# 基本语法文档(五)

*Author: limzh*

## Section1. 序言

本节我们学习递归、循环和条件语句

## Section2. 条件控制

所谓条件控制，简单说就是`if...elif...else`语句的使用.

比如说我们要去判断一个数是否大于10，如果是，则再次判断它是否大于20,如果大于20，则返回1，如果小于20则返回0，如果小于10则返回2.

```
def check(num):
    if num >= 10:
        if num > 20:
            return 1
        elif num <= 20:
            return 0
    else:
        return 2
```

通过对真值条件的判断来决定进行哪些流程的进行，就是所谓的条件控制。

## Section3. 循环遍历

试想，假如我要计算从1加到20的和，那应该怎样编写代码呢？一种自然的方式是对它进行数学优化，使用等差数列求和公式可以很容易地计算。但是如果是要对一个长度为10000的数组的每一个元素进行处理呢？比如，现有一个长度为10000的整数数组，我要去对里面每一个元素进行判断，如果该元素值为1则使其加一，如果不为1则不进行处理。

```
# 假若mylist为长度为10000的整数数组
if mylist[0] == 1:
    mylist[0] += 1
if mylist[1] == 1:
    mylist[1] += 1
#...省略很多相似的操作
if mylist[9999] == 1:
    mylist[9999] += 1
```

自然我们可以将这种操作实现10000遍，但是非常繁琐。不难发现，上面这种操作仅仅是对列表的index进行遍历，对每一个元素对进行检查和相同的处理。在这里我们引入循环遍历的概念用以简化类似的操作.

```
for i in range(10000):
    if mylist[i] == 1:
        mylist[i] += 1
```

还有另外一种while循环的实现，

```
i = 0
while(i != 10000):
    if mylist[i] == 1:
        mylist[i] += 1
```

在这里为止，循环遍历的基础语法和使用方法就已经介绍结束了。下面是介绍高级特性时间。

## 3.1 `for...in...`关键字的作用

`for...in...` 关键字是固定搭配，也是一种`语法糖`。但是它们和`range(...)`并不是一种固定搭配，仅仅只是因为在Python中这样使用可以做到从0遍历到某个传入的整数的效果，才慢慢约定成这样做。还是上面那个遍历数组的例子，一种替代的做法是:

```
i = 0
for k in mylist:
    if mylist[i] == 1:
        mylist[i] += 1
    i += 1
```

在这里，`for...in...`真正做的事情是遍历一个`可遍历对象`的所有元素。

`range(...)`函数返回一个0到...的可遍历对象，我们不妨暂时浅显地认为，这是一个类似于列表的可遍历对象。`range(...)`生成并返回一个包括0到9999整数的类似列表的对象。`for...in...`用以遍历.其实可以用自然语言相对容易地翻译过来,`for i in xxx:`就是`对于在xxx中的元素i,我们进行如下操作:`只不过这个i是顺序遍历的而已。这也不难理解为什么`for k in mylist:`也是合法的遍历方式，因为mylist 就是一个合适的可遍历列表。

## 3.2 `for...in...`中的变量引用机制：深浅复制

考虑这样语法的有效性，

```
mylist = [[],[],[],[]]
k = 0
for i in mylist:
    i.append(k)
    k += 1
```

请问最后mylist的结果是什么呢？答案是：

```
mylist =[[0],[1],[2],[3]]
```

考虑这样语法的有效性，

```
mylist = [0,1,2,3]
for i in mylist:
    i += 1
```

请问最后的mylist的结果是什么?答案是:

```
mylist = [0,1,2,3]
```

所以要注意的第一点，实质上每一次遍历进行了一次变量的复制引用，浅复制和深复制的认识在这里仍然有效。

### 3.3 `for...in...`中的变量引用机制：多变量赋值

我们知道，在同一行我们可以进行多个变量的赋值

```
x,y = 1,2 # 有效
x,y = (1,2)# 有效，结果同上
x,y = (1,2,3) # 无效，左侧接收的变量数小于右边释放的值的数目
x,y = [1,2] # 有效，只是将元组换成了列表
def func():
    return 1,2
x,y = func() # 有效，return返回的值的数目和接受值的变量数目一致
print(x, y)# 有效，输出12
```

因为`for...in...`的本质上就是迭代地将一个可迭代对象的内容赋值给变量的过程，因此自然可以进行多变量的赋值.

请看下面这个例子：

```
mylist = [(1,2),(2,3)]
for i in mylist:
    print(i)
```

```
mylist = [(1,2),(2,3)]
for i, j in mylist:
    print(i, ' ', j)
```

以上两段代码的的输出分别是什么呢？请自己尝试。

## Section4. 递归

递归是函数的自身调用，同时也是一种重要的算法思想的实现方式，在学习递归的过程中我们需要回答四个问题：

1. 递归解决问题的算法思想是什么？
2. 递归的实现结构是什么，也即，实现递归所必须包含的要素是什么？
3. 递归存在的必要性是什么？
4. 递归的不足是什么？

我们用一个著名的斐波那契数列的例子来做引入，

### Question:

> 已知斐波那契数列为1,1,2,3,5,8,13,21,34,55...
>
> 其迭代公式为: f(n) = f(n-1) + f(n-2), n代表数列的第几项
>
> 求斐波那契数列的第1000项的值。

考虑这段代码的有效性:

```
def f(x):
    if x <= 2:
        return 1
    return f(x-1) + f(x-2)
print(f(10))
```

同时考虑这段代码的有效性:

```
def f(n):
    x, y = 1, 1
    for i in range(n-2):
        tmp = x + y
        x = y
        y = tmp
    return y
print(f(10))
```

我们也可以将第二段代码写成:

```
def f(n):
    x, y = 1, 1
    for i in range(n-2):
        x, y = y, x + y
    return y
print(f(10))
```

这种写法和第二段等效，是一种语法糖包装，这和多变量赋值的机制有关。在这里稍微停顿，补充介绍一下。

## （补充） 多变量赋值的机制

```
x, y = 1, 2
```

其实并不等效于

```
x = 1
y = 2
```

而是等效于

```
tmp1 = 1
tmp2 = 2
x = tmp1
y = tmp2
```

这是是为了逻辑上的合理，当我们使用上述多变量赋值的时候，我们的逻辑是x 和 y **同时** 被赋予1 和 2，而并不应该有先后顺序的差别。但是实际执行上必须要有先执行谁，后执行谁的问题，于是就使用中间变量的方式来消除顺序的差异。这也使得在Python中，交换两个变量的值这种操作变得非常容易，如:

```
x, y = y, x
```

# （继续） 递归的介绍

两段代码输出的结果都是正确有效的，但是实现的思路却很值得玩味，简单地说： 递归函数通过不断调用自己，从终点处出发，按照迭代公式把大问题分割成若干个子问题，再不断递归子问题，直到子问题可以被直接解决。当子问题被解决后，再从底部（最基本的子问题）不断聚拢起来大问题的解，直到解决我们想要解决的最大的问题。而第二段代码是从起点出发，按照迭代公式把子问题聚拢成大问题的解，再不断聚拢更大问题的解，直至获得我们想要的结果。

我们会想问：第二段代码和第一段代码的关系是什么呢?

请略作思考，我把我的理解附在下面：

> 递归的方法的分为两部分，一部分将问题分解，另一部分将子问题的解聚拢。第二段代码仅将子问题的解聚拢。因此，第二段代码反映了第一段代码子问题解聚拢的过程。

递归方法和第二段代码方法谁所耗费时间最少呢?

> 不难发现，是第二段代码。

然而，我们肯定又想问：既然第二段代码和递归方法都有效，并且前者花费时间更少，那么递归存在的意义是什么?

对于这个问题，我们暂时按下不表。当完全将本节介绍完，这个问题的答案会自动浮现在我们面前。

我们回过头来，解决一下之前留下的四个问题：

## 4.1 递归的思想是什么?

实际上，递归不是一种思想，递归是一种算法思想具体的函数实现。它背后的思想是：`分治法(Divide and conquer)`

正如我们通过解决斐波那契数列问题所洞察到的那样，递归是从大问题出发，先将大问题分割成若干个子问题(通过迭代公式： `f(n) = f(n - 1) + f(n - 2)`， 在本题中，这是两个子问题。） 然后不断递归子问题，直到将子问题分割到一个直接得到答案的最基本子问题单元（在本题中，当n为1 或 2时，f(n)直接返回1.）。这是分治法的`分`的部分。

当我们得到最基本子问题的解之后，再不断地将子问题的解以某种规则聚拢起来，得到稍大一些的问题的解。在本题中，这种规则为`+`。稍大一些的问题的解又聚拢成更大问题的解，直到获取我们要解决的最终问题的解。这是分治法`治`的部分。简单理解成，解的聚拢。

这种分治思想可以得到优雅的实现，这种实现我们称之为递归。

# 练习：

```
def f(n):
    if x <= 2:
        return 1
    return f(n-1) + f(n-2)
f(10)
```

请尝试描述一下该代码的执行过程，并比对一下`分治法`思想，看哪些过程对应分，哪些过程对应治。

## 4.2 递归实现的要素是什么?

这个问题实际上是在问，要运用分治法思想解决问题，我们应当从问题中提取出哪些信息?

分治法无非分合两步，解决一个问题我们首先要知道如何分（分成怎样的子问题就可以得到问题的解？同时要注意，分出的子问题要和大问题有相同的结构，只是规模减小。），分到什么程度为止（分到一眼就可以看出解的子问题为止，因此这个问题又变成了：什么是问题的最基本子问题？）。这自然提炼出两个分的要素：

1. 分的规则 。在斐波那契问题里，由`f(n) = f(n-1)+f(n-2)`知，`f(n-1),f(n-2)`的解足以构成`f(n)`的解，故而分的规则就是将`f(n)`分为`f(n-1)`和`f(n-2)`
2. 分的边界条件（最基本子问题）在实例问题中，我们知道当问题被分到`f(1)`或`f(2)`的时候，就可以直接返回1作为该子问题的答案，因此`f(1)`或`f(2)`就是最最基本子问题，分到这里分就可以停止了。

分完就要合并，如果子问题的解不能充分解决大问题的解，那么分治法思想的链条断裂，解题无效。这也就要求我们提炼出合的要素：

3. 合的规则。在斐波那契问题里，由`f(n) = f(n-1) + f(n-2)`知，子问题合并到大问题的方式是将解相加。故而合的规则就是`+`.

以上，分治思想解题步骤完成。

在实际过程中，分的规则和合的规则往往会以迭代公式的形式同时出现。因为当我们考虑如何分的时候，其实必须考虑怎么分才能保证子问题的解充分解决大问题（本质上，充分解决大问题本身就体现了合的规则），也要保证子问题和大问题有相同的问题结构。

故而，实际编程递归函数的实现要素即为：

1. 自身调用更小规模输入：分的规则
2. 在函数开头设置边界停止条件：分的最基本子问题
3. 返回一个按某种规则组合的自身调用的值：合的规则

## 4.3 递归的意义是什么?

现在我们回到按下不表的问题，如果说从起点出发开始循环的方法比分治法方法更有效（指节省时间），那么分治法（递归）的意义是什么呢？ 我们还是从一个例子考虑，

### Question:

> 考虑ane要走上一段台阶，她一次既可以走一节台阶，又可以走两阶台阶。请问她走到第N阶台阶所耗费的最小步数是多少?

## 练习:

1. 请尝试编程，既使用递归方法又使用一般方法。（N = 3， 5， 10都请作尝试， 一般方法如果想不到也不要紧，主要实现分治法的递归函数）.
2. 请思考并尝试解答为什么递归在这个问题上表现地非常突出?

提示：按照递归所需要素从问题中提炼信息。问自己如下问题：如何将大问题分成子问题，以保证子问题的解可以充分地解决大问题，同时子问题还有着同大问题一样的问题结构？如何将子问题的解合在一起得到大问题的解？边界条件是什么？参考答案附在`day6/steps.py`中，也请注意注释规范和命名规范，写出有你自己习惯的优雅代码。

## 4.4 递归的不足是什么?

递归最大最大的问题是占用大量内存空间和时间。请尝试用`steps.py`的问题解决N = 1000的情形，体会一下当数据规模很大的时候，递归所消耗的时间和电脑的空间。至于为什么它会消耗这么多时间，占用如此多空间，这个按下不表。当我们进入数据结构的学习的时候，会透彻地理解它。如果你学有余力，可以去了解一下树这种数据结构。并可以看一下这个网址： ***上台阶问题的三种算法实现***