机器学习算法day01\_python快速上手

# 课程大纲

|  |  |
| --- | --- |
| Python快速上手 | Python简介 |
| Python集成开发环境 |
| Python基本语法 |
| Python的变量和集合 |
| Python流程控制语法 |
| Python函数 |
| Python模块 |
| Numpy矩阵运算包 | Numpy简介 |
| Numpy中的多维数组ndarray |
| Ndarray常用方法 |
| 数组的基本运算 |
| 数组的形状操作 |
| 数据挖掘与机器学习导论 | 数据挖掘概念 |
| 数据挖掘与机器学习的关系 |
| 机器学习简介 |
| 机器学习的应用步骤 |

课程目标：

1. 在已有JAVA语言的基础上迅速实现python上手开发
2. 掌握Numpy矩阵算法包的核心功能
3. 了解数据挖掘与机器学习算法的概念及其关系

# 1 Python快速上手

## 1.1.Python简介

Python 是一个高层次的结合了解释性、编译性、互动性和面向对象的脚本语言。

Python 的设计具有很强的可读性，相比其他语言经常使用英文关键字，其他语言的一些标点符号，它具有比其他语言更有特色语法结构。

* Python是一种解释型语言： 这意味着开发过程中没有了编译这个环节。类似于PHP和Perl语言。
* Python是交互式语言： 这意味着，您可以在一个Python提示符，直接互动执行写你的程序。
* Python是面向对象语言: 这意味着Python支持面向对象的风格或代码封装在对象的编程技术。
* Python是初学者的语言：Python 对初级程序员而言，是一种伟大的语言，它支持广泛的应用程序开发，从简单的文字处理到WWW浏览器再到游戏

## 12.Python集成开发环境

### 1.2.1 Python安装

Python已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。

可以直接下载相应平台的二进制代码，然后安装Python，或者使用C编译器手动编译源代码。编译的源代码，功能上有更多的选择性，为python安装提供了更多的灵活性。

Python版本的选择：

有两大系列 python 2.x

Python 3.x

以下为不同平台上安装Python的方法：

**1、Unix & Linux 平台安装 Python:**

打开WEB浏览器访问http://www.python.org/download/

选择适用于Unix/Linux的源码压缩包。

下载及解压压缩包。

如果你需要自定义一些选项修改Modules/Setup

执行 ./configure 脚本

make

make install

执行以上操作后，Python会安装在 /usr/local/bin目录中，Python库安装在/usr/local/lib/pythonXX，XX为你使用的Python的版本号。

**2、Window 平台安装 Python:**

打开WEB浏览器访问http://www.python.org/download/

在下载列表中选择Window平台安装包，包格式为：python-XYZ.msi 文件 ， XYZ 为你要安装的版本号。

下载后，双击下载包，进入Python安装向导，安装非常简单，你只需要使用默认的设置一直点击"下一步"直到安装完成即可。

**3、环境变量配置**

程序和可执行文件可以在许多目录，而这些路径很可能不在操作系统提供可执行文件的搜索路径中。

path(路径)存储在环境变量中，这是由操作系统维护的一个命名的字符串。这些变量包含可用的命令行解释器和其他程序的信息。

Unix或Windows中路径变量为PATH（UNIX区分大小写，Windows不区分大小写）。

vi /etc/profile

export PATH="$PATH:/usr/local/bin/python"

在 Windows 设置环境变量

在环境变量中添加Python目录：

在命令提示框中(cmd) : 输入

path %path%;C:\Python , 按下"Enter"。

注意: C:\Python 是Python的安装目录。

**4、运行Python**

有三种方式可以运行Python：

1. **交互式解释器：**

你可以通过命令行窗口进入python并开在交互式解释器中开始编写Python代码。

你可以在Unix，DOS或任何其他提供了命令行或者shell的系统进行python编码工作。

$python # Unix/Linux

或者

C:>python # Windows/DOS

以下为Python命令行参数：

选项 描述

-d 在解析时显示调试信息

-O 生成优化代码 ( .pyo 文件 )

-S 启动时不引入查找Python路径的位置

-v 输出Python版本号

-X 从 1.6版本之后基于内建的异常（仅仅用于字符串）已过时。

-c cmd 执行 Python 脚本，并将运行结果作为 cmd 字符串。

file 在给定的python文件执行python脚本。

1. **命令行脚本**

在你的应用程序中通过引入解释器可以在命令行中执行Python脚本，如下所示：

#在 Unix/Linux下

$python script.py

# 在Windows下

C:>python script.py

注意：在执行脚本时，请检查脚本是否有可执行权限。

1. **集成开发环境（IDE：Integrated Development Environment）**

您可以使用图形用户界面（GUI）环境来编写及运行Python代码。以下推荐各个平台上使用的IDE：

IDLE 是 Linux上最早的 Python IDE。

Pycharm 是jetbrain出品的Python 集成开发环境

### 1.2.2 python依赖库管理工具pip

**pip 是一个安装和管理 Python 包的工具**，python安装包的工具有easy\_install, setuptools, pip，distribute。使用这些工具都能下载并安装python依赖包

**1、安装pip**

安装和升级之前，先下载get-pip.py

然后使用下面的命令：

python get-pip.py

不过注意一下，linux或osX下，需要权限，使用下面的命令，输入密码后即可。

sudo python get-pip.py

windows下需要管理员权限启动终端。

**2、安装setuptools**

如果你还没有安装了setuptools，get-pip.py 会帮你安装。

如果你已经安装了setuptools，可以运行下面的命令进行升级。

pip install -U setuptools

windows下，注意将pip路劲加到系统的path中。

**3、升级pip**

Linux or OS X系统，运行下面的命令:

pip install -U pip

windows系统运行下面的命令：

python -m pip install -U pip

**4、安装依赖包**

使用下面的命令来安装包

pip install SomePackage # 默认下载安装最新版本

pip install SomePackage==1.0.4 # 指定安装版本

pip install 'SomePackage>=1.0.4' # 指定最低版本

要看更多地例子，可以看这里pip install

例如要安装web开发框架库 Django，用下面的一条命令即可，方便快捷：

pip install Django==1.7

### 1.2.3 python环境一键安装

在用python做科学计算的场景中，需要安装的依赖库非常多且非常麻烦，建议用python科学计算集成环境 anaconda

**一句话点评：省事！！！给力！！！**

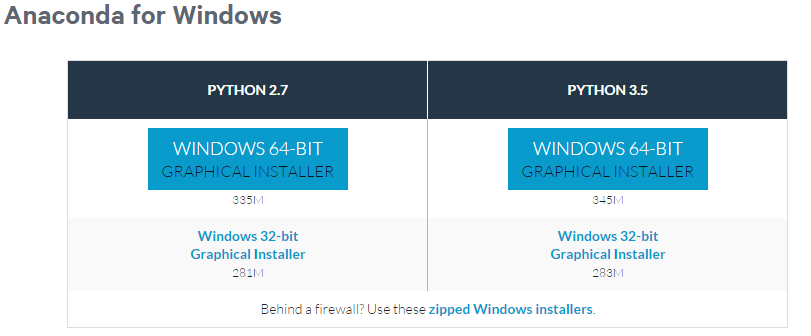
请see官方介绍：

Anaconda is a completely free Python distribution (including for commercial use and redistribution). It includes more than 400 of the most popular Python packages for science, math, engineering, and data analysis.

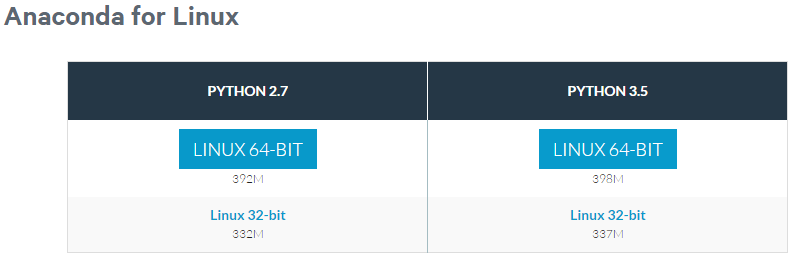
官网下载地址

<https://www.continuum.io/downloads>

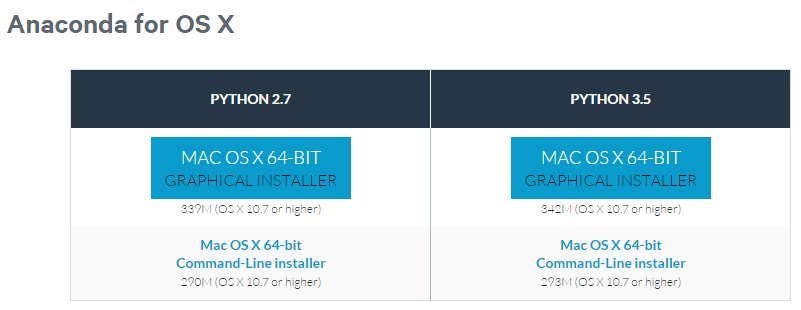
WINDOW版本：



LINUX版本：



MAC OS版本：



## 1.3.Python基本语法

### 1.3.1 行和缩进

Python中，不使用括号来表示代码的类和函数定义块或流程控制。

代码块是由行缩进，缩进位的数目是可变的，但是在块中的所有语句必须缩进相同的量。

如下所示：

|  |
| --- |
| if True:  print "True"  else:  print "False" |

然而，在本实施例中的第二块将产生一个错误：

|  |
| --- |
| if True:  print "Answer"  print "True"  else:  print "Answer"  print "False" |

### 1.3.2 Python引号

Python接受单引号（'），双引号（“）和三（''或”“”）引用，以表示字符串常量，只要是同一类型的引号开始和结束的字符串。

三重引号可以用于跨越多个行的字符串。例如，所有下列是合法的：

|  |
| --- |
| word = 'word'  sentence = "This is a sentence."  paragraph = """This is a paragraph. It is  made up of multiple lines and sentences.""" |

### 1.3.3 Python注释

“＃”号之后字符和到物理行是注释的一部分，Python解释器会忽略它们。

|  |
| --- |
| #!/usr/bin/python  # First comment  print "Hello, Python!"; # second comment  这将产生以下结果：  Hello, Python!  注释可能会在声明中表达或同一行之后：  name = "Madisetti" # This is again comment  你可以使用多行注释如下：  # This is a comment.  # This is a comment, too.  # This is a comment, too.  # I said that already. |

### 1.3.4 分号的使用

python中一个语句的结束不需要使用分号

如果想在一行中输入多个语句，可使用分号：

|  |
| --- |
| import sys; x = 'foo'; sys.stdout.write(x+"""  """) |

## 1.4.Python的变量和集合

Python有五个标准的数据类型：

1. 数字
2. 字符串
3. 列表
4. 元组
5. 字典

f) set

python中定义变量时不需要显示指定变量类型，以下为python中变量使用的典型语法：

### 1.4.1变量定义和赋值

|  |
| --- |
| #基本使用  counter = 100 # 整型  miles = 1000.0 # 浮点  name = "John" # 字符串  print counter  print miles  print name  #多重赋值  a = b = c = 1  d, e, f = 1, 2, "john" |

### 1.4.2字符串的使用

|  |
| --- |
| str = 'Hello World!' #字符串在python中本质上是一个字符序列Seq  print str # 打印整个字符串  print str[0] # 打印字符串第一个字母  print str[2:5] # 打印第3到第5个字母  print str[2:] # 打印从第3个字母到末尾  print str \* 2 # 字符串重复2次  print str + "TEST" # 字符串拼接 |

### 1.4.3列表的使用

|  |
| --- |
| list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]  tinylist = [123, 'john']  print list # 打印整个列表  print list[0] # 打印第一个元素  print list[1:3] # 打印第二和第三个元素  print list[2:] # 打印第三个元素到末尾  print tinylist \* 2 # 打印2次  print list + tinylist # 拼接两个list  #修改list中的元素  list[0]=”python”  print(list) |

将输出以下结果：

|  |
| --- |
| ['abcd', 786, 2.23, 'john', 70.200000000000003]  abcd  [786, 2.23]  [2.23, 'john', 70.200000000000003]  [123, 'john', 123, 'john']  ['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john'] |

### 1.4.4元组使用

元组是类似于列表中的序列数据类型，一个元组由数个逗号分隔的值组成。

列表和元组之间的主要区别是：列表用方括号[]，**列表的长度和元素值是可以改变的**

而**元组用圆括号()，不能被更新**。

元组可以被认为是只读列表。

|  |
| --- |
| tuple = ( 'abcd', 786 , 2.23, 'john', 70.2)  tinytuple = (123, 'john')  print tuple # 打印整个元组  print tuple[0] # 打印第一个元素  print tuple[1:3] # 打印第2、3两个元素  print tuple[2:] #  print tinytuple \* 2 # 重复2遍  print tuple + tinytuple # 拼接 |

这将产生以下结果：

|  |
| --- |
| ('abcd', 786, 2.23, 'john', 70.200000000000003)  abcd  (786, 2.23)  (2.23, 'john', 70.200000000000003)  (123, 'john', 123, 'john')  ('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john') |

### 1.4.5字典

Python字典是一种哈希表型。由“键-值”对组成。

键可以是任何Python类型，但通常是数字或字符串。

值可以是任意Python的对象。

字典是由花括号括号{}，可分配值，并用方括号[]访问。例如：

|  |
| --- |
| dict = {}  dict['one'] = "This is one"  dict[2] = "This is two"  tinydict = {'name': 'john','code':6734, 'dept': 'sales'}  print dict['one'] # Prints value for 'one' key  print dict[2] # Prints value for 2 key  print tinydict # Prints complete dictionary  print tinydict.keys() # Prints all the keys  print tinydict.values() # Prints all the values |

这将产生以下结果：

|  |
| --- |
| This is one  This is two  {'dept': 'sales', 'code': 6734, 'name': 'john'}  ['dept', 'code', 'name']  ['sales', 6734, 'john'] |

### 1.4.6 set

定义一个set：

a={1,2,3,4,5}

print a

a.remove(3)

a.add(6)

a.union(b)

### 1.4.7数据类型转换

有时候，可能需要执行的内置类型之间的转换。

类型之间的转换，只需使用类名作为函数。

|  |  |
| --- | --- |
| int(x [,base]) | 将x转换为整数。基数指定为base（进制） |
| long(x [,base] ) | 将x转换为一个长整数。基数指定为base， |
| float(x) | 将x转换到一个浮点数。 |
| complex(real [,imag]) | 创建一个复数。 |
| str(x) | 转换对象x为字符串表示形式。 |
| eval(str) | 计算一个表达式字符串，并返回一个对象。 |
| tuple(s) | 把s（序列）转换为一个元组。 |
| list(s) | 把s（序列）转换为一个列表。 |
| set(s) | 把s（序列）转换为一个set集合。 |
| dict(d) | 转成字典,d必须是（键，值）元组序列。 |

例如：

a=int(‘A’,16)

print(a)

结果为: 10

a=tuple(range(1,10,2))

print(a)

b=tuple("hello")

print b

c=complex(1,2)

print c

x=1

e=eval('x+1')

print e

f=dict([(1,2),(3,4),('a',100)])

print f

结果为：

(1, 3, 5, 7, 9)

('h', 'e', 'l', 'l', 'o')

(1+2j)

2

{'a': 100, 1: 2, 3: 4}

## 1.5.Python流程控制语法

### 1.5.1 if语句

|  |
| --- |
| var1 = 100  if var1:  print "1 - Got a true expression value"  print var1  var2 = 0  if var2:  print "2 - Got a true expression value"  print var2  print "Good bye!"  #if的条件可以是数字或字符串或者布尔值True和False（布尔表达式）  #如果是数字，则只要不等于0，就为true  #如果是字符串，则只要不是空串，就为true |

if else

|  |
| --- |
| var = 100  if var == 200:  print "1 - Got a true expression value"  print var  elif var == 150:  print "2 - Got a true expression value"  print var  elif var == 100:  print "3 - Got a true expression value"  print var  else:  print "4 - Got a false expression value"  print var  print "Good bye!" |

嵌套if else

|  |
| --- |
| var = 100  if var < 200:  print "Expression value is less than 200"  if var == 150:  print "Which is 150"  elif var == 100:  print "Which is 100"  elif var == 50:  print "Which is 50"  elif var < 50:  print "Expression value is less than 50"  else:  print "Could not find true expression"  print "Good bye!" |

### 1.5.2 while循环

|  |
| --- |
| count = 0  while count < 5:  print count, " is less than 5"  count = count + 1  else:  print count, " is not less than 5" |

### 1.5.3 for循环

|  |
| --- |
| # 求素数  for num in range(10,20):  for i in range(2,num):  if num%i == 0:  j=num/i  print '%d equals %d \* %d' % (num,i,j)  break  else:  print num, 'is a prime number'  #遍历集合  r=range(10,20)  r={1,2,3,4,5} #set类型  r=["aaa",3,"c"]  print(r)  for num in r:  print(num)  r={"a":9,"b":10}  print(r)  for num in r.values():  print(num) |

当执行上面的代码，产生以下结果：

|  |
| --- |
| 10 equals 2 \* 5  11 is a prime number  12 equals 2 \* 6  13 is a prime number  14 equals 2 \* 7  15 equals 3 \* 5  16 equals 2 \* 8  17 is a prime number  18 equals 2 \* 9  19 is a prime number |

## 1.6.Python函数

### 1.6.1 基本形式

|  |
| --- |
| #定义函数  def changeme( mylist ):  "This changes a passed list into this function"  mylist.append([1,2,3,4]);  print "Values inside the function: ", mylist  return (mylist,"haha")  # 调用函数  mylist = [10,20,30];  changeme( mylist );  print "Values outside the function: ", mylist |

python的函数调用是**引用传递，**这将产生以下结果：

|  |
| --- |
| Values inside the function: [10, 20, 30, [1, 2, 3, 4]]  Values outside the function: [10, 20, 30, [1, 2, 3, 4]] |

**默认参数和可变参数**

|  |
| --- |
| # 默认参数  #有默认值的参数后面不能再跟无默认值的参数  def printinfo( name, age = 35 ):  "This prints a passed info into this function"  print "Name: ", name;  print "Age ", age;  return;  #调用  #如果调换了参数的顺序，则必须把参数名都带上  printinfo( age=50, name="miki" );  printinfo( name="miki" );  #可变参数  def printinfo( arg1, \*vartuple ):  "This prints a variable passed arguments"  print "Output is: "  print arg1  for var in vartuple:  print var  return;  # 调用  printinfo( 10 );  printinfo( 70, 60, 50 ); |

### 1.6.2 匿名函数

* 可以使用lambda关键字来创建小的匿名函数。这些函数被称为匿名，因为它们不是以标准方式通过使用def关键字声明。
* Lambda形式可以采取任何数量的参数，但在表现形式上只返回一个值。它们不能包含命令或多个表达式。
* 匿名函数不能直接调用打印，因为需要lambda表达式。
* lambda函数都有自己的命名空间，并且不能访问变量高于在其参数列表和那些在全局命名空间的变量。

示例：

|  |
| --- |
| # 定义  sum = lambda arg1, arg2: arg1 + arg2 #lambda表达式  # 调用  print "Value of total : ", sum( 10, 20 )  print "Value of total : ", sum( 20, 20 )  ##返回多个值  tup**=lambda** x,y**:**(x**+**1,y**+**1)  c**=**tup(2,3)  print c[0],c[1] (a,b)**=**tup(2,3)  **print** a,b **print** c[0],c[1] |

利用lambda可以实现类似于scala中的高阶函数效果：

|  |
| --- |
| #!/usr/bin/python  # -\*- coding: UTF-8 -\*-  def outfunc(func,x,y):  c=func(x,y)  print(c)  outfunc(lambda x,y:x+y,1,2) |

## 1.7.Python模块

简单地说，**一个模块是由Python代码的文件**。一个模块可以定义函数，类和变量。模块还可以包括可运行的代码。

### 1.7.1 模块的定义和导入

例：以下代码定义在support.py文件中

|  |
| --- |
| def print\_func( par ):  print "Hello : ", par  return |

在别的模块比如(hello.py)中可以导入已定义好的模块

|  |
| --- |
| #!/usr/bin/python  #导入模块  import cn.itcast.test.support  # 使用导入的模块中的函数  cn.itcast.test.support.print\_func("Zara")  #------------------------------------------------  #或者  from cn.itcast.test.support import print\_func  print\_func("Zara") |

### 1.7.2 模块和包

在python中一个文件可以被看成一个独立模块，而包对应着文件夹，模块把python代码分成一些有组织的代码段，通过导入的方式实现代码重用。

#### 1.7.1 模块搜索路径

导入模块时，是按照sys.path变量的值搜索模块，sys.path的值是包含每一个独立路径的列表，包含当前目录、python安装目录、PYTHONPATH环境变量，搜索顺序按照路径在列表中的顺序（一般当前目录优先级最高）。

|  |
| --- |
| [‘/home/zhoujh/study\_workspace/studynotes/python/python\_base’, ‘/usr/local/lib/python2.6/site-packages/setuptools-0.6c11-py2.6.egg’, ‘/usr/local/lib/python2.6/site-packages/redis-2.2.1-py2.6.egg’, ‘/usr/local/lib/python2.6/site-packages/Flask-0.8-py2.6.egg’, ‘/usr/local/lib/python2.6/site-packages/Jinja2-2.6-py2.6.egg’, ‘/usr/local/lib/python2.6/site-packages/Werkzeug-0.8.3-py2.6.egg’, ‘/usr/local/lib/python2.6/site-packages/tornado-2.2.1-py2.6.egg’, ‘/usr/local/lib/python2.6/site-packages/MySQL\_python-1.2.3-py2.6-linux-x86\_64.egg’, ‘/usr/local/lib/python2.6/site-packages/PIL-1.1.7-py2.6-linux-x86\_64.egg’, ‘/usr/local/lib/python2.6/site-packages/SQLAlchemy-0.7.8-py2.6-linux-x86\_64.egg’, ‘/home/zhoujh/python\_workspace/python\_app’, ‘/usr/local/lib/python26.zip’, ‘/usr/local/lib/python2.6’, ‘/usr/local/lib/python2.6/plat-linux2’, ‘/usr/local/lib/python2.6/lib-tk’, ‘/usr/local/lib/python2.6/lib-old’, ‘/usr/local/lib/python2.6/lib-dynload’, ‘/usr/local/lib/python2.6/site-packages’] |

#### 1.7.2 导入模块

**1.7.2.1 使用import语句导入模块**

有下面两种方式

import module1

import module2

import module3

import module1,module2,module3

这两种方式的效果是一样的，但是第一种可读性比第二种好，推荐按照下面的顺序导入模块，并且一般在文件首部导入所有的模块：

* python标准库
* 第三方模块
* 应用程序自定义模块

也可以在函数内部导入模块，这样被导入的模块作用域是局部的

**1.7.2.2 使用from-import语句导入模块的属性**

单行导入

from module import name1,name2,name3

多行导入

from module import name1,name2,\

name3

导入全部属性（由于容易覆盖当前名称空间中现有的名字，所以一般不推荐使用，适合模块中变量名很长并且变量很多的情况）

from module import \*

如果你不想某个模块的属性被以上方法导入，可以给该属性名称前加一个下划线(\_test)，如果需要取消隐藏，可以显示的导入该属性（from module import \_test）

**1.7.2.3 扩展的import语句**

使用自定义的名称替换模块的原始名称

import simplejson as json

模块被导入时，加载的时候模块顶层代码会被执行，如：设定全局变量、类和函数的声明等，所以应该把代码尽量封装到类和函数中。一个模块无论被导入多少次，只加载一次，可以防止多次导入时代码被多次执行。

**1.7.2.4 重新导入模块**

reload(module)

内建函数reload可以重新导入一个已经存在的模块

#### 1.7.3.包结构

**1.包定义结构**

包将有联系的模块组织在一起，有效避免模块名称冲突问题，让应用组织结构更加清晰。

一个普通的python应用程序目录结构：

app/

\_\_init\_\_.py

a/

\_\_init\_\_.py

a.py

b/

\_\_init\_\_.py

b.py

app是最顶层的包，a和b是它的子包，可以这样导入：

from app.a import a

from app.b.b import test

a.test()

test()

上面代码表示：导入app包的子包a和子包b的属性test，然后分别调用test方法。

**2. \_\_init\_\_.py的作用**

每个目录下都有\_\_init\_\_.py文件，这个是初始化模块，from-import语句导入子包时需要它，可以在里面做一些初始化工作，也可以是空文件。

ps：\_\_init\_\_.py定义的属性直接使用 顶层包.子包 的方式导入，如在目录a的\_\_init\_\_.py文件中定义init\_db()方法，调用如下：

from app import a

a.init\_db()

**3. 指定python文件编码方式**

python默认是使用ASCII编码，可以指定编码方式，如

|  |
| --- |
| #!/usr/bin/env python  #coding=utf-8  或者  #!/usr/bin/env python  # -\*- coding:utf-8 -\*- |

**4. 解决导入循环问题**

有下面两个模块，a.py和b.py

|  |
| --- |
| a.py  #!/usr/bin/env python  #coding=utf-8    import b    if \_\_name\_\_ == '\_\_main':  print 'hello,I'm a' |

b.py

|  |
| --- |
| #!/usr/bin/env python  #coding=utf-8    import a    if \_\_name\_\_ == '\_\_main':  print 'hello,I'm b' |

在这里a尝试导入b，而b也尝试导入a，导入一个先前没有完全导入的模块，会导致导入失败。解决办法：移除一个导入语句，把导入语句放到函数内部，在需要的时候导入。

b.py

|  |
| --- |
| #!/usr/bin/env python  #coding=utf-8    if \_\_name\_\_ == '\_\_main':  import a  print 'hello,I'm b' |

## 1.8.Python文件IO

### 1.8.1 文件读写

**[Python](http://lib.csdn.net/base/11" \o "Python知识库" \t "http://blog.csdn.net/cnmilan/article/details/_blank)**进行文件读写的函数为open或file:

file\_handler = open(filename,,mode）

**open mode**

|  |  |
| --- | --- |
| w | 以写方式打开文件，可向文件写入信息。如文件存在，则清空该文件，再写入新内容 |
| a | 以追加模式打开文件（即一打开文件，文件指针自动移到文件末尾），如果文件不存在则创建 |
| r+ | 以读写方式打开文件，可对文件进行读和写操作。 |
| w+ | 消除文件内容，然后以读写方式打开文件。 |
| a+ | 以读写方式打开文件，并把文件指针移到文件尾。 |
| b | 以二进制模式打开文件，而不是以文本模式。该模式只对Windows或Dos有效，类Unix的文件是用二进制模式进行操作的。 |

**操作文件对象方法**

|  |  |
| --- | --- |
| f.close() | 关闭文件，记住用open()打开文件后一定要记得关闭它，否则会占用系统的可打开文件句柄数。 |
| f.fileno() | 获得文件描述符，是一个数字 |
| f.flush() | 刷新输出缓存 |
| f.isatty() | 如果文件是一个交互终端，则返回True，否则返回False。 |
| f.read([count]) | 读出文件，如果有count，则读出count个字节。 |
| f.readline() | 读出一行信息。 |
| f.readlines() | 读出所有行，也就是读出整个文件的信息。 |
| f.seek(offset[,where]) | 把文件指针移动到相对于where的offset位置。where为0表示文件开始处，这是默认值 ；1表示当前位置；2表示文件结尾。 |
| f.tell() | 获得文件指针位置。 |
| f.truncate([size]) | 截取文件，使文件的大小为size。 |
| f.write(string) | 把string字符串写入文件。 |
| f.writelines(list) | 把list中的字符串一行一行地写入文件，是连续写入文件，没有换行。 |

例1：从文本文件中每读取一行文本便输出

|  |
| --- |
| #!/usr/bin/env/ python  #coding=utf-8  fileHandler = open('/root/a.txt', 'a+') #以读写方式处理文件IO  fileHandler.seek(0)  line = fileHandler.readline()  while line:  print line  line = fileHandler.readline()  fileHandler.close |

例2:其他文件IO函数的使用

|  |
| --- |
| #!/usr/bin/env/ python  #coding=utf-8  fileHandler = open('/root/a.txt', 'a+') #以读写方式处理文件IO  fileHandler.seek(0)  #读取整个文件  contents = fileHandler.read()  print contents  #读取所有行,再逐行输出  fileHandler.seek(0)  lines = fileHandler.readlines()  for line in lines:  print line  #当前文件指针的位置  print fileHandler.tell()  fileHandler.close |

例3:用file(...)替换open(...)

|  |
| --- |
| #!/usr/bin/env/ python  #coding=utf-8  fileHandler = file('/root/a.txt', 'a+') #以读写方式处理文件IO  fileHandler.seek(0)  line = fileHandler.readline()  while line:  print line  line = fileHandler.readline() |

例4:文件的写操作

|  |
| --- |
| #!/usr/bin/env/ python  #coding=utf-8  fileHandler = file('/root/a.txt','a+') #或者调用open()函数  fileHandler.write("\r\n")  fileHandler.write("thank you")  fileHandler.seek(0)  contents = fileHandler.read()  print contents  fileHandler.close |

### 1.8.2 文件夹相关操作

**[Python](http://lib.csdn.net/base/11" \o "Python知识库" \t "http://blog.csdn.net/cnmilan/article/details/_blank)中对文件、文件夹（文件操作函数）的操作需要涉及到os模块和shutil模块。**

得到当前工作目录，即当前Python脚本工作的目录路径: **os.getcwd()**

返回指定目录下的所有文件和目录名:**os.listdir()**

删除一个文件:**os.remove()**

删除多个目录（只能删除空目录）：**os.removedirs（r”c：\python”）**

检验给出的路径是否是一个文件：**os.path.isfile()**

检验给出的路径是否是一个目录：**os.path.isdir()**

判断是否是绝对路径：**os.path.isabs()**

检验给出的路径是否存在:**os.path.exists()**

返回一个路径的目录名和文件名:**os.path.split()**

Eg：

os.path.split('/home/swaroop/byte/code/poem.txt')

结果：('/home/swaroop/byte/code', 'poem.txt')

分离扩展名：**os.path.splitext()**

获取路径名：**os.path.dirname()**

获取文件名：**os.path.basename()**

运行shell命令: **os.system()**

读取和设置环境变量:**os.getenv() 与os.putenv()**

给出当前平台使用的行终止符:**os.linesep**Windows使用'\r\n'，Linux使用'\n'而Mac使用'\r'

指示你正在使用的平台：**os.name**      对于Windows，它是'nt'，而对于Linux/Unix用户，它是'posix'

重命名：**os.rename（old， new）**

创建多级目录：**os.makedirs（r“c：\python\test”）**

创建单个目录：**os.mkdir（“test”）**

获取文件属性：**os.stat（file）**

修改文件权限与时间戳：**os.chmod（file）**

终止当前进程：**os.exit（）**

获取文件大小：**os.path.getsize（filename）**

## 1.9 Python多线程

Python中的多线程是伪线程；不能充分利用cpu中的多核，但是在io等待型的场景下多线程还是可以提高效率

Python中的多线程有多种实现方式，利用threading包实现是比较普遍的做法

示例代码如下：

|  |
| --- |
| import threading  from time import ctime,sleep  def music(func):  for i in range(2):  print("i was listening to %s. %s" %(func,ctime()))  sleep(1)  def movie(func):  for i in range(2):  print("i was at the %s! %s" %(func,ctime()))  sleep(5)  threads=[]  t1=threading.Thread(target=music,args=(u'爱情买卖'))  threads.append(t1)  t2=threading.Thread(target=movie,args=(u'阿凡达',))  threads.append(t2)  # if \_\_name\_\_ == '\_\_main\_\_' :  for t in threads:  # t.setDaemon(True)  t.start()  # t.join()  print("all over %s" %ctime()) |

## 1.10面向对象

### 1.10.1 创建类

使用class语句来创建一个新类，class之后为类的名称并以冒号结尾，如下实例:

|  |
| --- |
| class ClassName:  '类的帮助信息' #类文档字符串  class\_suite #类体 |

类的帮助信息可以通过ClassName.\_\_doc\_\_查看。

class\_suite 由类成员，方法，数据属性组成。

### 1.10.2 实例

以下是一个简单的Python类实例:

|  |
| --- |
| #!/usr/bin/python  # -\*- coding: UTF-8 -\*-  class Employee:  '所有员工的基类'  empCount = 0  #构造函数  def \_\_init\_\_(self, name, salary):  self.name = name  self.salary = salary  Employee.empCount += 1    def displayCount(self):  print "Total Employee %d" % Employee.empCount  def displayEmployee(self):  print "Name : ", self.name, ", Salary: ", self.salary |

empCount变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用Employee.empCount访问。

第一个方法\_\_init\_\_()方法是一种特殊的方法，被称为类的**构造函数或初始化方法**，当创建了这个类的实例时就会调用该方法

类的方法

使用def关键字可以为类定义一个方法，与一般函数定义不同，类方法必须包含参数self,且为第一个参数

### 1.10.3 创建实例对象

要创建一个类的实例，你可以使用类的名称，并通过\_\_init\_\_方法接受参数。

|  |
| --- |
| "创建 Employee 类的第一个对象"  emp1 = Employee("Zara", 2000)  "创建 Employee 类的第二个对象"  emp2 = Employee("Manni", 5000) |

访问属性

可以使用点(.)来访问对象的属性。使用如下类的名称访问类变量:

|  |
| --- |
| emp1.displayEmployee()  emp2.displayEmployee()  print "Total Employee %d" % Employee.empCount |

完整实例：

|  |
| --- |
| #!/usr/bin/python  # -\*- coding: UTF-8 -\*-  class Employee:  '所有员工的基类'  empCount = 0  def \_\_init\_\_(self, name, salary):  self.name = name  self.salary = salary  Employee.empCount += 1    def displayCount(self):  print "Total Employee %d" % Employee.empCount  def displayEmployee(self):  print "Name : ", self.name, ", Salary: ", self.salary  "创建 Employee 类的第一个对象"  emp1 = Employee("Zara", 2000)  "创建 Employee 类的第二个对象"  emp2 = Employee("Manni", 5000)  emp1.displayEmployee()  emp2.displayEmployee()  print "Total Employee %d" % Employee.empCount |

执行以上代码输出结果如下：

|  |
| --- |
| Name : Zara ,Salary: 2000  Name : Manni ,Salary: 5000  Total Employee 2 |

你可以添加，删除，修改类的属性，如下所示：

|  |
| --- |
| emp1.age = 7 # 添加一个 'age' 属性  emp1.age = 8 # 修改 'age' 属性  del emp1.age # 删除 'age' 属性  你也可以使用以下函数的方式来访问属性：  getattr(obj, ‘name’[, default]) : 访问对象的属性。  hasattr(obj,’name’) : 检查是否存在一个属性。  setattr(obj,’name’,value) : 设置一个属性。如果属性不存在，会创建一个新属性。  delattr(obj, ‘name’) : 删除属性。  hasattr(emp1, 'age') # 如果存在 'age' 属性返回 True。  getattr(emp1, 'age') # 返回 'age' 属性的值  setattr(emp1, 'age', 8) # 添加属性 'age' 值为 8  delattr(empl, 'age') # 删除属性 'age' |

### 1.10.4 Python内置类属性

|  |
| --- |
| \_\_dict\_\_ : 类的属性（包含一个字典，由类的数据属性组成）  \_\_doc\_\_ :类的文档字符串  \_\_name\_\_: 类名  \_\_module\_\_: 类定义所在的模块（类的全名是'\_\_main\_\_.className'，如果类位于一个导入模块mymod中，那么className.\_\_module\_\_ 等于 mymod）  \_\_bases\_\_ : 类的所有父类构成元素（包含了以个由所有父类组成的元组） |

Python内置类属性调用实例如下：

|  |
| --- |
| #!/usr/bin/python  # -\*- coding: UTF-8 -\*-  class Employee:  '所有员工的基类'  empCount = 0  def \_\_init\_\_(self, name, salary):  self.name = name  self.salary = salary  Employee.empCount += 1    def displayCount(self):  print "Total Employee %d" % Employee.empCount  def displayEmployee(self):  print "Name : ", self.name, ", Salary: ", self.salary  print "Employee.\_\_doc\_\_:", Employee.\_\_doc\_\_  print "Employee.\_\_name\_\_:", Employee.\_\_name\_\_  print "Employee.\_\_module\_\_:", Employee.\_\_module\_\_  print "Employee.\_\_bases\_\_:", Employee.\_\_bases\_\_  print "Employee.\_\_dict\_\_:", Employee.\_\_dict\_\_ |

执行以上代码输出结果如下：

Employee.\_\_doc\_\_: 所有员工的基类

Employee.\_\_name\_\_: Employee

Employee.\_\_module\_\_: \_\_main\_\_

Employee.\_\_bases\_\_: ()

Employee.\_\_dict\_\_: {'\_\_module\_\_': '\_\_main\_\_', 'displayCount': <function displayCount at 0x10a939c80>, 'empCount': 0, 'displayEmployee': <function displayEmployee at 0x10a93caa0>, '\_\_doc\_\_': '\xe6\x89\x80\xe6\x9c\x89\xe5\x91\x98\xe5\xb7\xa5\xe7\x9a\x84\xe5\x9f\xba\xe7\xb1\xbb', '\_\_init\_\_': <function \_\_init\_\_ at 0x10a939578>}

### 1.10.5 私有属性

1、类的私有属性

\_\_private\_attrs：**两个下划线开头，声明该属性为私有**，不能在类地外部被使用或直接访问。在类内部的方法中使用时 self.\_\_private\_attrs

2、类的私有方法

\_\_private\_method：两个下划线开头，声明该方法为私有方法，不能在类地外部调用。在类的内部调用 self.\_\_private\_methods

3、实例

|  |
| --- |
| #!/usr/bin/python  # -\*- coding: UTF-8 -\*-  class JustCounter:  \_\_secretCount = 0 # 私有变量  publicCount = 0 # 公开变量  def count(self):  self.\_\_secretCount += 1  self.publicCount += 1  print self.\_\_secretCount  counter = JustCounter()  counter.count()  counter.count()  print counter.publicCount  print counter.\_\_secretCount # 报错，实例不能访问私有变量 |

运行结果会报错：

|  |
| --- |
| Traceback (most recent call last):  File "test.py", line 17, in <module>  print counter.\_\_secretCount # 报错，实例不能访问私有变量  AttributeError: JustCounter instance has no attribute '\_\_secretCount' |

Python不允许实例化的类访问私有数据，但你可以使用 object.\_className\_\_attrName 访问属性，将如下代码替换以上代码的最后一行代码：

.........................

print counter.\_JustCounter\_\_secretCount

执行以上代码，执行结果如下：

|  |
| --- |
| 1  2  2  2 |

### 1.10.6 python对象销毁(垃圾回收)

同Java语言一样，Python使用了引用计数这一简单技术来追踪内存中的对象。

在Python内部记录着所有使用中的对象各有多少引用。

一个内部跟踪变量，称为一个引用计数器。

当对象被创建时， 就创建了一个引用计数， 当这个对象不再需要时， 也就是说， 这个对象的引用计数变为0 时， 它被垃圾回收。但是回收不是"立即"的， 由解释器在适当的时机，将垃圾对象占用的内存空间回收。

a = 40 # 创建对象 <40>

b = a # 增加引用， <40> 的计数

c = [b] # 增加引用. <40> 的计数

del a # 减少引用 <40> 的计数

b = 100 # 减少引用 <40> 的计数

c[0] = -1 # 减少引用 <40> 的计数

垃圾回收机制不仅针对引用计数为0的对象，同样也可以处理循环引用的情况。循环引用指的是，两个对象相互引用，但是没有其他变量引用他们。这种情况下，仅使用引用计数是不够的。Python 的垃圾收集器实际上是一个引用计数器和一个循环垃圾收集器。作为引用计数的补充， 垃圾收集器也会留心被分配的总量很大（及未通过引用计数销毁的那些）的对象。 在这种情况下，解释器会暂停下来，试图清理所有未引用的循环。

实例

析构函数 \_\_del\_\_

\_\_del\_\_在对象销毁的时候被调用，当对象不再被使用时，\_\_del\_\_方法运行：

|  |
| --- |
| #!/usr/bin/python  # -\*- coding: UTF-8 -\*-  class Point:  def \_\_init\_\_( self, x=0, y=0):  self.x = x  self.y = y  def \_\_del\_\_(self):  class\_name = self.\_\_class\_\_.\_\_name\_\_  print class\_name, "销毁"  pt1 = Point()  pt2 = pt1  pt3 = pt1  print id(pt1), id(pt2), id(pt3) # 打印对象的id  del pt1  del pt2  del pt3 |

以上实例运行结果如下：

3083401324 3083401324 3083401324

Point 销毁

### 1.10.7 类的继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。继承完全可以理解成类之间的类型和子类型关系。

1、语法：

派生类的声明，与他们的父类类似，继承的基类列表跟在类名之后，如下所示：

class SubClassName (ParentClass1[, ParentClass2, ...]):

'Optional class documentation string'

class\_suite

2、实例：

|  |
| --- |
| #!/usr/bin/python  # -\*- coding: UTF-8 -\*-  class Parent: # 定义父类  parentAttr = 100  def \_\_init\_\_(self):  print "调用父类构造函数"  def parentMethod(self):  print '调用父类方法'  def setAttr(self, attr):  Parent.parentAttr = attr  def getAttr(self):  print "父类属性 :", Parent.parentAttr  class Child(Parent): # 定义子类  def \_\_init\_\_(self):  print "调用子类构造方法"  def childMethod(self):  print '调用子类方法 child method'  c = Child() # 实例化子类  c.childMethod() # 调用子类的方法  c.parentMethod() # 调用父类方法  c.setAttr(200) # 再次调用父类的方法  c.getAttr() # 再次调用父类的方法 |

以上代码执行结果如下：

调用子类构造方法

调用子类方法 child method

调用父类方法

父类属性 : 200

你可以继承多个类

class A: # 定义类 A

.....

class B: # 定义类 B

.....

class C(A, B): # 继承类 A 和 B

.....

可以使用issubclass()或者isinstance()方法来检测。

issubclass() - 布尔函数判断一个类是另一个类的子类或者子孙类，语法：issubclass(sub,sup)

isinstance(obj, Class) 布尔函数如果obj是Class类的实例对象或者是一个Class子类的实例对象则返回true。

3、方法重写

如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法：

实例：

|  |
| --- |
| #!/usr/bin/python  # -\*- coding: UTF-8 -\*-  class Parent: # 定义父类  def myMethod(self):  print '调用父类方法'  class Child(Parent): # 定义子类  def myMethod(self):  print '调用子类方法'  c = Child() # 子类实例  c.myMethod() # 子类调用重写方法 |

执行以上代码输出结果如下：

4、基础重载方法

下表列出了一些通用的功能，你可以在自己的类重写：

1/ \_\_init\_\_ ( self [,args...] )

构造函数

简单的调用方法: obj = className(args)

2/ \_\_del\_\_( self )

析构方法, 删除一个对象

简单的调用方法 : dell obj

3/ \_\_str\_\_( self )

用于将值转化为适于人阅读的形式

简单的调用方法 : str(obj)

4/ \_\_cmp\_\_ ( self, x )

对象比较

简单的调用方法 : cmp(obj, x)

|  |
| --- |
| #!/usr/bin/python  class Vector:  def \_\_init\_\_(self, a, b):  self.a = a  self.b = b  def \_\_str\_\_(self):  return 'Vector (%d, %d)' % (self.a, self.b)    def \_\_add\_\_(self,other):  return Vector(self.a + other.a, self.b + other.b)  v1 = Vector(2,10)  v2 = Vector(5,-2)  print v1 + v2 |

以上代码执行结果如下所示:

Vector(7,8)

# 2 Numpy快速上手

## 2.1. 什么是Numpy

Numpy是Python的一个科学计算的库

**主要提供矩阵运算的功能，而矩阵运算在机器学习领域应用非常广泛**

Numpy一般与Scipy、matplotlib一起使用。

虽然python中的list已经提供了类似于矩阵的表示形式，不过numpy为我们提供了更多的函数。

### 2.1.2 安装导入了Numpy

（通用做法import numpy as np 简单输入）

|  |
| --- |
| >>> import numpy as np  >>> print np.version.version  1.6.2 |

### 2.1.3 Numpy组成

Numpy基础部分中，有两个主要内容，如下：

任意维数的数组对象（ndarray，n-dimensional array object）

通用函数对象（ufunc，universal function object）

## 2.2. 多维数组

### 2.2.1 Numpy中的数组<矩阵>

Numpy中，最重要的数据结构是：多维数组的类型（numpy.ndarray）

ndarray由两部分组成：

实际所持有的数据；

描述这些数据的元数据（metadata）

与Python原生支持的List类型不同，数组的所有元素必须同样的类型。

**数组（即矩阵）的维度被称为axes，维数称为 rank**

ndarray 的重要属性包括:

* ndarray.ndim：数组的维数，也称为rank
* ndarray.shape：数组各维的大小，对一个n行m列的矩阵来说， shape 为 (n,m)
* ndarray.size：元素的总数。
* ndarray.dtype：每个元素的类型，可以是numpy.int32, numpy.int16, and numpy.float64等
* ndarray.itemsize：每个元素占用的字节数。
* ndarray.data：指向数据内存。

### 2.2.2 ndarray常用方法示例

#### 2.2.2.2 使用numpy.array方法

以list或tuple变量为参数产生一维数组：

|  |
| --- |
| >>> print np.array([1,2,3,4])  [1 2 3 4]  >>> print np.array((1.2,2,3,4))  [ 1.2 2. 3. 4. ]  >>> print type(np.array((1.2,2,3,4)))  <type 'numpy.ndarray'> |

以list或tuple变量为元素产生二维数组或者多维数组：

|  |
| --- |
| >>> x = np.array(((1,2,3),(4,5,6)))  >>> x  array([[1, 2, 3],  [4, 5, 6]])  >>> y = np.array([[1,2,3],[4,5,6]])  >>> y  array([[1, 2, 3],  [4, 5, 6]])  index 和slicing ：第一数值类似数组横坐标，第二个为纵坐标  >>> x[1,2]  6  >>> y=x[:,1] #取第二列  >>> y  array([2, 5])  涉及改变相关问题，我们改变上面y是否会改变x？这是特别需要关注的！  >>> y[0] = 10  >>> y  array([10, 5])  >>> x  array([[ 1, 10, 3],  [ 4, 5, 6]]) |

通过上面可以发现改变y会改变x ，因而我们可以推断，y和x指向是同一块内存空间值，系统没有为y 新开辟空间把x值赋值过去。

#### 2.2.2.3 使用numpy.arange方法

|  |
| --- |
| >>> print np.arange(15)  [ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14]  >>> print type(np.arange(15))  <type 'numpy.ndarray'>  >>> print np.arange(15).reshape(3,5)  [[ 0 1 2 3 4]  [ 5 6 7 8 9]  [10 11 12 13 14]]  >>> print type(np.arange(15).reshape(3,5))  <type 'numpy.ndarray'> |

#### 2.2.2.4 使用numpy.linspace方法

例如，在从1到10中产生20个数：

|  |
| --- |
| >>> print np.linspace(1,10,20)  [ 1. 1.47368421 1.94736842 2.42105263 2.89473684  3.36842105 3.84210526 4.31578947 4.78947368 5.26315789  5.73684211 6.21052632 6.68421053 7.15789474 7.63157895  8.10526316 8.57894737 9.05263158 9.52631579 10. ] |

使用numpy.zeros，numpy.ones，numpy.eye等方法可以构造特定的矩阵

构造“0”矩阵：

|  |
| --- |
| >>> print np.zeros((3,4))  [[ 0. 0. 0. 0.]  [ 0. 0. 0. 0.]  [ 0. 0. 0. 0.]] |

构造“1”矩阵

|  |
| --- |
| >>> print np.ones((3,4))  [[ 1. 1. 1. 1.]  [ 1. 1. 1. 1.]  [ 1. 1. 1. 1.]] |

构造单位矩阵(E矩阵)

|  |
| --- |
| >>> print np.eye(3)  [[ 1. 0. 0.]  [ 0. 1. 0.]  [ 0. 0. 1.]] |

#### 2.2.2.5 获取数组的属性：

|  |
| --- |
| >>> a = np.zeros((2,2,2))  >>> print a.ndim #数组的维数  3  >>> print a.shape #数组每一维的大小  (2, 2, 2)  >>> print a.size #数组的元素数  8  >>> print a.dtype #元素类型  float64  >>> print a.itemsize #每个元素所占的字节数  8 |

### 2.2.3 数组的基本运算

数组的算术运算是按元素逐个运算。数组运算后将创建包含运算结果的新数组。

与其他矩阵语言不同，NumPy中的乘法运算符\*按元素逐个计算，矩阵乘法可以使用dot函数或创建矩阵对象实现（后续介绍）

#### 2.2.3.1 数组的加减运算

|  |
| --- |
| >>> a= np.array([20,30,40,50])  >>> b= np.arange( 4)  >>> b  array([0, 1, 2, 3])  >>> c= a-b  >>> c  array([20, 29, 38, 47]) |

将运算结果更新原数组，不创建新数组

|  |
| --- |
| >>> a= np.ones((2,3), dtype=int)  >>> b= np.random.random((2,3)) ##生成2\*3矩阵，元素为[0,1)范围的随机数  >>> a\*= 3  >>> a  array([[3, 3, 3],  　　 [3, 3, 3]])  >>> b+= a #a转换为浮点类型相加  >>> b  array([[ 3.69092703, 3.8324276, 3.0114541],  　　　 [ 3.18679111, 3.3039349, 3.37600289]])  >>> a+= b # b转换为整数类型报错  TypeError: Cannot cast ufunc add output from dtype('float64') to dtype('int32') with casting rule 'same\_kind' |

当数组中存储的是不同类型的元素时，数组将使用占用更多位（bit）的数据类型作为其本身的数据类型，也就是偏向更精确的数据类型(这种行为叫做upcast)。

|  |
| --- |
| >>> a= np.ones(3, dtype=np.int32)  >>> b= np.linspace(0,np.pi,3)  >>> b.dtype.name  'float64'  >>> c= a+b  >>> c  array([ 1., 2.57079633, 4.14159265])  >>> 'float64' |

#### 2.2.3.2 数组乘法运算

|  |
| --- |
| >>> b\*\*2  array([0, 1, 4, 9])  >>> 10\*np.sin(a)  array([ 9.12945251,-9.88031624, 7.4511316, -2.62374854])  >>> a<35  array([True, True, False, False], dtype=bool) |

#### 2.2.3.3 数组内部运算

许多非数组运算，如计算数组所有元素之和，都作为ndarray类的方法来实现，使用时需要用ndarray类的实例来调用这些方法。

二维数组：

|  |
| --- |
| >>> np.sum([[0, 1], [0, 5]])  6  >>> np.sum([[0, 1], [0, 5]], axis=0)  array([0, 6])  >>> np.sum([[0, 1], [0, 5]], axis=1)  array([1, 5]) |

|  |
| --- |
| >>> b= np.arange(12).reshape(3,4)  >>> b  array([[ 0, 1, 2, 3],  [ 4, 5, 6, 7],  [ 8, 9, 10, 11]])  >>> b.sum(axis=0) # 计算每一列的和  array([12, 15, 18, 21])  >>> b.min(axis=1) # 获取每一行的最小值  array([0, 4, 8])  >>> b.cumsum(axis=1) # 计算每一行的累积和  array([[ 0, 1, 3, 6],  [ 4, 9, 15, 22],  [ 8, 17, 27, 38]]) |

三维数组：

|  |
| --- |
| >>> x  array([[[ 0, 1, 2],  [ 3, 4, 5],  [ 6, 7, 8]],  [[ 9, 10, 11],  [12, 13, 14],  [15, 16, 17]],  [[18, 19, 20],  [21, 22, 23],  [24, 25, 26]]])  >>> x.sum(axis=1)  array([[ 9, 12, 15],  [36, 39, 42],  [63, 66, 69]])  >>> x.sum(axis=2)  array([[ 3, 12, 21],  [30, 39, 48],  [57, 66, 75]]) |

求元素最值

|  |
| --- |
| >>> a= np.random.random((2,3))  >>> a  array([[ 0.65806048, 0.58216761, 0.59986935],[ 0.6004008, 0.41965453, 0.71487337]])  >>> a.sum()  　 3.5750261436902333  >>> a.min()  0.41965453489104032  >>> a.max()  0.71487337095581649 |

#### 2.2.3.4 数组的索引、切片

和列表和其它Python序列一样，一维数组可以进行索引、切片和迭代操作。

|  |
| --- |
| >>> a= np.arange(10)\*\* 3 #记住，操作符是对数组中逐元素处理的！  >>> a  array([0, 1, 8, 27, 64, 125, 216, 343, 512, 729])  >>> a[2]  8  >>> a[2:5]  array([ 8, 27, 64])  >>> a[:6:2]= -1000 # 等同于a[0:6:2]= -1000，从开始到第6个位置，每隔一个元素将其赋值为-1000  >>> a  array([-1000, 1,-1000, 27,-1000, 125, 216, 343, 512, 729])  >>> a[: :-1] # 反转a  array([ 729, 512, 343, 216, 125,-1000, 27,-1000, 1,-1000])  >>>for i in a:  ... print i\*\*2,  ...  1000000 1 1000000 729 1000000 15625 46656 117649 262144 531441 |

多维数组可以每个轴有一个索引。这些索引由一个逗号分割的元组给出。

|  |
| --- |
| >>>def f(x,y):  ... return 10\*x+y  ...  >>> b= np.fromfunction(f,(5,4),dtype=int) #fromfunction是一个函数  >>> b  array([[ 0, 1, 2, 3],  [10, 11, 12, 13],  [20, 21, 22, 23],  [30, 31, 32, 33],  [40, 41, 42, 43]])  >>> b[2,3]  23  >>> b[0:5, 1] # 每行的第二个元素  array([ 1, 11, 21, 31, 41])  >>> b[: ,1] # 与前面的效果相同  array([ 1, 11, 21, 31, 41])  >>> b[1:3,: ] # 每列的第二和第三个元素  array([[10, 11, 12, 13],  [20, 21, 22, 23]]) |

当少于提供的索引数目少于轴数时，已给出的数值按秩的顺序复制，缺失的索引则默认为是整个切片：

|  |
| --- |
| >>> b[-1] # 最后一行，等同于b[-1,:]，-1是第一个轴，而缺失的认为是：，相当于整个切片。  array([40, 41, 42, 43]) |

b[i]中括号中的表达式被当作i和一系列"："，来代表剩下的轴。NumPy也允许你使用“点”像b[i,...]。

点(…)代表许多产生一个完整的索引元组必要的冒号。如果x是秩为5的数组(即它有5个轴)，那么:

* x[1,2,…] 等同于 x[1,2,:,:,:],
* x[…,3] 等同于 x[:,:,:,:,3]
* x[4,…,5,:] 等同 x[4,:,:,5,:]

|  |
| --- |
| >>> c= array( [ [[ 0, 1, 2], #三维数组（n个2维数组叠加而成）  ...[ 10, 12, 13]],  ...  ...[[100,101,102],  ...[110,112,113]]] )  >>> c.shape  (2, 2, 3)  >>> c[1,...] #等同于c[1,:,:]或c[1]  array([[100, 101, 102],  [110, 112, 113]])  >>> c[...,2] #等同于c[:,:,2]  array([[ 2, 13],  [102, 113]]) |

#### 2.2.3.5 矩阵的遍历

|  |
| --- |
| >>>for row in b:  ... print row  ...  [0 1 2 3]  [10 11 12 13]  [20 21 22 23]  [30 31 32 33]  [40 41 42 43] |

如果想对数组中每个元素都进行处理，可以使用flat属性，该属性是一个数组元素迭代器：

|  |
| --- |
| >>>for element in b.flat:  ... print element,  ...  0 1 2 3 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43 |

#### 2.2.3.6 合并数组

使用numpy下的vstack（垂直方向）和hstack（水平方向）函数：

|  |
| --- |
| >>> a = np.ones((2,2))  >>> b = np.eye(2)  >>> print np.vstack((a,b))  [[ 1. 1.]  [ 1. 1.]  [ 1. 0.]  [ 0. 1.]]  >>> print np.hstack((a,b))  [[ 1. 1. 1. 0.]  [ 1. 1. 0. 1.]] |

看一下这两个函数有没有涉及到浅拷贝这种问题：

|  |
| --- |
| >>> c = np.hstack((a,b))  >>> print c  [[ 1. 1. 1. 0.]  [ 1. 1. 0. 1.]]  >>> a[1,1] = 5  >>> b[1,1] = 5  >>> print c  [[ 1. 1. 1. 0.]  [ 1. 1. 0. 1.]] |

通过上面可以知道，这里进行是深拷贝，而不是引用指向同一位置的浅拷贝。

#### 2.2.3.7 深度拷贝

数组对象自带了浅拷贝和深拷贝的方法，但是一般用深拷贝多一些：

|  |
| --- |
| >>> a = np.ones((2,2))  >>> b = a  >>> b is a  True  >>> c = a.copy() #深拷贝  >>> c is a  False |

#### 2.2.3.8 矩阵转置运算

|  |
| --- |
| >>> a = np.array([[1,0],[2,3]])  >>> print a  [[1 0]  [2 3]]  >>> print a.transpose()  [[1 2]  [0 3]] |

### 2.2.4 数组的形状操作

#### 2.4.1 reshape更改数组的形状

数组的形状取决于其每个轴上的元素个数：

|  |
| --- |
| >>> a= np.floor(10\*np.random.random((3,4)))  >>> a  array([[ 7., 5., 9., 3.],  [ 7., 2., 7., 8.],  [ 6., 8., 3., 2.]])  >>> a.shape  (3, 4) |

可以用多种方式修改数组的形状：

|  |
| --- |
| >>> a.ravel() # 平坦化数组  array([ 7., 5., 9., 3., 7., 2., 7., 8., 6., 8., 3., 2.])  >>> a.shape= (6, 2)  >>> a.transpose()  array([[ 7., 9., 7., 7., 6., 3.],  [ 5., 3., 2., 8., 8., 2.]]) |

由ravel()展平的数组元素的顺序通常是“C风格”的，就是以行为基准，最右边的索引变化得最快，所以元素a[0,0]之后是a[0,1]。如果数组改变成其它形状(reshape)，数组仍然是“C风格”的。NumPy通常创建一个以这个顺序保存数据的数组，所以ravel()通常不需要创建起调用数组的副本。但如果数组是通过切片其它数组或有不同寻常的选项时，就可能需要创建其副本。还可以同过一些可选参数函数让reshape()和ravel()构建FORTRAN风格的数组，即最左边的索引变化最快。

#### 2.4.2 resize更改数组形状

reshape函数改变调用数组的形状并返回该数组，而resize函数改变调用数组自身。

|  |
| --- |
| >>> a  array([[ 7., 5.],  [ 9., 3.],  [ 7., 2.],  [ 7., 8.],  [ 6., 8.],  [ 3., 2.]])  >>> a.resize((2,6))  >>> a  array([[ 7., 5., 9., 3., 7., 2.],  [ 7., 8., 6., 8., 3., 2.]])  ##如果调用reshape，则会返回一个新矩阵  >>> a.reshape((2,6))  array([[ 7., 5., 9., 3., 7., 2.],  [ 7., 8., 6., 8., 3., 2.]]) |

# 3 数据挖掘与机器学习导论

----机器学习算法最适用的场景就是：不便用规则处理的场合

## 3.1数据挖掘

简而言之，数据挖掘（Data Mining）是有组织有目的地收集数据，通过分析数据使之成为信息，从而在大量数据中寻找潜在规律以形成规则或知识的技术。

## 3.2 数据挖掘与机器学习的关系

机器学习可以用来作为数据挖掘的一种工具或手段；

数据挖掘的手段不限于机器学习，譬如还有诸如统计学等众多方法；

但机器学习的应用也远不止数据挖掘，其应用领域非常广泛，譬如**人工智能**；

## 3.2机器学习

### 3.2.1定义

机器学习(Machine Learning, ML)是一门多领域交叉学科，涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多门学科。专门研究计算机怎样模拟或实现人类的学习行为，以获取新的知识或技能，重新组织已有的知识结构使之不断改善自身的性能。

它是人工智能的核心，是使计算机具有智能的根本途径，其应用遍及人工智能的各个领域。目前，世界上共有几百种不同的机器学习算法。

### 3.2.2机器学习算法类别

#### 分类与聚类

* Classification (分类)：

给定一堆样本数据，以及这些数据所属的类别标签，通过算法来对预测新数据的类别

有先验知识

* Clustering(聚类)：

事先并不知道一堆数据可以被划分到哪些类，通过算法来发现数据之间的相似性，从而将相似的数据划入相应的类，简单地说就是把相似的东西分到一组

没有先验知识

#### 常见的分类与聚类算法

* 常用的分类算法：k-最近邻法(k-nearest neighbor，kNN)，决策树分类法，朴素贝叶斯分类算法(native Bayesian classifier)、**支持向量机(SVM)的分类器，神经网络法**，模糊分类法等等。
* 常见聚类算法： **K均值(K-means clustering)聚类算法**、K-MEDOIDS算法、CLARANS算法；BIRCH算法、CURE算法、CHAMELEON算法等；基于密度的方法：DBSCAN算法、OPTICS算法、DENCLUE算法等；基于网格的方法：STING算法、CLIQUE算法、WAVE-CLUSTER算法；

#### 监督学习与无监督学习

机器学习按照训练数据是否有“先验知识”，一般划分为三类：

1. 监督学习(supervised learning)
2. 无监督学习(unsupervised learning)
3. 半监督学习(semi-supervised learning)

* 监督式学习技术需要关于结果的先验知识

例如，如果我们正在研究一个市场活动的历史数据，我们可以根据市场是否产生预期的反应来对数据进行分类，或决定下一步要花多少钱。监督式学习技术为预测和分类提供了强大的工具。

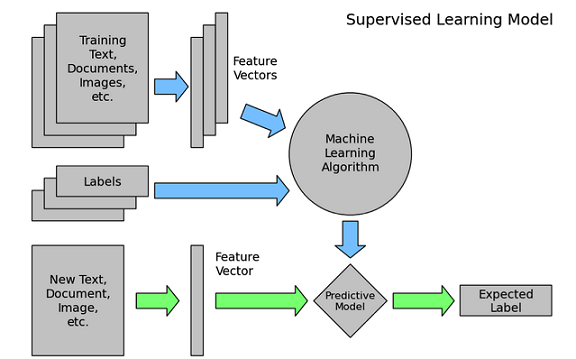
* 无监督学习技术不需要先验知识。

例如，在某些欺诈的案例中，只有当事情发生很久以后，我们才可能知道某次交易是不是欺诈。在这种情况下，与其试图预测哪些交易是欺诈，我们不如使用机器学习来识别那些可疑的交易，并做出标记，以备后续观察。我们对某种特定的结果缺乏先验知识、但仍希望从数据中汲取有用的洞察时，就要用到无监督式学习。

## 3.3 机器学习的应用步骤

1. 需求分析
2. 收集数据
3. 探索数据特性
4. **提取数据特征并建模**
5. 开发代码（常用语言：R语言，Python语言，spark mllib库）
6. 训练模型
7. 应用系统集成（比如将训练好的算法模型集成到推荐系统中）

通用机器学习算法应用工程技术架构



## 3.4 机器学习必需数学知识

在数据挖掘所用的机器学习算法中，很大一部分问题都可以归结为以下三个方面的数学知识：概率、距离、线性方程

#### 3.4.1 概率

基本概念：

概率描述的是随机事件发生的可能性

比如，抛一枚硬币，出现正反两面的概率各为50%

基本计算：

设一个黑箱中有8个黑球2个红球，现随机抽取一个球，则

取到黑球的概率为：8/(8+2) =0.8

取到红球的概率：2 /(8+2) =0.2

条件概率：

假如有两个黑箱A/B，A中有7黑球+1红球，B中有1黑球+1红球，假如随机抽取到一个球为红球，问，球来自A箱的概率——这就是条件概率问题

所求概率可表示为： p(A|红球) 即在已知结果是红球的条件下，是来自A的概率

条件概率的计算：

P(A|红球) = P(A,红球)/P(A)

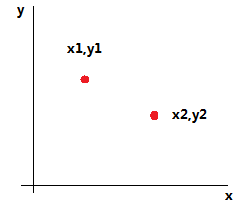
<补充：具体运算过程>

#### 3.4.2 距离（相似度）

在机器学习中，距离通常用来衡量两个样本之间的相似度，当然，在数学上，距离这个概念很丰满，有很多具体的距离度量，最直白的是“欧氏距离”，即几何上的直线距离

* 图示：

如图，在二维平面上有两个点(x1,y1) , (x2,y2)，求两点之间的距离



* 计算方法：

D12 = 

而在机器学习中，通常涉及的是多维空间中点的距离计算，计算方式一样：

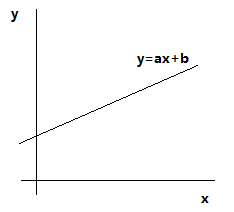
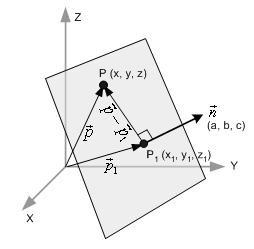
Dn = 

#### 3.4.3 线性方程

机器学习中的线性拟合或回归分类问题都需要理解线性方程

* 图示

线性方程用来描述二维空间中的直线或多维空间中的平面，比如在二维空间中，如图

y=ax+b即是图中直线的线性方程：

* x是自变量，y是因变量
* a b 是参数，决定直线的斜率和截距

如果在多维空间中，线性方程则是表示平面，方程形式如：ax+by+cz+d=0

* 计算方法

初等数学经常已知a, b求解x y，而在高等数学中，我们往往是知道大量的(y,x)样本比如(x1,y1)，(x2,y2),(x3,y3)要求反推参数列表(a,b,..)

在维度小，样本数据都“正确+精确”的情况下，可以通过线性方程求解的方式来解出a,b,....

但在机器学习中，我们拿到的大量样本数据本身都是“不精确且充满噪点”的，所以代入方程来求解a,b...显然不可行，此时，一般都是采用逼近的思想来求解：

|  |
| --- |
| 1. 设定参数的初始值——>代入样本试探——>根据试探结果调整参数——>再次代入样本试探——>再调整参数 2. 一直循环迭代直到获得一组满意的参数 |

<补充：一个运算实例>

#### 3.4.5 向量和矩阵

在以上3大数学问题中，都涉及到大量样本数据大量特征值的“批量运算”，此时，可运用数学中的工具：“向量和矩阵”

N维向量：就是一个一维的数组(x1,x2,x3,x4,.....)，数组中的元素个数即为向量的“维度数”

矩阵：将多个(比如M个) N维向量写在一起，就是矩阵（M\*N）：

x11,x12,x13,x14,.....

x21,x22,x23,x24,.....

x31,x32,x33,x34,.....

x41,x42,x43,x44,.....

矩阵和向量的意义主要在哪呢？就是为了方便快速地进行大量数据（尤其是线性方程问题）的批量运算

如：

矩阵相加



矩阵相乘

