## 第八章 条件和循环

### 8.1case语句替代方法

Python没有case，可以用序列和成员关系操作符来简化它:

if user.cmd in ('create', 'delete', 'update'):

action = '%s item' % user.cmd

else:

action = 'invalid choice... try again!'

用 Python 字典给出更加优雅的case语句解决方案：

msgs = {'create': 'create item',

'delete': 'delete item',

'update': 'update item'}

default = 'invalid choice... try again!'

action = msgs.get(user.cmd, default)

### 8.2三元操作符：

>>> x, y = 4, 3

>>> smaller = x if x < y else y

>>> smaller

3

### 8.3 for语句用于序列类型

3.1通过序列项迭代

>>> nameList = ['Walter', "Nicole", 'Steven', 'Henry']

>>> for eachName in nameList:

... print eachName, "Lim"

3.2通过序列索引迭代

另个方法就是通过序列的索引来迭代:

>>> nameList = ['Cathy', "Terry", 'Joe', 'Heather', 'Lucy']

>>> for nameIndex in range(len(nameList)):

... print "Liu,", nameList[nameIndex]

...

Liu, Cathy

Liu, Terry

Liu, Joe

Liu, Heather

Liu, Lucy

3.3使用项和索引迭代

两全其美的办法是使用内建的 enumerate() 函数, 它是 Python 2.3 的新增内容. 代码如下:

>>> nameList = ['Donn', 'Shirley', 'Ben', 'Janice', 'David', 'Yen', 'Wendy']

>>> for i, eachLee in enumerate(nameList):

... print "%d %s Lee" % (i+1, eachLee)

...

1 Donn Lee

2 Shirley Lee

3 Ben Lee

。。。

### 8.4 for语句用于迭代器类型

迭代器对象有一个 next() 方法, 调用后返回下一个条目. 所有条目迭代完后, 迭代器引发一个 StopIteration 异常告诉程序循环结束. for 语句在内部调用 next() 并捕获异常.

使用迭代器做 for 循环的代码与使用序列条目几乎完全相同. 事实上在大多情况下, 你无法分辨出你迭代的是一个序列还是迭代器, 因此，这就是为什么我们在说要遍历一个迭代器时，实际上可能我们指的是要遍历一个序列，迭代器，或是一个支持迭代的对象（它有 next()方法）。

### 8.5 for语句使用内建函数range

>>> for eachVal in range(2, 19, 3):

... print "value is:", eachVal

三种调用方法

range(start, end, step =1)

range(end)

range(start, end)

### 8.6与序列相关的内建函数

sorted(),zip() 返回一个序列

reversed(), enumerate() 返回迭代器 reversed(), enumerate() 这两个类中有next和\_\_iter\_\_ 方法

下边是使用循环相关和序列相关函数的例子. 为什么它们叫"序列相关"呢? 是因为其中两个函数( sorted() 和 zip() )返回一个序列(列表), 而另外两个函数( reversed() 和 enumerate() ) 返回迭代器(类似序列)

### 8.7 迭代器和序列

可以用在 for 语句进行循环的对象就是**可迭代对象**。除了内置的数据类型（列表、元组、字符串、字典等）可以通过 for 语句进行迭代，我们也可以自己创建一个容器，包含一系列元素，可以通过 for 语句依次循环取出每一个元素，这种容器就是**迭代器（iterator）**。除了用 for 遍历，迭代器还可以通过 next() 方法逐一读取下一个元素。要创建一个迭代器有3种方法，其中前两种分别是：

1.为容器对象添加 \_\_iter\_\_() 和 \_\_next\_\_() 方法（Python 2.7 中是 next()）；\_\_iter\_\_() 返回迭代器对象本身 self，\_\_next\_\_() 则返回每次调用 next() 或迭代时的元素；

2.内置函数 iter() 将可迭代对象转化为迭代器

# iter(IterableObject)

ita = iter([1, 2, 3])

print(type(ita))

print(next(ita))

print(next(ita))

print(next(ita))

# Create iterator Object

class Container:

    def \_\_init\_\_(self, start = 0, end = 0):

        self.start = start

        self.end = end

    def \_\_iter\_\_(self):

        print("[LOG] I made this iterator!")

        return self

    def \_\_next\_\_(self):

        print("[LOG] Calling \_\_next\_\_ method!")

        if self.start < self.end:

            i = self.start

            self.start += 1

            return i

        else:

            raise StopIteration()

c = Container(0, 5)

for i in c:

print(i)

[LOG] I made this iterator!

[LOG] Calling \_\_next\_\_ method!

0

[LOG] Calling \_\_next\_\_ method!

1

[LOG] Calling \_\_next\_\_ method!

2

[LOG] Calling \_\_next\_\_ method!

3

[LOG] Calling \_\_next\_\_ method!

4

[LOG] Calling \_\_next\_\_ method!

3.第三种就是**生成器（generator）**。生成器通过 yield 语句快速生成迭代器，省略了复杂的 \_\_iter\_\_() & \_\_next\_\_() 方式：

def container(start, end):

    while start < end:

        yield start

        start += 1

c = container(0, 5)

print(type(c))

print(next(c))

next(c)

for i in c:

print(i)

<class 'generator'>

0

2

3

4

简单来说，yield 语句可以让普通函数变成一个生成器，并且相应的 \_\_next\_\_() 方法返回的是 yield 后面的值。一种更直观的解释是：程序执行到 yield 会返回值并暂停，再次调用 next() 时会从上次暂停的地方继续开始执行：

def gen():

    yield 5

    yield "Hello"

    yield "World"

    yield 4

for i in gen():

    print(i)

5

Hello

World

4

### 8.8列表解析

语法: [expr for iter\_var in iterable]

扩展版本的语法: [expr for iter\_var in iterable if cond\_expr]

===矩阵样例===

你需要迭代一个有三行五列的矩阵么? 很简单:

>>> [(x+1,y+1) for x in range(3) for y in range(5)]

[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 1), (2, 2), (2,

3) (2 4) (2 5) (3 1) (3 2) (3 3) (3 4) (3 5)]

### 8.9 生成器表达式

生成器表达式是列表解析的一个扩展。

生成器是特定的函数，允许你返回一个值，然后"暂停"代码的执行，稍后恢复。

列表解析的一个不足就是必要生成所有的数据，用以创建整个列表。这可能对有大量数据的迭代器有负面效应。生成器表达式通过结合列表解析和生成器解决了这个问题。

生成器表达式在 Python 2.4 被引入, 它与列表解析非常相似，而且它们的基本语法基本相同; 不过它并不真正创建数字列表, 而是返回一个生成器，这个生成器在每次计算出一个条目后，把这个条目“产生”(yield)出来. 生成器表达式使用了"延迟计算"(lazy evaluation), 所以它在使用内存上更有效. 我们来看看它和列表解析到底有多相似:

列表解析:

[expr for iter\_var in iterable if cond\_expr]

生成器表达式:

(expr for itervar in iterable if condexpr)

两者差别：

**[python]** [view plain](http://blog.csdn.net/gaochizhen33/article/details/45169765) [copy](http://blog.csdn.net/gaochizhen33/article/details/45169765)

1. >>> [i **for** i **in** range(10)]
2. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

注意，列表解析会产生一个新的列表，如上所示。

**[python]** [view plain](http://blog.csdn.net/gaochizhen33/article/details/45169765) [copy](http://blog.csdn.net/gaochizhen33/article/details/45169765)

1. >>> (i **for** i **in** range(10) **if** i % 2 == 0)
2. <generator object <genexpr> at 0xb772cc0c>

如上所示，生成器表达式并不会生成列表，只是生成了一个生成器。我们在使用该生成器的时候，它才会计算相应的值。

### 第九章 文件和输入输出

### 第十章 错误和异常

### 10.3检测和处理异常

try 语句有两种主要形式: try-except 和 try-finally . 这两个语句是互斥的, 也就是说你只能使用其中的一种 . 一个 try 语句可以对应一个或多个 except 子句 , 但只能对应一个 finally 子句, 或是一个 try-except-finally 复合语句.

***处理单个异常***

try-except：

try:

try\_suite # watch for exceptions here 监控这里的异常

except Exception[, reason]:

exceptsuite # exception-handling code 异常处理代码

try:

try\_suite # watch for exceptions here 监控这里的异常

***处理多个异常***

try:

try\_suite # watch for exceptions here 监控这里的异常

except Exception1[, reason1]:

suite\_for\_exception\_Exception1

except Exception2[, reason2]:

suite\_for\_exception\_Exception2

***处理多个异常***

try:

try\_suite # watch for exceptions here 监控这里的异常

except (Exc1[, Exc2[, ... ExcN]])[, reason]:

suite\_for\_exceptions\_Exc1\_to\_ExcN

***捕获所有异常***

如果查询异常继承的树结构, 我们会发现 Exception 是在最顶层的, 所以我们的代码可能看起来会是这样:

try:

try\_suite # watch for exceptions here 监控这里的异常

except Exception, e:

# error occurred, log 'e', etc.

***异常参数***

异常也可以有参数, 异常引发后它会被传递给异常处理器. 当异常被引发后参数是作为附加帮助信息传递给异常处理器的. 虽然异常原因是可选的, 但标准内建异常提供至少一个参数, 指示异常原因的一个字符串.

# single exception

except Exception[, reason]:

suite\_for\_Exception\_with\_Argument

# multiple exceptions

except (Exception1, Exception2, ..., ExceptionN)[, reason]:

suite\_for\_Exception1\_to\_ExceptionN\_with\_Argument

无论 reason 只包含一个字符串或是由错误编号和字符串组成的元组, 调用 str(reason) 总会返回一个良好可读的错误原因. 不要忘记 reason 是一个类实例 - 这样做你其实是调用类的特殊方法 \_\_str\_\_() .

***else 子句***

try:

3rd\_party\_module.function()

except:

log.write("\*\*\* caught exception in module\n")

else:

log.write("\*\*\* no exceptions caught\n")

log.close()

***finally 子句***

finally 子句是无论异常是否发生,是否捕捉都会执行的一段代码

下面是 try-except-else-finally 语法的示例:

try:

A

except MyException: B

else: C

finally: D

### 10.4 上下文管理

***with 语句***

with context\_expr [as var]:

with\_suite

with open('/etc/passwd', 'r') as f:

for eachLine in f:

# ...do stuff with eachLine or f...

这个代码片段干了什么呢...嗯,这是 Python,因而你很可能的已经猜到了.它会完成准备工作,比如试图打开一个文件,如果一切正常,把文件对象赋值给 f.然后用迭代器遍历文件中的每一行,当完成时,关闭文件.无论的在这一段代码的开始,中间,还是结束时发生异常,会执行清理的代码,此外文件仍会被自动的关闭.

### 10.5 \*字符串作为异常

### 10.6 触发异常

raise 语句对所支持是参数十分灵活,对应到语法上就是支持许多不同的格式.rasie 一般的用法是:

raise [SomeException [, args [, traceback]]]

第一个参数,SomeExcpetion,是触发异常的名字.如果有,它必须是一个字符串,类或实例(详见下文).如果有其他参数(arg 或 traceback),就必须提供 SomeExcpetion.Python 所有的标准异常见表10.2.

第二个符号为可选的 args(比如参数,值),来传给异常.这可以是一个单独的对象也可以是一个对象的元组.当异常发生时,异常的参数总是作为一个元组传入.如果 args 原本就是元组,那么就将其传给异常去处理;如果 args 是一个单独的对象,就生成只有一个元素的元组(就是单元素元组).大多数情况下,单一的字符串用来指示错误的原因.如果传的是元组,通常的组成是一个错误字符串,一个错误编号,可能还有一个错误的地址,比如文件,等等.

最后一项参数,traceback,同样是可选的(实际上很少用它),如果有的话,则是当异常触发时新生成的一个用于异常-正常化(exception—normally)的追踪(traceback)对象.当你想重新引发异常时,第三个参数很有用(可以用来区分先前和当前的位置).如果没有这个参数,就填写 None.

最常见的用法为 SomeException 是一个类.不需要其他的参数,但如果有的话,可以是一个单一对象参数,一个参数的元组,或一个异常类的实例.如果参数是一个实例,可以由给出的类及其派生类实例化(已存在异常类的子集).若参数为实例,则不能有更多的其他参数.

raise 语句的用法

rasie 语法 描述

raise exclass 触发一个异常,从 exclass 生成一个实例(不含任何异常参数)

raise exclass() 同上,除了现在不是类;通过函数调用操作符(function calloperator: "()")作用于类名生成一个新的 exclass 实例,同样也没有异常参数

raise exclass, args 同上,但同时提供的异常参数 args,可以是一个参数也可以元组

raise exclass(args) 同上

raise exclass,args, tb 同上,但提供一个追踪(traceback)对象 tb 供使用

Edit By Vheavens

Edit By Vheavens

raise exclass,instance 通过实例触发异常(通常是 exclass 的实例);如果实例是 exclass的子类实例,那么这个新异常的类型会是子类的类型(而不是exclass);如果实例既不是 exclass 的实例也不是 exclass 子类的

实例,那么会复制此实例为异常参数去生成一个新的 exclass 实例.

raise instance 通过实例触发异常:异常类型是实例的类型;等价于 raise

instance.\_\_class\_\_, instance (同上).

raise string (过时的) 触发字符串异常

raise string, args 同上,但触发伴随着 args

raise string, args, tb 同上,但提供了一个追踪(traceback)对象 tb 供使用

raise (1.5 新增)重新触发前一个异常,如果之前没有异常,触发 TypeError.