xiaolai From wangpeiran 11 selfteaching#697

2 contributors

Raw Blame

276 lines (155 sloc) 24.4 KB

# 如何从容应对含有过多“过早引用”的知识?

“过早引用”（Forward References，另译为“前置引用”），原本是计算机领域的术语。

在几乎所有的编程语言中，对于变量的使用，都有“先声明再使用”的要求。直接使用未声明的变量是被禁止的。Python 中，同样如此。如果在从未给 `an_undefined_variable` 赋值的情况下，直接调用这个变量，比如，`print(an_undefined_variable)`，那就会报错：`NameError: name 'an_undefined_variable' is not defined`。

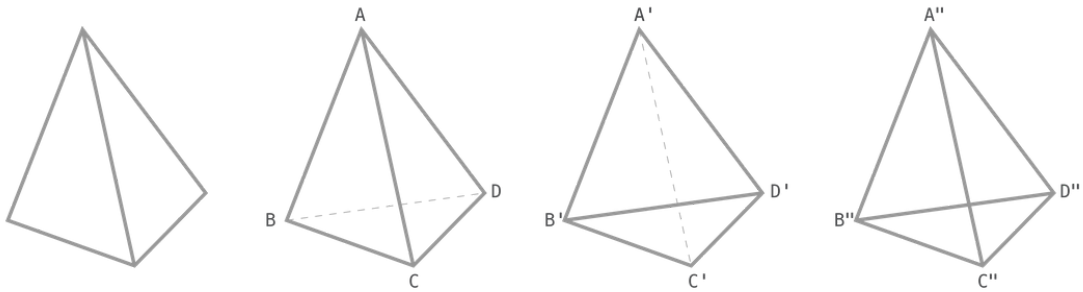
```
print(an_undefined_variable)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)

<ipython-input-1-7e0e1cc14e37> in <module>
----> 1 print(an_undefined_variable)

NameError: name 'an_undefined_variable' is not defined
```

充满过早引用的知识结构，在大脑中会构成类似 M.C. Escher 善画的那种“不可能图形”那样的“结构”。

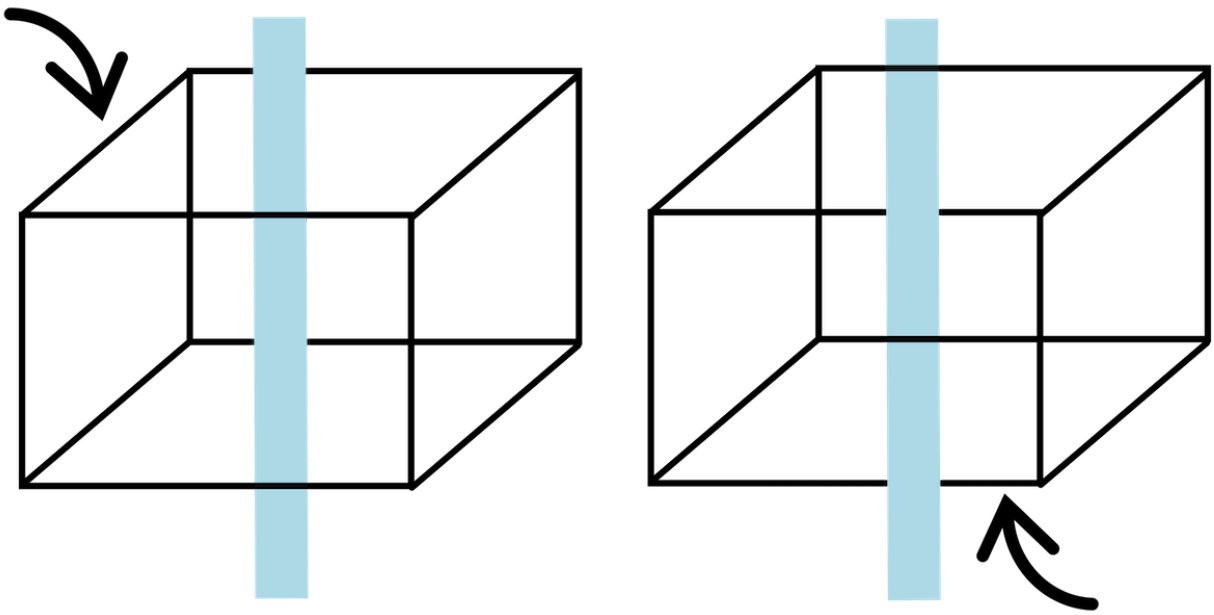



在上图中，前三个椎形一般不会造成视觉困惑 —— 尤其是第一个。

若是加上虚线，比如，第二个和第三个，那么由于我们预设虚线表示“原本应该看不见的部分”，于是，`C` 点的位置相对于 `B` 和 `D` 应该更靠近自己；`C'` 的位置，相对于 `B'` 和 `D'` 应该更远离自己……

然而，在第四个椎形中，由于 `B"D"` 和 `A"C"` 都是实线，于是，我们一下子就失去了判断依据，不知道 `C"` 究竟是离自己更近还是更远？

对一个点的位置困惑，连带着它与其它三个点之间的关系。可若那不是锥体，而是立方体呢？每个点的位置困惑会造成对它与更多点之间的更多联系的困惑…… 若是更多面体呢？



把这些令人困惑的点，比喻成“过早引用”，你就明白为什么“很多过早引用”的知识结构会那么令人困惑，处理起来那么令人疲惫了吧？

## 过早引用就是无所不在

可生活、学习、工作，都不是计算机，它们可不管这套，管它是否定义过，管它是否定义清晰，直接甩出来就用的情况比比皆是。

对绝大多数“不懂事”的小朋友来说，几乎所有痛苦的根源都来自这里：“懂事”的定义究竟是怎样的呢？什么样算作懂事，什么样算作不懂事？弄不好，即便整个童年都在揣摩这个事，到最后还是迷迷糊糊。他们的父母，从未想过对孩子说话也好提要求也好，最好“先声明再使用”，或者即便事先声明过也语焉不详…… 于是，这些可怜的孩子，能做的只有在惶恐中摸索，就好像在黑暗中拼图一样。

可事实上，他们的父母也不容易。因为确实有太多细节，给小朋友讲了也没用，或者讲也讲不清楚，又或者拼命解释清楚了，但小朋友就是听不进去…… 所以，令人恼火的“过早引用”，有时候真的是只能那样的存在。

谈恋爱的时候也是这样。太多的概念，千真万确地属于过早引用。爱情这东西究竟是什么，刚开始的时候谁都弄不大明白。并且事实证明，身边的绝大多数人跟自己一样迷糊。至于从小说电影里获得的“知识”，虽然自己看心神愉悦，但几乎肯定给对方带来无穷无尽的烦恼 —— 于对方来说你撒出来的是漫天飞舞的过早引用……

到了工作阶段，技术岗位还相对好一点，其他领域，哪哪儿都是过早引用，并且还隐藏着不可见，搞得人们都弄出了一门玄学，叫做“潜规则”。

人们岁数越大，交朋友越来越不容易。最简单的解释就是，每个人的历史，对他人来说都构成“过早引用”。所以，理解万岁？太难了吧，幼儿园、小学的时候，人们之间几乎不需要刻意相互理解，都没觉得有这个必要；中学的时候，相互理解就已经开始出现不同程度的困难了，因为过早引用的积累。大学毕业之后，再工作上几年，不仅相互理解变得越来越困难，还有另外一层更大的压力 —— 生活中要处理的事情越来越多，脑力消耗越来越大，遇到莫名其妙的过早引用，哪儿有心思处理？

## 不懂也要硬着头皮读完

这是事实：大多数难以掌握的技能都有这个特点。人们通常用“学习曲线陡峭”来形容这类知识，只不过，这种形容只限于形容而已，对学习没有实际的帮助。面对这样的实际情况，有没有一套有效的应对策略呢？

首先是要学会一个重要的技能：

> **读不懂也要读完，然后重复很多遍。**

这是最重要的起点。听起来简单，甚至有点莫名其妙 —— 但以后你就会越来越深刻地体会到，这么简单的策略，绝大多数人竟然不懂，也因此吃了很多很多亏。

充满了过早引用的知识结构，就不可能是一遍就读懂的。别说这种信息密度极高的复杂且重要的知识获取了，哪怕你去看一部好电影，也要多刷几遍才能彻底看懂，不是嘛？比如，Quentin Tarantino 导演的 Pulp Fiction (1994)、David Fincher 导演的 Fight Club (1999)、Christopher Nolan 导演的 Inception (2010)、或者 Martin Scorsese 导演的 Shutter Island (2010)……

所以，从一开始就要做好将要重复很多遍的准备，从一开始就要做好第一次只能读懂个大概的准备。

古人说，读书百遍其义自见，道理就在这里了 —— 只不过，他们那时候没有计算机术语可以借用，所以，这道理本身成了“过早引用”，对那些根本就没有过“读书百遍”经历的人，绝对以为那只不过是在忽悠自己……

有经验的读书者，拿来一本书开始自学技能的时候，他会先翻翻目录（Table Of Contents），看看其中有没有自己完全没有接触过的概念；然后再翻翻术语表（Glossary），看看是否可以尽量理解；而后会看看索引（Index），根据页码提示，直接翻到相关页面进一步查找…… 在通读书籍之前，还会看看书后的参考文献（References），看看此书都引用了哪些大牛的书籍，弄不好会顺手多买几本。

这样做，显然是老到 —— 这么做的最大好处是“尽力消解了大量的过早引用”，为自己减少了极大的理解负担。

所以，第一遍的正经手段是“**囫囵吞枣**地读完”。

囫囵吞枣从一开始就是贬义词。但在当前这个特殊的情况下，它是最好的策略。那些只习惯于一上来就仔细认真的人，在这里很吃亏，因为他们越是仔细认真，越是容易被各种过早引用搞得灰心丧气；相应地，他们的挫败感积累得越快；到最后弄不好最先放弃的是他们 —— 失败的原因竟然是因为“太仔细了”……

第一遍囫囵吞枣，用个正面一点的描述，就是“为探索未知领域先画个潦草的地图”。地图这东西，有总比没有好；虽然说它最好精确，但即便是“不精确的地图”也比“完全没地图”好一万倍，对吧？更何况，这地图总是可以不断校正的，不是吗？世界上哪个地图不是一点一点校正过来才变成今天这般精确的呢？

## 磨练“只字不差”的能力

通过阅读习得新技能（尤其是“尽量只通过阅读习得新技能”），肯定与“通过阅读获得心灵愉悦”很不相同。

读个段子、读个小说，读个当前热搜文章，通常情况下不需要“精读” —— 草草浏览已经足够，顶多对自己特别感兴趣的地方，慢下来仔细看看……

但是，若是为了习得新技能去阅读，就要施展“**只字不差地阅读**”这项专门的技能。

对，“只字不差地阅读”是所有自学能力强的人都会且都经常使用的技能。尤其是当你在阅读一个重要概念的定义之时，你就是这么干的：定义中的每个字都是有用的，每个词的内涵外延都是需要进行推敲的，它是什么，它不是什么，它的内涵外延都是什么，因此，在使用的时候需要注意什么……

很有趣的一个现象是，绝大多数自学能力差的人，都是把一切都当作小说去看，随便看看，粗略看看……

你有没有注意到一个现象，人们在看电影的时候，绝大多数人会错过绝大多数细节；但这好像并不会削减他们的观影体验；并且，他们有能力使用错过了无数细节之后剩下的那些碎片拼接出一个“完整的故事”—— 当然，通常干脆是“另一个貌似完整的故事”。于是，当你在跟他们讨论同一个电影的时候，常常像是你们没坐在同一个电影院，看的不是同一个电影似的……

所谓的自学能力差，很可能最重要的坑就在这里：

> 每一次学习新技能的时候，很多人只不过是因为做不到只字不差地阅读，于是总是会错过很多细节；于是，最终就好像“看了另外一个山寨版电影一样”，实际上“习得了另外一个山寨版技能”……

在学习 Python 语言的过程中，有个例子可以说明以上的现象。

在 Python 语言中，`for` 循环可以附加一个 `else` 部分。你到 Google 上搜索一下 `for else python` 就能看到有多少人在“追问”这是干什么的？还有另外一些链接，会告诉你“for... else”这个“秘密”的含义，将其称为“语法糖”什么的……

其实，官方教程里写的非常清楚的，并且还给出了一个例子：

> Loop statements may have an else clause; it is executed when the loop terminates through exhaustion of the list (with for) or when the condition becomes false (with while), but not when the loop is terminated by a break statement. This is exemplified by the following loop, which searches for prime numbers:
>
> ```
> >>> for n in range(2, 10):
> ...     for x in range(2, n):
> ...         if n % x == 0:
> ...             print(n, 'equals', x, '*', n//x)
> ...             break
> ...     else:
> ...         # loop fell through without finding a factor
> ...         print(n, 'is a prime number')
> ...
> 2 is a prime number
> 3 is a prime number
> 4 equals 2 * 2
> 5 is a prime number
> 6 equals 2 * 3
> 7 is a prime number
> 8 equals 2 * 4
> 9 equals 3 * 3
> ```

只有两种情况，

> - 要么干脆就没读过，
> - 要么是读了，却没读到这个细节……

—— 后者更为可怕，跟花了同样的钱看了另外一个残缺版本的电影似的……

为什么说“只字不差地阅读”是一项专门的技能呢？你自己试过就知道了。明明你已经刻意让自己慢下来，也刻意揣摩每个字每个词的含义，甚至为了理解正确，做了很多笔记…… 可是，当你再一次“只字不差地阅读”的时候，你经常会“惊讶地发现”，自己竟然有若干处遗漏的地方！对，这就是一种需要多次练习、长期训练才能真正掌握的技能。绝对不像听起来那么简单。

所以，到了第二遍第三遍就必须施展“只字不差地阅读”这项专门的技能了，只此一点，你就已然**与众不同**了。

## 好的记忆力很重要

“就算读不懂也要读完”的更高境界，是“**就算不明白也要先记住**”。

人们普遍讨厌“死记硬背”…… 不过，说实话，这很肤浅。虽然确实也有“擅长死记硬背却就是什么都不会的人”，但是，其实有更多记忆力强的人，实际上更可能是“博闻强识”。

面对“过早引用”常见的知识领域，好记忆力是超强加分项。记不清、记不住、甚至干脆忘了 —— 这是自学过程中最耽误事的缺点。尤其在有“过早引用知识点”存在的时候，更是如此。

然而，很多人并没有意识到的是，记忆力也是“一门手艺”而已。并且，事实上，它是任何时候都可以通过刻意练习加强的“手艺”。

更为重要的是，记忆力这个东西，有一百种方法去弥补 —— 比如，最明显、最简单的办法就是“好记性不如烂笔头”……

所以，在绝大多数正常情况下，所谓的“记不清、记不住、甚至干脆忘了”，都只不过是懒的结果 —— 若是一个人懒，且不肯承认自己懒，又因为不肯承认而已就不去纠正，那…… 那就算了，那就那么活下去罢。

然而，提高对有效知识的记忆力还有另外一个简单实用的方法 —— 而市面上有各种“快速记忆法”，通常相对于这个方法来看用处并不大。

这个方法就是以下要讲到的“整理归纳总结” —— **反复做整理归纳总结，记不住才怪呢！**

## 尽快开始整理归纳总结

从另外一个角度，这类体系的知识书籍，对作者来说，不仅是挑战，还是摆脱不了的负担。

Python 官方网站上的 The Python Tutorial，是公认的最好的 Python 教材 —— 因为那是 Python 语言的作者 Guido van Rossum 写的……

虽然 Guido van Rossum 已经很小心了，但还是没办法在讲解上避免大量的过早引用。他的小心体现在，在目录里就出现过五次 More：

> - More Control Flow Tools
> - More on Defining Functions
> - More on Lists
> - More on Conditions
> - More on Modules

好几次，他都是先粗略讲过，而后在另外一处再重新深入一遍…… 这显然是一个最尽力的作者了 —— 无论是在创造一个编程语言上，还是在写一本教程上。

然而，即便如此，这本书对任何初学者来说，都很难。当个好作者不容易。

于是，这只能是读者自己的工作了 —— 因为即便是最牛的作者，也只能到这一步了。

第一遍囫囵吞枣之后，马上就要开始“**总结、归纳、整理、组织** 关键知识点”的工作。自己动手完成这些工作，是所谓学霸的特点。他们只不过是掌握了这样一个其他人从未想过必须掌握的简单技巧。他们一定有个本子，里面是各种列表、示意图、表格 —— 这些都是最常用的知识（概念）整理组织归纳工具，这些工具的用法看起来简单的要死。

这个技巧说出来、看起来都非常简单。然而，也许正因为它看起来如此简单，才被绝大多数人忽略…… 与学霸们相对，绝大多数非学霸都有一模一样的糊弄自己的理由：反正有别人做好的，拿过来用就是了！—— 听起来那么理直气壮……

可实际上，**自己动手做做就知道了** —— 整理、归纳、组织，再次反复，是个相当麻烦的过程。非学霸们自己不动手做的真正原因只不过是：嫌麻烦、怕麻烦。一个字总结，就是，**懒**！可是，谁愿意承认自己懒呢？没有人愿意。于是，都给自己找个冠冕堂皇的理由，比如，上面说的“反正别人已经做好了，我为什么还要再做一遍呢？”再比如，“这世界就是懒人推进的！”

久而久之，各种爱面子的说法完美地达成了自我欺骗的效果，最后连自己都信了！于是，身上多了一个明明存在却永远找不到的漏洞 —— **且不自知**。

我在第一次粗略读过整个 Python Official Tutorial 中的第五章之后，顺手整理了一下 Containers 的概念表格：

| Containers in Python | | |
|---|---|---|
| Sequence type | Set | Map |
| String | Set | Dictionary |
| range() | | |
| List | | |
| Tuple | | |

- ***Immutable***: background color, gray
- ***Ordered***: font color, red

### 可这张图错了！

因为我最早“合理囫囵吞枣”的时候，`Bytes` 这种数据类型全部跳过；而后来多轮反复之后继续深入，又去读 The Python Language Reference 的第五章 `Data Model` 之后，`发现 Set` 也有 Immutable，是 `Frozen Set`..... 当然，最错的是，整理的过程中，一不小心把“Ordered”给弄反了！

于是肯定需要再次整理，若干次改进之后，那张图就变成了下面这个样子：

| Containers in Python | | |
|---|---|---|
| Sequence Type | Set | Map |
| *String* | *Set* | *Dictionary* |
| *range()* | *Frozen Set* | |
| *List* | | |
| *Tuple* | | |
| *Bytes* | | |
| **Immutable**: background color, gray<br>**Ordered**: font color, red | | |

> 另外，从 Python 3.7 开始，Dictionary 是 insertion ordered 了：
> https://docs.python.org/3/library/collections.html#ordereddict-objects

这个自己动手的过程其实真的“很麻烦”，但它实际上是帮助自己强化记忆的过程，并且对自我记忆强化来说，绝对是不可或缺的过程。习惯于自己动手做罢！习惯于自己不断修改罢！

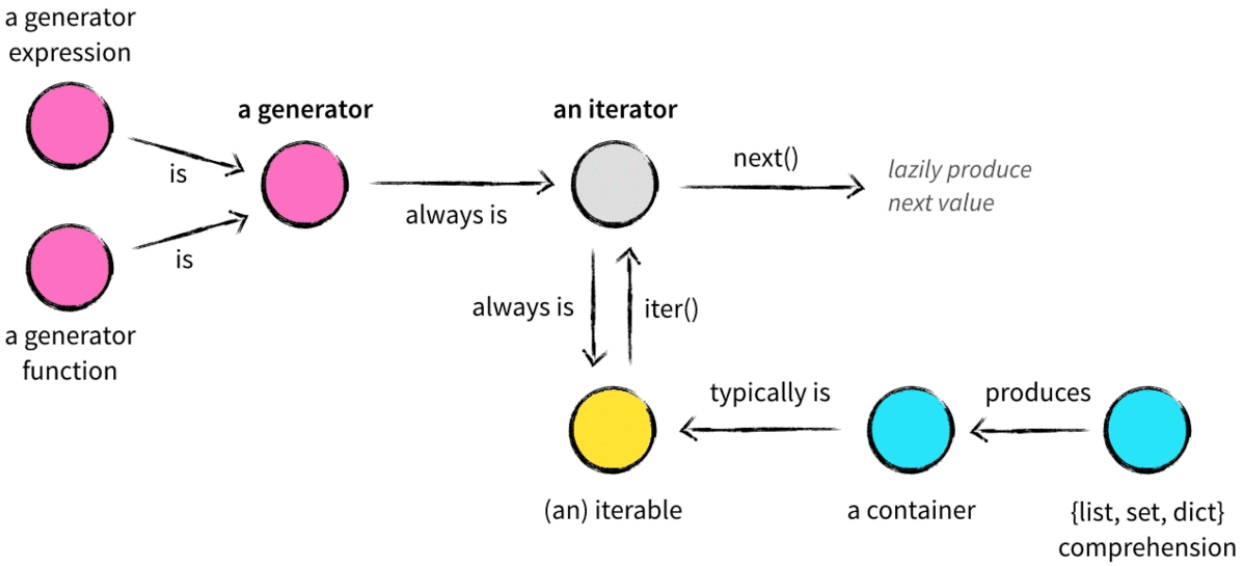
再给你看个善于学习的人的例子：

> https://nvie.com/posts/iterators-vs-generators/

作者 Vincent Driessen 在这个帖子里写到：

> I'm writing this post as a pocket reference for later.

人家随手做个图，都舍不得不精致：




自学能力强的人有个特点，就是不怕麻烦。小时候经常听到母亲念叨，“怕麻烦！那还活着干嘛啊？活着多麻烦啊！” —— 深刻。

## 先关注使用再研究原理

作为人类，我们原本很擅长运用自己并不真正理解的物件、技能、原理、知识的……

三千多年以前，一艘欧洲腓尼基人的商船在贝鲁斯河上航行的时候搁浅了…… 于是，船员们纷纷登上沙滩。饿了怎么办？架火做饭呗。吃完饭，船员们惊讶地发现锅下面的沙地上有很多亮晶晶、闪闪发光的东西！今天的化学知识对当年的他们来说，是那一生不可触摸的“过早引用”。他们并不懂这个东西的本质、原理，但稍加研究，他们发现的是，锅底沾有他们运输的天然苏打…… 于是，他们的总结是，天然苏打和沙子（我们现在知道沙子中含有石英砂）被火烧，可能会产生这个东西。几经实验，成功了。于是，腓尼基人学会了制做玻璃球，发了大财……

两千五六百年之前，释加牟尼用他的理解以及在那个时代有限的概念详细叙述了打坐的感受 —— 他曾连续打坐 6 年。今天，西方科学家们在深入研究脑科学的时候，发现 Meditation 对大脑有特别多的好处…… 这些好处就是好处，与宗教全然没有任何关系的好处。

> - Harvard neuroscientist: Meditation not only reduces stress, here's how it changes your brain
> - This Is Your Brain on Meditation –– The science explaining why you should meditate every day
> - Researchers study how it seems to change the brain in depressed patients
> - Meditation's Calming Effects Pinpointed in the Brain
> - Different meditation types train distinct parts of your brain

你看，我们原本就是可以直接使用自己并不真正理解的物件、技能、原理、知识的！可为什么后来就不行了呢?

或者说，从什么时候开始，我们开始害怕自己并不真正理解的东西，不敢去用，甚至连试都不敢去试了呢？

有一个相当恼人的解释：**上学上坏了**。

学校里教的全都是属于“先声明再使用”的知识。反过来，不属于这种体系架构的知识，学校总是回避的 —— 比如，关于投资与交易的课程，从来看不见地球上有哪个义务教育体系把它纳入教学范围。虽然，投资与交易，是每个人都应该掌握、都必须掌握的不可或缺的技能，某种意义上它甚至比数学语文都更重要，然而，学校就是不会真教这样的东西。

而且，现在的人上学的时间越来越长。小学、初中、高中、本科加起来 16 年…… 这么长时间的“熏陶”，只能给大多数人造成幻觉，严重、深刻，甚至不可磨灭的幻觉，误以为所有的知识都是这种类型…… 可偏偏，这世界上真正有用的、真正必要的知识，几乎全都不是这种类型 —— 颇令人恼火。

现在的你，不一样了 —— 你要跳出来。养成一个习惯：

> 不管怎么样，先用起来，反正，研究透原理，不可能马上做到，需要时间漫漫。

用错了没关系，改正就好。用得不好没关系，用多了就会好。只要开始用起来，理解速度就会加快 —— 实践出真知，不是空话。

有的时候，就是因为没有犯过错，所以不可能有机会改正，于是，就从未做对过。

## 尊重前人的总结和建议

生活中，年轻人最常犯的错误就是把那句话当作屁：

> 不听老人言，吃亏在眼前。

对年轻人来讲，老人言确实很讨厌，尤其是与自己当下的感受相左的时候。

然而，这种“讨厌”的感觉，更多的时候是陷阱，因为那些老人言只不过是过早引用，所以，在年轻人的脑子里“无法执行”，“报错为类型错误”……

于是，很多人一不小心就把“不听老人言”和“独立思考”混淆起来，然后最终自己吃了亏。可尴尬在于，等自己意识到自己吃亏了的时候吧，大量的时间早已飘逝，是为“无力回天”。

你可以观察到一个现象，学霸（好学生）的特点之一就是“老师让干啥就干啥”，没废话。

比如，上面告诉你了，“必须自己动手”，那你就从现在开始老老实实地在一切必要的情况下自己动手去“**总结**、**归纳**、**整理**、**组织** 关键知识点”…… 那你就必然能够学好。但针对这么个建议，你反复在那里问，“为什么呀？”，“有没有更简单的办法啊？”…… 那你就完了，死定了。

学写代码的过程中，有很多重要的东西实际上并不属于“编程语言范畴”。比如，如何为变量命名、如何组织代码，这些“规范”，不是违背了就会马上死掉的[1]；并且，初来乍到的时候，这些东西看起来就是很啰嗦、很麻烦的…… 然而，这些东西若是不遵守，甚至干脆不了解，那么最终的结果是，你永远不可能写出大项目，永远是小打小闹 —— 至于为什么，可以用那句你最讨厌的话回答你：

> 等你长大了就懂了……

自学编程的好处之一，就是有机会让一个人见识到“规范”、“建议”的好处。也有机会让一个人见识到不遵守这些东西会吃怎样的亏（往往是现世报）。

Python 中有一个概念叫 PEP，Python Enhancement Proposals，必须找时间阅读，反复阅读，牢记于心：

> https://www.python.org/dev/peps/pep-0008/

到最后，你会体会到，这不只是编程的事，这种东西背后的思考与体量，对整个人生都有巨大帮助。

**脚注**

[1]：也可能真的会死…… 请看一篇 2018 年 9 月份的一则新闻，发生在旧金山的事情：Developer goes rogue, shoots four colleagues at ERP code maker

↑ Back to Content ↑