第1章

Web搜索引擎开发

Elasticsearch 是一个分布式和高可用的搜索引擎。本章介绍如何使用 Python 语言和 Elasticsearch 实现 Web 搜索引擎。

1.1 准备工作环境

首先要准备一个 Python 的开发环境。当前可以使用 Python 3.9 版本。Python 3.9 可以从 Python 的官方网站 https://www.python.org/下载得到。使用默认方式安装即可。

在 Windows 下安装 Python 以后，在控制台输入 python 命令进入交互式环境。

```
C:\Users\Administrator>python
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在 Windows 操作系统下，利用如下命令检查 Python 3 是否已经正确安装，及其版本号：

```
>python3 -V
Python 3.9.6
```

检查 Python 3 所在的路径：

```
>where python3
C:\Users\Administrator\AppData\Local\Microsoft\WindowsApps\python3.exe
```

可以准备一个用于编写代码的集成开发环境。例如，可以使用 PyCharm 或者 Visual Studio。也可以使用 Notepad++这样的文本编辑器写 Python 代码。

1.2 Linux 操作系统基础

很多 Web 应用的后台运行在 Linux 操作系统中。Linux 来源于 UNIX，是 UNIX 操作系统的开放源代码实现。Linux 通过 SSH 客户端软件连接到远程的 Linux 服务器。SSH 服务器通常作为大多数 Linux 发行版上易于安装的软件包提供。可以尝试使用 ssh localhost 命令来测试它是否正在运行。

如果有现成的 Linux 服务器可用，可以使用支持 SSH（Secure Shell，安全外壳）协议的终端仿真程序 SecureCRT 连接到远程 Linux 服务器。因为它可以保存登录密码，所以使用比较方便。除了 SecureCRT，还可以使用开源软件 PuTTY（http://www.chiark.greenend.org.uk/~sgtatham/putty），或者保存登录密码的 KiTTY（https://www.fosshub.com/KiTTY.html）及 Xshell。如果用 root 账户登录，则终端提示符是#；否则，终端提示符是$。

也可以在 Windows 下安装 Cygwin，使用它来练习 Linux 常用命令。

小袋鼠在袋鼠妈妈的袋子里长大。使用 VMware，Linux 可以运行在 Windows 系统下。VMware 让 Linux 运行在虚拟机中，而且不会破坏原来的 Windows 操作系统。首先要准备好 VMware，当然仍然需要 Linux 光盘文件。

就好像华山派有剑宗和气宗，Linux 也有很多种版本，如 RedHat、CentOS、Ubuntu 及 SUSE。这里介绍 Ubuntu（https://www.ubuntu.com）和 CentOS（http://www.centos.org/）。

操作系统中可能会安装好几个版本的 JDK。在 Linux 中，为了切换 JDK 版本，只需要修改/etc/alternatives 中的符号链接指向。

在 Ubuntu 中，如果需要安装软件，可以下载 deb 安装包，然后使用 dpkg 命令安装。但一个软件包可能依赖其他的软件包。为了安装一个软件可能需要下载其他的好几个它所依赖的软件包。

为了简化安装操作，可以使用高级包装工具（Advanced Packaging Tool，APT）。APT 会自动计算出程序之间的相互关联性，并且计算出完成软件包的安装需要哪些步骤。这样在安装软件时，不会再被那些关联性问题所困扰。

在/etc/apt/sources.list 文件中指示了包的来源存储库。包的来源可以是 CD 或 DVD、硬盘上的目录、HTTP 或 FTP 服务器上的目录。请求的数据包位于服务器或本地硬盘上，它将自动下载并安装。APT 主要关注采购包、包的可用版本的比较，以及包档案的管理。实际上，可以通过浏览器浏览在 HTTP 或 FTP 服务器上的存储库。

如果需要修改/etc/apt/sources.list 文件，可以先备份这个文件：

```
#sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak
```

如果这一步出现

```
sudo: unable to resolve host t-000004
```

这样的错误，则可以考虑执行如下命令修改/etc/hosts 文件的内容：

```
#echo $(hostname -I | cut -d\  -f1) $(hostname) | sudo tee -a /etc/hosts
```

如果安装过程中出现“E: Could not get lock /var/lib/dpkg/lock”这样的错误，则可以尝试使用如下命令修复：

```
#sudo fuser -cuk /var/lib/dpkg/lock
#sudo rm -f /var/lib/dpkg/lock
```

在 CentOS 中，如果需要安装软件，可以下载 RPM 安装包，然后使用 RPM 安装。例如，下载 Elasticsearch 软件的安装包 elasticsearch-6.6.0.rpm：

```
#wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.6.0.rpm
```

使用如下命令安装：

```
#rpm -ivh elasticsearch-6.6.0.rpm
```

但有些操作系统对应的 RPM 安装包找起来比较麻烦。一个软件包可能依赖其他的软件包。为了安装一个软件可能需要下载其他的好几个它所依赖的软件包。

为了简化安装操作，可以使用黄狗升级管理器（Yellow dog Updater，Modified），一般简称 yum。yum 会自动计算出程序之间的相互关联性，并且计算出完成软件包的安装需要哪些步骤。这样在安装软件时，不会再被那些关联性问题所困扰。

yum 软件包管理器自动从网络下载并安装软件。yum 有点类似 360 软件管家，但是不会有商业倾向的推销软件。例如，安装支持 wget 命令的软件：

```
#yum install wget
```

为了方便在服务器端编写 Shell 脚本，可以采用 Micro（https://github.com/zyedidia/micro）这样的终端文本编辑器。

在 Linux 上，可以通过 snap 安装 micro：

```
#snap install micro --classic
```

保存文件后，按 Ctrl+Q 组合键退出。

1.3　Elasticsearch 的 Python 客户端

首先介绍 Elasticsearch 的安装以及 Python 客户端的基本使用，然后介绍如何定义索引结构。

1.3.1　安装 Elasticsearch

如果在 Windows 操作系统下，则可以从 https://www.elastic.co/cn/downloads/elasticsearch 下载 Elasticsearch 的安装包。这里使用的版本为 7.8.0。得到文件 elasticsearch-7.8.0.zip。

直接解压至某目录，例如 D:\elasticsearch-7.8.0。下载完解压后有以下几个路径：bin 是运行的脚本；config 是设置文件；lib 中放的是依赖包。到目录 D:\elasticsearch-7.8.0\bin 下，运行 elasticsearch.bat。

如果显示 Java 虚拟机（Java Virtual Machine，JVM）内存不够，则可以在 D:\elasticsearch-7.8.0\config\jvm.options 配置文件中调整内存大小。其中，Xms 参数表示堆空间的初始大小；Xmx 参数表示堆空间的最大大小。应该把最小和最大 JVM 堆设置成相同的值。例如：

```
-Xms4g
-Xmx4g
```

成功启动 Elasticsearch 后，在浏览器中打开网址：http://localhost:9200/。

启动成功后，会在解压目录下增加 2 个文件夹：data 用于存储索引数据；logs 用于日志记录。因为创建索引耗时，所以预先将文档写入一个日志目录。

如果使用 Linux 的 Ubuntu 发行版，则可以用 debian 安装包安装 Elasticsearch；如果使用 RedHat Enterprise Linux，则可以使用 RPM 包安装 Elasticsearch。

在 Windows 操作系统下，启动 Elasticsearch 服务：

```
> bin\elasticsearch.bat
```

通过 HTTP（Hyper Text Transfer Protocol，超文本传输协议）发送指令和 Elasticsearch 交互。curl 是一个知名的网络命令行工具，可以用来发送 GET 或者 POST 命令。在 Linux 下默认已经安装了这个命令行工具，可以用 curl 命令行访问网址 http://localhost:9200/：

```
> curl http://localhost:9200/
```

或者在命令行外壳程序 PowerShell 中运行：

```
> Invoke-RestMethod http://localhost:9200
```

在 Windows 下可以用 Python 代码测试 Elasticsearch 服务：

```
import requests
url = 'http://localhost:9200'

response = requests.get(url)

print(response.status_code)
print(response.content)
```

Kibana 是 Elasticsearch 的开源数据可视化仪表板，可以在 Kibana 控制台发送命令给 Elasticsearch。在 Windows 操作系统下，Kibana 可以使用.zip 软件包安装，从 https://www.elastic.co/cn/downloads/kibana 下载 Kibana 安装包。

在编辑器中打开配置文件 config/kibana.yml 设置 Elasticsearch 实例：

```
elasticsearch.hosts: ["http://localhost:9200"]
```

在 Windows 操作系统下运行 bin\kibana.bat，以启动 Kibana。

在浏览器中打开网址 http://localhost:5601 进入 Kibana 的管理界面。

如果无法正常访问，则可以用如下命令检查端口占用情况。

```
>netstat -ano | findstr 5601
```

如果仍然无法正常使用，则可以先测试 X-Pack 插件或者检查 Elasticsearch 中插件的安装情况。

1.3.2　基本使用

elasticsearch-py（https://github.com/elastic/elasticsearch-py）是 Elasticsearch 的官方底层 Python 客户端。Elasticsearch DSL（https://github.com/elastic/elasticsearch-dsl-py）是高层客户端。

先介绍 elasticsearch-py 的使用。使用 pip 命令安装 elasticsearch 模块：

```
pip install elasticsearch
```

定义索引：

```
from elasticsearch import Elasticsearch

#By default we connect to localhost:9200
es = Elasticsearch()

es.indices.create(index='accesslog', ignore=400)

es.indices.put_mapping(
    index="accesslog",
    body={
        "properties": {
               "logdate": {
                   "type":"date",
                   "format":"dd/MM/yyy HH:mm:ss"
               }
            }
         }
     )
#删除索引
es.indices.delete(index='accesslog')
```

简单的用法如下：

```
>>> from datetime import datetime
>>> from elasticsearch import Elasticsearch

#默认情况下连接到 localhost:9200
>>> es = Elasticsearch()
```

```
    #在 elasticsearch 中创建一个索引,忽略状态码 400(索引已经存在)
    >>> es.indices.create(index='my-index', ignore=400)
    {u'acknowledged': True}

    #会序列化 datetime
    >>> es.index(index="my-index", doc_type="test-type", id=42, body={"any": "data",
"timestamp": datetime.now()})
    {u'_id': u'42', u'_index': u'my-index', u'_type': u'test-type', u'_version': 1,
u'ok': True}

    #但没有反序列化
    >>> es.get(index="my-index", doc_type="test-type", id=42)['_source']
    {u'any': u'data', u'timestamp': u'2016-05-12T19:45:31.804229'}
```

索引和查询员工的例子：

```
    from elasticsearch import Elasticsearch

    es = Elasticsearch()

    e1={
        "first_name":"nitin",
        "last_name":"panwar",
        "age": 27,
        "about": "Love to play cricket",
        "interests": ['sports','music'],
    }
    print(e1)

    #插入文档
    res = es.index(index='megacorp',id=1,body=e1)
    print(res)

    e2={
        "first_name" :  "Jane",
        "last_name" :   "Smith",
        "age" :        32,
        "about" :      "I like to collect rock albums",
        "interests":  [ "music" ]
    }

    res=es.index(index='megacorp',id=2,body=e2)
    print(res)

    e3={
        "first_name" :  "Douglas",
```

```
    "last_name" :   "Fir",
    "age" :         35,
    "about":        "I like to build cabinets",
    "interests":  [ "forestry" ]
}
res=es.index(index='megacorp',id=3,body=e3)

res=es.get(index='megacorp',id=3)
print(res)

#检索文档
res= es.search(index='megacorp',body={'query':{'match_all':{}}})
print("Got %d Hits:" % res['hits']['total']['value'])

res= es.search(index='megacorp',body={'query':{'match':{'first_name':'nitin'}}})
print(res['hits']['hits'])

#布尔运算符
res= es.search(index='megacorp',body={
       'query':{
          'bool':{
             'must':[{
                    'match':{
                       'first_name':'nitin'
                    }
                }]
          }
       }
    })

print(res['hits']['hits'])

#过滤运算符
res= es.search(index='megacorp',body={
       'query':{
          'bool':{
             'must':{
                'match':{
                   'first_name':'nitin'
                }
             },
             "filter":{
                "range":{
                   "age":{
                      "gt":25
                   }
```

```
                    }
                }
            }
        }
    })

print(res['hits']['hits'])

#增加数据,尝试全文搜索
e4={
    "first_name":"asd",
    "last_name":"pafdfd",
    "age": 27,
    "about": "Love to play football",
    "interests": ['sports','music'],
}
res=es.index(index='megacorp',id=4,body=e4)
print(res)

res= es.search(index='megacorp',body={
       'query':{
           'match':{
               "about":"play cricket"
           }
       }
    })
for hit in res['hits']['hits']:
    print(hit['_source']['about'] )
    print(hit['_score'])
    print('**********************')

#短语匹配
res= es.search(index='megacorp',body={
       'query':{
           'match_phrase':{
               "about":"play cricket"
           }
       }
    })
for hit in res['hits']['hits']:
    print(hit['_source']['about'])
    print(hit['_score'])
    print('**********************')

#使用聚合功能分析员工的兴趣爱好
res= es.search(index='megacorp',body={
```

```
        "aggs": {
            "all_interests": {
            "terms": { "field": "interests" }
            }
        }
    })
```

从指定位置返回结果：

```
#设置 size 选项,确定要返回的搜索命中数
test=es.search(index=['test'], size=1000, from_=0)
```

使用函数 enumerate()返回结果：

```
result = elastic_client.search(index="some_index", body=query_body, size=999)
all_hits = result['hits']['hits']

#see how many "hits" it returned using the len() function
print ("total hits using 'size' param:", len(result["hits"]["hits"]))

#iterate the nested dictionaries inside the ["hits"]["hits"] list
for num, doc in enumerate(all_hits):
    print ("DOC ID:", doc["_id"], "--->", doc, type(doc), "\n")

    #Use 'iteritems()` instead of 'items()' if using Python 2
    for key, value in doc.items():
        print (key, "-->", value)

    #print a few spaces between each doc for readability
    print ("\n\n")
```

接下来介绍 Elasticsearch DSL 的使用。安装模块：

```
pip install elasticsearch-dsl
```

直接写成一个 dict 的典型的搜索请求如下：

```
from elasticsearch import Elasticsearch
client = Elasticsearch()

response = client.search(
    index="my-index",
    body={
      "query": {
        "bool": {
          "must": [{"match": {"title": "python"}}],
          "must_not": [{"match": {"description": "beta"}}],
          "filter": [{"term": {"category": "search"}}]
        }
      },
```

```
        "aggs" : {
          "per_tag": {
            "terms": {"field": "tags"},
            "aggs": {
              "max_lines": {"max": {"field": "lines"}}
            }
          }
        }
    }
)

for hit in response['hits']['hits']:
    print(hit['_score'], hit['_source']['title'])

for tag in response['aggregations']['per_tag']['buckets']:
    print(tag['key'], tag['max_lines']['value'])
```

这种方法的问题是它非常冗长，容易出现语法错误，例如错误的嵌套、难以修改（如添加另一个过滤器）。

用 Python DSL 重写示例：

```
from elasticsearch import Elasticsearch
from elasticsearch_dsl import Search

client = Elasticsearch()

#将术语查询放在布尔查询的过滤器上下文中
s = Search(using=client, index="my-index") \
    .filter("term", category="search") \
    .query("match", title="python")   \
    .exclude("match", description="beta")

s.aggs.bucket('per_tag', 'terms', field='tags') \
    .metric('max_lines', 'max', field='lines')

response = s.execute()

for hit in response:
    print(hit.meta.score, hit.title)

for tag in response.aggregations.per_tag.buckets:
    print(tag.key, tag.max_lines.value)
```

用一个简单的 Python 类，代表博客系统中的一篇文章：

```
from datetime import datetime
from elasticsearch_dsl import Document, Date, Integer, Keyword, Text, connections
```

```
#定义一个默认的 Elasticsearch 客户端
connections.create_connection(hosts=['localhost'])

class Article(Document):
    title = Text(analyzer='snowball', fields={'raw': Keyword()})
    body = Text(analyzer='snowball')
    tags = Keyword()
    published_from = Date()
    lines = Integer()

    class Index:
        name = 'blog'
        settings = {
          "number_of_shards": 2,
        }

    def save(self, ** kwargs):
        self.lines = len(self.body.split())
        return super(Article, self).save(** kwargs)

    def is_published(self):
        return datetime.now() > self.published_from

#在 elasticsearch 中创建映射
Article.init()

#创建并保存文章
article = Article(meta={'id': 42}, title='Hello world!', tags=['test'])
article.body = ''' looong text '''
article.published_from = datetime.now()
article.save()

article = Article.get(id=42)
print(article.is_published())

#显示群集运行状况
print(connections.get_connection().cluster.health())
```

接下来介绍摄取处理器插件的使用。

首先安装 attachment 插件：

```
#sudo bin/elasticsearch-plugin install ingest-attachment
```

然后重新启动 Elasticsearch，让插件生效。

在 Python 客户端创建一个管道，然后使用摄取处理器插件 attachment，如下所示：

```
from elasticsearch import Elasticsearch
es = Elasticsearch()
body = {
```

```
        "description" : "Extract attachment information",
         "processors" : [
          {
              "attachment" : {
              "field" : "data"
                 }
           }
         ]
       }
es.index(index='_ingest', doc_type='pipeline', id='attachment', body=body)
```

1.3.3 定义索引结构

要将 JSON 数据传递给方法的 body 参数，用于创建 Elasticsearch 映射的 Python 字典，代码如下：

```
mapping = {
    "mappings": {
        "properties": {
            "some_string": {
                "type": "text" #formerly "string"
            },
            "some_bool": {
                "type": "boolean"
            },
            "some_int": {
                "type": "integer"
            },
            "some_more_text": {
                "type": "text"
            }
        }
    }
}
```

可以更改分片和副本设置：

```
#mapping dictionary that contains the settings and
#_mapping schema for a new Elasticsearch index:
mapping = {
    "settings": {
        "number_of_shards": 2,
        "number_of_replicas": 1
    },
    "mappings": {
        "properties": {
            "some_string": {
```

```
                "type": "text" #formerly "string"
            },
            "some_bool": {
                "type": "boolean"
            },
            "some_int": {
                "type": "integer"
            },
            "some_more_text": {
                "type": "text"
            }
        }
    }
}
```

index.create()方法的 API 调用的基本布局如下：

```
#create an index with the mapping passed to the 'body' parameter
indices.create(
    index="__INDEX_NAME_GOES_HERE__",
    body=MAPPING_DICT_GOES_HERE
)
```

调用该方法时，只有两个必需的参数：一个是以字符串形式传递到索引的索引名称；另一个是将 Python 字典传递给方法的 body 参数。但是，有一个选项可以指示 Elasticsearch 忽略指定的 HTTP 错误代码。

```
#make an API call to the Elasticsearch cluster
#and have it return a response:
response = elastic_client.indices.create(
    index="some_new_index",
    body=mapping,
    ignore=400 #ignore 400 already exists code
)

#print out the response:
print ('response:', response)
```

如果执行得当，Elasticsearch 集群应返回一个 Python 字典。该字典在响应中具有值为“True”的“acknowledged”键，如下所示：

```
response: {'acknowledged': True, 'shards_acknowledged': True, 'index':
'some_new_index'}
```

评估命令的 Python 条件语句如下所示：

```
if 'acknowledged' in response:
    if response['acknowledged'] == True:
        print ("INDEX MAPPING SUCCESS FOR INDEX:", response['index'])
```

```
#catch API error response
elif 'error' in response:
    print ("ERROR:", response['error']['root_cause'])
    print ("TYPE:", response['error']['type'])
```

使用 Kibana 控制台或 curl 请求可以验证索引及其映射是否已正确创建。

定义索引结构的完整 Python 代码如下：

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

from elasticsearch import Elasticsearch
elastic = Elasticsearch(hosts=["localhost"])
#or: elastic = Elasticsearch(hosts=["localhost"])

#mapping dictionary that contains the settings and
#_mapping schema for a new Elasticsearch index:
mapping = {
    "settings": {
        "number_of_shards": 2,
        "number_of_replicas": 1
    },
    "mappings": {
        "properties": {
            "some_string": {
                "type": "text" #formerly "string"
            },
            "some_bool": {
                "type": "boolean"
            },
            "some_int": {
                "type": "integer"
            },
            "some_more_text": {
                "type": "text"
            }
        }
    }
}

#make an API call to the Elasticsearch cluster
#and have it return a response:
response = elastic.indices.create(
```

```
    index="some_new_index",
    body=mapping,
    ignore=400 #ignore 400 already exists code
)

if 'acknowledged' in response:
    if response['acknowledged'] == True:
        print ("INDEX MAPPING SUCCESS FOR INDEX:", response['index'])

#catch API error response
elif 'error' in response:
    print ("ERROR:", response['error']['root_cause'])
    print ("TYPE:", response['error']['type'])

#print out the response:
print ('\nresponse:', response)
```

Elasticsearch 不会分析 Keyword 数据类型，这意味着索引的字符串将保持不变。Keyword 数据类型可用于聚合列。

与 Keyword 字段数据类型不同，索引到 text 类型的字符串在存储到反向索引中之前将经过分析器处理。默认情况下，Elasticsearch 的标准分析器将拆分并小写化索引的字符串。

第 2 章

Python 技术基础

本章介绍开发 Web 应用所用到的 Python 技术基础。

2.1 变量

在 Python 中定义变量时，不需要声明类型，但变量在内部是有类型的。例如，在交互式环境输入如下代码，即输出变量 a 的类型：

```
>>> a='test'
>>> type(a)  #调用函数 type()得到变量 a 的类型
<class 'str'>
```

2.2 注释

与 Shell 类似，Python 脚本中用#表示注释。但是，如果#位于第一行开头，并且是#!（称为 Shebang），则例外。它表示该脚本使用后面指定的解释器/usr/bin/python3 解释执行。每个脚本程序只能在开头包含这条语句。

为了能够在源代码中添加中文注释，需要把源代码保存为 UTF-8 格式。例如：

```
#-*- coding: UTF-8 -*

import spacy
en_nlp = spacy.load('en')  #加载英文模型
```

2.3 简单数据类型

本节介绍包括数值、字符串和数组在内的简单数据类型。

2.3.1　数值

Python 中有三种不同的数值类型：int（整数）、float（浮点数）和 complex（复数）。与 Java 或者 C 语言中的 int 类型不同，Python 中的 int 类型是无限精度的。例如：

```
>>> i=32432444444444444444444444444444444444444444448797687567567657000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000564564
>>> i
32432444444444444444444444444444444444444444448797687567567657000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000564564
>>> type(i)
<class 'int'>
```

Python 依据 IEEE 754 标准使用二进制表示 float（浮点数），所以存在表示精度的问题。例如：

```
>>> 0.1 == 0.10000000000000000000001
True
```

可以使用 decimal 模块采用十进制表示完整的小数。例如：

```
>>> import decimal
>>> a = decimal.Decimal('0.1')
>>> b = decimal.Decimal('0.10000000000000000000001')
>>> a == b
False
```

在傅里叶变换中会用到复数。复数在 Python 中是一个基本数据类型（complex）。例如：

```
>>> complex(2,3)
(2+3j)
```

一个复数有一些内置的访问器：

```
>>> z = 2+3j
>>> z.real
2.0
>>> z.imag
3.0
>>> z.conjugate()
(2-3j)
```

几个内置函数支持复数：

```
>>> abs(3 + 4j)
5.0
>>> pow(3 + 4j, 2)
(-7+24j)
```

标准模块 cmath 具有处理复数的更多功能：

```
>>> import cmath
```

```
>>> cmath.sin(2 + 3j)
(9.15449914691143-4.168906959966565j)
```

用于数值运算的算术运算符说明见表 2-1。

表 2-1　算术运算符

语　　法	数 学 含 义	运算符名字
a+b	$a+b$	加
a-b	$a-b$	减
a*b	$a \times b$	乘法
a/b	$a \div b$	除法
a//b	$\lfloor a \div b \rfloor$	地板除
a%b	$a \bmod b$	模
-a	$-a$	取负数
abs(a)	$\lvert a \rvert$	绝对值
a**b	a^b	指数
math.sqrt(a)	$\sqrt{a}$	平方根

对于“/”运算，就算分子和分母都是 int，返回的也将是浮点数。例如：

```
>>> print(1/3)
0.3333333333333333
```

Python 支持不同的数字类型相加，它使用数字类型强制转换的方式来解决数字类型不一致的问题。也就是说，它会将一个操作数转换成与另一个操作数相同的数据类型。

如果有一个操作数是复数，则另一个操作数被转换为复数：

```
>>> 3.0 + (5+6j)      #非复数转换为复数
(8+6j)
```

整数转换为浮点数：

```
>>> 6 + 7.0           #非浮点型转换为浮点型
13.0
```

Python 代码中一般一行就是一条语句，但是可以使用斜杠（\）将一条语句分为多行显示。例如：

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> total = a + \
... b + \
... c
```

```
>>> total
6
```

2.3.2　字符串

使用 strip()方法可以去掉字符串首尾的空格或者指定的字符。例如：

```
term = "   hi      ";
print(term.strip());          #去除首尾空格
```

使用 split()方法将句子分成单词。例如下面的代码中，mary 是一个单一的字符串，尽管这是一个句子，但这些词语并没有表示成严谨的单位。为此，需要一种不同的数据类型：字符串列表，其中每个字符串对应一个单词。使用 split()方法把句子切分成单词：

```
>>> mary = 'Mary had a little lamb'
>>> mary.split()
['Mary', 'had', 'a', 'little', 'lamb']
```

split()方法根据空格拆分 mary，返回的结果是 mary 中的单词列表。此列表包含函数 len()演示的 5 个项目。对于 mary，函数 len()返回字符串中的字符数（包括空格）。

```
>>> mwords = mary.split()
>>> mwords
['Mary', 'had', 'a', 'little', 'lamb']
>>> len(mwords)              #mwords 中的项目数
5
>>> len(mary)                #字符数
22
```

空白字符包括空格' '，换行符'\ n'和制表符'\ t'等。.split()分隔这些字符的任何组合序列：

```
>>> chom = ' colorless    green \n\tideas\n'
>>> print(chom)
 colorless    green
   ideas

>>> chom.split()
['colorless', 'green', 'ideas']
```

通过提供可选参数，.split('x')可用于在特定子字符串'x'上拆分字符串。如果没有指定'x'，则.split()只是在所有空格上分割，例如：

```
>>> mary = 'Mary had a little lamb'
>>> mary.split('a')              #根据'a'切分
['M', 'ry h', 'd ', ' little l', 'mb']
>>> hi = 'Hello mother,\nHello father.'
>>> print(hi)
Hello mother,
```

```
Hello father.
>>> hi.split()                    #没有给出参数：在空格上分割
['Hello', 'mother,', 'Hello', 'father.']
>>> hi.split('\n')                #仅在'\n'上分割
['Hello mother,', 'Hello father.']
```

但是，如果想将一个字符串拆分成一个字符列表呢？在 Python 中，字符只是长度为 1 的字符串。函数 list()将字符串转换为单个字母的列表：

```
>>> list('hello world')
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

如果有一个单词列表，可以使用.join()方法将它们重新组合成一个单独的字符串。在“分隔符”字符串'x'上调用，'x'.join(y)连接列表 y 中由'x'分隔的每个元素。例如下面的代码，mwords 中的单词用空格连接回句子字符串：

```
>>> mwords
['Mary', 'had', 'a', 'little', 'lamb']
>>> ' '.join(mwords)
'Mary had a little lamb'
```

也可以在空字符串"上调用该方法作为分隔符。效果是列表中的元素连接在一起，元素之间没有任何内容。例如下面的代码，将一个字符列表放回到原始字符串中：

```
>>> hi = 'hello world'
>>> hichars = list(hi)
>>> hichars
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> ''.join(hichars)
'hello world'
```

对一个字符串取子串的例子代码如下：

```
>>> x = "Hello World!"
>>> x[2:]
'llo World!'
>>> x[:2]
'He'
>>> x[:-2]
'Hello Worl'
>>> x[-2:]
'd!'
>>> x[2:-2]
'llo Worl'
```

使用函数 ord()和 chr()实现字符串和整数之间的相互转换：

```
>>> a = 'v'
>>> i = ord(a)
>>> chr(i)
```

```
'v'
```

2.3.3　数组

创建一个数组，然后向这个数组添加元素的代码如下：

```
>>> temp_list = []
>>> print(temp_list)
[]
>>> temp_list.append("one")
>>> temp_list.append("two")
>>> print(temp_list)
['one', 'two']
>>>
```

创建一个指定长度的数组：

```
>>> size = 10
>>> lst = [None] * size
>>> lst
[None, None, None, None, None, None, None, None, None, None]
```

2.4　字面值

Python 包括以下几种类型的字面值。

- 数字：整数、浮点数、复数。
- 字符串：以单引号、双引号或者三引号定义字符串。
- 布尔值：True 和 False。
- 空值：None。

Python 中有 4 种不同的字面值集合：列表字面值、元组字面值、字典字面值和集合字面值。示例代码如下：

```
fruits = ["apple", "mango", "orange"]              #列表
numbers = (1, 2, 3)                                #元组
alphabets = {'a':'apple', 'b':'ball', 'c':'cat'}  #字典
vowels = {'a', 'e', 'i' , 'o', 'u'}                #集合

print(fruits)
print(numbers)
print(alphabets)
print(vowels)
```

2.5 控制流

完成一件事情要有流程控制。例如，洗衣服有三个步骤：把脏衣服放进洗衣机，等洗衣机洗好衣服，晾衣服。这是顺序控制结构。

顺序执行的代码采用相同的缩进，叫作一个代码块。Python 没有像 Java 或者 C#语言那样采用{}分隔代码块，而是采用代码缩进和冒号来区分代码之间的层次。

缩进的空格数量是可变的，但是所有代码块语句必须包含相同的缩进空格数量。NodePad++这样的文本编辑器支持选择多行代码后，按 Tab 键可以改变代码块的缩进格式。

控制流用来根据运行时的情况调整语句的执行顺序。流程控制语句可以分为条件语句和迭代语句。

2.5.1 if 语句

路径不存在，就创建它，可以使用条件语句实现。条件语句的一般形式如下：

```
if 条件:
    语句 1
    语句 2
    ...
elif 条件:
    语句 1
    语句 2
    ...
else:
    语句 1
    语句 2
    ...
语句 x
```

例如，判断一个数是否是正数的代码如下：

```
x = -32.2;
isPositive = (x > 0);
if isPositive:
   print(x, " 是正数");
else:
print(x, " 不是正数");
```

这里的 if 复合语句，首行以关键字开始，以冒号（:）结束。

使用关系运算符和条件运算符作为判断依据。关系运算符返回一个布尔值。关系运算符的说明见表 2-2。

表 2-2 关系运算符

运 算 符	用 法	返回 true，如果……
>	a > b	a 大于 b
>=	a >= b	a 大于或等于 b

续表

运　算　符	用　　法	返回 true，如果……
<	a < b	a 小于 b
<=	a <= b	a 小于或等于 b
==	a == b	a 等于 b
!=	a != b	a 不等于 b

如果针对多个值测试一个变量，则可以在 if 条件判断中使用一个集合：

```
x = "Wild things"
y = "throttle it back"
z = "in the beginning"
if "Wild" in {x, y, z}: print (True)
```

2.5.2　循环

使用复印机复印一个证件，可以设定复制的份数。例如，复制 3 份。在 Python 中，可以使用 for 循环或者 while 循环实现多次重复执行一个代码块。

for 循环可以遍历任何序列。例如，输出数组中的元素：

```
mylist = [1,2,3]
for item in mylist:
    print(item)
```

输出字符串中的字符：

```
>>> for c in '风调雨顺' :
...     print(c,type(c))
...
风 <class 'str'>
调 <class 'str'>
雨 <class 'str'>
顺 <class 'str'>
```

或者借助函数 range()遍历字符串中的字符：

```
>>> word = '风调雨顺'
>>> for i in range(len(word)):
...     print(word[i])
...
风
调
雨
顺
```

因为 Python 3 中并不存在表示单个字符的数据类型，所以返回的变量 c 仍然是 str 类型。

输出字符串'banana'中每个出现的字符及其位置：

```
>>> for c in enumerate('banana'):
...     print(c)
...
(0, 'b')
(1, 'a')
(2, 'n')
(3, 'a')
(4, 'n')
(5, 'a')
```

在 Python 中，可以将一个可选的“else”块与循环关联。“else”块仅在循环完成所有迭代后才执行。例如：

```
for val in range(5):
   print(val)
else:
    print("The loop has completed execution")
```

输出：

```
0
1
2
3
4
The loop has completed execution
```

每一次在执行循环代码块之前，根据循环条件决定是否继续执行循环代码块，当满足循环条件时，继续执行循环体中的代码。在循环条件之前写上关键词 while。这里的 while 就是“当”的意思。例如，当用户直接按 Enter 键时退出循环：

```
import sys

while True:
    line = sys.stdin.readline().strip()
    if not line:
        break
    print(line)
```

2.6 列表

使用一个列表可以存储任何类型的对象：

```
list1 = ['physics', 'chemistry', 1997, 2000];
print("list1[0]: ", list1[0])
```

输出：

```
list1[0]:  physics
```

可以通过切片运算来获得一个列表：

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

应用 lambda 表达式截短字符串：

```
>>> lines = ['this', 'is', 'a', 'list', 'of', 'words']
>>> list(map(lambda it: it[:3], lines))
['thi', 'is', 'a', 'lis', 'of', 'wor']
```

2.7　元组

元组是一个不可变的 Python 对象序列。元组变量的赋值要在定义时就进行，定义时赋值后就不允许有修改。

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print( "tup1[0]: ", tup1[0]);
print( "tup2[1:5]: ", tup2[1:5]);
```

通常将元组用于异构（不同）数据类型，将列表用于同类（相似）数据类型。

包含多个项目的文字元组可以分配给单个对象。当发生这种情况时，就好像元组中的项目已经“打包”到对象中。

```
>>> t = ('foo', 'bar', 'baz', 'qux')
```

将元组中的元素分别赋给变量称为拆包。

```
>>> (s1, s2, s3, s4) = t
>>> s1
'foo'
>>> s2
'bar'
>>> s3
'baz'
>>> s4
'qux'
```

包装和拆包可以合并为一条语句，以进行复合分配：

```
>>> (s1, s2, s3, s4) = ('foo', 'bar', 'baz', 'qux')
>>> s1
'foo'
```

```
>>> s2
'bar'
>>> s3
'baz'
>>> s4
'qux'
```

可以构建一个由元组组成的数组：

```
>>> pairs = [("a", 1), ("b", 2), ("c", 3)]
>>> for a, b in pairs:
...    print(a, b)
...
a 1
b 2
c 3
```

使用命名元组可以给元组中的元素起一个有意义的名字：

```
import collections

#声明一个名为 Person 的命名元组,这个元组包含 name 和 age 两个键
Person = collections.namedtuple('Person', 'name age')

#使用命名元组
bob = Person(name='Bob', age=30)
print('\nRepresentation:', bob)

jane = Person(name='Jane', age=29)
print('\nField by name:', jane.name)

print('\nFields by index:')
for p in [bob, jane]:
   print('{} is {} years old'.format(*p))
```

2.8 集合

使用 in 运算符可以检查给定元素是否存在于集合中。如果集合中存在指定元素，则返回 True；否则，返回 False。

```
>>> s = {1,2,3,4,5}            #创建 set 对象并将其分配给变量 s
>>> contains = 1 in s          #判断是否包含的例子
>>> print(contains)
True
>>> contains = 6 in s
>>> print(contains)
```

```
False
```

输出字符串'banana'中的字符集合：

```
>>> set(c for (i,c) in enumerate('banana'))
{'n', 'a', 'b'}
```

可以使用 set.update()方法增加项目到集合。

```
>>> A = [1, 2, 3]
>>> S = set()
>>> S.update(A)
>>> S
{1, 2, 3}
```

可以使用 set.intersection()方法找出两个集合都包含的元素。

```
A = {2, 3, 5, 4}
B = {2, 5, 100}

print(B.intersection(A))  #输出 2,5
```

2.9　字典

使用字典可以存取键/值对。若访问字典元素，则可以使用熟悉的方括号和键来获取字典中的值。

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print("dict['Name']: ", dict['Name'])
print("dict['Age']: ", dict['Age'])
```

通过为该键指定值可以在字典上创建新的键/值对。如果该键不存在，则将该键/值对添加到字典。如果该键已经存在，则覆盖它指向的当前值。

```
d = {'key':'value'}
print(d)    #输出 {'key': 'value'}
d['mynewkey'] = 'mynewvalue'
print(d)    #输出 {'mynewkey': 'mynewvalue', 'key': 'value'}
```

运行如下语句会抛出 KeyError 异常：

```
print(d['noexists'])
```

运行如下语句会返回 None：

```
print(d.get('noexists'))
```

如果只需要判断键是否在字典中存在，则可以使用 in 关键字实现：

```
>>> d = {'a': 1, 'b': 2}
>>> 'a' in d #<==评估为 True
```

```
True
>>> 'c' in d #<==评估为 False
False
```

键可以是复杂数据类型：

```
>>> transProb = {}
>>> transProb[('yes','是')] = 0.6
>>> transProb[('yes','好')] = 0.4
>>> transProb[('good','好')]                          #访问不存在的键触发异常
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: ('good', '好')
```

为了避免异常，可以通过函数 collections.defaultdict()设置默认值：

```
>>> from decimal import Decimal
>>> import collections
>>> transProb = collections.defaultdict(Decimal)  #设定默认值
>>>
>>> transProb[('yes','是')] = 0.6
>>> transProb[('yes','好')] = 0.4
>>>
>>> transProb[('good','好')]                          #返回默认值
Decimal('0')
```

如果需要根据字典中的值排序，由于字典本质上是无序的，则可以把排序结果保存到有序的列表。

```
>>> x = {1: 2, 3: 4, 4: 3, 2: 1, 0: 0}
>>> sorted_by_value = sorted(x.items(), key=lambda kv: kv[1])
>>> print(sorted_by_value)
[(0, 0), (2, 1), (1, 2), (4, 3), (3, 4)]
```

OrderedDict 是一个字典子类，它会记住键/值对的顺序。

```
import collections

print('普通的字典:')
d = {}
d['a'] = 'A'
d['b'] = 'B'
d['c'] = 'C'

for k, v in d.items():
    print(k, v)

print('\n有序的字典:')
d = collections.OrderedDict()
d['a'] = 'A'
```

```
    d['b'] = 'B'
    d['c'] = 'C'
    d['a'] = 'a'

    for k, v in d.items():
        print(k, v)
```

2.10　位数组

位数组（也称为位图）通常用于快速内存访问上的一种数据结构。位图可以表示文档的切分结果。不幸的是，位图可能会使用太多内存。为了补偿，可以使用压缩位图。

PyRoaringBitMap（https://github.com/Ezibenroc/PyRoaringBitMap）是一个 C 语言库 CRoaring 的 Python 包装器。

可以使用 PyPi 安装 pyroaring：

```
    #pip3 install pyroaring
```

或者从 whl 文件安装：

```
    #pip3 install --user https://github.com/Ezibenroc/PyRoaringBitMap/releases/download/0.2.1/pyroaring-0.2.1-cp36-cp36m-linux_x86_64.whl
```

几乎可以像使用经典的 Python 集合那样在代码中使用 BitMap：

```
    from pyroaring import BitMap
    bm1 = BitMap()
    bm1.add(3)
    bm1.add(18)
    bm2 = BitMap([3, 27, 42])
    print("bm1      = %s" % bm1)
    print("bm2      = %s" % bm2)
    print("bm1 & bm2 = %s" % (bm1&bm2))   #按位运算
    print("bm1 | bm2 = %s" % (bm1|bm2))
```

输出：

```
    bm1      = BitMap([3, 18])
    bm2      = BitMap([3, 27, 42])
    bm1 & bm2 = BitMap([3])
    bm1 | bm2 = BitMap([3, 18, 27, 42])
```

遍历位数组：

```
    >>> a = iter(bm1)                      #取得 iterator
    >>> print(next(a, None))               #取得下一个元素,如果没有,则返回 None
    3
    >>> print(next(a, None))
```

```
    18
    >>> print(next(a, None))
    None
```

2.11 模块

使用 import 语句可以导入一个在.py 文件中定义的函数。一个.py 文件称为一个模块（Module）。例如，存在一个 re.py 文件，可以使用 import re 语句导入这个正则表达式模块。

使用正则表达式模块去掉一些标点符号的例子代码如下：

```
import re

line = 'Hi.'
normtext = re.sub(r'[\.,:;\?]', '', line)
print(normtext)
```

从 re 模块直接导入 sub 函数的例子代码如下：

```
from re import sub

line = 'Hi.'
normtext = sub(r'[\.,:;\?]', '', line)
print(normtext)
```

模块越来越多以后，会难以管理。例如，可能会出现重名的模块。又如，一个班里有两个叫陈晨的同学，如果他们在不同的小组，可以叫第一组的陈晨或者第三组的陈晨，这样就能区分同名。为了避免名字冲突，模块可以位于不同的命名空间，叫作包。在模块名前面可以加上包名限定，这样即使模块名相同，也不会冲突。

为了查看本地有哪些模块可用，可以在 Python 交互式环境中输入：

```
help('modules')
```

对于大项目，可以把多个模块文件置于同一个目录下组成包，而且必须在该目录下放置一个__init__.py 文件来让 Python 识别出这是一个包。

2.12 函数

把一段多次重复出现的函数命名成一个有意义的名字，然后通过名字来执行这段代码。有名字的代码段就是一个函数。

Python 解释器内置了一些函数。其中，函数 str()将对象转换为易于阅读的字符串；函数 len()用于获取对象的长度；函数 id()返回对象的标识；函数 print()将给定对象打印到标准

输出设备（屏幕）或文本流文件。

2.12.1　print 函数

显示某个目录下的文件数量的代码如下：

```
import os

folderlist = os.listdir('/home/soft/kaldi/')
total_num_file = len(folderlist)

print ('total '+total_num_file+' files')
```

这样会出错，因为 Python 不支持+运算中的整数自动转换成字符串。调用函数 str()可以将整数转换成字符串：

```
print ('total '+str(total_num_file)+' files')
```

或者格式化：

```
print ('total have %d files' % (total_num_file))    #%d 表示输出整数
```

另外一种格式化输出的方法是使用 str.format()方法，下面的代码比较了这两种方法：

```
>>> sub1 = "python string!"
>>> sub2 = "an arg"
>>> a = "i am a %s" % sub1
>>> b = "i am a {0}".format(sub1)
>>> print(a)
i am a python string!
>>> print(b)
i am a python string!
>>> c = "with %(kwarg)s!" % {'kwarg':sub2}
>>> print(c)
with an arg!
>>> d = "with {kwarg}!".format(kwarg=sub2)
>>> print(d)
with an arg!
```

如下的代码会出错：

```
>>> name=(1, 2, 3)
>>> print("hi there %s" % name)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not all arguments converted during string formatting
```

函数 print()用到的格式化字符串的约定见表 2-3。

表 2-3　格式化字符串中的转换类型

转 换 类 型	含　义
d,i	带符号的十进制整数
o	不带符号的八进制数
u	不带符号的十进制数
x	不带符号的十六进制数（小写）
X	不带符号的十六进制数（大写）
e	科学记数法表示的浮点数（小写）
E	科学记数法表示的浮点数（大写）
f,F	十进制浮点数
g	如果指数大于-4 或者小于精度值，则和 e 相同，其他情况和 f 相同
G	如果指数大于-4 或者小于精度值，则和 E 相同，其他情况和 F 相同
C	单字符（接受整数或者单字符字符串）
r	字符串（使用 repr()转换任意 Python 对象）
s	字符串（使用 str()转换任意 Python 对象）

2.12.2　定义函数

使用关键字 def 定义一个函数。例如：

```
def square(number):              #定义一个名为 square 的函数
    return number * number       #返回一个数的平方
print(square(3))                 #输出：9
```

代码中可以给函数增加说明：

```
def square_root(n):
    """计算一个数字的平方根。

    Args:
        n：用来求平方根的数字。
    Returns:
        n 的平方根。
    Raises:
        TypeError：如果 n 不是数字。
        ValueError：如果 n 是负数。

    """
    pass
```

参数可以有默认值。例如，定义一个名为 RunKaldiCommand 的函数：

```
import subprocess
```

```
    def RunKaldiCommand(command, wait = True):        #wait 的默认值是 True
        """通常执行由管道连接的一系列命令,所以使用 shell=True """
        p = subprocess.Popen(command, shell = True,
                             stdout = subprocess.PIPE,
                             stderr = subprocess.PIPE)

        if wait:
            [stdout, stderr] = p.communicate()
            if p.returncode is not 0:                 #执行命令出现错误
                raise Exception("There was an error while running the command
{0}\n".format(command)+"-"*10+"\n"+stderr)
            return stdout, stderr
        else:
            return p
```

使用这个函数：

```
    RunKaldiCommand("ls -lh")
```

这里只给 RunKaldiCommand()方法的第一个参数传递了值，第二个值采用默认的 True。

如果需要声明可变数量的参数，则在这个参数前面加*。例如：

```
    def myFun(*argv):
        for arg in argv:
            print (arg)

    myFun('Hello', 'a', 'to', 'b')
```

函数定义中的特殊语法**kwargs 用于传递一个键/值对的，且可变长度的参数列表。例如：

```
    def myFun(**kwargs):
        for key, value in kwargs.items():
            print ("%s == %s" %(key, value))

    #调用函数
    myFun(first ='test', mid ='for', last='abc')
```

输出结果如下：

```
    first == test
    mid == for
    last == abc
```

每个 Python 文件/脚本（模块）都有一些未明确声明的内部属性。其中，一个属性是__builtins__属性，它本身包含许多有用的属性和功能。在这里可以找到__name__属性，根据模块的使用方式，它可以具有不同的值。

当把 Python 模块作为程序直接运行时（无论是从命令行还是双击它），__name__中包

含的值都是文字字符串"__main__"。

相比之下，当一个模块被导入到另一个模块中（或者在 Python REPL 被导入）时，__name__属性中的值是模块本身的名称（即隐式声明它的 Python 文件/脚本的名称）。

Python 脚本执行的方式是自上而下的。指令在解释器读取它们时执行。这可能是一个问题，如果想要做的就是导入模块并利用它的一个或两个方法，则可以有条件地执行这些指令——将它们包装在一个 if 语句块中。

这是'main 函数'的目的。它是一个条件块，因此除非满足给定的条件，否则不会处理函数 main()。

函数 main()的例子代码如下：

```
import sys

def main():
    if len(sys.argv) != 2:
        sys.stderr.write("Usage: {0} <min-count>\n".format(sys.argv[0]))
        raise SystemExit(1)

    words = {}
    for line in sys.stdin.readlines():
        parts = line.strip().split()
        words[parts[1]] = words.get(parts[1], 0) + int(parts[0])

    for word, count in words.iteritems():
        if count >= int(sys.argv[1]):
            print ("{0} {1}".format(count, word))

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

2.13 面向对象编程

定义一个 Token 类描述词在文本中的位置：

```
class Token(object):                                    #Token 类是 object 的子类
    """标记"""
    def __init__(self, text, offset, data=None):        #构造方法
        self.offset = offset                            #词在文档中的开始位置
        self.text = text                                #词
        self.end = offset + len(text)                   #词在文档中的结束位置
        self.data = data if data else {}

    def set(self, prop, info):
```

```
        self.data[prop] = info

    def get(self, prop, default=None):
        return self.data.get(prop, default)
```

调用这个构造方法来创造对象。例如，有个词出现在文档的开始位置：

```
t = Token("剧情", 0)  #出现在开始位置的“剧情”这个词
```

例如，有个根据指定字符分割输入字符串的 StringTokenizer 类实现如下：

```
class StringTokenizer(object):
    """字符串分隔类"""

    def __init__(self,text:str, delim:str):
        self.currentPosition = 0
        self.newPosition = -1
        self.text = text
        self.maxPosition = len(text)
        self.delimiters = delim

    def skip_delimiters(self,startPos:int)->int:
        """跳过分隔符"""
        position = startPos
        while ( position < self.maxPosition):
            c = self.text[position]
            if( self.delimiters.find(c) == -1  ):
                break
            position+=1
        return position

    def scan_token(self,startPos:int)->int:
        """扫描符号"""
        position = startPos
        while (position < self.maxPosition):
            c = self.text[position]
            if( self.delimiters.find(c) != -1 ):
                break
            position+=1
        return position

    def next_token(self):
        """取得下一个标记"""
        self.currentPosition = self.skip_delimiters(self.currentPosition)
        start = self.currentPosition
        self.currentPosition = self.scan_token(self.currentPosition)
        return self.text[start: self.currentPosition]
```

使用这个 StringTokenizer 类：

```
st = StringTokenizer("2#7#8#9#道","#")
print(st.next_token())
print(st.next_token())
print(st.next_token())
print(st.next_token())
print(st.next_token())
print(st.next_token())
```

2.13.1 静态方法

静态方法是一种属于类中的函数，同时表明它不需要访问类。@classmethod 创建了一个方法，其第一个参数是从中调用的类（而不是类实例），@staticmethod 没有任何隐式参数。例如：

```
class A(object):
    def foo(self, x):
        print( "executing foo(%s, %s)" % (self, x) )

    @classmethod
    def class_foo(cls, x):
        print( "executing class_foo(%s, %s)" % (cls, x) )

    @staticmethod
    def static_foo(x):
        print( "executing static_foo(%s)" % x )
a = A()
```

下面是对象实例调用方法的常用方法。对象实例 a 作为第一个参数隐式传递。

```
a.foo(1)
#executing foo(<__main__.A object at 0xb7dbef0c>,1)
```

使用类可以调用 class_foo。

```
A.class_foo(1)
#executing class_foo(<class '__main__.A'>,1)
```

对于静态方法，self（对象实例）和 cls（类）都不会作为第一个参数隐式传递。调用静态方法：

```
A.static_foo('hi')
#executing static_foo(hi)
```

在类定义中声明而不是在方法内部声明的变量是静态变量：

```
class MyClass:
    i = 3
```

```
print(MyClass.i)          #输出静态变量 i 的值

m = MyClass()
m.i = 4                   #实例变量
print(MyClass.i, m.i)     #输出静态变量和实例变量的值 3 4
```

生成唯一 ID：

```
import itertools

class BarFoo:

    id_iter = itertools.count()

    def __init__(self):
        self.id = next(self.id_iter)
```

2.13.2　__call__方法

Python 有一组内置方法，而__call__是其中之一。__call__方法可以使 Python 程序员编写实例的行为类似于函数的类。当实例作为函数调用时，如果定义了此方法，则 x(arg1, arg2, ...)是 x.__call__(arg1, arg2, ...)的简写。

object()是 object.__call__()的简写。

示例 1：

```
class Example:
    def __init__(self):
        print("Instance Created")

    #Defining __call__ method
    def __call__(self):
        print("Instance is called via special method")

#Instance created
e = Example()

#__call__ method will be called
e()
```

输出：

```
Instance Created
Instance is called via special method
```

示例 2：

```
class Product:
    def __init__(self):
```

```
        print("Instance Created")

    #Defining __call__ method
    def __call__(self, a, b):
        print(a * b)

#Instance created
ans = Product()

#__call__ method will be called
ans(10, 20)
```

输出：

```
Instance Created
200
```

2.14 使用 StringIO 模块

StringIO 模块是内存中类似文件的对象。创建 StringIO 对象时，可通过将字符串传递给构造函数来对其进行初始化。如果未传递任何字符串，则 StringIO 将以空开始。在这两种情况下，文件上的初始光标都从零开始。

例子代码如下：

```
from io import StringIO

#The arbitrary string.
string = 'Hello and welcome to GeeksForGeeks.'

#Using the StringIO method
#to set as file object.
file = StringIO(string)

#Reading the initial file:
print(file.read())

#To set the cursor at 0.
file.seek(0)

#This will drop the file after
#index 18.
file.truncate(18)

#File after truncate.
```

```
print(file.read())
file.seek(0)
print(file.read())
```

使用 StringIO 实现的字符缓存：

```
import io

class StringBuilder(object):

    def __init__(self):
        self._stringio = io.StringIO()

    def __str__(self):
        return self._stringio.getvalue()

    def append(self, *objects, sep=' ', end=''):
        print(*objects, sep=sep, end=end, file=self._stringio)

sb = StringBuilder()
sb.append('a')
sb.append('b', end='\n')
sb.append('c', 'd', sep=',', end='\n')
print(sb)  #'ab\nc,d\n'
```

2.15　文件操作

文件的绝对路径由目录和文件名两部分构成，示例代码如下：

```
import os.path

path = '/home/data/file.wav'

print(os.path.abspath(path))      #返回绝对路径（包含文件名的全路径）
print(os.path.basename(path))     #返回路径中包含的文件名
print(os.path.dirname(path))      #返回路径中包含的目录
```

输出：

```
/home/data/file.wav
file.wav
/home/data
```

2.15.1　读写文件

调用函数 open(fileName)返回一个_io.TextIOWrapper 对象。例如，文本文件 a.txt 包含

以下内容：

```
    the quick person did not realize his speed and the quick person bumped
```

统计文本中的词频：

```
    import re
    from collections import Counter
    words = re.findall('\w+', open('a.txt').read())
    print( Counter(words) )
```

输出如下：

```
    Counter({'the': 2, 'quick': 2, 'person': 2, 'did': 1, 'not': 1, 'realize': 1,
'his': 1, 'speed': 1, 'and': 1, 'bumped': 1})
```

逐行读入文本文件：

```
    lexicon = open("lexicon.txt")

    for line in lexicon:
       line = line.strip()
       print(line,"\n")

    lexicon.close()
```

读入 UTF-8 编码格式的文本文件：

```
    import codecs
    import sys

    transcript = codecs.open(sys.argv[1], "r", "UTF-8")    #第一个参数传入文件名

    for line in transcript:
       print(line)

    transcript.close()
```

为了实现写入文本文件，可以使用'w'模式的函数 open()以写模式打开新文件。

```
    new_path = "a.speaker_info"
    fout = open(new_path,'w')
```

需要注意，如果 new_days.txt 在打开文件前已经存在，它的旧内容将被破坏，所以在使用'w'模式时要小心。

一旦打开新文件，可以使用写入操作<file>.write()将数据放入文件中。写入操作接受单个参数，该参数必须是字符串，并将该字符串写入文件。如果要在文件中开始新行，则必须明确提供换行符。例子代码如下：

```
    fout.write("\nID:\t1212")
```

关闭文件可确保磁盘上的文件和文件变量之间的连接已完成。关闭文件还可确保其他

程序能够访问它们并保证数据安全。所以，一定要确保关闭文件。现在，使用<file>.close()函数关闭所有文件。

```
fout.close()
```

使用函数 open()创建文件对象可以采用的模式总结如下：

- 'r'用于读取现有文件（默认值；可以省略）。
- 'w'用于创建写入的新文件。
- 'a'用于将新内容附加到现有文件。

取得文件的修改时间：

```
>>> import os
>>> os.stat('test.txt').st_mtime
1559512015.9773836
```

对于 JSON 格式的文件，可以导入 json 模块读取：

```
import json
data = json.load(open('my_file.json', 'r'))
```

演示 JSON 文件的内容如下：

```
{"hello":"lietu"}
```

演示读取 JSON 格式的文件如下：

```
>>> import json
>>> print(json.load(open('my_file.json','r')))
{u'hello': u'lietu'}
```

2.15.2　重命名文件

使用 os.rename()方法可以重命名文件。首先用 touch 命令创建一个空文件：

```
#touch ./test1
```

然后把 test1 重命名为 test2：

```
import os
src= 'test1'
dst= 'test2'
os.rename(src, dst)
```

2.15.3　遍历文件

使用 os.scandir()方法遍历一个目录。os.scandir()方法返回一个迭代器。

```
import os

with os.scandir('/home/') as entries:
```

```
    for entry in entries:
        print(entry.name)
```

这里通过 with 语句使用上下文管理器关闭迭代器并在迭代器耗尽后自动释放获取的资源。

只打印出一个目录下的文件：

```
dir_entries = os.scandir('/home/')
for entry in dir_entries:
    if entry.is_file():                          #判断项目是否为文件
        print(f'{entry.name}')
```

如果要遍历一个目录树并处理树中的文件，则可以使用 os.walk()方法。os.walk()方法默认以自上而下的方式遍历目录：

```
import os
for root, dirs, files in os.walk("/home/"):
    for name in files:
        print(os.path.join(root, name))          #打印文件
    for name in dirs:
        print(os.path.join(root, name))          #打印目录
```

2.16 迭代器

迭代器（iterator）用来遍历生成器中的元素。通过函数 next()迭代 iterator 对象中的元素。

```
>>> r = range(5)
>>> itr = iter(r)
>>> next(itr)
0
```

如果没有元素，则抛出 StopIteration 异常。为了避免函数 next()抛出 StopIteration 异常，可以指定一个默认值。

```
>>> print(next(itr, None))    #取得下一个元素,如果没有,则返回 None
1
```

支持迭代的生成器 StringTokenizer 类需要实现__iter__()方法。

```
class StringTokenizer(object):
    """字符串分隔类"""

    def __init__(self,text:str, delim:str):
        self.currentPosition = 0
        self.newPosition = -1
        self.text = text
        self.maxPosition = len(text)
```

```
        self.delimiters = delim

    def skip_delimiters(self,startPos:int)->int:
        """跳过分隔符"""
        position = startPos
        while ( position < self.maxPosition):
            c = self.text[position]
            if( self.delimiters.find(c) == -1  ):
                break
            position+=1
        return position

    def scan_token(self,startPos:int)->int:
        """扫描标记"""
        position = startPos
        while (position < self.maxPosition):
            c = self.text[position]
            if( self.delimiters.find(c) != -1 ):
                break
            position+=1
        return position

    def next_token(self):
        """取得下一个标记"""
        self.currentPosition = self.skip_delimiters(self.currentPosition)
        start = self.currentPosition
        self.currentPosition = self.scan_token(self.currentPosition)
        return self.text[start: self.currentPosition]

    def __iter__(self):
        """这个方法用于支持以迭代的方式返回符号"""
        while True:
            token = self.next_token()
            if(token==None):
                break
            yield token
```

使用 StringTokenizer 类：

```
it = iter(StringTokenizer("2#7#8#9#道","#"))
for x in range(5):  #取得 5 个标记
   print(next(it))
```

2.16.1　zip 函数

函数 zip()遍历多个可迭代的对象，并聚合它们成为一个可迭代的 zip 对象。例如，把

两个列表聚合成一个：

```
x = [1,2,3,4]
y = [7,8,3,2]
z = ['a','b','c','d']

for a,b in zip(x,y):
   print(a,b)
```

输出如下：

```
1 7
2 8
3 3
4 2
```

也可以聚合两个以上列表：

```
for a,b,c in zip(x,y,z):
   print(a,b,c)
```

输出如下：

```
1 7 a
2 8 b
3 3 c
4 2 d
```

可以使用计数器统计 zip 对象中元素出现的频次：

```
from collections import Counter

x = [1,2,3,4,4]
y = [7,8,3,2,2]

print(Counter(zip(x,y)))
```

输出如下：

```
Counter({(4, 2): 2, (1, 7): 1, (2, 8): 1, (3, 3): 1})
```

Counter.most_common()方法返回出现频次最高的 *n* 个元素，例如返回最常见的 3 个元素：

```
print(c.most_common(3))
```

输出如下：

```
[((4, 2), 2), ((1, 7), 1), ((2, 8), 1)]
```

2.16.2 itertools 模块

itertools 模块实现了许多迭代器构建块。函数 itertools.islice()用于从可迭代对象取切片。

```
import itertools

def zigzag(period):
    while True:
        for n in range(period):
            yield n
        for n in range(period, 0, -1):
            yield n

it1 = zigzag(5)                          #创建一个无限长度的可迭代对象
it2 = itertools.islice(it1, 0, 20)    #取前 20 个元素
li = list(it2)
print(li)    #输出：[0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 4, 3, 2, 1]
```

使用函数 itertools.count()可创建自增 ID：

```
>>> import itertools
>>> id_iter = itertools.count()
>>> next(id_iter)
0
```

2.17　数据库

这里以数据库 MariaDB 为例，介绍 Python 的数据库客户端。

安装数据库 MariaDB：

```
#sudo apt-get install mariaDB-server mariaDB-client
```

以 MariaDB root 用户身份登录：

```
#sudo mysql -u root -p
```

输入密码后，将进入 MariaDB shell。

如果要启动 MariaDB，可以使用以下命令。

```
#sudo systemctl start mariadb
```

可以使用以下命令停止 MariaDB：

```
#sudo systemctl stop mariadb
```

安装 Python MySQL 客户端：

```
#python3 -m pip install PyMySQL
```

连接数据库：

```
import pymysql.cursors

#Connect to the database
```

```
    connection = pymysql.connect(host='localhost',
                                 user='user',
                                 password='',
                                 db='mysql',
                                 charset='utf8mb4',
                                 cursorclass=pymysql.cursors.DictCursor)
```

如果出现错误“pymysql.err.InternalError: (1698, "Access denied for user 'root'@'localhost'")”，则可以考虑使用 mysql_native_password 插件设置用户。

```
    $ sudo mysql -u root
    MariaDB [(none)]> USE mysql;

    MariaDB [mysql]> UPDATE user SET plugin='mysql_native_password' WHERE
User='root';
    Query OK, 1 row affected (0.00 sec)
    Rows matched: 1  Changed: 1  Warnings: 0

    MariaDB [mysql]> FLUSH PRIVILEGES;
    Query OK, 0 rows affected (0.00 sec)

    MariaDB [mysql]> exit;
    Bye
```

重新启动 MariaDB 服务：

```
    #sudo systemctl restart mariadb
```

在以下示例中，获取 MariaDB 的版本信息。

```
    import pymysql

    con = pymysql.connect('localhost', 'root',  '', 'mysql')

    with con:

        cur = con.cursor()
        cur.execute("SELECT VERSION()")  #使用游标执行 SQL 语句
        version = cur.fetchone()

        print("Database version: {}".format(version[0]))
```

输出：

```
    Database version: 10.1.38-MariaDB-0ubuntu0.18.04.2
```

插入数据和查找数据的例子：

```
    import pymysql.cursors

    #连接到数据库
```

```
    connection = pymysql.connect(host='localhost',
                            user='root',
                            password='',
                                        db='mysql',
                                        charset='utf8mb4',
                                        cursorclass=pymysql.cursors.DictCursor)

    cursor=connection.cursor()

    cursor.execute("CREATE TABLE IF NOT EXISTS results (dataset text, wer float)")
    cursor.execute('INSERT   INTO   results(dataset,   wer)   VALUES(%s,   %s)',
("LibriSpeech", 0.0583))

    connection.commit()  #提交更新

    cursor.close()

    cursor=connection.cursor()

    cursor.execute("select dataset, wer from results;")
    rows = cursor.fetchall()

    for row in rows:
       print(row["dataset"], row["wer"])
```

可以将数据库连接参数写入配置文件。Python 标准库中的 configparser 模块定义了用于读取和写入 Windows 操作系统使用的配置文件的功能。此类文件通常具有.ini 扩展名。INI 格式文件由 section 下的键/值对组成。sampleconfig.ini 文件内容示例如下：

```
    [SectionName]
    keyname1=value
    ;comment
    keyname2=value
```

以下脚本读取并解析 sampleconfig.ini 文件：

```
    import configparser
    parser = configparser.ConfigParser()
    parser.read('sampleconfig.ini')
    for sect in parser.sections():
       print('Section:', sect)
       for k,v in parser.items(sect):
          print(' {} = {}'.format(k,v))
       print()
```

2.18 日志

Python 在标准库中随附了一个日志记录模块。该模块提供了一个灵活的框架从 Python 程序中发出日志消息。

日志记录模块的最重要的对象如下：

- Logger：实际的日志接口。
- Handler：处理日志语句并输出日志。
- Formatter：将输入数据格式化为字符串。
- Filter：允许过滤某些消息。

使用日志记录可以实现：

- 为消息设置不同的级别，例如 info()、debug()、error()、warning()。
- 添加消息的不同格式，即放置日期时间戳、文件名、进程、路径，以及更多 LogRecord 属性。
- 同时输出日志到文件、套接字等。

根据需要在 Python 程序中跟踪的消息或事件的严重性，可以使用不同的预定义级别。日志级别见表 2-4。

表 2-4 日志级别

级　别	数　值	什么时候使用
CRITICAL	50	严重错误，表明程序本身可能无法继续运行
ERROR	40	由于存在更严重的问题，该软件无法执行某些功能
WARNING	30	表示发生了意外情况，或者表示在不久的将来会出现一些问题（例如“磁盘空间不足”）。该软件仍按预期运行
INFO	20	确认一切正常
DEBUG	10	详细信息，通常仅在诊断问题时才需要
NOTSET	0	创建日志时，级别设置为 NOTSET

日志的最低级别为 DEBUG，最高级别为 CRITICAL。因此，建议始终将日志级别定义为最小值，即 DEBUG，以便在需要时也可以使用较高的日志级别。

在如下的 Python 脚本中，导入了 logging 模块，但尚未定义任何级别。

```
#!/usr/bin/env python3

import logging

logging.debug('debug')
logging.info('info')
logging.warning('warning')
logging.error('error')
logging.critical('critical')
```

当执行这个脚本时，只得到了函数 warning()、error()和 critical()的输出。这是因为默认日志记录级别设置为 warning()，并且脚本将仅考虑所有具有与 warning()级别相等或更高数值的级别。

```
#python3 /tmp/logging_ex.py
WARNING:root:warning
ERROR:root:error
CRITICAL:root:critical
```

在上面例子的日志输出中得到了“root”。日志层次结构的根称为根日志。这就是函数 debug()、info()、warning()、error()和 critical()使用的日志，它们只是调用根日志的同名方法。根日志的名称在日志的输出中打印为“root”。