第二章流程控制

> 本文是Python通用编程系列教程，已全部更新完成，实现的目标是从零基础开始到精通Python编程语言。本教程不是对Python的内容进行泛泛而谈，而是精细化，深入化的讲解，共5个阶段，25章内容。所以，需要有耐心的学习，才能真正有所收获。虽不涉及任何框架的使用，但是会对操作系统和网络通信进行全局的讲解，甚至会对一些开源模块和服务器进行重写。学完之后，你所收获的不仅仅是精通一门Python编程语言，而且具备快速学习其他编程语言的能力，无障碍阅读所有Python源码的能力和对计算机与网络的全面认识。对于零基础的小白来说，是入门计算机领域并精通一门编程语言的绝佳教材。对于有一定Python基础的童鞋，相信这套教程会让你的水平更上一层楼。

一 运算方式

1. 数学运算

既然我们编程的目的是为了控制计算机能够像人脑一样工作，那么人脑能做什么，就需要程序中有相应的机制去模拟。人脑无非是数学运算和逻辑运算，对于数学运算就是加减乘除，很简单，我们先来看一下。计算机的核心部件就是CPU，CUP有两个功能，控制和运算，接下来就看一下计算机是如何进行运算的，关于计算的控制功能会在第五阶段有详细的说明。

```
a = 10
b = 20
print(a + b)   # 加 （在python中，#可用来添加注释，即#后面的内容计算机时不会管你的，
可用来给代码添加注释，即解释该代码）
print(a - b)   # 减
print(a * b)   # 乘
print(a / b)   # 除
print(a % b)   # 取模
print(a ** b)   # 幂
print(a // b)   # 取整除
```

2. 比较运算

```
a = 10
b = 20
print(a == b)
print(a != b) # 不等于
# print(a <> b)  # 弃用
print(a > b)
print(a < b)
print(a >= b)
print(a <= b)
```

3. 赋值运算

```
a = 10
b = 20
a = b # 将b赋值给a
print(a)
a += b # 效果等同于a = a + b，即将a+b的值赋给a，下同
print(a)
a *= b
print(a)
a %= b
print(a)
a **= b
print(a)
a //= b
print(a)
```

4. 逻辑运算

```
print(True and False) # 与运算，True和False进行与运算结果应该为False
print(True or False) # 或运算，True和False进行或运算结果应该为True
print(not (True and False))#非运算，将结果置非，即True变False，False变True
```

5. 身份运算

```
Python 3.7.2 (default, Dec 29 2018, 00:00:04)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 'abhdskjLhfafh3417469ahflkgaghf4980185357843.vd/f,/cmbcmv,.bxj;lfgow385682dsjklbvaskc'
>>> b = 'abhdskjLhfafh3417469ahflkgaghf4980185357843.vd/f,/cmbcmv,.bxj;lfgow385682dsjklbvaskc'
>>> a is b
False
>>> a == b
True
>>>
```

```
# is比较的是id，在python中，id为变量的内存地址，id相同意味着在内存中保存的地址也相同，即为同一个变量
# 而==比较的是值
```

二 分支语句

1. if，else

对于逻辑运算，即人根据外部条件的变化而做出不同的反映，比如
如果：一切都是天意，那么：谁也逃不离

```
if "everything" is "God's will":
    print('everyone can not flee')
```

或许这个例子太抽象了，我们再举一个简单的例子：身高，体重和视力都合格才可以做飞行员。

```
height = 183
weight = 75
vision = 5.0
# 只有同时满足三个条件才可以开飞机
if height == 183 and weight == 70 and vision > 5.0:
```

```
    print('可以做飞行员')
else:
    print('不可以')
```

2. if循环嵌套使用

```
height = 183
weight = 75
vision = 5.0
# 1 只要视力低于5.0就不能飞
if not vision >= 5.0:  # 为了试一下not的使用，当然也可以写成vison < 5.0
    print('视力太差，上天无望')
else:
    # 2 身高是硬伤，如果不达标就放弃吧，孩子
    user_height = input('请输入身高')
    user_height = int(user_height)
    if user_height == 183:
        # 3 再来判断用户体重
        user_weight = input('请输入体重')
        user_weight = int(user_weight)
        if user_weight == 75:
            print('恭喜你，你可以上天了')
        else:
            print('体重不合适，建议增重或减重后再来尝试')
    else:
        print('身高无望')
```

3. if，elif，else 使用

如果：成绩>=90，那么：优秀
如果成绩>=80且<90,那么：良好
如果成绩>=70且<80,那么：普通
其他情况：很差

```
score=input('>>: ')# 当运行到input()时，程序会暂停运行，并需要你在命令行中输入信息，
并将输入的信息以str的方式给到该函数的返回值，如果你输入50，
那么score就等于"50"（注意引号哦，表示str类型）
score=int(score)# 转化为int类型
if score >= 90:
```

```
    print('优秀')
elif score >= 80:
    print('良好')
elif score >= 70:
    print('普通')
else:
    print('很差')
```

使用模板

```
if 条件1:
    缩进的代码块
elif 条件2:
    缩进的代码块
elif 条件3:
    缩进的代码块
else:
    缩进的代码块
```

三 循环语句

1. while 循环

(1) while循环(条件循环)

为什么要使用循环，先来看下面一段代码

```
albert_age = 18
guess = int(input(">>:"))
if guess > albert_age :
    print("猜的太大了，往小里试试...")
elif guess < albert_age :
    print("猜的太小了，往大里试试...")
else:
    print("恭喜你，猜对了...")
#第2次
guess = int(input(">>:"))
if guess > albert_age :
```

```
    print("猜的太大了，往小里试试...")
elif guess < albert_age :
    print("猜的太小了，往大里试试...")
else:
    print("恭喜你，猜对了...")
#第3次
guess = int(input(">>:"))
if guess > albert_age :
    print("猜的太大了，往小里试试...")
elif guess < albert_age :
    print("猜的太小了，往大里试试...")
else:
    print("恭喜你，猜对了...")
```

毫无疑问，这是一段low逼的代码。。。。。。因为重复的代码太多了，这种情况的发生是因为有些场景下我们需要反复验证某种情况，比如验证用户名和密码时，如果用户输入的用户名和密码不匹配，我们一般会反复提醒用户输入用户名和密码，直到输入正确为止。那么我们无法判断用户在输入几次之后能够输入正确，上述代码格式也就不能使用了。这种情况我们完全可以使用while 条件循环来处理，while循环又叫做条件循环，既满足条件才会执行，语法如下：

```
albert_age = 18  # albert才18岁？OMG~
guess = int(input(">>:"))
while guess != albert_age:  # 如果猜的年龄和实际年龄不符
  if guess > albert_age :
    print("猜的太大了，往小里试试...")
  elif guess < albert_age :
     print("猜的太小了，往大里试试...")
  guess = int(input(">>:"))
print("恭喜你，猜对了...")
```

```
while 条件:
    # 循环体
    # 如果条件为真，那么循环体则执行，执行完毕后再次循环，重新判断条件。
    # 如果条件为假，那么循环体不执行,循环终止
```

注意：满足条件就是指条件为真，一般我们会用True直接表示条件为真，或者使用下面小练习的 count <= 10 的运算判断的形式，如果表示条件为假可以这样表示：

```
False    None    0    ""    ()    []    {}
# 总结一下：False，0或者空都为False
```

(2) while循环小练习

```
#打印0-10
count=0 # count的初始值为0
while count <= 10:# 如果count小于等于0
    print('loop',count) # loop1 loop2 ...
    count+=1# 前面提过的，该行等价于count = count + 1
#打印0-10之间的偶数
count=0
while count <= 10:
    if count%2 == 0: # ==用来比较左右两边的值是否相等，count%2如果等于0，表示count为偶数
        print('loop',count)
    count+=1
#打印0-10之间的奇数
count=0
while count <= 10:
    if count%2 == 1:
        print('loop',count)
    count+=1
```

(3) while循环之死循环

死循环就是会一直执行的循环，因为条件一直成立

```
import time
num=0
while True:
    print('count',num)
    time.sleep(1)# 让程序在这里睡（暂停）1秒
    num+=1
```

(4) while循环嵌套与tag

tag只是一个变量，不过他是布尔类型，只有True和False，你也可以写成0或者1，当我们有多层循环的时候，使用tag可以迅速退出所有循环

```
tag=True
    while tag:
        ......
        while tag:
            ........
            while tag:
                tag=False # 循环逐层判断，当tag为false时，循环会逐层退出
```

(5) while循环break与continue

这是理解的重点，在后续的购物车、网盘等python练习中会经常出现，也是python等编程语言比较重要的语法知识。

```
#break用于退出本层循环
while True:
    print "123"
    break
    print "456"

#continue用于退出本次循环，继续下一次循环
while True:
    print "123"
    continue
    print "456"
```

所以上面猜年龄的程序可以这样改写

```
albert_age = 18
while True:
    guess = int(input(">>:"))
    if guess > albert_age :
        print("猜的太大了，往小里试试...")
    elif guess < albert_age :
        print("猜的太小了，往大里试试...")
    else:
```

```
        print("恭喜你，猜对了...")
        break  # 用户猜对的时候退出循环
```

(6) while与else组合使用

与其它语言else 一般只与if 搭配不同，在Python 中还有个while ...else 语句，while 后面的else 作用是指，当while 循环正常执行完，中间没有被break 中止的话，就会执行else后面的语句。

```
count = 0
while count <= 5 :
    count += 1
    print("Loop",count)
else:
    print("循环正常执行完啦")
print("-----out of while loop ------")

"""
输出
Loop 1
Loop 2
Loop 3
Loop 4
Loop 5
循环正常执行完啦
-----out of while loop ------
"""
# 如果执行过程中被break啦，就不会执行else的语句啦
count = 0
while count <= 5 :
    count += 1
    if count == 3:break
    print("Loop",count)
else:
    print("循环正常执行完啦")
print("-----out of while loop ------")

"""
输出
Loop 1
Loop 2
```

```
-----out of while loop ------
"""
```

2. for循环

(1) for循环(迭代循环)

for循环是迭代式循环，for 遍历 被循环中的每一项内容，比如下面的代码，for会循环遍历range(10)中的每一个元素，即0， 1， 2， 3...9，语法如下：

```
for i in range(10):
    缩进的代码块
```

说明：

- 其中i为迭代出来的一个个对象，i只是一个变量名，可以任意
- 关键字for 和 in是必须的
- range(10) 是一个被迭代的对象，只要能存多个值，他就可以被迭代，你直接写一个列表也是一样的
- 迭代循环可以理解被迭代的对象就是一个老母鸡，她肚子里有的是蛋，迭代出来的对象就是蛋

计算0到9的和：

```
sum_value = 0  # 将和的初值置0
for i in range(10):
    sum_value += i  # 第三次出现啦！再不会的话~
print(sum_value)
```

(2) break与continue（同上while循环）

(3) for循环与else连用

我们常常会在for循环遍历一个序列或者字典后,接着语句else,很多新手会误以为是判断执行else后面的代码,其实不然,for循环里面也不存在判断,而已执行完遍历的对象后,再执行else后面的代码。

```
# 实例1
L = [1,2,3,4,5,6,7,8,9,10]
for i  in L:
    print(i)
else:
    print('ending')
"""
#输出
>>>
1
2
3
4
5
6
7
8
9
10
ending
"""

# 实例2

dir = {1:'kobe',2:'LBJ',3:'CP3',4:'TDK'}# 字典格式，1 2 3 4为键，Kobe LBJ...为键所对应的值
for k,v in dir.items():
    print (k,v)
else:
    print('ending')

"""
#输出
1 kobe
2 LBJ
3 CP3
4 TDK
ending
"""
```


四 流程控制语句用法说明

1. 避免多层分支嵌套

如果这篇文章只能删减成一句话就结束，那么那句话一定是“要竭尽所能的避免分支嵌套”。过深的分支嵌套是很多编程新手最容易犯的错误之一。假如有一位新手程序员写了很多层分支嵌套，那么你可能会看到一层又一层的大括号：`if: if: if: ... else: else: else: ...` 俗称“*嵌套 if 地狱（Nested If Statement Hell）*”。如果能够避免的话，尽可能用其他的方式代替，这种多层嵌套非常不利于代码的可读性，尤其是当一个 if 分支下代码的量比较多的时候。

2. 封装那些过于复杂的逻辑判断

如果条件分支里的表达式过于复杂，出现了太多的 `not/and/or`，那么这段代码的可读性就会大打折扣，这时我们可以把他拆解，或者先用not 的形式取反，取反的意思即为原来值为True,not后为False，原来值为False,not后为True。

3. 留意不同分支下的重复代码

重复代码是代码质量的天敌，而条件分支语句又非常容易成为重复代码的重灾区。所以，当我们编写条件分支语句时，需要特别留意，不要生产不必要的重复代码。当你编写分支代码时，请额外关注由分支产生的重复代码块，如果可以简单的消灭它们，那就不要迟疑。

4. 使用“德摩根定律”

在做分支判断时，我们有时候会写成这样的代码：

```
# 如果用户账户没有余额或者用户透支额度，拒绝用户购买
# 以下是伪代码
if not “没有余额” or not “有透支额度”:
    print("拒绝用户购买")
```

第一眼看到代码时，是不是需要思考一会才能理解它想干嘛？这是因为上面的逻辑表达式里面出现了 2 个 `not` 和 1 个 `or`。而我们人类恰好不擅长处理过多的“否定”以及“或”这种逻辑关系。这个时候，就该 德摩根定律 出场了。通俗的说，德摩根定律就是 `not A or not B` 等价于 `not (A and B)`。通过这样的转换，上面的代码可以改写成这样：

```
if not （“没有余额” and “有透支额度”）:
    print("拒绝用户购买")
```

5. 在条件判断中使用 all() / any()

`all()` 和 `any()` 两个函数非常适合在条件判断中使用。这两个函数接受一个可迭代对象，返回一个布尔值，其中：

- `all(seq)`：仅当 `seq` 中所有对象都为布尔真时返回 `True`，否则返回 `False`
- `any(seq)`：只要 `seq` 中任何一个对象为布尔真就返回 `True`，否则返回 `False`

假如我们有下面这段代码：

```
def all_numbers_gt_10(numbers):
    # 仅当序列中所有数字大于 10 时，返回 True
    if not numbers: # 如果numbers为空，因为在这里numbers代表一个列表，[1, 2, 3...]这种格式，
        在列表中，空列表[]为False,这行代码就用来判断numbers是否为空，为空就返回False
        return False
    for n in numbers:# 遍历numbers中的每一个元素
        if n <= 10:
            return False# 如果有元素小于等于10，该函数马上返回False
    return True# 如果numbers列表中的所有元素都大于10，那么返回True
```

如果使用 `all()` 内建函数，再配合一个简单的生成器表达式，上面的代码可以写成这样：

```
def all_numbers_gt_10_2(numbers):
    return bool(numbers) and all(n > 10 for n in numbers)
```

简单、高效，同时也没有损失可用性。

6. 使用 try/while/for 中 else 分支

让我们看看这个函数：

```
def do_stuff():
    first_thing_successed = False
    try:
        do_the_first_thing()# 做第一件事
        first_thing_successed = True# 第一件事成功了，把标志位置为True
    except Exception as e: # 如果上面两行代码（try中的两行代码）有错误，第一件事没有成功，执行下面语句
        print("Error while calling do_some_thing")
        return
    # 仅当 first_thing 成功完成时，做第二件事
    if first_thing_successed:
        return do_the_second_thing()
```

在函数 `do_stuff` 中，我们希望只有当 `do_the_first_thing()` 成功调用后*（也就是不抛出任何异常）*，才继续做第二个函数调用。为了做到这一点，我们需要定义一个额外的变量 `first_thing_successed` 来作为标记。其实，我们可以用更简单的方法达到同样的效果：

```
def do_stuff():
    try:
        do_the_first_thing()
    except Exception as e:
        print("Error while calling do_some_thing")
        return
    else:
        return do_the_second_thing()
```

在 `try` 语句块最后追加上 `else` 分支后，分支下的`do_the_second_thing()` 便只会在 try 下面的所有语句正常执行（也就是没有异常，没有 return、break 等）完成后执行。类似的，Python 里的 `for/while` 循环也支持添加 `else` 分支，它们表示：当循环使用的迭代对象被正常耗尽、或 while 循环使用的条件变量变为 False 后才执行 else 分支下的代码。

7. 与 None 值的比较

在 Python 中，有两种比较变量的方法：`==` 和 `is`，二者在含义上有着根本的区别：

- `==`：表示二者所指向的的值是否一致
- `is`：表示二者是否指向内存中的同一份内容，也就是 `id(x)` 是否等于 `id(y)`

`None` 在 Python 语言中是一个单例对象，如果你要判断某个变量是否为 None 时，记得使用 `is`而不是 `==`，因为只有 `is` 才能在严格意义上表示某个变量是否是 None。
否则，可能出现下面这样的情况：

```
class Foo(object):
    def __eq__(self, other):# 魔法方法，当该类在==判断中做左值时触发
        return True


foo = Foo()
print(foo)
print(foo == None)
print(foo is None)
```

在上面代码中，Foo 这个类通过自定义 `__eq__` 魔法方法的方式，很容易就满足了 `== None` 这个条件。
所以，当你要判断某个变量是否为 None 时，请使用 `is` 而不是 `==`。

8. 留意 and 和 or 的运算优先级

看看下面这两个表达式，猜猜它们的值一样吗？

> (True or False) and False
> True or False and False

答案是：不一样，它们的值分别是 `False` 和 `True`，你猜对了吗？问题的关键在于：`and` 运算符的优先级大于 `or`。因此上面的第二个表达式在 Python 看来实际上是 `True or (False and False)`。所以结果是 `True` 而不是 `False`。在编写包含多个 `and` 和 `or` 的表达式时，请额外注意 `and` 和 `or` 的运算优先级。即使执行优先级正好是你需要的那样，你也可以加上额外的括号来让代码更清晰。