第 1 章　网络爬虫快速入门

人工智能应用需要源源不断的数据来让应用跟上变化的世界。可以使用网络爬虫从互联网中自动获取数据。

在搜索引擎优化（SEO）领域，可以通过网络爬虫补充数据来实现更多的关键词收录。

网络爬虫经过最近几十年的快速发展，已经改变了人们获取信息的方式。从搜索引擎到推荐系统，都会用到网络爬虫技术。本书介绍采用流行的 Python 编程语言实现网络爬虫。

1.1　各种网络爬虫

有运行在大规模云计算平台的通用网络爬虫，还有一些行业垂直爬虫以及网站定向爬虫。通用网络爬虫是大鳄，每一只都有自己独立的领地。行业垂直爬虫是领头雁，是各行业的旗帜。而网站定向爬虫则像一只只小麻雀，麻雀虽小，五脏俱全。

1.1.1　通用爬虫

目前通用网络爬虫的组织方式主要有网络综合爬虫和网络主题资源爬虫两种。其中网络综合爬虫能够广泛地采集各互联网站点资源，并对其进行页面搜索，将索引结果存入索引数据库，供网络用户检索，并且能够提供互联网网络资源地导航功能的工具，如 Google、百度等。

Google、百度这样的公司需要大量的服务器和专业开发人员，运营开销大，如何在经济上可行就是一个问题。通用网络爬虫的主要收入是在搜索结果页中展示和用户输入的关键词相关的广告。条幅广告比关键词广告更早出现。按点击付费的关键词广告比条幅广告的收费额度低许多，点击一次广告可能只收几分钱，而条幅广告的计价单位至少在几百块。那些曾经被忽视的中小企业，一度被认为是游离在广告市场之外的客户，现在突然进入了互联网广告的生态系统。地球上最大的动物鲸鱼吃的是小鱼小虾，只有让更多的生物进入生态链，才能够产生庞大的顶级生物。

通用网络爬虫的企业是资本密集型企业，这样的公司往往前期有风险投资，有一定盈利后成为上市公司。

1.1.2 定向爬虫

垂直定向爬虫是针对某一个行业的专业爬虫，例如搜房（http://www.soufun.com/），39 健康网上的搜索。垂直搜索是搜索引擎的细分和延伸，是对网页库中的某类专门的数据进行处理后再以某信息进行一次整合，定向分字段抽取出需要的数据进行处理后再以某种形式返回给用户。

垂直爬虫需要从茫茫的互联网中获取行业信息，信息按行业过滤和分类是必不可少的。垂直搜索引擎和普通的网页搜索引擎的一个最大区别是对网页信息进行结构化信息抽取，也就是将网页的非结构化数据抽取成特定的结构化信息数据，好比网页搜索是以网页为最小单位，基于视觉的网页块分析是以网页块为最小单位，而垂直搜索是以结构化数据为最小单位。然后将这些数据存储到数据库中，并进行进一步的加工处理，如去重、分类等。最后分词、索引再以搜索的方式满足用户的需求。

整个过程中，数据由非结构化数据抽取成结构化数据，经过深度加工处理后以非结构化的方式和结构化的方式返回给用户。

垂直爬虫的应用方向很多，比如企业库爬虫、供求信息爬虫、购物爬虫、房产爬虫、地理信息爬虫、音乐爬虫、图片爬虫……几乎各行各业各类信息都可以进一步细化成各类的垂直爬虫。

垂直爬虫的技术评估应从以下几点来判断。

（1）全面性：应该能从众多的来源采集信息。

（2）更新性：用户最好可以在几秒或几分钟内看到最新发布的信息。

（3）准确性：数据分类准确，不能包含重复冗余信息。

（4）功能性：功能完善，可以同时搜索文字信息、图片、视频、地理信息等。

1.2 网络爬虫基本技术

一个基本的爬虫包括采集数据下载器和运行状况监控面板等部分。

网络爬虫（Crawler）的主要目的是为获取互联网上的信息。网络爬虫利用主页中的超文本链接遍历 Web，通过 URL 引用从一个 HTML 文档爬行到另一个 HTML 文档。http://dmoz.org 是整个互联网抓取的入口。网络爬虫收集到的信息可有多种用途，如建立索引、HTML 文件的验证、URL 链接验证、获取更新信息、站点镜像等。网络爬虫建立的页面数据库，包含有根据页面内容生成的文摘，这是一个重要特色。

网站本身可以声明不想被网络爬虫抓取的内容。可以有两种方式实现：第一种方式是

在站点增加一个纯文本文件，例如 http://www.baidu.com/robots.txt；另外一种方式是直接在 HTML 页面中使用 robots 的 meta 标签。在抓取网页时大部分网络爬虫会遵循 robot.txt 协议。

1.3　Windows 命令行

为了提高学习和工作效率，需要学会使用一些有用的软件工具，这里先介绍 Windows 命令行的使用。

假设有一个标准件工厂，在车间生产产品，在工地使用这些产品，类似地，往往在集成开发环境中开发软件。如果在 Windows 操作系统中运行开发的软件，则往往通过 Windows 命令行来运行。

在图形化用户界面出现之前，人们就是用命令行来操作计算机的。Windows 命令行是通过 Windows 系统目录下的 cmd.exe 执行的。可以在开始菜单的运行窗口直接输入程序名，回车后运行这个程序。打开开始→运行，这样就会打开资源管理器中的运行程序窗口。或者使用快捷键——窗口键 +R 键，打开运行程序窗口。总之，输入程序名 cmd 后单击确定，出现命令提示窗口。因为能够通过这个黑屏的窗口直接输入命令来控制计算机，所以也称为控制台窗口。

通常用扩展名来表示文件的类别，例如，exe 表示可执行文件。文件名称由文件名和扩展名组成。文件名和扩展名之间由小数点分隔，例如 calc.exe。

使用 rmdir 命令删除目录，例如删除 opencvsharp 目录：

```
D:\soft>rmdir /s opencvsharp
```

为了方便从命令行安装软件，可以先安装软件包管理工具软件 Chocolatey。使用如下命令安装 Chocolatey：

```
@"%SystemRoot%\System32\WindowsPowerShell\v1.0\powershell.exe" -NoProfile
-InputFormat None -ExecutionPolicy Bypass -Command "iex ((New-Object System.
Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))" && SET
"PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"
```

然后可以使用 choco 命令安装一些开发用的软件，例如安装 git：

```
>choco install git
```

当我们建立或修改一个文件时，必须向 Windows 指明这个文件的位置。文件的位置由三部分组成：驱动器、文件所在路径和文件名。路径是由一系列路径名组成的，这些路径名之间用“\”分开，例如：

```
C:\Windows\System32\calc.exe
```

开始的路径往往是 C:\Users\Administrator。公园的地图上往往会标出游客的当前位置，Windows 命令行也有个当前目录的概念，这个 C:\Users\Administrator 就是当前路径。

可以用 cd 命令改变当前路径，例如改变到 C:\Windows\System32 路径。

```
C:\Users\Administrator>cd C:\Windows\System32
```

如果写 cd d:，这样的效果是改变当前路径到 d: 子目录。所以切换盘符不能使用 cd 命令，而是直接输入盘符的名称，例如想要切换到 d 盘，可以使用如下命令：

```
C:\Users\Administrator>d:
```

执行一个可执行文件：

```
C:\Users\Administrator>C:\Windows\System32\calc.exe
```

也可以不指定可执行文件的路径，系统约定从指定的路径找可执行文件，这个路径通过 PATH 环境变量指定。环境变量是一个“变量名 = 变量值”的对应关系，每一个变量都有一个或者多个值与之对应。如果是多个值，则这些值之间用分号分开，例如 PATH 环境变量可能对应这样的值：“C:\Windows\system32;C:\Windows”，表示 Windows 会从 C:\Windows\system32 和 C:\Windows 两个路径找可执行文件。

设置或者修改环境变量的具体操作步骤是：首先在 Windows 桌面右击此电脑→属性→高级系统设置→环境变量，然后设置用户变量，或者系统变量，然后再设置环境变量 PATH 的值。

其实打开桌面上我的电脑就是运行资源管理器。打开资源管理器的另外一种方法是：首先按住键盘上的窗口键不放，然后再按 E 键。

需要重新启动命令行才能让环境变量设置生效。为了检查环境变量是否设置正确，可以在命令行中显示指定环境变量的值，例如显示 PATH 的值：

```
C:\Users\Administrator>PATH
```

为了从命令行修改环境变量，使用 Chocolatey 安装环境变量编辑器 Rapid Environment Editor。

```
>choco install rapidee
```

服务器的名称通过 DNS 服务器转换成对应的 IP 地址，也就是说，通过 DNS 取得该 URL 域名的 IP 地址。电脑需要选择一个好的 DNS 服务器，常用的有 114.114.114.114，或者 8.8.8.8。

使用 WMIC（Windows Management Instrumentation Command-line）设置 DNS：

```
wmic nicconfig where (IPEnabled=TRUE) call SetDNSServerSearchOrder ("114.114. 114.114")
```

为了快速访问网站，还可以直接在本机设置网站的 IP 地址，例如，首先得到 stackoverflow.com 的 IP 地址：

```
C:\Users\Administrator>nslookup stackoverflow.com
```

服务器：public1.114dns.com

Address：114.114.114.114

非权威应答 :

名称：stackoverflow.com

Addresses: 151.101.1.69

151.101.129.69

151.101.193.69

151.101.65.69

然后修改 HOSTS 文件，它其实就是一个文本文件。每行一个域名对应一个 IP 地址。

```
151.101.1.69      stackoverflow.com
```

1.4　上手 Scrapy 网络爬虫开发

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的开发工作。简单地，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。

Scrapy（https://github.com/scrapy/scrapy）是一个流行的爬虫框架。要安装 Scrapy，请在终端使用以下命令：

```
pip install Scrapy
```

Scrapy shell 是一个交互式 shell，可以在其中非常快速地尝试和调试爬虫代码。通常，

我们通过传递网页的 URL 来启动一个 shell，如下所示：

```
语法：scrapy shell <url_to_be_scraped>
```

例如：

```
scrapy shell http://quotes.toscrape.com/tag/friends/
```

获得网页标题：

```
In [1]: response.xpath('//title/text()').get()
Out[1]: 'Quotes to Scrape'
```

在浏览器中查看网页源代码，可以看到网页标题：

```
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Quotes to Scrape</title>
    <link rel="stylesheet" href="/static/bootstrap.min.css">
    <link rel="stylesheet" href="/static/main.css">
</head>
…
```

一旦我们学会了启动 shell，我们就可以用它来测试爬取代码。在编写任何 Python 爬虫代码之前，应该使用 shell 测试网页以进行抓取。Scrapy shell 有一些可用的快捷方式，一旦我们启动了 shell，它们就可用了。快捷方式介绍如下。

shelp()：shelp() 命令，显示 Scrapy 对象列表和有用的快捷方式。可以看到，Request 对象代表发送到链接 http://quotes.toscrape.com/tag/friends/ 的 GET 请求。此外，如果 Response 对象包含一个 200 HTTP 代码，表示请求成功，除此之外，它还提到了 Crawler 和 Spider 对象的位置。

fetch（URL）：“URL”是指向需要抓取的网页的链接。fetch 快捷方式接受一个 URL，即要抓取的网页，它返回爬虫信息，以及响应是成功还是失败。在下面的示例中，我们有一个有效的 URL 和一个无效的 URL，根据请求的性质，fetch 会显示错误或成功代码。

```
    In [4]: fetch('http://quotes.toscrape.com/tag/friends/')
    2022-07-27 09:56:31 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://quotes.
toscrape.com/tag/friends/> (referer: None)

    In [5]: fetch('http://quotes.toscrape.com/tafrnd/')
    2022-07-27 09:57:10 [scrapy.core.engine] DEBUG: Crawled (404) <GET http://quotes.
toscrape.com/tafrnd/> (referer: None)
```

fetch（request）：我们可以创建一个 Request 对象，并将其传递给 fetch() 方法，为此，需要创建一个 Scrapy 对象。Request 类提及到了所需的 HTTP 方法、网页的 URL、标头（如果有的话）。我们要抓取 URL=‘http://quotes.toscrape.com/tag/friends/’ 的网页，我们需要准

备请求对象为：

```
    fetch(request_object)
```

终端输入输出为：

```
    In [6]: url = 'http://quotes.toscrape.com/tag/friends/'

    In [7]: request = scrapy.Request(url,method='GET')

    In [8]: fetch(request)
    2022-07-27 11:07:40 [scrapy.downloadermiddlewares.retry] DEBUG: Retrying <GET
http://quotes.toscrape.com/tag/friends/> (failed 1 times): [<twisted.python.failure.
Failure twisted.internet.error.ConnectionLost: Connection to the other side was lost
in a non-clean fashion.>]
    2022-07-27 11:07:41 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://quotes.
toscrape.com/tag/friends/> (referer: None)
```

view（Response）：在默认浏览器中打开网页，网页是作为 Request 对象或 fetch() 方法中的 URL 发送的网页。当我们输入 view（Response）时，在上述 fetch（Request）之后，网页会在默认浏览器中打开。

1.5　本章小结

本章介绍了各种网络爬虫以及开发网络爬虫所需要的软件工具。最后，介绍了从 Scrapy shell 上手网络爬虫开发。

第 2 章　Python 开发快速入门

为了方便采用自底向上的方式实现网络爬虫，需要复习下 Python 编程基础，并准备好相关的算法基础，有了这些基础就可以实现简单的信息采集或者分布式网络爬虫系统。

需要熟练掌握 Python 开发方面的基础知识，才能使用 Python 实现网络爬虫。

2.1　变量

定义变量时不声明类型，但变量在内部是有类型的。在交互式环境下输入如下代码会输出变量 a 的类型：

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

2.2　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。例如：

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

from bs4 import BeautifulSoup
import requests
url = "https://www.tutorialspoint.com/index.htm"
req = requests.get(url)
soup = BeautifulSoup(req.text, "html.parser")  # 解析网页
print(soup.title)
```

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）。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=32432444444444444444444444444444444444444444448797687567567657000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000564564
>>> i
3243244444444444444444444444444444444444444444879768756756765700000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000564564
>>> 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 == 0.1000000000000000000000001
True
```

可以导入 decimal 模块并使用十进制表示完整的小数，例如：

```
>>> import decimal
>>> a = decimal.Decimal('0.1')
>>> b = decimal.Decimal('0.1000000000000000000000001')
>>> 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)
```

几个内置函数都支持复数运算：

```
>>> 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列表如表 2-1 所示。

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×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为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)    # 非复数转复数
(8+6j)
```

整数转换为浮点数：

```
>>> 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
```

2.3.2　字符串

在计算机编程中，字符串是一个字符序列，例如，"hello" 是一个包含字符序列 'h'、'e'、'l'、'l' 和 'o' 的字符串。

我们使用单引号或双引号来表示 Python 中的字符串，例如：

```
# create a string using double quotes
string1 = "Python programming"

# create a string using single quotes
string1 = 'Python programming'
```

可以通过三种方式访问字符串中的字符。

（1）索引。一种方法是将字符串视为列表并使用索引值，例如：

```
greet = 'hello'

# access 1st index element
print(greet[1]) # "e"
```

（2）负索引。与列表类似，Python 允许对其字符串进行负索引，例如：

```
greet = 'hello'

# access 4th last element
print(greet[-4]) # "e"
```

（3）切片。使用切片运算符冒号（:）访问字符串中的字符范围，例如：

```
greet = 'Hello'
```

```
# access character from 1st index to 3rd index
print(greet[1:4])  # "ell"
```

可以在 Python 中创建多行字符串，为此，我们使用三个双引号 """ 或三个单引号 '''，例如：

```
# multiline string
message = """
Never gonna give you up
Never gonna let you down
"""

print(message)
```

在上面的示例中，封闭三引号内的任何内容都是一个多行字符串。

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。下面的 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() 方法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割，如下所示。

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

```
>>> 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) 会连接列表 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示例代码如下：

```
>>> 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'
```

字符串插值是将变量的值替换为字符串中的占位符的过程，例如：

```
# Python program to demonstrate
# string interpolation

n1 = 'Hello'
n2 = 'GeeksforGeeks'

# f tells Python to restore the value of two
# string variable name and program inside braces {}
print(f"{n1}! This is {n2}")
```

2.3.3 数组

创建一个数组，然后向这个数组中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 包括如下几种类型的字面值。

（1）数字：整数、浮点数、复数；

（2）字符串：以单引号、双引号或者三引号定义字符串；

（3）布尔值：True 和 False；

（4）空值：None。

有 4 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)
```

2.5　控制流

完成一件事情要有流程控制，例如，洗衣的 3 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

当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 2-2 所示。

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
<	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)
```

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)
```

可以使用 range() 函数循环一组代码指定的次数。range() 函数返回一个数字序列，默认从 0 开始，默认以 1 递增，并以指定的数字结束，例如：

```
for num in range(1, 23):
   url = f"https://slickdeals.net/computer-deals/?page={num}"
   print(url)
```

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)
```

2.6　列表

可以使用一个列表（List）存储任何类型的对象，例如：

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print("list1[0]: ", list1[0])
print("list2[1:5]: ", list2[1:5])
```

输出：

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

此外，列表甚至可以将另一个列表作为项目，这称为嵌套列表。

```
my_list = ["mouse", [8, 4, 6], ['a']]    # 嵌套列表
```

使用 range 函数生成列表：

```
>>> list(range(10))                      # 从 0 开始到 10，步长为 1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(0, 30, 5))                # 从 0 开始到 30，步长为 5
[0, 5, 10, 15, 20, 25]
```

可以使用赋值运算符（=）来更改一个项目或项目范围。

```
odd = [2, 4, 6, 8]                       # 错误的值

odd[0] = 1                               # 改变第一项

print(odd)                               # 输出:  [1, 4, 6, 8]

odd[1:4] = [3, 5, 7]                     # 改变第 2 至第 4 项

print(odd)                               # 输出: [1, 3, 5, 7]
```

可以使用 append() 方法将一个项添加到列表中，或使用 extend() 方法添加多个项。

```
odd = [1, 3, 5]

odd.append(7)

print(odd)                               # 输出:  [1, 3, 5, 7]

odd.extend([9, 11, 13])

print(odd)                               # 输出:  [1, 3, 5, 7, 9, 11, 13]
```

可以使用 + 运算符来连接两个列表。* 运算符重复列表给定次数。

```
odd = [1, 3, 5]

print(odd + [9, 7, 5])                   # 输出:  [1, 3, 5, 9, 7, 5]

print(["re"] * 3)                        # 输出:  ["re", "re", "re"]
```

此外，我们可以使用方法 insert() 在所需位置插入一个项目，或者通过将多个项目挤压到列表的空白切片中来插入多个项目。

```
odd = [1, 9]
odd.insert(1,3)

print(odd)                          # 输出：  [1, 3, 9]

odd[2:2] = [5, 7]

print(odd)                          # 输出：  [1, 3, 5, 7, 9]
```

可以使用关键字 del 从列表中删除一个或多个项目。

```
my_list = ['p','r','o','b','l','e','m']

del my_list[2]                      # 删除一个项目

print(my_list)                      # 输出：  ['p', 'r', 'b', 'l', 'e', 'm']

del my_list[1:5]                    # 删除多个项目

print(my_list)                      # 输出：  ['p', 'm']
```

甚至可以完全删除列表。

```
del my_list                         # 删除整个列表

print(my_list)                      # 错误：列表未定义
```

可以使用 remove() 方法删除给定的项目，或使用 pop() 方法删除给定索引处的项目，也可以使用 clear() 方法清空列表。

```
my_list = ['p','r','o','b','l','e','m']
my_list.remove('p')

print(my_list)                      # 输出：  ['r', 'o', 'b', 'l', 'e', 'm']

print(my_list.pop(1))               # 输出：  'o'

print(my_list)                      # 输出：  ['r', 'b', 'l', 'e', 'm']

print(my_list.pop())                # 输出：  'm'

print(my_list)                      # 输出：  ['r', 'b', 'l', 'e']

my_list.clear()

print(my_list)                      # 输出：  []
```

最后，我们还可以通过为一个元素片段分配一个空列表来删除列表中的项目。

```
>>> my_list = ['p','r','o','b','l','e','m']
>>> my_list[2:3] = []
>>> my_list
['p', 'r', 'b', 'l', 'e', 'm']
>>> my_list[2:5] = []
>>> my_list
['p', 'r', 'm']
```

for–in 语句可以轻松遍历列表中的项目：

```
for fruit in ['apple','banana','mango']:
    print("I like",fruit)
```

为了复制出一个新的列表，可以使用内置的 list.copy() 方法（从 Python 3.3 开始提供）。

```
>>> old_list = [1, 2, 3]
>>> new_list = old_list.copy()
```

使用 new_list = my_list，实际上没有两个列表，赋值仅复制对列表的引用，而不是实际列表，因此 new_list 和 my_list 在赋值后引用相同的列表。

通常，我们只想收集符合特定条件的项目。下面，我们有一个单词列表，我们只想从中提取包含 'wo' 的单词，为此，我们需要先创建一个新的空列表，然后遍历原始列表以查找要放入的项目：

```
>>> wood = 'How much wood would a woodchuck chuck if a woodchuck could
chuck wood?'.split()
>>> wood
['How', 'much', 'wood', 'would', 'a', 'woodchuck', 'chuck', 'if', 'a',
'woodchuck', 'could', 'chuck', 'wood?']
>>> wolist = []                                    # 创建一个空的列表
>>> for x in wood:
        if 'wo' in x:
            wolist.append(x)                       # 向列表增加项目
>>> wolist
['wood', 'would', 'woodchuck', 'woodchuck', 'wood?']
```

打印列表的内容：

```
>>> mylist = ['x', 3, 'b']
>>> print('[%s]' % ', '.join(map(str, mylist)))
[x, 3, b]
```

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，赋值之后就不允许有修改。

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

包装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))
```

2.8 集合

可以使用运算符 in 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}                    # 创建集合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'}
```

2.9　字典

字典是另一种可变容器模型，且可存储任意类型的对象。要访问字典元素，可以使用熟悉的方括号和键来获取它的值。

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

如果需要根据字典中的值排序，由于字典本质上是无序的，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)
```

2.10　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。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 greet_person 的函数：

```
def greet_person(person, number=2):
    for greeting in range(number):
        print(f"Hello {person}! How are you doing today?")
# 1.
greet_person("Sara", 5)
# 2.
greet_person("Kevin")
```

输出结果如下：

```
Hello Sara! How are you doing today?
Hello Sara! How are you doing today?
Hello Sara! How are you doing today?
Hello Sara! How are you doing today?
Hello Sara! How are you doing today?
Hello Kevin! How are you doing today?
Hello Kevin! How are you doing today?
```

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，示例代码如下：

```
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。你会怎么做？你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 main 函数。

main 函数的示例代码如下：

```
def main():
    print("Hello World!")


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

在 Python 中，函数是一级对象。这意味着函数就像其他任何对象一样，可以从函数返回函数。

在下面的程序中，我们定义了两个函数：function1() 和 function2()。function1() 返回 function2() 作为返回值。

```
def function1():
    return function2

def function2():
    print('Function 2')

x = function1()
x()
```

2.11 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示例代码如下：

```
import re

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

从 re 模块直接导入 sub 函数的示例代码：

```
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。

Python 中的外部模块也可以使用包管理器 pip 下载和安装，例如，安装模块 bs4：

```
pip install bs4
```

另一方面，一些模块，例如 Math 模块，不需要安装，我们只需要在 import 后加模块名称。

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')
```

2.12　检查字符串是否包含子字符串

检查 Python 字符串是否包含子字符串的最简单方法是使用 in 运算符。

in 运算符用于在 Python 中检查数据结构的成员身份，它返回布尔值（True 或 False）。要使用 in 运算符检查字符串是否包含 Python 中的子字符串，我们只需在超字符串上调用它：

```
fullstring = "StackAbuse"
substring = "tack"

if substring in fullstring:
    print("Found!")
else:
    print("Not found!")
```

此运算符是调用对象的 __contains__ 方法的简写，也适用于检查列表中是否存在项。值得注意的是，它不是空安全的，因此如果 fullstring 指向 None，则会抛出异常：

```
TypeError: argument of type 'NoneType' is not iterable
```

为了避免这种情况，首先要检查它是否指向 None：

```
fullstring = None
substring = "tack"

if fullstring != None and substring in fullstring:
    print("Found!")
else:
    print("Not found!")
```

Python 中的 String 类型有一个名为 index() 的方法，可用于查找字符串中子字符串第一个出现的位置。

如果未找到子字符串，将引发 ValueError 异常，可以使用 try-except else 块来处理：

```
fullstring = "StackAbuse"
substring = "tack"

try:
    fullstring.index(substring)
```

```
except ValueError:
    print("Not found!")
else:
    print("Found!")
```

如果需要知道子字符串的位置，而不是只知道它在整个字符串中是否出现，则此方法非常有用。

String 类型有另一个方法 find()，它比 index() 更方便使用，因为我们不需要担心处理任何异常。

如果 find() 找不到匹配项，则返回 -1，否则返回较大字符串中子字符串的最左边索引。

```
fullstring = "StackAbuse"
substring = "tack"

if fullstring.find(substring) != -1:
    print("Found!")
else:
    print("Not found!")
```

如果希望避免捕获错误，那么该方法应该优于 index()。

正则表达式提供了一种更灵活（尽管更复杂）的方式来检查字符串的模式匹配。Python 附带了一个用于正则表达式的内置模块，称为 re。re 模块包含一个名为 search() 的函数，我们可以使用它来匹配子字符串模式：

```
from re import search

fullstring = "StackAbuse"
substring = "tack"

if search(substring, fullstring):
    print("Found!")
else:
    print("Not found!")
```

如果需要更复杂的匹配函数，如不区分大小写的匹配，则此方法是最好的，否则，对于简单的子字符串匹配用例，应该避免正则表达式的复杂性和较慢的速度。

2.13 面向对象编程

为了能够封装细节，需要抽象出对象。对象只是数据（变量）和作用于这些数据的方法（函数）的集合。类本质上是用于创建对象的模板。

就好像函数定义以关键字 def 开头一样，在 Python 中，可以使用关键字 class 定义一个类。

这是一个简单的类定义。

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

一个类创建一个新的本地命名空间，其中定义了所有属性。属性可以是数据或函数。其中还有一些特殊属性，这些属性以双下画线（__）开头，例如，__doc__ 为我们提供了该类的文档字符串，示例代码如下：

```
class MyClass:
   "This is my second class"
   a = 10
   def func(self):
          print('Hello')

print(MyClass.a)          # 输出: 10
print(MyClass.func)       # 输出:  <function MyClass.func at 0x0000000003079BF8>
print(MyClass.__doc__)    # 输出: This is my second class
```

可以根据类模板来创建对象，创建对象的过程类似于函数调用。

```
ob = MyClass()
```

这将创建一个名为 ob 的新实例对象。我们可以使用对象名称前缀来访问对象的属性。

```
ob.func()  # 输出:  Hello
```

您可能已经注意到了类中函数定义的 self 参数，但是我们将该方法简单地称为 ob.func() 而没有任何参数，它仍然奏效。这是因为，只要对象调用其方法，对象本身就作为第一个参数传递。因此，ob.func() 转换为 MyClass.func(ob)。

方法与对象实例或类相关联，函数则不是。当 Python 调度（调用）一个方法时，它会将该调用的第一个参数绑定到相应的对象引用（对于大多数方法，这个参数通常称为 self）。

在 Python 中，除了用户定义的属性外，每个对象都有一些默认的属性和方法。要查看对象的所有属性和方法，可以使用内置的 dir() 函数。执行以下脚本可以查看 ob 对象的所有属性：

```
ob = MyClass()

print(dir(ob))
```

在 Python 中，静态变量是在类的所有实例之间共享的变量，而不是每个实例唯一的变量，它有时也被称为类变量，因为它属于类本身而不是类的任何特定实例。

静态变量通常用一个值初始化，就像实例变量一样，但是可以通过类本身而不是通过实例来访问和修改它们，示例代码如下：

```
class Example:
    staticVariable = 5


print(Example.staticVariable)    # Prints '5'

instance = Example()
print(instance.staticVariable)   # Prints '5'

instance.staticVariable = 6
print(instance.staticVariable)   # Prints '6'
print(Example.staticVariable)    # Prints '5'

Example.staticVariable = 7
print(Example.staticVariable)    # Prints '7'
```

2.14 泛型

泛型类型允许我们定义可以处理不同数据类型的类、函数或方法，而无须事先指定确切的数据类型。当我们想要创建一个可以处理多种数据类型的单个实现时，这很有用。

要在 Python 中定义泛型类型，我们可以使用内置的模块 typing，它为 Python 3.5 及更新版本提供了一组类型提示。typing 模块定义了许多泛型类型，如 List、Tuple、Dict 和 Union，这些类型可用于定义泛型函数、方法和类。

在生活中，一般把同一个容器存放同一类东西，例如，棉签盒专门用来放棉签，糖果盒专门用来放糖果。可以使用泛型来检查集合中存储的数据类型，例如， List<str> 指定放字符串。

在这个基本示例中，我们将定义一个可以处理不同类型列表的泛型函数：

```
from typing import List, TypeVar

T = TypeVar('T')

def reverse_list(lst: List[T]) -> List[T]:
    return lst[::-1]
```

```
# Example usage
num_lst = [1, 2, 3, 4, 5]
str_lst = ['a', 'b', 'c', 'd', 'e']
print(reverse_list(num_lst))
print(reverse_list(str_lst))
```

定义一个可以使用不同类型值的泛型类：

```
from typing import TypeVar

T = TypeVar('T')

class Box:
    def __init__(self, value: T):
        self.value = value

    def get_value(self) -> T:
        return self.value

# Example usage
box1 = Box(10)
box2 = Box('Sling Academy')
print(box1.get_value())
print(box2.get_value())
```

创建一个通用的数据存储库类，该类可以处理不同的数据类型。

```
from typing import TypeVar, Generic, List

T = TypeVar('T')

class DataRepository(Generic[T]):
    def __init__(self):
        self.data = []

    def add_data(self, item: T) -> None:
        self.data.append(item)

    def remove_data(self, item: T) -> None:
        self.data.remove(item)

    def get_all_data(self) -> List[T]:
        return self.data

# Example usage
```

```
repo = DataRepository[int]()
repo.add_data(10)
repo.add_data(20)
repo.add_data(30)
print(repo.get_all_data()) # Output: [10, 20, 30]
repo.remove_data(20)
print(repo.get_all_data()) # Output: [10, 30]

repo2 = DataRepository[str]()
repo2.add_data('apple')
repo2.add_data('banana')
repo2.add_data('orange')
print(repo2.get_all_data()) # Output: ['apple', 'banana', 'orange']
repo2.remove_data('banana')
print(repo2.get_all_data()) # Output: ['apple', 'orange']
```

DataRepository 类使用泛型类型 T 来定义存储库可以存储的数据类型。泛型类型 T 确保添加到存储库的数据是正确的类型，泛型返回类型 List[T] 确保从存储库检索的数据也是正确的类型。

2.15 日志记录

网络爬虫的运行过程可能很长，为了监控运行状态，可以用日志记录。

```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug('crawling...')
```

日志级别大小关系为：CRITICAL > ERROR > WARNING > INFO > DEBUG > NOTSET，当然也可以自己定义日志级别。

处理器将日志记录发送到任何输出，这些输出用自己的方式处理日志记录。

例如，FileHandler 会获取日志记录并将其附加到文件中。

标准日志记录模块已经配备了多个内置处理器，如：

（1）可以写入文件的多个文件处理器（TimeRotated、SizeRotated、Watched）；

（2）StreamHandler 可以输出到 stdout 或 stderr 等流；

（3）SMTPHandler 通过电子邮件发送日志记录；

（4）SocketHandler 将日志记录发送到流套接字。

此外，还有 SyslogHandler、NTEventHandler、HTTPHandler、MemoryHandler 等处理器。

格式器负责将元数据丰富的日志记录序列化为一个字符串。如果没有提供，则有一个默认格式器。记录库提供的通用格式器类将模板和样式作为输入，然后可以为日志记录对象中的所有属性声明占位符。

举个例子，'%(asctime)s %(levelname)s %(name)s: %(message)s' 会生成类似这样的日志：

2017-07-19 15:31:13,942 INFO parent.child: Hello EuroPython。

请注意，属性消息是使用提供的参数对日志的原始模板进行插值的结果，例如，对于 logger.info("Hello %s", "Laszlo")，消息将是 "Hello Laszlo"。

TestStreamHandler.py 中的示例代码如下：

```
    import logging

    logger = logging.getLogger(__name__)
    logger.setLevel(logging.INFO)
    handler = logging.StreamHandler()
    handler.setLevel(logging.INFO)
    formatter = logging.Formatter('%(asctime)s [%(filename)s:%(lineno)s - %(funcName)s
- %(levelname)s ] %(message)s')
    handler.setFormatter(formatter)
    logger.addHandler(handler)

    string = ''
    logger.info("trainning... \n {0}".format(string))
```

输出：

```
    2022-06-24 22:01:28,465 [TestStreamHandler.py:12 - <module> - INFO ] trainning...
```

2.16　数据库

Python 编程语言具有强大的数据库编程功能。Python 支持各种数据库，如 SQLite、MySQL、Oracle、Sybase、PostgreSQL 等。数据库接口的 Python 标准是 Python DB-API，大多数 Python 数据库接口都遵循这个标准。

Python DB-API 包括的方法如下。

（1）module.connect()：连接到数据库，要连接的参数因模块而异。connect() 方法返回 Connection 对象或引发异常。

（2）Connection.cursor()：从连接生成游标对象。游标用于将 SQL 语句发送到数据库并获取结果。

（3）Connection.commit()：提交当前连接中所做的更改。如果要保存更改（如插入、

更新或删除），必须在关闭连接之前调用 commit()。未提交的更改从当前连接中可见，但从其他连接中看不到。

（4）Connection.rollback()：回滚（撤销）当前连接中所做的更改。如果遇到异常时要继续使用同一连接，则必须回滚。

（5）Connection.close()：关闭连接。当程序退出时，连接总是隐式关闭的，但手动关闭连接是个好主意，尤其是当代码可能在循环中运行时。

（6）Cursor.execute(statement)：对数据库执行 SQL 语句。

（7）Cursor.execute(statement, tuple)：对数据库执行 SQL 语句。如果要将变量替换到 SQL 语句中，请使用这种形式。

（8）Cursor.fetchall()：从当前语句获取所有结果。

（9）Cursor.fetchone()：只获取一个结果，返回元组，如果没有结果，则返回 None。

SQLAlchemy 是一个开放源码的 SQL 工具包和对象关系映射器。

为了与数据库交互，我们需要获得它的句柄。session 对象是数据库的句柄。session 类是使用 sessionmaker() 定义的，这是一个可配置的 session 工厂方法，它绑定到前面创建的引擎对象，示例代码如下：

```
    from sqlalchemy.orm import sessionmaker
    import sqlalchemy as sa
    from sqlalchemy import Column, Integer, String
    from sqlalchemy.ext.declarative import declarative_base

    #For learning purposes, we normally use a SQLite memory-only database for convenience.
    engine = sa.create_engine("sqlite:///:memory:")
    Session = sessionmaker(bind = engine)
    session = Session()

    Base = declarative_base()

    class Customers(Base):
        __tablename__ = 'customers'

        id = Column(Integer, primary_key=True)
        name = Column(String)
        address = Column(String)
        email = Column(String)

    Base.metadata.create_all(engine)

    c1 = Customers(name = 'Ravi Kumar', address = 'Station Road Nanded', email =
'ravi@gmail.com')
```

```
session.add(c1)
session.commit()
```

2.17　本章小结

本章介绍了使用 Python 开发网络爬虫所需要的 Python 基础。

Python 于 20 世纪 80 年代后期由荷兰的 Guido van Rossum 设计，作为 ABC 语言的继承者，能够处理异常并与阿米巴操作系统连接。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 Python 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 的代码转换为 Python 3。

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。

Monty Python 引用经常出现在 Python 的代码和文化中，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。