问题呢？在正式统计以前，我们已经抽取了同格式的测试语料，并顺利统计了词频。就整个程序过程来看，也不应该是死循环。原因何在？  
我们将在示例程序9-5中加入一些语句，中间每运行一段就输出运行时间以及行号。

#coding: utf-8

#示例程序9-6

import time

def count\_words\_freq(filename):

'''

给定文本文件，建立词表并同时统计词频

'''

word\_freq\_pairs = []

linenum = 0 #当前处理行数

total\_line\_number = 0

with open(filename) as f:

for line in f:

total\_line\_number += 1

print('总行数为：', total\_line\_number)

f.seek(0) #重新回到文件起始位置(即0位置)

start\_time = time.time()

for line in f:

words = [word.split('/')[0] for word in line.split()]

for word in words:

for item in word\_freq\_pairs:

if word == item[0]:

item[1] += 1

break

else:

word\_freq\_pairs.append([word, 1])

if linenum % 1000 == 0:

end\_time = time.time()

print('...当前已经处理到第{}行...已经处理了{:.2f}秒...'.format(linenum, end\_time - start\_time))

linenum += 1

return word\_freq\_pairs

#测试用

filename = r'd:**\t**emp**\语**料.txt'

table = count\_words\_freq(filename)

print(table[:100])

示例程序9-6中：

* time模块是python标准库中的一个专门提供时间相关函数的模块。
* time()函数能够把当前时间转化为秒，返回浮点数。
* 首先遍历整个文件，得到文件总行数total\_line\_number。
* f.seek(int)方法能够回到文件对象f的指定字节位置，如果参数是0，则回到文件的起始位置。这里回到文件起始位置的原因是，在前面遍历整个文件得到文件总行数以后，f已经指向了文件的末尾，后面再进行读取时，如果不关闭后重新打开，就需要利用seek()调整f重新指向文件起始位置。

将任意两个语句运行时间点的时间相减，就会得到两个语句间运行了多长时间，这里以每统计1000行作为一个时间点。  
根据统计前10000行的输出结果，从9000行到10000行之间的这1000行需要的统计时间约为25秒（intel i5-5200U，主频2.2G，内存8G，win10下），我们以这个时间为标准来估算统计完整个语料.txt可能需要多少时间。文件总行数为43938000行，则总时间为：43938000/1000\*25/3600 = 305.125小时。 也就是说，理想情况下(实际上时间会更长，因为随着统计文本的增加，词表的长度会更长)，快到2周才可以得到统计结果。  
2周太久，只争朝夕，我们希望能尽量减少这个统计时间，因此让我们一起分析下建立词表的方法，看看是否可以改进、加速。示例程序9-6中的这段：

# 示例程序9-7

for line in f:

words = [word.split('/')[0] for word in line.split()]

for word in words:

for item in word\_freq\_pairs:

if word == item[0]:

item[1] += 1

break

else:

word\_freq\_pairs.append([word, 1])

示例程序9-7中：

* line是文件中所有的行，因此最终word会遍历到文本内所有的词，总次数随统计文本大小的不同而变化，我们将其设为m。
* word\_freq\_pairs是词-词频对列表，初始为空，最终将为文本内所有不同的词-词频，我们将最终其大小设为n。
* 程序多重循环的核心语句就是将要进行word == item[0]的判断字符串是否相同(字符串比较运算的耗时远大于整数加一运算的耗时，因此这里只关注本段程序中最耗时的语句)。

我们同时考虑原始示例程序9-3中，得到词表后，利用词表进行词频统计的函数：

# 示例程序9-8

def count\_words\_freq(filename, words):

word\_freq\_pairs = []

with open(filename) as f:

text = f.read()

for word in words:

number = text.count(word)

word\_freq\_pairs.append([word, number])

return word\_freq\_pairs

示例程序9-8中：

* words是词表总大小，应与示例程序9-7中的词-词频对列表大小相同，设为n。
* text是文本大小，假设其长度为k(如果m为文本中的单词书，则k正比于m)
* 整个程序是要对词表中所有n个词，来进行text.count(word)，也就是放到长度为k的字符串中，寻找每一个word出现的次数。这个比较，平均每次比较的次数，一般将大于m次(具体可参考字符串匹配的BF算法及KMP算法)。
* 因此，最后整个程序，将需要进行判断字符串是否相同m\*n次(近似次数)

对示例程序9-7与9-8，程序/算法的核心都是对每一个word，我们都要遍历一个table，比较其中的词与当前的word是否相同，相同就是找到，否则就是没有找到。这种查找方法称为**顺序查找**，或**顺序查找算法**。

（图）

对算法4-7，假设table中词汇是均匀分布的，则对每一个查询词，平均只要遍历到table的一半。这样对一个词汇进行查找所需的比较次数就约为：n/2。 可见，算法执行实际比较(运算)次数与表的大小n直接相关，是n的函数，称为算法的**时间复杂度**。同时，称n为**问题规模**。   对问题规模为n的词表，对长度为m个单词的文本，统计词频需要重复n/2\*m的比较。虽然算法4-8需要重复约m\*n次比较，但两者在时间复杂度上，是同一个量级的，同阶的，可用O(m\*n)表示这个量级或阶。有关算法及时间复杂度，请参考《数据结构》或《算法》等相关课程。

（时间复杂度图）

我们必须设法降低统计词频算法的时间复杂度。  
考虑查找算法，该算法找一个单词必须遍历单词列表，但在猜数游戏中，我们找到一个数，并没有从头遍历，而是采用了每次折半的方式，很快就找到了目标数字，这就是二分查找或折半查找算法。  
如果我们利用二分查找，那岂不是将O(m\*n)的时间复杂度，降低到m\*log2(n)(请读者自行大致估算，忽略常数项系数)了？仅看这个式子，大家也许没有感觉。假设n=200000(一般词表大小)，则log2(n)=17.6096...，这是什么概念？这意味着，假设我们利用二分查找，成功的进行词频统计，进行粗略估算，则该算法平均查找次数为200000/17.6=11363.6...，比200000/2=100000性能提高了5000多倍！  
在实现二分查找之前，我们需要进行一些知识铺垫。

9.3 递归

n的阶乘可以定义为：n！= 1X2X3...n，我们据此写出了求n!的函数。  
与此同时，n的阶乘也可以这样定义：n!=n\*(n-1)!且0！=1。根据这样的定义，可以写出求n!函数如下：

# 示例程序9-9

def recusive\_factorial(n):

if n == 0:

return 1

else:

return n\*recusive\_factorial(n-1)

在示例程序9-9中：

* 程序两个分支恰是n的阶乘定义(第二个定义)
* 函数recusive\_factorial()自己调用了自己。
* 函数自己直接/间接调用自己的方法/技巧，称为**递归**。这种函数也称为递归函数。 我们可以模仿求n阶乘的递归方法，递归求1+2+3+...n。首先写出便于递归的公式：recusive\_sum(n) = n + recusive\_sum(n-1)，recusive\_sum(0) = 0。

# 示例程序9-8

def recusive\_sum(n):

if n == 0:

return 0

else:

return n + recusive\_sum(n-1)

对这类可以写出递归形式公式的，均可以直接用递归函数求解。再比如斐波那契数列：1,1,2,3,5,8,13,21....，求其第n项。  
首先将其写成递归形式的公式：

fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)  
fibonacci(0) = 1  
fibonacci(1) = 1  
然后写出递归函数：

# 示例程序9-9

def fibonacci(n):

if n <= 2:

return 1

else:

return fibonacci(n-1) + fibonacci(n-2)

递归程序也可以转化成非递归程序(循环)，利用循环求和及求阶乘的代码，之前的几节中已经讲述，这里就利用循环来求斐波那契数列第n项。  
由于递归程序在最后需要return前两个函数值，因此需要两个变量来保存这两个函数值。

# 示例程序9-10

def fibonacci\_iter(n):

a, b = 1, 1

for i in range(n-2):

a, b = b, a+b

return b

# 测试

print(fibonacci\_iter(10))

示例程序9-10中：

* 利用a与b来保存数列前两个值，数列初始两个值为1和1。  
  如果是需要保存前三个值的数列，则可以写成如下：

# 示例程序9-11

def fibonacci3\_iter(n):

a, b, c = 1, 1, 1

for i in range(n-3):

a, b, c = b, c, a+b+c

return c

# 测试

print(fibonacci3\_iter(10))

示例程序9-11中：

* 我们实现了**进阶版**的斐波那契数列：1, 1, 1, 3, 5, 9, 17, 31, 57.... 也就是：  
  fibonacci3(n) = fibonacci3(n-1) + fibonacci3(n-2) + fibonacci3(n-3)  
  fibonacci(0), fibonacci(1), fibonacci(2) = 1, 1, 1

还有一类问题，解决该问题的过程可以用递归来描述的。  
最经典的如汉诺塔问题：有三根相邻的柱子，标号为A,B,C，A柱子上从下到上按金字塔状叠放着n个不同大小的圆盘，要把所有盘子一个一个移动到柱子C上，每次只能挪动一个圆盘，并且每次移动同一根柱子上都不能出现大盘子在小盘子上方，请问如何移动？  
该问题的完整描述可自行网上搜索。

（图）

下面我们从宏观上来分析一下这个问题的实际操作过程：  
首先**假设存在一种能够完成这个任务的移动办法，称之为hanoi()**，该方法能够对任意n>=1的圆盘进行符合要求的移动，过程是：

* 当n=1时，从A柱挪动圆盘到C柱只要直接移动即可，可表示为:print(A,'--->',C)
* 对n>1，即要挪动多个圆盘时，A柱圆盘要挪动到C柱，需要通过辅助柱B来完成。为使A中n个圆盘都挪到C柱，需要：
  + 将A中n-1个圆盘 **采用办法：hanoi()** 挪到B柱，辅助柱为C；
  + 将A柱中最后剩下的一个圆盘直接挪到C柱；
  + 将B柱的n-1个圆盘 **采用办法：hanoi()** 挪到C柱，辅助柱为A；

根据以上递归框架，我们来实现下hanoi方法/函数。

def hanoi(n,a,b,c):

if n==1: # 1个环的情况，也是递归的出口

print(a,'->',c)

else: # n > 1时，用抽象出的3步来移动

hanoi(n-1,a,c,b) # 将n-1个环从a移动到b上

print(a,'->',c) # 将底盘从a移动到c上

hanoi(n-1,b,a,c) # 将b上的n-1个环移动到c上

9.4 二分查找

我们回到二分查找，先从宏观上对其进行分析，将整个过程用递归框架来描述：

* 假定存在一个有效的二分查找的方法，名字叫bi\_search()，能够在正序numbers序列的索引区间[low, high]之间查找number的索引。
* 如果low>high，说明没有找到，返回-1；
* 否则，取中点位置mid = (low+high)//2；
  + 如果number == numbers[mid]，则返回mid，此即为所得索引；
  + 否则如果 number > numbers[mid]，就 **采用方法：bi\_search()** 在[mid+1, high]之间查找number；
  + 否则，就 **采用方法：bi\_search()** 在[low, mid-1]之间查找number。

(图)

# coding: utf-8

# 示例程序9-13

def bi\_search(number, numbers, low, high):

if low > high:

return -1

else:

mid = (low+high)//2

if number == numbers[mid]:

return mid

elif number > numbers[mid]:

return bi\_search(number, numbers, mid+1, high)

else:

return bi\_search(number, numbers, low, mid-1)

# 测试用

nums = [x\*x for x in range(10)]

print(nums)

print(bi\_search(26, nums, 0, 9))

print(bi\_search(9, nums, 0, 9))

在此基础上，能够相对容易地实现二分查找的非递归算法，该算法实现可参考《数据结构》及《算法》等书籍，这里暂不介绍。  
因为二分查找应用十分广泛，python早已经为我们事先实现。

# coding: utf-8

# 示例程序9-14

import bisect

a = [i for i in range(20)]

print(a)

print(bisect.bisect\_left(a, 5))

print(a)

print(bisect.insort\_left(a, 5.5))

print(a)

示例程序9-14中：

* bisect是标准库中的二分查找模块，内置有多个方法。
* bisect.bisect\_left(..., x)方法接受两个参数，第一个是正序序列，第二个是任意可与序列中元素进行大小比较的对象x。该方法会查找x在序列中的插入位置，这意味着，在序列中插入x以后，序列仍然保持有序。如果x大于序列中最大的对象，则返回序列长度，如果x小于序列中最小的对象，则返回0。在序列中如果有对象与对象x大小相同，返回该对象在序列中的索引。
* bisect.insort\_left(..., x)方法接受两个参数，第一个是正序序列，第二个是任意可与序列中元素进行大小比较的对象x。该方法会查找x在序列中的插入位置，并将其插入，并使序列仍然保持有序。如果x大于序列中最大的对象，插入在序列尾部，如果x小于序列中最小的对象，则插入在序列首部。在序列中如果有对象与对象x大小相同，则插入在该位置对象的左侧。
* bisect模块详细介绍参见：<https://docs.python.org/3/library/bisect.html?highlight=bisect#module-bisect>

利用bisect模块，对词表进行查找，可以使上述统计词频的性能大为提高。  
二分查找的前提是，初始序列有序，下面让我们考虑下排序算法。

9.5 排序

由于前几个任务中，我们多次进行了求最大值及最小值的实现，这里就介绍基础排序算法中的简单选择排序。  
给定一个序列，假设序列内的元素可比较大小，欲对其进行简单选择排序(正序)。简单选择排序算法的伪码如下：

从头(索引i=0)开始遍历序列seq：

令最小值的索引为i，即：min=i

在区间[i+1, len(seq))，令索引为j，遍历seq：

if seq[j] < seq[min]:

min = j

交换i与min位置的元素，使seq[i]为[i, len(seq))区间的最小值

(图)

根据上述伪码，可实现如下：

# coding: utf-8

# 示例程序9-15

# 以seq为数字序列为例

def sort\_simple\_selection(seq):

for i in range(len(seq)-1):

min = i

for j in range(i+1, len(seq)):

if seq[j] < seq[min]:

min = j

seq[i], seq[min] = seq[min], seq[i]

numbers = [23,45,12,1,2,333,5,1,222,34,-9]

sort\_simple\_selection(numbers)

print(numbers)

示例程序9-15中：

* sort\_simple\_selection(...)函数的参数是一个列表，在主程序调用该函数时，实参为numbers，指向[23,45,12,1,2,333,5,1,222,34,-9]，形式参数seq也指向这个列表。
* 在sort\_simple\_selection(...)函数内部将seq指向的这个列表排序后，程序返回，此时，变量number仍然指向这个列表，而这个列表已经排好了序。
* 因此，自定义函数中利用可变参数作为参数传递时，函数内部对可变参数的更改，能够反映到该函数外部，如果疏忽，则会得到预期之外的结果。当然，福利就是，提供了无需返回这个可变参数的便利。

（画图）

经过简单分析可得，简单选择排序算法的时间复杂度是O(n\*n)，对问题规模n=20万，即对词表为20万的序列进行排序，则需要比较字符串大小20万x20万=400亿次。   我们可以利用与二分查找的思路来设计性能更高的排序算法，算法思路是，在待排序序列中，每次选定一个元素p，将其余元素与p依次进行比较，比这个元素小的元素均放到p的左侧，其余元素放到p的右侧。然后，对p的左侧、右侧，再进行同样的算法操作。   这，就是快速排序！

（图）

容易想到，快速排序可以利用递归框架来实现，为了便于理解，算法中利用了2个辅助序列(实际算法实现可无额外序列，具体实现就下学期来上我的数据结构课程上听吧)，伪码如下：

假设有快速排序算法quick\_sort(seq)，可以实现快速排序。

令left\_seq = [], right\_seq = []

令待排序序列区间的第一个元素为p，即p=seq[0]

对seq的[start+1,end]区间中的每一个元素：

如果元素 < p:

将该元素加入到left\_seq中

否则：

将该元素加入到right\_seq中

如left\_seq非空，利用快速排序算法quick\_sort，对left\_seq进行快速排序

如right\_seq非空，利用快速排序算法quick\_sort，对right\_seq进行快速排序

返回：left\_seq + p + right\_seq

根据上述伪码，可实现如下：

# coding: utf-8

# 示例程序9-16

# 以seq为数字序列为例

def quick\_sort(seq):

left\_seq = []

right\_seq = []

p=seq[0]

for number in seq[1:]:

if number <= p:

left\_seq.append(number)

else:

right\_seq.append(number)

if left\_seq:

left\_seq = quick\_sort(left\_seq)

if right\_seq:

right\_seq = quick\_sort(right\_seq)

return left\_seq + [p] + right\_seq

#测试

numbers = [23,45,12,1,2,333,5,1,222,34,-9,-9,-9]

sorted\_numbers = quick\_sort(numbers)

print(numbers)

print(sorted\_numbers)

示例程序9-15中：

* 由于quick\_sort(...)函数内部并没有对参数seq进行排序，因此最终要返回排好序的序列，而原有的seq也即主程序中的numbers保持不变。
* 快速排序的非递归算法，读者可自行查阅资料实现。

可以证明(详见各类算法及数据结构教程)，快速排序算法的时间复杂度为O(nlog2(n))，以n=20万为例进行粗略估算，则性能比简单选择排序算法提升200000/log2(200000)倍，也即10000倍以上！   这么强大的且常用的算法，python当然已经为我们实现了，直接内置(内置的是基于快速排序算法的改进版本)！

# coding: utf-8

# 示例程序9-17

numbers = [23,45,12,1,2,333,5,1,222,34,-9,-9,-9]

nums = sorted(numbers)

print('nums = ', nums)

print('numbers in sorted(numbers) = ', numbers)

numbers.sort()

print('numbers after numbers.sort() = ', numbers)

示例程序9-17中：

* sort()是python的list对象的快速排序方法，用法是：列表.sort()，可将列表中的元素进行排序(默认为正序)。注意，列表自身进行了排序。
* sorted(...)是python内置的快速排序方法，用法是：sorted(seq)，可返回对序列seq排序后新生成的一个列表。注意，原有序列不变。

我们回到任务本身，实际上，考虑到词表初始为空列表，空列表自然有序，同时，bisect.bisect\_left()方法能够直接找到插入位置，因此每次插入新对象后，词表仍然可以保持有序。这也是一种排序算法，即**二分插入排序**，比较符合当前任务需求，缺点是，每次插入元素会导致O(n)的元素移动(具体原因请同学们下学期好好上我的数据结构课程^-^)，将在很大程度上影响程序性能，但预期仍然会比原来顺序查找的词频统计方法要快很多。   我们利用bisect及上述思路重新设计词频统计方法：

#coding: utf-8

#示例程序9-18

import time, bisect

def count\_words\_freq\_bisect(filename):

'''

给定文本文件，建立词表并同时统计词频

'''

word\_freq\_pairs = []

linenum = 0 #当前处理行数

total\_line\_number = 0

with open(filename) as f:

for line in f:

total\_line\_number += 1

print('总行数为：', total\_line\_number)

f.seek(0) #重新回到文件起始位置(即0位置)

start\_time = time.time()

for line in f:

words = [word.split('/')[0] for word in line.split()]

for word in words:

i = bisect.bisect\_left(word\_freq\_pairs, [word])

if i != len(word\_freq\_pairs) and word\_freq\_pairs[i][0] == word:

word\_freq\_pairs[i][1] += 1

else:

word\_freq\_pairs.insert(i, [word,1])

if linenum % 1000 == 0:

end\_time = time.time()

print('...当前已经处理到第{}行...已经处理了{:.2f}秒...'.format(linenum, end\_time - start\_time))

linenum += 1

return word\_freq\_pairs

#测试用

filename = r'd:**\t**emp**\语**料.txt'

table = count\_words\_freq\_bisect(filename)

print(table[:100])

示例程序9-18中：

* i = bisect.bisect\_left(word\_freq\_pairs, [word])是找到文本中的词word在词表word\_freq\_pairs中的插入位置，用[word]的原因是比较对象的类型需要一致。
* i != len(word\_freq\_pairs) and word\_freq\_pairs[i][0] == word，当其为True则说明词表中有word词汇，且为i。
* word\_freq\_pairs[i][1] += 1，找到词汇位置i后，词频加一
* word\_freq\_pairs.insert(i, [word,1])，没有相同的词汇，则在i位置插入词汇以及频次1。

运行上述代码，不难发现，统计词频程序的性能得到了很大提升，几周的运行时间，缩短到天。  
计算机不分昼夜不知疲劳的运算，人们仍不满足。人们不是心疼计算机的付出，而是需要更快，并非一味的贪心，而是尽量追求完美。

9.6 散列、set及dict

从bisect.bisect\_left()函数以及示例程序9-18，我们隐约有所启发，对一个对象为其找到合适的插入位置，同时也能知道是否有相同的对象。如果每一个对象都自带一个位置信息就好了，就像序列中的每一个元素，都有一个索引位置，我们可以直接利用索引位置来找到元素。  
也就是说，如果每一个对象，都能得到独立的不同地址，将这些对象根据地址存储在一个容器中。对任意欲查询的对象，得到地址，就可以直接在这个容器中查询了。  
更进一步，对待查找元素集合X，我们需要一个映射f(X)---->Y（下标），使X与Y一一对应，且X在Y上能较为均匀的分布，Y的大小能够控制在与X集合的大小成一定比例。

（图）

若结构中存在和关键字X相等的记录，则必定在f(X)的存储位置上。由此，不需比较便可直接取得所查记录。称这个对应关系f为散列函数(Hash function)，按这个思想建立的表为散列表或哈希表-Hash\_table。这样，查找任意一个元素，并不需要排序，同时，其时间复杂度是O(1)与n无关！！！。以我们任务中的词表n为例，将比二分查找的性能又提升了log2(200000)=16.7倍，是不是很神奇？！！  
具体哈希的思想在《数据结构》及《算法》教程中有详细讲解，或者访问：<https://en.wikipedia.org/wiki/Hash_table>

根据散列/哈希思想，python中内置实现了集合set与词典dict。首先来看set：

#coding: utf-8

#示例程序9-19

words = set() #建立一个空的set

print('空set：',words)

basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'} #集合赋值

print(basket) # show that duplicates have been removed

letters = set('aadfijskdfsakdfjlasd') #字符串直接给集合赋值

print(letters)

print('分割-'\*50)

print('apple' in basket) #判断对象是否在集合中

print('apple' not in basket)

print('分割-'\*50)

letters = {letter.upper() for letter in 'abcde'} #集合解析/推导

print(letters)

letters.add('h') #向集合中加入元素

print('after add h:'',letters, len(letters)) #len()求集合元素个数

letters.remove('h') #从集合中移除指定元素

print('after remove h:'letters)

letters.clear() #清空集合

print('after clear():', letters)

示例程序9-19中列出了集合初始化以及各种常用函数与用法。读者需要了解：

* 集合是可变数据类型。
* 集合内不存在相同的对象，如在建立集合时有相同对象，则只保留一个。
* 集合是哈希表，判断对象是否在集合中，比判断对象是否在列表/元组中要快很多，因此这类任务，尽量使用集合而不是列表或元组。
* 集合内的元素，没有固定的顺序，没有下标，因此你不能用类似list一样的数字索引来取得键值对。

有了set这个利器，得到词表就异常容易：

#coding: utf-8

#示例程序9-20

filename = r'd:**\t**emp**\语**料.txt'

word\_table = set()

with open(filename) as f:

for line in f:

word\_table |= {word.split('/')[0] for word in line.split()}

for i, word in enumerate(word\_table):

print(word)

if i==20: #词表较长，本例仅取20个打印出来作为示例

break

还是需要一定的耐心，不是所有的事情都能够一蹴而就，立等可取。10G的语料，用个人电脑来统计，是需要一些时间。  
示例程序9-20中：

* word\_table |= {word.split('/')[0] for word in line.split()}也就是word\_table = word\_table | {word.split('/')[0] for word in line.split()}，其中|可以将两侧的集合合并。
* enumerate(...)函数可将参数转换为enumberate类型，该类型可用for...in进行迭代，每次返回返回一个整数，以及其参数中的一个元素。参数可以是集合也可以是列表或元组。对集合，则所取得的整数仅用来计数。如果参数是列表或者元组，则每次取得的整数恰是取得元素的索引。

虽然有了set，我们能较快的得到词表，但是对当前的词频统计任务，却难以进一步提升性能，因为其实我们理想中是期望能有可以容纳词:词频的一个容器，且每个词可以像集合中的对象一样，可以一步定位找到。  
好吧，这个套路大家想必早已经习惯：我们所期望的，python必已经为我们提供并内置了。这就是词典dict。

#coding: utf-8

#示例程序9-21

words = {} #建立一个空的dict，与words = dict()等价

print('空dict：',words)

basket = dict(apple=1, orange=2, pear=3, banana=10) #词典赋值，键不能有重复

print(basket)

tel = {'jack': 4098, 'sape': 4139}

print(tel)

print(tel['jack']) #词典某个具体键的值

print('分割-'\*10)

print('apple' in basket) #判断某个键是否在词典中

print('apple' not in basket)

print('分割-'\*10)

tel = dict([('sape', 4139), ('guido', 4127), ('jack', 4098)]) #利用列表生成词典

print(tel)

letters = {letter:10 for letter in 'abcde'} #词典解析/推导

print(letters, len(letters)) #len()求词典内键值对的个数

tel['pengyuan'] = 4078 #词典中加入键:值对

print('after insert a key and value into tel:', tel)

del tel['jack'] #删除词典的键。注意值也随之删除

print('after del a key from tel:', tel)

letters.clear() #清空词典

print('after clear():', letters)

tel['test'] #对没有的键试图得到其value，则会给出`KeyError`异常

示例程序9-21中：

* dict是词典类型，内部存储0，1或多个键值对，也称为key value pairs，表示为key:value或键:值，dict采用哈希的方式利用键进行索引，因此查询键速度很快，与集合相当。
* dict是可变数据类型。
* dict中的键值对也是没有固定顺序的，没有下标，因此你不能用类似list一样的数字索引来取得键值对。
* dict中的键不能重复，值没有要求。
* dict中，键key必须是不可变数据类型(因此list不能作为键)，值value没有这个要求。

再看一下dict的遍历：

#coding: utf-8

#示例程序9-22

letters = {letter:10 for letter in '人生苦短我用python'}

for key in letters: #对dict直接遍历，就是对key进行遍历

print(key)

for key in letters.keys(): #得到dict\_keys类型对象，可称为键视图

print(key)

for value in letters.values(): #得到dict\_values类型对象，可称为值视图，可对其遍历得到各个value。

print(value)

for key, value in letters.items(): #得到dict\_items类型对象，可称为键值对视图，可对其进行遍历同时得到键与值

print(key, value)

将dict视图转换为list类型：

#coding: utf-8

#示例程序9-23

letters = {letter:10 for letter in '人生苦短我用python'}

values = list(letters.values())

print(values)

items = list(letters.items())

print(items)

print(list(letters.keys()))

有了以上知识，我们编写利用dict进行词频统计的程序。

#coding: utf-8

#示例程序9-24

def count\_words\_freq\_dict(filename):

words\_freq\_dict = {}

with open(filename) as f:

for line in f:

words = [word.split('/')[0] for word in line.split()]

for word in words:

if word in words\_freq\_dict: #如果词典中有键为word，则该word的value+1

words\_freq\_dict[word] += 1

else: #否则，词典中加入键word，其value为1

words\_freq\_dict[word] = 1

return words\_freq\_dict

#测试用

filename = r'd:**\t**emp**\语**料.txt'

table = count\_words\_freq\_dict(filename)

for i, item in enumerate(table):

print(item, table[item])

if i==20:

break

以上代码的性能已经足够，但是我们很多时候其实希望得到一个按照value逆序或者正序排列的key:value(也即词-词频)结果。  
如何利用sort()/sorted()对value进行排序，请结合本任务后续第9.8小节的相关知识点自行完成。  
此外，count\_words\_freq\_dict()函数部分还可以精简。

9.7 collections

collections是python标准库中的容器类型(Container datatypes)，可以视为内置的dict,set,list,tuple等的特殊形式。 下面首先用collections中的defaultdict类型将count\_words\_freq\_dict()函数简化：

#coding: utf-8

#示例程序9-25

from collections import defaultdict

def count\_words\_freq\_dict(filename):

words\_freq\_dict = defaultdict(int)

with open(filename) as f:

for line in f:

words = [word.split('/')[0] for word in line.split()]

for word in words:

words\_freq\_dict[word] += 1

return words\_freq\_dict

示例程序9-25中：

* 首先从collections模块中引入了defaultdict类型
* defaultdict(int)是建立了一个defaultdict类型的对象，该对象类似dict，但对没有在defaultdict中的键key，设缺省值为整型数0。
* words\_freq\_dict[word] += 1，如遇到原先不在words\_freq\_dict中的word，由于缺省值为0，再自增加1后，为1次。

我们还可以继续精简代码：

#coding: utf-8

#示例程序9-26

from collections import Counter

def count\_words\_freq\_dict(filename):

words\_freq\_dict = Counter()

with open(filename) as f:

for line in f:

words\_freq\_dict.update([word.split('/')[0] for word in line.split()])

return words\_freq\_dict

示例程序9-26中：

* 首先从collections模块中引入了Counter类型
* words\_freq\_dict = Counter()是建立了一个空的Counter对象，Counter类型类似于dict，也可以视为一种特殊的dict。
* words\_freq\_dict.update([word.split('/')[0] for word in line.split()])，其中update()是Counter()类型变量的一个函数，会自动更新其参数(一般为一个序列)中的对象频次，并以键值对的形式存入Counter对象，且与dict类似，这个Counter对象的key是可哈希的，即可利用哈希函数存取，时间复杂度为O(1)。

好吧，我们将词频统计的代码精简到了6行，且效率很高，这就是python语言的威力。  
在本节任务中，我们利用词频统计任务，将程序的效率逐步提升，与此同时，所用数据类型逐步复杂，代码量逐步减少。  
之所以这样安排而不是直接给出最优答案确实是期望在这个过程中，读者能对各种相关数据类型和算法有更清醒的认识和理解，也能够进一步锻炼使用python进行编程的能力。  
通过1-9这几个任务，我们已经基本介绍并掌握了python语言的常用基本语法，并已对python编程有一定的实践能力。  
我们清楚的感觉到，python通过内置或者标准库，已经实现了很多非常有用的类型或者功能，从现在开始，我们将直接利用python、标准库以及各种外部库来进行编程。  
同时，恭喜读者，你已经掌握了python语言基础，且已经具有基本的编程能力。接下来读者可以自行探索与实践。我们也可以继续一同步入python的进阶与应用。

9.8 辅助任务

* 任务1：回文字符串的判定 在task 7中，我们根据一个字符串实现了一个回文塔。现在的任务是，给定一个序列，判断其是否是回文序列。这个任务比较简单，有多种解法，我们先练习下递归解法：

#coding: utf-8

#示例程序9-27

def palindrome(seq):

if len(seq) in [0, 1]:

return True

else:

if seq[0] != seq[-1]:

return False

return palindrome(seq[1:-1])

seq = 'abcdedcba'

print(palindrome(seq))

上述代码可以简化如下：

#coding: utf-8

#示例程序9-28

def palindrome(seq):

if len(seq) in [0, 1]:

return True

else:

return False if seq[0] != seq[-1] else palindrome(seq[1:-1])

seq = 'abcdedcba'

print(palindrome(seq))

示例程序9-28中：

* False if seq[0] != seq[-1] else palindrome(seq[1:-1])是**三元表达式**，一般形式为：值1 if 条件 else 值2。当条件为真，表达式的值为值1，否则表达式的值为值2。也可以直接用在赋值语句中如：x = 5 if len('python')<10 else 10。

利用三元表达式，上述代码还可以进一步简化：

#coding: utf-8

#示例程序9-29

def palindrome(seq):

return True if len(seq) in [0, 1] else False if seq[0] != seq[-1] else palindrome(seq[1:-1])

seq = 'abcdedcba'

print(palindrome(seq))

好吧，递归函数主体只剩下了一行代码，虽然代码精简有卖点，但是并不建议总这样做，有时候会影响程序可读性。 再考虑下非递归的解法：

#coding: utf-8’p

#示例程序9-30

def palindrome(seq):

for i in range(len(seq)//2):

if seq[i] != seq[len(seq)-i-1]:

return False

break

return True

seq = 'abcdedcba'

print(palindrome(seq))

如果善用切片，还可以这样：

#coding: utf-8

#示例程序9-31

def palindrome(seq):

return seq == seq[::-1]

seq = 'abcdedcba'

print(palindrome(seq))

* 任务2：字频的统计

完全可以利用与词表词频类似的方法来统计一个文本文件的字表字频，但是我们现在已经得到了词频统计的结果，如果可以在这个基础上直接得到字表字频，统计时间就会大大缩短。

#coding: utf-8

#示例程序9-32

from collections import defaultdict

#假设之前已经词频统计完成，并已放入Counter或dict对象word\_table中

ch\_table = defaultdict(int)

for word, freq in word\_table.items():

for ch in word:

ch\_table[ch] += freq

for ch, freq in ch\_table.items():

print(ch, freq, sep=':', end='|')

示例程序9-32中：

* 对词频统计结果中的每个词进行遍历，对组成每个词的字进行遍历，每个词中包含的字的频次为当前的对应词的词频。

9.9 拓展与汇总

* bisect模块 主要有六个方法：

bisect.bisect\_left(a, x, lo=0, hi=len(a))。

Locate the insertion point for x in a to maintain sorted order. The parameters lo and hi may be used to specify a subset of the list which should be considered; by default the entire list is used. If x is already present in a, the insertion point will be before (to the left of) any existing entries. The return value is suitable for use as the first parameter to list.insert() assuming that a is already sorted.  
The returned insertion point i partitions the array a into two halves so that all(val < x for val in a[lo:i]) for the left side and all(val >= x for val in a[i:hi]) for the right side.

bisect.bisect\_right(a, x, lo=0, hi=len(a))，bisect.bisect(a, x, lo=0, hi=len(a))。

Similar to bisect\_left(), but returns an insertion point which comes after (to the right of) any existing entries of x in a.  
The returned insertion point i partitions the array a into two halves so that all(val <= x for val in a[lo:i]) for the left side and all(val > x for val in a[i:hi]) for the right side.

bisect.insort\_left(a, x, lo=0, hi=len(a))。

Insert x in a in sorted order. This is equivalent to a.insert(bisect.bisect\_left(a, x, lo, hi), x) assuming that a is already sorted. Keep in mind that the O(log n) search is dominated by the slow O(n) insertion step.

bisect.insort\_right(a, x, lo=0, hi=len(a))，bisect.insort(a, x, lo=0, hi=len(a))。

Similar to insort\_left(), but inserting x in a after any existing entries of x.

详细用法见：<https://docs.python.org/3/library/bisect.html>

* 集合的基本运算

令A = {1,2,3,4,5}，B = {6,7,8,9,1}，D = {'1','2','3','4','5'}，D = {'6','7','8','9','1'}。则有：

| **函数或操作** | **示例** | **结果** | **说明** |
| --- | --- | --- | --- |
| union(\*others) | A.union(B, C) | {1,2,3,4,5,6,7,8,9,'1','2','3','4','5'} | 求并集 |
| set | other| ... | A|B|C | 同上 | 求并集 |
| intersection(\*others) | A.intersection(B, C, D) | {1,'1'} | 求交集 |
| set & other &... | A & B & C & D | 同上 | 求交集 |
| difference(\*others) | A.difference(B) | {2,3,4,5} | 求差集 |
| set - other -... | A - B | 同上 | 求差集 |
| symmetric\_difference(other) | A.symmetric\_difference(B) | {2,3,4,5,6,7,8,9} | A|B - A&B |
| set - other -... | A ^ B | 同上 | A|B - A&B |

更多操作以及不可变集合frozenset等，见：<https://docs.python.org/3/library/stdtypes.html?set#set-types-set-frozenset>

* sort()与sorted()

两者用法、相同与不同主要如下：

| **. . .** | **sort()** | **sorted** |
| --- | --- | --- |
| 用法 | sort(\*, key=None, reverse=None) | sorted(\*, key=None, reverse=None) |
| 相同1 | 内置，快速排序，针对列表 | 内置，快速排序，针对列表 |
| 不同1 | 就地排序，原列表改变 | 返回排序后的新列表，原列表不变 |
| 不同2 | 2个关键字参数(可选) | 1个序列参数(必选)，2个关键字参数(可选) |
| 相同2 | 关键字参数用法相同，任一关键字参数，须加上关键字才能使用 | 关键字参数用法相同，任一关键字参数，须加上关键字才能使用 |
| 不同3 | 仅针对列表 | 可用于任何可迭代对象 |

基本用法示例：

#coding: utf-8

#示例程序9-33

a = [5,6,3,2,1,90]

print(sorted(a))

print('after sorted, a is still old a:', a)

a.sort()

print('after sort, a is not old a:', a)

print(sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})) # 直接对dict进行sort，将仅单独作用于key

a.sort(reverse=True)

print('reverse a:', a)

sort()及sorted()中的key参数指定一个可返回排序关键字的函数，该函数作用于待排序对象中的每一个元素，sort()及sorted()根据排序关键字进行排序。

#coding: utf-8

#示例程序9-34

a = sorted("This is a test string from Andrew".split(), key=str.lower) #返回每个字符的小写形式，以其为关键字进行排序

print(a)

student\_tuples = [

('john', 'A', 15),

('jane', 'B', 12),

('dave', 'B', 10),

]

from operator import itemgetter

a = sorted(student\_tuples, key=itemgetter(2)) # 返回每个对象索引2的元素，以其为关键字对student\_tuples排序

print(a, student\_tuples)

student\_tuples.sort(key=itemgetter(2), reverse=True) # key及reverse参数一起使用

print(student\_tuples)

示例程序9-34中：

* str.lower()是一个函数/方法，可接受一个字符串参数，返回字符串的小写形式。如：str.lower('AB')，会返回ab。key=str.lower中，是将指定的函数/方法名称str.lower赋值给key，以供后续sorted()的调用。
* from operator import itemgetter，从operator模块中引入itemgetter函数，参数代表作用对象中对应索引的元素，后续sort()及sorted()将以这个元素作为关键字来进行排序。

有关key参数的使用涉及到了函数式编程，相关知识将在后续章节详述。

* Counter模块

键入以下代码，观察执行结果。

#coding: utf-8

#示例程序9-34

#Counter 初始化

c = Counter() # a new, empty counter

print(c)

c = Counter('gallahad') # a new counter from an iterable

print(c)

c = Counter({'red': 4, 'blue': 2}) # a new counter from a mapping

print(c)

c = Counter(cats=4, dogs=8) # a new counter from keyword args

print(c)

c = Counter(['eggs', 'ham'])

print(c)

c['bacon'] # count of a missing element is zero

print(c)

Counter 常用方法：

#coding: utf-8

#示例程序9-35

#Counter 方法

c = Counter(a=4, b=2, c=0, d=-2)

print(sorted(c.elements())) # Return an iterator over elements repeating each as many times as its count.

# Elements are returned in arbitrary order.

text = 'abracadabra'

n=3

Counter(text).most\_common(n) # Return a list of the n most common elements and their counts from the most common to the least.

# 相当于根据值value逆序前n个key-value pairs。

c = Counter(a=3, b=1)

d = Counter(a=1, b=2)

print(c + d) # add two counters together: c[x] + d[x]

print(c - d) # subtract (keeping only positive counts)

print(c & d) # intersection: min(c[x], d[x])

print(c | d) # union: max(c[x], d[x])

Counter详细介绍，见：<https://docs.python.org/3/library/collections.html?Counter#counter-objects>

* 数据类型汇总（以后再写）

9.10 完整代码

#coding: utf-8

#示例程序9-36

from collections import Counter

def count\_words\_freq\_dict(filename):

words\_freq\_dict = Counter()

with open(filename) as f:

for line in f:

words\_freq\_dict.update([word.split('/')[0] for word in line.split()])

return words\_freq\_dict

def main():

filename = r'd:**\t**emp**\语**料.txt'

table = count\_words\_freq\_dict(filename)

print(table.most\_common(50))

if \_\_name\_\_ == '\_\_main\_\_':

main()

#1、写函数，给定符号和行数，如’\*’，5，可打印相应行数的如下菱形。主程序输入符号和行数调用该函数进行验证。

def diamond(n,s):

for i in range(1,n+1):

print(' '\*(n-i) + s\*i)

for i in range(1,n):

print(' '\*i + s\*(n-i))

s = input('请输入你想要输入的符号')

s = s +' '

n =int( input('请输入边长'))

diamond(n,s)

# 打印空心菱形

def hollow\_diamond(n,s):

for i in range(1,n+1):

if (i == 1) or (i == 2):

print(' '\*(n-i) + s\*i)

else :

print(' '\*(n-i) +(s + ' '\*(2\*(i-2)) + s))

for i in range(1,n):

if (i == n-1) or (i == n-2):

print(' '\*i + s\*(n-i))

else :

print(' '\*i + (s + ' '\*(2\*(n-2-i)) + s))

s = input('请输入一个你喜欢的符号:')

s = s + ' '

n = int(input('请输入边长:'))

hollow\_diamond(n,s)

# 打印实心三角形

def triangle(n,s):

for i in range(1,n+1):

print(' '\*(n-i) + s\*i)

s = input('请输入一个你喜欢的符号:')

s = s + ' '

n = int(input('请输入边长:'))

triangle(n,s)

# 打印空心三角形

def hollow\_triangle(n,s):

for i in range(1,n+1):

if (i == 1) or (i == 2) or (i == n):

print(' '\*(n-i) + s\*i)

else :

print(' '\*(n-i) +(s + ' '\*(2\*(i-2)) + s))

s = input('请输入一个你喜欢的符号:')

s = s + ' '

n = int(input('请输入边长:'))

hollow\_triangle(n,s)

# 打印空心等腰梯形

def trapezoid(n,m,k,s):

for i in range(1,k+1):

if (i == 1):

print(' '\*(k-i) + s\*n)

elif (i == k):

print(s\*m)

else:

print(' '\*(k-i) + (s + ' '\*(2\*(m-2-k+i)) + s))

s = input('请输入一个你喜欢的符号:')

s = s + ' '

n = int(input('请输入上底的长度:'))

m = int(input('请输入下底的长度：'))

k = int(input('请输入高：'))

trapezoid(n,m,k,s)

# 打印实心梯形

def tixing():

for i in range(1,k+1):

print(' '\*(k-i) + s\*(n+(i+1)))

s = input('请输入一个你喜欢的符号:')

s = s + ' '

n = int(input('请输入上底的长度:'))

k = int(input('请输入高：'))

tixing()

**期末考试题（时间为90分钟）**

1、写函数，给定符号和行数，如’\*’，5，可打印相应行数的如下菱形。主程序输入符号和行数调用该函数进行验证。（20分）

\*

\* \*

\* \* \*

\* \* \* \*

\* \* \* \* \*

\* \* \* \*

\* \* \*

\* \*

\*

1. 用递归和非递归分别实现函数求1!+2!+3!+...+n!，主程序以n=10分别调用。（20分）

#递归

def jiecheng(n):

if n==0:

return 1

else:

return n\*jiecheng(n-1)

m=int(input('输入n以求得n的阶乘'))

n=m

print(m,'的阶乘是:',jiecheng(n))

#非递归

m=int(input('输入m以求得m的阶乘'))

n=m

total=1

while n>0:

total=n\*total

n-=1

print(m,'的阶乘是',total)

1. 北京车牌号的一般形式为：“京X-YYYYY”，其中X为字母，Y为字母或者数字，字母不能为I或者O，数字只能0-9之间。

请编写程序模拟选号过程：一次可以随机生成10个车牌号（不能有重复），依次将其编号为0-9，显示给用户。（20分）

import random

alphabet=['A','B','C','D','E','F','G','H','J','K','L','M','N','P','Q','R','S','T','U','V','W','X','Y','Z']

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

i=0

chepai=set()

while len(chepai)!=10:

chepai.add('京{}-{}{}{}{}{}'.format(random.choice(alphabet),random.choice(alphabet+number),random.choice(alphabet+number),random.choice(alphabet+number),random.choice(alphabet+number),random.choice(alphabet+number)) )

chepai=list(chepai)

for i in range(10):

print(chepai[i])

4、两个向量间的距离可定义为两个向量间的夹角余弦值，给定三个向量，求向量间距离的最小值。三个向量为：[1,2,3,4],[4,5,6,7],[7,8,9,10]（20分）

import math

a=[1,2,3,4]

b=[4,5,6,7]

c=[7,8,9,10]

ab,ac,bc=0,0,0

a\_,b\_,c\_=0,0,0

for i in range(4):

ab=a[i]\*b[i]+ab

ac=a[i]\*c[i]+ack

bc=b[i]\*c[i]+bc

a\_+=a[i]\*\*2

b\_+=b[i]\*\*2

c\_+=c[i]\*\*2

ab=ab/(math.sqrt(a\_)\*math.sqrt(b\_))

ac=ac/(math.sqrt(a\_)\*math.sqrt(c\_))

bc=bc/(math.sqrt(b\_)\*math.sqrt(c\_))

list=[ab,ac,bc]

print(min(list))

5、随机生成100000个整数（1-10000之间）作为集合A，随机生成100000个整数（1-15000之间）作为集合B。

（a）得到A,B所有出现的数及出现次数，分别输出到文件a.txt，b.txt中（新建文件），请输出多行，每行为两个数，两个数之间用逗号分隔（5分）；

（b）列出A中所有回文数（例：12321）以及出现的次数（5分）；

（c）列出B中分别由1-5个数字组成的数各自的总次数与总和，并按照总次数排序输出（如223就是3个数字组成的数，2834就是4个数字组成的数）（5分）；

（d）从a.txt,b.txt读入得到A与B中出现的数（注意不要次数），计算既出现在A又出现在B中的数，追加输出到文件a.txt（5分）。

import random

A=[]

B=[]

for i in range(100000):

A.append(random.randint(1,10001))

B.append(random.randint(1,15001))

#a

with open(r'd:\temp\a.txt','w') as a:

dict\_A={}

for number1 in A:

if number1 in dict\_A:

dict\_A[number1]+=1

else:

dict\_A[number1]=1

for i in dict\_A:

a.write(str(i))

a.write(',')

a.write(str(dict\_A[i]))

a.write('\n')

with open(r'd:\temp\b.txt','w') as b:

dict\_B={}

for number2 in B:

if number2 in dict\_B:

dict\_B[number2]+=1

else:

dict\_B[number2]=1

for i in dict\_B:

b.write(str(i))

b.write(',')

b.write(str(dict\_B[i]))

b.write('\n')

#b

for i in dict\_A:

s=str(i)

if s==s[::-1]:

print(i,dict\_A[i])

#c

#d

with open(r'd:\temp\c.txt','w') as fh:

for i in dict\_A:

if i in dict\_B:

fh.write(str(i))

fh.write('\n')

代码文件上传至：

[liupengyuan.github.io](https://github.com/liupengyuan/liupengyuan.github.io)/[chapter2](https://github.com/liupengyuan/liupengyuan.github.io/tree/master/chapter2)/[homework](https://github.com/liupengyuan/liupengyuan.github.io/tree/master/chapter2/homework)/computer/end/ 仍然是学号

如github不好使，上传至邮箱：[[13552940428@163.com](mailto:13552940428@163.com)@163.com](mailto:13552940428@163.com)

上传/邮件截止时间：20:30，过时无效。