# Python编程积累

### Python数据类型

##### 1. Tuple：元祖数据类型

小括号( )：代表tuple元组数据类型，元组是一种不可变序列。

>>> tup = (1,2,3)

>>> tup

(1, 2, 3)

Python 规定，单元素 tuple 要多加一个逗号“,”，这样就避免了歧义。

多元素 tuple 加不加这个额外的","效果是一样的。

##### 2. List：列表数据类型

中括号[ ]：代表list列表数据类型，列表是一种可变的序列。

>>> list('python')

['p', 'y', 't', 'h', 'o', 'n']

>>>[‘python’,’c++’,’java’]

[‘python’,’c++’,’java’]

##### 3. Dict：字典数据类型

大括号{ }花括号：代表dict字典数据类型，字典是由键对值组组成。冒号':'分开键和值，逗号','隔开组。

>>> dic={'jon':'boy','lili':'girl'}

>>> dic

{'lili': 'girl', 'jon': 'boy'}

>>> NETS = {'vgg16': ('VGG16', 'vgg16\_faster\_rcnn\_iter\_70000.caffemodel'),

'zf': ('ZF', 'ZF\_faster\_rcnn\_final.caffemodel')}

##### Numpy array

**（1）与list区别**

|  |
| --- |
| （1）初始化  List：  a=[[1,2],[3,4]]  Numpy Array:  b=np.array([[1,2],[3,4]]) |
| 1. 直接显示   >>>a  [[1, 2,], [3, 4]]  >>>b  array([[1,2],[3,4]]) |
| （3）打印显示  >>>print(a)  [[1, 2,], [3, 4]]  >>>print(b)  [[1, 2,] [3, 4]] |
| 1. numpy array可以进行很多数据操作   包括shape、reshape、stack、hstack、vstack等 |

**（2）切片操作**

>>>a=np.array([[1,2],[2,3],[3,4]])

>>>a[:-1]

array([[1,2],[2,3]])

### Python脚本路径相关

##### 1. 获取python脚本的当前路径

this\_dir=os.path.dirname(\_\_file\_\_)

‘\_\_file\_\_'这个参数代表的是python解释器正在执行的脚本文件；

this\_dir即为当前脚本的路径

##### 2. 添加python脚本搜索的路径

path\_dir=os.path.join(‘\*\*\*’, ’\*\*\*’, ’\*\*\*’, ....)

#将几个路径拼接在一起，形成一个新的路径

sys.path.insert(0,path\_dir)

#将路径加入python的搜索路径下，这样路径下的python脚本即可直接利用以下几种方式调用相关脚本或者函数：

1）import 脚本名

2）from 脚本名 Import 函数名

3）import 搜索路径下的文件夹名.脚本名

### python命令行参数解析

##### 1. sys.argv

适合解析简单的命令行

|  |
| --- |
| #!/usr/bin/python  # -\*- coding: utf-8 -\*-  '''  需要模块：sys  参数个数：len(sys.argv)  文件名： sys.argv[0]  参数1： sys.argv[1]  参数2： sys.argv[2]  ......  '''  import sys  print "file = ", sys.argv[0]  for i in range(1, len(sys.argv)):  print "parameter%s = %s"%(i, sys.argv[i]) |

##### 2.argparse

argparse是python用于解析命令行参数和选项的标准模块，用于代替已经过时的optparse模块。

**（1）使用步骤**

1：import argparse #首先导入模块

2：parser = argparse.ArgumentParser() #然后创建一个解析对象

3：parser.add\_argument() #然后向该对象中添加你要关注的命令行参数和选项，每一个add\_argument对应一个要关注的参数或选项

4：parser.parse\_args() #最后调用parse\_args()方法解析

**（2）创建解析器--ArgumentParser()**

class ArgumentParser(prog=None, usage=None, description=None, epilog=None, parents=[], formatter\_class=argparse.HelpFormatter, prefix\_chars='-', fromfile\_prefix\_chars=None, argument\_default=None, conflict\_handler='error', add\_help=True)

创建一个ArgumentParser实例，ArgumentParser的参数都为关键字参数。

prog ：文件名，默认为sys.argv[0]，用来在help信息中描述程序的名称。

usage ：描述程序用途的字符串

description ：help信息前显示的信息

epilog ：help信息之后显示的信息

parents ：由ArgumentParser对象组成的列表，它们的arguments选项会被包含到新ArgumentParser对象中。(类似于继承)

formatter\_class ：help信息输出的格式，为了美观

prefix\_chars ：参数前缀，默认为’-‘(最好不要修改)

fromfile\_prefix\_chars ：前缀字符，放在文件名之前

conflict\_handler ：解决冲突的策略，默认情况下冲突会发生错误

这些参数都有默认值，当调用parser.print\_help()或者运行程序时由于参数不正确(此时python解释器其实也是调用了pring\_help()方法)时，会打印这些描述信息。

一般只需要传递description参数。

**（3）添加参数选项--add\_argument()**

add\_argument(name or flags...[, action][, nargs][, const][, default][, type][, choices][, required][, help][, metavar][, dest])

|  |
| --- |
| **name or flags** ：参数有两种，可选参数和位置参数。  添加可选参数  >>> parser.add\_argument('-f', '--foo')) #-f是--foo的缩写形式，使用时用哪个都行  添加位置参数  >>> parser.add\_argument('bar')  parse\_args()运行时，默认会用’-‘来认证可选参数，剩下的即为位置参数。解析时缺少位置参数就会报错了 |
| **action： 默认为store**  **store\_const**：值存放在const中：  >>> parser = argparse.ArgumentParser()  >>> parser.add\_argument('--foo', action='store\_const', const=42)  >>> parser.parse\_args('--foo'.split())  Namespace(foo=42)  **store\_true和store\_false**：值存为True或False  >>> parser = argparse.ArgumentParser()  >>> parser.add\_argument('-x', action='store\_true')  >>> parser.add\_argument('-y', action='store\_false')  >>> parser.add\_argument('-z', action='store\_false')  >>> parser.parse\_args('-x -y'.split())  Namespace(x=True, y=False, z=True)  **append**：存为列表，可以有多个参数  >>> parser = argparse.ArgumentParser()  >>> parser.add\_argument('--l', action='append')  >>> parser.parse\_args('--l 1 --l 2'.split())  Namespace(l=['1', '2'])  **append\_const**：存为列表，会根据const关键参数进行添加：  >>> parser = argparse.ArgumentParser()  >>> parser.add\_argument('--str', dest='types', action='append\_const', const=str)  >>> parser.add\_argument('--int', dest='types', action='append\_const', const=int)  >>> parser.parse\_args('--str --int --str --int'.split())  Namespace(l=None, types=[<type 'str'>, <type 'int'>, <type 'str'>, <type 'int'>])  **count**：统计参数出现的次数  >>> parser = argparse.ArgumentParser()  >>> parser.add\_argument('--counte', '-c', action='count')  >>> parser.parse\_args('-cccc'.split())  Namespace(counte=4)  **help**： help信息  **version**：版本  >>> parser = argparse.ArgumentParser()  >>> parser.add\_argument('--version', action='version', version='version 2.0')  >>> parser.parse\_args(['--version'])  version 2.0 |
| **metaver**：帮助信息中显示的参数名称 |
| **nargs：** 参数的数量  值可以为整数N(N个)，\*(任意多个，可以为0个)，+(一个或更多)，有点像正则表达式啊  >>> parser = argparse.ArgumentParser()  >>> parser.add\_argument('-x', nargs='\*')  >>> parser.add\_argument('-y', nargs='\*')  >>> parser.add\_argument('-z', nargs='\*')  >>> parser.parse\_args('1 2 -x 3 4 -y 5 6'.split())  Namespace(x=['3', '4'], y=['5', '6'], z=['1', '2'])  值为?时，首先从命令行获得参数，如果有-y后面没加参数，则从const中取值，如果没有-y，则从default中取值  >>> parser = argparse.ArgumentParser()  >>> parser.add\_argument('x', nargs='?',default='default')  >>> parser.add\_argument('-y', nargs='?',const='const', default='default')  >>> parser.parse\_args('1 -y 2'.split())  Namespace(x='1', y='2')  >>> parser.parse\_args('1 -y'.split())  Namespace(x='1', y='const')  >>> parser.parse\_args([])  Namespace(x='default', y='default') |
| **const** ：保存一个常量 |
| **default** ：默认值 |
| **type** ：参数类型，默认为str |
| **choices** ：设置参数值的范围，如果choices中的类型不是字符串，记得指定type  >>> parser = argparse.ArgumentParser()  >>> parser.add\_argument('x', type=int, choices=range(1, 4))  >>> parser.parse\_args(['3'])  Namespace(x=3)  >>> parser.parse\_args(['4'])  usage: [-h] {1,2,3}  : error: argument x: invalid choice: 4 (choose from 1, 2, 3) |
| **required** ：该选项是否必选，默认为True |
| **dest** ：参数名  >>> parser = argparse.ArgumentParser()  >>> parser.add\_argument('--foo', dest='f\_name')  >>> parser.parse\_args('--foo XXX'.split())  Namespace(f\_name='XXX') |

1. **常用情形例子**

filename = argv\_argparse.py

|  |
| --- |
| #!/usr/bin/python  # -\*- coding: utf-8 -\*-  import sys  import argparse  def cmd():  args = argparse.ArgumentParser(description = 'Personal Information ',epilog = 'Information end ')  #必写属性,第一位  args.add\_argument("name", type = str, help = "Your name")  #必写属性,第二位  args.add\_argument("birth", type = str, help = "birthday")  #可选属性,默认为None  args.add\_argument("-r",'--race', type = str, dest = "race", help = u"民族")  #可选属性,默认为0,范围必须在0~150  args.add\_argument("-a", "--age", type = int, dest = "age", help = "Your age", default = 0, choices=range(150))  #可选属性,默认为male  args.add\_argument('-g',"--gender", type = str, dest = "gender", help = 'Your gender', default = 'male', choices=['male', 'female'])  #可选属性,默认为None,-p后可接多个参数  args.add\_argument("-p","--parent",type = str, dest = 'parent', help = "Your parent", default = "None", nargs = '\*')  #可选属性,默认为None,-o后可接多个参数  args.add\_argument("-o","--other", type = str, dest = 'other', help = "other Information",required = False,nargs = '\*')  args = args.parse\_args() #返回一个命名空间,如果想要使用变量,可用args.attr  print "argparse.args=",args,type(args)  print 'name = %s'%args.name #打印name参数  d = args.\_\_dict\_\_  for key,value in d.iteritems():  print '%s = %s'%(key,value)  if \_\_name\_\_=="\_\_main\_\_":  cmd() |
| python argv\_argparse.py xiaoming 1991.11.11  或  python argv\_argparse.py xiaoming 1991.11.11 -p xiaohong xiaohei -a 25 -r han -g female -o 1 2 3 4 5 6 |

### Python 内置函数

##### 1. enumerate()函数

enumerate() 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 for 循环当中。

语法:

enumerate(sequence, [start=0])

参数

sequence -- 一个序列、迭代器或其他支持迭代对象。

start -- 下标起始位置。

返回值

返回 enumerate(枚举) 对象。

|  |
| --- |
| >>>seasons = ['Spring', 'Summer', 'Fall', 'Winter']  >>> list(enumerate(seasons))  [(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]  >>> list(enumerate(seasons, start=1)) # 小标从 1 开始  [(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')] |
| 普通for循环  i = 0  seq = ['one', 'two', 'three']  for element in seq:  print i, seq[i]  i +=1  使用enumerate  seq = ['one', 'two', 'three']  for i, element in enumerate(seq):  print i, seq[i] |

### Numpy内置函数

##### 1. stack()，hstack()，vstack()数组堆叠函数

|  |
| --- |
| **stack()函数**  函数原型为：stack(arrays, axis=0)，arrays可以传数组和列表。  （需要再学） |
| **hstack()函数**  函数原型：hstack(tup) ，参数tup可以是元组，列表，或者numpy数组，返回结果为numpy的数组。  作用：将多个数组的行的内容依次拼接起来，列数不变。其实就是水平把数组给堆叠起来，vstack()函数正好和它相反。  import numpy as np  a=[[1],[2],[3]]  b=[[1],[2],[3]]  c=[[1],[2],[3]]  d=[[1],[2],[3]]  print(np.hstack((a,b,c,d)))  输出：  [[1 1 1 1]  [2 2 2 2]  [3 3 3 3]] |
| **vstack()函数**  函数原型：vstack(tup) ，参数tup可以是元组，列表，或者numpy数组，返回结果为numpy的数组。  作用：将多个数组的列的内容依次拼接起来，列数不变。其实就是垂直把数组给堆叠起来。  import numpy as np  a=[1,2,3]  b=[4,5,6]  print(np.vstack((a,b)))  输出：  [[1 2 3]  [4 5 6]] |

##### 2. where()函数

**（1）用法1**

numpy.where(condition[, x, y])

condition : array\_like, bool

x, y : array\_like, optional，x 和y 与condition尺寸相同

if conditon == True: 取当前位置的x的值

else： 取当前位置的y的值

返回值：

返回一个array或包含多个array的tuple。

当条件对象为一维，返回（array, )。

条件对象为二维，返回(array1,array2)。**第一部分为矩阵行的坐标，第二部分为矩阵列的坐标。**

根据定义条件返回元素，这些元素或者从x中获得，或者从y中获得。

|  |
| --- |
| >>> np.where([[True, False], [True, True]],  ... [[1, 2], [3, 4]],  ... [[9, 8], [7, 6]])  array([[1, 8],[3, 4]]) # True时从x取值，False时从y取值 |

1. **用法2**

numpy.where(condition)

返回值：

返回条件中非零（True）元素的坐标。

|  |
| --- |
| >>> x=np.arange(16)  >>> np.where(x>5)  (array([ 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]),)  >>>x = np.arange(16).reshape(-1,4)  >>>np.where(x>5)  (array([1, 1, 2, 2, 2, 2, 3, 3, 3, 3]), array([2, 3, 0, 1, 2, 3, 0, 1, 2, 3])) #注意这里是坐标，前面的一维的坐标，后面是二维的坐标  >>>ix = np.array([[False, False, False],  [ True, True, False],  [False, True, False]], dtype=bool)  >>>np.where(ix)  (array([1, 1, 2]), array([0, 1, 1])) |

### Python常用功能

##### 1. 统计时间

|  |
| --- |
| import time |