2017年1月11日

**学 士 学 位 论 文**

作者:[杨迪]

[python自动化运维学习探索]

[重庆邮电大学]

指导教师：[杨迪]

专业名称：[软件工程]



**错误！未找到目录项。**

**第一部分：基础语法知识**

1.1基础语法

**编码**

默认情况下，Python 3源码文件以 UTF-8 编码，所有字符串都是 unicode 字符串。 当然你也可以为源码文件指定不同的编码：  
# -\*- coding: UTF-8 -\*-  
**python保留字**

保留字即关键字，我们不能把它们用作任何标识符名称。Python的标准库提供了一个keyword module，可以输出当前版本的所有关键字：

>>> import keyword

>>> keyword.kwlist

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

**注释**

Python中单行注释以#开头，多行注释用三个单引号（'''）或者三个双引号（"""）将注释括起来。  
**行与缩进**

python最具特色的就是使用缩进来表示代码块。缩进的空格数是可变的，但是同一个代码块的语句必须包含相同的缩进空格数。  
**数据类型**

python中数有四种类型：整数、长整数、浮点数和复数。

整数， 如 1

长整数 是比较大的整数

浮点数 如 1.23、3E-2

复数 如 1 + 2j、 1.1 + 2.2j  
**字符串**

python中单引号和双引号使用完全相同。

使用三引号('''或""")可以指定一个多行字符串。

转义符 '\'

自然字符串， 通过在字符串前加r或R。 如 r"this is a line with \n" 则\n会显示，并不是换行。

python允许处理unicode字符串，加前缀u或U， 如 u"this is an unicode string"。

字符串是不可变的。

按字面意义级联字符串，如"this " "is " "string"会被自动转换为this is string。  
**Python 3中有六个标准的数据类型：**

Numbers（数字）

String（字符串）

List（列表）

Tuple（元组）

Sets（集合）

Dictionaries（字典）

**Numbers（数字）**

Python 3支持int、float、bool、complex（复数）。

数值类型的赋值和计算都是很直观的，就像大多数语言一样。内置的type()函数可以用来查询变量所指的对象类型。

>>> a, b, c, d = 20, 5.5, True, 4+3j

>>> print(type(a), type(b), type(c), type(d))

数值运算：

>>> 5 + 4 # 加法

9

>>> 4.3 - 2 # 减法

2.3

>>> 3 \* 7 # 乘法

21

>>> 2 / 4 # 除法，得到一个浮点数

0.5

>>> 2 // 4 # 除法，得到一个整数

0

>>> 17 % 3 # 取余

2

>>> 2 \*\* 5 # 乘方

32

注意：

1、Python可以同时为多个变量赋值，如a, b = 1, 2。

2、一个变量可以通过赋值指向不同类型的对象。

3、数值的除法（/）总是返回一个浮点数，要获取整数使用//操作符。

4、在混合计算时，Pyhton会把整型转换成为浮点数。  
**String（字符串）**

Python中的字符串str用单引号(' ')或双引号(" ")括起来，同时使用反斜杠(\)转义特殊字符。

s='he doesn \t'

print (s,type(s),len(s))

》》he doesn <class 'str'> 10  
如果你不想让反斜杠发生转义，可以在字符串前面添加一个r，表示原始字符串：

>>> print('C:\some\name')

C:\some

ame

>>> print(r'C:\some\name')

C:\some\name  
另外，反斜杠可以作为续行符，表示下一行是上一行的延续。还可以使用"""..."""或者'''...'''跨越多行。

字符串可以使用 + 运算符串连接在一起，或者用 \* 运算符重复：

>>> print('str'+'ing', 'my'\*3)

string mymymy  
Python中的字符串有两种索引方式，第一种是从左往右，从0开始依次增加；第二种是从右往左，从-1开始依次减少。

注意，没有单独的字符类型，一个字符就是长度为1的字符串。

>>> word = 'Python'

>>> print(word[0], word[5])

P n

>>> print(word[-1], word[-6])

n P  
还可以对字符串进行切片，获取一段子串。用冒号分隔两个索引，形式为变量[头下标:尾下标]。

截取的范围是前闭后开的，并且两个索引都可以省略：

>>> word = 'ilovepython'

>>> word[1:5]

'love'

>>> word[:]

'ilovepython'

>>> word[5:]

'python'

>>> word[-10:-6]

'love'  
与C字符串不同的是，Python字符串不能被改变。向一个索引位置赋值，比如word[0] = 'm'会导致错误。

注意：

1、反斜杠可以用来转义，使用r可以让反斜杠不发生转义。

2、字符串可以用+运算符连接在一起，用\*运算符重复。

3、Python中的字符串有两种索引方式，从左往右以0开始，从右往左以-1开始。

4、Python中的字符串不能改变。

**List（列表）**

List（列表） 是 Python 中使用最频繁的数据类型。

列表是写在方括号之间、用逗号分隔开的元素列表。列表中元素的类型可以不相同：  
>>> a = ['him', 25, 100, 'her']

>>> print(a)

['him', 25, 100, 'her']  
和字符串一样，列表同样可以被索引和切片，列表被切片后返回一个包含所需元素的新列表。详细的在这里就不赘述了。

列表还支持串联操作，使用+操作符：  
>>> a = [1, 2, 3, 4, 5]

>>> a + [6, 7, 8]

[1, 2, 3, 4, 5, 6, 7, 8]  
与Python字符串不一样的是，列表中的元素是可以改变的：  
>>> a = [1, 2, 3, 4, 5, 6]

>>> a[0] = 9

>>> a[2:5] = [13, 14, 15]

>>> a

[9, 2, 13, 14, 15, 6]

>>> a[2:5] = [] # 删除

>>> a

[9, 2, 6]  
List内置了有很多方法，例如append()、pop()等等，这在后面会讲到。

注意：

1、List写在方括号之间，元素用逗号隔开。

2、和字符串一样，list可以被索引和切片。

3、List可以使用+操作符进行拼接。

4、List中的元素是可以改变的。  
**Tuple（元组）**

元组（tuple）与列表类似，不同之处在于元组的元素不能修改。元组写在小括号里，元素之间用逗号隔开。

元组中的元素类型也可以不相同：  
>>> a = (1991, 2014, 'physics', 'math')

>>> print(a, type(a), len(a))

(1991, 2014, 'physics', 'math') 4  
元组与字符串类似，可以被索引且下标索引从0开始，也可以进行截取/切片（看上面，这里不再赘述）。

其实，可以把字符串看作一种特殊的元组。  
>>> tup = (1, 2, 3, 4, 5, 6)

>>> print(tup[0], tup[1:5])

1 (2, 3, 4, 5)

>>> tup[0] = 11 # 修改元组元素的操作是非法的  
虽然tuple的元素不可改变，但它可以包含可变的对象，比如list列表。

构造包含0个或1个元素的tuple是个特殊的问题，所以有一些额外的语法规则：  
tup1 = () # 空元组

tup2 = (20,) # 一个元素，需要在元素后添加逗号  
另外，元组也支持用+操作符：

>>> tup1, tup2 = (1, 2, 3), (4, 5, 6)

>>> print(tup1+tup2)

(1, 2, 3, 4, 5, 6)  
string、list和tuple都属于sequence（序列）。

注意：

1、与字符串一样，元组的元素不能修改。

2、元组也可以被索引和切片，方法一样。

3、注意构造包含0或1个元素的元组的特殊语法规则。

4、元组也可以使用+操作符进行拼接。  
**Sets（集合）**

集合（set）是一个无序不重复元素的集。

基本功能是进行成员关系测试和消除重复元素。

可以使用大括号 或者 set()函数创建set集合，注意：创建一个空集合必须用 set() 而不是 { }，因为{ }是用来创建一个空字典。  
>>> student = {'Tom', 'Jim', 'Mary', 'Tom', 'Jack', 'Rose'}

>>> print(student) # 重复的元素被自动去掉

{'Jim', 'Jack', 'Mary', 'Tom', 'Rose'}

>>> 'Rose' in student # membership testing（成员测试）

True

>>> # set可以进行集合运算

...

>>> a = set('abracadabra')

>>> b = set('alacazam')

>>> a

{'a', 'b', 'c', 'd', 'r'}

>>> a - b # a和b的差集

{'b', 'd', 'r'}

>>> a | b # a和b的并集

{'l', 'm', 'a', 'b', 'c', 'd', 'z', 'r'}

>>> a & b # a和b的交集

{'a', 'c'}

>>> a ^ b # a和b中不同时存在的元素

{'l', 'm', 'b', 'd', 'z', 'r'}  
**Dictionaries（字典）**

字典（dictionary）是Python中另一个非常有用的内置数据类型。

字典是一种映射类型（mapping type），它是一个无序的键 : 值对集合。

关键字必须使用不可变类型，也就是说list和包含可变类型的tuple不能做关键字。

在同一个字典中，关键字还必须互不相同。  
>>> dic = {} # 创建空字典

>>> tel = {'Jack':1557, 'Tom':1320, 'Rose':1886}

>>> tel

{'Tom': 1320, 'Jack': 1557, 'Rose': 1886}

>>> tel['Jack'] # 主要的操作：通过key查询

1557

>>> del tel['Rose'] # 删除一个键值对

>>> tel['Mary'] = 4127 # 添加一个键值对

>>> tel

{'Tom': 1320, 'Jack': 1557, 'Mary': 4127}

>>> list(tel.keys()) # 返回所有key组成的list

['Tom', 'Jack', 'Mary']

>>> sorted(tel.keys()) # 按key排序

['Jack', 'Mary', 'Tom']

>>> 'Tom' in tel # 成员测试

True

>>> 'Mary' not in tel # 成员测试

False  
构造函数 dict() 直接从键值对sequence中构建字典，当然也可以进行推导，如下：  
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

{'jack': 4098, 'sape': 4139, 'guido': 4127}

>>> {x: x\*\*2 for x in (2, 4, 6)}

{2: 4, 4: 16, 6: 36}

>>> dict(sape=4139, guido=4127, jack=4098)

{'jack': 4098, 'sape': 4139, 'guido': 4127}  
另外，字典类型也有一些内置的函数，例如clear()、keys()、values()等。

注意：

1、字典是一种映射类型，它的元素是键值对。

2、字典的关键字必须为不可变类型，且不能重复。

3、创建空字典使用{ }。  
**Python算术运算符**

以下假设变量a为10，变量b为21：

运算符 描述 实例

+ 加 - 两个对象相加 a + b 输出结果 31

- 减 - 得到负数或是一个数减去另一个数 a - b 输出结果 -11

\* 乘 - 两个数相乘或是返回一个被重复若干次的字符串 a \* b 输出结果 210

/ 除 - x 除以 y b / a 输出结果 2.1

% 取模 - 返回除法的余数 b % a 输出结果 1

\*\* 幂 - 返回x的y次幂 a\*\*b 为10的21次方

// 取整除 - 返回商的整数部分 9//2 输出结果 4 , 9.0//2.0 输出结果 4.0  
**Python比较运算符**

以下假设变量a为10，变量b为20：

运算符 描述 实例

== 等于 - 比较对象是否相等 (a == b) 返回 False。

!= 不等于 - 比较两个对象是否不相等 (a != b) 返回 True.

> 大于 - 返回x是否大于y (a > b) 返回 False。

< 小于 - 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。 (a < b) 返回 True。

>= 大于等于 - 返回x是否大于等于y。 (a >= b) 返回 False。

<= 小于等于 - 返回x是否小于等于y。 (a <= b) 返回 True。

**Python赋值运算符**

以下假设变量a为10，变量b为20：

运算符 描述 实例

= 简单的赋值运算符 c = a + b 将 a + b 的运算结果赋值为 c

+= 加法赋值运算符 c += a 等效于 c = c + a

-= 减法赋值运算符 c -= a 等效于 c = c - a

\*= 乘法赋值运算符 c \*= a 等效于 c = c \* a

/= 除法赋值运算符 c /= a 等效于 c = c / a

%= 取模赋值运算符 c %= a 等效于 c = c % a

\*\*= 幂赋值运算符 c \*\*= a 等效于 c = c \*\* a

//= 取整除赋值运算符 c //= a 等效于 c = c // a  
**Python位运算符**

按位运算符是把数字看作二进制来进行计算的。Python中的按位运算法则如下：

下表中变量 a 为 60，b 为 13。

运算符 描述 实例

& 按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0 (a & b) 输出结果 12 ，二进制解释： 0000 1100

| 按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。 (a | b) 输出结果 61 ，二进制解释： 0011 1101

^ 按位异或运算符：当两对应的二进位相异时，结果为1 (a ^ b) 输出结果 49 ，二进制解释： 0011 0001

~ 按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1 (~a ) 输出结果 -61 ，二进制解释： 1100 0011， 在一个有符号二进制数的补码形式。

<< 左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。 a << 2 输出结果 240 ，二进制解释： 1111 0000

>> 右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数 a >> 2 输出结果 15 ，二进制解释： 0000 1111  
**Python逻辑运算符**

Python语言支持逻辑运算符，以下假设变量 a 为 10, b为 20:

运算符 逻辑表达式 描述 实例

and x and y 布尔"与" - 如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值。 (a and b) 返回 20。

or x or y 布尔"或" - 如果 x 是 True，它返回 True，否则它返回 y 的计算值。 (a or b) 返回 10。

not not x 布尔"非" - 如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。 not(a and b) 返回 False  
**Python成员运算符**

除了以上的一些运算符之外，Python还支持成员运算符，测试实例中包含了一系列的成员，包括字符串，列表或元组。

运算符 描述 实例

in 如果在指定的序列中找到值返回 True，否则返回 False。 x 在 y 序列中 , 如果 x 在 y 序列中返回 True。

not in 如果在指定的序列中没有找到值返回 True，否则返回 False。 x 不在 y 序列中 , 如果 x 不在 y 序列中返回 True。  
**Python身份运算符**

身份运算符用于比较两个对象的存储单元

运算符 描述 实例

is is是判断两个标识符是不是引用自一个对象 x is y, 如果 id(x) 等于 id(y) , is 返回结果 1

is not is not是判断两个标识符是不是引用自不同对象 x is not y, 如果 id(x) 不等于 id(y). is not 返回结果 1

**字符串例子：**

或者，字符串可以被 """ （三个双引号）或者 ''' （三个单引号）括起来。使用三引号时，换行符不需要转义，它们会包含在字符串中。以下的例子使用了一个转义符，避免在最开始产生一个不需要的空行。

print("""\

Usage: thingy [OPTIONS]

-h Display this usage message

-H hostname Hostname to connect to

""")

其输出如下：

Usage: thingy [OPTIONS]

-h Display this usage message

-H hostname Hostname to connect to  
有一个方法可以让您记住分切索引的工作方式，想像索引是指向字符之间，第一个字符左边的数字是 0。接着，有n个字符的字符串最后一个字符的右边是索引n，例如：

+---+---+---+---+---+

| H | e | l | p | A |

+---+---+---+---+---+

0 1 2 3 4 5

-5 -4 -3 -2 -1

**Python列表：**  
也可以使用嵌套列表（在列表里创建其它列表），例如：

>>> a = ['a', 'b', 'c']

>>> n = [1, 2, 3]

>>> x = [a, n]

>>> x

[['a', 'b', 'c'], [1, 2, 3]]

>>> x[0]

['a', 'b', 'c']

>>> x[0][1]

'b'  
python元组：  
Python 的元组与列表类似，不同之处在于元组的元素不能修改。

元组使用小括号，列表使用方括号。

元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

如下实例：

tup1 = ('Google', 'W3CSchool', 1997, 2000);

tup2 = (1, 2, 3, 4, 5 );

tup3 = "a", "b", "c", "d";  
创建空元组

tup1 = ();

元组中只包含一个元素时，需要在元素后面添加逗号

tup1 = (50,);

元组与字符串类似，下标索引从0开始，可以进行截取，组合等。  
访问元组

元组可以使用下标索引来访问元组中的值，如下实例:

#!/usr/bin/python3

tup1 = ('Google', 'W3CSchool', 1997, 2000)

tup2 = (1, 2, 3, 4, 5, 6, 7 )

print ("tup1[0]: ", tup1[0])

print ("tup2[1:5]: ", tup2[1:5])

以上实例输出结果：

tup1[0]: Google

tup2[1:5]: (2, 3, 4, 5)

修改元组

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合，如下实例:

#!/usr/bin/python3

tup1 = (12, 34.56);

tup2 = ('abc', 'xyz')

# 以下修改元组元素操作是非法的。

# tup1[0] = 100

# 创建一个新的元组

tup3 = tup1 + tup2;

print (tup3)

以上实例输出结果：

(12, 34.56, 'abc', 'xyz')  
**删除元组**

元组中的元素值是不允许删除的，但我们可以使用del语句来删除整个元组，如下实例:

#!/usr/bin/python3

tup = ('Google', 'W3CSchool', 1997, 2000)

print (tup)

del tup;

print ("删除后的元组 tup : ")

print (tup)  
**元组运算符**

与字符串一样，元组之间可以使用 + 号和 \* 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的元组。

Python 表达式 结果 描述

len((1, 2, 3)) 3 计算元素个数

(1, 2, 3) + (4, 5, 6) (1, 2, 3, 4, 5, 6) 连接

['Hi!'] \* 4 ['Hi!', 'Hi!', 'Hi!', 'Hi!'] 复制

3 in (1, 2, 3) True 元素是否存在

for x in (1, 2, 3): print x, 1 2 3 迭代  
**元组内置函数**

Python元组包含了以下内置函数  
len(tuple)

计算元组元素个数。

max(tuple)

返回元组中元素最大值。

min(tuple)

返回元组中元素最小值。

tuple(seq)

将列表转换为元组。

**Python3 字典**

字典是另一种可变容器模型，且可存储任意类型对象。

字典的每个键值(key=>value)对用冒号(:)分割，每个对之间用逗号(,)分割，整个字典包括在花括号({})中 ,格式如下所示：

d = {key1 : value1, key2 : value2 }

键必须是唯一的，但值则不必。

值可以取任何数据类型，但键必须是不可变的，如字符串，数字或元组。

一个简单的字典实例：

dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}

也可如此创建字典：

dict1 = { 'abc': 456 };

dict2 = { 'abc': 123, 98.6: 37 };  
**删除字典元素**

能删单一的元素也能清空字典，清空只需一项操作。

显示删除一个字典用del命令，如下实例：

#!/usr/bin/python3

dict = {'Name': 'W3CSchool', 'Age': 7, 'Class': 'First'}

del dict['Name'] # 删除键 'Name'

dict.clear() # 删除字典

del dict # 删除字典

print ("dict['Age']: ", dict['Age'])

print ("dict['School']: ", dict['School'])  
**字典键的特性**

字典值可以没有限制地取任何python对象，既可以是标准的对象，也可以是用户定义的，但键不行。

两个重要的点需要记住：

不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住，如下实例  
键必须不可变，所以可以用数字，字符串或元组充当，而用列表就不行  
**字典内置函数&方法**

Python字典包含了以下内置函数：

len(dict)

计算字典元素个数，即键的总数。  
str(dict)

输出字典以可打印的字符串表示。  
type(variable)

返回输入的变量类型，如果变量是字典就返回字典类型。  
Python字典包含了以下内置方法：

序号 函数及描述

1 radiansdict.clear()

删除字典内所有元素

2 radiansdict.copy()

返回一个字典的浅复制

3 radiansdict.fromkeys()

创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值

4 radiansdict.get(key, default=None)

返回指定键的值，如果值不在字典中返回default值

5 key in dict

如果键在字典dict里返回true，否则返回false

6 radiansdict.items()

以列表返回可遍历的(键, 值) 元组数组

7 radiansdict.keys()

以列表返回一个字典所有的键

8 radiansdict.setdefault(key, default=None)

和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default

9 radiansdict.update(dict2)

把字典dict2的键/值对更新到dict里

10 radiansdict.values()

以列表返回字典中的所有值  
**Python3 条件控制**  
if 语句

Python中if语句的一般形式如下所示：

if condition\_1:

statement\_block\_1

elif condition\_2:

statement\_block\_2

else:

statement\_block\_3  
Python中用elif代替了else if，所以if语句的关键字为：if – elif – else。

注意：

1、每个条件后面要使用冒号（:），表示接下来是满足条件后要执行的语句块。

2、使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。

3、在Python中没有switch – case语句。  
**while 循环**

Python中while语句的一般形式：

while 判断条件：

statements  
**for语句**

Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

for循环的一般格式如下：  
for <variable> in <sequence>:

<statements>

else:

<statements>  
Python loop循环实例：

>>> languages = ["C", "C++", "Perl", "Python"]

>>> for x in languages:

... print x

...

C

C++

Perl

Python

以下实例for实例中使用了 break语句，break语句用于跳出当前循环体：

#!/usr/bin/env python3

edibles = ["ham", "spam","eggs","nuts"]

for food in edibles:

if food == "spam":

print("No more spam please!")

break

print("Great, delicious " + food)

else:

print("I am so glad: No spam!")

print("Finally, I finished stuffing myself")  
**range()函数**

**如果你需要遍历数字序列，可以使用内置range()函数。它会生成数列，例如:**

>>> for i in range(5):

... print(i)

...

0

1

2

3

4  
**你也可以使用range指定区间的值：**

>>> for i in range(5,9) :

print(i)

5

6

7

8

>>>

也可以使range以指定数字开始并指定不同的增量(甚至可以是负数;有时这也叫做'步长'):

>>> for i in range(0, 10, 3) :

print(i)

0

3

6

9

>>>   
您可以结合range()和len()函数以遍历一个序列的索引,如下所示:

>>> a = ['Mary', 'had', 'a', 'little', 'lamb']

>>> for i in range(len(a)):

... print(i, a[i])

...

0 Mary

1 had

2 a

3 little

4 lamb

还可以使用range()函数来创建一个列表：

>>> list(range(5))

[0, 1, 2, 3, 4]

>>>  
**break和continue语句及循环中的else子句**

break语句可以跳出for和while的循环体。如果你从for或while循环中终止，任何对应的循环else块将不执行。

continue语句被用来告诉Python跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

循环语句可以有else子句;它在穷尽列表(以for循环)或条件变为假(以while循环)循环终止时被执行,但循环被break终止时不执行  
**pass语句**

pass语句什么都不做。它只在语法上需要一条语句但程序不需要任何操作时使用.例如:

>>> while True:

... pass # 等待键盘中断 (Ctrl+C)  
**迭代器**

迭代是Python最强大的功能之一，是访问集合元素的一种方式。。

迭代器是一个可以记住遍历的位置的对象。

迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：iter() 和 next()。

字符串，列表或元组对象都可用于创建迭代器：  
>>> list=[1,2,3,4]

>>> it = iter(list) # 创建迭代器对象

>>> print (next(it)) # 输出迭代器的下一个元素

1

>>> print (next(it))

2

>>>

迭代器对象可以使用常规for语句进行遍历：

#!/usr/bin/python3

list=[1,2,3,4]

it = iter(list) # 创建迭代器对象

for x in it:

print (x, end=" ")

执行以上程序，输出结果如下：

1 2 3 4

也可以使用 next() 函数：

#!/usr/bin/python3

import sys # 引入 sys 模块

list=[1,2,3,4]

it = iter(list) # 创建迭代器对象

while True:

try:

print (next(it))

except StopIteration:

sys.exit()

执行以上程序，输出结果如下：

1

2

3

4

**生成器**

在 Python 中，使用了 yield 的函数被称为生成器（generator）。

跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器。

在调用生成器运行的过程中，每次遇到 yield 时函数会暂停并保存当前所有的运行信息，返回yield的值。并在下一次执行 next()方法时从当前位置继续运行。

**Python 函数**

def 函数名（参数列表）：

函数体

让我们使用函数来输出"Hello World！"：

>>> def hello() :

print("Hello World!")

>>> hello()

Hello World!

>>>

**函数返回值**

Python的函数的返回值使用return语句，可以将函数作为一个值赋值给指定变量：

def return\_sum(x,y):

c = x + y

return c

res = return\_sum(4,5)

print(res)

你也可以让函数返回空值：

def empty\_return(x,y):

c = x + y

return

res = empty\_return(4,5)

print(res)   
**可变参数列表**

最后,一个最不常用的选择是可以让函数调用可变个数的参数.这些参数被包装进一个元组(查看元组和序列).在这些可变个数的参数之前,可以有零到多个普通的参数:

def arithmetic\_mean(\*args):

sum = 0

for x in args:

sum += x

return sum

print(arithmetic\_mean(45,32,89,78))

print(arithmetic\_mean(8989.8,78787.78,3453,78778.73))

print(arithmetic\_mean(45,32))

print(arithmetic\_mean(45))

print(arithmetic\_mean())

**Python数据结构**

**列表**

Python中列表是可变的，这是它区别于字符串和元组的最重要的特点，一句话概括即：列表可以修改，而字符串和元组不能。

以下是 Python 中列表的方法：  
list.append(x)  
list.extend(L)  
list.insert(i, x)  
list.remove(x)  
list.pop([i])：从列表的指定位置删除元素，并将其返回。如果没有指定索引，a.pop()返回最后一个元素。元素随即从列表中被删除。（方法中 i 两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号，你会经常在 Python 库参考手册中遇到这样的标记。）  
list.clear()  
list.index(x)：返回列表中第一个值为 x 的元素的索引。如果没有匹配的元素就会返回一个错误。

list.count(x)  
list.sort()

list.reverse()

list.copy()： 返回列表的浅复制，等于a[:]。  
**将列表当做堆栈使用**

列表方法使得列表可以很方便的作为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放（后进先出）。用 append() 方法可以把一个元素添加到堆栈顶。用不指定索引的 pop() 方法可以把一个元素从堆栈顶释放出来。例如：  
**将列表当作队列使用**

也可以把列表当做队列用，只是在队列里第一加入的元素，第一个取出来；但是拿列表用作这样的目的效率不高。在列表的最后添加或者弹出元素速度快，然而在列表里插入或者从头部弹出速度却不快（因为所有其他的元素都得一个一个地移动）。  
**列表推导式**

列表推导式提供了从序列创建列表的简单途径。通常应用程序将一些操作应用于某个序列的每个元素，用其获得的结果作为生成新列表的元素，或者根据确定的判定条件创建子序列。

每个列表推导式都在 for 之后跟一个表达式，然后有零到多个 for 或 if 子句。返回结果是一个根据表达从其后的 for 和 if 上下文环境中生成出来的列表。如果希望表达式推导出一个元组，就必须使用括号。

这里我们将列表中每个数值乘三，获得一个新的列表：  
>>> vec = [2, 4, 6]

>>> [3\*x for x in vec]

[6, 12, 18]  
现在我们玩一点小花样：

>>> [[x, x\*\*2] for x in vec]

[[2, 4], [4, 16], [6, 36]]  
**这里我们对序列里每一个元素逐个调用某方法：**

>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']

>>> [weapon.strip() for weapon in freshfruit]

['banana', 'loganberry', 'passion fruit']  
**我们可以用 if 子句作为过滤器：**

>>> [3\*x for x in vec if x > 3]

[12, 18]

>>> [3\*x for x in vec if x < 2] []

**以下是一些关于循环和其它技巧的演示：**

>>> 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]  
**列表推导式可以使用复杂表达式或嵌套函数：**

>>> [str(round(355/113, i)) for i in range(1, 6)]

['3.1', '3.14', '3.142', '3.1416', '3.14159']  
嵌套列表解析

Python的列表还可以嵌套。

以下实例展示了3X4的矩阵列表：

>>> matrix = [

... [1, 2, 3, 4],

... [5, 6, 7, 8],

... [9, 10, 11, 12],

... ]  
以下实例将3X4的矩阵列表转换为4X3列表：

>>> [[row[i] for row in matrix] for i in range(4)]

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]  
**以下实例也可以使用以下方法来实现：**

>>> transposed = []

>>> for i in range(4):

... transposed.append([row[i] for row in matrix])

...

>>> transposed

[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]  
**del 语句**

使用 del 语句可以从一个列表中依索引而不是值来删除一个元素。这与使用 pop() 返回一个值不同。可以用 del 语句从列表中删除一个切割，或清空整个列表（我们以前介绍的方法是给该切割赋一个空列表）。例如：  
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]

>>> del a[0]

>>> a

[1, 66.25, 333, 333, 1234.5]

>>> del a[2:4]

>>> a

[1, 66.25, 1234.5]

>>> del a[:]

>>> a

[]

也可以用 del 删除实体变量：

>>> del a  
**元组和序列**

元组由若干逗号分隔的值组成，例如：

>>> t = 12345, 54321, 'hello!'

>>> t[0]

12345

>>> t

(12345, 54321, 'hello!')

>>> # Tuples may be nested:

... u = t, (1, 2, 3, 4, 5)

>>> u

((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可能有或没有括号， 不过括号通常是必须的（如果元组是更大的表达式的一部分）。

**集合**

集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。

可以用大括号({})创建集合。注意：如果要创建一个空集合，你必须用 set() 而不是 {} ；后者创建一个空的字典，下一节我们会介绍这个数据结构。

以下是一个简单的演示：

>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}

>>> print(basket) # show that duplicates have been removed

{'orange', 'banana', 'pear', 'apple'}

>>> 'orange' in basket # fast membership testing

True

>>> 'crabgrass' in basket

False

**集合也支持推导式：**

>>> a = {x for x in 'abracadabra' if x not in 'abc'}

>>> a

{'r', 'd'}

**字典**

另一个非常有用的 Python 内建数据类型是字典。

序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。

理解字典的最佳方式是把它看做无序的键=>值对集合。在同一个字典之内，关键字必须是互不相同。

一对大括号创建一个空的字典：{}。

这是一个字典运用的简单例子：

>>> tel = {'jack': 4098, 'sape': 4139}

>>> tel['guido'] = 4127

>>> tel

{'sape': 4139, 'guido': 4127, 'jack': 4098}

>>> tel['jack']

4098

>>> del tel['sape']

>>> tel['irv'] = 4127

>>> tel

{'guido': 4127, 'irv': 4127, 'jack': 4098}

>>> list(tel.keys())

['irv', 'guido', 'jack']

>>> sorted(tel.keys())

['guido', 'irv', 'jack']

>>> 'guido' in tel

True

>>> 'jack' not in tel

False

**构造函数 dict() 直接从键值对元组列表中构建字典。如果有固定的模式，列表推导式指定特定的键值对：**

>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

{'sape': 4139, 'jack': 4098, 'guido': 4127}

**此外，字典推导可以用来创建任意键和值的表达式词典：**

>>> {x: x\*\*2 for x in (2, 4, 6)}

{2: 4, 4: 16, 6: 36}

**如果关键字只是简单的字符串，使用关键字参数指定键值对有时候更方便：**

>>> dict(sape=4139, guido=4127, jack=4098)

{'sape': 4139, 'jack': 4098, 'guido': 4127}

**遍历技巧**

在字典中遍历时，关键字和对应的值可以使用 items() 方法同时解读出来：

>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}

>>> for k, v in knights.items():

... print(k, v)

...

gallahad the pure

robin the brave

在序列中遍历时，索引位置和对应值可以使用 enumerate() 函数同时得到：

>>> for i, v in enumerate(['tic', 'tac', 'toe']):

... print(i, v)

...

0 tic

1 tac

2 toe

同时遍历两个或更多的序列，可以使用 zip() 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.

要反向遍历一个序列，首先指定这个序列，然后调用 reversesd() 函数：

>>> for i in reversed(range(1, 10, 2)):

... print(i)

...

9

7

5

3

1

要按顺序遍历一个序列，使用 sorted() 函数返回一个已排序的序列，并不修改原值：

>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

>>> for f in sorted(set(basket)):

... print(f)

...

apple

banana

orange

Pear

**输出格式美化**

Python两种输出值的方式: 表达式语句和 print() 函数。(第三种方式是使用文件对象的 write() 方法; 标准输出文件可以用 sys.stdout 引用。)

如果你希望输出的形式更加多样，可以使用 str.format() 函数来格式化输出值。

如果你希望将输出的值转成字符串，可以使用 repr() 或 str() 函数来实现。

str() 函数返回一个用户易读的表达形式。

repr() 产生一个解释器易读的表达形式。

>>> for x in range(1, 11):

... print(repr(x).rjust(2), repr(x\*x).rjust(3), end=' ')

... # 注意前一行 'end' 的使用

... print(repr(x\*x\*x).rjust(4))

注意：在第一个例子中, 每列间的空格由 print() 添加。

这个例子展示了字符串对象的 rjust() 方法, 它可以将字符串靠右, 并在左边填充空格。

还有类似的方法, 如 ljust() 和 center()。 这些方法并不会写任何东西, 它们仅仅返回新的字符串。

另一个方法 zfill(), 它会在数字的左边填充 0，如下所示：

>>> '12'.zfill(5)

'00012'

>>> '-3.14'.zfill(7)

'-003.14'

>>> '3.14159265359'.zfill(5)

'3.14159265359'

**str.format() 的基本使用如下:**

>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))

We are the knights who say "Ni!"

括号及其里面的字符 (称作格式化字段) 将会被 format() 中的参数替换。

在括号中的数字用于指向传入对象在 format() 中的位置，如下所示：

>>> print('{0} and {1}'.format('spam', 'eggs'))

spam and eggs

>>> print('{1} and {0}'.format('spam', 'eggs'))

eggs and spam

如果在 format() 中使用了关键字参数, 那么它们的值会指向使用该名字的参数。

>>> print('This {food} is {adjective}.'.format(

... food='spam', adjective='absolutely horrible'))

This spam is absolutely horrible.

位置及关键字参数可以任意的结合:

>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',

other='Georg'))

The story of Bill, Manfred, and Georg.

'!a' (使用 ascii()), '!s' (使用 str()) 和 '!r' (使用 repr()) 可以用于在格式化某个值之前对其进行转化:

>>> import math

>>> print('The value of PI is approximately {}.'.format(math.pi))

The value of PI is approximately 3.14159265359.

>>> print('The value of PI is approximately {!r}.'.format(math.pi))

The value of PI is approximately 3.141592653589793.

可选项 ':' 和格式标识符可以跟着字段名。 这就允许对值进行更好的格式化。 下面的例子将 Pi 保留到小数点后三位：

>>> import math

>>> print('The value of PI is approximately {0:.3f}.'.format(math.pi))

The value of PI is approximately 3.142.

在 ':' 后传入一个整数, 可以保证该域至少有这么多的宽度。 用于美化表格时很有用。

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}

>>> for name, phone in table.items():

... print('{0:10} ==> {1:10d}'.format(name, phone))

...

Jack ==> 4098

Dcab ==> 7678

Sjoerd ==> 4127

如果你有一个很长的格式化字符串, 而你不想将它们分开, 那么在格式化时通过变量名而非位置会是很好的事情。

最简单的就是传入一个字典, 然后使用方括号 '[]' 来访问键值 :

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}

>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '

'Dcab: {0[Dcab]:d}'.format(table))

Jack: 4098; Sjoerd: 4127; Dcab: 8637678

也可以通过在 table 变量前使用 '\*\*' 来实现相同的功能：

>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}

>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(\*\*table))

Jack: 4098; Sjoerd: 4127; Dcab: 8637678

**旧式字符串格式化**

% 操作符也可以实现字符串格式化。 它将左边的参数作为类似 sprintf() 式的格式化字符串, 而将右边的代入, 然后返回格式化后的字符串. 例如:

>>> import math

>>> print('The value of PI is approximately %5.3f.' % math.pi)

The value of PI is approximately 3.142.

因为 str.format() 比较新的函数， 大多数的 Python 代码仍然使用 % 操作符。但是因为这种旧式的格式化最终会从该语言中移除, 应该更多的使用 str.format().

**读和写文件**

open() 将会返回一个 file 对象，基本语法格式如下:

open(filename, mode)

实例:

>>> f = open('/tmp/workfile', 'w')

第一个参数为要打开的文件名。

第二个参数描述文件如何使用的字符。 mode 可以是 'r' 如果文件只读, 'w' 只用于写 (如果存在同名文件则将被删除), 和 'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾. 'r+' 同时用于读写。 mode 参数是可选的; 'r' 将是默认值。

**文件对象的方法**

本节中剩下的例子假设已经创建了一个称为 f 的文件对象。

f.read()

为了读取一个文件的内容，调用 f.read(size), 这将读取一定数目的数据, 然后作为字符串或字节对象返回。

size 是一个可选的数字类型的参数。 当 size 被忽略了或者为负, 那么该文件的所有内容都将被读取并且返回。

>>> f.read()

'This is the entire file.\n'

>>> f.read()

''

f.readline()

f.readline() 会从文件中读取单独的一行。换行符为 '\n'。f.readline() 如果返回一个空字符串, 说明已经已经读取到最后一行。

>>> f.readline()

'This is the first line of the file.\n'

>>> f.readline()

'Second line of the file\n'

>>> f.readline()

''

f.readlines()

f.readlines() 将返回该文件中包含的所有行。

如果设置可选参数 sizehint, 则读取指定长度的字节, 并且将这些字节按行分割。

>>> f.readlines()

['This is the first line of the file.\n', 'Second line of the file\n']

**另一种方式是迭代一个文件对象然后读取每行:**

>>> for line in f:

... print(line, end='')

...

This is the first line of the file.

Second line of the file

这个方法很简单, 但是并没有提供一个很好的控制。 因为两者的处理机制不同, 最好不要混用。

**f.write()**

f.write(string) 将 string 写入到文件中, 然后返回写入的字符数。

>>> f.write('This is a test\n')

15

如果要写入一些不是字符串的东西, 那么将需要先进行转换:

>>> value = ('the answer', 42)

>>> s = str(value)

>>> f.write(s)

18

**f.tell()**

f.tell() 返回文件对象当前所处的位置, 它是从文件开头开始算起的字节数。

**f.seek()**

如果要改变文件当前的位置, 可以使用 f.seek(offset, from\_what) 函数。

from\_what 的值, 如果是 0 表示开头, 如果是 1 表示当前位置, 2 表示文件的结尾，例如：

seek(x,0) ： 从起始位置即文件首行首字符开始移动 x 个字符

seek(x,1) ： 表示从当前位置往后移动x个字符

seek(-x,2)：表示从文件的结尾往前移动x个字符

from\_what 值为默认为0，即文件开头。下面给出一个完整的例子：

>>> f = open('/tmp/workfile', 'rb+')

>>> f.write(b'0123456789abcdef')

16

>>> f.seek(5) # 移动到文件的第六个字节

5

>>> f.read(1)

b'5'

>>> f.seek(-3, 2) # 移动到文件的倒数第三字节

13

>>> f.read(1)

b'd'

**f.close()**

在文本文件中 (那些打开文件的模式下没有 b 的), 只会相对于文件起始位置进行定位。

当你处理完一个文件后, 调用 f.close() 来关闭文件并释放系统的资源，如果尝试再调用该文件，则会抛出异常。

>>> f.close()

>>> f.read()

Traceback (most recent call last):

File "<stdin>", line 1, in ?

ValueError: I/O operation on closed file

<pre>

<p>

当处理一个文件对象时, 使用 with 关键字是非常好的方式。在结束后, 它会帮你正确的关闭文件。 而且写起来也比 try - finally 语句块要简短:</p>

<pre>

>>> with open('/tmp/workfile', 'r') as f:

... read\_data = f.read()

>>> f.closed

True

文件对象还有其他方法, 如 isatty() 和 trucate(), 但这些通常比较少用。

**pickle 模块**

python的pickle模块实现了基本的数据序列和反序列化。

通过pickle模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去，永久存储。

通过pickle模块的反序列化操作，我们能够从文件中创建上一次程序保存的对象。

**基本接口：**

pickle.dump(obj, file, [,protocol])

有了 pickle 这个对象, 就能对 file 以读取的形式打开:

x = pickle.load(file)

注解：从 file 中读取一个字符串，并将它重构为原来的python对象。

file: 类文件对象，有read()和readline()接口。

实例1：

#使用pickle模块将数据对象保存到文件

import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],

'b': ('string', u'Unicode string'),

'c': None}

selfref\_list = [1, 2, 3]

selfref\_list.append(selfref\_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.

pickle.dump(data1, output)

# Pickle the list using the highest protocol available.

pickle.dump(selfref\_list, output, -1)

output.close()

实例2：

#使用pickle模块从文件中重构python对象

import pprint, pickle

pkl\_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl\_file)

pprint.pprint(data1)

data2 = pickle.load(pkl\_file)

pprint.pprint(data2)

pkl\_file.close()

**Python3 File(文件) 方法**

file 对象使用 open 函数来创建，下表列出了 file 对象常用的函数：

file.close()

关闭文件。关闭后文件不能再进行读写操作。

file.flush()

刷新文件内部缓冲，直接把内部缓冲区的数据立刻写入文件, 而不是被动的等待输出缓冲区写入。

file.fileno()

返回一个整型的文件描述符(file descriptor FD 整型), 可以用在如os模块的read方法等一些底层操作上。

file.isatty()

如果文件连接到一个终端设备返回 True，否则返回 False。

file.next()

返回文件下一行。

file.read([size])

从文件读取指定的字节数，如果未给定或为负则读取所有。

file.readline([size])

读取整行，包括 "\n" 字符。

file.readlines([sizehint])

读取所有行并返回列表，若给定sizeint>0，返回总和大约为sizeint字节的行, 实际读取值可能比sizhint较大, 因为需要填充缓冲区。

file.seek(offset[, whence])

设置文件当前位置

file.tell()

返回文件当前位置。

file.truncate([size])

截取文件，截取的字节通过size指定，默认为当前文件位置。

file.write(str)

将字符串写入文件，没有返回值。

file.writelines(sequence)

向文件写入一个序列字符串列表，如果需要换行则要自己加入每行的换行符。

**Python3 OS 文件/目录方法**

os 模块提供了非常丰富的方法用来处理文件和目录。常用的方法如下表所示：

**Python3 OS 文件/目录方法**

os 模块提供了非常丰富的方法用来处理文件和目录。常用的方法如下表所示：

os.access(path, mode)

检验权限模式

os.chdir(path)

改变当前工作目录

os.chflags(path, flags)

设置路径的标记为数字标记。

os.chmod(path, mode)

更改权限

os.chown(path, uid, gid)

更改文件所有者

os.chroot(path)

改变当前进程的根目录

os.close(fd)

关闭文件描述符 fd

os.closerange(fd\_low, fd\_high)

关闭所有文件描述符，从 fd\_low (包含) 到 fd\_high (不包含), 错误会忽略

os.dup(fd)

复制文件描述符 fd

os.dup2(fd, fd2)

将一个文件描述符 fd 复制到另一个 fd2

os.fchdir(fd)

通过文件描述符改变当前工作目录

os.fchmod(fd, mode)

改变一个文件的访问权限，该文件由参数fd指定，参数mode是Unix下的文件访问权限。

[http://www.w3cschool.cn/python3/python3-os-file-methods.html](http://www.w3cschool.cn/python3/python3-os-file-methods.html详细介绍。)

**异常**

即便Python程序的语法是正确的，在运行它的时候，也有可能发生错误。运行期检测到的错误被称为异常。

大多数的异常都不会被程序处理，都以错误信息的形式展现在这里:

异常以不同的类型出现，这些类型都作为信息的一部分打印出来: 例子中的类型有 ZeroDivisionError，NameError 和 TypeError。

错误信息的前面部分显示了异常发生的上下文，并以调用栈的形式显示具体信息。

**异常处理**

以下例子中，让用户输入一个合法的整数，但是允许用户中断这个程序（使用 Control-C 或者操作系统提供的方法）。用户中断的信息会引发一个 KeyboardInterrupt 异常。

>>> while True:

try:

x = int(input("Please enter a number: "))

break

except ValueError:

print("Oops! That was no valid number. Try again ")

try语句按照如下方式工作；

首先，执行try子句（在关键字try和关键字except之间的语句）

如果没有异常发生，忽略except子句，try子句执行后结束。

如果在执行try子句的过程中发生了异常，那么try子句余下的部分将被忽略。如果异常的类型和 except 之后的名称相符，那么对应的except子句将被执行。最后执行 try 语句之后的代码。

如果一个异常没有与任何的except匹配，那么这个异常将会传递给上层的try中。

一个 try 语句可能包含多个except子句，分别来处理不同的特定的异常。最多只有一个分支会被执行。

处理程序将只针对对应的try子句中的异常进行处理，而不是其他的 try 的处理程序中的异常。

一个except子句可以同时处理多个异常，这些异常将被放在一个括号里成为一个元组，例如:

except (RuntimeError, TypeError, NameError):

pass

最后一个except子句可以忽略异常的名称，它将被当作通配符使用。你可以使用这种方法打印一个错误信息，然后再次把异常抛出。

try except 语句还有一个可选的else子句，如果使用这个子句，那么必须放在所有的except子句之后。这个子句将在try子句没有发生任何异常的时候执行。例如:

for arg in sys.argv[1:]:

try:

f = open(arg, 'r')

except IOError:

print('cannot open', arg)

else:

print(arg, 'has', len(f.readlines()), 'lines')

f.close()

抛出异常

Python 使用 raise 语句抛出一个指定的异常。例如:

>>> raise NameError('HiThere')

Traceback (most recent call last):

File "<stdin>", line 1, in ?

NameError: HiThere

raise 唯一的一个参数指定了要被抛出的异常。它必须是一个异常的实例或者是异常的类（也就是 Exception 的子类）。

如果你只想知道这是否抛出了一个异常，并不想去处理它，那么一个简单的 raise 语句就可以再次把它抛出。

用户自定义异常

你可以通过创建一个新的exception类来拥有自己的异常。异常应该继承自 Exception 类，或者直接继承，或者间接继承，例如:

>>> class MyError(Exception):

def \_\_init\_\_(self, value):

self.value = value

def \_\_str\_\_(self):

return repr(self.value)

>>> try:

raise MyError(2\*2)

except MyError as e:

print('My exception occurred, value:', e.value)

**定义清理行为**

try 语句还有另外一个可选的子句，它定义了无论在任何情况下都会执行的清理行为。 例如:

>>> try:

raise KeyboardInterrupt

finally:

print('Goodbye, world!')

Goodbye, world!

KeyboardInterrupt

以上例子洪不管try子句里面有没有发生异常，finally子句都会执行。

如果一个异常在 try 子句里（或者在 except 和 else 子句里）被抛出，而又没有任何的 except 把它截住，那么这个异常会在 finally 子句执行后再次被抛出。

下面是一个更加复杂的例子（在同一个 try 语句里包含 except 和 finally 子句）:

>>> def divide(x, y):

try:

result = x / y

except ZeroDivisionError:

print("division by zero!")

else:

print("result is", result)

finally:

print("executing finally clause")

预定义的清理行为

一些对象定义了标准的清理行为，无论系统是否成功的使用了它，一旦不需要它了，那么这个标准的清理行为就会执行。

这面这个例子展示了尝试打开一个文件，然后把内容打印到屏幕上:

for line in open("myfile.txt"):

print(line, end="")

以上这段代码的问题是，当执行完毕后，文件会保持打开状态，并没有被关闭。

关键词 with 语句就可以保证诸如文件之类的对象在使用完之后一定会正确的执行他的清理方法:

with open("myfile.txt") as f:

for line in f:

print(line, end="")

以上这段代码执行完毕后，就算在处理过程中出问题了，文件 f 总是会关闭。

**视屏笔记**