为什么从函数开始?

读完第一部分之后,你多多少少已经"写"了一些程序,虽然我们总是说,"这就是让你脱盲";也就是说,从此之后,你多多少少能够读懂程序,这就已经很好了。

可是你无论如何都避免不了已经写了一些,虽然,那所谓的"写",不过是"改"而已 —— 但,毕竟也是一大步。

绝大多数编程书籍并不区分学习者的"读"与"写"这两个实际上应该分离的阶段 —— 虽然现实中这两个阶段总是多多少少重叠一部分。

在一个比较自然的过程中,我们总是先学会阅读,而后才开始练习写作;并且,最终,阅读的量一定远远大于写作的量——即,输入远远大于输出。当然,貌似也有例外。据说,香港作家倪匡,他自己后来很少读书,每天咣当咣当地像是打扫陈年旧物倒垃圾一样写作——他几乎是全球最具产量的畅销小说作家,貌似地球另外一端的史蒂芬·金都不如他多。又当然,他的主要输入来自于他早年丰富的人生经历,人家读书,他阅世,所以,实际上并不是输入很少,恰恰相反,是输入太多……

所以,正常情况下,输入多于输出,或者,输入远远多于输出,不仅是自然现象,也是无法改变的规则。

于是,我在安排内容的时候,也刻意如此安排。

第一部分,主要在于启动读者在编程领域中的"阅读能力",到第二部分,才开始逐步启动读者在编程领域中的"写作能力"。

在第二部分启动之前,有时间有耐心的读者可以多做一件事情。

Python 的代码是开源的,它的代码仓库在 Github 上:

https://github.com/python/ (https://github.com/python/)

在这个代码仓库中,有一个目录下,保存着若干 Python Demo 程序:

https://github.com/python/cpython/tree/master/Tools/demo (https://github.com/python/cpython/tree/master/Tools/demo)

这个目录的下 README 中有说明:

This directory contains a collection of demonstration scripts for various aspects of Python programming.

- beer.py Well-known programming example: Bottles of beer.
- eiffel.py Python advanced magic: A metaclass for Eiffel post/preconditions.
- hanoi.py Well-known programming example: Towers of Hanoi.
- life.py Curses programming: Simple game-of-life.
- markov.py Algorithms: Markov chain simulation.
- mcast.py Network programming: Send and receive UDP multicast packets.
- queens.py Well-known programming example: N-Queens problem.
- redemo.py Regular Expressions: GUI script to test regexes.
- rpython.py Network programming: Small client for remote code execution.
- rpythond.py Network programming: Small server for remote code execution.
- sortvisu.py GUI programming: Visualization of different sort algorithms.
- ss1.py GUI/Application programming: A simple spreadsheet application.
- vector.py Python basics: A vector class with demonstrating special methods.

最起码把这其中的以下几个程序都精读一下,看看自己的理解能力:

- <u>beer.py</u>
 (<u>https://github.com/python/cpython/blob/master/Tools/demo/beer.py</u>).

 Well-known programming example: Bottles of beer.
- eiffel.py.
 (https://github.com/python/cpython/blob/master/Tools/demo/eiffel.py).

 Python advanced magic: A metaclass for Eiffel post/preconditions.
- <u>hanoi.py</u>
 (<u>https://github.com/python/cpython/blob/master/Tools/demo/hanoi.py</u>)

 Well-known programming example: Towers of Hanoi.
- <u>life.py (https://github.com/python/cpython/blob/master/Tools/demo/life.py)</u>. Curses programming: Simple game-of-life.
- <u>markov.py</u>
 (https://github.com/python/cpython/blob/master/Tools/demo/markov.py)
 Algorithms: Markov chain simulation.
- <u>queens.py</u>
 (<u>https://github.com/python/cpython/blob/master/Tools/demo/queens.py)</u>

 Well-known programming example: N-Queens problem.

就算读不懂也没关系,把读不懂的部分标记下来,接下来就可以"带着问题学习"......

在未来的时间里,一个好的习惯就是,有空了去读读别人写的代码——理解能力的提高,就靠这个了。你会发现这事儿跟其他领域的学习没什么区别。你学英语也一样,读多了,自然就读得快了,理解得快了,并且在那过程中自然而然地习得了很多"句式",甚至很多"说理的方法"、"讲故事的策略"……然后就自然而然地会写了,从能写一点开始,慢慢到"很能写"!

为了顺利启动第一部分的"阅读",特意找了个不一样的入口,"布尔运算";第二部分,从"阅读"过渡到"写作",我也同样特意寻找了一个不一样的入口:**从函数开始写起**。

从小入手,从来都是自学的好方法。我们没有想着一上来就写程序,而是写"子程序"、"小程序"、"短程序"。从结构化编程的角度来看,写函数的一个基本要求就是:

- 完成一个功能;
- 只完成一个功能;
- 没有任何错误地只完成一个功能......

然而,即便是从小入手,任务也没有变得过分简单。其中涉及的话题理解起来并不容易,尽管我们尽量用最简单的例子。涉及的话题有:

- 参数的传递
- 多参数的传递
- 匿名函数以及函数的别称
- 递归函数
- 函数文档
- 模块
- 测试驱动编程
- 可执行程序

这些都是你未来写自己的工程时所必须仰仗的基础,马虎不得,疏漏不得。

另外,这一部分与第一部分有一个刻意不同的编排,这一部分的每一章之后,**没有写总结** —— 那个总结需要读者自己动手完成。你需要做的不仅仅是每一个章节的总结,整个第二部分读完之后,还要做针对整个"深入了解函数"(甚至应该包括第一部分已经读过的关于函数的内容)的总结…… 并且,关于函数,这一章并未完全讲完呢,第三部分还有生成器、迭代器、以及装饰器要补充 —— 因为它们多多少少都涉及到下一部分才能深入的内容,所以,在这一部分就暂时没有涉及。

你要习惯,归纳、总结、整理的工作,从来都不是一次就能完成的,都需要反复多次之后才能彻底完成。必须习惯这种流程——而不是像那些从未自学过的人一样,对这种东西想当然地全不了解。

另外,从现代编程方法论来看,"写作"部分一上来就从函数入手也的确是"更正确"的,因为结构化编程的核心就是拆分任务,把任务拆分到不能再拆分为止——什么时候不能再拆分了呢?就是当一个函数只完成一个功能的时候……