

简明 python 指南

本来这个是某系列连载文章中的第二期的第十五篇，不过限于篇幅，我觉得还是独立出来比较好。写这个指南的目的，是帮助大多数想接触 python 的人更方便更全面的认识这门编程语言并学会如何运用这套工具集。

与此同时我觉得现如今关于 python 的书大多都是多余的，它们花了很大的篇幅在讲些没用或者不相干的东西。很多书假定你对编程的理解还很模糊，假定你可能用着不同的操作系统，其实这对于大多数我的目标读者来讲，都是多余的。当然那也是出版社的无奈，我自然无所谓，我可以假定大家都用的 windows，还都是 xp 以上，只要不影响行文就好。另外这个标题里的简明一词，是相对一本书的厚度而言的。

我最早知道 python 是 09 年，那时候还在用塞班的 s60¹手机。后来是 GAE，那时候版本还是 2.5.x，不过因为已经高三了也就没去折腾。最初开始用 python 大概是从 2012 年秋季休学的时候算起，现在转眼都 2.7.9 了。眼看着 python2 的时代已然走到了尽头，虽然大家都还会继续用，只是过去的时光早已凶猛地从身边流逝了。

最初打算写关于 python 的教程是在 2013 年春。之前 2011 年刚高考完的时候想自己做网站，于是 2012 年春的时候决定学 python（主要受豆瓣的启发）。然而拖拖拉拉，直到那年秋末休学才开始投入起来。到 2013 年的时候基本的东西都会的差不多了，正好社团里有不少新人感兴趣，于是也打算教他们来一起玩，大概写教程的想法最早是这时候有的。后来就是一个悲剧的故事，我复学之后没能掌控好自己的节奏挂了许多课，社团这边的活动也没组织得好，外面帮公益机构做的东西也是一拖再拖直到烂尾。最后我选择降级到 13 级，虽然去年秋季有写些教程的草稿，但后来今年上半年因为个人问题不得不在家待半年，差不多都忘了。直到七月我回到学校，八月有人有重新跟我提起 python 的时候，这个中断了 N 次的想法才得以重新唤醒，虽然最后那个网站我并没有做出来。其实相比 13/14 年时候的想法，这些年来我对 python 的理解也有不少变化，尤其是在我接触了 erlang 和 APL 以后。这也是我决定丢掉去年的草稿重新写过的主要原因之一（另一方面是我忘记存哪了不太好找……）。

不过由于第一次写这种稍微正式一点的教程(严谨的说法只能算指南，教程最好看官方的)，字数会有点多。也许精简一下，或者说真正有用的部分估计不到 6000 字，然而这个环节要我自己察觉出来很不容易，所以如果能收到些反馈建议，而我也对应做些改进，两全之策，岂不美哉。另外，我建议记性不太好或者动手能力较差的，去打印一份（或者一部分）纸质版放在床头，虽然第一版是有点坑，不过我想说的其实是，有些时候纸质版的东西看上去更入眼。

1 S60 是诺基亚一个的手机操作系统，文中往后所有的缩写和名词如果不影响内容主体则不加描述。

目标读者

其实并没有很具体的目标，用这个二级标题只是在假装很严肃的样子。一开始定位在一大二的小盆友（那是 14 年的时候，现在他们都大三大四了），不过就现在的情况发展下去，小学六年级以上也是可以的。

为什么学 python?

这是很多书要跟你讲的第一章，很遗憾，我不会告诉你为什么。因为老实说 python 不是一门很适合用来教学的编程语言²，不过呢，它就是能用来偷懒，所以大家都用它。本质上来说，学会 python 并不是简单的学习它的语法和各种概念，而是学会读懂别人的 python 代码，学会自己写 python，以及熟悉整个 python 的工具链和生态环境以致如何做项目等等。就是这样。而实际上学 python 是一个实践和积累的过程，也就是说本文并不能帮你太多。

准备工作

你需要装一个 python 解释器。我不会跟你解释具体解释器是什么鬼以及和编译器有什么差别，你所需要做的，就是理解这个过程并照着做一遍，碰到看不懂的地方，自己去搜一下或者直接问我也行 -> absente@live.cn。

同样的，我也不会跟你讲太多什么是 python2 和 python3 的区别。这里你只需要下载个 anaconda python2 的集成环境就好。

然后是编辑器。初学我觉得 sublime text 3 或者 notepad2-mod 都可以，当然如果要推荐的话，我觉得 sublime 和 pycharm³就够了。vs2015 其实也可以，不过有点太重型了，而且关键是我并不太熟悉...

还有就是备上两份文档，python2 和 python3 的。Anaconda 实际上会附上一份 python2 的 chm 文档，然而要自己去目录里找出来。这里个人觉得 chm 文档还是够用的，当然，如果你习惯多标签阅读，那么 html 的更适合你，如果是电子书的话，官网也有 pdf。另外我推荐用那种专门的离线文档工具，比如 zealdocs（mac 下的是 dash）。

这里附上 windows 下可用的 anaconda 和 sublime text 3 以及 pycharm⁴，zeal 和官方的 chm/html/pdf 文档：

² 这年头没有模式匹配功能的语言还好意思出来混？如果是我当老师的话我肯定首选 erlang。

³ 在国内就用专业版吧，社区版是给版权强迫症留的。

⁴ 另外，不要跟我讲版权，我没有要传播盗版的意思，你问我资瓷不资瓷正版，我当然是资瓷的，但是下载东西也是符合天朝国情嘛，如果有条件的，我也推荐你们用正版。其实开源的软件有不错的，像什么 e macs 之类的。但是我只是在讲 python 嘛，不想涉及太多周边的，不然就不够简明了。

百度网盘链接（合集）：<http://pan.baidu.com/s/1dD6nVn7>

- ♦ Python: <https://www.python.org/>
- ♦ Anaconda: <http://continuum.io/downloads>
- ♦ Sublime text 3: <http://www.sublimetext.com/>
- ♦ Pycharm 4: <https://www.jetbrains.com/pycharm/>
- ♦ Zeal / Dash: <https://zealdocs.org/>

装 anaconda 的时候一路默认 next，这样 python 的执行路径应该也就在 PATH 环境变量里了，当然如果你实在是闲 anaconda 太大，也可以自己下个官方的 Python⁵，然后选择加到环境变量里。其实加到 PATH 里面并不是总是必须的，只是通常为了方便调试就成了潜规则。

语法概要

你可能觉得上来就讲语法会不会有点消化不良？其实不会的。所谓编程语言的语法，就是按照一定的符号规则去表达你的逻辑和程序流程。

在这里我觉得有必要列举一下其他编程语言的语法。首先是 c 语言（以下都是单行代码）：`void main(){}`

上面是一段合法且并没有什么卵用的 c 代码。本来想举个 java 的例子，不过好像不太确定怎么写，就搜了一下 `hello world`，然后改了改，但是不确定是否一定合法⁶，于是就放弃了。换成 js: `function main(){}` 或者匿名函数形式的 `main = function(){}`

然后是 python:

```
def main(): pass 或 main = lambda: None
```

我们在这里不是要对比出什么优劣，那是西方人喜欢干的事情，而且相对没什么意义。这里我们需要做的是寻找其中的共性。比如你会发现它们都可以用圆括号来表示参数区域。

感觉真要讲语法的话，到这里就差不多了，最多给你强调一下，缩进一定要用四个空格，区块注释要用双引号。实际上 python 的语法并没有多复杂。很多语法关键字刚入门的话根本用不到，而且也不是那么容易搞懂（比如 `with` 和 `yield`）。于是你只需明白几个基本概念就可以写 python 代码了。当然这里并不鼓励你马上去写个 `hello world`，因为那也没什么实际意义，而且不太符合我国的教育习惯。这里我的思路是先讲点概念，然后教你怎么读懂 python 代码，然后再去改或者去写。当然，语法不可能就说这么点，那是开玩笑的。在讲概念之前，正经地补充点语法常识还是有必要的。

⁵ 一般来说 CPython 指的就是官方版的 python，其他比较常见的解释器还有 pypy，另外 Cython 和 CPython 不是一个东西。

⁶ 合法（legal）的意思放到编程里面就是符合语法规则的意思，习惯就好。

首先，python 用缩进替代了大括号来限定作用域（缩进这个设定有好有坏，不过算是 python 比较明显的几个特征之一），当然，和传统编程语言一样，实际上限定作用域（或者说语法单元）的还是小括号。然后你可能会说为什么不强调没有分号这个问题，其实，python 是允许分号的，只是依照潜规则，大家都不用而已。而且，如果你喜欢的话，还可以开启大括号缩进模式... 不过出于人身安全考虑，不要这么做，这样只会被当作异教徒。

先说逻辑相关的，有几个关键字：`and`，`or`，`not` 和 `is`，`==`，`!=`，虽然这个用来替换 `&`，`||`，`!` 好像更直观了，但是有时候也会搞晕人。另外，逻辑运算的返回值只有两个，`True` 和 `False`，首字母大写。至于 `None` 这个是独立出来的一个值，对应 C 里面的 `null`。容易搞混的地方自己动手试试最好做个笔记。`is` 和 `==` 通常是不会等价的，所以如果一个值你能确定全等的时候，才尽量用 `is`。用 `is` 的好处是有时候逻辑表达可以更连贯一点（有些时候计算效率也更高），比如 `A is B`。但有时候也不一定符合表达习惯，比如 `A is not B`。还有就是明白，`[] is []` 这种肯定是 `False` 的，因为不同的列表指向的内存空间肯定不同。

除了前面说到 `None`，Python 和 C 里面很多概念都是可以对应上的，这个往后用久了会有更多体会，但是要注意的是 python 也有自己提倡的一些潜规则，比如 `PEP8`，然而这里的很多规则实际上和 C 是不完全对应的，这也是我为什么个人并不提倡完全遵循 `pep8` 的原因之一。

说到这种潜规则，不得不提一下 python 的命名规则，变量和函数命名也是很重要的一个细节，python 推荐用小写字母加下划线比如：`blue_shit`，如果是 `class` 的话会推荐你写作 `HolyShit`，至于 `class method` 很多人也推荐小写加下划线（`eat_shit`）不过个人觉得继续用 `java` 那种小写字母开头的驼峰写法（`eatShit`）也不是什么坏事（毕竟 `Google` 也是这么玩的不是么）。不过通常来讲，个人自从用惯 `erlang` 之后就不再特别讲究这个了。比如前面的 `blue_shit` 可以缩写作 `S` 或 `BS/SB` 随意，有时候变量本身的意义并不是那么重要，只要不影响逻辑就好。所谓的代码可读性，其实只在你需要读懂代码的前提下才有意义，而有时候我们不用搞懂代码，只要会用就好了，这个概念比较类似道家的无用，不过外国人多半是搞不懂的。

流程控制方面，主要就是 `if` 和 `for`，然后 `while` 用的很少（一般也就是结合 `yield` 或者来个 `mainloop`），没有 `switch`，`switch` 用 `if` 和 `elif` 的组合代替。然后，python 的 `else` 比较奇葩，`if`，`for`，`while` 都可以在屁股后面加个 `else`。`if...else` 比较好理解，`for...in` 后面的 `else` 只在 `for` 循环没有 `break` 的情况下最后执行（也就是补刀用，如果 `for` 没有完整走完是不会有），`while` 后面的 `else` 同理。然后，`break` 跟 `continue` 是想反的，和 `c` 一样。不过 python 里面有个 `pass`，这个语句没有啥作用，一般用在 `def` 后面临时写个空函数，然后结果和 `def a():return None` 是一样的。下写 `if` 分支的时候，你可以用 `pass` 临时占个位置。因为单独写个注释在 `build` 的时候还是会报错的，而补上 `pass` 就没事了。另外如果习惯了 `pass` 之后，可以用来标注某些未实现的功能，方便查找。还有就是，如果你习惯 `c` 里面的三元表达式即 `Z=X?A:B` 这种，python 也提供了一个类似的语法糖即 `Y=A if X else B`。后者可能对于印欧语系的人更为直观一点，然而对于天朝群众开讲可能还是 `c` 的写法更好记些吧。

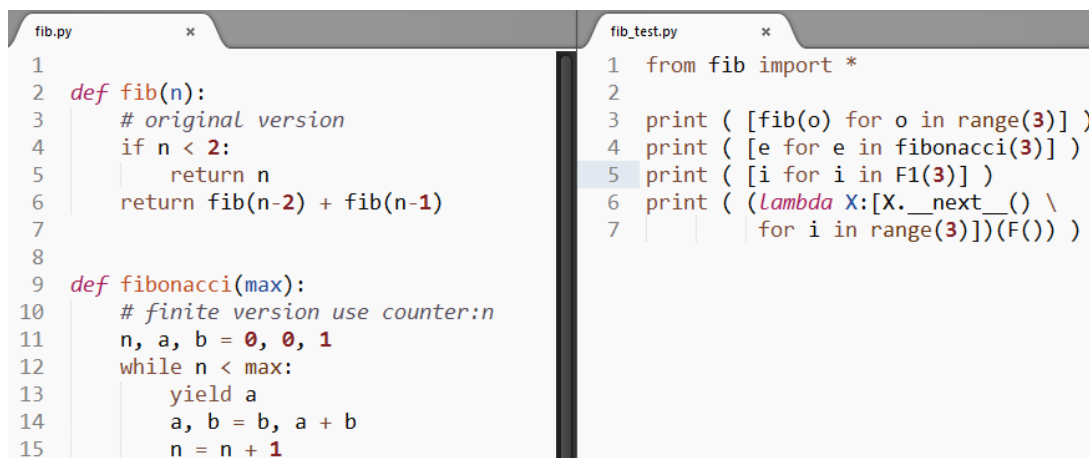
补充说明一下，python 的 for 和 c 看上去是不太一样的，因为 python 默认采取 for in 这种遍历的形式。如果要当 c 的 for 来用，一般是 for i in range(N)。原理是这样的，range 这个内置函数可以自动生成一个[0...9]的整数列表，然后 for in 会逐个提取出来赋值到 i 上面。

流程控制这部分，最重要的还是异常处理。老实说 python 的异常处理并不是很高明，只是勉强够用的那种（这方面 erlang 是目前做的最好的，没有之一）。一般最简形式是 try...except，然后是 try...except...else 还有 try...except...else...finally。except 的意思就是 catch 住某个异常。最基础的异常是 Exception，只要出错了都可以抓住它。else 就是没异常的时候走的流程，finally 就是无论怎样都会有的最后一步，之所以会有这个设定是因为缩进层级会影响到作用域。简单说，如果你在 try 的下一级缩进里写了个 a=1，那么在 try 语句的外层是访问不到的，而 finally 可以解决这个问题因为缩进的层级是一样的。

其实 try 语句除了用来写这种防御性的代码，还有一个用途就算做一些判定，比如类型比较，因为对于动态编程语言来讲有时候你要确定一个对象的类型不一定很容易，类似 isinstance 这种，只能在有限个基类中做判定，无法应对那种过于未知的情况，而 try 就不同了，你只用确定你的假设是否正确，那就可以认为这个是以满足需求的。实际上，我用 try 比较少，反而是用 raise 更多。这个可能是受了 erlang 和其他参考资料的某些影响。

剩下的，补充一下 with，装饰器 decorator 和匿名函数 lambda 我放到后面再讲。这个就是个语法糖，可以免去一个子流程过程中的临时变量定义及预处理步骤或后续必要操作对整体流程的干扰，最常见的是 open 和 with 的结合，例如 with open('target.txt') as f: len(f)，然后你就可以省略掉 f.close() 这个过程了，一般都是类似这样的。不过要想自己写个 with 的流程，放在这里来讲就有点复杂了。

然后是操作相关的语法。这里我不太想讲 yield，因为篇幅可能会更长⁷。不过我想举个斐波那契数列⁸的例子应该就够了。下面的代码如果看不懂也没关系，我会到后面的章节重新解释一遍，现在只要关注里面的 yield 就好。



```
fib.py
1
2 def fib(n):
3     # original version
4     if n < 2:
5         return n
6     return fib(n-2) + fib(n-1)
7
8
9 def fibonacci(max):
10    # finite version use counter:n
11    n, a, b = 0, 0, 1
12    while n < max:
13        yield a
14        a, b = b, a + b
15    n = n + 1

fib_test.py
1 from fib import *
2
3 print ( [fib(o) for o in range(3)] )
4 print ( [e for e in fibonacci(3)] )
5 print ( [i for i in F1(3)] )
6 print ( (lambda X:[X.__next__() \
7         for i in range(3)])(F()) )
```

⁷ Python3 后来还引入了 yield from，总之是个好用但不太好理解的东西，大多数用法都太诡异了。

⁸ 注意，不要用那些国内土鳖的错误缩写 fab，因为全名叫 fibonacci。

```
16
17 def F1(N):
18     # finite version
19     a,b = 0,1
20     while a<N:
21         yield a
22         a,b = b,a+b
23
24 def F():
25     # infinite version
26     a,b = 0,1
27     yield a
28     yield b
29     while True:
30         a, b = b, a + b
31         yield b
32
33
```

```
[0, 1, 1]
[0, 1, 1]
[0, 1, 1, 2]
[0, 1, 1]
[Finished in 0.2s]
```

图 1: yield 演示

然后是 `raise` 和 `return`。`return` 比较好理解，就是返回值，不过 `python` 里面允许忽略返回值，也就是说直接写 `return` 是合法的，即等价于 `return None`。`return` 和 `c` 里面比较的区别用法就是返回多个值，其原理后面讲元组的时候会补充。`raise` 可能没那么好理解，不过通常来讲，`raise` 是补刀用的⁹。有时候显式地抛出异常，更利于维护整体的代码和业务逻辑。

倒数第三个是赋值，包括匹配赋值。`Python` 里面的赋值和 `c` 以及 `js` 不太一样，比如你不能这么写¹⁰：`(a=(b=1))`。不过你可以这么写：`a=b=1`。当然，还可以这么写来交换两个变量的值：`a,b = b,a`。这个交换赋值的原理实际上是等长元组（`tuple`）的匹配，即在`(a,b) = (b,a)`的基础上省略了括号¹¹。当然，其他编程语言里面会有用下划线来表示匿名变量，不过 `python` 没有这个强制设定¹²，习惯的话也可以用单个下划线来表示不需要的变量，比如`[_b,_] = [1,2,3]`。

倒数第二个是 `del`，虽然 `python` 自带垃圾回收，但有时候也需要自己删除不需要的东西，比如某个集合（`set`）里不需要的值。

最后是 `print`，因为这个在 `python3` 里面被踢掉了，所以放到后面意为可选。所以如果要在这一点上兼容 `python2` 和 `python3`，有个小技巧，就是在 `print` 和括号之间加一个空格，这样在 `python2` 里面会被理解为打印一个元组，`python3` 会理解为一次函数调用。

操作之后来讲下结构相关的语法。首先是函数构造，用 `def` 就可以了（记得处理缩进）。

⁹ 参见沈崑当年留下的幻灯片《`python` 编程艺术》，背后的理念和 `erlang` 的 `let it crash` 比较相近。

¹⁰ `C` 之所以能这么写是因为它遵从表达式有值这个思路，其实挺乱的。

¹¹ 这个原理在后面讲元组的时候再回顾补充。

¹² `Python` 是没有模式匹配这个技能的，所以即便是`[a,a]=[1,2]`也不会有问题，最后 `a` 的值为 2。

至于匿名函数，python 有很多限制，不能像其他脚本语言那样写好几行。类定义的话用 `class`，允许多重继承，不过通常来讲除了用 `mixin` 之外都不推荐这么弄。导入的话用 `import` 和 `from...import`。声明全局变量用 `global`，python2 没办法声明局部变量，这个在 python3 里面用 `local` 可以搞定。

基本概念

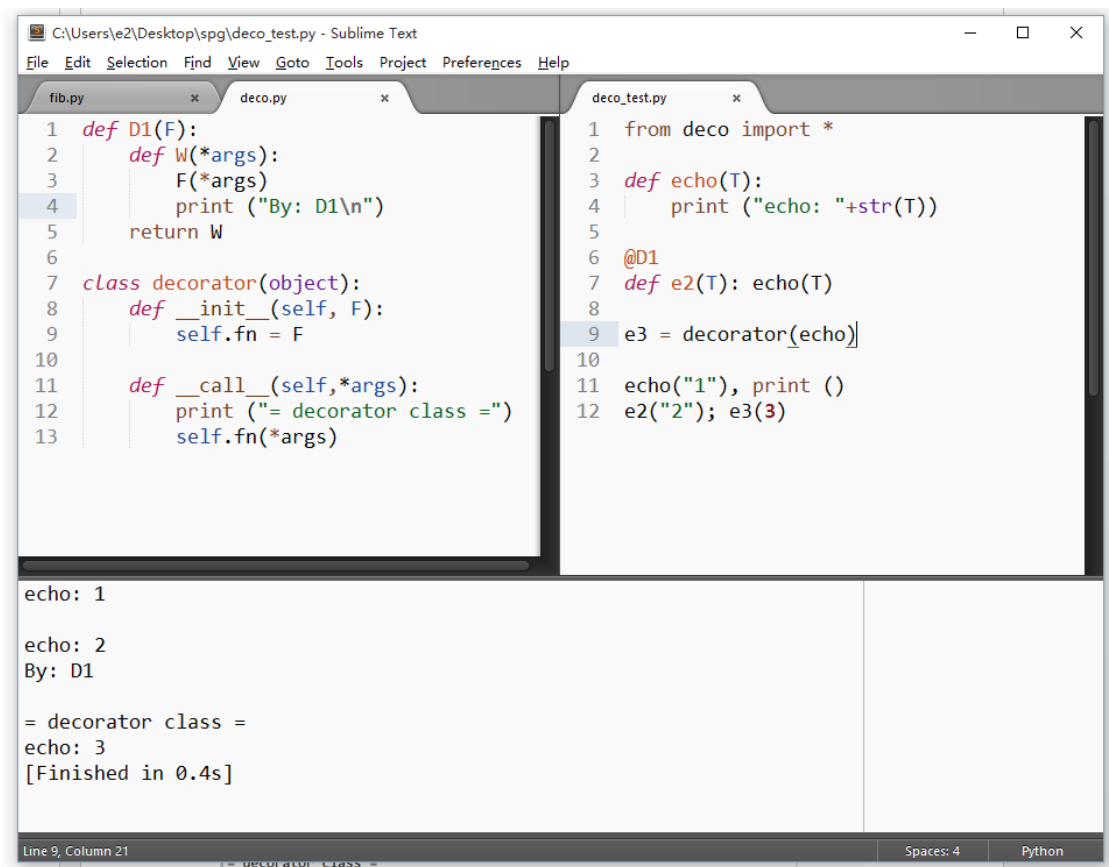
这一节会有点长，想跳过的可以先跳过，不过还是尽量看完先。

说到概念，首先是 **package**（包）。在 `c` 里面，代码是可以很方便的 `include` 的，这种概念很原始，也很实用。就是把一段代码插到另一个地方，而且为了方便，通常都是另一端代码的开头。`js` 也有类似的概念，效果也很相似，不过现在大多数人写 `js` 都不喜欢直接那么干了，讲究个什么命名空间（`namespace`），这个也是有点用的，不过也有点多余。Python 里面主要靠 **package** 来维护这么一个类似 `namespace` 的概念。其实就是学 `java` 的，然而学得很挫。建立 **package** 的方法很简单。新建个文件夹 `abc`，在底下建一个叫 `__init__.py` 的文件，那么回到 `abc` 的上级目录，在这里打开 `cmd`，输入 `python -c "import abc"` 你会发现没有任何问题，那就对了。Import 一个 **package** 跟 `c` 里面 `include` 是比较像的，不过有点不同。比如 `import os` 之后，你可以用 `os.listdir` 去调用，这样保留了 `os` 这个前缀，省得跟当前代码内的函数名冲突。当然必要的时候也可以用 `from os import listdir` 这种方式来省去 `package name`，具体怎么用，这个看应用场景。通常标准库和成熟的第三方库，如果名字不是很长，那么用 `import` 比较多。本质上来讲，`import` 的机制和 `include` 的机制差很多，不过不必深究，用得习惯就好。有人说 `python` 或许比较适合女生学，因为有很多包包。

还有一个概念是模块（**module**），指的是可以被 `import` 的东西。这个概念包含了两个概念，一个是前面说的包，另一个是 `python` 源文件，还是比较好理解的。

然后是类（**class**）和函数（**function**）。Python 的类比较简陋，没有 `java` 里面的 `interface`（接口）。另外所有的类属性和方法（**method**）都存储在类的一个 `__dict__` 属性里。其实在 `python` 里并不需要掌握太多使用类的技巧，只要记得不用太多继承就好。函数的话，`python` 是区分有名函数和匿名函数的，而且 `python` 算是强制你很多时候都别去写匿名函数。这也就是 `python` 作者的一种强迫症而已，不用太在意，偶尔用用 `lambda` 还是很方便的。类和函数的定位不同，如果之前习惯 `c`，那么用函数会比较多，即便写几个类也可能只用 `staticmethod` 而不是 `property` 这种。如果是之前用 `java` 的估计除了一堆类还有各种类装饰器吧。说到装饰器¹³，这是个很方便的语法糖，用类和函数都可以构造，看具体复杂度和各自喜好了。我这里先举个不太复杂的例子就好：

¹³ 这里有很多装饰器的例子：<https://wiki.python.org/moin/PythonDecoratorLibrary>



```
1 def D1(F):
2     def W(*args):
3         F(*args)
4         print ("By: D1\n")
5     return W
6
7 class decorator(object):
8     def __init__(self, F):
9         self.fn = F
10
11     def __call__(self,*args):
12         print ("= decorator class =")
13         self.fn(*args)
```

```
1 from deco import *
2
3 def echo(T):
4     print ("echo: "+str(T))
5
6 @D1
7 def e2(T): echo(T)
8
9 e3 = decorator(echo)
10
11 echo("1"), print ()
12 e2("2"); e3(3)
```

```
echo: 1
echo: 2
By: D1
= decorator class =
echo: 3
[Finished in 0.4s]
```

图 2：装饰器示例

关于对象，python 和 js 的设定其实很多还是相似的。不过没必要在意太多，一般在 python 里就是 class+method 的组合，类似 js 里面用一个对象然后像这样{"main":function(){}这样框住一坨代码。现在普遍承认的面向对象编程实际上并没有什么太多意义，所以尽量忘掉这个概念对迎接未来的软件工程应该有一定的帮助。

对象之下，还有类型（type），这个很重要。不过按理说类型是比对象更基本的概念，但是一旦引入了 OOP，那么不可避免会造成这样的混淆，比如 class 是 type 还是说 type 可以用 class 定义这种新手容易搞混的问题。老实说这点其实就是 python3 主要想改的一个地方，那就是把类和类型设计的更规范化。不过既然我说的是 Python2，那么 python3 之后的东西，可以自己往后再补。就像在前期第十八节¹⁴里提到的，学点旧东西并没有啥坏处，也方便对新事物有更深入的认识。当然即便是 python2，在这里我们不去深究 class 和 type 的关系，只要会用就好。

关于类型，首先你要知道，python 是动态类型的，不像 java 那么死。这个有好有坏，此处不多做分析，只要习惯这种设定就好。然后，你需要知道就是几个基本的类型和扩充类型：数字分整数 int 和浮点 float（一般不用在意长度的问题，而实际上 python 里的 float 是 c 里面的 double）；字符串在 python2 里面是 str，也和 c 一样可以当不可变的字符列表来用，

¹⁴ 还没发布，大意是说，现在学新的软件不如学旧版本的，因为离线文档方便查阅，而且功能精简。

基于 `str` 的有 `unicode` 类型；然后数据集方面，最基本的有列表 `list`，然后是字典 `map`，还有元组 `tuple`，这三个基本的之外，还有集合 `set`，还有基于 `list+tuple` 的有序字典 `ordereddict` 等等。简单说是这个样子的，具体后面展开。不过在这篇指南里不会过多涉及 `ctypes`，之前有 `c` 语言基础的可以自己去了解（比如你想使用些 `c` 里面的专用类型），毕竟现在大多数人之所以还用 `Cpython` 和 `Cython` 主要就是性能考虑和 `c` 语言的扩展性了。

有了前面的铺垫，现在来具体讲下常用的（或者说真正意义上的）一些基本概念（以及基本类型）。首先要讲的是元组。元组这个类型比较特别，因为圆括号在 `python` 里面有很多种不同的用法，当然实际上是一个意思。比如：`(1)` 实际上并不是构造了一个元组，而是给 `1` 加了个括号表面这是一个单元，也就是说 `(1)` 不等于 `(1,)`。再一个，`[i for i in range(2)]` 和 `(i for i in range(2))` 的结果也是不一样的，前者构造了一个列表，后者构造了一个生成器（`generator`）。元组和列表的主要区别在于元组不可变（`immutable`），这个比较符合常规的函数式编程（`FP: Functional Programming`）的思想，前面提到的 `return` 多个值和交换变量的用法都是基于此。

列表是很多编程语言的基本概念，比如 `lisp`。`Python` 的列表基本可以对应 `C` 里面的数组（`Array`），不过 `python` 不限定列表内数据的类型，存什么都可以。另外列表的索引方法比较多样，比如 `L[-1]` 可以表示列表的最后一个元素，`L[:]` 可以表示列表内的所有元素¹⁵，`L[::-1]` 可以让一个列表反转等等（这里背后用到的也是对象的 `__getitem__` 方法加上 `slice`，即切片）。关于列表有很多常用的技巧，而且列表在 `FP` 里面用到比较多，感兴趣的自己去了解下。

再就是字典，这个在其他编程语言里面可能叫做 `map`（映射）或 `hash` 之类的。简单说就是一个符合 `key-value` 的数据集。字典的 `key` 具有唯一性，且 `key` 必须 `immutable`（官方说法叫 `hashable`，其实意思差不多）。众所周知，现在比较常见的数据传输格式比如 `json`¹⁶，里面的 `object` 实际上和 `dict` 是类似的（不过严格的 `json` 不允许把匿名函数当作 `value`）。对于 `dict`，用法也很多。不过本质上一个 `dict` 实际上就是 `tuple+list` 的组合，这个如果你调用一个 `dict` 的 `items()` 就会吐出来一个这样的结构。具体表现为你可以这样解压（`unpack`）一个字典的所有 `key`: `[k for k,v in D]`。另外 `dict` 也可以用下标，不过 `python` 的 `dict` 默认是无序（或者说不符合存储顺序）的，所以不能用像 `list` 一样用下标索引。访问一个 `dict` 的 `value` 可以通过它的某个 `key` 来访问：`D[K]`，如果找不到的话，就会报错（`KeyError`）。当然更多的情况我们不太想捕获这个异常，所以用 `D.get(K)` 就好（这个其实是默认返回了 `None`，如果想返回一个指定值比如 `2`，可以 `D.get(K,2)`）。

`Python` 的集合同 `dict` 一样，比较符合数学规范（这个也是有好有坏）。现在的 `python` 对于集合都支持用大括号来创建了¹⁷，比如：`{1,2,3}`。集合的操作也很多，不过个人通常用的一

¹⁵ `L[:]` 这个操作是很有用的，可以用来复制一个列表。因为有时候直接让 `L2=L` 会碰到修改了 `L` 的值会影响到 `L2`。这里体现了列表的可变性（`mutable`）。

¹⁶ 很久以前还都是用 `xml` 的，于是就有了 `SOAP` 之类的那一套东西。`Json` 大概是 `09` 年左右跟着 `ajax` 流行起来的。

¹⁷ 以前 `2.5` 还是 `2.6` 之前好像是不行的，只能用 `set([1,2])` 这种形式来构造。类似的情况可能还有，所以通常只要用 `2.7` 或 `python3` 就好了。

个技巧就是用集合去排除一个列表里的重复元素，比如：`list(set([1,2,1]))`的最后结果只会留下`[1,2]`。当然，对于这种偏数学运算的东西，比如矩阵什么的，还是用 `numpy` 之类的第三方库比较好。

当然，最基本的两个类型我好想还没展开，即数（`int`, `float`）和字（`str`, `unicode`）。Python 进行数运算的坑不是很多，除了除法和精度可能有些问题，其他都还好。字符串的问题倒是一开始可以坑掉很多人，老实说到今天我都不能完全避免被 `unicode` 坑一下，不过真正被坑的机会其实没那么多的，只要用好 `decode` 和 `encode` 方法，能够避免 python2 里面多数的 `unicode` 坑。

类型讲完，接下来概念只要了解一下就好，比如前面提到过的生成器，这个可以用来实现惰性求值（`lazy evaluation`）和协程（`coroutine`）¹⁸，而迭代器（`iterator`）跟生成器是个很相近的东西，具体差别老实讲我自己都说不清楚，不过对于 python 来讲，如果一个类具有一个能用的 `__iter__` 方法的话，那么这个类的实例就可以拿来 `for in` 遍历了。装饰器（`decorator`）前面讲过了，还有一个在构造方法上类似的概念叫描述符（`descriptor`）。用 `class` 定义的装饰器主要需要提供 `__call__` 方法（通常来讲 python 的类定义大多需要提供一个 `__init__` 方法做初始化），描述符则需要实现 `__get__` 和 `__set__`。这个和 `dict` 的 `__getitem__` 和 `__setitem__` 是有差别的。类似的还有切片（`slice`），道理跟前几个相似，感兴趣的可以看官方文档自己了解下¹⁹。

还有些常见的概念我觉得有必要顺带提一下，比如：递归（`recursive`），循环（`loop`），尾递归（`tail recursion`），回调（`callback`），同步（`synchronous`），异步（`asynchronous`，简称 `async`），协程，操作符（也叫运算符：`operator`），语法糖（`syntax sugar`），编程范式（`paradigm`），函数式编程，面向对象编程（`OOP: object oriented programming`），多态（`Polymorphism`），`duck typing`，模式匹配（`pattern matching`），声明式编程（`Declarative`），命令式编程（`Imperative`），冯诺依曼式编程语言（`von Neumann language`），组合子（`combinator`），`lambda calculus`， π 演算，消息传递（`Message passing`）。

以上这些概念我不会一个个解释以及演示如何在 python 里面去运用，因为很多东西是 python 不自带或者不支持的，比如尾递归优化就需要借助第三方库，模式匹配的话 python 语法原生不支持。还有一些用的比较少，比如运算符重载（`override`）。简单说，关于递归，在 python 里面（`stackless` 除外）还是少用比较好（其他关键词都是打酱油的）。

读懂代码

OK，总算进入正题。看懂 python 代码，从本篇指南的例子开始。我相信有些人看不太懂前面图 1 的 `yield` 演示，于是现在我来解释一下。图片我就不重复插了，自己翻页就好（顺

¹⁸ https://en.wikipedia.org/wiki/Coroutine#Implementations_for_Python

¹⁹ 善用文档和搜索引擎：<https://docs.python.org/2/howto/descriptor.html>

便锻炼一下自己的记忆力 or 码字速度)。

首先这里有两个文件：`fib.py` 和 `fib_test.py`，放在同一个目录下。`fib.py` 定义了四个不同版本的函数。`fib(n)`的话是原始斐波那契数列的递归版本（前两个数是 0 和 1，第三个数是前两个数之和，第四个数是第二个和第三个之和以此类推），这个好懂，简单说逻辑是这样的：如果 `n` 小于 2 就直接返回 `n`，`n` 大于 2 就调用前两位的 `fib(n)`。`fibonacci(max)`的话用了个 `yield`，同时还用了个变量 `n` 来计数，算是标准的生成器版本。简单说逻辑是这样的，给定一个个数上限 `max`，第一次调用的时候，`yield a`，即为 0，后来 `a` 变成了 1，`b` 变成了 2，也就是下次调用函数的时候 `yield` 的就是 1，以此类推。后面的两个函数 `F1` 和 `F` 道理是类似的，只是没有用 `n` 来计数，`F1` 也能生成数列，不过在个数控制上会存在问题（你会看到底下输出的时候只有第三行是输出了四个数）。`fib_test.py` 是用来测试 `fib.py` 的，第一行是导入了所有 `fib` 里的函数所以在 `fib_test` 里可以直接调用。第三行就是一次 `list comprehension` 了，这个效率很低，因为 `n>3` 之后每次 `fib(o)`都要递归一次。第 4,5 行也是构造一个列表，方法一致。第 6,7 行可能不太好理解，因为用到了匿名函数 `lambda`。先从右往左看去，`F()`调用后生成了一个 `generator`，然后传递给了 `lambda`，匿名函数里就是调用三次 `__next__` 方法的意思。但你不能直接在 `lambda` 里面写 `F().__next__()`，因为这样每次都会 `F()`是一个新的生成器。

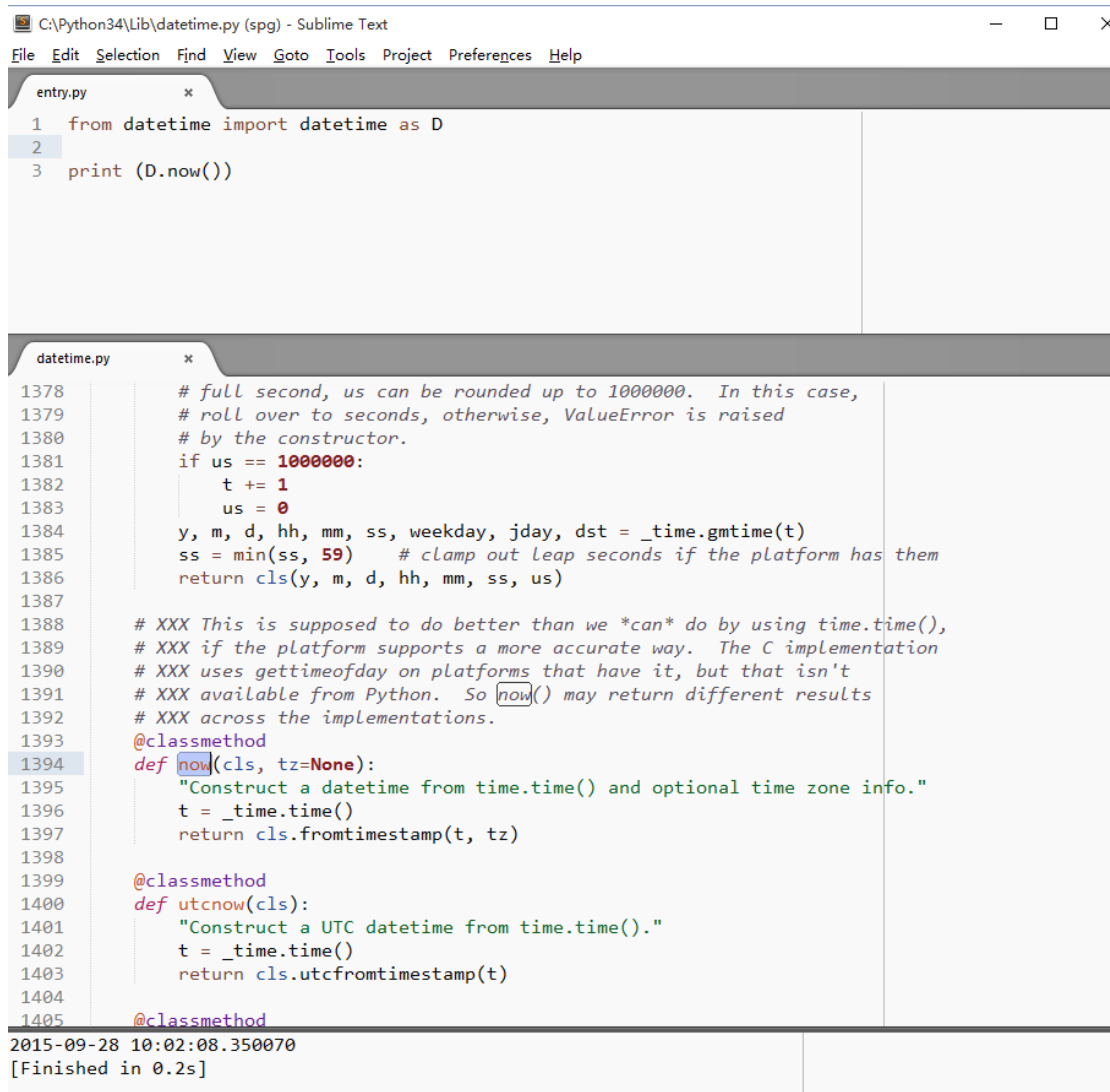
接下来解释一下图 2 里的装饰器。文件结构和图 1 的例子是类似的，不再解释。这里有两个装饰器，一个是函数 `D1`。另一个是类 `decorator`。先看装饰器做了什么，第 11 行前面的一次 `echo(T)`调用就是 `print` 出一句第 4 行说的文本，这里是给参数 `T` 做了强制类型转换的。第 12 行分号前面是用 `D1` 装饰后的函数 `e2(T)`，输出结果的变化在于原来 `echo(T)`的后面多了一句 `By: D1`。类似的，`e3` 是用 `decorator` 类来装饰过的 `echo`，区别在于 `e3` 是在调用 `echo` 前多输出了一行。简单说这两个装饰器的作用都只是在给做原函数做类似打补丁的事情，当然装饰器的复杂用法也有，比如用来模仿模式匹配的效果²⁰。现在来讲基本原理。其实构造的原理是这样的，我们先看右边的 `deco_test.py` 里的第 9 行和左边 `deco.py` 第 7 行开始的 `class` 定义，这个是装饰器的本质。`echo` 函数作为 `decorator` 实例构造的参数传到了 `__init__` 方法里，然后绑到了实例自己的 `fn` 属性上。然后在实例调用的过程即 `__call__` 方法中，先于 `self.fn` 调用了一句 `print`。函数构造装饰器的方法也是类似的，`D1` 里面的 `W` 函数（一般叫做 `wrapper`）实际上就是个包装，最后用 `return` 的 `W` 来替代原函数。而 `@` 语法糖实际上就是做了个同名函数替换的过程，即右边第 6、7 行可以理解为：先定义好一个 `e2`，然后让 `e2 = D1(e2)`。

以上差不多是前面代码的讲解，如果搞不懂的话自己写一遍试试，或者 Google 一下。回顾前面我写的代码，如果你了解 `pep8` 的话，可能会替我的人身安全担忧。是的，我之所以这么不符合规范地写 `python` 代码的用意，是在于让大家了解 `python` 的语法其实没有那么死，至少没有传说中那么恶心。你可以用逗号，用分号，用反斜杠，或者写一堆 `lambda`，甚至用 `type` 函数来定义类²¹，只要不影响代码的组织结构和 `debug`，我觉得都是可以的。

²⁰ 许多 web 框架都有类似的设定，比如 `flask` 的 `@app.route` 和 `pyramid` 的 `@view_config`

²¹ 用 `type` 函数来定义类属于黑魔法范畴，最简单的形式是 `a = type('a'(),{})`，感兴趣的自己去了解。

当然，就这点代码量不足以认为可以读懂大多数的 python 代码，所以这里我觉得有必要多加点示例。首先是 python 自己的标准库。首先举个比较简单的例子：



```
C:\Python34\Lib\datetime.py (spg) - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help

entry.py
1 from datetime import datetime as D
2
3 print (D.now())

datetime.py
1378 # full second, us can be rounded up to 1000000. In this case,
1379 # roll over to seconds, otherwise, ValueError is raised
1380 # by the constructor.
1381 if us == 1000000:
1382     t += 1
1383     us = 0
1384 y, m, d, hh, mm, ss, weekday, jday, dst = _time.gmtime(t)
1385 ss = min(ss, 59) # clamp out leap seconds if the platform has them
1386 return cls(y, m, d, hh, mm, ss, us)
1387
1388 # XXX This is supposed to do better than we *can* do by using time.time(),
1389 # XXX if the platform supports a more accurate way. The C implementation
1390 # XXX uses gettimeofday on platforms that have it, but that isn't
1391 # XXX available from Python. So now() may return different results
1392 # XXX across the implementations.
1393 @classmethod
1394 def now(cls, tz=None):
1395     "Construct a datetime from time.time() and optional time zone info."
1396     t = _time.time()
1397     return cls.fromtimestamp(t, tz)
1398
1399 @classmethod
1400 def utcnow(cls):
1401     "Construct a UTC datetime from time.time()."
1402     t = _time.time()
1403     return cls.utctimestamp(t)
1404
1405 @classmethod
2015-09-28 10:02:08.350070
[Finished in 0.2s]
```

图 3: datetime 模块源码概览

图中上半部分的代码是一次调用标准库的示范，即调用 `datetime.datetime.now()` 方法。假设你不知道 `datetime` 这个模块怎么用，最简单的方法其实就是去查文档或者网上找资料。稍微原始一点的，可以在命令行里面调用 `help(datetime.datetime.now)` 或者 `dir(datetime.datetime)` 这种（如果用 `ipython qt console` 的话在键入过程中会带有提示的），更原始的就是上图这种查看源代码的方式了。

但是具体来说，怎么找源码呢？最土的方法，自己在 python 安装目录找（通常就在 `lib` 文件夹里），或者，借助操作系统或第三方软件来搜（我偶尔也会用 `everything` 来找 python 的模块文件），又或者，借助编辑器/IDE 的功能，这个 `emacs` / `sublime` / `pycharm` 都有，不过个人还是习惯在 `pycharm` 里面翻源码：

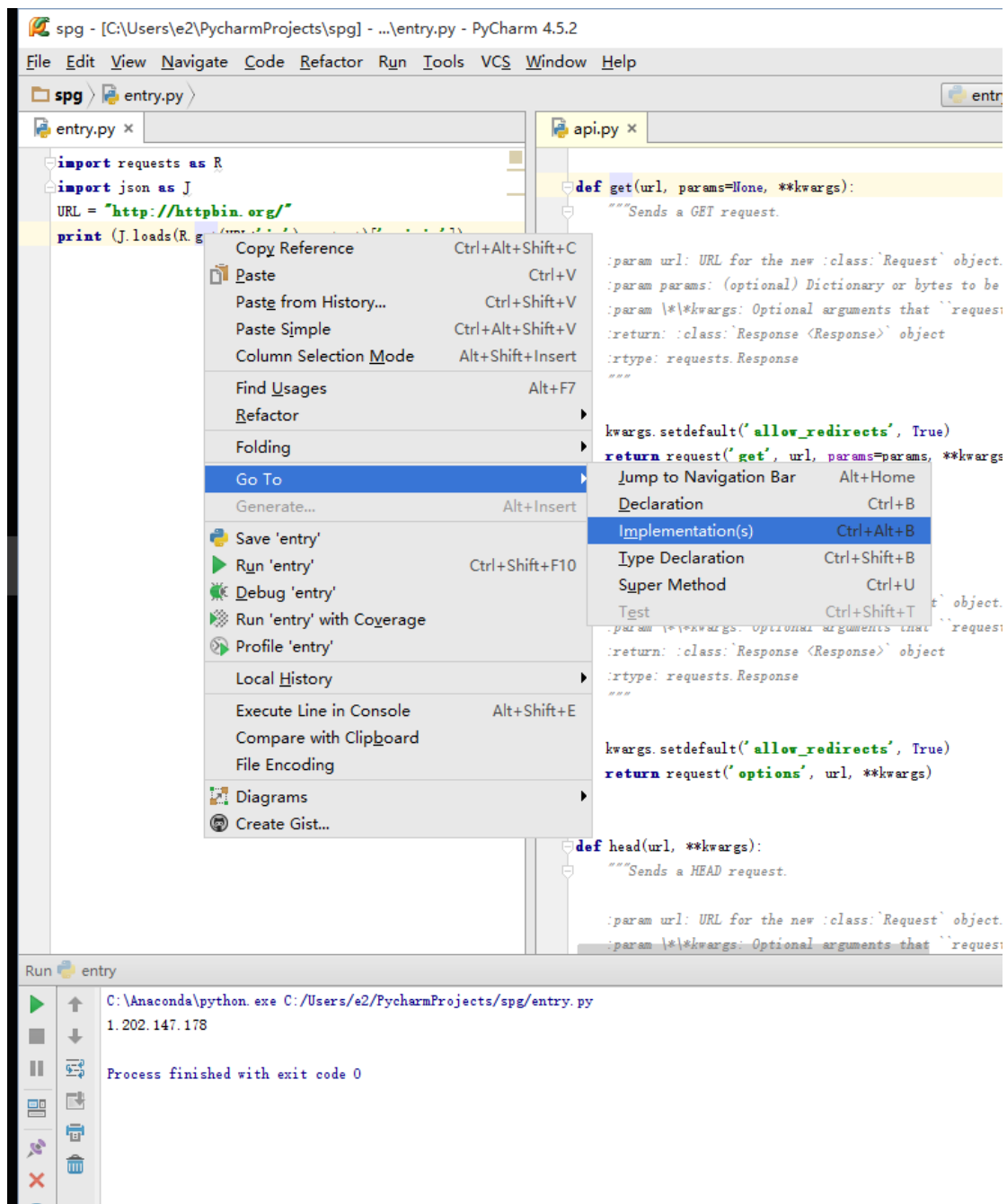


图 4: pycharm 下查看第三方库的源代码

简言之，对于想快速上手 python 的人来说，anaconda+pycharm+zeal 无疑是最佳组合。

上述所讲主要是读 python 代码的方式，没有讲到具体要点。其实要读懂 python 代码不难，通常（对于一个有规模的 python 项目来讲）情况下，首先，有文档的先看文档，因为文档一般都有 api reference 这一块，这样可以快速理清目标的代码结构，当然如果没有文档的话，就靠 pycharm 之类的。通常大部分 project 都会带有一部分测试代码，通过测试代码来反查也是一种思路。当然，以上思路对于某些具体的代码片段并不管用，尤其是某些算法题。通常看不懂 python 代码有两种情况，一是不知道整个代码是怎么工作的，二是看不

懂这里面写的是什么意思。前者基本上方法和思路已经说了，现在来讲下后者。先举个简单的例子：

Reverse Words in a String

一行代码解决。。。

class Solution:

def reverseWords(self, s):

return ''.join(s.strip().split()[::-1])

发布于 2015-09-10  收起评论  感谢  分享  收藏 · 没有帮助 · 举报 · 作者保留权利

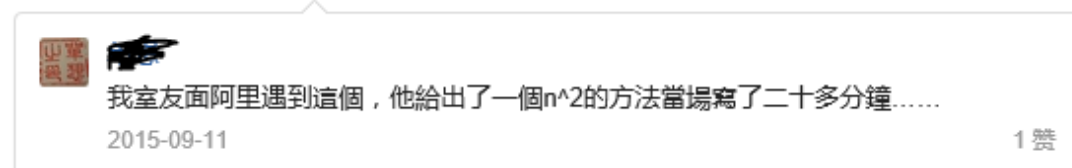


图 5：单行代码示例

上图的单行代码如果看不懂，分解就可以了。`str().strip()`会去掉字符串两端的空格（因为空格是默认参数），`split()`会以空格分割字符串列表，最后那个`[::-1]`前面讲过了，一个简单的逆序方法，最后`.join`把前面的列表合并成一个字符串。

上面这个例子还是很好懂的，因为涉及的冷门技巧不多。现在再换一个：

Golfed version (157 strokes):

```
class Solution:generateParenthesis=lambda _,n:s(n,n)
s=lambda o,c:o+c<1and['']or(o
and map(''.__add__,s(o-1,c))or[])+(c>o
and map(''.__add__,s(o,c-1))or[])
```

Golfing my [Solution 3 from here](#) (134 strokes):

```
class Solution:generateParenthesis=g=lambda s,n,o:n>0<=o and[''+p
for p in s.g(n-1,o+1)]+['']+p for p in s.g(n,o-1)]or['']*o]*(n<1)
```

132 strokes (saving 2) thanks to generating the strings "from right to left", see comments below:

```
class Solution:generateParenthesis=g=lambda s,n,o:n>0<=o and[p+'']for
p in s.g(n-1,o+1)]+[p+'('for p in s.g(n,o-1)]or['']*o]*(n<1)
```



answered Sep 15 by [StefanPochmann](#) (109,520 points)
edited Sep 16 by [StefanPochmann](#)

[ask related question](#) [comment](#)

Err you're right, s/code golf/one liners/. Thanks for the golfed version, I think the `map` together with `''.__add__` and `''.__add__` are too verbose for a golf version anyway.

图 6: leetcode 代码讨论概览

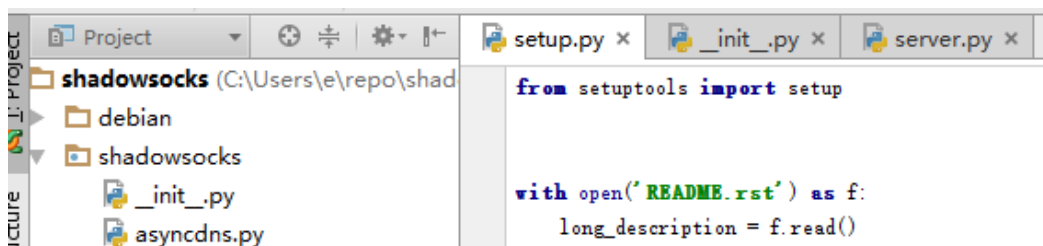
老实说,对于上面这种代码,我不指望一下就能看懂,其实,对于多数情况,只要会用了改就可以了。即,读懂 python 代码的关键,还是要会用先,再是改写,读懂是其次。

本来关于读代码,我觉得讲这些就差不多了,剩下的可以多练习练习,找些简单的项目源码看起。本来有想过举些稍微有点代码量的实际项目做例子的,但是感觉单讲源码分析的话会写好长,也就不好详讲,最后一个例子我们来简单分析一下某 socks 工具的源码:

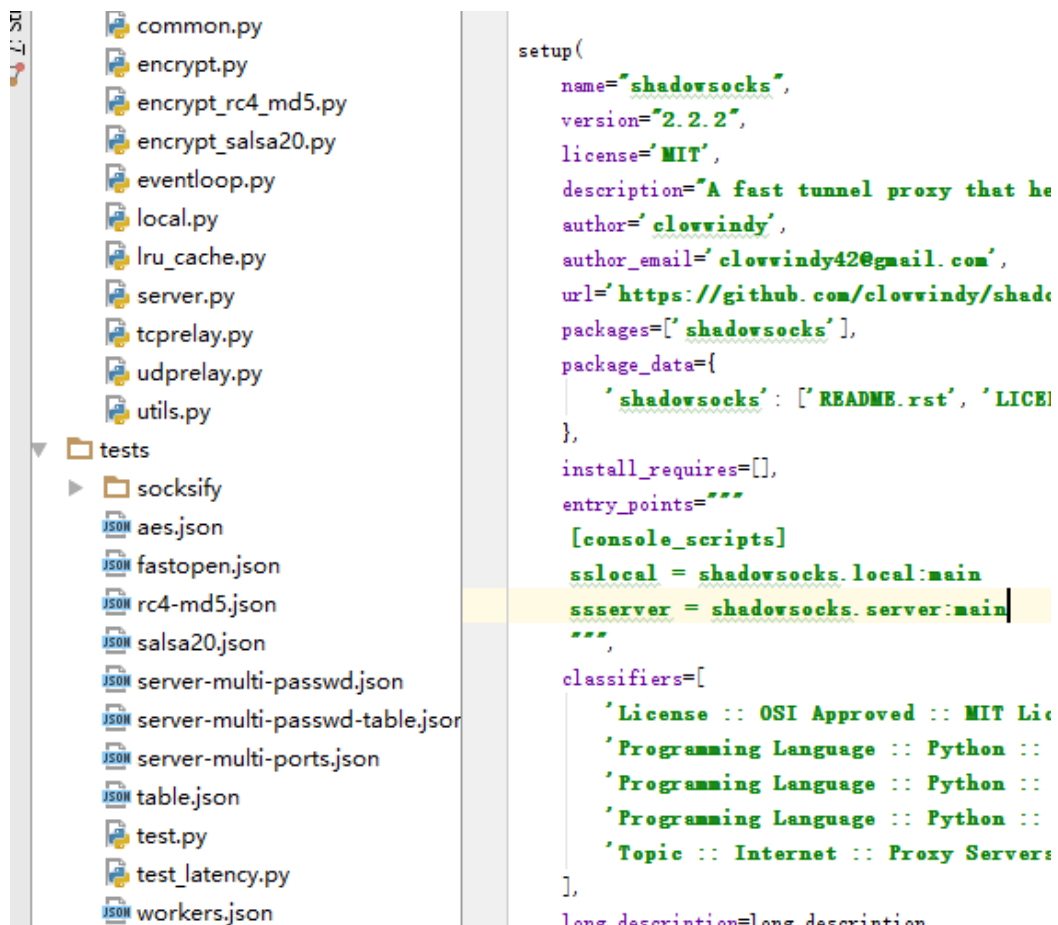
1. 首先我们要把源码下载下来,这里只需要 git clone 一下就可以了:

```
e@AIIPC ~/repo
$ git clone "https://github.com/e42s/shadowsocks"
Cloning into 'shadowsocks'...
remote: Counting objects: 1788, done.
remote: Total 1788 (delta 0), reused 0 (delta 0), pack-reused 1788
Receiving objects: 100% (1788/1788), 393.60 KiB | 130.00 KiB/s, done.
Resolving deltas: 100% (1141/1141), done.
Checking connectivity... done.
```

2. 用 pycharm 打开目录,上来先看一眼 setup.py²²以及文件结构:

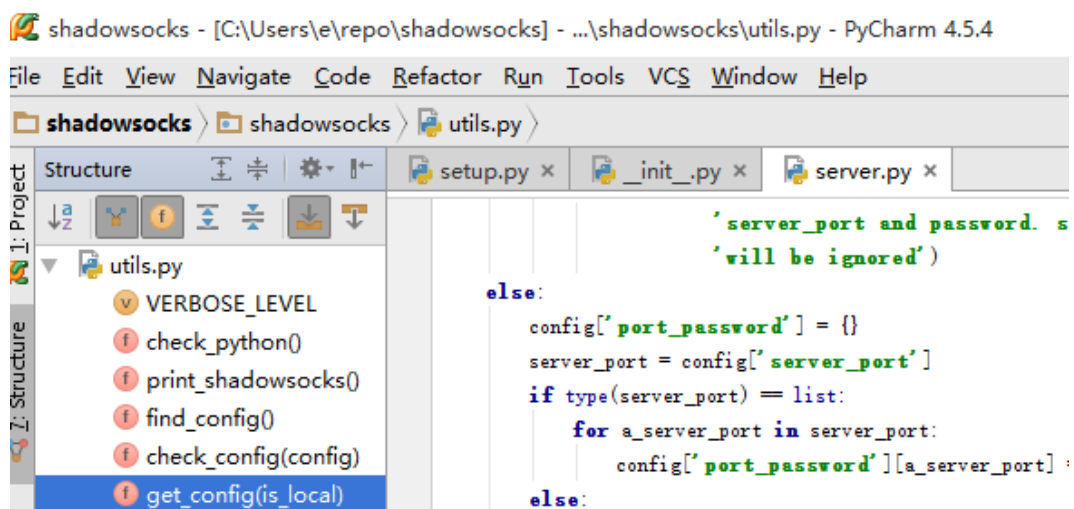


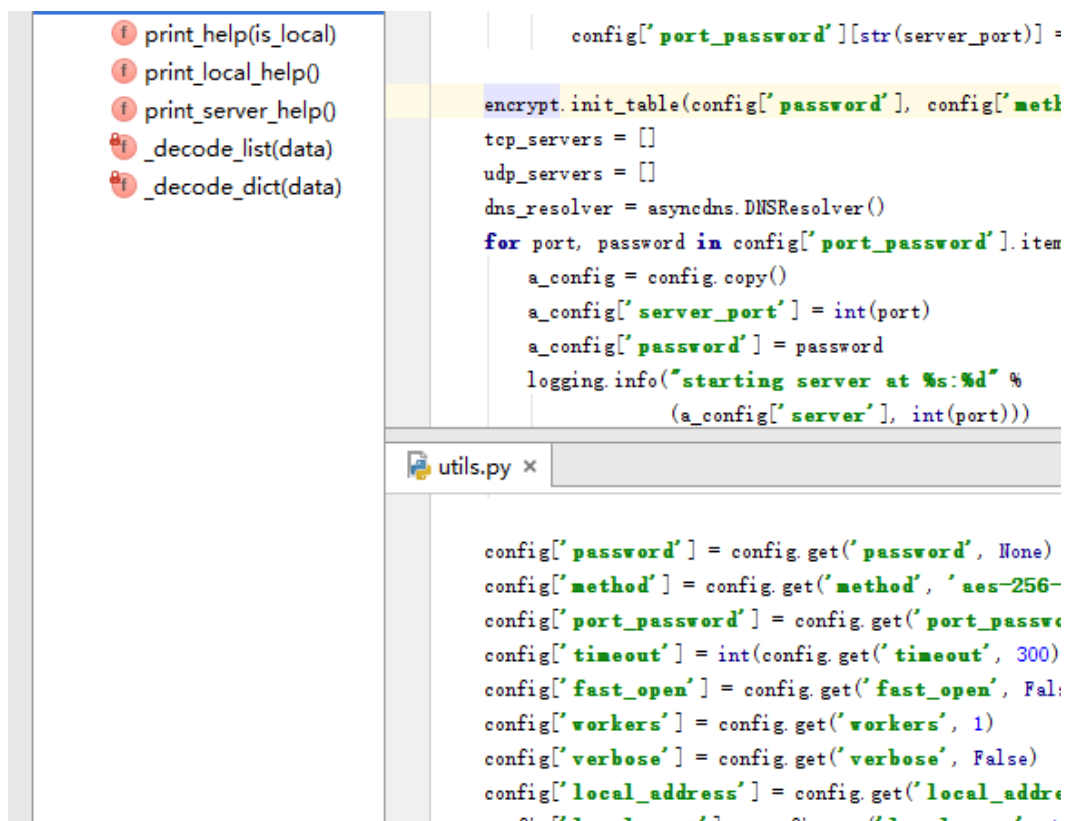
²² 关于 setup.py 会涉及到 easy_install 和 setuptools 之类的东西,这个后面再补充说明。



这一步是为了弄清楚 package 的依赖关系（不过这个项目的 install_requires 是空的所以也就没什么依赖了），有些项目可能会把这部分抽离到一个 requirements.txt 里面。然后我们还可以看到 entry_points 里面有两行，对应生成两个可执行命令 console_scripts。

3. 接下来就看项目的具体用途而定了。如果是一个工具型的库（比如 SQLAlchemy）那么就从__init__.py 入手，当然也有人不喜欢在__init__.py 里放东西（比如 django），那就对应的各找各咯，思路是一样的，找一个入口。而对于非工具类的（比如我们现在看的这个），也有对应的入口，那就是它的执行文件：





然后再一步步深入，比如具体上面的 `tcp_server` 要怎么实现。

4. 最后就是改动代码和反复调试了，这里截图演示不了，就略过了，其实是最关键的一步。不过没关系，大家可以当作联系。不过上面这个例子的代码量并不大，如果觉得这个代码量不够塞牙缝的，可以换一个：<https://github.com/Pylons/pyramid>

以上都是比较基础的分析方法，没有借助到 `pycharm` 之类的 IDE 独有特性。

最后补充一下。要读懂代码呢，基础是很必要的，而基础之中的基础就是规范了。说道规范，首先是注释。`python` 的注释分两种，而且差别其实还挺大的。注释也有很多用法，可以方便调试，可以写文档，可以做文档测试。不过总的来说，规范只有一条，那就是文档注释用三个双引号（以及前面提到过的四空格缩进），不要用单引号，具体为什么，可以自己留到后面去思考，也可以参照 `pep8`，这里不解释。

代码调试也是基础，新手上路，首先要习惯看 `trackback` 异常。单条异常日志要倒着看，进阶之后，可以自己试着用日志来替代调试过程中临时写的 `print`。往后，用 `ide` 的 `debug` 功能或者各种 `debug` 工具（`gdb`、`pdb`、`pudb` 等等），或者自己手动监视 `locals()`，再往后，也可以引入 `inspect` 和自省（`interrogation`）。当然，个人目前还介于 `print` 和 `log` 之间的位置，感觉基本够用了...

再就是熟悉标准库，需要的时候做点笔记也是应该的。虽然对于大多数人来讲每星期写点技术总结很困难（包括我），但试着写写还是很有帮助的，无论内容的好坏。关于读代码的部分就讲到这里。其实从注释开始，读代码和写代码之间的章节界限就已经模糊了。

写代码

这里我只说一条原则：若无必要，勿增实体。或者反其道行之。不过由于 `python` 的语法限制，通常还是精简比较好。如果是元编程吐出来的 `python` 例外。

我记得有些 `c` 语音或 `java` 书会教你先不要用 IDE，用记事本之类的文本编辑器写起，这个其实是有一定道理的，然而，这个道理并不太适合 `java`，因为 `java` 的大部分代码并不能在一行之内搞定。当然，很多时候单靠人来纠正代码里的错误还是比较费时间的，于是就有了 `pyflakes`、`pylint` 之类的第三方工具。另外，说到文本编辑器，`windows` 自带的记事本并不是一个很好的选择，如果觉得 `sublime` 比较重量级(?)，那么 `notepad2-mod` / `notepad++` 之类的或许比较适合你。

当然，我还是假定大家可以先试用一下 `sublime`，因为这个文本编辑器的扩展性还可以，且不如 `emacs` 那么难上手。一开始写代码，对于完全没有编程经验的人来讲，难免会有语法错误 (`syntax error`, `indentation error` 等等)，这个还是先适应一下比较好，不然如果一上来用 `pycharm`，结果碰到一个只有 `vim` 还没有安装权限的 `linux` 虚拟机，那就准备哭吧。关于工具的选择，无论是软件(以上)还是硬件(`ssd` 或机械键盘)，到头来还是看个人需求。

除了语法错误，还可能碰到很多其他的错误，解法嘛，一般先自己思考，不行就网上找找。当然，养成良好的代码组织习惯之后可以避免一些错误的发生。比如有的时候一个分支逻辑判断写太长，那就最好拆开来，封装涉及的实体比较多，就不要用太多继承。

说写代码的话，差不多就这样了。毕竟编码(`coding`)在我看来是一件很枯燥的事情，建议有目的性的写或改，比如你喜欢或有意向搞数据分析，那就不要开多一个窗口去折腾 `web` 框架。

写代码的风格也是因人而异的，有人喜欢上来写大纲一样先把结构列出来，有人喜欢编写边调试。一般来说，对于较大的项目，后者比较实际点。这里说的写代码风格和前面提到过的编程范式不是一个意思，编程范式的话有很多种分类，`OO` 还是 `FP`，命令式还是声明式等等。个人偏向于声明式+函数式的组合。当然这里说到优劣，也还是见仁见智的。比如我虽然习惯函数式，但并不死磕纯函数无副作用这些，毕竟实用就好。总之，接触的编程语言越多，自己的编程范式会逐渐趋于稳定，这算是从我个人经历总结出来的吧。

最后再补充点相关知识。如果需要长期编写一个项目，无论是否团队协作，最好还是引入版本控制系统(`VCS`, `version control system`)，方便做变更日志记录。比如今天做了哪些改动，还差哪些没实现等等。虽然这不是 `VCS` 的标准用法，但也不违反规范。一般推荐用 `git` 和 `hg`。`hg` 的话比较适合那种单人玩一玩性质的项目，因为配置很简单。如果一开始定位就比较高的话，还是用 `git` 吧，这个基本上已经成为业界标准了，省得以后还要迁移。

另外，不要懒得写文档写测试，哪怕只有一点点。虽然很多时候自己也不写测试...

好啦，回到本小节最开头那句话，其实就是控制复杂度的另一种表述形式吧，或许这样说会比较易懂一点。

还有，这里可以顺便留点简单的作业题：

1. 写个标准的冒泡排序（可选：用递归实现）
2. 用一行代码，把当前目录下文件名中的某段固定字符串批量删除掉，长度越短越好。
比如：abc-xyz.mp3 和 bcd-xyz.txt 中的 xyz。

误区

这里主要讲些常见的关于 python 的误区。首当其冲是 python2 和 python3 的争执。其实现在 python3 都到 3.5 了，随着新特性越来越多，守旧的人也只会继续守下去，比如我，不过个人同样支持 python3 的发展。之所以不用 python3 是有原因的，如果你开发的环境需要用到很多历史悠久的第三方库，那么用 Python2 通常要保险一点。如果单纯是运维的话，一般没问题，因为多数第三方库存在的问题都出现在那些关键特性的变更上，比如字符串。当初碰到过 BeautifulSoup4 在 3.4 里面处理的结果和 2.7 完全不同情况，最后实在无解了只能把部分方法覆盖掉敷衍了事。现在 python 在 web 领域的地位已经受 js 和 ruby 冲击得所剩无几了，除了爬虫代理各种小工具各种成熟的库以外。而在科学计算和数据分析领域，python 这几年倒是增幅比较大。换句话说，工具总是要适应场景的。

关于 OOP，python 虽然也号称一切是面向对象的，其实并没有那么纯粹。当然个人认为还是不纯粹的好。习惯 java 的人可能觉得 python 里面没有 interface 会有点奇怪，不过要说 python 没有 interface 也不完全对，只是没有严格的实现而已。有个第三方库叫 zope.interface，可以满足大部分需求（虽然这个库的主要功能不局限于此）。一般的 python 项目需要用到它的机会不多，也只有在工程量比较大情况下才会用到，比如 twisted 这样的。有人可能之前看到 zope 就觉得卧槽这是什么鬼还能用吗，不过这又是另一个误区了。

续上。很多人觉得 python 的所谓的哲学就像《unix 编程艺术》里面描述的那样，差不多可以用 simple is better 来描述。其实这是一种片面的认知。首先我觉得编程语言根本就没什么哲学概念一说，用比较形象的词来说，只是一种虚无缥缈的情怀罢了。具体到 python 的话，只要它存在着 c 扩展，那就不可能 simple 下去，只能一味的掩盖罢了。所以真正要用好 python，必须要直面 python 的历史。而讲到历史，这里有必要概括性科普一下 zope。

Python 的早期发展和 zope 有千丝万缕的联系。Python 的作者 GVR 在 1994 年发布的 1.0 版本，那时候还在 CWI (Centrum Wiskunde & Informatica)，后来去了 CNRI (Corporation for National Research Initiatives)。2000 年左右，在发布了 2.0 之后，pythonlabs 的一席人加入了 Digital Creations，也就是 Zope 公司的前身。2003 年，在完成 python2.3 之前，GVR 离开了 zope，但当初那些 python 的核心开发人员不少还在这家公司。好吧，其实 zope 是一

家公司，也是他们做的一套 web 框架体系的名字，自然也是最早的 python web 框架之一。

Zope 的大致思路就是仿当时的 J2EE，把 python 当 java 使。这个其实是可行的，而且效果还不错，但是社区的大多数人都不太接受这种又带接口又是 xml 的玩法。Zope2 可能接受的人还有，毕竟很酷炫还实用，直接能在浏览器里面修改业务逻辑。但 zope3 反而更激进了，于是到最后 zope3 也只相当于是个半成品。Python 社区里现存的和 zope 相关的，还有人记得的，估计就剩 plone 了，但这个相对还是太复杂了，现在的人一般搞 cms 基本都不会考虑它。所以简单说，zope 沦落到没人用的地步只能说太超前了，nosql (ZODB)、在线写代码这些它都老早就实现了，但九几年的时候，带宽还是个问题。超前也就算了，对社区也不是很重视那就只能作死了。往后的框架，中期冒出来的 Django 现在的普及度就很高，它是仿 ruby on rails 的。后期的框架都是比较轻量级的，诸如 flask, tornado 这些。实际上 zope 这一支还没死，zope3 之后有了 reposer 这个项目，把 zope 的遗产解耦了，到现在还有人用的主要就 ZCA²³ (对应 zope.interface)、ZODB 和 traversal²⁴这三个了。Reposer 里面有个 web 框架叫 bfg，后来跟一个中后期 web 框架 pylons 合并了，成了今天的 pyramid (我主要用的就是这个)。虽然 pyramid 相对 django 来说可能没那么方便快捷，不过熟悉的过程中倒是让我对 python 的发展有了更深刻的认识。

科普 zope 就到这里，我想说的其实只是希望做选择的时候还是尽量盲从 (没看错，这样省时间)，但还是要广开思路，不要局限了自己的视野。Pyramid 是我觉得 python 里面最实用的 web 框架，但限于历史背景，很多人都对这个代码量巨大的框架有一种畏惧心理，这大可不必操心。经得起时间考验的开源项目就这个好处，实在不行了自己改一改就好。

关于 python 的语法，误区不多，缩进、分号都不是什么大问题。坑人比较多的还是逻辑判断，比如前面有提到过的 is 语句。if 语句也是，有时候省略一些描述觉得像 if a 这样会省事，但需求改了之后可能限定 a 的条件可能就是另一个意思了，因为 if 语句默认 None 和 False 是归为一类的，还包括空列表等等。

关于 python，我觉得最大的误区可能还是认为什么都可以用 python 来做。其实也不是不可以，但多数情况这是不科学的。我举个例子，VCS 里面，现在主流的三个有 SVN, Hg 和 git。豆瓣早年觉得 hg 是用 python 写的，应该好扩展吧，结果最后还是把代码库整体切换到了 git 上面。有时候就是这么现实。类似的还有个例子，现在自动编译代码或者部署之类的，多少会用到 makefile，但是 python 历史上出现过 buildout 这样的东西，就是完全靠 python 实现项目的部署等一系列操作。我觉得这个没有流行起来是也有原因的，既然 makefile 用的人那么多也不难学，为何要多了解一个更复杂的体系呢？

还有，有的人觉得有了 pip 就可以不用 easy_install 了，实际上这么讲也不科学。而实际情况，easy_install 要比 pip 好用的多。因为多数环境下你根本不需要用到 uninstall 这个功能，

²³ ZCA = Zope Component Architecture，现在一些大型 python 项目比如 twisted 也还有用到这个。

²⁴ traversal 是一套路由机制，区别于传统的 url 匹配，采用对象目录的方式进行分发。豆瓣用的那个 quixote 主要也是模仿了这一套机制。

而且，现在的 `virtualenv`²⁵都是一次性的，出问题了就重新建一个，也不费事。

效率问题可能也有很多人在意，但也有很多人不在意。而我觉得，不把效率当回事才是真正的陷入误区了。有些事情，能用一个 `for` 循环何必用两个呢？可以硬编码的，何必要放到数据库里面？所以说，环保节能，要从我做起。

设计模式也是常困惑新手的一个问题，`python` 到底需不需要类似 `java` 里面那样的设计模式。其实是有的，但不多，一般只要搞懂 `adapter` 和 `factory` 就够了。认为需要完全照搬或者完全不需要的，那都是走极端。

另外，有些人觉得 `python` 在 `windows` 下开发会很痛苦，其实那只是一些没用过 `msbuild` 的人的臆想。当然，在 `windows` 系统上摸索编译 `c` 扩展是一件很累人的事情，这里给个参考²⁶，大致的流程就是：装好 `visual studio`（2008 以上的 `express` 或 2015 社区版都可以），还有系统对应的 `windows sdk`（一般来说 `win7sdk` 比较通用）；然后打开 `vs` 的命令行（分 64 位和 32 位，这里以 64 位为例），设置两个参数（`set DISTUTILS_USER_SDK=1` 和 `setenv /x 64 /release`），然后再用 `python setup.py` 就可以了。

而说到误区，我觉得最严重的，可能是关于 `python` 所谓的胶水语言的这个定位。很多人以为 `python` 作为胶水语言是为其他各工具集之间建立桥梁，实则不然。我觉得胶水语言的本质，是用来缝合历史遗留问题与当今环境之间裂隙的，说简单点，就是用来弥补一些类似 `c` 语言这样的旧工具的不足之处，以便让这些旧工具能适应新时代的需求。

最后，请不要忽略测试代码的重要性。再次强调。但，是不是说代码测试通过了就一定管用呢？这个还看你的测试覆盖率，但，即便是 100% 的覆盖率，也不等价于你的代码就没有任何问题了。这里没有绝对的因果关系，只有可能和不可能。以上内容主要是对新手而言的，如果你觉得这些误区你都了解，那么就该看一下沈巍²⁷当年留下的幻灯片《`python` 编程艺术》里面关于误区部分的深入探讨了。

参考资料

本来想在这里留一些参考资料的，但是想想，除了那些文档链接以外，好像也没什么特别的。找东西，能用 `google` 就 `google` 吧，不能用就先翻过去²⁸。以下可以算作附录，也就列举写 `python` 相关的第三方库和工具的名字：

- ◆ 数据 / 计算： `sqlite`、`bsddb`、`mysql`、`sqlalchemy`、`redis`、`mongodb`、`zodb`、`numpy`、`scikit`、`pandas`、`PIL`、`matplotlib`、`openxml`、`cython`、`pypy`

²⁵ `Python3` 自带 `venv`，不过大部分人都还是统一用这个。

²⁶ <https://github.com/cython/cython/wiki/CythonExtensionsOnWindows#using-windows-sdk-cc-compiler-works-for-all-python-versions>

²⁷ 关于此人的信息，后面还会提到。

²⁸ 这里我是不会教你的，自己摸索是个很有意思的过程，实在搞不定再找我。

- ♦ 实用 / 基础: anaconda(conda)|ipython(notebook, kernel)、pywin32、setuptools、virtualenv、pip、buildout、mr.developer、nose、mock、uncompyle、pdb、pu db、fn、toolz、patterns、pysistent、pp、multipledispatch、gevent、pyqt、pygments、sphinx、you-get、sh、zope.interface
- ♦ Web / 网络: django、flask/jinja2、bottle、tornado、pyramid、zope2、webtest、beautifulsoup、requests、twisted、scrapy、httpie、oauth、lamson、shadowsocks、celery、rabbitmq、fabric、saltstack、sentry、planet、pelican

暂时就想到这些，我书读得少，见识不广，有些库也只是道听途说，和 web 开发不相干的就更知道了，以后有好东西再慢慢填充吧。顺便科普一下 pypi(python package index)，这个相当于 node.js 里面的 npm，用 easy_install 安装东西，默认都要先在这里面找一遍。当然，如果自己用东西打算开源，除了在 github 上放代码，也推荐在 pypi 上传自己的包。

新手路线

我把新手路线放到最后是有用意的，如果前面不看完的话，无论是不是新手，这一节对你都没有意义。而且对于不同程度不同领域来说，每个人的路线也是不一样的。每个人都有每个人自己的路，而我所讲的路只有一条，因为那是我自己的，所以肯定不是你的。

前面我已经大致说过我的经历了，不过很简略，没有涉及到一些有用的细节，所以这里我想补充一下那些我没讲到的比较有用的细节和一些比较没用的个人往事。So，如果不感兴趣的，跳过或无视这段就好，直接看下一节。

最开始我没打算用 python 来做网站，那是 2011 年的夏天，高考后的我还在打算学习 php 的正确用法。可惜半年过去了，PHP 没能有多少进展，SAE 上的应用一直闲置着。我试过去看一些国内的 PHP 视频教程，可惜他上来就复制粘贴了几行 jquery 和 mysql 代码，看得我很纠结（最纠结的还是他们的幻灯片 banner 设计太挫我有点不能忍）。我也还试过去看一本叫 head first php 的书，然而影印书的厚度加上枯燥无味的内容导致我根本没心情看完（讲真，这系列的书其实很水的（也可能只是不太适合非拉丁语系的读者），还不如那个 xx x for dummy 系列）。

然后我就在思考一个问题，或许我（当时）不太适合学（用）PHP。于是我想，不如退回到之前做选择的阶段，而既然要选择，不如借鉴一下其他网站，然而机缘巧合就想起了豆瓣（之前之所以选择用 php 很大程度受了百度、wordpress 和维基百科的影响）。然后一看豆瓣是用 python 写的，觉得挺有意思，就这么开始了一条新的不归路。

那是 2012 年的春天，去图书馆还了那本 PHP 砖头书之后，我就另外找了本 python 砖头书（封面是只老鼠，应该是 python 学习指南），结果...半年后，依旧没什么进展，可以说一行 python 代码都没写过。我停留在了一个很疑惑的阶段，因为感觉 python 好像很好懂的样子，但是要怎么做出一个网站来，感觉像是隔了几座山。于是为了不阻塞这个环节上，我就开始去了解怎么部署一个网站。快到期末的时候，买了个 VPS（其实这时候真正的动力不

是部署网站，而是科学上网...），发现 linux 命令行挺有意思的，就开始玩这个了，结果没停下来控制好节奏，然后物理也就没花心思复习，挂掉了（这是我挂掉的第一门课）。

Linux 其实早在 09 年的时候已有接触过了，不过那时候也仅限于玩玩图形界面，试下双系统而已。真正接触 linux shell 还是从这时候开始的，因为 VPS 没有 GUI（即便是 x11 forward 或 vnc，效率也太低了，毕竟服务器在美帝那边）。于是乎，在这种艰难且优越的条件下，我学会了 Vim 的基本用法之类的，后来不记得是因为什么的影响（可能有打算研究内核或者只是图个稳定也可能只是为了个 ZFS²⁹），还把 ubuntu 换成了 debian 再换成了 freebsd³⁰。

这一时期，顺着豆瓣的思路，对 python 的生态环境（ecosystem）开始有了初步的了解。那时候感觉豆瓣当年的选择还是不错的，用了 quixote³¹和 twisted 而没有用 zope2（那时候还没有 django）。而在了解过 django 之后感觉这类型的框架并不太适合我，flask 貌似可以，于是就试着跟 tutorial 走了一段时间（虽然最后并没有多少进展）。

同一时期，我在学院的科技协会认识了一个技术相对很厉害的学长（id: jasl），我是 11 级的，他 09 级，那时候他大三。相对我而言他经验已经很丰富了，写过 python，当时已经开始转 ruby on rails 了。同时，我也加入了他在科协带领的一个部门，叫程序兴趣小组。这个事情对我的帮助其实挺大，就跟他的第一次对话，便我知道了 sqlalchemy 这个东西往后必然有大用处（之前我是没听过的），他也提到了 django，还有 rails 等等（之前我也听过 rails，但相对来说还是更喜欢 python），总的而言，帮助还是挺大的。

However，即便如此，也没能促使我写下一行真正有用的代码。而真正促使我去写下 python 代码的，是暑假参加一个公益项目时候的事情，我在那边整理维护一个小型图书馆，但是当时的图书管理系统并不怎么好用，于是就想着怎么搞可以方便点。那时候找过一些现成的方案，但都好像不太适用，于是就自己想着怎么自己做一个，然后开始自己着手做一个图书管理系统。一开始我想这是尽快重建目录，于是就用几个 excel 表格简单模拟了一个系统，但很多操作还是需要手动完成（那时候不会 vba 啊，只用了几个稍微复杂点的公式，如果会 vba 的话我估计后面应该就不需要 python 了）。

这时候时间已经转到了 2012 年的九月中旬。大二的课业开始变得繁重，而我的心思感觉根本回不来了，类似这样的杂事让我越想越投入。到了秋天，我的思维开始变得发散，于是总能了解到很多有意思的东西，比如那时候我就发现了一个叫 pyramid 的 web 框架，粗略的了解了一下感觉挺适合我的，于是就它了。但这个框架有个问题，代码巨多，文档忒长³²... 所以最后的结果是，大约一个月的时间，我都没能搞懂这个框架的正确用法，然后，我休学了，虽然不是因为这个文档（具体原因可能只是先事情多太烦了），但整个管理系统的

²⁹ ZFS 是个很牛逼的文件系统，sun 之前的遗产之一。当年 opensolaris 以及基于它的 smartos 还没成型，即便不是学院派，freebsd 似乎也是最主流的选择。

³⁰ 最后用的最多的还是 ubuntu。现在主要用 suse 和 windows server，主要是 kvm 的费用开销有点大。

³¹ Quixote 这个框架就是相对当年 zope 的自嘲（堂吉珂德 vs 风车巨人的意味），这个框架一开始我也试过，没搞懂。现在虽然能搞懂了，但是相对 pyramid 来说没啥实用价值。

³² 只是相对其他 python 框架，不能跟微软比：[//media.readthedocs.org/pdf/pyramid/latest/pyramid.pdf](http://media.readthedocs.org/pdf/pyramid/latest/pyramid.pdf)

开发进程确实被这个框架的学习进度卡住了。

最开始预想那个 excel 的模拟系统最多只能用三个月，因为按照书籍的流通和数据量的增长速度，很快就会扛不住的。然而，最后这个 demo 都不如的半成品用了差不多一年... 所以，在 13 年初的时候我也意识到这个问题，于是也考虑过换用国产的 uliweb 来快速实现，不过最后还是换回来了。因为这时候，我好像开始考虑了很多看似有必要实际上太超前的问题，比如分布式存储，数据同步，缓存等等。实际上这些都可以该需求来妥协，然而那时候脑子没能转过来。倒是用 pyramid+zodb 开始写一个新的系统（原本是 couchdb，不过最后还是用了 zodb）。是的，从 12 年到 13 年整整一年的时间，我都没有写出过一行真正有用的 python 代码。不过一旦开始写了，进度也挺快，代码量很快就破了 1k 还是 3k(已经记不清了，不过很多代码都没有起到实际的作用，倒是感觉不知不觉堆出了个自己的函数库)，然而，我又意识到一个问题，就算后端写好了，前端的交互有点跟不上进度了。而当我开始补救前端的时候，又到了 13 年的暑假。

其实 13 年上半年，还有个事情也掺和进来了，就是和学校几个人打算参加个比赛做个 app，代号 CDT（其他事情也有，比如帮别人用 Jinja2 和 pelican 做静态站）。于是 13 年的暑假，很多事情同时交织在我的脑海里，让我来忙乱地有些不及躲闪。这时候我也逐渐意识到一个问题，就是如果一个东西一开始就是由我来设计的，那么如果没有太多的条件约束，那么很快这个东西就会变成一个无底洞。我说这个的意思也许是委婉的表达，最后，直到今天，那个图书管理系统（最后我起的名字叫大黄，一只猫的名字），以及 cdt（虽然有中文名字，但是我们一般都只用英文的 package name），都没有完整实现。唯一的收获是，我对编程语言有了更深的认识，因为这一年我开始接触另一门名叫 erlang 的编程语言（没错，就是从前面提到分布式存储的 couchdb 展开的）以及，我开始意识到，自己并不适合写代码。

我说我不适合写代码意思，具体来说，不是说我完全不适合做软件开发，而是当前写代码这种形式不利于我对软件的具体实现。听起来有点拉屎不成怪茅厕的意味（家乡的土话），不过实际上我觉得这个有一定因素在里面。

扯远了。当然真正阻碍我实现 cdt 和大黄的原因不是这个，而是自己因为太负责任了导致自己开始变得不负责（大黄就是个很好的例子，因为我太想帮她们早点做好这个系统了，然而欲速则不达，而我即便休学了早晚也还要上学）。至于最根本的原因，还在于，我只是一个人在瞎折腾，没用跟人进行太多的沟通（或者说我已经不知道该怎么沟通了）。退一步来讲，当初我完全可以谨慎评估一下，有些东西完全可以不用自己做的，或者在当时那个条件下如果自己去做，你根本找不到能够给你提供实质帮助的人（题外话：其实是我错过了），或者说，我所做的那些，并不符合历史的客观规律，太理想化了³³。

后来，我多次尝试重启大黄和 cdt 的进程，然而最终都没能进行下去。实际上现在那边也不太需要大黄了，因为图书馆的定位也变了。而 cdt，当初我们一起搞项目的另外两个人，

³³ 虽然不是决定性的，不过有时候大环境确实能决定很多，比如在联大当时根本没几个玩 python 的。类似的情况还有很多，所以很多事情不被理解也是正常。所以 15 年之前的一些举措确实是 too young 了

都工作了，只有我因为休学和留级继续蹲在学校里。

2014 年是对我来说是曲折而平淡的一年，我主动留级了，原先跟我一个班的人则大四了。这一年开始，我没写多少代码，只是偶尔还接触点 IT 方面的消息，稍微了解一下 `erlang/prolog/apl`³⁴，本来上半年有个微信公众号的活要弄，但感觉自己有点打不起精神了（不出意外，这个项目最后也没能做出啥有用的贡献）。下半年也就九月份回光返照了一段时间，写了点东西，有了些新发现和新的计划。然而，随着十一月开始的降温，我的状态便一落千丈，最后，实在撑不住了，寒假回了家，然后一学期没能回学校。在深圳的那段调养的过程基本可以忽略了，因为并没有太多和 `python` 相关的事情，唯一跟编程沾边的是我打算给 `erlang` 的几本书做批注。后来因为某些原因，我在深圳某公司干了两个月的主板 `layout`（布线），又匆匆的回了学校。然后呢，差不多就到今天了。

前面提到了两个失败的例子，其实如果那时候看过一些类似 软件随想录 / `dreaming in code` / 人月神话 一类的书的话，我估计就不会走得太偏了。

上面这些东西在 2015 年之前并没有跟太多人讲过，因为当时，我不知道该如何表达当时我的想法。然而人生总是要经历些什么的，所以无论成败，无所谓他人怎么看待我，该做的事情还是要继续下去，不然我还不如继续在深圳 `layout` 呢你说？回顾这段经历，我觉得在 `python` 学习过程中最难受的一段时间应该是死磕 `pyramid` 文档的时候。但这也是对我帮助最大地方。而且我的英语水平并不是特别好，就裸考过了个四级，六级裸考没过，然后就没有然后啦，觉得四级过了就够用了。啃文档那时候是 12 年的冬天，别说 `pyramid` 的文档，当时 `python` 的官方文档我都不太懂看。所以我觉得，新手刚接触 `python` 的过程中，学会看 `python` 的文档是个很重要的过程。

于是我觉得这里有必要着重讲一下怎么看懂 `python` 一类的文档，然后正片从这里开始。首先你要知道，`python` 也好，`flask` 也好 `pyramid` 也好很多很多的 `python` 项目的文档，无论输出的格式何种，大多都是用 `sphinx` 生成的（现在基本是主流规范了。以前的话可能就直接用 `rst` 生成个 `html`，或者干脆 `txt` 纯文本）。`Sphinx` 生成的文档有几个特点。首先，如果是 `html` 版的话，页尾或页首通常是这个样子的：



图 10: Pyramid 文档页首

³⁴ 可能有人好奇为什么我不去接触那些所谓的函数式语言，因为我觉得类似 `lisp` 那个方向在未来并没有太多价值。

这里只用关注三个链接: [next](#) | [modules](#) | [index](#) , [next](#) 或 [previous](#) 很好理解, 就是翻页用的。而这里的 [modules](#) 是最有用的地方, 通常项目的 [api](#) 文档 (一般都是代码里的文档注释) 就在这里了。类似的, [python](#) 官方文档也是这个样子的:

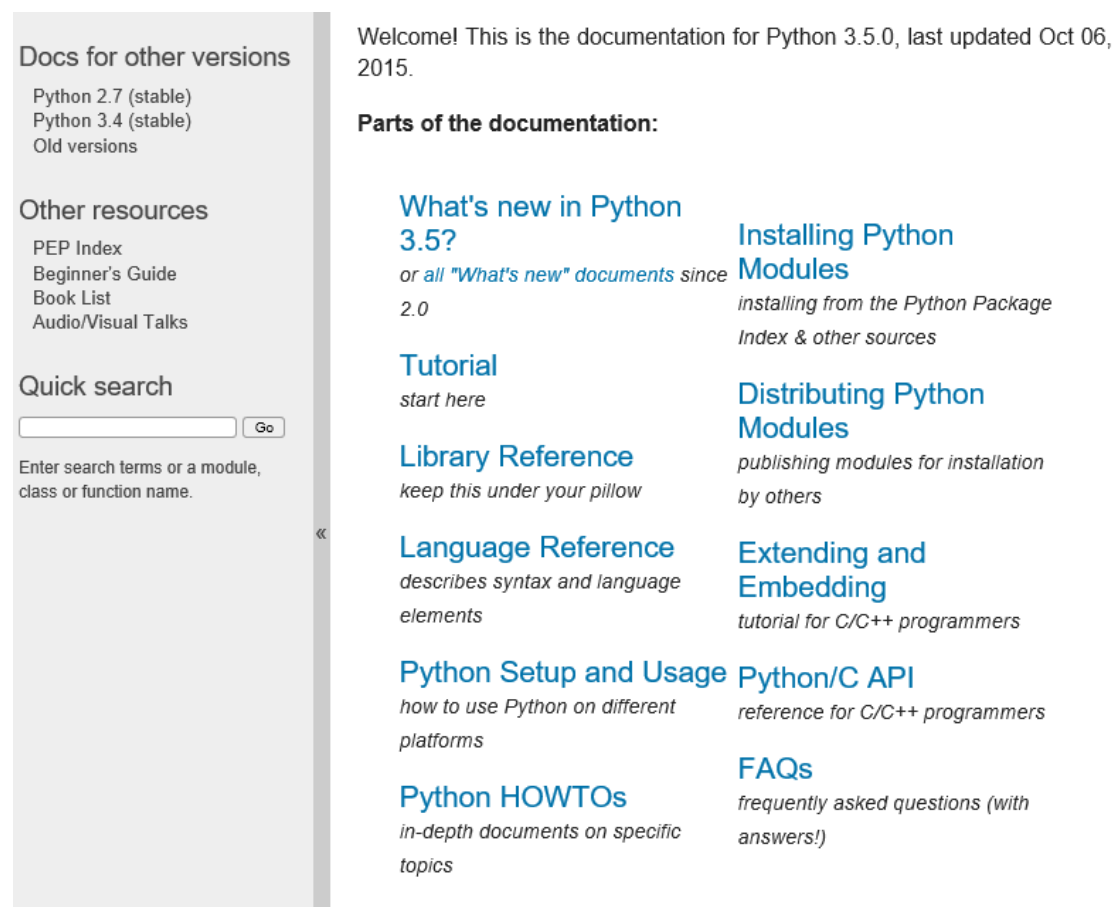


图 11:Python3.5 文档首页

当然 [python](#) 的官方文档和第三方库在组织结构上还是有区别的 (毕竟标准库很多, 还要教你语法)。对于官方文档来讲, 直接看 [modules](#) 就有点显得太冗长了, 尤其对新手而言。所以官方文档的正确用法是: 用左侧的 [search](#), 以及, 需要的时候直接转到 [tutorial](#) 或者 [library reference](#) 即可, 通常是不太需要去看 [language reference](#) 的, 因为里面的标注法 ([notation](#)) 可能会让一些没接触过语法解析以及 [parser](#) 之类概念的人感到头晕。不过如果你想彻底搞懂 [yield/with](#) 之类的, 还是要看看里面的细节以及对应的 [how to](#) 文章。

[Sphinx](#) 生成的 [HTML](#) 文档的方便之处在于可以直接 [search](#), 以及借助浏览器进行多标签浏览。当然, [sphinx](#) 也可以生成 [pdf](#)。pdf 格式的话一般更适合打印或者在平板等阅读器上看, 加书签做备注什么的, 比网页版方便, 劣势方面, 可能也就是一些网页版的 [css](#) 效果表现不出来吧。Epub 或者 [tex](#) 好像也可以, 一般主流的 [python](#) 文档都是用 [reStructuredText](#) 这种格式编写的, 有些也用 [markdown](#) (其实用 [rst](#) 主要是历史原因, 这个最早也是 [zope](#) 那批人弄出来的)。

只要英文不太差，且还能适应这样的环境，那文档方面应该就没多大问题了。必要的时候，再调出代码来看下具体实现就好了。老实说至今我没有看过 Cpython 用 c 实现的那部分源码，可能还没这方面需要吧，我记得国内有本书叫《python 源码剖析》讲的就是这方面的东西，感兴趣的或者有（很扎实）C 语言基础的，可以去看看。

Ok，回到我自己的新手路线。除了文档，其他还比较有帮助的，就是一些陈旧的资料了，比如 limodou（李迎辉）的博客，他是文本编辑器 ulipad 以及 web 框架 uliweb 的作者。记得他有用过好几个博客，donews 的较早，百度空间也有，后来百度空间停服了，那部分资料估计现在也找不到了。Stackoverflow 上的问答资料也帮到过不少忙，即便像 pyramid 这样冷门的框架，上面依然能看到一些优质的问答，其中还包括一些 web 框架作者的解答，总之很多很多。然后还有其他一些国内的早期 python 人物，比如前面提到过的沈巍（邮件列表里人称沈游侠），他的网易博客也留下了不少有用的文章，当然最具代表性的还是他那个《python 编程艺术》³⁵的幻灯片。一开始看的时候心里想这特么什么鬼，后来过了两年回过头来看才觉得这人当年是何等的牛逼。

同时期的我记得还有赖永浩和潘俊勇。老潘当年推 zope 几乎是不留余地，可惜我开始接触 zodb 的时候，已经是 N 年后的事情了，不过当年留下的一些翻译文档还是有些帮助（另外还有 EryxLee 写的一些 pyramid 教程），不知道现在还能否找到。科学计算方面，当年有个张若愚，还出了本书，虽然有些过时不过看着也挺不错的，貌似曾经说要出第二版，不知道情况如何。同时期相对年轻点的，还有清风，张沈鹏（人称教主），剩下的有影响力的我记着好像就没几个了，也许现在还活跃在知乎或类似的地方打打嘴炮，好像也没啥了，不少也早转方向了。

当然这些国内的 python 大牛，我记得好像都是从啄木鸟社区和 python-cn 的邮件列表找到的。而说到啄木鸟社区和 python-cn 的邮件列表，由不得不提一下国内 python 圈里面广为人知的大妈：zoom.quiet。此人对早期国内 python 社区的发展做出了很大的贡献，虽然有些方面的意见不敢苟同，但其付出是值得肯定的，包括好几次 pycon 的筹办工作，虽然 pycon 没几届办的特别好的，但也算是挺不容易的吧。国内的 Pycon，老实讲我都没太明白它的定位，虽然不能跟国内的 ruby conf 比，但或许还是有些存在价值吧。

我记得前面开始水 zope 的时候有讲过：历史很重要。是的，我的新手路线离不开对 python 历史的各种花式挖坟，国外的也好、国内的也好，基本都被我扒过了。而挖 python 历史的过程对我个人也有不少帮助，因为你会意识到某种历史的局限性，例如你会渐渐地会明白 python3 的各种纷争以及未来 python 的发展趋势。

我觉得可以一句话概括 Python 的宿命：领导者是对的，然而并没有什么卵用。这个不仅体现在 python 的 core team，比如当年做出 py3k 决定这件事情。我也觉得搞 python3 是

³⁵ 这个幻灯片最早挺熟是从一个 id 叫 shell909090 的前辈写的一篇新手入门里看到的，那篇入门指引比我的简练很多，定位比较明确，对我也有过一些帮助。而这位前辈我是在啄木鸟社区的 planet 上看到的：
<http://shell909090.org/blog/archives/2272>

对的，但是，不向下兼容的话，那也只能等历史来完成这个漫长的过渡过程了。同样的，在国内 python 社区也是这个问题，管事的那帮人想法也是对的，但是，现在这个年代坚持用着 google 邮件列表真的好吗？自己搞一个像 erlang 一样的独立邮件列表或者 ruby-china 一样的官方论坛会死啊？

谁知道呢，我也不是没跟人提过这个事情。而，整个学习 python 过程中我最不爽的问题就是这个，本来好好的一个 python 社区，随着 09 年天朝局域网体系的确立，变得分崩离析。如果不是因为大部分人上不了 google，或者说非 gmail 邮箱后来根本收不了里面的群发邮件，v2ex 后来也不会有那么多玩 python 的人在里面。我也知道，国内还有个水木清华，还有个 python 贴吧，但说真的，最早那一批玩 python 的死守 google 邮件列表的结果就是必将导致 python 社区的断层，而历史将无法得到延续，更别提文化的传承。

如果要我给他们提个建议，也很简单：去 google 化，接地气³⁶；面向大众，走向中立。今时今日的 google 早已不是曾经的 google 了，python 在脚本语言中的地位也开始受到更多的挑战，做出些艰难的决定，有时也是必要的。当然，即便 python 在中国做不活社区，我也觉得没啥，毕竟相濡以沫，不如相忘于江湖。而我最初喜欢 python 的，也是这种感觉。

写这篇东西的用意，一方面是希望能帮助到新接触 python 的人，或者学习过程中遇到了跟我类似问题的人，能够有个参考指南，不至于犯太多同样的低级错误。虽然很多时候犯错是无可避免的，于是真正有点用处的，也只是让很多新人能够了解一部分 python 在国内发展的前世今生。虽然我所说的很主观不完整也不一定对，但也是某种角度的参考，毕竟国内很少有人能像我这样凭空扯淡这么多字数的吧。

其实曾经也有人跟我有类似的想法，或者说我之所以有写这个打算，完全是受前人的影响。在国内 python 社区发展的比较好的那个年代，那时候 google 还没被墙掉，有一本挺不错的书叫《可爱的 python》。一开始我以为这是一本给新手看的书，然而我错了，可能当初大妈编辑的时候也觉得书的定位应该偏向新人，结果效果恰恰相反。这本书一开始我看的时候是云里雾里，因为我用的主力系统不是 linux，有些东西没办法实操，加上章节安排有点混乱（比我现在写的这个还混乱），读起来也费劲。而且，我至今没搞懂那个 CDays 是什么意思。后来再看的时候我又觉得，里面很多东西都过时了，好像除了怀旧并没有什么其他用途。不过再后来我就发现读这本书是一个了解国内 python 历史的绝佳途径，而且里面的一些小贴士（书里写作作弊条，cheatsheet）也挺实用的。

当然，很多 python 书籍还是不错的。挑选一本好书要看很多方面，封面设计、排版、标题、厚度、作者经历、译者水平以及出版社也都包括在内，看多自然就会挑了，没有捷径。

³⁶ 关于接地气我想举个例子，就是 limodou 的 uliweb qq 群。

真·新手路线

首先我想提一下图形界面这一块，先声明，个人不建议拿 python 搞 GUI。你会发现，在前面的参考资料部分，我都没有提到 wxpython。wx 还是有不少人在用的，而我只是象征性的放了一个 pyqt。这么做或者说不推荐 GUI 开发是有原因的，毕竟是 python 比较明显的一个鸡肋领域。如果真要弄，自带的 tkinter 基本就够了，pyqt 我觉得学习成本不亚于 twisted。而安卓方面，虽然现在有个叫 kivy 的项目，GVR 也推了，然而（github 上的关注了居然比 pyramid 还低）。所以，不要误入歧途。讲真，如果要搞，现在 c++/java/.net 甚至 js 都比 python 要实在。

然后是科学计算和数据分析领域，这个是目前 python 比较火的一个地方。如果你对这个感兴趣，那可能就会明白我推荐安装 anaconda 的用意。前面提到的有些库也是这个领域常用的，比如 numpy / cython / matplotlib / pandas 等等。但毕竟没接触过这个领域，所以其他有用的东西我可能还没来得及接触，仅仅是有耳闻的那种。不过 ipython 倒是比较常用，因为这个不分领域，qt console 能爆官方命令行十几条街，用过之后就不愿换回来了。不过据我一些在公司里做 DA(数据分析)的同学讲，其实真正用 python 的时候并不是很多。

然后是网络这一块，python 还是有一定优势的，因为库比较成熟，比如 twisted 这种。虽然 twisted 也跟 zope 一样很多人见了就怕，实际上也没那么夸张啦，一旦习惯了 ZCA 这种设定，再多的 python 代码看上去都不会觉得太恶心。爬虫方面，python 也是比较成熟的，抓站之流比较出名的有 scrapy。网络安全方面我不是很了解，感兴趣的自己去 google 一下。

Web 开发这块，水比较深，我接触的也最多，于是打算放到后面来讲。一般上来都会选个框架，主流的是 django，选它不会错，就看习不习惯了。次一点的有 flask 和 tornado，在国内都有些创业公司用在生产环境上了。Bottle/web.py 也有人用，不过相对比较少。而且 bottle 其实往深了讲，适合入门但并不太适合新手作为主力长期使用。其他更冷门的框架也有人用，比如 web2py，比如 pyramid。

我这里可以简要概括一下几个框架的主要特点：首先是 django。Django 最开始是仿 rails 这个应该都明白了，但是概念上有些许不同。比较适合快速开发，用起来也是简单粗暴。现成的扩展很多，比如你想给自己的项目加个 sentry 的支持，一搜就能找到 django-sentry。而且整个工具链都很齐全，数据库迁移(db migrate)什么的也已经很成熟了。但是，选择 django 意味着丧失了一定的可定制性。因为实际上你不可能真正去改动 django 的源码，太多了，10M 的大小啊。而且，即便你想给 django 修点 bug 或者加点新功能，也不是那么简单的事情。

相比之下，轻量级的框架就容易些，比如 flask 和 bottle。用 flask 的人多主要原因可能是集成度比较高且扩展多。至少，flask 记成的自家模板 jinja2 要甩 bottle 的原生模板几条街。而 Bottle 的买点无非是单文件 import，这个其实也是最大的局限性。Flask 的问题在于扩展多了，感觉和 django 也没啥区别，而且集成度可能还没 django 做的好。我就记得以前

用 flask 的时候,加上 flask-alchemy 老出错,但是自己手动接入 sqlalchemy 一点问题都没有。

Tornado 这个框架相对比较特别,因为说它是框架其实并不严谨。其实它应该是 web.py 的威力加强版。目标和 django 一样也是大而全,但是整体架构还算轻量级。因为我没试过这款,不太好对这个框架做过多的评价。优势可能是集成前端交互比较方便,还自带了 server 端的功能,应该来说是最适合新手的一款。缺点,也许是异步回调的一些写法不太符合 python 的风格(实际上也可以用同步逻辑写)。

高度集成的框架里有一个极端叫 Web2py,被人吐槽的很多。然而做些小站点也没有传说的那么糟糕,你看前面提到过的张若愚,他不照样用的很舒坦。

我的选择是 pyramid,当初选择的理由是我观察到 python 的 web 框架的发展呈现出一个整体发散的趋势,而 pyramid 在那个时候是由两个框架合并过来的(即 repoze.bfg+pylons),这一点我觉得挺有意思。深入了解之后,我发 repoze.bfg 原来继承了 zope 一脉的 traversal 功能,这个我挺喜欢的,还有 chameleon 模板等等。我比较喜欢这种有实力的项目,即便它看上去不那么出众,但是感觉很适合自己。总之,每个人都有自己的喜好,但有时候适合自己的未必适合别人,所以放到工作上,总有些东西需要去适应和妥协的。

Web 开发这块比较麻烦的,如果不是在大公司里,不可避免的要牵扯到一部分运维的工作。比如环境和代码部署啦,自动化测试啦,数据库维护等等,不过我觉得最要命的可能还是前端。现在部署的花样比以前多多了,docker 是 13 年左右开始冒出来的,现在用的人比以前要多不少,这个相当于一个系统级别的 virtualenv。我还没怎么正式用过,据说效果还可以。代码仓库基本上都一律用 git 了,除非公司用的 svn 之类的比较难迁移。CI (Continuous Integration) 方面,我没怎么试过,只知道有那么些工具还不错,比如 buildbot / Jenkins / travis 等等。运维方面,用过的也就 fabric, saltstack 据说还不错,不过也没试过。

数据库方面我觉得可以分出来讲讲。很多人刚上手 web 开发比较头疼的可能不是前端而是 sql。其实呢,sql 没那么恶心,习惯就好了,而且现在也还有很多 nosql 的选择,也很成熟,比如 mongodb 和 redis。不过关系型数据库还是广泛应用的,出来混,迟早要还。主流的还是 mysql,小型的可以用 sqlite,国内用 oracle 和 sql server (t-sql) 的也不少,不过个人比较喜欢 postgresql,其实初阶水平来用的话,差别都不是很大。但具体到开发,sql 还会涉及到数据库迁移的过程,如果用 django 你就轻松多了,其他的话,可能还得借助 sqlalchemy+alembic 的组合,或者,自己手动 alter table 吧。

前端之所以说头疼,是对于类似我这周单干的人来讲的。前端的工具和技术更新的快,搞后端的基本是跟不上节奏的。这倒不是重点,搞前端的知识储备和经验积累可能比大多数搞后端的人预估的还要多得多。而且,还需要一定的设计水平。很多搞后端的就是喜欢偷懒,以为用个 bootstrap 就可以省去前端设计的工作了,于是乎曾经好长一段时间,你都会看到很多网站的按钮都长这样:



图 12: bootstrap2 中文文档（局部）

所以有时候太偷懒也是不行的。不过还好现在 bootstrap 效果比以前简洁了许多，没那么强烈的违和感。现在前端用到比较多的工具，我所知的有：node.js、webpack / gulp / grunt、coffeescript / typescript、angular.js / react.js / vue.js、jquery / lo-dash、d3.js / three.js。其他的没列出来的说明名气还不是很大。由于自己的前端水平也不是很好，这里就不细讲个人偏好了。获取前端资讯的方法有很多，关注点新闻站点，水下知乎，刷下微博，搜些好文章，都可以的。当然最重要的还是参考与实践结合，这个道理无论前后端都是相通的。

Web 开发方面差不多就这些，现在大部分都是后端提供些 api，前端分离，手机应用也调 api 或者干脆用混合应用（html5 + native code）。等项目架构稍复杂点，不可避免的会引入消息队列、集群、冗余、周期性任务计划以及性能优化等等，有时候还不得不做点 seo 之类的杂活，总之是个深坑。而游戏的水更深，所以我就没太多能讲的了，只好一笔带过了。现在用 python 做游戏的还真不多，大多都转 lua 了。而听过且比较出名的可能是 eve，不过用到的好像是 c++ 结合 stackless python。

最后，我想说的是，python 不是一门什么好编程语言，比如你有打算深入的话，不可避免有一天还要去研究 c 代码，其实很多其他编程语言也都一样。所以，这里我没有针对 python 的意思，我都意思是说，现存的编程语言都是垃圾。所以，如果正片文章没能看懂，也没关系，你只需要记住，尽量不要把编程语言简称语言，那是对自然语言的侮辱。

感觉全文到这里就可以结束了，希望对坚持看到这一行你有所帮助吧。如果你还记得前面留的两道（其实是三道）作业题，最好也做一下吧。这种题目比较适合看到这里还对 python 的具体应用没有概念的朋友。我也不打算出更多的题目，因为软件开发实际上还是看实践的，你老是解别人假设的题目，最后与你在真实环境里碰到的情况，可能完全搭不上边。所以数量上，我觉得两三道足矣。还有，关于教程，我想最后强调一下，最好看官方的 tutorial，哪怕是中文版，再不济也可以是简明 python 教程，不要去看什么笨办法学 python，那个不适合英文解析速度不够快的人，效率很低，而且基本上没什么卵用（认真脸）。

以上所有，我想强调的东西，如果要概括一下的话，简单说就是“学以致用”。这是我们学校的校训，虽然很多人并没有真正的理解，但我希望所有搞技术的，或者有缘看到这篇文章的，能够最终领悟这个道理。

后记

这篇指南的篇幅远超出了我之前的计划，时间上也是经过了一个很长的跨度。最初建立这个 word 文件是在 9 月 10 日，现在一个多月过去了，或许也该花点时间总结些什么。

不是说想在此处再补充三千字，只是有感过去这些年的变化。这是我在大学的第五年，终于到了大三，而前四年说是一事无成我也认了。此次回北京，并没有什么很具体的计划，也不想搞出什么大新闻，我只想安静的做一只土鳖，争取老老实实过完这一学年。

有时候我也在想，自己坑过那么多人，以后的日子要怎么过？而实际上，如果心里没有些纠结的地方，那么整个人可能也会散掉吧。无论如何，能有这番经历，还要感谢大家，无论现在怎么想，再多的误解，我也不会再去作多余的解释。

黑桥的事情，我想，在我没有足够的能力之前，我是不会再踏足那片土地了。虽然我知道她们当中有些已经没再搭理我了，但没把我拉黑或许就算不错了吧。

科协也是，当然相对来说无奈的地方可能更多。有些东西，我说再多，做再多，你们不信，有屁用。当初你们想学，却不问我，那是几个意思。有想法，却不敢说，好像会死。我不是在唱衰，事实上我期待一个更好的明天，然而历史已将我拉进轮回，没有我的地方或许会少些阻碍。这当初是一个技术主导社团，也许现在不是，但以后也许还会回来。

程序兴趣小组的历史任务还没有完成，但在没有做够充分的准备之前，联大补完计划的主体进程只能先搁置了，小的项目可以先跑跑，积攒些实力和筹码。

本来想写点感激的话，但仔细想想，也许我错的地方固然不少，但另一方面大家都值得反思，于是写了一堆带有轻微负能量的话，或许这就是人性吧。因为这一切本无对错，如果硬要分清楚，那就是我们都错了。

本文作为回馈 python 社区的一部分努力，也许后记部分写的有点莫名其妙，但对我而言，四年大学生活的扭曲经历，和两年多的 python 使用经历，是分不开的。

感谢那些曾经以及现在帮助我、相信我、对我施以沉默打击的人和组织，没有他们今天我也不会这么有空写完这两万六千字或者也没什么可写的。最后，谨以此文纪念一个实用主义者在旧世界中的死亡。

2015 年 10 月 13 日 星期二

刘东熠，写于北京联合大学