Python基础教程

第5章 控制语句与作用域

1. Python控制结构

Python支持三种不同的控制结构：

- **if**
- **for**
- **while**

注意：在Python中没有switch和case语句，可通过多重elif来达到相同的效果。

1. Python控制结构

Python中的if由三部分组成：

（1）关键字if本身；

（2）用于判断结果真假的条件表达式；

（3）当表达式为真或者非零时执行的代码块。

if 语句语法如下：

```
if   EXPRESSION1:
        STATEMENT1
elif  EXPRESSION2:
        STATEMENT2
else:
        STATEMENT3
```

2.1 if语句

➢ 多重条件表达式

单个if语句可以通过布尔操作符and、or、not实现多重判断条件或是否定判断条件。

```
if not warn and ( var > 10 ):
    print ( “more than one expression”)
```

2.1 if语句

➢ 单一语句代码块

如果一个复合语句（例如if子句、while或for循环）的代码仅仅包含一行代码，那么它可以和前面的语句写在同一行上：

```
if single_line:
    do_something()
```

2.2 else语句

和其他语言一样，Python提供了与if语句搭配使用的else语句。

如果if语句的条件表达式的结果是False，那么程序将执行else后面的语句。

“悬挂else”：

```
1   if (iValue > 10 )
2       if( fValue> 0.1 )
3           printf("first if expression.\n");
4
5   else
6       printf("second if expression.\n");
```

问题：else属于哪一个if？

2.3 悬挂else

```
1   if (iValue > 10 )
2       if( fValue> 0.1 )
3           printf("first if expression.\n");
4
5   else
6       printf("second if expression.\n");
```

在C语言中，规则是else与其最近的if搭配。虽然是想和最外层的if搭配，但是事实上else属于内部的if，因为C编译器会忽略额外的空白。

分析：语法正确，但是不是预期的执行流程。通过执行结果很容易发现问题所在。但如果有大量的代码嵌入到了类似的框架中，那么修正程序bug将耗费大量精力。

2.3 悬挂else

对于Python，很难出现“悬挂else”这种问题，因为其强制性的缩进语法很容易让程序员决定else所属的if，并且代码也变得更容易阅读。

```
if ival > 1:
    if fval > 0.1:
        print ( "First case" )
    else:
        print ( "Second case" )
```

2.4 elif（else-if）语句

elif是Python的else-if语句，会检查多个表达式是否为真，并在为真时执行特定代码块中的代码。和else一样，elif声明是可选的。

不同的是：if只能有一个else语句，却可以有任意数量的elif语句。

```
1   cmd = 'update'
2   if cmd == 'create':
3       print "create item"
4
5   elif cmd == 'read':
6       print "read item"
7
8   elif cmd == 'update':
9       print "update item"
10
11  else:
12      print "other action"
13
```

2.4 elif（else-if）语句

假如需要判断的条件很多呢？

方法一：多个逻辑运算符一起使用，这也是最常用的写法。

```
if x == 1 or y == 1 or z == 1:
    print ('passed')
```

方法二：使用成员操作符in，比较Pythonic的一种用法。

```
if val in (1, 2, 3):
    print ('passed')
```

2.4 elif（else-if）语句

方法三：使用字典

```
22  actions={
23              "create":"create item",
24              "read"  :"read item",
25              "update":"update item"
26          }
27  default = "other action"
28
29  action = actions.get(cmd,default)
```

2.4 elif（else-if）语句

使用字典示例：

```
>>> people = {
                '乔峰':['降龙十八掌','丐帮'],
                '张无忌':['乾坤大挪移','明教']
            }
>>> name = input("输入大侠的名字: ")
输入大侠的名字: 乔峰
>>> if name in people:
        print ("{}大侠的门派是{}, 绝技是{}.".format(name, people[name][1],people[name][0]))


乔峰大侠的门派是丐帮, 绝技是降龙十八掌.
```

2.5 条件表达式（三元操作符）

C语言中的三元表达式（**C**？**X**：**Y**）：

- 其中**C**是条件表达式；
- **X**是C为真的结果；
- **Y**是C为假的结果。

Python没有三元操作符，类似的语法中确定为：

X if **C** else **Y**。

```
smaller = x if  x < y else y
```

3. While循环

用于在某条件下，循环执行某段程序，以处理需要重复处理的相同任务。

while condition（判断条件）：

执行语句……

执行语句可以是单个语句或语句块。

判断条件可以是任何表达式，任何非零、或非空（null）的值均为True。当判断条件假False时，循环结束。

判断条件可以是个常值，表示循环必定成立，如果常量为布尔真，那么就是“无限循环”。

3.1 计数循环

```
>>> count  = 0
>>> while( count < 9 ):
        print ('the index is:', count)
        count += 1


the index is: 0
the index is: 1
the index is: 2
the index is: 3
the index is: 4
the index is: 5
the index is: 6
the index is: 7
the index is: 8
```

3.2 While无限循环

如果condition永远为真True，无限循环成为可能！

```
while True：
    handle，indata = wait_for_client_connect()
    outdata = process_request( indata )
    ack_result_to_client(handle, outdata)
```

哪些情况可以成为无限循环？

4.1 for语句

for循环一般会访问一个可迭代的对象（如序列或者是迭代器），并且在所有条目都处理过之后结束循环。

for ivalue in (1,2,3)（迭代器）:

print(ivalue)

每次循环， **ivalue** 迭代变量被设置为可迭代对象的当前元素。

建议：能用for循环，就尽量不用while循环！

4.1 for语句

for语句循环的特点：

（1）通常用于遍历序列成员：字符串、列表、元组和字典；

◆三种迭代方式：

➢ 通过序列项迭代

```
1  action_list = ['create','read','update']
2  for action in action_list:
3      print action
```

4.2 for语句-迭代方式

- 通过序列索引迭代

```
5   action_list = ['create','read','update']
6   for index in range( len(action_list) ):
7       print action_list[index]
```

- 通过项和索引迭代

```
9   action_list = ['create','read','update']
10  for index , item in enumerate(action_list):
11      print index,item
```

4.3 range()函数

- 内建函数range( )用法：

  range(start , end, step = 1)

  range( )会返回一个包含所有k的可迭代对象：

  （1）start <= k < end；

  （2）k每次递增step，step不能为0，否则发生错误。

  range( )简略用法：

  range(end)（start默认为 0，step为 1）

  range(start , end)

range()函数

```
>>> range(10)
>>> for v in range(10):
        print (v)


0
1
2
3
4
5
6
7
8
9
>>> list(range(10,0,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

请编写程序计算
1+2+…+1000

for语句

for语句循环特点：

（2）会自动的调用迭代器的next( )方法，捕获StopIteration异常并结束循环；

- **for语句对容器对象调用iter( )函数，iter( )是Python内置函数。**
- **iter( )函数会返回一个定义了next( )方法的迭代器对象，它在容器中逐个访问容器内的元素。**
- **next( )也是Python内置函数。在没有后续元素时，next( )会抛出一个StopIteration异常，通知for语句循环结束。**

```
for x in c:
        statement( s )
```

和下面的代码等价：

```
_temporary_iterator = iter( c )
while True:
        try: x = next( _temporary_iterator )
        except StopIteration: break
        statement( s )
```

for语句-迭代器

- 迭代器是用来帮助记录每次迭代访问到的位置，当对迭代器使用**next( )**函数的时候，迭代器会返回它所记录位置的下一个位置的数据。
- 实际上，在使用**next( )**函数的时候，调用的就是迭代器对象的 **__next__** 方法。

理解：

是一个带状态的对象，能在调用next( )方法的时候返回容器中的下一个值。

任何实现了__iter__和__next__( )方法的对象都是迭代器，__iter__返回迭代器自身，__next__返回容器中的下一个值。

如果容器中没有更多元素了，则抛出StopIteration异常，至于到底是如何实现的这并不重要。

for语句-迭代器

生成无限序列：

```Python
>>> from itertools import count
>>> counter = count(start=13)
>>> next(counter)
13
>>> next(counter)
14
```

从一个有限序列中生成无限序列：

```Python
>>> from itertools import cycle
>>> colors = cycle(['red', 'white', 'blue'])
>>> next(colors)
'red'
>>> next(colors)
'white'
>>> next(colors)
'blue'
>>> next(colors)
'red'
```

4.5 for语句-并行迭代

- 同时迭代两个序列

```
>>> names = [ 'anne', 'beth', 'george', 'damon' ]
>>> ages = [ 12, 45, 32, 102 ]
>>> for i in range( len( names ) ):
        print( names[ i ], ' is ', ages[ i ], 'years old' )
```

```
anne is 12 years old
beth is 45 years old
george is 32 years old
damon is 102 years old
```

4.6 for语句-zip函数

- 内建函数**zip**用来进行并行迭代，可用作多个序列（能够处理不等长，最短原则）

```
>>> list( zip( range( 5 ), range( 100000 ) ) )
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
>>> zip( names, ages )
<zip object at 0x0000024C73834F40>
>>> list( zip( names, ages ) )
[('anne', 12), ('beth', 45), ('george', 32), ('damon', 102)]
>>> for name, age in zip( names, ages ) :
        print( name, ' is ', age, 'years old' )
```

```
anne is 12 years old
beth is 45 years old
george is 32 years old
damon is 102 years old
```

```
>>> names = ( ' Poe ', ' Gaudi ', ' Freud ', ' Poe2 ' )
>>> years = ( 1976, 1987, 1990, 2003 )
>>> for name in sorted( names ):
        print( name )


 Freud
 Gaudi
 Poe
 Poe2
>>> for name in reversed( names ):
        print( name )


 Poe2
 Freud
 Gaudi
 Poe
>>> for i, name in enumerate( names ):
        print( i, name )


0  Poe
1  Gaudi
2  Freud
3  Poe2
```

```
>>> years = ( 1976, 1987, 2020, 2003, 1990
>>> for i in sorted( years ):
        print( i )


1976
1987
1990
2003
2020
>>> for i in reversed( years ):
        print( i )


1990
2003
2020
1987
1976
>>> for i, year in enumerate( years ):
        print( i, year )


0 1976
1 1987
2 2020
3 2003
4 1990
```

5.1 break语句

结束当前循环，然后跳转到下一条语句，类似C中的传统break。需要立即从循环中退出时，break语句可以用在while和for循环中。

```
>>> def lgf1( num ):
        count = int( num/2 )
        while count > 0 :
                if num % count == 0 :
                        print( count, 'is the largest factor of', num )
                        break
                count -= 1
        return count

>>> lgf1( 238595 )
47719 is the largest factor of 238595
47719
```

5.2 else语句与break

他大多数语言中，条件语句范围外不会有**else**语句，在**while**和**for**循环中使用**else**语句。

在循环中使用时，**else**子句只在循环完成后执行，**br**

```
>>> def showMaxFactor( num ):
        count = int(num / 2)
        while count > 1:
                if num % count ==0:
                        print(u'%d 的最大公约数
                        break
                count -= 1
        else:
                print (u'%d 是素数' % num)


>>> for i in range(10, 30):
        showMaxFactor( i )
```

```
10 的最大公约数是: 5
11 是素数
12 的最大公约数是: 6
13 是素数
14 的最大公约数是: 7
15 的最大公约数是: 5
16 的最大公约数是: 8
17 是素数
18 的最大公约数是: 9
19 是素数
20 的最大公约数是: 10
21 的最大公约数是: 7
22 的最大公约数是: 11
23 是素数
24 的最大公约数是: 12
25 的最大公约数是: 5
26 的最大公约数是: 13
27 的最大公约数是: 9
28 的最大公约数是: 14
29 是素数
```

6. continue语句

与其他高级语言中传统的**continue**并没有不同，也用在**while**和**for**循环中。在开始下一次循环前要满足一些先决条件，否则循环正常结束。

```
>>> valid = False
>>> count = 3
>>> while count > 0 :
        inp = input( ' enter password ' )
        # check for valid password
        for eachPasswd in passwordList:
            if inp == eachPasswd:
                valid = True
                break
        if not valid:          #( or valid == 0 )
            print( ' invalid input ' )
            count -= 1
            continue
        else:
            break
```

7. pass语句

Python中pass是空语句，是为了保持程序结构的完整性。

如同C语言中使用分号“;”表示“不做任何事”。

有时候编写代码前把程序逻辑结构定下来，但是不需要做任何事，就可以使用pass语句来进行“占位”。

常用于异常处理。

```
def foo_func( ):
    pass
```

或是

```
if user_choice == 'do_calc':
        pass
else
        pass
```

8.1 断言

if语句有一个“近亲”，工作方式如下：

if not condition:

crash program

```
>>> age = 10
>>> assert 0 < age < 100
>>> age = -1
>>> assert 0 < age < 100
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    assert 0 < age < 100
AssertionError
```

8.2 断言assert存在目的

与其让程序在不可预计的时候崩溃，不如在错误条件出现时让它崩溃！

作为初期测试和调试过程中的辅助条件，在程序中植入检查点。需要确保程序中的某个条件一定为真才能让程序正常工作的话，使用**assert**！

```
>>> age = -1
>>> assert 0 < age < 100, 'The age must be realistic'
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    assert 0 < age < 100, 'The age must be realistic'
AssertionError: The age must be realistic
```

9. Python变量作用域

Python中标识符的作用域是定义其声明在程序里可应用的范围。变量分为全局变量和局部变量。

全局变量的一个特征是除非被删除掉，否则它们一直“存活”到脚本运行结束，且对于所有的函数，它们的值都是可以被访问的。

局部变量仅依赖于定义它们的函数是否被调用，当声明它的函数被调用时，局部变量就会进入声明它们的作用域，一旦函数调用完成，变量就离开作用域。

9.1 Python变量作用域-定义

```
>>> def foo():
        print ('\ncalling foo()...')
        bar = 200
        print ('in foo(),bar is ',bar)


>>> bar = 100
>>> print ('in __main__,bar is ',bar)
in __main__,bar is  100
>>> foo()

calling foo()...
in foo(),bar is  200
>>> print ('\nin __main__,bar is (still)',bar)

in __main__,bar is (still) 100
```

9.1 Python变量作用域-定义

```
>>> bar = 100
>>> def foo():
        global bar
        print ('bar1 = ', bar)
        bar = 200
        print ('bar2 = ', bar)


>>> foo()
bar1 =  100
bar2 =  200
>>> bar = 300
>>> foo()
bar1 =  300
bar2 =  200
```

9.2 Python变量作用域- lambda匿名函数

变量作用域的**lambda**匿名函数，一个**lambda**表达式定义了新的作用域，它和标准函数一样遵循作用域的规则。

```
>>> x = 10
>>> def foo():
        y = 5
        bar = lambda: x+y
        print(bar())


>>> foo()
15
```

9.2 Python变量作用域- lambda匿名函数

若要使外面的局部作用域传递一个变量到内部则需要将**lambda**表达式改为：**bar = lambda y = y : x + y**，即外部的**y**值作为一个参数传入，成为**lambda**函数的局部变量。

```
>>> x = 10
>>> def foo():
        y =5
        bar = lambda y = y: x + y
        print (bar())
        y = 8
        print (bar())


>>> foo()
15
15
```

9.2 Python变量作用域- lambda匿名函数

输出的结果是“错误”（未满足预期）的，因为外部的y值被传入并在lambda中“设置”，虽然稍后改变了，但是lambda的定义没有改变。在lambda表达式中加入对函数局部变量y进行引用的局部变量z。

```
>>> x = 10
>>> def foo():
        y = 5
        bar = lambda z: x+z
        print (bar(y))
        y = 8
        print (bar(y))


>>> foo()
15
18
```

9.3 Python变量作用域-嵌套作用域

嵌套作用域的引入使得作用域不仅仅只是停留在局部作用域和全局作用域的层次。

```
>>> def foo():
        m = 3
        def bar():
            n = 4
            print (m + n)
        print (m)
        bar()


>>> foo()
3
7
```

10. Python 闭包

定义：

在一个内部函数里面对在外部作用域（不是全局作用域）的变量进行引用，则内部函数被认为是闭包。

定义在外部函数内的但由内部函数引用或者使用的变量称为自由变量。

闭包对于在函数对象和作用域中随意地切换是很有用的，在**GUI**或者很多**API**支持回调函数的事件驱动编程中很有用处。

10. Python 闭包

```
>>> def add(x):
        def adder(y):return x+y
        return adder

>>> c = add(8)
>>> type(c)
<class 'function'>
>>> c.__name__
'adder'
>>> c(10)
18
```

adder(y)就是这个内部函数，对在外部作用域（但不是在全局作用域）的变量进行引用：**x**就是被引用的变量，**x**在外部作用域**add**里面，但不在全局作用域里，则这个内部函数**adder**就是一个闭包。

概括起来：闭包 = 函数块 + 定义函数时的环境，**adder**就是函数块，**x**就是环境，这个环境可以有很多，不止一个简单的**x**。

10. Python 闭包

注意：闭包中是不能修改外部作用域的局部变量的。

```
>>> def foo():
        m = 0
        def foo1():
            m = 1
            print (m)
        print (m)
        foo1()
        print(m)


>>> foo()
0
1
0
```

从执行结果可以看出，虽然在闭包里面也定义了一个变量**m**，但是其不会改变外部函数中的局部变量**m**。

10. Python 闭包

```
>>> def counter(start_at = 0):
        count = [start_at]
        def incr():
            count[0] += 1
            return count[0]
        return incr

>>> count = counter(5)
>>> print(count)
<function counter.<locals>.incr at 0x034E5A98>
>>> type(count)
<class 'function'>
>>> print(count())
6
>>> print(count())
7
>>> count1 = counter(100)
```

分析：函数可以作为另一个函数的参数或返回值，可以赋给一个变量。

函数可以嵌套定义，有了嵌套函数这种结构，便会产生闭包问题。

10. Python 闭包

总结：

- 闭包函数嵌套存在于函数体内；
- 闭包函数必须引用外部变量（一般不能是全局变量），不一定要return；
- 闭包函数必须作为对象被逐级return，直至作为主函数（最外层函数）的返回值。

用途：

（1）当闭包执行完后，仍然能够保持住当前的运行环境；

（2）闭包可以根据外部作用域的局部变量来得到不同的结果，类似配置功能的作用，可以修改外部的变量，闭包根据这个变量展现出不同的功能。

- 需要对某些文件的特殊行进行分析，先要提取出这些特殊行。

思考

1.While循环和for循环特点?
2.什么时候需要死循环?
3.如何区分全局和局部变量作用域?
4.闭包是什么？如何使用闭包?