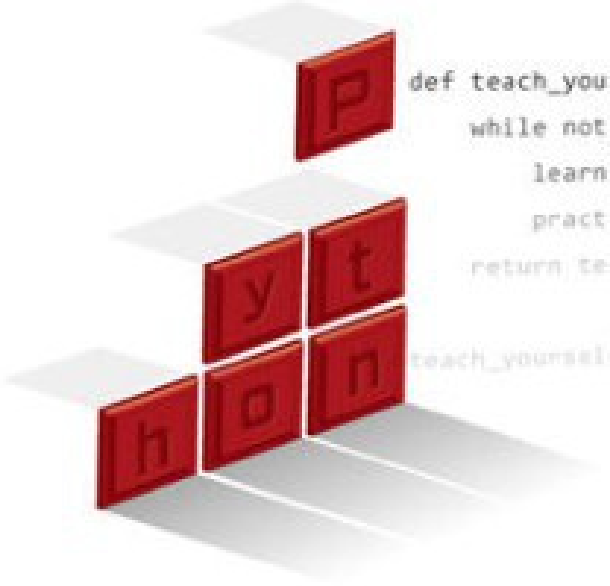
本书为您开启全新的编程自学体验！
您将收获持续的、优秀的自学能力！
Learn，Practice，Build！

Broadview
www.broadview.com.cn

自学是门手艺

没有自学能力的人没有未来

李笑来 著



中国工信出版集团
電子工業出版社
http://www.phei.com.cn

自学是门手艺
没有自学能力的人没有未来
李笑来 著
電子工業出版社

自学是门手艺

没有自学能力的人没有未来

李笑来 2019

本书内容无非是：

def TeachYourSelf(anything):

while not create(somthing):

learn()

practice()

return TeachYourSelf(another)

TeachYourSelf(coding)

特别感谢霍炬（@virushuo）、洪强宁（@hongqn) 两位良师诤友在此书写作过程中给予我的巨大

帮助！

PDF版本由Oritaro（https://readfree.me/accounts/profile/Oritaro/wish/）通过HTML批量转换而来，

图片内容丢失。请访问本书的github仓库获取全部内容。（原书版权协议为CC-BY-NC-ND

license。）

Cover Picture from:

https://glumpy.github.io/_static/screenshots/spiral.png

完整版带图片地址：https://github.com/selfteaching/the-

craft-of-selfteaching

前言

想写一本关于自学能力的书，还真的不是一天两天的事儿，所以肯定不是心血来潮。

等我快把初稿框架搭完，跟霍炬说起我正在写的内容时，霍炬说：

你还记得吗，你第一次背个包来我家的时候，咱们聊的就是咋写本有意思的编程

书……

我说：

真是呢！十三年就这么过去了……

不过，这次真的写了，写出来的其实并不是，或者说，并不仅仅是“一本编程书”。

这本“书”是近些年我一直在做却没做完整的事情，讲清楚“学习学习再学习”：

学会学习之后再去学习……

只不过，这一次我更具体了一点：不是“学习”，而是“自学” —— 这一点点的变化，也是之前十多年

没写顺，可这一次却倾盆而出自成体系的重要原因。

以前，我在写作课里讲，写好的前提就是 “Narrow down your topic” —— 把话题范

围缩小缩小再缩小…… 这次算是给读者一个活生生的实例了罢。

自学能力，对每个个体来说，是在这个变化频率和变化幅度都在不断加大的时代里最具价值的能

力。具备这个能力，不一定能直接增加一个人的幸福感（虽然实际上常常确实能），但，它一定会

缓解甚至消除一个人的焦虑 —— 若是在一个以肉眼可见的方式变化着的环境里生存，却心知肚明自

己已然原地踏步许久，自己正在被这个时代甩在身后，谁能不焦虑呢？

实际上，这些年来我写的书，都是关于学习的，无论是《把时间当作朋友》，还是《通往财富自由

之路》，甚至《韭菜的自我修养》，你去看就知道，背后都是同样的目标：学习、进步 —— 甚至进

化。

这一次的《自学是门手艺》，首先，可以看作是之前内容的“实践版”：

完成这本书的内容，起码会习得一个新技能：编程。

更为重要的是，可以把《自学是门手艺》当作之前内容的“升级版”：

自学能力，是持续学习持续成长的发动机……

仔细观察整个人群，你就会发现一个惊人且惊悚的事实：

至少有 **99%** 的人终生都没有掌握自学能力！

其实这个数字根本不夸张。根据 2017 年的统计数据，从 1977 年到 2017 年，40 年间全国大学录

取人数总计为 1.15 亿左右（11518.2 万），占全国人口数量的 10% 不到，另外，这其中一半以上

是专科生…… 你觉得那些 4% 左右的本科毕业生中，带着自学能力走入社会的比例是多少？不夸张

地讲，我觉得 1% 都是很高的比例了 —— 所以，前面提到的 99% 都是很客气的说法。

绝大多数人，终其一生都没有自学过什么。他们也不是没学过，也不是没辛苦过，但，事实是：他

们在有人教、有人带、有人逼的情况下都没真学明白那些基础知识……更可怕的是，他们学的那些

东西中，绝大多数终其一生只有一个用处：考试。于是，考试过后，那些东西就“考过即弃”了……

不得不承认，应试教育的确是磨灭自学能力的最有效方法。

在随后的生活里，尽管能意识到自己应该学点什么，常有“要是我也会这个东西就好了”的想法，

但，基本上百分之百以无奈结束 —— 再也没有人教、再也没有人带、再也没有人逼…… 于是，每

次“决心重新做人”都默默地改成“继续做人”而后逢年过节再次许愿“重新做人”……

这是有趣而又尴尬的真相：

没有不学习的人。

你仔细观察就知道了，就算被你认为不学无术的人，其实也在学习，只不过，他们的选择不同，他

们想学的是投机取巧，并天天琢磨怎样才能更好地投机取巧……

但，他们不是最倒霉的人。最倒霉的人是那种人，也“认真学了”，可总是最终落得个越来越焦虑的

下场…… 经常有一些人指责另外一些人“贩卖焦虑” —— 根据我的观察，这种指责的肤浅在于，焦虑

不是被卖方贩卖的产品，焦虑其实是买方长期自行积累的结果。

别人无法贩卖给你焦虑，是你自己焦虑 —— 是你自己在为自己不断积累越来越多的焦虑……

然而，又有谁不想解决掉焦虑呢？又有谁不想马上解决掉焦虑呢？

于是，你焦虑，你就要找解决方案。而焦虑的你找到的解决方案，就是花个钱买本书，报个班，找

个老师，上个课……这能说是别人贩卖焦虑给你吗？

自学能力强的人，并非不花钱，甚至他们花的钱可能更多。他们也花钱买书，而且买更多的书；他

们也可能花钱上课，而且要上就上最好的课、最好的班；他们更经常费尽周折找到恰当的人咨询、

求教、探讨 —— 所以，事实上，他们更可能花了更多的钱……

但，自学能力强的人不焦虑，起码他们不会因为学习以及学习过程而焦虑 —— 这是重大差别。

而焦虑的大多数，并不是因为别人贩卖焦虑给他们他们才“拥有”那些焦虑 —— 他们一直在焦虑，并

且越来越焦虑……

为什么呢？总也学不会、学不好，换做是你，你不焦虑吗？！

生活质量就是这样一点一点被消磨掉的 —— 最消耗生活质量的东西，就是焦虑。

我相信，若是《自学是门手艺》这本书真的有用，它的重要用处之一就是能够缓解你的焦虑，让你

明白，首先焦虑没用，其次，有办法也有途径让你摆脱过往一事无成的状况，逐步产生积累，并且

逐步体会到那积累的作用，甚至最后还能感觉到更多积累带来的加速度……到那时候，焦虑就

是“别人的事情”了。

自学没有什么“秘诀”。它是一门手艺，并且，严格意义上来讲，它只是一门手艺。

手艺的特点就是无需天分。手艺的特点就是熟练程度决定一切。从这一点上来看，自学这门手艺和

擀饺子皮没什么区别 —— 就那点事儿，刚开始谁都笨手笨脚，但熟练了之后，就那么回事儿……

而已。

做什么事儿都有技巧，这不可否认。

自学当然也有技巧…… 不过，请做好思想准备：

这儿的空间，没什么新鲜……

—— 这是崔健一首歌里的歌词片段，但放在这里竟然非常恰当到位。

一切与自学相关的技巧都是老生常谈。

中国人说，熟能生巧；老外说，Practice makes perfect —— 你看，与自学相关的技巧，干脆不分

国界……

—— 因为这事儿人类从起点开始就没变过 —— 每代人都有足够多的人在自学这件事儿上挣扎……

有成的有不成的；成的之中有大成有小成……　　可有一件事儿同样不变：留下的文字、留下的信息，

都是大成或者小成之人留下的，不成的人不声不响就销声匿迹。

并且，从各国历史上来看，自学技巧这个话题从未涉及到政治，无论是在东方还是西方都是如此。

结果就是，在自学能力这个小领域中，留下并流传下来的信息，几乎从未被审查，从未被摧毁，从

未被侵犯，从未被扭曲 —— 真的是个特别罕见的“纯净的领域” —— 这真的是整个人类不可想像之

意外好运。

这就是为什么一切的自学技巧到最后肯定是老生常谈的原因。

大部分年轻人讨厌老生常谈。

但这还真的是被误导的结果。年轻人被什么误导了呢？

每一代人都是新鲜出生，每一代人出生时都在同一水准……　随着时间的推移，总是庸者占绝大多

数，这个“绝大多数”不是 51%，不是 70%， 而是 99%！ —— 年轻人吃亏就吃在没把这个现象考虑

进来。

也就是说，虽然有用的道理在不断地传播，可终究还是 99% 的人做不到做不好，于是：

讲大道理的更可能是庸者、失败者，而不是成功者。

人类有很多天赋。就好像我反复提到的那样，“就算不懂也会用”是人类的特长。同样的道理，人类

在这方面同样擅长：

无论自己什么样，在“判断别人到底是不是真的很成功”上，基本上有 99% 的把

握……

所以，十岁不到的时候，绝大多数小朋友就“看穿”了父母，后来再“看穿”了老师…… 发现他们整天

说的都是他们自己做不到的事情……　于是误以为自己“看穿”了整个世界。

那时候小朋友们还没学、或者没学好概率这个重要知识，于是，他们并不知道那只不过是 99% 的情

况，而且更不知道“因素的重要性与它所占的比例常常全无正相关”，所以当然不知道那自己尚未见

到的 1% 才可能是最重要的……

于是，99% 的小朋友们一不小心就把自己“搭了进去”：

不仅讨厌老生常谈，而且偏要对着干，干着干着就把自己变成了另外一个属于那

99% 的另外一个老生……

这是 99% 的人终其一生的生动写照。

做 1% 很难吗？真的很简单，有时仅仅一条就可能奏效：

在自学这件事儿上，重视一切老生常谈**……**

很难吗？不难，只不过是一个“开关”而已。

当我动手写这本“书”的时候，是 47 岁那年（2019）的春节前 —— 显然，这个时候我也早就是一

位“老生”了…… 并且，这些道理我已经前后讲了二十年！算是“常谈”甚至“长谈”了罢……

开始在新东方开始教书那年，我 28 岁；用之前那一点三脚猫的编程能力辅助着去写《TOEFL 核心

词汇 21 天突破》是 2003 年；后来写《把时间当作朋友》是 2007 年，这本书的印刷版出版发行是

在 2009 年；再后来陆续写了很多内容，包括没有纸质版发行只有在线版的《人人都能用英语》

（2013）；以及因为在罗振宇的得到 App 上开专栏，把之前写过的

《学习学习再学习》重构且扩充

而出版的《通往财富自由之路》（2017）；甚至连《韭菜的自我修养》（2018）都是讲思考、学

习、和认知升级的……

说来说去，就那些事儿 —— 没什么新鲜。

这中间也有很多写了却没写完，或者因为写得自己不满意扔在柜子里的东西，比如《人人都是工程

师》（2016）—— 哈！我就是这么坚韧，有了目标就会死不放弃…… 3 年后的今天，我终于用那

个时候完全想不到的方式完成了当时的目标，并且，做到了很多 3 年前自己都完全想象不到的事

情。

在写当前这本《自学是门手艺》的过程中，我从一开始就干脆没想给读者带来什么“新鲜”的或者“前

所未见”的自学技巧 —— 因为真的就没有，根本就没有什么新鲜的自学技巧…… 没有，真的没有

—— 至少，我自己这么久了还是真的没见识过。

然而，我算是最终能做到的人。知到、知道、得到、做到之间，均各不相同。

二十年前，在拥挤的课堂里坐在台下听我讲课的小朋友们，绝大多数在当时应该没有想到他们遇到

了这样一个人 —— 二十年后，刚认识我的人也不会自动知道我是这样的人。

但是，这些年里，看到我在一点一点进步、从未原地踏步的人很多很多…… 我猜，所谓的“榜样”，

也不过如此了罢。

不夸张地讲，这可能是当前世界上最硬核的鸡汤书了 —— 因为，虽然它就是鸡汤（李笑来自认就是

个鸡汤作者），但，它不是“只是拿话糊弄你”那种，也不是“只不过是善意的鼓励”那种，它是那

种“教会你人生最重要的技能”的鸡汤，并且还不仅仅只有一种技能，起码两个：“自学能力” 和 “编程

能力”…… 而这两个能力中的无论哪一种，都是能确定地提高读者未来收入的技能，对，就是

100% 地确定 —— 有个会计专业的人求职的时候说“我还会编程”且还能拿出作品，你看看他可不

可能找不到工作？你看看他是不是能薪水更高？

#! —— 这是个程序员能看懂的梗。

关键在于，这个老生不是说说而已的老生，他是能够做到的人：

一个末流大学的会计专业毕业的人不得已去做了销售；

这个销售后来去国内最大的课外辅导机构当了 7 年 TOEFL/GRE/GMAT 老师；

这个英语老师后来同时成了很多畅销书、长销书的作者；

这个作者后来居然成了著名天使投资人；

这个投资人后来竟然写了本关于编程入门的书籍；

这本“书”最终竟然还是一个完整的产品，不仅仅是“一本书”……

然而呢？

—— 然而，即便是这样的老生，也讲不出什么新鲜道理。

因为啊，历史上留下来的所有关于自学的技巧，都是人类史上最聪明的人留下来的 —— 你我这样的

人，照做就可以了…… 现在你明白怎么回事儿了吧？

记住罢 ——

千万不要一不小心就把自己搭进去……

李笑来

初稿完成于 *2019* 年 *2* 月 *27* 日

如何证明你真的读过这本书？

積ん読

日语里有个很好玩的词，“積ん読”（tsundoku）：

指那些买回来堆在那里还没读过的（甚至后来干脆不看了的）书……

细想想，每个人都有很多很多“積ん読”。小时候我们拿回家的教科书中就有相当一部分，其实就

是“積ん読”，虽然那时候掏钱买书的是父母，不仔细看、或者干脆不看的时候，也知道自己在偷

懒…… 再后来就是“主动犯罪”了 —— 比如，很多人买到手里的英语词汇书是根本就没有翻到过第二

个列表的，乃至于过去我常常开玩笑说，中国学生都认识一个单词，*abandon*，不是吗？这个单词

是很多很多人“决心重新做人”而后“就这样罢”的铁板钉钉的见证者。

在没有电子书的时代，印刷版书籍多少还有一点“装饰品”功用，可是到了电子书时代，谁知道你的

设备里有多少付费书籍呢？攒下那么多，其实并没有炫耀的地方，给谁

看呢？据说，Kindle 的后台

数据里可以看到清楚的

1

1

2

3

“打开率”，大抵上也是在 /4 ~ /3 之间，也就是说，差不多有 /3 ~ /4 的

电子书籍被购买下载之后，从来就没有被打开过。

如此看来，付费之后并不阅读，只能欺骗一个对象了：自己。跟心理学家们之前想象的不同，我认

为人们通常是不会欺骗自己的，至少很难“故意欺骗自己”。所以，对于“卖了之后坚决不读”这个现

象，我不认为“给自己虚妄的满足感”是最好的解释。

更朴素一点，更接近真相的解释是：

那百分之七八十的人，其实是想着给自己一个希望……

—— 等我有空了一定看。嗯。

说来好笑，其实每个人共同拥有的目标之一是这样的：

成为前百分之二十的少数人……

然而，PK 掉百分之七八十的人的方法真的很简单很简单啊：

把买来的书都真真切切地认真读过就可以了。

这实在是太简单了罢？！可是…… 我知道你刚刚那个没出息的闪念：

那我少买书甚至不买书不就可以了吗？

你自己都知道这是荒谬的，却忍不住为你的小聪明得意 —— 其实吧，幸亏有你们在，否则我们怎么

混进前百分之二十呢？

Processing math: 100%

PoW

比特币这个地球上第一个真正被证明为可行的区块链应用中有一个特别重要的概念，叫做“工作证

明”（Proof of Work）—— 你干活了就是干活了，你没干活就是没干活，你的工作是可被证明

的……

借用这个思路，我设计了个方法，让你有办法证明自己就是看过这本书，就是读完了这本书 —— 你

能向自己也向别人证明自己曾经的工作…… 是不是挺好？

证明的方法是使用 github.com 这个网站以及它所提供的工具 **git**。

具体步骤

请按照以下步骤操作：

1. 注册 github.com 帐号 —— 无论如何你都必须有 github 账户；

2. 使用浏览器访问 https://github.com/xiaolai/the-craft-of-selfteaching；

3. 在页面右上部找到“Fork”按钮，将该仓库 Fork 到你自己的账户中；

4. 使用 git clone 命令或者使用 Desktop for Github 将 the craft of

selfteaching 这个你 Fork 过来的仓库克隆到本地；

5. 按照 Jupyterlab 的安装与配置 的说明在本地搭建好 Jupyterlab —— 如果在

Jupyterlab 中浏览本书的话，其中的所有代码都是可以“当场执行”的，并且，你

还可以直接改着玩……

6. 在阅读过程中，可以不断通过修改文章中的代码作为练习 —— 这样做的结果就

是已阅读过的文件会发生变化…… 每读完一章，甚至时时刻刻，你都可以通过

git commit 命令向你自己 Fork 过来的仓库提交变化 —— 这就是你的阅读

工作证明；

7. 仓库里有一个目录， my-notes ，你可以把你在学习过程中写的笔记放在那

里；

8. 仓库里还有一个另外一个目录， from-readers ；那是用来收集读者反馈的

—— 将来你可以写一篇《我的自学之路》，放在这个目录里，单独创建一个分

支，而后提交 pull request ，接受其他读者投票，若是达到一定的赞同

率，那么你的文章就会被收录到主仓库中被更多人看到，激励更多的人像你一

样走上自学之路……

当然，为了这么做，你还要多学一样反正你早晚都必须学会的东西，Git —— 请参阅附录 《Git 入

门》。

时间就是这样，我们没办法糊弄它。而有了 git 这样的工具之后，我们在什么时候做了什么样的工

作，是很容易证明的 —— 这对我们来说真是天大的好事。

如何使用 **Pull Request** 为这本书校对

另外，在你阅读的过程中，发现有错别字啊、代码错误啊，甚至有“更好的表述”等等，都可以通过

pull request 来帮我改进 —— 这也是一种“工作证明”。

1. 使用浏览器访问 https://github.com/selfteaching/the-craft-of-selfteaching

2. 点击右上角的“Fork 按钮”，将该仓库 Fork 到你的 Github 账户中

1. 创建一个新分支，可以取名为 from- ，比如， from-

git.basic.tutorial ；

1. 在新分支下进行修改某个文件（示例图片中是修改了 README.md 文件），而后提交：

1. 在页面顶部选择 Pull request 标签：

而后点击 Compare & pull request 按钮

确认无误之后，点击 Create pull request 按钮。

1. 随后，Github 用户 @xiaolai —— 就是我，即，the-craft-of-selfteaching 这个仓库的所有者，

会被通知有人提交了 Pull request，我会看到：

在我确认这个 Pull request 修改是正确的、可接受的之后，我就会按 Merge pull request 按

钮 —— 如此这般，一个修正就由你我共同完成了。

注意

提交 Pull request 的时候，最佳策略如下：

尽量每次只提交一个小修改；

提交时尽量简短且清楚地说明修改原因；

耐心等待回复。

当自己的 Fork 过来的仓库已经被你在本地“玩残”了的时候，它千万不能当作用来提交 Pull request

的版本。自己本地怎么玩都无所谓，但，需要向别人提交 Pull request 的时候，必须重新弄一个当

前最新版本到本地，而后再在其基础上修改。

为什么一定要掌握自学能力？

一句话解释清楚：

没有自学能力的人没有未来。

有两个因素需要深入考虑：

未来的日子还很长

这世界进步得太快

我有个观察：

很多人都会不由自主地去复刻父母的人生时刻表。

比如，你也可能观察到了，父母晚婚的人自己晚婚的概率更高，父母晚育的人自己晚育的概率也更

高……

再比如，绝大多数人的内心深处，会不由自主地因为自己的父母在五十五岁的时候退休了，所以就

默认自己也会在五十五岁前后退休……　于是，到了四十岁前后的时候就开始认真考虑退休，在不知

不觉中就彻底丧失了斗志，早早就活得跟已经老了很多岁似的。

但是，这很危险，因为很多人完全没有意识到自己所面临的人生，与父母所面临的人生可能完全不

一样 —— 各个方面都不一样。单举一个方面的例子，也是比较容易令人震惊的方面：

全球范围内都一样，在过去的五十年里，人们的平均寿命预期增长得非常惊人……

拿中国地区做例子，根据世界银行的数据统计，中国人在出生时的寿命预期，从 1960 年的 *47.73*

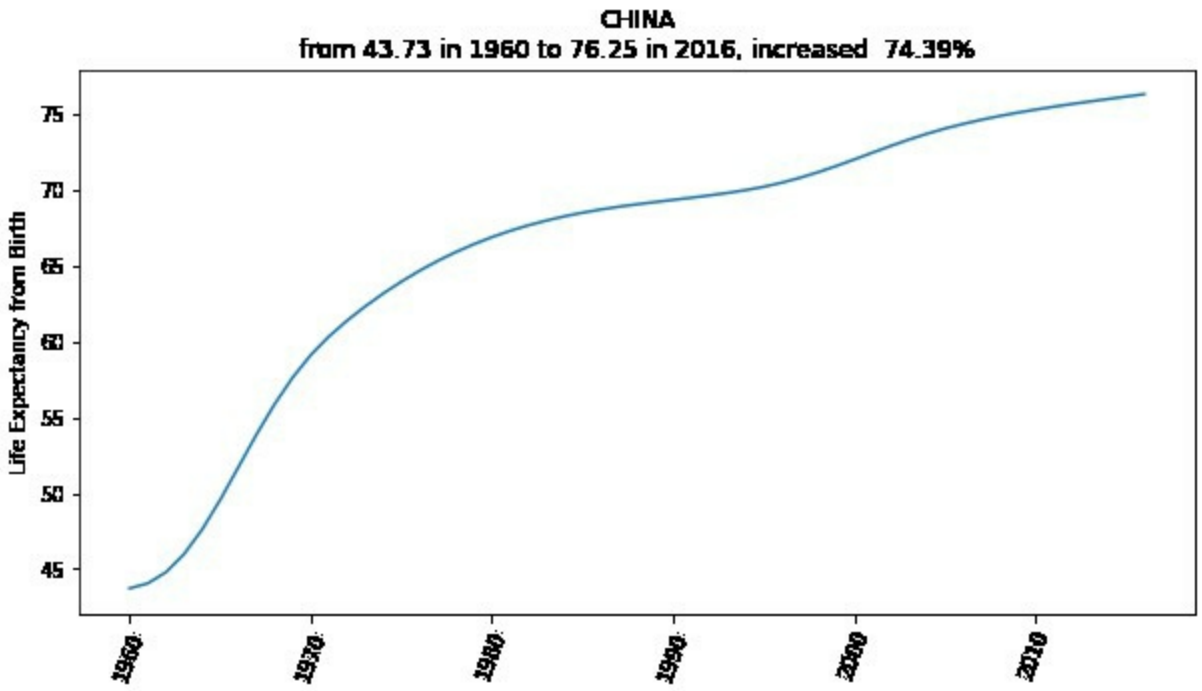
岁，增长到了 2016 年的 *76.25* 岁，56 年间的增幅竟然由 **74.39%** 之多！

In [4]:

```
import matplotlib.pyplot as plt

import numpy as np
```

```
data = np.genfromtxt('life-expectancy-china-1960-2016.txt',
delimiter=',',
names=['x', 'y'])
da1960 = data[0][1]
da2016 = data[len(data)-1][1]
increase = (da2016 - da1960)/da1960
note = 'from {:.2f} in 1960 to {:.2f} in 2016, increased {:.2%}'\
.format(da1960, da2016, increase)
plt.figure(figsize=(10,5))
plt.plot(data['x'], data['y'])
plt.ylabel('Life Expectancy from Birth')
plt.tick_params(axis='x', rotation=70)
plt.title('CHINA\n' + note)
# plt.savefig('life-expectancy-china-1960-2016.png', transparent=True)
plt.show()
```



 # data from:

 # https://databank.worldbank.org/data/reports.aspx?source=2&series=SP.DY

N.LE00.IN

如此发展下去，虽然人类不大可能永生不死，但平均寿命依然在持续延长是个不争的事实。与上一

代不同，现在的千禧一代，需要面对的是百岁人生 —— 毫无疑问，不容置疑。

这么长的人生，比默认的想象中可能要多出近一倍的人生，再叠加上另外一个因素 —— 这是个变化

越来越快的世界 —— 会是什么样子？

我是 1972 年出生的。从交通工具来看，我经历过出门只能靠步行，大街上都是牛车马车，机动车

顶多见过拖拉机，到有自行车，到见过摩托车，到坐小汽车，到自己开

车，到开有自动辅助驾驶功

能的电动车……　从阅读来看，我经历过只有新华书店，到有网络上的文字，到可以在当当上在线买

到纸质书，到有了国际信用卡后可以在 Amazon 上第一时间阅读新书的电子版、听它的有声版，到

现在可以很方便地获取最新知识的互动版，并直接参与讨论……　从技能上来看，我经历过认为不识

字是文盲，到不懂英语是文盲，到不懂计算机是文盲，到现在，不懂数据分析的基本与文盲无

异……

我也见识过很多当年很有用很赚钱很令人羡慕的技能“突然”变成几乎毫无价值的东西，最明显的例

子是驾驶。也就是二十多年前，的哥还是很多人羡慕的职业呢！我本科的时候学的是会计专业，那

时候我们还要专门练习打算盘呢！三十年之后的今天，就算有人打算盘打得再快，有什么具体用处

嘛？我上中学的时候，有个人靠出版字帖赚了大钱 —— 那时候据说只要写字漂亮就能找到好工作；

可今天，写字漂亮与否还是决定工作好坏的决定性因素吗？打印机很便宜啊！

这两个因素叠加在一起的结果就是，这世界对很多人来说，其实是越来越残忍的。

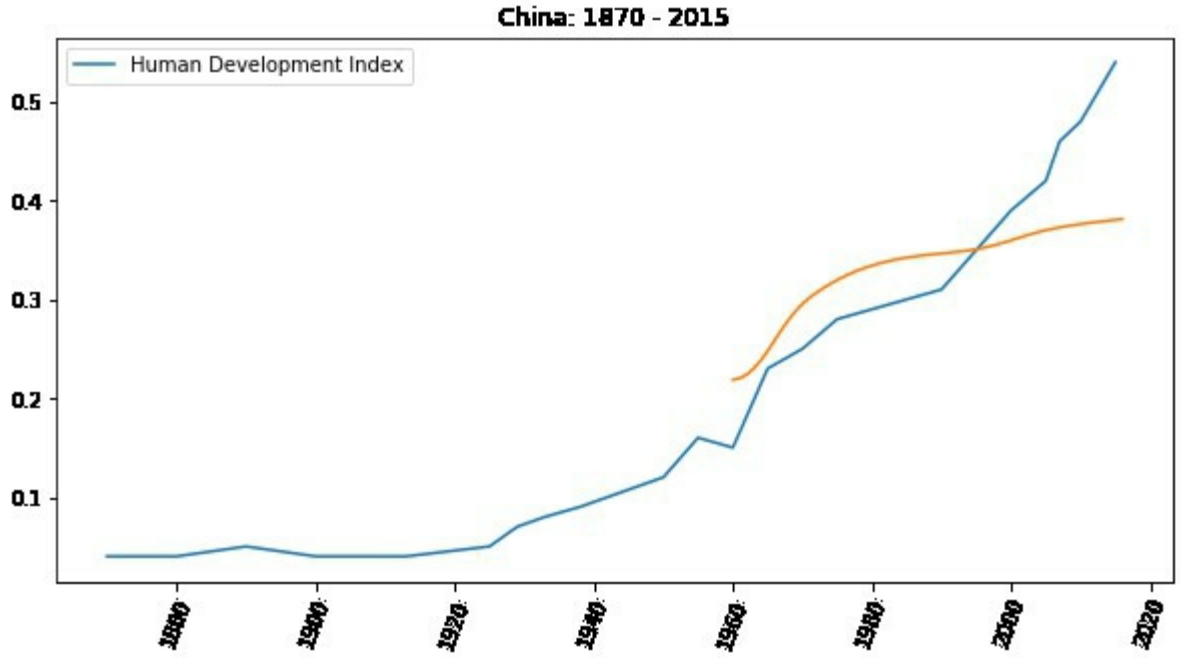
我见过太多的同龄人，早早就停止了进步，早早就被时代甩在身后，早早就因此茫然不知所措 ——

早早晚晚，你也会遇到越来越多这样的人。他们的共同特征只有一个：

没有自学能力

有一个统计指数，叫做人类发展指数（Human Development Index），它的曲线画出来，怎么看都

有即将成为指数级上升的趋势。



In [3]:

import matplotlib.pyplot as plt

import numpy as np

plt.figure(figsize=(10,5))

lebdata = np.genfromtxt('life-expectancy-china-1960-2016.txt',

delimiter=',',

names=['x', 'y'])

hdidata = np.genfromtxt('hdi-china-1870-2015.txt',

delimiter=',',

names=['x', 'y'])

plt.plot(hdidata['x'], hdidata['y'])

plt.tick_params(axis='x', rotation=70)

plt.title('China: 1870 - 2015')

 # plt.savefig('human-development-index.png', transparent=True)

plt.legend(['Human Development Index'])

plt.plot(lebdata['x'], lebdata['y']*0.005)

plt.plot(secondary_y=**True**)

 # plt.savefig('human-development-index-china-1870-2015.png', transparent

=True)

plt.show()

 # data from:

 # https://ourworldindata.org/957e133e-f3e9-4bf2-9627-dbf30ebc9b4d

社会发展越来越快，你要面对的人生越来越长，在那一段与你的直觉猜想并不相同的漫漫人生路

上，你居然没有磨练过自学能力，竟然只能眼睁睁地看着自己被甩下且无能为力，难道接下来要在

那么长的时间里“苦中作乐”吗？

没有未来的日子，怎么过呢？

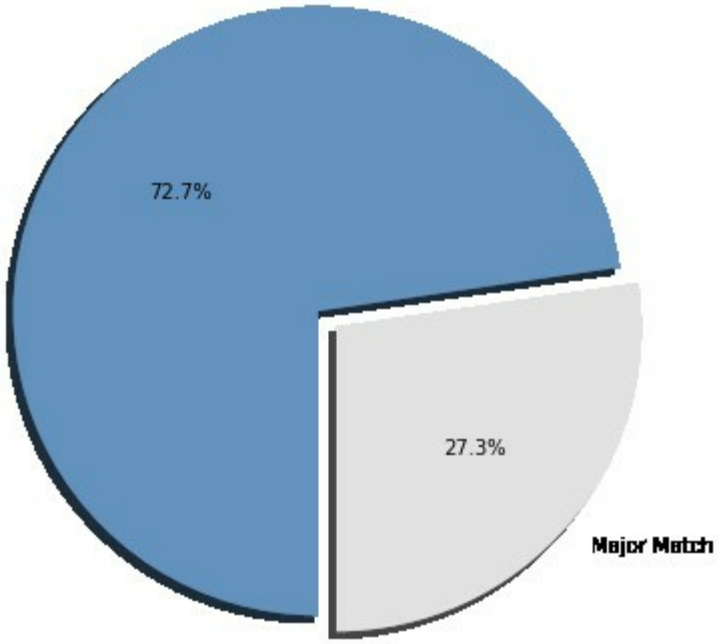
我本科学的是会计，研究生跑到国外读宏观经济学没读完，跑回国内做

计算机硬件批发，再后来去

新东方应聘讲授托福课程，离开新东方之后创业，再后来做投资，这期间不断地写书…… 可事实

上，我的经历在这个时代并不特殊。有多少人再后来的职业生涯中所做的事情与当年大学里所学的

专业相符呢？



纽约联邦储蓄银行在 2012 年做过一个调查，发现人们的职业与自己大学所学专业相符的比例连

30% 都不到。而且，我猜，这个比例会持续下降的 —— 因为这世界变化快，因为大多数教育机构

与世界发展脱钩的程度只能越来越严重……

In [7]:

```
import matplotlib.pyplot as plt

labels = ['Major Match', '']

sizes = [273, 727]

colors = ['#E2E2E2', '#6392BF']

explode = (0, 0.08)

plt.figure(figsize=(7, 7))

plt.pie(sizes,

labels=labels,

explode=explode,

autopct=' %1.1f%%',

colors=colors,

startangle=270,

shadow=True)

# plt.savefig('major-match-job.png', transparent=True)

plt.show()

# data from:

#  https://libertystreeteconomics.newyorkfed.org/2013/05/do-big-cities-he  lp-college-graduates-find-better-jobs.html
```

绝大多数人终生都饱受时间幻觉的拖累。

小时候觉得时间太长，那是幻觉；长大了觉得时间越来越快，那还是幻觉 —— 时间从来都是匀速

的。最大的幻觉在于，总是以为“时间不够了” —— 这个幻觉最坑人。许多年前，有一次我开导我老

婆。她说，“啊？得学五年才行啊？！太长了！”我说，

“你回头看看呗，想想呗，五年前你在做什么？是不是回头一看的时候，五年前就好

像是昨天？道理是一样的，五年之后的某一天你回头想今天，也是‘一转眼五年就过

去’了…… 只不过，你今天觉得需要时间太多，所以不肯学 —— 但是，不管你学还

是不学，五年还是会‘一转眼就过去’的…… 到时候再回头，想起这事儿的时候，没

学的你，一定会后悔 —— 事实上，你已经有很多次后悔过‘之前要是学了就好了’，

不是吗？”

现在回头看，开导是非常成功的。十多年后的今天，她已经真的可以被称为“自学专家” —— 各种运

动在她那儿都不是事儿。健身，可以拿个北京市亚军登上健与美杂志封面；羽毛球，可以参加专业

比赛；潜水，潜遍全球所有潜水胜地，到最后拿到的各种教练证比她遇到的各地教练的都多、更高

级；帆船，可以组队横跨大西洋；爬山，登上喜马拉雅……

都说，人要有一技之长。那这一技究竟应该是什么呢？

自学能力是唯一值得被不断磨练的长技。

磨练出自学能力的好处在于，无论这世界需要我们学什么的时候，我们都可以主动去学，并且还是

马上开始 —— 不需要等别人教、等别人带。

哪怕有很强的自学能力的意思也并不是说，什么都能马上学会、什么都能马上学好，到最后无所不

精无所不通……　因为这里有个时间问题。无论学什么，都需要耗费时间和精力，与此同时更难的事

情在于不断填补耐心以防它过早耗尽。另外，在极端的情况下，多少也面临天分问题。比如身高可

能影响打篮球的表现，比如长相可能影响表演的效果，比如唱歌跑调貌似是很难修复的，比如有些

人的粗心大意其实是基因决定的，等等。不过，以我的观察，无论是什么，哪怕只是学会一点点，

都比不会强。哪怕只是中等水平，就足够应付生活、工作、养家糊口的需求。

我在大学里学的是会计专业，毕业后找不到对口工作，只好去做销售 —— 没人教啊！怎么办？自

学。也有自学不怎么样的时候，比如当年研究生课程我就读不完。后来想去新东方教书 —— 因为听

说那里赚钱多 —— 可英语不怎么样啊！怎么办？自学。离开新东方去创业，时代早就变了，怎么

办？自学，学的不怎么样，怎么办？硬挺。虽然创业这事儿后来也没怎么大成，但竟然在投资领域

开花结果 —— 可赚了钱就一切平安如意了吗？并不是，要面对之前从来没可能遇到的一些险恶与困

境，怎么办？自学。除了困境之外，更痛苦的发现在于对投资这件事来说，并没有受过任何有意义

的训练，怎么办？自学。觉得自己理解的差不多了，一出手就失败，怎

么办？接着学。

我出身一般，父母是穷教师。出生在边疆小镇，儿时受到的教育也一般，也是太淘气 —— 后来也没

考上什么好大学。说实话，我自认天资也一般，我就是那种被基因决定了经常马虎大意的人。岁数

都这么大了，情商也都不是一般的差 —— 还是跟年轻的时候一样，经常莫名其妙就把什么人给得罪

透了……

但，我过得一直不算差。

靠什么呢？人么，一个都靠不上。到最后，我觉得只有一样东西真正可靠 —— 自学能力。于是，经

年累月，我磨练出了一套属于我自己的本领：只要我觉得有必要，我什么都肯学，学什么都能学会

到够用的程度……　编程，我不是靠上课学会的；英语，不是哪个老师教我的；写作，也不是谁能教

会我的；教书，更没有上过师范课程；投资，更没人能教我　——　我猜，也没人愿意教我…… 自己

用的东西自己琢磨，挺好。

关键在于，自学这事儿并不难，也不复杂，挺简单的，因为它所需要的一切都很朴素。

于是，从某个层面上来看，我每天都过的很开心。为什么？因为我有未来。凭什么那么确信？因为

我知道我自己有自学能力。

—— 我希望你也有。

准确地讲，希望你有个更好的未来。

而现在我猜，此刻，你心中也是默默如此作想的罢。

为什么把编程当作自学的入口？

很多人误以为“编程”是很难的事情。

…… 实则不然 —— 这恰恰是我们选择“编程”作为自学的第一个“执行项目”的原因。

一本关于自学能力的书，若是真的能够起到作用，那么它就必须让读者在读之前和读之后不一样

——　比如，之前可能没有自学能力，或者自学能力很差，之后就有了一定的自学能力……

然而，这很难。不但对读者来说很难，对作者来说更难 —— 我当过那么多年被学生高度评价的老

师，出版过若干本畅销且长销的书籍，所以更是清楚地知道例子的重要性。

道理当然很重要；可是，在传递道理的时候，例子相对来看好像更重要。

同样的道理，例子不准，人就可能会理解错；例子不精彩，人就可能听不进去；例子居然可以令人

震惊，那就可以做到让听众、让读者“永生不忘”。

许多年前，有位后来已经在美国读书博士毕业了的学生来信，大意是说：

好多年前，我在新东方上课，听您讲，人学习就好像是动物进化一样…… 很多人很

早就开始停止了进化，本质上跟猴子没啥区别。

那段类比好长，我记不太清楚细节了……　可是，当时我是出了一身汗的，因为我忽

然觉得自己是一只猴子。可是，突然之间，我不想继续做猴子，更不想一直做猴

子！

从那之后，我好像变了一个人似的……　现在我已经博士毕业了，觉得应该写封信告

诉您，我不再是猴子了，最起码是大猩猩，而且我保证，我会一直进化。

……

所以啊，在我看来，写书讲课之前，最重要的工作，也是做得最多的事情，其实就是“找到好例子”

——　那即意味着说，先要找到很多很多恰当合适的例子，而后再通过反复比较试验，挑出那个效果

最好的例子。了解了这一点，将来你准备任何演讲，都会不由自主地多花一点时间在这方面，效果

肯定比“把幻灯片做得更花哨一些”要好太多了罢？

后来，我选中了一个例子：“自学编程” —— “尽量只通过阅读学会编程”。

（一）

选择它的理由，首先就在于：

事实证明，它就是无论是谁都能学会的 —— 千万别不信。

它老少皆宜 —— 也就是说，“只要你愿意”，根本没有年龄差异。十二岁的孩子可以学；十八岁的大

学生可以学；在职工作人员可以学……　就算你已经退休了，想学就能学，谁也拦不住你。

它也不分性别，男性可以学，女性同样可以学，性别差异在这里完全不存在。

它也不分国界，更没有区域差异 —— 互联网的恩惠在于，你在北京、纽约也好，老头沟、门头沟也

罢，在这个领域里同样完全没有任何具体差异。

尤其是在中国。现状是，中国的人口密度极高，优质教育资源的确就是稀缺…… 但，在计算机科学

领域，所有的所谓“优质教育资源”事实上完全没有任何独特的竞争力 —— 编程领域，实际上是当今

世上极为罕见的“教育机会公平之地”。又，不仅在中国如此，事实上，在全球范围内也都是如此。

（二）

编程作为“讲解如何习得自学能力的例子”，实在是太好了。

首先，编程这个东西反正要自学 —— 不信你问问计算机专业的人，他们会如实告诉你的，学校里确

实也教，但说实话都教得不太好……

其次，编程这个东西最适合“仅靠阅读自学” —— 这个领域发展很快，到最后，新东西出来的时候，

没有老师存在，任由你是谁，都只能去阅读“官方文档”，只此一条路。

然后，也是最重要的一条，别管是不是很多人觉得编程是很难的东西，事实上它就是每个人都应该

具备的技能。

许多年前，不识字，被称为文盲……

后来，人们反应过来了，不识英文，也是文盲，因为科学文献的主导语言是英文，读不懂英文，什

么都吃不上热乎的；等菜好不容易端上来了吧，早就凉了不说，味道都常常会变……

再后来，不懂基本计算机操作技能的，也算是文盲，因为他们无论做什么事情，效率都太低下了，

明明可以用快捷键一下子完成的事情，却非要手动大量重复……

到了最近，不懂数据分析的，也开始算作文盲了。许多年前人们惊呼信息时代来了的时候，其实暂

时体会不到什么太多的不同。然而，许多年过去，互联网上的格式化数据越来越多，不仅如此，实

时产出的格式化数据也越来越多，于是，数据分析不仅成了必备的能力，而且早就开始直接影响一

个人的薪资水平。

你作为一个个体，每天都在产生各种各样的数据，然后时时刻刻都在被别人使用着、分析着…… 然

而你自己却全然没有数据分析能力，甚至不知道这事儿很重要，是不是感觉很可怕？你看看周边那

么多人，有多大的比例想过这事儿？反正那些天天看机器算法生成的信息流的人好像就是全然不在

意自己正在被支配……

怎么办？学呗，学点编程罢 —— 巧了，这还真是个正常人都能学会的技能。

（三）

编程作为“讲解如何习得自学能力的例子”最好的地方在于，这个领域的知识结构，最接近每个人所

面对的人生中的知识结构。

这是什么意思呢？

编程入门的门槛之所以高，有个比较特殊的原因：

它的知识点结构不是线性的。

我们在中小学里所遇到的教科书，其中每个章节所涉及到的知识点之间，全都是线性关联。第一章

学好了，就有基础学第二章；在第二章的概念不会出现在第一章之中……

很遗憾，编程所涉及到的知识点没办法这样组织 —— 就是不行。编程教材之所以难以读懂，就是因

为它的各章中的知识点结构不是线性排列的。你经常在某一章读到不知道后面第几章才可能讲解清

楚的概念。

比如，几乎所有的 Python 编程书籍上来就举个例子：

```
print('Hello, world!')
```

姑且不管这个例子是否意义非凡或者意义莫名，关键在于， print() 是个函数，而函数这个概

念，不可能一上来就讲清楚，只能在后面若干章之后才开始讲解……

想要理解当前的知识点，需要依赖对以后才能开始学习的某个甚至多个知识点的深

入了解……

这种现象，可以借用一个专门的英文概念，叫做“**Forward References**”——原本是计算机领域里

的一个术语。为了配合当前的语境，姑且把它翻译为“过早引用”罢，或者“前置引用”也行。

学校里的课本，都很严谨——任何概念，未经声明就禁止使用。所以，学完一章，就能学下一章；

跳到某一章遇到不熟悉的概念，往前翻肯定能找到……

在学校里习惯了这种知识体系的人，离开学校之后马上抓瞎——社会的知识结构不仅不是这样的，

而且_几乎全都不是_这样的。工作中、生活里，充满了各式各样的“过早引用”。为什么总是要到多

年以后你才明白父母曾经说过的话那么有道理？为什么总要到孩子已经长大之后才反应过来当初自

己对孩子做错过很多事情？为什么在自己成为领导之前总是以为他们只不过是在忽悠你？为什么那

么多人创业失败了之后才反应过来当初投资人提醒的一些观念其实是千真万确的？——因为很多概

念很多观念是“过早引用”，在当时就是非常难以理解……

自学编程在这方面的好处在于，在自学的过程中，其实你相当于过了一遍“模拟人生”——于是，面

对同样的“过早引用”，你不会觉得那么莫名其妙，你有一套你早已在“模拟人生”中练就的方法论去应

对。

（四）

另外一个把编程作为“讲解如何习得自学能力的例子”最好的地方在于，

你在这个过程中将不得不习

得英语 —— 起码是英文阅读能力，它能让你在不知不觉中“脱盲”。

学编程中最重要的活动就是“阅读官方文档”。学 Python 更是如此。Python 有很多非常优秀的地

方，其中一个令人无法忽视的优点就是它的文档完善程度极好。它甚至有专门的文档生成工

具，Sphinx：

Sphinx is a tool that makes it easy to create intelligent and beautiful

documentation, written by Georg Brandl and licensed under the BSD license.

It was originally created for the Python documentation, and it has excellent facilities for the documentation of software projects in a range of languages. Of course, this site is also created from reStructuredText sources using Sphinx!

最好的 Python 教程，是官方网站上的 The Python Tutorial，读它就够了。我个人完全没兴趣从头

到尾写一本 Python 编程教材，不仅因为人家写得真好，而且它就放在那里。

虽然你在官方网站上就是很难找到它的中文版，虽然就不告诉你到底在哪里也显得很不厚道，但

是，我建议你就只看英文版 —— 因为离开了这个教程之后，还是要面对绝大多数都是英文的现实。

为了照顾那些也想读完本书，但因为种种原因想着读中文可以快一些的人，链接还是放在这里：

https://docs.python.org/zh-cn/3/tutorial/index.html (for v.3.7.2）

http://www.pythondoc.com/pythontutorial3/ (for v.3.6.3) 我曾经专门写过一

本书发布在网上，叫《人人都能用英语》。其中的观点就是，大多数人之所以在

英语这事儿上很挫，是因为他们花无数的时间去“学 ”，但，就是“不用 ”。学以致用，用以促学。可就

是不用，无论如何就是不用，那英语学了那么多年能学好吗？

自学编程的一个“副作用”就是，你不得不用英语。而且还是天天用，不停地用。

当年我上大学的时候，最初英语当然也不好。不过，因为想读当时还是禁书的《动物庄园》

（Animal Farm），就只好看原版（当时好不容易搞到的是本英法对照版）…… 然后英语阅读就基

本过关了。

这原理大抵上是这样，刚开始，英语就好像一层毛玻璃，隔在你和你很想要了解的内容之间。然

而，由于你对那内容的兴趣和需求是如此强烈，乃至于即便隔着毛玻璃你也会挣扎着去看清楚……

挣扎久了（其实没两天就不一样），你的“视力”就进化了，毛玻璃还在那里，但你好像可以穿透它

看清一切……

自学编程，也算是一举两得了！

（五）

当然，把编程作为“讲解如何习得自学能力的例子”，实在是太好了的最重要原因在于，自学编程对

任何人来说都绝对是：

现实的（Practical）

可行动的（Actionable）

并且还是真正是可达成的（Achievable）

最重要的就是最后这个“可达成的”。虽然对读者和作者来说，一个做到没那么容易，另一个讲清楚

也非常难，但是，既然是所有人都“可达成的”，总得试试吧？但是，请相信我，这事儿比减肥容易

多了 —— 毕竟，你不是在跟基因作斗争。

这只是个起点。

尽量只靠阅读学会编程，哪怕仅仅是入门，这个经历和经验都是极为宝贵的。

自学是门手艺。只不过它并不像卖油翁的手艺那样很容易被别人看到，也不是很容易拿它出来炫耀

——　因为别人看不到么！然而，经年累月，就不一样了，那好处管他别人知不知道，自己却清楚得

很！

你身边总有些人能把别人做不好的事儿做得极好，你一定很羡慕。可他们为什么能做到那样呢？很

简单啊，他们的自学能力强，所以他们能学会大多数自学能力差的人终生学不到的东西。而且他们

的自学能力会越来越强，每学会一样新东西，他们就积累了更多自学经验，难以对外言表的经验，

再遇到什么新东西，相对没那么吃力。

另外，自学者最大的感受就是万物相通。他们经常说的话有这么一

句：“…… 到最后，都是一样的

呢。”

（六）

最后一个好处，一句话就能说清楚，并且，随着时间的推移，你对此的感触会越来越深：

在这个领域里，自学的人最多……

没有什么比这句话更令人舒心的了：相信我，你并不孤独。

只靠阅读习得新技能

习得自学能力的终极目标就是：

有能力只靠阅读就能习得新技能。

退而求其次，是“尽量只靠阅读就习得新技能” —— 当然，刚开始的时候可能需要有人陪伴，一起

学，一起讨论，一起克服困难…… 但，就是要摆脱“没人教、没人带，没人逼，就彻底没戏”的状

态。

小时候总是听大人说：

不是什么东西都可以从书本里学到的……

一度，我觉得他们说的有道理。再后来，隐约感觉这话哪儿有毛病，但竟然又感觉无力反驳……

那时，真被他们忽悠到了；后来，也差点被他们彻底忽悠倒！

幸亏后来我渐渐明白，且越来越相信：

自己生活工作学习上遇到的所有疑问，书本里应该都有答案 —— 起码

有所参考。

“不是什么东西都可以从书本里学到的……”这话听起来那么有道理，只不过是因为自己读书不够

多、不够对而已。

过了 25 岁，我放弃了读小说，虚构类作品，我只选择看电影；而非虚构类作品，我选择尽量只读

英文书，虽然那时候买起来很贵也很费劲，但我觉得值 —— 英文世界和中文世界的文化风格略有不

同。在英文世界里，你看到的正常作者好像更多地把“通俗易懂”、“逻辑严谨”当作最基本的素养；而

在中文世界里，好像“故弄玄虚”、“偷梁换柱”更常见一些；在英文世界里，遇到读不懂的东西可以很

平静地接受自己暂时的愚笨，心平气和地继续努力就好；在中文世界里，遇到装神弄鬼欺世盗名

的，弄不好最初根本没认出来，到最后跟“认贼作父”一样令人羞辱难当不堪回首。

说实话，我真觉得这事儿跟崇洋媚外没什么关系。我是朝鲜族，去过韩国，真觉得韩国的书更没法

看（虽然明显是个人看法）…… 2015 年年底，我的律师告诉我，美国移民就快帮我办下来了，可

那时候我开始觉得美国政府也各种乱七八糟，于是决定停止办理。我是个很宅的人，除了餐馆基本

上哪儿都不去，陪家人朋友出去所谓旅游的时候，我只不过是换个房间继续宅着…… 可这些都不是

重点，重点在于：

知识原本就应该无国界…… 不是吗？不是吗！

再说，这些年我其实还读了不少中国人写的英文书呢，比如，张纯如的书很值得一看；郑念的 Life

and Death in Shanghai，真的很好很好。我也读了不少老外写的关于中国的书 —— 这些年我一直

推荐费正清的剑桥中国史（The Cambridge History of China），当然有中文版的，不过，能读英文

版的话感受很不一样。

当然，英文书里同样烂书也很多，烂作者也同样一大堆，胡说八道欺世盗名的一大串…… 但，总体

上来看，非小说类著作质量的确更高一点。

还有，英语在科学研究领域早已成为“主导语言”（Dominant Language）也是不争的事实。不过，

英语成为主导语言的结果，就是英语本身被不断“强奸”，外来语越来越多，“Long time no see”被辞

典收录就是很好的例子。事实上，英语本身就是个大杂烩……

读书越多越明白读书少会被忽悠…… 很多人真的会大头捣蒜一般地认同“不是什么东西都可以从书

本里学到的……”

另外，很多人在如此教导小朋友的时候，往往是因为“人心叵测”啊，“江湖险恶”啊，所以害怕小朋友

吃亏。可事实上，如若说，人间那些勾心斗角的事儿貌似从书本里学不来的话，其实也不过还是历

史书看少了 —— 勾心斗角的套路历史上全都被反复用过了。倒是有本中文书值得吐血推荐，民国时

代的作者连阔如先生写的《江湖丛谈》，粗略扫过你就知道了，江湖那点事儿，也早就有人给你里

里外外翻了个遍…… 只不过这书不太容易买到就是了。

我也遇到过这样的反驳：

书本能教会你做生意吗？！

说实话，去回驳这个反驳还真挺难，因为反驳者是脑容量特别有限才能说出这种话 —— 世界上有那

么多商学院都是干嘛的？搞得它们好像不存在一样。首先，它们的存在说明，商业这事儿是有迹可

循的，是可学习的。其次，商业类书籍非常多，是非虚构类书籍中的一大品类；更为重要的是，做

生意这事儿，看谁做 —— 有本事（即，比别人拥有更多技能）的人做生意和没本事的人做生意，用

同样的商业技巧，能有一样的效果吗？最后啊，这世界在这方面从来没有变过：一技傍身的人，总

是不愁生活……

更为重要的是，这才几年啊，互联网本身已经成了一本大书 —— 关于全世界的一整本大书。仅仅是

10 多年前，大约在 2008 年前后，经过几年发展的 Wikipedia 被众多西方大学教授们群起而攻，指

责它错误百出…… 可现在呢？Wikipedia 好像有天生的自我修复基因，它变得越来越值得信赖，越

来越好用。

七零后八零后长大的过程中，还经常被父母无故呵斥：“怎么就你事儿这么多！”或者无奈敷衍：“等

你长大了就明白了……” 九零后、零零后呢？他们很少有什么疑问需要向父母提问，直接问搜索引

擎，效果就是父母们天天被惊到甚至吓倒。最近两年更不一样了，我有朋友在旧金山生活，他的孩

子整天跟 Google 说话，有点什么问题，就直接“Hi Google...”

我长大的那个年代，一句“通过阅读了解世界”好像还是很抽象甚至很不现实的话，现在呢？现在，

除了阅读之外，你还能想出什么更有效的方法吗？反正我想不出。

有个很有趣的现象：

人么，只要识字，就忍不住阅读……

只不过，人们阅读的选择很不同而已。有自学能力的人和没有自学能力的人，在这一点上很容易分

辨：

有自学能力的人，选择阅读“有繁殖能力”的内容；

没有自学能力的人，阅读只是为了消磨时光……

我把那些能给你带来新视野，能让你改变思考模式，甚至能让你拥有一项新技能的内容称之为“有繁

殖能力的内容”。

人都一样，拥有什么样的能力之后，就会忍不住去用，甚至总是连下意识中也要用。

那些靠阅读机器算法推送的内容而杀时间的人，恰恰就是因为他们有阅读能力才去不断地读，读啊

读，像是那只被打了兴奋剂后来死在滚轮上的小白鼠。如果这些人哪怕有一点点自学能力，那么他

们很快就会分辨出自己正在阅读的东西不会刺激自己的产出，只会消磨自己的时间；那么，他们就

会主动放弃阅读那些杀时间的内容，把那时间和精力自然而然地用在筛选有繁殖能力的内容，让自

己进步，让自己习得更多技能上去了。

所以，只要你有一次“只靠阅读习得一项新技能”的经验，你就变成另外一个人了。你会不由自主、

哪怕下意识里都会去运用你新习得的能力……　从这个角度看，自学很上瘾！能上瘾，却不仅无害，

还好处无穷，这样的好事，恐怕也就这一个了罢。

我有过只靠阅读学会游泳的经历……　　听起来不像真的吧？更邪门的是，罗永浩同学的蛙泳，是我站

在游泳池边，仅靠言语讲解，就让他从入水就扑腾开始三十分钟之内可以开始蛙泳的 —— 虽然当天

他第一次蛙泳，一个来回五十米都坚持不下来。

仅靠阅读学会新技能不仅是可能的，并且，你随后会发现的真相是：

绝大多数情况下，没人能教你，也不一定有人愿意教你……　到最后，你想学会或你

必须学会什么东西的时候，你只能靠阅读！　—— 因为其实你谁都靠不上……

我有很多偶像，英国数学家乔治·布尔就是其中一个 —— 因为他就是个基本上只靠阅读自学成才的

人。十八、九岁，就自学了微积分　　——　　那是将近两百年前，没有

Google，没有 Wikipedia…… 然

后他还自己创办了学校，给自己打工…… 从来没有上过大学，后来却被皇家学院聘请为该学院第一

个数学教授。然后，人家发明的布尔代数，在百年之后引发了信息革命…… 达芬奇也是这样的人

—— 要说惨，他比所有人都惨…… 因为几乎从一开始就貌似没有谁有资格有能力教他。

这些例子都太遥远了。给你讲个我身边的人，我亲自打过很长时间交道的人 —— 此人姓邱，人称邱

老板。

邱老板所写的区块链交易所引擎，在 Github 上用的是个很霸气的名字，“貔貅”（英文用了一个生造

的词，Peatio） —— 这个 Repo 至 2019 年春节的时候，总计有 2,913 个 Star，有 2,150 个

Fork…… 绝对是全球这个领域中最受关注的开源项目。2017 年 9 月，云币应有关部门要求关闭之

前，是全球排名前三的区块链交易所。

邱老板当年上学上到几年级呢？初中未读完，所以，跟他聊天随口说成语是很有负担的，因为他真

的可能听不懂…… 然而，他的编程、他的英语，全是自学的…… 学到什么地步呢？学到可以创造极

有价值的商业项目的地步。他什么学习班都没上过，全靠阅读 —— 基本上只读互联网这本大书。

讲真，你没有选择，只靠阅读习得新技能，这是你唯一的出路。

开始阅读前的一些准备

内容概要

关于 Python 编程的第一部分总计 7 章，主要内容概括为：

1. 以布尔值为入口开始理解程序本质

2. 了解值的分类和运算方法

3. 简要了解流程控制的原理

4. 简要了解函数的基本构成

5. 相对完整地了解字符串的操作

6. 了解各种容器的基础操作

7. 简要了解文件的读写操作

阅读策略

首先，不要试图一下子就全部搞懂。这不仅很难，在最初的时候也完全没必要。

因为这部分的知识结构中，充满了“过早引用”。请在第一遍粗略完成第 1 部分中的 E1 ~ E7 之后，

再去阅读《如何从容应对“过早引用”？》。

其次，这一部分，注定要反复阅读若干遍。

在开始之前，要明确这一部分的阅读目标。

这一部分的目标，不是让你读完之后就可以开始写程序；也不是让你读完之后就对编程或者 Python

编程有了完整的了解，甚至不是让你真的学会什么……　这一部分的目标，只是让你“脱盲”。

不要以为脱盲是很容易的事情。你看，所有人出生的时候，都天然

是“文盲”；人们要上好多年的

学，才能够真正“脱盲” —— 仔细想想吧，小学毕业的时候，所有人就真的彻底脱盲了吗？

以中文脱盲为例，学字的同时，还要学习笔划；为了学更多的字，要学拼音，要学如何使用《新华

字典》……

学会了一些基础字之后，要学更多的词，而后在练习了那么多造词造句之后，依然会经常用错……

你看，脱盲，和阅读能力强之间距离很长呢；不仅如此，阅读能力强和写作能力强之间的距离更

长……

反复阅读这一部分的结果是：

你对基本概念有了一定的了解

你开始有能力相对轻松地阅读部分官方文档

你可以读懂一些简单的代码

仅此而已。

心理建设

当我们开始学习一项新技能的时候，我们的大脑会不由自主地紧张。可这只不过是多年之间在学校

里不断受挫的积累效应 —— 学校里别的地方不一定行，可有个地方特别行：给学生制造全方位、无

死角、层层递进的挫败感。

可是，你要永远记住两个字：

别怕！

用四个字也行：

啥也别怕！

六个字也可以：

没什么可怕的！

我遇到最多的孱弱之语大抵是这样的：

我一个文科生……

哈哈，从某个层面望过去，其实吧，编程既不是文科也不是理科……它更像是“手工课”。你越学就

越清楚这个事实，它就好像是你做木工一样，学会使用一个工具，再学会使用另外一个工具，其实

总共就没多少工具。然后，你更多做的是各种拼接的工作，至于能做出什么东西，最后完全靠你的

想象力……

十来岁的孩子都可以学会的东西，你怕什么？

别怕，无论说给自己，还是讲给别人，都是一样的，它可能是人生中最重要的鼓励词。

关于这一部分内容中的代码

所有的代码，都可以在选中代码单元格（Code Cell）之后，按快捷键

或 ^

执行，查看

结果。

少量执行结果太长的代码，其输出被设置成了“Scrolled”，是可以通过触摸板或鼠标滑轮上下滑动

的。

为了避免大量使用 print() 才能看到输出结果，在很多的代码单元格中，开头插入了以下代码：

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

你可以暂时忽略它们的意义和工作原理。注意：有时，你需要在执行第二次的时候，才能看到全部

输出结果。

另外，有少量代码示例，为了让读者每次执行的时候看到不同的结果，使用了随机函数，为其中的

变量赋值，比如：

```
import random

r = random.randrange(1, 1000)
```

同样，你可以暂时忽略它们的意义和工作原理；只需要知道因为有它们在，所以每次执行那个单元

格中的代码会有不同的结果就可以了。

如果你不是直接在网站上浏览这本“书”、或者是在阅读印刷版，而是在本地自己搭建 Jupyterlab 环

境使用，那么请参阅附录《Jupyterlab 的安装与配置》。

注意：尤其需要仔细看看《Jupyterlab 的安装与配置》的《关于 Jupyter lab themes》这一小节 —— 否则，阅读体验会有很大差别。

另外，如果你使用的是 nteract 桌面版 App 浏览 .ipynb 文件，那么有些

使用了 input() 函数的代

码是无法在 nteract 中执行的。

入口

“速成”，对绝大多数人[1]来说，在绝大多数情况下，是不大可能的。

编程如此，自学编程更是如此。有时，遇到复杂度高一点的知识，连快速入门都不一定是很容易的

事情。

所以，这一章的名称，特意从“入门”改成了“入口” —— 它的作用是给你“指一个入口”，至于你能否从

那个入口进去，是你自己的事儿了……

不过，有一点不一样的地方，我给你指出的入口，跟别的编程入门书籍不一样 —— 它们几乎无一例

外都是从一个 “Hello World!” 程序开始的…… 而我们呢？

让我们从认识一个人开始罢……

乔治·布尔

1833 年，一个 18 岁的英国小伙脑子里闪过一个念头：

逻辑关系应该能用符号表示。

这个小伙子叫乔治·布尔（George Boole，其实之前就提到过我的这位偶像），于 1815 年出生于距

离伦敦北部 120 英里之外的一个小镇，林肯。父亲是位对科学和数学有着浓厚兴趣的鞋匠。乔治·布

尔在父亲的影响下，靠阅读自学成才。14 岁的时候就在林肯小镇名声大噪，因为他翻译了一首希腊

语的诗歌并发表在本地的报纸上。

到了 16 岁的时候，他被本地一所学校聘为教师，那时候他已经在阅读微积分书籍。19 岁的时候布

尔创业了 —— 他办了一所小学，自任校长兼教师。23 岁，他开始发表数学方面的论文。他发明

了“操作演算”，即，通过操作符号来研究微积分。他曾经考虑过去剑桥读大学，但后来放弃了，因

为为了入学他必须放下自己的研究，还得去参加标准本科生课程。这对一个长期只靠自学成长的人

来说，实在是太无法忍受了。

1847 年，乔治 32 岁，出版了他人生的第一本书籍，THE MATHEMATICAL ANAYSIS OF LOGIC

—— 18 岁那年的闪念终于成型。这本书很短，只有 86 页，但，最终它竟然成了人类的瑰宝。在书

里，乔治·布尔很好地解释了如何使用代数形式表达逻辑思想。

1849 年，乔治·布尔 34 岁，被当年刚刚成立的女皇学院（Queen's College）聘请为第一位数学教

授。随后他开始写那本最著名的书，AN INVESTIGATION OF THE LAWS OF THOUGHT。他在前

言里写到：

“The design of the following treatise is to investigate the fundamental laws of those operations of the mind by which reasoning is performed; to give expression to them in the symbolical language of a Calculus, and upon this foundation to

establish the science of Logic and construct its method; …”

“本书论述的是，探索心智推理的基本规律；用微积分的符号语言进行表达，并在此

基础上建立逻辑和构建方法的科学……”

Processing math: 100%

在大学任职期间，乔治·布尔写了两本教科书，一本讲微积分方程，另外一本讲差分方程，而后

者，A TREATISE ON DIFFERENTIAL EQUATIONS，直到今天，依然难以超越。

乔治·布尔于 1864 年因肺炎去世。

乔治·布尔在世的时候，人们并未对他的布尔代数产生什么兴趣。直到 70 年后，克劳德·香农

（Claude Elwood Shannon）发表那篇著名论文，A SYMBOLIC ANALYSIS OF RELAY AND

SWITCHING CIRCUITS 之后，布尔代数才算是开始被大规模应用到实处。

有本书可以闲暇时间翻翻，The Logician and the Engineer: How George Boole and Claude

Shannon Created the Information Age。可以说，没有乔治·布尔的布尔代数，没有克劳德·香农的逻

辑电路，就没有后来的计算机，就没有后来的互联网，就没有今天的信息时代 —— 世界将会怎样？

2015 年，乔治·布尔诞辰 200 周年，Google 设计了专门的 Logo 纪念这位为人类作出巨大贡献的自

学奇才。

Google Doodle 的寄语是这样的：

A very happy **11001000** *th* birthday to genius George Boole!

布林运算

从定义上来看，所谓程序（Programs）其实一点都不神秘。

因为程序这个东西，不过是按照一定_顺序_完成任务的流程（Procedures）。根据定义，日常生活

中你做盘蛋炒饭给自己吃，也是完成了一个“做蛋炒饭”的程序 —— 你按部就班完成了一系列的步

骤，最终做好了一碗蛋炒饭给自己吃 —— 从这个角度望过去，所有的菜谱都是程序……

只不过，菜谱这种程序，编写者是人，执行者还是人；而我们即将要学会写的程序，编写者是人，

执行者是计算机 —— 当然，菜谱用自然语言编写，计算机程序由程序员用编程语言编写。

然而，这些都不是最重要的差异 —— 最重要的差异在于计算机能做布林运算（Boolean

Operations）。

于是，一旦代码编写好之后，计算机在执行的过程中，除了可以“按照顺序执行任务”之外，还可

以“根据不同情况执行不同的任务”，比如，“如果条件尚未满足则重复执行某一任务”。

计算器和计算机都是电子设备，但，计算机更为强大的原因，用通俗的说法就是它“可编

程”（Programable） —— 而所谓可编程的核心就是_布林运算_及其相应的流程控制（Control

Flow）；没有布林运算能力就没有办法做流程控制；没有流程控制就只能“按顺序执行”，那就显

得“很不智能”……

布尔值

在 Python 语言中，布尔值（Boolean Value）用 True 和 False 来表示。

注意：请小心区分大小写 —— 因为 Python 编译器是大小写敏感的，对它来说， True 和 true 不

是一回事。

任何一个逻辑表达式都会返回一个布尔值。

In [1]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

# 请暂时忽略以上两行 ……

1 == 2

1 != 2
```

Out[1]: True

1 == 2 ，用自然语言描述就是“*1* 等于 *2* 吗？” —— 它的布尔值当然是 False 。

1 != 2 ，用自然语言描述就是“*1* 不等于 *2* 吗？” —— 它的布尔值当然是 True 。

注意：自然语言中的“等于”，在 Python 编程语言中，使用的符号是 ==

，不是一个等号！

请再次注意：单个等号 = ，有其他的用处。初学者最不适应的就是，在编程语言里所使用的操作

符，与他们之前在其他地方已经习惯了的使用方法并不相同 —— 不过，适应一段时间就好了。

逻辑操作符

Python 语言中的逻辑操作符（Logical Operators）如下表所示 —— 为了理解方便，也可以将其称

为“比较操作符”。

比较操作符

意义

示例

布尔值

==

等于

1 == 2

False

!=

不等于

1 != 2

True

>

大于

1 > 2

False

>=

大于等于

1 >= 1

True

<

小于

1 < 2

True

<=

小于等于

1 <= 2

True

in

属于

'a' in 'basic'

True

除了等于、大于、小于之外，Python 还有一个逻辑操作符， in ：

这个表达式 'a' in 'basic' 用自然语言描述就是：

“ 'a' 存在于 'basic' 这个字符串之中吗？”（属于关系）

布林运算操作符

以上的例子中，逻辑操作符的运算对象（Operands）是数字值和字符串值。

而针对布尔值进行运算的操作符很简单，只有三种：与、或、非：

分别用 and 、 or 、 not 表示

注意：它们全部是小写。因为只有两个值，所以布林运算结果只有几种而已，如下图所示：

先别管以下代码中 print() 这个函数的工作原理，现在只需要关注其中布林运算的结果：

In [2]:

print('(True and False) yields:', **True and False**)

print('(True and True) yields:', **True and True**)

print('(False and True) yields:', **False and True**)

print('(True or False) yields:', **True or False**)

print('(False or True) yields:', **False or True**)

print('(False or False) yields:', **False or False**)

print('(not True) yields:', **not True**)

print('(not False) yields:', **not False**)

(True and False) yields: False

(True and True) yields: True

(False and True) yields: False

(True or False) yields: True

(False or True) yields: True

(False or False) yields: False

(not True) yields: False

(not False) yields: True

千万不要误以为布林运算是_理科生_才必须会、才能用得上的东西……文理艺分科是中国的特殊分

类方式，真挺害人的。比如，设计师们在计算机上创作图像的时候，也要频繁使用_或与非_的布林

运算操作才能完成各种图案的拼接…… 抽空看看这个网页： Boolean Operations used by Sketch

App —— 这类设计软件，到最后是每个人都用得上的东西呢。另，难道艺术生不需要学习文科或者

理科？—— 事实上，他们也有文化课……

流程控制

有了布林运算能力之后，才有_根据情况决定流程_的所谓流程控制（Flow Control）的能力。

In [3]:

import random

r = random.randrange(1, 1000)

请展示忽略以上两句的原理，只需要了解其结果：

引入随机数，而后，每次执行的时候， *r* 的值不同

if r % 2 == 0:

print(r, 'is even.')

else:

print(r, 'is odd.')

693 is odd.

你可以多执行几次以上程序，看看每次不同的执行结果。执行方法是，选中上面的 Cell 之后按快捷

键 shift + enter 。

现在看代码，先忽略其它的部分，只看关键部分：

...

if r % 2 == 0:

...

else:

...

这个 if/else 语句，完成了流程的分支功能。 % 是计算余数的符号，如果 r 除以 2 的余数等于

0 ，那么它就是偶数，否则，它就是奇数 —— 写成布林表达式，就是 r

% 2 == 0 。

这一次，你看到了单个等号 = ： r = random.randrange(1, 1000) 。

这个符号在绝大多数编程语言中都是“赋值”（Assignment）的含义。

在 r = 2 之中， r 是一个 名称为 r 的变量（Variable）—— 现在只需要将变量理解为程序保存数

值的地方；而 = 是赋值符号， 2 是一个整数常量（Literal）。

语句 r = 2 用自然语言描述就是：

“把 2 这个值保存到名称为 r 的变量之中”。

现在先别在意头两行代码的工作原理，只关注它的工作结果：random.randrange(1, 1000) 这部分代

码的作用是返回一个 *1* 到 *1000* 之前（含 1 但不含 1000 ）的随机数。每次执行以上的程序，它

就生成一个新的随机数，然后因为 = 的存在，这个随机数就被保存到 r 这个变量之中。

计算机程序的所谓“智能”（起码相对于计算器），首先是因为它能做布林运算。计算机的另外一个

好处是“不知疲倦”（反正它也不用自己交电费），所以，它最擅长处理的就是“重复”，这个词在程序

语言中，术语是循环（Loop）。以下程序会打印出 10 以内的所有奇数：

In [4]:

```
for i in range(10):

if i % 2 != 0:
```

print(i)

1

3

5

7

9

其中 range(10) 的返回值，是 0～9 的整数序列（不包含 10 ）。

用自然语言描述以上的程序，大概是这样的 —— 自然语言写在 # 之后：

for i **in** range(10): # 对于 *0*～ *9* 中的所有数字都带入 *i* 这个变量，执行一遍以下任

务：

if i % 2 != 0: # 如果 *i* 除以 *2* 的余数不等于零的话，执行下面的语句：

print(i) # 向屏幕输出 *i* 这个变量中所保存的值

就算你让它打印出一百亿以内的奇数，它也毫不含糊 —— 你只需要在 range() 这个函数的括号里

写上一个那么大的整数就行……

让它干一点稍微复杂的事儿吧，比如，我们想要打印出 100 以内所有的质数（Primes）。

根据质数的定义，它大于等于 2 ，且只有在被它自身或者 1 做为除数时余数为 0 。判断一个数

字是否是质数的算法是这样的：

设 n 整数， n >= 2 ；

若 n == 2 ， n 是质数；

若 n > 2 ，就从 2 开始，一直到 n - 1 作为除数，_逐一_算算看看余数

是否等于 0 ？

如果是，那就不用接着算了，它不是质数；

如果全部都试过了，余数都不是 0 ，那么它是质数。

于是，你需要两个嵌套的循环，第一个负责 n 从 2 到 99 （不包含 100 ）的循环；而这内部，

需要另外一个负责 i 从 2 到 n 的循环：

In [5]:

```
for n in range(2, 100):

if n == 2:

print(n)

continue

for i in range(2, n):

if (n % i) == 0:

break

else: # 这里目前你可能看不懂 …… 但，先关注结果吧。

print(n)
```

2

3

5

7

11

13

17

19

23

29

31

37

41

43

47

53

59

61

67

71

73

79

83

89

97

所谓算法

以上的算法可以改进（程序员们经常用的词汇是“优化”）：

从 2 作为除数开始试，试到 √n 之后的一个整数就可以了……

In [6]:

```
for n in range(2, 100):
if n == 2:
print(n)
continue
for i in range(2, int(n ** 0.5)+1): #为什么要 +1 以后再说 …… n 的 1/2 次方，
相当于根号 n。
if (n % i) == 0:
break
else:
print(n)
```

2

3

5

7

11

13

17

19

23

29

31

37

41

43

47

53

59

61

67

71

73

79

83

89

97

你看，寻找更有效的算法，或者说，不断优化程序，提高效率，最终是程序员的工作，不是编程语

言本身的工作。关于判断质数最快的算法，可以看 Stackoverflow 上的讨论，有更多时间也可以翻

翻 Wikipedia。

到最后，所有的工具都一样，效用取决于使用它的人。所以，学会使用工具固然重要，更为重要的

是与此同时自己的能力必须不断提高。

虽然写代码这事儿刚开始学起来好像门槛很高，那只不过是幻觉，其实门槛比它更高的多的去了。

到最后，它就是个最基础的工具，还是得靠思考能力 —— 这就好像识字其实挺难的 —— 小学初中

高中加起来十来年，我们才掌握了基本的阅读能力；可最终，即便是本科毕业、研究生毕业，真的

能写出一手好文章的人还是少之又少一样 —— 因为用文字值得写出来的是思想，用代码值得写出来

的是创造，或者起码是有意义的问题的有效解决方案。有思想，能解决问题，是另外一门手艺 ——

需要终生精进的手艺。

所谓函数

我们已经反复见过 print() 这个函数（Functions）了。它的作用很简单，就是把传递给它的值输出

到屏幕上 —— 当然，事实上它的使用细节也很多，以后慢慢讲。

现在，最重要的是初步理解一个函数的基本构成。关于函数，相关的概念有：函数名（Function

Name）、参数（Paramters）、返回值（Return Value）、调用（Call）。

拿一个更为简单的函数作为例子， abs() 。它的作用很简单：接收一个数字作为参数，经过运算，

返回该数字的绝对值。

In [7]:

a = abs(-3.1415926)

a

Out[7]: 3.1415926

在以上的代码的第 1 行中，

我们_调用_了一个_函数名_为 abs 的函数；写法是 abs(-3.1415926) ；

这么写，就相当于向它_传递_了一个参数，其值为： -3.1415926 ；

该函数接收到这个参数之后，根据这个参数的_值_在函数内部进行了运算；

而后该函数返回了一个值，_返回值_为之前接收到的参数的值的绝对值

3.1415926 ；

而后这个_值_被保存到变量 a 之中。

从结构上来看，每个函数都是一个完整的程序，因为一个程序，核心构成部分就是输入、处理、输

出：

它有输入 —— 即，它能接收外部通过参数传递的值；

它有处理 —— 即，内部有能够完成某一特定任务的代码；尤其是，它可以根

据“输入”得到“输出”；

它有输出 —— 即，它能向外部输送返回值……

被调用的函数，也可以被理解为子程序（Sub-Program） —— 主程序执行到函数调用时，就开始执

行实现函数的那些代码，而后再返回主程序……

我们可以把判断一个数字是否是质数的过程，写成函数，以便将来在多处用得着的时候，随时可以

调用它：

In [8]:

```
def is_prime(n): # 定义 is_prime()，接收一个参数

if n < 2: # 开始使用接收到的那个参数（值）开始计算 …..

return False # 不再是返回给人，而是返回给调用它的代码 …..

if n == 2:

return True

for m in range(2, int(n**0.5)+1):

if (n % m) == 0:

return False
```

else:

return True

for i **in** range(80, 110):

if is_prime(i): # 调用 *is_prime()* 函数，

print(i) # 如果返回值为 *True*，则向屏幕输出 *i*

83

89

97

101

103

107

109

细节补充

语句

一个完整的程序，由一个或者多个语句（Statements）构成。通常情况下，建议每一行只写一条语

句。

In [9]:

for i **in** range(10):

if i % 2 != 0:

```
print(i)
```

```
1
```

```
3
```

```
5
```

```
7
```

```
9
```

语句块

在 Python 语言中，行首空白（Leading whitespace，由空格 ' ' 或者 Tab

构成）有着特殊的

含义。

如果有行首空白存在，那么，Python 将认为这一行与其他邻近有着相同行首空白的语句同属于一

个语句块 —— 而一个语句块必然由一个行末带有冒号 ： 的语句起始。同属于一个语句块中的语

句，行首空白数量应该相等。这看起来很麻烦，可实际上，程序员一般都使用专门的文本编辑器，

比如 Visual Studio Code，其中有很多的辅助工具，可以让你很方便地输入具备一致性的行首空

白。

以上程序，一共三个语句，两个语句块，一个 for 循环_语句块_中包含着一个 if 条件语句块。注

意第一行和第二行末尾的冒号：。

在很多其他的语言中，比如，Javascript，用大括号 {} 作为语句块标示 —— 这是 Python 比较特

殊的地方，它组织语句块的方式如下图所示：

注意：在同一个文件里，不建议混合使用 Tab 和 Space；要么全用空格，要么全用

制表符。

注释

在 Python 程序中可以用 # 符号标示注释语句。

所谓的注释语句，就是程序文件里写给人看而不是写给计算机看的部分。本节中的代码里就带着很

多的注释。

人写的 Python 语言代码，要被 Python 编译器翻译成机器语言，而后才能让计算机“读懂”，随后计

算机才可以按照指令执行。编译器在编译程序的过程中，遇到 # 符号，就会忽略其后的部分（包

括这个注释符号）。

操作符

在本节，我们见到的比较操作符可以比较它左右的值，而后返回一个布尔值。

我们也见过两个整数被操作符 % 连接，左侧作为被除数，右侧作为除数， 11 % 3 这个表达式的

值是 2 。对于数字，我们可用的操作符有 + 、 - 、 * 、 / 、 // 、 % 、 ** —— 它们分别代表

加、减、乘、除、商、余、幂。

赋值符号与操作符的连用

你已经知道变量是什么了，也已经知道赋值是什么了。于是，你看到 x = 1 就明白了，这是为 x

赋值，把 1 这个值保存到变量 x 之中去。

但是，若是你看到 x += 1 ，就迷惑了，这是什么意思呢？

这只是编程语言中的一种惯用法。它相当于 x = x + 1 。

看到 x = x + 1 依然会困惑…… 之所以困惑，是因为你还没有习惯把单等号 = 当作赋值符号，把

双等号 == 当作逻辑判断的“等于”。

x = x + 1 的意思是说，把表达式 x + 1 的值保存到变量 x 中去 —— 如此这般之后， x 这个变

量中所保存的就不再是原来的值了……

In [10]:

```
x = 0

x += 1

print(x)
```

1

其实不难理解，只要习惯了就好。理论上，加减乘除商余幂这些操作符，都可以与赋值符号并用。

In [11]:

```
x = 11

x %= 3 # x = x % 3
```

```
print(x)
```

2

总结

以下是这一章中所提到的重要概念。了解它们以及它们之间的关系，是进行下一步的基础。

数据：整数、布尔值；操作符；变量、赋值；表达式

函数、子程序、参数、返回值、调用

流程控制、分支、循环

算法、优化

程序：语句、注释、语句块

输入、处理、输出

编译器

你可能已经注意到了，这一章的小节名称罗列出来的话，看起来像是一本编程书籍的目录 —— 只不

过是概念讲解顺序不同而已。事实上还真的就是那么回事儿。

这些概念，基本上都是独立于某一种编程语言的（Language Independent），无论将来你学习哪一

种编程语言，不管是 C++，还是 Javascript，抑或是 Golang，这些概念都在那里。

学会一门编程语言之后，再学其它的就会容易很多 —— 而且，当你学会了其中一个之后，早晚你会

顺手学其它的，为了更高效使用微软办公套件，你可能会花上一两天时间研究一下 VBA；为了给自

己做个网页什么的，你会顺手学会　Javascript；为了修改某个编辑器插件，你发现人家是用 Ruby

写的，大致读读官方文档，你就可以下手用 Ruby 语言了；为了搞搞数据可视化，你会发现不学会

R 语言有点不方便……

你把这些概念装在脑子里，而后就会发现几乎所有的编程入门教学书籍结构都差不多是由这些概念

构成的。因为，所有的编程语言基础都一样，所有的编程语言都是我们指挥计算机的工具。无论怎

样，反正都需要输入输出，无论什么语言，不可能没有布林运算，不可能没有流程控制，不可能没

有函数，只要是高级语言，就都需要编译器……　所以，掌握这些基本概念，是将来持续学习的基

础。

脚注

[1]：对于自学能力强、有很多自学经验的人来说，速成往往是真的是可能、可行的。因为他们已经

积累的知识与经验会在习得新技能时发挥巨大的作用，乃至于他们看起来相对别人花极少的时间就

能完成整个自学任务。也就是说，将来的那个已经习得自学能力、且自学能力已经磨练得很强的

你，常常真的可以做到在别人眼里“速成”。

↑Back to Content↑

值及其相应的运算

从结构上来看，一切的计算机程序，都由且只由两个最基本的成分构成：

运算（Evaluation）

流程控制（Control Flow）

没有流程控制的是计算器而已；有流程控制的才是可编程设备。

看看之前我们见过的计算质数的程序：（按一下

，即 ESC，确保已经进入命令模式，

L

可以切换是否显示代码行号）

In [1]:

```
def is_prime(n): # 定义 is_prime()，接收一个参数
if n < 2: # 开始使用接收到的那个参数（值）开始计算 ……
return False # 不再是返回给人，而是返回给调用它的代码 ……
if n == 2:
return True
for m in range(2, int(n**0.5)+1):
if (n % m) == 0:
return False
else:
return True
for i in range(80, 110):
```

if is_prime(i): # 调用 *is_prime()* 函数，

print(i) # 如果返回值为 *True*，则向屏幕输出 *i*

83

89

97

101

103

107

109

if... ， for... 在控制流程：在什么情况下运算什么，在什么情况下重复运算什么；

第 13 行 is_prime() 这个函数的调用，也是在控制流程 —— 所以我们可以把函数看作是**“**子程

序**”**；

一旦这个函数被调用，流程就转向开始执行在第 1 行中定义的 is_prime() 函数内部的代码，而这段

代码内部还是_计算_和流程控制，决定一个返回值 —— 返回值是布尔值；再回到第 13 行，将返回

值交给 if 判断，决定是否执行第 14 行……

而计算机这种可编程设备之所以可以做流程控制，是因为它可以做布林运算，即，它可以_对布尔

值_进行操作，而后将布尔值交给_分支_和_循环_语句，构成了程序中的流程控制。

值

从本质上看，程序里的绝大多数语句包含着运算（Evaluation），即，在对某个值进行评价。这里

的“评价”，不是“判断某人某事的好坏”，而是“计算出某个值究竟是什么” —— 所以，我们用中文

的“运算”翻译这个“*Evaluation*”可能表达得更准确一些。

在程序中，被运算的可分为常量（Literals）和变量（Variables）。

```
a = 1 + 2 * 3

a += 1

print(a)
```

在以上代码中，

1 、 2 、 3 ，都是常量。 *Literal* 的意思是是“字面的”，顾名思义，常量的_值_就是它字面上的

值。 1 的值，就是 1 。

a 是变量。顾名思义，它的值将来是可变的。比如，在第 2 句中，这个变量的_值_发生了改变，

之前是 7 ，之后变成了 8 。

第 1 句中的 + 、 * ，是操作符（Operators），它用来对其左右的值进行相应的_运算_而后得到一

个值。先是由操作符 * 对 2 和 3 进行运算， 生成一个值， 6 ；然后再由操作符 + 对 1 和

6 进行运算，生成一个值 7 。先算乘除后算加减，这是操作符的优先级决定的。

= 是赋值符号，它的作用是将它右边的值保存到左边的变量中。

_值_是程序的基础成分（Building　blocks），它就好像盖房子用的砖块一样，无论什么样的房子，

到最后都主要是由砖块构成。

常量，当然有个值 —— 就是它们字面所表达的值。

_变量_必须先_赋值_才能使用，也就是说，要先把一个_值_保存到变量中，它才能在其后被运算。

在　Python　中每个函数都有返回值，即便你在定义一个函数的时候没有设定返回值，它也会加上默

认的返回值 None ……（请注意 None 的 大小写！）

In [2]:

```
def f():
pass
print(f()) # 输出 f() 这个函数被调用后的返回值， None
print(print(f())) # 这一行最外围的 print() 调用了一次 print(f())，所以输出一个 None，
# 而后再输出这次调用的返回值，所以又输出一次 None
```

```
None
None
None
```

当我们调用一个函数的时候，本质上来看，就相当于：

我们把一个值交给某个函数，请函数根据它内部的运算和流程控制对其

进行操作而

后返回另外一个值。

比如， abs() 函数，就会返回传递给它的_值_的绝对值； int() 函数，会将传递给它的值的小数部

分砍掉； float() 接到整数参数之后，会返回这个整数的浮点数形式：

In [3]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

abs(-3.14159)

int(abs(-3.14159))

float(abs(-3.14159))
```

Out[3]: 3.14159

值的类型

在编程语言中，总是包含最基本的三种数据类型：

布尔值（Boolean Value)

数字（Numbers）：整数（Int）、浮点数（Float）、复数（Complex

Numbers）

字符串（Strings）

既然有不同类型的数据，它们就分别对应着不同类型的值。

运算的一个默认法则就是，通常情况下应该是相同类型的值才能相互运算。

显然，数字与数字之间的运算是合理的，但你让 + 这个操作符对一个字符串和一个数字进行运算

就不行：

In [4]:

from **IPython.core.interactiveshell** **import** InteractiveShell InteractiveShell.ast_node_interactivity = "all"

11 + 10 - 9 * 8 / 7 // 6 % 5

'3.14' + 3 # 这一句会报错

Out[4]: 20.0

TypeError Traceback (most recent call last)

in

3

4 11 + 10 - 9 * 8 / 7 // 6 % 5

----> 5 '3.14' + 3 # 这一句会报错

TypeError: can only concatenate str (not "int") to str

所以，在不得不对不同类型的值进行运算之前，总是要事先做 **Type Casting**（类型转换）。比如，

将字符串转换为数字用 int() 、 float() ；

将数字转换成字符串用 str() ；

另外，即便是在数字之间进行计算的时候，有时也需要将整数转换成浮点数字，或者反之：

将整数转换成浮点数字用 float()；

将浮点数字转换成整数用 int()；

有个函数， type()，可以用来查看某个值属于什么类型：

In [ ]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

```
type(3)
```

```
type(3.0)
```

```
type('3.14')
```

```
type(True)
```

```
type(range(10))
```

```
type([1,2,3])
```

```
type((1,2,3))
```

```
type({1,2,3})
```

```
type({'a':1, 'b':2, 'c':3})
```

操作符

针对不同类型的数据，有各自专用的操作符。

数值操作符

针对数字进行计算的操作符有加减乘除商余幂： + 、 - 、 * 、 / 、 // 、 % 、 ** 。

其中 + 和 - 可以对单个值进行操作， -3 ；其它的操作符需要有两个值

才能操作。

从优先级来看，这些操作符中：

对两个值进行操作的 + 、 - 的优先级最低；

而后是 * 、 / 、 // 、 % ；

而后是对单个值进行操作的 + 、 - ；

而后是 ** 。

完整的操作符优先级列表，参见官方文档：

https://docs.python.org/3/reference/expressions.html#operator-precedence

布尔值操作符

针对与布尔值，操作符有 and 、 or 、 not 。

它们之中，优先级最低的是 or ，然后是 and , 优先级最高的是 not ：

In [ ]:

True and False or not True

最先操作的是 not ，因为它优先级最高。所以，上面的表达式相当于 True and False or (not

True) ， 即相当于 True and False or False ；

然后是 and ，所以， True and False or False 相当于是 (True and False) or False ，即相当于 False or False ；

于是，最终的值是 False 。

逻辑操作符

数值之间还可以使用逻辑操作符， 1 > 2 返回布尔值 False 。逻辑操作符

有： <（小

于）、 <= （小于等于）、 > （大于）、 >= （大于等于）、 != （不等于）、 == （等于）。

逻辑操作符的优先级，高于布尔值的操作符，低于数值计算的操作符。

In [ ]:

```
n = -95

n < 0 and (n + 1) % 2 == 0
```

字符串操作符

针对字符串，有三种操作：

拼接： + 和 ' '（后者是空格）

拷贝： *

逻辑运算： in 、 not in ； 以及， < 、 <= 、 > 、 >= 、 != 、 ==

In [ ]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

'Awesome' + 'Python'

'Awesome' 'Python'

'Python, ' + 'Awesome! ' * 3

'o' in 'Awesome' and 'o' not in 'Python'
```

字符之间，字符串之间，除了 == 和 != 之外，也都可以被逻辑操作符 < 、 <= 、 > 、 >= 运

算：

In [ ]:

'a' < 'b'

这是因为字符对应着 Unicode 码，字符在被比较的时候，被比较的是对应的 Unicode 码。

In [ ]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

'A' > 'a'

ord('A')

ord('a')

当字符串被比较的时候，将从两个字符串各自的第一个字符开始逐个比较，“一旦决出胜负马上停

止”：

In [ ]:

'PYTHON' > 'Python 3'

列表的操作符

数字和字符串（由字符构成的序列）是最基本的数据类型，而我们往往需要批量处理数字和字符

串，这样的时候，我们需要数组（Array）。不过，在 Python 语言中，它提供了一个容

器（Container）的概念，用来容纳批量的数据。

Python 的容器有很多种 —— 字符串，其实也是容器的一种，它的里面容纳着批量的字符。

我们先简单接触一下另外一种容器列表（List）。

列表的标示，用方括号 [] ；举例来说， [1, 2, 3, 4, 5] 和 ['ann', 'bob', 'cindy', 'dude', 'eric'] ，或者

['a', 2, 'b', 32, 22, 12] 都是一个列表。

因为列表和字符串一样，都是有序容器（容器还有另外一种是无序容器），所以，它们可用的操作

符其实相同：

拼接： + 和 ' '（后者是空格）

拷贝： *

逻辑运算： in 、 not in ；以及， < 、 <= 、 > 、 >= 、 != 、 ==

两个列表在比较时（前提是两个列表中的数据元素类型相同），遵循的还是跟字符串比较相同的规

则：“一旦决出胜负马上停止”。但，实际上，由于列表中可以包含不同类型的元素，所以，通常情

况下没有实际需求对他们进行“大于、小于”的比较。（比较时，类型不同会引发 TypeError ……）

In [ ]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

a_list = [1, 2, 3, 4, 5]

b_list = [1, 2, 3, 5]
```

c_list = ['ann', 'bob', 'cindy', 'dude', 'eric']

a_list > b_list

10 **not in** a_list

'ann' **in** c_list

更复杂的运算

对于数字进行加、减、乘、除、商、余、幂的操作，对于字符串进行拼接、拷贝、属于的操作，对

布尔值进行或、与、非的操作，这些都是相对简单的运算。

更为复杂一点的，我们要通过调用函数来完成 —— 因为在函数内部，我们可以用比“单个表达式”更

为复杂的程序针对传递进来的参数进行运算。换言之，函数就相当于各种事先写好的子程序，给它

传递一个值，它会对其进行运算，而后返回一个值（最起码返回一个 None ）。

以下是 Python 语言所有的内建函数（Built-in Functions）：

(

Python

Built-in

Functions

)

abs()

delattr()

hash()

memoryview()

set()

all()

dict()

help()

min()

setattr()

any()

dir()

hex()

next()

slice()

ascii()

divmod()

id()

object()

sorted()

bin()

enumerate()

input()

oct()

staticmethod()

bool()

eval()

int()

open()

str()

breakpoint()

exec()

isinstance()

ord()

sum()

bytearray()

filter()

issubclass()

pow()

super()

bytes()

float()

iter()

print()

tuple()

callable()

format()

len()

property()

type()

chr()

frozenset()

list()

range()

vars()

classmethod()

getattr()

locals()

repr()

zip()

compile()

globals()

map()

reversed()

import()

complex()

hasattr()

max()

round()

现在倒不用着急一下子全部了解它们 —— 反正早晚都会的。

这其中，针对数字，有计算绝对值的函数 abs() ，有计算商余的函数 divmod() 等等。

In [ ]:

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

abs(-3.1415926)

divmod(11, 3)

这些内建函数也依然只能完成“基本操作”，比如，对于数字，我们想计算三角函数的话，内建函数

就帮不上忙了，于是，我们需要调用标准库（Standard Library）中的 math 模块（Module）：

In [ ]:

import math

math.sin(5)

代码 math.sin(5) 这里的 . ，也可以被理解为“操作符”，它的作用是：

从其它模块中调用函数。

代码 math.sin(5) 的作用是：

把 5 这个值，传递给 math 这个模块里的 sin() 函数，让 sin() 根据它内部的代

码对这个值进行运算，而后返回一个值（即，计算结果）。

类（Class）中定义的函数，也可以这样被调用 —— 虽然你还不明白类（Class）究竟是什么，但，

从结构上很容易理解，它实际上也是保存在其他文件中的一段代码，于是，那段代码内部定义的函

数，也可以这样调用。

比如，数字，其实属于一个类，所以，我们可以调用那个类里所定义的函数，比

如， float.as_integer_ratio() ，它将返回两个值，第一个值除以第二个值，恰好等于传递给它的那个

浮点数字参数：

In [ ]:

3.1415926.as_integer_ratio()

关于布尔值的补充

当你看到以下这样的表达式，而后再看看它的结果，你可能会多少有点迷惑：

In [ ]:

True or 'Python'

这是因为 Python 将 True 定义为：

By default, an object is considered true unless its class defines either a __bool__() method that returns False or a __len__() method that returns zero, when called with the object.

https://docs.python.org/3/library/stdtypes.html#truth-value-testing

这一段文字，初学者是看不懂的。但，下一段就好理解了：

Here are most of the built-in objects considered False :

constants defined to be false: None and False .

zero of any numeric type: 0 , 0.0 , 0j , Decimal(0) ,

Fraction(0, 1)

empty sequences and collections: '' , () , [] , {} ,

set() , range(0)

所以， 'Python' 是个非空的字符串，即，不属于是‘empty sequences’，所以它不被认为是

False ，即，它的布尔值是 True

于是，这么理解就轻松了：

每个变量或者常量，除了它们的值之外，同时还相当于有一个对应的布尔值。

关于值的类型的补充

除了数字、布尔值、字符串，以及上一小节介绍的列表之外，还有若干数据类型，比如

range() （等差数列）、 tuple （元组）、 set （集合）、 dictionary （字典），再比如 Date

Type （日期）等等。

它们都是基础数据类型的各种组合 —— 现实生活中，更多需要的是把基础类型组合起来构成的数

据。比如，一个通讯部，里面是一系列字符串分别对应着若干字符串和数字。

entry[3662] = {

'first_name': 'Michael',

'last_name': 'Willington',

'birth_day': '12/07/1992',

'mobile': {

'+714612234',

'+716253923'

}

'id': 3662,

...

}

针对不同的类型，都有相对应的操作符，可以对其进行运算。

这些类型之间有时也有不得相互运算的需求，于是，在相互运算之前同样要 *Type Casting*，比如将

List 转换为 Set，或者反之：

In [ ]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

a = [1, 2, 3, 4, 5, 6, 7]

b = set(a)

c = list(b)

a

b

c
```

总结

回到最开始：从结构上来看，一切的计算机程序，都由且只由两个最基本的成分构成：

运算（Evaluation）

流程控制（Control Flow）

这一章主要介绍了基础数据类型的运算细节。而除了基础数据类型，我们需要由它们组合起来的更

多复杂数据类型。但，无论数据的类型是什么，被操作符操作的总是该数据的值。所以，虽然绝大

多数编程书籍按照惯例会讲解“数据类型”，但，为了究其本质，我们在这里关注的是“值的类型”。虽

然只是关注焦点上的一点点转换，但，实践证明，这一点点的不同，对初学者更清楚地把握知识点

有巨大的帮助。

针对每一种_值_的类型，无论简单复杂，都有相应的操作方式：

操作符

值运算

逻辑运算

函数

内建函数

其他模块里的函数

其本身所属类之中所定义的函数

所以，接下来要学习的，无非就是熟悉各种数据类型，及其相应的操作，包括能对它们的_值_进行

操作的操作符和函数；无论是操作符还是函数，最终都会返回一个相应的值，及其相应的布尔值

—— 这么看来，编程知识结构没多复杂。因为换句话讲，

接下来你要学习的无非是各种_数据类型_的_运算_而已。

另外，虽然现在尚未来得及对函数进行深入讲解，但最终你会发现它跟操作符一样，在程序里无所

不在。

备注

另外，以下几个链接先放在这里，未来你会返回来参考它们，还是不断地参考它们：

关于表达式：https://docs.python.org/3/reference/expressions.html

关于所有操作的优先

级：https://docs.python.org/3/reference/expressions.html#operator-

precedence

上一条链接不懂 BNF 的话根本读不懂：https://en.wikipedia.org/wiki/Backus-Naur_form

Python 的内建函数：https://docs.python.org/3/library/functions.html

Python 的标准数据类型：https://docs.python.org/3/library/stdtypes.html

另外，其实所有的操作符，在 Python 内部也是调用函数完成的……

https://docs.python.org/3.7/library/operator.html

流程控制

在相对深入了解了值的基本操作之后，我们需要再返回来对流程控制做更深入的了解。

之前我们看过这个寻找质数的程序：

```
for n in range(2, 100):
if n == 2:
print(n)
continue
for i in range(2, n):
if (n % i) == 0:
break
else:
print(n)
```

这其中，包含了_分支_与循环 —— 无论多复杂的流程控制用这两个东西就够了，就好像无论多复

杂的电路最终都是由通路和开路仅仅两个状态构成的一样。

今天的人们觉得这是“天经地义”的事情，可实际上并非如此。这是 1966 年的一篇论

文所带来的巨大改变 —— *Communications of the ACM* by Böhm and Jacopini

(1966)。实际上，直到上个世纪末， GOTO 语句才从各种语言里近乎“灭绝”……

任何进步，无论大小，其实都相当不容易，都非常耗时费力 —— 在哪儿都一样。有

兴趣、有时间，可以去浏览 Wikipedia 上的简要说明 —— Wikipedia: Minimal

structured control flow。

if 语句

if 语句的最简单构成是这样 —— 注意第 1 行末尾的冒号：

if expression:

statements

如果表达式 expression 返回值为真，执行 if 语句块内部的 statements ，否则，什么都不做，执

行 if 之后的下一个语句。

In [12]:

import random

r = random.randrange(1, 1000)

if r % 2 == 0:

print(f' **{r}** is even.')

如果，表达式 expression 返回值无论真假，我们都需要做一点相应的事情，那么我们这么写：

if expression:

statements_for_True

else:

statements_for_False

如果表达式 expression 返回值为真，执行 if 语句块内部的 statements_for_True ， 否则，就执行

执行 else 语句块内部的 statements_for_False

In [15]:

import random

r = random.randrange(1, 1000)

if r % 2 == 0:

print(f' **{r}** is even.')

else:

print(f' **{r}** is odd.')

126 is even.

有时，表达式 返回的值有多种情况，并且针对不同的情况我们都要做

相应的事情，

那么可以这么写：

if expression_1:

statements_for_expression_1_True

elif expression_2:

statements_for_expression_2_True

elif expression_3:

statements_for_expression_3_True

elif expression_...:

statements_for_expression_..._True

Python 用 elif 处理这种多情况分支，相当于其它编程语言中使用 swith 或者 case ……

elif 是 else if 的缩写，作用相同。

以下程序模拟投两个骰子的结果 —— 两个骰子数字加起来，等于 7 算平，大于 7 算大，小于

7 算小：

In [5]:

import random

r = random.randrange(2, 13)

if r == 7:

print('Draw!')

elif r < 7:

print('Small!')

elif r > 7:

print('Big!')

Big!

当然你还可以模拟投飞了的情况，即，最终的骰子数是 0 或者 1 ，即，< 2 ：

In [4]:

import random

r = random.randrange(0, 13) # 生成的随机数应该从 *0* 开始了；

if r == 7:

print('Draw!')

elif r >= 2 **and** r < 7: # 如果这里直接写 *elif r < 7:* ， 那么， *else:* 那一部分永远不会

被执行 ……

print('Small!')

elif r >= 2 **and** r > 7:

print('Big!')

else:

print('Not valid!')

Small!

for 循环

Python 语言中， for 循环不使用其它语言中那样的计数器，取而代之的是 range() 这个我称其

为“整数等差数列生成器”的函数。

用 C 语言写循环是这样的：

for(a = 0; a < 10; a = a + 1){

printf("value of a: %d**n**", a);

}

用 Python 写同样的东西，是这样的：

In [1]:

for a **in** range(10):

print(f'value of a: **{a}**') #每次 *a* 的值都不同，从 *0* 递增至 *9*

value of a: 0

value of a: 1

value of a: 2

value of a: 3

value of a: 4

value of a: 5

value of a: 6

value of a: 7

value of a: 8

value of a: 9

range() 函数

range() 是个内建函数，它的文档是这样写的：

range(*stop*)

range(*start*, *stop[*, *step]*)

只有一个参数的时候，这个参数被理解为 stop ，生成一个从 0 开始，到 stop - 1 的整数数列。

这就解释了为什么有的时候我们会在 for ... in range(...): 这种循环内的语句块里进行计算的时候，

经常会在变量之后写上 + 1 ，因为我们 range(n) 的返回数列中不包含 n ，但我们有时候却需要

n 。点击这里返回看看第一章里提到的例子：所谓算法那一小节。

In [4]:

from **IPython.core.interactiveshell** **import** InteractiveShell InteractiveShell.ast_node_interactivity = "all"

range(10)

list(range(10)) # 将 *range(10)* 转换成 *list*，以便清楚看到其内容。

Out[4]: range(0, 10)

Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

start 参数的默认值是 0 。如需指定起点，那么得给 range() 传递两个参数，比如， range(2,

13) ……

In [7]:

list(range(2, 13))

Out[7]: [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

第三个参数可选； step ，步长，就相当于是“等差数列”当中的“差”，默认值是 0 。例如， range(1,

10, 2) 生成的是这样一个数列 [1, 3, 5, 7, 9] 。所以，打印 0 ～ 10 之间的所有奇数，可以这样写：

In [8]:

for i **in** range(1, 10, 2):

print(i)

1

3

5

7

9

我们也可以生成负数的数列：

In [6]:

list(range(0, -10, -1))

Out[6]: [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]

Continue、**Break** 和 **Pass**

在循环的过程中，还可以用 continue 和 break 控制流程走向，通常是在

某条件判断发生的情况下

—— 正如你早就见过的那样：

for n **in** range(2, 100):

if n == 2:

print(n)

continue

for i **in** range(2, n):

if (n % i) == 0:

break

else:

print(n)

continue 语句将忽略其后的语句进入开始下次循环，而 break 语句将从此结束当前循环，开始执

行循环之后的语句：

for 语句块还可以附加一个 else —— 这是 Python 的一个比较有个性的地方。附加在 for 结尾的

else 语句块，在没有 *break* 发生的情况下会运行。

In [10]:

for n **in** range(2, 100):

if n == 2:

print(n)

continue

for i **in** range(2, n):

if (n % i) == 0:

break

else: # 下一行的 *print(n)* 事实上属于语句块 *for i in range(2, n):* print(n) # 整个循环结束，都没有发生 *break* 的情况下，才执行一次 *print(n)* 2

3

5

7

11

13

17

19

23

29

31

37

41

43

47

53

59

61

67

71

73

79

83

89

97

试比较以下两段代码：

In [ ]:

```
for n in range(2, 100):
if n == 2:
print(n)
continue
for i in range(2, n):
if (n % i) == 0:
break
print(n)  # 事实上相当于针对 range(2, 100) 中每个 n 都执行了一次 print(n)
# 这个 print(n) 属于语句块 for n in range(2, 100):
```

In [16]:

```
for n in range(2, 100):

if n == 2:

print(n)

continue

for i in range(2, n):

if (n % i) == 0:

break

print(n) # 事实上相当于针对 range(2, 100) 中每个 n 都执行了一次 print(n)
```

```
2

3

5

5

5

7

7

7

7

7

9
```

11

11

11

11

11

11

11

11

11

13

13

13

13

13

13

13

13

13

13

13

15

17

17

17

17

17

17

17

17

17

17

17

17

17

17

17

19

19

19

19

19

19

19

19

19

19

19

19

19

19

19

19

19

19

21

23

23

23

23

23

23

23

23

23

23

23

23

23

23

23

23

23

23

23

23

23

23

25

25

25

27

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

29

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

31

33

35

35

35

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

37

39

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

41

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

43

45

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

47

49

49

49

49

49

51

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

53

55

55

55

57

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

59

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

61

63

65

65

65

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

67

69

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

71

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

73

75

77

77

77

77

77

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

79

81

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

83

85

85

85

87

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

89

91

91

91

91

91

93

95

95

95

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

97

99

pass 语句什么都不干：

再比如，

def someFunction():

pass

又或者：

for i **in** range(100):

pass

if i % 2 == 0:

pass

换个角度去理解的话可能更清楚： pass 这个语句更多是给写程序的人用的。当你写程序的时候，

你可以用 pass 占位，而后先写别的部分，过后再回来补充本来应该写

在 pass 所在位置的那一段

代码。

写嵌套的判断语句或循环语句的时候，最常用 pass ，因为写嵌套挺费脑子的，一不小心就弄乱

了。所以，经常需要先用 pass 占位，而后逐一突破。

while 循环

今天，在绝大多数编程语言中，都提供两种循环结构：

Collection-controlled loops（以集合为基础的循环）

Condition-controlled loops（以条件为基础的循环）

之前的 for ... in ... 就是 Collection-controlled loops；而在 Python 中提供的 Condition-controlled loops 是 while 循环。

while 循环的格式如下：

while expression:

statements

输出 1000 以内的斐波那契数列的程序如下：

In [13]:

```
n = 1000

a, b = 0, 1

while a < n:

print(a, end=' ')

a, b = b, a+b
```

print()

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

for 和 while 的区别在哪里？什么时候应该用哪个？

for 更适合处理序列类型的数据（Sequence Type）的迭代，比如处理字符串中的每一个字符，比

如把 range() 返回的数列当作某种序列类型的索引。

while 更为灵活，因为它后面只需要接上一个逻辑表达式即可。

一个投骰子赌大小的游戏

虽然还不可能随心所欲写程序，但是，你现在具备了起码的“阅读能力”。有了以上大概的介绍，你

也许可以读懂一些代码了 —— 它们在你眼里再也不是天书了……

以下是一个让用户和程序玩掷骰子赌大小的程序。规则如下：

每次计算机随机生成一个 2... 12 之间的整数，用来模拟机器人投两个骰子

的情况；

机器人和用户的起始资金都是 10 个硬币

要求用户猜大小：

用户输入 b 代表“大”；

用户输入 s 代表“小”；

用户输入 q 代表“退出”；

用户的输入和随机产生的数字比较有以下几种情况：

随机数小于 7，用户猜小，用户赢；

随机数小于 7，用户猜大，用户输；

随机数等于 7，用户无论猜大还是猜小，结局平，不输不赢；

随机数大于 7，用户猜小，用户输；

随机数大于 7，用户猜大，用户赢；

游戏结束条件：

机器人和用户，若任意一方硬币数量为 0，则游戏结束；

用户输入了 q 主动终止游戏。

In [ ]:

```
from random import randrange

coinUser, coinBot = 10, 10 # 可以用一个赋值符号分别为多个变量赋值

roundOfGames = 0

def bet(dice, wager): # 接收两个参数，一个是骰子点数，另一个用户的输入

if dice == 7:

print(f'The dice is {dice}; \nDRAW! \n') # \n 是换行符号

return 0

elif dice < 7:

if wager == 's':

print(f'The dice is {dice}; \nYou WIN! \n')

return 1
```

else:

print(f'The dice is **{dice}**; **\n**You LOST! **\n**')

return -1

elif dice > 7:

if wager == 's':

print(f'The dice is **{dice}**; **\n**You LOST! **\n**')

return -1

else:

print(f'The dice is **{dice}**; **\n**You WIN! **\n**')

return 1

while True: # 除 *for* 之外的另外一个循环语句

print(f'You: **{coinUser}\t** Bot: **{coinBot}**')

dice = randrange(2, 13) # 生成一个 *2* 到 *12* 的随机数

wager = input("What's your bet? ")

if wager == 'q':

break

elif wager **in** 'bs': # 只有当用户输入的是 *b* 或者 *s* 得时候，才 “掷骰子”……

result = bet(dice, wager)

coinUser += result # *coinUser += result* 相当于 *coinUser = coinUser + result* coinBot -= result

```
roundOfGames += 1
if coinUser == 0:
print("Woops, you've LOST ALL, and game over!")
break
elif coinBot == 0:
print("Woops, the robot's LOST ALL, and game over!")
break
print(f"You've played {roundOfGames} rounds. \n")
print(f"You have {coinUser} coins now. \nBye!")
```

总结

有控制流，才能算得上是程序。

只处理一种情况，用 if ...

处理 True / False 两种情况，用 if ... else ...

处理多种情况，用 if ... elif ... elif ... else ...

迭代有序数据类型，用 for ... in ... ，如果需要处理没有 break 发生的情况，用 for ... else ...

其它循环，用 while ...

与循环相关的语句还有 continue 、 break 、 pass

函数从控制流角度去看其实就是子程序

函数

函数，实际上是可被调用的完整的程序。它具备输入、处理、输出的功能。又因为它经常在主程序

里被调用，所以它总是更像是个子程序。

了解一个函数，无非是要了解它的两个方面：

它的输入是怎么构成的（都有哪些参数？如何指定？）；

以及它的输出是什么（返回值究竟是什么？）……

从这个角度看，牛，对人类来说就是个函数，它吃的是草，挤出来的是奶…… 开玩笑了。

在我们使用函数的过程中，我们常常_有意忽略_它的内部如何完成从输出到输出之间的处理过程

——　这就好像我们平日里用灯泡一样，大多数情况下，我们只要知道开关的使用方法就够了 ——

至于为什么按到这个方向上灯会亮，为什么按到另外一个方向上灯会灭，并不是我们作为用户必须

关心的事情……

当然，如果你是设计开关的人就不一样了，你必须知道其中的运作原理；但是，最终，你还是希望

你的用户用最简单方便的操作界面，而不是必须搞懂所有原理才能够使用你所设计的产品……

当我们用 Python 编程的时候，更多的情况下，我们只不过是在使用别人已经写好的函数，或者用

更专业一点的词藻，叫做“已完好封装的函数”。而我们所需要做的事情（所谓的“学习使用函数”），

其实只不过是“通过阅读产品说明书了解如何使用产品”而已，真的没多神秘……

注意

这一章的核心目的，不是让你学会如何写函数；而是通过一些例子，让你大抵上学

会“如何阅读官方文档中关于函数的使用说明”。也请注意之前的那个词：“大抵上”，

所以千万别怕自己最初的时候理解不全面。

另外，这一章中用来举例的函数，全部来自于同一个官方文档页面，Built-in Functions：

https://docs.python.org/3/library/functions.html

示例 **print()**

基本的使用方法

print() 是初学者最常遇到的函数 —— 姑且不说是不是最常用到的。

它最基本的作用就是把传递给它的值输出到屏幕上，如果不给它任何参数，那么它就输出一个空

行：

In [3]:

```
print('line 1st')

print('line 2nd')
```

```
print()

print('line 4th')
```

```
line 1st
```

line 2nd

line 4th

你也可以向它传递多个参数，它就会把那些值逐个输出到屏幕，每个值之间用空格分开。

In [5]:

print('Hello,', 'jack', 'mike', '...', 'and all you guys!')

Hello, jack mike ... and all you guys!

当我们想把变量或者表达式的值插入字符串中的时候，可以用 f-string：

In [12]:

name = 'Ann'

age = '22'

print(f' **{name}** is **{age}** years old.')

Ann is 22 years old.

但，这并不是 print() 这个函数的功能，这实际上是 f-string 的功能， f-string 中用花

括号 {} 扩起来的部分是表达式，最终转换成字符串的时候，那些表达式的值（而不是变量或者表

达式本身）会被插入相应的位置……

In [11]:

name = 'Ann'

age = '22'

f' **{name}** is **{age}** years old.'

Out[11]: 'Ann is 22 years old.'

所以， print(f'{name} is {age} years old.') 这一句中，函数 print() 完成的还是

它最基本的功能：给它什么，它就把什么输出到屏幕上。

print() 的官方文档说明

以下，是 print() 这个函数的官方文档：

最必须读懂的部分，就是这一行：

print(*object, sep=' ', end='\n', file=sys.stdout,

flush=False) [1]

先只注意那些有着 = 的参数， sep=' ' 、 end='\n' 、 file=sys.stdout ，和

flush=False 。

这其中，先关注这三个 sep=' ' 、 end='\n' 、 file=sys.stdout ：

sep=' ' ：接收多个参数之后，输出时，分隔符号默认为空格， ' ' ；

end='\n' ：输出行的末尾默认是换行符号 '\n' ；

file=sys.stdout ： 默认的输出对象是 sys.stout （即，用户正在使用

的屏幕）……

也就是说，这个函数中有若干个具有默认值的参数，即便我们在调用这个函数的时候，就算没有指

定它们，它们也存在于此。

即，当我们调用 print('Hello', 'world!') 的时候，相当于我们调用的是

print('Hello', 'world!', sep=' ', end='\n', file=sys.stdout,

flush=False)

In [22]:

import sys # 如果没有这一行，代码会报错

print('Hello', 'world!') # 下一行的输出和这一行相同

print('Hello', 'world!', sep=' ', end=' **n**', file=sys.stdout, flush=**False**

)

print('Hello', 'world!', sep='-', end=' **t**')

print('Hello', 'world!', sep='~') # 上一行的末尾是 *t*，所以，这一行并

没有换行显示

print('Hello', 'world!', sep=' **n**') # 参数之间用换行 *n* 分隔

Hello world!

Hello world!

Hello-world! Hello~world!

Hello

world!

很多人只看各种教材、教程，却从来不去翻阅官方文档 —— 到最后非常吃亏。只不过是多花一点点

的功夫而已，看过之后，就会知道：原来 print() 这个函数是可以往文件里写数据的，只要指定

file 这个参数为一个已经打开的文件对象就可以了（真的有很多人完全

不知道）……

另外，现在可以说清楚了：

print() 这个函数的返回值是 None —— 注意，它向屏幕输出的内容，与

print() 这个函数的返回值不是一回事儿。

做为例子，看看 print(print(1)) 这个语句 —— print() 这个函数被调用了两次，第一次

是 print(1) ，它向屏幕输出了一次，完整的输出值实际上是 str(1) + '\n' ，而后返回一

个值， None ；而第二次调用 print()，这相当于是向屏幕输出这个 None ：

In [1]:

print(print(1))

1

None

“看说明书”就是这样，全都看了，真不一定全部看懂，但，看总是比不看强，因为总是有能看懂的

部分……

关键字参数

在 Python 中，函数的参数，有两种：

位置参数（Positional Arguments，在官方文档里常被缩写为 *arg*）

关键字参数（Keyward Arguments，在官方文档里常被缩写为 *karg*）

在函数定义中，带有　=　的，即，已为其设定了默认值的参数，叫做 Keyword Arguments，其它的

是 Positional Arguments。

在调用有 Keyword Arguments 的函数之时，如若不提供这些参数，那么参数在执行时，启用的是它

在定义的时候为那些 Keyword Arguments 所设定的默认值；如若提供了这些参数的值，那么参数在

执行的时候，启用的是接收到的相应值。

比如， sorted() 函数，它的定义如下：

sorted(iterable, *, key=None, reverse=False)

现在先只关注它的 Keyword Arugment， reverse ：

In [31]:

from **IPython.core.interactiveshell** **import** InteractiveShell InteractiveShell.ast_node_interactivity = "all"

sorted('abdc')

sorted('abdc', reverse=**True**)

Out[31]: ['a', 'b', 'c', 'd']

Out[31]: ['d', 'c', 'b', 'a']

位置参数

位置参数，顾名思义，是“由位置决定其值的参数”。拿 divmod() 为例，它的官方文档是这样写

的：

它接收且必须接收两个参数。

当你调用这个函数的时候，括号里写的第一个参数，是被除数，第二个参数是

除数 —— 此为该函数的输入；

而它的返回值，是一个元组（Tupple，至于这是什么东西，后面讲清楚），其

中包括两个值，第一个是商，第二个是余 —— 此为该函数的输出。

作为“这个函数的用户”，你不能（事实上也没必要）调换这两个参数的意义。因为，根据定义，被

传递的值的意义就是由参数的位置决定的。

In [2]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

divmod(11, 3)

a, b = divmod(11, 3)

a

b

divmod(3, 11)

a, b = divmod(3, 11)

a

b
```

Out[2]: (3, 2)

Out[2]: 3

Out[2]: 2

Out[2]: (0, 3)

Out[2]: 0

Out[2]: 3

可选位置参数

有些函数，如 pow() ，有可选的位置参数（Optional Positional Arguments）。

于是， pow() 有两种用法，各有不同的结果：

pow(x, y) —— 返回值是 x ** y

pow(x, y, z) —— 返回值是 x ** y % z

In [40]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

pow(2, 3)

pow(2, 3, 4)

Out[40]: 8

Out[40]: 0

注意 pow() 函数定义部分中，圆括号内的方括号 [, z] —— 这是非常严谨的标注，如果没有

z ， 那么那个逗号 , 就是没必要的。

看看 exec() 的官方文档（先别管这个函数干嘛用的），注意函数定义中的两个嵌套的方括号：

这些方括号的意思是说：

没在方括号里的 object 是不可或缺的函数，调用时必须提供；

可以有第二个参数，第二个参数会被接收为 globals ；

在有第二个参数的情况下，第三个参数会被接收为 locals ；

但是，你没办法在不指定 *globals* 这个位置参数的情况下指定

locals ……

可接收很多值的位置参数

再回头看看 print() ， 它的第一个位置参数， object 前面是有个星号的：*object,

... 。

对函数的用户来说，这说明，这个位置可以接收很多个参数（或者说，这个位置可以接收一个列表

或者元组）。

再仔细看看 print() ，它只有一个位置参数：

因为位置决定了值的定义，一般来说，一个函数里最多只有一个这种可以接收很多值的位置参数

—— 否则如何获知谁是谁呢？

如果与此同时，还有若干个位置参数，那么，能够接收很多值的位置参数只能放置最后，就好像

max() 函数那样：

Class 也是函数

虽然你现在还不一定知道 Class 究竟是什么，但，在阅读官方文档的时候，遇到一些内建函数前面

写着 Class，比如 Class bool([x]) ，千万别奇怪，因为 Class 本质上来看就是一种特殊类型

的函数，也就是说，它也是函数：

In [44]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

bool()

bool(3.1415926)

bool(-3.1415926)

bool(1 == 2)

bool(**None**)

Out[44]: False

Out[44]: True

Out[44]: True

Out[44]: False

Out[44]: False

总结

本章需要（大致）了解的重点如下，其实很简单：

你可以把函数当作一个产品，而你自己是这个产品的用户；

既然你是产品的用户，你要养成好习惯，一定要亲自阅读产品说明书；

调用函数的时候，注意_可选位置函数的使用方法_和关键字函数的默认值；

函数定义部分，注意两个符号就行了， [] 和 = ；

所有的函数都有返回值，即便它内部不指定返回值，也有一个返回值：None

；

另外，一定要耐心阅读该函数在使用的时候需要注意什么 —— 产品说明书的主

要作用就在这里……

知道这些就很好了！

这就好像你拿着一张地图，不可能一下子掌握其中所有的细节，但，花几分钟搞清楚“图

例”（Legend）部分总是可以的，知道什么样的线标示的是公交车，什么样的线标示的是地铁，什

么样的线标示的是桥梁，然后知道上北下南左西右东 —— 这之后，就可以开始慢慢研究地图了……

为了学会使用 Python，你以后最常访问的页面一定是这个：

https://docs.python.org/3/library/index.html

而最早反复阅读查询的页面肯定是其中的这两个：

https://docs.python.org/3/library/functions.html

https://docs.python.org/3/library/stdtypes.html

对了，还有就是，在这一章之后，你已经基本上“精通”了 print() 这个函数的用法。

脚注

[1]： print() 函数的官方文档里，sep='' 肯定是 sep=' ' 的笔误 —— 可以用以下代

码验证（2019.02.14）：

print('a', 'b', sep='')

print('a', 'b')

↑Back to Content↑

字符串

在任何一本编程书籍之中，关于字符串的内容总是很长 —— 就好像每本英语语法书中，关于动词的

内容总是占全部内容的至少三分之二。这也没什么办法，因为处理字符串是计算机程序中最普遍的

需求 —— 因为程序的主要功能就是完成人机交互，人们所用的就是字符串而不是二进制数字。

在计算机里，所有的东西最终都要被转换成数值。又由于计算机靠的是电路，所以，最终只能处理

1 和 0 ，于是，最基本的数值是二进制；于是，连整数、浮点数字，都要最终转换成二进制数

值。这就是为什么在所有编程语言中 1.1 + 2.2 并不是你所想象的 3.3 的原因。

In [1]:

1.1 + 2.2

Out[1]: 3.3000000000000003

因为最终所有的值都要转换成二进制 —— 这时候，小数的精度就有损耗，多次浮点数字转换成二进

制相互运算之后再从二进制转换为十进制之后返回的结果，精度损耗就更大了。因此，在计算机

上，浮点数字的精度总有极限。有兴趣进一步可以看看关于 decimal 模块的文档。

字符串也一样。一个字符串由 0 个字符或者多个字符构成，它最终也要被转换成数值，再进一步被

转换成二进制数值。空字符串的值是 None ，即便是这个 None —— 也最终还是要被转换成二进

制的 0 。

字符码表的转换

很久以前，计算机的中央处理器最多只能够处理 8 位二进制数值，所以，那时候的计算机只能处理

256 个字符，即，2

8 个字符。那个时候计算机所使用的码表叫 ASCII。现在计算机的中央处理器，

大多是 64 位的，所以可以使用 264 容量的码表，叫做 Unicode。随着多年的收集，2018 年 6 月 5

日公布的 11.0.0 版本已经包含了 13 万个字符 —— 突破 10 万字符是在 2005 年[1]。

把单个字符转换成码值的函数是 ord() ，它只接收单个字符，否则会报错；它返回该字母的

unicode 编码。与 ord() 相对的函数是 chr() ，它接收且只一个整数作为参数，而后返回相应

的字符。 ord() 接收多个字符的话会报错。

In [2]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

ord('a')

chr(122)

ord('鼙') # 完了，汉字也有你不认识的吧？

chr(25354) # 这个字估计你也不认识 ……

# ord('Python') # 这一句会报错
```

Out[2]: '抃'

字符串的标示

标示一个字符串，有 4 种方法，用单引号、用双引号，用三个单引号或者三个双引号：

In [3]:

```
'Simple is better than complex.' # 用单引号
```

Out[3]: 'Simple is better than complex.'

In [4]:

"Simple is better than complex." # 用双引号

Out[4]: 'Simple is better than complex.'

In [5]:

用三个单引号。注意输出结果中的 \n

这个字符串，看起来是两行，保存在内存或者变量之中的时候，

是一整串，其中的换行是用 \n 表示的。

'''

Simple is better than complex.

Complex is better than complicated.

'''

Out[5]: '\nSimple is better than complex.\nComplex is better than complicated.\n'

In [6]:

#用三个双引号。注意输出结果中的 \n

"""

Simple is better than complex.

Complex is better than complicated.

"""

Out[6]: '\nSimple is better than complex.\nComplex is better than complicated.\n'

In [7]:

print(

"""

Simple is better than complex.

Complex is better than complicated.

"""

) #用 *print()* 输出的时候， *\n* 就是不可见字符，字符串本身如下：

 # '\nSimple is better than complex.\nComplex is better than complicated.\n'

其中的 *\n* 被打印出来的时候显示成换行

Simple is better than complex.

Complex is better than complicated.

字符串与数值之间的转换

由数字构成的字符串，可以被转换成数值，转换整数用 int() ，转换浮点数字用 float() 。

与之相对，用 str() ，可以将数值转换成字符串类型。

注意，int() 在接收字符串为参数的时候，只能做整数转换。下面代码最后一行会报错：

In [8]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

int('3')

float('3')

str(3.1415926)

 # int('3.1415926') # 这一行会报错

Out[8]: 3

Out[8]: 3.0

Out[8]: '3.1415926'

input() 这个内建函数的功能是接收用户的键盘输入，而后将其作为字符串返回。它可以接收一

个字符串作为参数，在接收用户键盘输入之前，会把这个参数输出到屏幕，作为给用户的提示语。

这个参数是可选参数，直接写 input() ，即，没有提供参数，那么它在要求用户输入的时候，就

没有提示语。

以下代码会报错，因为 age < 18 不是合法的逻辑表达式，因为 age 是由 input() 传递过

来的字符串；于是，它不是数字，那么它不可以与数字比较……

In [9]:

age = input('Please tell me your age: ')

if age < 18:

print('I can not sell you drinks...')

else:

print('Have a nice drink!')

TypeError Traceback (most recent call last)

in

1 age = input('Please tell me your age: ')

----> 2 if age < 18:

3 print('I can not sell you drinks...')

4 else:

5 print('Have a nice drink!')

TypeError: ' <' not supported between instances of 'str' and 'int'

要改成这样才行：

In [9]:

age = int(input('Please tell me your age: '))

if age < 18:

print('I can not sell you drinks...')

else:

print('Have a nice drink!')

Have a nice drink!

注意：如果你用来浏览当前 .ipynb 文件的是那个桌面 App Nteract，它目前不支持 input() 这个

函数的调用……

转义符

有一个重要的字符，叫做“转义符”， \ ，也有的地方把它称为“脱字符”，因为它的英文原文是

Escaping Character。它本身不被当作字符，你要想在字符串里含有这个字符，得这样写 \\ ：

In [9]:

' \\'

Out[9]: '\\'

In [10]:

' \'

File " ", line 1

'\'

^

SyntaxError: EOL while scanning string literal

上面这一行报错信息是 SyntaxError: EOL while scanning string literal 。这是因

为 \' 表示的是单引号字符 '（Literal） —— 是可被输出到屏幕的 '，而不是用来标示字符串

的那个 ' —— 别急，无论哪个初学者第一次读到前面的句子都觉得有点莫名其妙…… —— 于

是，Python 编译器扫描这个“字符串”的时候，还没找到标示字符串末尾的另外一个 ' 的时候就读到

了 EOL （End Of Line）。

如果你想输出这么个字符串， He said, it's fine. ，如果用双引号扩起来 "

倒没啥问题，

但是如果用单引号扩起来就麻烦了，因为编译器会把 it 后面的那个单引号 ' 当作字符串结尾。

In [11]:

'He said, it's fine.'

File " ", line 1

'He said, it's fine.'

^

SyntaxError: invalid syntax

于是你就得用转义符 \ ：

In [12]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

要么你这么写：

'He said, it\' s fine.'

要么你这么写：

"He said, it's fine."

要么，不管用单引号还是双引号标示字符串，都习惯于用 \' 和 \" 书写属于字符串内部的引号 ……

"He said, it\' s fine."

Out[12]: "He said, it's fine."

Out[12]: "He said, it's fine."

Out[12]: "He said, it's fine."

转义符号 \ 的另外两个常用形式是和 t 、 n 连起来用， \t 代表制表符（就是用 TAB

键

敲出来的东西）， \n 代表换行符（就是用 Enter

敲出来的东西）。

由于历史原因，Linux/Mac/Windows 操作系统中，换行符号的使用各不相同。Unix 类操作系统（包

括现在的 MacOS），用的是 \n ； Windows 用的是 \r\n ， 早期苹果公司的 Macintosh 用的是

\r （参见 Wikipedia: Newline）。

所以，一个字符串，有两种形式，**raw** 和 **presentation**，在后者中， \t 被转换成制表符， \n

被转换成换行。

在写程序的过程中，我们在代码中写的是 *raw*，而例如当我们调用 print() 将字符串输出到屏幕

上时，是 *presentation*：

In [13]:

s = "He said, it's file." *# raw*

print(s) *# presentation*

He said, it's file.

以后有时间去看看这两个内建函数，能了解更多细节：

ascii(*object*) https://docs.python.org/3/library/functions.html#ascii

repr(*object*) https://docs.python.org/3/library/functions.html#repr

字符串的操作符

字符串可以用空格 ' ' 或者 + 拼接：

In [14]:

'Hey!' + ' ' + 'You!' # 使用操作符 +

Out[14]: 'Hey! You!'

In [15]:

'Hey!' 'You!' # 空格 与 + 的作用是相同的。

Out[15]: 'Hey!You!'

字符串还可以与整数被操作符 * 操作， 'Ha' * 3 的意思是说，把字符串 'Ha' 复制三遍：

In [16]:

'Ha' * 3

Out[16]: 'HaHaHa'

In [18]:

'3.14' * 3

Out[18]: '3.143.143.14'

字符串还可以用 in 和 not in 操作符 —— 看看某个字符或者字符串是否被包含在某个字符串

中，返回的是布尔值：

In [19]:

'o' **in** 'Hey, You!'

Out[19]: True

字符串的索引

字符串是由一系列的字符构成的。在 Python 当中，有一个容器（Container）的概念，这个概念后

面会深入讲解。现在需要知道的是，字符串是容器的一种；容器可分为两种，有序的和无序的 ——

字符串属于有序容器。

字符串里的每个字符，对应着一个从 0 开始的索引。比较有趣的是，索引可以是负数：

0

1

2

3

4

5

P

y

t

h

o

n

-6

-5

-4

-3

-2

-1

In [20]:

```
s = 'Python'

for char in s:

print(s.index(char), char)
```

0 P

1 y

2 t

3 h

4 o

5 n

对于有序容器中的元素 —— 字符串就是字符的有序容器 —— 由于它们是有索引的，所以我们可以

根据索引提取容器中的值，你可以把 [] 当作是有序容器的操作符之一，我们姑且将其成为“索引

操作符”。注意以下代码第 3 行中， s 后面的 [] ，以及里面的变量 i ：

In [21]:

```
s = 'Python'

for i in range(len(s)):

print(s[i])
```

P

y

t

h

o

n

我们可以使用_索引操作符_根据索引提取字符串这个_有序容器_中的一个或多个元素，即，其中的

字符或字符串。这个“提取”的动作有个专门的术语，叫做“Slicing”（切片）。索引操作符 [] 中可

以有一个、两个或者三个整数参数，如果有两个参数，需要用 ： 隔开。它最终可以写成以下 4 种

形式：

s[index] —— 返回索引值为 index 的那个字符

s[start:] —— 返回从索引值为 start 开始一直到字符串末尾的所有字

符

s[start:stop] —— 返回从索引值为 start 开始一直到索引值为 stop 的那个字符_之前_的所有字符

s[:stop] —— 返回从字符串开头一直到索引值为 stop 的那个字符_之前_的所有字符

s[start:stop:step] —— 返回从索引值为 start 开始一直到索引值为 stop 的那个字符_之前_的，以 step 为步长提取的所有字符

In [2]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

s = 'Python'

s[1]

s[2:]

s[2:5]

s[:5]

s[1:5:2]
```

Out[2]: 'y'

Out[2]: 'thon'

Out[2]: 'tho'

Out[2]: 'Pytho'

Out[2]: 'yh'

处理字符串的内建函数

Python 内建函数中，把字符串当做处理对象的

有： ord() 、 input() 、 int() 、 float() 、 len() 、 print() 。再次注意，ord()

只接收单个字符为参数。

In [23]:

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

ord(' **n**')

ord(' **t**')

ord(' **r**')

chr(65) # 与 *ord()* 相对的函数

s = input('请照抄一遍这个数字 3.14: ')

int('3')

int(s) 这一句会报错 …… 所以暂时注释掉了

float(s) * 9

len(s)

print(s*3)

Out[23]: 10

Out[23]: 9

Out[23]: 13

Out[23]: 'A'

Out[23]: 3

Out[23]: 28.26

Out[23]: 4

3.143.143.14

处理字符串的 **Method**

在 Python 中，字符串是一个对象 —— 更准确地讲，是 str 类（ Class str ）的对象。

第一部分尚未读完的你，暂时不用了解对象究竟是什么；只需要知道的是，一个对象的内部有很多

函数……　　这些写在对象内部的函数，有个专门的名称，类的方法（Method）。问题在于，在讲解

编程的内容里，“方法”这个词（比如，处理数值的_方法_是……）随处可见；所以，为了避免歧

义，以后的文字里，提到“类的方法”的时候，直接用 **Method** 这个英文单词……

字符串有很多可以调用 Methods。以下介绍的 str Methods，在官方文档 "Text Sequence Type"

中都可以找到。

调用 str 类的 Methods 的方法是使用 . 这个符号，比如：

pytnon

'Python'.upper()

大小写转换

转换字符串大小写的是 str.upper() 、 str.lower() 和 str.swapcase() ， 以及

str.casefold() ； 另外，还有专门针对行首字母大写的 str.capitalize() 和针对每个词

的首字母大写的 str.title() ：

In [24]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

'Now is better than never.'.upper()

在 *Python* 命令行工具之中，单个下划线，是个特殊变量；

保存着最近的语句或者表达式的结果

上一个 *Cell* 执行过后， _ 中保存着 *'NOW IS BETTER THAN NEVER.'*

_.lower()

Out[24]: 'NOW IS BETTER THAN NEVER.'

Out[24]: 'now is better than never.'

In [25]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

 # casefold() 也是转换成小写，但它能处理更多欧洲语言字符

'ß'.casefold() # 德语字符中，大写 *ß* 的小写形式是 *ss*

len('ß'.casefold())

'ß'.lower() *# lower()* 对这类字符无能为力 ……

len('ß'.lower())

 # casefold

' **u0132**' *# IJ* 这个字符的 *Unicode* 编码

' **u0132**'.casefold()

' **u0132**'.lower() # 对这个字符来说， *lower()* 和 *casefold* 的效果一样

len(' **u0132**'.casefold())

这是一篇有用的文章：

 # Truths programmers should know about case

 # https://www.b-list.org/weblog/2018/nov/26/case/

Out[25]: 'ss'

Out[25]: 2

Out[25]: 'ß'

Out[25]: 1

Out[25]: 'IJ'

Out[25]: 'ij'

Out[25]: 'ij'

Out[25]: 1

In [26]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

s = 'Now is better than never.'

s.capitalize() # 句首字母大写

s.title() # 每个单词首字母大写

Out[26]: 'Now is better than never.'

Out[26]: 'Now Is Better Than Never.'

In [27]:

s = 'Now is better than never.'

s.swapcase() # 逐个字符更替大小写

s.title()

s.title().swapcase()

Out[27]: 'nOW IS BETTER THAN NEVER.'

Out[27]: 'Now Is Better Than Never.'

Out[27]: 'nOW iS bETTER tHAN nEVER.'

另外，还有个 str.encode() 在处理非英文字符串（比如中文）的时候，经常会用到：

In [28]:

 # str.encode(encoding="utf-8", errors="strict")

 # 关于更多可能得 encoding list, 请参阅：

 # https://docs.python.org/3/library/codecs.html#standard-encodings

s = '简单优于复杂。'

s.encode()

Out[28]:

b'\xe7\xae\x80\xe5\x8d\x95\xe4\xbc\x98\xe4\xba\x8e\xe5\xa4\x8d\xe6\x9d\x8

搜索与替换

让我们从 str.count() 这个搜寻子字符串出现次数的 Method（即， str 这个 Class 中定义

的函数）开始。

它官方文档是这么写的：

str.count(sub[,start[,end]])

下面的函数说明加了默认值，以便初次阅读更容易理解：

str.count(sub [,start=0[, end=len(str)-1]])

这里的方括号 [] 表示该参数可选；方括号里再次嵌套了一个方括号，这个意思是说，在这个可选

参数 start 出现的情况下，还可以再有一个可选参数 end ；

而 = 表示该参数有个默认值。

只给定 sub 一个参数的话，于是从第一个字符开始搜索到结束；

如果，随后给定了一个可选参数的话，那么它是 start ，于是从 start 开

始，搜索到字符串结束；

如果 start 之后还有参数的话，那么它是 end ；于是从 start 开始，搜

索到 end 结束。

返回值为在字符串中 sub 出现的次数。

注意：字符串总第一个字符的索引值是 0 。

In [29]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

s = """Simple is better than complex.

Complex is better than complicated."""

s.lower().count('mp')

s.lower().count('mp', 10)

s.lower().count('mp', 10, 30)

Out[29]: 4

Out[29]: 3

Out[29]: 1

以下是 str 的搜索与替换的 Methods： str.find() , str.rfind() , str.index() 的示

例：

In [3]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

 # str.find(sub[, start[, end]])

print('Example of str.find():')

s = """Simple is better than complex.

Complex is better than complicated."""

s.lower().find('mpl')

s.lower().find('mpl', 10)

s.lower().find('mpl', 10, 20) # 没有找到就返回 *-1*， *-1* 的布尔值是 *False*
print()

print('Example of str.rfind():')

 # str.rfind(sub[, start[, end]])

 # brfind() 返回最后 *sub* 出现的那次的位置； *find()*是最早的那次

s.lower().rfind('mpl')

s.lower().rfind('mpl', 10)

s.lower().rfind('mpl', 10, 20) # 没有找到就返回 *-1*， *-1* 的布尔值是 *False*
print()

print('Example of str.index():')

 # str.index(sub[, start[, end]])

作用与 *find()* 相同，但，如果没找到的话，会触发 *ValueError* 异常

 # https://docs.python.org/3/library/exceptions.html#ValueError

s.lower().index('mpl')

 # str.rindex(sub[, start[, end]])

作用与 *rfind()* 相同，但，如果没找到的话，会触发 *ValueError* 异常

s.lower().rindex('mpl')

print()

Example of str.find():

Out[3]: 2

Out[3]: 24

Out[3]: -1

Example of str.rfind():

Out[3]: 56

Out[3]: 56

Out[3]: -1

Example of str.index():

Out[3]: 2

Out[3]: 56

str.startswith() 和 str.endswith() 是用来判断一个_字符串_是否以某个_子字符串_起

始或者结束的：

In [2]:

```
s = """Simple is better than complex.

Complex is better than complicated."""

# str.startswith(prefix[, start[, end]])

print("s.lower().starstwith('S'):", \

s.lower().startswith('S'))

print("s.lower().starstwith('b'):", \

s.lower().startswith('b', 10))

print("s.lower().startswith('e', 11, 20):", \
```

```
s.lower().startswith('e', 11, 20))

# str.endswith(suffix[, start[, end]])

print("s.lower().endswith('.'):", \

s.lower().endswith('.'))

print("s.lower().endswith('.', 10):", \

s.lower().endswith('.', 10))

print("s.lower().endswith('.', 10, 20):", \

s.lower().endswith('.', 10, 20))

# 好玩：中国人一不小心就会把这两个函数写成或者记成

# startwith() 和 endwith() —— 少写一个 s!

s.lower().starstwith('S'): False

s.lower().starstwith('b'): True

s.lower().startswith('e', 11, 20): True

s.lower().endswith('.'): True

s.lower().endswith('.', 10): True

s.lower().endswith('.', 10, 20): False
```

为了找到位置而进行搜索之前，你可能经常需要事先确认需要寻找的字符串在寻找对象中是否存

在，这个时候，可以用 in 操作符：

```
In [32]:

s = """Simple is better than complex.
```

Complex is better than complicated."""

如果你只想知道“有没有”，而无需知道“在哪里”，那么可以用：

print('mpl' **in** s)

True

能搜索，就应该能替换 —— str.replace()，它的函数说明是这样的：

str.replace(old, new[, count])

用 new 替换 old ，替换 count 个实例，其中， count 这个参数是可选的。

In [3]:

s = """Simple is better than complex.

Complex is better than complicated."""

 # str.startswith(prefix[, start[, end]])

print("s.lower().replace('mp', '[ ]', 2):**n**")

print(s.lower().replace('mp', '[ ]', 2))

s.lower().replace('mp', '[ ]', 2):

si[ ]le is better than co[ ]lex.

complex is better than complicated.

另外，还有个专门替换 TAB（ \t ）的 Method，

str.expandtabs(tabsize=8)

它的作用非常简单，就是把字符串中的 TAB（ \t ）替换成空格，默认是替换成 8 个空格 —— 当

然你也可以指定究竟替换成几个空格

In [5]:

from **IPython.core.interactiveshell** **import** InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

 # str.expandtabs(tabsize=8)

s = "Special**t**cases**t**aren't**t**special**t**enough**t**to**t**break**t**the**t**rules."

s.expandtabs()

s.expandtabs(2)

Out[5]: "Special cases aren't special enough to break the rules."

Out[5]: "Special cases aren't special enough to break the rules."

去除子字符

str.strip([chars])

它最常用的场景是去除一个字符串首尾的所有空白，包括空格、TAB、换行符等等。

In [35]:

from **IPython.core.interactiveshell** **import** InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

s = " **r** **t** Simple is better than complex. **t** **n**"

s

s.strip()

Out[35]: '\r \t Simple is better than complex. \t \n'

Out[35]: 'Simple is better than complex.'

但是，如果给定了一个字符串作为参数，那么参数字符串中的所有字母都会被当做需要从首尾剔除

的对象：

In [4]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

s = "Simple is better than complex."

s

s.strip('Six.p') # p 全部处理完之后， p 并不在首尾，所以原字符串中的 p 字母不受影响；

s.strip('pSix.mle') # 这一次，首尾的 p 被处理了 ….. 参数中的字符顺序对结果没有影响，换成 Sipx.ml e 也一样 ……
```

Out[4]: 'Simple is better than complex.'

Out[4]: 'mple is better than comple'

Out[4]: ' is better than co'

还可以只对左侧处理， str.lstrip() 或者只对右侧处理， str.rstrip()

In [37]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

# str.lstrip([chars])

s = "Simple is better than complex."
```

s

s.lstrip('Six.p') # *p* 全部处理完之后， *p* 并不在首尾，所以原字符串中的 *p* 字母不受影响；

s.lstrip('pSix.mle') # 这一次，首尾的 *p* 被处理了 ….. 参数中的字符顺序对结果没有影响，换乘 *Sipx.ml e* 也一样 ……

Out[37]: 'Simple is better than complex.'

Out[37]: 'mple is better than complex.'

Out[37]: ' is better than complex.'

In [38]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

 # str.rstrip([chars])

s = "Simple is better than complex."

s

s.rstrip('Six.p') # *p* 全部处理完之后， *p* 并不在首尾，所以原字符串中的 *p* 字母不受影响；

s.rstrip('pSix.mle') # 这一次，首尾的 *p* 被处理了 ….. 参数中的字符顺序对结果没有影响，换乘 *Sipx.m le* 也一样 ……

Out[38]: 'Simple is better than complex.'

Out[38]: 'Simple is better than comple'

Out[38]: 'Simple is better than co'

拆分字符串

在计算机里，数据一般保存在文件之中。计算机擅长处理的是“格式化数据”，即，这些数据按照一

定的格式排列 —— 电子表格、数据库，就是种保存方式。Microsoft 的 Excel 和 Apple 的

Numbers，都可以将表格导出为 .cvs 文件。这是文本文件，里面的每一行可能由多个数据构

成，数据之间用 ,（或 ; 、 \t ）分隔：

Name,Age,Location

John,18,New York

Mike,22,San Francisco

Janny,25,Miami

Sunny,21,Shanghai

文本文件中的这样一段内容，被读进来之后，保存在某个变量，那么，那个变量的值长成这个样

子：

'Name,Age,Location\nJohn,18,New York\nMike,22,San

Francisco\nJanny,25,Miami\nSunny,21,Shanghai'

我们可以对这样的字符串进行很多操作，最常用的比如， str.splitlines() ,

str.split() ； 还有个 str.partition() ，有空的人可以去官方文档看看说明。

str.splitlines() 返回的是个列表（List）—— 这又是一个后面的章节才能详细讲解的概念

—— 由被拆分的每一行作为其中的元素。

In [39]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

s = """Name,Age,Location

John,18,New York

Mike,22,San Francisco

Janny,25,Miami

Sunny,21,Shanghai"""

s # *s* 被打印出来的时候， *\n* 都被转换成换行了

s.splitlines() # 注意输出结果前后的方括号， *[]*，表示这个返回结果是一个 *List*

Out[39]: 'Name,Age,Location\nJohn,18,New York\nMike,22,San Francisco\nJanny,25,Miami\nSunny,21,Shanghai'

Out[39]: ['Name,Age,Location',

'John,18,New York',

'Mike,22,San Francisco',

'Janny,25,Miami',

'Sunny,21,Shanghai']

str.split() , 是将一个字符串，根据分隔符进行拆分：

str.split(sep=None, maxsplit=-1)

In [40]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

s = """Name,Age,Location

John,18,New York

Mike,22,San Francisco

Janny,25,Miami

Sunny,21,Shanghai"""

r = s.splitlines()[2] # 取出返回列表中索引值为 2 的那一行

r

r.split() # 如果没有给 str.strip() 传递参数，那么默认为用 None 分割（各种空白，比如， \t 和

\r 都被当作 None）

r.split(sep=',') # 如果没有给 str.strip() 传递参数，那么默认为用 None 分割（各种空白，比如， \t 和 \r 都被当作 None）

r.split(',') # 上一行可以这样写。

r.split(sep=',', maxsplit=1) # 第二个参数指定拆分几次

# r.split(sep=',', 1)) # 上一行不能这样写。

r.split(sep=',', maxsplit=0) # 0 次，即不拆分

r.split(sep=',', maxsplit=-1) # 默认值是 -1，拆分全部

Out[40]: 'Mike,22,San Francisco'

Out[40]: ['Mike,22,San', 'Francisco']
```

Out[40]: ['Mike', '22', 'San Francisco']

Out[40]: ['Mike', '22', 'San Francisco']

Out[40]: ['Mike', '22,San Francisco']

Out[40]: ['Mike,22,San Francisco']

Out[40]: ['Mike', '22', 'San Francisco']

拼接字符串

str.join() 是将来非常常用的，它的官方文档说明却很少：

str.join(_iterable_)

Return a string which is the concatenation of the strings in iterable. A

TypeError will be raised if there are any non-string values in iterable, including

bytes objects. The separator between elements is the string providing this method.

它接收的参数是 iterable，虽然你还没办法知道 iterable 究竟是什么，但这个 Method 的例子貌似可

以看懂（除了“那个方括号究竟是干什么的”之外）：

In [41]:

s = ''

t = ['P', 'y', 't', 'h', 'o', 'n']

s.join(t)

Out[41]: 'Python'

字符串排版

将字符串居中放置 —— 前提是设定整行的长度：

str.center(width[, fillchar])

注意，第 2 个参数可选，且只接收单个字符 —— char 是 *character* 的缩写。

In [42]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

s = 'Sparse is better than dense!'

s.title().center(60)

s.title().center(60, '=')

s.title().center(10) # 如果宽度参数小于字符串长度，则返回原字符串

s = 'Sparse is better than dense!'

s.title().rjust(60)

s.title().rjust(60, '.')

Out[42]: ' Sparse Is Better Than Dense! '

Out[42]: '==========Sparse Is Better Than Dense!=========='

Out[42]: 'Sparse Is Better Than Dense!'

Out[42]: ' Sparse Is Better Than Dense!'

Out[42]: '.....Sparse Is Better Than Dense!'

将字符串靠左或者靠右对齐放置：

str.ljust(width)

str.rjust(width)

另外，还有个字符串 Method 是，将字符串转换成左侧由 0 填充的指定长度字符串。例如，这在

批量生成文件名的时候就很有用……

In [43]:

```
for i in range(1, 11):

filename = str(i).zfill(3) + '.mp3'

print(filename)
```

001.mp3

002.mp3

003.mp3

004.mp3

005.mp3

006.mp3

007.mp3

008.mp3

009.mp3

010.mp3

格式化字符串

所谓对字符串进行格式化，指的是将特定变量插入字符串特定位置的过程。常用的 Methods 有两

个，一个是 str.format() ，另外一个是 f-string 。

使用 **str.format()**

这个 Method 的官方文档说明，你现在是死活看不懂的：

str.format(*args, **kwargs)

参数前面多了个 * …… 没办法，现在讲不清楚，讲了也听不明白…… 先跳过，以下只关注怎么用

这个 Method。

它的作用是：

在一个字符串中，插入一个或者多个占位符 —— 用大括号 {} 括起来；

而后将 str.format() 相应的参数，依次插入占位符中；

占位符中可以使用由零开始的索引。

In [44]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

name = 'John'

age = 25

' **{}** is **{}** years old.'.format(name, age)

 # 不写占位符索引就默认每个占位符的索引从第一个开始是 0, 1, 2 ... (占位符数量 - 1)

 # '{} {}'.format(a, b) 和 *'{0} {1}'.format(a, b)* 是一样的。

 # '{0} is {2} years old.'.format(name, age)

这一句会报错，因为 *2* 超出实际参数索引极限

两个连续使用的大括号，不被认为是占位符；且只打印出一对大括号

"Are you **{0}**? :-{{+}}".format(name)

 # "%s is %d years old." % (name, age)

上一行这是兼容 *Python 2* 的老式写法，可以从此忽略 ……

 # str.fomrat() 里可以直接写表达式 ……

' **{}** is a grown up? **{}**'.format(name, age >= 18)

Out[44]: 'John is 25 years old.'

Out[44]: 'Are you John? :-{+}'

Out[44]: 'John is a grown up? True'

使用 **f-string**

f-string 与 str.format() 的功用差不多，只是写法简洁一些 —— 在字符串标示之前加上一个字

母 f：

In [45]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

 # https://docs.python.org/3/library/stdtypes.html#printf-style-bytes-formatting

 # f-string

name = 'John'

age = 25

f' **{name}** is **{age}** years old.'

f' **{name}** is a grown up? {age >= 18}'

Out[45]: 'John is 25 years old.'

Out[45]: 'John is a grown up? True'

只不过，str.format() 的用法中，索引顺序可以任意指定，于是相对更为灵活，下面的例子只是为了

演示参数位置可以任意指定：

In [46]:

name = 'John'

age = 25

' **{1}** is **{0}** years old.'.format(name, age)

Out[46]: '25 is John years old.'

字符串属性

字符串还有一系列的 Methods，返回的是布尔值，用来判断字符串的构成属性：

In [7]:

 # str.isalnum()

print("'1234567890'.isalnum():", \

'1234567890'.isalnum()) *# '3.14'.isalnum()* 返回的是 *False*

 # str.isalpha()

print("'abcdefghij'.isalpha():", \

'abcdefghij'.isalpha())

 # str.isascii()

print("'山巅一寺一壶酒'.isascii():", \

'山巅一寺一壶酒'.isascii())

 # str.isdecimal()

print("'0.123456789'.isdecimal():", \

'0.1234567890'.isdecimal())

 # str.isdigit()

print("'0.123456789'.isdigit():", \

'0.1234567890'.isdigit()) *# 注意，如果字符串是 identifier，返回值也是 True*

 # str.isnumeric()

print("'0.123456789'.isnumeric():", \

'0.1234567890'.isnumeric())

 # str.islower()

print("'Continue'.islower():", \

'Continue'.islower())

 # str.isupper()

print("'Simple Is Better Than Complex'.istitle():", \

```
'Simple Is Better Than Complex'.isupper())
# str.istitle()
print("'Simple Is Better Than Complex'.istitle():", \
'Simple Is Better Than Complex'.istitle())
# str.isprintable()
print("' \t'.isnumeric():", \
' \t'.isprintable())
# str.isspace()
print("' \t'.isnumeric():", \
' \t'.isspace())
# str.isidentifier()
print("'for'.isidentifier():", \
'for'.isidentifier())
'1234567890'.isalnum(): True
'abcdefghij'.isalpha(): True
'山巅一寺一壶酒'.isascii(): False
'0.123456789'.isdecimal(): False
'0.123456789'.isdigit(): False
'0.123456789'.isnumeric(): False
'Continue'.islower(): False
```

'Simple Is Better Than Complex'.istitle(): False

'Simple Is Better Than Complex'.istitle(): True

'

'.isnumeric(): False

'

'.isnumeric(): True

'for'.isidentifier(): True

总结

这一章节显得相当繁杂。然而，这一章和下一章（关于容器），都是“用来锻炼自己耐心的好材

料”……

不过，若是自己动手整理成一个表格，总结归纳一下这一章节的内容，你就会发现其实没多繁杂，

总之就还是那点事儿，怎么处理字符串？用操作符、用内建函数，用 Methods。只不过，字符串的

操作符和数值的操作符不一样 —— 类型不一样，操作符就当然不一样了么！—— 最不一样的地方

是，字符串是有序容器的一种，所以，它有索引，所以可以根据索引提取…… 至于剩下的么，就是

很常规的了，用函数处理，用 Methods 处理，只不过，Methods 相对多了一点而已。

整理成表格之后，就会发现想要全部记住其实并没多难……

为了表格在一屏中可以正确显示，本来应该规规矩矩写 str.xxx ，但，写成

了 s.xxx ……

另外，操作那一行，为了分类记忆方便，把 len() 和 s.join() 也放进去

了……

“记住”的方法并不是马上就只盯着表格看…… 正确方法是反复阅读这一章内容中的代码，并逐一运

行，查看输出结果；还要顺手改改看看，多多体会。多次之后，再看着表格回忆知识点，直到牢记

位置。

为什么数值没有像字符串值这样详细论述？

上一章中，我们概括地讲了各种类型的值的运算。而后并没有继续深入讲解数字的运算，而是直

接“跳”到了这一章关于字符串的内容。其实，只要一张表格和一个列表就足够了（因为之前零零散

散都讲过）：

Python 针对数字常用的操作符和内建函数，按照优先级从低到高排列：

名称

操作

结果 官方文档链接

加

1 + 2

3

减

2 - 1

1

乘

3 * 5

15

除

6 / 2

3.0

商

7 // 3

2

余

7 % 3

1

负

-6

-6

正

+6

6

绝对值

abs(-1)

1

abs()

转换为整数

int(3.14)

3

int()

转换为浮点数

float(3)

3.0

float()

商余

divmod(7, 3)

2, 1

divmod()

幂

pow(2, 10)

1024

pow()

幂

3 ** 2

9

Python 用来处理数值的内建函数：

abs(n) 函数返回参数 n 的绝对值；

int(n) 用来将浮点数字 n 换成整数；

float(n) 用来将整数 n 转换成浮点数字；

divmod(n, m) 用来计算 n 除以 m ， 返回两个整数，一个是商，另外一个

是余；

pow(n, m) 用来做乘方运算，返回 n 的 m 次方；

round(n) 返回离浮点数字 n 最近的那个整数。

Python 做更为复杂的数学计算的模块（Module）是 math module，参阅：

https://docs.python.org/3/library/math.html

脚注

[1]：请查阅 https://en.wikipedia.org/wiki/Unicode

↑Back to Content↑

数据容器

在 Python 中，有个数据容器（Container） 的概念。

其中包括字符串、由 range() 函数生成的等差数列、列表（List）、元组（Tuple）、集

合（Set）、字典（Dictionary）。

这些容器，各有各的用处。其中又分为_可变_容器（Mutable） 和_不可变_容器（Immutable）。

可变的有列表、集合、字典；不可变的有字符串、 range() 生成的等差数列元组。集合，又分为

Set 和 *Frozen Set*；其中，Set 是可变的，Frozen Set 是不可变的。

字符串、由 range() 函数生成的等差数列、列表、元组是有序类型（Sequence Type），而集合与

字典是_无序_的。

另外，集合没有_重合_元素。

迭代（**Iterate**）

数据容器里的元素是可以被迭代的（Iterable），它们其中包含的元素，可以被逐个访问，以便被处

理。

对于数据容器，有一个操作符， in ，用来判断某个元素是否属于某个容器。

由于数据容器的可迭代性，再加上这个操作符 in ，在 Python 语言里写循环格外容易且方便（以字

符串这个字符的容器作为例子）：

In [1]:

```
for c in 'Python':
print(c)
```

```
P
y
t
h
o
n
```

在 Python 出现之前，想要完成这样一个访问字符串中的每一个字符的循环，大抵上应该是这样的（比如 C 语言）：

```
# Written in C
char *string;
scanf("%s",string);
int i=strlen(string);
int k = 0;
while(k
printf("%c", string[k]);
k++;
}
```

在 Python 中，简单的 for 循环，只需要指定一个次数就可以了，因为有 range() 这个函数：

In [2]:

```
for i in range(10):

print(i)
```

0

1

2

3

4

5

6

7

8

9

即便是用比 C 更为“现代”一点的语言，也大抵上应该是这样的：

```
var i;

for (i = 0; i < 10; i++) {

console.log（i）

}
```

当然，有时候我们也需要比较复杂的计数器，不过，Python 也不只有

for 循环，还有 while 循

环，在必要的时候可以写复杂的计数器。

列表（**List**）

列表和字符串一样，是个有序类型（Sequence Type）的容器，其中包含着有索引编号的元素。

列表中的元素可以是不同类型。不过，在解决现实问题的时候，我们总是倾向于创建由同一个类型

的数据构成的列表。遇到由不同类型数据构成的列表，我们更可能做的是想办法把不同类型的数据

分门别类地拆分出来，整理清楚 —— 这种工作甚至有个专门的名称与之关联：数据清洗。

列表的生成

生成一个列表，有以下几种方法：

a_list = []

b_list = [1, 2, 3]

list(), **or** list(literable) # 这是 *Type Casting*

(expression **with** x) **for** x **in** iterable

In [3]:

a_list = []

a_list.append(1)

a_list.append(2)

print(a_list, f'has a length of {len(a_list)}.')

#range() 返回的不是 *list*，需要用 *list()* 转换，否则也没办法调用 *.append()*

```
b_list = list(range(1, 9))

b_list.append(11)

print(b_list, f'has a length of {len(b_list)}.')

c_list = [2**x for x in range(8)]

print(c_list, f'has a length of {len(c_list)}.')
```

[1, 2] has a length of 2.

[1, 2, 3, 4, 5, 6, 7, 8, 11] has a length of 9.

[1, 2, 4, 8, 16, 32, 64, 128] has a length of 8.

这最后一种方法颇为神奇：

```
[2**x for x in range(8)]
```

这种做法，叫做 **List Comprehension**。

Comprehend 这个词的意思除了“理解”之外，还有另外一个意思，就是“包括、囊括” —— 这样的

话，你就大概能理解这种做法为什么被称作 *List Comprehension* 了。中文翻译中，怎么翻译的都

有，“列表生成器”、“列表生成式”等等，都挺好。但是，被翻译成“列表解析器”，就不太好了，给人

的感觉是操作反了……

List comprehension 可以嵌套使用 for ，甚至可以加上条件 if 。官方文档里有个例子，是用来把

两个元素并不完全相同的两个列表去同后拼成一个列表（下面稍作了改写）：

In [4]:

```
import random

n = 10

# 生成一个 n 个元素的序列，每个元素是 1~100 之间的随机数

a_list = [random.randrange(1, 100) for i in range(n)]

print(f'a_list comprehands {len(a_list)} random numbers: {a_list}')

# 从 a_list 里把偶数都挑出来

b_list = [x for x in a_list if x % 2 == 0]

print(f'... and it has {len(b_list)} even numbers: {b_list}')
```

a_list comprehands 10 random numbers: [98, 93, 84, 66, 58, 66, 9, 75, 11, 21]

... and it has 5 even numbers: [98, 84, 66, 58, 66]

列表的操作符

列表的操作符和字符串一样，因为它们都是有序容器。列表的操作符有：

拼接： + （与字符串不一样的地方是，不能用空格了 ' ' ；

复制： *

逻辑运算： in 和 not in ， < 、 <= 、 > 、 >= 、 != 、 ==

而后两个列表也和两个字符串一样，可以被比较，即，可以进行逻辑运算；比较方法也跟字符串一

样，从两个列表各自的第一个元素开始逐个比较，“一旦决出胜负马上停止”：

In [5]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

a_list = [1, 2, 3]

b_list = [4, 5, 6]

c_list = a_list + b_list * 3

c_list

7 not in c_list

a_list > b_list
```

Out[5]: False

根据索引提取列表元素

列表当然也可以根据索引操作，但，由于列表是可变序列，所以，不仅可以提取，还可以删除，甚

至替换。

In [1]:

```
import random

n = 3

a_list = [random.randrange(65, 91) for i in range(n)]

b_list = [chr(random.randrange(65, 91)) for i in range(n)]
```

print(a_list)

c_list = a_list + b_list + a_list * 2

print(c_list)

print()

根据索引提取（ *Slicing*）

print(c_list[3]) # 返回索引值为 *3* 的元素值

print(c_list[:]) # 相当于 *a_list*，返回整个列表

print(c_list[5:]) # 从索引为 *5* 的值开始直到末尾

print(c_list[:3]) # 从索引 *0* 开始，直到索引 *3* 之前（不包括 *3*）

print(c_list[2:6]) # 从索引 *2* 开始，直到索引 *6* 之前（不包括 *6*）

print()

根据索引删除

del c_list[3]

print(c_list) *# del* 是个命令， *del a_list[3]* 是一个语句；不能这么写：*print(del a _list[3])*

del c_list[5:8]

print(c_list)

print()

根据索引替换

c_list[1:5:2] = ['a', 2] *# s[start:stop:step] = t*，跟 *range* 的三个参数类似；

 # len(t) = len([start:stop:step]) 必须为真

print(c_list)

[77, 80, 86]

[77, 80, 86, 'E', 'U', 'J', 77, 80, 86, 77, 80, 86]

E

[77, 80, 86, 'E', 'U', 'J', 77, 80, 86, 77, 80, 86]

['J', 77, 80, 86, 77, 80, 86]

[77, 80, 86]

[86, 'E', 'U', 'J']

[77, 80, 86, 'U', 'J', 77, 80, 86, 77, 80, 86]

[77, 80, 86, 'U', 'J', 77, 80, 86]

[77, 'a', 86, 2, 'J', 77, 80, 86]

需要注意的地方是：列表（List）是可变序列，而字符串（str）是不可变序列，所以，对字符串来

说，虽然也可以根据索引提取，但没办法根据索引删除或者替换。

In [7]:

s = 'Python'[2:5]

print(s)

del(s[3]) # 这一句会报错

tho

TypeError Traceback (most recent call last)

in

1 s = 'Python'[2:5]

2 print(s)

----> 3 del(s[3]) # 这一句会报错

TypeError: 'str' object doesn't support item deletion

之前提到过：

字符串常量（String　Literal）是不可变有序容器，所以，虽然字符串也有一些

Methods 可用，但，那些 Methods 都不改变它们自身，而是在操作后返回一个值给

另外一个变量。

而对于列表这种可变容器，我们可以对它进行操作，结果是它本身被改变了。

In [8]:

s = 'Python'

L = list(s)

print(s)

print(L)

del(L[2])

print(L) # 用 *del()* 对 *L* 操作之后， *L* 本身少了 *1* 个元素

Python

['P', 'y', 't', 'h', 'o', 'n']

['P', 'y', 'h', 'o', 'n']

列表可用的内建函数

列表和字符串都是容器，它们可使用的内建函数也其实都是一样的：

len()

max()

min()

In [9]:

```
import random

n = 3

# 生成 3 个随机数，构成一个列表 ''

a_list = [random.randrange(65, 91) for i in range(n)]

b_list = [chr(random.randrange(65, 91)) for i in range(n)]

print(a_list)

print(b_list)

# 列表可以使用操作符 + 和 *

c_list = a_list + b_list + a_list * 2

print(c_list)

a_list *= 3

print(a_list)
```

内建函数操作 *len()*、 *max*（）、 *min*

print(len(c_list))

print(max(b_list)) # 内建函数内部做了异常处理，可以比较字符和数字 —— 初学者最讨厌这

种事情了 ……

print(min(b_list))

print('X' **not in** b_list)

[66, 70, 72]

['Q', 'W', 'G']

[66, 70, 72, 'Q', 'W', 'G', 66, 70, 72, 66, 70, 72]

[66, 70, 72, 66, 70, 72, 66, 70, 72]

12

W

G

True

Methods

字符串常量和 range() 都是不可变的（Imutable）；而列表则是可变类型（Mutable type），所以，

它最起码可以被排序 —— 使用 sort() Method：

In [10]:

import random

n = 10

a_list = [random.randrange(1, 100) **for** i **in** range(n)]

print(f'a_list comprehands {len(a_list)} random numbers:**n**', a_list)

a_list.sort()

print('the list sorted:**n**', a_list)

a_list.sort(reverse=**True**) *#reverse* 参数，默认是 *False*

print('the list sorted reversely:**n**', a_list)

a_list comprehands 10 random numbers:

[78, 49, 36, 68, 99, 99, 47, 56, 73, 21]

the list sorted:

[21, 36, 47, 49, 56, 68, 73, 78, 99, 99]

the list sorted reversely:

[99, 99, 78, 73, 68, 56, 49, 47, 36, 21]

如果列表中的元素全都是由字符串构成的，当然也可以排序：

In [11]:

import random

n = 10

a_list = [chr(random.randrange(65, 91)) **for** i **in** range(n)]

 # chr() 函数会返回指定 *ascii* 码的字符， *ord('A')* 是 *65*

print(f'a_list comprehands {len(a_list)} random string elements:**n**', a_l ist)

```
a_list.sort()
print('the list sorted:\n', a_list)
a_list.sort(reverse=True) #reverse 参数，默认是 False
print('the list sorted reversely:\n', a_list)
print()
b_list = [chr(random.randrange(65, 91)) +\
chr(random.randrange(97, 123))\
for i in range(n)]
# 可以在行末加上 \ 符号，表示 “该行未完待续 ……”
print(f'b_list comprehands {len(b_list)} random string elements:\n', b_l ist)
b_list.sort()
print('the sorted:\n', b_list)
b_list.sort(key=str.lower, reverse=True)
# key 参数，默认是 None
#   key=str.lower   的意思是，在比较的时候，先全都转换成小写再比较 ……
# —— 但并不改变原有值
print('the sorted reversely:\n', b_list)
a_list comprehands 10 random string elements:
['O', 'W', 'Z', 'I', 'R', 'H', 'G', 'L', 'W', 'L']
the list sorted:
```

['G', 'H', 'I', 'L', 'L', 'O', 'R', 'W', 'W', 'Z']

the list sorted reversely:

['Z', 'W', 'W', 'R', 'O', 'L', 'L', 'I', 'H', 'G']

b_list comprehands 10 random string elements:

['Ax', 'Uh', 'Gg', 'Co', 'Zh', 'Wi', 'Di', 'Is', 'Hu', 'Br']

the sorted:

['Ax', 'Br', 'Co', 'Di', 'Gg', 'Hu', 'Is', 'Uh', 'Wi', 'Zh']

the sorted reversely:

['Zh', 'Wi', 'Uh', 'Is', 'Hu', 'Gg', 'Di', 'Co', 'Br', 'Ax']

注意：不能乱比较…… 被比较的元素应该是同一类型 —— 所以，不是由同一种数据类型元素构成

的列表，不能使用 sort() Method。下面的代码会报错：

In [12]:

a_list = [1, 'a', 'c']

a_list = a_list.sort() # 这一句会报错

TypeError Traceback (most recent call last)

in

1 a_list = [1, 'a', 'c']

----> 2 a_list = a_list.sort() # 这一句会报错

TypeError: ' <' not supported between instances of 'str' and 'int'

可变序列还有一系列可用的

Methods： a.append()， a.clear()， a.copy()， a.extend(t)， a.insert(i，x)， a.pop([i])，

a.remove(x)， a.reverse() ……

In [13]:

```
import random

n = 3

a_list = [random.randrange(65, 91) for i in range(n)]

b_list = [chr(random.randrange(65, 91)) for i in range(n)]

print(a_list)

c_list = a_list + b_list + a_list * 2

print(c_list)

# 在末尾追加一个元素

c_list.append('100')

print(c_list)

# 清空序列

print()

print(a_list)

a_list.clear()

print(a_list)

print()
```

```
# 拷贝一个列表
d_list = c_list.copy()
print(d_list)
del(d_list[6:8])
print(d_list)
print(c_list) # 对一个拷贝操作，不会更改“原件”
print()
# 演示拷贝 .copy() 与赋值 = 的不同
e_list = d_list
del(e_list[6:8])
print(e_list)
print(d_list) # 对 e_list 操作，相等于对 d_list 操作
# 在末尾追加一个列表
print()
print(a_list)
a_list.extend(c_list) # 相当于 a_list =+ c_list
print(a_list)
# 在某索引位置插入一个元素
print()
print(a_list)
```

a_list.insert(1, 'example') # 在索引 *1* 的位置插入 *'example'*

a_list.insert(3, 'example') # 在索引 *3* 的位置插入 *'example'* ；

print(a_list)

排序

 # a_list.sort() 这一句会出错，因为当前列表中的元素，是 *int* 和 *str* 混合的。

print()

print(a_list)

a_list.reverse()

print(a_list)

x = a_list.reverse() *# reverse()* 只对当前序列操作，并不返回一个逆序列表；返回值是 *Non*

e

print(x)

[65, 66, 68]

[65, 66, 68, 'C', 'T', 'Y', 65, 66, 68, 65, 66, 68]

[65, 66, 68, 'C', 'T', 'Y', 65, 66, 68, 65, 66, 68, '100']

[65, 66, 68]

[]

[65, 66, 68, 'C', 'T', 'Y', 65, 66, 68, 65, 66, 68, '100']

[65, 66, 68, 'C', 'T', 'Y', 68, 65, 66, 68, '100']

[65, 66, 68, 'C', 'T', 'Y', 65, 66, 68, 65, 66, 68, '100']

[65, 66, 68, 'C', 'T', 'Y', 66, 68, '100']

[65, 66, 68, 'C', 'T', 'Y', 66, 68, '100']

[]

[65, 66, 68, 'C', 'T', 'Y', 65, 66, 68, 65, 66, 68, '100']

[65, 66, 68, 'C', 'T', 'Y', 65, 66, 68, 65, 66, 68, '100']

[65, 'example', 66, 'example', 68, 'C', 'T', 'Y', 65, 66, 68, 65, 66, 68, '100']

[65, 'example', 66, 'example', 68, 'C', 'T', 'Y', 65, 66, 68, 65, 66, 68, '100']

['100', 68, 66, 65, 68, 66, 65, 'Y', 'T', 'C', 68, 'example', 66, 'example', 65]

None

有一个函数、两个 Methods 与删除单个元素相关联， del()， a.pop[i]，a.remove(x)，请注意它

们之间的区别。

In [14]:

import random

n = 3

a_list = [random.randrange(65, 91) **for** i **in** range(n)]

print(a_list)

插入

print()

a_list.insert(1, 'example') # 在索引 *1* 的位置插入 *'example'*

删除

print()

print(a_list)

a_list.remove('example') # 去除 *'example'* 这个元素，如果有多个 *'example'* ，只删除

第一个

print(a_list)

 # pop() 删除并返回被删除的值

print()

print(a_list)

p = a_list.pop(2) # 去除索引为 *2* 的元素，且返回元素的值，赋值给 *p*

print(a_list)

print(p)

 # pop() 与 *del()*，或者 *remove()* 的区别

print()

a_list.insert(2, 'example')

a_list.insert(2, 'example')

print(a_list)

del(a_list[2])

print(a_list)

print()

print(a_list.remove('example')) *# a_list.remove()* 这个方法的返回值是 *None*

print(a_list)

[86, 69, 81]

[86, 'example', 69, 81]

[86, 69, 81]

[86, 69, 81]

[86, 69]

81

[86, 69, 'example', 'example']

[86, 69, 'example']

None

[86, 69]

小结

看起来是个新概念，例子全部读完也很是要花上一段时间，然而，从操作上来看，操作列表和操作

字符串的差异并不大，重点在于一个是 Immutable，另外一个是 Mutable，所以，例如像

a.sort() ， a.remove() 这样的事儿，列表能做，字符串不能做 —— 字符串也可以排序，但，那是排

序之后返回给另外一个变量；而列表可以直接改变自身……

而整理成表格之后呢，理解与记忆真的是零压力：

元组（**Tuple**）

在完整掌握列表的创建与操作之后，再理解元组（Tuple） 就容易了，因为它们之间的主要区别只

有两个：

List 是可变有序容器，Tuple 是不可变有序容器。

List 用方括号标识 [] ，Tuple 用圆括号 标识 () ；

创建一个元组的时候，用圆括号：

a = ()

这样就创建了一个空元组。

多个元素之间，用 , 分离。

创建一个含多个元素的元组，可以省略这个括号。

In [15]:

a = 1, 2, 3 # 不建议这种写法

b = (1, 2, 3) # 在创建元组的时候建议永远不省略圆括号 ……

print(a)

print(b)

a == b

(1, 2, 3)

(1, 2, 3)

Out[15]: True

注意：创建单个元素的元组，无论是否使用圆括号，在那唯一的元素后

面一定要补上一个逗号 , ：

In [16]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

a = 2, # 注意这个末尾的逗号 , 它使得 *a* 变量被定义为一个元组，而不是数字

a

b = 2 # 整数，赋值

b

c = (2) # 不是元组

c

type(c) # 还是 *int*

d = (2,) # 这才是元组

d

a == d

Out[16]: (2,)

Out[16]: 2

Out[16]: 2

Out[16]: int

Out[16]: (2,)

Out[16]: True

元组是不可变序列，所以，你没办法从里面删除元素。

但是，你可以在末尾追加元素。所以，严格意义上，对元组来讲，“不可变”的意思是说，“当前已有

部分不可变”……

In [17]:

```
a = 1,

print(a)

print(id(a))

a += 3, 5

print(a)

print(id(a)) # id 并不相同 —— 实际上是在内存中另外新创建了一个元组
…..
```

(1,)

4339120112

(1, 3, 5)

4338763312

初学者总是很好奇 List 和 Tuple 的区别。首先是使用场景，在将来需要更改的时候，创建 List ； 在

将来不需要更改的时候，创建 Tuple。其次，从计算机的角度来看，Tuple 相对于 List 占用更小的

内存。

In [18]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

n = 10000 *#@param {type:"number"}*

a = range(n)

b = tuple(a) # 把 *a* 转换成元组

c = list(a) # 把 *a* 转换成列表

a.__sizeof__()

b.__sizeof__()

c.__sizeof__()

Out[18]: 48

Out[18]: 80024

Out[18]: 90088

等你了解了 Tuple 的标注方法，你就会发现， range() 函数返回的等差数列就是一个 Tuple ——

range(6) 就相当于 (0, 1, 2, 3, 4, 5) 。

集合（**Set**）

集合（Set）这个容器类型与列表不同的地方在于，首先它不包含重合元素，其次它是无序的；进

而，集合又分为两种，Set，可变的，Frozen Set，不可变的。

创建一个集合，用花括号 {} 把元素括起来，用 , 把元素隔开：

In [19]:

primes = {2, 3, 5, 7, 11, 13, 17}

primes

Out[19]: {2, 3, 5, 7, 11, 13, 17}

创建

注意：创建空集合的时候，必须用 set()，而不能用 {}：

In [20]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

a = {} # 注意这样创建的是一个 *dict*（字典），而不是 *set* 集合

b = set() # 这样创建的才是空集合

type(a)

type(b)

Out[20]: dict

Out[20]: set

也可以将序列数据转换（Casting）　为集合。转换后，返回的是一个已去重的集合。

In [21]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

a = "abcabcdeabcdbcdef"

b = range(10)

c = [1, 2, 2, 3, 3, 1]

d = ('a', 'b', 'e', 'b', 'a')

set(a)

set(b)

set(c)

set(d)

Out[21]: {'a', 'b', 'c', 'd', 'e', 'f'}

Out[21]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Out[21]: {1, 2, 3}

Out[21]: {'a', 'b', 'e'}

Set 当然也可以进行 *Comprehension*：

In [22]:

a = "abcabcdeabcdbcdef"

b = {x **for** x **in** a **if** x **not in** 'abc'}

b

Out[22]: {'d', 'e', 'f'}

操作

将序列类型数据转换成 Set，就等于去重。当然，也可以用 in 来判断某个元素是否属于这个集

合。 copy()、 len()、 max()、 min()， 也都可以用来操作 Set，但 del() 却不行 —— 因为 Set 中

的元素没有索引（它不是有序容器）。从 Set 里删除元素，得用 set.remove(elem) 方法；而

Frozen Set 是不可变的，所以不能用 set.remove(elem) 方法操作。

对于集合，有相应的操作符对它们可以进行集合运算：

并集：|

交集：&

差集：-

对称差集：^

之前用 set('abcabcdeabcdbcdef') 作为简单例子还凑合能用；但，这样对读者无意义的集合，无助于

进一步的理解。

事实上，每种数据结构（Data Structures —— 在这一章里，我们一直用的概念是“容器”，其实是指

同一事物的两种称呼）都有自己的应用场景。比如，当我们需要管理很多用户时，集合就可以派上

很大用场。

假定两个集合中有些人是 admins，所有人都是 moderators：

admins = {'Moose', 'Joker', 'Joker'}

moderators = {'Ann', 'Chris', 'Jane', 'Moose', 'Zero'}

那么：

In [23]:

admins = {'Moose', 'Joker', 'Joker'}

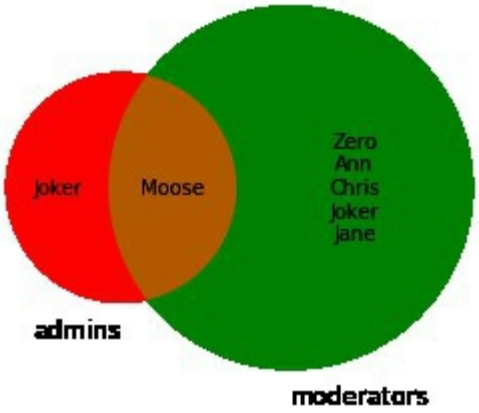
moderators = {'Ann', 'Chris', 'Jane', 'Moose', 'Zero'}

admins # 去重自动完成

'Joker' **in** admins # *Joker* 是否是 *admins*？

'Joker' **in** moderators # *Joker* 是否是 *moderator*？

admins | moderators # *admins*、 *moderator*，或者身兼两职的，即，两个角色中的所有人 *i*



n admins or moderators or both

admins & moderators # 既是 *admins* 又是 *moderator* 的都有谁？ *in both admins and moderat ors*

admins - moderators # 是 *admins* 但不是 *moderator* 的都有谁？ *in admins but not in moderat ors*

admins ^ moderators # *admins* 和 *moderator* 中不是身兼两职的都有谁？ *in admins or users but not both*

Out[23]: {'Joker', 'Moose'}

Out[23]: True

Out[23]: False

Out[23]: {'Ann', 'Chris', 'Jane', 'Joker', 'Moose', 'Zero'}

Out[23]: {'Moose'}

Out[23]: {'Joker'}

Out[23]: {'Ann', 'Chris', 'Jane', 'Joker', 'Zero'}

In [2]:

 # 这个 cell 集合运算图示需要安装 matplotlib 和 matplotlib-venn

 # !pip install matplotlib

 # !pip install matplotlib-venn

```
import matplotlib.pyplot as plt
from matplotlib_venn import venn2
admins = {'Moose', 'Joker', 'Joker'}
moderators = {'Ann', 'Chris', 'Jane', 'Moose', 'Zero'}
v = venn2(subsets = (admins, moderators), set_labels =('admins', 'modera
tors'))
v.get_label_by_id('11').set_text(' \n'.join(admins&moderators))
v.get_label_by_id('10').set_text(' \n'.join(admins-moderators))
v.get_label_by_id('01').set_text(' \n'.join(admins^moderators))
plt.show()
```

以上的操作符，都有另外一个版本，即，用 Set 这个类的方法完成。

意义

操作符

方法

方法相当于

并集

|

set.union(*others)

set | other | ...

交集

&

set.intersection(*others)

set & other & ...

差集

-

set.difference(*others)

set - other - ...

对称差集

^

set.symmetric_difference(other)

set ^ other

注意，并集、交集、差集的方法，可以接收多个集合作为参数 (*other)，但对称差集方法只接收一

个参数 (other) 。

对于集合，推荐更多使用方法而不是操作符的主要原因是：更易读
—— 对人来说，因为有意义、有

用处的代码终将需要人去维护。

In [25]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

admins = {'Moose', 'Joker', 'Joker'}

moderators = {'Chris', 'Moose', 'Jane', 'Zero'}

admins.union(moderators)

admins.intersection(moderators)

admins.difference(moderators)

admins.symmetric_difference(moderators)
```

Out[25]: {'Chris', 'Jane', 'Joker', 'Moose', 'Zero'}

Out[25]: {'Moose'}

Out[25]: {'Joker'}

Out[25]: {'Chris', 'Jane', 'Joker', 'Zero'}

逻辑运算

两个集合之间可以进行逻辑比较，返回布尔值。

set == other

True : set 与 other 相同

set != **other**

True : set 与 other 不同

isdisjoint(*other*)

True : set 与 other 非重合；即， set & other == None

issubset(*other*)，**set** <= **other**

True : set 是 other 的子集

set < **other**

True : set 是 other 的真子集，相当于 set <= other && set != other

issuperset(*other*)，**set** >= **other**

True : set 是 other 的超集

set > **other**

True : set 是 other 的真超集，相当于 set >= other && set != other

更新

对于集合，有以下更新它自身的方法：

add(*elem*)

把 elem 加入集合

remove(*elem*)

从集合中删除 elem；如果集合中不包含该 elem，会产生 KeyError 错误。

discard(*elem*)

如果该元素存在于集合中，删除它。

pop(*elem*)

从集合中删除 elem，并返回 elem 的值，针对空集合做此操作会产生 KeyEroor 错

误。

clear() 从集合中删除所有元素。

set.update(* *others*) ，相当于 set |= other | ...

更新 set, 加入 others 中的所有元素；

set.intersection_update(* *others*)，相当于 set &= other & ...

更新 set, 保留同时存在于 set 和所有 others 之中的元素；

set.difference_update(* *others*)，相当于 set -= other | ...

更新 set, 删除所有在 others 中存在的元素；

set.symmetric_difference_update(*other*)，相当于 set ^= other

更新 set, 只保留存在于 set 或 other 中的元素，但不保留同时存在于 set 和 other 中

的元素；注意，该方法只接收一个参数。

冻结集合

还有一种集合，叫做冻结集合（Frozen Set），Frozen Set 之于 Set，正如 Tuple 之于 List，前者

是不可变容器（Immutable），后者是可变容器（Mutable），无非是为了节省内存使用而设计的类

别。

有空去看看这个链接就可以了：

https://docs.python.org/3/library/stdtypes.html#frozenset

字典（**Dictionary**）

Map 是容器中的单独一类，映射（Map）容器。映射容器只有一种，叫做字典（Dictionary）。先

看一个例子：

In [26]:

phonebook = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225}

phonebook

Out[26]: {'ann': 6575, 'bob': 8982, 'joe': 2598, 'zoe': 1225}

字典里的每个元素，由两部分组成， *key*（键）和 *value*（值），二者由一个冒号连接。

比如， 'ann':6575 这个字典元素， *key* 是 'ann' ， *value* 是 6575 。

字典直接使用 *key* 作为索引，并映射到与它匹配的 *value*：

In [27]:

phonebook = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225}

phonebook['bob']

Out[27]: 8982

在同一个字典里， *key* 都是唯一的。当创建字典的时候，如果其中有重复的 *key* 的话，就跟 Set 那

样会“自动去重” —— 保留的是众多重复的 *key* 中的最后一个 *key:value*（或者说，最后一个

key:value “之前那个 *key* 的 *value* 被更新了”。字典这个数据类型之所以

叫做 Map（映射），是因为

字典里的 *key* 都映射且只映射一个对应的 *value*。

In [28]:

phonebook = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225, 'ann':6585}

phonebook

Out[28]: {'ann': 6585, 'bob': 8982, 'joe': 2598, 'zoe': 1225}

在已经了解如何操作列表之后，再去理解字典的操作，其实没什么难度，无非就是字典多了几个

Methods。

提蓄一下自己的耐心，把下面的若干行代码都仔细阅读一下，猜一猜输出结果都是什么？

字典的生成

In [29]:

from IPython.core.interactiveshell import InteractiveShell

InteractiveShell.ast_node_interactivity = "all"

aDict = {}

bDict = {'a':1, 'b':2, 'c':3}

aDict

bDict

Out[29]: {}

Out[29]: {'a': 1, 'b': 2, 'c': 3}

更新某个元素

In [30]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

phonebook1 = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225, 'ann':6585

}

phonebook1['joe']

phonebook1['joe'] = 5802

phonebook1

phonebook1['joe']
```

Out[30]: 2598

Out[30]: {'ann': 6585, 'bob': 8982, 'joe': 5802, 'zoe': 1225}

Out[30]: 5802

添加元素

In [31]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

phonebook1 = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225, 'ann':6585

}

phonebook2 = {'john':9876, 'mike':5603, 'stan':6898, 'eric':7898}

phonebook1.update(phonebook2)
```

phonebook1

Out[31]: {'ann': 6585,

'bob': 8982,

'joe': 2598,

'zoe': 1225,

'john': 9876,

'mike': 5603,

'stan': 6898,

'eric': 7898}

删除某个元素

In [32]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

phonebook1 = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225, 'ann':6585

}

del(phonebook1['ann'])

phonebook1
```

Out[32]: {'bob': 8982, 'joe': 2598, 'zoe': 1225}

逻辑操作符

In [33]:

```
phonebook1 = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225, 'ann':6585
```

}

'ann' **in** phonebook1

phonebook1.keys()

'stan' **in** phonebook1.keys()

phonebook1.values()

1225 **in** phonebook1.values()

phonebook1.items()

('stan', 6898) **in** phonebook1.items()

Out[33]: True

Out[33]: dict_keys(['ann', 'bob', 'joe', 'zoe'])

Out[33]: False

Out[33]: dict_values([6585, 8982, 2598, 1225])

Out[33]: True

Out[33]: dict_items([('ann', 6585), ('bob', 8982), ('joe', 2598), ('zoe', 1225)])
Out[33]: False

可用来操作的内建函数

In [34]:

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

phonebook1 = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225, 'ann':6585

}

```
phonebook2 = {'john':9876, 'mike':5603, 'stan':6898, 'eric':7898}
phonebook1.update(phonebook2)
len(phonebook1)
max(phonebook1)
min(phonebook1)
list(phonebook1)
tuple(phonebook1)
set(phonebook1)
sorted(phonebook1)
sorted(phonebook1, reverse=True)
Out[34]: 8
Out[34]: 'zoe'
Out[34]:
'ann'
Out[34]: ['ann', 'bob', 'joe', 'zoe', 'john', 'mike', 'stan', 'eric']
Out[34]: ('ann', 'bob', 'joe', 'zoe', 'john', 'mike', 'stan', 'eric')
Out[34]: {'ann', 'bob', 'eric', 'joe', 'john', 'mike', 'stan', 'zoe'}
Out[34]: ['ann', 'bob', 'eric', 'joe', 'john', 'mike', 'stan', 'zoe']
Out[34]: ['zoe', 'stan', 'mike', 'john', 'joe', 'eric', 'bob', 'ann']
```

常用 **Methods**

In [35]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

phonebook1 = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225, 'ann':6585

}

phonebook2 = {'john':9876, 'mike':5603, 'stan':6898, 'eric':7898}

phonebook3 = phonebook2.copy()

phonebook3

phonebook3.clear()

phonebook3

phonebook2 # .copy() 的“原件”不会发生变化

p = phonebook1.popitem()

p

phonebook1

p = phonebook1.pop('adam', 3538)

p

phonebook1

p = phonebook1.get('adam', 3538)

p

phonebook1

p = phonebook1.setdefault('adam', 3538)
```

p

phonebook1

Out[35]: {'john': 9876, 'mike': 5603, 'stan': 6898, 'eric': 7898}

Out[35]: {}

Out[35]: {'john': 9876, 'mike': 5603, 'stan': 6898, 'eric': 7898}

Out[35]: ('zoe', 1225)

Out[35]: {'ann': 6585, 'bob': 8982, 'joe': 2598}

Out[35]: 3538

Out[35]: {'ann': 6585, 'bob': 8982, 'joe': 2598}

Out[35]: 3538

Out[35]: {'ann': 6585, 'bob': 8982, 'joe': 2598}

Out[35]: 3538

Out[35]: {'ann': 6585, 'bob': 8982, 'joe': 2598, 'adam': 3538}

迭代各种容器中元素

我们总是有这样的需求：对容器中的元素逐一进行处理（运算）。这样的时候，我们就用 for 循环

去迭代它们。

对于迭代 range() 和 list 中的元素我们已经很习惯了：

In [36]:

for i **in** range(3):

print(i)

0

1

2

In [37]:

```
for i in [1, 2, 3]:

print(i)
```

1

2

3

迭代的同时获取索引

有时，我们想同时得到有序容器中的元素及其索引，那么可以调用 enumerate() 函数来帮我们：

In [38]:

```
s = 'Python'

for i, c in enumerate(s):

print(i, c)
```

0 P

1 y

2 t

3 h

4 o

5 n

In [39]:

```
for i, v in enumerate(range(3)):
print(i, v)
```

0 0

1 1

2 2

In [40]:

```
L = ['ann', 'bob', 'joe', 'john', 'mike']
for i, L in enumerate(L):
print(i, L)
```

0 ann

1 bob

2 joe

3 john

4 mike

In [41]:

```
t = ('ann', 'bob', 'joe', 'john', 'mike')
for i, t in enumerate(t):
print(i, t)
```

0 ann

1 bob

2 joe

3 john

4 mike

迭代前排序

可以用 sorted() 和 reversed() 在迭代前先排好序：

In [42]:

```
t = ('bob', 'ann', 'john', 'mike', 'joe')
for i, t in enumerate(sorted(t)):
print(i, t)
```

0 ann

1 bob

2 joe

3 john

4 mike

In [43]:

```
t = ('bob', 'ann', 'john', 'mike', 'joe')
for i, t in enumerate(sorted(t, reverse=True)):
print(i, t)
```

0 mike

1 john

2 joe

3 bob

4 ann

In [44]:

t = ('bob', 'ann', 'john', 'mike', 'joe')

for i, t **in** enumerate(reversed(t)):

print(i, t)

0 joe

1 mike

2 john

3 ann

4 bob

同时迭代多个容器

可以在 zip() 这个函数的帮助下，同时迭代两个或者两个以上的容器中的元素（这样做的前提是，

多个容器中的元素数量最好相同）：

In [1]:

chars='abcdefghijklmnopqrstuvwxyz'

nums=range(1, 27)

for c, n **in** zip(chars, nums):

print(f"Let's assume **{c}** represents **{n}**.")

Let's assume a represents 1.

Let's assume b represents 2.

Let's assume c represents 3.

Let's assume d represents 4.

Let's assume e represents 5.

Let's assume f represents 6.

Let's assume g represents 7.

Let's assume h represents 8.

Let's assume i represents 9.

Let's assume j represents 10.

Let's assume k represents 11.

Let's assume l represents 12.

Let's assume m represents 13.

Let's assume n represents 14.

Let's assume o represents 15.

Let's assume p represents 16.

Let's assume q represents 17.

Let's assume r represents 18.

Let's assume s represents 19.

Let's assume t represents 20.

Let's assume u represents 21.

Let's assume v represents 22.

Let's assume w represents 23.

Let's assume x represents 24.

Let's assume y represents 25.

Let's assume z represents 26.

迭代字典中的元素

In [46]:

```
phonebook1 = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225, 'ann':6585
}
for key in phonebook1:
print(key, phonebook1[key])
```

ann 6585

bob 8982

joe 2598

zoe 1225

In [47]:

```
phonebook1 = {'ann':6575, 'bob':8982, 'joe':2598, 'zoe':1225, 'ann':6585
```

}

for key, value **in** phonebook1.items():

print(key, value)

ann 6585

bob 8982

joe 2598

zoe 1225

总结

这一章的内容，只不过是“多”而已，一旦逻辑关系理顺，就会觉得很简单。而这一章的开头，已经

是最好的总结了。

最后需要补充的，只是两个参考链接，以后有什么搞不明白的地方，去那里翻翻就能找到答案：

https://docs.python.org/3/tutorial/datastructures.html#dictionaries

https://docs.python.org/3/library/stdtypes.html#typesmapping

文件

我们需要处理的数据，一定是很多，所以才必须由计算机帮我们处理——大量的数据保存、读取、

写入，需要的就是文件（Files）。在这一章里，我们只介绍最简单的文本文件。

创建文件

创建一个文件，最简单的方法就是用 Python 的内建函数 open() 。

open() 函数的官方文档很长，以下是个简化版：

open(file, mode='r')

第二个参数， mode ，默认值是 'r' ，可用的 mode 有以下几种：

参数字符

意义

'r'

只读模式

'w'

写入模式（重建）

'x'

排他模式 —— 如果文件已存在则打开失败

'a'

追加模式 —— 在已有文件末尾追加

'b'

二进制文件模式

't'

文本文件模式 (默认)

'+'

读写模式（更新）

创建一个新文件，用这样一个语句就可以：

In [1]:

open('test-file.txt', 'w')

Out[1]: <_io.TextIOWrapper name='test-file.txt' mode='w' encoding='UTF-8'>

当然，更多的时候，我们会把这个函数的返回值，一个所谓的 file object，保存到一个变量中，以便

后面调用这个 file object 的各种方法，比如获取文件名 file.name ，比如关闭文件

file.close() ：

In [2]:

f = open('test-file.txt', 'w')

print(f.name)

f.close()

test-file.txt

删除文件

删除文件，就得调用 os 模块了。删除文件之前，要先确认文件是否存在，否则删除命令会失败。

In [3]:

import os

f = open('test-file.txt', 'w')

print(f.name)

if os.path.exists(f.name):

os.remove(f.name)

print(f' **{f.name}** deleted.')

else:

print(f' **{f.name}** does not exist')

test-file.txt

test-file.txt deleted.

读写文件

创建文件之后，我们可以用　f.write()　方法把数据写入文件，也可以用 f.read() 方法读取

文件。

In [4]:

f = open('test-file.txt', 'w')

f.write('first line**n**second line**n**thrid line**n**')

f.close()

f = open('test-file.txt', 'r')

s = f.read()

print(s)

f.close()

first line

second line

thrid line

文件有很多行的时候，我们可以用 file.readline() 操作，这个方法每次调用，都会返回文件

中的新一行。

In [5]:

f = open('test-file.txt', 'w')

f.write('first line**n**second line**n**thrid line**n**')

f.close()

f = open('test-file.txt', 'r')

s = f.readline() # 返回的是 *'first line\n'*

print(s)

s = f.readline() # 返回的是 *'second line\n'*

print(s)

f.close()

first line

second line

注意，返回结果好像跟你想的不太一样。这时候，之前见过的 str.strip() 就派上用场了：

In [6]:

f = open('test-file.txt', 'w')

f.write('first line**n**second line**n**thrid line**n**')

f.close()

f = open('test-file.txt', 'r')

s = f.readline().strip() # 返回的是 *'first line'*， *'\n'* 被去掉了 ……

print(s)

s = f.readline().strip() # 返回的是 *'second line'*， *'\n'* 被去掉了 ……

print(s)

f.close()

first line

second line

与之相对的，

In [7]:

f = open('test-file.txt', 'w')

f.write('first line**n**second line**n**thrid line**n**')

f.close()

f = open('test-file.txt', 'r')

s = f.readlines() # 返回的是一个列表，注意， *readlines*，最后的 *'s'*

print(s)

f.close()

['first line\n', 'second line\n', 'thrid line\n']

既然返回的是列表，那么就可以被迭代，逐一访问每一行：

In [8]:

```
f = open('test-file.txt', 'w')

f.write('first line\nsecond line\nthrid line\n')

f.close()

f = open('test-file.txt', 'r')

for line in f.readlines():

print(line)

f.close()
```

```
first line

second line

thrid line
```

与之相对的，我们也可以用 file.writelines() 把一个列表写入到一个文件中，按顺序每一行

写入列表的对应元素：

```
In [9]:

a_list = ['first line\n', 'second line\n', 'third line\n']

f = open('test-file.txt', 'w')

f.writelines(a_list)

f.close()

f = open('test-file.txt', 'r')

for line in f.readlines():

print(line)
```

f.close()

first line

second line

third line

with 语句块

针对文件操作，Python 有个另外的语句块写法，更便于阅读：

with open(...) **as** f:

f.write(...)

...

这样，就可以把针对当前以特定模式打开的某个文件的各种操作都写入同一个语句块了：

In [10]:

import os

with open('test-file.txt', 'w') **as** f:

f.write('first line**n**second line**n**thrid line**n**')

with open('test-file.txt', 'r') **as** f:

for line **in** f.readlines():

print(line)

if os.path.exists(f.name):

os.remove(f.name)

print(f' **{f.name}** deleted.')

else:

print(f' **{f.name}** does not exist')

first line

second line

thrid line

test-file.txt deleted.

另外，用 with 语句块的另外一个附加好处就是不用写 file.close() 了……

另一个完整的程序

若干年前，我在写某本书的时候，需要一个例子 —— 用来说明“即便是结论正确，论证过程乱七八

糟也不行！”

作者就是这样，主要任务之一就是给论点找例子找论据。找得到不仅_恰当_且又_精彩_的例子和论

据的，就是好作者。后面这个“精彩”二字要耗费很多时间精力，因为它意味着说“要找到_很多_例子

而后在里面选出_最精彩_的那个！” —— 根本不像很多人以为的那样，是所谓的“信手拈来”。

找了很多例子都不满意…… 终于有一天，我看到这么个说法：

如果把字母 a 计为 1 、 b 计为 2 、 c 计为 3 …… z 计为 26 ，那么：

knowledge = 96

hardwork = 98

attitude = 100

所以结论是：

知识（ *knowledge*）与勤奋（ *hardwork*）固然都很重要；

但是，决定成败的却是态度（**attitude**）！

结论虽然有道理 —— 可这论证过程实在是太过分了罢……

我很高兴，觉得这就是个好例子！并且，加工一下，会让读者觉得很精彩 —— 如果能找到一些按照

同样的计算方法能得到 100 的单词，并且还是那种一看就是“反例”的单词……

凭直觉，英文单词几十万，如此这般等于 100 的单词岂不是数不胜数？并且，一定会有很多负面意

义的单词如此计算也等于 100 罢？然而，这种事情凭直觉是不够的，手工计算又会被累死…… 于

是，面对如此荒谬的论证过程，我们竟然“无话可说”。

幸亏我是会写程序的人。所以，不会“干着急没办法”，我有能力让计算机帮我把活干了。

很快就搞定了，找到很多很多个如此计算加起来等于 100 的英文单词，其中包括：

connivance（纵容）

coyness（羞怯）

flurry (慌张)

impotence（阳痿）

stress（压力）

tuppence（微不足道的东西）

……

所以，决定成败的可以是“慌张”（flurry），甚至是“阳痿”（impotence）？这不明显是胡说八道嘛！

—— 精彩例子制作完毕，我把它放进了书里。

那，具体的过程是什么样的呢？

首先我得找到一个英文单词列表，很全的那种。这事儿用不着写程序，Google 一下就可以了。我搜

索的关键字是“english word list” ，很直观吧？然后就找到一个：https://github.com/dwyl/english-

words；这个链接里有一个 words-alpha.txt 文件，其中包含接近 37,0101 个单词，应该够用了！下

载下来让机器人计算就可以了！

因为文件里每行一个单词，所以，就让程序打开文件，将文件读入一个列表，而后迭代这个列表，

逐一计算那个单词每个字母所代表的数字，并加起来看看是否等于 100？如果是，就将它们输出到

屏幕…… 好像不是很难。

In [11]:

```
with open('words_alpha.txt', 'r') as file:

for word in file.readlines():

pass # 先用 pass 占个位，一会儿再写计算过程
```

按照上面那说法，把 a 记为 1 ，直至把 z 记为 26 ，这事儿并不难，因为有 ord() 函数啊

—— 这个函数返回字符的 Unicode 编码： ord('a') 的值是 97 ， 那按上面的说法，用

ord('a') - 96 就相当于得到了 1 这个数值…… 而 ord('z') - 96 就会得到 26 这个数

值。

In [12]:

```
ord('a')
```

Out[12]: 97

那么，计算 'knowledge' 这个字符串的代码很简单：

In [13]:

```
word = 'knowledge'

sum = 0

for char in word:

sum += ord(char) - 96

print(sum)
```

96

果然，得到的数值等于 96 —— 不错。把它写成一个函数罢： sum_of_word(word) :

In [14]:

```
def sum_of_word(word):

sum = 0
```

```
for char in word:
sum += ord(char) - 96
return sum
sum_of_word('attitude')
```

Out[14]: 100

那让程序就算把几十万行都算一遍也好像很简单了：

In [15]:

```
def sum_of_word(word):
sum = 0
for char in word:
sum += ord(char) - 96
return sum
with open('words_alpha.txt', 'r') as file:
for word in file.readlines():
if sum_of_word(word) == 100:
print(word)
```

abstrusenesses

acupuncturist

adenochondrosarcoma

adenomyxosarcoma

adscititiously

adsorptiveness

anaglyptography

anesthetization

anisophyllous

annihilationistic

anomophyllous

anthropopathite

anthropophaginian

antiagglutinating

antianaphylactogen

antibacteriolytic

antichristianism

anticyclolysis

anticytolysin

anticommutative

anticonvulsive

antieducationally

antiexpressive

antimilitaristic

antimissionary

antipestilential

antiprofiteering

antirepublicanism

aponogetonaceous

apostrophising

apperceptionist

aristocratically

aristodemocratical

arthrolithiasis

associationalist

assortatively

astigmatometry

atherosclerosis

autoeciousness

autointoxicant

autoprothesis

autosymbiontic

benzoglyoxaline

benzothiodiazole

bioastronautics

bisymmetrically

blennocystitis

carposporangium

chondropterygii

chromolithograph

cytoplasmically

cochromatography

collenchymatous

commercialization

communalization

competitorship

complementalness

compossibility

congressionist

conjecturableness

consolitorily

contemplatingly

contradictorily

contrapositive

controllableness

cooperativeness

cosponsorship

counsellorship

counterpassant

counterpotent

counterspying

countertendency

counterworker

countrywomen

craspedodromous

cryptanalytics

cryptostomata

cultivatability

cultrirostral

curvilinearity

dacryostenosis

dehydrosparteine

demonstrability

demonstrations

denominationalize

dephlogistication

dermorhynchous

desophistication

desoxycinchonine

desterilization

diathermotherapy

dyschromatopsia

discriminatingly

dyscrystalline

discursiveness

disenfranchisement

disfranchisements

disgustingness

dispensatorily

draftswomanship

duplicitously

electrobiologist

electrocardiograms

electrodepositable

electromagnetist

electromuscular

electroresection

electroviscous

embourgeoisement

endotheliolysin

epistolography

erythrolitmin

erythropoietic

eurytopicity

eventognathous

existentialist

experientialist

experimentative

expostulator

exquisiteness

extemporalness

exterminations

externalisation

extraconscious

extradictionary

ferromolybdenum

fibroligamentous

firmisternous

flavorlessness

flavorousness

forerunnership

forthrightness

frictionlessly

fruitlessness

gastrolatrous

geomorphogenist

gymnospermism

gyrophoraceous

gyrostabilizer

governmentalize

gravitationally

guanidopropionic

hastatosagittate

helminthologist

hematocytometer

hemotherapeutics

hydrargyrosis

hydroergotinine

hydronitrous

hydrophyllium

hydroquinoline

hydrotherapist

hyperalkalinity

hyperbrachycephalic

hyperimmunized

hyperthermally

hyporrhythmic

hyposensitive

hypostatically

hypostomous

hippocastanaceous

hipponosology

hipponosological

hysterocleisis

historiographer

homeomorphisms

homotransplant

iliohypogastric

immunologists

immunotherapies

imposturous

impoverishment

impropriatrix

incommutability

inconstantness

incorporations

incuriousness

indispensableness

industrializes

intensifications

intensitometer

intercursation

interirrigation

interjectionary

interpretress

intersituating

interstimulate

intraformational

intraleukocytic

intuitionless

intuitiveness

intussuscept

irrespectively

isopropylacetic

yttrogummite

lactiferousness

laryngocentesis

levorotatory

limonitization

liverwursts

locomotiveness

lotophagously

lubriciousness

luciferousness

maldistribution

mammatocumulus

manuscriptural

martyrolatry

masculinization

melanospermous

mercuriammonium

mesmerizability

mesopterygium

metempsychosical

metropolitancy

metroscirrhus

mycosymbiosis

microphyllous

microprocessor

ministerialness

myodynamometer

myringodectomy

misinterpretable

monoeciousness

monopropellant

monumentalising

morphologists

multicomputer

multilinguist

multimetallist

multistratified

necessitously

nesslerization

neurapophysis

neurilemmatous

neurosurgery

neutroclusion

nonagricultural

nonappointment

nonarticulation

nonattainability

nonbituminous

nonblunderingly

noncohesiveness

noncollectively

noncommittally

nonconstraining

noncontiguity

noncruciformly

noncuriously

nondeductibility

nonderogatively

nondisruptive

nondissipatedly

nondistortion

nonegregiously

nonemulously

nonequivocating

nonestimableness

noneuphonious

nonexternalized

nonextrusive

nonfestiveness

nonformidability

nonfrequently

nonimperialistic

nonindulgently

noninfiniteness

noninheritabness

nonintersecting

noniridescently

nonleprously

nonmanipulative

nonmathematically

nonmischievous

nonnarcissistic

nonoutlawries

nonparticipating

nonpermanently

nonpersecutive

nonperseverant

nonphilosophy

nonphilosophical

nonpoisonous

nonpreciously

nonpredatorily

nonprotesting

nonpsychopathic

nonpuerilities

nonrecuperative

nonrehabilitation

nonrenunciation

nonresonantly

nonretrenchment

nonromantically

nonsanctification

nonsculptural

nonsecretively

nonsensibility

nonsensification

nonsentiently

nonseriously

nonsolubleness

nontemperamental

nontheistically

nontypicalness

nonubiquitary

nonupholstered

nonusurious

nonvulgarities

normalizations

octophthalmous

oligophyllous

omniproduction

operationalistic

ophthalmencephalon

ophthalmoscopy

ophthalmoscopical

opinionatively

optimizations

ornithopteris

orthodoxality

orthotropism

ossiculotomy

ostensibilities

osteomyelitis

ostreiculture

ovariectomizing

overaccentuation

overbashfulness

overcaustically

overcommendation

overcourtesy

overdefensively

overfearfulness

overgesticulated

overjoyfully

overliveliness

overnarrowly

overnationalize

overnumerous

overroughness

oversensitize

overslowness

overstrictly

overthrowers

overtruthful

overvigorous

palaeodictyoptera

paleoethnography

paleomammologist

parametrization

parasigmatismus

parentheticality

participatively

patronizingly

penetratingness

perityphlitis

permittivity

perniciousness

persnicketiness

petrographically

petromyzonidae

petrosiliceous

phylloptosis

physicochemically

phytogeography

phytogeographical

photographically

photoptometer

photosynthate

phthisiologist

pyrenocarpous

pyroantimonate

pyrophorous

pyrotechnically

pleocrystalline

pleurosaurus

plumbosolvent

plurivorous

pneumatogenous

pneumatometry

pneumonotomy

poecilocyttares

pointlessness

polariscopically

polychromatize

polyembryonate

popularisation

posterishness

posteruptive

posteternity

postexistent

posttreatment

pratiyasamutpada

preacknowledgment

preassumption

precorruptive

predesirously

predetermination

preindependently

preintercourse

preprocessors

prereconciliation

presymphysial

presuitability

presupplicating

preteressential

preventionist

prioristically

proacquisition

proauthority

procompulsion

prodeportation

professionalised

professionally

profitmongering

progeotropism

prognostically

prohostility

proletarianness

propolization

proportionated

proportioning

protomagnesium

protomeristem

protorosauria

protosulphate

prototrophy

protractility

providentialism

proximolingual

pseudodipteros

pseudofoliaceous

pseudonymity

pseudopermanent

pseudosophist

psychiatrists

psychoanalysis

pterylography

pterylographical

pteroclomorphic

pumpkinification

pupilloscoptic

purposefully

pussyfooting

quadricentennials

quarterstaves

querulously

questionnaires

quinquelocular

quinquepetaloid

quinquevalency

rapturously

reassortments

recollectiveness

reconstructing

reconsultation

refamiliarization

reformulations

reharmonization

rehypnotizing

relentlessness

repetitiveness

repressibility

repressionist

reproducibility

requisitioning

restaurateurs

retributively

reverentialness

rhinocerotiform

rhythmization

rhodospermous

rontgenoscopy

rumblegumption

saponaceousness

satisfyingness

scintillatingly

sclerotization

scrofulously

scutelligerous

sedimentologist

semianatropous

semimembranosus

seminuliferous

semiphenomenally

semipictorially

semipropagandist

semitendinosus

sentimentality

shirtlessness

sympatholytic

syphilologist

siphonognathus

slaveownership

snippersnapper

sparrowwort

spermatocystic

spermatogonium

spermophorium

sphygmographies

splanchnopleuric

sporomycosis

sputteringly

squamotemporal

stereoisomerical

stylistically

stillatitious

stylomandibular

stylotypite

strawberrylike

stroboradiograph

subantiqueness

subantiquities

subassociations

subcompensation

subconjunctival

subcontiguous

subcorporation

subdirectorship

subexpression

subhypothesis

subnaturalness

subpartnership

subpharyngeally

substantialist

sulphurosyl

sunburntness

superacuteness

superconsecrated

superfecundity

supergravitated

superinfusion

supermilitary

superocularly

superponderant

superprinting

superproducing

superreflection

supportress

suppositional

suprahumanity

supraocclusion

suspensively

sussultorial

telegraphonograph

tendenciousness

terminalization

terpsichoreally

terrestrialism

territorialism

tetrapneumones

tetrapneumonian

thelyotokous

theomythologer

theriomorphism

thermotelephonic

thymolphthalein

thyrocalcitonin

thyrotrophin

thirtytwomo

thoracomyodynia

tylosteresis

tyroglyphus

tithymalopsis

tocodynamometer

topochemistry

toponeurosis

torrentiality

torrentuous

tortuously

toxophorous

transitivity

transparentize

transversally

trichoepithelioma

trimeresurus

trinitrocarbolic

trypanosomacidal

trophoplasmatic

tubercularizing

ultrabenevolent

ultrabrachycephalic

ultrainclusive

unattestedness

unattributably

unaudaciousness

unauthentically

unavertibleness

unbenevolently

uncalamitously

uncapriciously

uncategoricalness

uncompromising

uncondensableness

uncongressional

uncontemningly

uncontinently

uncontinuous

uncontortedly

uncrossableness

underconstumble

undergoverness

undeteriorative

undetestability

undisinterested

undispassionate

undistortedly

unembellishedness

unexceptionably

unexpectability

unexplanatory

unexpropriated

ungratuitous

unhistorically

unhumourous

uninvigorative

uninvolvement

unirritableness

universalizing

universitatis

unjournalistic

unlibidinously

unmunificently

unnervously

unorientalness

unorthographical

unperemptory

unphysiological

unplutocratical

unprecipitous

unpredisposing

unprotrudent

unputatively

unreluctantly

unremunerative

unreproachingly

unsecretarylike

unsensitizing

unsensualistic

unstealthiness

unstimulative

unstorminess

unsubstantiate

unsufficingness

unsusceptibly

unsuspicious

untenantableness

unterminational

unulcerously

ununduLatory

unveritableness

unvictorious

unwatchfulness

uredosporous

ureterolysis

vapourishness

vasoinhibitory

vasostimulant

ventrifixation

ventriloquise

ventripotency

violoncellists

virtuosities

viscometrically

vitreousness

whitlowwort

wondrousness

worshipability

zeuctocoelomatic

zygapophysis

嗯？怎么输出结果跟想得不一样？找到的词怎么都“奇形怪状”的…… 而且，输出结果中也没有

attitude 这个词。

插入个中止语句， break ，把找到的第一个词中的每个字符和它所对应的值都拿出来看看？

In [16]:

```
def sum_of_word(word):

sum = 0

for char in word:

sum += ord(char) - 96

return sum

with open('words_alpha.txt', 'r') as file:

for word in file.readlines():
```

```
if sum_of_word(word) == 100:
print(word)
for c in word: # 把字母和值都打出来，看看对不对？
print(c, ord(c) - 96)
break # 找到一个之后就停下来。
```

```
abstrusenesses
a 1
b 2
s 19
t 20
r 18
u 21
s 19
e 5
n 14
e 5
s 19
s 19
e 5
s 19
```

-86

怎么有个 -86 ？！仔细看看输出结果，看到每一行之间都被插入了一个空行，想到应该是从文件

里读出的行中，包含'\n'这种换行符…… 如果是那样的话，那么 ord('\n') -96 返回的结果是

-86 呢，怪不得找到的词都“奇形怪状”的……

In [17]:

ord(' **n**') -96

Out[17]: -86

改进一下呗 —— 倒也简单，在计算前把读入字符串前后的空白字符都给删掉就好了，用

str.strip() 就可以了：

In [18]:

```
def sum_of_word(word):

sum = 0

for char in word:

sum += ord(char) - 96

return sum

with open('words_alpha.txt', 'r') as file:

for word in file.readlines():

if sum_of_word(word.strip()) == 100:
```

print(word)

abactinally

abatements

abbreviatable

abettors

abomasusi

abreption

abrogative

absconders

absinthol

absorbancy

acceptavit

acceptors

acclimation

accounter

accumulate

acenaphthene

achronism

achroous

acylation

acknowledge

acolytes

acquisita

acquitted

acriflavine

acromegaly

acronychal

acronycta

acronyx

actinocarp

activates

acuminose

acurative

addressing

adelocodonic

ademonist

adiabatically

adipopexia

adsessor

adulthood

advantaging

adventual

adverting

aeolipyle

aequorin

aeriality

aerofoils

aerometer

aetosaur

affectation

affricative

afghanistan

africanist

aftercareer

agalactous

agamogenetic

agapornis

agariciform

aggresses

agnations

agrypniai

agrology

agrological

aichmophobia

aydendron

airdrops

airmonger

aistopoda

albedometer

albuminoid

alchemising

alertest

aleurodes

alfaquins

algaeology

algaeological

alienation

alineation

aliturgic

alkalinize

alkoxyl

allenarly

allentato

alligation

alloquial

allottable

allthorn

almoravide

alopecist

alphonso

alpinery

alpinist

alrighty

altarist

alternated

altiplano

altiscope

amanitins

amarillos

ambulating

amelcorns

ameloblast

ametropic

amiableness

amyluria

ammiaceous

amoebobacter

amoralize

amortise

amphiboles

amphisbaenic

amphiumidae

amputees

amusedly

anacamptics

analysis

anapodeictic

anastaltic

anchoritic

andesinite

androclinia

anepiploic

aneurism

angelologic

angerless

angiectopia

anginous

angioblast

angiotribe

angiport

anglemeter

anglophobia

angulates

anhydrated

anhydremic

anilinism

animately

animaters

anisocercal

annapurna

annually

annullate

anomalipod

anomalure

anophelinae

anorchism

answerable

antalkaline

antalkalis

antarctalia

antepaschal

anteporch

anterior

anthozoa

anticathode

antichlor

anticomet

anticult

antifowl

antigalactic

antimarian

antislip

anvilling

apheresis

aphrodisia

apiology

apneumona

apoharmine

apokreos

apoplectic

aporetical

apostacy

apotracheal

appeasers

apperceive

appertain

applicancy

appliedly

applying

appointed

appraisable

apropos

aquamarine

aquiform

araneiform

arbitrages

arbuscles

archaeolith

archaicness

archdiocesan

archenemies

archibenthic

archigony

archilithic

archimagus

archiplasm

archsewer

arcticward

arenilitic

areometer

argillitic

argiopoidea

argumenta

arhatship

aryanism

arightly

arrastre

arrests

arrivals

arrowy

arsenium

artilize

arugulas

asbestos

ascendants

ascyrum

ascophore

asyndetic

asparagyl

aspergilla

asphalter

asplanchnic

assafoetida

asswaging

asterales

asterioid

asthenies

atheromas

athetosic

athyris

athyroid

athreptic

athrogenic

atonally

attargul

attitude

attunes

auctorial

audiophile

augments

aulophobia

auntlier

aureoline

aureous

auriculo

auriscalp

auslaute

autoclave

autoharp

automated

avanters

avernus

aversant

avidious

avocation

avouching

awfully

aworry

azurite

babesiosis

bacilliform

backslashes

backswept

backtracking

backwardly

baconianism

baculiform

baguettes

baldnesses

balistarii

balletomane

ballonets

bananaquit

bandwagons

bannerols

barbarising

bardolphian

bariatrics

barytone

barkeepers

barkpeeling

barleybrake

barleybreak

barometz

baronetical

barracouta

barspoon

bartizaned

bartonella

basidorsal

basilateral

basiotribe

basipodite

bassanello

bassetite

bastinading

bathyscape

batidaceous

batonist

batrachoididae

battailant

battement

baulkiest

bawsunt

beautihood

becarpeting

becrowding

bedazzles

bedposts

beeftongue

beestings

beetroot

beginnings

beguileful

belinuridae

bellwaver

bemajesty

benediction

benzolate

bergamots

beryllate

berlinize

beshrivel

besmircher

bespangles

bespreading

bestirred

bestridden

beswarms

betattered

bettering

bevellers

bewhisker

bewitching

biathlons

bibitory

bibliophobia

bibliotics

biddulphiaceae

bifarious

bigaroons

bikukulla

bilifaction

bilocation

biloculate

bimillennia

bimorphs

bintangor

bioassayed

biologize

biophyte

biovular

bipartite

byplays

birthmark

bisantler

biscutate

bisinuate

bismarckian

bissonata

bittings

biunity

biweeklies

biwinter

blackfriars

blarneyer

blasphemes

blastoffs

blazoning

blennogenic

blethering

blighters

blissful

blockwood

bloodwit

blowtube

bluegums

boatloading

bodyplate

bogberries

bogwoods

bogwort

boycott

boilerful

bolstered

bombacaceous

bonhomies

bonneting

boobyism

boohooing

boomless

boondoggled

bootblacks

bootery

bootmaker

boozers

bordroom

borneols

borrowed

boskiest

bosporan

botanist

bouillon

boulevard

bounceably

boundary

boundure

bountree

boviform

bowerlet

bowerly

bowerlike

bowknot

bowlmaker

boxboards

brachiation

bractlets

brahmaness

braquemard

brawlys

breakshugh

breathily

breediness

breezeful

bremeness

brevetcy

breviary

breviconic

brewises

brezhnev

bridgeless

bridgemaking

brigatry

brightish

bromauric

bronchocele

bronchus

bronzy

broodily

brooklime

broomweed

browser

browsick

bruiting

brushier

brushmen

bufotalin

bugproof

bulgurs

bulliest

bullnose

bullpup

bullskin

bumblekite

burnished

bushlands

busticate

butcherer

buzzy

cacidrosis

cacogenesis

cacomistle

cacophony

cacophonical

caecotomy

calaminaris

calciphyre

calcitonin

calculist

caligraphy

calypter

callitriche

calvarium

calvities

campanular

camphorate

cancriform

candidature

canephoroe

cannibalized

cantrips

capiteaux

capitolian

caponiers

caponiser

caponniere

captaincies

carbazylic

carboluria

carboxyl

carburised

carcharodon

cardiograph

cardioplegia

cardiorenal

caressant

caryatids

carkingly

carlyleian

carlylese

carotenes

carouser

carpenter

carpetweed

carrioles

carroty

cartelism

cashdrawer

cassalty

cassiopeid

castores

catalanist

catamneses

cataphracted

cataphracti

catarinite

catechisms

catechistic

catenative

catholicon

catteries

catwort

causeries

cavernlike

cavitates

celibatist

cellarway

cellfalcicula

celticist

celtophil

cencerros

cenogenetic

censorate

censurable

centipedes

centupled

cephalous

ceramicist

ceratites

ceratomania

cerithioid

cerusite

chaenomeles

chaetosoma

chaiseless

chamecephaly

champignon

chaplaincies

charwomen

chastening

chattery

chaucerism

chaussees

cheesemaking

chemigraphic

chemotactic

chemurgy

chemurgical

cherishing

chickories

chiliasts

chimpanzee

chippewas

chiropodic

chirpily

chivariing

chloranaemia

chlorellaceae

chloropal

choiceness

chondrioma

chondrule

choppers

chorioids

chorisis

chortles

chowries

chrysalida

chromium

chucklers

churchful

churingas

chutist

cyaphenine

cyathium

cycadaceous

cicatricula

cyclitis

cilicious

cymation

cymophane

cinchotine

cingulum

cyniatria

circleting

circumduce

cirurgian

cisjurane

cisleithan

cysteine

cystidean

civilizade

civilizee

claytonia

clangoring

clangoured

clarifiers

classily

claudetite

clausure

cleanliest

clearhearted

clementine

clerically

clerkdoms

cleveites

climatarchic

clinically

clockwise

clomiphene

clotting

clovery

clubster

clumsier

coannexes

coarsest

coassert

coassumed

coasters

coatroom

coattails

coauthered

cobleskill

cobwebbery

cockneyfied

cockshut

cocottes

coderives

coenures

cofactors

cognatus

cogredient

coyness

coislander

cojuror

colascioni

colazione

colleagues

collecting

colloque

colonials

colopexia

colophenic

colubrinae

columbite

columels

cometary

commandeered

commercing

companero

companion

compering

competible

complanate

complicacy

comport

concavely

conchitis

concludible

condensate

confabulate

confederated

confrater

congeners

congenital

congiaries

congress

conjoint

conjugated

conjunct

connivance

conniver

conodont

consigns

consumo

contented

contoise

contrude

cooingly

cookeries

cookout

cooniest

coonskin

coonties

coparceny

copyism

coplots

coproduce

coprosma

coquets

coquito

corbeilles

coresidence

corinthiac

coryphaei

coryzal

corkiest

corkwing

cornette

cornmeals

cornuted

corollet

corollike

coromandel

corridor

corticole

cosmetical

cossets

costumed

cotabulate

cotenancy

cotillon

cotters

couchette

coulombs

courses

courter

courtin

cousins

covalency

coverlet

coverside

coverup

cowardish

coxcomby

coxcombical

crackleware

craniomalacia

crankiest

craspedum

cravenly

crebrity

credentialed

creepiest

creosote

crepehanger

crepeiest

cresoline

cryalgesia

crimison

crimsoned

criticule

crowdweed

crudity

cruising

cruisken

crummier

crusados

crusts

cuadrillas

cubicities

cuculus

culicoides

cullises

cultrate

culture

culvers

cumulated

curcuddoch

curdlers

curettage

curioso

curledly

curlily

curtalax

cuspidine

customed

cutdown

cutesier

cutinise

cutlases

cutlets

cutlips

cutout

cuttles

cutups

czarship

dacryuria

daywrit

danewort

dartrose

dawsoniaceae

deadeningly

deadworks

deathshot

debamboozle

debarkation

debaucheries

debordment

debussing

decalcomanias

decerebrize

deciduity

declarative

declension

decompiler

decorous

decouples

decremental

decrypted

deepfroze

deerberry

deescalating

defections

defeminized

deferrized

definitise

definitor

deflators

deflexure

degausses

deglaciation

degreewise

dehorting

deywoman

delectating

deliberates

delineating

deliquesce

delivery

delouses

deltation

demibuckram

demigriffin

demisuit

demivolt

demobilize

demodulate

demoniast

dempster

denasalized

dendrocoele

denitrated

denominate

denounces

dentalized

denumerable

denunciated

dephycercal

dephlegmated

depilator

deplaster

depletion

deploring

deprivate

derbylite

deresinate

derivers

derogating

derrickmen

describably

descriers

desegregated

desiccative

designers

desmidiales

desmodus

desolates

despatches

desponder

despotat

destuffs

deterring

developpe

dewberry

diagonally

diakinesis

dialysing

dialyzer

diamondize

diaphyseal

diaphonies

diaschisis

dichotomal

dichromasia

differencing

digestive

digladiator

dihexagonal

dykereeve

dilatants

dilatator

dimensive

dimethoate

dinginess

dinitrate

dinitrile

dioctahedral

diodontidae

dioecious

diopsides

diphygenic

diphtheric

diplodus

diplomaing

diplotene

disarranged

discanting

discernible

discipline

disclaiming

discommode

discophile

discredited

discrepate

disembogue

disgavelled

disgress

dishelming

dishouse

disimpark

disjecting

disjoinable

disjoint

disjunct

dyslectic

dislodging

dismarket

disobliger

disomus

disparageable

disparple

dispeller

dysphemia

disponer

dispraise

disroot

disrump

disseized

dissuader

distancing

distrait

disunified

ditchdown

dithyramb

ditroite

diureses

diurons

divekeeper

divertila

divinely

diviners

divorcees

dochmiasis

dockyards

dogmatize

dogwinkle

dollymen

dolomitic

domiciliate

dominates

dooryard

doormaker

dormette

dormeuse

dorosoma

dorsigrade

doubleheader

doughty

downcut

downlier

downset

dowsabels

draftsmen

dragonism

drawlers

drawplate

drybrained

dryrot

driveling

drizzle

droopier

droshky

droskies

drossy

drumbling

drumlier

druttle

dualities

duckhearted

ducklings

ducktails

duetting

dulcimore

dumbfound

dumpily

duologue

durative

durdenite

durions

duvetine

earthmaker

earthwall

earwigging

earwort

ebullient

ecchymosed

echiniform

echinoderidae

echopraxia

eclecticize

ecstasies

ectosteal

educatedly

effectless

efoliolate

egoistical

eyeleteer

eightsmen

ejections

elaeothesia

elargement

eldmother

electives

electorial

elementate

elephantidae

elephants

elevenfold

elohistic

elotillo

elsewhere

emasculate

embodiment

emboldening

embolismic

embowers

emendation

emotioned

empathetic

emperess

empresse

emptily

enaluron

encapsuled

encarpium

enclasping

encrust

encumbers

endochrome

endocritic

endoplasma

endostraca

endothecate

energiser

enflagellate

enfoulder

engagedness

enharmonic

enlivens

enneadianome

enoplion

enrolles

ensorceled

enstyle

enstool

ensurance

ensurer

entohyal

entrains

envoys

enweaving

epibolism

epicostal

epigaster

epigonos

epilimnial

epimerite

epimerum

episcleral

episematic

epistlar

epitaphical

epitaxy

epizoon

eponymic

equalized

equative

equipages

equiradical

equison

equispaced

erasement

erasions

erubescence

espouse

esprove

essoins

esterified

estheses

esthesio

estopped

estuant

eteoclus

ethanoyl

etherized

ethylated

ethnicism

ethnogenic

eucrites

eugenist

eulogize

eunuchoid

eupatridae

euphenics

eutony

evechurr

eventual

everting

evolute

evolves

exanthine

excavates

excellent

excerpted

excoriate

excussed

exdividend

execrates

exergonic

exhibiter

exigencies

exintine

exister

exocyclica

exophasic

exoterica

expalpate

explains

explodes

exsculp

exserted

extendible

extracted

exzodiacal

faculative

fairyism

faithwise

falciparum

falseness

falsities

fanflower

fantasts

fantoddish

farnovian

farriery

farrows

fasciculate

fascinery

fatalisms

fatalistic

fatherhood

fatherling

fattiest

fauvette

featherlet

featherlike

featherweed

feldspars

felinity

fellowred

feltness

fenestral

feoffeeship

ferments

fermillet

ferrament

ferryman

ferrites

ferrums

fervanite

festally

fewterer

fiddlerfish

fierasferid

filiciform

filipiniana

fillagreing

filtering

finitely

firebombing

firebricks

firmity

fishnets

fishpool

fissioned

fissipedal

fivepins

flaccidities

flagellates

flaggingly

flangeless

flankwise

flatlings

flavanilin

fleawort

fleysome

flyboats

flichters

flyness

flywinch

flockiest

floodwall

floosies

fluidist

fluorine

fluoroid

flurry

fluxweed

focometer

foenngreek

foetalism

fogscoffer

follying

fondlings

foolship

footfolk

footpaces

footsy

forbearant

forebearing

foredoomed

forefoot

foreiron

forepast

foreshock

foresides

forespeech

forewarn

forgiver

formeret

forsaking

forsung

fortranh

forwoden

forwore

foujdary

fountain

fourbagger

foveolet

foziest

fractural

fraughts

freetrader

freewoman

frenchily

freshest

freshets

friendlier

frigorific

fringillid

frisson

frogeyes

frolicly

froughy

fuehrers

fugacities

fulgour

fulimart

fumaroidal

fumeless

fundless

furcular

furless

furnacemen

fursemide

furuncle

gablatores

galactagogue

gallberry

galleasses

gallicanism

galliwasp

gangliomata

garlandry

garnishes

gascoigny

gasconism

gasoliner

gasoscope

gasteralgia

gauntlet

gauntly

gawkhammer

gecarcinus

gemsbucks

generating

genetmoil

gentianella

gentisate

gentrices

geococcyx

geographer

geomantical

geophagism

geotaxis

geraniols

gerenuks

germanely

germanhood

gestening

ghostland

giftwrap

gigartinaceae

gilravager

gymnogen

gingerline

ginkgoales

ginneries

gypsies

gyrally

girasols

gyromele

giustina

gladatorial

gladfully

gladiolus

glairiest

glassfish

glassteel

glauberite

glaucidium

glycogenic

glycolate

glyconian

glimpses

glissando

glittered

glomerella

glorifies

glossoid

glowfly

glucinum

glucosin

gluemaking

glutamate

glutelin

gnatcatcher

gnathopod

gnostical

gobbledegook

gobernadora

gobletful

goblinism

godparent

goldbricks

goldurned

golliwog

goloshes

goniatite

gonothecal

gooneys

gorgonacean

gorgonian

goutweed

governs

graybeards

grainsmen

graithly

grandaunt

granogabbro

granville

grapeskin

grappling

gravamens

greatening

greenflies

grenadiers

griefless

grillers

grimiest

grindery

grivets

grizelin

groenendael

groggily

grooty

groover

grouper

grouts

grovelled

growly

gruelly

grumpy

grunzie

guayaberas

guaiasanol

guarantied

guildsman

gullery

gullibly

gunshop

gusseted

guttered

guttier

hackbarrow

haematinon

haemophiliac

hailproof

hainberry

hairgrass

halakists

halftones

halleflinta

hamesucken

hammerers

hammerfish

hammerkop

handstone

haranguing

harmonics

harrying

haruspice

hastifly

haustral

hawknosed

headclothes

headhunts

healthily

healthward

heartblood

heatedness

heelpost

hegemonies

heydeguy

helianthin

heliophobia

hellbroth

helleborin

helmetlike

helotize

hematites

hematogenic

hemiablepsia

hemicollin

hemidactyl

hemidomatic

hemiekton

hemiplegy

hemiterata

hemoglobin

hemology

hennebique

henrietta

hepatomata

hepteris

heraldship

herbarist

heritably

hermetics

herpetoid

heterocerc

hexacoralla

hexarchies

hexaster

hyalescence

hydrachnidae

hydrates

hydrazide

hydromel

hifalutin

highways

hyingly

hylarchical

hillfort

hinderers

hindwards

hinnying

hyperemia

hypohemia

hypopnea

hypothec

hypoxic

hippidion

hippuric

hyraxes

hiroshima

hirsute

hysons

hitchhikes

hlidhskjalf

hobbyist

hobbistical

hoboisms

hodoscope

hoggaster

hollandaise

hollandite

hollantide

hollering

holocarpic

holograph

homomeral

honduras

hondurean

honeycomb

honoring

honoured

hoofbound

hookerman

hooligans

hooters

hoppercar

horation

horntip

horologia

horseherd

horsify

hospitage

hospital

hostaging

hotblooded

hotelize

hotness

housemen

houtou

howlers

huccatoon

huehuetl

humblest

hungrier

huronian

hurrayed

husbandland

hussies

hutzpah

yahooish

yalensian

yankton

yaourt

yappers

yardbirds

yarners

yarrow

yarwhip

ichorrhoea

ichthyol

ideagenous

identifies

ideologies

ideologise

idyllion

idiogenetic

idiotry

idolisms

idolistic

yeasting

yelpers

ignatius

ignifying

iguanodon

yirring

illyrian

illoricata

illuminee

illutate

imbroglio

immature

immitigable

immusical

impaction

impaneling

impartance

imparter

impellor

impendent

imperent

imported

impotence

impresas

impressa

imprest

impugnable

inadequacy

inaneness

inanities

inapplicable

inbreather

incensive

inceptor

inchoating

incogitance

incongealable

increeping

incubating

indetectable

indicolite

indignancy

indirubin

indologian

inductees

indument

indurite

ineconomic

inefficient

inexist

infantive

infernos

inferring

infidelism

infixion

inflation

infringer

infuneral

ingroup

ingrown

inkiness

innately

innovate

inoculate

inrooted

insecticide

inshoot

inshrined

insooth

inspinne

instars

instore

intellect

intelsat

interbreed

interfaces

interfere

intermat

interpale

interpeal

interplea

intertie

intexine

intrados

intrapial

intrenched

introfied

inulases

inwound

inwraps

iodonium

iodophor

yohimbine

yokeldom

ioniums

irideous

irishly

irritate

isleless

ismaelism

ismaelitic

isocephalic

isocheims

isochimes

isolates

isophasal

isthmics

ytterbia

yukaghir

yuruna

iwberry

jackpudding

jaguarondi

jailhouse

japishly

jarveys

jasminum

jatrophic

javitero

jawfishes

jazzlike

jean-christophe

jessamies

jillions

jimberjawed

jimmying

jostles

journaled

judiciary

jumblers

jumbucks

juncoides

junkets

jurassic

justing

juvavian

kailyards

keelsons

keeshonden

keeshonds

kehilloth

keyboards

keyholes

ketonimid

keurboom

kibitzer

kiddushes

kimeridgian

kingcups

kingsize

kissers

kytoon

kleistian

knappers

knifeless

knockoffs

knothole

knouts

knowledged

knulling

kohlrabies

koimesis

kolinski

kookery

kousso

krakowiak

kristin

kuvasz

laborsome

laceflower

lachrymable

laciniform

lacquerer

ladderwise

lageniform

lagniappes

laliophobia

lamarckism

lambskins

laminating

lampstand

landholding

landscaping

landsting

languished

lapulapu

larcinry

largeness

laterigrade

latinized

latirus

lavature

lavenders

lazybone

lazyish

lazuline

lazulis

leaderless

leaseholds

leashless

leaveless

leftness

legatorial

legumins

lemmitis

lennilite

lepidity

leucocism

leucojum

leucophane

leveraging

levigates

libellist

liberalism

liberator

libytheidae

libration

lyctus

lievrite

liferoot

lifeways

ligaments

lightest

lightning

likelihood

limbuses

liminess

limpidly

lyngbyaceae

lingtow

lyricise

lysimachia

litation

literato

litotes

lyxose

localness

lodicules

loftsman

logicity

loginess

logophobia

logrolled

longeron

longobardic

longship

longsome

loopers

loricarian

loviers

lowigite

lowlands

lowlifer

lubricant

luckiest

lugsails

lumpens

lunchers

luniest

lunkers

lupinine

luteins

lutraria

lutrinae

macabreness

macartney

machismos

macrocosm

macrograph

macromazia

macrozamia

madreporacea

madrilenian

maenadically

magistral

magnetify

magnetize

mahzors

maieutics

mailboxes

maintains

makership

mallophagan

mamelukes

mammalogy

mammalogical

mammutidae

mandarinate

mandritta

mangonels

mannoses

mantises

marauders

margarins

marginicidal

marinates

marmatite

marshalcy

marshalman

martiloge

martinet

martingale

maskanonge

masochism

masseter

masticot

mastoids

matagory

matchless

matchmaking

matiness

matrons

mattedly

mattulla

maximize

maximus

mealworm

meaningly

measurer

medianity

medicamental

medusalike

meetness

megalensian

melampus

melanoderm

melanous

melopiano

membranelle

membranula

meningioma

menoplania

menoxenia

mentery

mephitinae

mercantile

merciment

mercurial

merganser

meridional

merocrine

merrily

mesentera

mesitite

mesocranic

mesohepar

mesomeric

mesosoma

metaborate

metallised

metamery

metanomen

metaplasm

metatheria

metergram

methanolic

metregram

metrized

mezuzah

myatonic

mycoplana

microbrachia

microbus

micromelic

microseme

microzoa

midparent

midpoint

miffiness

milkless

millimole

milwaukee

mimickers

mineraloid

minious

minitant

mynpacht

minuses

minuter

myosis

myrabolam

miracidium

myrmicidae

mirrored

misatoned

misbiassed

misbinding

miscipher

miscopy

miscredit

misdirect

miseducate

misenus

misguggle

mislearned

mismanager

mismoved

misogallic

misparse

missus

mistcoat

mistful

miswired

miswish

mythos

mitoses

mittens

mniaceous

mobproof

moderates

modiation

modulant

modumite

molecular

mollient

mollifies

moneyman

monetise

mongolian

monilioid

monocline

monogerm

monokini

monomark

montaging

montanin

montero

moodiest

moosebird

moralism

morceaux

morfrey

morocota

morphemic

morpion

moshavim

moslemin

motors

moulten

moults

mourns

mousier

moussaka

mouther

mowstead

mozarabian

mozetta

muckment

mucusin

mufflers

muyusa

muktatma

mullers

multiped

multum

mummify

munnion

muradiyah

muraenoid

murillo

murkly

murrey

murzim

muscicole

mushru

muskish

musths

mutagens

mutedly

mutillid

nahuatlecan

nayword

naloxone

nanocurie

napellus

nappiest

nastily

natuary

nauticals

necrophil

necturidae

needlemaking

negotiated

negroism

neoblastic

neoclassic

neonomian

neossine

neotype

nervism

nesters

nestling

netheist

neurergic

neurofil

neuronal

newsful

newstand

nextly

nidulus

niellist

niggardize

niggertoe

nighters

nightfish

nihilist

nimious

ninepins

nitrolic

nivellate

nodality

nonaccepted

nonamendable

nonanarchic

nonaphasic

nonblockaded

nonblooded

nonbreakable

nonbreeder

noncasual

noncausal

nonchokable

nonciteable

noncredence

nondairy

nondeist

nonfebrile

nonhuman

nonplacet

nonpliable

nonsabbatic

nonsaleable

nonserif

nonspecie

nonteachable

norites

nosairian

nosebleeds

nosohaemia

nostalgic

notating

notchboard

notional

noveldom

novellas

nucleates

nugacity

nullism

numeracy

nutates

nutty

oarfishes

obligatos

obligatum

obliques

obouracy

obscenely

obsidional

obtests

obtuser

occultate

oceanarium

octachordal

octactiniae

octaeterid

octameter

octodecimal

odontalgic

oecumenian

oestrin

offhandedly

offishly

ogrisms

oilheating

oiltight

okruzi

oligochaete

olonets

omittance

omitter

omniarchs

omophagy

onymatic

oniscus

onomancy

oogametes

opaquing

operagoer

operatical

operetta

ophiurid

opticly

orality

orbitelar

orchestic

ordures

oreamnos

orients

ornament

orphancy

orthose

ortygan

oscillated

osculated

osnappar

osseins

ossifier

osteitic

osteogen

ostracode

otorrhea

outbaking

outbragged

outcept

outcook

outcut

outfawn

outflank

outflue

outgambled

outhired

outkill

outlier

outlined

outoffice

outpaces

outpeer

outray

outrank

outrate

outreached

outscape

outset

outshake

outslid

outsped

outtalk

outtear

ovately

overably

overblanch

overboard

overbore

overbow

overcapable

overchased

overcup

overdazed

overdried

overfilm

overflog

overgamble

overlash

overlave

overleer

overline

overmind

overneat

overrim

overtoe

overwake

overweak

ovulated

owelty

oxammite

oxcarts

oxyacids

oxydasic

oxygon

oxtails

padcluoth

paedology

paedological

paillette

paintably

pairwise

pakistani

palaeograph

palaeophile

palagonite

palatally

palmipedes

palpitate

palpless

paludrine

panathenaean

panderers

pandurate

panhandling

pansmith

pantoum

papality

papyrian

papists

parablepsia

parachaplain

parachutic

paracmasis

paracress

paradeless

parageusic

paralleler

paramecium

paranuclei

pararctalia

parentate

parleyer

parousia

parricidial

pasteur

pastiches

pasture

patellula

pathless

patinize

pauraque

paviors

pavisor

paxillate

peculium

pedagogues

pedantize

pederastic

pediadontic

pediculated

pediculati

pediculicide

pedicures

peirastic

pelargonic

pelleting

penalises

penchants

penduline

penlites

penorcon

penutian

peoplish

perborate

percents

perforata

performed

periauger

peridermic

perijove

perilune

peripherad

perisome

permits

peroxided

perscent

personage

personal

perturb

pesthole

pestify

pettiagua

petunse

phagocyte

phalangitic

phalanxes

phantasmag

pharisees

phasiron

phenacetine

phenetole

phenixes

phenocoll

phenolated

phenoxide

phycitidae

physicked

phlebitis

phlogisma

pholidota

phoronida

phosphide

photechy

photism

photoeng

photogen

photogs

photonic

photopia

phrynidae

phulwara

pianisms

pianistic

piblokto

picarooned

pictural

pigeonite

piggybacks

pigroot

pilaster

pililloo

pilloried

pilotman

pinecones

piniform

pintails

piperidin

pipkinet

piquiere

pyralidan

pyralis

pyraloid

pyrexic

pyridine

pyritic

pyrogen

pishogue

pisidium

pistaches

pistachio

pistoled

pitcairnia

pitiedly

pitying

pitless

pitmaking

pituite

pixilated

placentoma

placodermal

placodont

plagioclase

playgirl

playoffs

plaister

plaiters

playward

plantlet

plantlike

plasmagenic

plastered

plastids

plateaux

platinoid

plebianism

plebiscite

plecotinae

plenties

plenums

pleuroid

pliskies

plosion

plouky

plumbagine

plumcot

plumdamas

plummet

plummy

plumule

plunging

pluries

plushes

plussage

pneograph

pocketing

podicipedidae

podsols

podzolic

poetito

poetized

poitrail

pokomoo

polyamide

polygalic

pollenate

pollux

pompilidae

pondokkie

ponying

poorish

poplitic

popply

poriferal

portland

poseuse

postdate

postfact

posting

postpaid

potbellied

potentee

pothook

potoos

potpies

pouchlike

pounding

pouring

powders

practicing

pratty

preachings

preacness

preadapts

preatomic

prebenefit

precambrian

precents

prechordal

precising

preclaimer

precombine

preconfer

predative

predawns

predeceases

predicates

preeligible

preeners

preentail

prefeudalic

preformed

prelithic

preludes

preludio

premorbid

prenaris

prenomen

preobtain

preofficial

preordain

prepacking

prepays

prerefined

prereject

prerelate

presley

presser

presteel

pretonic

prevalue

prevent

priapus

priestal

priggess

primary

primeur

primitiae

princeps

princify

printer

prionine

prismy

prisoned

prittle

privant

proavis

probings

problems

procaines

proclive

procured

prodroma

producer

profaning

profiles

profundae

profuse

progamete

progeny

proleague

promisee

pronging

prononce

proofing

propjet

proreader

prorebate

prorecall

proscenia

prosier

prosodal

prosodiac

prothmia

proximad

proxime

pseudaphia

psychon

pteridoid

ptomainic

publicly

pucksey

puddingy

pueblito

puerpera

pugilant

pulicose

pullalue

pumpkin

punaluan

punctate

pupilate

purbeckian

pureayn

pursue

purty

pussy

putchuk

putois

putter

quadplex

quadrangle

qualmish

quarry

quarter

quartin

quassin

quatrin

quebrith

quemeful

querent

quesited

quicklime

quietened

quillais

quillon

quinyie

quinnet

quintar

quints

quippu

quittal

quomodo

racemisms

rachitism

rackettail

rackingly

rackwork

radarscope

radiocarbon

radiolitic

radioteria

raillery

raincoats

ramellose

rammermen

randannite

rangeless

rankwise

ransomable

rasters

rastling

ratatats

ratchety

ratheripe

rattails

raughty

ravelers

ravelins

ravelling

ravisher

reaccelerated

reaccount

reaccusing

reappliance

reapplier

rearguing

rearising

reattaches

reattired

rebozos

receptor

recidivated

reciprocal

recommendee

recompete

recompiled

reconclude

recreating

rectrices

redamation

redeclining

redeploy

redesigns

redespise

redoubler

redounds

redrying

redrives

reduzate

reeffishes

reenclosed

reevokes

reexhibit

referring

refinancing

refinery

reflation

reflexing

refought

refugium

refuting

regalement

regalness

regimented

regionals

regorging

regraduate

regressed

regroup

regrown

rehammers

reimpart

reimpose

reinjure

reinvoice

reissued

relacquer

relatival

releasably

remedying

remeditate

remigrated

renegating

renovate

reophore

repaginated

repower

repress

reprint

reprobate

repugns

repulsed

reputable

reradiates

reschedule

rescous

researcher

resecting

resents

reserval

resewing

reshipped

reshoeing

reshoot

reshuffle

residencer

residues

resimmer

resinfiable

resizer

resolved

respicing

responded

restaging

restocked

restore

restow

resumes

resurfaced

retaught

retemper

retinite

retinker

retinula

retiring

retotaled

retreatal

retrenched

retroact

retrocecal

revacating

reversal

revibrate

revives

rewearing

rewound

rewraps

rhabdosome

rhagadiform

rhyton

ricinulei

ricochets

ridicules

rigescent

rimeless

rindless

ringnecks

riptides

rituals

riverbank

riverine

riverman

roadblocks

roadster

rocketeer

rockiest

rodentian

romagnole

rondelier

roofward

rookeried

roomful

roommate

rootages

rootier

rooving

rosery

rosillo

rosiny

roubouh

rouping

ruction

ruddiest

rufous

rulings

rumpadder

runkles

runneth

runtime

rutelian

saccharifier

saddleless

sagebrush

sagginess

saintly

saintlike

salacious

salampore

salariats

saltery

saltmaker

salvifics

samolus

samshus

sanitised

santirs

santols

sanukite

sapiencies

sapindales

sapogenin

sarcolemma

sarcophagic

sarothra

sassagum

sateless

satiating

satinlike

satirise

satrapy

satrapical

saucepot

savory

saxifragaceae

scabiophobia

scannings

scaphites

scapiform

scapolite

scarfskin

scawtite

scenarize

scentful

schediastic

schematics

schiavoni

schnapper

scholarian

scholium

schoolmaam

schoolman

schorly

sciaticky

scincoidian

scintler

scivvy

scleranth

sclerosed

scobiform

scoreboard

scorepads

scotchmen

scotopic

scrapbook

scratchman

screwier

scrieves

scriggly

scripto

scrublike

scrummage

sculpted

scumbling

scuppet

scurril

scuttle

seafowls

seamster

searness

seashells

seatless

secours

secreting

secundum

seemlily

seignorage

seignoral

seiyukai

selective

selenates

semaphore

sementera

semibarbaric

semichemical

semicircled

semiclose

semicrome

semicurl

semideltaic

semihoral

semimatt

seminifical

semipupa

semishaft

semitone

senilism

sennits

sensoria

septicemia

septleva

serfship

serioso

serjeancy

serphoidea

sertion

sertule

services

sescuple

session

setdown

setout

settles

setups

setwise

sextole

shadberry

shadowing

shaggymane

shaivism

shakedown

shakeout

shakeups

shaktism

shamoys

shanksman

sharezer

sharpens

shattered

shawllike

sheeneys

sheepwalk

shellshake

shelterage

sheraton

sheroot

shibboleth

shieldfern

shipbuild

shipcraft

shipyard

shiplaps

shipplane

shivaism

shivers

shlimazl

shochetim

shoofly

shooter

shopboy

shortcake

shovels

shredders

shrewder

shrieking

shrives

syagush

sicklily

sycoceric

sycones

sienites

sightly

sigillarid

significian

signiori

silicifies

silicons

syllabaria

sillery

sylviid

simulate

synacmy

sinapism

synapte

sincipita

syndicate

synergic

sinewing

syntagma

syphered

siphons

syrens

siskins

sissone

sixtine

skatings

skewers

skiddiest

skimpier

skirret

skirter

skittled

skyugle

sklents

skully

slangrell

slaverer

sleepyhead

sleepry

slenderer

slyest

slighty

slithered

slumps

smackeroo

smashboard

smellful

smirching

smirking

smitten

smokelike

smokers

smoorich

snarlish

sneerful

sneeshing

sniffily

snipelike

snipers

snivels

snobbery

snorker

snubbers

snuffkin

sobersided

socialism

sociogram

socrates

sodiums

sodomite

softboard

soybeans

solidism

solunar

somites

sompner

sonship

soonly

soother

sordello

sorehawk

sotols

soughing

soundheaded

soupfin

soupon

sourball

sources

sourdre

spacewalked

spanceling

sparganiaceae

sparily

sparkler

spavins

speakablies

specificated

speckledy

speltoid

spenders

speranza

sperling

sphecius

sphendone

sphenion

spherula

sphygmic

spiciest

spiflicate

spyhole

spikehole

spillages

spinelet

spinelike

spiracula

spirifer

spirited

spitish

spizella

splashy

splining

spoliaria

spondaics

spooler

sporangia

spouted

springald

springle

sprucer

sprugs

spunks

spurdog

squares

squaws

squibbing

squinched

squint

srikanth

stabilised

stagecraft

staggerer

stagiary

stagiest

stagskin

staynil

stalemated

stalklet

stalklike

staminode

standards

standers

stanzas

starers

starfish

starling

statize

status

steadiers

steamers

stearyl

steckling

stemming

steppes

steptoe

stereome

stiacciato

stickboat

styles

stillman

stinkball

stipels

stockier

stockmen

stokavci

stomatic

stonehand

stonier

stooges

stools

stoping

stoves

stower

straddleback

strains

strangled

strawen

straws

streetage

stress

striatal

striding

striker

strobile

strolld

stroth

struv

stubbles

stuccos

stupes

sturdied

subadjacent

subadult

subalgebraic

subcosta

subcranial

subdeliria

subducing

subduple

subgallate

subhooked

subjectable

subjugable

sublethal

sublimes

subloral

subnets

subpool

subradiate

subrepand

subscience

subsellia

subsmile

subsume

subucula

succour

sucrose

sufficience

suffocated

suffragial

sugarloaf

sulfamine

sulfonal

sumbulic

sumerian

summula

sunbeamy

sunders

sunfast

sunnier

sunward

superceded

supracaecal

surcharge

surely

surmit

suttas

suzanne

swampine

swankier

swanmark

swashing

swattle

sweatier

sweepdom

sweepier

sweetened

sweetman

sweptback

swifter

swimmer

swinebread

swingman

swipple

swirls

swythe

swollen

tabernacles

tablature

taborets

tabourer

tabourin

tabulary

tagliarini

tailcoats

tailgating

tailory

tailspin

takhtadjy

tallying

talocalcaneal

talpetate

tamehearted

tanchelmian

tangents

tanzanian

tariffize

tarquin

tastably

tatarize

tattooed

taurus

tautens

taxeopod

taxying

taxless

teagardeny

teardown

tectricial

teethers

telegrams

telephone

telescope

tellinoid

telluric

teloptic

temperance

temperer

templum

temporal

tensely

tenurial

teroxide

terranes

tersion

tessaradecad

tetanize

tetanus

tetrachical

tetragon

tetrahedra

tetramer

tetramin

tetrapoda

tetryl

tettigidae

tewsome

thalidomide

thaneship

thegither

theists

theomagics

theorum

therapsid

therebeside

therefore

thermels

thickening

thievish

thinghood

thynnidae

thiobacilli

thiolactic

thiophene

thioxene

thirdendeal

thirty

thondraki

thorny

thrawing

threatened

thrifts

thriver

thumbnail

tyburn

tidemarks

tiderips

tiefenthal

tigerism

tightish

tilesherd

timbrelled

tinkerer

tinselled

tipburn

typefaces

tipless

tyramin

tiredest

tiremaker

tiresias

tironian

toasty

tobaccoism

tobaccosim

toboggans

toddymen

toffyman

toilets

toitish

tolerated

toluids

toluole

tonguer

toomly

tooting

tornesi

toroids

torqued

torrefied

torsalo

torteau

tosily

totty

totuava

tousy

tovariaceae

towards

towers

townman

towser

toxicaemia

toxicon

toxotae

trachyte

traction

trailboard

tramells

tramyard

trammelhead

trampcock

trampdom

tranceful

transbay

transect

transom

travelog

trecento

treeship

trembling

tremellaceae

trepangs

triadist

tricarbon

trickful

trierarch

triones

triplet

triply

tristam

trisula

tritor

trizoic

trochlear

troller

tropics

troppo

troths

trotol

trounced

trudging

truism

trutta

tsarevna

tsktsk

tsunamic

tubeform

tubules

tularemia

tummies

tumoral

tuneably

tunnland

tuppence

turgite

turkey

turnoff

turtled

tussled

twangler

tweenies

tweesht

twisted

twitchel

twoling

uigurian

uitotan

uitspan

umbratical

umpirer

unaborted

unabsurd

unabundance

unadmirable

unadopted

unalerted

unaltered

unamerceable

unangry

unappended

unattacked

unavailable

unbalconied

unbarreled

unbarricaded

unbedraggled

unbeguiled

unbeholden

unbeloved

unbendably

unbenight

unboring

unbounded

unboxes

unbragging

unbreast

unbreeches

unbridgeable

unbroiled

unbroken

unbullied

uncenter

uncentre

unchaining

unchiseled

uncialize

unclever

unclipped

unclubby

uncoaxial

uncombined

uncrest

uncrisp

unculted

uncupped

undamnified

undeducible

undeferred

undelight

underbalance

underbodice

underboil

underchime

underclub

underdone

underfiend

underfire

undergird

underkind

underlay

underseam

undertide

undertied

undowned

undraws

undress

unduloid

unelectable

unenchant

unequaled

unevocable

unfasten

unfauceted

unferried

unfevered

unfiducial

unfinish

unfoaming

unfogging

unfooted

unformal

unfreeze

unfriended

unfrugal

unfuelled

unfundable

ungambling

ungloved

unhaggling

unhorse

unhumbled

unhushed

uniflow

unimbibing

unindexed

unironed

unitive

unjapanned

unkempt

unkindred

unlaving

unleveled

unlighted

unlinks

unloathed

unloyal

unmackly

unmaligned

unmanly

unmanlike

unmanner

unmantle

unmatchable

unmental

unmetred

unminted

unmiracled

unmiry

unmiter

unmitre

unmodified

unmould

unmown

unpalpable

unpapered

unpasted

unpegging

unpitched

unplashed

unprimed

unprince

unprop

unramified

unrandom

unrankled

unreceding

unrecent

unrelated

unrisen

unrobing

unrowed

unsaccharic

unsayable

unscaling

unseven

unshamefaced

unshelled

unshocked

unshore

unsicker

unsmeared

unsnatch

unsolar

unstanch

unstate

unsteep

unsunk

untacking

untaste

untaxable

untermed

unthewed

unthreaded

untidier

untipt

untoiled

unturf

unugly

unuseable

unvalued

unvendable

unwaking

unwarbled

unwarned

unwearied

unwhite

unwilled

unwove

upbuoy

upcover

upcrowd

upcurled

uperize

upgrow

uppluck

upsets

upspew

upspread

upstare

uptears

urbanest

urbanized

urceolate

ureteral

urobilin

urohyal

urologic

urticaria

useless

uspanteca

utensil

uucpnet

uxorial

vaccinist

vacuolate

vacuums

valbellite

vallums

vamosing

vampyre

variety

varnished

vasovagal

vedettes

vegetism

vehemency

velatura

veliform

velocipeded

veloute

venantes

venation

venditate

venially

ventifact

ventose

verbalize

verbenone

verbomania

verdancies

verdicts

vergery

verriere

vetchling

veuglaire

vibists

victrola

vietminh

vilipended

villancico

villanella

vinagron

vincular

vinous

violins

virbius

virgater

visammin

visionic

vitameric

vitaminic

vitasti

vitrailed

vitrics

vivency

vivifier

voyaging

volleyed

volvell

vortical

vouchsafe

vulgars

vulpinae

wachuset

walycoat

wamefuls

wanrest

wanters

warpers

warragals

washdays

washings

washway

wastemen

wasterie

wastern

watchdogs

watergate

wattles

waveshape

weariest

webworn

wedgewise

wednesday

weelfaured

weeniest

wehrlite

wellyard

wellmaker

wellring

whaleries

wheatears

wheelabrate

wheelsman

whelpish

whenever

wherves

whichway

whinnier

whipray

whirley

whirling

whiskey

whisking

whistled

whitebill

whitfinch

wholely

wholesale

whooper

whorish

whumps

widdendream

widewhere

wigglers

wigwagger

wildsome

willets

windfallen

windfalls

windigos

wiredancer

wirehaired

wiselier

wissing

wistaria

wistit

witchlike

witchweed

witcraft

withnay

witloof

witwall

wizards

woeness

wolframic

wollomai

woodblock

woolfell

wordably

worldman

wormgear

worset

wouhleche

wounder

wrathing

wrawler

wreakers

wreathy

wriggles

writing

wronger

xylose

zamouse

zaptiahs

zaratite

zesting

zeuxian

zincifies

zinkify

zithern

zoogleas

zorgite

如果想把符合条件的词保存到一个文件 result.txt 里的话，那么：

In [19]:

```
def sum_of_word(word):
sum = 0
for char in word:
sum += ord(char) - 96
return sum
with open('results.txt', 'w') as result:
```

```
with open('words_alpha.txt', 'r') as file:

for word in file.readlines():

if sum_of_word(word.strip()) == 100:

result.write(word)
```

竟然这么简单就搞定了？！

这 12 行的代码，在几秒钟内从接近 370,101 个英文单词中找到 3,771 个如此计算等于 100 的词

汇。

喝着咖啡翻一翻 result.txt ，很快就找到了那些可以用来做反例格外恰当的词汇。

真无法想象当年的自己若是不懂编程的话现在会是什么样子……

总结

这一章我们介绍了文本文件的基本操作：

打开文件，直接用内建函数， open() ，基本模式有 r 和 w ；

删除文件，得调用 os 模块，使用 os.remove() ，删除文件前最好确认文

件确实存在……

读写文件分别有

file.read() 、 file.write() 、 file.readline() 、 file.readlines() 、

可以用 with 把相关操作都放入同一个语句块……

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 anUndefinedVariable 赋值的情况下，直接调用

这个变量，比如，print(anUndefinedVariable)，那就会报错：NameError: name 'aNewVariable' is not defined 。

In [1]:

print(an_undefined_variable)

NameError Traceback (most recent call la

st)

in

----> 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儿，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，就好像找黑暗中拼图

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，竟然是绝大多数人并不懂的，也因此吃了很多很多亏。

充满了过早引用的知识讲解，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…… 再通读书籍之

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:

pyhton

>>> 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 fact

or

... 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儿的缺点。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：

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：

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。

学些代码的过程中，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，这不只是编程的事儿，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↑

官方教程：**The Python Tutorial**

虽然，第一部分总计七章关于编程内容的编排是非常特别且相当有效的：

它并没有像其它教程那样，从“Hello world!”入手；

它也没有使用与市面上所有编程教材一样的内容先后顺序；

它一上来就让你明白了程序的灵魂：布林运算；

它很快就让你明白有意义的程序其实只有两个核心构成：运算和流程控制；

它让你很快理解函数从另外一个角度看只不过是“程序员作为用户所使用的产

品”；

它让你重点掌握了最初级却最重要的数据类型，字符串；

它让你从容器的角度了解了 Python 中绝大多数“重要的数据类型”；

最重要的是，它不承诺你“速成”，但承诺“领你入门”…… 显然，它做到了。

但是，第一部分的内容核心目标是让你“脱盲” —— 它的作用还做不到让你“已然学会编程”，它更多

是让你从此开始有能力去阅读更多的重要资源，比如，官方的教程和参考。第一部分的内容更像地

图上的“图例”，而不是地图本身。

第一部分反复读过之后，最重要的结果就是：

现在你有能力自己查询官方文档了……

起码，在此之后，再去阅读 The Python Tutorial，不那么费力了，最起

码，可以靠自己理解绝大多

数内容……

在继续阅读本书内容的同时，有空就要反复翻 The Python Tutorial。

官方文档中最重要的链接

Python　也许是目前所有编程语言中在文档建设（Documenting）方面做得最好的（好像真的不需要

在这句话后面加上“之一”）。Python 社区为了建设完善的文档，甚至有专门的文档制作工具 ——

得益于 Python 社区从一开始就非常重视文档规范 —— Sphinx。你在网络上最经常看到的计算机类

文档，很可能都在这个网站上：Read the Docs……

Python 的官方文档网址是：

https://docs.python.org/3/

其中对初学者最重要的两个链接是：

Tutorial: https://docs.python.org/3/tutorial/index.html

Library Reference: https://docs.python.org/3/library/index.html

理论上来讲，只要有了基础的概念，自己反复阅读官方的　The　Python Tutorial 是最好的，没什么入

门书籍比它更好 —— 因为它的作者是 Python 的作者，那个被称为“善意独裁者”的 Guido van

Rossum。

此人很帅，但更帅的是他的车牌（摘自 Guido van Rossume 的个人主页）：

为什么一定要阅读官方文档

买一两本 Python 教程是不可能完整掌握 Python 的 —— 其实，这句话里的 Python 替换成任何一种

语言也是一样的。

教程和官方文档的各种属性是非常不一样的，比如，针对读者群，组织方式，语言表达…… 最不一

样的地方在“全面性”。任何一本单独的教程，都不可能像官方文档那样全面。各种单独教程的优势

在于，它们更多地针对初学者、入门者设计，但与此同时，在全面性、深入性上做了妥协。

比如，在当前这本书里，不会涉及 Bytes Object —— 并非只有我一个人这么做，著名的 Python 教

程《Think Python: How to Think Like a Computer Scientist》、《Dive into Python》等等都没有涉

及 Bytes Object 这个话题。

由于官方文档实际上没办法对入门者、初学者过分友好 —— 毕竟，全面、权威、准确才是它更应该

做到的事情 —— 所以，很多人在刚开始的时候求助于各类非官方的教材、教程。原本应该是入门以

后就理应“只读官方文档”，或者“第一查询对象只能是官方文档”，但，在很多人那里竟然变成了“从

一开始到最后都在回避官方文档（或者说‘最专业的说明文字’），这就不好了，真的很吃亏，且自己

都无法知道自己究竟吃了多少亏 —— 总以为自己已经学完了，但实际上从一开始就一点都不全面。

请牢记且遵守这个原则：

第一查询对象只能是官方文档。

所以，当我用 Google 查询的时候，经常使用这样的格式：

site:python.org

有时甚至会指定在那个目录里搜索：

bytes site:python.org/3/library，你试试这个连接：bytes

site:python.org/3/library

这个原则对任何语言都适用。将来你在学习任何新软件包（库）、语言更新后的新特性、甚至另外

一个新语言的时候，都要这么做。所谓的超强自学能力，基本上就是由一些类似这样的小习惯和另

外一些特别基础的方法构成的强大能力。

将官方文档拉回本地

把 The Python Tutorial 拉回本地阅读，可能更为方便，尤其是可以用 Sphinx 重新制作之后，页面

左侧可以总是显示完整的目录：

也可以把这个教程转换成 epub 格式，以便在移动设备上阅读；甚至可以把这些个页面的 .rst 源

文件转换成 .ipynb 文件，以便用 Jupyter Lab 浏览时可以直接执行其中的代码……

注意

此页的 Code Cell 中都是可执行的 bash 命令……

在此页执行它们对你来说是没意义的 —— 因为它们的执行结果在服务器上；这其中的命令，应该在

你本地计算机上的 Terminal 中执行，你才能在本地计算机上获取结果。

安装 **git**

In [ ]:

%%bash

which git

git --version

 # 没有的话就执行以下命令安装 git

 # conda install -c anaconda git

下载源文件

The Python Tutorial 的源文件位置在：

https://github.com/python/cpython/tree/master/Doc/tutorial

repo 地址是：

https://github.com/python/cpython.git

使用 git 将 repo 下载到 ~/Download/ 目录：

In [ ]:

%%bash

cd ~/Downloads

 # 总计 241 M，所以需要一点时间

git clone https://github.com/python/cpython.git

cd cpython/Doc/tutorial

ls

安装 **rst2ipynb**

In [ ]:

%%bash

 # rst2ipynb needs pandoc and notedown...

which pandoc

which notedown

没有这两样东西的话，需要执行下面两行进行安装 ……

 # conda install -c conda-forge pandoc

 # conda install -c conda-forge notedown

 # install rst2ipynb

cd ~/Downloads

git clone https://github.com/nthiery/rst-to-ipynb.git

cd rst-to-ipynb

pip install .

which rst2ipynb

批量转换 **rst** 至 **ipynb**

这个 rst2ipynb 的程序有点讨厌，一次只能处理一个文件…… 下面是一个 bash 程序，其实将来学

起来也不难，看着跟 Python 差不多…… 下面的代码执行过后会出现很多“警告” —— 没关系，文件

会正常转换的。

In [ ]:

%%bash

cd ~/Downloads/cpython/Doc/tutorial/

for f in *.rst

do

rst2ipynb $f -o " **${**f/%.rst/.ipynb**}**"

done

mkdir ipynbs

mv *.ipynb ipynbs/

如此这般，你就把 rst 文件都转换成 ipynb 文件，保存在

~/Downloads/cpython/Doc/tutorial/ipynbs/ 之中了。随便把它挪到你喜欢的什么地

方。用本地的 Jupyterlab 浏览，或者用 Nteract App 浏览。

如果以后你经常需要批量转换某个目录内的 rst 文件，那就把 bash function 放在

~/.bash_profile 文件里，在最后面追加以下代码：

function rsti {

for f in *.rst

do

rst2ipynb $f -o " **${**f/%.rst/.ipynb**}**"

done

}

而后在 Terminal 里执行一遍：

source ~/.bash_profile

而后，在有 .rst 文件的目录下输入 rsti 执行即可……

用 **Sphinx** 生成 **html/epub** 版本

In [ ]:

%%bash

which sphinx-quickstart

没有的话就执行下一行：

 # conda install -c anaconda sphinx

sphinx-quickstart --version

sphinx-quickstart --help

生成 html 版本和 epub 版本：

In [ ]:

%%bash

cd ~/Downloads/cpython/Doc/tutorial/

sphinx-quickstart -q output --sep -p 'The Python Tutorial' -a 'Guido van

Rossum' -r '1.0' -v '1.0' -l 'en' --suffix '.rst' --master 'index' --ex

t-autodoc --ext-doctest --ext-intersphinx --ext-todo --ext-coverage --ex

t-imgmath --ext-mathjax --ext-ifconfig --ext-viewcode --makefile --no-ba

tchfile --no-use-make-mode

cp -f *.rst output/source/

cd output

make html

make epub

 # 生成的 html 版本应该在 output/build/html 目录下；

 # 生成的 epub 版本应该在 output/build/epub 目录下。

*# sphinx-quickstart -q output *

*# --sep *

*# -p 'The Python Tutorial' *

*# -a 'Guido van Rossum' *

 # -v '1.0'

*# -r '1.0' *

*# -l 'en' *

*# --suffix '.rst' *

*# --master 'index' *

*# --ext-autodoc *

*# --ext-doctest *

*# --ext-intersphinx *

*# --ext-todo *

*# --ext-coverage *

*# --ext-imgmath *

*# --ext-mathjax *

*# --ext-ifconfig *

*# --ext-viewcode *

*# --makefile *

*# --no-batchfile *

 # --no-use-make-mode

用 Sphinx 这样生成的版本，支持本地目录内搜索，也确实比在网站上看更方便一点：

下载已经转换好的版本

万一有什么情况中间出错又搞不定的话，就直接下载已经转换好的版本：

In [ ]:

%%bash

cd ~/Downloads

git clone https://github.com/xiaolai/the-python-tutorial-in-other-format

s.git

完整的 **Python Doc** 制作

其实，Python 的整个文档，已经是做好了制作文件的文档库：

cpython/Doc/Makefile

只不过，将所有文件编译到一个 epub 里，在 iPad 之类的移动设备上打开有点费劲 —— 在我的设

备上显示有 7701 页，翻页都要顿一顿……

想要使用这个官方的 Makefile 的话，首先确认自己已经安装过 Sphinx，

其次需要补充安装一个

包：

pip install blurb

而后在 Terminal 中转到 Doc 所在目录，执行以下命令：

make html

make epub

笨拙与耐心

自学的过程，实际上需要拆解为以下四个阶段，虽然它们之间常常有部分重叠：

学

练

用

造

只要识字，就会忍不住阅读；只要感觉上有“值得学”的，就会忍不住去学 —— 事实上每个人时时刻

刻都在学习，只不过，学习的目标选择与学习的方式以及效率均不相同而已。

绝大多数人从未区分过这些个阶段，也从未研究过这几个阶段分别应该如何对待。这就解释了为什

么那么多人虽然总是忍不住阅读，总是忍不住学习，但，终其一生都没有真正掌握过像样的技

能……　因为他们在第一个阶段就出错，到了第二个阶段就放弃，第三个阶段是直接跳进去的，总

是“对付着用”，至于第四个阶段么，想都没想过……

第一部分的内容，基本用来展示“学”的过程。学，就需要重复，甚至很多次重复，尤其是在面对充

满了“过早引用”现象的知识结构的时候。

反复学，最锻炼的是“归纳整理”的能力。而且，最有意思的，这在大多数情况下还是自动发生的

——　只要你不断重复，你的大脑会在不自主之间把那些已经掌握的知识点与当前尚未掌握的知识点

区分开来，前者处理起来轻松容易，甚至可以跳过；后者需要投入更多的注意力去仔细处理…… 在

这个过程中，绝大多数的归纳整理工作自动完成了。最后再加上一点“刻意的、收尾性的归纳总结整

理工作” —— 大功告成。

绝大多数人总是希望自己一遍就能学会 —— 于是，注定了失败；而面对注定的失败，却并不知道那

与天分、智商全无关系，因为谁都是那样的；于是，默默认定自己没有天分，甚至怀疑自己的智

商；于是，默默地离开，希望换个地方验证自己的天分与智商；于是，再次面临注定的失败；于

是，一而再再而三地被“证明”为天分不够智商不够…… 于是，就变成了那条狗：

在一项心理学实验中，狗是被测试对象。把狗关进栅栏里；然而那栅栏并不算太

高，原本狗一使劲就能跳过去。但，狗带着电项圈，只要它被发现有要跳过栅栏的

企图，它就会被电击…… 几次尝试之后，狗就放弃了跳出栅栏 —— 虽然它其实能

跳过去。更为惊人的结果是，随后把这条狗关进很低的栅栏，甚至是它都不用跳直

接就可以跨过去的栅栏，它也老老实实地呆在栅栏里。

自学是门手艺，编程很大程度上也是一门手艺，掌握它在绝大多数情况下与天分智商全无关系 ——

很多人是在十来岁的时候学会编程的基本技能的。所有的手艺，最基本特征就是：

主要靠时间

这就跟你看人们的车技一样，二十年安全驾龄和刚上路的肯定不一样，但，这事儿跟天分、智商真

的没什么关系……

到了第二部分，我们终于进入“用”的阶段 —— 嗯？“练”怎么跳过去了？没有，我们的策略是，以用

带练 —— 在不断应用的过程中带动刻意练习。

练和学，是多少有些重合部分的。比如，你可能反复“学”了第一部分两三遍，而后就进入了第二部

分，开始接触“用”，在“用”的过程中，只要有空，或者只要有需求，你就可能回去“复习”一遍第一部

分的内容……

无论之前的“学”，重复了多少遍，一旦开始练，就注定体会到各式各样的笨拙。一会儿漏写了一个

引号或者一个括号，一会儿不小心使用了非英文字符的全角符号，要么就是发现自己犯的错误愚蠢

且反复发生，比如，语句块起始语句末尾忘了写冒号……

这再正常不过了。

每次自学什么新东西的时候，你就把自己想象成“再次出生的婴儿” —— 起始每次自学，的的确确都

是重生。一旦掌握了一项新的技能，你就不再是从前的那个自己，你是另外一个人了。

看着婴儿蹒跚学步，的确笨拙，但谁会觉得它不可爱呢？

同样的道理，刚开始用一个技能的时候，笨拙其实就是可爱 —— 只不过这时候旁人不再这么觉得而

已了，只不过因为你披着一张成年人的皮。然而，你的大脑中正在学习的那一部分，和新生婴儿的

大脑没有任何区别。

在第一部分的时候，“练”的必要其实并不大…… 甚至，因为这是“过早引用”太多的知识结构，所

以，急于练习反倒会有副作用。由于对自己所面对的复杂知识结构（就是过早引用太多的知识结

构）认识不够，没有提前的应对策略，所以，他们根据原本的习惯，边学边练，学不明白当然就练

不明白，于是，走着走着就挫败感太强，于是，就自然而然地放弃了…… 而且，弄不好，越练越容

易出现“不必要的挫败感”。

一切“主要靠时间”的活动都一样，都需要在从事之前认真做“心理建设”。通常情况下，读一本教程，

上个学习班，就“会”了 —— 几乎肯定是错觉或者幻觉。

首先要明白，这肯定是个比“天真的想象”要长得多的过程。

其次要明白，并且要越来越自然地明白，哪儿哪儿都需要很多重复。读，要读

很多遍；练，要练很多遍；做，要做很多遍……

有了这样的心理建设，就相对更容易保持耐心。

人们缺乏耐心的常见根源就是“之前根本就没准备花那么多时间和精力”，所以当然很容易超出“时间

和精力的预算”，当然相对更容易焦虑 —— 就好像没多少本钱的人做生意常常更容易失败一样。

这也解释了为什么我在写这本书的过程中，心中锁定的目标群体是大一学生和高一学生（甚至也有

初一学生）：

他们最有自学的“本钱”……

离开学校之后，绝大多数人很难再有“一看一下午”、“一练一整天”、“一玩一整夜”的本钱。又，由于

生活的压力越来越大，对“能够使用”新技能的“需求”越来越紧迫，于是，对任何一次自学的“时间精

力投资”都缩手缩脚，小里小气……

预算观念非常重要 —— 这个观念的存在与否，成熟与否，基本上决定一个人未来的盈利能力。

大多数人对此不仅不成熟，甚至干脆没有预算观念！ —— 这也是为什么绝大多数人不适合创业的最

根本原因。

不夸张地讲，未来的你只需要恪守一个原则，就很可能会因此超越 99%

的人：

绝对不做预算不够的事情。

说来惭愧，我是四十多岁之后，在创业和投资中经历了大量的失败，付了不可想像的学费之后，才

深刻理解这条看起来过分简单的原则 —— 亏得本科还是学会计毕业的呢！我的运气在于，在自学这

件事儿上，从来给出的预算都很多……

大约 1984 年，我在远在边疆的延吉市的本地青少年宫参加了一个要交 10 元学费的暑期计算机兴趣

班，老师姓金，教的是 BASIC 语言，用的机器是这样的：

它要外接上一个九寸的单色显示器，那时候还没有磁盘，所以，只要一断电，就什么都没有了……

后来上了大学，买书自学 C++，结果在一个地方被卡住了：

我写的代码死活编译不过去…… 当时的编译器错误提醒还不像今天这么完善，当时也没有什么

Google 可以随时看看网上是否早就有人遇到过跟我一样的问题…… 当时我身边根本就没有什么别

人在玩这些东西，当时的学校里的电脑还很少，需要提前申请所谓的“上机时间”，后来呢？后来就

放弃了。

当时是什么东西卡住我了呢？说来都能笑死，或者都能被气死：

if c = 1 {

...

}

习惯于写 BASIC 的我，“被习惯蒙蔽了双眼”，反复检查那么多遍，完全没看到应该写的是 c ==

1！

一晃就好几年过去，有一天在书店看到一本 C++ 教程，想起来多年前放弃的东西，就把书买回来，

在家里的电脑上重新玩了起来…… 然后就想起来问题在哪儿了，然后这事儿就被我重新捡起来

了……

学完了也就学完了，真没地儿用。没地儿用，当然就很少练。

又过了好几年，我去新东方教书。2003 年，在写词汇书的过程中，需要统计词频，C++ 倒是用不

上，用之前学过它的经验，学了一点 Python，写程序统计词频 ——《TOELF 核心词汇 21 天突

破》到今天还在销售。一个当年 10 块钱学费开始学的技能，就因为这本书，这些年给我“变现”了很

多钱。

我有很多东西都是这样，隔了好多年，才重新捡起来，而后继续前行的。

最搞笑的是弹吉他。

十五六岁的时候，父亲给我买了一把吉他，理由是：

你连唱歌都跑调，将来咋学英语啊？

然后我就开始玩。花 5 块钱上了个培训班，第一天学了一个曲子，第二天就因为打架把吉他砸坏

了，害得父亲又去给我买了一把更便宜的……

那个年代学吉他的人，第一首肯定是“Romance d'Amour”（爱的罗曼史），我当然不例外。那曲子

好听啊，好听到弹得不好也好听的地步。

然后吧，有一天，在一姑娘面前显摆，竟然没有弹错！弹完之后很得意…… 结果那姑娘一脸迷茫，

隔了两三秒钟说，“不还有第二段吗？” —— 我一脸懵蛋，“…… 啊？”

可是吧，那第二段我终究没有学……　其实也练过，但后来因为指骨摔断了，所以再后来的许多年，

弹吉他只用拨片。直到……　直到四十五岁那年。有一天，忽然想起这事儿，于是找来琴谱试了一

下，花了一会的时间顺了下来。

所以，我猜我的“自学能力强”这件事儿本身，其实只不过是我投入的预算很多造成的 —— 活到老学

到老，在我这里不是空话，所以，相对于别人，我这里只不过是预算太多、太充裕了而已。

于是，我学的时候重复得比别人多；练的时候重复得比别人多；到最后用得也相对比别人多很多

—— 这跟是否有天分或者聪明与否全然没有关系。

当然，学到的东西多了，就变得更聪明了似的。有高中学历的人通常情况下比只有小学学历的人更

聪明 —— 其实就是这个原因而已。而这个现象与天生的智商并不总是正相关。

有个现象，不自学的人不知道。

真正开始自学且不断自学之后，刚开始总是觉得时间不够用 —— 因为当时的自己和其他人没什么太

大区别。

随着时间的推移，不仅差异会出现，自我认知差异也开始越来越明显：

别人的时间都白过了，自己的时间都有产出……

到了下一个阶段，在其他人不断焦虑的情况下，自己却开始越来越淡定：

因为早已习惯了投入大量时间换取新技能……

等后来真的开始用这些技能做事，不断地做其他人因为时间白过了或者因为投入的“预算”不够而学

不会做不到的事情 —— 并且还能充分明白，这并不是自己聪明、有天分的结果；只不过是做了该做

的事情，投入了该投入的“成本”和“预算”而已……

于是，就真的能够理解下面这句话背后的深意：

人生很长，何必惊慌。

反正，这事儿跟天分与智商几乎没有任何关系。

刻意练习

在自学的过程中，总是有一些部分需要刻意练习。就好像小时候我们学习汉字，有些人总是

把“武”这个字上加上一个撇 —— 对他们来说，不去写那个不该有的撇，就是需要刻意练习的。另外

一些人倒是不在这个字上出错，但，候和侯傻傻地分不清楚（类似的例

子不计其数），那么，对他

们来说就有了另外需要刻意练习的地方……

手艺这个东西，尤其需要刻意练习。我们说，手艺么，主要是靠时间…… 这里的“时间”，准确地

讲，就是“刻意练习”的时间，而不是任何时间。

我当过很长时间的英语老师。异常苦于一件事：最有用的道理最没人听。

学英语根本就不应该那么难，学了十六年也学不明白 —— 至于嘛！然而，最简单的道理，人们就是

听不进去。他们之所以不行，就是因为从未刻意练习。学英语最简单的刻意练习就是朗读。每天朗

读一小时，一百天下来就会超越绝大多数人，也会超越自己原本可能永远跨不去的那个坎 —— 神奇

的是，朗读什么无所谓，反正现在有声书那么多…… 更神奇的是，刚开始朗读得好不好听，甚至好

不好都无所谓，反正没几天就能体会到大幅进步…… 最神奇的是，这么简单的事儿，99.99% 的人

不做 —— 你身在中国，能理解这个比例真的没有夸张。

顺带推荐一下王渊源（John Gordan）的微信公共号：

清晨朗读会

到 2019 年 2 月 21 日，王渊源同学的清晨朗读会带着大伙朗读了第 1000 天……

许多年前，资质平庸的我，一直苦恼一件事儿：

为什么自己无论干什么都笨手笨脚、差这儿差那儿的……

为什么与此同时，总是能看到另外一些人，给人感觉“一出手就是高手”呢？！

这事儿折磨了我好多年……　直到后来我当了老师，每年面前流过几万名学生之后，我才“羞耻”地反

应过来：

我花在刻意练习上的时间太少了；

并且，也没有刻意思考哪些地方我应该去刻意练习。

而那些看起来“一出手就是高手”的人，则恰恰相反，他们不仅花很多时间刻意练习，还总是刻意思

考在哪些地方尤其要刻意练习 —— 就是这一点差别造成了那么大的差距。

比如，小时候玩琴，因为手骨摔断了，于是就中断了很多刻意练习 —— 后来换成拨片之后，也习惯

不好，不做很多基础练习，只是顺着感觉“胡搞瞎搞”…… 于是，我的琴艺永远是自娱自乐也就刚刚

够用的水准，永远上不了下一个台阶。我认识的人之中，许岑同学在这方面就是我这种情况的反向

典范。

然而，我深刻地意识到，在另外一些地方，若是再“混”下去，那这辈子就别想有什么名堂了。所

以，我就下决心在必要的地方一定要刻意地练习。印象中我第一个应用这种思考模式与决心的地方

就是写书。我花了很长时间去准备第一本书，并且刻意地思考在哪些地方应该刻意地用力 —— 比

如，在取书名这件在别人眼里可能并不是很重要的事儿上，我每天都琢

磨，前后换了二十几个名

字，最终选定…… 其后每一本出版的书籍，在书名选择上我都“殚精竭虑” —— 最终的结果是，我的

第一本书就是畅销书、长销书 —— 后面的每一本都是。

对，所谓的“混”，解释很简单：

不做刻意练习的人就是在混时间。

需要刻意练习的地方，因人而异。有的人就是不可能让“武”字带把刀，不需要刻意练习，但另外一

些人不是。有些人就是朗读十分钟的效果跟别人朗读一小时的效果一样地好，但更多的人并不

是……

然而，这并不是所谓的“天分”差异，这大抵上只相当于正态分布坐标略不相同而已。每个人都一

样，都有各自必须刻意练习的地方，都有对别人来说无比容易可偏偏对自己来说就是很难的地方，

而且，在这件事儿上，大家的点虽然各不相同，但总体上需要刻意练习的部分比例都差不多 —— 虽

然说不清楚那个比例到底是多少。

比如，在学一门新的编程语言时，我常常做这样的刻意练习：

在纸上用笔写程序……

而后，看着纸上的程序，把自己的大脑当作解析器，去判断每一句执行后的结果……

反复确认之后，再在编辑器里输入这个程序 —— 用很慢的速度，确保自己的输入无误……

然后再一运行，十有八九出错 —— 要再反复检查修改很多次才能顺利执行到最后……

为什么要这么做呢？因为我发现自己一旦学另外一个语言的时候，自己的大脑就会经常把这个新的

语言与之前学过的其他语言相混淆，这很痛苦。我必须想出一个办法，让之前的也好，之后的也

罢，干脆刻在自己的脑子里，不可能相互混淆。

我相信不是所有人都有我这样的烦恼和痛苦，虽然他们在我不烦恼不痛苦的地方也可能烦恼痛

苦……

然而，于我来讲，这就是我需要刻意练习的地方 —— 这也是我刻意思考之后才找到的需要刻意练习

的地方。

你也一样。你需要刻意练习的地方，需要你自己去刻意思考 —— 你和别人不一样，没有人和你一

样，就这样。

这种事情，过去还真的是所谓“书本上学不到”的东西 —— 因为没有哪个作者能做到“遍历世上所有人

的所有特殊情况”…… 不过，互联网这本大书貌似正在突破这种限制，因为有无数的作者在写书，

每个人所关注的点也不一样，再加上搜索引擎，所以，你总是可以在互联网这本大书中找到“竟然与

我一样的人”！

于是，你可能感受到了，其实吧，所谓“刻意练习”，其实是“刻意思考哪里需要刻意练习”之后最自然

不过的事情 —— 所以，“刻意思考”才是关键。

应对策略很简单：

准备个专门的地方记录

我现在用的最多的就是 iPhone 上的 Notes，一旦遇到什么“疑似需要刻意练习”的点，就顺手记录在

那里以防不小心忘记或者不小心遗漏。

而后有时间的时候就拿出来看看，排列一下优先级，琢磨一下刻意练习的方式，而后找时间刻意练

习，如此这般，做到“尽量不混日子”……

有时候，刻意练习是很容易的，比如，为了让自己记住当前正在学习的编程语言的语法规则，直

至“刻在脑子里一般”，需要做的无非是把编辑器中的 “Auto completion”（自动补全）先关掉三个月

—— 麻烦一点就麻烦一点，坚决不让 Tab 键帮自己哗啦哗啦写上一大片…… 那不叫麻烦，那叫刻

意练习。

人们常说：

凡事儿，就怕琢磨……

那些高手，无一例外都是善于琢磨的人…… 可是，他们在琢磨什么呢？为什么他们会琢磨那些事儿

呢？

你看，所谓的琢磨，其实真的不是很难，只不过，在此之前，你不知道该琢磨什么而已，一旦知道

了，剩下的都再自然不过，都会自然而然地发生 —— 事实上，所谓的差别差距，只不过一线间而

已。

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，香港已过世的作家

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/

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

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：

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.

queens.py 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，“布林运算”；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……

关于参数（上）

之前就提到过，从结构上来看，每个函数都是一个完整的程序，因为一个程序，核心构成部分就是

输入、处理、输出：

它可以有输入 —— 即，它能接收外部通过参数传递的值；

它可以有处理 —— 即，内部有能够完成某一特定任务的代码；尤其是，它可以

根据“输入”得到“输出”；

它可以有输出 —— 即，它能向外部输送返回值……

所以，在我看来，有了一点基础知识之后，最早应该学习的是“如何写函数” —— 这个起点更好一

些。

这一章的内容，看起来会感觉与Part1 F4 函数那一章部分重合。但，这两章的出发点不一样：

Part1.E.4 函数那一章，只是为了让读者有能力“阅读”函数的说明文档；

这一章，是为了让读者能够开始动手写函数给自己或别人用……

为函数取名

哪怕一个函数内部什么都不干，它也得有个名字，然后名字后面要加上圆括号 () ，以明示它是个

函数，而不是某个变量。

定义一个函数的关键字是 def ，以下代码定义了一个什么都不干的函数：

In [1]:

```
def do_nothing():
    pass
```

do_nothing()

为函数取名（为变量取名也一样）有些基本的注意事项：

首先，名称不能以数字开头。能用在名称开头的有，大小写字母和下划线 _ ；

其次，名称中不能有空格，要么使用下划线连接词汇，如， do_nothing ，

要么使用 Camel Case，如 doNothing —— 更推荐使用下划线；

再次，名称不能与关键字重合 —— 以下是 Python 的 Keyword List：

-

Python

Keyword

List

-

and

as

assert

async

await

continue

def

del

elif

else

finally

for

from

global

if

is

lambda

None

nonlocal

not

raise

return

True

try

while

and

as

assert

async

await

你随时可以用以下代码查询关键字列表：

In [15]:

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import keyword

keyword.kwlist # 列出所有关键字

keyword.iskeyword('if') # 查询某个词是不是关键字

Out[15]: ['False',

'None',

'True',

'and',

'as',

'assert',

'async',

'await',

'break',

'class',

'continue',

'def',

'del',

'elif',

'else',

'except',

'finally',

'for',

'from',

'global',

'if',

'import',

'in',

'is',

'lambda',

'nonlocal',

'not',

'or',

'pass',

'raise',

'return',

'try',

'while',

'with',

'yield']

Out[15]: True

关于更多为函数、变量取名所需要的注意事项，请参阅：

PEP 8 -- Style Guide for Python Code: Naming Conventions

PEP 526 -- Syntax for Variable Annotations

注：PEPs，是 Python Enhancement Proposals 的缩

写：https://www.python.org/dev/peps/

不接收任何参数的函数

在定义函数的时候，可以定义成不接收任何参数；但，调用函数的时候，也依然需要写上函数名后

面的圆括号 () ：

In [3]:

def do_something():

print('This is a hello message from do_something().')

do_something()

This is a hello message from do_something().

没有 **return** 语句的函数

函数内部，不一定非要有 return 语句 —— 上面 do_somthing() 函数就没有 return 语

句。但，如果函数内部并未定义返回值，那么，该函数的返回值是 None ， 当 None 被当作布尔

值对待的时候，相当于是 False 。

这样的设定，使得函数调用总是可以在条件语句中被当作判断依据：

In [2]:

def do_something():

print('This is a hello message from do_something().')

if not do_something(): # 由于该函数名称的缘故，这一句代码的可读性

很差 ……

print("The return value of 'do_something()' is None.")

This is a hello message from do_something().

The return value of 'do_something()' is None.

if not do_something(): 翻译成自然语言，应该是，“如果 do_something() 的返回值

是‘非真’，那么：……”

接收外部传递进来的值

让我们写个判断闰年年份的函数，取名为 is_leap() ， 它接收一个年份为参数，若是闰年，则返

回 True ， 否则返回 False 。

根据闰年的定义：

年份应该是 4 的倍数；

年份能被 100 整除但不能被 400 整除的，不是闰年。

所以，相当于要在能被 4 整除的年份中，排除那些能被 100 整除却不能被 400 整除的年份。

In [12]:

```
def is_leap(year):

leap = False

if year % 4 == 0:

leap = True

if year % 100 == 0 and year % 400 != 0:

leap = False

return leap

is_leap(7)

is_leap(12)

is_leap(100)

is_leap(400)
```

Out[12]: False

Out[12]: True

Out[12]: False

Out[12]: True

In [6]:

```
# 另外一个更为简洁的版本，理解它还挺练脑子的
```

 # cpython/Lib/datetime.py

def _is_leap(year):

return year % 4 == 0 **and** (year % 100 != 0 **or** year % 400 == 0) _is_leap(300)

Out[6]: False

函数可以同时接收多个参数。比如，我们可以写个函数，让它输出从大于某个数字到小于另外一个

数字的斐波那契数列；那就需要定义两个参数，调用它的时候也需要传递两个参数：

In [7]:

def fib_between(start, end):

a, b = 0, 1

while a < end:

if a >= start:

print(a, end=' ')

a, b = b, a + b

fib_between(100, 10000)

144 233 377 610 987 1597 2584 4181 6765

当然可以把这个函数写成返回值是一个列表：

In [8]:

def fib_between(start, end):

```
r = []
a, b = 0, 1
while a < end:
if a >= start:
r.append(a)
a, b = b, a + b
return r
fib_between(100, 10000)
```

Out[8]: [144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]

变量的作用域

下面的代码，经常会让初学者迷惑：

In [9]:

```
def increase_one(n):
n += 1
return n
n = 1
print(increase_one(n))
# print(n)
```

2

当 increase_one(n) 被调用之后， n 的值究竟是多少呢？或者更准确点问，随后的

print(n) 的输出结果应该是什么呢？

输出结果是 1 。

在程序执行过程中，变量有全局变量（Global Variables）和局域变量（Local Variables）之分。

首先，每次某个函数被调用的时候，这个函数会开辟一个新的区域，这个函数内部

所有的变量，都是局域变量。也就是说，即便那个函数内部某个变量的名称与它外

部的某个全局变量名称相同，它们也不是同一个变量 —— 只是名称相同而已。

其次，更为重要的是，当外部调用一个函数的时候，准确地讲，传递的不是变量，

而是那个变量的值。也就是说，当 increase_one(n) 被调用的时候，被传递给

那个恰好名称也叫 n 的局域变量的，是全局变量 n 的值， 1 。

而后， increase_one() 函数的代码开始执行，局域变量 n 经过 n += 1 之

后，其中存储的值是 2 ，而后这个值被 return 语句返回，所

以， print(increase(n)) 所输出的值是函数被调用之后的返回值，即， 2 。

然而，全局变量 n 的值并没有被改变，因为局部变量 n （它的值是 2 ）和 全局

变量 n （它的值还是 1 ）只不过是名字相同而已，但它们并不是同一个变量。

以上的文字，可能需要反复阅读若干遍；几便下来，消除了疑惑，以后就彻底没问题了；若是这个

疑惑并未消除，或者关键点并未消化，以后会反复被这个疑惑所坑害，浪费无数时间。

不过，有一种情况要格外注意 —— 在函数内部处理被传递进来的值是可变容器（比如，列表）的时

候：

In [10]:

```
def be_careful(a, b):

a = 2

b[0] = 'What?!'

a = 1

b = [1, 2, 3]

be_careful(a, b)

a, b
```

Out[10]: (1, ['What?!', 2, 3])

所以，一个比较好的习惯是，如果传递进来的值是列表，那么在函数内部对其操作之前，先创建一

个它的拷贝：

In [11]:

```
def be_careful(a, b):

a = 2

b_copy = b.copy()
```

```
b_copy[0] = 'What?!'

a = 1

b = [1, 2, 3]

be_careful(a, b)

a, b

Out[11]: (1, [1, 2, 3])
```

关于参数（下）

可以接收一系列值的位置参数

如果你在定义参数的时候，在一个位置参数（Positional Arguments）前面标注了星号，*，那

么，这个位置参数可以接收一系列值，在函数内部可以对这一系列值用 for ... in ... 循环

进行逐一的处理。

带一个星号的参数，英文名称是“Arbitrary Positional Arguments”，姑且翻译为“随意的位置参数”。

还有带两个星号的参数，一会儿会讲到，英文名称是“Arbitrary Keyword Arguments”，姑且翻译

为“随意的关键字参数”。

有些中文书籍把 “Arbitrary Positional Arguments” 翻译成“可变位置参数”。事实上，

在这样的地方，无论怎样的中文翻译都是令人觉得非常吃力的。前面的这个翻译还

好了，我还见过把 “Arbitrary Keyword Arguments” 翻译成“武断的关键

字参数”的

—— 我觉得这样的翻译肯定会使读者产生说不明道不白的疑惑。

所以，入门之后就尽量只用英文是个好策略。虽然刚开始有点吃力，但后面会很省

心，很长寿 —— 是呀，少浪费时间、少浪费生命，其实就相当于更长寿了呀！

In [7]:

```
def say_hi(*names):

for name in names:

print(f'Hi, {name}!')

say_hi()

say_hi('ann')

say_hi('mike', 'john', 'zeo')
```

```
Hi, ann!

Hi, mike!

Hi, john!

Hi, zeo!
```

say_hi() 这一行没有任何输出。因为你在调用函数的时候，没有给它传递任何值，于是，在函

数内部代码执行的时候， name in None 的值是 False ，所以， for 循环内部的代码没有被

执行。

在函数内部，是把 names 这个参数当作容器处理的 —— 否则也没办法用 for ... in ... 来

处理。而在调用函数的时候，我们是可以将一个容器传递给函数的 Arbitrary Positional Arugments

的 —— 做法是，在调用函数的时候，在参数前面加上星号 * ：

In [54]:

```
def say_hi(*names):

for name in names:

print(f'Hi, {name}!')

names = ('mike', 'john', 'zeo')

say_hi(*names)
```

```
Hi, mike!

Hi, john!

Hi, zeo!
```

实际上，因为以上的 say_hi(*names) 函数内部就是把接收到的参数当作容器处理的，于是，

在调用这个函数的时候，向它传递任何容器都会被同样处理：

In [52]:

```
def say_hi(*names):

for name in names:

print(f'Hi, {name}!')
```

```
a_string = 'Python'

say_hi(*a_string)

a_range = range(10)

say_hi(*a_range)

a_list = list(range(10, 0, -1))

say_hi(*a_list)

a_dictionary = {'ann':2321, 'mike':8712, 'joe': 7610}

say_hi(*a_dictionary)
```

```
Hi, P!

Hi, y!

Hi, t!

Hi, h!

Hi, o!

Hi, n!

Hi, 0!

Hi, 1!

Hi, 2!

Hi, 3!

Hi, 4!

Hi, 5!
```

Hi, 6!

Hi, 7!

Hi, 8!

Hi, 9!

Hi, 10!

Hi, 9!

Hi, 8!

Hi, 7!

Hi, 6!

Hi, 5!

Hi, 4!

Hi, 3!

Hi, 2!

Hi, 1!

Hi, ann!

Hi, mike!

Hi, joe!

在定义可以接收一系列值的位置参数时，建议在函数内部为该变量命名时总是用复数，因为函数内

部，总是需要 for 循环去迭代元组中的元素，这样的时候，名称的复数形式对代码的可读性很有

帮助 —— 注意以上程序第二行。以中文为母语的人，在这个细节上常常感觉“不堪重负” —— 因为

中文的名词没有复数 —— 但必须习惯。（同样的道理，若是用拼音命名变量，就肯定是为将来挖

坑……）

注意：一个函数中，可以接收一系列值的位置参数只能有一个；并且若是还有其它位置参数存在，

那就必须把这个可以接收一系列值的位置参数排在所有其它位置参数之后。

In [10]:

```
def say_hi(greeting, *names):

for name in names:

print(f' {greeting}, {name.capitalize()}!')

sayHi('Hello', 'mike', 'john', 'zeo')
```

Hello, Mike!

Hello, John!

Hello, Zeo!

为函数的某些参数设定默认值

可以在定义函数的时候，为某些参数设定默认值，这些有默认值的参数，又被称作关键字参数

（Keyword Arguments）。从这个函数的“用户”角度来看，这些设定了默认值的参数，就成了“可选

参数”。

In [1]:

```
def say_hi(greeting, *names, capitalized=False):

for name in names:

if capitalized:

name = name.capitalize()

print(f' {greeting}, {name}!')

say_hi('Hello', 'mike', 'john', 'zeo')

say_hi('Hello', 'mike', 'john', 'zeo', capitalized=True)
```

Hello, mike!

Hello, john!

Hello, zeo!

Hello, Mike!

Hello, John!

Hello, Zeo!

可以接收一系列值的关键字参数

之前我们看到，可以设定一个位置参数（Positional　Argument），接收一系列的值，被称

作“Arbitrary Positional Argument”；

同样地，我们也可以设定一个可以接收很多值的关键字参数（Arbitrary Keyword Argument）。

In [63]:

def say_hi(**names_greetings):

for name, greeting **in** names_greetings.items():

print(f' **{greeting}**, **{name}**!')

say_hi(mike='Hello', ann='Oh, my darling', john='Hi')

Hello, mike!

Oh, my darling, ann!

Hi, john!

既然在函数内部，我们在处理接收到的 Arbitrary Keyword Argument 时，用的是对字典的迭代方

式，那么，在调用函数的时候，也可以直接使用字典的形式：

In [64]:

def say_hi(**names_greetings):

for name, greeting **in** names_greetings.items():

print(f' **{greeting}**, **{name}**!')

a_dictionary = {'mike':'Hello', 'ann':'Oh, my darling', 'john':'Hi'}

say_hi(**a_dictionary)

say_hi(**{'mike':'Hello', 'ann':'Oh, my darling', 'john':'Hi'})

Hello, mike!

Oh, my darling, ann!

Hi, john!

Hello, mike!

Oh, my darling, ann!

Hi, john!

至于在函数内部，你用什么样的迭代方式去处理这个字典，是你自己的选择：

In [65]:

```
def say_hi_2(**names_greetings):

for name in names_greetings:

print(f' {names_greetings[name]}, {name}!')

say_hi_2(mike='Hello', ann='Oh, my darling', john='Hi')
```

Hello, mike!

Oh, my darling, ann!

Hi, john!

函数定义时各种参数的排列顺序

在定义函数的时候，各种不同类型的参数应该按什么顺序摆放呢？对于之前写过的 say_hi() 这

个函数，

In [66]:

```
def say_hi(greeting, *names, capitalized=False):

for name in names:

if capitalized:

name = name.capitalize()
```

print(f' **{greeting}**, **{name}**!')

say_hi('Hi', 'mike', 'john', 'zeo')

say_hi('Welcome', 'mike', 'john', 'zeo', capitalized=**True**)

Hi, mike!

Hi, john!

Hi, zeo!

Welcome, Mike!

Welcome, John!

Welcome, Zeo!

如果，你想给其中的 greeting 参数也设定个默认值怎么办？写成这样好像可以：

In [24]:

def say_hi(greeting='Hello', *names, capitalized=**False**):

for name **in** names:

if capitalized:

name = name.capitalize()

print(f' **{greeting}**, **{name}**!')

say_hi('Hi', 'mike', 'john', 'zeo')

say_hi('Welcome', 'mike', 'john', 'zeo', capitalized=**True**)

Hi, mike!

Hi, john!

Hi, zeo!

Welcome, Mike!

Welcome, John!

Welcome, Zeo!

但， greeting 这个参数虽然有默认值，可这个函数在被调用的时候，还是必须要给出这个参

数，否则输出结果出乎你的想象：

In [25]:

```
def say_hi(greeting='Hello', *names, capitalized=False):

for name in names:

if capitalized:

name = name.capitalize()

print(f' {greeting}, {name}!')

say_hi('mike', 'john', 'zeo')
```

mike, john!

mike, zeo!

设定了默认值的 greeting ，竟然不像你想象的那样是“可选参数”！所以，你得这样写：

In [67]:

```
def say_hi(*names, greeting='Hello', capitalized=False):

for name in names:
```

if capitalized:

name = name.capitalize()

print(f' **{greeting}**, **{name}**!')

say_hi('mike', 'john', 'zeo')

say_hi('mike', 'john', 'zeo', greeting='Hi')

Hello, mike!

Hello, john!

Hello, zeo!

Hi, mike!

Hi, john!

Hi, zeo!

这是因为函数被调用时，面对许多参数，Python 需要按照既定的规则（即，顺序）判定每个参数究

竟是哪一类型的参数：

Order of Arguments

1. Positional

2. Arbitrary Positional

3. Keyword

4. Arbitrary Keyword

所以，即便你在定义里写成

def say_hi(greeting='Hello', *names, capitalized=False):

...

在调用该函数的时候，无论你写的是

say_hi('Hi', 'mike', 'john', 'zeo')

还是

say_hi('mike', 'john', 'zeo')

Python 都会认为接收到的第一个值是 Positional Argument —— 因为在定义中， greeting 被放

到了 Arbitrary Positional Arguments 之前。

化名与匿名

化名

在 Python 中，我们可以给一个函数取个化名（alias）。

以下的代码，我们先是定义个了一个名为 _is_leap 的函数，而后为它另取了一个化名：

In [2]:

from **IPython.core.interactiveshell** **import** InteractiveShell InteractiveShell.ast_node_interactivity = "all"

def _is_leap(year):

return year % 4 == 0 **and** (year % 100 != 0 **or** year % 400 == 0) year_leap_bool = _is_leap

year_leap_bool #

year_leap_bool(800) # *_is_leap(800) -> True*

id(year_leap_bool) # *id()* 这个函数可以查询某对象的内存地址

id(_is_leap) # *year_leap_bool* 和 *_is_leap* 其实保存在同一个地址中，也就是说，它们是同一个对象。

type(year_leap_bool)

type(_is_leap) # 它们都是 *function*

Out[2]:

Out[2]: True

Out[2]: 4547071648

Out[2]: 4547071648

Out[2]: function

Out[2]: function

我们可以看到的是， id(year_leap_bool) 和 id(_is_leap) 的内存地址是一样的 —— 它

们是同一个对象，它们都是函数。所以，当你写 year_leap_bool = _is_leap 的时候，相当

于给 _is_leap() 这个函数取了个化名。

在什么样的情况下，要给一个函数取一个化名呢？

在任何一个工程里，为函数或者变量取名都是很简单却不容易的事情 —— 因为可能会重名（虽然已

经尽量用变量的作用域隔离了），可能会因取名含混而令后来者费解……

所以，仅仅为了少敲几下键盘而给一个函数取个更短的化名，实际上并不是好主意，更不是好习

惯。尤其现在的编辑器都支持自动补全和多光标编辑的功能，变量名再长都不构成负担。

更多的时候，为函数取一个化名，应该是为了提高代码可读性 —— 对自己或其他人都很重要。

lambda

写一个很短的函数可以用 lambda 关键字。

下面是用 def 关键字写函数：

In [14]:

```
def add(x, y):
return x + y
add(3, 5)
```

Out[14]: 8

下面是用 lambda 关键字写函数：

In [17]:

```
add = lambda x, y: x + y
add(3, 5)
```

Out[17]: 8

lambda 的语法结构如下：

```
lambda_expr ::= "lambda" [parameter_list] ":" expression
```

以上使用的是 BNF 标注。当然，BNF 是你目前并不熟悉的，所以，有疑惑别当回事儿。

反正你已经见到示例了：

lambda x, y: x + y

先写上 lambda 这个关键字，其后分为两个部分， ：之前是参数，之后是表达式；这个表达式的

值，就是这个函数的返回值。

注意： lamda 语句中， :之后有且只能有一个表达式。

而这个函数呢，没有名字，所以被称为“匿名函数”。

add = lambda x, y: x + y

就相当于是给一个没有名字的函数取了个名字。

lambda 的使用场景

那 lambda 这种匿名函数的用处在哪里呢？

作为某函数的返回值

第一个常见的用处是作为另外一个函数的返回值。

让我们看看 The Python Tutorial 中的一个例子。

In [22]:

def make_incrementor(n):

return lambda x: x + n

f = make_incrementor(42)

f(0)

f(1)

Out[22]: 42

Out[22]: 43

这个例子乍看起来很令人迷惑。我们先看看 f = make_incrementor(42) 之后， f 究竟是什

么东西：

In [23]:

```
def make_incrementor(n):

return lambda x: x + n

f = make_incrementor(42)

f

id(make_incrementor)

id(f)
```

Out[23]: . (x)>

Out[23]: 4428443296

Out[23]: 4428726888

首先，要注意， f 并不是 make_incrementor() 这个函数的化名，如果是给这个函数取个化

名，写法应该是：

```
f = make_incrementor
```

那 f 是什么呢？ 它是 .

(x)> ：

f = make_incrementor(42) 是将 make_incrementor(42) 的返回值

保存到 f 这个变量之中；

而 make_incrementor() 这个函数接收到 42 这个参数之后，返回了一个

函数： lambda x: x + 42 ；

于是， f 中保存的函数是 lambda x: x + 42 ；

所以， f(0) 是向这个匿名函数传递了 0 ，而后，它返回的是 0 + 42 。

作为某函数的参数

可以拿一些可以接收函数为参数的内建函数做例子。比如， map() 。

map (*function, iterable, ...*)

Return an iterator that applies *function* to every item of *iterable*, yielding the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. With multiple iterables, the iterator stops when the shortest iterable is exhausted. For cases

where the function inputs are already arranged into argument tuples, see

itertools.starmap() .

map() 这个函数的第一个参数，就是用来接收函数的。随后的参数，是 iterable —— 就是可

被迭代的对象，比如，各种容器，例如：列表、元组、字典什么的。

In [35]:

```
def double_it(n):

return n * 2

a_list = [1, 2, 3, 4, 5, 6]
```

b_list = list(map(double_it, a_list))

b_list

c_list = list(map(**lambda** x: x * 2, a_list))

c_list

Out[35]: [2, 4, 6, 8, 10, 12]

Out[35]: [2, 4, 6, 8, 10, 12]

显然用 lambda 更为简洁。另外，类似完成 double_it(n) 这种简单功能的函数，常常有“用

过即弃”的必要。

In [60]:

phonebook = [

{

'name': 'john',

'phone': 9876

},

{

'name': 'mike',

'phone': 5603

},

{

'name': 'stan',

'phone': 6898

},

{

'name': 'eric',

'phone': 7898

}

]

phonebook

list(map(**lambda** x: x['name'], phonebook))

list(map(**lambda** x: x['phone'], phonebook))

Out[60]: [{'name': 'john', 'phone': 9876},

{'name': 'mike', 'phone': 5603},

{'name': 'stan', 'phone': 6898},

{'name': 'eric', 'phone': 7898}]

Out[60]: ['john', 'mike', 'stan', 'eric']

Out[60]: [9876, 5603, 6898, 7898]

可以给 map() 传递若干个可被迭代对象：

In [63]:

a_list = [1, 3, 5]

b_list = [2, 4, 6]

list(map(**lambda** x, y: x * y, a_list, b_list))

Out[63]: [2, 12, 30]

以上的例子都弄明白了，再去看 The Python Tutorial 中的例子，就不会有任何疑惑了：

In [1]:

pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

pairs.sort(key=**lambda** p: p[1])

pairs

Out[1]: [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]

递归函数

递归（**Recursion**）

在函数中有个理解门槛比较高的概念：递归函数（Recursive Functions）—— 那些在自身内部调用

自身的函数。说起来都比较拗口。

先看一个例子，我们想要有个能够计算 n 的阶乘（factorial） n! 的函数， f() ，规则如下：

n! = n × (n-1) × (n-2)... × 1

即， n! = n × (n-1)!

且， n >= 1

注意：以上是数学表达，不是程序，所以， = 在这一小段中是“等于”的意思，不是

程序语言中的赋值符号。

于是，计算 f(n) 的 Python 程序如下：

In [1]:

```
def f(n):

if n == 1:

return 1

else:

return n * f(n-1)

print(f(5))
```

120

递归函数的执行过程

以　factorial(5)　为例，让我们看看程序的流程（注意，图片里的诸如 n=5>1 之类的标注，并不是程

序语言表达式，只是对读者的说明）：

当 f(5) 被调用之后，函数开始运行……

因为 5 > 1 ，所以，在计算 n * f(n-1) 的时候要再次调用自己 f(4) ；所以必须等待

f(4) 的值返回；

因为 4 > 1 ，所以，在计算 n * f(n-1) 的时候要再次调用自己 f(3) ；所以必须等待

f(3) 的值返回；

因为 3 > 1 ，所以，在计算 n * f(n-1) 的时候要再次调用自己 f(2) ；所以必须等待

f(2) 的值返回；

因为 2 > 1 ，所以，在计算 n * f(n-1) 的时候要再次调用自己 f(1) ；所以必须等待

f(1) 的值返回；

因为 1 == 1 ，所以，这时候不会再次调用 f() 了，于是递归结束，开始返回，这次返回的

是 1 ；

下一步返回的是 2 * 1 ；

下一步返回的是 3 * 2 ；

下一步返回的是 4 * 6 ；

下一步返回的是 5 * 24 ——至此，外部调用 f(5) 的最终返回值是 120 ……

加上一些输出语句之后，能更清楚地看到大概的执行流程：

In [1]:

```
def f(n):

print(' \tn =', n)

if n == 1:

print('Returning...')

print(' \tn =', n, 'return:', 1)

return 1
```

else:

r = n * f(n-1)

print(' \tn =', n, 'return:', r)

return r

print('Call f(5)...')

print('Get out of f(n), and f(5) =', f(5))

Call f(5)...

n = 5

n = 4

n = 3

n = 2

n = 1

Returning...

n = 1 return: 1

n = 2 return: 2

n = 3 return: 6

n = 4 return: 24

n = 5 return: 120

Get out of f(n), and f(5) = 120

有点烧脑……　不过，分为几个层面去逐个突破，你会发现它真的很好玩。

递归的终点

递归函数在内部必须有一个能够让自己停止调用自己的方式，否则永远循环下去了……

其实，我们所有人很小就见过递归应用，只不过，那时候不知道那就是递归而已。听过那个无聊的

故事罢？

山上有座庙，庙里有个和尚，和尚讲故事，说……

山上有座庙，庙里有个和尚，和尚讲故事，说……

山上有座庙，庙里有个和尚，和尚讲故事，说……

写成 Python 程序大概是这样：

```
def a_monk_telling_story():

print('山上有座庙，庙里有个和尚，和尚讲故事，他说……')

return a_monk_telling_story()

a_monk_telling_story()
```

这是个_无限循环_的递归，因为这个函数里没有设置中止自我调用的条件。无限循环还有个不好听

的名字，叫做“死循环”。

在著名的电影盗梦空间（ *2010*）里，从整体结构上来看，“入梦”也是个“递归函数”。只不过，这个函

数和 a_monk_telling_story() 不一样，它并不是死循环 —— 因为它设定了中止自我调用的条件：

在电影里，醒过来的条件有两个

一个是在梦里死掉；

一个是在梦里被 kicked 到……

如果这两个条件一直不被满足，那就进入 limbo 状态 —— 其实就跟死循环一样，出

不来了……

为了演示，我把故事情节改变成这样：

入梦， in_dream() ，是个递归函数；

入梦之后醒过来的条件有两个：

一个是在梦里死掉， dead is True ；

一个是在梦里被 kicked， kicked is True ……

以上两个条件中任意一个被满足，就再睡上一天……

至于为什么会死掉，如何被 kick，我偷懒了一下：管它怎样，管它如何，反正，每个条件被满足的

概率是 1/10…… (也只有这样，我才能写出一个简短的，能够运行的“盗梦空间程序”。）

把这个很抽象的故事写成 Python 程序，看看一次入梦之后能睡多少天，大概是这样：

In [1]:

```
import random

def in_dream(day=0, dead=False, kicked=False):

dead = not random.randrange(0,10) # 1/10 probability to be dead

kicked = not random.randrange(0,10) # 1/10 probability to be kicked day +=
```

1

print('dead:', dead, 'kicked:', kicked)

if dead:

print((f"I slept **{day}** days, and was dead to wake up..."))

return day

elif kicked:

print(f"I slept **{day}** days, and was kicked to wake up...")

return day

return in_dream(day)

print('The in_dream() function returns:', in_dream())

dead: False kicked: False

dead: False kicked: False

dead: False kicked: False

dead: False kicked: False

dead: False kicked: False

dead: False kicked: False

dead: False kicked: False

dead: True kicked: True

I slept 8 days, and was dead to wake up...

The in_dream() function returns: 8

如果疑惑为什么 random.randrange(0,10) 能表示 1/10 的概率，请返回去重新阅读第一部分

中关于布尔值的内容。

另外，在 Python 中，若是需要将某个值于 True 或者 False 进行比较，尤其是再条件语句中，推荐

写法是（参见 PEP8）：

if condition:

pass

就好像上面代码中的 if dead: 一样。

而不是（虽然这么写通常也并不妨碍程序正常运行[1]）：

if condition **is** True:

pass

抑或：

if condition == True:

pass

让我们再返回来接着讲递归函数。正常的递归函数一定有个退出条件。否则的话，就_无限循环_下

去了…… 下面的程序在执行一会儿之后就会告诉你： RecursionError: maximum recursion

depth exceeded （上面那个“山上庙里讲故事的和尚说”的程序，真要跑起来，也是这样）：

In [3]:

def x(n):

return n * x(n-1)

x(5)

RecursionError Traceback (most recent call la

st)

in

1 def x(n):

2 return n * x(n-1)

----> 3 x(5)

in x(n)

1 def x(n):

----> 2 return n * x(n-1)

3 x(5)

... last 1 frames repeated, from the frame below ...

in x(n)

1 def x(n):

----> 2 return n * x(n-1)

3 x(5)

RecursionError: maximum recursion depth exceeded

不用深究上面盗梦空间这个程序的其它细节，不过，通过以上三个递归程序 —— 两个很扯淡的例

子，一个正经例子 —— 你已经看到了递归函数的共同特征：

1. 在 return 语句中返回的是自身的调用（或者是含有自身的表达式）

2. 为了避免死循环，_一定要有至少一个条件_下返回的不再是自身调用……

变量的作用域

再回来看计算阶乘的程序 —— 这是正经程序。这次我们把程序名写完整， factorial() :

In [6]:

```
def factorial(n):

if n == 1:

return 1

else:

return n * factorial(n-1)

print(factorial(5))
```

120

最初的时候，这个函数的执行流程之所以令人迷惑，是因为初学者对_变量_的作用域把握得不够充

分。

变量根据作用域，可以分为两种：全局变量（Global）和局部变量

（Local）。

可以这样简化理解：

在函数内部被赋值而后使用的，都是局部变量，它们的作用域是局部，无法被

函数外的代码调用；

在所有函数之外被赋值而后开始使用的，是全局变量，它们的作用域是全局，

在函数内外都可以被调用。

定义如此，但，通常程序员们会严格地遵守一条原则：

在函数内部绝对不调用全局变量。即便是必须改变全局变量，也只能通过函数的返

回值在函数外改变全局变量。

你也必须遵守同样的原则。而这个原则同样可以在日常的工作生活中“调用”：

做事的原则：自己的事儿自己做，别人的事儿，最多通过自己的产出让他们自己去

搞……

再仔细观察一下以下代码。当一个变量被当做参数传递给一个函数的时候，这个变量本身并不会被

函数所改变。比如， a = 5 ，而后，再把 a 当作参数传递给 f(a) 的时候，这个函数当然应该

返回它内部任务完成之后应该传递回来的值，但， a 本身不会被改变。

In [7]:

```
def factorial(n):

if n == 1:

return 1

else:

return n * factorial(n-1)

a = 5

b = factorial(a) # a 并不会因此改变；

print(a, b)

a = factorial(a) # 这是你主动为 a 再一次赋值 ……

print(a, b)
```

5 120

120 120

理解了这一点之后，再看 factorial() 这个递归函数的递归执行过程，你就能明白这个事实：

在每一次 factorial(n) 被调用的时候，它都会形成一个作用域， n 这个变量作为参

数把它的值传递给了函数，但是， n 这个变量本身并不会被改变。

我们再修改一下上面的代码：

In [8]:

```
def factorial(n):

if n == 1:
```

return 1

else:

return n * factorial(n-1)

n = 5 # 这一次，这个变量名称是 *n*

m = factorial(n) # *n* 并不会因此改变；

print(n, m)

5 120

在 m = factorial(n) 这一句中， n 被 factorial() 当做参数调用了，但，无论函数内部

如何操作，并不会改变变量 n 的值。

关键的地方在这里：在函数内部出现的变量 n ，和函数外部的变量 n 不是一回事儿 —— 它们只

是名称恰好相同而已，函数参数定义的时候，用别的名称也没什么区别：

In [1]:

def factorial(x): # 在这个语句块中出现的变量，都是局部变量

if x == 1:

return 1

else:

return x * factorial(x-1)

n = 5 # 这一次，这个变量名称是 *n*

m = factorial(n) # *n* 并不会因此改变；

print(n, m)

这个例子和之前再之前的示例代码有什么区别吗？

本质上没区别，就是变量名称换了而已 ……

5 120

函数开始执行的时候， x 的值，是由外部代码（即，函数被调用的那一句）传递进来的。即便函

数内部的变量名称与外部的变量名称相同，它们也不是同一个变量。

In [1]:

观察一下名称相同的一个全局变量和局部变量的不同内存地址

def f(n):

return id(n)

n = 5

print(id(n)) # 全局变量 *n* 的内存地址

print(id(f(n))) # 局部变量 *n* 的内存地址。

4430918896

4467206608

递归函数三原则

现在可以小小总结一下了。

一个递归函数，之所以是一个有用、有效的递归函数，因为它要遵守递归三原则。正如，一个机器

人之所以是个合格的机器人，因为它遵循阿莫西夫三铁律（Three Laws of Robotics）一样[2]。

1. 根据定义，递归函数必须在内部调用自己；

2. 必须设定一个退出条件；

3. 递归过程中必须能够逐步达到退出条件……

从这个三原则望过去， factorial() 是个合格有效的递归函数，满足第一条，满足第二条，尤

其还满足第三条中的“逐步达到”！

而那个扯淡的盗梦空间递归程序，说实话，不太合格，虽然它满足第一条，也满足第二条，第三条

差点蒙混过关：它不是逐步达到，而是不管怎样肯定能达到 —— 这明显是两回事儿…… 原谅它

罢，它的作用就是当例子，一次正面的，一次负面的，作为例子算是功成圆满了！

刚开始的时候，初学者好不容易搞明白递归函数究竟是怎么回事儿之后，就不由自主地想“我如何才

能学会递归式思考呢？” —— 其实吧，这种想法本身可能并不是太正确或者准确。

准确地讲，递归是一种解决问题的方式。当我们需要解决的问题，可以被逐步拆分成很多越来越小

的模块，然后每个小模块还都能用同一种算法处理的时候，用递归函数最简洁有效。所以，只不过

是在遇到可以用递归函数解决问题的时候，才需要去写递归函数。

从这个意义上来看，递归函数是程序员为了自己方便而使用的，并不是为了计算机方便而使用 ——

计算机么，你给它的任务多一点或者少一点，对它来讲无所谓，反正有电就能运转，它自己又不付

电费……

理论上来讲，所有用递归函数能完成的任务，不用递归函数也能完成，只不过代码多一点，啰嗦一

点，看起来没有那么优美而已。

还有，递归，不像“序列类型”那样，是某个编程语言的特有属性。它其实是一种特殊算法，也是一

种编程技巧，任何编程语言，都可以使用递归算法，都可以通过编写递归函数巧妙地解决问题。

但是，学习递归函数本身就很烧脑啊！这才是最大的好事儿。从迷惑，到不太迷惑，到清楚，到很

清楚，再到特别清楚 —— 这是个非常有趣，非常有成就感的过程。

这种过程锻炼的是脑力 —— 在此之后，再遇到大多数人难以理解的东西，你就可以使用这一次积累

的经验，应用你已经磨炼过的脑力。有意思。

至此，封面上的那个“伪代码”应该很好理解了：

```
def TeachYourSelf(anything):

while not create(somthing):

learn()

practice()

return TeachYourSelf(another)

TeachYourSelf(coding)
```

自学还真的就是递归函数呢……

思考与练习

MIT 的一个网页，有很多递归的例子

https://introcs.cs.princeton.edu/java/23recursion/

脚注

[1]：参见 Stackoverflow 上的讨论：Boolean identity == True vs is True

↑Back to Content↑

[2]：关于阿莫西夫三铁律（Three Laws of Robotics）的类比，来自著名的 Python 教

程，Think Python: How to Think Like a Computer Scientist

↑Back to Content↑

函数的文档

你在调用函数的时候，你像是函数这个产品的用户。

而你写一个函数，像是做一个产品，这个产品将来可能会被很多用户使用 —— 包括你自己。

产品，就应该有产品说明书，别人用得着，你自己也用得着 —— 很久之后的你，很可能把当初的各

种来龙去脉忘得一干二净，所以也同样需要产品说明书，别看那产品曾经是你自己设计的。

Python 在这方面很用功，把函数的“产品说明书”当作语言内部的功能，这也是为什么 Python 有

Sphinx 这种工具，而绝大多数其他语言没有的原因之一罢。

Docstring

在函数定义内部，我们可以加上 **Docstring**；将来函数的“用户”就可以通过 help() 这个内建函

数，或者 .__doc__ 这个 Method 去查看这个 Docstring，即，该函数的“产品说明书”。

先看一个 Docstring 以及如何查看某个函数的 Docstring 的例子：

In [21]:

```
def is_prime(n):
"""
Return a boolean value based upon
whether the argument n is a prime number.
"""
if n < 2:
return False
if n == 2:
return True
for m in range(2, int(n**0.5)+1):
if (n % m) == 0:
return False
else:
return True
```

help(is_prime)

print(is_prime.__doc__)

is_prime.__doc__

Help on function is_prime in module __main__:

is_prime(n)

Return a boolean value based upon

whether the argument n is a prime number.

Return a boolean value based upon

whether the argument n is a prime number.

Out[21]: '\n Return a boolean value based upon\n whether the argument n is a prime number.\n '

Docstring 可以是多行字符串，也可以是单行字符串：

In [16]:

```
def is_prime(n):
"""Return a boolean value based upon whether the argument n is a pri
me number."""
if n < 2:
return False
if n == 2:
return True
for m in range(2, int(n**0.5)+1):
```

if (n % m) == 0:

return False

else:

return True

help(is_prime)

print(is_prime.__doc__)

is_prime.__doc__

Help on function is_prime in module __main__:

is_prime(n)

Return a boolean value based upon whether the argument n is a prime

number.

Return a boolean value based upon whether the argument n is a prime numb

er.

Out[16]: 'Return a boolean value based upon whether the argument n is a prime num ber.'

Docstring 如若存在，必须在函数定义的内部语句块的开头，也必须与其它语句一样保持相应的缩进

（Indention）。Docstring 放在其它地方不起作用：

In [12]:

def is_prime(n):

if n < 2:

return False

if n == 2:

return True

for m **in** range(2, int(n**0.5)+1):

if (n % m) == 0:

return False

else:

return True

"""

Return a boolean value based upon

whether the argument n is a prime number.

"""

help(is_prime)

print(is_prime.__doc__)

is_prime.__doc__

Help on function is_prime in module __main__:

is_prime(n)

None

书写 **Docstring** 的规范

规范，虽然是人们最好遵守的，但其实通常是很多人并不遵守的东西。

既然学，就要像样 —— 这真的很重要。所以，非常有必要认真阅读 Python PEP 257 关于

Docstring 的规范。

简要总结一下 PEP 257 中必须掌握的规范：

1. 无论是单行还是多行的 Docstring，一概使用三个双引号扩起；

2. 在 Docstring 内部，文字开始之前，以及文字结束之后，都不要有空行；

3. 多行 Docstring，第一行是概要，随后空一行，再写其它部分；

4. 完善的 Docstring，应该概括清楚以下内容：参数、返回值、可能触发的错误类

型、可能的副作用，以及函数的使用限制等等；

5. 每个参数的说明都使用单独的一行……

由于我们还没有开始研究 Class，所以，关于 Class 的 Docstring 应该遵守什么样的规范就暂时略

过了。然而，这种规范你总是要反复去阅读参照的。关于 Docstring，有两个规范文件：

PEP 257: Docstring Convensions

PEP 258: Docutils Design Specification

需要格外注意的是：

Docstring 是写给人看的，所以，在复杂代码的 Docstring 中，写 **Why** 要远比写

What 更重要 —— 你先记住这点，以后的体会自然会不断加深。

Sphinx 版本的 **Docstring** 规范

Sphinx 可以从 .py 文件里提取所有 Docstring，而后生成完整的 Documentation。将来若是你写

大型的项目，需要生成完善的文档的时候，你就会发现 Sphinx 是个“救命”的家伙，省时、省力、省

心、省命……

在这里，没办法一下子讲清楚 Sphinx 的使用，尤其是它还用它自己的一种标记语

言，reStructureText，文件尾缀使用 .rst ……

但是，可以看一个例子：

In [22]:

```
class Vehicle(object):
    '''
    The Vehicle object contains lots of vehicles

    :param arg: The arg is used for ...
    :type arg: str
    :param `*args`: The variable arguments are used for ...
    :param `**kwargs`: The keyword arguments are used for ...
    :ivar arg: This is where we store arg
    :vartype arg: str
    '''
    def __init__(self, arg, *args, **kwargs):
```

self.arg = arg

def cars(self, distance, destination):

'''We can't travel a certain distance in vehicles without fuels,

so here's the fuels

:param distance: The amount of distance traveled

:type amount: int

:param bool destinationReached: Should the fuels be refilled to

cover required distance?

:raises: :class:`RuntimeError`: Out of fuel

:returns: A Car mileage

:rtype: Cars

'''

pass

help(Vehicle)

Help on class Vehicle in module __main__:

class Vehicle(builtins.object)

| Vehicle(arg, *args, **kwargs)

|

| The Vehicle object contains lots of vehicles

| :param arg: The arg is used for ...

| :type arg: str

| :param `*args`: The variable arguments are used for ...

| :param `**kwargs`: The keyword arguments are used for ...

| :ivar arg: This is where we store arg

| :vartype arg: str

|

| Methods defined here:

|

| __init__(self, arg, *args, **kwargs)

| Initialize self. See help(type(self)) for accurate signature.

|

| cars(self, distance, destination)

| We can't travel a certain distance in vehicles without fuels, so

here's the fuels

|

| :param distance: The amount of distance traveled

| :type amount: int

| :param bool destinationReached: Should the fuels be refilled to

cover required distance?

| :raises: :class:`RuntimeError`: Out of fuel

|

| :returns: A Car mileage

| :rtype: Cars

|

| --

--

| Data descriptors defined here:

|

| __dict__

| dictionary for instance variables (if defined)

|

| __weakref__

| list of weak references to the object (if defined)

通过插件，Sphinx 也能支持 Google Style Docstring 和 Numpy Style Docstring。

以下两个链接，放在这里，以便你将来查询：

sphinx.ext.napoleon – Support for NumPy and Google style docstrings

sphinx.ext.autodoc – Include documentation from docstrings

保存到文件的函数

写好的函数，当然最好保存起来，以便将来随时调用。

模块

我们可以将以下内容保存到一个名为 mycode.py 的文件中 —— 这样可以被外部调用的 .py 文

件，有个专门的称呼，模块（Module）—— 于是，其实任何一个 .py 文件都可以被称为模块：

In [3]:

```
# %load mycode.py

# 当前这个 Code Cell 中的代码，保存在当前文件夹中的 mycode.py 文件中

# 以下的代码，是使用 Jupyter 命令 %load mycode.py 导入到当前 Code Cell 中的：

def is_prime(n):

"""

Return a boolean value based upon

whether the argument n is a prime number.

"""

if n < 2:

return False

if n == 2:

return True

for m in range(2, int(n**0.5)+1):

if (n % m) == 0:

return False
```

else:

return True

def say_hi(*names, greeting='Hello', capitalized=**False**):

"""

Print a string, with a greeting to everyone.

*:param *names: tuple of names to be greeted.*

:param greeting: 'Hello' as default.

:param capitalized: Whether name should be converted to capitalzed b efore print. False as default.

:returns: None

"""

for name **in** names:

if capitalized:

name = name.capitalize()

print(f' **{greeting}**, **{name}**!')

而后，我们就可以在其它地方这样使用（以上代码现在已经保存在当前工作目录中的

mycode.py）：

In [2]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

import mycode

help(mycode.is_prime)

help(mycode.say_hi)

mycode.__name__

mycode.is_prime(3)

mycode.say_hi('mike', 'zoe')

Help on function is_prime in module mycode:

is_prime(n)

Return a boolean value based upon

whether the argument n is a prime number.

Help on function say_hi in module mycode:

say_hi(*names, greeting='Hello', capitalized=False)

Print a string, with a greeting to everyone.

:param *names: tuple of names to be greeted.

:param greeting: 'Hello' as default.

:param capitalized: Whether name should be converted to capitalzed b

efore print. False as default.

:returns: None

Hello, mike!

Hello, zoe!

以上这个模块（Module）的名称，就是 mycode 。

模块文件系统目录检索顺序

当你向 Python 说 import ... 的时候，它要去寻找你所指定的文件，那个文件应该是 import

语句后面引用的名称，再加上 .py 构成的名字的文件。Python 会按照以下顺序去寻找：

先去看看内建模块里有没有你所指定的名称；

如果没有，那么就按照 sys.path 所返回的目录列表顺序去找。

你可以通过以下代码查看你自己当前机器的 sys.path ：

In [1]:

import sys

sys.path

在 sys.path 所返回的目录列表中，你当前的工作目录排在第一位。

有时，你需要指定检索目录，因为你知道你要用的模块文件在什么位置，那么可以用

sys.path.append() 添加一个搜索位置：

import sys

sys.path.append("/My/Path/To/Module/Directory")

import my_module

系统内建的模块

你可以用以下代码获取系统内建模块的列表：

In [22]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

sys.builtin_module_names

"_sre" in sys.builtin_module_names # True

"math" in sys.builtin_module_names # True
```

Out[22]: ('_abc',

'_ast',

'_codecs',

'_collections',

'_functools',

'_imp',

'_io',

'_locale',

'_operator',

'_signal',

'_sre',

'_stat',

'_string',

'_symtable',

'_thread',

'_tracemalloc',

'_warnings',

'_weakref',

'atexit',

'builtins',

'errno',

'faulthandler',

'gc',

'itertools',

'marshal',

'posix',

'pwd',

'sys',

'time',

'xxsubtype',

'zipimport')

Out[22]: True

Out[22]: False

跟变量名、函数名，不能与关键字重名一样，你的模块名称也最好别与系统内建模块名称重合。

指定引入模块中特定函数

当你使用 import mycode 的时候，你向当前工作空间引入了 mycode 文件中定义的所有函

数，相当于：

In [ ]:

from mycode import *

你其实可以只引入当前需要的函数，比如，只引入 is_prime()

In [6]:

from mycode import is_prime

这种情况下，你就不必使用 mycode.is_prime() 了；而是就好像这个函数就写在当前工作空间

一样，直接写 is_prime()：

In [7]:

from mycode import is_prime

is_prime(3)

Out[7]: True

注意，如果当前目录中并没有 mycode.py 这个文件，那么， mycode 会被当作目录名再被尝试

一次，如果当前目录内，有个叫做 mycode 的目录（或称文件夹），那么， from mycode

import * 的作用就是把 mycode 这个文件夹中的所有 .py 文件全部导入……

如果我们想要导入 foo 这个目录中的 bar.py 这个模块文件，那么，可以这么写：

import foo.bar

或者

from foo import bar

引入并使用化名

有的时候，或者为了避免混淆，或者为了避免输入太多字符，我们可以为引入的函数设定 化

名（alias），而后使用化名调用函数。比如：

In [8]:

from mycode import is_prime **as** isp

isp(3)

Out[8]: True

甚至干脆给整个模块取个化名：

In [13]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

import mycode as m

m.is_prime(3)

m.say_hi('mike', 'zoe')

Out[13]: True

Hello, mike!

Hello, zoe!

模块中不一定只有函数

一个模块文件中，不一定只包含函数；它也可以包含函数之外的可执行代码。只不过，在

import 语句执行的时候，模块中的非函数部分的可执行代码，只执行一次。

有一个 Python 的彩蛋，恰好是可以用在此处的最佳例子 —— 这个模块是 this ，它的文件名是

this.py ：

In [16]:

import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

这个 this 模块中的代码如下：

In [ ]:

```
s = """Gur Mra bs Clguba, ol Gvz Crgref

Ornhgvshy vf orggre guna htyl.

Rkcyvpvg vf orggre guna vzcyvpvg.

Fvzcyr vf orggre guna pbzcyrk.

Pbzcyrk vf orggre guna pbzcyvpngrq.

Syng vf orggre guna arfgrq.

Fcnefr vf orggre guna qrafr.

Ernqnovyvgl pbhagf.

Fcrpvny pnfrf nera'g fcrpvny rabhtu gb oernx gur ehyrf.
```

Nygubhtu cenpgvpnyvgl orngf chevgl.

Reebef fubhyq arire cnff fvyragyl.

Hayrff rkcyvpvgyl fvyraprq.

Va gur snpr bs nzovthvgl, ershfr gur grzcgngvba gb thrff.

Gurer fubhyq or bar-- naq cersrenoyl bayl bar --boivbhf jnl gb qb vg.

Nygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er Qhgpu.

Abj vf orggre guna arire.

Nygubhtu arire vf bsgra orggre guna *evtug* abj.

Vs gur vzcyrzragngvba vf uneq gb rkcynva, vg'f n onq vqrn.

Vs gur vzcyrzragngvba vf rnfl gb rkcynva, vg znl or n tbbq vqrn.

Anzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!"""

```
d = {}
for c in (65, 97):
for i in range(26):
d[chr(i+c)] = chr((i+13) % 26 + c)
print("".join([d.get(c, c) for c in s]))
```

这个 this.py 文件中也没有什么函数，但，这个文件里所定义的变量，我们都可以在 import

this 之后触达：

In [19]:

```
from IPython.core.interactiveshell import InteractiveShell
```

InteractiveShell.ast_node_interactivity = "all"

import this

this.d

this.s

Out[19]: {'A': 'N',

'B': 'O',

'C': 'P',

'D': 'Q',

'E': 'R',

'F': 'S',

'G': 'T',

'H': 'U',

'I': 'V',

'J': 'W',

'K': 'X',

'L': 'Y',

'M': 'Z',

'N': 'A',

'O': 'B',

'P': 'C',

'Q': 'D',

'R': 'E',

'S': 'F',

'T': 'G',

'U': 'H',

'V': 'I',

'W': 'J',

'X': 'K',

'Y': 'L',

'Z': 'M',

'a': 'n',

'b': 'o',

'c': 'p',

'd': 'q',

'e': 'r',

'f': 's',

'g': 't',

'h': 'u',

'i': 'v',

'j': 'w',

'k': 'x',

'l': 'y',

'm': 'z',

'n': 'a',

'o': 'b',

'p': 'c',

'q': 'd',

'r': 'e',

's': 'f',

't': 'g',

'u': 'h',

'v': 'i',

'w': 'j',

'x': 'k',

'y': 'l',

'z': 'm'}

Out[19]: "Gur Mra bs Clguba, ol Gvz Crgref\n\nOrnhgvshy vf orggre guna htyl.\nRkc

yvpvg vf orggre guna vzcyvpvg.\nFvzcyr vf orggre guna pbzcyrk.\nPbzcyrk vf orggre guna pbzcyvpngrq.\nSyng vf orggre guna arfgrq.\nFcnefr vf orgg

re guna qrafr.\nErnqnovyvgl pbhagf.\nFcrpvny pnfrf nera'g fcrpvny rabhtu

gb oernx gur ehyrf.\nNygubhtu cenpgvpnyvgl orngf chevgl.\nReebef fubhyq arire cnff fvyragyl.\nHayrff rkcyvpvgyl fvyraprq.\nVa gur snpr bs nzovt hvgl, ershfr gur grzcgngvba gb thrff.\nGurer fubhyq or bar-- naq cersren oyl bayl bar --boivbhf jnl gb qb vg.\nNygubhtu gung jnl znl abg or boivb hf ng svefg hayrff lbh'er Qhgpu.\nAbj vf orggre guna arire.\nNygubhtu ar ire vf bsgra orggre guna *evtug* abj.\nVs gur vzcyrzragngvba vf uneq gb rkcynva, vg'f n onq vqrn.\nVs gur vzcyrzragngvba vf rnfl gb rkcynva, vg znl or n tbbq vqrn.\nAnzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb z ber bs gubfr!"

试试吧，试试能否独立读懂这个文件里的代码 —— 对初学者来说，还是挺练脑子的呢！

它先是通过一个规则生成了一个密码表，保存在 d 这个字典中；而后，将 s 这个变量中保存

的“密文”翻译成了英文……

或许，你可以试试，看看怎样能写个函数出来，给你一段英文，你可以把它加密成跟它一样的“密

文”？

dir() 函数

你的函数，保存在模块里之后，这个函数的用户（当然也包括你），可以用 dir() 函数查看模块

中可触达的变量名称和函数名称：

In [23]:

import mycode

dir(mycode)

Out[23]: ['__builtins__',

'__cached__',

'__doc__',

'__file__',

'__loader__',

'__name__',

'__package__',

'__spec__',

'is_prime',

'say_hi']

测试驱动的开发

写一个函数，或者写一个程序，换一种说法，其实就是“实现一个算法”——而所谓的“算

法”，Wikipedia 上的定义是这样的：

In mathematics and computer science, an **algorithm** is an unambiguous specification of how to solve a class of problems. Algorithms can perform *calculation*, *data processing*, and *automated reasoning tasks*.

“算法”，其实没多神秘，就是“解决问题的步骤”而已。

在第三部分的第一章里，我们看过一个判断是否为闰年的函数：

让我们写个判断闰年年份的函数，取名为 is_leap()，它接收一个年份为参数，若是

闰年，则返回 True，否则返回 False。

根据闰年的定义：

年份应该是 4 的倍数；

年份能被 100 整除但不能被 400 整除的，不是闰年。

所以，相当于要在能被 4 整除的年份中，排除那些能被 100 整

除却不能被 400 整除的年份。

不要往回翻！现在自己动手尝试着写出这个函数？你会发现其实并不容易的……

In [3]:

def is_leap(year):

pass

第一步，跟很多人想象得不一样，第一步不是上来就开始写……

第一步是先假定这个函数写完了，我们需要验证它返回的结果对不对……

这种“通过先想办法验证结果而后从结果倒推”的开发方式，是一种很有效的方法论，叫做“Test

Driven Development”，以测试为驱动的开发。

如果我写的 is_leap(year) 是正确的，那么：

is_leap(4) 的返回值应该是 True

is_leap(200) 的返回值应该是 False

is_leap(220) 的返回值应该是 True

is_leap(400) 的返回值应该是 True

能够罗列出以上四种情况，其实只不过是根据算法“考虑全面”之后的结果 —— 但，你自己试试就知

道了，无论多简单的事，想要“考虑全面”好像并不容易……

所以，在写 def is_leap(year) 中的内容之前，我只是用 pass 先把位置占上，而后在后面

添加了四个用来测试结果的语句 —— 它们的值，现在当然都是 False …… 等我把整个函数写完

了，写正确了，那么它们的值就都应该变成 True 。

In [4]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

def is_leap(year):

pass

is_leap(4) is True

is_leap(200) is False

is_leap(220) is True

is_leap(400) is True
```

Out[4]: False

Out[4]: False

Out[4]: False

Out[4]: False

考虑到更多的年份不是闰年，所以，排除顺序大抵上应该是这样：

先假定都不是闰年；

再看看是否能被 4 整除；

再剔除那些能被 100 整除但不能被 400 整除的年份……

于是，先实现第一句：“先假定都不是闰年”：

In [13]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

def is_leap(year):

r = False

return r

is_leap(4) is True

is_leap(200) is False

is_leap(220) is True

is_leap(400) is True
```

Out[13]: False

Out[13]: True

Out[13]: False

Out[13]: False

然后再实现这部分：“年份应该是 4 的倍数”：

In [14]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

def is_leap(year):

r = False

if year % 4 == 0:

r = True

return r

is_leap(4) is True

is_leap(200) is False

is_leap(220) is True

is_leap(400) is True
```

Out[14]: True

Out[14]: False

Out[14]: True

Out[14]: True

现在剩下最后一条了：“剔除那些能被 100 整除但不能被 400 整除的年份”…… 拿一个参数值，

比如， 200 为例，

因为它能被 4 整除，所以，使 r = True ，

然后再看它是否能被 100 整除 —— 能 —— 既然如此再看它能不能被

400

整除，

如果不能，那就让 r = False ；

如果能，就保留 r 的值…… 如此这般， 200 肯定使得 r = False 。

In [17]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

def is_leap(year):

r = False

if year % 4 == 0:

r = True

if year % 100 == 0:

if year % 400 !=0:

r = False

return r

is_leap(4) is True

is_leap(200) is False

is_leap(220) is True

is_leap(400) is True
```

Out[17]: True

Out[17]: True

Out[17]: True

Out[17]: True

尽管整个过程读起来很直观，但真的要自己从头到尾操作，就可能四处出错，不信你就试试 —— 这

一页最下面添加一个单元格，自己动手从头写到尾试试……

当然，Python 内建库中的 datetime.py 模块里的代码更简洁，之前给你看过：

In [1]:

```
# cpython/Lib/datetime.py

def _is_leap(year):

return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0) _is_leap(300)
```

Out[1]: False

你自己动手，从写测试开始，逐步把它实现出来试试？—— 肯定不能允许你拷贝粘贴，哈哈。

在 Python 语言中，有专门用来“试错”的流程控制 —— 今天的绝大多数编程语言都有这种“试错语

句”。

当一个程序开始执行的时候，有两种错误可能会导致程序执行失败：

语法错误（Syntax Errors）

意外（Exceptions）

比如，在 Python3 中，你写 print i ，而没有写 print(i) ，那么你犯的是语法错误，于是，

解析器会直接提醒你，你在第几行犯了什么样的语法错误。语法错误存在的时候，程序无法启动执

行。

但是，有时会出现这种情况：语法上完全正确，但出现了意外。这种错误，都是程序已经执行之后

才发生的（Runtime Errors） —— 因为只要没有语法错误，程序就可以启动。比如，你写的是

print(11/0)：

In [2]:

print(11/0)

ZeroDivisionError Traceback (most recent call la

st)

in

----> 1 print(11/0)

ZeroDivisionError: division by zero

虽然这个语句本身没有语法错误，但，这个表达式是不能被处理的。于是，它触发了

ZeroDivisionError，这个“意外”使得程序不可能继续执行下去。

在 Python 中，定义了大量的常见“意外”，并且按层级分类：

在第三部分阅读完毕之后，可以回来重新查看以下官方文档：

https://docs.python.org/3/library/exceptions.html

BaseException

+-- SystemExit

+-- KeyboardInterrupt

+-- GeneratorExit

+-- Exception

+-- StopIteration

+-- StopAsyncIteration

+-- ArithmeticError

| +-- FloatingPointError

| +-- OverflowError

| +-- ZeroDivisionError

+-- AssertionError

+-- AttributeError

+-- BufferError

+-- EOFError

+-- ImportError

| +-- ModuleNotFoundError

+-- LookupError

| +-- IndexError

| +-- KeyError

+-- MemoryError

+-- NameError

| +-- UnboundLocalError

+-- OSError

| +-- BlockingIOError

| +-- ChildProcessError

| +-- ConnectionError

| | +-- BrokenPipeError

| | +-- ConnectionAbortedError

| | +-- ConnectionRefusedError

| | +-- ConnectionResetError

| +-- FileExistsError

| +-- FileNotFoundError

| +-- InterruptedError

| +-- IsADirectoryError

| +-- NotADirectoryError

| +-- PermissionError

| +-- ProcessLookupError

| +-- TimeoutError

+-- ReferenceError

+-- RuntimeError

| +-- NotImplementedError

| +-- RecursionError

+-- SyntaxError

| +-- IndentationError

| +-- TabError

+-- SystemError

+-- TypeError

+-- ValueError

| +-- UnicodeError

| +-- UnicodeDecodeError

| +-- UnicodeEncodeError

| +-- UnicodeTranslateError

+-- Warning

+-- DeprecationWarning

+-- PendingDeprecationWarning

+-- RuntimeWarning

+-- SyntaxWarning

+-- UserWarning

+-- FutureWarning

+-- ImportWarning

+-- UnicodeWarning

+-- BytesWarning

+-- ResourceWarning

拿 FileNotFoundError 为例 —— 当我们想要打开一个文件之前，其实应该有个办法提前验证

一下那个文件是否存在。如果那个文件并不存在，就会引发“意外”。

In [3]:

f = open('test_file.txt', 'r')

FileNotFoundError Traceback (most recent call la

st)

in

----> 1 f = open('test_file.txt', 'r')

FileNotFoundError: [Errno 2] No such file or directory: 'test_file.txt'

在 Python 中，我们可以用 try 语句块去执行那些可能出现“意外”的语句， try 也可以配合

except 、 else 、 finally 使用。从另外一个角度看， try 语句块也是一种特殊的流程控

制，“专注于当意外发生时应该怎么办？”

In [13]:

```
try:
f = open('test_file.txt', 'r')
except FileNotFoundError as fnf_error:
print(fnf_error)
```

[Errno 2] No such file or directory: 'test_file.txt'

如此这般的结果是：

当程序中的语句 f = open('test_file.txt', 'r') 因为

test_file.txt 不存在而引发意外之时， except 语句块会接管流程；而后，

又因为在 except 语句块中我们指定了 FileNotFoundError ，所以，若是

FileNotFoundError 真的发生了，那么， except 语句块中的代码，

即， print(fnf_error) 会被执行……

你可以用的试错流程还有以下变种：

```
try:
do_something()
except built_in_error as name_of_error:
do_something()
else:
do_something()
```

或者：

```
try:
    do_something()
except built_in_error as name_of_error:
    do_something()
else:
    do_something()
finally:
    do_something()
```

甚至可以嵌套：

```
try:
    do_something()
except built_in_error as name_of_error:
    do_something()
else:
    try:
        do_something()
    except built_in_error as name_of_error:
        do_something()
    ...
```

更多关于错误处理的内容，请在阅读完第三部分中与 Class 相关的内容之后，再去详细阅读以下官

方文档：

Errors and Exceptions

Built-in Exceptions

Handling Exceptions

理论上，这一章不应该套上这么大的标题：《测试驱动开发》，因为在实际开发过程中，所谓测试

驱动开发要使用更为强大更为复杂的模块、框架和工具，比如，起码使用 Python 内建库中的

unitest 模块。

在写程序的过程中，为别人（和将来的自己）写注释、写 Docstring；在写程序的过程中，为了保障

程序的结果全面正确而写测试，或者干脆在最初写的时候就考虑到各种意外所以使用试错语句块

—— 这些明明是天经地义的事情，却是绝大多数人不做的…… 因为感觉有点麻烦。

这里是“聪明反被聪明误”的最好示例长期堆积的地方。很多人真的是因为自己很聪明，所以才觉

得“没必要麻烦” —— 这就好像当年苏格拉底仗着自己记忆力无比强大甚至干脆过目不忘于是鄙视一

切记笔记的人一样。

但是，随着时间的推移，随着工程量的放大，到最后，那些“聪明人”都被自己坑死了 —— 聪明本身

搞不定工程，能搞定工程的是智慧。苏格拉底自己并没完成任何工程，是他的学生柏拉图不顾他的

嘲笑用纸笔记录了一切；而后柏拉图的学生亚里士多德才有机会受到苏格拉底的启发，写了《前分

析篇》，提出对人类影响至今的“三段论”……

千万不要因为这第三部分中所举的例子太容易而把自己迷惑了。刻意选择简单的例子放在这里，是

为了让读者更容易集中精力去理解关于自己动手写函数的方方面面 —— 可将来你自己真的动手去

做，哪怕真的去阅读真实的工程代码，你就会发现，难度还是很高的。现在的轻敌，会造成以后的

溃败。

现在还不是时候，等你把整本书都完成之后，记得回来再看这个链接：

doctest — Test interactive Python examples

unittest — Unit testing framework

可执行的 **Python** 文件

理论上来讲，你最终可以把任何一个程序，无论大小，都封装（或者囊括）到仅仅一个函数之中。

按照惯例（Convention），这个函数的名称叫做 main() ：

In [4]:

```
def routine_1():

print('Routine 1 done.')

def routine_2():
```

```
sub_routine_1()

sub_routine_2()

print('Routine 2 done.')

def sub_routine_1():

print('Sub-routine 1 done.')

def sub_routine_2():

print('Sub-routine 2 done.')

def main():

routine_1()

routine_2()

print('This is the end of the program.')

if __name__ == '__main__':

main()
```

```
Routine 1 done.

Sub-routine 1 done.

Sub-routine 2 done.

Routine 2 done.

This is the end of the program.
```

当一个模块（其实就是存有 Python 代码的 .py 文件）被导入，或者被执行的时候，这个模块的

__name__ 被设定为 __main__ 。

把一个程序整个封装到 main() 之中，而后在模块代码里加上：

if __name__ == '__main__':

main()

这么做的结果是：

1. 当 Python 文件被当作模块，被 import 语句导入时， main() 函数不被直

接运行；

2. 当 Python 文件被 python -m 执行的时候， main() 才被执行。

还记得那个 Python 的彩蛋吧？ this.py 的代码如下：

In [ ]:

s = """Gur Mra bs Clguba, ol Gvz Crgref

Ornhgvshy vf orggre guna htyl.

Rkcyvpvg vf orggre guna vzcyvpvg.

Fvzcyr vf orggre guna pbzcyrk.

Pbzcyrk vf orggre guna pbzcyvpngrq.

Syng vf orggre guna arfgrq.

Fcnefr vf orggre guna qrafr.

Ernqnovyvgl pbhagf.

Fcrpvny pnfrf nera'g fcrpvny rabhtu gb oernx gur ehyrf.

Nygubhtu cenpgvpnyvgl orngf chevgl.

Reebef fubhyq arire cnff fvyragyl.

Hayrff rkcyvpvgyl fvyraprq.

Va gur snpr bs nzovthvgl, ershfr gur grzcgngvba gb thrff.

Gurer fubhyq or bar-- naq cersrenoyl bayl bar --boivbhf jnl gb qb vg.

Nygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er Qhgpu.

Abj vf orggre guna arire.

Nygubhtu arire vf bsgra orggre guna *evtug* abj.

Vs gur vzcyrzragngvba vf uneq gb rkcynva, vg'f n onq vqrn.

Vs gur vzcyrzragngvba vf rnfl gb rkcynva, vg znl or n tbbq vqrn.

Anzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!"""

```
d = {}
for c in (65, 97):
for i in range(26):
d[chr(i+c)] = chr((i+13) % 26 + c)
print("".join([d.get(c, c) for c in s]))
```

所以，只要 import this， this.py 中的代码就被执行：

In [ ]:

```
import this
```

我在当前目录下，保存了一个文件 that.py ，它的内容如下 —— 其实就是把 this.py 之中的代码封

装到 main() 函数中了：

In [2]:

 # %load that.py

def main():

s = """Gur Mra bs Clguba, ol Gvz Crgref

Ornhgvshy vf orggre guna htyl.

Rkcyvpvg vf orggre guna vzcyvpvg.

Fvzcyr vf orggre guna pbzcyrk.

Pbzcyrk vf orggre guna pbzcyvpngrq.

Syng vf orggre guna arfgrq.

Fcnefr vf orggre guna qrafr.

Ernqnovyvgl pbhagf.

Fcrpvny pnfrf nera'g fcrpvny rabhtu gb oernx gur ehyrf.

Nygubhtu cenpgvpnyvgl orngf chevgl.

Reebef fubhyq arire cnff fvyragyl.

Hayrff rkcyvpvgyl fvyraprq.

Va gur snpr bs nzovthvgl, ershfr gur grzcgngvba gb thrff.

Gurer fubhyq or bar-- naq cersrenoyl bayl bar --boivbhf jnl gb qb vg

.

Nygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er Qhgpu.

Abj vf orggre guna arire.

Nygubhtu arire vf bsgra orggre guna *evtug* abj.

Vs gur vzcyrzragngvba vf uneq gb rkcynva, vg'f n onq vqrn.

Vs gur vzcyrzragngvba vf rnfl gb rkcynva, vg znl or n tbbq vqrn.

Anzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!"""

```
d = {}

for c in (65, 97):

for i in range(26):

d[chr(i+c)] = chr((i+13) % 26 + c)

print("".join([d.get(c, c) for c in s]))

if __name__ == '__main__':

main()
```

于是，当你在其它地方导入它的时候， import that ， main() 函数的内容不会被执行：

In [ ]:

```
import that
```

但是，你在命令行中，用 python that.py ，或者 python -m that 将 that.py 当作可执行模块运行的

时候， main() 就会被执行 —— 注意，不要写错， python -m that.py 会报错的 —— 有 -m 参数，

就不要写文件尾缀 .py ：

In [ ]:

```
%%bash
```

python that.py

In [ ]:

%%bash

python -m that

像 that.py 那样把整个程序放进 main() 函数之后， import that 不会自动执行 main 函数里的代

码。不过，你可以调用 that.main() ：

In [ ]:

import that

that.main()

当然， that.py 之中没有任何 Docstring，所以 help(that) 的结果是这样的：

In [ ]:

import that

help(that)

所以，之前那个从 37 万个词汇中挑出 3700 个字母加起来等于 100 的词汇的程序，也可以写成以

下形式：

In [8]:

#!/usr/bin/env python

def sum_of_word(word):

```
sum = 0

for char in word:

sum += ord(char) - 96

return sum

def main(wordlist, result):

with open(result, 'w') as result:

with open(wordlist, 'r') as file:

for word in file.readlines():

if sum_of_word(word.strip()) == 100:

result.write(word)

if __name__ == '__main__':

main('words_alpha.txt', 'results.txt')
```

至于以上代码中的第一行， #!/usr/bin/env python 是怎么回事儿，建议你自己动手解决一下，去

Google：

python3 script executable

你会很快弄明白的……

另外，再搜索一下：

python3 script executable parameters retrieving

你就可以把以上程序改成在命令行下能够接收指定参数的 Python 可执行文件……

顺带说， import this 的彩蛋有更好玩的玩法：

In [6]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

import this

love = this

this **is** love *# True*

love **is True** *# False*

love **is False** *# False*

love **is not True or False** *# True* love **is not True or False**; love **is** love *# True True* Out[6]: True

Out[6]: False

Out[6]: False

Out[6]: True

Out[6]: True

Out[6]: True

在 Terminal 里输入 python

而后在 Interactive Shell 里逐句输入试试。 love = this 后面的每一

句，都是布林运算，想想看为什么是那样的结果？

import this

love = this

this **is** love

 # True, 试试看， *id(this)* 和 *id(love)* 是同一个值

即，它们的内存地址相同

love **is** True

 # False, id(love) 和 *id(True)* 不是同一个值

love **is** False

同上

love **is not** True **or** False

 # is not 的优先级比 *or* 高；所以相当于是：

 # (love is not True) or False，于是返回 *True*

love **is not** True **or** False; love **is** love

重复一次上一句 —— `;` 是语句分隔符

而后 *love is love* 当然是 *True*

注意以下代码中， id() 函数的输出结果：

In [9]:

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import this

love = this

this **is** love

love **is True**

love **is False**

love **is not True or False**

love **is not True or False**; love **is** love id(love)

id(this)

id(**True**)

id(**False**)

love **is not True**

Out[9]: True

Out[9]: False

Out[9]: False

Out[9]: True

Out[9]: True

Out[9]: True

Out[9]: 4345330968

Out[9]: 4345330968

Out[9]: 4308348176

Out[9]: 4308349120

Out[9]: True

Python 的操作符优先级，完整表格在这里：

Operator precedence

Python 的更多彩蛋：

Python Easter Eggs

刻意思考

随着时间的推移，你会体会到它的威力：

刻意思考哪儿需要刻意练习

只不过是一句话而已，却竟然知道与不知道之间终究会形成天壤之别的差异，也是神奇。

刻意思考，就是所谓的琢磨，琢磨这事儿，一旦开始就简单得要死，可无从下手的时候就神秘无

比。让我们再看一个“刻意思考” —— 即，琢磨 —— 的应用领域：

这东西能用在哪儿呢？

很多人学了却没怎么练，有一个很现实的原因 —— 没什么地方用得上。

这也怪我们的应试教育，大学前上 12 年学，“学”（更多是被逼的）的绝大多数东西，只有一个能够

切实体会到的用处，考试 —— 中考、高考，以及以它们为目标的无数“模考”…… 于是，反过来，不

管学什么东西，除了考试之外，几乎无法想象其他的用处。

一旦我们启动了对某项技能的自学之后，在那过程中，最具价值的刻意思考就是，时时刻刻琢磨“这

东西能用在哪儿呢？”

比如，当你看到字符串的 Methods 中有一个 str.zfill() 的时候，马上就能

想到，“嗯！这可

以用来批量更名文件……”

虽然现在的 Mac OS 操作系统里已经有相当不错的批量更名工具内建在 Finder 之中（选中多个文件

之后，在右键菜单中能看到 rename 命令），但，这是近期才加进去的功能，几年前却没有 ——

也就是说，几年前的时候，有人可以用 str.zfill() 写个简单的程序完成自己的工作，而另外

一些人仅因为操作系统没有提供类似的功能就要么手工做，要么干脆忍着忘了算了……

但，更多的时候，需要你花时间去琢磨，才能找到用处。

找到用处，有时候还真挺难的 —— 因为人都一样，容易被自己的眼界所限，放眼望过去，没有用

处，自然也就不用了，甚至不用学了，更不用提那就肯定是感觉不用练了……

所以，仔细想想罢 —— 那些在学校里帮老师干活的小朋友们，更多情况下还真不是很多人以为

的“拍马屁”（不排除肯定有哈），只不过是在“主动找活干”……

找活干，是应用所学的最有效方式，有活干，所以就有问题需要解决，所以就有机会反复攻关，在

这个过程中，以用带练……

所以，很多人在很多事儿上都想反了。

人们常常取笑那些呼哧呼哧干活的人，笑着说，“能者多劳”，觉得他们有点傻。

这话真的没错。但，这么说更准：劳者多能 —— 你看，都想反了吧？

到最后，一切自学能力差的人，外部的表现都差不多，都起码包括这么一条：眼里没活。他们也不

喜欢干活，甚至也没想过，玩乐也是干活（每次逢年过节玩得累死那种） —— 从消耗或者成本的角

度来看根本没啥区别 —— 只不过那些通常都是没有产出的活而已。

在最初想不出有什么用处的时候，还可以退而求其次，看看“别人想出什么用处没有？” —— 比如，

我去 Google best applications of python skill ，在第一个页面我就发现了这么篇文

章：“What exactly can you do with Python? ”，翻了一会儿觉得颇有意思……

再高阶一点的刻意思考（琢磨），无非是在“这东西能用在哪儿呢？”这句话里加上一个字而已：

这东西还能用在哪儿呢？

我觉得这个问题对思维训练的帮助非常深刻 —— 别看只是多了一个字而已。

当我读到在编程的过程中有很多的“约定”的时候，就琢磨着：

哦，原来约定如此重要……

哦，原来竟然有那么多人不重视约定……

哦，原来就应该直接过滤掉那些不遵守约定的人…… —— 那这个原理（东西）

还能用在哪儿呢？ —— 哦，在生活中也一样，遇到不遵守约定的人或事，直接

过滤，不要浪费自己的生命……

学编程真的很有意思，因为这个领域是世界上最聪明的人群之一开辟出来并不断共同努力着发展

的，所以，在这个世界里有很多思考方式，琢磨方式，甚至可以干脆成为“做事哲学”的东西，可以

普遍应用在其它领域，甚至其它任何领域。

比如，在开发方法论中，有一个叫做 MoSCoW Method 的东西，1994 年由 Clegg Dai 在《 Case Method Fast-Track: A RAD Approach》一书中提出的 —— 两个 o 字母放在那里，是为了能够把

这个缩写读出来，发音跟莫斯科一样。

简单说，就是，凡事儿都可以分为：

Must have

Should have

Could have

Won't have

于是，在开发的时候，把所谓的需求打上这 4 个标签中的某一个，以此分类，就很容易剔除掉那些

实际上做了还不如不做的功能……

琢磨一下罢，这个东西还可以用在什么地方？

显然，除了编程之外，其他应用领域挺多的，这个原则相当地有启发性……

我写书就是这样的。在准备的过程中 —— 这个过程比绝大多数人想象得长很多 —— 我会罗列所有

我能想到的相关话题……　等我觉得已经再也没有什么可补充的时候，再为这些话题写上几句话构成

大纲……　　这时候就会发现很多话题其实应该是同一个话题。如此这般，一次扩张，一次搜索之后，

就会进行下一步，应用 MoSCow 原则，给这些话题打上标签 —— 在这过程中，总是发现很多之前

感觉必要的话题，其实可以打上 Won't have 的标签，于是，把它们剔除，然后从 Must

have 开始写起，直到 Should have ，至于 Could have 看时间是否允许，看情况，比如，

看有没有最后期限限制……

在写书这事儿上，我总是给人感觉很快，事实上也是，因为有方法论 —— 但显然，那方法论不是从

某一本“如何写书”的书里获得的，而是从另外一个看起来完全不相关的领域里习得后琢磨到的……

所谓的“活学活用”，所谓的“触类旁通”，也不过如此。

战胜难点

无论学什么，都有难点。所谓的“学习曲线陡峭”，无非就是难点靠前、难点很多、难点貌似很难而

已。

然而，相信我，所有的难点，事实上都可以被拆解成更小的单元，而后再逐一突破的时候，就没那

么难了。逐一突破全部完成之后，再拼起来重新审视的时候就会发现那所谓的难常常只不过是错

觉、幻觉而已 —— 我把它称为困难幻觉。

把一切都当作手艺看的好处之一就是心态平和，因为你知道那不靠天分和智商，它靠的就一件事

儿，不混时间，刻意思考、刻意练习 —— 其实吧，老祖宗早就有总结：

天下无难事，只怕有心人……

大家都是人，咋可能没“心”呢？

想成为有心人，其实无非就是学会拆解之后逐一突破，就这么简单。

第三部分所所用的例子依然非常简单 —— 这当然是作者的刻意；但是，所涉及的话题都是被认

为“很难很难”，“很不容易理解”，“初学者就没必要学那些了”之类的话题：

类，以及面向对象编程（Class，OOP）

迭代器、生成器、装饰器（Iterators、Generators、Decorators）

正则表达式（Regular Expressions）

巴科斯-诺尔范式（Backus Normal Form）

尤其是最后一个，巴科斯-诺尔范式，几乎所有的编程入门书籍都不会提到……

然而，这些内容，在我设计《自学是门手艺》内容的过程中，是被当作 Must have ， 而不是

Should have ， 当然更不是 Could have 或者 Won't have 的。

它们属于 Must have 的原因也很简单：

无论学什么都一样，难的部分不学会，就等于整个没学。

—— 仅因为不够全面。

有什么必要干前功尽弃的事情呢？要么干脆别学算了，何必把自己搞成一个半吊子？—— 可惜，这

偏偏是绝大多数人的习惯，学什么都一样，容易的部分糊弄糊弄，困难的部分直接回避…… 其实，

所有焦虑，都是这样在许多年前被埋下、在许多年后生根发芽，茂盛发达 —— 你想想看是不是如

此？

虽然别人认为难，你刚开始也会有这样的错觉，但，只要你开始施展“读不懂也要读完，读完之后再

读很多遍”的手段，并且还“不断自己动手归纳总结整理”，你就会“发现”，其实没什么大不了的，甚

至你会有错觉：

“突然”之间一切都明了了！

那个“突然”，其实就是阿基米德的 Eureka，那个他从澡堂里冲出来大喊大叫的词汇。

其实吧，泡澡和冥想，还真是最容易产生 Eureka 状态的两种活动。原理在于，泡澡和打坐的时

候，大脑都极其放松，乃至于原本相互之间并无联系的脑神经突出之间突然产相互关联；而那结果

就是日常生活中所描述的“融会贯通”，于是，突然之间，Eureka！

自学者总是感觉幸福度很高，就是这个原因。日常中因为自学，所以总是遇到更多的困难。又因为

这些东西不过是手艺，没什么可能终其一生也解决不了，恰恰相反，都是假以时日必然解决的“困

难”……　　于是，自学者恰恰因为遇到的“困难”多，所以才有更多遇到“Eureka”的可能性，那种幸福，

还真的难以表述，即便表述清楚了，身边的人也难以理解，因为自学者就是很少很少。

对很多人来说，阅读的难点在于起初的时候它总是显得异常枯燥。

刚识字、刚上学的时候，由于理解能力有限，又由于年龄的关系于是耐心有限，所以，那时需要老

师耐心陪伴、悉心引导。这就好像小朋友刚出生的时候，没有牙齿，所以只能喝奶差不多…… 然

而，到了一定程度之后一定要断奶，是不是？可绝大多数人的实际情况却是，小学的时候爱上了“奶

嘴”（有人带着阅读），而后一生没有奶嘴就吃不下任何东西。

他们必须去“上课”，需要有人给他们讲书。不仅如此，讲得“不生动”、“不幽默”还不行；就算那职业

提供奶嘴的人（这是非常令人遗憾的真相：很多的时候，所谓的“老师”本质上只不过就是奶妈而

已）帅气漂亮、生动幽默、尽职尽力…… 最终还是一样的结果 —— 绝大多数人依然没有完整掌握

所有应该掌握的细节。

开始“自学”的活动，本质上来看，和断奶其实是一回事儿。

知识就是知识，它没有任何义务去具备幽默生动的属性；

手艺就是手艺，它没有任何义务去具备有趣欢乐的属性。

幽默与生动，是要自己去扮演的角色；有趣与欢乐，是要自己去挖掘的幸福 —— 它们从来都并不自

动包含在知识和手艺之中。只有当它们被“有心人”掌握、被“有心人”应用、甚至被“有心人”拿去创造

的时候，也只有“有心人”才能体会到那幽默与生动、那有趣与欢乐。

所以，有自学能力的人，不怕枯燥 —— 因为那本来就理应是枯燥的。这就好像人生本无意义，有意

义的人生都是自己活出来的一样，有意义的知识都是自己用出来的 —— 对不用它的人，用不上它的

人来说，只能也只剩下无法容忍的枯燥。

能够耐心读完那么多在别人看来“极度枯燥”的资料，是自学者的擅长。可那在别人看来“无以伦比”的

耐心，究竟是哪儿来的呢？如何造就的呢？没断奶的人想象不出来。其实也很简单，首先，平静地

接受了它枯燥的本质；其次，就是经过多次实践已然明白，无论多枯燥，总能读完；无论多难，多

读几遍总能读懂..... 于是，到最后，只不过是习惯了而已。

第三部分关于编程的内容过后，还有若干关于自学的内容。

在反复阅读编程部分突破难点的过程之中、过程之后，你会对那些关于自学的内容有更深更生动的

认识。很多道理过去你都知道是对的，只不过因为没有遇到过生动的例子 —— 主要是没遇到过能让

自己感到生动的例子 —— 于是你就一直没有重视起来，于是，就还是那句话，那一点点的差异，造

成了后来那么大的差距。

然而，已然知道了真相的你，以后就再也没办法蒙蔽自己了 —— 这就是收获，这就是进步。

类 —— 面向对象编程

面向对象编程

注意：当前这一小节所论述的内容，不是专属于哪个编程语言（比如 Python、Javascript 或者

Golang）。

面向对象编程（Object Oriented Programming, OOP）是一种编程的范式（Paradigm），或者说，

是一种方法论（Methodology）—— 可以说这是个很伟大的方法论，在我看来，现代软件工程能做

那么复杂的宏伟项目，基本上都得益于这个方法论的普及。

争议

现在，OOP 的支持者与反对者在数量上肯定不是一个等级，绝大多数人支持 OOP 这种编程范式。

但是，从另外一个角度，反对 OOP 的人群中，牛人比例更高 —— 这也是个看起来颇为吊诡的事

实。

比如，Erlang 的发明者，Joe Armstrong) 就很讨厌 OOP，觉得它效率低下。他用的类比也确实令

人忍俊不禁，说得也挺准的：

支持 OOP 的语言的问题在于，它们总是随身携带着一堆并不明确的环境 —— 你明

明只不过想要个香蕉，可你所获得的是一个大猩猩手里拿着香蕉…… 以及那大猩猩

身后的整个丛林！

—— Coders at Work

创作 UTF-8 和 Golang 的的程序员 Rob Pike，更看不上 OOP，在 2004 年的一个讨论帖里直接把

OOP 比作 “Roman numerals of computing” —— 讽刺它就是很土很低效的东西。八年后又挖坟把

一个 Java 教授写的 OOP 文章嘲弄了一番：“也不知道是什么脑子，认为写 6 个新的 Class 比直接

用 1 行表格搜索更好？”

Paul Graham) —— 就是那个著名的 Y-Combinator 的创始人 —— 也一样对 OOP 不以为然，在

Why Arc isn't Especially Objedct-Oriented 中，说他认为 OOP 之所以流行，就是因为平庸程序员

（Mediocre programers）太多，大公司用这种编程范式去阻止那帮家伙，让他们捅不出太大的娄

子……

然而，争议归争议，应用归应用 —— 就好像英语的弊端不见得比其他语言少，可就是最流行，那怎

么办呢？用呗 —— 虽然该抱怨的时候也得抱怨抱怨。

从另外一个角度望过去，大牛们如此评价 OOP 也是很容易理解的 —— 因为他们太聪明，又因为他

们太懒得花时间去理解或容忍笨蛋…… 我们不一样，最不一样的地方在于，我们不仅更多容忍他

人，而且更能够容忍自己的愚笨，所以，视角就不同了，仅此而已。

并且，上面两位大牛写的编程语言，现在也挺流行，Joe Armstrong 的 Erlang 和 Rob Pike 的

Golang，弄不好早晚你也得去学学，去用用……

基本术语

面向对象编程（OOP），是使用对象（Objects）作为核心的编程方式。进而就可以把对象

（Objects）的数据和运算过程封装（Encapsulate）在内部，而外部仅能根据事先设计好的界

面（Interface）与之沟通。

比如，你可以把灯泡想象成一个对象，使用_灯泡_的人，只需要与_开关_这个界面（Interface）打

交道，而不必关心灯泡内部的设计和原理 —— 说实话，这是个很伟大的设计思想。

生活中，我们会遇到无数有意无意应用了这种设计思想的产品 —— 并不仅限于编程领域。你去买个

车回来，它也一样是各种封装之后的对象。当你转动方向盘（操作界面）的时候，你并不需要关心

汽车设计者是如何做到把那个方向盘和车轮车轴联系在一起并如你所愿去转向的；你只需要知道的

是，逆时针转动方向盘是左转，而顺时针转动方向盘是右转 —— 这就可以了！

在程序设计过程中，我们常常需要对标现实世界创造对象。这时候我们用的最直接手段就是抽

象（Abstract）。抽象这个手段，在现实中漫画家们最常用。为什么你看到下面的图片觉得它们俩

看起来像是人？尤其是在你明明知道那肯定不是人的情况下，却已然接受那是两个漫画小人的形

象？

这种描绘方法，就是抽象，很多“没必要”的细节都被去掉了（或者反过来说，没有被采用），留下

的两个特征，一个是头，一个是双眼 —— 连那双“眼睛”都抽象到只剩下一个黑点了……

这种被保留下来的“必要的特征”，叫做对象的属性（Attributes），进而，这些抽象的对象，既然

是“人”的映射，它们实际上也能做一些抽象过后被保留下来的“必要的行为”，比如，说话，哭笑，这

些叫做对象的方法（Methods）。

从用编程语言创造对象的角度去看，所谓的界面，就由这两样东西构成：

属性 —— 用自然语言描述，通常是名词（Nouns）

方法 —— 用自然语言描述，通常是动词（Verbs）

从另外一个方面来看，在设计复杂对象的时候，抽象到极致是一种必要。

我们为生物分类，就是一层又一层地抽象的过程。当我们使用“生物”这个词的时候，它并不是某一

个特定的我们能够指称的东西…… 然后我们开始给它分类……

所以，当我们在程序里创建对象的时候，做法常常是

先创建最抽象的类（Class）

然后再创建 子类（Subclass）……

它们之间是从属关系是：

Class

Subclass

在 OOP 中，这叫继承（Inheritance）关系。比如，狗这个对象，就可以是从哺乳动物这个对象_继

承_过来的。如果哺乳动物有“头”这个属性（Attributes），那么在狗这个对象中就没必要再重新定义

这个属性了，因为既然狗是从哺乳动物继承过来的，那么它就拥有哺乳动物的所有属性……

每当我们创建好一个类之后，我们就可以根据它创建它的许多个实例（Instances）。比如，创建好

了“狗”这个类之后，我们就可以根据这个类创建很多条狗…… 这好多条狗，就是狗这个类的实例。

现在能把这些术语全部关联起来了吗？

对象，封装，抽象

界面，属性，方法

继承，类，子类，实例

这些就是关于“面向对象编程”方法论的最基本的术语 —— 无论在哪种编程语言里，你都会频繁地遇

到它们。

对象、类，这两个词，给人的感觉是经常被通用 —— 习惯了还好，但对有些初学者来说，就感觉那

是生命不能承受之重。——　这次不是英文翻译中文时出现的问题，在英文世界里，这些词的互通使

用和滥用也使相当一部分人（我怀疑是大部分人）最终掌握不了　OOP 这个方法论。

细微的差异在于“视角”的不同。

之前提到函数的时候，我用的说辞是，

你写了一个函数，而后你要为这个产品的使用者写说明书……

—— 当然，产品使用者之中也包括未来的你……

类（Class）这个东西也一样，它也有创作者和使用者。

你可以这样分步理解：

你创造了一个类（Class），这时候你是创作者，从你眼里望过去，那就是个类

（Class）；

而后你根据这个类的定义，创建了很多实例（Instances）；

接下来一旦你开始使用这些实例的时候，你就成了使用者，从使用者角度望过

去，手里正在操作的，就是各种对象（Objects）……

最后，补充一下，不要误以为所有的 Classes 都是对事物（即，名词）的映射 —— 虽然大多数情

况下确实如此。

对基本概念有了一定的了解之后，再去看 Python 语言是如何实现的，就感觉没那么难了。

类 **—— Python** 的实现

既然已经在不碰代码的情况下，把 OOP 中的主要概念梳理清楚了，以下的行文中，那些概念就直

接用英文罢，省得理解上还得再绕个弯……

Defining Class

Class 使用 class 关键字进行定义。

与函数定义不同的地方在于，Class 接收参数不是在 class Classname(): 的括号里完成 ——

那个圆括号有另外的用处。

让我们先看看代码，而后再逐一解释：

In [3]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

import datetime

class Golem:

def __init__(self, name=**None**):

self.name = name

self.built_year = datetime.date.today().year

def say_hi(self):

print('Hi!')

g = Golem('Clay')

g.name

g.built_year

g.say_hi

g.say_hi()

type(g)

type(g.name)

type(g.built_year)

type(g.__init__)

type(g.say_hi)

Out[3]: 'Clay'

Out[3]: 2019

Out[3]: >

Hi!

Out[3]: __main__.Golem

Out[3]: str

Out[3]: int

Out[3]: method

Out[3]: method

以上，我们创建了一个 Class:

class Golem:

def __init__(self, name=None):

self.name = name

self.built_year = datetime.date.today().year

其中定义了当我们根据这个 Class 创建一个实例的时候，那个 Objedct 的初始化过程，即

__init__() 函数 —— 又由于这个函数是在 Class 中定义的，我们称它为 Class 的一个

Method。

这里的 self 就是个变量，跟程序中其它变量的区别在于，它是一个系统默认可以识别的变量，

用来指代将来用这个 Class 创建的 Instance。

比如，我们创建了 Golem 这个 Class 的一个 Instance， g = Golem('Clay') 之后，我们写

g.name ，那么解析器去就去找 g 这个实例所在的 Scope 里有没有 self.name ……

注意： self 这个变量的定义，是在 def __init__(self, ...) 这一句里完成的。对于这个

变量的名称取名没有强制要求，你实际上可以随便用什么名字，很多 C 程序员会习惯于将这个变量

命名为 this —— 但，根据惯例，你最好还是只用 self 这个变量名，省得给别人造成误会。

在 Class 的代码中，如果定义了 __init__() 函数，那么系统就会将它当作用来 Instance 在创

建后被初始化的函数。这个函数名称是强制指定的，初始化函数必须使用这个名称；注意 init

两端各有两个下划线 _ 。

当我们用 g = Golem('Clay') 这一句创建了一个 Golam 的 Instance 的时候，以下一连串的事

情发生了：

g 从此之后就是一个根据 Golem 这个 Class 创建的 Instance，对使用者来

说，它就是个 Object；

因为 Golem 这个 Class 的代码中有 __init__() ，所以，当 g 被创建的时

候， g 就需要被初始化……

在 g 所在的变量目录中，出现了一个叫做 self 的用来指代 g 本身的变

量；

self.name 接收了一个参数， 'Clay' ，并将其保存了下来；

生成了一个叫做 self.built_year 的变量，其中保存的是 g 这个 Object

被创建时的年份……

对了，Golem 和 Robot 一样，都是机器人的意思；Golem 的本义来自于犹太神话，一个被赋予了

生命的泥人……

Inheritance

我们刚刚创建了一个 Golem Class，如果我们想用它 Inherite 一个新的 Class，比

如， Running_Golem ，一个能跑的机器人，那就像以下的代码那样做 —— 注意 class

Running_Golem 之后的圆括号：

In [1]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import datetime

class Golem:

def __init__(self, name=None):

self.name = name

self.built_year = datetime.date.today().year

def say_hi(self):

print('Hi!')

class Running_Golem(Golem): # 刚刚就说，这个圆括号另有用途 ……

def run(self):

print("Can't you see? I'm running...")

rg = Running_Golem('Clay')

rg.run

rg.run()

rg.name

rg.built_year

rg.say_hi()
```

```
Can't you see? I'm running...

Hi!
```

如此这般，我们根据 Golem 这个 Class 创造了一个 Subclass —— Running_Golem，既然它是

Golem 的 Inheritance，那么 Golem 有的 Attributes 和 Methods 它都有，并且还多了一个 Method

—— self.run 。

Overrides

当我们创建一个 Inherited Class 的时候，可以重写（Overriding）Parent Class 中的 Methods。比

如这样：

In [2]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import datetime

class Golem:

def __init__(self, name=None):

self.name = name

self.built_year = datetime.date.today().year

def say_hi(self):

print('Hi!')

class runningGolem(Golem):

def run(self):

print("Can't you see? I'm running...")
```

def say_hi(self): # 不再使用 *Parent Class* 中的定

义，而是新的 ……

print('Hey! Nice day, Huh?')

rg = runningGolem('Clay')

rg.run

rg.run()

rg.name

rg.built_year

rg.say_hi()

Out[2]:

6f9358>>

Can't you see? I'm running...

Out[2]: 'Clay'

Out[2]: 2019

Hey! Nice day, Huh?

Inspecting A Class

当我们作为用户想了解一个 Class 的 Interface，即，它的 Attributes 和 Methods 的时候，常用的有

三种方式：

1. help(object)

2. dir(object)

3. object.__dict__

In [3]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import datetime

class Golem:

def __init__(self, name=None):

self.name = name

self.built_year = datetime.date.today().year

def say_hi(self):

print('Hi!')

class runningGolem(Golem):

def run(self):

print('Can\' t you see? I\' m running...')

def say_hi(self): # 不再使用 Parent Class 中的定

义，而是新的 ……

print('Hey! Nice day, Huh?')

rg = runningGolem('Clay')

help(rg)

dir(rg)

rg.__dict__
```

```
hasattr(rg, 'built_year')
Help on runningGolem in module __main__ object:
class runningGolem(Golem)
| runningGolem(name=None)
|
| Method resolution order:
| runningGolem
| Golem
| builtins.object
|
| Methods defined here:
|
| run(self)
|
| say_hi(self)
|
| ----------------------------------------------------------------------
--
| Methods inherited from Golem:
|
```

| __init__(self, name=None)

| Initialize self. See help(type(self)) for accurate signature.

|

| --

--

| Data descriptors inherited from Golem:

|

| __dict__

| dictionary for instance variables (if defined)

|

| __weakref__

| list of weak references to the object (if defined)

Out[3]: ['__class__',

'__delattr__',

'__dict__',

'__dir__',

'__doc__',

'__eq__',

'__format__',

'__ge__',

'__getattribute__',

'__gt__',

'__hash__',

'__init__',

'__init_subclass__',

'__le__',

'__lt__',

'__module__',

'__ne__',

'__new__',

'__reduce__',

'__reduce_ex__',

'__repr__',

'__setattr__',

'__sizeof__',

'__str__',

'__subclasshook__',

'__weakref__',

'built_year',

'name',

'run',

'say_hi']

Out[3]: {'name': 'Clay', 'built_year': 2019}

Out[3]: True

Scope

每个变量都属于某一个 **Scope**（变量的作用域），在同一个 Scope 中，变量可以被引用被操

作…… 这么说非常抽象，难以理解 —— 只能通过例子说明。

我们先给 Golem 这个 Class 增加一点功能 —— 我们需要随时知道究竟有多少个 Golem 处于活跃

状态…… 也因此顺带给 Golem 加上一个 Method： cease() —— 哈！机器人么，想关掉它，说

关掉它，就能关掉它；

另外，我们还要给机器人设置个使用年限，比如 10 年；

…… 而外部会每隔一段时间，用 Golem.is_active() 去检查所有的机器人，所以，不需要外

部额外操作，到了年头，它能应该关掉自己。—— 当然，又由于以下代码是简化书写的，核心目的

是为了讲解 Scope，所以并没有专门写模拟 10 年后某些机器人自动关闭的情形……

在运行以下代码之前，需要先介绍三个 Python 的内建函数：

hasattr(object, attr) 查询这个 object 中有没有 这个 attr ， 返

回布尔值

getattr(object, attr) 获取这个 object 中这个 attr 的值

setattr(object, attr, value) 将这个 object 中的 attr 值设置

为 value

现在的你，应该一眼望过去，就已经能掌握这三个内建函数的用法
—— 还记得之前的你吗？眼睁睁

看着，那些字母放在那里对你来说没任何意义…… 这才多久啊！

In [17]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import datetime

class Golem:

population = 0

__life_span = 10

def __init__(self, name=None):

self.name = name

self.built_year = datetime.date.today().year

self.__active = True

Golem.population += 1

def say_hi(self):

print('Hi!')

def cease(self):
```

self.__active = **False**

Golem.population -= 1 # 执行一遍之后，试试把这句改成 *population += 1*

def is_active(self):

if datetime.date.today().year - self.built_year >= Golem.__life_span:

self.cease()

return self.__active

g = Golem()

hasattr(Golem, 'population') *# True*

hasattr(g, 'population') *# True*

hasattr(Golem, '__life_span') *# False*

hasattr(g, '__life_span') *# False*

hasattr(g, '__active') *# False*

Golem.population *# 1*

setattr(Golem, 'population', 10)

Golem.population *# 10*

x = Golem()

Golem.population *# 11*

x.cease()

Golem.population *# 10*

getattr(g, 'population') *# 10*

g.is_active()

Out[17]: True

Out[17]: True

Out[17]: False

Out[17]: False

Out[17]: False

Out[17]: 1

Out[17]: 10

Out[17]: 11

Out[17]: 10

Out[17]: 10

Out[17]: True

如果你试过把第 17 行的 Golem.population += 1 改成 population += 1，你会被如下信

息提醒：

16 self.__active = True

---> 17 population += 1

UnboundLocalError: local variable 'population' referenced before

assignment

—— 本地变量 population 尚未赋值，就已经提前被引用…… 为什么会这样呢？因为在你所创

建 g 之后，马上执行的是 __init()__ 这个初始化函数，而 population 是在这个函数之外

定义的……

如果你足够细心，你会发现这个版本中，有些变量前面有两个下划线 __ ，比如， __life_span

和 self.__active 。这是 Python 的定义，变量名前面加上下划线（Underscore）一个以上 _

的话，那么该变量是“私有变量”（Private Variables），不能被外部引用。而按照 Python 的惯例，

我们会使用两个下划线起始，去命名私有变量，如： __life_span 。你可以回去试试，把所有

的 __life_span 改成 _life_span （即，变量名开头只有一个 _ ，那

么， hasattr(Golem, '_life_span') 和 hasattr(g, '_life_span') 的返回值就都变

成了 True 。

看看下面的图示，理解起来更为直观一些：

整个代码启动之后，总计有 4 个 Scopes 如图所示：

① class Golem 之外；

② class Golem 之内；

③ __init__(self, name=None) 之内；

④ cease(self) 之内；

在 Scope ① 中，可以引用 Golem.population ，在生成一个 Golem 的 实例 g 之后，也可以

引用 g.population ；但， Golem.__life_span 和 g.__active 在 Scope ① 是不存在

的；

在 Scope ② 中，存在两个变量， population 和 __life_span ；而 __life_span 是

Private（私有变量，因为它的变量名中前两个字符是下划线 __ ；于是，在 Scope ① 中，不存在

Golem.__life_span —— hasattr(Golem, '__life_span') 的值为 False ；

在 Scope ③ 中和 Scope ④ 中，由于都给它们传递了 self 这个参数，于是，在这两个 Scope

里，都可以引用 self.xxx ，比如 self.population ，比如 self.__life_span ；

在 Scope ③ 中， population 是不存在的，如果需要引用这个值，可以用

Golem.population ，也可以用 self.population 。同样的道理，在 Scope ③ 中

__life_span 也不存在，如果想用这个值，可以用 Golem.__life_span 或者

self.__life_span ；

Scope ④ 与 Scope ③ 平行存在。所以在这里， population 和 __life_span 也同样并不存

在。

Encapsulation

到目前为止，Golem 这个 Class 看起来不错，但有个问题，它里面的数据，外面是可以随便改的

—— 虽然，我们已经通过给变量 life_span 前面加上两个下划线，变成 __life_span ，使其称为

私有变量，外部不能触达（你不能引用 Golem.__life_span ），可 Golem.population 就不一

样，外面随时可以引用，还可以随时修改它，只需要写上一句：

Golem.population = 1000000

我们干脆把 population 这个变量也改成私有的罢： __population ，而后需要从外界查看这

个变量的话，就在 Class 里面写个函数，返回那个值好了：

In [4]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import datetime

class Golem:

__population = 0

__life_span = 10

def __init__(self, name=None):

self.name = name

self.built_year = datetime.date.today().year

self.__active = True

Golem.__population += 1

def say_hi(self):
```

print('Hi!')

def cease(self):

self.__active = **False**

Golem.__population -= 1

def is_active(self):

if datetime.date.today().year - self.built_year >= Golem.__life_

span:

self.cease

return self.__active

def population(self):

return Golem.__population

g = Golem('Clay')

g.population

g.population()

Out[4]:

>

Out[4]: 1

如果，你希望外部能够像获得 Class 的属性那样，直接写 g.population，而不是必须加上一个

括号 g.population() 传递参数（实际上传递了一个隐含的 self 参数），那么可以在 def

population(self): 之前的一行加上一句 @property：

class Golem:

__population = 0

...

@property

def population(self):

return Golem.__population

如此这般之后，你就可以用 g.population 了：

In [5]:

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import datetime

class Golem:

__population = 0

__life_span = 10

def __init__(self, name=**None**):

self.name = name

self.built_year = datetime.date.today().year

self.__active = **True**

Golem.__population += 1

def say_hi(self):

```
print('Hi!')

def cease(self):

self.__active = False

Golem.__population -= 1

def is_active(self):

if datetime.date.today().year - self.built_year >= Golem.__life_

span:

self.cease

return self.__active

@property

def population(self):

return Golem.__population

g = Golem('Clay')

g.population

# g.population = 100

Out[5]: 1
```

如此这般之后，不仅你可以直接引用 g.population ，并且，在外部不能再直接给

g.population 赋值了，否则会报错：

```
---------------------------------------------------------------

---------
```

AttributeError Traceback (most recent

call last)

in

26 g = Golem('Clay')

27 g.population

---> 28 g.population = 100

AttributeError: can't set attribute

到此为止，Encapsulation 就做得不错了。

如果你非得希望从外部可以设置这个值，那么，你就得再写个函数，并且在函数之前加上一句：

...

@property

def population(self):

return Golem.__population

@population.setter

def population(self, value):

Golem.__population = value

这样之后， .population 这个 Attribute 就可以从外部被设定其值了（虽然在当前的例子中显得

没必要让外部设定 __population 这个值…… 以下仅仅是为了举例）：

In [1]:

```
from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

import datetime

class Golem:

__population = 0

__life_span = 10

def __init__(self, name=None):

self.name = name

self.built_year = datetime.date.today().year

self.__active = True

Golem.__population += 1

def say_hi(self):

print('Hi!')

def cease(self):

self.__active = False

Golem.__population -= 1

def is_active(self):

if datetime.date.today().year - self.built_year >= Golem.__life_

span:

self.cease

return self.__active
```

```
@property
def population(self):
return Golem.__population
@population.setter
def population(self, value):
Golem.__population = value
g = Golem('Clay')
g.population
g.population = 100
ga = Golem('New')
g.population
ga.population
help(Golem)
Golem.__dict__
g.__dict__
hasattr(Golem, 'population')
getattr(Golem, 'population')
setattr(Golem, 'population', 10000)
g.population  # 所以，在很多的情况下，不把数据封装在 Class 内部的话，后面会很有很多
麻烦。
```

```
Help on class Golem in module __main__:

class Golem(builtins.object)
 | Golem(name=None)
 |
 | Methods defined here:
 |
 | __init__(self, name=None)
 | Initialize self. See help(type(self)) for accurate signature.
 |
 | cease(self)
 |
 | is_active(self)
 |
 | say_hi(self)
 |
 | ----------------------------------------------------------------------
--
 | Data descriptors defined here:
 |
 | __dict__
```

| dictionary for instance variables (if defined)

|

| __weakref__

| list of weak references to the object (if defined)

|

| population

Out[1]: 10000

函数工具

这一章要讲的是迭代器、生成器，和装饰器，这些都是函数工具。有人把它们称为

DIG（Decorator, Iterator，Generator）—— 它们都是真正掌握 Python 的关键。

迭代器（**Iterator**）

我们已经见过 Python 中的所有容器，都是可迭代的 —— 准确地讲，是可以通过迭代遍历每一个元

素：

In [5]:

string = "this is a string."

list = ['item 1', 'item 2', 3, 5]

set = (1, 2, 3, 4, 5)

for c **in** string:

```
print(c, end=', ')

print()

for L in list:

print(L, end=', ')

print()

for s in set:

print(s, end=', ')

print()
```

t, h, i, s, , i, s, , a, , s, t, r, i, n, g, .,

item 1, item 2, 3, 5,

1, 2, 3, 4, 5,

有个内建函数，就是用来把一个对象转换成“可迭代对象”的 —— iter() 。

In [17]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

i = iter("Python")

type(i)

s = iter((1, 2, 3, 4, 5))

type(s)

L = iter(['item 1', 'item 2', 3, 5])
```

type(L)

Out[17]: str_iterator

Out[17]: tuple_iterator

Out[17]: list_iterator

可迭代对象如何使用呢？有个 next() 函数：

In [26]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

i = iter("Python")

next(i)

next(i)

next(i)

next(i)

next(i)

next(i)

# next(i) 前面已经到 'n' 了，再调用就会有 StopIteration 错误提示。
```

Out[26]: 'P'

Out[26]: 'y'

Out[26]: 't'

Out[26]: 'h'

Out[26]: 'o'

Out[26]: 'n'

在 i 这个迭代器里一共有 6 个元素，所以， next(i) 在被调用 6 次之后，就不能再被调用了，

一旦再被调用，就会触发 StopIteration 错误。

那我们怎么自己写一个迭代器呢？

迭代器是个 Object，所以，写迭代器的时候写的是 Class，比如，我们写一个数数的迭代

器，Counter：

In [85]:

```
class Counter(object):

def __init__(self, start, stop):

self.current = start

self.stop = stop

def __iter__(self):

return self

def __next__(self):

if self.current > self.stop:

raise StopIteration

else:

c = self.current

self.current += 1
```

return c

c = Counter(11, 20)

next(c)

next(c)

next(c)

for c **in** Counter(101, 105):

print(c, sep=', ')

type(Counter)

Out[85]: 11

Out[85]: 12

Out[85]: 13

101

102

103

104

105

Out[85]: type

这里的重点在于两个函数的存在， __iter__(self) 和 __next__(self) 。

def __iter__(self):

return self

这两句是约定俗成的写法，写上它们， Counter 这个类就被会被识别为 Iterator 类型。而后再有

__next__(self) 的话，它就是个完整的迭代器了。除了可以用 for loop 之外，也可以用

while loop 去遍历迭代器中的所有元素：

In [64]:

```
class Counter(object):

def __init__(self, start, stop):

self.current = start

self.stop = stop

def __iter__(self):

return self

def __next__(self):

if self.current > self.stop:

raise StopIteration

else:

c = self.current

self.current += 1

return c

for c in Counter(101, 103):

print(c, sep=', ')
```

c = Counter(201, 203)

while True:

try:

print(next(c), sep=', ')

except StopIteration:

break

101

102

103

201

202

203

生成器（**Generator**）

那用函数（而不是 Class）能不能写一个 Counter 呢？答案是能，用生成器（Generator）就行。

In [67]:

def counter(start, stop):

while start <= stop:

yield start

start += 1

for i **in** counter(101, 105):

print(i)

101

102

103

104

105

哎呀！怎么感觉这个简洁很多呢？

不过，是否简洁并不是问题，这次看起来用生成器更简单，无非是因为当前的例子更适合用生成器

而已。在不同的情况下，用迭代器和用生成器各有各的优势。

这里的关键在于 yield 这个语句。它和 return 最明显的不同在于，在它之后的语句依然会被

执行 —— 而 return 之后的语句就被忽略了。

但，正因为这个不同，在写迭代器的时候，只能用 yield ，而没办法使用 return —— 你现在

可以回去把上面代码中的 yield 改成 return 看看，然后体会一下它们之间的不同。

生成器函数被 next() 调用后，执行到 yield 生成一个值返回（然后继续执行剩余的语句）；

下次再被 next() 调用的时候，从上次生成返回值的 yield 语句处继续执行…… 如果感觉费

解，就多读几遍 —— 而后再想想若是生成器中有多个 yield 语句会是什么情况？

还有一种东西，叫做生成器表达式。先看个例子：

In [84]:

even = (e **for** e **in** range(10) **if not** e % 2)

 # odd = (o for o in range(10) if o % 2)

print(even)

for e **in** even:

print(e)

at 0x107cc0048>

0

2

4

6

8

其实，这种表达式我们早就在 List Comprehension 里见过 —— 那就是通过生成器表达式完成的。

注意

仔细看 even = (e for e in range(10) if not e % 2) 中最外面那层括号，用了圆括

号， even 就是用生成器创造的迭代器（Iterator），若是用了方括号，那就是用生成器创造的列表

（List）—— 当然用花括号 {} 生成的就是集合（Set）……

In [77]:

```
# even = (e for e in range(10) if not e % 2)
odd = [o for o in range(10) if o % 2]
print(odd)
for o in odd:
print(o)
```

[1, 3, 5, 7, 9]

1

3

5

7

9

In [78]:

```
# even = (e for e in range(10) if not e % 2)
odd = {o for o in range(10) if o % 2}
print(odd)
for o in odd:
print(o)
```

{1, 3, 5, 7, 9}

1

3

5

7

9

生成器表达式必须在括号内使用（参见官方__HOWTOS），包括函数的参数括号，比如：

In [81]:

sum_of_even = sum(e **for** e **in** range(10) **if not** e % 2) print(sum_of_even)

20

In [ ]:

函数内部当然可以包含其它的函数，以下就是一个函数中包含着其它函数的结构示例：

def a_func():

def b_func():

pass

def c_func():

pass

def d_func():

pass

b_func()

return True

想象一下，如果，我们让一个函数返回的是另外一个函数呢？我们一步一步来：

In [2]:

```
def a_func():

def b_func():

print("Hi, I'm b_func!")

print("Hi, I'm a_func!")

a_func()
```

Hi, I'm a_func!

In [3]:

```
def a_func():

def b_func():

print("Hi, I'm b_func!")

print("Hi, I'm a_func!")

b_func()

a_func()
```

Hi, I'm a_func!

Hi, I'm b_func!

上一个代码，我们可以写成这样 —— 让 a_func() 将它内部的 b_func() 作为它的返回值：

In [5]:

def a_func():

def b_func():

print("Hi, I'm b_func!")

print("Hi, I'm a_func!")

return b_func()

a_func()

Hi, I'm a_func!

Hi, I'm b_func!

如果我们在 return 语句里只写函数名呢？好像这样：

In [6]:

def a_func():

def b_func():

print("Hi, I'm b_func!")

print("Hi, I'm a_func!")

return b_func

a_func()

Hi, I'm a_func!

Out[6]: .b_func()>

这次返回的不是调用 b_func() 这个函数的执行结果，返回的是 b_func 这个函数本身。

装饰器（**Decorator**）

函数也是对象

这是关键：

函数本身也是对象（即，Python 定义的某个 Class 的一个 Instance）。

于是，函数本身其实可以与其它的数据类型一样，作为其它函数的参数或者返回值。

让我们分步走 —— 注意，在以下代码中， a_decorator 返回的一个函数的调用 wrapper()

而不是 wrapper 这个函数本身：

In [7]:

```
def a_decorator(func):

def wrapper():

print('We can do sth. before a func is called...')

func()

print('... and we can do sth. after it is called...')

return wrapper()

def a_func():

print("Hi, I'm a_func!")

a_func()

a_decorator(a_func)
```

Hi, I'm a_func!

We can do sth. before a func is called...

Hi, I'm a_func!

... and we can do sth. after it is called...

如果返回的是函数本身， wrapper ，输出结果跟你想的并不一样：

In [8]:

```
def a_decorator(func):

def wrapper():

print('We can do sth. before a func is called...')

func()

print('... and we can do sth. after it is called...')

return wrapper #

def a_func():

print("Hi, I'm a_func!")

a_func()

a_decorator(a_func)
```

Hi, I'm a_func!

Out[8]: .wrapper()>

装饰器操作符

不过，Python 提供了一个针对函数的操作符 @ ，它的作用是…… 很难一下子说清楚，先看看以下

代码：

In [9]:

```
def a_decorator(func):

def wrapper():

print('We can do sth. before calling a_func...')

func()

print('... and we can do sth. after it was called...')

return wrapper

@a_decorator

def a_func():

print("Hi, I'm a_func!")

a_func()

We can do sth. before calling a_func...

Hi, I'm a_func!

... and we can do sth. after it was called...
```

注意：以上的代码中， a_decorator(func) 返回的是 wrapper 这个函数本身。

在我们定义 a_func() 的时候，在它之前，加上了一句 @a_decorator ；这么做的结果是：

每次 a_func() 在被调用的时候，因为它之前有一句 @a_decorator ，所以它

会先被当作参数传递到 a_decorator(func) 这个函数中…… 而后，真正的执

行，是在 a_decorator() 里被完成的。

—— 被 @ 调用的函数，叫做“装饰器”（Decorator），比如，以上代码中的

a_decorator(func) 。

现在可以很简单直接地说清楚装饰器的作用了：

@a_decorator

def a_func():

...

等价与

def a_func():

...

a_func = a_decorator(a_func)

就是用 a_decorator 的调用结果替换掉原来的函数。 a_decorator 返回值是什么，以后调用

a_func 时就是在调用这个返回值，而 a_decorator 本身此时已经执行完毕了。

装饰器的用途

Decorator　最常用的场景是什么呢？最常用的场景就是用来改变其它函数的行为。

In [46]:

def an_output():

return 'The quick brown fox jumps over the lazy dog.'

print(an_output())

The quick brown fox jumps over the lazy dog.

In [47]:

```
def uppercase(func):
def wrapper():
original_result = func()
modified_restult = original_result.upper()
return modified_restult
return wrapper
@uppercase
def an_output():
return 'The quick brown fox jumps over the lazy dog.'
print(an_output())
```

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.

你还可以给一个函数加上一个以上的装饰器：

In [50]:

```
def uppercase(func):
def wrapper():
original_result = func()
modified_restult = original_result.upper()
return modified_restult
```

return wrapper

def strong(func):

def wrapper():

original_result = func()

modified_restult = ' '**+original_result+**' '

return modified_restult

return wrapper

@strong

@uppercase

def an_output():

return 'The quick brown fox jumps over the lazy dog.'

print(an_output())

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.

你把两个装饰器的顺序调换一下写成下面这样试试：

@uppercase

@strong

def an_output():

...

装饰器的执行顺序是“自下而上” —— 其实是“由里到外”更为准确。体会一下。

装饰带有参数函数

到现在我们见到的使用装饰器的函数都是没有参数的： an_output 以及之前的 a_func 。

如果被装饰的函数有参数怎么办？装饰器自身内部又应该怎么写？

这时候，Python 的 *args and **kwargs 的威力就显现出来了 —— 之前怕麻烦没有通过仔细

反复阅读搞定这“一个星号、两个星号、直接晕倒”的知识点的人，现在恐怕要吃亏了……

装饰器函数本身这么写：

```
def a_decorator(func):

def wrapper(*args, **kwargs):

return original_result

# ...

return wrapper
```

在这里， (*args, **kwargs) 非常强大，它可以匹配所有函数传进来的所有参数…… 准确地

讲， *args 接收并处理所有传递进来的位置参数， **kwargs 接收并处理所有传递进来的关键

字参数。

假设我们有这么个函数：

In [54]:

```
def say_hi(greeting, name=None):

return greeting + '! ' + name + '.'
```

print(say_hi('Hello', 'Jack'))

Hello! Jack.

如果我们想在装饰器里对函数名、参数，都做些事情 —— 比如，我们写个 @trace 用来告诉用户

调用一个函数的时候都发生了什么……

In [65]:

def trace(func):

def wrapper(*args, **kwargs):

print(f"Trace: You've called a function: **{func.__name__}**(),",

f"with args: **{args}**; kwargs: **{kwargs}**")

orginal_result = func(*args, *kwargs)

print(f"Trace: **{func.__name__}{args}** returned: **{orginal_result}**"

)

return orginal_result

return wrapper

@trace

def say_hi(greeting, name=**None**):

return greeting + '! ' + name + '.'

print(say_hi('Hello', 'Jack'))

Trace: You've called a function: say_hi(), with args: ('Hello', 'Jack');

kwargs: {}

Trace: say_hi('Hello', 'Jack') returned: Hello! Jack.

Hello! Jack.

有了以上的基础知识之后，再去阅读 Python Decorator Library 的 Wiki 页面就会轻松许多：

https://wiki.python.org/moin/PythonDecoratorLibrary

学会装饰器究竟有多重要？

装饰器一定要学会 —— 因为很多人就是不会。

Oreilly.com 上有篇文章，《5 reasons you need to learn to write Python decorators》中，其中的第

五条竟然是：**Boosting your career**!

Writing decorators isn't easy at first. It's not rocket science, but takes enough effort to learn, and to grok the nuances involved, that many developers will never go to the trouble to master it. And that works to your advantage. When you become the person on your team who learns to write decorators well, and write decorators that solve real problems, other developers will use them. Because once the hard work of writing them is done, decorators are so easy to use. This can massively magnify the positive impact of the code you write. And it just might make you a hero, too.

As I've traveled far and wide, training hundreds of working software engineers to use Python more effectively, teams have consistently reported writing decorators to be one of the most valuable and important tools they've learned in my advanced Python programming workshops.

为什么有那么多人就是学不会呢？——　只不过是因为在此之前，遇到 *args **kwargs 的时

候，“一个星号、两个星号、直接晕倒”…… 而后并未再多挣扎一下。

正则表达式

正则表达式本质上是个独立的语言，短小却格外强悍 —— 乃至于，如果你竟然没学会它的话，你的

之前学的编程技能干脆与残疾无异。

Wikipedia 上对正则表达式的说明如下：

正则表达式（英语：Regular Expression，在代码中常简写为 regex、regexp 或

RE），又称正规表示式、正规表示法、正规运算式、规则运算式、常规表示法，是

计算机科学的一个概念。正则表达式使用单个字符串来描述、匹配一系列符合某个

句法规则的字符串。在很多文本编辑器里，正则表达式通常被用来检索、替换那些

符合某个模式的文本。许多程序设计语言都支持利用正则表达式进行字符串操作。

例如，在 Perl 中就内建了一个功能强大的正则表达式引擎。正则表达式这个概念最

初是由 Unix 中的工具软件（例如 sed 和 grep）普及开的。

以下是绝大多数翻译成中文的教程中对正则表达式进行讲解时所使用的描述：

一个正则表达式（Regular Expression）通常被称为一个模式（Pattern）。

我常常觉得当初要是它被翻译成“规则表达式”，那么很可能初学者不会感到那么大的压力 —— 谁都

一样，看着由“每个都认识的字构成的词组”却不能直观地想到它究竟是什么东西，都会感到莫名的

压力。

Regular，其实在它的众多语义中，取以下释义最符合 Regular Expression 的原意[1]：

Linguistics 规则的

regular verbs 规则动词

而 **Pattern** 这个词，在词典里有好几个对应的中文词汇：

① 图案；② 式样；③ 图样；④ 榜样；⑤ 模式；⑥ 样品；⑦ 模子

在当前语境之下，把 Pattern 翻译成“模式”，显然不如“模子”更好（甚至连“样品”感觉都比“模式”更恰

当）—— “模子”这个词很直观啊，拿着一个模子去找与它一致的字符串…… “与规则一致”，英文用

的是 **Match**，一般被翻译作“匹配”。

在自学编程的过程中，处处都是这种语言翻译带来的迷惑、障碍，或者耽误。既然应该把 Regular

Expression 理解为“规则表达式”更好，那其实吧，把 Pattern 直接理解为中文的“规则”，可能更直观

更准确，理解上更是毫无障碍：

一个规则表达式（Regular Expression）通常被称为一个规则（Pattern）。

那么，规则表达式里写的是什么呢？只能是规则了…… 到最后好像也就“捕获”（Capture）这个词没

什么歧义。

现在，我们已经把术语全部“解密”了，然后再看看下面的表述：

我们可以用书写特定的规则，用来在文本中捕获与规则一致的字符串，而后对其进

行操作……

理解起来相当顺畅。

以下的 Python 代码中， \wo\w 就是一个规则表达式（或称为规则）；

而 re.findall(pttn, str) 的作用就是，在 str 里找到所有与这个规则（Pattern，模

式）一致（Match，匹配）的字符串：

In [226]:

import re

str = 'The quick brown fox jumps over the lazy dog'

pttn = re.compile(r'\wo\w')

re.findall(pttn, str)

Out[226]: ['row', 'fox', 'dog']

总结一下：

规则表达式（Regular Expressions，通常缩写为 Regex）是最强大且不可或缺的文

本处理工具 —— 它的用处就是在文本中扫描/搜索（Scan/Search）与某一规

则（Pattern）匹配（Match，即，与规则一致）的所有实例，并且还可以按照规

则捕获（Capture）其中的部分或者全部，对它们进行替换

（Replece）。

接下来为了避免歧义，我们干脆用 Regex 这个缩写，以及与它相关的英文单词：pattern, match,

capture, replace(ment)……

有时，使用 Regex 并不是为了 Replace，而是为了检查格式，比如，可以用 Regex 检查用户输入

的密码是否过于简单（比如，全部都由数字构成），比如可以用来验证用户输入的电话号码、证件

号码是否符合特定格式等等。

另外，在自学的过程中，想尽一切办法把一切术语用简单直白的“人话”重新表述，是特别有效的促

进进步的行为模式。

视觉体验

所谓百闻不如一见。

眼见为实 —— 想办法让一个陌生的概念视觉上直观，是突破大多学习障碍的最简单粗暴直接有效的

方式。

我们最好先直接看看 Regex 的工作过程。以下，是用微软发行的代码编辑工具 Visual Studio Code

针对一小段文本使用若干条 Regex 进行匹配的过程：

Python 的项目代码仓库里有一个很简短的 Demo 程序，叫 redemo.py ，它使用 Tcl/Tk 作为图形

界面，也可以用来测试正则表达式。

它的代码地址是：

https://raw.githubusercontent.com/python/cpython/master/Tools/demo/redemo

它运行起来长成这样：

目前（2019）网上最方便的 Regex 测试器，是 regex101.com：

以下，就是在一段文本中，找出所有首写字母大写的词汇的过程，并将其先全部替换成小写，再将

其全部替换为大写的过程；使用的正则表达式是 ([A-Z])\w+ ，替换表达式分别是 \L$1 和

\U$1 ：

这个网站太好了，所以，平日里我是用 Nativefier 工具将这个网站打包为一个 Mac Desktop App 使

用。不过，它也有局限，就是被搜索文件略微大点就报错，说 timeout ……

准备工作

我们需要个文本文件，用来当作练习使用正则表达式去搜索替换的目标。这个文件保存在当前的根

目录，文件名称是： regex-target-text-sample.txt 。

以下代码中， pttn = r'beg[iau]ns?' 这一句中的 beg[iau]ns? 就是 Regex 的 Pattern。

注意：在 Python 代码中，写 Pattern 的时候，之所以要在字符串 '...' 之前加上 r ，写成

r'...' ，是因为如果不用 raw string 的话，那么，每个转义符号都要写成 \\

；如果用 raw

string，转义符号就可以直接使用 \ 本身了…… 当然，如果你想搜索 \ 这个符号本身的话，那么

还是得写 \\ 。

而 re.findall(pttn, str) 的意思是说，把 str 中所有与 pttn 这个规则一致的字符串

都找出来：

In [5]:

import re

with open('regex-target-text-sample.txt', 'r') **as** f:

str = f.read()

pttn = r'beg[iau]ns?'

re.findall(pttn, str)

Out[5]: ['begin', 'began', 'begun', 'begin']

文件 regex-target-text-sample.txt 中的内容如下：

<**ol**>

<**li**><**pre**>begin began begun bigins beginingpre>li>

<**li**><**pre**>google gooogle goooogle gooooooglepre>li>

<**li**><**pre**>coloured color coloring colouring coloredpre>li

>

<**li**><**pre**>never ever verb however everestpre>li>

<**li**><**pre**>520 52000 5200000 520000000 520000000000pre>li>

<**li**><**pre**>error wonderer achroiocythaemia achroiocythemiapre

>li>

<**li**><**pre**>The white dog wears a black hat. pre>li>

<**li**><**pre**>Handel, Händel, Haendelpre>li>

ol>

<**dl**>(843) 542-4256dl> <**dl**>(431) 270-9664dl>

<**dl**>3336741162dl> <**dl**>3454953965dl>

<**ul**>

<**li**>peoplesr@live.comli> <**li**>jaxweb@hotmail.comli>

<**li**>dhwon@comcast.netli> <**li**>krueger@me.comli>

ul>

<**h3**>URLsh3>

https://docs.python.org/3/howto/regex.html

https://docs.python.org/3/library/re.html

<**h3**>passwordsh3>

Pasw0rd~

i*Eh,GF67E

a$4Bh9XE&E

<**h3**>duplicate wordsh3>

<**p**>It's very very big. p>

<**p**>Keep it simple, simple, simple! p>

在以下的示例中，有时直接设定了 str 的值，而不是使用以上整个文本文件 —— 因为读者在阅读的

时候，最好能直接看到被搜索的字符串。另外，如果使用整个文件，所得到的 Match 太多，也确实

影响阅读。

优先级

毕竟，你已经不是“啥都不懂”的人了。你已经知道一个事实：编程语言无非是用来运算的。

所谓的运算，就有操作符（Operators）和操作元（Operands）—— 而操作符肯定是有优先级的，

不然的话，那么多操作元和操作符放在一起，究竟先操作哪个呢？

Regex 也一样，它本身就是个迷你语言（Mini Language）。在 Regex 中，操作符肯定也有优先

级。它的操作元有个专门的名称，原子（Atom）。

先大致看看它的操作符优先级，你就会对它有相当不错的了解：

排

原子与操作符优先级

从高到低

列

)

1

转义符号 (Escaping Symbol)

\

分组、捕获 (Grouping or

(...) (?:...) (?=...) (?!...) (? <=...) (?

2

Capturing)

3

数量 (Quantifiers)

a* a+ a? a{n, m}

序列与定位（Sequence and

4

abc ^ $ \b \B

Anchor

5

或（Alternation）

a|b|c

6

原子 (Atoms)

a [^abc] \t \r \n \d \D \s \S \w \W .

当然，你若是在之前，没有自学过、理解过 Python （或者任何其它编程语言）表达式中的操作符

优先级，那么一上来就看上面的表格不仅对你没有帮助，只能让你更迷惑。

—— 这就是理解能力逐步积累逐步加强的过程。

原子

在 Regex 的 Pattern 中，操作元，即，被运算的“值”，被称为原子（Atom）。

本义字符

最基本的原子，就是本义字符，它们都是单个字符。

本义字符包括从 a 到 z， A 到 Z， 0 到 9，还有 _ —— 它们所代表的就是它们的字面

值。

即，相当于， string.ascii_letters 和 string.digits 以及 _ 。

In [50]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import string

string.ascii_letters

string.digits
```

Out[50]: 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

Out[50]: '0123456789'

以下字符在 Regex 中都有特殊含义：

\ + * . ? - ^ $ | ( ) [ ] { } < >

当你在写 Regex 的时候，如果你需要搜索的字符不是本义字符，而是以上这些特殊字符时，_建

议_都直接加上转义符号 \ 来表示，比如，你想搜索 ' ，那你就写 \' ，或者你想搜索 # 那你

就写 \# （事实上， # 并不是 Regex 的特殊符号，所以，它之前的转义符号可有可无） —— 这

对初学者来说可能是最安全的策略。

跟过往一样，所有的细节都很重要，它们就是需要花时间逐步熟悉到牢记。

集合原子

集合原子还是原子。

标示集合原子，使用方括号 [] 。 [abc] 的意思是说，“ a or b or c ”，即，abc 中的任意一

个字符。

比如， beg[iau]n 能够代表 begin 、 began ， 以及 begun 。

In [25]:

import re

str = 'begin began begun bigins begining'

pttn = r'beg[iau]n'

re.findall(pttn, str)

Out[25]: ['begin', 'began', 'begun', 'begin']

在方括号中，我们可以使用两个操作符： -（区间）和 ^（非）。

[a-z] 表示从小写字母 a 到小写字母 z 中的任意一个字符。

[^abc] 表示 abc 以外的其它任意字符，即，非 [abc] 。

注意，一个集合原子中， ^ 符号只能用一次，只能紧跟在 [之后。否则不起作用。

类别原子

类别原子，是指那些能够代表“一类字符”的原子，它们都得使用转义符号再加上另外一个符号表

达，包括：

\d 任意数字；等价于 [0-9]

\D 任意非数字；等价于 [^0-9]

\w 任意本义字符；等价于 [a-zA-z0-9_]

\W 任意非本义字符；等价于 [^a-zA-z0-9_]

\s 任意空白；相当于 [\f\n\r\t\v] （注意，方括号内第一个字符是空格符号）

\S 任意非空白；相当于 [^ \f\n\r\t\v] （注意，紧随 ^ 之后的是一个空格符号）

. 除 \r \n 之外的任意字符；相当于 [^\r\n]

类别原子挺好记忆的，如果你知道各个字母是哪个词的首字母的话：

d 是 digits

w 是 word characters

s 是 spaces

另外，在空白的集合 [\f\n\r\t\v] 中： f 是分页符； \n \r 是换行符； \t 是制表

符； \v 是纵向制表符（很少用到）。各种关于空白的转义符也同样挺好记忆的，如果你知道各个

字母是那个词的首字母的话：

f 是 flip

n 是 new line

r 是 return

t 是 tab

v 是 vertical tab

In [8]:

import re

str = '

(843) 542-4256

(431) 270-9664

'

pttn = r'\d\d\d\-'

re.findall(pttn, str)

Out[8]: ['542-', '270-']

边界原子

我们可以用边界原子指定边界。也可以称作“定位操作符”。

^ 匹配被搜索字符串的开始位置；

$ 匹配被搜索字符串的结束位置；

\b 匹配单词的边界； er\b ， 能匹配 wonderer 中的 er ， 却不能匹配 error 中的 er ；

\B 匹配非单词边界； er\B ， 能匹配 error 中的 er ， 却不能匹配 wonderer 中的 er 。

In [26]:

import re

str = 'never ever verb however everest'

pttn = r'er\b'

re.findall(pttn, str)

pttn = r'er\B'

re.findall(pttn, str)

Out[26]: ['er', 'er', 'er']

Out[26]: ['er', 'er']

注意： ^ 和 $ 在 Python 语言中被 \A 和 \Z 替代。

事实上，每种语言或多或少都对 Regex 有自己的定制。不过，本章讨论的绝大多数细节，都是通用

的。

组合原子

我们可以用圆括号 () 将多个单字符原子组合成一个原子 —— 这么做的结果是， () 内的字符串

将被当作一整个原子，可以被随后我们要讲解的数量操作符操作。

另外， () 这个操作符，有两个作用：组合（Grouping），就是我们刚刚讲到的作用；而另外一个

作用是捕获（Capturing)，后面会讲到。

注意区别， er 、 [er] 和 (er)) 各不相同。

er 是两个原子， 'e' 和紧随其后的 'r'

[er] 是一个原子，或者 'e' 或者 'r' ；

(er) 是一个原子， 'er'

下一节中讲到数量操作符的时候，会再次强调这点。

数量操作符

数量操作符有： + ? * {n, m} 。

它们是用来限定位于它们之前的原子允许出现的个数；不加数量限定则代表出现一次且仅出现一

次：

+ 代表前面的原子必须至少出现一次，即： 出现次数 ≧ 1

例如， go+gle 可以匹配 google gooogle goooogle 等;

? 代表前面的原子最多只可以出现一次，即： 0 ≦ 出现次数 ≦ 1

例如， colou?red 可以匹配 colored 或者 coloured ;

* 代表前面的原子可以不出现，也可以出现一次或者多次，即： 出现次数 ≧ 0

例如， 520* 可以匹配 520 52000 5200000 520000000000 等。

{n} 之前的原子出现确定的 n 次；

{n,} 之前的原子出现至少 n 次；

{n, m} 之前的原子出现至少 n 次，至多 m 次

例如， go{2,5}gle ，能匹配 google gooogle goooogle 或

gooooogle ，但不能匹配 gogle 和 gooooooogle

In [2]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
import re
with open('regex-target-text-sample.txt', 'r') as f:
str = f.read()
pttn = r'go+gle'
re.findall(pttn, str)
pttn = r'go{2,5}gle'
re.findall(pttn, str)
pttn = r'colou?red'
re.findall(pttn, str)
pttn = r'520*'
re.findall(pttn, str)
```

Out[2]: ['google', 'gooogle', 'goooogle', 'gooooogle']

Out[2]: ['google', 'gooogle', 'goooogle']

Out[2]: ['coloured', 'colored']

Out[2]: ['520', '52000', '5200000', '520000000', '520000000000']

数量操作符是对它之前的原子进行操作的，换言之，数量操作符的操作元是操作符之前的原子。

上一节提到，要注意区别： er 、 [er] 和 (er) 各不相同。

er 是两个原子， 'e' 之后 'r'

[er] 是一个原子，或者 'e' 或者 'r' ；

(er) 是一个原子， 'er'

In [27]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import re

str = 'error wonderer severeness'

pttn = r'er'

re.findall(pttn, str)

pttn = r'[er]'

re.findall(pttn, str)

pttn = r'(er)'

re.findall(pttn, str)
```

Out[27]: ['er', 'er', 'er', 'er']

Out[27]: ['e', 'r', 'r', 'r', 'e', 'r', 'e', 'r', 'e', 'e', 'r', 'e', 'e']

Out[27]: ['er', 'er', 'er', 'er']

在以上的例子中，看不出 er 和 (er) 的区别，但是，加上数量操作符就不一样了 —— 因为数

量操作符只对它之前的那一个原子进行操作：

In [28]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

str = 'error wonderer severeness'

pttn = r'er+'

re.findall(pttn, str)

pttn = r'[er]+'

re.findall(pttn, str)

pttn = r'(er)+'

re.findall(pttn, str)

Out[28]: ['err', 'er', 'er', 'er']

Out[28]: ['err', 'r', 'erer', 'e', 'ere', 'e']

Out[28]: ['er', 'er', 'er']

或操作符 |

或操作符 | 是所有操作符中优先级最低的，数量操作符的优先级比它高，所以，在 | 前后的原

子被数量操作符（如果有的话）操作之后才交给 | 操作。

于是， begin|began|begun 能够匹配 begin 或 began 或 begun 。

In [29]:

import re

str = 'begin began begun begins beginn'

pttn = r'begin|began|begun'

re.findall(pttn, str)

Out[29]: ['begin', 'began', 'begun', 'begin', 'begin']

在集合原子中（即， [] 内的原子）各个原子之间的关系，只有“或”——相当于方括号中的每个原

子之间都有一个被省略的 | 。

注意：中括号的 | 不被当作特殊符号，而是被当作 | 这个符号本身。在中括号中的圆括号，也

被当作圆括号 () 本身，而无分组含义。

In [1]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

import re

str = 'achroiocythaemia achroiocythemia a|e'

pttn = r'[a|ae]'

re.findall(pttn, str)

pttn = r'[a|e]'

re.findall(pttn, str)

pttn = r'[ae]'

re.findall(pttn, str)

pttn = r'[(ae)]'

re.findall(pttn, str)

pttn = r'[a|ae|(ae)]'

re.findall(pttn, str)

Out[1]: ['a', 'a', 'e', 'a', 'a', 'e', 'a', 'a', '|', 'e']

匹配并捕获

捕获（Capture），使用的是圆括号 () 。使用圆括号得到的匹配的值被暂存成一个带有索引的列

表，第一个是 $1 ，第二个是 $2 …… 以此类推。随后，我们可以在替换的过程中使用 $1 $2

中所保存的值。

注意：在 Python 语言中调用 re 模块之后，在 re.sub() 中调用被匹配的值，用的索引方法是

\1 、 \2 …… 以此类推。

In [23]:

import re

str = 'The white dog wears a black hat.'

pttn = r'The (white|black) dog wears a (white|black) hat.'

re.findall(pttn, str)

repl = r'The \2 dog wears a \1 hat.'

re.sub(pttn, repl, str)

repl = r'The \1 dog wears a \1 hat.'

re.sub(pttn, repl, str)

Out[23]: [('white', 'black')]

Out[23]: 'The black dog wears a white hat.'

Out[23]: 'The white dog wears a white hat.'

非捕获匹配

有时，你并不想捕获圆括号中的内容，在那个地方你使用括号的目的只是分组，而非捕获，那么，

你就在圆括号内最开头加上 ?: —— (?:...) ：

In [24]:

import re

str = 'The white dog wears a black hat.'

pttn = r'The (?:white|black) dog wears a (white|black) hat.'

re.findall(pttn, str) # 只捕获了一处，也就是说只有一个值将来

可以被引用

repl = r'The \1 dog wears a \1 hat.' # 不过，可替换的位置却有两个（即便非捕

获匹配，也有可替换位置）

re.sub(pttn, repl, str)

Out[24]: ['black']

Out[24]: 'The black dog wears a black hat.'

需要注意的是，虽然非匹配捕获不将匹配值暂存以便随后替换时调用，但，匹配处依然是将来可被

替换的位置。

在 Python 代码中使用正则表达式，匹配和捕获以及随后的替换，有更灵活的方式，因为可以对那

些值直接编程。 re.sub() 中， repl 参数甚至可以接收另外一个函数作为参数 —— 以后你肯

定会自行认真阅读以下页面中的所有内容：

https://docs.python.org/3/library/re.html

非捕获匹配，还有若干个操作符：

(?=pattern)

正向肯定预查（look ahead positive assert），在任何匹配规则的字符串开始处匹配

查找字符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例

如， Windows(?=95|98|NT|2000) %60) 能匹配 Windows2000 中

的 Windows ，但不能匹配 Windows3.1 中的 Windows 。预查不消耗字符，也就

是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的

搜索，而不

是从包含预查的字符之后开始。

(?!pattern)

正向否定预查（negative assert），在任何不匹配规则的字符串开始处匹配查找字

符串。这是一个非获取匹配，也就是说，该匹配不需要获取供以后使用。例

如 Windows(?=95|98|NT|2000)) 能匹配 Windows3.1 中的 Windows ，但不

能匹配 Windows2000 中的 Windows 。预查不消耗字符，也就是说，在一个匹配

发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字

符之后开始。

(? <=pattern)

反向（look behind）肯定预查，与正向肯定预查类似，只是方向相反。例如， (?

<=95|98|NT|2000)Windows Windows) 能匹配 2000Windows 中

的 Windows ，但不能匹配 3.1Windows 中的 Windows 。

(?

反向否定预查，与正向否定预查类似，只是方向相反。例如 (?

配 2000Windows 中的 Windows 。

控制标记

有几个全局控制标记（Flag）需要了解，其中最常默认指定的有 G 和 M ：

A / ASCII ， 默认为 False

\d , \D , \w , \W , \s , \S , \b , 和 \B 等只限于ASCII 字符

行内写法： (?a)

Python re 模块中的常量： re.A re.ASCII

I / IGNORECASE ， 默认为 False

忽略字母大小写

行内写法： (?i)

Python re 模块中的常量： re.I re.IGNORECASE

G / GLOBAL ， 默认为 True

找到第一个 match 之后不返回

行内写法： (?g)

Python re 模块中这个标记不能更改，默认为 TRUE

L / LOCALE ， 默认为 False

由本地语言设置决定 \d , \D , \w , \W , \s , \S , \b , 和 \B 等等的内容

行内写法： (?L)

Python re 模块中的常量： re.L re.LOCALE

M / MULTILINE ， 默认为 True

使用本标志后， ^ 和 $ 匹配行首和行尾时，会增加换行符之前和之后的位

置。

行内写法： (?m)

Python re 模块中的常量： re.M re.MULTILINE

S / DOTALL ， 默认为 False

使 . 完全匹配任何字符，包括换行；没有这个标志， . 匹配除了 n r 之外

的任何字符。

行内写法： (?s)

Python re 模块中的常量： re.S re.DOTALL

X / VERBOSE ， 默认为 False

当该标志被指定时，Pattern 中的的空白符会被忽略，除非该空白符在圆括号或

方括号中，或在反斜杠 \ 之后。这样做的结果是允许将注释写入 Pattern，这

些注释会被 Regex 解析引擎忽略。注释用 # 号 来标识，不过该符号不能在字

符串或反斜杠之后。

行内写法： (?x)

Python re 模块中的常量： re.X re.VERBOSE

几个最常用的 **Regex**

以下是几个常用的 Regex ， 值得保存：

[2]

matching username

/^[a-z0-9_-]{3,16}$/

matching password[3]

/^[a-z0-9_-]{6,18}$/

matching a HEX value

/^#?([a-f0-9]{6}|[a-f0-9]{3})$/ $/)

matching a slug

/^[a-z0-9-]+$/

matching email address

/^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]

{2,6})$/ @([%5Cda-z%5C.-]+)%5C.([a-z%5C.]%7B2,6%7D)$/)

matching a URL

/^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-

]*)*\/?$/ ?([%5Cda-z%5C.-]+)%5C.([a-z%5C.]%7B2,6%7D)

([%5C/%5Cw%20%5C.-])%5C/?$/)

matching an IP address

/^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}

(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/

matching a HTML tag

[/^<([a-z]+)([^<]+)*(?:>(.*)<\/\1>|\s+\/>)$/]

(https://regexper.com/#%2F%5E<%28%5Ba-

z%5D%2B%29%28%5B%5E<%5D%2B%29 *%28%3F%3A>%28.*
%29<%5C%2F%5C1>

对了，告诉你一个小秘密：

以上的正则表达式，点击都能连接到 regexper.com，在那里你能查看这些正则表达

式的图形化示意图。另外，本文中，处于 markdown cell 的绝大多数正则表达式都

有这样的连接…… 你可以重读的时候试试。

写 Regex 最烧脑的地方在于“使其全面” —— 要考虑到各种意外情况。

当然，除非必要，也不要在“全面”这事儿上面花太多时间 —— 给你看一个据说是“最大程度上能够匹

配所有 email 地址的 Regex”，我都懒得测试的一个正则表达式：

[4]

regex

(?:(?:\r\n)?[\t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(

?:\r\n)?[\t]

)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[

\t]))*"(?:(?:

\r\n)?[\t])*)(?:\.(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-

\031]+(?:(?:(

?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(

?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[\t])*))*@(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\"
.\[\] \000-\0
31]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\
]\r\\]|\\.)*\
](?:(?:\r\n)?[\t])*)(?:\.(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\
] \000-\031]+
(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\
\]|\\.)*\](?:
(?:\r\n)?[\t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\
n)?[\t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[\t]))
*"(?:(?:\r\n)
?[\t])*)*\<(?:(?:\r\n)?[\t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\03
1]+(?:(?:(?:\
r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[
\t])*)(?:\.(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(
?:(?:(?:\r\n)
?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(

?:\r\n)?[\t]

)*))*(?:,@(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:

(?:(?:\r\n)?[

\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:

\r\n)?[\t])*

)(?:\.(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(

?:\r\n)?[\t]

)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n

)?[\t])*))*)

:(?:(?:\r\n)?[\t]))?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:(?:

\r\n)?[\t])+

|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[\t

]))*"(?:(?:\r

\n)?[\t])*)(?:\.(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-\0

31]+(?:(?:(?:(?:

\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:

(?:\r\n)?[\t

]))*"(?:(?:\r\n)?[\t])*))*@(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\

[\] \000-\031

]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\

r\\]|\\.)*\](

?:(?:\r\n)?[\t])*)(?:\.(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\]

\000-\031]+(?

:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]

|\\.)*\](?:(?

:\r\n)?[\t])*))*\>(?:(?:\r\n)?[\t])*)|(?:[^()<>@,;:\\".\[\] \00

0-\031]+(?:(?

:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.

|(?:(?:\r\n)?

[\t]))*"(?:(?:\r\n)?[\t])*)*:(?:(?:\r\n)?[\t])*(?:(?:(?:[^()<>

@,;:\\".\[\]

\000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"

(?:[^\"\r\\]|

\\.|(?:(?:\r\n)?[\t]))*"(?:(?:\r\n)?[\t])*)(?:\.(?:(?:\r\n)?[\

t])*(?:[^()<>

@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:

\\".\[\]]))|"

(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[\t]))*"(?:(?:\r\n)?[\t])*))*@(?:(

?:\r\n)?[\t]

)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[

\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\t])*)(?:\.(?:(?:\r
\n)?[\t])*(?
:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["(
)<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\t])*))*|(?:[^()<>@,;:\
\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^
\"\r\\]|\\.|(
?:(?:\r\n)?[\t]))*"(?:(?:\r\n)?[\t])*)*\<(?:(?:\r\n)?[\t])*(?:
@(?:[^()<>@,;
:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\"
.\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\t])*)(?:\.(?:(?:\r\n)?[\t])*(?:
[^()<>@,;:\\"
.\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\
]]))|\[([^\[\
]\r\\]|\\.)*\](?:(?:\r\n)?[\t])*))*(?:,@(?:(?:\r\n)?[\t])*(?:[^
()<>@,;:\\".\
[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]

```
))|\[([^\[\]\
r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>
@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\
[([^\[\]\r\\]
|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,
;:\\".\[\] \0
00-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?
:[^\"\r\\]|\\
.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,
;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?
:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:
\r\n)?[ \t])*
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
"()<>@,;:\\".
\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n
)?[ \t])*(?:[
^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<
```

>@,;:\\".\[\]

]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\t])*))*\>(?:(?:\r\n)?[\

t])*)(?:,\s*(

?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[

\["()<>@,;:\\

".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[\t]))*"(?:(?:\r\n)?[\t

])*)(?:\.(?:(

?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[

\t])+|\Z|(?=[

\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[\t]))*"(?:

(?:\r\n)?[\t

])*))*@(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:

(?:\r\n)?[\t

])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\

n)?[\t])*)(?

:\.(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\

r\n)?[\t])+|

\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[

\t])*))*|(?:

[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\["()

<>@,;:\\".\[\

]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[\t]))*"(?:(?:\r\n)?[\t])*)*

\<(?:(?:\r\n)

?[\t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t]

)+|\Z|(?=[\["

()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\t])*)(?:

\.(?:(?:\r\n)

?[\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\

Z|(?=[\["()<>

@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\t])*))*(?:,@

(?:(?:\r\n)?[

\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|

(?=[\["()<>@,

;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\t])*)(?:\.(?:(

?:\r\n)?[\t]

)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[

\["()<>@,;:\\

".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\t])*))*)*:(?:(?:\r

\n)?[\t])*)?

(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t])+|\Z|(?=[\[

"()<>@,;:\\".

\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[\t]))*"(?:(?:\r\n)?[\t])

*)(?:\.(?:(?:

\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[\t

])+|\Z|(?=[\[

"()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[\t]))*"(?:(?

:\r\n)?[\t])

))@(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?

:\r\n)?[\t])

+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)

?[\t])*)(?:\

.(?:(?:\r\n)?[\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\

n)?[\t])+|\Z

|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[\

t])*))*\>(?:(

?:\r\n)?[\t])*))*)?;\s*)

趣事一则

ELIZA 是人类史上第一个“聊天机器人”程序，上个世纪六十年代，1966 年实现的。核心就是使用正

则表达式匹配对方语句中的关键词汇，再经过替换，形成看似有意义的“回应” —— 事实上，ELIZA

完全不知道自己在说什么……

几年后，就是七十年代初，1972 年，另外一个聊天机器人 Parry 被写了出来

[5]

…… 随后第二年，

这两个聊天机器人在 ARPANET “碰面” 好好地聊了一下：

[6]

ELIZA: Good Evening. Tell me your problems. Please terminate input with a period or a question mark.

PARRY: People get on my nerves sometimes.

ELIZA: I am not sure I understand you fully.

PARRY: You should pay more attention.

ELIZA: Suppose you should pay more attention.

PARRY: You're entitled to your own opinion.

ELIZA: What makes you think I am entitled to my own opinion?

PARRY: It bothers me just to be around people in general.

ELIZA: Please go on.

PARRY: I went to the track at Bay Meadows a while back.

ELIZA: What does that suggest to you?

PARRY: I went to the races.

.....

以下是后人用 Python 重新实现的 ELIZA 的程序代码，你可以直接执行它，尝试一下于 ELIZA 聊聊

天：

In [ ]:

 # %load https://raw.githubusercontent.com/jezhiggins/eliza.py/master/eliza.py

#---

 # eliza.py

#

 # a cheezy little Eliza knock-off by Joe Strout

 # with some updates by Jeff Epler

 # hacked into a module and updated by Jez Higgins

#---

import string

import re

import random

class eliza:

def __init__(self):

self.keys = list(map(**lambda** x:re.compile(x[0], re.IGNORECASE),gPats)

)

self.values = list(map(**lambda** x:x[1],gPats))

```
#---------------------------------------------------------------------
# translate: take a string, replace any words found in dict.keys()
# with the corresponding dict.values()
#---------------------------------------------------------------------
def translate(self,str,dict):
words = str.lower().split()
keys = dict.keys();
for i in range(0,len(words)):
if words[i] in keys:
words[i] = dict[words[i]]
return ' '.join(words)
#---------------------------------------------------------------------
# respond: take a string, a set of regexps, and a corresponding
# set of response lists; find a match, and return a randomly
# chosen response from the corresponding list.
#---------------------------------------------------------------------
def respond(self,str):
# find a match among keys
for i in range(0, len(self.keys)):
match = self.keys[i].match(str)
```

```
if match:
# found a match ... stuff with corresponding value
# chosen randomly from among the available options
resp = random.choice(self.values[i])
# we've got a response... stuff in reflected text where indicated
pos = resp.find('%')
while pos > -1:
num = int(resp[pos+1:pos+2])
resp = resp[:pos] + \
self.translate(match.group(num),gReflections) + \
resp[pos+2:]
pos = resp.find('%')
# fix munged punctuation at the end
if resp[-2:] == '?.': resp = resp[:-2] + '.'
if resp[-2:] == '??': resp = resp[:-2] + '?'
return resp
#---------------------------------------------------------------------
# gReflections, a translation table used to convert things you say
# into things the computer says back, e.g. "I am" --> "you are"
#---------------------------------------------------------------------
```

```
gReflections = {
"am" : "are",
"was" : "were",
"i" : "you",
"i'd" : "you would",
"i've" : "you have",
"i'll" : "you will",
"my" : "your",
"are" : "am",
"you've": "I have",
"you'll": "I will",
"your" : "my",
"yours" : "mine",
"you" : "me",
"me" : "you"
}
#-----------------------------------------------------------------------
# gPats, the main response table. Each element of the list is a
# two-element list; the first is a regexp, and the second is a
# list of possible responses, with group-macros labelled as
```

```
# %1, %2, etc.
#----------------------------------------------------------------------
gPats = [
[r'I need (.*)',
[ "Why do you need %1?",
"Would it really help you to get %1?",
"Are you sure you need %1?"]],
[r'Why don\' ?t you ([^\?]*)\??',
[ "Do you really think I don't %1?",
"Perhaps eventually I will %1.",
"Do you really want me to %1?"]],
[r'Why can\' ?t I ([^\?]*)\??',
[ "Do you think you should be able to %1?",
"If you could %1, what would you do?",
"I don't know -- why can't you %1?",
"Have you really tried?"]],
[r'I can\' ?t (.*)',
[ "How do you know you can't %1?",
"Perhaps you could %1 if you tried.",
"What would it take for you to %1?"]],
```

[r'I am (.*)',

["Did you come to me because you are %1?",

"How long have you been %1?",

"How do you feel about being %1?"]],

[r'I\' ?m (.*)',

["How does being %1 make you feel?",

"Do you enjoy being %1?",

"Why do you tell me you're %1?",

"Why do you think you're %1?"]],

[r'Are you ([^\?]*)\??',

["Why does it matter whether I am %1?",

"Would you prefer it if I were not %1?",

"Perhaps you believe I am %1.",

"I may be %1 -- what do you think?"]],

[r'What (.*)',

["Why do you ask?",

"How would an answer to that help you?",

"What do you think?"]],

[r'How (.*)',

["How do you suppose?",

"Perhaps you can answer your own question.",

"What is it you're really asking?"]],

[r'Because (.*)',

["Is that the real reason?",

"What other reasons come to mind?",

"Does that reason apply to anything else?",

"If %1, what else must be true?"]],

[r'(.*) sorry (.*)',

["There are many times when no apology is needed.",

"What feelings do you have when you apologize?"]],

[r'Hello(.*)',

["Hello... I'm glad you could drop by today.",

"Hi there... how are you today?",

"Hello, how are you feeling today?"]],

[r'I think (.*)',

["Do you doubt %1?",

"Do you really think so?",

"But you're not sure %1?"]],

[r'(.*) friend (.*)',

["Tell me more about your friends.",

"When you think of a friend, what comes to mind?",

"Why don't you tell me about a childhood friend?"]],

[r'Yes',

["You seem quite sure.",

"OK, but can you elaborate a bit?"]],

[r'(.*) computer(.*)',

["Are you really talking about me?",

"Does it seem strange to talk to a computer?",

"How do computers make you feel?",

"Do you feel threatened by computers?"]],

[r'Is it (.*)',

["Do you think it is %1?",

"Perhaps it's %1 -- what do you think?",

"If it were %1, what would you do?",

"It could well be that %1."]],

[r'It is (.*)',

["You seem very certain.",

"If I told you that it probably isn't %1, what would you feel?"]],

[r'Can you ([^\?]*)\??',

["What makes you think I can't %1?",

"If I could %1, then what?",

"Why do you ask if I can %1?"]],

[r'Can I ([^\?]*)\??',

["Perhaps you don't want to %1.",

"Do you want to be able to %1?",

"If you could %1, would you?"]],

[r'You are (.*)',

["Why do you think I am %1?",

"Does it please you to think that I'm %1?",

"Perhaps you would like me to be %1.",

"Perhaps you're really talking about yourself?"]],

[r'You\' ?re (.*)',

["Why do you say I am %1?",

"Why do you think I am %1?",

"Are we talking about you, or me?"]],

[r'I don\' ?t (.*)',

["Don't you really %1?",

"Why don't you %1?",

"Do you want to %1?"]],

[r'I feel (.*)',

["Good, tell me more about these feelings.",

"Do you often feel %1?",

"When do you usually feel %1?",

"When you feel %1, what do you do?"]],

[r'I have (.*)',

["Why do you tell me that you've %1?",

"Have you really %1?",

"Now that you have %1, what will you do next?"]],

[r'I would (.*)',

["Could you explain why you would %1?",

"Why would you %1?",

"Who else knows that you would %1?"]],

[r'Is there (.*)',

["Do you think there is %1?",

"It's likely that there is %1.",

"Would you like there to be %1?"]],

[r'My (.*)',

["I see, your %1.",

"Why do you say that your %1?",

"When your %1, how do you feel?"]],

```
[r'You (.*)',
[ "We should be discussing you, not me.",
"Why do you say that about me?",
"Why do you care whether I %1?"]],
[r'Why (.*)',
[ "Why don't you tell me the reason why %1?",
"Why do you think %1?" ]],
[r'I want (.*)',
[ "What would it mean to you if you got %1?",
"Why do you want %1?",
"What would you do if you got %1?",
"If you got %1, then what would you do?"]],
[r'(.*) mother(.*)',
[ "Tell me more about your mother.",
"What was your relationship with your mother like?",
"How do you feel about your mother?",
"How does this relate to your feelings today?",
"Good family relations are important."]],
[r'(.*) father(.*)',
[ "Tell me more about your father.",
```

"How did your father make you feel?",

"How do you feel about your father?",

"Does your relationship with your father relate to your feelings tod

ay?",

"Do you have trouble showing affection with your family?"]],

[r'(.*) child(.*)',

["Did you have close friends as a child?",

"What is your favorite childhood memory?",

"Do you remember any dreams or nightmares from childhood?",

"Did the other children sometimes tease you?",

"How do you think your childhood experiences relate to your feelings

today?"]],

[r'(.*)\?',

["Why do you ask that?",

"Please consider whether you can answer your own question.",

"Perhaps the answer lies within yourself?",

"Why don't you tell me?"]],

[r'quit',

["Thank you for talking with me.",

"Good-bye.",

"Thank you, that will be $150. Have a good day!"]],

[r'(.*)',

["Please tell me more.",

"Let's change focus a bit... Tell me about your family.",

"Can you elaborate on that?",

"Why do you say that %1?",

"I see.",

"Very interesting.",

"%1.",

"I see. And what does that tell you?",

"How does that make you feel?",

"How do you feel when you say that?"]]

]

#----------

 # command_interface

#----------

def command_interface():

print('Therapist**n**---------')

print('Talk to the program by typing in plain English, using normal up

per-')

```
print('and lower-case letters and punctuation. Enter "quit" when done
.')
print('='*72)
print('Hello. How are you feeling today?')
s = ''
therapist = eliza();
while s != 'quit':
try:
s = input('> ')
except EOFError:
s = 'quit'
print(s)
while s[-1] in '!.':
s = s[:-1]
print(therapist.respond(s))
if __name__ == "__main__":
command_interface()
Therapist
---------
Talk to the program by typing in plain English, using normal upper-
```

and lower-case letters and punctuation. Enter "quit" when done.

==========

Hello. How are you feeling today?

脚注

[1]：释义摘自苹果电脑上系统内建的《牛津英汉双解辞典》

↑Back to Content↑

[2]：8 Regular Expressions You Should Know by Vasili

↑Back to Content↑

[3]：关于校验密码强度的正则表达式，往往需要设置更为复杂的规则，Stackoverflow 上的一则答

复中有很好的示例：https://stackoverflow.com/a/21456918

↑Back to Content↑

[4]：http://www.ex-parrot.com/pdw/Mail-RFC822-Address.html

↑Back to Content↑

[5]：Parry 的源代码（用 Lisp 写的）在这里：http://www.cs.cmu.edu/afs/cs/project/ai-

repository/ai/areas/classics/parry/

↑Back to Content↑

[6]：ELIZA 和 Parry 的完整聊天记录在这里：https://tools.ietf.org/html/rfc439

↑Back to Content↑

BNF 以及 **EBNF**

通常情况下，你很少会在入门书籍里读到关于 Backus-Naur Form（BNF，巴科斯-诺尔范式）和

Extended Backus-Naur Form（EBNF）的话题 —— 它们都被普遍认为是“非专业人士无需了解的话

题”，隐含的另外一层含义是“反正就算给他们讲他们也无论如何看不懂”……

然而，在我眼里，这事儿非讲不可 —— 这是这本“书”的设计目标决定的。

严格意义上来讲，在《自学是门手艺》中，以自学编程为例，我完全没必要自己动手耗时费力写那

么多东西，仅仅是为了让读者“入门”。编程入门书籍，或者 Python 编程入门书籍，都已经太多太多

了，其中质量过硬的书籍也多得去了 —— 并且，如果你没有英文阅读障碍，那你就会发现网上有太

多非常优质的免费教程…… 真的轮不到李笑来同学再写一次。

我写这本书的目标是：

让读者从认知自学能力开始，通过自学编程作为第一个实践，逐步完整掌握自学能

力，进而在随后漫长的人生中，需要什么就去学什么，

…… 不用非得找人教、找人带 —— 只有这样，前途这两个字才会变得实在。

于是，我最希望能做到的是，从这里了解了自学方法论，也了解了编程以及 Python 编程的基础概

念之后，《自学是门手艺》的读者能够自顾自地踏上征程，一路走下去

—— 至于走到哪里，能走到

哪里，不是我一个作者一厢情愿能够决定的，是吧？

当然，会自学的人运气一定不会差。

于是，这本“书”的核心目标之一，换个说法就是：

我希望读者在读完《自学是门手艺》之后，有能力独立地去全面研读官方文档 ——

甚至是各种编程语言、各种软件的相关的文档（包括它们的官方文档）。

自学编程，很像独自一人冲入了一个丛林，里面什么动物都有……　而且那个丛林很大很大，虽然丛

林里有的地方很美，可若是没有地图和指南针，你就会迷失方向。

其实吧，地图也不是没有 —— 别说 Python 了，无论什么编程语言（包括无论什么软件）都有很翔

实的官方文档……　可是吧，绝大多数人无论买多少书、上多少课，就是不去用官方“地图”，就不！

—— 其实倒不是说“第三方地图”更好，实际的原因很不好意思说出来：

这首先吧，觉得官方文档阅读量太大了……（嗯？那地图不是越详细越好

吗？）

那还有吧……　也不是没去看过，看不懂……（嗯……　这对初学者倒是个问

题！）

所以，我要认为这本“书”的最重要工作是：

为读者解读清楚地图上的“图例”，从此之后读者在任何需要的时候能够彻底读懂地

图。

在阅读官方文档的时候，很多人在 The Python Tutorial 上就已经觉得吃力了…… 如果到了

Standard Libraries 和 Language References 的部分，就基本上完全放弃了，比如，以下这段摘自

string — Common string operations：

Format Specification Mini-Language ... The general form of a standard format specifier is:

format_spec ::= [[fill]align][sign][#][0][width][

grouping_option][.precision][type]

fill ::=

align ::= " <" | ">" | "=" | "^"

sign ::= "+" | "-" | " "

width ::= digit+

grouping_option ::= "_" | ","

precision ::= digit+

type ::= "b" | "c" | "d" | "e" | "E" | "f"

| "F" | "g" | "G" |

"n" | "o" | "s" | "x" | "X" | "%"

...

读到这，看着一大堆的 ::= [] | 当场傻眼了……

这是 BNF 描述，还是 Python 自己定制的 EBNF…… 为了理解它们，以后当然最好有空研究一

下“上下文无关文法” （Context-free Grammar），没准未来你一高兴就会去玩一般人不敢玩的各种

Parser，甚至干脆自己写门编程语言啥的…… 不过，完全可以跳过那些复杂的东西的 —— 因为你

当前的目标只不过是“能够读懂那些符号的含义”。

其实吧，真的不难的 —— 它就是语法描述的方法。

比如，什么是符合语法的整数（Integer）呢？符合以下语法描述的是整数（使用 Python 的

EBNF）：

integer ::= [sign] [digit]*

sign ::= "+" | "-"

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"

| "9"

以上的描述中，基本符号没几个，它们各自的含义是：

::= 表示定义；

< > 尖括号里的内容表示必选内容；

[ ] 中是可选项；

" " 双引号里的内容表示字符；

| 竖线两边的是可选内容，相当于or；

+ 表示零个或者一个……

* 表示一个或者多个……

于是：

1. interger 定义是：由“可选的 sign ”和“一个或者多个 digit 的集合” 构成

—— 第一行末尾那个 * 的作用和正则表达式里的 * 一样；

2. sign 的定义是什么呢？要么是 + 要么是 -；

3. digit 的定义是什么呢？从 "0" 到 "9" 中的任何一个值……

于是， 99 、 +99 、 -99 ， 都是符合以上语法描述的 integer ； 但， 99+ 和 99- 肯定不符

合以上语法描述的 integer 。

很简单吧？反正就是在 ::= 左边逐行列出一个语法构成的所有要素，而后在右边逐行逐一定义，

直至全部要素定义完毕。

也许那些在此之前已经熟悉 BNF 范式的人会有点惊讶，“你怎么连‘终结符’和‘非终结符’这种最基本

的概念都跳过了？” —— 是呀，即便不讲那俩概念也能把这事儿讲清楚到“能马上开始用”了的地

步…… 这就是我经常说的，“人类有这个神奇的本领，擅长使用自己并不懂的东西……”

Python 对 BNF 的拓展，借鉴了正则表达式 ——

[1]

从最后两个符号的使用（ + * ）你可以看得出

来。顺带说，这也是为什么这本“书”里非要讲其他入门书籍里不讲的正则表达式的原因之一。

又，由于 Python 的社区文档是二十来年长期积累的，有时标注方法并不一致。比如，在描述

Python Full Grammar specification 的时候，他们用的语法标注符号体系就跟上面描述 String 的语

法不一样了，是这样的：

: 表示定义；

[ ] 中是可选项；

' ' 双引号里的内容表示字符；

| 竖线两边的是可选内容，相当于or；

+ 表示零个或者一个……

* 表示一个或者多个……

—— 用冒号：替代了 ::= ，用单引号 ' 替代了双引号 "" ，而尖括号 <> 干脆不用了：

 # Grammar for Python

 # NOTE WELL: You should also follow all the steps listed at

 # https://devguide.python.org/grammar/

 # Start symbols for the grammar:

 # single_input is a single interactive statement;

 # file_input is a module or sequence of commands read from

an input file;

 # eval_input is the input for the eval() functions.

 # NB: compound_stmt in single_input is followed by extra NEWLINE!

single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE

file_input: (NEWLINE | stmt)* ENDMARKER

eval_input: testlist NEWLINE* ENDMARKER

decorator: '@' dotted_name ['(' [arglist] ')'] NEWLINE

decorators: decorator+

decorated: decorators (classdef | funcdef | async_funcdef)

async_funcdef: 'async' funcdef

funcdef: 'def' NAME parameters ['->' test] ':' suite

parameters: '(' [typedargslist] ')'

typedargslist: (tfpdef ['=' test] (',' tfpdef ['=' test])* [',' [

'*' [tfpdef] (',' tfpdef ['=' test])* [',' ['**' tfpdef [

',']]]

| '**' tfpdef [',']]]

| '*' [tfpdef] (',' tfpdef ['=' test])* [',' ['**' tfpdef [',']

]]

| '**' tfpdef [','])

tfpdef: NAME [':' test]

varargslist: (vfpdef ['=' test] (',' vfpdef ['=' test])* [',' [

'*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [

',']]]

| '**' vfpdef [',']]]

| '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']

]]

| '**' vfpdef [',']

)

vfpdef: NAME

stmt: simple_stmt | compound_stmt

simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE

small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |

import_stmt | global_stmt | nonlocal_stmt | assert_s

tmt)

expr_stmt: testlist_star_expr (annassign | augassign (yield_expr|

testlist) |

('=' (yield_expr|testlist_star_expr))*)

annassign: ':' test ['=' test]

testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [','

]

augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|='

| '^=' |

' <<=' | '>>=' | '**=' | '//=')

 # For normal and annotated assignments, additional restrictions e

nforced by the interpreter

del_stmt: 'del' exprlist

pass_stmt: 'pass'

flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt

| yield_stmt

break_stmt: 'break'

continue_stmt: 'continue'

return_stmt: 'return' [testlist]

yield_stmt: yield_expr

raise_stmt: 'raise' [test ['from' test]]

import_stmt: import_name | import_from

import_name: 'import' dotted_as_names

 # note below: the ('.' | '...') is necessary because '...' is tok

enized as ELLIPSIS

import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+

)

'import' ('*' | '(' import_as_names ')' | import_as

_names))

import_as_name: NAME ['as' NAME]

dotted_as_name: dotted_name ['as' NAME]

import_as_names: import_as_name (',' import_as_name)* [',']

dotted_as_names: dotted_as_name (',' dotted_as_name)*

dotted_name: NAME ('.' NAME)*

global_stmt: 'global' NAME (',' NAME)*

nonlocal_stmt: 'nonlocal' NAME (',' NAME)*

assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_

stmt | funcdef | classdef | decorated | async_stmt

async_stmt: 'async' (funcdef | with_stmt | for_stmt)

if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':'

suite]

while_stmt: 'while' test ':' suite ['else' ':' suite]

for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suit

e]

try_stmt: ('try' ':' suite

((except_clause ':' suite)+

['else' ':' suite]

['finally' ':' suite] |

'finally' ':' suite))

with_stmt: 'with' with_item (',' with_item)* ':' suite

with_item: test ['as' expr]

 # NB compile.c makes sure that the default except clause is last

except_clause: 'except' [test ['as' NAME]]

suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

test: or_test ['if' or_test 'else' test] | lambdef

test_nocond: or_test | lambdef_nocond

lambdef: 'lambda' [varargslist] ':' test

lambdef_nocond: 'lambda' [varargslist] ':' test_nocond

or_test: and_test ('or' and_test)*

and_test: not_test ('and' not_test)*

not_test: 'not' not_test | comparison

comparison: expr (comp_op expr)*

 # <> isn't actually a valid comparison operator in Python. It's h

ere for the

 # sake of a __future__ import described in PEP 401 (which really

works :-)

comp_op: ' <'|'>'|'=='|'>='|' <='|' <>'|'!='|'in'|'not' 'in'|'is'|'i

s' 'not'

star_expr: '*' expr

expr: xor_expr ('|' xor_expr)*

xor_expr: and_expr ('^' and_expr)*

and_expr: shift_expr ('&' shift_expr)*

shift_expr: arith_expr ((' <<'|'>>') arith_expr)*

arith_expr: term (('+'|'-') term)*

term: factor (('*'|'@'|'/'|'%'|'//') factor)*

factor: ('+'|'-'|'~') factor | power

power: atom_expr ['**' factor]

atom_expr: ['await'] atom trailer*

atom: ('(' [yield_expr|testlist_comp] ')' |

'[' [testlist_comp] ']' |

'{' [dictorsetmaker] '}' |

NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False

')

testlist_comp: (test|star_expr) (comp_for | (',' (test|star_expr

))* [','])

trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME

subscriptlist: subscript (',' subscript)* [',']

subscript: test | [test] ':' [test] [sliceop]

sliceop: ':' [test]

exprlist: (expr|star_expr) (',' (expr|star_expr))* [',']

testlist: test (',' test)* [',']

dictorsetmaker: (((test ':' test | '**' expr)

(comp_for | (',' (test ':' test | '**' expr))*

[',')) |

((test | star_expr)

(comp_for | (',' (test | star_expr))* [',']))

)

classdef: 'class' NAME ['(' [arglist] ')'] ':' suite

arglist: argument (',' argument)* [',']

 # The reason that keywords are test nodes instead of NAME is that

using NAME

 # results in an ambiguity. ast.c makes sure it's a NAME.

 # "test '=' test" is really "keyword '=' test", but we have no su

ch token.

 # These need to be in a single rule to avoid grammar that is ambi

guous

*# to our LL(1) parser. Even though 'test' includes '*expr' in sta*

r_expr,

 # we explicitly match '' here, too, to give it proper precedence*

.

 # Illegal combinations and orderings are blocked in ast.c:

 # multiple (test comp_for) arguments are blocked; keyword unpacki

ngs

 # that precede iterable unpackings are blocked; etc.

argument: (test [comp_for] |

test '=' test |

'**' test |

'*' test)

comp_iter: comp_for | comp_if

sync_comp_for: 'for' exprlist 'in' or_test [comp_iter]

comp_for: ['async'] sync_comp_for

comp_if: 'if' test_nocond [comp_iter]

 # not used in grammar, but may appear in "node" passed from Parse

r to Compiler

encoding_decl: NAME

yield_expr: 'yield' [yield_arg]

yield_arg: 'from' test | testlist

现在你已经能读懂 BNF 了，那么，可以再读读用 BNF 描述的 Regex 语法，就当复习了

[2]

—— 很

短的：

BNF grammar for Perl-style regular expressions

<**RE**> ::= <**union**> | <**simple-RE**>

<**union**> ::= <**RE**> "|" <**simple-RE**>

<**simple-RE**> ::= <**concatenation**> | <**basic-RE**>

<**concatenation**> ::= <**simple-RE**> <**basic-RE**>

<**basic-RE**> ::= <**star**> | <**plus**> | <**elementary-RE**>

<**star**> ::= <**elementary-RE**> "*"

<**plus**> ::= <**elementary-RE**> "+"

<**elementary-RE**> ::= <**group**> | <**any**> | <**eos**> | <**char**> | <**set**>

<**group**> ::= "(" <**RE**> ")"

<**any**> ::= "."

<**eos**> ::= "$"

<**char**> ::= any non metacharacter | "\" metacharacter

<**set**> ::= <**positive-set**> | <**negative-set**>

<**positive-set**> ::= "[" <**set-items**> "]"

<**negative-set**> ::= "[^" <**set-items**> "]"

<**set-items**> ::= <**set-item**> | <**set-item**> <**set-items**>

<**set-items**> ::= <**range**> | <**char**>

<**range**> ::= <**char**> "-" <**char**>

真的没原来以为得那么神秘，是不？[3]

都学到这儿了…… 顺带再自学个东西吧。

这个东西叫 glob ，是 Global 的缩写。你可以把它理解为“超级简化版正则表达式” —— 它最初是

Unix/Posix 操作系统中用来匹配文件名的“通配符”。

先看一张 1971 的 Unix 操作系统中关于 glob 的截图：

A screenshot of the original 1971 Unix reference page for glob – note the owner is dmr, short for Dennis Ritchie.

glob 的主要符号只有这么几个：

*

?

[abc]

[^abc]

现在的你，打开 Wikipedia 上的关于 glob 和 Wilcard character 的页面，肯定能做到“无障碍”理解：

https://en.wikipedia.org/wiki/Glob_(programming)

https://en.wikipedia.org/wiki/Wildcard_character

顺带说，现在你再去读关于 Format String 的官方文档，就不会再觉得“根本看不懂”了，恰恰相反，

你会觉得“我怎么之前连这个都看不懂呢？”

https://docs.python.org/3/library/string.html#format-string-syntax

在自学这件事儿上，失败者的死法看起来千变万化，但其实都是一样的…… 只不过是因为怕麻烦或

者基础知识不够而不去读最重要的文档。

比如，学英语的时候死活不读语法书。其实英文语法书也没多难啊？再厚，不也是用来“查”的吗？

不就是多记几个标记就可以读懂的吗？比如，词性标记， v. , n. , adj. , adv. , prep. ... 不

就是相当于地图上的图例吗？那语法书，和现在这里提到的官方文档，不都是“自学者地图”吗？

但，就是这么一点点简单的东西，挡住了几乎所有人，真是可怕。

脚注

[1]: The Python Language Reference » 1.2 Notation —— 这个链接必须去看一看……

↑Back to Content↑

[2]: Perl Style Regular Expressions in Prolog CMPT 384 Lecture Notes Robert D. Cameron November 29 - December 1, 1999

↑Back to Content↑

[3]：很少有人注意到：在很多编程语言的文法文档中， "$" 被称为 —— 2017 年 5 月我

投资了一个初创公司，听说他们的资产名称叫做 eos …… 我当场就被这

个梗逗乐了。

↑Back to Content↑

拆解

在学习编程的过程中，你会不由自主地学会一个重要技能：

拆解

这么简单的两个字，在人生中的作用重大到不可想像……　而且它也的确是自学能力中最重要的底层

能力之一。

横向拆解

我很幸运，12　岁的时候有机会学习编程（习得了最基本的概念，那时候学的编程语言是

BASIC），所以，相对其他人在“拆解任务”方面有更强的初始意识。

后来，15 岁开始学着玩吉他的时候，发现道理其实是一样的。

有个曲子很难（当然也非常好听），曲名是 Recrerdes Da La Alhambra 阿罕布拉宫的回忆。你看

看曲谱就知道它多难了：

那怎么办？怎么办？！—— 我的办法听起来看起来都很笨：

每次只弹一个小节；

而且还是放慢速度弹，刚开始很慢很慢；

等熟悉了之后逐渐快起来，直到正常速度；

再开始弹下一个小节；

同样是放慢速度弹，刚开始很慢很慢；

等熟悉了之后逐渐快起来，直到正常速度；

再把两个小节拼起来；

有些小节拼起来相对容易，另外一些需要挣扎很久才顺畅；

如此这般，最终就把这个很难的曲子弹出来了 —— 其实所有的初学者都是这么干的。

可以听听这个曲子放松一下（当然肯定不是我弹的哈哈）：

In [2]:

```
from IPython.display import IFrame

IFrame('https://www.youtube.com/embed/OOsRMECWKAE?', width='800', height

='450')
```

Out[2]:

提起这事儿，总是会不由自主地叹口气 —— 因为在这事儿上我运气太差，刚把这个曲子练完没多

久，还没来得及找人显摆，就摔断了掌骨和指骨，给我的手指灵活性造成了不可修复的损伤，于

是，后来只能用拨片玩玩吉他了……

话说回来，自学的一个重要技巧就是，

把那些很难的任务无限拆分 —— 直至每个子任务都很小，小到都可操作为止。

比如，正则表达式，这个你必须学会的东西，学会学好真的不那么容易。一切的技能都一样，没学

会之前都是很难很难的，可学会之后用熟了，你就会“发现”拿东西其实也没多难……

那刚开始的时候怎么办？你其实需要运用拆分的本领：

先囫囵吞枣至少读一遍教程；

给自己搭好测试的环境（或许在 Regex101.com 上，或许用一个编辑器，比如

VS Code）；

先不管什么意思，找一些 Regex 自己试试；

正式进入“精度”状态，每一小节每一小节地突破；

搞定一小节之后，就把它与之前的小节再反复翻两三遍；

把学习任务拆分成若干块，再重新逐个突破，比如，匹配，替换，在编辑器中

使用，在 Python 代码中使用；

把各种操作符与特殊字符拆分成若干个组，而后，熟悉到牢记（而不用将来反

复回来查询）； ……

事实上，当你习惯这么做了之后，就会“发现”一切的自学任务，其实都不是“难”，不过是“繁杂程度

不一”而已。

很多人最终自学失败，要么是不懂拆分任务，要么就仅仅是怕麻烦而已——还是那句话，人活着就

挺麻烦的……

纵向拆解

拆解的第一种方法是把某个任务拆分成若干个小任务，正如上面的讲解那样，我称它为“横向拆

解”。

另外一种方法，我称它为“纵向拆解”（有时，我也会用“分层拆解”这个说法）。

这种方式在自学复杂的概念体系时特别管用。

编程这种东西，之所以刚开始觉得难学，是因为它其中涉及的概念除了之前我们强调的“过早引

用”之外，还有个特征：

有的概念虽然并不同属一个层面，却常常纠缠在一起，没有明确的提示……

比如，常量、变量、字符串、函数、文件等等的概念，其实并不是某个特定的编程语言的专有概

念，它们是所有编程语言都会涉及到的概念，因为计算机处理的就是这些东西，而无论哪个编程语

言最终都要通过计算机去处理这些东西。

又比如说，分支与循环，每个编程语言都有对应的语句，所以，分支循环在逻辑判断、流程管理这

个层面，而分支循环的“实现”应该划分到另外一个层面中去；而每个语言实现分支循环语句句法多

少有点差异 —— 这些细节属于那个编程语言本身：

 # Python 这么写：

for i **in** range(n):

...

// Javascript 这么写：

var i;

for (i = 0; i < n; i++) {

...;

}

自学正则表达式的时候也如此。最基本的规则是属于 Regex 自己的；而后各种语言的实现各不相

同，那是各个编程语言层面的；在各种编辑器中，除了基础的规则之外，也有它们自己的定制……

看起来细节很多，但分层分类之后，就会变得很容易理解、很容易记住。

遇到“面向对象编程”也是如此。类、实例、对象、继承、多态…… 这些其实并不属于某一个编程语

言，但它们也确实在几乎所有编程语言中被实现、被应用 —— 所谓的难，无非是因为属于两个层面

甚至多个层面的概念被拧在一起教、学、练……

再比如说，在我把这个用编程当作习得自学能力的第一个“实战项目”之时，甚至要把“读”和“写”分成

两个层面，先照顾“读”，至于“写”，要等到有了基本的“读”的能力之后再说；即便是到了“写”，还要

划分至少两个层面，首先是从“简单的函数开始”，而非上来就要写个“大程序”…… 这种拆分层面的

技能好像可以用在方方面面呢！

所以，要在自学的过程中，不停地想办法把它们从层面上区分开来——不能总是把它们混在一块

儿“大锅烩”。

日常生活中，我们会遇到被评价为“理解能力强”的人，而另外那些不被如此评价的人就很不理解，

很迷惑：

我到底差哪儿了，你不说我理解能力强？难道我的理解能力很差吗？

当老师当久了，经常被这种现象震惊：

原来很简单的东西竟然可能成为很多人一生的障碍。

——并且，这话重复多少遍都不过分。

大多数人不太在意自己脑中的概念之间的关系，因为平日里这也不怎么耽误事儿。但，一旦遇到复

杂一点的知识体系，就完全搞不定了……　而所谓知识体系的复杂，无非就是新的概念多一些，概念

之间的关联更复杂一些……　而概念之间的关联更复杂一些，无非是各个概念之间不仅只有一种联

系，最后会形成网状连接……

——　在《通往财富自由之路》那本书里，我几乎用了整本书的篇幅去讲解、厘清概念及其之间关系

的重要性。

复杂吗？其实并不复杂——在横向纵向分别逐步分清之后。

可问题在于，脑子里概念关联一团糟的人，自己并不觉得，甚至无法知

道 —— 他们是那种跟你一块

去看一场电影却能看到另外一部电影的人。说他们理解能力差过分吗？他们不能理解被评价为理解

能力差，难道不是很自然吗？

分清的方法？其实也不难，就是不断拆解，不断主动整理。每次用图表整理那些概念的时候，就会

发现比原来更清晰一些，多次整理，最终就谙熟于心了。

想要再举更恰当更惊人的例子很难，勉为其难再举个例子。

当我在 2011 年遇到比特币的时候，现在回头看，在当时的情况下，我平日里习惯对概念及其关联

进行各种纵向横向的拆解这件事儿给我“创造”了巨大的好运。后来我在《INBlockchain 的开源区块

链投资原则》里提过这事儿：

“比特币”这个概念，可以有多重的理解 —— 这也是为什么人们感到迷惑，或者相互

之间很难达成一致理解的根本原因。

首先，比特币是世界上第一个，也是迄今为止最成功的区块链应用。

其次，比特币是一家世界银行，只不过它不属于任何权威管辖，它是由一个去中心

化网络构成的。

另外，这家叫做比特币的，去中心化的世界银行，发行了一个货币，恰好也叫“比特

币”。有些人更喜欢使用相对小心的说法，把这个货币指称为 BTC，而不是“比特

币”（Bitcoin）。

最后，即便在比特币横空出世的七年后（2017），也很少有人意识到比特币（或者

BTC） 其实也可以被理解为这家叫做比特币的去中心化的世界银行的股票。

—— 这无非就是把一个概念拆分成若干个层面再对每个层面准确理解而已。

但，毫无疑问，这点靠很简单很简单的方法练就的理解能力，帮了我大忙。

触类旁通

无论听起来多么简单的任务，落实成代码肯定没那么简单，没那么容易。

以后你会越来越清楚的：写程序的主要工作量，往往并非来自于在编辑器里敲代码那个阶段。

更多的工作量，其实在于如何才能在脑子里把整个流程拆解清楚，考虑到各个方

面……

所以，编程，更多是拿着纸笔梳理细节的工作。一旦所有的细节都想明白了，落实成代码其实是飞

快的 —— 越是工程量大的项目越是如此。

这个道理在哪里都是相同、相通的。不说编程，说写书，也是一样的。

随着时间的推移，你花在“拆解”上的时间会越来越多，因为所有大的工程，都可以被拆解成小工程

—— 于是，也为了做出大工程，拆解的工作首先是必须，其次是最耗时费力但最值得的。

我身边很多人，包括出版社的专业编辑，都慨叹过我的“写书速度”。我猜，实际上把他们惊到甚至

惊倒的，并不是他们以为的“李笑来写书的速度”，而是“李笑来的打字速度”而已。

当我告诉他们我要写一本什么书了的时候，实际上，有个工作早就完成了：“系统梳理要写的那本书

的所有细节”，剩下的只是落笔把那些东西写出来而已 —— 当然，我是敲出来，用我那几乎无以伦

比的输入速度敲出来 —— 那当然“显得”很快了！

创业也好，投资也罢，还是一样的。因为我这个人脸皮厚，不怕人们笑话，所以我可以平静地说这

事儿：

我参与过（或投资）很多失败的创业项目……

对所有复盘的结果，无一例外，根源都是当初立项的时候，很多重要细节还没搞清楚，甚至没想到

要去搞清楚，就已经开始行动……　于是，在成本不断积累的情况下，没完没了地处理各种“意外”，

没完没了地重新制定目标，没完没了地拖延，没完没了地“重新启动”…… 直至开始苟延残喘，最后

不了了之。

拆解得不够，就容易导致想不清楚，想错，想歪……

也许，有人可能会理直气壮地反问，“怎么可能从一开始就把所有情况都想清楚么！” 唉，是呀，以

前我也是这么想的……　直到吃了很多亏，很多很多亏，很多很多很大很大的亏，才“发现”且不得不

痛下决心去接收：事先想不清楚的，就不要去做。

这是一种特殊、且重要、又极有价值的能力。现实生活中，后来我也见过若干有这种能力的高人，

比如，你可以到网上搜一个人名，庄辰超，他就是我见过的能做到干什么事儿之前都能全都想清楚

的真人活人之一。

自学的时候，拆解任务的重要性更是如此。

这本“书”的一个特点，就是把“自学”（或者平日里称为“学习”）这个流程，拆解为“学”、“练”、“用”、

后面还会讲到“造”总计四个环节来处理 —— 从内容编排本身就这么干，甚至，在开头相当一部分，

就明确说明，“根本不指望你读过一遍就会了”，还反复提醒，“要重复读很多遍，虽然第一遍必须囫

囵吞枣”……

对于初学者常面临的尴尬，我们也从一开始就提醒，编程语言，和你之前在学校里学的语文，本质

上没什么区别，先学会读，而后在多读多读再多读的同时，开始练习写 —— 这才真的很自然。

即便是开始讲如何写，我们的做法也是“从写函数”开始，而不是“来，让我们写个程序……” —— 这

一点点看起来不起眼的差异，作用是很大的，因为从“小而完整”的东西开始做（任何事）非常重

要。

“小”无所谓，“完整”才是关键。

刚需幻觉

在前言之中，就举过一个例子，人们一不小心就把自己搭了进去……只不过因为没搞明白，道理就

是道理，跟讲道理的老生其实全然没什么关系。

在自学中，耽误人的幻觉很多。比如，时间幻觉。人们总觉得自己时间不够了，所以学东西的时候

总是很急……　　可实际上，练成一门手艺，到够用的地步，一两年足够；到很好的地步，三五年足够

——　至于极好么，那是一辈子的事儿。结果呢，很多人瞎着急，乱“省时间”，学啥都不全面，练啥

都不足数足量，一晃三五年就过去，然后又开始焦虑，换个手艺再学学再试试…… 然后循环焦虑。

最坑人的幻觉，在我看来，就是刚需幻觉　——　这又是我杜撰的一个词…… 听我慢慢讲。

感觉总是最大的坑

我的结论是：

绝大多数人的自学能力，基本上都是被“自己的感觉”耽误掉的。

即，原本每个人都有一定的自学能力，但，最终，都被自己的感觉给干掉了，直至全然失去自学能

力　——　虽然其后也经常学习，但，其后一生从事的全都是学习这个行为里最初级的模式：“模仿”。

为什么“模仿”是学习行为最初级的模式呢？

首先，模仿必须依赖模仿对象，受自己的视野所限。

其次，模仿只能处理表里如一的简单知识和技能；一旦遇到那些深层次

的、表面上看不大出来的，

模仿就无效了，即便是非要模仿、愣模仿，结果就只能是“东施效颦”。

在《把时间当作朋友》中，我就反复强调一件事：

不要问学它有什么用，学就是了……

这原本是自学的最佳策略之一，也是自学的最根本策略。然而，听进去这句话的人不多，即便是我

举了无数的例子，即便是他可能当时也有一点认同……　然而，转瞬间又变成了原来的状态，无论遇

到什么自学机会都不由自主地问：

我学它有什么用啊？

如果，得到的答案中，那“用处”对自己不是“刚需”，瞬间就失去了动力，瞬间就放弃了追求…… 直

至某一天，突然“发现”那竟然是个刚需…… 于是，临时抱佛脚。人么，总是对追求的事情心影相

随；又有谁会对曾经放弃的事情念念不忘呢？于是，下一次还是会再做“预算不足”的决定。

最终失去自学能力的人，都是因为对“刚需”的判断失误 —— 这就是我说的“刚需幻觉”。

刚需幻觉

“刚需幻觉”的根源在于：

你不会的东西，对你来说感觉上就不是刚需。

要不是这本“书”里有个自学编程那么大且立体的一个例子存在，供我反复地、多角度地阐述道理，

上面这句话还真的不好解释 —— 即便解释清楚了，也因为缺乏生动的例证而失去效力。

正则表达式（Regex）就是很好的例子。

当你没有学会它的时候，甚至不知道它的时候，感觉上，你不可能觉得那是个刚需 —— 因为你一直

都活得好好的，甚至在你全然不知道那个东西竟然存在的情况下。

你略微学了一下，没学会、没学好、或者干脆放弃了之后，在感觉上，你也同样不会觉得那是刚

需。因为你不会用它，所以你当然“用不上它”…… 你根本没办法直到你不懂正则表达式这个事实让

你“未得到”什么（不是“失去”，因为你不曾拥有）。

然而，只要你花了一点力气，真的掌握了它，你会有两个“发现”：

你根本离不开它[1]；

它其实真没多难……

第二个“发现”总是让你无比庆幸 —— 幸亏学了，幸亏“发现”了，否则，自己耽误自己一辈子且完全

不自知。庆幸！庆幸！！但，第一个“发现”更为重要，因为但凡哪怕只有过一次这样的经历，你就

可以永久摆脱“刚需幻觉”的诅咒。

编程也是一样的，当你开始学编程的时候，被身边的人发现，他们中的大多数一定会问你：

你学它要干嘛啊？

无论你怎么回答，他们的反应应该都是差不多的：反正就是不理解你的投入是否合理。

然后，等你学会了编程，掌握了这个技能，你的“发现”肯定包括那两点：

你根本离不开它；

它其实真没多难……

哪怕很简单的技能都是如此。许多年前，我专门花一两周的时间练习键盘盲打，以及把输入法方案

选择改为微软双拼方案……　　也有人揶揄过我：“我靠，你练这玩意干嘛？难道将来要当一辈子打字

员？”

当然，我很早就练就了一定的自学能力，所以很早就有一些自学经验，所以我用不着事后“发现”，

而是“直接就知道”，将来：

我根本离不开它；

它其实真没多难……

事实上呢？事实上就是如此。打字速度极快，直接导致了后来我成为多产作家。无法想象如果我是

个打字速度极慢的人，我如何写书，如何写那么多书和那么多文章；以及，后来我又是如何于 2018

年 9 月的某一天开始在 72 小时之内完成《韭菜的自我修养》初稿……可问题在于，这个技能难

吗？在我长大的年代里，它算个技能；在今天，全都是从小就用键盘的人的世界里，谁不会呢？

所以，当面对一项新技能的时候，“觉得并非刚需”肯定是幻觉。因为一个技能到底是不是刚需，在

掌握它之前是不知道的……

只有真正掌握了那个技能之后，那个技能才会“变”成刚需。

并且，

一旦掌握了某项技能，它只能是刚需。

这种幻觉非常坑人。

我们的大脑，有一种神奇的功能 —— “无论如何都会用已有信息拼成一个完整图像的功能”。

“无论如何”的意思是说，不管那已有信息有多么凌乱、有多么残缺、有多么无意义、人脑竟然还是

可以拼出一个完整的“有意义”的图像。

这一点也经常被人利用。在新闻学里，就有著名的“Framing Effect”：

这个原理也经常被各路营销使用：

很容易想象，经常被这些东西误导的人，“脑力”欠缺到什么程度 —— 这当然也是他们全无自学能力

的结果之一。

而当我们被刚需幻觉所左右的时候，我们明显是使用“片面的信息”拼出了“完整的意义”，而后被其误

导，严重误导，且全然不自知 —— 最气人的是，误导我们的竟然不是别人，竟然是我们自己的大

脑！刚需幻觉如此，时间幻觉、困难幻觉亦如是。后面会讲到的注意力漂移，也依然是这种情况，

自己才是真正的元凶……

所以，在决定学任何东西的时候，最好不要去咨询身边的人 —— 除非你确定对方是高手，最好是自

学高手。否则，你遇到的永远是怀疑、打击、甚至嘲笑。最令人遗憾、最令人无奈的是，那些人其

实也不是坏人，他们其实也不是故意，他们只是被自己的“刚需幻觉”误导了而已，他们（竟然）以

为所有人都和他们一样……　然后，若是你（居然）听信了他们的话，那就实在是太惨了！

要学，想学，那就自顾自去学吧，用不着征求别人的意见！

优势策略

如何不被“刚需幻觉”所迷惑、所限制呢？

首先，基础策略是，要深刻理解这个现象及其解释：

对任何一项技能来说，刚需是自学的结果；而不是自学的原因；

用“是否为刚需”作为自己是否开始学习的理由，注定会吃亏的，因为如此做的下场就是肯定被“刚需

幻觉”所迷惑、所限制。

而后，也是更为重要的一个策略：

做个自驱动的人，而非被外部驱动的被动的人。

这基本上是个越早建立越好的习惯。绝大多数人一生都是被动者，终生只是被外界驱动而已。所

以，他们会在某一时刻，“发现”某个技能是刚需，然后再去学，但，可惜却肯定的是，到那时候预

算总是并不充裕，总是捉襟见肘。

最后一个很简单却很有效的策略是三个字，之前提到过的：

找活干

有活干，真的很幸福。

影响下一代

想象一下有些父母是这样跟孩子对话的：

爸爸（妈妈），你在干什么呀？

我在学 _____（请自行填空）

那你学它干什么用啊？

等我学会就知道了……

而后又过了段时间……

孩子，过来看看！这是爸爸（妈妈）做的东西！

我猜，这样的孩子，会从小就自然而然地破解掉“刚需幻觉”。

所谓的“潜移默化”，所谓的“耳闻目染”，其实是很简单的东西。

然而，效果呢？效果不会因为策略的简单或者容易而消减。通常的情况是，越是简单容易的策略，

效用越是惊人。

所以，正确结论是这样的：

一切的技能都是刚需。

虽然，这并不意味着你不需要选择。

然而，至于学还是不学，其实从来都不是从“有没有用”来判断的，真正

有意义的判断依据只有一

条：

有没有时间？

有时间就学呗！没时间就挤时间呗！学得不足够好怎么办？花更多时间呗……

脚注

[1]：写这本书的时候，前后弄出来那么多 .ipynb 文件 —— 于是，到最后哪怕“生成个目录”这样

看起来简单的活，若是会用正则表达式，就能几分钟完成；但若是不会，那就得逐一手工提取、排

序、编辑…… 对我来说怎么可能不是刚需！

In [2]:

```
import re

import os

files = [f for f in os.listdir('.') if os.path.isfile(f)]

files.sort()

for f in files:

if '.ipynb' in f:

with open(f, 'r') as file:

str = file.read()

pttn = r'"# (.*)"\n'
```

r = re.findall(pttn, str)

if len(r) > 0:

print(f'> - [{f.replace(".ipynb", "")}（**{r[0]}**）]({f})') # 生成 *markdown*

> - [01.preface（**前言**）](01.preface.ipynb)

> - [02.proof-of-work（**如何证明你真的读过这本书？**）](02.proof-of-work.ipynb)

> - [Part.1.A.better.teachyourself（**为什么一定要掌握自学能力？**）](Part.1.A.better.teachyourself.ip ynb)

> - [Part.1.B.why.start.from.learning.coding（**为什么把编程当作自学的入口？**）](Part.1.B.why.start

.from.learning.coding.ipynb)

> - [Part.1.C.must.learn.sth.only.by.reading（**只靠阅读习得新技能**）](Part.1.C.must.learn.sth.only.by

.reading.ipynb)

> - [Part.1.D.preparation.for.reading（**开始阅读前的一些准备**）](Part.1.D.preparation.for.reading.ipy nb)

> - [Part.1.E.1.entrance（**入口**）](Part.1.E.1.entrance.ipynb)

> - [Part.1.E.2.values-and-their-operators（**值及其相应的运算**）](Part.1.E.2.values-and-their-operator s.ipynb)

> - [Part.1.E.3.controlflow（**流程控制**）](Part.1.E.3.controlflow.ipynb)

> - [Part.1.E.4.functions（**函数**）](Part.1.E.4.functions.ipynb)

> - [Part.1.E.5.strings（**字符串**）](Part.1.E.5.strings.ipynb)

> - [Part.1.E.6.containers（**数据容器**）](Part.1.E.6.containers.ipynb)

> - [Part.1.E.7.files（**文件**）](Part.1.E.7.files.ipynb)

> - [Part.1.F.deal-with-forward-references（**如何从容应对含有过多“过早引用”的知识？**）](Part.1.F

.deal-with-forward-references.ipynb)

> - [Part.1.G.The-Python-Tutorial-local（**官方教程：The Python Tutorial**）](Part.1.G.The-Python-Tut orial-local.ipynb)

> - [Part.2.A.clumsy-and-patience（**笨拙与耐心**）](Part.2.A.clumsy-and-patience.ipynb)

> - [Part.2.B.deliberate-practicing（**刻意练习**）](Part.2.B.deliberate-practicing.ipynb)

> - [Part.2.C.why-start-from-writing-functions（**为什么从函数开始？**）](Part.2.C.why-start-from-wri ting-functions.ipynb)

> - [Part.2.D.1-args（**关于参数（上）**）](Part.2.D.1-args.ipynb)

> - [Part.2.D.2-aargs（**关于参数（下）**）](Part.2.D.2-aargs.ipynb)

> - [Part.2.D.3-lambda（**化名与匿名**）](Part.2.D.3-lambda.ipynb)

> - [Part.2.D.4-recursion（**递归函数**）](Part.2.D.4-recursion.ipynb)

> - [Part.2.D.5-docstrings（**函数的文档**）](Part.2.D.5-docstrings.ipynb)

> - [Part.2.D.6-modules（**保存到文件的函数**）](Part.2.D.6-modules.ipynb)

> - [Part.2.D.7-tdd（**测试驱动的开发**）](Part.2.D.7-tdd.ipynb)

> - [Part.2.D.8-main（**可执行的 Python 文件**）](Part.2.D.8-main.ipynb)

> - [Part.2.E.deliberate-thinking（**刻意思考**）](Part.2.E.deliberate-

thinking.ipynb)

> - [Part.3.A.conquering-difficulties（**战胜难点**）](Part.3.A.conquering-difficulties.ipynb)

> - [Part.3.B.1.classes-1（**类 —— 面向对象编程**）](Part.3.B.1.classes-1.ipynb)

> - [Part.3.B.2.classes-2（**类 —— Python 的实现**）](Part.3.B.2.classes-2.ipynb)

> - [Part.3.B.3.decorator-iterator-generator（**函数工具**）](Part.3.B.3.decorator-iterator-generator.ipynb

)

> - [Part.3.B.4.regex（**正则表达式**）](Part.3.B.4.regex.ipynb)

> - [Part.3.B.5.bnf-ebnf-pebnf（**BNF 以及 EBNF**）](Part.3.B.5.bnf-ebnf-pebnf.ipynb)

> - [Part.3.C.breaking-good-and-bad（**拆解**）](Part.3.C.breaking-good-and-bad.ipynb)

> - [Part.3.D.indispensable-illusion（**刚需幻觉**）](Part.3.D.indispensable-illusion.ipynb)

> - [Part.3.E.to-be-thorough（**全面 —— 自学的境界**）](Part.3.E.to-be-thorough.ipynb)

> - [Part.3.F.social-selfteaching（**自学者的社交**）](Part.3.F.social-selfteaching.ipynb)

> - [Part.3.G.the-golden-age-and-google（**这是自学者的黄金时代**）](Part.3.G.the-golden-age-and-go ogle.ipynb)

> - [Part.3.H.prevent-focus-drifting（**避免注意力漂移**）](Part.3.H.prevent-focus-drifting.ipynb)

> - [Q.good-communiation（**如何成为优秀沟通者**）](Q.good-communiation.ipynb)

> - [R.finale（**自学者的终点**）](R.finale.ipynb)

> - [S.whats-next（**下一步干什么？**）](S.whats-next.ipynb)

> - [T-appendix.editor.vscode（**Visual Studio Code 的安装与配置**）](T-appendix.editor.vscode.ipynb

)

> - [T-appendix.git-introduction（**Git 简介**）](T-appendix.git-introduction.ipynb)

> - [T-appendix.jupyter-installation-and-setup（**Jupyterlab 的安装与配置**）](T-appendix.jupyter-instal lation-and-setup.ipynb)

> - [T-appendix.symbols（**这些符号都代表什么？**）](T-appendix.symbols.ipynb)

↑Back to Content↑

全面 —— 自学的境界

之前提到过那些“貌似一出手就已然是高手”的人，也为此做过一番解释：

他们的特点就是善于刻意练习……

为了真正做到刻意练习，更重要的是需要不断地进行刻意思考 —— 刻意思考自己究竟应该在哪些地

方必须刻意练习？

之前也说过，人和人非常不同，于是，需要刻意练习的地方也各不相同。

不过，倒是有一个方面，所有的自学者都必须刻意练习 —— 这是谁都

逃不过的地方：

全面

那些“貌似一出手就已然是高手”的人就是在这一方面超越了绝大多数人——在每个层面上，他们都

学习得更全面，练习得更全面，使用得最全面，在使用此技能去创造的时候，思考得也就自然更为

全面。于是，就产生了“全面碾压”的效果。

然而，这是很难被人发现的“秘密”，因为“全面”这个事实，只存在于这些高人的大脑之中，很难被展

示出来……而他们不会想到这是个什么“秘密”——因为他们一直就是这么做的，他们会误以为所有

人都是这么做的。

小时候，我经常看到父亲备课到深夜。他手中的教科书，每一页的页边都密密麻麻地写着各种注

释，实在没地方写了，就在那个地方插上一张纸，于是能写得更多……到最后，他的那本书，要比

别人同样的书看起来厚很多。

许多年后，我竟然成了老师，于是，我就备课。我备课的方法自然是“拷贝”过来的，我父亲怎么做

的，我见过，于是我也那么做。到最后，都到了这个地步：只有那书已经成了别人的两倍厚度心里

才踏实。

又过了一段时间，在一个内部分享会中，我听到一位老师的说法，他问我：

……　李老师啊，您已经讲了这么久了，都熟到干脆不用备课的地步了吧？

我愣了一下，讲课前不用备课这事儿根本就不在我的想象范围之内啊！我父亲讲了那么多年的课，

不还是经常备课到深夜嘛？我也一样做了，总是觉得“还是有很多可以补充的地方”啊！

然而，这个小插曲提醒了我一个现象：

我会那么做，我就会误以为所有人都会那么做……

我猜，那些“貌似一出手就已然是高手”的人，也一样，他们从未觉得这是什么“秘密”，他就是那么做

的，他们很久以来就是那么做的，他们误以为所有人都是那么做的。

从另外一方面，外人更希望他们拥有的是个“秘密”。于是，因为自己并不知道那个“秘密”，所以，自

己做不到他们那样　——　这样看起来就合理了，自己的心里也够舒服了，毕竟看起来理所应当了么。

把自学当作一门手艺，把所有的技能也都当作一门手艺，那就相对容易理解了：

全面，是掌握一门手艺的基本。

为了全面，当然要靠时间。所以，关于“混与不混”，我们有了更深刻却又更朴素的认识：

所谓的不混时间，无非就是刻意练习、追求全面。

也正是这个原因，几乎所有自学高手都懂这个道理：

绝对不能只靠一本书

有个特别有趣的现象，我觉得绝大多数人平日里都挺大手大脚的，都挺

舍不得对自己过分苛刻的，

但，一到买书这件事儿上，绝大多数人真的很节俭，真的很苛刻 ——对待越严肃的知识越是如此。

倒是在买本小说啊或者买张电影票的时候，基本不用过脑子。

他们好像完全不知道自己正在疯狂地虐待自己…… 的大脑。

对自己的胃好一点，我绝对认同 —— 因为我自己就是个吃货。

可是，在很长一段时间里，我完全不能理解人们为什么不由自主地对自己的大脑不好，不仅是不

好，还是格外地不好，甚至干脆是虐待。

观察学生多了，也就慢慢理解了。

绝大多数人事实上从来没有习得过自学能力，他们终生都在被指导、被引领。而在校教育少则九

年，多则十几年二十年，他们体验过太多“不过尔尔”的学习过程。他们肯定不是没花过钱，九年义

务教育的过程中，就花了父母很多钱，后面若是上了大学，花钱更多…… 花过那么多钱，却总是没

什么收获，在他们的经验中，“这次我应该小心点”是再自然不过的事。

“第一次突破”很重要。

如果一个人有过一次只通过阅读书籍即获得一项新技能的体验，那么，他们内心深处（更准的说法

是大脑的底层操作系统）的那个成本计算方法就会发生改变，心里想的更可能是：

这肯定是有用的，一旦学会，收益可不是几十块钱的书价或几百块课价那么一

点点……

至于是否能学会，主要看我投入的时间精力预算有多少……

我身边有很多自学能力非常强的人。

这些人买书的方式都是一样的，一旦他们决定学习什么技能的时候，第一个想到的是去买书，而不

是去找人。他们之前体验过，他们就是很清楚：

首先，书里什么都有；

其次，仅靠阅读在大多数情况下绝对够了……

更为不一样的是，他们一定会买回来一大堆书 —— 而不是四处去问，“关于 xx 的哪一本书最好

啊？”

在他们眼里，书是最低成本的东西，比起最终的收益来讲更是不值一提。

更为重要的是，一本书绝对不够 —— 无论是谁写的，无论那个作者多么著名，影响力有多大……

因为，书也好、教程也罢，这种东西就是有“篇幅限制”的。更为关键的是，每个作者都有不一样的

视角、不一样的出发点和不一样的讲解方式、组织方法。

比如，我这本就跟别人写的很不一样。我的出发点是把编程当作一个自学的例子，重点在于学会如

何自学，并且通过实践真的习得一个起初你觉得不是刚需，学会之后发现干脆离不开的、不可或缺

的技能。这本书的内容组织方式也不一样 —— 反正你现在已经知道了。

另外，这本书的目标里有更重要的另外一个：“让你有能力靠自己能够理解所有的官方文档” —— 书

里不用讲官方标准库里的每个模块、每个函数究竟如何使用，因为那些在官方文档里定义得非常清

楚……

而其他人写的呢？比如 Think Python，比如 A Bite of Python，再比如 Dive into Python ，以及网

[1]

上很多很多免费的 Python 教程都写得很好呢！

没有经验的人不懂而已。当你搞明白了一本书，后面再多读哪怕很多本的时间精力成本都是很低

的，但每多读一本，都能让你在这个话题中变得更为完整。

针对同一个话题读很多本书的最常见体验就是：

嗯？这个点很好玩！这个角度有意思！

看看比比前面几个作者怎么论述的呢？

嗯？！怎么我看过却竟然没注意到呢！

这最后一条真的是很令人恼火却又享受的体验。它令你恼火，是因为你竟然错过；它令你享受，是

因为虽然错过却竟然还有弥补的机会！

总有一天你会明白的，一切的“学会”和“学好”之间的差异，无非是全面程度的差异。

于是，翻译过来，“学好”竟然如此简单：

多读几本书。狠一点，就是多读很多本书。

到最后，这种习惯会慢慢延伸到生活中。比如，我在遇到好歌的时候，总是想尽一切办法找到那首

歌的很多版本，首唱者可能有很多版本，录音版、现场版，不同年份的版本等等；还有很多翻

唱…… 看电影也一样，若是有翻拍版本，一定会找过来看 —— 不同国家的翻拍版本对比起来特别

好玩。

甚至，到最后，你做东西的时候都会想着顺手多做几个版本。我这本“书”，就肯定会有印刷版、电

子版…… 到最后还会有个产品版 —— 这基本上目前尚无其它作者做到的事情。

提高对所学知识技能的“全面程度”，有个最狠的方法 —— 再次说出来不惊人，但实际效果惊到爆：

教是最好的学习方法。

这真的不是我的总结，人类很早就意识到这个现象了罢？

孔老夫子在《礼记·学记》里就“曰”过：

“学然后知不足，教然后知困。知不足，然后能自反也；知困，然后能自强也。故

曰：教学相长也。”

到了孔子三十二代孙，孔颖达，解读《兑命》中所提“学学半”时，说到：

“学学半者，上学为教，下学者谓习也……”

许嘉璐先生[2]有一本书，《未央续集》，提到这段解读的时候讲了个自己的例子：

“我当了五十年的教师，经常遇到这种情况：

备好课了，上讲台了，讲着讲着，突然发现有的地方疏漏了，某个

字的读音没有查，文章前后的逻辑没有理清楚，下完课回去补救，

下次就不会出现同样的情况了，这就是教学相长。”

所以，别说老师了，学生更是如此。

我经常讲我所观察到的班里的第一和第二的区别 —— 因为这是很好的例子。

第一总是很开放，乐于分享，别人问他问题，他会花时间耐心解答；第二总是很保守，不愿分享，

不愿把时间“浪费”在帮助他人身上…… 注意，在“浪费”这个词上我加了引号 —— 这是有原因的。

我的观察室，这不是现象，这是原因：

第一之所以比第二强，更可能是因为他开放、乐于分享，才成了第一。

而不是因为他是第一，所以才开放，所以才乐于分享。

因为到最后，你会发现，第一并没有因为时间被占用而成绩退步，反而成绩更好。这是因为他总是

在帮助其他同学的过程里，看到了自己也要避免的错误、发现了其它的解题思路、巩固了自己的知

识点，所以他在社交的过程中学到了更多，同时还收获了同学们的友谊 —— 换言之，通过分享，通

过反复讲解，他自己的“全面程度”得到了最快的提高。

而第二呢？第二其实有可能比第一更聪明呢 —— 他可是全靠自己走到那个地步的！可是，他没有用

最狠的方式提高自己的全面程度，虽然排名第二，可他其实只不过是一个“下学”者，于是，他很吃

力的 —— 虽然他实际上很聪明…… 于是，在这种感受下，他怎么肯愿意把那么吃力才获得的东西

分享出去呢？

这真是个有趣且意味深长的现象。

另外一个有趣的现象是，“下学”者永远等待“上学”者整理好的东西。之前在《如何从容应对含有过

多“前置引用”的知识？》提到过一个对应策略：

尽快开始整理归纳总结

同时给出了建议：

一定要自己动手去做……

不仅如此，还描述了个我自己的例子，通过“自己动手整理”才发现自己之前竟然完全理解错了。

我这方面运气非常好，因为父母全都是大学教师，从小父母就鼓励我帮同学解答问题，这让我不知

不觉在很早很早就开始了“上学”的阶段。

这一次写这本“书”的过程中，同样的“奇迹”再次发生在我身上。

说实话，正则表达式我一直没有完全掌握 —— 虽然偶尔用用，也都是边查边用。实在解决不了，就

算了…… 现在回头想想，多少就是因为“仅凭感觉，并没觉得那绝对是刚需” [3] —— 当然，真正会了

之后，马上开始时时刻刻都有可能使用，离开它简直活不下去 —— 写这书的后半程，有大量的重新

组织的需要，很多文字替换，若是没有正则表达式，就干脆没法干……

我是如何完全掌握正则表达式的呢？就是因为写这本书。既然是写书，当然害怕自己在不经意中出

错，此为其一。更为重要的是，必须先完整掌握之后才能有诸如“为读者提供更好一点的理解起

点”，“理解起来相对更简单直接的组织结构”，以及“挖掘必须习得它的真正原因以便鼓励读者”之类

的畅销书卖点 —— 对作者来说，有什么比销量更重要的呢？

写一本好书，对我来讲，这个需求太刚了，刚到好像是钛钢的地步。

于是，本来就习惯于同一个话题多读好多本书的我，读了更多的书，翻了更多的教程，官方文档翻

了更多遍，做了更多的笔记，那一章反复废掉原稿再次重写了很多遍…… 在这样的刺激下，“全面

程度”若是没有极速提高，那才怪了呢！

还有啊，我的英语，也是这么搞出来的。

所以，很多的时候，我这个人并不是“好为人师”，细想想，貌似“好为己师”更为恰当一点。写书也好

讲课也罢，其实自己的进步是最大的。

若是我没有在新东方教书七年，我连当前这个半吊子英文水准都不会有…… 为了在讲台上不犯错，

多多少少都得多费一些功夫吧？我的经验和许嘉璐先生是一样的，无论备课多努力，后面还是会有

纰漏…… 可是，这若只是做个“下学”者，岂不更惨？

哦，对了，其实所有的读者，都可以用这个简单的方法影响下一代：

有同学问，你就一定要耐心讲讲 —— 对自己有好处。

当然，最直接的方法是把自己变成“上学”者，保持开放，乐于分享，而孩子只需通过“耳闻目染”就可

以了。

脚注

[1]：一点八卦：Dive into Python 的作者是 Mark Pilgrim；他是互联网上最著名的自绝于信息世界

（infosuicide）的三人之一。另外两位分别是比特币的作者 Satoshi Nakamoto，和 Why the lucky

stiff 的作者 Jonathan Gillette。

↑Back to Content↑

[2]：许嘉璐，1998年至2000年全国人大常委会副委员长，民进中央主席，国家语言文字工作委员会

主任。

↑Back to Content↑

[3]：写完这一段，给霍炬看，他当场嘲笑我，“哈！当年我就说，你应该学学 Vim，是不是到现在你

都没学？”我无言以对，因为真的就没学…… 然后，我想了想，回复他说，“好吧，我决定写一个

Vim 教程出来，嗯。”

↑Back to Content↑

自学者的社交

很多人有莫名其妙的误解，以为“自学”（self-teaching）就一定是“自己

独自学”（solo-learning），

殊不知，自学也需要社交。也有另外一些人，因为“专心”到一定程度，觉得社交很累，所以开始抵

触一切社交。这些都不是全面的看法。

事实上，在任何领域，社交都是必须的，只不过，很多人没有建立、打磨过自己的社交原则，所以

才被各种无效社交所累。就算讨厌，讨厌的也不应该是社交，而是无效社交。

在自学的任何一个阶段，学、练、用、造，社交都可能存在。

哪怕是在最枯燥，看起来最不需要社交的“练”的阶段，社交也会起很大的作用 —— 在自己累了的时

候，看到有人还在练，看到很多人都在练，看到很多人其实也挺累的但还在练…… 这些都是让自己

感觉没那么费劲的好办法。

实际上，在最初“学”的阶段，社交也是极为重要的。

生活中，你遇到过这样的现象没有：“看见别人打针，自己先疼得受不了……” 这是因为我们的大脑

中有一种神经元，叫做镜像神经元（Mirror　Neuron），它会让我们“感同身受”，当我们看到另外一

个人正在做什么的时候，镜像神经元会尽力给我们足够的刺激，让我们“体验”那个人的感受。以前

人们不知道为什么哈欠竟然会“传染”，现在科学家们很清楚了 —— 那就是镜像神经元在起作用。

镜像神经元的存在，使得我们有模仿能力、有通感能力、有同情心、有同理心…… 这也是为什么人

类天然有社交需求的重要原因，因为我们的大脑皮层上都有很多的镜像神经元。

一般来说，物品、书籍之类非人的东西，都不大可能激活镜像神经元。只有看到人的时候，镜像神

经元才会被激发。所以，你送给小朋友一把吉他，他不会有什么兴趣的。可若是你在弹琴的时候被

他看见，他的镜像神经元就会因为你的行为而被触发，进而对弹奏吉他感兴趣 —— 注意，不是对吉

他本身感兴趣。若是你在弹琴的时候，带着某种能够打动他的情绪，那他更容易被影响，因为情绪

更能激发镜像神经元。也就是说，一切的学习起初都基于模仿，一切的模仿，都源自于看到真人的

行为 —— 哪怕是在电影里看到，虽然只不过是影相而已，并非真人，但毕竟是真人的影相。

所以，无论学什么技能，都要找到用那种技能的人，这样我们的镜像神经元才可能更容易被激发，

学习效果才会好。若是能找到热爱那项技能，乃至于一使用那项技能就很开心（最好的情绪之一）

的人，那就更好了。激情这东西，是少数幸运儿才长期持有的东西，大多数人小时候挺多，过了十

五六岁之后就开始有意无意磨灭了激情，且并不自知。

之前提到，

当我们看到另外一个人正在做什么的时候，镜像神经元会尽力给我们足够的刺激，

让我们“体验”那个人的感受。

这句话里有个词很重要，“尽力”。因为镜像神经元只能调用我们大脑里已有的信息去模拟对方的感

受，所以，它最多也就是“尽力”，无法做到“确保正确”。今天的糖尿病患者使用的皮下注射针头，已

经可以做到很细，细到让使用者“无感”的地步，所以，当一个糖尿病患者给自己注射胰岛素的时

候，他自己并不觉得疼，可是看的人却能“疼”到紧皱眉头的地步，为什么？因为旁观者的大脑里没

有实际用那么细的针头注射胰岛素的经验，所以镜像神经元在旁观者“感同身受”时所调用的，其实

是过往旁观者自己打针的体验 —— 被很粗的针头做静脉注射的痛苦体验。

所以，很多人误以为他们眼里的成功者靠的是“坚持”、靠的是“毅力”，这完全是自己的镜像神经

元“尽力”的结果，是“调用自己过往经验去感同身受的结果”…… 事实上呢？那些“成功者”其实并不在

意成功，因为到死之前成长不应该也不可能结束，因为那是他们的生活方式，学习、进步、探索、

迂回，甚至折腾、挫败和迷茫，都是他们生活中必不可少的内容，这是最初不自觉的选择，谈不上

什么“坚持”，谈不上什么“毅力”…… 说实话，对他们来说，不让折腾才真痛苦呢，不学习才需要坚

持和毅力呢！

再进一步，这也是为什么要选择朋友的原因。人与人之间有很大的差异，最大的差异来自于性格养

成，大多数人会沦为表现型人格，只有少数人才会在不断调整中保持、

呵护、进一步培养“进取

型”人格。他们自然而然地更为乐观，更有耐心，更有承受力，更有战斗力，更能生产更能体验学习

与进步的乐趣。与这样的人在一起，学习会更容易 —— 只因为镜像神经元会更容易地被正确激发。

说清楚了，道理其实挺简单的。

有一次朋友跟我聊起他苦于没办法培养自己孩子的正经兴趣爱好……我说，其实很简单，只不过是

你方法错了。你不用告诉孩子“应该学什么，应该对什么感兴趣”，而是，想尽一切办法让孩子见识

到拥有那个技能的，令他产生羡慕情绪的人 —— 只要孩子羡慕那个人，他就自然而然地有“我也想

这样”的想法，到最后，谁都拦不住。这就是镜像神经元的力量。进而，所谓的社交，还真不一定是

非要跟人说话、聊天…… 见识到，也是社交的最主要组成部分。

你看，谁说社交不重要？

进而，想要把一门手艺搞到真正“精湛”的地步，最有效的方法就是尽早进入“造”的阶段 —— 所谓

的“造”，就是不断创造的“造”。

自学这门手艺，很简单，就是不断地学：

def TeachYourSelf(anything):

while not create(somthing):

learn()

practice()

return TeachYourSelf(another)

TeachYourSelf(coding)

学上几个，自然就很精湛。而其它的用自学这门手艺习得的手艺，基本上都可以用“是否做出了像样

的作品”作为检验自己的那门手艺是否达到了精湛的衡量指标。

硅谷有一家著名的孵化器，叫 Y-Combinator，现在的掌门人是个很年轻的人，Samuel H. Altman。

他在那篇著名的文章《Advice for ambitious 19 year olds》中有一个精彩的建议：

No matter what you choose, build stuff and be around smart people.

无论你选择了什么，都要造出东西来，要与聪明人打交道。

当然，对于“聪明人”这个概念，我和 Sam 的看法并不一致。在我看来，有好作品的人都很聪明，

但，还是那句话，那不是天分和智商，那分明是有效积累。

我个人最看重的个人品质之一，就是有没有像样的作品。

很少有人有像样的作品。人群中只有少数人最终能拿出完整的作品——人与人之间的差异是如此之

大，乃至于少数人有作品，更少数人有好的作品，只极少数极少数人才可能做出传世的作品；而与

此同时，绝大多数人（万分之九千九百九十九的人）一辈子都没有像样的作品，他们连一篇作文都

写不明白。于是，与有像样作品的人打交道，总是非常值得。

并且，跟他们打交道也不费劲，都是思考非常通透的人，通常沟通能力极强。哪怕沟通起来貌似费

劲的那一小部分，事实上也不是难以沟通，那只不过是人家简单朴实而已。

我甚至经常建议我的合伙人们，在招人的时候，把这一点当作最靠谱的判断方式。少废话，少吹牛

逼，给我看看你的作品。这个原则可以一下子过滤掉所有的不合格者。另外一个很自然的现象是，

如果一个人能做出像样的东西来，那么他身边的聪明人密度无论如何都会比其他人的高出很多。

地球上有效社交密度最高的地方，是 Github。有些程序员们常开玩笑，说 Github 是全球最大的同

性社交网站，事实上，他们不知道，女性程序员的比例正在逐步提高，而且女性在科学上，从来就

没有屈居二线过[1]。

在 Github 上，找到自己感兴趣的项目，而后为那项目贡献一己之力，用自己的工作赢得社区的认

同…… 这就是 Github 上的社交方式。若是自己做了有意义的项目，就会有更多人关注；若是那项

目对很多人有用，那就不仅有很多人关注，更有很多人会像当初的你一样为这个项目做贡献…… 这

就是程序员们的有效社交。

Github 能成为地球上最大的有效社交网络，没毛病，因为用作品社交肯定是最高效的。

所以，无论学什么，都要想尽一切办法尽快做出自己的作品。做一个产品出来的过程中，会磨练另

外一项自学者不可或缺的能力和素质：

完整

与之前提到的另外一项加起来，就构成了自学者的最基本素养：

学就学得全面；

做就做得完整。

无论多小的作品，都会让创作者感受到“单一技能的必然无效性” ——你试试就知道了。哪怕你想做

个静态网站，你都会发现，仅仅学会 html/css 是不够的，因为部署到远端服务器上的时候，你无论

如何都得学学 Linux 基本操作…… 而已然具备了自学者基本素养的你，自然会想办法“全面掌握”，

而不是糊弄一下而已。

更为重要的是，一旦你开始创作作品，你更大的“发现”就是，你肯定需要很多“之前看起来并不相干

的知识与技能”，而非“只靠专业就够了”……

还是拿我出第一本书为例。那之前我没有写过书，若是出版了书放在书店，没有人知道李笑来是

谁…… 于是，只有内容本身，并不保证那书能卖出去。除了把内容写出来之外，我必然要去学习很

多之前完全没碰过的东西，比如“如何才能做到系统持续地修订内容”；又比如，“如何与出版社编辑

正常沟通”；再比如，“如何取一个好书名”…… 一个赛一个地“与专业无关”。

所以，“做得完整”，从来都不是容易的事情。

从这个角度去理解，你就会明白那些高明的手艺人为什么总是做小东西——那是因为在追求完整的

过程中，你必然会发现，越小越容易完整。这也是为什么庸人总是好高骛远，因为他们不顾完整，

所以就可以妄图建造海市蜃楼。

手艺人不怕做的事儿小。而且，“小”无所谓，“完整”才是关键。

有作品和没作品的人理解能力也不一样。做过作品的人，看到类似 MoSCow Method 的做事原则，

瞬间就能有所感悟，而没有作品的人，却不见得有一样的感受。

顺带给你看个 Wikipedia 上的链接列表，在编程领域里，有无数可以借鉴到生活中

的哲学、方法论：

If it ain't broke, don't fix it

KISS principle

Don't repeat yourself

Feature creep

List of software development philosophies

Minimum viable product

MoSCoW method

Overengineering

Worse is better

S.O.L.I.D.)

Unix philosophy

给自己足够长的时间去学；在充足“预算”之下耐心地练；不断找活干，以用带练；然后，最重要的

是，一定要尽快尝试着做出属于自己的完整作品，无论大小。

只有这样，你才是个值得被交往的人。

脚注

[1]：NPR：Most Beautiful Woman' By Day, Inventor By Night

上图是保存在美国专利局的一个存档文件（US Patent 2,292,387, Aug 11, 1942）截图。这项专利

的发明者是 Hedy Lamarr，人家长得是这样的：

Hedy Lamarr 是好莱坞最知名的演员之一，并且，她也是自学高手：

Although Lamarr had no formal training and was primarily self-taught, she worked in her spare time on various hobbies and inventions, which included an improved traffic stoplight and a tablet that would dissolve in water to create a carbonated drink. The beverage was unsuccessful; Lamarr herself said it tasted like Alka-Seltzer.

↑Back to Content↑

这是自学者的黄金时代

历史上，自学者从未像今天这样幸福。

以前不是这样的。比如，几乎每个中国人从小就听说过无数次那种“为

了拜师学艺一跪就几天甚至几

个月也不一定被收下”的故事。

在古代，拜师学艺的确是很难的事情。首先，真正的好老师确实难寻；其次，高手也没空当老师；

再次，就算是肯收徒授艺的老师也总是时间精力极其有限……　更为重要的是，那时候想自学也真不

行 —— 根本就没有什么称得上是文献的东西可供阅读或检索，很多重要信息甚至干脆只存在于某些

人的脑中，就算它们被落实成了文献，也相当有限，且散落深藏在各处 —— 就算凑齐了，也没有

Google！

对，最关键的是那时候没有 Google……

今天的互联网，已经不再是二十几年前刚出现的那样“激进而简陋”的东西了。经过多年的发展，互

联网上的内容已经构成了真正意义上的“全球唯一通用图书馆” —— 而针对它可用的检索工具中最好

的，当然是 Google。

于是，今天，自学者在真正的意义上身处于一个黄金时代 —— 没有什么是不能自学的。注意措辞，

在这句话前面甚至根本不用加上“几乎”这样的限定以示准确 —— 你想学什么，就能学什么，而不

是“只有先拜师才能开始学艺”。

今天的你，想学什么就去问 Google；在学习中遇到什么问题，就直接问 Google —— 直接问它，通

常总是比向某个人提问有效率得多。Google 就是这样，越用越离不开它……

其实很多人并不真的懂如何用好 Google 的，可是连这个它也能解决，因为你可以直接问它：

How to use google effectively

经过多年的发展，Google 的使用体验越来越好，2019 年的今天，你搜索以上语句返回的页面

里，Google 甚至在众多搜索结果中选了一条它“认为”是“最佳”的搜索结果：

lifehack.com 上的这篇文章也的的确确值得细读 —— 读过且真正理解之后，不夸张地讲，你的“搜

索技能”已经足够灭掉整个人类当前活跃群体中的 99% 了…… 这个说法真的完全没有夸张，绝大多

数人就是不会在搜索的过程中使用那些符号的，比如 - * ~ @ # : " .. —— 还有很多

其它技巧…… 话说，你在 Google 上用过 Time *place* 吗？

已经掌握了正则表达式以及 glob 的你，学习一下如何使用这种符号，实在是太小菜一碟了 —— 然

而，这么简单的东西所能带来的未来收益是非常惊人的，不信走着瞧。

可实际上，还是得应用我们之前说过的原则：

首选查询，肯定是官方文档。

这么做的重要理由之一，是为了让你少受“二手知识”的蒙蔽和误导。这里有一个绝佳的例子让你理

解二手知识的局限性：我写过的一本“书”，发布在网上 —— 这本“书”的另外一个“神”之处，在于它能

让你“顿悟”阅读的力量，不管你是男生还是女生…… 若是这个链接错过了，你竟然会错过整个生

活！

Google Search 的官方文档在这里：

https://support.google.com/websearch

Google 还有更为强大的工具给你使用，叫做 Google Custom Search，官方文档在这里：

https://support.google.com/customsearch/

对编程工作来说，Google 当然格外地更为重要 —— 互联网上积累的最多最专业的信息，当然是计

算机相关信息。所以，当你遇到什么错误提示的时候，不仅要问 Google，还要优先问问

Stackoverflow —— 连 Google 自己都这么干。在 colab.research.google.com （Google 为它的

TensorFlow 服务搭建的 Jupyterlab 环境）上，如果你运行什么代码出错了的话，那么，出错信息

下面会出现一个按钮，上面写着： SEARCH STACK OVERFLOW ，点击它就直接给你

Stackoverflow 上的搜索结果 …… 真够意思！

Google、Stackoverflow、Wikipedia、Youtube，这些都是你经常要去搜索的好地方。

二十年前，Google 刚出现的时候，谁能想象它今天这个样子呢？以下是 1998 年 11 月 11 日

http://google.com 这个网址的截图：

当时，网站还没有正式上线，第一个链接是一个原型设计，用一个二级域名发布在斯坦福的网站

上：http://google.stanford.edu/：

那个时候，Google 还要向外人强调一下，他们已经有 2500 万页面可供检索！

事实上，到了 2008 年，Google 公告称，可供检索页面已经超过一万亿（One Trillion），到了

2016 年年底，这个数字已经超过 130 万亿……

换个角度看，这个数字同时也是互联网上信息的积累 —— 世界上再没有比互联网更大更全

的“书”了。并且，由于 Google 的存在，互联网这本大书，是可检索的！

于是，有事儿先问 **Google** 就成了自学者的必备修养。

能 Google 出答案的问题，就不需要去麻烦别人。

这也是一个自学者的基本素养。

偶尔，也确实会遇到 Google 了很久，就是没有找到答案的情况…… 这样的时候，你可能需要想办

法“问人”了。然而，最靠谱的通常并不见得是“身边的人”，而是互联网上各种垂直社区里的其他用

户……

向人问，也是有学问的 —— 很多人张口就问，结果呢？结果没人理。

为什么呢？

作为一个有素养的自学者，有一篇文章必须精读：

How To Ask Questions The Smart Way

这是大神 Eric S. Raymond 和 Rick Moen 于 2001 年在网上发布的文章，被人们奉为经典；迄今为

止经历了很多次的修订，最后一次是在 2014 年， Rivision 3.10 —— 前后被翻译成了许多种

语言。

不认真使用 **Google**，你就错过了整个人类历史上自学者最黄金的时代。

避免注意力漂移

注意力漂移，是我杜撰的一个词，用来作为“注意力集中”的反义词 —— 因为更多的时候，我们并不

是“注意力不集中”，而是…… 而是更令人恼火的一个现象：

“注意力所集中的焦点总是不断被自己偷偷换掉……”

比如，你本来只不过就想着去 Google 一个编程错误信息的解读，结果不知道怎么就“注意到”了另外

一个东西，比如，“某编辑器的皮肤”，然后你就“顺手”把它下载下来，然后“很自然地顺手把它装

上”，而后又看了看，觉得有点必要于是“顺手做了点定制”…… 然后欣赏了一会儿，并自我得意一番

之后“突然发现”自己还没有解决两小时之前要解决的问题！

说这种现象“令人恼火”，是因为那注意力所集中的焦点，是被自己偷偷换掉的！

好奇心越重的人，越是容易被注意力漂移所拖累。

好奇心是好东西，而且是必须认真呵护的东西 —— 几乎最重要、最强劲的自学动力，都混合着好奇

心出现并持续。

在我成长的那个年代里，很多孩子本来是可以成为自学专家的，结果，99.99% 都被父母给毁了

—— 而且还是不经意地毁的。那些父母毁掉自己孩子的方法简单直接又粗暴、且毫不犹豫。

刚开始图好玩，小孩子问啥都回答…… 当然，最初的时候，小孩子问的问题也很容易回答；就算不

容易，也很容易糊弄过去。没多久，父母就开始应付不过来了，一方面自己也没那么多耐心了，另

外一方面是更为严重的问题 —— 自己的脑力不够。再加上那时候也没有 Google，所以，父母的反

应惊人地一致：“去去去，赶紧睡觉！怎么就你事儿这么多？！”

一个个小朋友就这样被毁掉了，他们的好奇心就这样成了他们自己要主动避免的东西 —— 否则就会

挨骂，谁愿意动不动就被一通数落呢？

好奇心是驱动一个人不断进步的最重要动力之一。所以必须不断呵护，呵护到老才对。

然而，就是这个如此金贵的东西，也会成为拖累；而且，若是真的被它拖累，那么最终真的会感觉

非常遗憾，被好东西拖累 —— 太可惜了。

刚才所描述的，还只不过是两个小时而已的“损失”。事实上，被注意力漂移所拖累的人，损失肯定

远不止如此。在做“工程”或者“项目”的时候 —— 尤其是那种非实物类的工程或项目，比如，写个

书，写个软件之类的 —— 注意力漂移导致的结果就是：

时间不断流逝，可是工程却永远没有结果。

这种损失，完全是任何正常人都没办法承受的……　这话其实并不准，因为事实上据我观察，绝大多

数人受到这种拖累的结果，自己其实无法想象 —— 因为永远没有完成过什么项目，永远没有完成过

什么工程，又如何知道自己损失的究竟是什么呢？

到今天为止，我自己依然还是个需要不断与“注意力漂移”争斗的人 —— 许多年前，我注意到这个现

象的时候，经过思考，就接受了一个事实：

注意力漂移不是能杜绝的现象，但必须在关键时刻有所应对……

如果当年的我没认真想过这事儿，没思索出对策，那么后来的我也不可能写那么多书，转行那么多

次，自学那么多大大小小的技能……　当然，各位读者也完全看不到现在正在阅读的文字 —— 因为

它们不可能被完整地写出来，甚至干脆就不应该存在。

在罗列并比较众多策略之后，我选了一个看起来最不相干，却最本质的策略：

把**“全面完整”**放到最高优先级。

而后，这些年全靠这个策略挺了过来……

当我想做什么事儿的时候，或者想学什么东西的时候，我会投入一定的时间去琢磨，这个事儿或者

这个东西，要做得全面完整，或者要学得全面完整，那都应该做什么呢？在思考如此严肃的问题的

时候，我还是习惯用纸和笔，写写画画 —— 迄今为止没有找到合适的电子设备和软件替代。

我买笔记本，不是为了记笔记的，因为记笔记这个东西，实在是在电脑上做更方便，许多年前开始

就是如此了。我的笔记本主要用来做一件事儿：

罗列整理那些为了做到“全面完整”而必须优先做的事。

用列表也好、或者用图表也罢，反正都是要不断整理修订的，而它的存在，给了我一个优先级：

除了这上面罗列的这些东西之外，在当前时间段，别的都不如它们重要。

一旦发现自己的注意力没有集中在这上面的关键之时，一旦发现自己的注意力已经漂移到其它当前

并不重要的事项上，就马上纠正。

谁都知道应该先做重要且紧急的事情，可问题在于，如何判断“是否重要”呢？全面完整这四个字就

会给我指引。

一方面，是用全面完整来保持自己对重要事情的关注，另外一方面，还需要提高对抗不相关完美诱

惑的能力。十年前，我写《把时间当作朋友》的时候，还把这东西叫做“脆弱的完美主义倾向”，现

在我已经把这个概念升级了 —— 因为更准确地讲，那不是“完美主义者的脆弱”，那是“能力不及格

者”的“轻重不分”。

早些年我，跟很多人一样痴迷于电脑这个东西，也跟很多人那样，用 Windows 惯出来了坏毛病

—— 动不动就重装系统…… 重装系统很浪费时间的，但那时也不知道为什么总是忍不住去干那事

儿，哪怕有些小毛病，马上就受不了，弄的好像重装一个干净的操作系统会让自己的世界焕然一新

一样。

再后来就明白了，这绝对就是自己的毛病 —— 做事不分轻重。

说实话，这也不是自己想明白的 —— 我没那么聪明。是因为遇到了一个高人。他的电脑桌面上，乱

七八糟摆满了各种图标，从不整理。我问他这不影响效率吗？他说，明明有搜索你不用，到底是谁

效率低下？我被问愣了，无言以对。

我又发现他根本没有装杀毒软件…… 我问为什么？他说，“养几个虫子玩玩也没什么不好……” 不

过，他转念又告诉了我他的思考。他说，只要平时习惯好，病毒进来了也没啥可偷的，但，更为关

键的是，他用电脑是干活的，而不是干杀毒的活的…… 脑子如此清楚，让我自愧弗如。

但，学到了。

虽然我还是做不到桌面上图标很乱，虽然我是因为改用了 Mac OS，所以不管杀毒软件，但，注意

力要放到应该放的地方，这事儿我记住了，牢牢记住，从此之后许多年，从未忘过。每次发现自己

轻重不分的时候，就会想起他，然后就改过自新。

如何成为优秀沟通者

一般认为，“手艺人”的普遍特征之一就是缺乏沟通能力，或者沟通能力差 —— 也许是因为平时把所

有的时间精力都投入到磨练手艺上去了罢。

但，这肯定并不是最主要的原因。你看手艺不怎么样的人沟通能力更差；手艺顶级的人却常常反过

来沟通能力很强很强…… 为什么呢？

所以，最核心的理由，应该是一个人最基本的选择而已：

看一个人是否重视沟通能力。

因为若是一个人重视沟通能力，那么，他就自然而然地会想办法去进行刻意练习。如果他本人并不

重视沟通能力，那么，自然就没有任何时间精力投入在这方面，结果就非常自然而然了。

非常遗憾，绝大多数人对沟通能力的重视远远不够 —— 他们也不是不重视，就是重视的程度实际上

太差了。别说双向沟通了，即便是单向沟通，向别人问个问题这么“简单”的事儿，其实也需

要“学”与“练” —— 之前提到过的文章，事实上一定有读者并没有去认真阅读：

…… 作为一个有素养的自学者，有一篇文章必须精读：

How To Ask Questions The Smart Way

经过多次尝试就知道了，连“描述清楚自己的问题”都不是一件很容易做得足够清楚、足够好的事儿

呢。

对沟通能力进行“刻意练习”的最佳方式是什么呢？其实还是你已经知道的：

教是最好的学习方法

讲课、写教程，甚至写书 —— 这是最高效的提升沟通能力的刻意练习手段。

人们常说，“杀鸡焉用宰牛刀”…… 在我看来，既然如此，备上一把宰牛刀挺好的，平时宰牛就用宰

牛刀，备不住哪天需要杀鸡的时候，并不需要非得换把刀才行，依然可以用那把宰牛刀……

讲演能力、写作能力，其实同样也是手艺而已，但，它们也确实是很必要的手艺。尤其是，具备这

两项手艺的人，在现在这样的社会里收入一定不差，不信你就观察一下身边的世界、你的眼界可以

触达的人群。

这两个手艺若是好像是宰牛刀的话，而日常生活里与单人沟通，只不过是杀鸡而已 —— 哈，这个类

比真是不能说给沟通对象听，实在有太大的误会可能性。然而，说正经的，其实这也是为什么“教

师”这个行业的人有不少能够跑出来成为优秀创业者、优秀投资人的原因之一。

当然，绝大多数连一门手艺都没有弄明白的初学者，阅读以下内容时，会觉得“与当前的自己没关

系”…… 不过，请注意，这绝对是“刚需幻觉”，千万不要被它误导。

已经说了无数遍了：绝大多数手艺都是这样的：

原理很简单，精湛与否取决于重复的次数。

哼，这句话不就是卖油翁说的吗？“无他，唯手熟尔” —— 是哦，卖油翁如是说。

以下，让我从入门开始讲起，而后步入进阶，最后到达高级……

入门

内容第一

无论是平日里讲话、还是台上讲课，抑或是写篇文章、写本书，永远都是内容第一，至于形式，并

非不重要，但绝对不能喧宾夺主。

通常，我们用“干货”来描述内容的重要性。第一步就是准备干货，至于修辞啊、笑点啊、酷炫幻灯

片啊等等等等，都必须是次要的，否则就是喧宾夺主。干货不够好，其它的做得越多、越好，最终

越是容易露怯 —— 这很可怕。

所以，在你还没有确定自己值得讲、别人值得听的内容的情况下，就去学习如何制作幻灯片，在我

看来完全是浪费时间。

使用工具的技巧之一，就是用最朴素的方法使用最好的工具，这样成本最低效果最好。所以，我在

讲课、讲演的时候，通常就只用最简单的模版，空白、单色背景那种，而后一页里只写一句话 ——

就是接下来三五分钟里我要讲的重点（甚至干脆只是标题）……　这样的幻灯片，我只需要几分钟就

能做完。而后，我的所有时间精力都放在精心准备内容上去了。高效、低成本。

做事不分轻重，这不对。

内容质量

内容第一，就决定了另外一个事实：

不要讲或写你并不擅长的事。

换句话说，不要分享你做得不够好、做得不够精的手艺。讲课也好、写书也罢，都是分享。分享的

意思是说，你有别人想要的东西 —— 别人想要的，就是他们学不好、做不好的东西..... 若是你自

己手艺不强，手艺不精，其实就没什么可分享的，就算你想“分享”，别人也不要。

你值得讲，别人值得听的，一定是你做得比别人更好的东西，就这么简单。

所以，在自己磨练手艺的时候，可以同时磨练沟通能力；然而，一旦需要讲，需要写，那么就说

明，你自己确信自己做得比别人更好、比别人更精，所以，你值得讲，所以，人家值得听。

一旦你确定自己有值得讲、人家值得听的东西，那么，内容质量的第一要素已经完成了，接下来要

注意哪些呢？只需要关注最重要的三个方面就可以了：

重点突出

例证生动

消除歧义

首先，“重点突出”，是最简单朴素、成本最低的“优秀结构”。

既然是沟通，就要了解对方。在绝大多数情况下，对方想要的才是重点。

可问题在于，我们平日里面对单个人的时候都觉得“了解对方很困难”，我们又如何判断“一个群

体”呢？与很多人想象得相反，判断群体远比判断个体容易得多…… 因为你可以用粗暴分类，比

如，你把听众或读者划分为“小白”和“专家”，那你就知道了，你面对的群体中，更可能“大部分是小

白，小部分是专家”。于是，你就可以思考，“小白们最想知道的是什么？”，“专家们最重视的是什

么？”，于是，你就可以在脑子里对所谓的“重点”有应对策略。

再比如说，你可以“粗暴”地把群体分为友善者和刺儿头两种。有经验的老师，都会专门准备“针对刺

儿头”的内容，因为，人群中永远存在刺儿头 —— 至少一个，所以，在现场，必须要有应对他们的

策略。比如，当一个刺儿头说了一句蛮不讲理却引发哄堂大笑的话，你怎么办？每个人的策略不

同，但你必须找到属于你的最佳策略。

进而，找到重点之后，就要做到紧接着必须做到的一步，“例证生动”。寻找好例子，是需要很多时

间精力的事情 —— 从来就没有任何“信手拈来”的生动例子。信手拈来，是读者或者听众的感觉而

已，对你来说，肯定是举重若轻的 —— 你明明是在举重，却被认为“如若举轻”。在第一部分，讲文

件的那一章里，我举了个“我自己费尽心机找好例子”的例子，相信你读过之后，一定有所感触。

最后一步，就是在前两步都完成之后，反复确认一件事情，消除一切歧义。这是真功夫，因为这东

西很难把自己关在屋子里自顾自练成。并且，每个人有自己的“容易引发误解”的特殊属性，大家各

不相同 —— 于是，只能靠自己探索。

无论如何都不能骂听众读者傻逼，他们的所有看起来傻逼的反应，都是你所说、你所写引发的 ——

这是百分之百清楚无误的事实。当年我写博客的时候，决不删除任何留言。其中最大的一个理由就

是，无论那留言显得多么荒谬，甚至干脆是谩骂，都值得我认真思考：

我到底说什么了？居然引来这种反应？！

看多了，思考多了，你就有你自己的策略了。

内容组织

只有一个重点的时候，其实并不需要组织；但，若是有一个以上的重点，那么这些重点之间会产生

逻辑关系：

并列

递进

转折

这是上中学的时候，所有人在语文课上都学习并掌握了的知识 —— 现在终于需要“活学活用”了。

在讲演、讲课、写教程、写非小说类书籍的时候，最有效的组织方式竟然是最简单的，并且只有一

个：

层层递进

你有两个重点需要分享，那么把更重要的那个重点放在后面；你有三个重点要分享，那么就把最重

要的放在最后面……　无论你有多少个重点，都按这种方式排列，准没错！

并且，另外一个建议是：

3 是重点数量的极限

如果你必须有 11 个重点，那怎么办？那就把它们分到最多 3 个分组中，比如，像当前文章这

样，“入门”、“进阶”、“高级”……

另外一个策略，是与“递进”的逻辑关系组合使用的 —— 你要区分针对你所谓的重点，对方的已知状

态。大家都知道的，何必当作重点？于是，未知程度最高的，放在未知程度不那么高的重点后面。

这都是听起来无比简单，甚至好像“无需讲述”的“重点”，但，若是你多观察一下周遭的世界，你就明

白了，很多人可能并不是不知道这些方法，但，不知道为什么，他们就是不去应用这么简单有效的

方法，也是怪得很！

进阶

当入门的手段都已经熟练了，就可以做很多所谓“锦上添花”的事情。锦上添花据说有很多种手段，

比如，制造笑点啊，使用炫酷的幻灯片啊等等……　但，我只想给你讲

一个学会了就够用，却也是最

简单、最直接，然而又最有效的手段：

输送价值观

事实上，你可以把当前这本“书”当作一个巨大且又生动的例子：

你把李笑来想象成一位在某个学校里为学生讲编程课的老师。那么，你就可以把当

前这本书当作“李笑来的讲义”…… 也许我和学生手里都拿着另外一本更著名的计算

机专家所著的编程入门书籍，然而，我的讲义，就是按照我的顺序、我的内容编排

来讲述的。

又，事实上，我在写这本书的时候，从某个层面上来看，真的写的就是“Python 官方文档”以

及“Python 官方教程”的辅助讲义，写作目标如此，写作方式亦如是。

那我做了什么最重要的事情呢？

我向我的群体输送了我觉得更有意义的价值观：

自学是门手艺……

当年我在学校里讲英语课的时候，除了讲英语本身之外，我输送的价值观是：

能管理好自己的时间的人，学英语学起来更容易……

后来，这一部分单独被提取出来，在我离开那所学校之后，写成了长销书，《把时间当作朋

友》—— 你看，是一样的道理。

而所谓的价值观，定义很简单：

你的价值观，就是你认为什么比什么更重要。

价值观可大可小。大到集体利益与个人利益之间的比较，小到自学中“全面”压倒一切……然而，这

世界总有独立于任何人存在的“客观的价值比较”，只不过，每个人的“价值观”是自己的“观点”。而我

们每个人都希望自己的观点尽量摆脱自己的主观，尽量靠近那个“客观的价值比较”…… 而一旦我们

确定自己比原来的自己，甚至相对于其他人更进一步的时候，就很可能值得认真分享。

这个方法着实简单，然而却非常有效。这有点像什么呢？这有点像人家弹个吉他是“蹦单音”，可你

弹的却是“曲调与和弦”相辅相成的音乐…… 给你听一个 Youtube 上，《一生所爱》这首曲子的我最

喜欢的指弹版本 —— 这种弹法，不仅有旋律，还有和弦，并且还有打击乐器效果“伴奏”：

In [6]:

```
from IPython.display import IFrame

IFrame('https://www.youtube.com/embed/AjWTop5O5jo?', width='800', height

='450')
```

Out[6]:

最初的时候，有个看起来很难以跨越的障碍：

感觉总是需要为自己塑造权威感 —— 否则就害怕没人听、没人看、没

人信……

这是很多人掉进去的坑。刚开始的时候，自己就不是权威啊！无论怎么装神弄鬼，事实上就不是

么！

很多人没想明白，因为害怕没人听、害怕没人看、害怕没人信，所以就开始各种作弊，包括各种装

神弄鬼，各种欺世盗名……　这么做，暂时管用，长期来看，肯定是吃亏的。

因为作弊其实并不难，装神弄鬼其实并不难，欺世盗名其实并不难 —— 真正难的是长期作弊……

长期欺骗有多难呢？难到根本不可能的地步。尤其是在寿命越来越久的今天。《庄子》里说，“寿则

多辱”，今天有了新解释 —— 大家都寿命很长，所以别骗人，因为早晚会露馅……

花那么长时间作弊什么的，还不如花那么长时间磨练手艺，你想想看是不是这个道理？

给你看一段视频，加州伯克利大学的　Brian　Harvey　在课堂上告诉学生，“为什么不要作弊”的真正原

因：

In [2]:

```
from IPython.display import IFrame

IFrame('https://www.youtube.com/embed/hMloyp6NI4E?', width='800', height

='450')
```

Out[2]:

于是，千万别扭曲了自己，是什么样就是什么样，该怎么做就怎么做。而另外一件事是确定的：

分享多了，就自然进步了……

在求知的领域里，分享得越多，进步越快，且社交有效性提高得更多。

高级

无论什么手艺，大多数人都可以入门、少数人可以进阶……　再往后，就通常被认为是个人“造

化”了。

可这所谓的“造化”究竟指的是什么呢？

我觉得通过这本书，我可以向绝大多数普通人解释这个“玄学词汇”了……

这里所谓的“造化”，指的应该是一个人的“融会贯通的能力” —— 有“造化”的人，不过

是把大量其他领域里的技能、技巧、甚至手艺学来而后应用到自己的手艺之中……

就这么简单。

有个特别好玩的例子。

现在大家已经熟悉了的吉他演奏中的“指弹”（Percussive Guitar），可这种玩法在没有 Youtube 的

时代里并不多见 —— 在我长大的年代里，甚至“前所未闻”。不知道是谁，把打击乐器的手法融合到

吉他演奏手法中去了，于是，在 Youtube 这样的视频工具出现之后，人们的“见识”成本降低了（过

去也许要“去西天取经才行”），很快就有人模仿，很快就有人更为擅长……

In [7]:

```
from IPython.display import IFrame

IFrame('https://www.youtube.com/embed/nY7GnAq6Znw?',       width='800', height

='450')
```

Out[7]:

有这种能力的人，普遍有两个特征：

他们自学很多看起来不相干的手艺

他们对自己的手艺充满尊重与热爱

我也只能猜个大概。这其中的第二个特征，很可能是第一个特征的根源，因为他们对自己的手艺充

满了尊重与热爱，所以，他们追求全面，他们刻意练习，他们还刻意思考…… 由此引发了对一切可

能与自己的手艺相关的东西都感兴趣 —— 虽然在外界看来那两样东西可能全无联系。于是，他们利

用已经在自己的手艺中练就的自学能力，不断自学新的东西，不断“发现”所谓的“新大陆”，不断用他

们的所见所闻回过来锤炼或更新自己的手艺……

所以，弹钢琴或者弹吉他的也去学了打击乐器；讲课的人会去听相声专场，学习相声演员是如何抖

包袱的；写书的人可能会像我琢磨“取名”的艺术 —— 甚至不惜去研究一下自己并不屑于相信的易经

八卦；学会计的去研究了物理或者经济学；学编程的也去学了设计，学设计的也去学了编程；做前

端的去学了后端，做后端的也学会了前端；做统计的学会了数据可视化…… 搞来搞去，计算机行业

里有个著名的词汇诞生：全栈工程师。

其实，所有精湛的手艺人，都是全栈，不信你就仔细观察一下。

于是，所有在入门、进阶之后走得更远的手艺人，都明白且认同这个道理：

学无止境

于是，最后一个重要技巧，不仅仅是“不断磨练当前的手艺”，还有就是不断向所有的手艺人学习。

再进一步，技巧没用了…… 想再进一步，靠的是另外一个层次的东西 —— 那就是尊重与热爱。

这么多年来，在互联网上我最喜爱的老师，是麻省理工大学的 Walter Lewin 教授。

In [8]:

```
from IPython.display import IFrame

IFrame('https://www.youtube.com/embed/sJG-rXBbmCc?', width='800', height

='450')
```

Out[8]:

十多年前，MIT 出了一套 Open Course，在线的免费课程。授课老师绝大多数都是本校的著名教授

—— 在那么多课程里，我一下子就爱上了这位教授。建议你有时间把

他的所有课程看完，虽然你可

能觉得物理这东西你并不感兴趣 —— 可事实上，看完你就知道了，你只不过是运气不好，你从来没

遇到过这么可爱牛逼的教授而已…… 在他身上，你可以学到无数，甚至是“不可言说”的技巧、秘密

和手艺。

关键在于，你一定会非常生动地、深刻地体会到他对物理、他对授课的尊重和热爱。看过之后，你

一定会跟我有一样的慨叹：“是哦，**Love is the power.** ”

没有什么比“热爱”和“尊重”更为高级的了。就这样。

自学者的终点

…… 磨练自学手艺的你，早晚会遇到这种情况：

必须学会一个没人教、没人带，甚至没有书可参考的技能。

这也许是我自己能够想象到的自学者所能经历的最高境界吧，因为那么多年过去之后的我也只不过

走到这个地方而已…… 许多年过去，我通过自学习得的没有人教、没有人带、甚至没有书可参考的

技能，很拿得出手的只有两个而已：

制作长销书

区块链投资

先说说区块链投资。

2011 年，我开始投资比特币的时候，不像现在。现在你能在 Amazon 上找到一大堆书，给你讲区

块链技术、区块链投资…… 2011 年的时候，互联网这本大书上，连篇像样的文章都几乎没有。

在这样的领域里成为专家，其实比外界想象得容易 —— 当然，无论什么，外界总是觉得很难。为什

么比外界想象得更容易呢？因为大家都不懂，仅此而已。

所以，剩下的事情很简单：

谁能持续研究谁就可能更先成为专家。

到最后，还是一样的，决定因素在于有效时间投入，再次仅此而已。

了解我的人都知道，在知识技能的分享方面，我从来都没有“藏着掖着”的习惯。我说“仅此而已”的时

候，那就是“仅此而已”，没有任何其它保留。

说研读比特币白皮书，就真的研读，反复研读，每年都要重复若干遍 —— 有人问了，有那个必要

吗？是哦，对我有没有那个必要，我自己说了算。并且，就算真有，告诉别人有什么用？对吧？

这里是《比特币白皮书》我翻译的版本。

说投资就真的投资，不是“买两个试试”、“买几个玩玩” —— 我的做法是重仓。持仓之后继续研究，

和两袖清风地读书研读肯定不一样。有一个与我极端相反的例子。此人大名鼎鼎，是《精通比特

币》（Mastering Bitcoin）的作者，Andreas M. Antonopoulos，与我同岁，也是 1972 年生人。

他也是国外公认的比特币专家，但他不是投资人 —— 他几乎没有比特币。2017 年牛市的时候，人

们听说大神 Andreas M. Antonopoulos 竟然几乎没有比特币，大为惊讶，向他捐款总计 102 个比特

币左右 —— 千万不要误会我，我没有任何鄙视他的意思。这不是我说说而已，我是用行动支持他的

人。他的书，《精通比特币》的中文版，还是我组织人翻译的。

我只是说，我和他不一样的地方在于，在某些方面我比他更实在，这也是事实。我相信我在实际投

资后，对比特币也好区块链也罢，理解力会更深一些，因为驱动力不一样么，多简单的道理。

然而，仅仅“谁能持续研究谁就更可能先成为专家”这一条，其实并不够完成“学会一个没人教、没人

带，甚至没有书可参考的技能”。

这么多年来，我能够“学会一些没人教、没人带，甚至没有书可参考的技能”（比如赚钱这事儿就是

如此），更多的时候仰仗的是一个我已经告诉过你的“秘密”…… 也许你想不起来了，但，我一说你

就能“发现” 其实你真的已经听我说过：

刻意思考：这东西我还能用在哪儿呢？

并且，我还讲过在自学编程的过程中，见识到的 MoSCoW Method 给我在写书方式上的影响。

我写书就是这样的。在准备的过程中 —— 这个过程比绝大多数人想象得长很多

—— 我会罗列所有我能想到的相关话题…… 等我觉得已经再也没有什么可补充的时

候，再为这些话题写上几句话构成的大纲…… 这时候就会发现很多话

题其实应该是

同一个话题。如此这般，一次扩张，一次搜索之后，就会进行下一步，应用

MoSCow 原则，给这些话题打上标签 —— 在这过程中，总是发现很多之前感觉必

要的话题，其实可以打上 Won't have 的标签，于是，把它们剔除，然后从

Must have 开始写起，直到 Should have ，至于 Could have 看时间是否

允许，看情况，比如，看有没有最后期限限制……

在写书这事儿上，我总是给人感觉很快，事实上也是，因为有方法论 —— 但显然，

那方法论不是从某一本“如何写书”的书里获得的，而是从另外一个看起来完全不相

关的领域里习得后琢磨到的……

你看，把另外一个领域里的知识拿过来用，是在一个“没人教、也没人带，甚至没有书籍可供参

考”的领域中，最基本的生存技巧。

再进一步，当我们在最开始的时候说，“尽量只靠阅读习得一项新技能”的时候，有一个重点现在终

于在之前的很多解释与讲解之后能说清楚了：

我们并不是不要老师这个角色了，准确地讲，我们只不过是不再需要“传统意义上的

老师”了而已。

首先，我们把自己当作老师 —— 英文中，Self-teaching 这个词特别好，

它描述得太准确了。很多

的时候，想在身边找到好老师是很难的，甚至是不可能的。在这种情况下，我们没有别的选择，我

们只能把自己当作老师去教自己。

其次，就算我们 100% 只依靠阅读，那内容不还是别人写的吗？写那内容的人，实际上就是老师。

没错，书本，是历史上最早的远程教育形式，即便到今天也依然是最重要最有效的远程教学形式。

阅读的好处在于，对老师的要求中没有地理位置的限制，若是能自由阅读英文，那就连国界限制都

没有。“书中自有颜如玉”这句话，显然是并不好色的人说的，因为他更爱书…… 这句话的意思其实

是说：

各路牛人都在书里……

反正，写书的人群中，牛人比例相对较高，这是事实 —— 古今中外都一样。

进而，更为重要的是，一旦当你把整个互联网当作一本大“书”，把 Google 当作入口，实际发生的效

果是：

你把“老师”这个角色去中心化了……

一方面“老师”这个角色的负担降低了，他们不用管你是谁，也不用管你怎么样了，他们该干嘛就干

嘛；而另外一方面则对你更重要 —— 你学不同的东西，就可以找不同

的老师；即便是相同的东西，

你也可以找很多老师；对于任何一个老师，你都可以“弱水三千只取一瓢”，也就是说，只挑他最厉

害的部分去学……　不就是多买几本书吗？不就是多搜索几次、多读一些文档吗？

最后，你竟然还有最厉害的一个小招数：

无论学会什么，都要进一步刻意思考：这东西我还能用在哪儿呢？

于是，你“一下子”就升级了 —— 用这样的方式，相对于别人，你最可能“学会几乎任何一个没人教、

没人带，甚至没有书可参考的技能”……

你看看自己的路径罢：从“不得不把自己当作老师去教自己”开始　——虽然起步是不得不，但这个“不

得不”，恰好是后来你变得更为强大的原因和起点…… 这就解释了为什么历史上有很多牛人的很多

成就其实都是这样“被迫”获得的。

于是，我们终于可以好好总结一下了：

你一定要想办法启动自学，否则你没有未来；

你把自学当作一门手艺，长期反复磨练它；

你懂得学、练、用、造各个阶段之间的不同，以及针对每个阶段的对应策略；

面对“过早引用”过多的世界，你有你的应对方式；

你会“囫囵吞枣”，你会“重复重复再重复”，你深刻理解“读书百遍其义自见”；

以后你最擅长的技能之一就是拆解拆解再拆解；

你用你的拆解手艺把所有遇到的难点都拆解成能搞定的小任务；

自学任何一门手艺之前你都不会去问“有什么用”，而是清楚地知道，无论是什

么只要学会了就只能也必然天天去用；

你没有刚需幻觉，你也没有时间幻觉，你更没有困难幻觉，反正你就是相对更

清醒；

不管你新学什么手艺，你都知道只要假以时日你就肯定能做好，因为所有的手

艺精湛，靠的只不过是充足的预算；

你知道如何不浪费生命，因为只要不是在刻意练习、不是在刻意思考，那就是

在“混时间”；

你总是在琢磨你能做个什么新作品；

你刻意地使用你的作品作为有效社交工具，也用作品去过滤无效社交；

你乐于分享，乐于阅读也更乐于写作 —— 因为这世界怎么帮助你的，你就想着

要怎样回报；

你把全面和完整当作最高衡量标准，也用这个标准去克制、应对自己的注意力

漂移；

你会不断自学新的手艺，因为你越来越理解单一技能的脆弱，越来越理

解多项

技能的综合威力；

你越来越依赖互联网，它是你最喜欢的“书”，而Google 是你最好的朋友——

他总是能帮你找到更好的老师；

偶尔，你会学会没人教、没人带、甚至没书可参考的手艺，别人都说你“悟

性”高，可你自己清楚地知道那其实是怎么回事儿；

你越来越明白，其实没什么“秘密”，越简单、越朴素的道理越值得重视；

你发现你用来思考的时间越来越多 —— 准确地讲，是“琢磨”…… 只不过是因为

你真会琢磨了 —— 你很清楚你应该花时间琢磨的是什么。

没有人教过我怎么写一本长销书（而不仅仅是畅销书），这显然是我通过自学习得的能力 —— 我也

只能把自己当作老师教自己，这是不得已。然而，不得不的选择同样常常能给我带来好运…… 甚

至，它也不是我通过阅读哪本书习得的能力 —— 因为这方面还真的没什么靠谱的书籍。然而，我竟

然学会了 —— 靠什么？靠的就是上面说的那点事儿而已。

“秘密”是什么？说出来后，你听起来肯定是感觉“太简单了”乃至于“有点不像真的”…… 每次，我都很

认真的问自己以下几个问题：

我要写的内容，的确是正确的吗？

我要写的内容，确实会对读者有用吗？

有什么内容是必须有的、应该有的？

我写的这个内容，十年后人们再看，还会觉得跟新的一样有用嘛？

我的书名，就那么放在那里，会直接让买家产生不由自主购买的吸引力吗？

一旦这几个问题我有了清楚的答案，我就知道，我有能力制作一本新的长销书了 —— 真的没有什么

别的“秘密”。

在《通往财富自由之路》中，我分享过以下内容：

我认为一个人的自学能力（当时还在用“学习能力”这个词）分为三个层次：

1. 学会有人手把手教授的技能

2. 学会书本上所教授的技能

3. 学会没有人能教授的技能

这一次，我无非是把后两个层面用一个特大号的实例掰开了揉碎了讲清楚而已。

到最后，没什么不能自学的，反正都只不过是手艺 —— 只不过，我们每个人都受限于自己的时间精

力而已。所以，若是你正在读高一，正在读大一，那就好好珍惜自己有机会可以随意设置充裕预

算的时光罢。若是你已为人父母，那就想办法用自己的行动影响下一代罢。然而，其实更为重要的

是，无论什么时候，都要这么想：

若是还有很长的未来，现在真不算晚……

自学不过是一门手艺，而且还是谁都能掌握的。不要“试试”而已，而是“直接开干” —— 这样才好。

最后还有个需要补充的是：很多人崇尚“刻苦”，并且刻意强调其中的“苦” —— 古训中所谓的“吃得苦

中苦，方为人上人” —— 这一点我并不认同，而且还是深刻地不认同。

我的观察是，所谓的“苦”，是那些完全不会自学的人对自学者的所作所为的错误理解。

自学一点都不苦，道理也很简单：

因为自学者是自发去学的，原动力在于自己。而不像其他人，是被动地学，原动力

并非在于自己。

由于原动力在于自己，遇到困难时，当然同样苦恼；可不一样的是，有持续的原动力去克服那些困

难，于是，总是在不断克服困难之后获得更大的愉悦、更大的满足感。

所以，“刻”，我们很认同，刻意地练习、刻意地思考，刻意地保持好奇心，刻意地去学习一些看起

来与当前所掌握的手艺完全不相干的知识…… 至于“苦”么，那是别人的误解，我们自己开心着呢

—— 无所不在、无处诉说的幸福。

人生苦长，无需惊慌。

祝你好运！

李笑来

初稿完成于 *2019* 年 *2* 月 *27* 日

下一步干什么？

理论上，下一步你的选择很多。自学是门手艺，你可以用它去学任何你想要掌握的其它手艺。如

果，你有意在编程这个领域继续深入，那么，以下就是一些不错的线索。

当然，最先应当做的是，去检查一下自己的“突击”的结果，去 Pythonbasics.org 做做练习：

https://pythonbasics.org/Exercises/

除了我在这里介绍的之外，请移步 The Hitchhiker's Guide to Python，它更为全面：

https://docs.python-guide.org/

Python 必读书籍

无论学什么，一本书肯定不够，以下是学习 Python 的基本必读书籍：

The Python Tutorial

The Hitchhiker’s Guide to Python!

Think Python: How to think like a computer scientist

Automate the Boring Stuff with Python

Effective Python

Python Cookbook

Fluent Python

Problem Solving with Algorithms and Data Structures using Python

Mastering Object-oriented Python - Transform Your Approach to Python

Programming

更多 Python 书籍：

https://pythonbooks.revolunet.com

千万别觉得多，只要真的全面掌握，后面再学别的，速度上都会因此快出很多很多……

Python Cheatsheet

你已经知道了，这种东西，肯定是自己整理的才对自己真的很有用……不过，你也可以把别人整理

的东西当作“用来检查自己是否有所遗漏”的工具。

网上有无数 Python Cheatsheets，以下是 3 个我个人认为相当不错的：

Comprehensive Python Cheatsheet

Python Crash Course - Cheat Sheets

Pysheeet

Awesome Python

Github 上的“居民”现在已经养成了一个惯例，无论什么好东西，他们都会为其只做一个“Awesome

...”的页面，在里面齐心协力搜集相关资源。比如，你想学 Golang，那你去 Google 搜索 Awesome

Go，一定会给你指向到一个 Github 上的 “Awesome Go”的页面……

以下是 Awesome Python 的链接：

https://github.com/vinta/awesome-python

CS 专业的人都在学什么？

如果你真有兴趣把这门手艺学精，不妨看看 Computer Science 专业的人都在学什么……

下面这个链接值得认真阅读：

http://matt.might.net/articles/what-cs-majors-should-know/

全栈工程师路径图

既然学了，就肯定不止 Python —— 在扎实的基础之上，学得越多学得越快。以下是一个“全栈工程

师路径图”，作者是位迪拜的帅哥 Kamran Ahmed：

https://github.com/kamranahmedse/developer-roadmap

Below you find a set of charts demonstrating the paths that you can take and the technologies that you would want to adopt in order to become a frontend, backend or a devops. I made these charts for an old professor of mine who wanted

something to share with his college students to give them a perspective; sharing them here to help the community.

Check out my blog and say "hi" on Twitter.

Disclaimer

The purpose of these roadmaps is to give you an idea about the landscape and to guide you if you are confused about what to learn next and not to encourage you to pick what is hip and trendy. You should grow some understanding of why one tool would better suited for some cases than the other and remember hip and

trendy never means best suited for the job

Introduction

Frontend Roadmap

Back-end Roadmap

DevOps Roadmap

路漫漫其修远兮……

但，多有意思啊？这完全就是一场闯关游戏。

Visual Studio Code 的安装与配置

官方文档请查询：

https://code.visualstudio.com/docs

允许命令行启动 **VS Code**

使用快捷键

p 呼出 Command Palette，在其中输入 shell command ， 而后选中 Install 'code'

command in PATH 。此后，就可以在 Terminal 命令行中使用 code 命令了。

选择 **Python** 解析器版本

使用快捷键

p 呼出 Command Palette，在其中输入 select interpreter ，而后选中 Python:

Select Interpreter 。

而后，在系统中已安装的若干个版本中选择你需要的那一个。MacOS 系统自带一个 Python 2.7，

而我们安装的 Anaconda 为系统另外安装了一个 Python 3.7。

安装扩展

使用快捷键

x 呼出扩展面板。安装 anaconda 扩展，它会连带装上 python 扩展：

另外，为了输入方便，有两个扩展可选安装：

Tabout 有它之后，可以使用 TAB 键跳出光标后的括号、引号等等；

Sublime Text Keymap and Settings Importer 有它之后，可以在 VS Code 中使

用 SublimeText 的快捷键，最重要的当属多光标编辑

l ……

自动补全

专业编辑器最重要的功能之一，就是能够在你输入的时候它帮你做到“自动补全”，通常使用的快捷

键是 TAB 键

。

TAB 键

触发的自动补全有两种：

当前文件中已有的字符串。比如，之前你输入过 sum_of_word ；那么，之后，

你就可以输入 su 或者干脆 sow 而后按 TAB 键

，“自动补全”功能会帮你

完成输入 sum_of_word

已有的 Snippets。比如，当你需要输入 if ...: ... 的时候，实际上当你输入 if

或者甚至 i 之后，你就可以用 TAB 键

，“自动补全”功能会为你“自动完

成”语句块的输入。

字符串自动补全，使用的是所谓的 Fuzzy Match。输入 sum_of_word 中所包含的任意字符的任意

组合（按顺序），它都会尽量去匹配；所以， su 和 sow 都可以匹配 sum_of_word ，再比

如， rst 可以匹配 result 。

在 Snippet 自动补全的过程中，常常有若干个“TAB Stop”，即，有若干个位置可以使用 TAB 键

（或者， Shift +

）来回切换；这时，第一种字符串自动补全的功能就失效了，如果需要使用字

符串自动补全，那么需要按快捷键 ESC

退出 Snippet 自动补全模式。

以下的 gif 文件演示的是以下代码的输入过程：

```
def sum_of_word(word):

sum = 0

for char in word:

sum += ord(char) - 96

return sum

with open('results.txt', 'w') as results:

with open('words_alpha.txt', 'r') as file:

for word in file.readlines():

if sum_of_word(word.strip()) == 100:

results.write(word)
```

因为有这样的功能，所以你在输入程序的时候其实是非常从容的，可以很慢输入，边思考边输

入…… 可实际上，完成速度却很快。

另外，SublimeText 的多光标输入是很多程序员爱不释手的功能，于是，各种编辑器里都有第三方

写的 SublimeText Keymap 插件，连 Jupyterlab 都有：

https://github.com/ryantam626/jupyterlab_sublime

Git 简介

--- You should've learned Git yesterday.

内容目标

再一次，这一篇内容的目标，依然不是“教程”，而是“教程”的“图例”——如果我们把真正的教程比喻

成“地图”的话。最全面的 Git 教程在网上， **Pro Git**，是免费的 —— 把它反复阅读若干遍，理解完

整：

https://git-scm.com/book/en/v2

并且还有各种语言的翻译版本 —— 也包括中文。

为什么你必须学会使用 **Git**？

Git 是一个分布式版本控制软件 —— 听起来也许跟你没关系，但，无论是谁，都会因为能够使用

Git 而节约时间、提高效率。进而，如果你居然没有一个活跃的 Github 账户，那么你正在错过人类

史上前所未有的共同协作时代 —— 半点都没有夸张。同样提供 Git 工具云服务的还有 Gitlab,

Bitbucket 等等。

并且，Github 很可能是地球上第一个给人们提供“用作品社交” 方式的平台，你若是不能参与其中，

实在是太可惜了！

从逻辑上理顺 **Git** 基本命令

Git 的作用，基本上可以被划分为三部分：

备份文件

跟踪文件变化

与他人协作共同操作文件

在一个 git 仓库中，总计有四个“抽象层”，它们分别是：

upstream repository 保存在云端的仓库

local repository 本地仓库

staging area 缓存区

working directory 工作区

其中， local repository 和 staging area 这两个抽象层的数据，保存在 working

directory 根目录下的一个隐藏目录 .git/ 下；需要使用 ls -a 才能看到。

当你使用 git init 命令将一个本地文件夹 working directory 初始化为 local

repository 的之后，该文件夹内部的结构如下：

```
.
└── .git
    ├── HEAD
    ├── config
    ├── description
    ├── hooks
    │   ├── applypatch-msg.sample
```

```
│ ├── commit-msg.sample
│ ├── fsmonitor-watchman.sample
│ ├── post-update.sample
│ ├── pre-applypatch.sample
│ ├── pre-commit.sample
│ ├── pre-push.sample
│ ├── pre-rebase.sample
│ ├── pre-receive.sample
│ ├── prepare-commit-msg.sample
│ └── update.sample
├── info
│ └── exclude
├── objects
│ ├── info
│ └── pack
└── refs
├── heads
└── tags
```

以下示意图中仅包含最基本的 Git 命令 —— 并且基本上都是独自使用 Git 时的常用命令。

在工作区 workding directory 这个抽象层中，你完成各种日常工作，创

建、编辑、修改……

你可能需要用某个编辑器去修改文件，你也可能频繁使用各种 Bash 命令，如， rm mkdir cp

mv 等等。

时不时，你可能会把一些处理完的文件“加入缓存区”；等一个阶段的工作完成之后，你可能会把那

些已经放入缓存区的文件提交到（commit）本地仓库；而后继续工作…… 根据情况，你也会将本地

仓库的文件推到（push）云端，即，远端仓库。如果，你正在与他人协作，你也可能经常需要从云

端下拉（pull）最新版本到本地。

Git 的安装

Mac

Mac 的操作系统 Mavericks (10.9) 以上版本中都内建有 Git，你可以在 Terminal 中通过以下命令查

看是否有 Git：

git --version

which git

也可以通过 Homebrew 安装最新版本的 Git：

ruby -e " **$(**curl -fsSL https://raw.githubusercontent.com/Homebrew/

install/master/install**)**"

brew doctor

brew install git

还可以通过 Conda 安装：

conda install -c anaconda git

Windows

前往 https://gitforwindows.org 下载并安装 Git for Windows。

此外，它还会提供 Git Bash —— 在 Windows 操作系统中使用与 *Nix 操作系统一样的 Bash 命令行

工具。

另外，在 Windows 操作系统中推荐使用 Git Bash 或者 PowerShell，而非 CMD 作为命令行工具。

Linux

大多数 Linux 操作系统要么基于 Debain，要么基于 Red-Hat —— 请自行查看 List of Linux

distributions，确定自己的 Linux 发行版究竟基于哪一个。

基于 Debian 的 Linux 发行版：

sudo apt-get update

sudo apt-get upgrade

sudo apt-get install git

基于 Red-Hat 的 Linux 发行版：

sudo yum upgrade

sudo yum install git

Git 本地配置

在命令行工具中执行以下命令：

git config --global user.name " "

git config --global user.email

Git GUI

Git 的图形化版本客户端，有很多种选择，以下几个跨平台的客户端最受欢迎：

Github Desktop

GitAhead

Fork

GitKraken

更多选择，请移步查看 git-scm.com 上的 Git GUI Clients 列表。

需要了解的 **Bash** 基本命令

虽然 Git 也有图形化版本，但，无论如何你都会接触到命令行工具。并且，谁都一样，早晚会遇到

非使用命令行不可的情况。

以下是常用 Bash 命令的简要说明：

命令

简要说明

cd

Change Directory 的缩写；转到指定目录

ls

List 的缩写；列出当前目录中的内容

mkdir

Make Directory 的缩写；在当前目录中创建一个新的目录

pwd

Present Working Directory 的缩写；显示当前工作目录

touch

创建一个指定名称的空新文件

rm

Remove 的缩写；删除指定文件

rmdir

Remove Directory 的缩写；删除指定目录

cp

Copy 的缩写；拷贝指定文件

mv

Move 的缩写；移动指定文件

cat

Concatenate 的缩写；在屏幕中显示文件内容

chmod

Change Mode 的缩写；改变文件的权限

man

Manual 的缩写；显示指定命令的使用说明

其中， chmod 最常用的 4 个权限分别是：

文件权限模式

简要说明

777

任何人都可以读、写、执行该文件

755

任何人都可以读、执行该文件，但只有创建者可以修改

700

只有创建者才能进行读、写、执行操作

+x

将文件设置为可执行

在使用 man 命令时，系统会使用 vim 文本编辑工具以只读模式打开帮助文件，常用键盘命令如

下：

键盘命令

简要说明

f

向后翻屏

b

向前翻屏

d

向后翻半屏

u

向前翻半屏

j

向后翻一行

k

向前翻一行

h

查看 vim 帮助

q

退出

一些不错的 **Git** 教程

除了 Pro Git 这本书之外，还有很多值得去看：

GIT CHEATSHEET -- an interaction from nop software

Learn Git Branching

Learn Git in a Month of Lunches

Git How To

Git pretty - Solve Git Mess

Visualizing Git Concepts with D3

当然，你肯定早晚会去 Github 上找 “Awesome Git”：

https://github.com/dictcp/awesome-git

Jupyterlab 的安装与配置

下载并安装 **Anaconda**

Anaconda 是目前最方便的 Python 发行版，搭载了很多我们终将必用的软件包，除了 Python 之

外，还有 R 语言，还包括 Pandoc， NumPy，SciPy，Matplotlib…… 等等。

无论是图形化界面安装，还是命令行界面安装，建议都把 Anaconda 安装在本地用户目录

内， ~/ 。请下载并安装 Python 3.x 的版本。

图形化界面安装的教程，官方的很详细，各个操作平台的说明都有：

https://docs.anaconda.com/anaconda/install/

在 MacOS 的 Terminal 命令行下，可以直接下载并安装：

cd ~/Downloads/

wget https://repo.anaconda.com/archive/Anaconda3-2018.12-MacOSX-x

86_64.sh

./Anaconda3-2018.12-MacOSX-x86_64.sh

安装到最后一步，会问你是否要安装微软出品的 Visual Studio Code，选择 yes —— 反正以后你

的电脑上会不止一个文本编辑器…… 以后你可能还会安装的文本编辑

器包括 SublimeText, Atom 等

等。

安装完毕之后，打开 Terminal，继续安装几个组件：

conda update conda

conda update anaconda

conda install -c conda-forge nodejs

conda install -c conda-forge jupyterlab # 这是用来升级 *jupyter lab* 到最新

版的方法

安装完毕之后，可以看看各个你将要用到的可执行命令都在什么地方，用 which 命令：

which python

python --version

which node

node -v

which jupyter

jupyter lab --version

jupyter notebook --version

which pip

pip --version

第一次启动 **Jupyter lab**

打开 Terminal， cd 到你想打开 Jupyter lab 的目录（就是你保存 ipynb 文

件的地方，以便在

Jupyter lab 中打开、浏览、编辑 ipynb 文件），在这里以用户根目录为例 ~/ ：

cd ~

jupyter lab

此时的 Terminal 窗口不能关闭，否则 Jupyter lab 就停止运行了 —— 就将它放在那里。

随后会有个浏览器打开，指向 http://localhost:8888/lab? —— 你就看到 Jupyter lab 的操作界面了。

目前，Jupyter lab 和 Jupyter notebook 是并存的，虽然前者是后者的下一步替代者。如果你依然习

惯于使用 Jupyter notebook，那么，在浏览器中指向 http://localhost:8888/tree? 看到的就是 Jupyter notebook.

配置 **Jupyter lab**

打开 Terminal，输入以下命令：

jupyter lab --generate-config

这会在 ~/.jupyter/ 目录下生成一个 jupyter_notebook_config.py 文件。

cd ~/.jupyter

code jupyter_notebook_config.py

上面的 code 命令，需要你已经安装 Visual Studio Code，并且在已经在其中设置了 Install 'code'

command in PATH 。参见附录 Visual Studio Code 的安装与配置

事实上，你可以用你喜欢的任何编辑器打开

~/.jupyter/jupyter_notebook_config.py 文件。

文件内容很长，有空可以仔细看。可以直接将以下内容拷贝粘贴到文件底部，根据需求修改：

c.NotebookApp.token = ''

c.NotebookApp.open_browser = False

c.NotebookApp.notebook_dir = '~/'

c.NotebookApp.default_url = '/tree'

逐条解释一下：

c.NotebookApp.token = ''

每次打开 Jupter，它都会给你生成一个新的 Token —— 这是安全策略。但是，如果你只是在自己

的电脑上使用，那么，这就给你制造了麻烦，因为若是你想同时用另外一个浏览器打开它，那你就

需要从 Terminal 里拷贝那个 Token 出来。所以，你可以在配置文件里直接把它设置为空：

c.NotebookApp.open_browser = False

每次你执行 jupyter lab 或者 jupyter notebook 命令的时候，它都会使用系统默认浏览器。

每个人的习惯不一样。比如我，会想到用一个平时不怎么用的浏览器专门用在 Jupyter 上，这样会

防止自己在关闭其它网页的时候不小心把 Jupyter 关掉…… 那我就会把这项设定为 False 。

c.NotebookApp.notebook_dir = '~/'

在 Terminal 中执行 jupyter 命令的时候，它默认是在你当前所在的工作目录打开 jupyter ，这同样

是出于安全考虑。但是，如果你只是在自己的电脑上使用，且只有自己在使用，那么莫不如直接把

它设置成 ~/ ，即，你的用户根目录，这样会很方便地访问各种地方的文件……

c.NotebookApp.default_url = '/tree'

这一项留给那些依然习惯于使用 jupter notebook 的人，这样设置之后，即便是输入 jupyter lab 命

令，打开的还是 jupyter notebook。

在 Terminal 里常用的与 Jupyter 有关的命令有：

jupyter lab

jupyter lab --version

conda install -c conda-forge jupyterlab # 这是用来升级 *jupyter lab* 到最新

版的方法

jupyter notebook list # 查看正在运行的 *jupyter lab/notebo ok*

jupyter notebook stop # 停止 *jupyter lab/notebook* 服务

将 **Jupyter lab** 配置成系统服务

如果，你厌烦每次都要跑到 Terminal 里启动 Jupyter lab，可以把它配置成系统服务，每次开机启

动它就自动运行。而你需要做的只不过是直接从浏览器中访问 http://localhost:8888/。

code ~/Library/LaunchAgents/com.jupyter.lab.plist

这条命令会让 Visual Studio Code 创建 ~/Library/LaunchAgents/com.jupyter.lab.plist 文件并打开。

在其中拷贝粘贴以下内容，注意，要把其中的 your_username 修改为你的用户名：

apple.com/DTDs/PropertyList-1.0.dtd">

version="1.0" >

Label

local.job

ProgramArguments

/Users/your_username/anaconda3/bin/jupyter

g>

lab

--no-browser

--notebook-dir=/Users/your_username/

RunAtLoad

StandardErrorPath

/tmp/local.job.err

StandardOutPath

/tmp/local.job.out

如果之前在 **jupyter_notebook_config.py** 文件里已经设置过

c.NotebookApp.open_browser = False

c.NotebookApp.notebook_dir = '~/'

那么这两行就可以不要了：

--no-browser

--notebook-dir=/Users/your_username/

而后在 **Terminal** 里执行：

launchctl load ~/Library/LaunchAgents/com.jupyter.lab.plist

如果你想重新启动这个服务，那么执行：

launchctl unload ~/Library/LaunchAgents/com.jupyter.lab.plist

launchctl load ~/Library/LaunchAgents/com.jupyter.lab.plist

关于 **Jupyter lab themes**

对中文用户来说，**Jupyter** 的默认字号有点过小，阅读起来不是很舒适。但，最佳的方案不是去寻找

合适的 **themes**，而是直接使用支持 **Stylus** 这类终端 **CSS** 定制插件的浏览

器，**Chrome/Firefox/Opera** 都支持 **Stylus** 插件。

我用的 **Stylus** 定制 **CSS** 是这样的：

a {color: #2456A4 !important;}

strong {color:#6392BF;}

em {color: #A9312A; font-style: normal !important;}

table {font-size: 90% !important;}

#jp-main-dock-panel {background-color: #f9f9f9;}

. jp-RenderedHTMLCommon {font-family: "Yuanti SC"; font-size: 100%

;}

. jp-Notebook {background-color: #fbfafa;}

. CodeMirror, . jp-RenderedHTMLCommon pre {font-size: 90%;}

. jp-RenderedHTMLCommon pre {

padding: 10px 25px;

background-color: #fafafa;

border-left: 4px solid #dadada;

border-radius: 10px;

}

. jp-RenderedHTMLCommon pre code {

background-color: #fafafa;

}

. jp-RenderedHTMLCommon h1 code,

. jp-RenderedHTMLCommon h2 code,

. jp-RenderedHTMLCommon h3 code,

. jp-RenderedHTMLCommon h4 code,

. jp-RenderedHTMLCommon p code,

. jp-RenderedHTMLCommon li code,

. jp-RenderedHTMLCommon blockquote p code,

. jp-RenderedHTMLCommon blockquote li code,

. jp-RenderedHTMLCommon td code {

background-color: #f6f6f6;

font-size: 90%;

color:#2e2e2e;

padding: 4px 4px;

margin: 0 8px;

box-shadow: 0px 1px 2px 0px rgba(0,0,0,0.2);

border-radius: 4px;

}

这样就相当于我把 **JupyterLab Light** 这个 **Theme** 稍微 **Tweak** 了一下。

另，我写的内容里，为了重点突出，特别定制了 **strong** 和 **em** 两个元素的显示，让它们以不同的

颜色展示；又因为中文并不适合斜体展示，所以，把 **em** 的 **font-style** 设定为 **normal ……**

安装插件

Jupyter notebook 经过很多年的发展，现在有很多扩展插件，但也有其中一些并不兼容最新的

Jupyter lab。不过，刚开始的时候用不着那么多插件，你只用其中的两个就足够开始了：

@jupyterlab/toc

ryantam626/jupyterlab_sublime

首先在用快捷键

, 打开 **Jupter lab** 的 **Advanced Settings**，在 **Extension Manager** 中，添加

User Overrides:

{

"enabled" : true

}

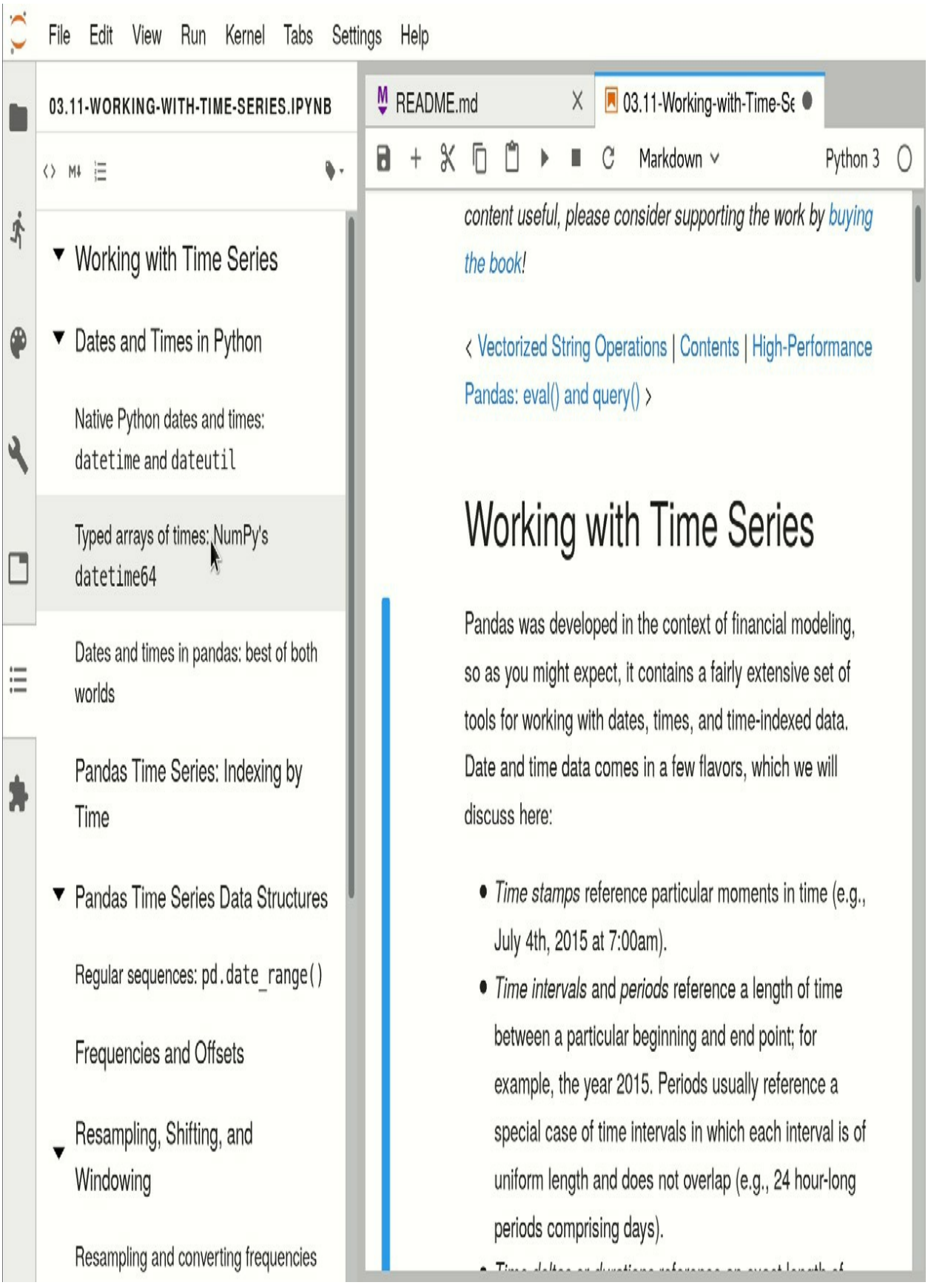
而后在 **Terminal** 执行以下命令安装插件：

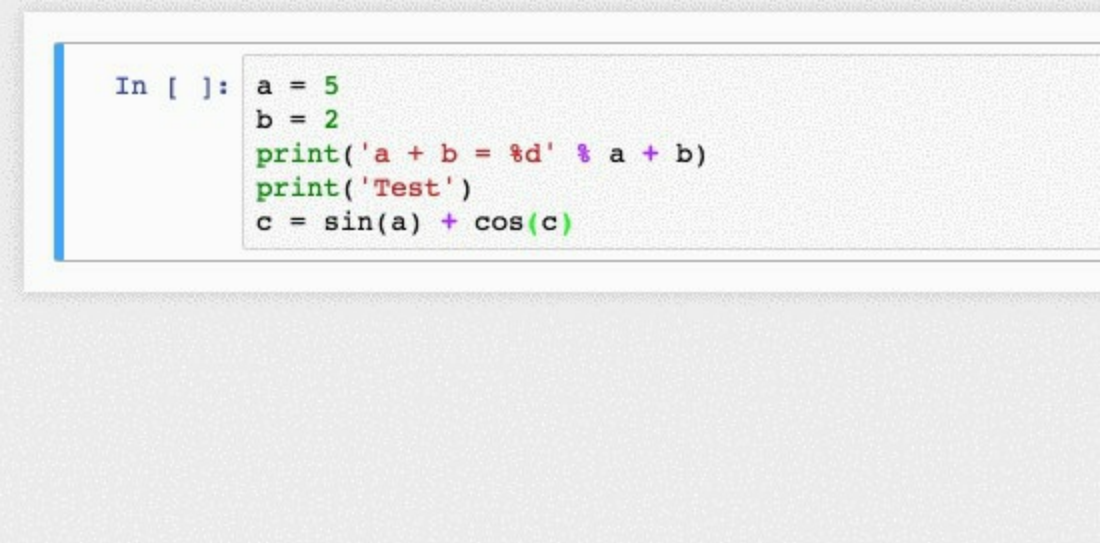
jupyter labextension install @jupyterlab/toc

jupyter labextension install @ryantam626/jupyterlab_sublime

jupyter lab build

toc 插件，自动将 **ipynb** 文件中的标题转换成目录。


File Edit View Run Kernel Tabs Settings Help
03.11-WORKING-WITH-TIME-SERIES.IPYNB
README.md
03.11-Working-with-Time-Se
Markdown
Python 3
Working with Time Series
Dates and Times in Python
Native Python dates and times: datetime and dateutil
Typed arrays of times: NumPy's datetime64
Dates and times in pandas: best of both worlds
Pandas Time Series: Indexing by Time
Pandas Time Series Data Structures
Regular sequences: pd.date_range()
Frequencies and Offsets
Resampling, Shifting, and Windowing
Resampling and converting frequencies
content useful, please consider supporting the work by buying the book!
‹ Vectorized String Operations | Contents | High-Performance Pandas: eval() and query() ›
Working with Time Series
Pandas was developed in the context of financial modeling, so as you might expect, it contains a fairly extensive set of tools for working with dates, times, and time-indexed data. Date and time data comes in a few flavors, which we will discuss here:
Time stamps reference particular moments in time (e.g., July 4th, 2015 at 7:00am).
Time intervals and periods reference a length of time between a particular beginning and end point; for example, the year 2015. Periods usually reference a special case of time intervals in which each interval is of uniform length and does not overlap (e.g., 24 hour-long periods comprising days).



jupyterlab_sublime 则可以让你在 **Jupyter lab** 的 **cell** 中，使用跟 **SublimeText** 一样的快捷键，比如

D 能够多选其它与当前选中内容一样的内容；比如

加鼠标点击，可以生成多个可编辑

点……

常用快捷键

以下是 **MacOS** 下 **Jupyter lab** 最常用的快捷键。快捷键在两种模式下执行，进入编辑模式用

，

回到命令模式用

（**ESC**）。

另外，代码编辑过程中需要安装 **Jupyterlab** 插件 **@ryantam626/jupyterlab_sublime** 之后才能使

用**“**多行同时编辑功能**”**。

快捷键

说明 模式

ESC

从编辑模式回到命令模式 命令

A

在当前 **Cell** 之前插入一个 **Cell**

B

在当前 **Cell** 之后插入一个 **Cell**

D , D

连续按两次 **d** 键，删除当前 **Cell**

Y

将当前 **Cell** 设置为 **Code Cell**

M

将当前 **Cell** 设置为 **Markdown Cell**

^

-

将当前 **Cell** 拆分为两个

M

合并选中的 **Cells**

J or

↓

连续向下选中 **Cells**

K or

↑

连续向上选中 **Cells**

or ^

运行当前 **Cell** 中的代码

L

显示/隐藏代码行号

当前 **Cell** 进入编辑模式 编辑

自动补全代码

呼出当前光标下词汇的 **Docstring**

D

Sublime Keymap: 选中下一个相同字符串

L

Sublime Keymap: 在选中的行内启动多行同时编辑

+ Mouse Click

生成下一个可同时编辑的光标点

增加一些必要的快捷键

在 **Settings > Keyboard Shortcuts** 中，可以设定一些常用但系统并未给出的快捷键：

```
{
"notebook:move-cells-down-down" : {
"command" : "notebook:move-cell-down",
"keys" : [
"Alt J"
],
"selector" : ".jp-Notebook:focus",
"title" : "Move Cells Down",
"category" : "Notebook Cell Operations"
},
"notebook:move-cells-down-up" : {
"command" : "notebook:move-cell-up",
"keys" : [
"Alt K"
],
"selector" : ".jp-Notebook:focus",
"title" : "Move Cells Down",
"category" : "Notebook Cell Operations"
},
"notebook:enable-output-scrolling" : {
```

"command" : "notebook:enable-output-scrolling",

"keys" : [

```
In [19]: 1

Out[19]: 1

In [20]: 2

Out[20]: 2

In [21]: 3

Out[21]: 3

In [22]: 4

Out[22]: 4

In [23]: 5

Out[23]: 5

In [24]: 6

Out[24]: 6

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:
```

"S"

],

"selector" : ".jp-Notebook:focus",

"title" : "Enable output scrolling",

"category" : "Notebook Cell Operations"

},

"notebook:disable-output-scrolling" : {

"command" : "notebook:disable-output-scrolling",

"keys" : [

"Alt S"

],

"selector" : ".jp-Notebook:focus",

"title" : "Enable output scrolling",

"category" : "Notebook Cell Operations"

}

}

这样就添加了 **4** 个快捷键：

J : Move selected cells up

K : Move selected cells down

S : Enable output scrolling

S : Disable output scrolling

比如 **Move Selected cells up**：

输出所有变量内容

默认情况下，**Code Cell** 只输出最后一个可以被 **evaluate** 的值，用 **_** 代表之前刚刚被 **evaluate** 的

值。

In [16]:

[1, 2, 3]

Out[16]: [1, 2, 3]

In [17]:

_ # 执行完上面的 ***Cell***，试试这个 ***Cell;*** 而后执行完下面的 ***Cell*** 之后再重新执行一次当前这个 ***Cell*** **Out[17]: [1, 2, 3]**

In [13]:

(1, 2, 3)

{1, 2, 3}

Out[13]: {1, 2, 3}

于是，为了显示最近 **evaluate** 的多个值，我们总是不得不使用很多的 **print() ……**

如果觉得这事儿比较烦的话，可以在 **Cell** 最上面写上：

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

如果还想更省事儿一点，就把这个设置写入配置文件：

c.InteractiveShell.ast_node_interactivity = "all"

In [21]:

from IPython.core.interactiveshell import InteractiveShell InteractiveShell.ast_node_interactivity = "all"

(1, 2, 3)

{1, 2, 3}

Out[21]: (1, 2, 3)

Out[21]: {1, 2, 3}

魔法函数

在 **Code Cell** 里，可以运行一些**“**魔法函数**”**（**Magic Functions**），这是秉承了 **IPython** 的特性。绝

大多数在 **IPython** 里能够使用的魔法函数在 **Jupyterlab** 里都可以直接使用。完整的 **IPython** 魔法函

数请参照：

https://ipython.readthedocs.io/en/stable/interactive/magics.html

Jupyterlab 里较为常用的魔法函数整理如下：

魔法函数

说明

%lsmagic

列出所有可被使用的 **Jupyter lab** 魔法函数

%run

在 **Cell** 中运行 **.py** 文件： **%run file_name**

%who

列出所有当前 **Global Scope** 中的变量；类似的还有： **%who df** ，**%whos**

%env

列出当前的环境变量

%load

将其他文件内容导入 **Cell**， **%load source** ， **source** 可以是文件名，也可以是 **URL**。

%time

返回 **Cell** 内代码执行的时间，相关的还有 **%timeit**

%writefile

把 **Cell** 的内容写入文件， **%write file_name** ； **%write -a file_name**， **-a** 是追加

%matplotlib

行内展示 **matplotlib** 的结果

inline

运行随后的 **shell** 命令，比如 **%%bash ls**；与之类似的还有 **%%HTML** ， **%%python2** ，

%%bash

%%python3 ， **%%ruby** ， **%%perl ……**

桌面版 **Jupyter App**

Nteract

支持各个操作系统，很好看、很好用。有一个小缺点是，不支持 **input()** 函数的调用。

https://nteract.io/desktop

~/code/src/github.com/nteract/nteract/example-notebooks/pandas-to-geojson.ipynb - Python 3 - idle *

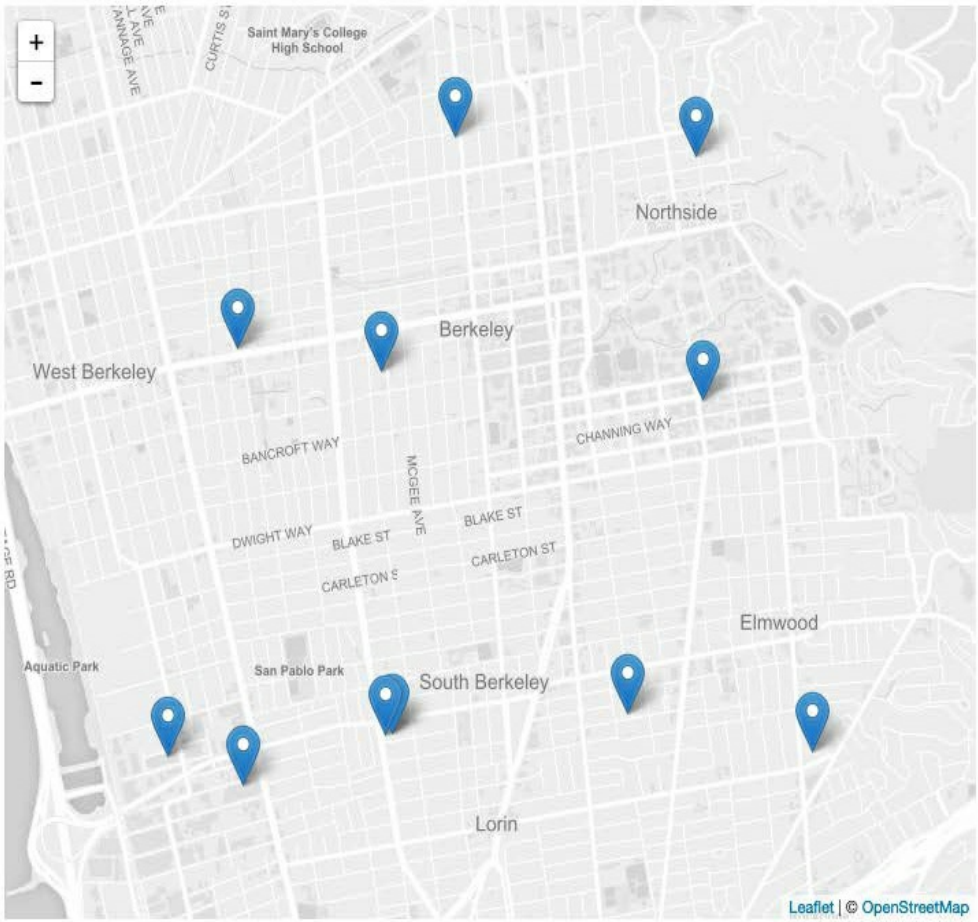
Bamboo

Converting from Pandas DataFrame to GeoJSON to Leaflet Map

[12]
```
import IPython
cols = ['street_address', 'issue_description', 'issue_type', 'ticket_status']
geojson = df_to_geojson(df_geo, cols)

def plot_geo(geojson):
  IPython.display.display({'application/vnd.geo+json': geojson}, raw=True)
```

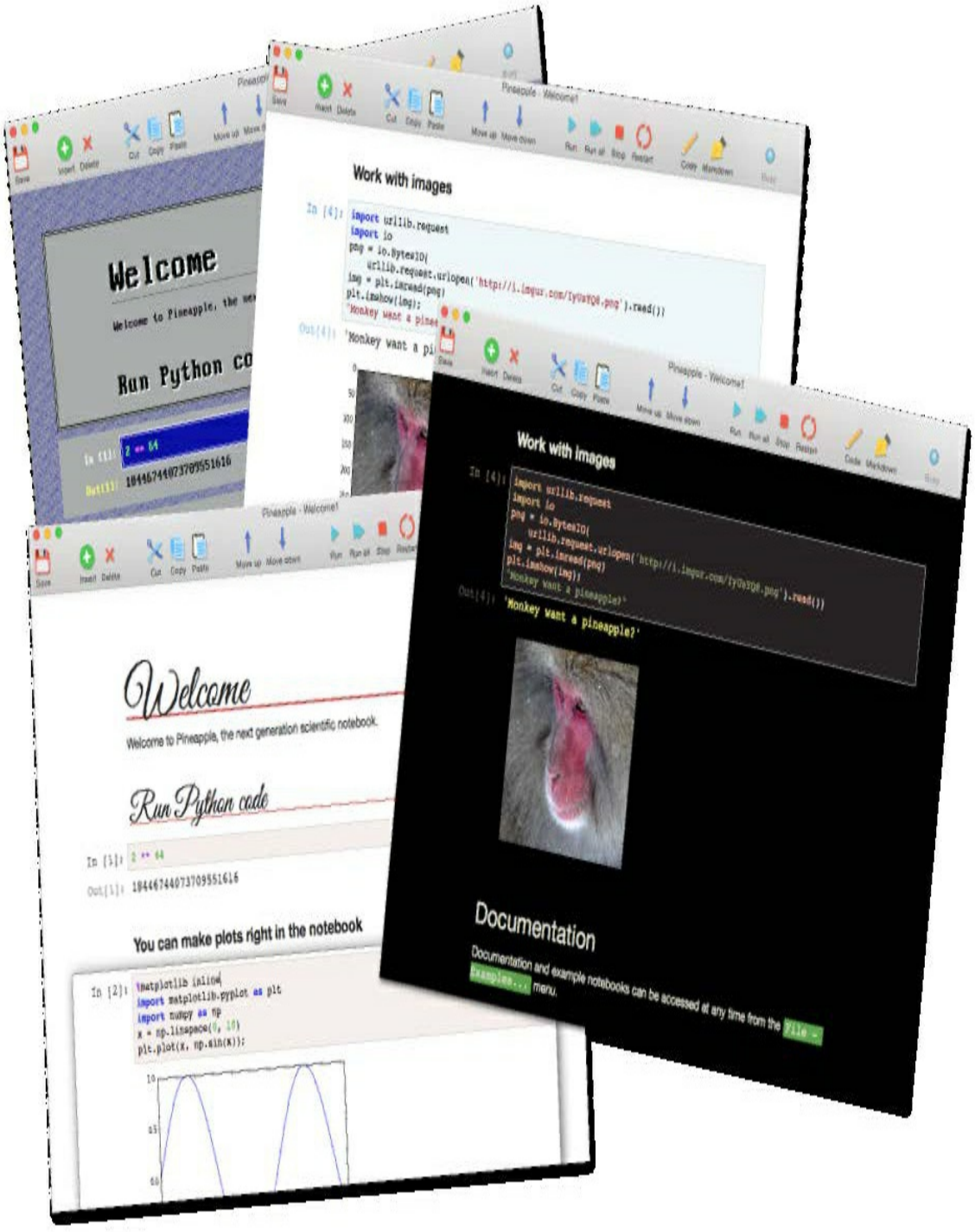
[22]
```
plot_geo(geojson)
```



Pineapple

只支持 **MacOS**，也很好用 **——** 缺点就是很难看**……**

https://nwhitehead.github.io/pineapple/


Welcome
Run Python co
Work with images
Welcome
Welcome to Pineapple, the next generation scientific notebook.
Run Python code
You can make plots right in the notebook
Documentation
Documentation and example notebooks can be accessed at any time from the File - Examples... menu.

这些符号都代表什么？

以下的表格你可以用很多次 —— 每次学一门新语言的时候，都可以拿它整理一遍思路……

最初的时候，人们能向计算机输入的只能是 **ASCII** 码表中的字符。于是从一开始，计算机科学家们

就绞尽脑汁去琢磨怎么把这些符号用好、用足……

于是，**ASCII** 码表中的字符常常在不同的地方有不同的用处。比如，**.**，在操作系统中，常常当作

文件名和扩展名之间的分隔符；在很多编程语言中，都被当作调用 **Class Attributes** 和 **Class**

Methods 的符号；在正则表达式中，**.** 代表除 **\r \n** 之外的任意字符……

把下面的表格打印出来，整理一下，在表格里填写每个符号在 **Python** 中都是用来做什么的？**[1]**

当前文件夹之中，有 **symbols.numbers** 文件，是用来打印以下表格的……

以后不管学什么语言，就拿这个表格过一遍，到时候只有一个感觉：越学越简单！

很多人最初的时候学不进去，就是因为“一些符号的用法太多了，经常混淆，于是就觉得累了、烦

了……” 然而，只要多重复几次多在脑子里过几遍就熟悉了 —— 若是真的熟悉了、若是真的掌握

了，你就会觉得：真的没什么啊！哪儿有那么难啊？！

学编程的一个好玩之处就是：

但凡你学会了任何一门编程语言，你就会发现你的键盘上的每一个键

你都用得

上……

可是在此之前，你跟人家花了同样的价钱买来的硬件设备，你竟然有那么多根本就没用过的东西！

普通人花同样价钱买来的是台机器而已；可是你却不一样 —— 用同样的价钱买来的是一个特别听话

的机器人，一个可以替你不分昼夜不辞辛苦干活的机器人 —— 这感觉非常神奇。

脚注

[1]：Python 语法，请参阅 **The Python Language Reference** **2. Lexical analysis**

↑Back to Content↑

Document Outline

- 00.cover
- 01.preface
- 02.proof-of-work
- Part.1.A.better.teachyourself
- Part.1.B.why.start.from.learning.coding
- Part.1.C.must.learn.sth.only.by.reading
- Part.1.D.preparation.for.reading
- Part.1.E.1.entrance
- Part.1.E.2.values-and-their-operators
- Part.1.E.3.controlflow
- Part.1.E.4.functions
- Part.1.E.5.strings
- Part.1.E.6.containers
- Part.1.E.7.files
- Part.1.F.deal-with-forward-references
- Part.1.G.The-Python-Tutorial-local
- Part.2.A.clumsy-and-patience
- Part.2.B.deliberate-practicing
- Part.2.C.why-start-from-writing-functions
- Part.2.D.1-args
- Part.2.D.2-aargs
- Part.2.D.3-lambda
- Part.2.D.4-recursion
- Part.2.D.5-docstrings
- Part.2.D.6-modules
- Part.2.D.7-tdd
- Part.2.D.8-main
- Part.2.E.deliberate-thinking
- Part.3.A.conquering-difficulties
- Part.3.B.1.classes-1
- Part.3.B.2.classes-2
- Part.3.B.3.decorator-iterator-generator
- Part.3.B.4.regex

- Part.3.B.5.bnf-ebnf-pebnf
- Part.3.C.breaking-good-and-bad
- Part.3.D.indispensable-illusion
- Part.3.E.to-be-thorough
- Part.3.F.social-selfteaching
- Part.3.G.the-golden-age-and-google
- Part.3.H.prevent-focus-drifting
- Q.good-communiation
- R.finale
- S.whats-next
- T-appendix.editor.vscode
- T-appendix.git-introduction
- T-appendix.jupyter-installation-and-setup
- T-appendix.symbols

Table of Contents

00.cover
01.preface
02.proof-of-work
Part.1.A.better.teachyourself
Part.1.B.why.start.from.learning.coding
Part.1.C.must.learn.sth.only.by.reading
Part.1.D.preparation.for.reading
Part.1.E.1.entrance
Part.1.E.2.values-and-their-operators
Part.1.E.3.controlflow
Part.1.E.4.functions
Part.1.E.5.strings
Part.1.E.6.containers
Part.1.E.7.files
Part.1.F.deal-with-forward-references
Part.1.G.The-Python-Tutorial-local
Part.2.A.clumsy-and-patience
Part.2.B.deliberate-practicing
Part.2.C.why-start-from-writing-functions
Part.2.D.1-args
Part.2.D.2-aargs
Part.2.D.3-lambda
Part.2.D.4-recursion
Part.2.D.5-docstrings
Part.2.D.6-modules
Part.2.D.7-tdd
Part.2.D.8-main
Part.2.E.deliberate-thinking
Part.3.A.conquering-difficulties
Part.3.B.1.classes-1
Part.3.B.2.classes-2
Part.3.B.3.decorator-iterator-generator
Part.3.B.4.regex
Part.3.B.5.bnf-ebnf-pebnf

Part.3.C.breaking-good-and-bad
Part.3.D.indispensable-illusion
Part.3.E.to-be-thorough
Part.3.F.social-selfteaching
Part.3.G.the-golden-age-and-google
Part.3.H.prevent-focus-drifting
Q.good-communiation
R.finale
S.whats-next
T-appendix.editor.vscode
T-appendix.git-introduction
T-appendix.jupyter-installation-and-setup
T-appendix.symbols