给深度学习入门者的Python快速教程 - 基础篇

達聞西 · 1 年前

下篇：给深度学习入门者的Python快速教程 - numpy和Matplotlib篇

Life is short, you need Python

人生苦短，我用Python

-- Bruce Eckel

5.1 Python简介

本章将介绍Python的最基本语法，以及一些和深度学习还有计算机视觉最相关的基本使用。

5.1.1 Python简史

Python是一门解释型的高级编程语言，特点是简单明确。Python作者是荷兰人Guido van Rossum，1982年他获得数学和计算机硕士学位后，在荷兰数学与计算科学研究所（**C**entrum **W**iskunde & **I**nformatica, **CWI**）谋了份差事。在CWI期间，Guido参与到了一门叫做ABC的语言开发工作中。ABC是一门教学语言，所以拥有简单，可读性好，语法更接近自然语言等特点。在那个C语言一统天下的年代，ABC就是一股简单的清流，毕竟是门教学语言，最后没有流行起来，不过这段经历影响了Guido。1989年的圣诞假期，闲得蛋疼的Guido决定设计一门简单易用的新语言，要介于C和Shell之间，同时吸取ABC语法中的优点。Guido用自己喜欢的一部喜剧电视剧来命名这门语言：《Monty **Python**'s Flying Circus》。

1991年，第一版基于C实现的Python编译器诞生，因为简单，拓展性好，Python很快就在Guido的同事中大受欢迎，不久Python的核心开发人员就从Guido一人变成了一个小团队。后来随着互联网时代的到来，开源及社区合作的方式蓬勃发展，Python也借此上了发展的快车道。因为Python非常容易拓展，在不同领域的开发者贡献下，许多受欢迎的功能和特征被开发出来，渐渐形成了各种各样的库，其中一部分被加入到Python的标准库中，这让本来就不需要过多用者底层细节的Python变得更加强大好用。在不过多考虑执行效率的前提下，使用

所以出bug的可能性也小了很多。因此有了语言专家Bruce Eckel的那句名言：Life is short, you need Python. 后来这句话的中文版“人生苦短，我用Python”被Guido印在了T恤上。发展至今，Python渐渐成了最流行的语言之一，在编程语言排行榜TOBIE中常年占据前5的位置。另外随着Python的用户群越来越壮大，慢慢在本身特点上发展出了自己的哲学，叫做Python的禅（The Zen of Python）。遵循Python哲学的做法叫做很Python（Pythonic），具体参见：

PEP 20 -- The Zen of Python

或者在Python中执行：

>> import this

Python拥有很好的扩充性，可以非常轻松地用其他语言编写模块供调用，用Python编写的模块也可以通过各种方式轻松被其他语言调用。所以一种常见的Python使用方式是，底层复杂且对效率要求高的模块用C/C++等语言实现，顶层调用的API用Python封装，这样可以通过简单的语法实现顶层逻辑，故而Python又被称为“胶水语言”。这种特性的好处是，无需花费很多时间在编程实现上，更多的时间可以专注于思考问题的逻辑。尤其是对做算法和深度学习的从业人员，这种方式是非常理想的，所以如今的深度学习框架中，除了MATLAB，或是Deeplearning4j这种摆明了给Java用的，其他框架基本上要么官方接口就是Python，要么支持Python接口。

5.1.2 安装和使用Python

Python有两个大版本，考虑到用户群数量和库的各种框架的兼容性，本文以Python2（2.7）为准，语法尽量考虑和Python3的兼容。

Unix/Linux下的Python基本都是系统自带的，一般默认为Python2，使用时在终端直接键入python就能进入Python解释器界面：

```
dlcv@arreat-top:~$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在解释器下就已经可以进行最基本的编程了，比如：

```
dlcv@arreat-top:~$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world!")
hello world!
>>>
```

写程序的话还是需要保存成文件再执行，比如我们写下面语句，并且保存为helloworld.py：

```
print("Hello world!")
```

然后在终端里执行：

```
dlcv@arreat-top:~$ python helloworld.py
Hello world!
dlcv@arreat-top:~$
```

安装更多的python库一般有两种方法，第一是用系统的软件包管理，以Ubuntu 16.04 LTS为例，比如想要安装numpy库（后面会介绍这个库），软件包的名字就是python-numpy，所以在终端中输入：

>> sudo apt install python-numpy

Python自己也带了包管理器，叫做pip，使用如下：

安装和深度学习相关的框架时，一般来说推荐使用系统自带的包管理，出现版本错误的可能性低一些。另外也可以使用一些提前配置好很多第三方库的Python包，这些包通常已经包含了深度学习框架中绝大多数的依赖库，比如最常用的是Anaconda：

Download Anaconda Now!

Windows下的Python安装简单一些，从官方网站下载相应的安装程序就可以了，当然也有更方便的已经包含了很全的第三方库的选择，WinPython：

WinPython

并且是绿色的，直接执行就可以用了。

5.2 Python基本语法

There should be one-- and preferably only one --obvious way to do it.

对于一个特定的问题，应该只用最好的一种方法来解决。

-- Tim Peters

5.2.1 基本数据类型和运算

基本数据类型

Python中最基本的数据类型包括整型，浮点数，布尔值和字符串。类型是不需要声明的，比如：

```
a = 1       # 整数
b = 1.2     # 浮点数
c = True    # 布尔类型
d = "False" # 字符串
e = None    # NoneType
```

其中#是行内注释的意思。最后一个None是NoneType，注意不是0，在Python中利用type函数可以查看一个变量的类型：

```
type(a)     # <type 'int'>
type(b)     # <type 'float'>
type(c)     # <type 'bool'>
type(d)     # <type 'str'>
type(e)     # <type 'NoneType'>
```

注释中是执行type()函数后的输出结果，可以看到None是单独的一种类型NoneType。在很多API中，如果执行失败就会返回None。

变量和引用

Python中基本变量的赋值一般建立的是个引用，比如下面的语句：

```
a = 1
b = a
c = 1
```

a赋值为1后，b=a执行时并不会将a的值复制一遍，然后赋给b，而是简单地为a所指的值，也就是1建立了一个引用，相当于a和b都是指向包含1这个值的这块内存的指针。所以c=1执行的也是个引用建立，这三个变量其实是三个引用，指向同一个值。这个逻辑虽然简单，不过也还是常常容易弄混，这没关系，Python内置了id函数，可以返回一个对象的地址，用id函数可以让我们知道每个变量指向的是不是同一个值：

```
id(a)   # 35556792L
```

```
id(c)   # 35556792L
```

注释中表示的仍是执行后的结果。如果这时候我们接下面两个语句：

```
b = 2   # b的引用到新的一个变量上
id(b)   # 35556768L
```

可以看到b引用到了另一个变量上。

运算符

Python中的数值的基本运算和C差不多，字符串的运算更方便，下面是常见的例子：

```
a = 2
b = 2.3
c = 3
a + b           # 2 + 2.3 = 4.3
c - a           # 3 - 2 = 1
a / b           # 整数除以浮点数，运算以浮点数为准，2 / 2.3 = 0.8695652173913044
a / c           # Python2中，整数除法，向下取整 2 / 3 = 0
a ** c          # a的c次方，结果为8
a += 1          # Python中没有i++的用法，自增用+=
c -= 3          # c变成0了
d = 'Hello'
d + ' world!'   # 相当于字符串拼接，结果为'Hello world!'
d += ' "world"!'# 相当于把字符串接在当前字符串尾，d变为'Hello "world"!'
e = r'\n\t\\'
print(e)        # '\\n\\t\\\\'
```

需要提一下的几点：1）字符串用双引号和单引号都可以，区别主要是单引号字符串中如果出现单引号字符则需要用转义符，双引号也是一样，所以在单引号字符串中使用双引号，或者双引号字符串中使用单引号就会比较方便。另外三个双引号或者三个单引号围起来的也是字符串，因为换行方便，更多用于文档。2）Python2中两个数值相除会根据数值类型判断是否整数除法，Python3中则都按照浮点数。想要在Python2种也执行Python3中的除法只要执行下面语句：

```
from __future__ import division     # 使用Python3中的除法
1 / 2                                   # 0.5
```

3）字符串前加r表示字符串内容严格按照输入的样子，好处是不用转义符了，非常方便。

Python中的布尔值和逻辑的运算非常直接，下面是例子：

```
a = True
b = False
a and b     # False
a or b      # True
not a       # False
```

基本上就是英语，操作符优先级之类的和其他语言类似。Python中也有位操作：

```
~8      # 按位翻转，1000 --> -(1000+1)
8 >> 3  # 右移3位，1000 --> 0001
1 << 3  # 左移3位，0001 --> 1000
5 & 2   # 按位与，101 & 010 = 000
5 | 2   # 按位或，101 | 010 = 111
4 ^ 1   # 按位异或，100 ^ 001 = 101
```

==, !=和is

判断是否相等或者不等的语法和C也一样，另外在Python中也常常见到is操作符，这两者的区别在于==和!=比较引用指向的内存中的内容，而is判断两个变量是否指向一个地址，看下面的代码例子：

```
a = 1
b = 1.0
c = 1
a == b  # True，值相等
a is b  # False，指向的不是一个对象，这个语句等效于 id(a) == id(b)
a is c  # True，指向的都是整型值1
```

所以一定要分清要比较的对象应该用那种方式，对于一些特殊的情况，比如None，本着Pythonic的原则，最好用is None。

注意关键字

Python中，万物皆对象。不过这并不是这里要探讨的话题，想说的是一定要注意关键字，因为所有东西都是对象，所以一个简简单单的赋值操作就可以把系统内置的函数给变成一个普通变量，来看下边例子：

```
id(type)                            # 506070640L
type = 1                            # type成了指向1的变量
id(type)                            # 35556792L
id = 2                              # id成了指向2的变量
from __future__ import print_function
print = 3                           # print成了指向3的变量
```

注意print是个很特殊的存在，在Python3中是按照函数用，在Python2中却是个命令式的语句，最早print的用法其实是下边这样：

```
print "Hello world!"
```

这么用主要是受到ABC语法的影响，但这个用法并不Pythonic，后来加入了print函数，为了兼容允许两种用法并存。所以单纯给print赋值是不灵的，在Python2中使用Python3中的一些特性都是用from __future__ import来实现。

模块导入

因为提到了对象名覆盖和import，所以简单讲一下。import是利用Python中各种强大库的基础，比如要计算cos(π)的值，可以有下面4种方式：

```
# 直接导入Python的内置基础数学库
import math
print(math.cos(math.pi))

# 从math中导入cos函数和pi变量
from math import cos, pi
print(cos(pi))

# 如果是个模块，在导入的时候可以起个别名，避免名字冲突或是方便懒得打字的人使用
import math as m
print(m.cos(m.pi))

# 从math中导入所有东西
from math import *
print(cos(pi))
```

一般来说最后一种方式不是很推荐，因为不知道import导入的名字里是否和现有对象名已经有冲突，很可能会不知不觉覆盖了现有的对象。

5.2.2 容器

列表

Python中的容器是异常好用且异常有用的结构。这节主要介绍列表（list），元组（tuple），字典（dict）和集合（set）。这些结构和其他语言中的类似结构并无本质不同，来看例子了解下使用：

```
a = [1, 2, 3, 4]
b = [1]
c = [1]
d = b
e = [1, "Hello world!", c, False]
print(id(b), id(c))                # (194100040L, 194100552L)
print(id(b), id(d))                # (194100040L, 194100040L)
print(b == c)                      # True
f = list("abcd")
print(f)                           # ['a', 'b', 'c', 'd']
g = [0]*3 + [1]*4 + [2]*2          # [0, 0, 0, 1, 1, 1, 1, 2, 2]
```

因为变量其实是个引用，所以对列表而言也没什么不同，所以列表对类型没什么限制。也正因为如此，和变量不同的是，即使用相同的语句赋值，列表的地址也是不同的，在这个例子中体现在id(b)和id(c)不相等，而内容相等。列表也可以用list()初始化，输入参数需要是一个可以遍历的结构，其中每一个元素会作为列表的一项。“*”操作符对于列表而言是复制，最后一个语句用这种办法生成了分段的列表。

列表的基本操作有访问，增加，删除，和拼接：

```
a.pop()             # 把最后一个值4从列表中移除并作为pop的返回值
a.append(5)         # 末尾插入值，[1, 2, 3, 5]
a.index(2)          # 找到第一个2所在的位置，也就是1
a[2]                # 取下标，也就是位置在2的值，也就是第三个值3
a += [4, 3, 2]      # 拼接，[1, 2, 3, 5, 4, 3, 2]
a.insert(1, 0)      # 在下标为1处插入元素0，[1, 0, 2, 3, 5, 4, 3, 2]
a.remove(2)         # 移除第一个2，[1, 0, 3, 5, 4, 3, 2]
a.reverse()         # 倒序，a变为[2, 3, 4, 5, 3, 0, 1]
a[3] = 9            # 指定下标处赋值，[2, 3, 4, 9, 3, 0, 1]
b = a[2:5]          # 取下标2开始到5之前的子序列，[4, 9, 3]
c = a[2:-2]         # 下标也可以倒着数，方便算不过来的人，[4, 9, 3]
d = a[2:]           # 取下标2开始到结尾的子序列，[4, 9, 3, 0, 1]
e = a[:5]           # 取开始到下标5之前的子序列，[2, 3, 4, 9, 3]
f = a[:]            # 取从开头到最后的整个子序列，相当于值拷贝，[2, 3, 4, 9, 3, 0, 1]
a[2:-2] = [1, 2, 3] # 赋值也可以按照一段来，[2, 3, 1, 2, 3, 0, 1]
g = a[::-1]         # 也是倒序，通过slicing实现并赋值，效率略低于reverse()
a.sort()
print(a)            # 列表内排序，a变为[0, 1, 1, 2, 2, 3, 3]
```

因为列表是有顺序的，所以和顺序相关的操作是列表中最常见的，首先我们来打乱一个列表的顺序，然后再对这个列表排序：

```
import random
a = range(10)                 # 生成一个列表，从0开始+1递增到9
print(a)                       # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
random.shuffle(a)             # shuffle函数可以对可遍历且可变结构打乱顺序
print(a)                       # [4, 3, 8, 9, 0, 6, 2, 7, 5, 1]
b = sorted(a)
print(b)                       # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
c = sorted(a, reverse=True)
print(c)                       # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

元组

元组和列表有很多相似的地方，最大的区别在于不可变，还有如果初始化只包含一个元素的tuple和列表不一样，因为语法必须明确，所以必须在元素后加上逗号。另外直接用逗号分隔多个元素赋值默认是个tuple，这在函数多返回值的时候很好用：

```
a = (1, 2)
b = tuple(['3', 4]) # 也可以从列表初始化
c = (5,)
print(c)            # (5,)
d = (6)
print(d)            # 6
```

```
print(e)            # (3, 4, 5)
```

集合

集合是一种很有用的数学操作，比如列表去重，或是理清两组数据之间的关系，集合的操作符和位操作符有交集，注意不要弄混：

```
A = set([1, 2, 3, 4])
B = {3, 4, 5, 6}
C = set([1, 1, 2, 2, 2, 3, 3, 3, 3])
print(C)        # 集合的去重效果，set([1, 2, 3])
print(A | B)    # 求并集，set([1, 2, 3, 4, 5, 6])
print(A & B)    # 求交集，set([3, 4])
print(A - B)    # 求差集，属于A但不属于B的，set([1, 2])
print(B - A)    # 求差集，属于B但不属于A的，set([5, 6])
print(A ^ B)    # 求对称差集，相当于(A-B)|(B-A)，set([1, 2, 5, 6])
```

字典

字典是一种非常常见的“键-值”(**k**ey-**v**alue)映射结构，键无重复，一个键不能对应多个值，不过多个键可以指向一个值。还是通过例子来了解，构建一个名字->年龄的字典，并执行一些常见操作：

```
a = {'Tom': 8, 'Jerry': 7}
print(a['Tom'])             # 8
b = dict(Tom=8, Jerry=7)    # 一种字符串作为键更方便的初始化方式
print(b['Tom'])             # 8
if 'Jerry' in a:            # 判断'Jerry'是否在keys里面
    print(a['Jerry'])        # 7
print(a.get('Spike'))       # None，通过get获得值，即使键不存在也不会报异常
a['Spike'] = 10
a['Tyke'] = 3
a.update({'Tuffy': 2, 'Mammy Two Shoes': 42})
print(a.values())   # dict_values([8, 2, 3, 7, 10, 42])
print(a.pop('Mammy Two Shoes'))     # 移除'Mammy Two Shoes'的键值对，并返回42
print(a.keys())     # dict_keys(['Tom', 'Tuffy', 'Tyke', 'Jerry', 'Spike'])
```

注意到初始化字典和集合很像，的确如此，集合就像是没有值只有键的字典。既然有了人名到年龄的映射，也许你立马想到是否可以给字典排序？在Python3.6之前，这个问题是错误的，字典是一种映射关系，没有顺序。当然了，如果要把(键, 值)的这种对进行排序，是没有问题的，前提是先把字典转化成可排序的结构，items()或者iteritems()可以做到这件事，接上段代码继续：

```
b = a.items()
print(b)  # [('Tuffy', 2), ('Spike', 10), ('Tom', 8), ('Tyke', 3), ('Jerry', 7)]
from operator import itemgetter
c = sorted(a.items(), key=itemgetter(1))
print(c)  # [('Tuffy', 2), ('Tyke', 3), ('Jerry', 7), ('Tom', 8), ('Spike', 10)]
d = sorted(a.iteritems(), key=itemgetter(1))
print(d)  # [('Tuffy', 2), ('Tyke', 3), ('Jerry', 7), ('Tom', 8), ('Spike', 10)]
e = sorted(a)
print(e)  # 只对键排序，['Jerry', 'Spike', 'Tom', 'Tuffy', 'Tyke']
```

items()可以把字典中的键值对转化成一个列表，其中每个元素是一个tuple，tuple的第一个元素是键，第二个元素是值。变量c是按照值排序，所以需要一个操作符itemgetter，去位置为1的元素作为排序参考，如果直接对字典排序，则其实相当于只是对键排序。字典被当作一个普通的可遍历结构使用时，都相当于遍历字典的键。如果觉得字典没有顺序不方便，可以考虑使用OrderedDict，使用方式如下：

```
from collections import OrderedDict
a = {1: 2, 3: 4, 5: 6, 7: 8, 9: 10}
b = OrderedDict({1: 2, 3: 4, 5: 6, 7: 8, 9: 10})
print(a)    # {1: 2, 3: 4, 9: 10, 5: 6, 7: 8}
```

这样初始化时的顺序就保留了，除了有序的特性以外，用法上和字典没有区别。2016年9月，Guido宣布在Python3.6中，字典将默认有序，这样就不用纠结了。另外需要注意的一点是字典是通过哈希表实现的，所以键必须是可哈希的， list不能被哈希，所以也不能作为字典的键，而tuple就可以。

因为上上段代码中用到了iteritems()，所以这里顺带提一下迭代器（iterator），迭代器相当于一个函数，每次调用都返回下一个元素，从遍历的角度来看就和列表没有区别了。iteritems()就是一个迭代器，所以效果一样，区别是迭代器占用更少内存，因为不需要一上来就生成整个列表。一般来说，如果只需要遍历一次，用迭代器是更好的选择，若是要多次频繁从一个可遍历结构中取值，且内存够，则直接生成整个列表会更好。当然，用迭代器生成一个完整列表并不麻烦，所以有个趋势是把迭代器作为默认的可遍历方式，比如前面我们使用过用来生成等差数列列表的range()，在Python2中对应的迭代器形式是xrange()。在Python3中，range()就不再产生一个列表了，而是作为迭代器，xrange()直接没了。

5.2.3 分支和循环

从这节开始，代码就未必适合在Python终端中输入了，选个顺手的编辑器或者IDE。作者良心推荐PyCharm，虽然慢，但好用，社区版免费：

PyCharm

for循环

上面提到的4种容器类型都是可遍历的，所以该讲讲用来遍历的for循环了。for循环的语法也是简单的英语：

```
a = ['This', 'is', 'a', 'list', '!']
b = ['This', 'is', 'a', 'tuple', '!']
c = {'This': 'is', 'an': 'unordered', 'dict': '!'}

# 依次输出：'This', 'is', 'a', 'list', '!'
for x in a:
    print(x)

# 依次输出：'This', 'is', 'a', 'tuple', '!'
for x in b:
    print(x)

# 键的遍历。不依次输出：'This', 'dict', 'an'
for key in c:
    print(key)

# 依次输出0到9
for i in range(10):
    print(i)
```

注意到每个for循环中，print都有缩进，这是Python中一个让人爱恨交织的特点：**强行缩进**来表明成块的代码。这样做的好处是代码十分清晰工整，还有助于防止写出过长的函数或者过深的嵌套，坏处是有时候不知为什么tab和空格就一起出现了，又或是多重if-else不知怎得就没对齐，还是挺麻烦的。

回到for循环上，这种把每个元素拿出来的遍历方式叫做for_each风格，熟悉Java的话就不会陌生，C++11中也开始支持这种for循环方式。不过如果还是需要下标呢？比如遍历一个list的时候，希望把对应下标也打印出来，这时可以用enumerate：

```
names = ["Rick", "Daryl", "Glenn"]

# 依次输出下标和名字
for i, name in enumerate(names):
    print(i, name)
```

需要注意的是，通过取下标遍历当然是可行的，比如用len()函数获得列表长度，然后用range()/xrange()函数获得下标，但是并不推荐这样做：

```
words = ["This", "is", "not", "recommended"]

# not pythonic :(
for i in xrange(len(words)):
    print(words[i])
```

在使用for循环时，有时会遇到这样一种场景：我们需要对遍历的每个元素进行某种判断，如果符合这种判断的情况没有发生，则执行一个操作。举个例子某神秘部门要审核一个字符串列表，如果没有发现不和谐的字眼，则将内容放心通过，一种解决办法是下面这样：

```
wusuowei = ["I", "don't", "give", "a", "shit"]  # 无所谓

hexie = True                                    # 默认和谐社会
for x in wusuowei:
    if x == "f**k":
        print("What the f**k!")                 # 发现了不该出现的东西，WTF！
        hexie = False                           # 不和谐了
        break                                   # 赶紧停下！不能再唱了

if hexie:                                       # 未发现不和谐元素！
    print("Harmonious society!")                # 和谐社会！
```

这样需要设置一个标记是否发现不和谐因素的状态变量hexie，循环结束后再根据这个变量判断内容是否可以放心通过。一种更简洁不过有些小众的做法是直接和else一起，如果for循环中的if块内的语句没有被触发，则通过else执行指定操作：

```
wusuowei = ["I", "don't", "give", "a", "shit"]

for x in wusuowei:
    if x == "f**k":
        print("What the f**k!")
        hexie = False
        break
else:                               # for循环中if内语句未被触发
    print("Harmonious society!") # 和谐社会！
```

这样不需要一个标记是否和谐的状态变量，语句简洁了很多。

if和分支结构

上一个例子中已经出现if语句了，所以这部分讲讲if。Python的条件控制主要是三个关键字：if-elif-else，其中elif就是else if的意思。还是看例子：

```
pets =['dog', 'cat', 'droid', 'fly']

for pet in pets:
    if pet == 'dog':        # 狗粮
        food = 'steak'      # 牛排
    elif pet == 'cat':      # 猫粮
        food = 'milk'       # 牛奶
    elif pet == 'droid':    # 机器人
        food = 'oil'        # 机油
    elif pet == 'fly':      # 苍蝇
        food = 'sh*t'       #
    else:
        pass
    print(food)
```

需要提一下的是pass，这就是个空语句，什么也不做，占位用。Python并没有switch-case的语法，等效的用法要么是像上面一样用if-elif-else的组合，要么可以考虑字典：

```
pets = ['dog', 'cat', 'droid', 'fly']
food_for_pet = {
    'dog': 'steak',
```

```
    'droid': 'oil',
    'fly': 'sh*t'
}

for pet in pets:
    food = food_for_pet[pet] if pet in food_for_pet else None
    print(food)
```

这里还用到了一个if-else常见的行内应用，就是代替三元操作符，如果键在字典中，则food取字典的对应值，否则为None。

if表达式中的小技巧

通过链式比较让语句简洁：

```
if -1 < x < 1:  # 相较于 if x > -1 and x < 1:
    print('The absolute value of x is < 1')
```

判断一个值是不是等于多个可能性中的一个：

```
if x in ['piano', 'violin', 'drum']:    # 相较于 if x == 'piano' or x == 'violin' or
    print("It's an instrument!")
```

Python中的对象都会关联一个真值，所以在if表达式中判断是否为False或者是否为空的时候，是无需写出明确的表达式的：

```
a = True
if a:           # 判断是否为真，相较于 a is True
    print('a is True')

if 'sky':       # 判断是否空字符串，相较于 len('sky') > 0
    print('birds')

if '':          # 判断是否空字符串，同上
    print('Nothing!')

if {}:          # 判断是否空的容器(字典)，相较于len({}) > 0
    print('Nothing!')
```

隐式表达式为False的是如下状况：

- None

- False

- 数值0

- 空的容器或序列（字符串也是一种序列）

- 用户自定义类中，如果定义了__len__()或者__nonzero__()，并且被调用后返回0或者False

while循环

while的就是循环和if的综合体，是一种单纯的基于条件的循环，本身没有遍历的意思，这是和for_each的本质差别，这种区别比起C/C++中要明确得多，用法如下：

```
i = 0
while i < 100:  # 笑100遍
    print("ha")

while True:     # 一直笑
    print("ha")
```

5.2.4 函数、生成器和类

还是从几个例子看起：

```
def say_hello():
    print('Hello!')

def greetings(x='Good morning!'):
    print(x)

say_hello()                             # Hello!
greetings()                             # Good morning!
greetings("What's up!")                 # What's up!
a = greetings()                         # 返回值是None

def create_a_list(x, y=2, z=3): # 默认参数项必须放后面
    return [x, y, z]

b = create_a_list(1)                    # [1, 2, 3]
c = create_a_list(3, 3)                 # [3, 3, 3]
d = create_a_list(6, 7, 8)              # [6, 7, 8]

def traverse_args(*args):
    for arg in args:
        print(arg)

traverse_args(1, 2, 3)              # 依次打印1, 2, 3
traverse_args('A', 'B', 'C', 'D')   # 依次打印A, B, C, D

def traverse_kargs(**kwargs):
    for k, v in kwargs.items():
        print(k, v)

traverse_kargs(x=3, y=4, z=5)       # 依次打印('x', 3), ('y', 4), ('z', 5)
traverse_kargs(fighter1='Fedor', fighter2='Randleman')

def foo(x, y, *args, **kwargs):
    print(x, y)
    print(args)
    print(kwargs)

# 第一个pring输出(1, 2)
# 第二个print输出(3, 4, 5)
# 第三个print输出{'a': 3, 'b': 'bar'}
foo(1, 2, 3, 4, 5, a=6, b='bar')
```

其实和很多语言差不多，括号里面定义参数，参数可以有默认值，且默认值不能在无默认值参数之前。Python中的返回值用return定义，如果没有定义返回值，默认返回值是None。参数的定义可以非常灵活，可以有定义好的固定参数，也可以有可变长的参数(args: arguments)和关键字参数(kargs: keyword arguments)。如果要把这些参数都混用，则固定参数在最前，关键字参数在最后。

Python中万物皆对象，所以一些情况下函数也可以当成一个变量似的使用。比如前面小节中提到的用字典代替switch-case的用法，有的时候我们要执行的不是通过条件判断得到对应的变量，而是执行某个动作，比如有个小机器人在坐标(0, 0)处，我们用不同的动作控制小机器人移动：

```
moves = ['up', 'left', 'down', 'right']

coord = [0, 0]

for move in moves:
    if move == 'up':        # 向上，纵坐标+1
        coord[1] += 1
```

```
    elif move == 'left':    # 向左，横坐标-1
        coord[0] -= 1
    elif move == 'right':   # 向右，横坐标+1
        coord[0] += 1
    else:
        pass
    print(coord)            # 打印当前位置坐标
```

不同条件下对应的是对坐标这个列表中的值的操作，单纯的从字典取值就办不到了，所以就把函数作为字典的值，然后用这个得到的值执行相应动作：

```
moves = ['up', 'left', 'down', 'right']

def move_up(x):         # 定义向上的操作
    x[1] += 1

def move_down(x):       # 定义向下的操作
    x[1] -= 1

def move_left(x):       # 定义向左的操作
    x[0] -= 1

def move_right(x):      # 定义向右的操作
    x[0] += 1

# 动作和执行的函数关联起来，函数作为键对应的值
actions = {
    'up': move_up,
    'down': move_down,
    'left': move_left,
    'right': move_right
}

coord = [0, 0]

for move in moves:
    actions[move](coord)
    print(coord)
```

把函数作为值取到后，直接加一括号就能使了，这样做之后起码在循环部分看上去很简洁。有点C里边函数指针的意思，只不过更简单。其实这种用法在之前讲排序的时候我们已经见过了，就是operator中的itemgetter。itemgetter(1)得到的是一个可调用对象(callable object)，和返回下标为1的元素的函数用起来是一样的：

```
def get_val_at_pos_1(x):
    return x[1]

heros = [
    ('Superman', 99),
    ('Batman', 100),
    ('Joker', 85)
]

sorted_pairs0 = sorted(heros, key=get_val_at_pos_1)
sorted_pairs1 = sorted(heros, key=lambda x: x[1])

print(sorted_pairs0)
print(sorted_pairs1)
```

在这个例子中我们用到了一种特殊的函数：lambda表达式。Lambda表达式在Python中是一种匿名函数，lambda关键字后面跟输入参数，然后冒号后面是返回值（的表达式），比如上边例子中就是一个取下标1元素的函数。当然，还是那句话，万物皆对象，给lambda表达式取名字也是一点问题没有的：

生成器（Generator）

生成器是迭代器的一种，形式上看和函数很像，只是把return换成了yield，在每次调用的时候，都会执行到yield并返回值，同时将当前状态保存，等待下次执行到yield再继续：

```
# 从10倒数到0
def countdown(x):
    while x >= 0:
        yield x
        x -= 1

for i in countdown(10):
    print(i)

# 打印小于100的斐波那契数
def fibonacci(n):
    a = 0
    b = 1
    while b < n:
        yield b
        a, b = b, a + b

for x in fibonacci(100):
    print(x)
```

生成器和所有可迭代结构一样，可以通过next()函数返回下一个值，如果迭代结束了则抛出StopIteration异常：

```
a = fibonacci(3)
print(next(a))  # 1
print(next(a))  # 1
print(next(a))  # 2
print(next(a))  # 抛出StopIteration异常
```

Python3.3以上可以允许yield和return同时使用，return的是异常的说明信息：

```
# Python3.3以上可以return返回异常的说明
def another_fibonacci(n):
    a = 0
    b = 1
    while b < n:
        yield b
        a, b = b, a + b
    return "No more ..."

a = another_fibonacci(3)
print(next(a))  # 1
print(next(a))  # 1
print(next(a))  # 2
print(next(a))  # 抛出StopIteration异常并打印No more消息
```

类（Class）

Python中的类的概念和其他语言相比没什么不同，比较特殊的是protected和private在Python中是没有明确限制的，一个惯例是用单下划线开头的表示protected，用双下划线开头的表示private：

```
class A:
    """Class A"""
    def __init__(self, x, y, name):
        self.x = x
        self.y = y
        self._name = name
```

```
        print(self._name)

    def greeting(self):
        print("What's up!")

    def __l2norm(self):
        return self.x**2 + self.y**2

    def cal_l2norm(self):
        return self.__l2norm()

a = A(11, 11, 'Leonardo')
print(A.__doc__)                # "Class A"
a.introduce()                   # "Leonardo"
a.greeting()                    # "What's up!"
print(a._name)                  # 可以正常访问
print(a.cal_l2norm())   # 输出11*11+11*11=242
print(a._A__l2norm())   # 仍然可以访问，只是名字不一样
print(a.__l2norm())             # 报错: 'A' object has no attribute '__l2norm'
```

类的初始化使用的是__init__(self,)，所有成员变量都是self的，所以以self.开头。可以看到，单下划线开头的变量是可以直接访问的，而双下划线开头的变量则触发了Python中一种叫做name mangling的机制，其实就是名字变了下，仍然可以通过前边加上“_类名”的方式访问。也就是说Python中变量的访问权限都是靠自觉的。类定义中紧跟着类名字下一行的字符串叫做docstring，可以写一些用于描述类的介绍，如果有定义则通过“类名.__doc__”访问。这种前后都加双下划线访问的是特殊的变量/方法，除了__doc__和__init__还有很多，这里就不展开讲了。

Python中的继承也非常简单，最基本的继承方式就是定义类的时候把父类往括号里一放就行了：

```
class B(A):
    """Class B inheritenced from A"""
    def greeting(self):
        print("How's going!")

b = B(12, 12, 'Flaubert')
b.introduce()   # Flaubert
b.greeting()    # How's going!
print(b._name())        # Flaubert
print(b._A__l2norm())   # “私有”方法，必须通过_A__l2norm访问
```

5.2.5 map, reduce和filter

map可以用于对可遍历结构的每个元素执行同样的操作，批量操作：

```
map(lambda x: x**2, [1, 2, 3, 4])               # [1, 4, 9, 16]

map(lambda x, y: x + y, [1, 2, 3], [5, 6, 7])   # [6, 8, 10]
```

reduce则是对可遍历结构的元素按顺序进行两个输入参数的操作，并且每次的结果保存作为下次操作的第一个输入参数，还没有遍历的元素作为第二个输入参数。这样的结果就是把一串可遍历的值，减少（reduce）成一个对象：

```
reduce(lambda x, y: x + y, [1, 2, 3, 4])    # ((1+2)+3)+4=10
```

filter顾名思义，根据条件对可遍历结构进行筛选：

```
filter(lambda x: x % 2, [1, 2, 3, 4, 5])    # 筛选奇数，[1, 3, 5]
```

需要注意的是，对于filter和map，在Python2中返回结果是列表，Python3中是生成器。

5.2.6 列表生成（list comprehension）

列表生成是Python2.0中加入的一种语法，可以非常方便地用来生成列表和迭代器，比如上节中map的两个例子和filter的一个例子可以用列表生成重写为：

```
[x**2 for x in [1, 2, 3, 4]]                    # [1, 4, 9 16]

[sum(x) for x in zip([1, 2, 3], [5, 6, 7])] # [6, 8, 10]

[x for x in [1, 2, 3, 4, 5] if x % 2]       # [1, 3, 5]
```

zip()函数可以把多个列表关联起来，这个例子中，通过zip()可以按顺序同时输出两个列表对应位置的元素对。有一点需要注意的是，zip()不会自动帮助判断两个列表是否长度一样，所以最终的结果会以短的列表为准，想要以长的列表为准的话可以考虑itertools模块中的izip_longest()。如果要生成迭代器只需要把方括号换成括号，生成字典也非常容易：

```
iter_odd = (x for x in [1, 2, 3, 4, 5] if x % 2)

print(type(iter_odd))                       # <type 'generator'>

square_dict = {x: x**2 for x in range(5)}   # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

至于列表生成和map/filter应该优先用哪种，这个问题很难回答，不过Python创始人Guido似乎不喜欢map/filter/reduce，他曾在表示过一些从函数式编程里拿来的特性是个错误。

5.2.7 字符串

Python中字符串相关的处理都非常方便，来看例子：

```
a = 'Life is short, you need Python'
a.lower()               # 'life is short, you need Python'
a.upper()                       # 'LIFE IS SHORT, YOU NEED PYTHON'
a.count('i')                    # 2
a.find('e')                     # 从左向右查找'e', 3
a.rfind('need')                 # 从右向左查找'need', 19
a.replace('you', 'I')   # 'Life is short, I need Python'
tokens = a.split()      # ['Life', 'is', 'short,', 'you', 'need', 'Python']
b = ' '.join(tokens)    # 用指定分隔符按顺序把字符串列表组合成新字符串
c = a + '\n'                    # 加了换行符，注意+用法是字符串作为序列的用法
c.rstrip()                      # 右侧去除换行符
[x for x in a]                  # 遍历每个字符并生成由所有字符按顺序构成的列表
'Python' in a                           # True
```

Python2.6中引入了format进行字符串格式化，相比在字符串中用%的类似C的方式，更加强大方便：

```
a = 'I’m like a {} chasing {}.'
# 按顺序格式化字符串，'I’m like a dog chasing cars.'
a.format('dog', 'cars')

# 在大括号中指定参数所在位置
b = 'I prefer {1} {0} to {2} {0}'
b.format('food', 'Chinese', 'American')

# >代表右对齐，>前是要填充的字符，依次输出：
# 000001
# 000019
# 000256
for i in [1, 19, 256]:
    print('The index is {:0>6d}'.format(i))

# <代表左对齐，依次输出：
# *---------
```

```
# *******---
for x in ['*', '****', '*******']:
    progress_bar = '{:-<10}'.format(x)
    print(progress_bar)

for x in [0.0001, 1e17, 3e-18]:
    print('{:.6f}'.format(x))   # 按照小数点后6位的浮点数格式
    print('{:.1e}'.format(x))   # 按照小数点后1位的科学记数法格式
    print ('{:g}'.format(x))    # 系统自动选择最合适的格式

template = '{name} is {age} years old.'
c = template.format(name='Tom', age=8)) # Tom is 8 years old.
d = template.format(age=7, name='Jerry')# Jerry is 7 years old.
```

format在生成字符串和文档的时候非常有用，更多更详细的用法可以参考Python官网：

7.1. string - Common string operations - Python 2.7.13 documentation

5.2.8 文件操作和pickle

在Python中，推荐用上下文管理器（with-as）来打开文件，IO资源的管理更加安全，而且不用老惦记着给文件执行close()函数。还是举例子来说明，考虑有个文件name_age.txt，里面存储着名字和年龄的关系，格式如下：

```
Tom,8
Jerry,7
Tyke,3
...
```

读取文件内容并全部显示：

```
with open('name_age.txt', 'r') as f:    # 打开文件，读取模式
    lines = f.readlines()               # 一次读取所有行
    for line in lines:                          # 按行格式化并显示信息
        name, age = line.rstrip().split(',')
        print('{} is {} years old.'.format(name, age))
```

open()的第一个参数是文件名，第二个参数是模式。文件的模式一般有四种，读取(r)，写入(w)，追加(a)和读写(r+)。如果希望按照二进制数据读取，则将文件模式和b一起使用（wb, r+b...）。

再考虑一个场景，要读取文件内容，并把年龄和名字的顺序交换存成新文件age_name.txt，这时可以同时打开两个文件：

```
with open('name_age.txt', 'r') as fread, open('age_name.txt', 'w') as fwrite:
    line = fread.readline()
    while line:
        name, age = line.rstrip().split(',')
        fwrite.write('{},{}\n'.format(age, name))
        line = fread.readline()
```

有的时候我们进行文件操作是希望把对象进行序列化，那么可以考虑用pickle模块：

```
import pickle

lines = [
    "I'm like a dog chasing cars.",
    "I wouldn't know what to do if I caught one...",
    "I'd just do things."
]

with open('lines.pkl', 'wb') as f:      # 序列化并保存成文件
    pickle.dump(lines, f)
```

```
    lines_back = pickle.load(f)

print(lines_back)                       # 和lines一样
```

注意到，序列化的时候就得使用b模式了。Python2中有个效率更高的pickle叫cPickle，用法和pickle一样，在Python3中就只有一个pickle。

5.2.9 异常

相比起其他一些语言，在Python中我们可以更大胆地使用异常，因为异常在Python中是非常常见的存在，比如下面这种简单的遍历：

```
a = ['Why', 'so', 'serious', '?']

for x in a:
    print(x)
```

当用for进行遍历时，会对要遍历的对象调用iter()。这需要给对象创建一个迭代器用来依次返回对象中的内容。为了能成功调用iter()，该对象要么得支持迭代协议(定义__iter__())，要么得支持序列协议(定义__getitem__())。当遍历结束时，__iter__()或者__getitem__()都需要抛出一个异常。__iter__()会抛出StopIteration，而__getitem__()会抛出IndexError，于是遍历就会停止。

在深度学习中，尤其是数据准备阶段，常常遇到IO操作。这时候遇到异常的可能性很高，采用异常处理可以保证数据处理的过程不被中断，并对有异常的情况进行记录或其他动作：

```
for filepath in filelist:   # filelist中是文件路径的列表
    try:
        with open(filepath, 'r') as f:
            # 执行数据处理的相关工作
            ...

        print('{} is processed!'.format(filepath))
    except IOError:
        print('{} with IOError!'.format(filepath))
        # 异常的相应处理
        ...
```

5.2.10 多进程（multiprocessing）

深度学习中对数据高效处理常常会需要并行，这时多进程就派上了用场。考虑这样一个场景，在数据准备阶段，有很多文件需要运行一定的预处理，正好有台多核服务器，我们希望把这些文件分成32份，并行处理：

```
from multiprocessing import Process#, freeze_support

def process_data(filelist):
    for filepath in filelist:
        print('Processing {} ...'.format(filepath))
        # 处理数据
        ...

if __name__ == '__main__':
    # 如果是在Windows下，还需要加上freeze_support()
    #freeze_support()

    # full_list包含了要处理的全部文件列表
    ...

    n_total = len(full_list) # 一个远大于32的数
    n_processes = 32

    # 每段子列表的平均长度
```

```
    # 计算下标，尽可能均匀地划分输入文件列表
    indices = [int(round(i*length)) for i in range(n_processes+1)]

    # 生成每个进程要处理的子文件列表
    sublists = [full_list[indices[i]:indices[i+1]] for i in range(n_processes)]

    # 生成进程
    processes = [Process(target=process_data, args=(x,)) for x in sublists]

    # 并行处理
    for p in processes:
        p.start()

    for p in processes:
        p.join()
```

其中**if** __name__ == '__main__'用来标明在import时不包含，但是作为文件执行时运行的语句块。为什么不用多线程呢？简单说就是Python中线程的并发无法有效利用多核，如果有兴趣的读者可以从下面这个链接看起：

GlobalInterpreterLock - Python Wiki

5.2.11 os模块

深度学习中的数据多是文件，所以数据处理阶段和文件相关的操作就非常重要。除了文件IO，Python中一些操作系统的相关功能也能够非常方便地帮助数据处理。想象一下我们有一个文件夹叫做data，下边有3个子文件夹叫做cat，dog和bat，里面分别是猫，狗和蝙蝠的照片。为了训练一个三分类模型，我们先要生成一个文件，里面每一行是文件的路径和对应的标签。定义cat是0，dog是1，bat是2，则可以通过如下脚本：

```
import os

# 定义文件夹名称和标签的对应关系
label_map = {
    'cat': 0,
    'dog': 1,
    'bat': 2
}

with open('data.txt', 'w') as f:

    # 遍历所有文件，root为当前文件夹，dirs是所有子文件夹名，files是所有文件名
    for root, dirs, files in os.walk('data'):
        for filename in files:
            filepath = os.sep.join([root, filename])    # 获得文件完整路径
            dirname = root.split(os.sep)[-1]                    # 获取当前文件夹名称
            label = label_map[dirname]                          # 得到标签
            line = '{},{}\n'.format(filepath, label)
            f.write(line)
```

其中，os.sep是当前操作系统的路径分隔符，在Unix/Linux中是'/'，Windows中是'\\'。有的时候我们已经有了所有的文件在一个文件夹data下，希望获取所有文件的名称，则可以用os.listdir():

```
filenames = os.listdir('data')
```

os也提供了诸如拷贝，移动和修改文件名等操作。同时因为大部分深度学习框架最常见的都是在Unix/Linux下使用，并且Unix/Linux的shell已经非常强大（比Windows好用太多），所以只需要用字符串格式化等方式生成shell命令的字符串，然后通过os.system()就能方便实现很多功能，有时比os，还有Python中另一个操作系统相关模块shutil还要方便：

```
import os, shutil
```

```
filepath1 = 'data/bat/IMG_000000.jpg'

# 修改文件名
os.system('mv {} {}'.format(filepath0, filepath1))
#os.rename(filepath0, filepath1)


# 创建文件夹
dirname = 'data_samples'
os.system('mkdir -p {}'.format(dirname))
#if not os.path.exists(dirname):
#    os.mkdir(dirname)

# 拷贝文件
os.system('cp {} {}'.format(filepath1, dirname))
#shutil.copy(filepath1, dirname)
```

下篇：给深度学习入门者的Python快速教程 - numpy和Matplotlib篇

「真诚赞赏，手留余香」

赞赏

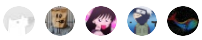
人赞赏

Python　Python 入门

深度学习（Deep Learning）

367

收藏　分享　举报



文章被以下专栏收录

From Beijing with Love

视觉、计算

进入专栏

57 条评论

写下你的评论...

1　2　3　4　...　6　下一页

推荐阅读

给深度学习入门者的Python快速教程 - 番外篇之 Python-OpenCV

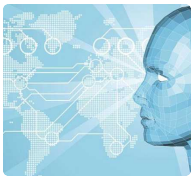
本篇是前面两篇教程：给深度学习入门者的Python快速教程 - 基础篇 给深度学习入门者的Python... 查看全文 >

逢閰西 · 1 年前

Python入门 函数 基础篇

一、什么是函数函数是最基本的一种代码抽象的方式，是组织好的可重复使用的用来实现单一或相关联功能的代码段。函数是对做相似的事情或相似的动作进行封装，它能提高应用的模块性和代码的重... 查看全文 >

木头人 · 4 个月前



Python深度学习完全路线指南

介绍深度学习目前已经成为了人工智能领域的突出话题。它在“计算机视觉”和游戏（AlphaGo）... 查看全文 >

地球的外星人君 · 4 个月前

Python基础入门

学习Python的秘诀：多敲代码 + 学会使用搜索引擎一、搭建编程环境工欲善其事，必先利其器，... 查看全文 >

陈容喜 · 9 天前 · 发表于 Python数据分析之路