第 1 章
Python 自然语言处理技术基础

一个组织往往使用一句话的愿景来描述组织或计划工作产生的明确和鼓舞人心的长期期望变化。自然语言处理是人工智能和语言学的交汇点。它涉及书面和口头语言的智能分析。可以从自然语言提取知识，形成知识库。机器系统与智慧生物的交互也需要自然语言处理技术。

作为人工智能的一部分，自然语言处理技术通过用户讲的话理解用户。相对于往往可以用上下文无关文法来处理的计算机编程语言而言，自然语言更灵活，因此往往借助一些统计和机器学习的方法。

目前，自然语言处理技术广泛应用在人们生活、学习和工作的各个方面，给人们带来了方便。

自然语言处理技术包括很多方面，如文本分类、文本摘要、对话系统、机器翻译、语音识别、语音合成等。社交软件可以使用自然语言处理技术来分析用户意图。

在搜索引擎技术中，需要更好地理解文本信息与用户输入。严格来说，自然语言处理（Natural Language Processing）包括自然语言理解和自然语言生成两部分。考虑到自然语言理解的基础地位，本书主要涉及自然语言理解部分。本书介绍使用流行的 Python 编程语言实现自然语言理解技术。

本章首先体验一些现成的自然语言理解技术实现，然后介绍 Python 基础知识。

1.1 体验自然语言处理技术

本节的介绍是通过浏览器来体验自然语言处理技术。

可以通过谷歌翻译体验机器翻译技术，将源语言文本翻译成目标语言文本。在浏览器中输入网址：https://translate.google.cn。目前将“有益的微生物”翻译成“Beneficial microorganism”，

将“你好，朋友”翻译成“your good friend”。

智能处理原始文本很困难：大多数单词很少见，而且看起来完全不同的单词通常意味着几乎相同的东西。例如，土豆又名马铃薯。不同顺序的相同单词可能意味着完全不同的东西，例如，“In the middle of the flower”和“Flower in the middle”。

通过一个Python语言开发的开源自然语言处理软件包spaCy(https://spacy.io)来体验对输入文本中的单词做标注。

在浏览器中输入网址：https://spacy.io/usage/linguistic-features。把要运行的代码修改成：

```
import spacy

nlp = spacy.load('en_core_web_sm')
doc = nlp(u'In the middle of the flower')   #待分析的文本

for token in doc:
   print(token.text, token.lemma_, token.pos_)
```

输出结果：

```
In In ADP
the the DET
middle middle NOUN
of of ADP
the the DET
flower flower NOUN
```

输出的第一列是词本身，第二列是词原型，第三列是词性。这里用编码来表示词性。英文词性编码表如表 1-1 所示。

表 1-1　英文词性编码表

代　码	名　称	代　码	名　称
NOUN	名词	NUM	数词
ADJ	形容词	ADP	介词
ADV	副词	PUNCT	标点符号
DET	冠词	INTJ	感叹词
ADP	所有格	PART	助词
PRON	代词	PROPN	专有名词
AUX	情态助动词	SCONJ	从属连词
CONJ	连接词	SYM	符号
VERB	动词	X	其他

1.2　Linux 基础

虽然很少人用 Linux 操作系统办公，但是很多自然语言处理应用都可以运行在 Linux 操作系统。Linux 有一些常用的发行版：CentOS 和 Ubuntu 等版本。Ubuntu 是由 Canonical 公司开发的基于 Debian 的开源 Linux 操作系统。CentOS 是 Red Hat Enterprise Linux 的免费克隆版。

可以通过在终端输入的各种命令和 Linux 操作系统打交道。为了远程登录 Linux 服务器，可以安装 Telnet 和 SSH 客户端 KiTTY(https://www.fosshub.com/KiTTY.html)。在 KiTTY 的配置界面输入 IP 地址、用户名和密码后登录 Linux 服务器。如果是用 root 账户登录，则终端提示符是#，否则终端提示符是$。

1.2.1　常用命令

通过输出文本文件/etc/issue 的内容来查看 Ubuntu 操作系统版本号：

```
$cat /etc/issue
Ubuntu 18.04 LTS \n \l
```

或者：

```
$ lsb_release -r
Release:       18.04
```

获取 Ubuntu 的代号：

```
$ lsb_release -c
Codename:      bionic
```

Ubuntu 操作系统可以使用 apt-get 安装软件，例如安装下载工具 wget：

```
# apt-get install wget
```

在 CentOS 下则可以使用黄狗升级管理器（Yellowdog Updater, Modified，一般简称 yum）安装软件包。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。例如安装 wget：

```
#yum install wget
```

使用 ls 命令列出当前目录下的文件。有的命令比较长，为了快速输入，可以用 Tab 键补全命令。history 显示历史命令。用上箭头选择最近运行过的命令再次执行。

Linux 手册页（man page）有助于解释每个命令可以做什么。可以使用 man 命令查看一个命令的手册页。这里的 man 是 manual 的缩写。例如，查看 touch 命令的用法：

```
# man touch
```

Linux 操作系统中的文件名区分大小写。例如，用 touch 命令创建两个文件：

```
# touch aaa.txt
# touch AAA.txt
# ls
aaa.txt
AAA.txt
```

文件和目录的模式定义了文件的可读写和可执行的权限。有三种基本模式：可读(r)，可写(w)和可执行(x)。另外，这些模式中的每一个都可以应用于用户、组或每个人。

查看当前目录下文件 build.sh 的属性：

```
# ls -lh build.sh
-rwxr-xr-x 1 root root 3.2K Dec 24  2017 build.sh
```

输出表明任何人都可以执行 build.sh。

给文件 myfile 增加可执行权限。

```
# chmod +x myfile
```

要让文件可执行且可由任何用户运行：

```
# chmod a+x myfile
```

开发自然语言处理应用时，经常会用到文本处理，awk 和 sed 命令是两个可以用来处理文本的工具。

可以把 awk 命令当作一个逐行处理文本的工具。awk 命令解释执行专门的编程语言。awk 中的代码由模式和动作组成。模式用来匹配输入，动作用来得到想要的输出。使用 awk 命令的基本形式是：

```
#awk 模式 { 动作 }
```

例如，输出 Python 的版本号：

```
# python3  --version | awk '{print $2}'
3.6.8
```

这里只用到了动作，而省略了模式。

可以使用 sed(Stream EDitor)命令查找和替换文件中的文本。例如：

```
# sed -i 's/original/new/g' file.txt
```

命令行参数说明如下：

- -i = in-place（保存回原始文件）

命令字符串说明如下：

- s =替换命令
- original =描述要替换的单词的正则表达式（或者只是单词本身）
- new =用来替换的目标文本
- g = global（即，替换所有而不仅仅是第一次出现）

- file.txt =文件名

例如把 cmd.sh 中的 queue.pl 替换成为 run.pl，替换结果输出到 cmd.local.sh 文件。

```
# sed 's/queue.pl/run.pl/g' cmd.sh > cmd.local.sh
```

1.2.2　Micro 编辑器

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

可以在/home/soft/micro 目录下运行：

```
# curl https://getmic.ro | bash
```

设置成在任意路径均可运行 Micro：

```
#cd /usr/bin
#sudo ln -s /home/soft/micro/micro micro
```

或者编辑/etc/profile 文件，增加 micro 所在的路径到 PATH 环境变量/home/soft/micro。

```
# ./micro /etc/profile
```

增加如下行：

```
export PATH=/home/soft/micro:$PATH
```

可以使用它编辑源代码文件 run.py：

```
#./micro  run.py
```

输入：

```
print("Hello nlp")
```

保存文件后，按 Ctrl-Q 组合键退出。

运行 run.py：

```
# python3  run.py
```

1.3　开发环境

Python 软件基金会维护的 Python 语言代码解释器可以从 Python 官方网站 https://www.Python.org 下载。

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

```
d:\data>python
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.1916 64 bit
(AMD64)] on win32
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

由于开源软件的迅速发展，可以借助开源软件简化自然语言处理开发工作。简而言之，可以使用 Sublime 这样的文本编辑器写 Python 代码，也可以使用 Eric(https://eric-ide.Python-projects.org) 或者 Microsoft Visual Studio 这样的集成开发环境。

1.4 变量

定义变量时不声明类型，但变量在内部是有类型的。在交互式环境输入的如下代码将输出变量 a 的类型：

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

1.5 注释

和 Shell 类似，Python 脚本中用#表示注释。但如果#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个语句。

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

```
# -*- coding: utf-8 -*

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

1.6 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。

1.6.1 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）。

和 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=3243244444444444444444444444444444444444444444487976875675676570000000000000
0000000000000000000000000000000000000000000000000000000000000564564
>>> i
3243244444444444444444444444444444444444444444487976875675676570000000000000000000000
0000000000000000000000000000000000000000000000000000000000000564564
>>> 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 模块通过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)
```

有若干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如表 1-2 所示。

表 1-2 用于数值运算的算术运算符说明

语 法	数 学 含 义	运算符名字
a+b	a+b	加
a-b	a−b	减
a*b	a×b	乘法
a/b	a ÷ b	除法
a//b	$\lfloor a \div b \rfloor$	地板除
a%b	a mod b	模
-a	-a	取负数
abs(a)	|a|	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，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，则另一个操作数将被转换为复数：

```
>>> 3.0 + (5+6j)   # 非复数转复数
(8+6j)
```

整数转为浮点数：

```
>>> 6 + 7.0 # 非浮点型转浮点型
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。例子代码如下：

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> total = a + \
... b + \
... c
>>> total
6
```

1.6.2　字符串

可以使用 strip()方法去掉字符串首尾的空格或者指定的字符。

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

使用 split()方法将句子分成单词。下面，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谨慎的单位。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()方法来把句子切分成单词：

```
>>> 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 len()函数演示的 5 个项目。对于 mary，len()函数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()可以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'，.split()只是在所有空格上分隔，如上所示。

```
>>> 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：在空格上分隔
['Hello', 'mother,', 'Hello', 'father.']
>>> hi.split('\n')                 #仅在'\n'上分隔
```

```
['Hello mother,', 'Hello father.']
```

但是如果你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。list()函数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。下面，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。下面，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 ord()函数和 chr()函数实现字符串和整数之间的互相转换：

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

1.6.3 数组

创建一个数组，然后向这个数组添加元素，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]
```

Python 中有三种不同的数值类型：int（整数）、float（浮点数）和 complex（复数）。

1.7　字面值

Python 包括如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。

有四种不同的字面值集合，分别是：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)
```

1.8　控制流

完成一件事情要有流程控制。例如，洗衣三个步骤：把脏衣服放进洗衣机、等洗衣机洗好

衣服再晾衣服。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。

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

控制流用来根据运行时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。

1.8.1 if 语句

若路径不存在，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：

```
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。关系运算符完整的列表如表 1-3 所示。

表 1-3　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
<	a < b	a 小于 b
<=	a <= b	a 小于或等于 b

续表

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

如果要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)
```

1.8.2　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 份副本。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 range()函数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')
```

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，当用户直接输入回车时退出循环：

```
import sys

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

1.9 列表

可以使用一个 List 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 表达式截断字符串：

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

1.10 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，定义后赋值就

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

打包和拆包可以合并为一个语句，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
```

可以使用命名元组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))
```

1.11 集合

可以使用 in 运算符来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)
Falsee
```

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}
```

set.intersection()方法可以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
```

1.12　字典

可以使用字典来存取键/值对。要访问字典元素，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'])
```

可以通过为该键指定值在字典上创建新的键/值对。如果键不存在，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'])
```

运行如下语句则会返回 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 # <== evaluates to True
True
>>> 'c' in d # <== evaluates to 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 collections.defaultdict()函数设置默认值：

```
>>> 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，由于字典本来是无序的，所以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)
```

1.13 位数组

位数组（也称为位图）通常用作快速数据结构。可以用位图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
```

1.14　模块

可以使用 import 语句导入一个.py 文件中定义的函数。一个.py 文件就称之为一个模块（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：

```
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。一个班里有两个叫作陈晨的同学。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，这样就能区分同名了。为了避免名字冲突，模块可以位于不同的命名空间，叫作包。可以在模块名前面加上包名限定，这样即使模块名相同，也不会冲突了。

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。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。

1.15 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，str()函数将对象转化为易于阅读的字符串。len()函数用于获取对象的长度。id()函数返回对象的标识。print()函数将给定对象打印到标准输出设备（屏幕）或文本流文件。

1.15.1 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。可以调用 str()函数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，以下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!
```

以下代码会出错：

```
>>> 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
```

print 函数用到的格式化字符串的约定如表 1-4 所示。

表 1-4　print 函数用到的格式化字符串的约定

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
u	不带符号的十进制
x	不带符号的十六进制（小写）
X	不带符号的十六进制（大写）
e	科学计数法表示的浮点数（小写）
E	科学计数法表示的浮点数（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）

1.15.2 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，所以我们使用 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*。示例代码如下：

```
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，可变长度的参数列表。例子代码如下：

```
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。我们可以在这里找到__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，如果你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——将它们打包在一个 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 main 函数。

main 函数的示例代码如下：

```
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()
```

1.16 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):
    """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隔输入字符串的 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())
```

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)
```

可以使用类调用 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)
```

1.17 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
```

1.17.1 读写文件

调用 open(fileName)函数返回一个_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 utf8 编码格式的文本文件：

```
import codecs
import sys

transcript = codecs.open(sys.argv[1], "r", "utf8")    #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 open()函数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 在打开文件之前已经存在，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，我们可以使用写入操作<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。如果想要在文件中开始新行，则必须明确提供换行符。示例代码如下：

```
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，可以使用<file>.close()函数关闭所有文件。

```
fout.close()
```

使用 open()函数创建文件对象可以使用的模式总结如下：

- 'r'用于读取现有文件（默认值，可以省略）。
- 'w'用于创建用于写入的新文件。
- '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'}
```

1.17.2 重命名文件

可以使用 os.rename 方法重命名文件。首先用 touch 命令创建一个空文件：

```
# touch ./test1
```

然后将 test1 重命名为 test2

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

1.17.3 遍历文件

使用 os.scandir 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():              #判断项目是否文件
        print(f'{entry.name}')
```

如果想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()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
```

1.18　迭代器

迭代器（iterator）用来遍历生成器中的元素。通过 next()函数迭代 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 next()函数抛出 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：

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

1.18.1 zip 函数

zip 函数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：

```
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：

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

1.18.2　itertools 模块

itertools 模块实现了许多迭代器构建块。itertools.islice()函数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]
```

可使用 itertools.count()函数创建自增 ID：

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

1.19 数据库

这里以 MariaDB 数据库为例，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 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()
```

1.20 读取 Excel 文件

待处理的文本存在于 Excel 文件中。openpyxl(https://bitbucket.org/openpyxl/openpyxl/)是一个用于读取/写入 Excel 2010 xlsx / xlsm / xltx / xltm 文件的 Python 库。

安装 openpyxl 模块：

```
# pip3 install openpyxl
```

首先，使用如下语句导入 openpyxl 模块：

```
>>> import openpyxl
```

如果没有错误消息，则表示 openpyxl 已正确安装，现在可以使用 Excel 文件。

接下来使用以下代码加载工作簿“testfile.xlsx”。

```
>>>wb= openpyxl.load_workbook('testfile.xlsx')
```

openpyxl.load_workbook()函数将文件名作为参数，并返回一个工作簿数据类型的对象。加载 testfile.xlsx 后，查看可用的类型或句柄：

```
>>> type (wb)
<class 'openpyxl.workbook.workbook.Workbook'>
```

要获取有关工作簿中的工作表数量及其名称的信息，请使用属性 sheetnames。此属性返回工作簿中工作表的名称。

```
>>> wb.sheetnames
['Sheet1', 'Sheet2', 'Sheet3']
```

知道名字后，我们可以同时访问任何表格。假设想要访问 Sheet2。

```
>>> sheet=wb['Sheet2']
>>> type(sheet)
<class 'openpyxl.worksheet.worksheet.Worksheet'>
>>> sheet.title
'Sheet2'
```

访问活跃的工作表。

```
>>> wb.active
<Worksheet "Sheet2">
```

为了从表格单元格访问数据，可以通过表格然后是单元格地址来引用。

```
>>> sheet['A2'].value
1
```

访问单元格数据的另一种方法是：

```
>>> e=sheet['B2']
>>> e.value
'KTV'
>>> e.row
2
>>> e.column
2
```

借助行和列从单元格中获取数据：

```
>>> sheet.cell(row=2, column=2)
<Cell 'Sheet2'.B2>
>>> sheet.cell(row=2, column=2).value
'KTV'
```

现在我们打印整列，而不是从列中获取一个值。使用迭代实现的打印整列代码如下：

```
>>> for x in range (1,9):
...     print(x,sheet.cell(row=x,column=2).value)
...
1 功能特征词
2 KTV
3 宾馆
4 部
5 参行
6 餐厅
7 长途客运站
8 长途汽车站
```

接下来打印多列：

```
>>> for y in range (1,9,1):
...     print(sheet.cell(row=y,column=1).value,sheet.cell(row=y,column=2).value)
...
ID 功能特征词
1 KTV
2 宾馆
3 部
4 参行
5 餐厅
6 长途客运站
7 长途汽车站
```

1.21 pytest 单元测试

pytest 是一个单元测试框架。在命令行中运行以下命令来安装 pytest：

```
#pip install -U pytest
```

检查是否安装了正确的版本：

```
>pytest --version
This is pytest version 4.0.2, imported from c:\python37\lib\site-packages\pytest.py
setuptools registered plugins:
  pytest-timeout-1.3.3 at c:\python37\lib\site-packages\pytest_timeout.py
```

用于测试的 test_sample.py 内容如下：

```
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

在 test_sample.py 所在的目录运行 pytest 命令，输出如下：

```
==================== test session starts ====================
platform win32 -- Python 3.7.2, pytest-4.0.2, py-1.8.0, pluggy-0.9.0
rootdir: E:\test, inifile:
plugins: timeout-1.3.3
collected 1 item

test_sample.py F                                           [100%]

====================== FAILURES ==========================
_____________________________________ test_answer _____________________________________

    def test_answer():
```

```
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:6: AssertionError
================ 1 failed in 0.13 seconds ======================
```

其中，失败的单元测试结果以红色输出。

1.22　异常处理

可以在运行时检查可能发生问题的代码是否有异常发生。因为代码包装在 try 关键词中，所以叫作 try 代码块。在 try 代码块中捕捉异常，而在 except 代码块中处理异常。这样把异常处理代码和正常的流程分开，使正常的处理流程代码能够连贯在一起。

except 代码块又叫作异常处理器，它的常见格式是：

```
try:
       //执行可能抛出异常的代码
except ExceptionClass as e:  //异常类型
       //处理代码
```

使用 raise 语句抛出异常。以下代码可捕捉路径创建中的异常：

```
import errno
import os

output_dir = "d:/test"

try:
       os.makedirs(output_dir)
except OSError as e:
       if e.errno == errno.EEXIST and os.path.isdir(output_dir):
          print("路径已经存在");
          pass
       else:
          raise e
```

1.23　日志

搜索引擎可以通过日志文件把用户查询词记录下来。语音识别系统也可以把语音识别结果写入日志。可以使用日志来追踪软件运行时所发生事件。一个事件可以用一个可包含可选变量

数据的消息来描述。Python 日志记录模块定义了为应用程序和库实现灵活事件日志记录系统的函数和类。

级别用于标识事件的严重性。Python 中有六个日志级别，每个级别都与一个表示日志严重性的整数相关联：NOTSET = 0，DEBUG = 10，INFO = 20，WARN = 30，ERROR = 40，CRITICAL = 50。

如果日志记录级别设置为 WARNING，则所有 WARNING、ERROR 和 CRITICAL 消息都将写入日志文件或控制台。如果将其设置为 ERROR，则仅记录 ERROR 和 CRITICAL 消息。

可以立即使用日志模块而无须任何配置。simple.py 将实现简单的日志记录，内容如下：

```
import logging

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

该示例调用日志记录模块的 5 种方法。控制台将输出如下消息。

```
# simple.py
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

可以使用 logging.getLogger()函数得到一个命名的 Logger 对象。

```
import logging
Log = logging.getLogger("my-logger")
Log.info("Hello, world")
```

如果在模块中记录日志，则可以使用模块名字命名 Logger：

```
import logging

Log = logging.getLogger(__name__)   # __name__属性包含当前模块的全名

def do_something():
   Log.debug("Doing something!")
```

通过 Logger.setLevel()方法设置级别：

```
Log = logging.getLogger('myLogger')
Log.setLevel(logging.DEBUG)
```

可以在 setLevel()方法中使用变量：

```
level = logging.getLevelName('INFO')  #得到 level 变量的值
Log.setLevel(level)
```

当 Logger 根据级别检查确定事件是否应该通过时（例如，如果日志级别低于 Logger 级别，则将忽略该事件），Logger 使用其“有效级别”而不是当前级别。 如果级别不是 NOTSET，则

有效级别与 Logger 级别相同，即从 DEBUG 到 CRITICAL 的所有值；但是，如果 Logger 级别为 NOTSET，则有效级别将是具有非 NOTSET 级别的第一个祖先级别。

以下将输出数值形式的有效级别：

```
>>> Log.getEffectiveLevel()
20
```

日志处理器是有效写出/显示日志的组件。以下将增加一个控制台日志处理器。

```
>>> consoleHandler = logging.StreamHandler()
>>> consoleHandler.setLevel(logging.INFO)  #设置日志级别
>>>
>>> Log.addHandler(consoleHandler)
>>>
>>> formatter = logging.Formatter('%(asctime)s  %(name)s  %(levelname)s: %(message)s')
>>> consoleHandler.setFormatter(formatter)  #设置日志输出格式
>>>
>>> Log.info('information message')
2019-07-08 17:07:34,060  myLogger  INFO: information message
INFO:myLogger:information message
```

以下将增加一个文件日志处理器。

```
>>> fileHandler = logging.FileHandler('test.log')
>>> fileHandler.setLevel(logging.INFO)
>>> Log.addHandler(fileHandler)
>>> Log.info('information message')
2019-07-09 09:11:36,336  myLogger  INFO: information message
INFO:myLogger:information message
```

查看 test.log 文件所在的位置：

```
>>> import os
>>> print(os.getcwd())    #返回当前工作目录
C:\Users\Administrator\AppData\Local\Programs\Python\Python37
```

1.24　Flask Web 框架

Flask 是一个轻量级的 WSGI（Web 服务器网关接口）Web 应用程序框架。

在 Linux 下安装 Flask：

```
# pip3 install -U Flask
```

hello.py 文件中一个完整的 Flask 应用程序如下所示。

```
# cat ./hello.py
from flask import Flask
```

```
app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, World!'
```

然后启动服务：

```
# env FLASK_APP=hello.py flask run
 * Serving Flask app "hello.py"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

1.25 本章小结

自然语言就是人交流用的语言。自然语言处理的研究方向主要包括：文本朗读、语音合成、语音识别、中文自动分词、词性标注、句法分析、自然语言生成、文本分类、信息检索、信息抽取、文字校对、问答系统、机器翻译以及自动摘要等。其中，信息检索方向最成熟。

Python 于 20 世纪 80 年代后期由荷兰的 Guido van Rossum 构思。Python 2.0 于 2000 年 10 月 16 日发布，具有许多主要的新功能，包括循环检测垃圾收集器和对 Unicode 的支持。Python 3.0 于 2008 年 12 月 3 日发布。它是该语言的一个重要修订，并非完全向后兼容。它的许多主要功能都被反向移植到 Python 2.6.x 和 2.7.x 版本系列。Python 3 的发布包括 2to3 实用程序，它可以自动（至少部分地）将 Python 2 代码转换为 Python 3。

除了官方的解释器，Python 代码也可以运行于 GraalVM。GraalVM 是一种高性能，可嵌入的多语言虚拟机。

Python 是一种多范式编程语言。Python 完全支持面向对象的编程和结构化编程，其许多功能支持函数编程和面向切面编程。

Python 的名字源于英国喜剧组织 Monty Python（巨蟒）。Monty Python 引用经常出现在 Python 代码和文化中。例如，Python 中经常使用的伪变量是 spam 和 eggs，而不是传统的 foo 和 bar。

当 Aaron Swartz 在 reddit.com 网站工作时，开发了 web.py。使用 web.py 的 reddit.com 发展成为 Alexa 的前 1000 个网站之一，并提供数百万的每日页面浏览量。

自然语言是文化的载体。可以根据文化，增加额外的标注信息。在许多语言中，即使将文本拆分为有用的类似单词的单元也很困难。虽然从原始字符开始可以解决一些问题，但通常最好使用语言知识来添加有用的信息。这正是 spaCy 旨在做的事情：输入原始文本，然后获得一个带有各种注释的 Doc 对象。

可以尝试使用 graalPython 在 graalvm 上运行 Python 代码。