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。

过程抽象

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。

```
1   def fib(n):
2       if n==0 or n==1:
3           return 1
4       else:
5           a = 1
6           b = 1
7           c = 0
8           for i in range(2,n+1):  #just like for(i = 2; i<=n ; ++i)
9               c = a+b
10              a = b
11              b = c
12          return c
```

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：

```
15  def fib_2(n):
16      if n==0 or n==1:
17          return 1
18      else:
19          a,b = 1,1
20          for i in range(2,n+1):
21              a,b = b,a+b
22          return b
```

但是，既然是在讲函数，我们就一定要让函数更加纯一些，减少 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：

```
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)
```

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：

```
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)
```

```
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类似 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。

[参考资料与注释]

《编程的本质》（*Elements of Programming*）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》（*Structure and Interpretation of Computer Programs*，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。

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   reversed(sorted(lst))
64   #; => [89, 55, 34, 21, 13, 8, 5, 3, 2, 1, 1]
```

同样的需求用 Ruby 表达起来看上去就稍微舒服一些：

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

```
66  def lst_comparator(o1,o2):
67      return o2-o1
68  sorted(lst,lst_comparator)
69  #; => [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。

函数作为对象

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：

```
71  lst_comparator2 = lambda o1,o2: o2-o1
72  sorted(lst,lst_comparator2)
73  #; => [89, 55, 34, 21, 13, 8, 5, 3, 2, 1, 1]
```

或者更简单的：

```
75  sorted(lst,lambda o1,o2:o2-o1)
76  #; => [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）：

```
78  sorted(lst,lambda o1,o2:-1)
79  #; => [89, 55, 34, 21, 13, 8, 5, 3, 2, 1, 1]
```

求各元素平方之和：

```
81  sum(map(lambda a:a**2,lst))
82  #; => 12816
```

或者：

```
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：

```
def spliter(n):
    return lambda a,b: a-n
sorted(lst,spliter(9))
#; => [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。

闭包与面向对象

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。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）：

```
function Pair(x,y){
    return function (fun){
        return fun(x,y);
    }
}

function first(a,b){
    return a;
}

function rest(a,b){
    return b;
}

var p = Pair(1,2);
var a = p(first); //; => a = 1
```

对应的等功能的 Python 代码：

```python
class Pair:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def first(self):
        return self.x
    def rest(self):
        return self.y

p = Pair(1,2)
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。

可变性与状态

```javascript
function MutablePair(a,b){
    return function(f){
        var x = f(a,b);
        a = x(rest)(first);
        b = x(rest)(rest)(first);
        return x(first);
    };
}
```

另外是这个对象所对应的方法：

```javascript
function get_first(a,b){
    return Pair(a, Pair(a, Pair(b, null)));
}

function get_rest(a,b){
    return Pair(b, Pair(a, Pair(b, null)));
}

function set_first(value){
    return function (a,b){
        return Pair(value, Pair(value, Pair(b, null)));
    };
}

function set_rest(value){
    return function (a,b){
        return Pair(value, Pair(a, Pair(value, null)));
    };
}
```

你会觉得很蛋疼。这到底是个什么东西！不妨我们试一试吧：

```javascript
var mp = MutablePair(1,2);
mp(get_first); //; => 1
mp(set_first(2));
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：

```
/**
Pair( <retval>,
    Pair( <val_a>,
        Pair( <val_b>, null)
    )
);
*/
```

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：

```
function MutablePair2(x,y){
    var _this = {
        get_first: function(){return x;},
        get_rest: function(){return y;},
        set_first: function(value){x = value;return value;},
        set_rest: function(value){y = value;return value;},
    };
    return _this;
}
```

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：

```
var mp2 = new MutablePair2(1,2);
mp2.get_first() //; => 1
mp2.set_first(3)
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。

面向对象也不过如此。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再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。

[参考资料与注释]

*《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。

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（我们会在后面深入解释这些内容）。

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》*。

数据抽象

数据产生器

这里的数据产生器与 Python 的 generator 还是有着区别的，跟*《具体数学》*和*《计算机编程艺术》*中提到的深奥的生成函数更加比不得。

我们回到第二讲开始的那个列表。

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

我们是直接就给出了这么一个列表。虽然一项项的看上去会发现他就是 fib(0)到 fib(10)这 11 个数，可是这些简单的数字要让人一个个记住然后写下来都也是很困难的。

既然 Python 都提供了 range 函数，为什么我们不模仿着做一个出来呢？

```
103  def fib_range(s,e):
104      lst = []
105      for i in range(s,e):
106          lst.append(fib(i))
107      return lst
```

这样一来，lst 的值就可以用 fib_range(0,11)来简单表示了。直观明白，而且省掉许多代码。嗯，我们在解释器里写段代码来测试一下吧：

```
>>> rng = fib_range(0,100000)
```

诶，怎么就卡住了！！

毕竟你只生成一个 fib range 还不是目的，目的是拿它来用。可是在这个地方就卡壳，哪里还有什么心情去用。

不过如果你知道 Python 的 range 了，自然就知道另外一个 xrange，xrange 和 range 的功能类似，但返回的并不是一个列表，而是一个叫做迭代器的对象（类型是 types.XRangeType），每对这个对象调用某个特殊的方法，就会返回下一个你所期待的结果。或者说，每次调用该方法之后，该对象的部分状态会被按需更改，然后再一次调用时会有同样的效果，依次这样进行下去就有了我们所要的效果。

而且，更重要的是，生成一个这样的对象要比生成一个上万项的列表要容易得多。

由于状态是要更改的，但 Python 2 中的闭包没有这么强，我们就只好继续使用 JavaScript 来实现了：

```
 99  function xfib_range(a,e){
100      return function(){
101          if (a != e){
102              return fib(a++);
103          }else{
104              throw undefined;
105          }
106      };
107  }
```

用起来得心应手：

```
rng = xfib_range(0,11);
rng() //; => 1
rng() //; => 1
rng() //; => 2
```

但是，你不觉得某些地方不对劲吗？

Python 中的迭代器也有这种不对劲的地方：你只能向前迭代，不能折回，更不能像数组那样进行随机位置访问（即通过下标的形式指定位置来进行访问）。

同样的，要解决这个问题我们还要好好的利用一下闭包。

高级迭代

我们用 JavaScript 构造一个 yfib_range 出来：

```
function yfib_range(a,e){
    c = a
    return function(fun){
        var x = fun(a,e,c);
        c = x[1];
        return x[0];
    }
}
```

```
function fib_forward(a,e,c){
    if(c < e){
        return [fib(c),c+1];
    }else{
        throw undefined;
    }
}

function fib_backward(a,e,c){
    if(c > a){
        return [ fib(c-1),c-1];
    }else{
        throw undefined;
    }
}

function fib_seek(i){
    return function(a,e,c){
        if(a + i < e){
            return [ fib(a+i),c ];
        }else{
            throw undefined;
        }
    };
}
```

啊哈，你看出来了？俨然就是 xfib_range 和第二讲的 get_*及 set_*方法的合体。这样子一个支持双向及随机访的问数据产生器就这么诞生了。而且，如果要追求通用性，还可以在生成方法中传入一个计算函数，这里固定了是使用 fib，你可以使用 fact、sum 或者其他的任何一个你所能够用到的函数。

Python 中的迭代器理论上只支持前向迭代。其中的关键部件是 next()方法（在 Python 3 里面是__next__()），所以要实现一个迭代器（无论是内部还是外部），只需在 next 中实现相应的逻辑就好。

比如 xfib_range 在 Python 中就可以这样子实现：

```
113  class xfib_range:
114      def __init__(self,s,e):
115          self.start = s
116          self.end   = e
117      def __iter__(self):
118          return self
119      def next(self):
120          if self.start != self.end:
121              res = fib(self.start)
122              self.start += 1
123              return res
124          else:
125              raise StopIteration()
```

因为 Python 中的 for...in..语句会默认通过__iter__和 next 来进行迭代，所以这个写法就成了一个标准的形式，而且一般 for 循环总是做前向迭代的，也就前向迭代这种变得更加常用而且一般化。在 C++11 的 range-based for 语句中则是通过比较是否到达了 end()的位置来进行迭代，所以 C++所支持的迭代器种类要多出不少（而且主要是外部迭代器）。

这种照规律生成数据的情况，Python 也给出了一个简单的解决方案，就是 yield 语句以及 generator 对象（类型为 types.GeneratorType）。虽然在更高级的编程应用中，yield 还有很大的发挥空间，但现在我们只管生成数据这一部分。

一个标准的生成器代码如下：

```
127 ▼ def x2fib_range(s,e):
128 ▼     for i in xrange(s,e):
129           yield fib(i)
```

代码简单到让你想哭，可是用法还是跟 xfib_range 一样。

这段代码的关键就是 yield 语句：yield 将其后的值返回给调用处，然后挂起执行。再一次调用的时候继续执行，然后遇到 yield 时会再次挂起。

描述起来很简单，但是其实 yield 相当于在单线程模型中加入了一个新的流程控制方式，比异常流还深了一层。文件 yieldex.py 中生凑出来一个例子，来将 yield 用作其他的用途，其实这个功能完全可以用 OO 来实现。Python 自称是“做事情的方法只有一种”，其实也还是可以简单就被破了的。

[参考资料与注释]

生成函数在 Donald E. Knuth 的《具体数学》和《计算机编程艺术》中分别有提到。虽然也是通过某种方式还还原一个序列的方法，但与偏向数学，已经远离了我们本来的目的。所以这里提到的数据产生器和 Python 中的生成器都跟其关系不大。不过作为计算机科学的根本，Knuth 的这两本经典作品还是很值得一读。

SICP 中讲类似于 xrange 或者 xfib_range 这一类的内容（称之为流）放在了第 3.5 节来讲述，其中特别提到的一点就是流的延迟计算特性。延迟计算也是函数式语言的典型特性之一。SICP 同时提到了 delay 和 force 原语的实现，以及对求值结果的记忆功能实现（这一点后面将会提及到）。

迭代器是经典的编程组件，同时迭代器模式也是《设计模式》中涉及到的一个非常重要的模式。大多数高级语言都对其进行了语言上的支持，而实现上也大同小异。在《松本行弘的编程世界》里面提到了关于内部和外部迭代器的区分。Python 的__iter__调用就是决定迭代器的内部和外部性。

在 C++中迭代器的实现转到了标准库层面，同时迭代器的种类（Category）也分成了很多种，比如 std::sort 就要求接受的迭代器必须是随机存取迭代器等。关于 C++迭代器的详细内容及实现，参见侯捷《STL 源代码剖析》。

列表处理（上）

无论你怎么说，List 总是最常见的结构之一，编程中如此，生活中亦是如此。FP 始祖的名字（LISP，LISt Processing）也深刻体现了这一点。Python 里面已经原生支持 list 这种结构，而且有特定的字面量语法来构建和使用。同时，如果你也有经常使用 Python，那么肯定也知道这一结构的重要性；当然我们要讲解原理，就不能用已有的结构，而是自己来实现一个新的 List。

首先，列表里面包含的是特定的元素，而且，列表是一种有序的（ordered）结构，元素之间是有先后位序的，所以，保存每一个元素的同时，还要有一部分空间来保存该元素的后一个位置。

我们前面提到过 Pair 这种结构，而用来保存列表的每一个单位的结构与之类似，遵照 Lisp 的传统，我们就把它称作 Cons。

构造列表

一个 List 的基本结构使用 Haskell 表示如下：

```
data List a' = Cons a' (List a') | None
```

None 用来表示我们已经到了这个 List 的尾部了。

Python 没这么高端的结构，所以我们需要自己构造相关的东西。首先，我们的闭包可以用来保存数据，那么就可以拿他来当作 Cons 用。于是一个 Cons 就只是一个简单的闭包表示：

```
141  ## constructors
142  def make_cons(first,rest):
143      return lambda fun: fun(first,rest)
```

而接下来我们就可以写出这一系列的 helper function：

```
145  def make_list(lead,*rest):
146      if rest :
147          return make_cons(lead,make_list(*rest))
148      else:
149          return make_cons(lead,None)
150
151  ## helper function
152  def head_of(lst):
153      return lst(lambda head,tail:head)
154
155  def tail_of(lst):
156      return lst(lambda head,tail:tail)
157
158  def print_list(lst):
159      if lst :
160          print head_of(lst),
161          print_list(tail_of(lst))
162      else:
163          pass
```

用法如下：

```
166  lst = make_list(1,2,3,4,5)
167  print_list(lst) #; >> 1 2 3 4 5
168
169  print head_of(lst) #; >> 1
170  print_list(tail_of(lst)) #; >> 2 3 4 5
```

make_cons 用来生成个保存元素的闭包，而 make_list 则通过利用 make_cons 来递归地把其参数生成为一个列表的形式：每个列表分为头和尾两部分，头是其首元素，尾是其剩余元素组成的列表，这两部分可以通过 head_of / tail_of 来访问到。

可以看到 make_list 和 print_list 这两个函数是典型的递归函数。原因很明显，我们最初构造的时候就把 List 做成了一种递归结构（首元素+尾列表），进一步地，由于列表是链式结构，所以处理的时候经常会对其进行整体遍历，所以与列表相关的函数会有大量的递归内容。

简单操作

我们现在有了对列表简单的构造和访问操作了，接下来进一步的就是更深一步的对列表的各种构造、访问和变换操作也需要进行实现，这样才能够洞识其本质。

首先是简单的获取列表的长度。Python 中有 len 函数，而对于我们构造的列表，同时也需要一个相应的函数来进行该操作：

```
174 def length(lst,len=0):
175     if lst:
176         return length(tail_of(lst),len+1)
177     return len
```

简单得很。不需要过度的进行赘述，随着我的 tail_of 的访问就能够获得到明显的结果（其实或者说这个 length 其实是在记录 tail_of 的访问次数）。

然后是添加列表元素的 append 和连接两个列表的 concat：

```
179 def append(elem,lst):
180     if lst:
181         return make_cons(head_of(lst),append(elem,tail_of(lst)))
182     return make_cons(elem,None)
183
184 def concat(lst1,lst2):
185     if lst1:
186         return make_cons(head_of(lst1),concat(tail_of(lst1),lst2))
187     else:
188         return lst2
```

两个函数的共同点在于，都要把其中一个 list 给拆开，然后生成一个新的 list，只是对于生成的 List，尾部的内容略有不同罢了。同样也是一直的递归。

这两个函数还能表现出来的一点就是，对于 append 和 concat 这种类型的操作，其实我们的原始的 list 并没有改变，不过是新构建了一个 list：深刻体现了函数式编程的 purity -- 一切皆值，函数使用值生成新值，而不会对原始数据有影响（no side-effect）。

无论我们的 list 有多么复杂，用它们来就构建完全足够了。

另外，还有一些实用的函数让我们利用已有的列表生成新的列表：

```
193 def take(lst,n = 0):
194     if n:
195         return make_cons(head_of(lst),take(tail_of(lst),n-1))
196     return None
197
198 def drop(lst,n = 0):
199     if n:
200         return drop(tail_of(lst), n-1)
201     return lst
```

一个是取列表的前 N 个元素，另外一个则是去掉列表的前 N 个元素。

方法还是极其简单粗暴，if 条件判断和递归帮我们搞定一切。当然，现在这样说还早，我们接下来的内容会尽力阐释这一切。

[参考资料与注释]

列表（list）的重要性已经提到了，不再多数，这里引用一些重要的参考资料。

SRFI-1 中定义了 scheme 中的 List Library，其中提及了本章及下一章所有的内容。并有相应的参考实现，作为一个标准文档，非常值得参考学习。SRFI（*scheme requests for implementation*）是 scheme 的标准扩展集，有很多经典的库 / 语言扩展从这里诞生。

列表处理（下）

一般来说，列表里面的元素必然有一些共同的特性，才能够用 list 这种构造给放在一起，而且，在静态类型语言中，元素的类型是要强制声明的（如 Haskell List a’，C++ List<T>等），所以有些时候我们需要某些机制对列表或列表每一个元素进行一个整体操作。

比如，针对一个数字组成的列表，需要对其进行求和操作。

```
203  def sum_list(lst):
204      if lst:
205          return head_of(lst) + sum_list(tail_of(lst))
206      return 0
```

然后我们就有了 sum_list 函数，来对所有可以做加法操作的列表进行求和。

倘若进一步地，我们要一个对列表所有元素相乘的结果，那么我们便能够根据这个模版改写出一个新的函数：

```
208  def prod_list(lst):
209      if lst:
210          return head_of(lst) * prod_list(tail_of(lst))
211      return 1
```

对比一下我们应该能发现，不同的地方只是函数名（sum 和 prod），运算符（+ 和*）以及默认值（幺元，分别对应为 0 和 1）而已。

那么我们就有理由对该操作进行抽象：

```
213  def acc_list(lst,fn,default):
214      if lst:
215          return fn( head_of(lst),acc_list(tail_of(lst),fn,default) )
216      return default
```

我们定义了一个聚集（accumulate）操作，第一个参数是该函数作用的列表（lst），第二个参数是加到列表元素上的操作（fn），第三个参数是该操作的初始值（default）。这样一来：

```
218  acc_list(make_list(1,2,3,4,5), lambda x,y: x+y, 0)  #; sums
219  acc_list(make_list(1,2,3,4,5), lambda x,y: x*y, 1)  #; products
220  acc_list(make_list(1,2,3,4,5), lambda x,y: x*x + y, 0) #;squares
```

当然，其实说这么半天，结果只是我们自己重新造了一个轮子。这在列表操作中成为 reduce（fold，规约）。完整的（并且改写成了迭代形式的）形式如下：

```
243  # reduce :: [a] -> (b -> a -> b) -> b -> b
244  def fold(lst,fun,default):
245      if lst:
246          return fold(tail_of(lst), fun, fun(head_of(lst), default))
247      return default
```

同样的还有如下的函数集合（each、map/collect、filter/select）：

```
222  # each :: [a] -> (a -> ()) -> ()
223  def each(lst,fn):
224      if lst:
225          fn(head_of(lst))
226          return each(tail_of(lst),fn)
227      return None
228
229  # map :: [a] -> (a -> b) -> [b]
230  def collect(lst,fun):
231      if lst:
232          return make_cons(fun(head_of(lst)),collect(tail_of(lst),fun))
233      return None
234
235  # filter :: [a] -> (a -> Bool) -> [a]
236  def select(lst,pred):
237      if lst:
238          if pred(head_of(lst)):
239              return make_cons(head_of(lst),select(tail_of(lst),pred))
240          return select(tail_of(lst),pred)
241      return None
```

当需要对所有的列表元素进行简单遍历的时候，将遍历操作作为一个函数（fn）传递给 each，each 就会按照你想要的方式顺序的执行下去。

而 map（collect，映射）操作则会把当前列表（lst）每一个元素进行变换（fn），然后得到一个新的列表。

然后就是 filter（select，过滤）操作，会使用一个谓词断言（pred），把符合条件的元素筛选出来，得到一个新的列表。

当然，上面那种方式写起来的 each/collect/select 简单易懂，但并不足够的高效：特别是尾递归技术没有用到，而且，作为这些操作同样都能够与 fold 牵扯上一些关系。

于是我们可以根据类型签名和一些推断，把这些操作利用 fold 改写：

```
249   def collect_over_fold(lst,fun):
250       return fold(lst,lambda head,tail: make_cons(fun(head),tail),None)
251
252   def select_over_fold(lst,pred):
253       return fold(lst,
254               lambda head,tail: make_cons(head,tail) if pred(head) else tail,
255               None)
```

所以由此我们也能基本上得出这么一点，在 map/reduce/filter 这些函子中，最重要的就是 reduce，由它可以导出其他的，同时可以更进一步的推广到更多的应用，那么我们所需做的就只是 if 判断和递归。

例如快速排序算法的实现：

```
# sort
def sort(lst,pred=lambda a,b: a > b):
    if lst and tail_of(lst):
        return concat(
                append(head_of(lst),
                    sort(select(tail_of(lst), lambda e: pred(head_of(lst),e)),
                        pred)),
                    sort(select(tail_of(lst), lambda e: pred(e,head_of(lst))),
                        pred))
    return lst
```

而同样用于分布式和并行计算上的 MapReduce 也是名噪一时，当然虽然实现要远比这几个函子复杂得多，但原理仍然没有超出此范围。

[参考资料与注释]

大多数包含函数式范式的编程语言都提供 map/reduce/filter 或类似的抽象操作（比如 C++的 std::transform/std::accumulate/std::remove_if），Python 同样也有这些函数，所以为了不破坏 Python 自身的环境，我们分别对他们进行了重命名。

本节中出现的各种结构在前文中已经出现，单独列出是因为这些内容特别重要。对于大部分人来说，掌握了 map/reduce 等函子的实现，对复杂数据和复杂逻辑的处理也不会是很困难的事情。