Meta-Python

我们脱离 Python 中关于 OO（或者说关于类/继承/多态）的话题，从函数开始走起，来深入 Python 的内容。

这其中可能会引用或者讲述一些其他语言的特性，跟主题无关的具体细节我们会避开，不过我会留下部分资源做为参考。

（一） 函数

函数是这么一个东西：你给他一点点输入，它就会返回给你一点点**输出**，同时还可能会做一点点其他的东西。

这里所指的输出是大多数编程语言中的 `return` 语句，而不是使用 ***stdout*** 或者图形接口给展示出来的输出内容。

本来函数就是这么简单的，可是我们总会忍不住在他里面做一些别的事情。

比如，对函数外围变量的修改和一些系统级别的调用（专业用语叫做 **Side-Effect**）。我们抛开这些话题不谈（当然，`print` 这种东西还是会用到的，不然连追溯执行流程都不太方便了）。

好的，那我们从一个简单的函数起步。

$$fib(n)=\begin{cases}1. & n=0,1\\ fib(n-1)+fib(n-2). & n>=2\end{cases}$$

很不幸，这个函数是**递归定义的**。结果就会导致我们写起来要蛋疼一阵子。

```python
def fib(n):
    if n==0 or n==1:
        return 1
    else:
        a = 1
        b = 1
        c = 0
        for i in range(2,n+1):  #just like for(i = 2; i<=n ; ++i)
            c = a+b
            a = b
            b = c
        return c


def fib_2(n):
    if n==0 or n==1:
        return 1
    else:
        a,b = 1,1
        for i in range(2,n+1):
            a,b = b,a+b
        return b
```

刚刚入门 Python 的同学大概能够写出第一段代码（`fib`），而一个了解并熟练掌握 Python 的并行赋值的同学写出的第二段代码就要在一定程度上简洁许多。

但是，既然是在讲函数，我们就一定要让函数更加纯粹一些。少做那些 `side-effect`，

而且，尽可能少的使用变量和赋值。

毕竟 Python 也支持定义**递归**函数。

所谓递归函数，就是指函数的定义体中，有直接或间接的调用自身的形式出现。一层层的调用类似于一个**递推**的过程，得出结果后再一层层的返回，则是一个**回归**的过程。所以中文称之为递归是一个绝佳的翻译。

递归版本如下：

```
23
24  def fib_3(n):
25      if n==0 or n==1:
26          return 1
27      else:
28          return fib_3(n-1) + fib_3(n-2)
29
```

跟原数学函数惊人的相似。当然，本来计算机科学就受数学影响颇深，这样的情况自然是不可避免的。

一般来说，代码除了优雅可读之外还要能不失效率才行。而 fib_3 这段代码就只剩下优雅了。如果你有耐心可以一步步的展开 fib_3(18)的调用树，会应该会有一些发现的。

解决效率这个问题有许多方法，当然，我们还是先讲最靠主题的那个。

尾递归。

尾递归仍然是一种递归，其区别在于相较于普通的递归，尾递归的返回结果就是一个值或者是具有返回值的简单函数调用。

尾递归版本的 fib：

```
30  def fib_helper(n,v,c):
31      if n == 0 or n == 1:
32          return c
33      else:
34          return fib_helper(n-1,c,v+c)
35
36  def fib_4(n):
37      return fib_helper(n, 1, 1)
```

或者更简单一点：

```
39  def fib_5(n,v=1,c=1):
40      if n == 0 or n == 1:
41          return c
42      else:
43          return fib_5(n-1,c,v+c)
```

第四个版本（fib_4）借助了一个 fib_helper（其实也可以看作 fib_helper 来借助 fib_4 来设置默认参数）实现 fib 的功能，简化之后就是 fib_5 的样子。至于为什么在递归的时候使用 n-1、c 和 v+c，我们看继续看一下：

```
44
45  def fib_6(n,v=1,c=1):
46      if n == 0 or n == 1:
47          return c
48      else:
49          n,v,c = n-1,c,v+c
50          return fib_6(n,v,c)
51
```

```
52  def fib_7(n):
53      v,c = 1,1
54      while 1:
55          if n == 0 or n == 1:
56              return c
57          n = n-1
58          v,c = c,v+c
```

第六个版本 fib_6 比第五个多出了一条赋值语句，于是递归调用的形参和实参名就变得一致。这样的尾递归我们称之为**严格的（strict）尾递归**。一个严格的尾递归可以可以看成一个拥有一定的退出条件的反复执行函数体的死循环。于是，我们可以继续把他简化成 fib_7 这种形式。

我故意把 fib_7 最后两个赋值语句分开来写，如果你把 v 和 c 再分别变为 a 和 b，跟 fib_2 做一个对比，就会发现两者本质上的相同之处。

而其实很多高端的编程语言（特别是 Lisp 系和 ML 系）是会自动把这种 fib_5 这种形式的尾递归转化为 fib_7 的形式，这种方法称之为**尾递归优化**（tail-recursive optimization）。这样一来，节省了不必要的调用栈的分配与销毁，能够节省不少时间和空间上的开销。但是，很不幸，Python 没有加入这个特性，所以，当你写递归函数的时候，就要考虑它的调用栈的限制了。

[参考资料及注释]

《编程的本质》（编程原本）第三章中对于递归和尾递归有详细的讲解，同时在第一章中对过程、函数、对象等给出了十分精确的定义和描述，能够促进理解编程的更深层的要义。

《计算机程序的构造和解释》（以下简称 SICP）在 1.2 节对递归和尾递归（称为迭代）有详细的讲解。并且提供了一个 fib 函数的调用树展示。SICP 是 MIT 前几年计算机科学的入门教材，但内容和思想的深度直逼国内硕士研究生的水平。许多内容丰富精彩，而且又不是一般的普适性，所以我们会多次引用到其中的内容。

递归，参见**递归**。

（二）深入函数

现在讲函数的更高端的应用。

首先，我们要知道函数究竟是一个什么东西。

我们给出一个预定义的列表，叫 lst，里面放着有限个**斐波那契数列项**。

```
60  lst = [1,1,2,3,5,8,13,21,34,55,89]
```

现在有个要求。我要得到这个列表排序后的结果。

Python 已经给你做好很多事情了，你不必过于纠结于该怎么实现**排序**算法。

```
61  sorted(lst) #; => [1,1,2,3,5,8,13,21,34,55,89]
```

较之于你还要纠结如何实现算法，Python 已经明智的选择了快速排序来帮你搞定了。Python 的 sorted 函数会以一个参数作为待排序的序列，然后返回一个新的序列作为排序的结果，这是符合函数式规范的。

然后你应该也发现了，我们给出的列表本来就是有序的，所以 sorted 几乎没做什么。

然后我们加入第二个要求：得到这个列表的**逆序**结果。

于是会有以下代码：

```
63  lst_1 = sorted(lst)
64  lst_1.reverse()
```

执行完之后的 lst_1 就保存了逆序之后的结果，而其实 reverse 方法返回的是 None。

整个逆序过程是在 lst_1 上进行。其中似乎有那么一些不尽人意的感觉，因为这个还是有一些 side-effect 在的。

同样的需求用 Ruby 表达起来看上去就稍微舒服一些：

```ruby
lst = [1,1,2,3,5,8,13,21,34,55,89]
lst.sort.reverse
```

其中，lst.sort.reverse 并不会对 lst 有任何影响，只是返回排序再反转之后的结果。

Python 也有自己的达到类似效果的方式，没有 Ruby 的链式调用这么直接，但也能实现类似的效果。

```python
def lst_comparator(o1,o2):
    return o2-o1
sorted(lst,lst_comparator)
#; => [89, 55, 34, 21, 13, 8, 5, 3, 2, 1, 1]
```

Python 的 sorted 与 Java 的 java.util.Arrays.sort 类似，在接受列表为参数的同时还可以接受一个 comparator（在 Java 里面则是实现了 java.util.Comparator<T> 接口的类型对象），根据 comparator 的返回值来排序。

那么，这里的 lst_comparator 就是作为 comparator 出现，作为 sorted 的参数值传递进而使用。也就是说，在这里，函数就是一种值，或者说，函数其实就是一种对象。

那么，对于一种对象，就应该有相应的生成方法。Python 的 def 语句块是一种，另外一种则就是 **lambda 表达式**（在其他编程语言里面或许有另外一个称呼，叫**匿名函数**，可是 Python 里的 lambda 表达式跟 def 比起来实在是太弱了）。因为有了 lambda 表达式，所以进阶的写法如下：

```python
lst_comparator2 = lambda o1,o2: o2-o1
sorted(lst,lst_comparator2)
#; => [89, 55, 34, 21, 13, 8, 5, 3, 2, 1, 1]
```

或者更简单的：

```python
sorted(lst,lambda o1,o2:o2-o1)
#; => [89, 55, 34, 21, 13, 8, 5, 3, 2, 1, 1]
```

在这里 lambda 表达式就有一些函数字面量的意思了。

更多示例比如：

返回列表反转的结果（no side-effect 的 reverse）：

```python
sorted(lst,lambda o1,o2:-1)
#; => [89, 55, 34, 21, 13, 8, 5, 3, 2, 1, 1]
```

求各元素平方之和：

```python
sum(map(lambda a:a**2,lst))
#; => 12816
```

或者：

```python
reduce(lambda e,v: e+v**2,lst)
#; => 12816
```

每一个都要比你苦苦思索着如何去设置变量然后再一步步的写 for 循环简单容易又方便得多。

而且，由于函数是作为值保存在变量中，那么我们同样可以用对待变量的方法来对待函数，而且作为一个特定的值，函数可以在任何可能的地方被定义。这样一来我们就能够根据条件和需求来动态的生成函数了。

一个简单的示例如下：

```
86  def spliter(n):
87      return lambda a,b: a-n
88  sorted(lst,spliter(9))
89  #; => [8, 5, 3, 2, 1, 1, 13, 21, 34, 55, 89]
```

这个示例中，spliter 函数会返回一个闭包（closure，一种特殊的函数），这个闭包会按照 spliter 接受的参数作为分割值，将一个列表分为两部分，小于该值的将会放在前半部分，并且反转顺序；大于该值的项则会移动到后半部分，位序关系不变。

所谓**闭包**，就是绑定了上层函数中局部变量的函数。Python 中对闭包支持得匮乏也是一个弱点（真丫是一个奇葩！）。所以深入闭包的内容，我们会移步 Ruby 或者 JavaScript 中讲述。

由于闭包会绑定**上层函数的局部变量**（又叫做 upvalue），并持有变量状态，那么我们就可以利用这一特性来实现更多高端的东西（JavaScript 代码）：

```
23  function Pair(x,y){
24      return function (fun){
25          return fun(x,y);
26      }
27  }
28
29  function first(a,b){
30      return a;
31  }
32
33  function rest(a,b){
34      return b;
35  }
36
37  var p = Pair(1,2);
38  var a = p(first); //; => a = 1
```

对应的等功能的 Python 代码：

```
91  class Pair:
92      def __init__(self,x,y):
93          self.x = x
94          self.y = y
95      def first(self):
96          return self.x
97      def rest(self):
98          return self.y
99
100 p = Pair(1,2)
101 a = p.first()
```

其实并不是我强拿 Python 的类来说事儿，而是因为类在某种程度算是这种闭包的简化体。闭包利用自身的特性可以很容易的实现面向对象中的**消息传递**和**封装**，不过目前这个对象是只读的，我们却需要另来把它变成更符合 OO 特性的可变对象：

```
41  function MutablePair(a,b){
42      return function(f){
43          var x = f(a,b);
44          a = x(rest)(first);
45          b = x(rest)(rest)(first);
46          return x(first);
47      };
48  }
```

另外是这个对象所对应的方法：

```
50  function get_first(a,b){
51      return Pair(a, Pair(a, Pair(b, null)));
52  }
53
54  function get_rest(a,b){
55      return Pair(b, Pair(a, Pair(b, null)));
56  }
57
58  function set_first(value){
59      return function (a,b){
60          return Pair(value, Pair(value, Pair(b, null)));
61      };
62  }
63
64  function set_rest(value){
65      return function (a,b){
66          return Pair(value, Pair(a, Pair(value, null)));
67      };
68  }
```

你会觉得很蛋疼。这特么到底是个什么东西！不妨我们试一试吧：

```
70  var mp = MutablePair(1,2);
71  mp(get_first); //; => 1
72  mp(set_first(2));
73  mp(get_first); //; => 2
```

我们来解释一下 41-68 行这段代码。

首先，一个 MutablePair 对象会接受一个函数作为消息，同时要求该函数有以下形式的返回结果：

```
75  /**
76  Pair( <retval>,
77      Pair( <val_a>,
78          Pair( <val_b>, null)
79      )
80  );
81  */
```

即一个嵌套的 Pair（Lisp 等语言中的 List 即是这种形式），第一个元素<retval>是整个消息的返回值，第二个元素<val_a>是消息执行结束后 a 的值，第三个元素<val_b>则是消息执行结束后 b 的值。

于是 get_*方法则只需要修改<retval>为特定的值即可，而同样，set_*方法也只是需要改动对应位置的值即可。

接下来在 MutablePair 对象闭包中，将<val_a>和<val_b>分别赋予对象的 a 和 b 字段，同时将<retval>返回。

其实 JavaScript 的对象机制就是用的类似的简化方法，所以上面的一坨坨初学者看起来蛋疼的代码，其实简化下来如下：

```
83
84  function MutablePair2(x,y){
85      var _this = {
86          get_first: function(){return x;},
87          get_rest: function(){return y;},
88          set_first: function(value){x = value;return value;},
89          set_rest: function(value){y = value;return value;},
90      };
91      return _this;
92  }
93
```

这样一来我们就能通过熟悉的点号表达式来进行熟悉的 OO 操作了：

```
94   var mp2 = new MutablePair2(1,2);
95   mp2.get_first() //; => 1
96   mp2.set_first(3)
97   mp2.get_first() //; => 2
```

至于如何在这基础上实现继承，则又是 JavaScript 的一套奇技淫巧了。其他的编程语言也从中吸取了很多，甚至设计模式中都有一个对应的 Prototype 模式。

继承、**封装**、**消息传递**、**多态**（动态类型语言本身就支持多态性），于是没有 class，没有 extends，没有乱七八糟的什么其他的东西，我们就把它给搞定了。

面向对象很牛吗？不过如此。而且本来函数式编程实现起来特别简单的东西，还非要用复杂的手段设计一系列的模式来解决。把复杂的事情简化之后在用复杂的办法实现简便的内容，这应该就是所谓的企业级吧。

[参考资料及注释]

《如何设计一门语言》微软陈梓瀚的一系列文章，仍在更新中。第六篇涉及到了有关闭包的内容，本讲的 JavaScript 代码大部分源自于此。 http://cppblog.com/vczh

SICP 中也有多处涉及到闭包、lambda 表达式和 map/reduce 等高阶函数的内容。对于对象和状态，在第三章提出了更多深刻的见解。SICP 使用 scheme（Lisp 的一种方言）作为编程语言来讲述。

Pair 作为 Lisp 中的最基本的数据处理单元（又称为 CONS），是进行基本运算的基础。其中 first 和 rest 在 Lisp 中以 car 和 cdr 函数表示，而 map 及 reduce 等函数也是多数 Lisp 方言的内置函数，基本的实现方式也是在 List 上进行递归运算。

Lambda 表达式的理论基础是 lambda calculus（λ演算），一个形式化的计算模型，与图灵机和冯诺依曼计算模型在计算机科学中具有同等重要的地位。同样也是 Lisp 编程语言的理论基础。Church encoding 基于无类型的 lambda 演算，可以用来模拟基本的数理逻辑运算。关于 lambda 演算的详细内容可以参考：wiki:Lambda Calculus / 《Lecture Notes on the Lambda Calculus》

高阶函数中，针对表操作的 map 和 reduce 是最经典和最常用的两个。Python 为 map 提供了语言级的支持（list comprehension），而 Google 在 2004 年发表的 MapReduce 并行计算模型的基本思想也是来源于这两个操作。

Lua 是一种轻量级的语言，同样没有基于类的对象机制。之前我曾经利用 Lua 的 table 和 function 实现过一种近似基于类继承的面向对象框架，那个时候还不理解 meta-table/meta-class 和 closure，只是一个简单的实现，参见《Lua 下的基础 OOP 框架实现》。