

# 第九章 常用标准库

## 9.1 创建与使用模块

## 9.2 了解模块的细节

## 9.3 标准库

# 模块

- **模块**是程序设计中为完成某一功能所需的一段程序或子程序。为了让代码可重用，请将其模块化
- 任何Python程序都可以作为模块导入，假设有如下程序

```
# hello.py  
print('Hello, world!')
```

将该程序保存为hello.py，然后在Python解释器中导入

```
>>> import hello  
Hello, world!                # 第一次导入，执行程序代码  
>>> import hello  
>>>                          # 第二次导入时，什么也没有发生
```

# 如何编写模块

- 如果将程序保存在自定义的目录中，例如c:\python，则可以使用以下方法将该目录信息告诉解释器

```
>>> sys.path.append('c:/python')
```

```
>>> import hello
```

```
Hello, world!
```

现在解释器知道，除了从默认的目录中查找外，还要从c:\python中寻找模块

- 多次导入模块和一次导入模块的效果是一样的，否则会陷入无限循环（考虑两个模块互相调用）

# 用于定义的模块

- 以下是一个包含函数的简单模块

```
# hello2.py
def hello():
    print('Hello, world!')
```

- 在使用`import hello2`导入后，可以通过以下方式访问该模块中的函数

```
>>> hello2.hello()
Hello, world!
```

- 此时函数在模块的作用域内被定义，实际上模块中定义的所有类、函数和赋值后的变量都成为了模块的特性

# 在模块中增加测试代码

- 有的时候在模块中添加一些代码用来检测模块本身是否能正常工作是很有用的，例如

```
# hello3.py
def hello():
    print('Hello, world!')
# A test:
hello()
```

此时如果你使用import语句导入该模块，会发现如下结果

```
>>> import hello3
```

```
Hello, world!      # 执行了测试代码hello(), 此结果并非我们所希望
```

# 在模块中增加测试代码

- 为避免前述情况，需要告诉解释器模块本身是作为程序运行还是导入到其他程序，为此可以使用\_\_name\_\_变量

```
>>> __name__  
'__main__'  
>>> hello3.__name__  
'hello3'
```

- 可以看出，如果是在“主程序”中运行，则变量\_\_name\_\_的值是'\_\_main\_\_'；如果是在导入的模块中运行，则变量\_\_name\_\_的值则被设定为模块的名字

# 在模块中增加测试代码

- 为合理利用模块中的测试代码，我们可以使用if语句，例如

```
# hello4.py
```

```
def hello():
```

```
    print('Hello, world!')
```

```
def test():
```

```
    hello()
```

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

- 此时可以得到如下结果

```
>>> import hello4    # 仅导入模块，不执行代码
```

```
>>> hello4.hello()
```

```
Hello, world!
```

# 如何找到模块

- 第一种方法：将自己定义的模块文件放在Python解释器的目录下，然后告诉解释器去那里查找模块

```
>>> import sys, pprint
>>> pprint.pprint(sys.path)
['C:/lectures',
 'C:\\Windows\\system32',
 'C:\\Users\\Melody\\AppData\\Local\\Programs\\Python\\Python35-32\\Lib\\idlelib',
 .....,
 'C:\\Users\\Melody\\AppData\\Local\\Programs\\Python\\Python35-32',
 'C:\\Users\\Melody\\AppData\\Local\\Programs\\Python\\Python35-32\\lib\\site-packages']
```



# 如何找到模块

- 前面利用pprint打印出来的这些字符串均对应于一个放置模块的目录，解释器可以从这些目录中找到所需模块
- 只要将自定义的模块放入这些目录中的某一个之中（推荐目录为site-packages），所有程序都可以将其导入了

# 如何找到模块

- 第二种方法：将自己定义的模块文件放在其他目录下，然后告诉解释器去那里查找模块（即本章最开始的例子中使用`sys.path.append`方法）
- 使用第二种方法的原因可能是：
  - 不想将自定义模块填满Python解释器的目录
  - 没有在Python解释器目录中存储文件的权利
  - 想将模块放在任意地方
- 编辑`sys.path`并不是通用方法，标准的实现方法是设置你所使用的计算机的用户变量`PYTHONPATH`，将其作为变量名，然后输入存放模块的目录作为变量值，其中目录之间用分号隔开

# 为模块命名

- 需要注意的是：包含模块代码的文件的名字要和模块的名字一致，然后加上.py扩展名

# 包（package）

- **包**是分组的模块集合，因此可以包含其他的模块；当模块存储在文件中时，包就是模块所在的目录
- 为了让Python将其作为包对待，则目录必须包含一个命名为\_\_init\_\_.py的文件（模块）；如果需要将某个模块放入包中，只需将其放在包目录内即可
- 例如，假设需要建立一个名为drawing的包，其中包括名为shapes和colors的模块，则需要创建的目录和文件应该如表所示

# 包的布局

文件/目录	描述
C:\python\	PYTHONPATH中的目录
C:\python\drawing	包目录（drawing包）
C:\python\drawing\__init__.py	包代码（drawing模块）
C:\python\drawing\colors.py	colors模块
C:\python\drawing\shapes.py	shapes模块

- 此时可以使用以下任意语句进行模块的导入

`import drawing`      # 导入drawing模块

`import drawing.colors`      # 导入colors模块

`from drawing import shapes`      # 导入shapes模块

# 第九章 常用标准库

## 9.1 创建与使用模块

## 9.2 了解模块的细节

## 9.3 标准库

# 模块中的内容

- 查看模块中包含的内容可以使用**dir函数**，它会将其所有特性（以及模块的所有函数、类和变量等）列出

```
>>> import copy
```

```
>>> dir(copy)
```

```
['Error', 'PyStringMap', '_EmptyClass', '__all__', '__builtins__',  
 '__cached__', '__doc__', ....., 'builtins', 'copy', 'deepcopy',  
 'dispatch_table', 'error', 'name', 't', 'weakref']
```

- 可以使用列表推导式将那些不是下划线开头的名字列出

```
>>> [n for n in dir(copy) if not n.startswith('_')]
```

```
['Error', 'PyStringMap', 'builtins', 'copy', 'deepcopy',  
 'dispatch_table', 'error', 'name', 't', 'weakref']
```

# \_\_all\_\_变量

- 前面完整的dir(copy)列表中，出现了\_\_all\_\_这个名字

```
>>> copy.__all__  
['Error', 'copy', 'deepcopy']
```

- \_\_all\_\_这个列表包含了模块的公有接口，它告诉解释器，从模块导入所有名字（**from copy import \***）代表是什么含义（即同时导入了哪些函数和类）
- 也就是说当编写模块时，通过设置\_\_all\_\_的值，可以将其他程序不需要或是不想用的变量、函数和类一起过滤出去；如果没有设定\_\_all\_\_的值，则**import \***语句导入所有不以下划线开头的全局名称



# 使用**help**获得帮助

- 可以对**copy**模块使用**help**函数

```
>>> help(copy)
```

```
Help on module copy:
```

```
NAME
```

```
copy - Generic (shallow and deep) copying operations.
```

```
DESCRIPTION
```

```
Interface summary:
```

```
import copy .....
```

# 使用**help**获得帮助

- 也可以对**copy**函数使用**help**函数

```
>>> help(copy.copy)
```

```
Help on function copy in module copy:
```

```
copy(x)
```

```
    Shallow copy operation on arbitrary Python objects.
```

```
    See the module's __doc__ string for more info.
```

# 查看文档

- 模块信息的来源是文档，通过查看文档可以得到关于函数的精确描述，例如

```
>>> print(range.__doc__)  
range(stop) -> range object  
range(start, stop[, step]) -> range object
```

Return an object that produces a sequence of integers from start (inclusive)

to stop (exclusive) by step. range(i, j) produces i, i+1, i+2, ..., j-1.

start defaults to 0, and stop is omitted! range(4) produces 0, 1, 2, 3.

These are exactly the valid indices for a list of 4 elements.

When step is given, it specifies the increment (or decrement).

# 阅读源代码

- 如果需要阅读某个模块的源代码，可以检查`sys.path`，然后自己找
- 另外一种方法是检查模块的`__file__`属性，例如

```
>>> print(hello5.__file__)
```

```
C:\Users\Melody\AppData\Local\Programs\Python\Python35-32\lib\site-packages\hello5.py
```

# 第九章 常用标准库

## 9.1 创建与使用模块

## 9.2 了解模块的细节

## 9.3 标准库

# sys模块

- **sys模块**支持访问与Python解释器联系紧密的变量和函数

函数/变量	描述
argv	命令行参数，包括脚本名称
exit([arg])	退出当前程序，可选参数为给定的返回值或错误信息
modules	将模块名映射到实际存在的模块上的字典
path	查找模块所在目录的目录名字列表
platform	解释器运行的平台名称
stdin	标准输入流
stdout	标准输出流
stderr	标准错误流

# sys模块

- 通过命令行调用Python脚本时，可能会加上一些命令行参数，这些参数放在sys.argv列表中

```
import sys
```

```
args = sys.argv[1:] # 获取命令行除第一个以外的所有参数的值放入  
#列表
```

```
args.reverse()
```

```
print(' '.join(args)) # 使用空格将列表中提供的若干字符隔开
```

- 如果在MS-DOS提示符下运行

```
C:\Users\Melody\AppData\Local\Programs\Python\Python35-  
32>python reverseargs.py this is a test
```

得到如下结果

```
test a is this
```

# os模块

- **os模块**提供访问操作系统服务的功能

函数/变量	描述
environ	对环境变量进行映射
system(command)	在子shell中执行操作系统命令
sep	路径名中的分隔符，如“/”、“\”或“:”
pathsep	分割路径的分隔符，如“:”、“;”或“::”
linesep	行分隔符（“\n”、“\r\n”或者“\r”
urandom(n)	返回n字节的加密强随机数据



# os模块

```
>>> import os
```

```
>>> print(os.environ['PYTHONPATH']) # 访问系统变量PYTHONPATH  
C:\python
```

```
>>> print(os.sep) # 打印路径名中的分隔符
```

```
\
```

```
>>> print(os.pathsep) # 打印分隔路径的分隔符
```

```
;
```

```
>>> s = os.linesep
```

```
>>> s
```

```
'\r\n' # 打印行分隔符
```

# os模块

```
>>> os.system(r'C:\Program Files (x86)\Mozilla Firefox\firefox.exe')
```

1 # 试图打开浏览器窗口，结果窗口一闪即逝

- 打开浏览器窗口更好的方法包括

```
>>> os.startfile(r'C:\Program Files (x86)\Mozilla Firefox\firefox.exe')
```

- 或者是

```
>>> import webbrowser
```

```
>>> webbrowser.open('http://by.cuc.edu.cn')
```

```
True
```

# fileinput模块

- **fileinput模块**支持遍历文本文件的所有行

函数	描述
<code>input([files[, inplace[, backup]])</code>	遍历多个输入流中的行，返回可以用于for循环遍历的对象
<code>filename()</code>	返回当前文件的名称
<code>lineno()</code>	返回当前（累计）的行数
<code>filelineno()</code>	返回当前文件的行数
<code>isfirstline()</code>	检查当前行是否是文件第一行
<code>isstdin()</code>	检查最后一行是否来自sys.stdin
<code>nextfile()</code>	关闭当前文件，移动到下一文件
<code>close()</code>	关闭序列

# fileinput模块示例

- 假设已经编写了一个Python脚本，现在想为其代码行进行编号

```
#numberlines.py
```

```
import fileinput
```

```
for line in fileinput.input(inplace=True): #对文件进行原地修改
    line = line.rstrip()    # 删除右边空格
    num = fileinput.lineno()
    print('%-50s # %2i' % (line, num))
```

# fileinput模块示例

- 在MS-DOS提示符下运行：  
python numberlines.py numberlines.py
- 得到如下结果

```
#numberlines.py                                     # 1
                                                       # 2
import fileinput                                       # 3
                                                       # 4
for line in fileinput.input(inplace=True):            # 5
    line = line.rstrip()                               # 6
    num = fileinput.lineno()                           # 7
    # print(num)                                       # 8
    print('%-50s # %2i' % (line, num))                # 9
```

# 集合的一点补充说明

- 集合是可变的，所以不能用作字典的键
- 集合本身只能包含不可变值，所以也就不能包含其它集合，如果需要产生集合的集合，则可以使用**frozenset**类型，代表不可变（即可散列）的集合

```
>>> a = set()
```

```
>>> b = set()
```

```
>>> a.add(b)
```

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

```
File "<pyshell#101>", line 1, in <module>
```

```
    a.add(b)
```

```
TypeError: unhashable type: 'set'
```

```
>>> a.add(frozenset(b))
```

```
>>>
```

# 堆

- **堆**是优先队列的一种，使用优先队列能够以任意顺序增加对象，并且能在任何时间找到最小的元素
- Python中没有独立的堆类型，只有heapq模块，须将列表作为堆对象本身

函数	描述
heappush(heap, x)	将x入堆
heappop(heap)	将堆中最小元素弹出
heapify(heap)	将heap属性强制应用到任一列表
heapreplace(heap, x)	将堆中最小元素弹出，将x入堆
nlargest(n, iter)	返回iter中第n大的元素
nsmallest(n, iter)	返回iter中第n小的元素

# 堆

- `heappush`函数将元素入堆，无须先使用`heapify`函数

```
>>> from heapq import *
>>> from random import shuffle
>>> lst = list(range(10))
>>> shuffle(lst)           # 对列表lst的元素进行随机排序
>>> heap = []
>>> for n in lst:
>>>     heappush(heap, n)

>>> heap
[0, 2, 1, 4, 6, 3, 8, 9, 5, 7]
```



# 堆

- **heapify**函数使用任意列表作为参数，通过尽可能少的移位操作将其转换为合法的堆

```
>>> heap = [5, 8, 0, 3, 6, 7, 9, 1, 4, 2]
```

```
>>> heapify(heap)
```

```
>>> heap
```

```
[0, 1, 5, 3, 2, 7, 9, 8, 4, 6]
```

# 双端队列

- 与双端队列（deque）有关的是**collections**模块

```
>>> from collections import deque
```

```
>>> q = deque(range(10))
```

```
>>> q
```

```
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> q.append(5)                                #元素从队列右侧入队
```

```
>>> q
```

```
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 5])
```

```
>>> q.appendleft(6)                            #元素从队列左侧入队
```

```
>>> q
```

```
deque([6, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 5])
```

# 双端队列

```
>>> q
deque([6, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 5])

>>> q.pop()          # 元素从队列右侧出队
5

>>> q.popleft()      # 元素从队列左侧出队
6

>>> q.rotate(3)      # 将队列中所有元素右移3位
>>> q
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6])
>>> q.rotate(-1)     # 将队列中所有元素左移1位
>>> q
deque([8, 9, 0, 1, 2, 3, 4, 5, 6, 7])
```

# time模块

- **time模块**中包括的函数能够实现以下功能:
  - 获得当前时间、操作时间和日期
  - 从字符串中读取时间
  - 将时间格式化为字符串
- 日期可以用实数表示，或者是包含9个整数的元组，例如（2008， 1， 21， 12， 2， 56， 0， 21， 0）表示2008年1月21日12时2分56秒，星期一，是当年的第21天，无夏令时
- *epoch*（新纪元）表示时间的开始，可以使用`time.gmtime(0)`获取系统使用的epoch的值

# time模块中的重要函数

函数	描述
<code>asctime([tuple])</code>	将时间元组转换为字符串
<code>localtime([secs])</code>	将秒数转换为日期元组，以本地时间为准
<code>mktime(tuple)</code>	将时间元组转换为本地时间，以秒计
<code>sleep(secs)</code>	休眠（不做任何事情） <b>secs</b> 秒
<code>strptime(string[,format])</code>	将字符串解析为时间元组
<code>time()</code>	当前时间（新纪元开始后的描述，以UTC为准）

- Python还提供了两个和时间密切相关的模块：**datetime**（支持日期和时间）和**timeit**（对代码段的执行时间进行计时）

```
>>> import time      # 导入time模块
```

```
>>> time.gmtime(0)    # 查看新纪元epoch的值
```

```
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1,  
tm_hour=0, tm_min=0, tm_sec=0, tm_wday=3, tm_yday=1,  
tm_isdst=0)
```

```
>>> time.asctime()    # 将当前时间以字符串的形式表示
```

```
'Wed Dec 9 21:36:33 2015'
```

```
>>> time.localtime()
```

```
time.struct_time(tm_year=2015, tm_mon=12, tm_mday=9,  
tm_hour=21, tm_min=36, tm_sec=46, tm_wday=2, tm_yday=343,  
tm_isdst=0)
```

```
>>> time.time()       # 当前时间，新纪元开始后的秒数
```

```
1449669095.365193
```

# random模块

- **random模块**包括返回随机数的函数，可以用于模拟或者用于产生随机输出的程序
- 事实上所产生的数字仍然是伪随机数，如果需要真的随机性，应该使用os模块的urandom函数

# random模块的重要函数

函数	描述
random()	返回 $0 < n \leq 1$ 之间的随机实数n
getrandbits(n)	以长整型形式返回n个随机位（二进制数）
uniform(a, b)	返回随机实数n，其中 $a \leq n < b$
randrange([start],stop,[step])	返回range(start,stop,step)中的随机数
choice(seq)	从序列中返回任意一个元素
shuffle(seq[,random])	将给定序列的元素进行随机移位
sample(seq, n)	从给定序列中选择给定数目的互不相同元素



```
>>> import random          # 导入random模块

>>> random.random()        # 0和1之间的随机数
0.21278499148668228

>>> random.getrandbits(3)   # 给定的3位数（二进制）
5

>>> random.uniform(2,3)     # 2和3之间的随机实数
2.6310004892208463

>>> random.randrange(1, 10, 2) # range(1,10,2)之间的一个随机数
3

>>> random.choice([1, 2, 3, 4, 5, 6,7]) # 给定序列中任一元素
2

>>> random.sample([1, 3, 4, 5, 6,7], 3) # 给定序列中返回n个随机
[6, 7, 4]                               # 选取的独立元素
```

# random模块的使用示例

- 假设需要找到2015年和2016年间的某个随机时刻，可以如下操作

```
>>> d1 = (2015,1,1,0,0,0,-1,-1,-1)
```

```
>>> t1 = time.mktime(d1)    # 利用时间元组生成时间下限的秒数
```

```
>>> d2 = (2016,1,1,0,0,0,-1,-1,-1)
```

```
>>> t2 = time.mktime(d2)    # 利用时间元组生成时间上限的秒数
```

```
>>> rt = random.uniform(t1, t2) # 随机选取其中的某个时间
```

```
>>> print(time.asctime(time.localtime(rt)))
```

```
Sun Nov 22 20:11:50 2015    # 将随机选取的时间点以字符串形式表示
```

# shelve模块

- **shelve模块**支持在文件中存储数据，只需为其提供文件名即可
- 调用**shelve**中的**open**函数时会返回一个**Shelf**对象，可以用它来存储内容；对于**Shelf**对象，应该将其当作一个普通的字典（字符串作为键）来操作，使用完毕调用其**close**方法

# shelve模块

```
>>> s = shelve.open('c:\\python\\test.data')
```

```
>>> s['x']=['a','b','c']
```

```
>>> s['x'].append('d')
```

```
>>> s['x']
```

```
['a', 'b', 'c']
```

```
>>> s.close()
```

‘d’去哪儿了？

- 由于没有在open方法的参数中设置writeback=True，当添加字符‘d’时，仅将其添加到键‘x’的值的一个副本中，并未保存到修改的版本中

# shelve模块

- 此时需要先将对应键'x'的值先抽取出来，进行修改，然后写回到文件中

```
>>> temp = s['x']
```

```
>>> temp
```

```
['a', 'b', 'c']
```

```
>>> temp.append('d')
```

```
>>> s['x'] = temp
```

```
>>> s['x']
```

```
['a', 'b', 'c', 'd']
```

# re模块：正则表达式

- re模块包含对正则表达式（regular expression）的支持
- 什么是**正则表达式**？— 可以匹配文本片段的模式（pattern），最简单的就是字符串，可以匹配其自身
- **通配符**：正则表达式可以匹配多于一个的字符串，因而可以使用一些特殊字符来创建这类模式，例点号（.）可以匹配任何（**一个，而非两个或多个!**）字符（除换行符），点号也称为通配符（wildcard）

# re模块：特殊字符的转义

- 对特殊字符进行转义
  - 在正则表达式中特殊字符作为普通字符会遇到一些问题，例如将点号作为普通字符进行匹配则会发生歧义（如python.org既可以匹配python.org又可以匹配pythonaorg）
  - 这样需要对特殊字符进行转义，可以在它前面加上反斜线，例如'python\\.org'就可以和'python.org'匹配
  - 使用两个反斜线的原因是需要进行两个级别的转义：  
(1)通过解释器转义；(2)通过re模块转义。或者还可以使用原始字符串r'python\\.org'

```
>>> p1 = re.compile('python\\.org')
>>> re.findall(p1, 'www.python.org')
['python.org']
```

根据前页的内容，普通字符串中为了表示和点号（.）匹配必须先  
用\进行转义，然后通过解释器转义，所以总共是两个\\

```
>>> p2 = re.compile(r'python\\.org')
>>> re.findall(p2, 'www.python.org')
['python.org']
```

在原始字符串中上面的\\.可以简化为\\.



```
>>> p3 = re.compile('\\\\')
```

```
>>> re.findall(p3, 'www\\python\\org\\cn')
```

```
['\\', '\\', '\\']    # 这里列表中每个元素中第1个\\是对第2个\\进行转义
```

同理，这里p3中提供的普通字符串中的4个\\\\应该解释为：先对一个\\使用一个\\进行转义，那么通过解释器转义的话，两个\\前面还要各加一个\\，所以总共是4个\\\\，用于匹配1个\\

```
>>> p4 = re.compile(r'\\')
```

```
>>> re.findall(p4, 'www\\python\\org\\cn')
```

```
['\\', '\\', '\\']    # 这里列表中每个元素中第1个\\是对第2个\\进行转义
```

参照前页中p2的例子，这里p4的原始字符串中的两个\\应该解释为：对1个\\进行转义，因此变为2个\\

试试这个： `p = re.compile('\\\\')`

请问 `re.findall(p, 'www\\\\\\python\\\\\\org\\\\\\cn')`的结果是什么？为什么？

# re模块：字符集

- **字符集**：可以使用中括号括住字符串来创建字符集（character set），从而支持对更多种类字符的匹配
- 例如，'[pj]ython'能够和'python'以及'jython'匹配；此外，还可以使用范围，比如'[a-zA-Z0-9]'匹配任意大小写字母和数字。注意：**字符集只能匹配一个这样的字符！**
- 为了反转字符集，可以在开头使用^字符，例如，'^abc'表示匹配除abc以外的字符

# re模块：选择符和子模式

- 如果需要匹配字符串'python'和'Perl'，字符集和通配符未必好用，此时可以使用用于选择项的特殊字符：**管道符号**（|），所需的模式可以写为'python|perl'
- 有的时候不需要对整个模式使用选择运算符，只是模式的一部分，这时可以使用圆括号括起需要的部分（注意：**可以是单个字符！**），称为**子模式**（subpattern），例如上面例子可以写为'p(ython|erl)'

# re模块：可选项和重复子模式

- 在子模式的后面加上问号，它就变成了**可选项**，可以出现在匹配字符串中，但并非必需
- 例如，模式`r'(http://)?(www\.)?python\.org`可以匹配如下字符串：
  - `http://www.python.org`, `http://python.org`, `www.python.org`, `python.org`
- **重复子模式**：可以指定子模式的出现次数
  - `(pattern)?` 表示子模式可以出现1次或者根本不出现
  - `(pattern)*` 表示子模式可以出现0次或多次
  - `(pattern)+` 表示子模式可以出现1次或多次
  - `(pattern){m, n}` 表示子模式可以出现m~n次

# re模块：字符串的开始和结尾

- 目前所出现的模式匹配都是针对整个字符串的，其实也能寻找与模式匹配的子字符串
- 如果想在字符串的开始位置而不是其它位置进行匹配，则可以使用脱字符（^），例如'^ht+p'会匹配'http://python.org'或者'http://python.org'，但是不匹配'www.http.org'
- 字符串结尾用美元符号（\$），表示对字符串的末尾进行匹配，即匹配整行

# re模块的内容

函数	描述
<code>compile(pattern[,flags])</code>	根据包含正则表达式的字符串创建模式对象
<code>search(pattern,string[,flags])</code>	在字符串中寻找模式
<code>match(pattern,string[,flags])</code>	在字符串的 <b>开始处</b> 匹配模式
<code>split(pattern, string[,maxsplit=0])</code>	根据模式的匹配项来分隔字符串
<code>findall(pattern,string)</code>	列出字符串中模式的所有匹配项
<code>sub(pat,repl,string[,count=0])</code>	将字符串中所有 <b>pat</b> 的匹配项用 <b>repl</b> 替换
<code>escape(string)</code>	将字符串中所有特殊正则表达式字符转义

- 以上可选参数**flags**用于改变对正则表达式的解释方法，具体用法可以参见Python 3.5.0 documentation

# re模块的内容

- **re.compile**将字符串书写的正则表达式转换为正则表达式对象，可以实现更有效率的匹配；使用一次**compile**完成一次转换之后，在每次使用模式的时候就不用再次转换
- 函数**re.search**和**re.match**也会在内部将字符串转换为正则表达式对象

# re模块的内容

- 模式对象本身也有查找/匹配的函数，所以 `re.search(pat,string)` 等价于 `pat.search(string)`，其中 `pat` 使用 `re.compile` 创建的模式对象。例如  
    `prog = re.compile(pattern)`  
    `result = prog.match(string)`    等价于  
    `result = re.match(pattern, string)`
- 基于以上原因，总是可以先用 `re.compile` 方法得到模式对象，然后调用模式对象的查找/匹配函数



# re模块的内容

- 函数`re.search`在给定字符串中寻找第一个匹配给定正则表达式的子字符串，找到返回匹配对象`MatchObject`(值为`True`)，否则返回`None`(值为`False`)。因此可以用于条件语句

```
>>> import re
```

```
>>> if re.search(r'p(ython|erl)', 'http://www.python.org'): print('Found it!')
```

```
Found it!
```

- 函数`re.match`在给定字符串的开头匹配正则表达式，找到返回`MatchObject`，否则返回假

```
>>> if not re.match(r'p(ython|erl)', 'www.python.org'): print('Not found!')
```

```
Not found!
```

# re模块的内容

- 函数`re.split`根据模式的匹配项来分隔字符串，类似于字符串的`split`方法，不过是用完整的正则表达式代替了固定的分隔符字符串

```
>>> text = 'alpha, beta,,,,,,gamma, delta'
```

```
>>> re.split('[,]+', text)
```

```
['alpha', ' beta', 'gamma', ' delta']
```

```
>>> re.split('[,]+', text, maxsplit = 2) # 使用maxsplit指明最多分割次数
```

```
['alpha', ' beta', 'gamma, delta']
```

```
>>> pat = r'[.?\-“,]+’      # 查找所有标点符号，此处”-”符号被转义
```

```
>>> text = "Hm... Err -- are you sure?" he said, sounding insecure.'
```

```
>>> re.findall(pat, text)
```

```
['"', '...', '--', '?', ',', '.', '']
```

# re模块的内容

- 函数`re.escape`可以对字符串中所有可能被解释为正则运算符的字符进行转义，例如

```
>>> re.escape('www.python.org')
```

```
'www\\.python\\.org'           # 对"."号进行转义
```

```
>>> re.escape('But where is the ambiguity?')
```

```
'But\\ where\\ is\\ the\\ ambiguity\\?' # 对"?"号和空格进行转义
```

# 匹配对象和组


- `re`模块中那些能够对字符串进行模式匹配的函数,当找到匹配项时均能返回`MatchObject`对象

```
>>> pat = re.compile(r'w+\.(python|perl)\.org')
```

```
>>> re.match(pat, 'w.perl.org')
```

```
<_sre.SRE_Match object; span=(0, 10), match='w.perl.org'>
```

一个组的开始和结束位置



- 这些`MatchObject`对象包括了匹配模式的子字符串的信息,还包括了哪个模式匹配了子字符串中哪部分信息(这些部分叫做**组**)
- **组**就是放置在圆括号中的子模式,其序号取决于它左侧的括号数,组0即为整个模式

# 匹配对象的重要方法

- 例如，在下面的模式中'There (was a (wee) (cooper)) who (lived in Fyfe)'包含了以下组：
  - 0 There was a wee cooper who lived in Fyfe
  - 1 was a wee copper
  - 2 wee
  - 3 cooper
  - 4 lived in Fyfe

函数	描述
<code>group([group1,...])</code>	获取给定子模式的匹配项
<code>start([group])</code>	返回给定组匹配项的起始位置
<code>end([group])</code>	返回给定组匹配项的结束位置
<code>span([group])</code>	返回给定组匹配项起始和结束位置

# 匹配对象的重要方法

```
>>> m = re.match(r'www\.(.*)\.{3}','www.python.org')
```

```
>>> m.group(1) # 组1包括(.)即0个或多个任意字符  
'python'
```

```
>>> m.start(1) # 组1的匹配项的起始位置的索引  
4
```


```
>>> m.end(1) # 组1的匹配项的结束位置的索引+1  
10
```

```
>>> m.span(1) # 组1的匹配项的起始和结束位置的索引，元组形式  
(4, 10)
```

# 作为替换的组号和函数

- 建立正则表达式 `emphasis_pat = r'\*([^\*]+)\*'` ，该表达式对`*`进行了转义
- 由于正则表达式一般难以理解，可以在`re`模块的函数中使用`VERBOSE`标志，它允许在模式中添加空白，函数会忽略这些空白；例如与上面模式等价的对象可以写为

```
>>> emphasis_pat = re.compile(r"  
    \*  # 替换模式标记的开始,星号*  
    (   # 进行匹配的组的开始  
    [^\*]+ # 对除星号*以外的所有字符匹配  
    )   # 进行匹配的组的结束  
    \*  # 替换模式标记的结束,星号*  
    ", re.VERBOSE)  
>>> re.sub(emphasis_pat, r'<em>\1</em>', 'Hello, *world*!')  
'Hello, <em>world</em>!' 
```



组号

# 贪婪和非贪婪模式

- 重复运算符（\*、+等等）默认是贪婪的，即会进行尽可能多的匹配

```
>>> emphasis_pat = r'\*(.+)\*'
>>> re.findall(emphasis_pat, '*This* is *it*!')
['This* is *it']
```

- 模式匹配了从开始星号到结束星号之间所有字符，如果不需要这种贪婪的行为，只需在重复运算符后加上一个问号（?）即可

```
>>> re.findall(emphasis_pat, '*This* is *it*!')
['This', 'it']
```



# 找出Email的发件人邮箱

Return-Path: <ccf\_mk@ccf.org.cn>

Delivered-To: zhoujing@cuc.edu.cn

Received: from mxgate1.cuc.edu.cn (unknown [202.205.16.252]) by smtp.cuc.edu.cn (Postfix) with SMTP id BA9C4A827A for <zhoujing@cuc.edu.cn>; Thu, 10 Dec 2015 19:26:19 +0800 (CST)

X-scanner: By EQAVSE AntiVirus Engine

X-scanresult: NOREP

**X-MAILFROM: <ccf\_mk@ccf.org.cn>**

X-RCPTTO: <zhoujing@cuc.edu.cn>

X-FROMIP: 115.28.56.183 X-EQManager-Scanned: 1

X-Received:unknown,115.28.56.183,20151210192618 .....

- 以上为文件message.txt中存放的某封电子邮件部分内容

# 找出Email的发件人邮箱

```
# Chapter9_findsender.py
```

```
# 发件人邮箱信息所在行应该为 X-MAILFROM: <ccf_mk@ccf.org.cn>
```

```
import fileinput, re
```

```
pat = re.compile('X-MAILFROM:(.*) <(.*)? >$')
```

匹配整行

```
for line in fileinput.input():
```

```
    m = pat.match(line)
```

```
    if m: print(m.group(2)) # 打印出与第2组匹配的项
```

# 找出Email的发件人邮箱

- 在MS-DOS提示符下运行

C:\python>python Chapter9\_findsender.py message.txt 得到如下结果  
ccf\_mk@ccf.org.cn

- 也可以将源代码中的fileinput.input()改为fileinput.input(r'c:\python'), 这样就能够直接在IDLE的文本编辑窗口中选用"Run Module"或直接按F5即可执行, 显示结果如下:

ccf\_mk@ccf.org.cn

>>>

# 小结

- **模块**：模块即为子程序，其主函数用于定义函数和类；若函数包含测试代码，则应将其放置在检查 `__name__ == '__main__'` 是否为真的if语句中；能够在PYTHONPATH中找到的模块都可以导入，语句 `import foo` 可以导入存储在 `foo.py` 中的所有模块
- **包**：包是包含其它模块的模块，是作为包含 `__init__.py` 文件的目录来实现的
- **模块的细节**：可以通过 `dir`、检查 `__all__` 变量以及使用 `help` 函数来查看模块的细节
- Python中包含了一些**标准库**，本章中涉及到的有
  - `sys`：访问多个和Python解释器联系紧密的变量和函数

# 小结

- **os**: 访问多个和操作系统联系紧密的变量和函数
- **fileinput**: 遍历多个文件和流中的所有行
- **heapq**和**collections**: 提供了堆和双端队列这些数据结构
- **time**: 获取当前时间，并可以进行时间日期操作和格式化
- **random**: 产生随机数、从序列中选取随机元素以及对序列中元素进行随机排序
- **shelve**: 创建持续性映射，并将映射内容保存在给定文件名的数据库中
- **re**: 支持正则表达式

# 本章的新函数

`dir(obj)`

返回按字母顺序排序的属性名称列表

`help([obj])`

提供特定对象的交互式帮助信息