## 一、基本的python使用概念

### ****1.1 Python脚本的调用方式****

在linux中，python、shell这些程序都是普通的文本格式，都需要一种程序去解释执行它。要么调用的时候指定，要么在文件头指定。

1. 直接使用python xxxx.py执行。其中python可以写成python的绝对路径。使用which python进行查询。

2、在文件的头部（第一行）写上**#!/usr/bin/python，**这个地方使用python的绝对路径，就是上面用which python查询来的结果。然后在外面就可以使用./xxx.py执行了。

### ****1.2 源程序编码方式****

默认情况下, Python 源码文件以UTF-8 编码. 在这种编码下, 世界上大多数语言的字符都可以用于, 字符串常量, 标识符, 以及注释——尽管标准库遵循一个所有可移植代码都应遵守的约定: 仅使用ASCII 字符作为标识符. 要正确地显示所有这些字符, 你的编辑器一定要有能力辨认出是UTF-8 编码, 还要使用一个支持所有文件中字符的字体.

也可以为源码文件指定不同的编码. 为此, 要在#! 行后面指定一个特

殊的注释行, 以定义源码文件的编码:

# -\*- coding: encoding -\*-

有了这样的声明，源文件中的所有字符都会被以encoding 的编码来解读，

而非是UTF-8.

## 二、数据结构

### 2.1 字符串相关

（1）定义

字符串是用单引号或双引号包裹起来的。

字符串有几种方法来跨越多行：

第一种，利用单引号或双引号，在一行最后加上一个反斜杠以表明下一行是这行的逻辑延续:

另一种方法, 字符串可以使用一对匹配的三引号对包围: """ 或'''. 当使用三引号时, 回车不需要被舍弃, 他们会包含在字符串里.

（2）操作

|  |
| --- |
| word = 'Help ' + 'A'  'str' 'ing' #该操作只支持常量  '<' + word\*5 + '>'  word[-1] # 最后一个字符  word[-2] # 倒数第二个字符  word[-2:] # 最后两个字符  word[:-2] #除了最后两个字符以外的字符  len(word) #表示字符串长度 |
|  |

### 2.2 列表相关

#### 2.2.1 简介

Python 有一些**复合数据类型**, 用来把其它值分组. 最全能的就是list,它可以写为在方括号中的通过逗号分隔的一列值(项). 列表的项并不需要是同一类型。

就像字符串索引, 列表的索引从0 开始, 列表也可以切片, 连接等等。

所有的切片操作返回一个包含请求元素的新列表。

|  |
| --- |
| a = ['spam ', 'eggs', 100 , 1234]  a[1:-1] #第二个到倒数第二个  3\*a[:3] + ['Boo!']  a[:] #返回a的浅复制  a[2] = a[2] + 23 #列表中的单个元素是可变的，包括列表的长度、内容、值等都是可以变的  a[0:2] = [1, 12] #替代一些项  a[0:2] = [] #移除一些项  a[1:1] = ['bletch ', 'xyzzy '] #插入一些项  a[:] = [] #清空列表  len(a) # len也有效  p = [1, [1,2], 4] #可以嵌套列表  p[1].append('xtra') #元素尾部加入一些内容 |

#### 2.2.2 常见用法

|  |
| --- |
| list.**append(x)**  在列表的尾部添加一个项; 等价于a[len(a):]=[x]  list.**extend(L)**  用给入的列表将当前列表接长; 等价于a[len(a):]=L  list.**insert(i, x)**  在给定的位置上插入项. 第一个参数就是准备在它之前插入的元素的索引, 因此a.insert(0,x) 会在列表的头部插入, 而a.insert(len(a),x) 则等价于a.append(x)  list.**remove(x)**  移除列表中第一个值为x 的项. 没有符合要求的项时, 会产生一个错误.  list.**pop([i])**  删除列表给定位置的项, 并返回它. 如果没有指定索引, a.pop 移除并返回列表的最后一项. (函式原型的i 在中方括号中意味着它是一个可选参数, 而不是你应当在那里键入一个方括号. 你将会在Python 库参考中经常见到这种表示法.)  list.**index(x)**  返回列表中第一个值为x 的项索引值. 如果没有匹配的项, 则产生一个错误  list.**count(x)**  返回列表中x 出现的次数  list.**sort()**  就地完成列表排序  list.**reverse()**  就地完成列表项的翻转 |

**（1）把列表当作堆栈用**

堆栈的特性是最后添加的元素就是第一个取出的元素(即“后入先出”). 要在栈顶添加一个项, 就使用**append()**. 要从栈顶取回一个项, 就使用不带显式索引的**pop()**

**（2）把列表或者collection.deque当作队列用**

从列表的尾部添加和弹出是很快的, 而在列表的开头插入或弹出是慢的(因为所有元素都得移动一个位置).

要实现一个队列, 使用collection.deque, 它被设计成在两端添加和弹出都很快

|  |
| --- |
| **from** collections **import** deque  >>> queue = deque(["Eric", "John", "Michael"])  >>> queue.append("Terry") # Terry 进入  >>> queue.append("Graham") # Graham 进入  >>> queue.popleft() # 第一个进入的现在离开  'Eric'  >>> queue.popleft() # 第二个进入的现在离开  'John' |

（3）列表推导式

列表推导式的结构是, 在一个方括号里, 首先是一个表达式, 随后是一个for 子句, 然后是零个或更多的for 或if 子句. 结果将是通过计算for和if 子句来获得的一个列表.

如果要使表达式推导式出元组, 就必须用圆括号.

|  |
| --- |
| >>> vec = [2, 4, 6]  >>> [[x, x\*\*2] **for** x **in** vec]  [[2, 4], [4, 16], [6, 36]]  >>> [3\*x **for** x **in** vec **if** x > 3]  [12 , 18]  >>> [(x, x\*\*2) **for** x **in** vec]  [(2, 4), (4, 16), (6, 36)]  >>> vec1 = [2, 4, 6]  >>> vec2 = [4, 3, -9]  >>> [x\*y **for** x **in** vec1 **for** y **in** vec2]  [8, 6, -18 , 16 , 12 , -36 , 24 , 18 , -54]  >>> [x+y **for** x **in** vec1 **for** y **in** vec2]  [6, 5, -7, 8, 7, -5, 10 , 9, -3]  >>> [vec1[i]\*vec2[i] **for** i **in** range(len(vec1))]  [8, 12 , -54]  >>> [str(round(355/113 , i)) **for** i **in** range(1, 6)]  ['3.1', '3.14', '3.142', '3.1416', '3.14159 '] |

**（4）del语句**

通过给定索引而不是值, 来删除列表中项的方法: 用**del** 语句. 它与返回一个值的**pop()** 方法不同. **del** 语句也可以移除列表中的切片, 或者清除整个列表(之前我们通过给切片赋值为空列表来完成这点)。

|  |
| --- |
| >>> **del** a[0]  >>> **del** a[2:4]  >>> **del** a[:]  >>> **del** a #删除变量实体，后面引用a就会出错 |

### 2.3元组和序列

另一种标准序列数据类型: 元组。元组由若干逗号分隔的值组成, 一般需要用圆括号包含，例如:

|  |
| --- |
| >>> t = 12345 , 54321 , 'hello!'  >>> t[0]  12345  >>> t  (12345 , 54321 , 'hello!')  >>> u = t, (1, 2, 3, 4, 5)  >>> u  ((12345 , 54321 , 'hello!'), (1, 2, 3, 4, 5)) |

元组有许多用途. 例如: (x, y) 坐标对, 数据库里的员工记录等. 元组同字串都是不可变的: 无法对元组指定项进行赋值(尽管可通过切片和连接来模拟这个操作). 元组中可以包含可变的对象, 如列表。

构造包含0 或1 个项的元组是个特殊问题: 语法上为了适应这一情况,有些额外的规则.

**空元组由一对空的圆括号构造;**

**一个项的元组由一个值后面跟着一个逗号构造**(把一个值放入一对圆括号里并不足以构造一个元组)。

**（1）元组打包和序列解包**

|  |
| --- |
| >>> t = 12345,54321,'hello!' #元组打包  >>> x, y, z = t #序列解包 |

### 2.4 集合（set）

集合是种无序不重复的元素集.基本用途包括成员关系测试和重复条目消除. 集合对象也支持合(union),交(intersection), 差(difference), 和对称差(sysmmetric difference) 等数学操作。

花括号或函式set() 可用于创建集合. 注意: 创建一个空集合只能使用set(), 而不能使用{}; 后者是创建一个空字典。

|  |
| --- |
| >>> basket = {'apple ', 'orange ', 'apple ', 'pear', 'orange ', 'banana '}  >>> **print**(basket) # 重复的被移除了  {'orange ', 'banana ', 'pear', 'apple '}  >>> 'orange ' **in** basket # 快速成员关系测试  True  >>> 'crabgrass ' **in** basket  False  >>> a = set('abracadabra ')  >>> b = set('alacazam ')  >>> a # a 中的不重复字母  {'a', 'r', 'b', 'c', 'd'}  >>> a - b # a 中有而b中没有的字母  {'r', 'd', 'b'}  >>> a | b # 在a 中或在b中的字母  {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}  >>> a & b # a 和b 中都有的字母  {'a', 'c'}  >>> a ^ b # a 或b中只有一个有的字母  {'r', 'd', 'b', 'm', 'z', 'l'}  >>> a = {x **for** x **in** 'abracadabra ' **if** x **not in** 'abc'}  >>> a  {'r', 'd'} |

### 2.5 字典

在其它语言中字典一般被叫做“关联存储”或“关联数组”. 与使用某个范围作为索引的序列不一样, **字典通过键来索引, 而键可以是任意不可变类型;**

通常用**字符串和数字作为键.** 如果**元组只包含字符串和数字, 元组也可以作为键**; 但是, 当元组直接或间接地包含可变对象时, 就不能用作一个键. **不能使用列表作为键,** 因为列表可以通过索引, 切片原地赋值而被改变。

**字典**看成是一个**没有顺序的键*:* 值对集合, 键必须是唯一的**(在一个字典里). 一对花括号创建一个空字典: {}. 在括号中间放置的以逗号分隔的键:值对列表就是字典的初始键:值对. 这也是字典输出时的格式.

**（1）存储、取值与删除**

字典最主要的操作是通过某键存储一个值, 以及从给定的键里提取它的值.

如果你使用一个已被使用的键进行存储操作, 该键的旧值就没有了.

使用一个不存在的键提取值会产生一个错误.

使用**del** 可以删除一个键: 值对.

**（2）查询字典中的键值**

在一个字典上执行list(d.keys()) 返回该字典中所使用键的列表, 该列表的顺序不确定(如果需要有序, 只要使用sorted(d.keys()))。

**（3）检查键是否存在**

要检查某一个键是否在字典里, 使用**in** 关键字.

|  |
| --- |
| >>> tel = {'jack ': 4098 , 'sape': 4139}  >>> tel['guido '] = 4127  >>> **del** tel['sape']  >>> tel['irv'] = 4127  >>> list(tel.keys())  ['irv', 'guido ', 'jack']  >>> sorted(tel.keys())  ['guido ', 'irv', 'jack']  >>> 'guido ' **in** tel  True |

**（4）构造器dict()**

|  |
| --- |
| >>> dict([('sape ', 4139), ('guido ', 4127), ('jack', 4098)])  {'sape': 4139 , 'jack': 4098 , 'guido ': 4127}  >>> dict([(x, x\*\*2) **for** x **in** (2, 4, 6)]) # 使用列表推导式  {2: 4, 4: 16 , 6: 36}  若键为字符串, 有时用关键字参数指定键值对更为简单:  >>> dict(sape=4139 , guido=4127 , jack=4098)  {'sape': 4139 , 'jack': 4098 , 'guido ': 4127} |

### 2.6 遍历技巧

#### 2.6.1 enumerate()

对序列遍历时, 可以使用**enumerate()** 函式来同时取回位置索引和相应的值.

|  |
| --- |
| >>> **for** i, v **in** enumerate(['tic', 'tac', 'toe']):  ... **print**(i, v)  ...  0 tic  1 tac  2 toe |

#### 2.6.2 zip()

同时对两个或更多的序列进行遍历时, 可用**zip()** 进行组合.

|  |
| --- |
| >>> questions = ['name', 'quest ', 'favorite color ']  >>> answers = ['lancelot ', 'the holy grail ', 'blue']  >>> **for** q, a **in** zip(questions , answers):  ... **print**('What is your {0}? It is {1}.'.format(q, a))  ...  What **is** your name? It **is** lancelot.  What **is** your quest? It **is** the holy grail.  What **is** your favorite color? It **is** blue. |

#### 2.6.3 reversed()、sorted()

反向遍历序列和有序遍历序列

|  |
| --- |
| >>> **for** i **in** reversed(range(1, 10 , 2)):  ... **print**(i)  >>> basket = ['apple ', 'orange ', 'apple ', 'pear', 'orange ', 'banana ']  >>> **for** f **in** sorted(set(basket)):  ... **print**(f) |

#### 2.6.4 items()

当对字典遍历时, 可用**items()** 方法同时取回键和对应的值。

|  |
| --- |
| >>> knights = {'gallahad ': 'the␣pure', 'robin ': 'the␣brave '}  >>> **for** k, v **in** knights.items():  ... **print**(k, v)  gallahad the pure  robin the brave |

### 2.7 条件控制

在**while** 和**if** 语句中使用的条件可以包含任何操作符。

#### 2.7.1 比较操作符

• 比较操作符**in** 和**not in** 检查一个值是否在序列中.

• 操作符**is** 和**is not** 比较两个对象是否为同一对象; 这只对诸如列表的可变对象有用.

所有比较操作符具有相同的优先级, 低于所有的数值操作.比较操作符可以连起来使用. 例如, a < b == c 测试a 小于b 且b 与c相等.

#### 2.7.2 逻辑操作符

• 逻辑操作符的优先级又低于比较操作符;

• 这其中, **not** 优先级最高, 而**or** 的优先级最低, 因此A **and not** B **or** C 等价于(A **and** (**not** B)) **or** C . 同样, 可以使用圆括号来表达想要的结果.

• 逻辑操作符**and** 和**or** 被称为短路操作符: 它从左至右计算参数, 并且当结果确定时计算就立即停止.

比较操作(或其它任何布尔表达式) 都能用逻辑操作符**and** 和**or** 连接,结果值可以用**not** 取反。

|  |
| --- |
| >>> string1 , string2 , string3 = '', 'Trondheim ', 'Hammer Dance '  >>> non\_null = string1 **or** string2 **or** string3  >>> non\_null  'Trondheim ' |

## 三、python函数

### 3.1 流程控制

（1）If控制

if …:

elif …:

else :

（2）For控制

for x in a[:]: # 制造整个列表的切片复本

if len(x) > 6: a.insert(0, x)

（3）break 和continue 语句, 以及循环中的else 子句

break 语句工作得如同C 语言一样, 跳出最小的for 或while 循环.

循环语句可以有一个else 子句; 该子句会在以下情况被执行: 循环因迭代到列表末尾而终止(for 语句), 或者, 当循环条件为假(while 语句), 同时它不会在循环因break 语句终止的情况下被执行. 下面搜索素数的示例说明了这一特性:

|  |
| --- |
| >>> for n in range(2, 10):  ... for x in range(2, n):  ... if n % x == 0:  ... print(n, 'equals ', x, '\*', n//x)  ... break  ... else:  ... # loop fell through without finding a factor  ... print(n, 'is a prime number ')  ...  2 is a prime number  3 is a prime number  4 equals 2 \* 2  5 is a prime number  6 equals 2 \* 3  7 is a prime number  8 equals 2 \* 4  9 equals 3 \* 3  (是的, 这是正确的代码. 仔细看: else 子句属于for 循环, 而非是if 语句) |

与循环搭配使用时, else子句的行为和它与try语句的搭配时相对于它与if语句的搭配时有更多共性: try语句的else子句在没有异常发生时被执行, 循环的else子句在没有break语句是被执行.

continue语句同样是从C语言借用的, 它终止当前迭代而进行循环的下一次迭代.

(4) range函数和len函数

|  |
| --- |
| range(5, 10)  range(0, 10 , 3)  range(-10 , -100 , -30)  a = ['Mary ', 'had', 'a', 'little ', 'lamb']  for i in range(len(  print(i, a[i]) |

（5）pass语句

pass 语句什么都不做. 当语法上需要一个语句, 但程序不要动作时, 就可以使用它. 例如:

while True:

pass # 忙等待键盘中断(Ctrl+C)

### 3.2 函数定义

#### 3.2.1 def

关键字def 引入了一个函式定义. 后面必须跟上函式名和在圆括号里的参数序列. 函式体从一行开始, 并且一定要缩进。

函式体的第一个语句可以是字串; 这个字串就是函式的文档字符串, 或称为docstring. (可以在文档字串一节找到更多信息) 有很多能将文档字串自动转换为在线或可打印文档的工具, 或让用户在代码中交互地浏览它的工具; 在代码里加上文档字符串是一个好的实践, 因此, 请养成这个习惯。

执行函式，会引入新的符号表(symbol table) 用于该函式的局部变量.更精确地说, 所有在函式中被赋值的变量和值都将存储在局部符号表中。

在函式中的尽管全局变量可以引用, 但是不可直接赋值(除非用global 语句进行声明).

函式的实参在它被调用时被引入到这个函式的局部变量表；因此，参数是按值传递的(值总是对象的一个引用，而不是对象本身的值)。

当一个函式调用另一个时, 对应这次调用, 一个新的局部符号表就会被创建.

#### 3.2.2 默认参数

例如：

def ask\_ok(prompt , retries=4, complaint='Yes or no ,please!'):

**重要警告: 默认参数的值只会被求一次值. 这使得当默认参数的值是可变对象时会有所不同，如列表, 字典, 或大多类的对象时。**

例如, 下面的函式在随后的调用中会累积参数值:

|  |
| --- |
| def f(a, L=[]):  L.append(a)  return L  print(f(1))  print(f(2))  print(f(3))  如上方式调用，L的内容会不断增加。 |
| 若要默认参数不会受影响，解决如下：  def f(a, L=None):  if L is None:  L = []  L.append(a)  return L |

#### 3.2.3 关键字参数

函式也可以通过keyword= value 形式的关键字参数来调用。

|  |
| --- |
| def parrot(voltage , state='a stiff ', action='voom', type='Norwegian Blue'):  parrot(1000)  parrot(action = 'VOOOOOM ', voltage = 1000000)  parrot('a thousand ', state = 'pushing up the daisies ') #此时voltage=’a thousand’  parrot('a million ', 'bereft of life', 'jump') #此时，type采用默认值  在函式调用时, 关键字参数必须跟在位置参数之后. 所有的关键字参数都必须与函式接受的形式参数匹配 |

\*name和\*\*name参数形式：

当最后一个形式参数的形式为\*\*name 时, 则除去其他的形参的值，它将以字典(参阅映射类型——字典) 的形式包含所有剩余关键字参数。 这种调用可以与具有\*name 形式的形式参数联合使用, 这种形式参数接受所有超出函式接受范围的位置参数.(\*name 必须在\*\*name之前使用)。

例如, 如果我们像这样定义一个函式:

|  |
| --- |
| **def** cheeseshop(kind , \*arguments , \*\*keywords):  **print**(kind )  **for** arg **in** arguments:  **print**(arg)  keys = sorted(keywords.keys())  **for** kw **in** keys:  **print**(kw , ":", keywords[kw])  cheeseshop(10, 1,2,3,shopkeeper="Michael␣Palin",client="John␣Cleese",sketch="Cheese␣Shop␣Sketch")  如上调用，kind=10，\*arguments代表1，2，3，\*\*keywords代表后面三个关键字参数  注意, 关键字参数名的列表是通过之前对字典keys() 进行排序操作而创建的; 如果不这样做, 参数打印的顺序是不确定的. |

#### 3.2.4 任意参数表

指定函式能够在调用时接受任意数量的参数. 这些参数会被包装进一个元组(参看元组和序列). 在变长参数之前, 可以使用任意多个正常参数:

**def** write\_multiple\_items(file , separator , \*args):

一般地, 这种variadic 参数必须在形参列表的末尾, 因为它们将接收传递给函式的所有剩余输入参数. 任何出现在\*arg 之后的形式参数只能是关键字参数, 这意味着它们只能使用关键字参数的方式接收传值, 而不能使用位置参数.

**def** concat(\*args , sep="/"):

#### 3.2.5 释放参数列表

当**参数存在于一个既存的列表、元组甚至字典之中**, 但却需要解包以若干位置参数的形式被函数调用。

**可以利用\*释放列表或元组中的桉树；**

**利用\*\*释放字典中的参数**

|  |
| --- |
| args = [3, 6]  list(range(\*args)) # 通过解包列表参数调用  [3, 4, 5]  **def** parrot(voltage , state='a stiff ', action='voom'):  print(voltage)  print(state+action)  d = {"voltage": "four million", "state": "bleed in demised", "action": "VOOM"}  parrot(\*\*d) |

#### 3.2.6 文档字符串

这里介绍一些文档字串有关内容和格式的约定.

第一行总应当是对该对象的目的进行简述. 这行应当以一个大写字母开始, 并以句号结束。

如果这个文档字符串不只一行, 那么第二行应当为空, 以能从视觉上分隔概述和其它部分. 接下来的行应当为一个或更多段来描述该对象的调用条件, 它的边界效应等等.

|  |
| --- |
| **def** my\_function():  """Do nothing , but document it.  No , really , it doesn 't do anything.  """  **pass**  **print**(my\_function.\_\_doc\_\_) |

## 四、模块

附录：

print()函数：

关键词end可以用来避免输出后的回车, 或者以一个不同的字符串结束输出:

Print(‘hello ‘, end=’!’)