第 1 章　深度学习快速入门

来源于人工神经网络的深度学习技术随着可用于机器学习数据的积累而迅速发展。深度学习方法可以从大量的训练数据中自动学习出实例的特征，在语音对话、图像识别、自然语言处理等领域得到了广泛的应用。PyTorch 是一个流行的深度学习框架。

1.1　各种深度学习应用

深度学习在很多领域都有广泛的应用，以下列举一些主要应用领域。

- 计算机视觉。深度学习在图像分类、目标检测、人脸识别、图像生成等方面应用广泛，如 Google 的 Inception 网络、Facebook 的 ResNet 等。
- 自然语言处理。深度学习在自然语言处理领域应用广泛，如机器翻译、情感分析、文本分类、语音识别、问答系统等，典型应用有 Google 的 BERT、OpenAI 的 GPT 等。
- 语音识别。深度学习在语音识别领域应用广泛，如语音转文字、语音合成等，如百度的 DeepSpeech2 等。
- 推荐系统。深度学习在推荐系统领域应用广泛，如电商推荐、社交媒体推荐等，如 Netflix 的推荐系统、Amazon 的推荐系统等。
- 医疗健康。深度学习在医疗健康领域应用广泛，如医学影像分析、疾病预测、药物发现等，如 Google 的 DeepVariant、IBM 的 Watson Health 等。
- 金融。深度学习在金融领域应用广泛，如风险管理、信用评估、股票预测等，如汇丰银行的风险管理系统、花旗银行的股票预测系统等。
- 自动驾驶。深度学习在自动驾驶领域应用广泛，如图像识别、目标检测、行驶路径规划等，如 Waymo 的自动驾驶系统、Tesla 的自动驾驶系统等。

这只是深度学习应用领域的冰山一角，随着深度学习技术的不断发展，未来还将涉及更多领域。

1.2 准备开发环境

当前，很多语音识别应用是在 Linux 操作系统下开发的。Linux 是围绕 Linux 内核构建的免费和开源软件操作系统系列。Linux 来源于 UNIX，Linux 是 UNIX 操作系统的开放源代码实现。通常，Linux 以桌面和服务器使用的称为 Linux 发行版的形式打包。Linux 有一些常用的发行版，如 CentOS 和 Ubuntu 等。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。本节首先介绍在 Ubuntu 和 CentOS 下安装 Python，然后介绍在 Linux 下开发 Python 应用的编辑器 Micro。

在 Windows 下，可以使用 Notepad++ 这样的文本编辑器编写 Python 代码，也可以使用 PyCharm 或者 PyDev 集成开发环境编写代码。

1.2.1 Linux 基础

有些深度学习系统运行在 Linux 服务器中，为了远程登录 Linux 服务器，可以安装 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 $。

查看 Ubuntu 操作系统版本号的代码如下：

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

或者

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

获取 Ubuntu 的代号的代码如下：

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

ls 命令用于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 命令用于显示历史命令。用上方向键可以选择最近运行过的命令再次执行。

可以使用支持 SSH 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秘密，所以比较方便。除了 SecureCRT，还可以使用开源软件 PuTTY（http://www.chiark.greenend.org.uk/~sgtatham/putty），以及可以保存登录密码的 PuTTY Connection Manager。

在终端启动的进程断开连接后会停止运行。为了让进程继续运行，可以使用 nohup 命令。

如果需要安装软件，可以下载对应的 RPM 安装包，然后使用 RPM 命令安装。但操作系统对应的 RPM 安装包找起来往往比较麻烦。一个软件包可能依赖其他的软件包，为了安装一个软件，需要下载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（Yellowdog 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 类似于 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 和 rzsz 命令的软件的代码如下：

```
#yum install wget
#yum install lrzsz
```

Windows 格式文本文件的换行符为 \r\n，而 Linux 文件的换行符为 \n。

dos2unix 是将 Windows 格式文件转换为 Linux 格式的实用命令。dos2unix 命令其实就是将文件中的 \r\n 转换为 \n。

开发深度学习系统的过程中，可能会用到大量的数据文件。如果需要在 Linux 操作系统中维护同一个文件的两份或多份副本，除了保存多份单独的物理文件副本，还可以采用保存一份物理文件副本和多个虚拟副本的方法。这种虚拟的副本就称为链接。链接是目录中指向文件真实位置的占位符。在 Linux 中有两种不同类型的文件链接：符号链接和硬链接。

符号链接就是一个实实在在的文件，它指向存放在虚拟目录结构中某个地方的另一个文件。这两个通过符号链接在一起的文件，彼此的内容并不相同。

如果现有空间不够用，可以增加存储设备后扩容。首先用 lsblk 命令查看现有空间情况。在一个 Linux 账号中显示如下：

```
[root@localhost ~]# lsblk
NAME    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sr0      11:0    1 1024M  0 rom
xvda    202:0    0  100G  0 disk
├─xvda1 202:1    0    8G  0 part [SWAP]
└─xvda2 202:2    0   32G  0 part /mnt
xvde    202:64   0 1000G  0 disk
```

创建一个要扩展的目录，代码如下：

```
# mkdir /ext
```

加载文件系统到这个目录下，代码如下：

```
# mount /dev/xvde /ext
```

确认加载成功，代码如下：

```
[root@localhost ~]# df -m
Filesystem     1M-blocks  Used Available Use% Mounted on
/dev/xvda2         32752 32496       256 100% /
devtmpfs           32108     0     32108   0% /dev
tmpfs              32020     1     32020   1% /dev/shm
tmpfs              32020   186     31835   1% /run
tmpfs              32020     0     32020   0% /sys/fs/cgroup
tmpfs               6374     1      6374   1% /run/user/0
/dev/xvde        1007801    77    956509   1% /ext
```

对于大的文件，可以使用 wget 命令在后台下载，代码如下：

```
#wget -bc <path>
```

这里的参数 b 表示在后台运行，参数 c 表示支持断点续传。

可以在 Windows 操作系统中编辑文本文件，然后使用 perl 命令把 Windows 操作系统中的文本文件转换成 Linux 可以识别的格式：

```
#perl -p -e 's/\r$//' < winfile.txt > unixfile.txt
```

在 Ubuntu 操作系统中安装 Python3，代码如下：

```
#apt install python3
```

Ubuntu 系统上默认的 root 密码是随机生成的，而 root 权限是通过 'sudo' 命令授予的。可以在终端输入命令 sudo passwd，然后输入当前用户的密码，按 Enter 键，终端会提示输入新的密码并确认，此时的密码就是 root 的新密码。修改成功后，输入命令 su root，再输入新的密码即可。

1.2.2 Micro 编辑器

为了方便在服务器端开发 Python\Perl\Shell\C++ 相关应用，可以使用 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，代码如下：

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

设置成在任意路径均可运行 Micro，代码如下：

```
#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，代码如下：

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

增加如下行：

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

可以使用它编辑配置文件，代码如下：

```
#./micro  test.py
```

输入：

```
print("Hello World")
```

保存文件后，按 Ctrl+Q 组合键退出。

1.2.3　在 Linux 系统中安装 Python

检查 Python 3 是否已经正确安装，并确认其版本号，代码如下：

```
# python3 -V
Python 3.4.5
```

检查 Python 3 所在的路径，代码如下：

```
# which python3
/usr/bin/python3
```

如果使用 CentOS，可以使用 yum 命令安装 Python 3。首先查找可供安装的 Python 版本，代码如下：

```
# yum search python3
```

然后安装想要的版本，代码如下：

```
# yum install python36
```

如果使用 Ubuntu 操作系统，首先运行以下命令更新软件包列表并将所有系统软件升级到可用的最新版本。

```
# sudo apt-get update && sudo apt-get -y upgrade
```

然后安装 Pip 包管理系统，代码如下：

```
#sudo apt-get install python3-pip
```

1.2.4　选择 Python 版本

Linux 系统中有可能同时存在多个可用的 Python 版本。每个 Python 版本都对应一

个可执行二进制文件。可以使用 ls 命令查看系统中有哪些 Python 的二进制文件可供使用，代码如下：

```
$ ls /usr/bin/python*
```

python 命令执行 Python 2。可以使用 python3 命令执行 Python 3。如何使用 python 命令执行 Python 3 ？

一种简单又安全的方法是使用别名。将如下命令放入 ~/.bashrc 或 ~/.bash_aliases 文件中：

```
alias python=python3
```

最好在终端使用 'python3' 命令，在 Python 3.x 文件中使用 shebang 行 '#!/usr/bin/env python3'。

1.2.5 在 Windows 系统中安装 Python

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。执行这个程序最直接的方式是找到这个程序，然后双击。因为 cmd.exe 并没有一个桌面的快捷方式，所以，这样太麻烦。

选择“开始”→“运行”命令，或者按窗口 +R 组合键，弹出“运行”对话框。输入 cmd 后单击“确定”按钮，出现命令提示窗口。因为能够能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。

开始的路径往往是 C:\Users\Administrator。Windows 命令行也有个当前目录的概念，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:\Python\Python37 路径，代码如下：

```
C:\Users\Administrator>cd C:\Python\Python37
```

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 环境变量可能对应如下值：“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如下：

（1）在 Windows 桌面上右击“此电脑”图标，在弹出的快捷菜单中选择“属性”命令。

（2）在“系统属性”界面中单击“高级系统设置”按钮。

（3）在“高级选项”模块中单击“环境变量”按钮。

（4）弹出“环境变量”对话框，可以看到“用户变量”和“系统变量”两个模块。如

果想修改或创建系统环境变量，可单击相应的“新建”按钮或选择已有的变量后单击“编辑”按钮进行修改。

（5）在“新建”对话框中输入变量的名称及变量值。变量值通常是文件路径，也可以是一个固定的值。

可以用如上步骤设置环境变量 PATH 的值。

重新启动命令行才能让环境变量的设置生效。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。这时需要用到 echo 命令。echo 命令用来显示一段文字，例如：

```
C:\Users\Administrator>echo Hello
```

执行上述命令，将在命令行输出 Hello。

如果要引用环境变量的值，可以用两个百分号把变量名包围起来，即“% 变量名 %”，代码如下：

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

假设把 Python 安装在 D:\Python\Python37 目录下，则可以在计算机属性中手工设置 PATH 环境变量，然后检查环境变量的值，代码如下：

```
C:\Users\Administrator.PC-201909301458>echo %PATH%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\
WindowsPowerShell\v1.0\;D:\apache-maven-3.5.2\bin;D:\Python\Python37
```

如下命令用于检查 Python 是否正确安装，以及所使用的版本号：

```
>python --version
```

或者用 where 命令检查系统是否已经安装了 Python。

```
C:\Users\Administrator>where python
C:\Python27\python.exe
D:\cygwin64\bin\python
D:\Programs\Python\Python37\python.exe
```

如果没有安装 Python，则可以使用 Chocolatey（https://chocolatey.org）安装 Python3 代码如下：

```
>choco install python3
```

或者访问 Python 的官方网站（https://www.python.org/）并下载最新版本。当前最新版本是 Python 3.12。

```
wget https://www.python.org/ftp/python/3.12.2/python-3.12.2-amd64.exe
```

确保选择“添加 python.exe 到 PATH”选项，然后单击“立即安装”按钮。

1.3 体验 PyTorch

本节以手写数字识别这个经典问题来体验 PyTorch。

1.3.1 安装 PyTorch

本节介绍在 Windows 操作系统中安装 PyTorch。

要仅为在 CPU 上使用而安装 PyTorch 的当前版本：

```
    pip install torch torchvision torchaudio
```

如果要使用支持 CUDA 的 GPU 卡，则安装 PyTorch 的 GPU 版本。如果有 CUDA 11.8 支持的 GPU，可以运行以下命令：

```
    pip install torch torchvision torchaudio --index-url https://download.pytorch.
org/whl/cu118
```

也可以先下载相应的安装包，然后从本地安装 whl 文件。whl 文件下载地址是 https://download.pytorch.org/whl/torch_stable.html。安装过程如下：

```
    wget -c https://download.pytorch.org/whl/cpu/torch-2.2.1%2Bcpu-cp312-cp312-win_
amd64.whl
    pip install torch-2.2.1+cpu-cp312-cp312-win_amd64.whl
```

在交互式环境测试 PyTorch。

首先，输入以下代码导入 PyTorch 包。

```
    >>> import torch
```

然后按 Enter 键。

定义一个由 0 组成的向量。现在，把向量想象成一个数字集合或一个数字列表。接下来，使用 Tensor() 创建一个包含 3 个数字的列表的向量。这是一个三维矢量，它是三维空间中的一个箭头。

运行以下 Python 代码：

```
    >>> torch.Tensor([0, 0, 0])
    tensor([0., 0., 0.])
```

这表明 PyTorch 安装成功。按 Ctrl+D 组合键退出 Python 交互式控制台。

1.3.2 PyTorch 中的计算图

PyTorch 中的计算图是一种图形结构，用于表示计算过程的依赖关系。在 PyTorch 中，计算图是通过自动微分（Autograd）实现的。Autograd 跟踪所有在张量上执行的操

作，并构建计算图，同时计算操作的梯度，以便在反向传播过程中更新参数。

PyTorch 张量表示计算图中的一个节点。如果 x 是一个张量，其 x.requires_grad=True，那么 x.grad 是另一个张量相对于某个标量值保持 x 的梯度。

下面是一个使用 PyTorch 构建计算图的示例代码。

```
import torch

# 定义输入张量
x = torch.tensor([[1.0, 2.0], [3.0, 4.0]], requires_grad=True)
# 定义权重张量
w = torch.tensor([[5.0, 6.0], [7.0, 8.0]], requires_grad=True)
# 定义偏置张量
b = torch.tensor([[9.0, 10.0]], requires_grad=True)

# 定义计算图
y = torch.matmul(x, w) + b
z = torch.sum(y)

# 计算梯度
z.backward()

# 输出梯度
print(x.grad)
print(w.grad)
print(b.grad)
```

在 PyTorch 中，每个张量都有一个 grad_fn 属性，该属性记录了创建该张量的操作，也称为梯度函数。梯度函数是用于计算张量的梯度（或导数）的函数，其根据链式法则将梯度向后传播到计算图中的先前计算节点，例如：

```
import torch

# 创建一个指定形状的张量，并将其所有元素初始化为 1
x = torch.ones(3, 2, requires_grad=True)
print(x)
print(x.grad_fn)    # 输出 :None
```

当在计算图上执行反向传播时，PyTorch 使用每个张量的 grad_fn 属性来构造反向传播图。这样，每个张量都知道如何将其梯度向后传播到先前的操作。

torch.no.grad() 是一个上下文管理器，它将禁用上下文中的所有梯度计算，例如：

```
import torch
import math
```

```
    dtype = torch.float
    device = torch.device("cpu")

    # Create tensors to hold input and outputs
    # As we don't need to compute gradients with respect to these Tensors, we can set
requires_grad = False. This is also the default setting.
    x = torch.linspace(-math.pi, math.pi, 2000)
    y = torch.sin(x)

    # Create random tensors for weights. For these Tensors, we require gradients,
therefore, we can set requires_grad = True
    a = torch.randn((), device = device, dtype = dtype, requires_grad=True)
    b = torch.randn((), device = device, dtype = dtype, requires_grad=True)
    c = torch.randn((), device = device, dtype = dtype, requires_grad=True)
    d = torch.randn((), device = device, dtype = dtype, requires_grad=True)

    learning_rate = 1e-6

    # Forward pass: we compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        # Manually zero the gradients after updating weights
        a.grad = None
        b.grad = None
        c.grad = None
```

```
    d.grad = None
```

1.3.3　用三阶多项式拟合函数

本节将拟合 $y=\sin(x)$ 从 $-\pi$ 到 π 上的三阶多项式。多项式有 4 个参数，将使用梯度下降来通过最小化预测输出和真实输出之间的欧几里得距离来拟合随机数据。

下面将介绍如下 3 种拟合多项式的方法。

- 使用 NumPy 实现多项式拟合并使用 NumPy 操作手动实现正向和反向通过。
- 利用 PyTorch 张量的概念。
- 使用 PyTorch 中的 Autograd 软件包，该软件包使用自动微分来自动计算后向传播。

NumPy 是一个很好的科学计算工具，但对于深度学习来说不是很方便，因为它对梯度或计算图一无所知。然而，将三阶多项式拟合到正弦函数是非常容易的。

程序用到了绘图库 Matplotlib。Matplotlib 是 Python 编程语言及其数值数学扩展 NumPy 的绘图库。安装 Matplotlib 的代码如下：

```
pip install matplotlib
```

拟合多项式实现代码如下：

```
import numpy as np
import math
import matplotlib.pyplot as plt

x = np.linspace(-math.pi, math.pi, 2000)
y = np.sin(x)

# We randomly initialize weights
a = np.random.randn()
b = np.random.randn()
c = np.random.randn()
d = np.random.randn()

# print randomly initialized weights
print(f'a = {a}, b = {b}, c = {c}, d = {d}')

# learning rate
lr = 1e-6

for i in range(5000):
  # y = a + bx + cx^2 + dx^3
```

```
    y_pred = a + b*x + c*x ** 2 + d*x ** 3

    # Compute and print loss
    loss = np.square(y_pred -y).sum()
    if i%100 == 0:
      print(i,loss)

    # Backprop to compute the gradients of a, b, c, d with respect to loss
    #dL/da = (dL/dy_pred) * (dy_pred/da)
    #dL/db = (dL/dy_pred) * (dy_pred/db)
    #dL/dc = (dL/dy_pred) * (dy_pred/dc)
    #dL/dd = (dL/dy_pred) * (dy_pred/dd)

    grad_y_pred = 2.0 * (y_pred-y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()


    # Update Weights
    a -= lr * grad_a
    b -= lr * grad_b
    c -= lr * grad_c
    d -= lr * grad_d

plt.plot(x,y,label = 'y = sin(x)', c = 'b')
plt.plot(x, y_pred, label = 'y = a + bx + cx^2 + dx^3', c = 'r',linestyle = 'dashed')
plt.xlabel('x')
plt.ylabel('y')
plt.ylim([-2,2])
plt.legend()
plt.show()

print(f'Result: y = {a} + {b} x + {c} x^2 + {d} x^3')
```

使用 NumPy 拟合三阶多项式很容易。但现代深度神经网络呢？不幸的是，NumPy 无法利用 GPU 来加速其数值计算。这就是 PyTorch 张量的有用之处。张量是一个 *n* 维数组，可以跟踪梯度和计算图。要在 GPU 上运行 PyTorch 张量，只需要指定正确的设备。但就目前而言，我们将坚持使用 CPU。

看看如何使用 PyTorch 张量来完成我们的任务。

```
import torch
import math
```

```
    dtype = torch.float
    device = torch.device("cpu")
    #device = torch.device("cuda:0") # Uncomment this if GPU is available.

    # Create random input and data

    x = torch.linspace(-math.pi, math.pi, 2000, device=device, dtype=dtype)
    y = torch.sin(x)

    # Randomly initialize weights

    a = torch.randn((), device=device, dtype=dtype)
    b = torch.randn((), device=device, dtype=dtype)
    c = torch.randn((), device=device, dtype=dtype)
    d = torch.randn((), device=device, dtype=dtype)

    learning_rate = 1e-6

    for t in range(5000):
        # Forward pass: compute predicted y
        y_pred = a + b * x + c * x ** 2 + d * x ** 3

        # Compute and print loss
        loss = (y_pred - y).pow(2).sum().item()
        if t % 100 == 99:
            print(t, loss)

        # Backprop to compute gradients of a, b, c, d with respect to loss
        grad_y_pred = 2.0 * (y_pred - y)
        grad_a = grad_y_pred.sum()
        grad_b = (grad_y_pred * x).sum()
        grad_c = (grad_y_pred * x ** 2).sum()
        grad_d = (grad_y_pred * x ** 3).sum()

        # Update weights using gradient descent
        a -= learning_rate * grad_a
        b -= learning_rate * grad_b
        c -= learning_rate * grad_c
        d -= learning_rate * grad_d

    plt.plot(x,y,label = 'y = sin(x)', c = 'b')
    plt.plot(x, y_pred, label = 'y = a + bx + cx^2 + dx^3', c = 'r',linestyle =
'dashed')
    plt.xlabel('x')
    plt.ylabel('y')
```

```
    plt.ylim([-2,2])
    plt.legend()
    plt.show()
    print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

我们在上面看到了张量也可以用于将三阶多项式拟合到 sin 函数。然而，我们不得不手动包含前向和后向传播。这对于拟合多项式这样的简单任务来说并不难，但对于深度神经网络来说，可能会变得非常混乱。幸运的是，PyTorch 的 Autograd 软件包可以用来自动计算向后传播，实现代码如下：

```
    import torch
    import math
    import matplotlib.pyplot as plt

    dtype = torch.float
    device = torch.device("cpu")

    # Create tensors to hold input and outputs
    # As we don't need to compute gradients with respect to these Tensors, we can set
requires_grad = False. This is also the default setting.

    x = torch.linspace(-math.pi, math.pi, 2000)
    y = torch.sin(x)

    # Create random tensors for weights. For these Tensors, we require gradients,
therefore, we can set requires_grad = True

    a = torch.randn((), device=device, dtype=dtype, requires_grad=True)
    b = torch.randn((), device=device, dtype=dtype, requires_grad=True)
    c = torch.randn((), device=device, dtype=dtype, requires_grad=True)
    d = torch.randn((), device=device, dtype=dtype, requires_grad=True)

    learning_rate = 1e-6

    for t in range(5000):
        # Forward pass: we compute predicted y using operations on Tensors.
        y_pred = a + b * x + c * x ** 2 + d * x ** 3

        # Compute and print loss using operations on Tensors.
        # Now loss is a Tensor of shape (1,)
        # loss.item() gets the scalar value held in the loss.
        loss = (y_pred - y).pow(2).sum()
        if t % 100 == 99:
            print(t, loss.item())
```

```
        # Use autograd to compute the backward pass. This call will compute the
        # gradient of loss with respect to all Tensors with requires_grad=True.
        # After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
        # the gradient of the loss with respect to a, b, c, d respectively.
        loss.backward()

        # Manually update weights using gradient descent. Wrap in torch.no_grad()
        # because weights have requires_grad=True, but we don't need to track this
        # in autograd.
        with torch.no_grad():
            a -= learning_rate * a.grad
            b -= learning_rate * b.grad
            c -= learning_rate * c.grad
            d -= learning_rate * d.grad

            # Manually zero the gradients after updating weights
            a.grad = None
            b.grad = None
            c.grad = None
            d.grad = None

    plt.plot(x, y, label='y = sin(x)', c='b')
    # We need to use tensor.detach().numpy() to convert our tensor into numpy array
for plotting
    plt.plot(x, y_pred.detach().numpy(), label='y = a + bx + cx^2 + dx^3', c='r',
linestyle='dashed')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.ylim([-2, 2])
    plt.legend()
    plt.show()
    print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

1.3.4　实现手写数字识别

首先建立第一个神经网络并对其进行训练。PyTorch 中的子库 torch.nn 用于神经网络操作，子库 torch.optim 用于神经网络优化器。优化器是一种函数或算法，用于修改神经网络的属性，如权重和学习率。

这里将使用以下 5 个步骤来构建和训练模型：构建计算图→设置优化器→设置标准→设置数据→训练模型。

首先使用可管理的数据集构建一个小型模型。

创建一个新文件 step_2_helloworld.py。

编写一个简短的 18 行代码片段，用于训练一个小模型。首先导入几个 PyTorch 模块，代码如下：

```
import torch
import torch.nn as nn
import torch.optim as optim
```

在这里，将 PyTorch 库别名分为以下几个常用的快捷方式：

- torch 包含所有 PyTorch 实用程序。然而，常规 PyTorch 代码包括一些额外的导入。在这里遵循相同的约定，这样就可以在线理解 PyTorch 教程和随机代码片段。
- torch.nn 包含用于构建神经网络的实用程序。这通常表示为 nn。
- torch.optim 包含训练实用程序。这通常被表示为 optim。

接下来，定义神经网络、训练使用程序和数据集，代码如下：

```
. . .
net = nn.Linear(1, 1)  # 1. Build a computation graph (a line!)
optimizer = optim.SGD(net.parameters(), lr=0.1)  # 2. Setup optimizers
criterion = nn.MSELoss()  # 3. Setup criterion
x, target = torch.randn((1,)), torch.tensor([0.])  # 4. Setup data
. . .
```

上述代码定义了任何深度学习训练脚本的几个必要部分。

- net = ... 定义了“神经网络”。在这种情况下，模型是一条形式为 y=m*x 的线；参数 nn.Linear(1, 1) 是直线的斜率。此模型参数 nn.Linear(1, 1) 将在训练期间更新。注意，torch.nn 包括许多深度学习运算，如这里使用的全连接层（nn.Lineral）和卷积层（nn.Conv2d）。
- optimizer = ... 定义优化器。这个优化器决定了神经网络将如何学习。torch.optim（别名为 optim）包括许多可以使用的此类优化器。
- criterion = ... 定义了损失。简而言之，损失定义了模型试图最小化的内容。对于直线的基本模型，目标是最小化直线的预测 y 值与训练集中的实际 y 值之间的差异。注意，torch.nn 包括许多其他可以使用的损失函数。
- x, target = ... 定义了数据集。现在，数据集只是一个坐标，一个 x 值和一个 y 值。在这里，torch 包本身提供 tensor 来创建新的张量，并提供 randn 来创建具有随机值的张量。

最后，通过在数据集上迭代 10 次来训练模型，每次都要调整模型的参数，代码如下：

```
. . .
# 5. Train the model
for i in range(10):
    output = net(x)
```

```
    loss = criterion(output, target)
    print(round(loss.item(), 2))

    net.zero_grad()
    loss.backward()
    optimizer.step()
```

总体目标是通过调整线的斜率来最大限度地减少损失。为了实现这一点，该训练代码实现了一种名为梯度下降的算法。

要找到损耗最低的最佳模型，可执行以下操作：

① 使用 net = nn.Linear(1, 1) 初始化一个随机模型。

② 在 for i in range(10) 循环中开始训练。

③ 每一步的方向由梯度决定。

④ 用 optimizer = ... 中的 lr=0.1 来指定步长。这决定了每个步骤可以有多大。

上述代码中的最后 3 行也很重要。

- net.zero_grad 清除上一步迭代中可能剩余的所有梯度。
- loss.backward 计算新的梯度。
- optimizer.step 使用这些梯度来采取措施。

“hello world”神经网络到此结束，保存并关闭文件即可。

运行脚本 step_2_helloworld.py。脚本将输出以下内容：

```
Output
0.33
0.19
0.11
0.07
0.04
0.02
0.01
0.01
0.0
0.0
```

注意，损失不断减少，这表明模型正在学习。使用 PyTorch 时，还需要注意如下两个实现细节。

- PyTorch 使用 torch.Tensor 保存所有数据和参数。这里，torch.randn 生成一个具有随机值的张量，该张量具有所提供的形状。例如，torch.randn((1, 2)) 创建一个 1×2 张量或二维行向量。
- PyTorch 支持多种优化器。这里使用了 torch.optim.SGD，也称为随机梯度下降（SGD）。还有更多的优化器在 SGD 的基础上添加了额外的功能。也有许多损失，torch.nn.MSELoss 只是其中之一。

这就结束了在玩具数据集上的第一个模型。下一步，将使用神经网络代替这个小模型，用常用的机器学习基准代替玩具数据集。

为了更好地理解 PyTorch 的优点，下面使用包含更多神经网络操作的 torch.nn.functional 和支持许多可以开箱即用的数据集的 torchvision.datasets 构建深度神经网络。接下来，将使用预制数据集构建一个相对复杂的自定义模型。

下面将使用卷积，这是一种模式查找器。对于图像，卷积在不同级别的“意义”上寻找 2D 模式：直接应用于图像的卷积在寻找“较低级别”的特征，如边缘。然而，应用于许多其他操作的输出的卷积可能正在寻找“更高层次”的特征，如门。

现在，将通过定义一个稍微复杂一点的模型来扩展构建的第一个 PyTorch 模型。神经网络现在将包含两个卷积和一个完全连接的层，以处理图像输入。

创建一个新文件 step_3_mnist.py，遵循与之前相同的五步算法：构建计算图→设置优化器→设置标准→设置数据→训练模型。

第一，定义深层神经网络。这是可能在 MNIST 上发现的其他神经网络的精简版本。这样就可以在计算机上训练你的神经网络，代码如下：

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR

# 1. Build a computation graph
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.fc = nn.Linear(1024, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 1)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        output = F.log_softmax(x, dim=1)
        return output
net = Net()
. . .
```

在这里，定义了一个继承自 nn.Module 的神经网络类。神经网络中的所有操作（包括神经网络本身）都必须继承自 nn.Module。神经网络类的典型范例如下：

- 在构造函数中，定义网络所需的任何操作。在这种情况下，有两个卷积和一个完全连接的层。需要记住的一点是，构造函数总是以 super().__init__() 开头。PyTorch 希望在将模块（如 nn.Conv2d）分配给实例属性（self.conv1）之前初始化父类。
- 在 forward() 方法中，运行初始化的操作。这个方法决定了神经网络的结构，明确定义了神经网络将如何计算其预测。

该神经网络使用了如下 4 种操作。

- nn.Conv2d：卷积。卷积在图像中寻找图案。早期的卷积寻找像边缘这样的“低级”模式。网络中后来的卷积寻找“高级”模式，如狗的腿或耳朵。
- nn.Linear：一个完全连接的层。完全连接的层将所有输入特征与所有输出维度相关联。
- F.relu，F.max_pool2d：这些都是非线性的类型。relu() 是函数 f(x) = max(x, 0)。max_pool() 在每个值块中取最大值。在这种情况下，将在整个图像中获取最大值。
- log_softmax：规范化向量中的所有值，使这些值之和为 1。

第二，像以前一样，定义优化器。这一次，将使用不同的优化器和不同的超参数设置。超参数配置训练，而训练调整模型参数。

```
. . .
optimizer = optim.Adadelta(net.parameters(), lr=1.)  # 2. Setup optimizer
. . .
```

第三，与以前不同，现在将使用不同的损失。这种损失用于分类问题，其中模型的输出是类索引。在这个特定的例子中，模型将输出包含在输入图像中的数字（可能是从 0 到 9 的任何数字）。

```
. . .
criterion = nn.NLLLoss()  # 3. Setup criterion
. . .
```

第四，建立数据。在这种情况下，将设置一个名为 MNIST 的数据集，该数据集以手写数字为特征。每个图像是一个包含手写数字的 28 像素 ×28 像素的小图像，目标是将每个手写数字分类为 0、1、2、…或 9。

```
. . .
# 4. Setup data
transform = transforms.Compose([
    transforms.Resize((8, 8)),
    transforms.ToTensor(),
```

```
        transforms.Normalize((0.1307,), (0.3081,))
    ])
    train_dataset = datasets.MNIST(
        'data', train=True, download=True, transform=transform)
    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=512)
    . . .
```

在这里，在 transform=…中对图像进行预处理。通过调整图像的大小，将图像转换为 PyTorch 张量，并将张量归一化为平均值为 0，方差为 1。

在接下来的两行中，设置 train=True，因为这是训练数据集，并设置 download=True，以便在数据集尚未下载的情况下下载该数据集。

batch_size ＝ 512 确定一次在多少个图像上训练网络。除非批量大得离谱（例如，数以万计），否则，更大的批量更适合进行大致更快的训练。

第五，训练模型。现在将在提供的数据集中对所有样本进行一次迭代，而不是在同一个样本上运行 10 次。通过一次遍历所有样本，以下是一个回合的训练：

```
    . . .
    # 5. Train the model
    for inputs, target in train_loader:
        output = net(inputs)
        loss = criterion(output, target)
        print(round(loss.item(), 2))

        net.zero_grad()
        loss.backward()
        optimizer.step()
    . . .
```

保存并关闭文件。

运行脚本 step_3_mnist.py。

```
python step_3_mnist.py
```

脚本将输出以下内容：

```
Output
2.31
2.18
2.03
1.78
1.52
1.35
1.3
1.35
1.07
```

```
1.0
...
0.21
0.2
0.23
0.12
0.12
0.12
```

注意，最终损失小于初始损失值的 10%。这意味着神经网络正在正确训练。

训练到此结束。然而，0.12 的损失很难解释：我们不知道 0.12 是“好”还是“坏”。为了评估模型的性能，接下来要计算此分类模型的准确性。

在数据集的训练拆分上计算了损失值。但是，最好对数据集进行单独的验证拆分。可以使用此验证拆分来计算模型的准确性，但是，不能用它来训练。接下来，将设置验证数据集并在其上评估模型。这一步将使用与以前相同的 PyTorch 实用程序，包括 MNIST 数据集的 torchvision.dataset。

将 step_3_mnist.py 文件复制到 step_4_eval.py 中，打开文件 step_4_eval.py。

设置验证数据集：

```
. . .
train_loader = ...
val_dataset = datasets.MNIST(
    'data', train=False, download=True, transform=transform)
val_loader = torch.utils.data.DataLoader(val_dataset, batch_size=512)
. . .
```

在训练循环之后，在文件的末尾添加一个验证循环：

```
    . . .
    optimizer.step()

correct = 0.
net.eval()
for inputs, target in val_loader:
    output = net(inputs)
    _, pred = output.max(1)
    correct += (pred == target).sum()
accuracy = correct / len(val_dataset) * 100.
print(f'{accuracy:.2f}% correct')
```

在这里，验证循环执行一些操作来计算准确性：

- 运行 net.eval() 可以确保神经网络处于评估模式并准备好进行验证。一些操作在评估模式下的运行方式与在训练模式下的不同。
- 对 val_loader 中的所有输入和标签进行迭代。

- 运行模型 net(inputs) 以获得每个类别的概率。
- 找到概率最高的类 output.max(1)。
- 计算正确分类的图像数量：pred==target 计算布尔值向量。.sum() 将这些布尔值强制转换为整数，并有效地计算真值的数量。
- correct / len(val_dataset) 最终计算正确分类的图像的百分比。

保存并关闭文件。

运行脚本 step_4_eval.py：

```
python step_4_eval.py
```

脚本将输出以下内容：

```
Output
2.31
2.21
...
0.14
0.2
89% correct
```

注意，具体损失值和最终精度可能会有所不同。

现在已经训练了第一个深度神经网络。可以通过调整训练的超参数来进行进一步的修改和改进，包括不同数量的回合、学习率和不同的优化器。

1.4 本章小结

PyTorch 是 Python 的一个深度学习框架，因其易用性和灵活性而广受欢迎。PyTorch 最初是由 Facebook AI Research 开发的，现在被许多公司和组织使用，包括谷歌、微软和优步。PyTorch 是开源的，可以在 GitHub 上使用。

2016 年 10 月，PyTorch 开始作为 Adam Paszke 的实习项目。当时，Adam Paszke 在 Torch 的核心开发者 Soumith Chintala 手下工作。Torch 是 LuaJIT 的科学计算框架。LuaJIT 是 Lua 语言的实时编译器。

PyTorch 目前由 Soumith Chintala、Gregory Chanan、Dmytro Dzhulgakov、Edward Yang 和 Nikita Shulga 维护，数百名才华横溢的个人以各种形式和方式做出了重大贡献。

PyTorch 是在经过修改的 BSD 许可证下发布的。

第 2 章　Python 技术基础

本章介绍开发深度学习应用需要的 Python 语言语法基础。

2.1　变量

在 Python 中，定义变量时是不声明类型的。变量在内部是有类型的。利用 type() 函数可以得到变量的类型。例如，在交互式环境中输入代码：

```
>>> a=3
>>> type(a)
<class 'int'>
```

表明变量 a 是整数类型。

2.2　注释

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 -*

# 从 torch 的 utils 工具箱中导入 tensorboard 模块，然后导入 SummaryWriter 类
from torch.utils.tensorboard import SummaryWriter
```

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 3 种不同的数值类型：int（整数）、float（浮点数）和 complex（复数）。与 Java、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'>
```

因为 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
```

可以使用 decimal 模块使用十进制表示完整的小数，例如：

```
>>> import decimal
>>> a = decimal.Decimal('0.1')
>>> b = decimal.Decimal('0.1000000000000000000001')
>>> 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，例如：

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

复数有一些内置的访问器，例如：

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

如下内置函数支持复数：

```
>>> 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，例如：

```
>>> 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 2-1 所示。

表 2-1　用于数值运算的算术运算符说明

语　　法	数学含义	运算符名称
a+b	a+b	加
a−b	a−b	减
a*b	a×b	乘
a/b	a÷b	除
a//b	$\lfloor a \div b \rfloor$	地板除
a%b	a mod 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，即便分子和分母都是 int 类型的，返回的也是浮点数，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，则另一个操作数被转换为复数，例如：

```
>>> 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
```

在 Python 中，一行就是一条语句，可以使用反斜杠（\）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      ";            # 去除首尾空格
print(term.strip());
```

使用 split() 方法可以将句子分成单词。下例中，mary 是一个单一的字符串。虽然这是一个句子，但这些词语并没有表示成严格的单位。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() 方法可以把句子切分成单词。

```
>>> 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'，则 .split() 只在所有空格上分割。

```
>>> 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，则在空格上分割
['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，字符是长度为 1 的字符串。list() 函数可以将字符串转换为单个字母的列表，例如：

```
>>> 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 join(y) 方法，'x'.join(y) 连接列表 y 中由 'x' 分隔的每个元素。下面，变量 mwords 中的 5 个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'
```

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() 函数可以实现字符串和整数之间的互相转换，例如：

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

2.3.3 数组

可以使用 array（数组）存储同样数据类型的数值类型。通过 import array 导入 python 的数组类型，就可以使用 array 类型了。

例如：

```
from array import array
node=array('H')    # 存储无符号短整型的数组

node.append(12)
```

2.4 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 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，洗衣服分 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，称为一个代码块。Python 没有像 Java 或 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。Notepad++ 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循环语句。

2.5.1　条件语句

当路径不存在时，就创建它，可以使用条件语句来实现。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，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。关系运算符如表 2-2 所示。

表 2-2　关系运算符

运　算　符	用　　法	含　　义
>	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，示例代码如下：

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

2.5.2　循环语句

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 'banana' :
...       print(c,type(c))
...
b <class 'str'>
a <class 'str'>
n <class 'str'>
a <class 'str'>
n <class 'str'>
a <class 'str'>
```

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' 中出现的每个字符及其位置，代码如下：

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

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，代码如下：

```
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，示例代码如下：

```
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])
```

输出如下：

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

此外，列表还可以将另一个列表作为项目，这称为嵌套列表，示例代码如下：

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

使用 range 函数可以生成列表，示例代码如下：

```
>>> 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（=）更改一个项目或项目范围，示例代码如下：

```
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，示例代码如下：

```
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，其中，* 运算符重复列表给定次数，示例代码如下：

```
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，还可以使用 insert() 方法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，示例代码如下：

```
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]
```

使用关键字 del 可以从列表中删除一个或多个项目，示例代码如下：

```
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']
```

还可以完全删除列表，示例代码如下：

```
del my_list                         # 删除整个列表

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

使用 remove() 方法可以删除给定的项目。使用 pop() 方法可以删除给定索引处的项目。使用 clear() 方法可以清空列表。示例代码如下：

```
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：  []
```

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，示例代码如下：

```
>>> 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，示例代码如下：

```
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），示例代码如下：

```
>>> 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。下面有一个单词列表，我们只想从中提取包含 'wo' 的单词。为此，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，代码如下：

```
>>> 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，代码如下：

```
>>> 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，定义时赋值之后就不允许修改了，示例代码如下：

```
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，示例代码如下：

```
>>> t = ( ‘foo’ , ‘bar’ , ‘baz’ , ‘qux’ )
```

将元组中的元素分别赋给变量称为拆包，示例代码如下：

```
>>> (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，示例代码如下：

```
>>> (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，示例代码如下：

```
>>> 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，示例代码如下：

```
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，示例代码如下：

```
>>> 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，代码如下：

```
>>> 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，可存储任意类型对象。要访问字典元素，可以使用方括号和键来获取它的值，示例代码如下：

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

可以按字典中的值排序，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中，示例代码如下：

```
>>> 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，示例代码如下：

```
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（BitSet）是一串可以按位运算的位。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，代码如下：

```
# pip3 install pyroaring
```

或者从 whl 文件安装，代码如下：

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

可以像使用经典的 Python 集合那样在代码中使用 BitMap，示例代码如下：

```
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))
print("bm1 | bm2 = %s" % (bm1|bm2))
```

输出如下：

```
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，代码如下：

```
>>> 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

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。一个班里有两个名为陈晨的同学。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 交互式环境中输入如下代码：

```
help( ‘modules’ )
```

2.12　函数

可以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 可以定义一个函数，示例代码如下：

```
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，示例代码如下：

```
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，示例代码如下：

```
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
```

使用如下函数：

```
    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，可在这个参数前面加 *，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。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），如果当前模块是 Python 程序的入口模块（也称顶级模块、脚本文件），则当前模块的 __name__ 属性的值是 "__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 脚本是自上而下的执行。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（使用 import module 方法）并利用它的一个或两个方法（使用 from ... import ... 方式），可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：

```
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　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，示例代码如下：

```
print (‘total ‘+str(total_num_file)+’ files’)
```

或者格式化，代码如下：

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

另一种格式化输出的方法是使用 str.format() 方法，如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!
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
```

print 函数用到的格式化字符串的约定如表 2-3 所示。

表 2-3　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）

续表

转换类型	含　义
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（接收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）

2.14　正则表达式

re 模块是 Python 的标准库，用于处理正则表达式的所有事情。与任何其他模块一样，可以从导入 re 开始，代码如下：

```
>>> import re
```

假设想在下面这个非常简短的文本中找到以 'wo' 开头的所有单词。想要使用的是 re.findall() 方法。它需要两个参数：①正则表达式模式；②用于查找匹配的目标字符串，代码如下：

```
>>> wood = 'How much wood would a woodchuck chuck if a woodchuck could
chuck wood?'
>>> re.findall(r'wo\w+', wood)         # r'...' 表示原始字符串
['wood', 'would', 'woodchuck', 'woodchuck', 'wood']
>>>
```

首先，注意正则表达式 r'wo\w+' 使用原始字符串表示，如 r'...' 字符串前缀所示。这是因为，正则表达式使用反斜杠（\）作为它们自己的特殊转义字符，没有 'r' 时反斜杠被解释为 Python 的特殊转义字符。原始字符串以大写字母 R 或者小写字母 r 开始。

findall() 方法将所有匹配的字符串部分作为列表返回。如果没有匹配项，则返回一个空列表，代码如下：

```
>>> re.findall(r'o+', wood)
['o', 'oo', 'o', 'oo', 'oo', 'o', 'oo']
>>> re.findall(r'e+', wood)
[]
```

如果想忽略匹配中的大小写该怎么办？可以将其指定为 findall() 方法的第三个可选参数：re.IGNORECASE。

```
>>> foo = 'This and that and those'
>>> re.findall(r'th\w+', foo)
['that', 'those']
>>> re.findall(r'th\w+', foo, re.IGNORECASE)    # case is ignored while matching
['This', 'that', 'those']
```

如果想用其他东西替换所有匹配的部分怎么办？可以使用 re.sub() 方法完成。下面找到所有元音序列并用 '-' 替换。该方法将结果作为新字符串返回，代码如下：

```
>>> wood
'How much wood would a woodchuck chuck if a woodchuck could chuck wood?'
>>> re.sub(r'[aeiou]+', '-', wood)       #3 个参数：正则表达式、替换字符串、目标字符串
'H-w m-ch w-d w-ld - w-dch-ck ch-ck -f - w-dch-ck c-ld ch-ck w-d?'
```

删除匹配部分也可以通过 re.sub() 实现，只需将替换字符串设为空字符串 '' 即可，代码如下：

```
>>> re.sub(r'[aeiou]+', '', wood)        # 用空字符串替换
'Hw mch wd wld  wdchck chck f  wdchck cld chck wd?'
```

如果必须在许多不同的字符串中匹配正则表达式，最好将正则表达式构造为 Python 对象。这样，正则表达式的有限状态自动机被编译一次并重复使用。由于构建 FSA（Finite State Automata）在计算上相当昂贵，因此减轻了处理负荷。为此，请使用 re.compile() 方法，代码如下：

```
>>> myre = re.compile(r'\w+ou\w+')       # 将 myre 编译为正则表达式对象
>>> myre.findall(wood)                   # 直接在 myre 上调用 .findall() 方法
['would', 'could']
>>> myre.findall('Colorless green ideas sleep furiously')
['furiously']
>>> myre.findall('The thirty-three thieves thought that they thrilled
the throne throughout Thursday.')
['thought', 'throughout']
```

编译完成后，就可以直接在正则表达式对象上调用一个 re 方法。在上面的例子中，myre 是对应于 r'\w+ou\w+' 的编译正则表达式对象，在 myre 上调用 .findall() 方法。相对于没编译时，现在需要指定少一个参数：目标字符串 myre.findall(wood) 是唯一需要的东西。

有时，我们只对确认给定字符串中是否存在匹配感兴趣。在这种情况下，re.search() 是一个很好的选择。re.search() 方法仅查找第一个匹配，然后退出。如果找到匹配项，则返回一个匹配对象。如果没有，则不会返回任何值。r'e+' 在 'Colorless ...' 字符串中成功

匹配，因此，返回匹配对象。Wood 字符串没有一个 'e'，所以，没有返回任何值。

```
>>> re.search(r'e+', 'Colorless green ideas sleep furiously')
<_sre.SRE_Match object at 0x02D9CB48>
>>> re.search(r'e+', wood)
>>>
```

可以在 if 条件语句的上下文中使用 re.search()。在下面的代码中，if ... 行检查 re.search 方法是否有返回的对象，然后打印匹配的部分和匹配的行（注意：if 条件判断中的 someobj 返回 True，只要 someobj 不是以下之一："nothing"、整数 0、空字符串 ""、空列表 [] 和空字典 {}）。

```
>>> f = open('D:\\Lab\\warpeace.txt')
>>> blines = f.readlines()
>>> f.close()
>>> smite = re.compile(r'sm(i|o)te\w*')
>>> for b in blines:
        matchobj = smite.search(b)
        if matchobj:          # True if matchobj is not "nothing"
            print(matchobj.group(), '-', b, end='')
```

输出如下：

```
smite - servants to rejoice in Thy mercy; smite down our enemies and destroy
smote - wives and children." The nobleman smote his breast. "We will all
smote - you, Papa" (he smote himself on the breast as a general he had heard
smite - Faith, the sling of the Russian David, shall suddenly smite his head
```

在这个例子中，我们想要找出“战争与和平”中所有含有“smite / smote ...”字样的行。首先使用 .readlines() 方法将文本文件作为行的列表加载。然后，因为多次进行匹配，所以，编译正则表达式。最后，循环遍历文本行，通过 .search() 方法创建一个匹配对象，并且只有匹配对象存在时才打印出匹配的部分和行。

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
```

输出如下：

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

2.15.1 读写文件

逐行读入文本文件，代码如下：

```
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，代码如下：

```
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，代码如下：

```
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，则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。写入操作接收单个参数，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用户的数据安全。所以，一定要关闭文件。使用 <file>.close() 函数关闭所有文件，代码如下：

```
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。

对于 json 格式的文件，可以导入 json 模块读取文件，代码如下：

```
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

Linux 操作系统中的文件名区分英文大小写，而 Window 操作系统中的文件名不区分英文大小写。

可以使用 os.rename 方法重命名文件。首先使用 touch 命令创建一个空文件，代码如下：

```
# 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 遍历一个目录。os.scandir() 方法返回一个迭代器，示例代码如下：

```
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已获取的资源。

只打印出一个目录下的文件，代码如下：

```
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，代码如下：

```
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 with 语句

with 语句在 Python 脚本中被广泛使用。

with 语句格式如下：

```
with context [as var]:
    pass
```

这里 context 是一个表达式，它将返回一个对象并保存在 var 中。

以下是一个示例：

```
with open("data.txt") as f:
    print(type(f))
```

在本例中，open("data.txt") 将返回一个 _io.TextIOWrapper 对象，该对象将保存到变量 f 中。

使用 with 语句的主要原因是 with 语句完成后将执行一些额外的操作，示例代码如下：

```
with open("data.txt") as f:
    print(type(f))
print(f.closed)
print("--end--")
```

运行这个 Python 脚本，将得到如下结果：

```
<class '_io.TextIOWrapper'>
True
--end--
```

从上面的输出中可以发现，with 语句将在完成时关闭文件。用户不需要手动关闭此文件。还可以发现，with 语句创建的变量是全局的。在上面的例子中，变量 f 在整个 Python 脚本中都能很好地工作，而不仅仅是在 with 语句中。

2.17　使用 pickle 模块序列化对象

可以把内存中的 Python 数据对象保存成二进制文件，这样下次需要它时就可以简单地加载它并获得原始对象。该过程称为“对象序列化”。

从文件恢复出来对象称为反序列化。切勿反序列化从不受信任的来源收到的数据，因为这可能会带来一些严重的安全风险。pickle 模块在挑选恶意数据时无法知道或引发错误。

如下是一个序列化字典的简单例子。

```
import pickle
emp = {1:"A",2:"B",3:"C",4:"D",5:"E"}
pickling_on = open("Emp.pickle","wb")            # 打开文件 "Emp.pickle"
pickle.dump(emp, pickling_on)
pickling_on.close()                              # 关闭文件
```

注意使用“wb”而不是“w”，因为所有操作都是使用字节完成的。

下面研究一下如何反序列化这个字典。

```
pickle_off = open("Emp.pickle","rb")    # 读取字节时，请注意 "rb" 而不是 "r" 的用法
emp = pickle.load(pickle_off)
print(emp)
```

2.18　面向对象编程

语音识别软件往往由很多代码组成，也是一个复杂的系统，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，使用关键字 class 定义一个类。

如下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__ 提供了该类的文档字符串，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，代码如下：

```
ob = MyClass()
```

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

```
ob.func()  # 输出:  Hello
```

读者可能已经注意到类中函数定义中的 self 参数，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() 将转换为 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，每个对象都有一些默认属性和方法。 要查看对象的所有属性和方法，可以使用内置的 dir() 函数，示例代码如下：

```
ob = MyClass()

print(dir(ob))
```

2.19　命令行参数

在采用多种编程语言开发的语音识别系统中，Python 脚本可能需要从命令行直接读取参数。如果脚本很简单或临时使用，没有多个复杂的参数选项，可以直接用 sys.argv 读取传入的命令行参数。

测试代码 TestArgv.py 内容如下：

```
import sys

print("This is the path of the script: ", sys.argv[0])  # 脚本的相对路径
print("Number of arguments: ", len(sys.argv))           # 长度最少是 1
print("The arguments are: " , str(sys.argv))            #str 函数输出 sys.argv 的内容
```

输出结果如下：

```
D:\PycharmProjects\Scripts\python.exe D:/PycharmProjects/untitled/TestArgv.py a b c
This is the path of the script:  D:/PycharmProjects/untitled/TestArgv.py
Number of arguments:  4
The arguments are:  ['D:/PycharmProjects/untitled/TestArgv.py', 'a', 'b', 'c']
```

相关的规范有 GNU getopt_long()。getopt_long() 是 GNU 版本的 getopt 模块中的一个函数，用于解析命令行选项和参数。在 Python 中，通常使用 argparse 模块来实现相同的功能，因为 getopt 模块并不是所有的 Python 实现都提供，而 argparse 是 Python 的标准库模块。

GNU 扩展 getopt_long() 可以解析可读的多字符选项，该选项前缀为双短线，而非单个短线。双短线选项 (如 --inum) 可以和单个短线选项区分开 (-abc)。GNU 扩展允许带参选项有不同的形式：--name=arg。

argparse 包使得这一工作变得简单而规范。它支持 GNU getopt_long()。

您首先导入 argparse 库并初始化 ArgumentParser 的对象。此对象将保存读取命令行参数所需的所有信息。

以下是入门的基本语法：

```
import argparse

# 初始化 ArgumentParser 对象
parser = argparse.ArgumentParser()
```

可以使用 add_argument 方法添加参数。初始化 argparse.ArgumentParser() 时，可以传递几个可选参数来自定义其行为。示例代码如下：

```
import argparse

parser = argparse.ArgumentParser(description='Example application.')
```

```
    parser.add_argument('integer', type=int, help='An integer.')
    parser.add_argument('-f', '--foo', default='bar', choices=['bar', 'baz'],
help='Foo option.')
    parser.add_argument('--flag', action='store_true', help='A boolean flag.')

    args = parser.parse_args()
```

无参数运行 Testargparse.py 的输出结果如下：

```
    usage: Testargparse.py [-h] [-f {bar,baz}] [--flag] integer
    Testargparse.py: error: the following arguments are required: integer
```

2.20 数据库

SQLite3 是一个非常易于使用的数据库引擎。SQLite3 是独立的、无服务器的、零配置和事务性的。它非常快速且轻量级，整个数据库存储在一个磁盘文件中，可以在语音识别应用中用作内部数据存储。Python 标准库包含一个名为“sqlite3”的模块，用于处理此数据库。

使用 sqlite3 模块在内存中创建一个 SQLite 数据库，代码如下：

```
    import sqlite3

    conn = sqlite3.connect('example.db')    # 连接数据库
```

接下来，通过游标创建表，代码如下：

```
    c = conn.cursor()
    c.execute('''CREATE TABLE results (dataset text, wer float)''')
    c.execute('INSERT INTO results(dataset, wer) VALUES(?, ?)', ("LibriSpeech",
0.0583))
```

为了实际上保存更改，需要调用连接对象的 .commit() 方法，代码如下：

```
    conn.commit()
```

从前面创建的 results 表中获取并显示记录，代码如下：

```
    # 获得所有结果
    c.execute("SELECT dataset, wer FROM results ")
    d = c.fetchall()

    for row in d:
        print ("dataset: {}".format( row[0] ))
        print ("wer: {}".format( row[1] ))
```

可以使用 DB Browser for SQLite（https://github.com/sqlitebrowser/sqlitebrowser）查看数据库文件中的数据。

2.21　JSON 格式

JSON（JavaScript Object Notation）是一种轻量级的数据交换格式。人类很容易阅读和编写 JSON。机器也很容易解析和生成 JSON。可以用 JSON 传输由名称 / 值对和数组数据类型组成的数据对象。

JSON 的基本数据类型有如下 6 种。

- 数字：有符号的十进制数字，可能包含小数部分，可能使用指数 E 表示法，但不能包括非数字，如 NaN。该格式不区分整数和浮点数。
- 字符串：零个或多个 Unicode 字符的序列。字符串用双引号分隔，并支持反斜杠转义语法。
- 布尔值：为 True 或 False 的任一值。
- 数组：零个或多个值的有序列表，每个值可以是任何类型。数组使用方括号符号，元素以半角逗号分隔。
- 对象：名称 / 值对的无序集合，其中名称（也称为键）是字符串。由于对象旨在表示关联数组，推荐每个键在对象内是唯一的。对象用大括号分隔，并使用逗号分隔每对，而在每一对中，冒号 ':' 字符将键或名称与其值分隔开。
- null：一个空值，使用单词 null。

json.dumps() 函数将字典转换为字符串对象。例如，如下代码将输出环境变量。

```
import json, os
print(json.dumps(dict(os.environ), indent = 2))
```

json.loads() 函数将 JSON 格式的字符串解析为 Python 中的字典或列表，示例代码如下：

```
import json

r = {'is_claimed': 'True', 'rating': 3.5}
r = json.dumps(r)
loaded_r = json.loads(r)
loaded_r['rating']        # 输出 3.5
type(r)                   # 输出 str
type(loaded_r)            # 输出 dict
```

2.22 日志记录

机器学习的训练过程可能很长，为了监控运行状态，可以用日志记录，代码如下：

```
import logging

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

日志级别大小关系如下：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 将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，例如：

- 可以写入文件的多个文件处理器（TimeRotated、SizeRotated、Watched）。
- StreamHandler 可以输出到 stdout 或 stderr 等流。
- SMTPHandler 通过电子邮件发送日志记录。
- 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。

例如，'%(asctime)s %(levelname)s %(name)s: %(message)s' 会生成如下日志：

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

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 中的例子代码如下：

```
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))
```

输出结果如下：

```
2018-06-24 22:01:28,465 [TestStreamHandler.py:12 - <module> - INFO ] trainning...
```

2.23　异常处理

当人处于危险的环境中时，血液中的肾上腺素会升高。可以在运行时可能发生问题的代码中检查是否有异常发生，因为代码包装在 try 关键词中，所以称为 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，就能使正常的处理流程代码能够连贯在一起。

except 代码块又称异常处理器，其常见格式如下：

```
except ExceptionClass as e:  //异常类型
          //处理代码
```

下列代码将捕捉路径创建中的异常：

```
import errno

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
```

2.24　本章小结

Python 于 20 世纪 80 年代后期由荷兰的 Guido van Rossum 构思，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 于 2000 年 10 月 16 日发布，具有许多新功能，包括循环检测垃圾收集器和对 Unicode 的支持。Python 3 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。

Python 是一种多范式编程语言，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 代码和文件中。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。