

python语言程序设计

python基本数据结构

数字

1. 包括整型、浮点型、复数（与其他语言重点区分）
2. 赋值方式和一般变量相同
3. 整数是不可变类型
4. 复数的常见属性
 - num.real 复数的实部
 - num.imag 复数的虚部
 - num.conjugate() 复数的共轭复数
5. 工厂函数
 - cmp () 比较两个数的大小
 - str() 返回对象的字符串表述
 - int() 返回整数字符串的数字类型
 - long() float() complex() 同int
6. 功能函数
 - abs() 求绝对值
 - divmod()根据除数和被除数得到商和余数的元组
 - pow() 指数函数
 - round()四舍五入
 - 进制转换函数hex() oct() ord() chr() unichr()

序列

序列是一种通用数据结构，比如数组，列表等，有一些操作序列的通用函数，如下

- in /not in 成员关系操作符判断某元素是否属于某序列
- seq[ind] 获取下标为ind的元素
- seq[start:stop:step] 获取下标为start到下标为stop-1的元素集合,步长为step
- seq * expr 重复序列expr次
- seq1 + seq2 连接序列seq1与seq2

通用内建函数

- list(iter) 将可迭代对象转换为列表
- tuple (iter)将可迭代对象转换为元组
- unicode() 将对象转换为unicode字符串
- enumerate (iter) 接受一个可迭代对象为参数，返回一个enumerate对象，可迭代，其指向元素由iter中所含元素与该元素位置组成的元组
- len(iter) 返回序列iter的长度

- `max(iter, key=None)`返回iter中的最大值，key是一个可以传递给sort函数的回调函数，可以规定排序的规则
- `min(iter, key=None)`与max雷同
- `sorted(iter, func=None, key=None, reverse=False)`排序函数，func指定排序函数，key指定每个元素中提取用于比较的关键字，reverse指定是否反向排序
- `zip(iter1, iter2...itern)`返回一个列表，该列表第一个元素是由iter的第一个元素组成的元组，依次类推
- `reversed(seq)`返回逆置的迭代器
- `any(seq)`如果seq中有一个为True，则返回true
- `all(seq)`如果seq中元素都为True，返回True

字符串

- 字符串的创建与赋值使用'与"并无差异，可以将字符串当作列表，使用切片操作符得到子串，序列通用函数均可适用于字符串，字符串是不可变对象
- 在字符串前加字母r表示原生字符串，其中的特殊字符均当作普通字符来看待
- 在字符串前加字母u表示unicode字符串
- 使用'''包围字符串，字符串可以跨越多行，并且可以包含特殊字符
- 仅适用于字符串的函数
 - 格式化操作符%（格式化使用最新的format函数）
 - `raw_input()`输出提示字符串，并得到用户输入
 - `ord('char')`返回某字符的ascii数值
 - `chr()`返回某数字的ascii字符
 - `string.capitalize()`将首字母大写
 - `string.center(width)`返回原字符串居中，并用空格填充至width的新字符串
 - `string.count(str, begin, end)`返回string中str出现的次数
 - `string.decode()`
 - `string.encode()`
 - `string.endswith(obj, beg, end)`判断string是否以obj结束
 - `string.startswith(obj)`判断string是否以obj开始
 - `string.expandtabs(tabsize)`将字符串中的tab替换为空格
 - `string.find(str, beg, end)`在string中查找str，返回找到的索引值，否则返回-1
 - `string.isalnum()`判断string是否仅由字母数字组成（不包括空格）
 - `string.isalpha()`判断是否所有字符都是字母
 - `string.isdigit()`判断是否所有字符都是数字
 - `string.isdecimal()`判断十进制数字
 - `string.islower()`判断小写
 - `string.isupper()`判断大写
 - `string.join(seq)`返回以string为分割符，将seq中所有元素拼接起来的新字符串
 - `string.lower()`小写
 - `string.upper()`大写
 - `string.strip()`裁掉string两端的空白字符

- `string.split(sep)`以`sep`为分隔符分隔`string`，返回列表
- `string.format()`格式化字符串输出 `'{} is better than {}'.format('red','yellow')` 其中`{}`标识的位置表示将要放入参数，将`format`函数参数格式化后放入原字符串中，`{}`中的内容可以像下面这样 `[[fill]align][sign][#][0][width][,][.precision][type]`

`fill` 表示填充字符，默认是空格

`align` 表示对齐方式 '`<`' 左对齐 '`>`' 右对齐 '`^`'居中 `sign` 在显示数字时，规定是否显示+ -号， '-'默认取值，正数省略，负数显示 '-'号 `width`显示的宽度 `precision`显示的精度 `type`显示的类型 一些示例

- 通过参数位置访问

```
'{} is better than {}'.format('beautiful', 'ugly')
```

- 通过下标访问

```
'{0} is better than {1}'.format('beautiful', 'ugly')
```

- 使用字典

```
In [12]: score = {'xiaoming':80, 'xiaohong':90}

In [13]: "xiaoming's score is {xiaoming}, xiaohong's score is {xiaohong}".format(**score)
Out[13]: "xiaoming's score is 80, xiaohong's score is 90"
```

列表

- 列表是序列的一种，序列通用操作函数均适用于列表，切片操作符，成员关系操作符
- 列表解析是一种快速构造列表的方法形如 `[2*i for i in range(10)]`
- 内建函数
 - `len`序列长度
 - `list()`列表构造
 - `zip`，`enumerate ()`，`sum`
 - `sorted()`，`reversed()`
 - `append()`向列表末尾添加一个元素
 - `seq.extend(seq2)`将`seq2`中的元素添加进`seq`
 - `list.count(obj)`返回`obj`在`list`中出现的次数
 - `list.remove(obj)`删除`obj`元素
 - `list.pop()`删除并返回指定位置的对象，默认最后一个
 - `list.insert()`插入元素
 - `list.sort()`

元组

- 元组是不可变类型
- 浅拷贝使用完全切片操作`[:]`，使用工厂函数`list()`，`turtle()`，使用`copy`函数，浅拷贝拷贝对象的引用，深拷贝使用`deepcopy ()`

字典

- 可以理解成一个可变的哈希表
- 字典的键必须是不可变类型，键必须是可哈希的，不可一个键对应多个值
- 创建字典
 - `dict`工厂函数
 - `{}.fromkeys(seq,default)`以`seq`中的元素为键创建字典

- 访问字典中的元素dict['key']
- dict.clear()删除所有元素
- dict.pop['key']返回并删除指定元素
- dict.items()返回包含字典键值对的列表
- dict.keys()返回包含字典键的列表
- dict.values()返回包含所有值的列表
- dict.update(dict2)将dict2中的元素添加进dict

集合

- 操作符 in , not in , == , >= , <= , > , < , & , | , - , ^
- 用set创建集合或用frozenset创建不可变集合
- s.add(e)添加元素
- s.update(t)
- s.pop()
- collections中的defaultdict对象
 - defaultdict接受一个类型作为参数创建一个字典，规定该字典默认值的类型，使用dict函数创建普通字典，当访问该字典中不存在的键时，会使程序抛出异常，使用defaultdict创建的字典，访问不存在的键时，返回默认类型的值。比如defaultdict(list),访问该字典中不存在的键值对时返回空的列表
- collections中的Counter对象
 - Counter('iter')可以从一个iterable的对象创建计数器
 - 访问：当指定的键不存在时返回0，否则返回该值的次数
 - c.update(element)新增元素
 - c.subtract(element)减少元素
 - c.most_common(n)返回topN的元素

文件处理

文件读写部分，python有一个基本的内置文件读取函数open `open(filename, mode)` 其中mode可以是r、w、x、a，模式a表示打开文件会在文件末尾追加内容，使用w模式打开文件，如果文件存在会将文件清空，再写入，如果使用x模式打开文件，如果文件存在会抛出异常。open函数返回文件对象，当操作完成时，使用close方法关闭文件流

常见文件读写函数

- read([size])读取指定长度文件内容，如果size为空，读取所有
- readline()读取文件一行
- readlines()读取文件所有内容到一个列表
- seek(offset[,whence])函数更改文件指针的位置，whence表示从何处偏移，默认值是0，表示从文件头开始
- write(str)将str写入文件
- writelines(strlist)将字符串列表写入文件
- 读取文件内容常用迭代（python文件对象是可迭代的）

```
1 with open("test.txt") as f:
2     for line in f:
3         print line
```

- 示例：将文件中所有单词首字母变大写

```
1 with open("program_log.txt") as inf, open("program.txt", 'w') as outf:
2     for line in inf:
3         outf.write(" ".join([word.capitalize() for word in line.split()]))
4         outf.write("\n")
```

os.path中的路径与文件管理函数

os模块

- os.listdir(dir)列出dir下所有的目录与文件
- os.getcwd()获取当前工作目录
- os.chdir(dir)改变工作目录
- os.rmdir('dir')删除空目录
- os.mkdir(dir)创建目录
- rename()重命名目录或文件
- os.chmod(file, mod)更改文件权限
- os.access(file, auth)判断用户对该文件是否有指定权限
其中auth可以取os.R_OK,os.W_OK,os.X_OK

拆分

- os.path.split(path)将path拆分为目录与文件名称，并返回二元组
- os.path.dirname(path)获取path中的目录名称
- os.path.basename(path)获取path中的文件名称
- os.path.splitext(path)获取path中文件扩展名和去处扩展名后的二元组

构建

- os.path.join(dir...)接收可变参数，组合为完整路径
- os.path.abspath(dir)得到某文件绝对路径
- os.path.expanduser(path)展开用户目录

```
os.path.expanduser('~ / test / txt')
```

获取文件属性

- os.path.getsize(file)获取文件大小
- os.path.{getctime(),getatime(),getmtime()}

判断文件类型

- os.path.exists(file)是否存在
- os.path.isfile(file)是否是文件
- os.path.isdir(file)是否是目录
- os.path.islink()

- `os.path.ismount()`

示例

- 获取用户目录下所有的文件列表 `[name for name in os.listdir(os.path.expanduser('~')) if os.path.isfile(name)]`
- 获取用户目录下所有的目录列表 `[name for name in os.listdir(os.path.expanduser('~')) if os.path.isdir(name)]`
- 获取用户目录下目录到绝对路径之间的字典 `{name: os.path.abspath(name) for name in os.listdir(os.path.expanduser('~')) if os.path.isdir(name)}`
- 获取最常用的10条命令

```
1 with open(os.path.expanduser('~/.bash_history')) as inf:
2     for line in inf:
3         cmd = line.strip().split()
4         if cmd[0] == 'sudo':
5             c[cmd[1]]+=1
6         else:
7             c[cmd[0]]+=1
8 c.most_common(10)
```

文件查找

使用字符串匹配

```
1 [item for item in os.listdir('.') if item.endswith('txt')]
2 [item for item in os.listdir('.') if item.startswith('pro')]
```

```
In [2]: ls
certificate/ Desktop/ Downloads/ Pictures/ SCORE-MIB.txt ssconfig/ tools/
CloudMusic/ dev-envir/ note/ program_log.txt secure/ TEST-MIB.txt winetools/
code/ Documents/ perl5/ program.txt sh/ tmp/ workspace/

In [3]: [ item for item in os.listdir('.') if item.endswith('txt')]
Out[3]: ['SCORE-MIB.txt', 'TEST-MIB.txt', 'program_log.txt', 'program.txt']

In [4]: [ item for item in os.listdir('.') if item.startswith('pro')]
Out[4]: ['program_log.txt', 'program.txt']
```

使用fnmatch

- fnmatch是一个文件名称匹配库，可以理解为简单的正则表达式匹配，支持的正则符号有
 - *匹配任意数量任意字符
 - ?匹配单个任意字符
 - [seq]匹配seq中的字符
 - ![seq]不匹配seq中的字符
- 常用函数
 - `fnmatch(file,pattern)`判断文件是否符合特定模式
 - `fnmatchcase`同上，忽略大小写
 - `filter(names,pat)`返回输入列表中符合pat的元素组成的列表

- 示例

- 找到当前目录所有jpg文件

```
1 [item for item in os.listdir('.') if fnmatch.fnmatch(item, '*.jpg')]
```

```
[item for item in os.listdir('.') if fnmatch.fnmatch(item, '*.jpg')]
['c2.jpg', 'a2.jpg', 'b2.jpg', 'd2.jpg']
```

- 返回所有a-c开头的文件

```
1 [item for item in os.listdir('.') if fnmatch.fnmatch('[a-c]*')]
```

```
[item for item in os.listdir('.') if fnmatch.fnmatch(item, '[a-c]*')]
['c1.txt', 'c2.jpg', 'a2.jpg', 'a1.txt', 'b2.jpg', 'b1.txt']
```

- 返回不是a-c开头的文件

```
1 fnmatch.filter(os.listdir('.'), '[!a-c]*')
```

```
fnmatch.filter(os.listdir('.'), '[!a-c]*')
['d1.txt', 'd2.jpg']
```

使用glob

也可以直接使用glob查找文件，相当于os.listdir与fnmatch `glob.glob('[a-c]*.jpg')`

```
In [21]: glob.glob('[a-c]*.jpg')
Out[21]: ['c2.jpg', 'a2.jpg', 'b2.jpg']
```

使用os.walk遍历目录树

os.walk(top, topdown=True, onerror=None, followlinks=False)walk函数递归返回三元组(dirpath, dirnames, filenames),dirpath保存当前目录，dirnames返回目录列表，filenames返回文件列表，如果想要忽略掉一个子目录，可以直接从dirnames中删除该目录 寻找某目录下某类型文件通用代码

```
1 import os
2 import fnmatch
3
4 # 判断某文件是否符合某模式
5 def is_file_match(filename, partterns):
6     for parttern in partterns:
7         if fnmatch.fnmatch(filename, parttern):
8             return True
9         else:
10            return False
11
12 # 查找root目录下除exculde_dir中符合parttern的文件
13 def find_specific_file(root, partterns=['*'], exclude_dir=[]):
```

```

14     for rootdir, dirnames, filenames in os.walk(root):
15         for filename in filenames:
16             if is_file_match(filename, partterns):
17                 yield os.path.join(rootdir, filename)
18
19         for dirname in dirnames:
20             if dirname in exclude_dir:
21                 dirnames.remove(dirname)

```

使用shutil

复制文件或目录

```

1 import shutil
2 shutil.copy('a.txt', 'b.txt')
3 shutil.copytree('dir1', 'dir2')

```

移动与重命名

```

1 shutil.move('a.txt', 'b.txt')
2 shutil.move('a.txt', 'dir')
3 shutil.move('dir1', 'dir2')

```

删除文件或目录

```

1 shutil.rmtree('dir')

```

通过计算md5校验码检查文件是否是同一个

```

1 import hashlib
2 d = hashlib.md5()
3 with open('backup.py') as f:
4     for line in f:
5         d.update(line)
6 d.hexdigest()

```

查找某目录下相同文件

```

1 from __future__ import print_function
2 import hashlib
3 import fnmatch
4 import os
5 import sys
6
7 # 每次从文件中读取8192个字节
8 CHUNK_SIZE = 8192
9 # 判断文件是否符合某模式

```



```

10 def is_file_match(filename, partterns):
11     for parttern in partterns:
12         if fnmatch.fnmatch(filename, parttern):
13             return True
14         else:
15             return False
16
17 # 查找指定目录下指定文件
18 def find_specific_file(root, partterns=['*'], exclude_dir=[]):
19     for rootdir, dirnames, filenames in os.walk(root):
20         for filename in filenames:
21             if is_file_match(filename, partterns):
22                 yield os.path.join(rootdir, filename)
23
24         for dirname in dirnames:
25             if dirname in exclude_dir:
26                 dirnames.remove(dirname)
27
28 # 循环获取文件内容
29 def get_chunk(filename):
30     with open(filename) as f:
31         while True:
32             chunk = f.read(CHUNK_SIZE)
33             if not chunk:
34                 break
35             else:
36                 yield chunk
37
38 # 获取文件md5校验和
39 def get_chsum(filename):
40     d = hashlib.md5()
41     for chunk in get_chunk(filename):
42         d.update(chunk)
43     return d.hexdigest()
44
45 # 执行查找比较过程
46 def main():
47     sys.argv.append("")
48     if not os.path.isdir(sys.argv[1]):
49         print("{0} is not dir!".format(sys.argv[1]))
50         exit()
51     record = {}
52     for item in find_specific_file(sys.argv[1]):
53         checkres = get_chsum(item)
54         if checkres in record:
55             print("find duplicate file {0} vs {1}".format(record[checkres], item))
56         else:
57             record[checkres] = item
58
59     print("filelist:", record)
60
61 if __name__ == '__main__':
62     main()

```

压缩包管理

使用tarfile创建与读取tar包

读取tar文件

```
1 with tarfile.open('mess.tar') as inf:
2     for mem in inf.getmembers():
3         print mem
```

创建tar文件

```
1 with tarfile.open("mess2.tar", mode='w') as outf:
2     outf.add('b2.c')
```

tarfile对象常用方法

- getnames()获取tar包的文件列表
- extract()提取单个文件
- extractall()提取所有文件
- 使用tarfile创建压缩包时，只需在mode参数指定压缩方式
`tarfile.open('test.tar', mode='w:bz2')`

数据存储

序列化与反序列化

程序运行时，所有内容保存在内存中，有时候需要把某些变量的内容或是运行时信息保存下来，下次程序运行时读取这些保存的内容，从而可以让程序从停止的地方继续执行。把变量、对象变为通用存储格式的过程叫序列化，从磁盘上的数据恢复出变量、对象的过程叫反序列化。python中的序列化与反序列化使用pickle模块或cpickle模块，其中cpickle使用c语言实现，速度上会快很多

导入

```
1 try:
2     import cPickle as pickle
3 except ImportError:
4     import pickle
```

dump与dumps

dump和dumps方法可以将对象转变为存储字节字符串，其中dumps方法，接受一个对象，返回该对象的字节字符串，dump接受一个对象和一个文件句柄，直接将序列化之后的内容写入文件。

```

1 In [3]: fp = open('dict.txt','w')
2 In [4]: test_dict = {'index_' + i: i for i in range(5)}
3 In [7]: cPickle.dumps(test_dict)
4 Out[7]: "(dp1\ns'index_4'\np2\nI4\nsS'index_2'\np3\nI2\nsS'index_3'\np4\nI3\nsS'index_0'\np5\nI0\nsS'index_1'\np6\nI1\ns."
5 In [8]: cPickle.dump(test_dict, fp)
6 In [9]: fp.close()

```

load与loads

load和loads方法将字节字符串反序列化为对象，load接受一个文件句柄，解析出文件中的对象，loads接受一个字节字符串，解析出对象

```

1 In [10]: dict_ori = cPickle.load(open('dict.txt','r'))
2
3 In [11]: dict_ori
4 Out[11]: {'index_0': 0, 'index_1': 1, 'index_2': 2, 'index_3': 3, 'index_4': 4}
5 In [15]: pickle.loads(pickle.dumps(test_dict))
6 Out[15]: {'index_0': 0, 'index_1': 1, 'index_2': 2, 'index_3': 3, 'index_4': 4}

```

json对象存储与解析

使用json模块进行json存储与解析，json模块常用的函数有dumps，dump，load，loads与pickle类比记忆，但是有一些需要注意的选项需要看一下，ensure_ascii参数指定了怎样表示非ascii字符，默认为\uXXX，当设置ensure_ascii为False时，可以正常显示。skipkeys指定，当键不是python基本类型时的操作，如果保持默认False，会报TypeError异常，设置为True则会跳过该键

```

1 In [18]: test_dict
2 Out[18]: {'index_0': 0, 'index_1': 1, 'index_2': 2, 'index_3': 3, 'index_4': 4}
3
4 In [19]: json.dumps(test_dict)
5 Out[19]: '{"index_4": 4, "index_2": 2, "index_3": 3, "index_0": 0, "index_1": 1}'
6
7 In [20]: json.dump(test_dict, open('json_dump.txt', 'w'))
8
9 In [21]: json.load(open('json_dump.txt'))
10 Out[21]: {u'index_0': 0, u'index_1': 1, u'index_2': 2, u'index_3': 3, u'index_4': 4}
11
12 In [22]: json.loads(json.dumps(test_dict))
13 Out[22]: {u'index_0': 0, u'index_1': 1, u'index_2': 2, u'index_3': 3, u'index_4': 4}

```

python操作csv与office文件

csv文件操作使用csv模块

csv文件是具有固定格式的一类文件，每行使用固定的行分隔符，每个域也是用固定的域分隔符（通常是','）。

写入csv文件

```

1 In [6]: title = ['host', 'port']
2 In [8]: row_1 = ['192.168.56.101', '80']
3
4 In [9]: row_2 = ['192.168.56.101', '8080']
5 fp = open('tmp.txt', 'w')
6
7 In [12]: csv_write = csv.writer(fp)
8
9 In [13]: csv_write.writerow(title)
10
11 In [14]: csv_write.writerows([row_1, row_2])
12
13 In [15]: fp.close()

```

读取csv文件

使用普通文件读取，或使用命名元组

```

1 In [18]: fp = open('tmp.txt', 'r')
2 In [19]: csv_reader = csv.reader(fp)
3 In [20]: title = next(csv_reader)
4 In [21]: print title
5 ['host', 'port']
6 In [22]: for line in csv_reader:
7     ...:     print line
8     ...:
9 ['192.168.56.101', '80']
10 ['192.168.56.101', '8080']
11
12 In [23]: from collections import namedtuple
13 In [24]: fp.close()
14 In [25]: fp = open('tmp.txt', 'r')
15 In [26]: csv_reader = csv.reader(fp)
16 In [27]: title = next(csv_reader)
17 In [28]: Row = namedtuple('Row', title)
18 In [29]: for line in csv_reader:
19     ...:     row = Row(*line)
20     ...:     print row.host, row.port
21     ...:     print row
22     ...:
23 192.168.56.101 80
24 Row(host='192.168.56.101', port='80')
25 192.168.56.101 8080
26 Row(host='192.168.56.101', port='8080')

```

多线程与多进程

进程

进程的概念是需要理解的，进程是操作系统中正在运行的一个程序实例，操作系统通过进程操作原语来对其进行调度。操作系统得到调用某个进程指令时，将硬盘上的程序调入内存，分配空间，初始化进程堆栈，然后进程开始运行。有时候我们有同时运行多个程序的需求，如果你的电脑只能做一件事，那是一件很抓狂的事。操作系统通过进程调度算法调度进程运行，使计算机看起来同时运行了很多程序。

python中多进程实现

fork

fork是linux下创建新进程的机制，通过fork父进程复制出一个相似，通过fork返回值判断执行子进程代码

```
1 import os
2 def main():
3     print 'current Process {} start...'.format(os.getpid())
4     pid = os.fork()
5     if pid < 0:
6         print 'fork error!'
7         exit(1)
8     elif pid == 0:
9         print 'child Process {} starting... , and my parent process is
{}'.format(os.getpid(), os.getppid())
10
11     else:
12         print 'I({}) created the child({})'.format(os.getpid(), pid)
13 if __name__ == '__main__':
14     main()
```

得到如下输出

```
1 current Process 5351 start...
2 I(5351) created the child(5352)
3 child Process 5352 starting... , and my parent process is 5351
```

multiprocess

使用multiprocess模块创建子进程，模块提供一个Process对象描述进程，创建进程时，只需要传入一个可调用的函数，以及函数运行时的参数即可

```
1 import os
2 import multiprocessing
3
4
5 def run_proc(name):
6     print 'child process {}({}) running...'.format(name, os.getpid())
7
8 def main():
9     print 'main process starting... {}'.format(os.getpid())
10    processes = []
11    for i in range(5):
12        p = multiprocessing.Process(target=run_proc, args=(str(i),))
```

```

13     processes.append(p)
14     print 'process {} will start'
15     p.start()
16
17     for p in processes:
18         p.join()
19     print 'processes end'
20 if __name__ == '__main__':
21     main()

```

使用进程池限制进程个数。multiprocessing模块中的Pool对象，用来表示进程池，Pool对象的apply_async函数用于创建进程，同样的给出可调用的函数与函数运行需要的参数

```

1  import os
2  from multiprocessing import Pool
3
4  def run_proc(name):
5      print 'child process {}({}) running...'.format(name, os.getpid())
6
7  def main():
8      print 'main process starting... {}'.format(os.getpid())
9      processes = Pool(processes=3)
10     for i in range(5):
11         processes.apply_async(run_proc, (str(i),))
12     processes.close()
13     processes.join()
14
15
16     print 'processes end'
17 if __name__ == '__main__':
18     main()

```

进程间通信

通过队列

队列，即multiprocessing模块中的Queue对象，队列中有某种资源，可以向队列中放入数据，另一个进程从队列中取出数据，当无数据可用时，消费者应该决定是阻塞等待资源还是返回一个错误，当队列已满，生产者应决定是阻塞等待可用空间还是返回错误。Queue对象有两个主要方法，get和put，get从队列中取出数据，put向队列中添加数据。blocked参数决定当队列不满足条件时是阻塞等待还是返回错误，默认为True，表示阻塞等待。timeout指定了队列阻塞的时间，如果超时，同样返回异常

```

1  from multiprocessing import Queue, Process
2  import os, time, random
3
4  def Proc_writer(q, urls):
5      print 'Process {} is writing...'.format(os.getpid())
6      for url in urls:
7          q.put(url)
8          print 'put {} to the Queue'.format(url)
9          time.sleep(random.random())

```

```

10
11 def Proc_reader(q):
12     print 'Process {} is reading...'.format(os.getpid())
13     while True:
14         url = q.get(True)
15         print 'get the {} from the Queue'.format(url)
16
17 def main():
18     print 'main process {} is running...'.format(os.getpid())
19     q = Queue()
20     process_1 = Process(target=Proc_writer, args=(q,['url_1', 'url_2', 'url_3']))
21     process_2 = Process(target=Proc_writer, args=(q,['url_4', 'url_5', 'url_6']))
22     process_3 = Process(target=Proc_reader, args=(q,))
23     process_1.start()
24     process_2.start()
25     process_3.start()
26     process_1.join()
27     process_2.join()
28     process_3.terminate()
29     print 'done'
30
31
32 if __name__ == '__main__':
33     main()
34

```

通过管道

multiprocessing模块的Pipe方法，返回一个二元组（conn1，conn2），Pipe方法有一个duplex参数，为True时代表管道连接是全双工的，为False时代表管道连接是单方向的，只能由conn2发送到conn1。send和recv方法用于发送与接受消息，如果没有消息可接受，recv阻塞，如果管道关闭，recv会抛出EOFError

```

1  import multiprocessing
2  import os, time, random
3
4  def proc_send(pipe, urls):
5      print 'process {} is read to send urls'.format(os.getpid())
6      for url in urls:
7          pipe.send(url)
8          print 'process {}: send {}'.format(os.getpid(), url)
9          time.sleep(random.random())
10
11 def proc_recv(pipe):
12     print 'process {} is ready to recv urls'.format(os.getpid())
13     while True:
14         print 'process {}: recv {}'.format(os.getpid(), pipe.recv())
15         time.sleep(random.random())
16
17 def main():
18     pipe = multiprocessing.Pipe()
19     process_send = multiprocessing.Process(
20         target=proc_send,

```

```

21     args=(pipe[0], ['url_' + str(i) for i in range(10)])
22     process_recv = multiprocessing.Process(
23         target=proc_recv,
24         args=(pipe[1],)
25     )
26     process_send.start()
27     process_recv.start()
28     process_send.join()
29     process_recv.join()
30     print 'done'
31
32 if __name__ == '__main__':
33     main()

```

分布式多进程

分布式也是一个比较重要的概念，通过将负载高的计算分摊到多台计算机上来提高系统性能。使用python完成分布式计算功能是简单的。需要用到的一个数据结构是队列，联想一下操作系统中的生产者消费者模型，一些进程放入数据，一些进程取出数据。程序开始需要在服务端维护一个网络队列管理器，服务端程序注册操作网络队列的方法，随后使用该方法从网络上获取队列，对该队列的操作，对网络上的其他进程是可见的。队列的put和get方法用于放入取出数据，注意服务端和客户端注册的接口方法需统一。使用multiprocessing子模块managers管理网络队列，其中的BaseManager类是一个基本的管理器，新建类继承该类。使用该类的register方法注册操作队列的方法，随后监听信道。如下例程 server

```

1  #!/usr/bin/env python
2  import Queue
3  from multiprocessing.managers import BaseManager
4
5  # 创建队列实体
6  task_queue = Queue.Queue()
7  result_queue = Queue.Queue()
8
9  class Queuemanager(BaseManager):
10     pass
11
12  # 注册方法
13  print 'register the func'
14  Queuemanager.register('get_task_queue', callable=lambda:task_queue)
15  Queuemanager.register('get_result_queue', callable=lambda:result_queue)
16
17  # 创建manager对象
18  print 'initialing the task manager'
19  manager = Queuemanager(address=('192.168.56.1', 8000), authkey='password')
20
21  # 开始监听
22  manager.start()
23  # 从网络得到队列
24  print 'get the queue from network...'
25  task = manager.get_task_queue()
26  result = manager.get_result_queue()
27  # 向队列中放入数据等待处理
28  print 'put urls to the task queue'

```



```

29 for url in ['ImageUrl_' + str(i) for i in range(10)]:
30     print 'put {} in task'.format(url)
31     task.put(url)
32     # 从队列中取出数据，阻塞等待
33     for i in range(10):
34         print 'result is {}'.format(result.get())
35
36 manager.shutdown()

```

client

```

1     #!/usr/bin/env python
2 from multiprocessing.managers import BaseManager
3 import Queue
4
5
6 class Queuemanager(BaseManager):
7     pass
8
9 Queuemanager.register('get_task_queue')
10 Queuemanager.register('get_result_queue')
11
12 server = '192.168.56.1'
13 port = 8000
14 key = 'password'
15 print 'try to connect to {}'.format(server)
16 manager = Queuemanager(address=(server, port), authkey=key)
17 manager.connect()
18
19 task = manager.get_task_queue()
20 result = manager.get_result_queue()
21
22 while not task.empty():
23     image_url = task.get(True, timeout=10)
24     print 'run task download {}'.format(image_url)
25     result.put(image_url + '----->completed!')
26
27 print 'worker exit!'

```

线程

线程是一个存在于进程中的概念，用于在进程中并行完成不同的工作。线程与进程的不同另做介绍

python中的多线程

threading推荐使用的多线程模块

threading中的模块对象

threading 模块对象	描 述
Thread	表示一个线程的执行的对象
Lock	锁原语对象（跟 thread 模块里的锁对象相同）
RLock	可重入锁对象。使单线程可以再次获得已经获得了的锁（递归锁定）
Condition	条件变量对象能让一个线程停下来，等待其他线程满足了某个“条件”。如，状态的变或值的改变
Event	通用的条件变量。多个线程可以等待某个事件的发生，在事件发生后，所有的线程会被激活
Semaphore	为等待锁的线程提供一个类似“等候室”的结构
BoundedSemaphore	与 Semaphore 类似，只是它不允许超过初始值
Timer	与 Thread 相似，只是它要等待一段时间后才开始运行

threading中的常见方法

activeCount()	当前活动的线程对象的数量
currentThread()	返回当前线程对象
enumerate()	返回当前活动线程的列表
settrace(func) ^a	为所有线程设置一个跟踪函数
setprofile(func) ^a	为所有线程设置一个 profile 函数

Thread类

函 数	描 述
start()	开始线程的执行
run()	定义线程的功能的函数（一般会被子类重写）
join(timeout=None)	程序挂起，直到线程结束；如果给了 timeout，则最多阻塞 timeout 秒
getName()	返回线程的名字
setName(name)	设置线程的名字
isAlive()	布尔标志，表示这个线程是否还在运行中
isDaemon()	返回线程的 daemon 标志
setDaemon(daemonic)	把线程的 daemon 标志设为 daemonic（一定要在调用 start()函数前调用）

初始化一个thread类来创建一个线程，我们可以

- 初始化Thread类，传入我们要运行的函数与参数
- 初始化Thread类，传入可调用对象，比如自定义可调用类
- 创建类继承Thread，覆盖run函数

threading模块实例

直接使用thread类

```
1  #!/usr/bin/env python
2  import threading
3  from time import ctime, sleep
4  secLoop = [6, 4]
5  def loop(sec, i):
6      print 'loop', i, 'start at', ctime()
7      sleep(sec)
```

```

8         print 'loop', i, 'finished at', ctime()
9     def main():
10         nloop = range(len(secLoop))
11         threads = []
12         for i in nloop:
13             threads.append(threading.Thread(target=loop, args=(secLoop[i], i)))
14
15         for i in nloop:
16             threads[i].start()
17
18         for i in nloop:
19             threads[i].join()
20
21     if __name__ == '__main__':
22         main()
23

```

自定义可调用类

```

1     import threading
2     from time import ctime, sleep
3
4     secLoop = [6, 4]
5     def loop(sec, i):
6         print 'loop', i, 'start at', ctime()
7         sleep(sec)
8         print 'loop', i, 'finished at', ctime()
9
10    class ThreadFunc(object):
11        def __init__(self, func, args):
12            self.func = func
13            self.args = args
14
15        def __call__(self):
16            apply(self.func, self.args)
17
18    def main():
19        nloop = range(len(secLoop))
20        threads = []
21        for i in nloop:
22            threads.append(threading.Thread(target=ThreadFunc(loop, (secLoop[i], i))))
23
24        for i in nloop:
25            threads[i].start()
26
27        for i in nloop:
28            threads[i].join()
29
30    if __name__ == '__main__':
31        main()

```

python中的线程池与进程池

concurrent.futures 中的ThreadPoolExecutor封装了线程池实现，初始化是的max_workers指定了线程池的大小，使用submit提交线程，提交的同时线程开始执行，将提交返回的线程句柄放入列表中，使用as_complete方法阻塞等待线程执行的结果，ProcessPoolExecutor接口相同

```
1  from concurrent.futures import ThreadPoolExecutor, as_completed
2  import time
3
4  # 参数times用来模拟网络请求的时间
5  def get_html(times):
6      time.sleep(times)
7      print("get page {}s finished".format(times))
8      return times
9
10 executor = ThreadPoolExecutor(max_workers=2)
11 urls = [3, 2, 4] # 并不是真的url
12 all_task = [executor.submit(get_html, (url)) for url in urls]
13
14 for future in as_completed(all_task):
15     data = future.result()
16     print("in main: get page {}s success".format(data))
```

done () 判断是否结束，result()获取执行结果，wait方法可以等待列表中的线程完成

python网络编程

要说python优雅在何处，与其他语言相比最为明显的，那一定是网络操作了。python可以让我们用最少的语句写出功能强大的程序，网络操作有相当多的开源库可以使用，而不用像其他语言一样，步骤繁琐。接下来从低到高介绍python网络编程

socket网络编程接口

基本概念

可以这样来理解socket，完成一段网络通信需要五个元素，协议族，协议类型，协议，目标ip地址，目标端口号，这是由TCP/IP网络结构决定的。socket作为一种数据结构，将以上五个信息组合起来，称为套接字。根据面向连接与非连接，套接字类型分为Datagram与Stream两种类型。Datagram套接字使用UDP协议，向目标地址发送数据包，不保证可靠送达，Stream使用TCP协议，是可靠的、具有差错与流量控制的传输协议，也就是说，stream类型的套接字传输信息，具有较好的可靠性，而Datagram类型的套接字具有更高的实时性。

初始化流式套接字的步骤

服务端

- 使用socket函数创建套接字
- 绑定到指定的ip地址
- 监听客户端连接

- 获取客户端连接后，使用返回的套接字收发数据
- 通信结束，关闭套接字

客户端

- 创建套接字
- 连接指定ip地址
- 连接成功后收发数据
- 关闭套接字

常用套接字函数

函 数	描 述
服务器端套接字函数	
<code>s.bind()</code>	绑定地址（主机名，端口号对）到套接字
<code>s.listen()</code>	开始 TCP 监听
<code>s.accept()</code>	被动接受 TCP 客户端连接，（阻塞式）等待连接的到来
客户端套接字函数	
<code>s.connect()</code>	主动初始化 TCP 服务器连接
<code>s.connect_ex()</code>	<code>connect()</code> 函数的扩展版本，出错时返回出错码，而不是抛出异常
公共用途的套接字函数	
<code>s.recv()</code>	接收 TCP 数据
<code>s.send()</code>	发送 TCP 数据
<code>s.sendall()</code>	完整发送 TCP 数据
<code>s.recvfrom()</code>	接收 UDP 数据
<code>s.sendto()</code>	发送 UDP 数据
<code>s.getpeername()</code>	连接到当前套接字的远端的地址（TCP 连接）
<code>s.getsockname()</code>	当前套接字的地址
<code>s.getsockopt()</code>	返回指定套接字的参数
<code>s.setsockopt()</code>	设置指定套接字的参数
<code>s.close()</code>	关闭套接字
面向模块的套接字函数	
<code>s.setblocking()</code>	设置套接字的阻塞与非阻塞模式
<code>s.settimeout()^a</code>	设置阻塞套接字操作的超时时间
<code>s.gettimeout()^a</code>	得到阻塞套接字操作的超时时间
面向文件的套接字函数	
<code>s.fileno()</code>	套接字的文件描述符
<code>s.makefile()</code>	创建一个与该套接字关联的文件对象

例程

时间戳服务器

```
1  #!/usr/bin/env python
2  import socket
3  from time import ctime
4
5
6  host='127.0.0.1'
7  port=8000
8  bufsize=1024
9  addr=(host, port)
10
11 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12 sock.bind(addr)
13 sock.listen(5)
14 while True:
15     print 'waiting for connection...'
16     clisock, cliaddr = sock.accept()
17     print 'connected from', cliaddr
18     print 'peername:', clisock.getpeername()
19     data = clisock.recv(bufsize)
20     if not data:
21         break
22     clisock.send('[%s] %s' % (ctime(), data))
23     clisock.close()
24 sock.close()
```

客户端

```
1  #!/usr/bin/env python
2  from time import ctime
3  import socket
4
5
6  host='127.0.0.1'
7  port=8000
8  addr=(host, port)
9  bufsize=1024
10
11 while True:
12     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     sock.connect(addr)
14     data = raw_input(u'>')
15     if not data:
16         break
17     sock.send(data)
18     data = sock.recv(bufsize)
19     print data
20     #!/usr/bin/env python
21 from time import ctime
22 import socket
23
```

```

24 host='127.0.0.1'
25 port=8000
26 addr=(host, port)
27 bufsize=1024
28
29
30 while True:
31     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
32     sock.connect(addr)
33     data = raw_input(u'>')
34     if not data:
35         break
36     sock.send(data)
37     data = sock.recv(bufsize)
38     print data
39
40     sock.close()
41     sock.close()

```

异步socket

使用select，poll和epoll，操作接口统一，一般使用较为新的epoll，

一个使用epoll的例子如下

```

1  import argparse
2  import logging
3  import os, sys
4  import select
5  import socket
6  import signal
7  import queue
8  logging.basicConfig(
9      level=logging.DEBUG, # 定义输出到文件的log级别,
10     format='%(asctime)s [%(filename)s %(lineno)d] %(levelname)s %(message)s', # 定义输出
log的格式
11     datefmt='%Y-%m-%d %A %H:%M:%S'
12 )
13
14
15 def parse_command_line(description: str):
16     argparser = argparse.ArgumentParser(description=description)
17     argparser.add_argument('--host', action='store', default='localhost',
18                             required=False, dest='host', help='the host server will listen
to')
19     argparser.add_argument('--port', action='store', default=1070, type=int,
20                             required=False, metavar='port', help="the port server will bind
to")
21     args = argparser.parse_args()
22     return (args.host, args.port)
23
24

```

```

25 # POLLIN    There is data to read
26 # POLLPRI   There is urgent data to read
27 # POLLOUT   Ready for output: writing will not block
28 # POLLERR   Error condition of some sort
29 # POLLHUP   Hung up
30 # POLLRDHUP Stream socket peer closed connection, or shut down writing half of connection
31 # POLLNVAL  Invalid request: descriptor not open
32
33 READ_ONLY = (select.EPOLLIN | select.EPOLLPRI | select.EPOLLHUP | select.EPOLLERR)
34 READ_WRITE = (READ_ONLY | select.EPOLLOUT)
35
36
37 class Server:
38
39     def __init__(self, server_addr: tuple):
40         self.server_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
41         self.server_sock.setblocking(False)
42         self.server_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
43         self.server_sock.bind(server_addr)
44         self.server_sock.listen(10)
45         # self.inputs = [self.server_sock]
46         self.fd_2_socks = {self.server_sock.fileno(): self.server_sock}
47
48         self.outputs = []
49         self.message_queues = {}
50         self.poller = select.epoll()
51         self.poller.register(self.server_sock.fileno(), READ_ONLY)
52         logging.info('server socket initialized!')
53         signal.signal(signal.SIGINT, self.destory_server)
54         signal.signal(signal.SIGTERM, self.destory_server)
55
56     def accept_connection(self):
57         connection, client_addr = self.server_sock.accept()
58         logging.info('connection from {}'.format(client_addr))
59         connection.setblocking(False)
60         self.fd_2_socks[connection.fileno()] = connection
61         self.poller.register(connection.fileno(), READ_ONLY)
62         self.message_queues[connection] = queue.Queue()
63
64     def read_msg(self, conn: socket):
65         data = conn.recv(1024)
66         if data:
67
68             logging.info('received msg({}): {}'.format(conn.getpeername(),
data.decode('utf-8')))
69             self.message_queues[conn].put(data)
70             self.poller.modify(conn.fileno(), READ_WRITE)
71
72         else:
73             logging.info('closing socket: {}'.format(conn.getpeername()))
74             self.poller.unregister(conn.fileno())
75             conn.close()
76             del self.message_queues[conn]

```



```

77
78     def send_msg(self, conn: socket):
79         try:
80             msg = self.message_queues[conn].get_nowait()
81         except queue.Empty:
82             logging.error('{} queue empty!'.format(conn.getpeername()))
83             self.poller.modify(conn.fileno(), READ_ONLY)
84         else:
85             logging.info('sending msg {} to {}'.format(msg, conn.getpeername()))
86             conn.send(msg)
87
88     def exception_handler(self, conn: socket):
89         logging.error('exception occured on socket: {}'.format(conn.getpeername()))
90
91         self.poller.unregister(conn.fileno())
92         conn.close()
93         del self.message_queues[conn]
94
95     def serv_forever(self):
96         while True:
97             logging.info('waiting for next event!')
98             # 对epoll对象,poll参数单位是秒,对poll对象,参数单位是毫秒
99             events = self.poller.poll(-1)
100             for fd, flag in events:
101                 sock = self.fd_2_socks[fd]
102                 if flag & (select.EPOLLIN | select.EPOLLPRI):
103                     if sock is self.server_sock:
104                         self.accept_connection()
105                     else:
106                         self.read_msg(sock)
107                 elif flag & select.EPOLLOUT:
108                     self.send_msg(sock)
109                 elif flag & (select.EPOLLERR | select.EPOLLHUP):
110                     self.poller.unregister(fd)
111                     sock.close()
112                     del self.message_queues[sock]
113
114     def destory_server(self, *args):
115         logging.info('exiting.....')
116         for arg in args:
117             print(arg)
118         sys.exit(0)
119
120
121 if __name__ == '__main__':
122     server_address = parse_command_line('asynchro socket server!')
123     server = Server(server_address)
124     server.serv_forever()
125

```

epoll会新建epoll对象，其poll方法会在已经注册的文件描述符上等待注册过的事件，当有事件发生或时间达到时，返回描述符与事件组成的元组。

网络http操作

使用urllib、urllib2、requests

一般来说，进行网络操作时可以使用以上三个模块任意一种，但是就功能上来说，requests较为强大，对底层模块进行了封装，使用更为简便，我们重点来讨论一下requestshttp请求有几种类型，最常用的是get与post，get请求向指定的资源请求数据，GET 请求可被缓存，GET 请求保留在浏览器历史记录中，GET 请求可被收藏为书签，GET 请求不应在处理敏感数据时使用，GET 请求有长度限制，GET 请求只应当用于取回数据。post请求向指定的资源提交数据，POST 请求不会被缓存，POST 请求不会保留在浏览器历史记录中，POST 不能被收藏为书签，POST 请求对数据长度没有要求。还有一个区别是，使用get请求时，参数直接显示在url中，安全性不高，而post请求将参数包含在请求体中。

使用requests

发出get与post请求

```
1 In [8]: res = requests.get('http://www.baidu.com')
2 In [9]: res2 = requests.post('http://www.baidu.com')
```

设置请求参数

通过params参数设置请求参数，需要传进一个字典，且值为none的键不会出现在url中

```
1 In [10]: para = {'username': 'xiaozi', 'password': 'testpass'}
2 In [11]: res3 = requests.get('http://www.baidu.com', params=para)
3 In [12]: res3.url
4 Out[12]: u'http://www.baidu.com/?username=xiaozi&password=testpass'
```

为post请求传递数据

使用data参数

```
1 In [3]: data = {'key': 'value'}
2 In [4]: res = requests.post('http://httpbin.org/post', data=data)
```

```
1 In [38]: payload = (('key1', 'value1'), ('key1', 'value2'))
2 In [39]: r = requests.post('http://httpbin.org/post', data=payload)
```

获取响应内容

字符串内容

```
1 In [5]: res.text
2 Out[5]: u'{"args": {}, "data": "", "files": {}, "form": {"key": "value"}, "headers":
{"Accept": "*/*", "Accept-Encoding": "gzip, deflate", "Connection": "close", "Content-
Length": "9", "Content-Type": "application/x-www-form-urlencoded", "Host": "httpbin.org", "User-
Agent": "python-
requests/2.18.4"}, "json": null, "origin": "113.140.11.6", "url": "http://httpbin.org/post"}\n'
```

查看与设置内容编码

```
1 In [8]: res.encoding
2 In [9]: res.encoding='utf-8'
```

使用content属性可以查看返回内容的字节内容，python会自动为我们解码gzip压缩的数据

```
1 In [15]: res.content
2 Out[15]: b'{"args": {}, "data": "", "files": {}, "form": {"key": "value"}, "headers":
{"Accept": "*/*", "Accept-Encoding": "gzip, deflate", "Connection": "close", "Content-
Length": "9", "Content-Type": "application/x-www-form-urlencoded", "Host": "httpbin.org", "User-
Agent": "python-
requests/2.18.4"}, "json": null, "origin": "113.140.11.6", "url": "http://httpbin.org/post"}\n'
```

json对象

```
1 In [16]: res.json()
2 Out[16]:
3 {'args': {},
4  'data': u'',
5  'files': {},
6  'form': {'key': u'value'},
7  'headers': {'Accept': u'*/*',
8  'Accept-Encoding': u'gzip, deflate',
9  'Connection': u'close',
10 'Content-Length': u'9',
11 'Content-Type': u'application/x-www-form-urlencoded',
12 'Host': u'httpbin.org',
13 'User-Agent': u'python-requests/2.18.4'},
14 'json': None,
15 'origin': u'113.140.11.6',
16 'url': u'http://httpbin.org/post'}
```

查看返回状态码

```
1 res.status_code
```

原始数据

```

1 In [31]: res = requests.get('https://api.github.com/events', stream=True)
2
3 In [32]: with open('temp.txt', 'wb') as fd:
4     ...:     for chunk in res.iter_content(512):
5     ...:         fd.write(chunk)

```

查看返回的http头部

```

1 In [44]: r.headers
2 Out[44]: {'Content-Length': '351', 'Via': '1.1 vegur', 'Server': 'unicorn/19.8.1',
'Connection': 'keep-alive', 'Access-Control-Allow-Credentials': 'true', 'Date': 'Thu, 24 May
2018 12:13:11 GMT', 'Access-Control-Allow-Origin': '*', 'Content-Type': 'application/json'}

```

使用headers参数定制请求头

```

1 In [33]: headers = {'user-agent': 'myapp'}
2 In [34]: res = requests.get('http://www.baidu.com', headers=headers)

```

返回对象的cookies属性保存了会话cookie，当需要重用cookies时，将cookie对象作为参数传递给请求的cookies参数

```

1 res = requests.get(url, cookies=res.cookies)

```

使用cookie模拟登陆豆瓣网站

```

1 cookies = {}
2
3 In [57]: raw_cookies='bid=-aa9P19eMB0;
_pk_ref.100001.8cb4=%5B%22%22%2C%22%22%2C1527208498%2C%22http
...: s%3A%2F%2Fwww.baidu.com%2Flink%3Furl%3DFpVIns4IT_ak-
F56yaoXBUJPtBtueoUd3nVtc4NWZV4QB08sdnc
...:
vDJTEYLuefv5%26ck%3D1583.5.192.64.190.324.478.646%26shh%3Dwww.baidu.com%26wd%3D%26eqid%3D89
...: b868ea00032e0c000000035b06b157%22%5D;
_pk_id.100001.8cb4=83b586dcea10d0e5.1519446169.2.1527
...: 209320.1519446169.; __utma=30149280.2141300691.1527208503.1527208503.1527208503.1;
__utmz=3
...: 0149280.1527208503.1.1.utmcsr=baidu|utmccn=(organic)|utmcmd=organic; ll="118371";
_pk_ses.1
...: 00001.8cb4=*; __utmb=30149280.9.10.1527208503; __utmc=30149280; ps=y;
push_noty_num=0; push
...: _doumail_num=0; __utmv=30149280.13784; ap=1; dbcl2="137847261:LmaeCQEH7Eo"; ck=x-
vi; __utmt
...: =1'
12
13 In [58]: for line in raw_cookies.split(';'):
14     ...:     key,value = line.split('=', 1)
15     ...:     cookies[key] = value

```

```
16 In [64]: url = 'https://www.douban.com/people/xiaozhiAXX/'
17
18 In [65]: res = requests.get(url,cookies=cookies)
19
20 In [66]: res.status_code
21 Out[66]: 200
22
```

使用proxies参数设置代理,访问google

```
1 In [1]: import requests
2
3 In [2]: proxy = {'http':'http://127.0.0.1:8118','https':'https://127.0.0.1:8118'}
4
5 In [3]: res = requests.get('https://www.google.com', proxies=proxy)
6
7 In [4]: res.status_code
8 Out[4]: 200
```

网络客户端

电子邮件

python发送电子邮件时, 使用标准库中的smtplib和email, smtplib中有一个SMTP类, 需要发送邮件时, 初始化该类返回smtpserver对象, 使用login登陆MUA, 使用sendmail方法发送邮件, 邮件的正文用email.mime.text.MIMEText对象进行描述

简单电子邮件发送程序

```
1 from email.mime.text import MIMEText
2 msg = MIMEText('hello message','plain', 'utf-8')
3 from_addr = 'yourPhone@163.com'
4 to_addr = 'yourQQ@qq.com'
5 sub_msg = 'hello'
6 smtp_server = 'smtp.163.com'
7 import smtplib
8 # 初始化smtp对象, 传入服务器地址与端口号
9 server = smtplib.SMTP(smtp_server,25)
10 # 设置调试模式可以让我们看到发送邮件过程中的信息
11 server.set_debuglevel(1)
12 # 登陆MUA, 使用账户与授权码登陆
13 server.login(from_addr, 'yourpassword')
14 msg['From'] = from_addr
15 msg['To'] = to_addr
16 msg['Subject'] = 'important message'
17 server.sendmail(from_addr, [to_addr], msg.as_string())
```

邮件被放入垃圾邮件中, 如下

important message ☆

发件人: **18392136027** <18392136027@163.com> 

时 间: 2018年5月24日(星期四) 晚上9:56

收件人: yangzhile <1786614260@qq.com>

这是一封垃圾箱中的邮件。请勿轻信中奖、汇款等虚假信息, 勿轻易拨打陌生

important very and miss you

发送带附件的电子邮件

```
1  from email.mime.text import MIMEText
2  from smtplib import SMTP
3  from email.mime.multipart import MIMEMultipart
4
5
6  from_addr = '18392136027@163.com'
7  to_addr = '1786614260@qq.com'
8  smtp_server = 'smtp.163.com'
9  smtp_port = 25
10 subject_msg = 'subject'
11
12 mul_msg = MIMEMultipart()
13 mul_msg['From'] = from_addr
14 mul_msg['To'] = to_addr
15 mul_msg['Subject'] = subject_msg
16
17 msg = MIMEText('\n\rimportant message\n\r', 'plain', 'utf-8')
18 mul_msg.attach(msg)
19
20 att1 = MIMEText(open('program.txt','rb').read(), 'base64', 'utf-8')
21 att1['Content-Type'] = 'application/octet-stream'
22 att1["Content-Disposition"] = 'attachment;filename="program.txt"'
23 mul_msg.attach(att1)
24
25 smtp = SMTP(smtp_server, smtp_port)
26 smtp.login(from_addr, 'youpass')
27 smtp.set_debuglevel(1)
28 smtp.sendmail(from_addr, to_addr, mul_msg.as_string())
29 smtp.close()
```


subject ☆

发件人: **18392136027** <18392136027@163.com> 

时 间: 2018年5月25日(星期五) 上午9:24

收件人: yangzhile <1786614260@qq.com>

附 件: 1 个 ( program.txt)

这不是腾讯公司的官方邮件②。 请勿轻信密保、汇款、中奖信息，勿轻易拨打陌生电话。  举报垃圾邮件

important message

 附件(1 个)

普通附件



program.txt (105字节)

预览 下载 收藏 转存▼

使用第三方开源库yagmail发送电子邮件

```
1 import yagmail
2 yag = yagmail.SMTP(user='youQQ@qq.com', password='you pass', host='smtp.qq.com', port=25)
3 contents = ['import message', 'program.txt']
4 yag.send(to='dest', subject='subject', contents=contents)
```

使用pop3协议用网易邮箱发送邮件时，容易被网易识别为垃圾邮件，可以使用qq邮箱

正则表达式

特殊字符

- 元字符

记 号	说 明	正则表达式样例
<i>literal</i>	匹配字符串的值	foo
<i>re1 re2</i>	匹配正则表达式 re1 或 re2	foo bar
.	匹配任何字符（换行符除外）	b.b
^	匹配字符串的开始	^Dear
\$	匹配字符串的结尾	/bin/*sh\$
*	匹配前面出现的正则表达式零次或多次	[A-Za-z0-9]*
+	匹配前面出现的正则表达式一次或多次	[a-z]+\com
?	匹配前面出现的正则表达式零次或一次	goo?
{N}	匹配前面出现的正则表达式 N	[0-9]{3}
{M,N}	匹配重复出现 M 次到 N 次的正则表达式	[0-9]{5,9}
[...]	匹配字符组里出现的任意一个字符	[aeiou]
[..x-y..]	匹配从字符 x 到 y 中的任意一个字符	[0-9], [A-Za-z]
[^...]	不匹配此字符集中出现的任何一个字符，包括某一范围的字符（如果在此字符集中出现）	[^aeiou],[^A-Za-z0-9_]
(* + ? { })?	用于上面出现的任何“非贪婪”。版本重复匹配次数符号	.*[a-z]
(...)	匹配封闭括号中正则表达式（RE），并保存为子组	([0-9]{3})?, f (oo u)bar

- 特殊字符

\d	匹配任何数字，和[0-9]一样（\D 是\d 的反义：任何非数字字符）	data\d+.txt
----	-------------------------------------	-------------

www.TopSage.com

444 第 15 章 正则表达式

续表

记 号	说 明	正则表达式样例
\w	匹配任何数字字母字符，和[A-Za-z0-9_]相同（\W 是\w 的反义）	[A-Za-z_]\w+
\s	匹配任何空白符，和[\n\t\r\f]相同，（\S 是\s 的反义）	of\sthe
\b	匹配单词边界（\B 是\b 的反义）	\bThe\b
\nn	匹配已保存的子组（请参考上面的正则表达式符号：(...)）	price:\16
\c	逐一匹配特殊字符 c（即，取消它的特殊含义，按字面匹配）	\\, \\, *
\A (Z)	匹配字符串的起始（结束）	\ADear

- 使用|来分隔多个正则符号
- 使用.来匹配单个字符
f.o -> f和o之间有任何单个字符
- 开头^结尾\$边界\b\B
^test 匹配test开头的字符串
test\$ 匹配test结尾的字符串
^\$ 匹配空行
\btest 匹配以test开始的字符串

\btest\b 匹配单词test

\Btest匹配包含test但不以test开头的字符串

- 使用[]匹配字符集合

[abc]

[a-z]

[^a-z]

正则表达式中的分组

重复单个字符只需在该单个字符后加上需要重复的次数 (*?{}), 当重复多个字符时, 将该多个字符用 () 括起来, 后面跟上重复次数即可, 这里我们看一个匹配ip地址的范例, 从ifconfig输出结果匹配到ip地址, 在vim中匹配如下, 由于 () {} 在shell中是特殊字符, 所以使用 \ 来转义 \(\d\{1,3\}\.\.\)\{3\}\d\{1,3\}

```
1 enp2s0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
2       ether 34:17:eb:81:67:5e txqueuelen 1000 (Ethernet)
3       RX packets 0 bytes 0 (0.0 B)
4       RX errors 0 dropped 0 overruns 0 frame 0
5       TX packets 0 bytes 0 (0.0 B)
6       TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
7
8 lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
9       inet 127.0.0.1 netmask 255.0.0.0
10      inet6 ::1 prefixlen 128 scopeid 0x10<host>
11      loop txqueuelen 1000 (Local Loopback)
12      RX packets 11725 bytes 30036896 (28.6 MiB)
13      RX errors 0 dropped 0 overruns 0 frame 0
14      TX packets 11725 bytes 30036896 (28.6 MiB)
15      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
16
17 virbr0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
18       inet 192.168.124.1 netmask 255.255.255.0 broadcast 192.168.124.255
19       ether 52:54:00:5c:2b:78 txqueuelen 1000 (Ethernet)
20       RX packets 0 bytes 0 (0.0 B)
21       RX errors 0 dropped 0 overruns 0 frame 0
22       TX packets 0 bytes 0 (0.0 B)
23
24 \(\d\{1,3\}\.\.\)\{3\}\d\{1,3\}
```

使用在线工具测试

在线正则表达式测试

```

RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 12881 bytes 20181602 (28.7 MiB)

```

正则表达式

☒ 全局搜索 ☐ 忽略大小写

▼测试匹配

匹配结果:

```

127.0.0.1
255.0.0.0
192.168.124.1
255.255.255.0
192.168.124.255
192.168.1.106
255.255.255.0
192.168.1.255

```

由于ip地址限制每个域不能超过255，使用 `((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)`

在线正则表达式测试

```
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 12881 bytes 20181602 (28.7 MiB)
```

正则表达式 `((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)` ☒ 全局搜索 ☐ 忽略大小写

测试匹配

匹配结果:

```
127.0.0.1
255.0.0.0
192.168.124.1
255.255.255.0
192.168.124.255
192.168.1.106
255.255.255.0
192.168.1.255
```

后向引用

默认情况下，每个 () 括起来的分组会被分配一个从1开始的组号，特殊的分组0代表整个正则表达式 使用 () 匹配一个子表达式后，匹配这个子表达式的文本，可以在后续正则中继续使用，\1表示分组一匹配的文本

在线正则表达式测试

```
enp2s0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
  ether 34:17:eb:81:67:5e txqueuelen 1000 (Ethernet)
  RX packets 0 bytes 0 (0.0 B)
  RX errors 0 dropped 0 overruns 0 frame 0
  TX packets 0 bytes 0 (0.0 B)
  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
  inet 127.0.0.1 netmask 255.0.0.0
```

正则表达式 ☒ 全局搜索 ☐ 忽略大小写

▼ 测试匹配

匹配结果:

共找到 4 处匹配:

0.0.
0.0.
255.255.
255.255.

常用的分组语法如下

分类	代码/语法	说明
捕获	(exp)	匹配exp,并捕获文本到自动命名的组里
	(?<name>exp)	匹配exp,并捕获文本到名称为name的组里，也可以写成(?'name'exp)
	(?:exp)	匹配exp,不捕获匹配的文本，也不给此分组分配组号
零宽断言	(?=exp)	匹配exp前面的位置
	(?<=exp)	匹配exp后面的位置
	(?!exp)	匹配后面跟的不是exp的位置
	(?<!exp)	匹配前面不是exp的位置
注释	(?#comment)	这种类型的分组不对正则表达式的处理产生任何影响，用于提供注释让人阅读

使

用命名分组 (\k)

```
enp2s0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    ether 34:17:eb:81:67:5e txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
```

正则表达式

☒ 全局搜索 ☐ 忽略大小写

▼测试匹配

匹配结果:

共找到 4 处匹配:
0.0.
0.0.
255.255.
255.255.

零宽断言

有时候我们想匹配某个模式之前或之后的部分，匹配的部分必须满足特定条件。

- (?=exp)该位置后面可以匹配exp，返回除了exp的部分

```
\b\w+(?=ing\b)
```

```
TX packets 12881 bytes 30181603 (28.7 MiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 12881 bytes 30181603 (28.7 MiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.124.1 netmask 255.255.255.0 broadcast 192.168.124.255
    ether 52:54:00:5c:2b:78 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
```

正则表达式

☒ 全局搜索 ☐ 忽略大小写

▼测试匹配

匹配结果:

共找到 2 处匹配:
broad
broad

- (?<=exp)该位置前面可以匹配exp，同样返回除了exp的部分

```
(?<=broad)\w+\b
```

```
RX packets 12881 bytes 30181603 (28.7 MiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 12881 bytes 30181603 (28.7 MiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500
    inet 192.168.124.1 netmask 255.255.255.0 broadcast 192.168.124.255
    ether 52:54:00:5c:2b:78 txqueuelen 1000 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    TX packets 0 bytes 0 (0.0 B)
```

正则表达式

☒ 全局搜索 ☐ 忽略大小写

▼ 测试匹配

匹配结果:

共找到 2 处匹配:

cast
cast

负向零宽断言

断言某位置前或后不能出现某些字符，我们也可以使用反义，但是问题在于使用反义时，`[^]`总会占用一个字符空间，可以匹配到单词分隔符。

- `(?!exp)`断言此位置后不匹配`exp`

124 1254356

正则表达式

☒ 全局搜索 ☐ 忽略大小写

▼ 测试匹配

匹配结果:

共找到 2 处匹配:

124
356

- `(?<!exp>)`断言此位置前不匹配exp

1234587 asd1254356

正则表达式 `(?<![a-z])\d{7}`

☒ 全局搜索 ☐ 忽略大小写

▼ 测试匹配

匹配结果:

共找到 1 处匹配:
1234587

贪婪匹配与懒惰匹配

当我们的正则表达式有表示重复次数的元字符时，默认情况下总是会自动匹配最长，比如ababab,用`a.*b`来匹配，会匹配到整个字符串，这就是贪婪匹配，容易理解懒惰匹配就是匹配最短。通过在元字符后加上`?`来将贪婪匹配变为懒惰匹配。如下

abababab

正则表达式 `a.*?b`

☒ 全局搜索 ☐ 忽略大小写

▼ 测试匹配

匹配结果:

共找到 4 处匹配:
ab
ab
ab
ab

abababab

正则表达式

a.*b

☒ 全局搜索 ☐ 忽略大小写

▼ 测试匹配

匹配结果:

共找到 1 处匹配:
abababab

python中的正则表达式re

核心函数与表达式

<code>compile(pattern,flags=0)</code>	对正则表达式模式 <code>pattern</code> 进行编译, <code>flags</code> 是可选标识符, 并返回一个 <code>regex</code> 对象
re 模块的函数和 regex 对象的方法	
<code>match(pattern,string,flags=0)</code>	尝试用正则表达式模式 <code>pattern</code> 匹配字符串 <code>string</code> , <code>flags</code> 是可选标识符, 如果匹配成功, 则返回一个匹配对象; 否则返回 <code>None</code>
<code>search(pattern,string,flags=0)</code>	在字符串 <code>string</code> 中搜索正则表达式模式 <code>pattern</code> 的第一次出现, <code>flags</code> 是可选标识符, 如果匹配成功, 则返回一个匹配对象; 否则返回 <code>None</code>
<code>findall(pattern,string[,flags])^a</code>	在字符串 <code>string</code> 中搜索正则表达式模式 <code>pattern</code> 的所有 (非重复) 出现; 返回一个匹配对象的列表
<code>finditer(pattern,string[,flags])^b</code>	和 <code>findall()</code> 相同, 但返回的不是列表而是迭代器; 对于每个匹配, 该迭代器返回一个匹配对象
<code>split(pattern,string,max=0)</code>	根据正则表达式 <code>pattern</code> 中的分隔符把字符串 <code>string</code> 分割为一个列表, 返回成功匹配的列表, 最多分割 <code>max</code> 次 (默认是分割所有匹配的地方)
<code>sub(pattern,repl,string,max=0)</code>	把字符串 <code>string</code> 中所有匹配正则表达式 <code>pattern</code> 的地方替换成字符串 <code>repl</code> , 如果 <code>max</code> 的值没有给出, 则对所有匹配的地方进行替换 (另外, 请参考 <code>subn()</code> , 它还会返回一个表示替换次数的数值)
匹配对象的方法	
<code>group(num=0)</code>	返回全部匹配对象 (或指定编号是 <code>num</code> 的子组)
<code>groups()</code>	返回一个包含全部匹配的子组的元组 (如果没有成功匹配, 就返回一个空元组)

- `compile()`对给出的`pattern`进行编译, 加快后续处理的速度, 该过程不是必须的, `search`, `match`函数可以直接使用字符串来进行匹配, 但是使用编译过后的字节码对象可以加快处理的速度
- 匹配对象的`group`和`groups`方法 `re`模块的`match`和`search`方法会返回一个匹配对象, 该对象保存了模式匹配的结果, 可以使用该对象的`groups`和`group`方法获取匹配的结果.`group`方法返回所有匹配对象或根据要求返回特定子组匹配的对象

- `match(pattern,string)`从string开头匹配pattern

```
In [4]: m=re.match('a.*?b', 'abcabab')

In [5]: m.group()
Out[5]: 'ab'

In [6]: m.group?
```

如果开头不匹配，返回

None

```
In [7]: m=re.match('a.*?b', 'cabcabab')

In [8]: m.group()

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-8-b9a6738fa293> in <module>()
----> 1 m.group()

AttributeError: 'NoneType' object has no attribute 'group'

In [9]: m is None
Out[9]: True
```

- `search(pattern,string)`从string中搜索pattern第一次出现的位置

```
In [10]: m=re.search('a.*?b', 'cabcabab')

In [11]: m.group()
Out[11]: 'ab'
```

- `findall(pattern, string)`从string中查找所有匹配，返回匹配字符串列表

```
In [17]: m= re.findall('a.*?b', 'cabcabab')

In [18]: m
Out[18]: ['ab', 'ab', 'ab']
```

- `sub(pattern, repl, string)`将匹配pattern的部分替换为repl

```
In [29]: re.sub('x\w+', 'test', info)
Out[29]: 'test hao test'

In [30]:
```

- `re.split(pattern,string)`使用pattern指定的字符分隔string，可以指定多个字符

```
In [2]: data = 'aa,bb.cc s'

In [3]: re.split('[,|. ]',data)
Out[3]: ['aa', 'bb', 'cc', 's']
```

- 在匹配时，通过指定flag参数为`re.I`可以忽略大小写匹配

```
1 In [8]: m = re.compile('a', re.I)
2
3 In [9]: m.findall('abcAAbbc')
4 Out[9]: ['a', 'A', 'A']
```

