

# 笨办法学Python3（Learn Python3 The Hard Way 中文版）

---

注：本书是基于 Zed Shaw 编写的《Learn Python 3 The Hard Way》一书所做的中文译注版。该中文版由“爱学习的ai酱”（微信号：xuexii2018）友情翻译，并在翻译和学习过程中对部分练习添加了批注，以帮助大家在入门 python 的旅途中更好地避坑。该翻译内容在简书/知乎（搜索“爱学习的ai酱”）也有发布，内容仅供学习交流使用，严禁用于商业行为。

## 目录

- [笨办法学Python3（Learn Python3 The Hard Way 中文版）](#)
- [目录](#)
- [前言](#)
  - [第四版更新](#)
  - [致谢](#)
  - [笨办法更简单](#)
    - [一、读和写](#)
    - [二、注意细节](#)
    - [三、发现不同](#)
    - [四、要问，不要盯着看](#)
    - [五、不要复制粘贴](#)
    - [六、一个关于坚持练习的忠告](#)
- [练习0 配置环境](#)
  - [MacOS](#)
    - [0.1.1 MacOS: 你应该看到的](#)
  - [Windows](#)
    - [1.2.1 Windows: 你应该看到](#)
  - [Linux](#)
    - [0.3.1 Linux: 你应该看到](#)
  - [从网上找答案](#)
  - [给初学者的忠告](#)

- 其他可选编辑器
- 练习 1. 第一个程序
  - 你应该看到
  - 课后练习
  - 常见问题
- 练习 2 注释和井号
  - 你应该看到
  - 课后练习
  - 常见问题
- 练习 3 数字和数学
  - 你应该看到
  - 课后练习
  - 常见问题
- 练习 4 变量和名字
  - 你会看到
  - 附加练习
  - 更多附加练习：
  - 常见问题
- 练习 5 更多变量和打印
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 6 字符串和文本
  - 你会看到
  - 附加练习
  - 把代码打乱
  - 常见问题
- 练习 7 更多打印
  - 你会看到
  - 附加练习
  - 把代码打乱
  - 常见问题
- 练习 8 打印，打印
  - 你会看到
  - 加分练习

- 常见问题
- 练习 9 打印, 打印, 打印
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 10 那是什么?
  - 你会看到
  - 转义字符
  - 附加练习
  - 常见问题
- 练习 11 问问题
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 12 提示用户
  - 附加练习
  - 常见问题
- 练习 13 参数, 解包, 变量
  - 等等! Features 还有另一个名字
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 14 提示和传递
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 15 阅读文件
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 16. 读写文件
  - 你应该看到
  - 附加练习
  - 常见问题
- 练习 17. 更多文件

- 你会看到
  - 附加练习
  - 常见问题
- 练习 18 名称, 变量, 代码, 函数
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 19 函数和变量
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 20 函数和文件
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 21 函数可以返回一些东西
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 22 你目前为止学到了什么?
  - 你正在学的
- 练习 23 字符串, 字节和字符编码
  - 初始研究
  - 开关、惯例 (conventions) 和编码
  - 分解输出结果
  - 分解代码
  - 深入了解编码
  - 拆解
- 练习 24 更多练习
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 25 更多练习
  - 你会看到
  - 附加练习

- 常见问题
- 练习 26 恭喜你，来做个测试吧！
  - 常见问题
- 练习 27 记忆逻辑
  - The Truth Terms
  - The Truth Tables
  - 常见问题
- 练习 28 布尔练习
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 29 if 语句
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 30 Else 和 if
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 31 做决定
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 32 循环和列表
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 33 While 循环
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 34 获取列表元素
  - 附加练习
- 练习 35 分支和函数
  - 你会看到

- 附加练习
  - 常见问题
- 练习 36 设计和调试
  - if 语句的规则：
  - 循环的规则
  - 调试建议
  - 课后作业
- 练习 37. 复习各种符号
  - 关键词
  - 数据类型
  - 字符串转义序列 (Escape Sequences)
  - 老式字符串格式化
  - 运算符
  - 阅读代码
  - 附加练习
  - 常见问题
- 练习 38. 操作列表
  - 你会看到
  - 列表能做什么
  - 何时使用列表
  - 附加练习
  - 常见问题
- 练习 39. 字典，可爱的字典
  - 一个字典示例
  - 你会看到
  - 字典能做什么
  - 附加练习
  - 常见问题
- 练习 40. 模块、类和对象
  - 模块就像字典
    - 40.1.1 类就像模块
    - 40.1.2 对象就像导入 (import)
    - 40.1.3 获取数据
    - 40.1.4 第一个类的例子
  - 你会看到

- 附加练习
  - 常见问题
- 练习 41. 学着去说面向对象
  - 词汇训练
  - 短语训练
  - 综合训练
  - 一个阅读测试
  - 练习从自然语言到代码
  - 读更多代码
  - 常见问题
- 练习 42. Is-A, Has-A, 对象和类
  - 鱼和三文鱼的区别是什么？
  - Mary 和三文鱼的区别是什么？
  - 代码怎么写
  - 关于 类名(object)
  - 附加练习
  - 常见问题
- 练习 43. 面向对象的分析和设计基础
  - 一个简单的游戏引擎分析
    - 43.1.1 写或画出这个问题
    - 43.1.2 抽取关键概念并予以研究
    - 43.1.3 为这些概念创建类的层级结构和对象地图
    - 43.1.4 编写类代码并通过测试来运行
    - 43.1.5 重复和改进
  - 自上而下 vs 自下而上
  - “来自25号行星的哥顿人”游戏代码
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 44. 继承和组合
  - 什么是继承？
    - 44.1.1 隐式继承 (Implicit Inheritance)
    - 44.1.2 显式继承 (Override Explicitly)
    - 44.1.3 修改前后
    - 44.1.4 三者结合

- 用 `super()` 的理由
  - 组合
- 何时使用继承或组合
- 附加练习
- 常见问题
- 练习 45. 你来做一个游戏
  - 评估你的游戏
  - 函数的风格
  - 类的风格
  - 代码的风格
  - 好的注释
  - 评估你的游戏
- 练习 46. 一个项目骨架
  - macOS/Linux 设置
  - Windows 10 设置
  - 创建项目骨架目录
    - 46.3.1 最终目录结构
  - 测试你的 Setup
  - 使用这个骨架
  - 课后测试
  - 常见问题
- 练习 47. 自动化测试
  - 写一个测试用例 (test case)
  - 测试指南
  - 你会看到
  - 附加练习
  - 常见问题
- 练习 48. 更复杂的用户输入
  - 我们的游戏词汇表 (lexicon)
    - 48.1.1 拆解句子
    - 48.1.2 词汇表元组 (Lexicon Tuples)
    - 48.1.3 扫描输入
    - 48.1.4 异常和数字
  - 一个测试优先挑战
  - 你需要测试



- 附加练习
  - 常见问题
- 练习 49. 创建句子
  - 匹配和窥探 (Peek)
  - 句子语法
  - 关于异常
  - 解析器代码 (The Parser Code)
  - 玩一玩解析器
  - 你需要测试
  - 附加练习
  - 常见问题
- 练习 50. 你的第一个网站
  - 安装 Flask
  - 创建一个简单的 “Hello World” 项目
  - 发生了什么？
  - 修正错误
  - 创建基本的模板
  - 附加练习
  - 常见问题
- 练习 51. 从浏览器获取输入
  - Web 是如何工作的？
  - 表单 (forms) 是如何工作的
  - 创建 HTML 表单
  - 创建布局模板(layout template)
  - 为表单撰写自动测试代码
  - 附加练习
  - 拆解
- 练习 52. 创建你的 web 游戏
  - 重构《练习 43》的游戏
  - 创建一个引擎
  - 你的期末考试
  - 常见问题
- 练习 53. 接下来的步骤
  - 如何学习任何编程语言
- 练习 54. 来自老程序员的建议

- 附录 A：命令行速成教程
  - 介绍：别说话，开始用 Shell
    - 55.1.1 如何使用附录
    - 55.1.2 你需要记东西
  - 附录练习 1 环境配置
    - 55.2.1 跟我做
      - macOS
      - Linux
      - Windows
    - 55.2.2 你学到的
    - 55.2.3 附加练习
      - Linux/macOS
      - Windows
  - 附录练习2 路径，文件夹，目录 (pwd)
    - 55.3.1 跟我做
      - Linux/macOS
      - Windows
    - 55.3.2 你学到的
    - 55.3.3 附加练习
  - 附录练习3 如果你迷路了
    - 55.4.1 跟我做
    - 55.4.2 你学到的
  - 附录练习4 创建目录 (mkdir)
    - 55.5.1 跟我做
      - Linux/macOS
      - Windows
    - 55.5.2 你学到的
    - 55.5.3 附加练习
  - 附录练习 5 切换目录 (cd)
    - 55.6.1 跟我做
      - Linux/macOS
      - Windows
    - 55.6.2 你学到的
    - 55.6.3 附加练习
  - 附录练习 6 列示目录 (ls)

- 55.7.1 跟我做
  - [Linux/macOS](#)
  - [Windows](#)
- 55.7.2 你学到的
- 55.7.3 附加练习
- 附录练习 7 移除目录 (rmdir)
  - 55.8.1 跟我做
    - [Linux/macOS](#)
    - [Windows](#)
  - 55.8.2 你学到的
  - 55.8.3 附加练习
- 附录练习 8 来回移动 (pushd, popd)
  - 55.9.1 跟我做
    - [Linux/macOS](#)
    - [Windows](#)
  - 55.9.2 你学到的
  - 55.9.3 附加练习
- 附录练习 9 创建空文件 (Touch, New-Item)
  - 55.10.1 跟我做
    - [Linux/macOS](#)
    - [Windows](#)
  - 55.10.2 你学到的
  - 55.10.3 附加练习
- 附录练习 10 复制文件 (cp)
  - 55.11.1 跟我做
    - [Linux/macOS](#)
    - [Windows](#)
  - 55.11.2 你学到的
  - 5.11.3 附加练习
- 附录练习 11 移动文件 (mv)
  - 55.12.1 跟我做
    - [Linux/macOS](#)
    - [Windows](#)
  - 55.12.2 你学到的
  - 55.12.3 附加练习

- [附录练习 12 浏览文件 \(less, MORE\)](#)
  - [55.13.1 跟我做](#)
    - [Linux/macOS](#)
    - [Windows](#)
  - [55.13.2 你学到的](#)
  - [55.13.3 附加练习](#)
- [附录练习 13 Stream 文件 \(cat\)](#)
  - [55.14.1 跟我做](#)
    - [Linux/macOS](#)
    - [Windows](#)
  - [55.14.2 你学到的](#)
  - [55.14.3 附加练习](#)
- [附录练习 14 移除文件 \(rm\)](#)
  - [55.15.1 跟我做](#)
    - [Linux](#)
    - [Windows](#)
  - [55.15.2 你学到的](#)
  - [55.15.3 附加练习](#)
- [附录练习 15 退出 Terminal \(exit\)](#)
  - [55.16.1 跟我做](#)
    - [Linux/macOS](#)
    - [Windows](#)
  - [55.16.2 你学到的](#)
  - [55.16.3 附加练习](#)
- [命令行后续](#)
  - [55.17.1 Unix Bash References](#)
  - [55.17.2 PowerShell References](#)

# 前言

这本简单的小书是为了让你开始编程。虽然书名说是“笨办法”，但其实不然。所谓“笨办法”只是本书教授的方法，也就是按照我的要求重复做一系列的练习来构建你的技能。这种方法对于零基础想要掌握基本编程技能的人来说非常有效，它几乎被用于所有的学习，从武术、音乐，到基础数学和阅读技巧。

这本书指导你通过练习和记忆逐渐建立起 Python 的使用技巧，然后用在更复杂的问题上。学完本书，你将会拥有开始学习更复杂编程所需的工具。我很喜欢告诉别人，我的书可以让你拥有“编程黑带”，也就是你已经掌握要开始学习编程的最基本的知识。

如果你肯努力，肯花时间来建立起这些技巧，你将可以正式学习编程。

## 第四版更新

《笨办法学 Python》第四版用了 Python 3.6。Python 3.6 升级了字符串格式系统，相比于之前的 4 或者 4 更好用，虽然对于初学者来说，学习 Python 3.6 会有很多问题，一个很明显的问题就是它的报错信息非常少，但是我将会帮助你理解从而解决这些问题。

同时，我也根据我过去五年来教授 Python 的经验，更新了视频课程。过去的视频只是简单地让你看着我做练习，而第四版加入了打乱再重新修复的练习，这个技巧叫做“调试”（debugging），它将会叫你如何修复你遇到的问题以及 Python 是如何运行你创建的程序。这个新方法的目标就是建立一种关于 Python 如何运行代码的思维模式，从而能够更容易看出来哪里出了问题。此外，你还会学到很多有用的调试错误程序的有用技巧。

最后，第四版从头到尾完全支持 Windows 10，以前版本更多地专注于基于 Unix 的系统比如 MacOS 和 Linux。而当我开始写第四版的时候微软已经开始认真对待开源工具和开发者，因为作为一个严肃的 Python 开发平台，真的很难忽略他们。视频教程将着重讲解 Windows 系统下 Python 的使用，当然也会展示 MacOS 和 Linux 系统下的操作。我将会告诉你每个平台的安装教程以及其他相关的技巧。

## 致谢

我想感谢 Angela 在这本书的前两版中对我的帮助，没有她我可能很难完成。她做了第一版的复制编辑工作，并且在我写作的过程中给我提供了极大的支持。

我同样要感谢 Greg Newman 为我设计封面，Brian Shumate 所做的网站设计，以及所有读过这本书并花时间给我反馈和更正意见的读者。

谢谢你们。

# 笨办法更简单

在这本书的帮助下，你将会做所有程序员学习一门编程语言都会做的非常简单的事情：

1. 做好每一个练习；
2. 准确敲好每一个程序；
3. 让它运行。

就是这样。刚开始可能会比较难，但坚持下去。如果你通读了这本书，每晚花个一两小时做做习题，你将能够为自己读下一本编程书打下良好的基础。这本书不会让你一夜之间变成程序员，但是它将会带你走上学习如何编程的道路。

这本书的目的是教会你作为编程新手所需的三种最重要的技能：**读和写、注重细节、发现不同**。

## 一、读和写

如果你连打字都不行，那你学习编程也会成问题。尤其如果你连程序源代码中的那些奇怪字符都打不出来，就别提编程了。没有这些基本技能，你将连最基本的软件工作原理都难以学会。

所以，把代码示例打出来并运行，能够帮助你学习各种符号的名称、更熟练地敲出来、以及读懂编程语言。

## 二、注意细节

区分好程序员和差程序员的一个重要标准，就是对细节的注重程度，事实上，这也是任何行业区分好坏的标准。如果缺乏对工作中每个微小细节的注意，你的工作成果将缺乏重要的元素。拿编程来讲，主意细节将会让你远离各种bug和难用的系统。

通过这本书的学习，以及准确打出每一个例子，你将能够训练你的大脑，在做练习的时候更多地关注细节。

## 三、发现不同

程序员长年累月的工作会培养出一个重要技能，那就是对于不同点的区分能力。一个有经验的程序员看到两个仅有细微差别的程序，可以立即指出其中的不同。程序员还造出工具来让这件事更加容易，不过我们不会用到这些工具。你要先用笨办法训练自己，然后再用工具。

在你做这些练习并敲代码的时候，你一定会出错。这是不可避免的，即使有经验的程序员也会偶

尔写错。你的任务是把自己写的东西和要求的正确答案对比，把所有的不同点都修正过来。这样做可以让你对程序里的错误、bug 以及其他问题更加敏感。

## 四、要问，不要盯着看

你只要写代码，就会出现 bug。Bug 意味着你写的代码有瑕疵、有错误、或者有问题。Bug 源于一个传说，从前有一只飞蛾飞进了第一台计算机，造成了故障。修复它就需要“debugging”。在软件世界里，有着不计其数的 bug。

就像第一只飞蛾，你的 bugs 将会藏在你代码的某处，你必须找到它们。你不能只是坐在电脑前盯着屏幕上的代码，希望答案能自己跳出来。这样做不会有额外的信息，你需要额外的信息来解决问题，所以你得起来寻找这只飞蛾。

怎么寻找呢？你需要审问你的代码，问它现在是怎么回事儿，或者从另一个不同的视角去看待这个问题。在这本书里，我将会频繁地告诉你“别盯着看，要问”。我将会向你演示如何让你的代码告诉你正在发生的一切，并且如何找到可能的解决方案。我还会教你一些从不同角度看代码的方法，让你能够获取更多信息和洞见。

## 五、不要复制粘贴

你必须手动将每个练习打出来。复制粘贴会让这些练习变得毫无意义。这些习题的目的是训练你的双手和大脑思维，让你有能力读代码、写代码、观察代码。如果你复制粘贴，那你就是在欺骗自己，这些练习的效果也将大打折扣。

## 六、一个关于坚持练习的忠告

在你通过这本书学习编程时，我正在学习弹吉他。我每天至少练习 2 个小时，至少花一个小时练习音阶、和声、和弦，剩下的时间用来学习音乐理论和歌曲演奏以及训练听力等。有时我一天会花 8 个小时来练习，因为我觉得这是一件有趣的事情。对我来说，重复性练习是学好一样东西最自然而然的方法。并且我深知，要掌握一件事情，只有每天坚持练习。虽然有时候，我整个人状态很差（甚至经常这样），或者觉得实在太难。没关系，坚持尝试，到最后你会发现它越来越简单，并且开始越来越有趣。

在我写《笨办法学 Python》和《笨办法学 Ruby》的过程中，我发现了绘画的乐趣。我在自己 39 岁的时候爱上了这门视觉艺术，并且像学习吉他、音乐和编程一样每天花时间学习画画。我收集了相关的教材，并且按照书中所说，每天坚持画，同时专注于享受这种学习过程的乐趣。我完全不是一个艺术家，甚至差得很远，但我现在至少可以说我会画画了。我学习画画的方法就跟

我在这本书里教你的一样。如果你把整个问题分解为一个个小练习和课程，并且每天做，你就可以学会几乎所有的东西。如果你专注于细微的进步，并且享受学习过程，你将会从中获益，无论你最后擅长到何种程度。

当你通过这本书学习编程的时候，要记住任何值得做的事情一开始都是困难的。也许你是一个害怕失败的人，一碰到困难就想放弃；也许你是一个缺乏自律的人，一碰到“无聊”的事情就不想上手；也许因为有人夸你“天赋异禀”而让你自视甚高，不愿意做这些看上去很笨拙的事情，怕有负你“神童”的称号；也许你太过激进，把自己跟有 20 多年经验的编程老手相比，让自己失去了信心。

无论是什么原因让你想要放弃，你一定要坚持下去。如果你碰到做不出来的课后练习，或者碰到一节看不懂的练习，你可以暂时跳过去，过一阵子回来再看。只要坚持下去，你总会弄懂的，因为编程的过程中总是会出现这样的问题。

一开始你可能什么都看不懂。这会让你感觉很不舒服，就像学习人类的自然语言一样。你会发现很难记住一些单词和特殊符号的用法，而且会经常感到很困惑。但是突然有一天，你一下子变得豁然开朗，以前不明白的东西忽然就明白了。如果你坚持练习下去，坚持去上下求索，你最终会学会这些东西。你可能不会成为一位编程大师，但你至少会明白程序是怎么运行的。

如果你放弃的话，你将永远达不到那种“豁然开朗”的时刻。你会在第一次碰到不明白的东西时（一开始就是所有东西）就选择放弃。如果你坚持尝试，坚持练习下去，坚持去弄懂习题的话，你最终一定会明白其中的内容。

如果你通读了这本书，却还是不知道怎么编程，那也没关系，至少你试过了。你可以说你已经尽过力但成效不佳，但至少你试过了。这也是一件值得你骄傲的事情。

## 练习0 配置环境

这个练习没有代码。它只是为了让你的计算机准备好运行 Python。你应该严格按照步骤来。（如果配置过程中遇到问题，可以在公众号“学习癌”后台留言，小编会为你答疑解惑。）

### 警告！

如果你不知道如何使用 Windows 上的 Powershell，MacOS 上的 Terminal,或者 Linux 上的 bash，你需要在进行下面的学习之前先做一下附录 A 中的练习。



# MacOS

做如下任务以完成练习:

1. 去 <https://www.python.org/downloads/release/python-360/> 上下载 “macOS 64-bit/ 32-bit installer”，像安装其他软件一样安装它。
2. 去 <https://atom.io/> 下载 Atom 文本编辑器，然后安装。如果你用不惯 Atom，可以在这个练习的最后选择其他可用的编辑器。
3. 把 Atom 放在 Dock 中，以便快速打开。
4. 用苹果电脑搜索功能找到你的 Terminal 程序，找不到的话就想想办法，你可以的。
5. 把 Terminal 也放在 Dock.
6. 运行 Terminal 程序，它看起来不咋滴。
7. 在 Terminal 中运行 python3.6。在 Terminal 中运行程序只需要输入程序名然后敲 `Return` 即可。
8. 输入 `quit()`，`Enter`，然后退出 `python3.6`。
9. 你应该回到你输入 `python` 之前看到的提示界面，如果不是，弄明白是什么原因。
10. 学会如何在 Terminal 中创建目录。
11. 学会如何在 Terminal 中切换目录。
12. 用你的编辑器在这个目录下创建一个文件，你可以先在编辑器里编辑，然后点击“保存”或者“另存为”，选择你创建的目录文件夹。
13. 用键盘快捷键切换回 Terminal 程序。
14. 回到 Terminal 后，用 `ls` 列示目录以查看你新建的文件。

## 0.1.1 MacOS: 你应该看到的

这是我在自己苹果电脑 Terminal 终端操作完的会话。你的可能会稍有不同，但大体应该是差不多的。

```
$ python3.6
Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang -700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information
>>>
~ $ mkdir lpthw
~ $ cd lpthw
lpthw $ ls
# ... Use your text editor here to edit test.txt....
lpthw $ ls
test.txt lpthw $
```

## Windows

1. 用浏览器访问 <https://atom.io>，下载 Atom 并安装，你可能需要用管理员身份运行。
2. 把 Atom 放在桌面或者快速启动栏以便快速访问，这些都可以在安装的时候进行设置。  
注：如果你的电脑运行很慢，打不开 Atom，你可以在本练习最后选择其他编辑器。
3. 在开始菜单搜索 Powershell，回车，运行。
4. 在桌面创建 Powershell 快捷键，或者把它添加到快速启动栏以方便打开。
5. 运行 Powershell（我之后会称它为 Terminal），它看起来不咋滴。
6. 从 <https://www.python.org/downloads/release/python-360/> 下载 Python 3.6 然后安装。记得勾选“把 Python 3.6 添加到路径”（add Python 3.6 to your path）复选框。
7. 在 Powershell（Terminal）中输入 `python` 并回车，以运行 Python。  
注：如果你输入 `Python` 但它没有运行，你需要重新安装 Python 并确保在安装过程中勾选了“把 Python 3.6 添加到路径”（add Python 3.6 to your path）复选框。
8. 输入 `quit()` 以退出 Python。
9. 你应该回到你输入 `python` 之前看到的提示界面，如果不是，弄明白是什么原因。
10. 学会如何在 Powershell 中创建目录。
11. 学会如何在 Powershell 中切换目录。
12. 用你的编辑器在这个目录下创建一个文件，你可以先在编辑器里编辑，然后点击“保存”或者“另存为”，选择你创建的目录文件夹。
13. 用键盘快捷键切换回 Powershell 程序。
14. 回到 Powershell 后，用 `ls` 列示目录以查看你新建的文件。

从现在起，当我说“Terminal”或“Shell”时指的就是 Powershell。当我让你运行 Python 3.6 的时候你只用输入 `python` 即可。

## 1.2.1 Windows: 你应该看到

```
> python
>>> quit()
> mkdir lpthw
> cd lpthw
... Here you would use your text editor to make test.txt in lpthw
>
> dir
Volume in drive C is
Volume Serial Number is 085C-7E02

Directory of C:\Documents and Settings\you\lpthw

04.05.2010      23:32      <DIR>          .
04.05.2010      23:32      <DIR>          ..
04.05.201 0      23:32                  6      test.txt
                1      File(s)          6 bytes
                2      Dir(s)          14 804 623 360 bytes free

>
```

如果你的显示跟我的略有不同也是正确的，但是大体上应该是一样的。

## Linux

Linux 系统五花八门，软件安装方式也各不相同。我假设如果你用的是 Linux，你是知道如何安装软件包的，下面是你的操作步骤：

1. 用你的安装包管理器（package manager）安装 Python 3.6，如果无法安装，就从 <https://www.python.org/downloads/release/python-360/> 上下载并安装。
2. 用你的安装包管理器安装 Atom 编辑器。如果 Atom 不好用，你可以选择本练习最后的其他编辑器。
3. 把 Atom 放到你的窗口管理（window manager）菜单，以方便快速访问。
4. 找到你的 Terminal 程序，它可能叫 GNOME Terminal、Konsole、或者 xterm。
5. 把你的 Terminal 也放在 Dock。
6. 运行你的 Terminal 程序，它看起来不咋滴。
7. 在你的 Terminal 程序中输入 `python3.6` 以运行 Python 3.6。如果无法运行，试试输入 `python`。
8. 输入 `quit()` 然后敲 `enter` 退出 Python。

9. 你应该回到你输入 `python` 之前看到的提示界面，如果不是，弄明白是什么原因。
10. 学会如何在 Terminal 中创建目录。
11. 学会如何在 Terminal 中切换目录。
12. 用你的编辑器在这个目录下创建一个文件，你可以先在编辑器里编辑，然后点击“保存”或者“另存为”，选择你创建的目录文件夹。
13. 用键盘快捷键切换回 Terminal 程序。
14. 回到 Terminal 后，用 `ls` 列示目录以查看你新建的文件。

## 0.3.1 Linux: 你应该看到

```
$ python
>>> quit()
$ mkdir lpthw
$ cd lpthw
# ...      Use your text editor here to edit test.txt ...
$ ls test.txt
$
```

如果你的显示跟我的略有不同也是正确的，但是大体上应该是一样的。

## 从网上找答案

这本书中很重要的一部分就是要学会从网上搜索编程相关的东西。我会告诉你“从网上搜索”，你需要做的就是用搜索引擎找到答案。我之所以不直接告诉你答案而是让你自己去找，就是为了让成为你成为一个独立的学习者，能够自己从网上找到答案而不依赖于书本，这是我的目标。

## 给初学者的忠告

这个练习已经结束了，它的难易程度可能取决于你对你电脑的熟悉程度。如果你觉得很难，试着花时间去学习和克服困难，因为只有攻克了这些最基础的东西，你才能继续学习更多的编程技能。

如果有人告诉你学到这本书的某个练习就可以停下来，或者跳过某些练习，别去相信。任何对你隐瞒知识的人，或者更糟糕——让你从他们那里获得知识而不是通过你自己的努力去获取，都是在让你对他们形成依赖。别听他们的，老老实实地做这些练习，从掌握自我学习的本领。

程序员可能会让你用 MacOS 或者 Linux，因为他们很喜欢苹果电脑的字体和排版设计，或者觉

得用 Linux 很酷。别听他们的，用你现在正在用的电脑系统就行，你需要的就是一个编辑器，一个终端，还有 Python。

最后，这个练习的目的就是让你准备好这三样东西，以便做后面的练习：

- 1. 用文本编辑器写练习。
- 2. 运行你写的练习。
- 3. 如果出错就试着修复。
- 4. 重复。

其他事情可能会烦扰到你，所以坚持按照以上计划来。

## 其他可选编辑器

文本编辑器对程序员来说非常重要，但是作为初学者，你只需要一个简单的编辑器即可。我推荐 Atom 是因为它是免费的，而且几乎在任何系统上都能运行。但是，Atom 可能不适合你的电脑，所以你也可以选择以下这些编辑器：

编辑器名称	适用系统	网址
Visual Studio Code	Windows, MacOS, Linux	<a href="https://code.visualstudio.com/">https://code.visualstudio.com/</a>
Notepad++	Windows	<a href="https://notepad-plus-plus.org/">https://notepad-plus-plus.org/</a>
gEdit	Linux, MacOS, Windows	<a href="https://github.com/GNOME/gedit">https://github.com/GNOME/gedit</a>
Textmate	MacOS	<a href="https://github.com/textmate/textmate">https://github.com/textmate/textmate</a>
SciTE	Windows, Linux	<a href="http://www.scintilla.org/SciTE.html">http://www.scintilla.org/SciTE.html</a>
jEdit	Linux, MacOS, Windows	<a href="http://www.jedit.org/">http://www.jedit.org/</a>

以上这些编辑器是根据好用程度排序的。由于这些项目可能被放弃、死掉、或者不再适用于你的电脑。所以如果你试了一个不行，就试试别的。而且以上排序是基于我自己的电脑，对于你的电脑来说情况可能不太一样。

如果你已经知道如何使用 Vim 或者 Emacs，那就放心用。如果你从来没用过，就不要考虑了。程序员可能会努力说服你用 Vim 或者 Emacs，但那只会让你误入歧途。你的目标是学习 Python，而不是学习 Vim 或者 Emacs。如果你尝试使用 Vim 但却不知道如何退出，就输入

:q! 或者 ZZ。

## 练习 1. 第一个程序

### 警告！

如果你跳过了练习 0，那你就没有按照这本书的正确学习方式。另外，也不要使用 IDLE 或者 IDE。如果你跳过了练习 0，拐回去重新学习。

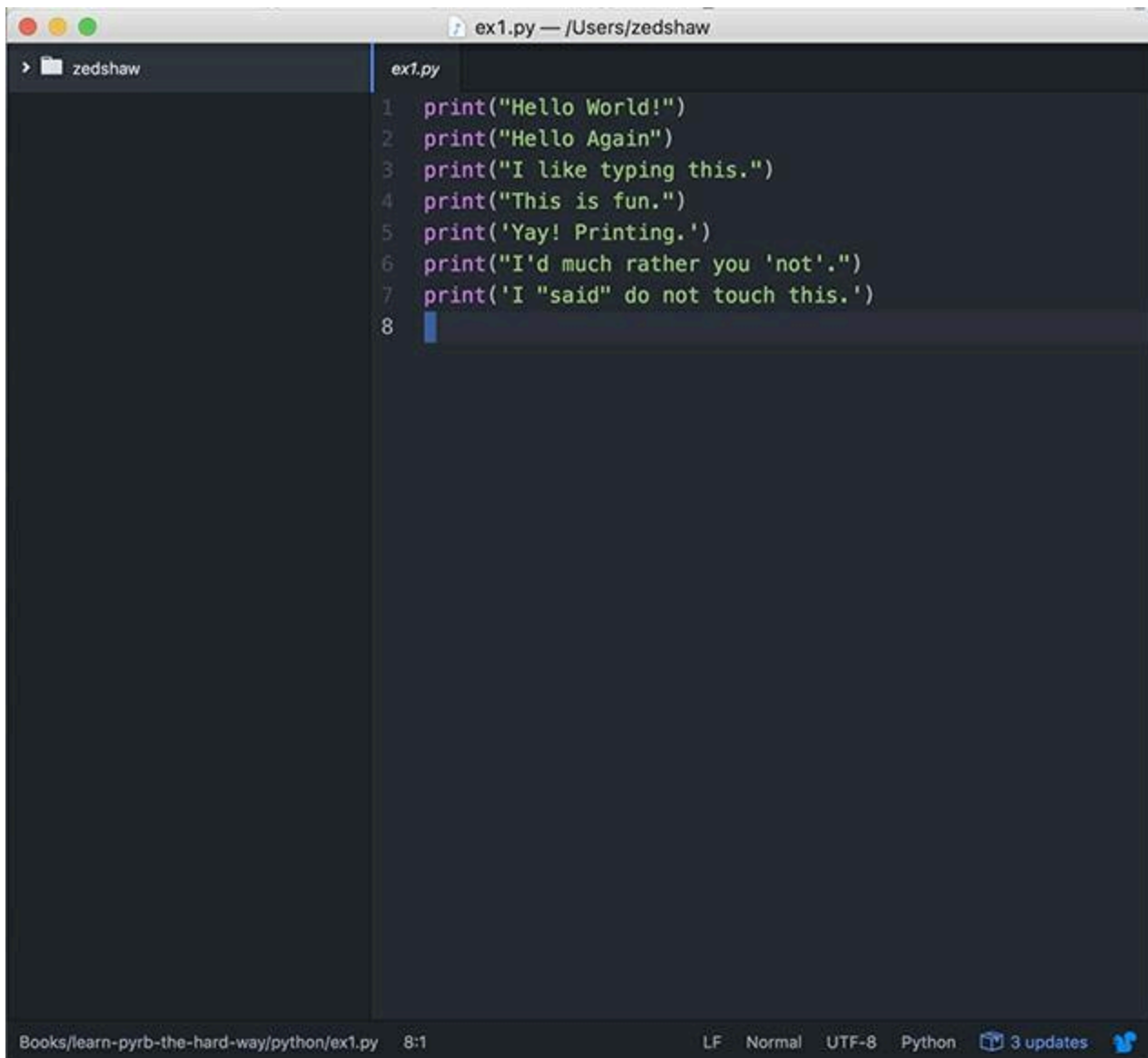
你可能已经花费了大量时间在练习 0 上，学习如何安装一个文本编辑器，运行编辑器以及 Terminal，并且学习如何操作它们。如果你还没有做这些，就不要再继续，否则你会后悔的。这种练习前的警告我再说最后一次：**不要擅自跳过练习自己往前学。**

把下面的本文输入到一个名为 `ex1.py` 单个文件中，python 文件一般以 `.py` 为后缀。

`ex1.py`

```
1     print("Hello World!")
2     print("Hello Again")
3     print("I like typing this.")
4     print("This is fun.")
5     print('Yay! Printing.')
6     print("I'd much rather you 'not'.")
7     print('I "said" do not touch this.')
```

你的 Atom 文本编辑器应该看起来像这样：



```
ex1.py — /Users/zedshaw
> zedshaw
ex1.py
1 print("Hello World!")
2 print("Hello Again")
3 print("I like typing this.")
4 print("This is fun.")
5 print('Yay! Printing.')
6 print("I'd much rather you 'not'.")
7 print('I "said" do not touch this.')
8
Books/learn-pyrb-the-hard-way/python/ex1.py 8:1
LF Normal UTF-8 Python 3 updates
```

如果你的编辑器看起来不完全一样也不用担心，只要大体一致就行。

当你敲这个文件时，你得明白：

1. 左边这些数字不是我敲上去的，它印在书上是为了讲解的时候方便说明。比如“看第 5 行...”之类，所以不要把行数敲进 Python 脚本里。
2. 我输出的结果跟我在练习 1 的 `ex1.py` 中让你们输入的内容是一模一样的，而不是大致相同。所以每一个字符你都要严格按照我的要求来输入，当然颜色无所谓。

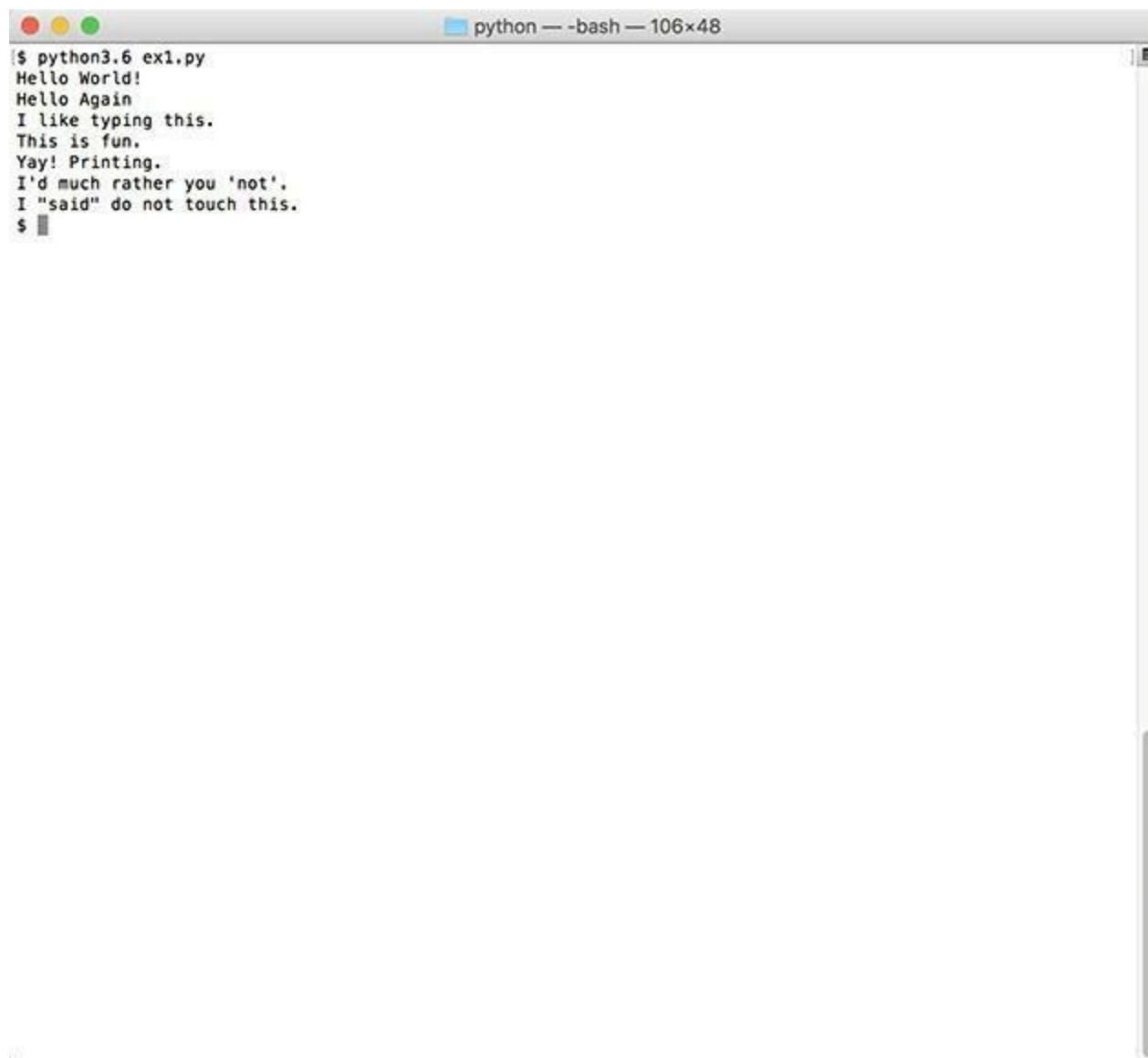
在终端这样输入就可以运行你的 Python 文件：

```
python3.6 ex1.py
```

如果你都做对了，你就会看到和我一样的结果，如果没有，你肯定是哪里出错了。不，相信我，不是电脑的问题。

## 你应该看到

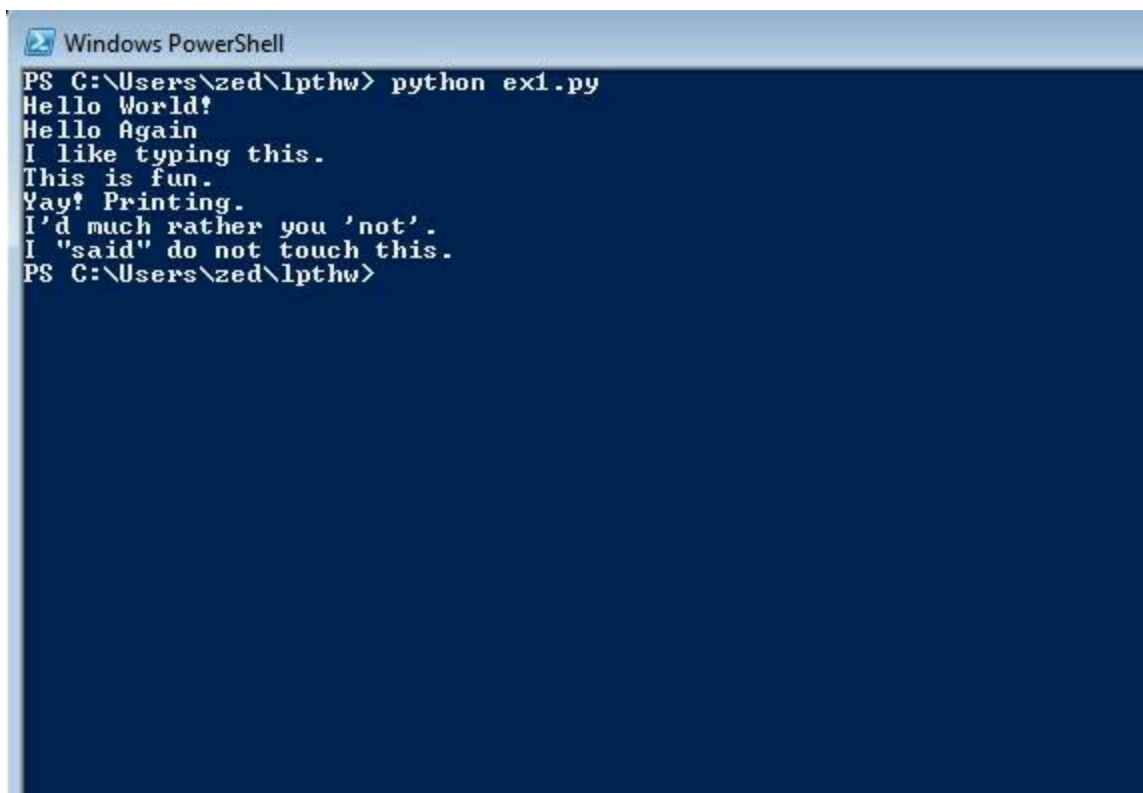
在 MacOS 的 Terminal 上你应该看到这样的结果：

A screenshot of a macOS Terminal window. The title bar at the top shows three colored window control buttons (red, yellow, green) on the left, and a title bar with a blue icon, the text "python — -bash — 106x48", and a close button on the right. The terminal content shows a command prompt "\$" followed by the command "python3.6 ex1.py". Below the command, the output of the script is displayed line by line: "Hello World!", "Hello Again", "I like typing this.", "This is fun.", "Yay! Printing.", "I'd much rather you 'not'.", and "I \"said\" do not touch this.". The prompt "\$" is followed by a cursor. A vertical scrollbar is visible on the right side of the terminal window.

```
$ python3.6 ex1.py
Hello World!
Hello Again
I like typing this.
This is fun.
Yay! Printing.
I'd much rather you 'not'.
I "said" do not touch this.
$
```

在 Windows 的 Powershell 上你应该看到这样的结果：



A screenshot of a Windows PowerShell terminal window. The title bar says "Windows PowerShell". The prompt is "PS C:\Users\zed\lpthw>". The user has entered "python ex1.py". The output of the script is displayed: "Hello World!", "Hello Again", "I like typing this.", "This is fun.", "Yay! Printing.", "I'd much rather you 'not'.", "I 'said' do not touch this.", and the prompt "PS C:\Users\zed\lpthw>".

```
PS C:\Users\zed\lpthw> python ex1.py
Hello World!
Hello Again
I like typing this.
This is fun.
Yay! Printing.
I'd much rather you 'not'.
I 'said' do not touch this.
PS C:\Users\zed\lpthw>
```

你可能会在 python3.6 命令行前面看到不同的名字，这不重要，重要的是你输入的命令要输出跟我一样的结果。

如果你出错了你可能会看到这个：

```
$ python3.6 python/ex1.py
File "python/ex1.py", line 3
print (" I like typing this.
                                ^
SyntaxError : EOL while scanning string literal
```

能读懂这些错误信息很重要，因为你接下来可能会出现很多这种错误，我也是。让我们一行一行来看。

1. 我们在 Terminal 运行 `ex1.py` 脚本。
2. Python 告诉我们 `ex1.py` 文件的第三行出错了。
3. 它把第三行的代码打印出来以便我们查看。
4. 然后它放了一个 `^`（插入符号）来指出哪里出了问题。注意到那个缺失的 `"` 了吗？
5. 最后，它输出“SyntaxError”，并告诉我们可能的错误信息。通常这些错误信息都比较隐晦，但是如果你用搜索引擎搜索以下，你就会发现别人也遇到过这样的错误，

然后你很可能就会找到问题的解决办法。

## 课后练习

这个课后练习包括一些你应该尝试去做的东西，如果你不会，可以先跳过，随后再拐回来做。

对于这个练习，试试以下操作：

1. 让你的脚本再打印出一行。
2. 让你的脚本只打印一行。
3. 在任一行的开头放一个 `#`，看看会发生什么？试着弄明白这个符号的作用。

从现在开始，我不会解释每个练习都是如何工作的，除非遇到特殊情况。

## 常见问题

以下是实际学习本节练习过程中学生们经常会问到的一些问题：

**我能用 IDLE 吗？** 不，你应该用 Terminal 或者 Powershell。如果你不知道怎么用，就去学习附录 A 的命令行速成教程。

**我怎样才能像你一样编辑代码时有颜色？** 先把你的文件另存为 `.py`，比如 `ex1.py`。然后你在输入的时候就会有颜色了。

**我在运行 `ex1.py` 时遇到了 `SyntaxError: invalid syntax` 提示。** 你很可能是已经运行了 Python，然后又输了一次 `python`。关闭 Terminal，重新打开，然后只输入 `python3.6 ex1.py`。

**我遇到了 `can't open 'ex1.py': [Errno 2] No such file or directory`。（“无法打开‘`ex1.py`’：[错误号 2] 没有该文件或目录”）。** 你需要和你创建的文件在同一个目录（文件夹）下。你要先使用 `cd` 命令切换到了那个目录下。比如，如果你把你的文件保存在 `lpthw/ex1.py`，那你就应该在运行 `python3.6 ex1.py` 之前先用 `cd lpthw/` 切换到 `lpthw/` 目录下。如果这段你看不懂，去学附录 A 的命令行速成教程。

**我的文件不运行，我只是返回了提示符，没有任何输出。** 你很可能以为我让你输入 `print("Hello World!")` 只是让你输 `"Hello World!"`。并不是，你要完整地、一字不差地把代码敲出来。

# 练习 2 注释和井号

注释在程序中非常重要，它们可以用自然语言告诉你某段代码的功能是什么，还能在你想要暂时移除某段代码时禁用程序的一部分。以下是如何在 Python 中使用注释：

ex2.py

```
1      # A comment, this is so you can read your program later.
2      # Anything after the # is ignored by python.
3
4      print("I could have code like this.") # and the comment after 5
6      # You can also use a comment to "disable" or comment out code
7      # print("This won't run.")
8
9      print("This will run.")
```

从现在开始，我都会这样写代码。你得明白不是所有的东西都得在字面上保持一致，你的屏幕和程序可能看起来跟我的不一样，但是只要你在编辑器里输入的文本一样就行。事实上，我用任何文本编辑器都可以输出同样的结果。

## 你应该看到

练习 2 会话

```
$ python3.6 ex2.py
I could have code like this. This will run.
```

再说一次，我之后可能不会将所有的截图都贴出来，你得明白第一个 `$` 和最后一个 `$` 之间的内容才是你应该关注的。

## 课后练习

1. 弄清楚 `#` 符号的作用，而且记住它的名字。(中文为井号，英文为 octothorpe 或者 pound character)。
2. 打开你的 `ex2.py` 文件，从后往前逐行检查每个单词，与要求输入的内容进行对比。

3. 有没有发现什么错误？有的话就改正过来。
4. 读你写的习题，把每个字符都读出来。有没有发现更多错误？有的话改正过来。

## 常见问题

**为什么在 `print("Hi # there.")` 里 `#` 就没有被忽略？** 因为 `#` 在一个字符串里，计算机会打印引号之间字符串的所有内容，`#` 在字符串里被认为是一个字符，而不是注释符号。

**我如何把很多行变成注释？** 在每一行前面加一个 `#`。

**为什么我要倒着检查代码？** 这是一个让你的大脑不专注于每行代码意思的小技巧，这样做能够让你更准确地检查出错误，可以说是一个很好用的纠错技巧了。

## 练习 3 数字和数学

每一种编程语言都得和数字、数学打交道。不用担心：程序员总是自诩为数学天才，其实事实并非如此。如果他们是数学天才，他们就会去研究数学，而不是去写那些 bug 连篇的网站框架以便能开上豪车。

这个练习包含了很多数学符号。让我们看看它们的名字，在你输入的时候，试着说出名字，直到你烂熟于心为止。以下是这些符号的名字：

- `+` plus, 加号
- `-` minus, 减号
- `/` slash, 斜杠
- `*` asterisk, 星号
- `%` percent, 百分号
- `<` less-than, 小于号
- `>` greater-than, 大于号
- `<=` less-than-equal, 小于等于号
- `>=` greater-than-equal, 大于等于号

有没有注意到以上只是些符号，没有运算操作呢？写完下面的练习代码后，再回到上面的列表，弄明白每个符号的作用。例如 `+` 是用来做加法运算的。

[ex3.py](#)

```
1     print("I will now count my chickens:")
2
3     print("Hens", 25 + 30 / 6)
4     print("Roosters", 100 - 25 * 3 % 4)
5
6     print("Now I will count the eggs:")
7
8     print(3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6)
9
10    print("Is it true that 3 + 2 < 5 - 7?")
11
12    print(3 + 2 < 5 - 7)
13
14    print("What is 3 + 2?", 3 + 2)
15    print("What is 5 - 7?", 5 - 7)
16
17    print("Oh, that's why it's False.")
18
19    print("How about some more.")
20
21    print("Is it greater?", 5 > -2)
22    print("Is it greater or equal?", 5 >= -2)
23    print("Is it less or equal?", 5 <= -2)
```

确保你在运行它之前准确输入了每一行代码，和我的要求做一下对比检查。

## 你应该看到

### 练习 3 会话

```
$ python3.6 ex3.py
I will now count my chickens: Hens 30.0
Roosters 97
Now I will count the eggs: 6.75
Is it true that 3 + 2 < 5 - 7? False
What is 3 + 2? 5
What is 5 - 7? -2
Oh, that's why it's False. How about some more.
Is it greater? True
Is it greater or equal? True

Is it less or equal? False
```

## 课后练习

1. 在每一行上面，用 `#` 写一句注释，向自己解释这行代码的作用。
2. 还记得你在练习 0 中是如何启动 Python 3.6 的吗？再次启动它，把 Python 当成一个计算器来做一些数学运算。
3. 找一些你需要计算的东西，然后写一个新的 `.py` 文件。
4. 用浮点数重新写一下 `ex3.py`，让它更精确一些，比如 20.0 就是一个浮点数。

## 常见问题

**为什么 `%` 是一个模数，而不是百分比？** 这很可能只是设计者们选用的一个符号。在正常情况下你可以把它读作百分号，但是，在编程中 `%` 只是一个符号。

**`%` 是如何工作的？** 可以这样讲，x 除以 y 余 J。比如 100 除以 16 余 4，`%` 求的就是余数 J。

**运算顺序是怎样的？** 在美国我们遵循 PEMDAS 规则，即“括号，指数，乘，除，加，减（Parentheses Exponents Multiplication Division Addition Subtraction）”。Python 也遵循这样的规则。很多人对 PEMDAS 规则存在误解，认为它们是严格按照先后次序来的，其实并不是，乘除是同时的，加减也是同时的，所以这个规则可能写成 `PE(M&D)(A&S)` 更合适。

## 练习 4 变量和名字

现在你已经能用 `print` 打印东西了，也能用 Python 做数学计算了。接下来我们要学习变量。在编程中，变量就是给某个东西起的名字，就像“写这本书的人”名叫“Zed”一样。程序员用这些变量让代码读起来更像自然语言。如果他们不给软件里面的东西起名字，当他们再次阅读他们写的代码时就会毫无头绪。

如果你在做这个练习的时候卡住了，记得我交给你的技巧，寻找不同点，并专注细节：

1. 在每行代码上面写上注释，跟自己解释这行代码的作用。
2. 倒着读你的 `.py` 文件。
3. 大声把你的 `.py` 文件读出来，字符也要读。

[ex4.py](#)

```
1 cars = 100
2 space_in_a_car = 4.0
3 drivers = 30
4 passengers = 90
5 cars_not_driven = cars - drivers
6 cars_driven = drivers
7 carpool_capacity = cars_driven * space_in_a_car
8 average_passengers_per_car = passengers / cars_driven
9
10
11 print("There are", cars, "cars available.")
12 print("There are only", drivers, "drivers available.")
13 print("There will be", cars_not_driven, "empty cars today.")
14 print("We can transport", carpool_capacity, "people today.")
15 print("We have", passengers, "to carpool today.")
16 print("We need to put about", average_passengers_per_car,
17       "in each car.")
```

### 警告！

`space_in_a_car` 中的 `_`

是下划线，我们会在以后的练习中经常用它来代替变量名之间的空格。

## 你会看到

### 练习 4 会话

```
$ python3.6 ex4.py
There are 100 cars available.
There are only 30 drivers available.
There will be 70 empty cars today.
We can transport 120.0 people today.
We have 90 to carpool today.
We need to put about 3.0 in each car.
```

## 附加练习

当我第一次写这个程序的时候我出了一个小错误，Python 是这样告诉我的：

```
Traceback ( most recent call last ):
  File "ex4.py" , line 8 , in <module>
    average_passengers_per_car = car_pool_capacity / passenger
NameError : name ' car_pool_capacity ' is not defined
```

用你自己的话解释这段错误信息，要用行号并解释清楚为什么。

## 更多附加练习：

1. 我给 `space_in_a_car` 赋予了 4.0 而不是 4，小数部分有必要加吗？如果只写 4 会怎么样？
2. 记住，4.0 是一个浮点数，浮点数就是有小数点的数字，要得到一个浮点数，你就得写成 `4.0` 而不是 `4`。
3. 给每一个变量写一些注释。
4. 确定你知道 `=` 就是给一个变量名（比如 `cars_driven`，`passengers`）赋一个值（可以是数字、字符串等等）。
5. 记住 `_` 是个下划线。
6. 像之前的练习一样把 Python3.6 当做一个计算器来运行，然后用变量名来做运算，比如用得比较多的 `i`、`x`、`j` 等。

## 常见问题

**`=` 和 `==` 有什么区别？** `=` 把右边的值赋给左边的变量。`==` 用来检测左右两边的东西是不是有同样的值。你会在练习 27 中学到这块内容。

**我们能把 `x = 100` 写成 `x=100` 吗？** 可以，但这种格式不好，加上空格阅读体验更好。

**你说的“倒着读文件”是什么意思？** 很简单，假如你有一个 16 行代码的文件，从第 16 行开始，和我文件中的第 16 行开始对比，然后是第 15 行等等，直到你把整个文件过完。

**为什么给 `space_in_a_car` 赋值要用 4.0？** 主要是为了让你知道什么是浮点数，以及问出这个问题，可以参考附加练习。



# 练习 5 更多变量和打印

现在我们要输入更多的变量并把它们打印出来。这次我们将用一个叫做“格式字符串”的东西。每次你用引号把一段文本引起来，你就是在创建一个字符串。字符串是你让计算机呈现给人看的内容。你可以打印字符串、把字符串保存到文件、或者发送到网络服务器等等。

字符串真的非常方便。你将在这个练习中学习如何创建包含变量的字符串。把你需要的变量放在 `{}` 里面就可以把变量嵌入在字符串中。你还需要在字符串前面加上字母 `f`（代表 format），比如 `f"Hello, {somevar}"`。双引号前面的 `f` 是为了告诉 python3：“这个字符串需要被格式化，把这些变量放在那儿。”

同样的，输入以下内容，哪怕你不理解，确保准确无误。

ex5.py

```
1     my_name = 'Zed A. Shaw'
2     my_age = 35 # not a lie
3     my_height = 74 # inches
4     my_weight = 180 # lbs
5     my_eyes = 'Blue'
6     my_teeth = 'White'
7     my_hair = 'Brown'
8
9     print(f"Let's talk about {my_name}.")
10    print(f"He's {my_height} inches tall.")
11    print(f"He's {my_weight} pounds heavy.")
12    print("Actually that's not too heavy.")
13    print(f"He's got {my_eyes} eyes and {my_hair} hair.")
14    print(f"His teeth are usually {my_teeth} depending on the coffee.")
15
16    # this line is tricky, try to get it exactly right
17    total = my_age + my_height + my_weight
18    print(f"If I add {my_age}, {my_height}, and {my_weight} I get {total}.")
```

## 你会看到

练习 5 会话

```
$ python3.6 ex5.py
Let's talk about Zed A. Shaw. He's 74 inches tall.
He's 180 pounds heavy.
Actually that's not too heavy.
He's got Blue eyes and Brown hair.
His teeth are usually White depending on the coffee.
If I add 35, 74, and 180 I get 289.
```

## 附加练习

1. 修改所有的变量，把前面的 `my_` 删掉。要更改所有的变量名，而不只是有 `=` 的部分。
2. 试着写一些变量，把英尺（inches）和英镑（pounds）换算成厘米（centimeters）和千克（kilograms），别自己直接把自己的数据进去，用 python 的数学运算来换算。

## 常见问题

**我能创建一个这样的变量吗：1 = 'Zed Shaw'？** 不能，`1` 不是一个有效的变量名。变量名需要以字母开头，比如 `a1` 就可以，但 `1` 不行。

**我如何给浮点数四舍五入取整数？** 你可以用 `round()` 函数，比如：`round(1.7333)`。

**为什么我还是不理解这些代码？** 试着把这些数字换成你自己的。虽然有点奇怪，但是与你自己相关能够让这些代码看起来更接地气。而且，你还刚开始学习，肯定会有不理解的地方。继续努力，再做一些练习你就会慢慢理解的。

## 练习 6 字符串和文本

你已经写过字符串了，但你还是不知道它们是用来干嘛的。在这个练习中，我们会创建一些包含复杂字符串的变量，让你看看它们的作用。首先解释一下字符串的含义。

字符串通常是一些你想要让你的程序呈现给别人或者“输出”出来的文本信息。当你把双引号或者单引号括在一段文本外面时，Python 就会知道你想要把这些文本变成字符串。你在学习 `print` 的时候应该多次看到这种用法了，当你想要打印一些文本的时候，你就把这些文本放在双引号或者

单引号里面。

字符串可以包含你的 Python 脚本中任意数量的变量。记住，变量就是让名字 = 一个值的那行代码。在本练习的代码中，`types_of_people = 10` 创建了一个名称为 `types_of_people`，值为 10 的变量。你可以用 `{types_of_people}` 的形式把这个变量放到任意字符串中。你还会看到我用了格式字符串（f-string），就像这样：

```
f"some stuff here { avariable }"  
f"some other stuff { anothervar }"
```

Python 还有其他种类的格式，就像你在第 17 行看到的 `.format()` 语法。你还会看到当我想对一个已经创建的字符串应用一种格式的时候，我就会这样用，比如在一个循环里，我们会在后面的内容中涉及到。

我们现在要输入一整段字符串、变量和格式，然后把它们打印出来。你还会练习使用缩写作为变量名。程序员都喜欢使用简短的缩写来节省时间，但是那些缩写在你看来会十分晦涩难懂。所以我们得尽早开始学习阅读和书写这些东西。

### ex6.py

```
1     types_of_people = 10  
2     x = f"There are {types_of_people} types of people."  
3  
4     binary = "binary"  
5     do_not = "don't"  
6     y = f"Those who know {binary} and those who {do_not}."  
7  
8     print(x)  
9     print(y)  
10  
11     print(f"I said: {x}")  
12     print(f"I also said: '{y}'")  
13  
14     hilarious = False  
15     joke_evaluation = "Isn't that joke so funny?! {}"  
16  
17     print(joke_evaluation.format(hilarious))  
18  
19     w = "This is the left side of..."  
20     e = "a string with a right side."  
21  
22     print(w + e)
```

# 你会看到

## 练习 6 会话

```
$ python3.6 ex6.py
There are 10 types of people.
Those who know binary and those who don't.
I said: There are 10 types of people.
I also said: 'Those who know binary and those who don't.'
Isn't that joke so funny?! False
This is the left side of...a string with a right side.
```

## 附加练习

1. 复习一遍这个程序，并在每一行上面写上注释来解释它。
2. 找到所有把字符串放在字符串里面的地方，一共有 4 处。
3. 你确定有 4 处吗？你怎么知道？也许我爱撒谎呢。
4. 解释一下为什么把 `w` 和 `e` 两个字符串用 `+` 连起来能够弄成一个更长的字符串。

## 把代码打乱

你现在已经可以把代码打乱了。把它当成一个游戏，用一种最聪明或者最简单的方式把代码打乱。打乱之后，你需要修复它们。如果你跟你的朋友一起学习，你们可以相互打乱对方的代码，然后再试着修复它。把你的 `ex6.py` 发给你的朋友，让他们打乱，然后你再试着找出它们的错误，并修复它。记住，如果你已经写了一遍这些代码了，你可以再写一次。如果你打乱得太彻底了，就试着重新写一遍。

## 常见问题

**为什么你在一些字符串外面放的是单引号，而其他的不是？** 大多数是因为格式。但是如果一个字符串已经用了双引号，我就会在这个字符串里面用单引号，看看第 6 行和第 15 行你就知道了。

**如果你觉得一个笑话很好笑，可以写 `hilarious = True` 吗？** 可以的，你会在练习 27 中学习到这些布尔值。

# 练习 7 更多打印

现在我们要做更多的练习，你只用输入代码让它运行即可。我不会做过多的解释，因为跟前面基本是一样的，目的是为了让你建立起自己的技能。千万别跳过，也别复制粘贴！

ex7.py

```
1     print("Mary had a little lamb.")
2     print("Its fleece was white as {}".format('snow'))
3     print("And everywhere that Mary went.")
4     print("." * 10) # what'd that do?
5
6     end1 = "C"
7     end2 = "h"
8     end3 = "e"
9     end4 = "e"
10    end5 = "s"
11    end6 = "e"
12    end7 = "B"
13    end8 = "u"
14    end9 = "r"
15    end10 = "g"
16    end11 = "e"
17    end12 = "r"
18
19    # watch that comma at the end. try removing it to see what h
20    print(end1 + end2 + end3 + end4 + end5 + end6, end=' ')
21    print(end7 + end8 + end9 + end10 + end11 + end12)
```

## 你会看到

### 练习 7 会话

```
$ python3.6 ex7.py
Mary had a little lamb.
Its fleece was white as snow.
And everywhere that Mary went.

.....
Cheese Burger
```

## 附加练习

接下来的附加练习基本也跟前面一样：

1. 回过头复习一遍代码，在每一行上面添加注释。
2. 倒着大声把每一行读出来，以发现你的错误。
3. 从现在开始，当你犯错了，就本子上写下你的错误。
4. 当你学习下个练习之前，看看这些错误，以避免再犯。
5. 记住每个人都会犯错。程序员就像音乐家一样总让别人觉得他们很完美，从不犯错，但其实他们经常犯错。

## 把代码打乱

在练习 6 中你还喜欢这种方式吗？从现在开始你要打乱你写的全部代码，或者你朋友的。我不会在每个练习中都写到这里，你要自觉来做这件事。你的目标是找到很多不同的方式来打乱你的代码，知道你试遍了所有可能的方法。在一些练习里我可能会提到一种人们通常使用的打乱方法。除此之外，把它当成一项标准的任务来完成吧。

## 常见问题

**为什么你要用这个叫做 `'snow'` 的变量？** 事实上那不是变量，它只是一个里面有 `snow` 这个单词的字符串，变量不会用单引号的。

**是不是写每一行代码都要加注释？** 不是，写注释只是为了向你自己解释一些难以理解的代码，或者你为什么要那样做。重要的是搞清楚为什么，然后你再试着写代码，让它实现一些事情。然而，有时候你不得不写一些让人讨厌的代码来解决一个问题，这个问题又需要你在每一行都写上注释，这时候你就应该严格地练练如何把代码用自然语言解释出来。

**单引号或者双引号都可以用来创建字符串吗？** 在 Python 里面，两个都可以，不过严格来讲，像 `a` 或者 `snow` 这种比较短的字符串应该用单引号。

## 练习 8 打印，打印

接下来我们要学习如何做更复杂的格式字符串。虽然它看起来很复杂，但是如果你认真做注释，



对于第 8 个练习来说，这样的信息量好像有点大，所以我希望你把它当成一道智力题。如果你真的不懂也没关系，因为这本书后面的内容会慢慢为你讲清楚。不过现在，试着学一学，看看会发生什么，然后再进行下面的练习。

## 加分练习

1. 检查你写的代码，把错误记下来，然后做下一个练习之前看一看，避免再犯同样的错误。

## 常见问题

为什么 `one` 要用引号，而 `True` 或者 `False` 却不用？Python 把 `True` 和 `False` 当成代表“对”和“错”的关键词。如果你给它们加引号，它们就会变成字符串而无法工作。你会在练习 27 中学到相关内容。

我能用 IDLE 来运行代码吗？不，你得学着用命令行。它对学习编程非常重要，并且是一个很好的起点。当你继续往下学这本书，你就会发现 IDLE 不管用了。

## 练习 9 打印，打印，打印

到现在为止，你应该意识到了这本书的模式，就是用很多练习来教你学习新东西。我先让你敲一些你可能不懂的代码，然后通过更多的练习来解释其中的概念。如果你现在有不懂的东西，在你完成更多练习之后你就会明白。先把你不理解的地方记下来，然后往下进行。

[ex9.py](#)



```
1      # Here's some new strange stuff, remember type it exactly.
2
3      days = "Mon Tue Wed Thu Fri Sat Sun"
4      months = "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"
5
6      print("Here are the days: ", days)
7      print("Here are the months: ", months)
8
9      print("""
10     There's something going on here.
11     With the three double-quotes.
12     We'll be able to type as much as we like.
13     Even 4 lines if we want, or 5, or 6.
14     """)
```

## 你会看到

### 练习 9 会话

```
$ python3.6 ex9.py
Here are the days: Mon Tue Wed Thu Fri Sat Sun
Here are the months:      Jan
Feb
Mar
Apr
May
Jun
Jul
Aug

There's something going on here.
With the three double-quotes.
We'll be able to type as much as we like.
Even 4 lines if we want, or 5, or 6.
```

## 附加练习

1. 检查你的代码，把错误记下来，以避免再犯。
2. 你有打乱你的代码然后重新修复吗？

# 常见问题

**为什么我在三个双引号之间加了空格就报错了呢？** 你必须这样输入 `"""`，而不能这样输入 `" " "`，也就是说中间不能加空格。

**如果我想让月份另起一行开始打印怎么办？** 你只需要在字符串前面加 `\n` 即可，就像这样：`"\nJan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"`。

**如果我的错误总是拼写错误是不是很糟糕？** 很多编程初学者（甚至非初学者）都会犯拼写错误，不用担心，细心点就行。

## 练习 10 那是什么？

在练习 9 中我教了你一些新东西。这两天我们一直在学习字符串。我教了你两种创建多行字符串的方式，第一种是在月份中间加 `\n`，它可以实现换行。

`\` 这个字符可以把没法输入的字符转化成字符串。有很多你可能会用到的“转义字符”（escape sequence），我们会在接下来的练习中学到一些，以便你理解我说的意思。

一个很重要的转义字符就是转义单引号或者双引号。比如你要在一个用双引号引起来的字符串中再加一对双引号，就像这样：`"I \"understand\" joe."`，python 就会懵掉，因为它会认为 `understand` 后面的双引号就代表这个字符串已经结束了。所以你需要用一种方式告诉 python 字符串里面的双引号并不是一个真正的双引号。

要解决这个问题，你得转义双引号和单引号，让 python 知道得把它们包含在字符串里。例如：

```
"I am 6'2\" tall." # escape double-quote inside string
'I am 6\'2" tall.' # escape single-quote inside string
```

第二种方法是用三个双引号，即 `"""`，这样就能像字符串一样运行，而且你可以多输入几行，最后再以 `"""` 结尾即可。我们来做个练习。

[ex10.py](#)

```
1 tabby_cat = "\tI'm tabbed in."
2 persian_cat = "I'm split\non a line."
3 backslash_cat = "I'm \\ a \\ cat."
4
5 fat_cat = """
6 I'll do a list:
7 \t* Cat food
8 \t* Fishies
9 \t* Catnip\n\t* Grass
10 """
11
12 print(tabby_cat)
13 print(persian_cat)
14 print(backslash_cat)
15 print(fat_cat)
```

## 你会看到

找一找你输入的 `tab` 符号（即 `\t` ），在这个练习中空格很重要，别弄错了。

```
$ python ex10 . py
    I'm tabbed in.
I'm split
on a line.
I'm \ a \ cat .

I'll do a list:
    *      Cat food
    *      Fishies
    *      Catnip
    *      Grass
```

## 转义字符

这是 python 支持的所有的转义字符了。你可能用不到这么多，但是记住它们的格式以及用法。在一些字符串里试着用用它们，看看能不能成功运行。

## 附加练习

1. 记住所有的转义字符。可以把它们添加到卡片上来记。
2. 改用三个单引号（`'''`），你知道什么情况下用它而不是三个双引号（`"""`）吗？
3. 把转义字符和格式字符串结合起来创建一个更复杂的字符串。

## 常见问题

**我还是没完全理解前面的练习，我该继续往下学吗？** 是的，继续学，别停在这儿。把你不明白的东西记在本子上，定期复习，等你做完更多的练习看你能不能理解。有时候你可能需要回过头去重新做做之前的练习才能明白。

**双反斜杠 `\\` 和其他符号有什么区别？** 它只是为了让你能把单反斜杠 `\` 打印出来，想想你为什么要用 `\\`。

**我要是用 `//` 或者 `/n` 就不行。** 因为你用的是斜杠而不是反斜杠。它们是不同的符号，有着不同的作用。

**我不明白附加练习的第 3 题。你说的把转义字符和格式字符串结合起来是什么意思？** 我需要一个理解一个概念，就是这些练习都可以结合起来解决问题。用你知道的关于格式字符串的东西和本练习学到的转义字符写一些新的代码。

**`'''` 和 `"""` 用哪个更好？** 这完全基于风格。现在先用 `'''`，当你感觉用 `"""` 更好或者别人都用它的时候你可以用 `"""`。

## 练习 11 问问题

现在可以缓一缓了。前面我们做了大量的打印练习，以让你熟悉这些简单的东西，但是的确，它们很无聊。我们现在要做的是在你的程序里放入数据。这块有点复杂，因为你得学着做两件你可能一下子理解不了的事情。但是相信我，无论如何先试试看。做几个练习之后你就会明白。

大多数软件就是做如下事情：

1. 从用户那里获得一些输入。
2. 改一改。
3. 打印出来一些东西以显示它变成了什么。

到现在为止你一直在打印东西，但是你还不知道怎么从用户那里获得 `input`（输入）。你甚至不知道“input”是什么意思。不管怎样，准确无误地输入这些代码，在下一个练习中我们会做更多的操作来解释 `input`。

ex11.py

```
1     print("How old are you?", end=' ')
2     age = input()
3     print("How tall are you?", end=' ')
4     height = input()
5     print("How much do you weigh?", end=' ')
6     weight = input()
7
8     print(f"So, you're {age} old, {height} tall and {weight} heavy.")
```

### 警告！

我们在每一个打印行末尾放一个 `end=' '`，是为了告诉 `print` 不要另起一行。

## 你会看到

### 练习 11 会话

```
$ python3.6 ex11.py
How old are you? 38
How tall are you? 6'2"
How much do you weigh? 180lbs
So, you're 38 old, 6'2" tall and 180lbs heavy.
```

## 附加练习

1. 上网查查 python 的 `input` 是干嘛的。
2. 你能找到它的其他使用方式吗？输入你找到的一些例子。
3. 再写一个像这样的格式，来问一些问题。

# 常见问题

我如何从别人那里获得一些数字来做数学运算？这就有点高级了，你可以试试输入

`x = int(input())`，这样就可以从 `input()` 里面获取到字符串形式的数字，再用 `int()` 把它们转化成数值。

我把我的体重作为 `input` 像这样输入进去：`input("6'2")`，但是不能正常运行。你别把你的体重放在那儿，你得直接在 Terminal 里面输入。首先，回去输入我让你输的代码；然后，运行脚本，当它暂停的时候，用你的键盘输入你的体重。这才是正确的做法。

## 练习 12 提示用户

当你输入 `()` 的时候，一定要确保输入完整，它们是成对出现的。对于 `input` 来说，你还可以给用户放一个提示，让他知道该输入什么。你可以把提示的字符串放在 `()` 里面，就像这样：

```
y = input("Name?")
```

这个提示告诉用户输入“名字”，然后把结果放到变量 `y` 里面。通过这种方式你就可以问用户问题然后得到他输入的答案。

这意味着我们可以重新写我们之前的练习，就用 `input` 来做所有的提示。

[ex12.py](#)

```
1     age = input("How old are you? ")
2     height = input("How tall are you? ")
3     weight = input("How much do you weigh? ")
4
5     print(f"So, you're {age} old, {height} tall and {weight} heavy.")
```

练习 12 会话

```
$ python3.6 ex12.py
How old are you? 38
How tall are you? 6'2"
How much do you weigh? 180lbs
So, you're 38 old, 6'2" tall and 180lbs heavy.
```

## 附加练习

1. 在 Terminal 里输入 `pydoc input`，看看它会说什么。如果你用的是 Windows，输入 `python3.6 -m pydoc input`。
2. 输入 `q`，退出 `pydoc`。
3. 到网上查查 `pydoc` 命令的作用。
4. 用 `pydoc` 读一读关于 `open`，`file`，`os`，和 `sys` 的内容；浏览一遍即可，把有意思的东西记下来。

## 常见问题

为什么我每次运行 `pydoc` 都会收到错误信息：`SyntaxError: invalid syntax`？要么你没在命令行运行 `pydoc`，要么你先运行了 `python3.6`，先退出 `python3.6` 再运行 `pydoc`。

为什么我的 `pydoc` 没有像你的一样暂停？有时候如果帮助文件很短，一屏足以放下的话，`pydoc` 就只会把它打印出来。

当我运行 `pydoc` 的时候我会收到 `more is not recognized as an internal`。一些 Windows 版本没有这个命令，你可以跳过这个小题，需要它的时候在网上搜搜 Python documentation 即可。

为什么我不能用 `print("How old are you?", input())`？你能，只不过 `input()` 的结果不会被保存到一个变量里，它会以一种奇怪的方式运行。你可以试试，然后试着打印你输入的东西，看看你能不能搞明白为什么它无法运行。

## 练习 13 参数，解包，变量

在这个练习中我们将会再涉及一种 `input` 方法，你可以用这种方法把变量传给一个脚本（也就是你的 `.py` 文件）。你知道如何运行 `ex13.py` 吧？输入 `python3.6 ex13.py` 就行（Windows 下输入 `python ex13.py`）。这句命令的 `ex13.py` 就叫做参数（argument）。我们现在要做的就是写一个也接受参数的脚本。

输入这个程序，然后我会详细解释：

[ex13.py](#)

```
1     from sys import argv
2     # read the WYSS section for how to run this
3     script, first, second, third = argv
4
5     print("The script is called:", script)
6     print("Your first variable is:", first)
7     print("Your second variable is:", second)
8     print("Your third variable is:", third)
```

第一行我们进行了“import”（导入），这能让你把 Python 功能库中的功能（features）添加到你的脚本中。Python 会问你你想用什么，而不是一次把所有的功能都给你。它会让你的程序很小，但是它同时也可以为其它阅读你代码的程序员提供参考。

这个 `argv` 是“argument variable”，一个在编程语言中非常标准的名字，你会在其它很多的语言中看到它的使用。当你运行 Python 脚本的时候，这个变量（variable）保存了你传给 Python 脚本的参数（argument）。在这个练习中，你会做更多相关的练习，看看会发生什么。

第三行“解包”（unpacks）了 `argv`，而不是保留所有的参数，它分成了四个变量：`script`，`first`，`second`，以及 `third`。这可能看起来很奇怪，但是“解包”这个词可能是对这个操作的最好定义，就好像在说：“把 `argv` 里面的东西解包，然后按顺序分配给从左到右每一个变量。最后就像平常一样把它们打印出来即可。

## 等等！Features 还有另一个名字

我在这儿把它们叫做 features（就是你导入进来让 python 做更多事情的东西），但是很少有人叫它们 features。我用这个名字只是因为我想让你在专业术语之外思考它们的真正含义。不过在你继续学习之前，你需要知道它们真正的名字：modules（模块）。

从现在开始我们会把这些 features 说成导入模块，比如，“你想导入 `sys` 模块”。它们还被有些程序员叫做“libraries”（库），但是我们就用模块这个名字吧。

## 你会看到

### 警告！

注意！你之前一直直接运行 python 脚本，不用输入命令行参数。如果你只输入 `python3.6 ex13.py` 你就错了！注意看我是怎么操作的，这在任何有 `argv`



## 警告！

的地方都会用到。

像这样运行这个程序，前面是你要传递的命令行参数：

### 练习 13 会话

```
$ python3.6 ex13.py first 2nd 3rd
The script is called: ex13.py
Your first variable is: first
Your second variable is: 2nd
Your third variable is: 3rd
```

当你做一些不同参数的运行时，你会看到：

### 练习 13 会话

```
$ python3.6 ex13.py stuff things that
The script is called: ex13.py
Your first variable is: stuff
Your second variable is: things
Your third variable is: that
$
$ python3.6 ex13.py apple orange grapefruit
The script is called: ex13.py
Your first variable is: apple
Your second variable is: orange
Your third variable is: grapefruit
```

事实上你还可以把 `first`，`2nd`，`3rd` 替换成任何你想替换的东西。如果你没有正确运行，你会收到这样的报错：

### 练习 13 会话

```
$ python3.6 ex13.py first 2nd
Traceback (most recent call last):
  File "ex13.py", line 3, in <module>
    script, first, second, third = argv
ValueError: not enough values to unpack (expected 4, got 3)
```

这种情况一般是当你运行脚本的时候没有在命令行放足够的变量（在本例中只有 `first`、`2nd`

）。注意当我运行的时候只给出 `first` 、 `2nd` ，就会出现错误说“需要三个以上的值来解包”，这就是告诉你，你没有给到足够多的参数。”

## 附加练习

1. 试着给你的脚本三个以内的参数，看看你会收到什么样的报错，你是否能解释它。
2. 写一个参数少的脚本和一个参数多的脚本，给未解包的变量起个合适的名字。
3. 把 `input` 和 `argv` 结合起来创建一个脚本，从用户那里获取更多 `input` 。别想得太难，就用 `argv` 来获取一些东西，再用 `input` 从用户那里获取一些东西。
4. 记住模块给我们一些特征，记住它叫模块（modules），我们之后会用到。

## 常见问题

当我运行的时候我收到了 `ValueError: need more than 1 value to unpack`。还记得我说过，学习编程的一项重要技能就是注意细节。如果你看了“你会看到”那部分，你就会看到我是如何在命令行上运行有参数的脚本的，你应该准确按照我做的来。

`argv` 和 `input()` 之间的区别是什么？区别取决于用户在哪被要求输入，如果是在命令行，就用 `argv`。如果你想让它们在程序已经运行的情况下用键盘输入，那就用 `input()` 。

命令行参数是字符串吗？是的，它们是以字符串的形式进来的，即使你在命令行输入的是数字。你可以用 `int()` 把它们转化成数值，就像 `int(input())` 。

你如何使用命令行？”你应该已经学过命令行的使用了，现在应该用得很 6 了。但是如果你还没有学，先去附录 A 学习命令行速成课程。

我不知道怎么把 `argv` 和 `input()` 结合在一起。别把它想得太难。就在脚本最后加两行，用 `input()` 获取一些东西，再打印出来。然后试着用更多方式在同一个脚本中使用这两样东西。

为什么我不能这样用：`input('? ') = x`？因为它写反了，按我的要求写，就能运行。

## 练习 14 提示和传递

让我们来做一個把 `argv` 和 `input` 结合在一起的练习来问用户一些特别的问题。这些问题会在你学习下一个练习中阅读和写文件的时候用到。在这个练习中我们会用一种不同的方式使用

`input`，就是让它打印出一个简单的 `>` 提示符。这有点像 Zork 或者 Adventure 这两款游戏。

ex14.py

```
1     from sys import argv
2
3     script, user_name = argv
4     prompt = '> '
5
6     print(f"Hi {user_name}, I'm the {script} script.")
7     print("I'd like to ask you a few questions.")
8     print(f"Do you like me {user_name}?")
9     likes = input(prompt)
10
11     print(f"Where do you live {user_name}?")
12     lives = input(prompt)
13
14     print("What kind of computer do you have?")
15     computer = input(prompt)
16
17     print(f"""
18     Alright, so you said {likes} about liking me.
19     You live in {lives}. Not sure where that is.
20     And you have a {computer} computer. Nice.
21     """)
```

我们把用户提示符设置成变量 `prompt`，然后把它赋给 `input` 而不是一遍遍地输入它们。现在如果我们想把提示符变成别的东西，只要修改一个地方，然后重新运行脚本即可，非常方便。

## 你会看到

当你运行脚本的时候，记住一定要把你的名字赋给这个脚本，让 `argv` 接收到你的名字。

练习 14 会话

```
$ python3.6 ex14.py zed
Hi zed, I'm the ex14.py script.
I'd like to ask you a few questions.
Do you like me zed?
>         Yes
Where do you live zed?
>         San Francisco
What kind of computer do you have?
>         Tandy 1000

Alright, so you said Yes about liking me.
You live in San Francisco. Not sure where that is.
And you have a Tandy 1000 computer. Nice.
```

## 附加练习

1. 查查看 Zork 和 Adventure 游戏是什么，找来玩玩。
2. 把 `prompt` 变量改成别的东西。
3. 在你的脚本里再加一个参数，就像之前练习中 `first, second = argv` 一样。
4. 确定你理解了我是如何在最后一行把 `"""` 多行格式字符（style multiline string）和 `{}` 格式激活器（format activator）结合起来的。

## 常见问题

我运行脚本的时候收到了 `SyntaxError: invalid syntax`。我再说一次，你得在命令行里运行它，而不是在 python 里。如果你输入 `python3.6`，然后再输入 `python3.6 ex14.py Zed`，就会无法运行，因为你是 python 里面运行 python。关掉窗口，然后只输入 `python3.6 ex14.py Zed`。

你说的“改变提示符”是什么意思？我不太理解。看到这个变量 `prompt = '>'` 了吗？改变它的值。你知道的，它只是一个字符串，你已经做了 13 个练习来创建它们了，所以好好想想，把它弄明白。

我收到了报错信息：`ValueError: need more than 1 value to unpack`。我前面说过，你需要看看“你会看到”那部分然后复制我的做法。这儿也一样，注意我是如何输入命令行的，以及我为什么有一个命令行参数。

我如何从 IDLE 来运行这些？不要用 IDLE。

我能在 `prompt` 变量外面用双引号吗？你完全可以，试试吧。

你有一台 `Tandy computer`？是的，在我很小的时候。

我运行的时候收到了报错信息：`NameError: name 'prompt' is not defined`。你要么把 `prompt` 变量拼写错了，要么把那行漏掉了。回过头去，从下到上比较每一行代码。你一旦遇到这种报错，就说明你拼写错误或者忘了创建变量。

## 练习 15 阅读文件

你已经知道如何用 `input` 或者 `argv` 来获取用户的输入了。现在你将学习如何阅读文件。你需要好好做这个练习，才能理解发生了什么，记住要仔细输入和检查。对文件进行操作很容易把文件删掉，所以你要千万小心。

在这个练习中你要写两个文件。一个是通常你要运行的 `ex15.py`，一个是叫做 `ex15_sample.txt` 的文本文件。以下是文本文件中要输入的内容：

```
This is stuff I typed into a file.  
It is really cool stuff.  
Lots and lots of fun to have in here.
```

我们要做的就是我们的脚本中打开这个文件并把它打印出来。然而，我们不想只是简单粗暴（hard coding）地把 `ex15_sample.txt` 这个文件名输入进去，hard coding 的意思是把一些应该从用户那里获取的信息直接放到源代码里。这样不好，因为我们随后会需要它载入别的文件。解决方法是用 `argv` 或者 `input` 来问用户应该打开哪个文件，而不是 hard coding 文件名。

[ex15.py](#)

```
1     from sys import argv
2
3     script, filename = argv
4
5     txt = open(filename)
6
7     print(f"Here's your file {filename}:")
8     print(txt.read())
9
10    print("Type the filename again:")
11    file_again = input("> ")
12
13    txt_again = open(file_again)
14
15    print(txt_again.read())
```

这个文件里发生了一些奇妙的东西，让我们快速分解来看一下：

第 1-3 行用了 `argv` 来获取一个文件名，然后第 5 行用了一个新的命令 `open`。现在，运行 `pydoc open` 然后阅读说明。看见了吧，就像你自己的脚本和输入，它用了参数（parameter）然后返回了一个值，你可以把它赋给你自己的变量。你只是打开了一个文件。

第 7 行打印了一些信息，第 8 行就有一些新东西了。我们对 `txt` 用了一个叫做 `read` 的函数，你从 `open` 那里得到的是一个文件，而且你还可以通过 . 命令名，以及参数，来给它一个命令，就像用 `open` 和 `input` 那样。区别是，`txt.(read)` 是说：`txt`，执行不带参数的 `read` 命令！

剩下的部分基本上类似，但是我们会把分析留到附加练习里。

## 你会看到

### 警告！

注意！你一直通过输入文件名来运行脚本，但是你要用 `argv` 加上参数来运行。看看下面例子的第一行，你会看我是通过输入 `python ex15.py ex15_sample.txt` 来运行它的。`ex15.py` 后面的内容就是你要输入的参数，如果你漏掉了，就会收到报错信息！所以千万要注意！

我创建了一个叫做 `ex15_sample.txt` 的文件来运行我的脚本。

## 练习 15 会话

```
$ python3.6 ex15.py ex15_sample.txt
Here's your file ex15_sample.txt:
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.

Type the filename again:
>      ex15_sample.txt
This is stuff I typed into a file.
It is really cool stuff.
Lots and lots of fun to have in here.
```

## 附加练习

这部分可能比较难，在你往下进行之前，确保你尽力去做这个附加练习。

1. 在每行上面添加注释解释其含义。
2. 如果你不确定，上网搜，或者问别人，比如你不知道 `open` 的用法，直接搜 `python3.6 open` 即可。
3. 我在这儿用的是“命令”（command）这个词，不过，它也叫“函数”（function）或者“方法”（method）。你会在本书后面学到 functions 和 methods。
4. 把第 10-15 行删掉（或者用别的方法使其失效）然后再运行脚本。
5. 只用 `input` 来试试运行这个脚本。为什么要获取文件名的话一种方法比另一种方法更好？
6. 开启 `python3.6 shell`，然后就像这个程序中一样从提示界面用 `open`。注意你是如何从 `python3.6` 里面打开文件并运行 `read` 的？
7. 在你的脚本中对 `txt` 调用 `close()` 以及 `txt_again` 变量。当你对它们完成操作后关掉文件是非常重要的。

## 常见问题

`txt = open(filename)` 会返回文件的内容吗？不会。它其实是创建了一个叫做“文件对象”（file object）的东西。你可以把它想象成曾经的 DVD 播放器，你可以在里面移动然后“读取”它们。但是 DVD 播放器不是 DVD 本身，就像文件对象也不是文件本身一样。

我无法像你附加练习 7 中说的那样在 Terminal/PowerShell 里输入代码。首先，在命令行输入 `python3.6` 然后敲回车。现在你已经在 `python3.6` 里面了。然后你可以输入代码，`python` 就会运行一些。试着这样玩玩，然后输入 `quit()` 并回车，退出。

### 为什么打开文件两次不会收到报错？

Python 不会限制你只能打开一次文件，事实上有时候确实需要打开多次。

`from sys import argv` 是什么意思？现在你只需要明白 `sys` 是一个包（package），这个短语是说从那个包里获取 `argv` 功能（feature）。你会在后面深入学习这块内容。

我把脚本文件名这样放进去：`ex15_sample.txt = argv`，但是无法运行。你不能这样做。严格按照我的代码来，然后用同样的方法在命令行运行它。你不用把文件名放进去，你得让 `python` 自己放。

## 练习 16. 读写文件

如果你做了上一节的附加练习，你应该看到了所有的命令（commands, modules, functions），你可以把这些命令施加给文件。以下是一些我想让你记住的命令：

- **close** - 关闭文件，就像编辑器中的“文件->另存为”一样。
- **read** - 读取文件内容。你可以把读取结果赋给一个变量。
- **readline** - 只读取文本文件的一行内容。
- **truncate** - 清空文件。清空的时候要当心。
- **write('stuff')** - 给文件写入一些“东西”。
- **seek(0)** - 把读/写的位置移到文件最开头。

这些都是你需要知道的一些非常重要的命令。其中一些要用到参数，但是我们暂且不去重点关注。你只需要记住 `write` 命令需要你提供一个你要写入的文件的字符串参数。

让我们用这些命令做一个小小的编辑器：

[ex16.py](#)



```
1     from sys import argv
2
3     script, filename = argv
4
5     print(f"We're going to erase {filename}.")
6     print("If you don't want that, hit CTRL-C (^C).")
7     print("If you do want that, hit RETURN.")
8
9     input("?")
10
11    print("Opening the file...")
12    target = open(filename, 'w')
13
14    print("Truncating the file. Goodbye!")
15    target.truncate()
16
17    print("Now I'm going to ask you for three lines.")
18
19    line1 = input("line 1: ")
20    line2 = input("line 2: ")
21    line3 = input("line 3: ")
22
23    print("I'm going to write these to the file.")
24
25    target.write(line1)
26    target.write("\n")
27    target.write(line2)
28    target.write("\n")
29    target.write(line3)
30    target.write("\n")
31
32    print("And finally, we close it.")
33    target.close()
```

这真是一个很大的文件，可能是你输入过的最大的文件了。所以慢一点，写完检查一下，然后再运行。你也可以写一点运行一点，比如先运行 1-8 行，然后再多运行 5 行，然后再多几行，直到所有的都完成和运行了。

## 你应该看到

事实上你应该看到两样东西，首先是你新脚本的输出结果：

### 练习 16 会话

```
$ python3.6 ex16.py test.txt We're going to erase test.txt.
If you don't want that, hit CTRL-C (^C). If you do want that, hit RETURN.
?
Opening the file...
Truncating the file.          Goodbye!
Now I'm going to ask you for three lines.
line 1:      Mary had a little lamb
line 2:      Its fleece was white as snow
line 3:      It was also tasty
I'm going to write these to the file.
And finally, we close it.
```

现在，用编辑器打开你创建的文件（比如我的是 test.txt），检查一下是不是对的。

## 附加练习

1. 如果你理解不了这个练习，回过头去按照给每行加注释的方法再过一遍，注释能帮助你理解每一行的意思，至少让你知道你理解不了的地方在哪里，然后动手去查找答案。
2. 写一个类似于上个练习的脚本，使用 `read` 和 `argv` 来读取你刚刚创建的文件。
3. 这个练习中有太多的重复，试着用一个 `target.write()` 命令来打印 line1、line2、line3，你可以使用字符串、格式字符串和转义字符。
4. 弄明白为什么我们要用一个 `'w'` 作为一个额外的参数来打开。提示：通过明确说明你想要写入一个文件，来安全地打开它。
5. 如果你用 `w` 模式打开文件，那你还需要 `target.truncate()` 吗？读一读 Python 的 `open` 函数文件，来搞明白这个问题。

## 常见问题

`truncate()` 对于 `'w'` 参数来说是必须的吗？详见附加练习 5。

`'w'` 到底是什么意思？它真的只是一个有字符的字符串，来表示文件的一种模式。如果你用了 `'w'`，就代表你说“用 `'write'` 模式打开这个文件。此外还有 `'r'` 表示 read 模式，`'a'` 表示增补模式，后面还可能加一些修饰符（modifiers）。

我能对文件使用哪些修饰符？目前最重要的一个就是 `+`，你可以用 `'w+'`，`'r+'` 以及 `'a+'`。这样会让文件以读和写的模式打开，取决于你用的是那个符号以及文件所在的位置等。

如果只输入 `open(filename)` 是不是就用 `'r'`（读）模式打开？是的，那是 `open()` 函数的默认值。

## 练习 17. 更多文件

现在让我们对文件做更多新的操作。我们会写一个 Python 脚本来把一个文件复制成另一个。代码会非常短，但是能让你学会对文件做更多的操作。

[ex17.py](#)

```
1     from sys import argv
2     from os.path import exists
3
4     script, from_file, to_file = argv
5
6     print(f"Copying from {from_file} to {to_file}")
7
8     # we could do these two on one line, how?
9     in_file = open(from_file)
10    indata = in_file.read()
11
12    print(f"The input file is {len(indata)} bytes long")
13
14    print(f"Does the output file exist? {exists(to_file)}")
15    print("Ready, hit RETURN to continue, CTRL-C to abort.")
16    input()
17
18    out_file = open(to_file, 'w')
19    out_file.write(indata)
20
21    print("Alright, all done.")
22
23    out_file.close()
24    in_file.close()
```

你应该会很快注意到我们输出了另一个常用命令 `exists`。它会基于一个字符串里面的变量文件名来判断，如果一个文件存在，它就会返回 `True`，不存在就会返回 `False`。我们会在这本书的下半部分经常使用这个函数，现在你只用知道你是如何输出它的。

使用 `import` 可以调出海量的免费代码，这些是程序员已经写过的代码，你就不用重复造轮子了。

# 你会看到

像你其他的脚本文件一样运行 `ex17.py`，再加上两个变量：一个是要复制的源文件，一个是要复制到的目标文件。我会使用一个叫 `test.txt` 的示例文件：

练习 17 会话

```
ParseError: KaTeX parse error: Expected 'EOF', got '#' at position 1: #_first make a s...
echo "This is a test file." > test.txt
```

```
ParseError: KaTeX parse error: Expected 'EOF', got '#' at position 1: #_then look at i... cat
test.txt
```

```
This is a test file.
```

```
ParseError: KaTeX parse error: Expected 'EOF', got '#' at position 1: #_now run our sc...
python3.6 ex17.py test.txt new_file.txt
```

```
Copying from test.txt to new_file.txt
```

```
The input file is 21 bytes long
```

```
Does the output file exist? False
```

```
Ready, hit RETURN to continue, CTRL-C to abort.
```

```
Alright, all done.
```

它应该对每一个文件都适用，多试一些看看会发生什么。注意不要动重要的文件。

## 警告！

你应该注意到我使用了 `cat` 命令来显示文件内容。如果你不懂，可以在附录 A 的命令行速成教程中学到这个命令。

# 附加练习

1. 这个脚本真的很烦人。在复制之前其实没必要问你，而且它打印了太多内容，试着通过删掉一些特征让这个脚本更简洁一些。
2. 看看你把这个脚本缩到多短，我可以把它变成一行。
3. 注意“你会看到”部分，我用了 `cat`，这是一种把文件打印到屏幕的简单办法，你可以输入 `man cat` 来看看关于这个命令的作用。
4. 弄明白你为什么得在代码里写 `out_file.close()`。
5. 去读读 Python 的 import statement，然后自己试试 import 一些东西，看能不能成功运行。不成功也没关系。

# 常见问题

为什么 `'w'` 要用引号？因为这是一个字符串。你已经用了一段时间字符串了，确保你知道它的含义。

你不可能把那些代码变成一行！那；取决于；你；如何；定义；一行；代码。

我觉得这个练习很难，这正常吗？是的，很正常。甚至到练习 36 的时候，或者学完这本书的时候，你可能还是会觉得很难。每个人的情况都不一样，所以坚持学下去，坚持做练习，要有耐心。

`len()` 函数是什么作用？它能够取字符串的长度，然后返回一个数字。你可以试着玩玩。

但我试着把这些代码缩短的时候，我在关闭文件时遇到了错误。你可能用了

`indata = open(from_file).read()`，这意味着你不需要在之后再输入 `in_file.close()`，因为你已经到了脚本的最后。一旦那一行运行过之后，它就已经被 Python 关掉了。

我收到了一个这样的错误：`SyntaxError: EOL while scanning string literal`。你忘了在字符串后面加引号了，再检查一遍的代码。

## 练习 18 名称，变量，代码，函数

这是一个很大的标题。接下来我要给你介绍一下函数。每一个程序员都要一遍一遍地用到函数，思考它们的作用以及如何使用它们，但是我会给你一些最简单的解释，让你能够快速上手。

函数一般就是做以下这些事情：

1. 它们为一些代码起名字，就像变量为字符串和数字起名字一样。
2. 它们像脚本获取 `argv` 一样获取参数（arguments）。
3. 通过 1 和 2 的操作，让你做一些你自己的“小脚本”或者“微命令”。

你可以通过在 Python 中使用 `def` 来创建一个函数。我会让你创建 4 个不同的函数，它们就像你的脚本一样运行，之后我还会想你展示每一个之间是如何关联的。

[ex18.py](#)

```

1      # this one is like your scripts with argv
2      def print_two(*args):
3          arg1, arg2 = args
4          print(f"arg1: {arg1}, arg2: {arg2}")
5
6      # ok, that *args is actually pointless, we can just do this
7      def print_two_again(arg1, arg2):
8          print(f"arg1: {arg1}, arg2: {arg2}")
9
10     # this just takes one argument
11     def print_one(arg1):
12         print(f"arg1: {arg1}")
13
14     # this one takes no arguments
15     def print_none():
16         print("I got nothin'.")
17
18
19     print_two("Zed", "Shaw")
20     print_two_again("Zed", "Shaw")
21     print_one("First!")
22     print_none()

```

让我们把第一个函数拆解一下，`print_two` 这是你从创建脚本中已经学到的最熟悉的东西：

1. 首先，我们告诉 Python 我们想要用 `def`（即 define）来创建一个函数。
2. 在 `def` 的同一行我们给了函数一个名字，本例中是 `print_two`，但是你也可以起名叫“peanuts”（花生），名字没关系，不过最好简短一些，并且能够说明这个函数的作用。
3. 然后我们告诉它我们想要 `*args`，它很像参数 `args`，只不过是给函数设的，必须放在括号里面才能工作。
4. 然后我们以 `:` 结束这一行，另起一行开始缩进。
5. 在 `:` 之后缩进四个空格的所有行都是关于 `print_two` 这个函数名的。我们第一个缩进的行就是用来解包这个参数（argument），跟之前的脚本一样。
6. 要表明它是如何工作的，我们把这些参数打印了出来，就像我们在脚本中所做的一样。

`print_two` 的问题是它不是创建一个函数最简单的方法。在 python 里面，我们可以跳过整个解包参数的过程，只用我们需要的 `()` 里面的名字即可，这也正是 `print_two_again` 所做的事情。

之后我们用一个参数创建了 `print_one` 这个函数。

最后我们创建了一个没有参数的函数 `print_none`。

### 警告！

这很重要。如果你现在不太明白，别急着灰心，我们会再做几个跟函数相关的练习来进一步学习。现在当

## 你会看到

如果你运行了 `ex18.py`，你会看到：

### 练习 18 会话

```
$ python3.6 ex18.py
arg1: Zed, arg2: Shaw
arg1: Zed, arg2: Shaw
arg1: First!
I got nothin'.
```

现在你已经看到了函数是如何工作的。注意你使用函数的方式就像你使用 `exists`、`open` 等其他一些“命令”一样。其实我一直在跟你卖关子，因为在 `python` 里面，这些“命令”就是“函数”（学习癌注：哈哈哈哈哈老肖太坏了）。这意味着你可以创建你自己的命令然后在你的脚本中使用。

## 附加练习

创建一个如下的函数 `checklist`（核查表）用于后面的练习。把这些内容写在索引卡上，一直保留到你完成所有剩余练习的时候或者当你感觉你不再需要这些索引卡的时候：

1. 你是否用 `def` 来创建函数了？
2. 你的函数名是只包含字符和 `_`（下划线）吗？
3. 你在函数名后面放 `(`（左圆括号）了吗？
4. 你在左圆括号后面放参数（argument）了吗？参数之间是以逗号隔开的吗？
5. 你的每个参数都是唯一的吗（即没有重名）？
6. 你在参数后面放 `)`（右圆括号）和 `:`（冒号）了吗？
7. 你在与这个函数相关的代码行前面加上四个空格的缩进了吗？（不能多，也不能少）

8. 你是通过另起一行不缩进来结束你的函数的吗？

当你运行（使用或者调用）一个函数时，检查以下事项：

1. 你是通过输入函数名称来运行/调用/使用一个函数的吗？
2. 你运行的时候有在名称后面加 `(` 吗？
3. 你有把你想要的值放在圆括号里并用逗号隔开了吗？
4. 你是以 `)` 来结束调用这个函数的吗？

在接下来的课程中用这两个 checklist，直到你不再需要它们为止。

最后，再强调以下，我说的“运行”（run）、“调用”（call）、“使用”（use）都是一个意思。

## 常见问题

函数名称有哪些要求？跟变量名一样，任何不以数字开头的字母、数字、下划线组合都可以。

`*args` 中的 `*` 是什么作用？这是告诉 Python 取所有的参数给函数，然后把它们放在 `args` 里放成一列，很像你之前学的 `argv`，只不过这个是为函数设置的。这种不常用，除非有特殊需要。

这部分好无聊好烦人啊。这就对了，这说明你已经开始一边输入代码一边思考它的作用了。如果想让它不这么无聊，按照我的要求一字不差地输入进去，然后再故意打乱它们，看看你能不能修复好。

## 练习 19 函数和变量

函数是一个信息量巨大的东西，但是别担心，老老实实做练习，仔仔细细核对 checklist，你最终会掌握它的。

有个小点你可能没注意到，我们会在之后进行强化：你函数里面的变量跟你脚本里面的变量没有关联。通过下面这个练习思考一下这个问题：

[ex19.py](#)



```
1     def cheese_and_crackers(cheese_count, boxes_of_crackers):
2         print(f"You have {cheese_count} cheeses!")
3         print(f"You have {boxes_of_crackers} boxes of crackers!")
4         print("Man that's enough for a party!")
5         print("Get a blanket.\n")
6
7
8     print("We can just give the function numbers directly:")
9     cheese_and_crackers(20, 30)
10
11
12 print("OR, we can use variables from our script:")
13 amount_of_cheese = 10
14 amount_of_crackers = 50
15
16 cheese_and_crackers(amount_of_cheese, amount_of_crackers)
17
18
19 print("We can even do math inside too:")
20 cheese_and_crackers(10 + 20, 5 + 6)
21
22
23 print("And we can combine the two, variables and math:")
24 cheese_and_crackers(amount_of_cheese + 100, amount_of_crackers + 1000)
```

这个练习展示了我们可以给函数 `cheese_and_crackers` 赋值的几种不同的方式，我们可以直接给它数字，或者变量，亦或是数学运算，甚至是数学运算和变量的结合。

从某种程度上说，函数的参数有点类似于我们给变量赋值时的 `=` 符号。事实上，如果你可以用 `=` 来定义一个东西，你就可以把它作为参数赋给函数。

## 你会看到

你应该研究一下这个脚本的输出结果，把它和你之前的脚本输出结果对比一下。

练习 19 会话

```
$ python3.6 ex19.py
We can just give the function numbers directly:
You have 20 cheeses!
You have 30 boxes of crackers!
Man that's enough for a party!
Get a blanket.
```

```
OR, we can use variables from our script:
You have 10 cheeses!
You have 50 boxes of crackers!
Man that's enough for a party!
Get a blanket.
```

```
We can even do math inside too:
You have 30 cheeses!
You have 11 boxes of crackers!
Man that's enough for a party!
Get a blanket.
```

```
And we can combine the two, variables and math:
You have 110 cheeses!
You have 1050 boxes of crackers!
Man that's enough for a party!
Get a blanket.
```

## 附加练习

1. 回顾一遍这个脚本，然后在每一行上方加上注释，解释它的作用。
2. 从下到上阅读每一行，说出所有重要的字符。
3. 写至少一个自己设计的函数，然后用 10 种不同的方式运行它。

## 常见问题

运行一个函数怎么可能有 **10 种不同的方式**？爱信不信，理论上讲，任何函数都有无数种调用方式。看看你对于函数、变量以及用户输入的创造力有多强。

有没有什么方法能分析函数是如何运行的，以帮助我更好地理解它？有很多方法，但是你先试试给每行加注释这种方式。其他方法包括大声把代码读出来，或者把代码打印出来然后在上面画图，来展示它是怎么运行的。

如果我想问用户关于 **cheese** 和 **crackers** 的数字呢？你需要用 `int()` 来把你通过 `input()` 获取的内容转化成数值。

在函数中创建 `amount_of_cheese` 这个变量会改变 `cheese_count` 这个变量吗？不会的，这些变量是相互独立并存在于函数之外的。它们之后会传递给函数，而且是“暂时版”，只是为了让函数运行。当函数退出之后，这些暂时的变量就会消失，其他一切正常运行。接着往下学，你会慢慢明白的。

像 `amount_of_cheese` 这样的全局变量（`global variables`）跟函数变量同名的话是不是不太好？是的，如果这样的话，你就不知道你说的到底是哪个变量了。不过你有时候可能不得不用同样的名字，或者你可能不小心同名了，不管怎么样，尽量避免这种情况。

一个函数里包含的参数有数量限制吗？这取决于 Python 的版本以及你的电脑，但是这个数量其实相当大。实践中一个函数包含 5 个参数为宜，再多就比较难用了。

你能在一个函数里面调用一个函数吗？可以，在之后的练习里你会创建一个小游戏，到时候就会用到这个。

## 练习 20 函数和文件

记住你的函数 `checklist`，然后在做这个练习的时候注意函数是如何和文件一起工作并发挥一些作用的。

[ex20.py](#)

```
1     from sys import argv
2
3     script, input_file = argv
4
5     def print_all(f):
6         print(f.read())
7
8     def rewind(f):
9         f.seek(0)
10
11 def print_a_line(line_count, f):
12     print(line_count, f.readline())
13
14 current_file = open(input_file)
15
16 print("First let's print the whole file:\n")
17
18 print_all(current_file)
19
20 print("Now let's rewind, kind of like a tape.")
21
22 rewind(current_file)
23
24 print("Let's print three lines:")
25
26 current_line = 1
27 print_a_line(current_line, current_file)
28
29 current_line = current_line + 1
30 print_a_line(current_line, current_file)
31
32 current_line = current_line + 1
33 print_a_line(current_line, current_file)
```

着重注意我们是如何在每次运行 `print_a_line` 的时候把当前行的数字传递出去的。

## 你会看到

### 练习 20 会话

```
$ python3.6 ex20.py test.txt
First let's print the whole file:

This is line 1
This is line 2
This is line 3

Now let's rewind, kind of like a tape.
Let's print three lines:
1      This is line 1

2      This is line 2

3      This is line 3
```

## 附加练习

1. 在每一行上方添加注释解释它的作用。
2. 每次 `print_a_line` 运行的时候，你都在传入一个 `current_line` 变量。写出每一次调用函数的时候 `current_line` 等于什么，然后找出它是如何变成 `print_a_line` 里面的 `line_count` 的。
3. 找出每一个用到函数的地方，然后检查它的 `def` 确保你给出了正确的参数。
4. 在网上搜搜 `seek` 这个函数的作用。试着输入 `pydoc file`，看看你能否从这里看明白。然后试着输入 `pydoc file.seek` 再看看 `seek` 是用来干嘛的。
5. 搜一下简化符号 `+=`，然后用 `+=` 重新写这个脚本。

## 常见问题

在 `print_all` 和其他函数里的 `f` 是什么东西？`f` 是一个变量，就像你在练习 18 中函数的变量一样，只不过这次它是一个文件。文件在 Python 里面有点类似于一个老式电脑里面的磁带驱动器，或者一个 DVD 播放机。它有一个“读取头”（read head），你可以在文件里 `seek`（寻找）这个读取头所在的位置，然后在那里工作。每次你做 `f.seek(0)` 的时候你都会从移动到文件最开始，每次你做 `f.readline()` 的时候，你都在从文件里读取一行内容，并且把读取头移动到 `\n` 后面，也就是每行结束的地方。我会在后面给你做更详细的解释。

**为什么 `seek(0)` 没有把 `current_line` 设置为 0？** 首先，`seek()` 函数处理的是字节（bytes），不是行。`seek(0)` 这个代码把文件移动到 0 字节（也就是第一个字节处）。其

次，`current_line` 只是一个变量并且跟这个文件没有任何实际联系。我们是在手动累加它。

什么是 `+=` ？你知道在英语里我们可以把 “it is” 写成 “it's”，或者把 “you are” 写成 “you're”，这叫缩写（contraction）。而 `+=` 就像 `=` 和 `+` 两种运算的缩写。也就是 `x = x + y` 就等同于 `x += y`。

`readline()` 是怎么知道每一行在哪儿的？`readline()` 里面的代码能够扫描文件的每个字节，当它发现一个 `\n` 字符，它就会停止扫描这个文件，然后回到它发现的地方。文件 `f` 就负责在每次调用 `readline()` 之后维持文件的当前位置，以此来保证它能阅读到每一行。

为什么文件中的行之间会有空行？`readline()` 函数返回文件中每行最后的 `\n`。又在 `print` 函数的结尾加上一个 `end = " "` 来避免给每行加上两个 `\n`。

## 练习 21 函数可以返回一些东西

你已经使用了 `=` 来命名变量并给变量赋予数值或字符串。接下来我会教你如何 `=` 和一个新的 python 字符 `return` 来把函数中的变量设置为一个值。有一点需要密切注意，但是先输入如下代码：

[ex21.py](#)

```

1     def add(a, b):
2         print(f"ADDING {a} + {b}")
3         return a + b
4
5     def subtract(a, b):
6         print(f"SUBTRACTING {a} - {b}")
7         return a - b
8
9     def multiply(a, b):
10        print(f"MULTIPLYING {a} * {b}")
11        return a * b
12
13 def divide(a, b):
14     print(f"DIVIDING {a} / {b}")
15     return a / b
16
17
18     print("Let's do some math with just functions!")
19
20     age = add(30, 5)
21     height = subtract(78, 4)
22     weight = multiply(90, 2)
23     iq = divide(100, 2)
24
25     print(f"Age: {age}, Height: {height}, Weight: {weight}, IQ: {iq}")
26
27
28     # A puzzle for the extra credit, type it in anyway.
29     print("Here is a puzzle.")
30
31     what = add(age, subtract(height, multiply(weight, divide(iq, 2))))
32
33     print("That becomes: ", what, "Can you do it by hand?")

```

我们现在要做我们自己的加减乘除数学运算了。我说的要密切注意的是 `add` 函数里面的 `return a + b`，这一步做的是这些事情：

1. 我们的函数是以两个参数被调用的：`a` 和 `b`。
2. 我们把函数所做的事情打印出来，在本例中是“ADDING”。
3. 然后我们让 Python 做一些反向的事情：我们返回 `a + b` 的和。你可以这样描述：我用 `a` 加上 `b`，然后返回它们的结果。
4. Python 把这两个数加起来。然后当函数终止的时候，运行了这个函数的任何一行都能够将 `a + b` 的结果赋予一个变量。

和这本书里其他内容比起来，这块你确实应该把节奏放慢一些，把代码打乱，然后试着琢磨一下每一步都发生了什么。

## 你会看到

### 练习 21 会话

```
$ python3.6 ex21.py
Let's do some math with just functions!
ADDING 30 + 5
SUBTRACTING 78 - 4
MULTIPLYING 90 * 2
DIVIDING 100 / 2
Age: 35, Height: 74, Weight: 180, IQ: 50.0
Here is a puzzle.
DIVIDING 50.0 / 2
MULTIPLYING 180 * 25.0
SUBTRACTING 74 - 4500.0
ADDING 35 + -4426.0
That becomes:          -4391.0
Can you do it by hand?
```

## 附加练习

1. 如果你还不能真正理解 `return` 是干什么的，试着写几个你自己的函数，并且让它们返回一些值。你可以让它 `return` 任何东西，只要你把它们放在 `=` 右边即可。
2. 脚本的最后是一个难题。我在用一个函数的返回值作为另一个函数的参数，这是在一个链（chain）里面进行的，这样就用函数创建了一个公式。它看起来确实很难，但是如果你运行这个脚本，你就可以看到结果。你要做的就是试着弄明白创建同样操作的平常的函数是什么样的。
3. 一旦你有了可以解出这个难题的公式，试着对函数的某些部分做做改动，看看会发生什么。有意改动一些数让它产生一些不同的值。
4. 做相反的操作。写一个简单的公式，然后用同一种方式通过函数来计算它。

这个练习可能真的很让你头大，但是放松，慢点学，把它当成是一个小游戏。正是解决这样的难题让编程如此有趣，所以之后我还会再给你一些小问题让你解决。



# 常见问题

**为什么 python 是“从后往前”（backward打印公式或者函数的？** 它其实不是从后往前，它是从里到外（inside out）。当你开始把代码打乱成分开的公式和函数时，你会看到它是如何工作的。试着理解我说的“inside out”而不是“backward”。

**我如何使用 `input()` 来输入我自己的值？** 还记得 `int(input())` 吗？这样做的问题是你不能输入浮点数，所以试着用 `float(input())` 来代替。

**你说的“写一个公式”是什么意思？** 先试试 `24 + 34 / 100 - 1023` 吧，变成使用函数来计算。然后自己想出一个类似的数学公式，要用变量让它看起来更像一个公式。

## 练习 22 你目前为止学到了什么？

这个练习和下一个练习不会有任何的代码，因此也不会有“你会看到”和“附加练习”这两个部分。事实上，这个练习就像一个大的附加练习，我会让你对之前所有学过的内容做一个回顾复习。

首先，回顾一下你目前为止做过的每一个练习，写下你用过的每个单词和符号，确保你的符号列表是完整的。

在每个单词或符号的旁边，写下它的名字和作用。如果你在这本书里面找不到某个符号的名字，可以试试在网上找找。如果你不知道某个单词或者符号的作用，认真读读与它相关的内容，然后试着在代码中用它们。

你可能会遇到一些你不明白或者无法解决的问题，把这些记在你的 list 上面，当你找到答案或者想明白以后可以补充上去。

一旦你有了自己的 list，花些时间重新写一遍这个 list，然后检查一下你写得对不对。这样做可能会很无聊，但是逼自己一下，把它落实下来。

如果你已经记住了这个 list 以及它们的作用，可以更进一步，写下字符表、它们的名字和作用，并争取记在脑子里。要是遇到你想不起来的内容，复习一下然后再记一遍。

### 警告！

做这些练习的时候，最重要的事情是：没有失败，只有尝试！

# 你正在学的

当你在做一件无聊的、不用动脑子去记的练习时，知道“为什么”很重要，它能帮助你专注一个目标，并且知道你是为了什么而做这些努力。

在这个练习中，你在学习符号的名称，以便更容易地阅读源代码。就像学习英文字母表和基础单词一样，只不过 python 的字母表还有你可能不认识的符号。

如果觉得难了，就把节奏放慢一些，不要想破脑袋地去学。看 list 的时候，最好每次看 15 分钟然后休息一下，让大脑放松一会儿可以帮助你写得更快更轻松。

## 练习 23 字符串，字节和字符编码

要做这个练习你需要去下载一个名为 `languages.txt` 的文本文件（下载地址：<https://learnpythonthehardway.org/python3/languages.txt>，点开，右键，“另存为”txt 格式，放在你的练习文件夹，再打开。）

这个文件列了一个人类的自然语言列表来说明一些有趣的概念：

1. 现代计算机是如何储存自然语言然后显示和加工的，还有 python 3 是如何调用这些字符串的。
2. 你是如何（同时也是必须）把 python 的字符串“编码”（encode）和“解码”（decode）成字节（byte）形式的。
3. 如何处理你字符串里以及字节处理过程中的错误。
4. 如何阅读代码并弄明白它的意思，即使你从来都没有看到过这些代码。

此外，你还会一睹 python 3 的 `if` 语句以及处理一系列东西的列表。你不用马上掌握这些代码或者理解这些概念，你会在接下来的练习中有足够的练习来学习。现在你只需要尝个鲜，并且弄明白前面所述的四个问题即可。

警告

这个练习很难，需要你理解的信息很多，而且这些信息深入到计算机理论中。这个练习之所以很难，是因为的字符串本身很复杂，很难用。我建议你在学这个练习的时候放慢节奏。写下你不明白的每一个单词，然

# 初始研究

我准备教你如何研究一段代码来发现它的奥秘。你要用到 `languages.txt` 文件，所以先确保你已经下载了这个文件。这个 `languages.txt` 文件包含了一个人类各种语言的列表，并且是以 UTF-8 进行编码的。

输入如下代码（新东西有点多，先敲完再说）：

`ex23.py`

```
1     import sys
2     script, encoding, error = sys.argv
3
4
5     def main(language_file, encoding, errors):
6         line = language_file.readline()
7
8         if line:
9             print_line(line, encoding, errors)
10        return main(language_file, encoding, errors)
11
12
13 def print_line(line, encoding, errors):
14     next_lang = line.strip()
15     raw_bytes = next_lang.encode(encoding, errors=errors)
16     cooked_string = raw_bytes.decode(encoding, errors=errors)
17
18     print(raw_bytes, "<===>", cooked_string)
19
20
21 languages = open("languages.txt", encoding="utf-8")
22
23 main(languages, encoding, error)
```

你肯定很好奇这个文件是用来干嘛的，可以运行它看看，以下是运行结果（注意运行时需要输入包括文件名在内的三个参数）：

```
python — bash — 80x24
$ python3.6 ex23.py utf-8 strict
b'Afrikaans' <====> Afrikaans
b'\xe1\x8a\xa0\xe1\x88\x9b\xe1\x88\xad\xe1\x8a\x9b' <====> አላጋጃ
b'\xd0\x90\xd2\xa7\xd1\x81\xd1\x88\xd3\x99\xd0\xb0' <====> Ἀρχαῖα
b'\xd8\xa7\xd9\x84\xd8\xb9\xd8\xb1\xd8\xa8\xd9\x8a\xd8\xa9' <====> العربية
b'V\xc3\xb5ro' <====> Võro
b'\xe6\x96\x87\xe8\xa8\x80' <====> 文言
b'\xe5\x90\xb4\xe8\xaf\xad' <====> 吴语
b'\xd7\x99\xd7\x99\xd6\xb4\xd7\x93\xd7\x99\xd7\xa9' <====> 𐤆𐤒𐤕
b'\xe4\xb8\xad\xe6\x96\x87' <====> 中文
$
```

### 警告！

我在这儿用了图片来展示你应该看到的内容。因为很多人的电脑不是用 UTF-8 来显示的，所以我得用图片来确保你知道我要呈现的是什麼。即使是我自己的 typesetting system (LaTeX) 也处理不了这些编码，迫使我必须使用图片。如果你看不到这些，很可能是你的终端没有用 UTF-8 来显示，你得想想办法。

这些例子用了 utf-8 、utf-16 和 big5 编码来说明这种转换，以及你可能会遇到的错误类型。这些名字在 Python 3 中被称为“codec”（编码器），但是你要用参数“encoding”。在这个练习的最后我列出了一个可用的编码表（encodings）以便你进行更多的练习（PDF文件里貌似没有这个编码表欸，找到的童鞋吱一声，大家也可以到网上查查）。我会在随后讲到这些东西的含义。你只用知道这些东西是如何工作的，这样我们就能在后面提及以及用到它们。

当你运行过几次之后，复习一遍你的符号列表，猜猜它们是做什么的，写下来。然后到网上找找它们的用法，是否跟你猜的一样。别担心你查不到，试试看。

## 开关、惯例（conventions）和编码

在我深入讲解这些代码的含义之前，你需要学习一些关于数据是如何存储在计算机中的基本知识。现代计算机非常复杂，但是核心就是大量的电灯开关。计算机用电来切换开关。这些开关可以以“开”代表 1，以“关”代表 0。以前有各种各样奇怪的计算机做的不只是 1 和 0 的事情，但现在所有的计算机都是一堆 1 和 0。1 代表着运行、有电、开着、进行、存在。0 代表着结束、完成、消失、关机、没电。我们把这些 1 和 0 叫做“比特”（bits）。

但是，一个只能让你用 1 和 0 操作的计算机将会非常低效和无聊。计算机接收了这些 1 和 0 之后，会用它们来编码更大的数字，比如用 8 个 1 和 0 来编码 256 个数（0-255）。那么编码到底是什么意思？它其实就是一个关于比特序列如何表示数字的公认标准，比如人们约定 00000000 就代表数字 0，11111111 就代表数字 255，00001111 就代表数字 15。即便是在计算机诞生早期的世界级战争中，计算机也是用这些约定的 1 和 0 来做大规模计算的。

现在我们把一个“字节”（byte）称为 8 个比特（1 和 0）的序列。过去每个人都有他们自己对于字节的惯例（convention），所以你还是会遇到一些人说，这项规定应该灵活一些，比如可以是 9 个、7 个或者 6 个字节序列。但是现在我们都说一个字节是 8 个比特，这是我们的惯例，它定义了我们对于字节的编码。当然还有用 16、32、64 甚至更多个比特来给字节编码的。

一旦你有了字节，你就可以开始存储和显示文本了，不过要用另一种惯例来让数字映射（map）成文字。在计算机发展的早期，有很多关于映射的惯例，有 8 个比特的，7 个比特的（或者更多或更少）。但是最终美国信息交换标准编码（即 ASCII 码）成为最流行的惯例。这个标准建立了从一个数字到一个字母的映射，比如 90 是 Z，用比特的话就是 1011010，对应到计算机里面的 ASCII 码表。

你可以在 Python 里面试试这个（Windows 系统下在 Powershell 输入 `python`，然后回车，就会出现 `>>>`，MacOS 输入 `Python3.6`）：

```
>>> 0b1011010
90
>>> ord ( ' Z ' )
90
>>> chr ( 90 )
' Z '
>>>
```

首先，我用二进制写了数字 90，然后我基于字母 'Z' 得到了对应的数字，接着我把这个数字转化成字母 'Z'。你不用记这些内容，我用 python 用了这么长时间好像只写过两次这个东西。

一旦我们有了 ASCII 惯例来用 8 个比特（即一个字节）给一个字符编码，我们就可以把它们“串”（string）在一起组成单词。比如如果我想写我的名字“Zed A. Shaw”，我只需要用 [90, 101, 100, 32, 65, 46, 32, 83, 104, 97, 119] 这样一系列字节就行了。大多数计算机上的早期文本都是存在存储器里的字节序列，计算机用它们把文字呈现给人看。同样的，这件事情的本质还是是一些约定俗成的开关转换。

不过 ASCII 有一个问题，它只能编码英文以及一些相似的语言，而且一个字节只能表示 256 个数字（0-255，或者 00000000-11111111）。很显然，世界上正在使用的语言远远超过 256 个

字符。因此不同国家创建了针对他们自己语言的编码惯例，虽然这些都管用，但是它们只适用一种语言。这就意味着，如果你想把一本英语书的书名放在一个泰语句子中，就会比较麻烦，你就需要一个泰语编码和一个英语编码。

为了解决这个问题，一群人创建了 Unicode，也就是针对所有人类语言的“统一编码”（Universal encoding）。Unicode 提供的解决方案跟 ASCII 码表类似，但是相比之下，前者更大。你可以用 32 个比特来编码一个 Unicode 字符，这比我们能找到的所有字符可能都要多。一个 32 位比特的数字意味着我们可以存储 4,294,967,295 个字符（ $2^{32}$ ），这对任何一种人类语言，甚至外星语言来说，都够用了。现在我们用多余的空间来表示一些重要的东西，比如 emoji 表情。

我们现在有了针对任何字符的编码协定，但是 32 比特是 4 个字节，这就意味着对于大多数我们想要编码的文本会浪费很多空间。我们也可以使用 16 比特（2 个字节），但仍然很浪费。因此后来出现了一种很妙的惯例：用 8 个比特来编码大多数通用字符，然后当我们需要编码更多字符的时候再使用更多的数字。这意味着我们有了一种压缩（compression）编码惯例，使得用 8 个比特来编码大多数常用字符，并在需要时切换成 16 或 32 个比特这件事成为可能。

在 Python 中编码文本的惯例叫做“utf-8”，即“Unicode Transformation Format 8 Bits”，它是一个把 Unicode 字符编码成字节序列（字节即比特序列，比特序列又即开关转换序列）的惯例。你也可以用其他编码惯例，但是 utf-8 是目前的标准。

## 分解输出结果

我们现在可以再看一下上面命令的输出结果。先看前面几行结果：

```
python — bash — 80x24
$ python3.6 ex23.py utf-8 strict
b'Afrikaans' <====> Afrikaans
b'\xe1\x8a\xa0\xe1\x88\x9b\xe1\x88\xad\xe1\x8a\x9b' <====> አሞላኛ
b'\xd0\x90\xd2\xa7\xd1\x81\xd1\x88\xd3\x99\xd0\xb0' <====> Ἀρχαῖα
b'\xd8\xa7\xd9\x84\xd8\xb9\xd8\xb1\xd8\xa8\xd9\x8a\xd8\xa9' <====> العربية
b'V\xc3\xb5ro' <====> Võro
b'\xe6\x96\x87\xe8\xa8\x80' <====> 文言
b'\xe5\x90\xb4\xe8\xaf\xad' <====> 吴语
b'\xd7\x99\xd7\x99\xd6\xb4\xd7\x93\xd7\x99\xd7\xa9' <====> བློ་གྲོ་བ་
b'\xe4\xb8\xad\xe6\x96\x87' <====> 中文
$
```

ex23.py 脚本其实就是把字节写在 `b' '` 里面，然后把它们转换成 UTF-8 编码（或者其他你设定的编码）。左边是每一个 utf-8 字节对应的数字，右边是 utf-8 实际输出的字符。之所以这样

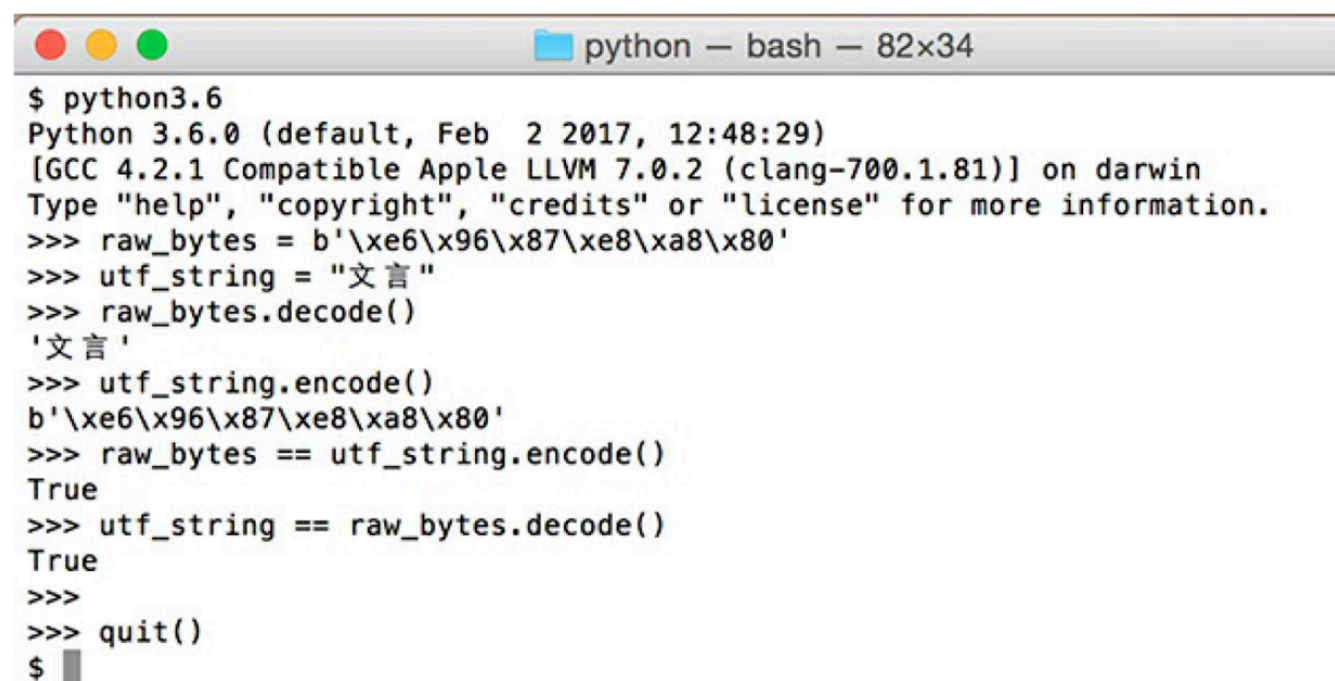


呈现，是为了让你明白 `<===>` 左边是 Python 用来存储字符串的数字字节或者“原始”（raw）字节，设置 `b' '` 是为了告诉 Python 这是“字节”（bytes）。这些原始字节之后被“加工”（cooked）然后显示在右边，以便让你看到你的终端呈现出来的真正的字符。

## 分解代码

我们已经对字符串和字节序列有了一定的理解。在 Python 中，一个字符串就是一个 UTF-8 编码的字符序列，用来显示或者进行文本操作。而字节是一些“原始”的字节序列，Python 用来存储这些 UTF-8 字符串并以 `b'` 开头来告诉 python 你正在处理原始字节。这些都基于 python 对于文本操作的惯例。

这是一个关于如何编码字符串和解码字节的 Python 会话展示：



```
python — bash — 82x34
$ python3.6
Python 3.6.0 (default, Feb  2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> raw_bytes = b'\xe6\x96\x87\xe8\xa8\x80'
>>> utf_string = "文言"
>>> raw_bytes.decode()
'文言'
>>> utf_string.encode()
b'\xe6\x96\x87\xe8\xa8\x80'
>>> raw_bytes == utf_string.encode()
True
>>> utf_string == raw_bytes.decode()
True
>>>
>>> quit()
$
```

你需要做的就是记住如果你有原始字节，那你必须用 `.decode()` 来获取字符串。原始字节没有相关惯例，它们只是一些没有意义的数字组成的字节序列。所以你必须告诉 Python“把这些解码成 utf 字符串”。

如果你有一个字符串，并且想要发送、保存、分享它，或者对它做一些其它的操作，通常情况下都可行，但是有时 Python 会扔出一个错误说它不知道如何编码。其实，Python 知道它内部的惯例，它只是不知道你需要的是哪个。在这种情况下，你必须用 `.encode()` 来获取你需要的字节。

记住这些方法（虽然我其实都是每次要用才查的）就是记住“DBES”这个记忆符号，它代表“Decode Bytes Encode Strings”（解码字节，编码字符串），当你思考如何转换字节和字符串的时候，可以在脑子里默念“迪拜斯”（DBES 发音），有字节要字符串，解码字节，有字符串要字节，编码字符串。

把这个放进脑子里之后，咱们来一行一行分解一下 `ex23.py` 的代码：

### ex23.py

```
1     import sys
2     script, encoding, error = sys.argv
3
4
5     def main(language_file, encoding, errors):
6         line = language_file.readline()
7
8         if line:
9             print_line(line, encoding, errors)
10        return main(language_file, encoding, errors)
11
12
13 def print_line(line, encoding, errors):
14     next_lang = line.strip()
15     raw_bytes = next_lang.encode(encoding, errors=errors)
16     cooked_string = raw_bytes.decode(encoding, errors=errors)
17
18     print(raw_bytes, "<====>", cooked_string)
19
20
21 languages = open("languages.txt", encoding="utf-8")
22
23 main(languages, encoding, error)
```

**第 1-2 行：**以通常的命令行参数开始，这个你已经学过了。

**第 5 行：**我把这段代码的主体部分定义为一个叫“main”的函数，这个函数会在脚本最后运行东西的时候被调用。

**第 6 行：**这个函数所做的第一件事就是从给出的 languages 文件中读取一行。你之前已经做过这个操作了，所以这儿没什么新内容，就像以前一样读取文本文件即可。

**第 8 行：**现在我用了一些新东西。你将会在这本书的后半部分学到相关内容，所以把这里当作一个尝鲜吧。这是一个 if 语句，它让你在 Python 代码中做决定。你可以“测试”一个变量的真



假，基于其真假，运行或者不运行这段代码。在本例中，我测试了一行中是否有内容。当 `readline` 函数到达文件末尾的时候，它会返回空字符串，if 这一行就是为了测试这个空字符串。只要 `readline` 给了我们一些东西，结果就会是 `true`，后面的代码就会运行（比如缩进的 9-10 行），当结果是 `false` 的时候，python 就会跳过 9-10 行。

**第 9 行：**然后我调用了一个单独的函数来做这一行的真正打印。这简化了我的代码，并且让我更容易理解。如果我想学习这个函数的作用，我可以跳到那儿进行学习。一旦我知道了 `print_line` 是做什么的，我就可以把我的记忆附到 `print_line` 这个名称下，然后忘掉细节。

**第 10 行：**我在这儿写了一小段非常神奇的代码。我在 `main` 函数内部又调用了 `main` 函数。其实也不神奇，因为在编程里面没有真正神奇的东西，所有你需要的信息都在那儿。这里我在一个函数里面又调用了它，好像看上去不太合理。但是问问你自己，为什么不合理？其实没有技术原因，如果一个叫 `main` 的函数只是跳到顶部，而我在这个函数的底部调用它，它就会回到顶部然后再次运行，这样就会形成一个循环（loop）。现在看第 8 行，你会看到 if 语句避免了这个函数无限循环。仔细研究研究这块内容，因为它是一个很重要的概念，不过如果你一下子理解不了也不用担心。

**第 11 行** 现在我开始定义 `print_line` 函数，它用来编码 `languages.txt` 文件中的每一行内容。

**第 13 行** 现在我终于获得了从 `languages.txt` 中收到的语言，并把它们编码成原始字节。还记得“DBES”这个辅助记忆词吗？“Decode Bytes, Encode Strings”，解码字节，编码字符串。`next_lang` 变量是一个字符串，因此要获得原始字节，我必须对它调用 `.encode()` 函数来“编码字符串”。我把我想要的编码以及如何处理错误传递给 `encode()`。

**第 14 行** 然后我做了额外一步，通过从 `raw_bytes` 创建一个 `cooked_string` 变量来逆向展示第 15 行。记住，“DBES”说的是“解码字节”，`raw_bytes` 是字节，所以我对它调用了 `.decode()` 来获取一个 python 字符串。这个字符串应该和 `next_lang` 变量是一样的。

**第 15 行** 我已经定义完了所有函数，现在我想打开 `languages.txt` 文件。

**第 16 行** 在这个脚本的结尾只是用所有正确的参数运行了 `main` 函数，以保证一切正常运行，避免循环。记住这个之后会跳转到第 5 行 `main` 函数被定义的地方，然后在第 10 行又被调用了一次，会造成它的循环。不过第 8 行的 if 语句又会阻止它无限循环。

## 深入了解编码

我们现在可以用我们的小脚本去探索其他的编码。下面是我针对其他不同编码所做的一些操作，

看看如何分解它们：

（注：英文版PDF中这里貌似把图片贴错了，贴的还是前面“分解代码”那一部分的图，所以这里大家就开下脑洞自己想象吧=.=）

首先，我做了一个简单的 UTF-16 编码，以便你了解和 UTF-8 比起来，它是如何变换的。你也可以用“utf-32”来看看它有多大，以及如何用 UTF-8 来节省空间。之后我尝试了 Big5，你会看到 Python 一点儿也不喜欢它。它扔了一个错误说 'big6' 无法在位置 0 上解码部分字符（**我也不知道为什么是 6**）。一个办法是告诉 Python 取代 Big5 编码下任何不好搞的字符。这是下一个例子，你会看到它在任何无法匹配 Big5 编码的地方放了一个 `?` 符号。

## 拆解

你可以试试做以下事情：

1. 找到其他编码方式编码的文本字符串，然后把它们放到 `ex23.py` 文件中，看它如何分解。
2. 给一个不存在的编码方式，看看会发生什么。
3. 额外挑战：重新用 `b''` 字节来写，不用 UTF-8 字符串，有效转换这个脚本。
4. 如果你可以做到上面这个，你也可以把这些字节通过移除部分的方式分解开，看看会发生什么。你需要移除多少来让 Python 分解？你能够移除多少来破坏字符串输出结果但是又能通过 Python 的解码系统。
5. 用你在第 4 点中学到的东西看看你能不能破坏这个文件。你得到的错误信息是什么？你能在文件通过 Python 的解码系统下带来多少破坏？

## 练习 24 更多练习

快到这部分的尾声了。现在你应该已经有了足够多的 Python 技能来继续学习编程到底是怎么回事儿，但是你应该做更多的练习，这个练习很长，纯粹是为了夯实基础。好好练吧，准确输入，仔细检查。

[ex24.py](#)

```

1     print("Let's practice everything.")
2     print('You\'d need to know \'bout escapes with \\ that do:')
3     print('\n newlines and \t tabs.')
4
5     poem = """
6         \tThe lovely world
7 with logic so firmly planted
8         cannot discern \n the needs of love
9         nor comprehend passion from intuition
10        and requires an explanation
11        \n\t\twhere there is none.
12    """
13
14    print("-----")
15    print(poem)
16    print("-----")
17
18
19    five = 10 - 2 + 3 - 6
20    print(f"This should be five: {five}")
21
22    def secret_formula(started):
23        jelly_beans = started * 500
24        jars = jelly_beans / 1000
25        crates = jars / 100
26        return jelly_beans, jars, crates
27
28
29    start_point = 10000
30    beans, jars, crates = secret_formula(start_point)
31
32    # remember that this is another way to format a string
33    print("With a starting point of: {}".format(start_point))
34    # it's just like with an f"" string
35    print(f"We'd have {beans} beans, {jars} jars, and {crates} crates.")
36
37    start_point = start_point / 10
38
39    print("We can also do that this way:")
40    formula = secret_formula(start_point)
41    # this is an easy way to apply a list to a format string
42    print("We'd have {} beans, {} jars, and {} crates.".format(*formula))

```

# 你会看到

## 练习 24 会话

```
$ python3.6 ex24.py
Let's practice everything.
You'd need to know 'bout escapes with \ that do:

    newlines and         tabs.
-----

    The lovely world
with logic so firmly planted cannot discern
    the needs of love
nor comprehend passion from intuition
and requires an explanation

        where there is none.
-----
This should be five: 5
With a starting point of: 10000
We'd have 5000000 beans, 5000.0 jars, and 50.0 crates.
We can also do that this way:
We'd have 500000.0 beans, 500.0 jars, and 5.0 crates.
```

## 附加练习

1. 确保你做了检查：从后往前读代码，大声读出来，然后在不明白的地方加上注释。
2. 有意打乱这个文件，然后运行，看看你会收到什么样的错误信息，确保你能把它修复好。

## 常见问题

**为什么你给变量叫 `jelly_beans` 但是后面又用的是 `beans` 这个名字？** 这是函数如何运行的一部分。记住，在函数内部变量是暂时的。当你返回它的时候，它可以被分配给一个变量以便之后使用。我只是创建了一个新的变量 `beans` 来保存返回的值。

**你说的从后往前读代码是什么意思？** 从最后一行开始，把你的代码跟我的代码进行比较，如果完全一样，就转到上一行，直到你检查到这个文件的第一行。

这首诗是谁写的？是我，其实我写的 not 全是烂诗。

## 练习 25 更多练习

接下来我们要做更多包含函数和变量的练习，来确保你完全掌握这些东西。这个练习你应该能直接输入、拆解并理解。

不过，这个练习有一些不同，你不是运行它，而是要把它导入 Python 然后自己运行这个函数。

[ex25.py](#)

```

1     def break_words(stuff):
2         """This function will break up words for us."""
3         words = stuff.split(' ')
4         return words
5
6     def sort_words(words):
7         """Sorts the words."""
8         return sorted(words)
9
10    def print_first_word(words):
11        """Prints the first word after popping it off."""
12        word = words.pop(0)
13        print(word)
14
15    def print_last_word(words):
16        """Prints the last word after popping it off."""
17        word = words.pop(-1)
18        print(word)
19
20    def sort_sentence(sentence):
21        """Takes in a full sentence and returns the sorted words
22        words = break_words(sentence)
23        return sort_words(words)
24
25    def print_first_and_last(sentence):
26        """Prints the first and last words of the sentence."""
27        words = break_words(sentence)
28        print_first_word(words)
29        print_last_word(words)
30
31    def print_first_and_last_sorted(sentence):
32        """Sorts the words then prints the first and last one."""
33        words = sort_sentence(sentence)
34        print_first_word(words)
35        print_last_word(words)

```

首先，用 `python3.6 ex25.py` 来运行这个脚本，找出你出错的地方，并把它们改正过来。然后对照“你会看到”部分看看运行结果是否一样。

## 你会看到

在这个练习中我们要在 python3.6 翻译器（interpreter）里与 `ex25.py` 文件交互，之前我们在做计算的时候也交互过。你在终端里这样运行 python3.6（Windows 下直接输入 `python`）：

```
$ python3.6
Python 3.6.0rc2 (v3.6.0rc2:800a67f7806d, Dec 16 2016, 14:12:21)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help " , "copyright" , "credits" or "license" for more info
>>>
```

你的输出结果应该和我的一样，你可以在提示符（即 `>` ）后面输入 Python 代码，它会直接运行。我希望你用这种方式输入这个练习的每一行代码，然后看看会如何：

## 练习 25 会话

```
1      import ex25
2      sentence = "All good things come to those who wait."
3      words = ex25.break_words(sentence)
4      words
5      sorted_words = ex25.sort_words(words)
6      sorted_words
7      ex25.print_first_word(words)
8      ex25.print_last_word(words)
9      words
10     ex25.print_first_word(sorted_words)
11     ex25.print_last_word(sorted_words)
12     sorted_words
13     sorted_words = ex25.sort_sentence(sentence)
14     sorted_words
15     ex25.print_first_and_last(sentence)
16     ex25.print_first_and_last_sorted(sentence)
```

以下是交互模式下输入的结果：

## 练习 25 Python 会话

```

Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwi
Type "help", "copyright", "credits" or "license" for more informa
>>> import ex25
>>> sentence = "All good things come to those who wait."
>>> words = ex25.break_words(sentence)
>>> words
['All', 'good', 'things', 'come', 'to', 'those', 'who', 'wait.']
>>> sorted_words = ex25.sort_words(words)
>>> sorted_words
['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
>>> ex25.print_first_word(words)
All
>>> ex25.print_last_word(words)
wait.
>>> words
['good', 'things', 'come', 'to', 'those', 'who']
>>> ex25.print_first_word(sorted_words)
All
>>> ex25.print_last_word(sorted_words)
who
>>> sorted_words
['come', 'good', 'things', 'those', 'to', 'wait.']
>>> sorted_words = ex25.sort_sentence(sentence)
>>> sorted_words
['All', 'come', 'good', 'things', 'those', 'to', 'wait.', 'who']
>>> ex25.print_first_and_last(sentence)
All
wait.
>>> ex25.print_first_and_last_sorted(sentence)
All
who

```

当你过完每一行，保证你能找到在 `ex25.py` 中运行的函数，并且理解了每个函数是如何运行的。如果你得到了不同的结果或者出现错误，你得把代码改正过来，然后退出 `python3.6`，重新进入。

## 附加练习

1. 弄明白“你会看到”中各行的作用是什么，确保你理解你是如何在 `ex25` 模块中运行你的函数的。
2. 试试输入 `help(ex25)` 以及 `help(ex25.break_words)`（要在交互练习后输入，否则



无法成功运行)。注意你是如何获取到关于这个模块的帮助的，以及帮助是如何放在 ex25 的每一个函数后面的 `"""` 字符串里的。这些特殊的字符串被称为文件注释，我们会在后面看到更多。

3. 输入 `ex25`。很无聊，可以走个捷径：`from ex25 import *`，意思就是从 `ex25` 导入所有东西，程序员总喜欢倒着说。打开一个新会话，看看你的函数会如何。
4. 试着拆解你的文件，看看当你用它的时候，它在 Python 里是什么样的。你得先输入 `quit()` 来退出 python，再重新加载它。

## 常见问题

有些函数我什么都没打印出来。你可能有些函数忘了在后面输入 `return`。检查一遍你的代码，确保每一行都是对的。

当我输入 `import ex25` 之后，我收到了 `-bash: import: command not found`。注意看“你会看到”部分我是怎么做的。我是在 Python 里面运行的，而不是在 Terminal，也就是说，你得先运行 Python。

当我输入 `import ex25.py` 时收到了这样的错误：`ImportError: No module named ex25`。不要在后面加 `.py`，Python 知道文件是以 `.py` 结尾的，所以你只用输入 `ex25` 即可。

我运行的时候遇到了这个错误：`SyntaxError: invalid syntax`。这意味着你漏掉了某些东西，比如少了一个 `"` 或者类似一对的符号。任何时候你只要收到这样的报错信息，你就从它提到的错的那行开始检查，看是不是所有字符都输入正确了，然后再回过头检查这一行上面的行是不是都输入正确了。

`words.pop` 函数是如何改变 `words` 变量的？这是个很复杂的问题，但是在本例中 `words` 是一个列表，正因为如此你可以给它一些命令。这就类似于当你操作文件和很多其他东西时候它们是如何运行的一样。

在函数里我什么时候应该用 `print` 而不是 `return` 呢？通过函数，`return` 能够给调用这个函数的那行代码返回一个结果，你可以把函数当成通过参数获取输入通过 `return` 返回输出。`print` 跟这个就完全不相关了，它只是把输出结果打印到终端。

# 练习 26 恭喜你，来做测试吧！

现在已经大体完成了这本书的一半内容，后面一半会更有意思，你会学习逻辑以及能够做一些有用的事情，比如做决定。

在你继续之前，我为你准备了一个小测验。这个小测验很难，因为它需要你修复别人的代码。当你成为一个程序员，你会经常需要处理其他程序员的代码，甚至跟他们正面杠。

有些程序员会宣称他们的代码很完美，这些人一般比较蠢，很少考虑别人的感受。好的程序员会像科学家一样，假设他们的代码总是存在一定概率是错的。好的程序员一般会在软件出现问题的情况下，用所有可能的方式排查自己会犯的错误，直到最后得出结论可能真的是其他人的代码出了问题。

在这个练习中，你将会通过修复一个烂程序员的代码来练习和他们打交道。我把练习 24 和 25 复制到了一个文件里，然后随机删除一些字符并加入一些错误。这些错误大多数 Python 都会告诉你，不过一些可能是计算错误，或者是一些字符格式或拼写错误，需要你自己发现。

这些错误都是程序员经常会犯的，哪怕是经验丰富的程序员。

在这个练习中，你的工作就是纠正这个文件。用你所学过的所有技能把这个文件变得更好。首先，你需要先分析一下这些代码，你可以把它打印出来，就像修改学校的学习论文一样编辑它。把其中每个错误都修改好，然后运行它，直到这个文件能够完美运行。试着不去寻求帮助。如果你卡住了，休息一下再回来。

这个练习的重点不在于输入，而在于修复好一个现有的文件。你需要去官网下载这个：

<https://learnpythonthehardway.org/python3/exercise26.txt>

把这些代码复制到一个文件中，并命名为 `ex26.py`。这是唯一一次允许你复制粘贴的地方。

## 常见问题

**我需要 `import ex25.py` 吗？或者我可以在里面引用它吗？** 都行。这个文件包含 `ex25` 中的函数，所以你可以先把它的引用补上。

**我在修复它的时候可以运行代码吗？** 你很多时候都需要这么做。计算机就是为了帮你的，所以尽可能多地使用它吧。

# 练习 27 记忆逻辑

今天你将开始学习逻辑（logic）。目前为止你已经完成了很可能让你会在终端读写文件的所有内容，并且还学习了相当多 Python 中的数学计算。

从现在开始，你将开始学习逻辑。你不会学习学院派喜欢教的那套复杂理论，而只是简单的基本逻辑，它们是一些能让真实的程序运行，以及真正的程序员每天都需要的东西。

学习逻辑需要你先做一些记忆工作。我希望能花一整个星期的时间来做这个练习。不要中途放弃，哪怕你已经烦闷不堪，坚持学下去。这个练习有一个逻辑表，你必须记住它们，这样在做后面的练习时才能更容易一些。

我得告诉你，这个练习一开始不会很有趣，甚至会冗长乏味，但是它会教给你作为程序员所需的一项重要技能。你需要记住这些重要的理念，当你掌握它们的时候，你会发现其中大多理念都非常令人激动。你会苦苦思索，就像跟章鱼搏斗（wrestling a squid），直到有一天你最终理解它们。所有记忆工作的付出会在之后给你丰厚的回报。

以下是一些让你不至于抓狂的记忆技巧：

一天少记一些内容，给需要强化记忆的部分做上标记；别妄图一坐坐两个小时一下子把这些表全记住，这不科学，你的大脑只会保留你前 15-30 分钟记的内容。相反，你应该创建一些索引卡，在表中左边列的内容（True or False）写在正面，右边的内容写在背面。然后你可以把它们拿出来，看着“True or False”直接说出“True！”保持练习，直到你能够做到这样。

一旦你能够做到，你就可以开始每天晚上在笔记本上默写 truth table。不要只是复制，试着凭记忆默写。如果你卡住了，快速看一眼来刷新你的记忆。这样做会训练你的大脑记住整张表。

别花超过一个星期时间在这上面，因为随着你往下学习，你会逐步应用这些内容。

## The Truth Terms

在 Python 中我们有下面这些词条（字符和短语）来判断某些东西是“True”还是“False”。计算机的逻辑就是看这些字符和变量的组合在特定程序和特定点下是不是 True。

- and
- or
- not

- != （不等于）
- == （等于）
- >= （大于等于）
- <= （小于等于）
- True
- False

你其实之前已经遇到过这些字符了，但是可能不是 terms。Terms （and, or, not）的运行方式就跟它们的意思一样。

## The Truth Tables

我们现在要用这些字符来帮你记忆 truth tables 的内容。

NOT	True?
not False	True
not True	False

OR	True?
True or False	True
True or True	True
False or True	True
False or False	False

AND	True?
True and False	False
True and True	True
False and True	False
False and False	False

NOT OR	True?
not (True or False)	False
not (True or True)	False
not (False or True)	False
not (False or False)	True

现在照着这些表来写下你自己的卡片，然后花一周的时间记忆它们。记住，这本书里没有失败，只有每天不断的尝试和坚持。

## 常见问题

我可以只学习布尔代数（boolean algebra）背后的理论而不去记忆这些内容吗？你当然可以这样做，但是之后你在写代码的时候就不得不地回顾布尔代数的那些规则了。如果你先把这些记住，不仅可以构建你的记忆技巧，还能让你的操作更自然。在这之后，布尔代数的理念就非常简单了。当然了，选择最适合你的方式吧。

## 练习 28 布尔练习

你上个练习所学的逻辑组合叫做“布尔”逻辑表达（Boolean logic expressions）。布尔逻辑在编程中无处不在。它是数学计算的基础模块，掌握它就跟掌握音乐里面的音阶一样重要。

在这个练习中，你将试着在 Python 中运用你在上个练习中所记忆的逻辑表。给以下每一个逻辑问题写下你认为的答案，要么是 True，要么是 False。等你把答案写下来，再在终端里运行 Python，输入每个逻辑问题，来确认你的答案是否正确。

```
1. True and True
2. False and True
3. 1 == 1 and 2 == 1
4. "test" == "test"
5. 1 == 1 or 2 != 1
6. True and 1 == 1
7. False and 0 != 0
8. True or 1 == 1
9. "test" == "testing"
10. 1 != 0 and 2 == 1
11. "test" != "testing"
12. "test" == 1
13. not (True and False)
14. not (1 == 1 and 0 != 1)
15. not (10 == 1 or 1000 == 1000)
16. not (1 != 10 or 3 == 4)
17. not ("testing" == "testing" and "Zed" == "Cool Guy")
18. 1 == 1 and (not ("testing" == 1 or 1 == 0))
19.      "chunky" == "bacon" and (not (3 == 4 or 3 == 3))
20.      3 == 3 and (not ("testing" == "testing" or "Python" == "Fun"))
```

我还会教你一个小诀窍来帮你弄明白更复杂的问题。

不论何时，当你看到这些布尔逻辑表达式，你可以通过以下简单的几步来解决它们：

1. 把每一个相等性测试（== 或者 !=）替换成真实性测试。
2. 先解决圆括号里面的 and/or。
3. 找到每一个 not，然后把它反转过来。
4. 找到剩余的 and/or，然后解决掉。
5. 当你完成的时候，你应该得到 True 或者 False。我会用一个变量来说明：

```
3 != 4 and not ("testing" != "test" or "Python" == "Python")
```

以下是我进行每一步逻辑运算的过程，最后我得出了一个单一的结果：

1、先解决每一个相等性测试：

```
3 != 4 是 True: True and not ("testing" != "test" or "Python" == "Python" ;
"testing" != "test" 是 True: True and not (True or "Python" == "Python") ;
"Python" == "Python": True and not (True or True) ;
```

2、找到圆括号里的每一个 and/or：

(True or True) 是 True: True and not (True)

3、找到每一个 not, 然后把它转换过来:

not (True) 是 False: True and False

4、找到其他剩余的 and/or 然后解决它们:

True and False 是 False。

这样我们就完成了这个测试, 并且知道结果是 False。

更复杂的测试可能一看非常难。你应该先试试, 不要一开始就气馁。我已经让你为做更难的“逻辑练习”做

## 你会看到

在你尝试给出所有答案后, 这是你可能会在 Python 运行后看到的会话结果:

```
$ python3.6
Python 2.5.1 (r251:54863, Feb 6 2009, 19:02:12)
[GCC 4.0.1 ( Apple Inc . build 5465)] on darwin
Type "help" , "copyright" , "credits" or "license" for more information
>>> True and True
True
>>> 1 == 1 and 2 == 2
True
```

## 附加练习

1. Python 中有很多类似于 `!=` 和 `==` 的运算符, 试着尽可能多地找到这类“比较运算符” (equality operators), 比如 `<` 或者 `<=`。
2. 写下这些比较运算符的名字, 比如我们把 `!=` 叫做“不等于”。
3. 在 Python 中输入新的布尔运算, 在你敲下回车之前先把答案说出来, 别思考, 说出你脑子里第一个冒出来的答案。写下来, 然后敲回车。算一算你对了多少, 错了多少。
4. 记完把纸扔掉, 防止你下次再用。

## 常见问题

**为什么 "test" and "test" 返回的是 test, 1 and 1 返回的是 1 而不是 True？** Python 和其他很多语言喜欢返回布尔表达式的运算数而不是只是 True 或者 False。这意味着，如果是 False and 1，你会得到第一个运算数（False），如果是 True and 1，你会得到第二个运算数（1），试着玩玩这个。

**!= 和 <> 有区别吗？** Python 已经不提倡使用 <>，而是更多地使用 !=，除此之外，二者没有任何区别。

**有捷径吗？** 有，任何包含一个 False 的 and 表达式结果都是 False。任何包含一个 True 的 or 表达式结果都是 True。但是你要掌握处理整个表达式的过程，后面会用到。

## 练习 29 if 语句

这个练习中的 Python 脚本将带你了解 if 语句。输入代码，准确运行，然后我们来看看你都学到了什么。

[ex29.py](#)



```
1     people = 20
2     cats = 30
3     dogs = 15
4
5
6     if people < cats:
7         print("Too many cats! The world is doomed!")
8
9     if people > cats:
10        print("Not many cats! The world is saved!")
11
12 if people < dogs:
13     print("The world is drooled on!")
14
15 if people > dogs:
16     print("The world is dry!")
17
18
19     dogs += 5
20
21 if people >= dogs:
22     print("People are greater than or equal to dogs.")
23
24 if people <= dogs:
25     print("People are less than or equal to dogs.")
26
27
28 if people == dogs:
29     print("People are dogs.")
```

## 你会看到

### 练习 29 会话

```
$ python3.6 ex29.py
Too many cats! The world is doomed!
The world is dry!
People are greater than or equal to dogs.
People are less than or equal to dogs.
People are dogs.
```

## 附加练习

在附加练习中，试着猜猜 if 语句是什么以及它是干什么的。在继续进行下个练习之前，试着用自己的话回答以下这些问题，

1. 你认为 if 对它下面的代码起什么作用？
2. 为什么 if 下面的代码要缩进 4 个空格？
3. 如果没有缩进会发生什么？
4. 你能从练习 27 里面把一些布尔表达式放进 if 语句吗？试试看。
5. 如果你改变 people, cats 和 dogs 的初始值会发生什么？

## 常见问题

`+=` 是什么意思？`x += 1` 就相当于 `x = x + 1`，但是输入的内容更少。你可以把它叫做“累加”（increment by）运算符。之后你还会学到 `-=` 这样类似的表达。

## 练习 30 Else 和 if

在上个练习中你学到了一些 if 语句，思考了它的含义和作用。在你学习更多内容之前，我会解释一下上个附加练习中的问题。首先确定你做了那些练习。

### 1. 你认为 if 对它下面的代码起什么作用？

if 语句在代码中创建了一个“分支”（branch），有点类似于在一本冒险书中，你选择了哪个答案，就翻到对应的一页，如果你选择了不同的答案，就会去到不同的地方。if 语句就是告诉脚本，如果这个布尔表达式是 True，那就运行它下面的代码，否则的话就跳过。

### 2. 为什么 if 下面的代码要缩进 4 个空格？

通过一行代码结尾的冒号告诉 Python 你在创建一个新的代码块，然后缩进四个空格告诉 Python 这个代码块中都有些什么。这就跟本书前半部分中你学的函数是一样的。

### 3. 如果没有缩进会发生什么？

如果没有缩进，你很可能收到一个错误提示。Python 一般会让你在一个带 `:` 的代码行下面缩进一些内容。

#### 4. 你能从练习 27 里面把一些布尔表达式放进 if 语句吗？试试看。

试试吧，你可以的。你可以把它们写得很复杂，不过复杂的东西一般风格都很糟糕。

#### 5. 如果你改变 people, cats 和 dogs 的初始值会发生什么？

因为你在比较数字，所以如果你改变了数字，不同的 if 语句将会得出不同的判断结果，那么下面某些代码块就有可能运行。回到练习中给这些变量一些不同的数值，然后看看你能否在脑中判断出来哪些代码块会运行。

把我的答案和你的比较一下，然后确保你真的理解了代码块的概念。这对你进行接下来的练习很重要。把下面的代码输入进去然后运行。

ex30.py

```
1     people = 30
2     cars = 40
3     trucks = 15
4
5
6     if cars > people:
7         print("We should take the cars.")
8     elif cars < people:
9         print("We should not take the cars.")
10    else:
11        print("We can't decide.")
12
13    if trucks > cars:
14        print("That's too many trucks.")
15    elif trucks < cars:
16        print("Maybe we could take the trucks.")
17    else:
18        print("We still can't decide.")
19
20    if people > trucks:
21        print("Alright, let's just take the trucks.")
22    else:
23        print("Fine, let's stay home then.")
```

## 你会看到

### 练习 30 会话

```
$ python3.6 ex30.py
We should take the cars.
Maybe we could take the trucks.
Alright, let's just take the trucks.
```

## 附加练习

1. 试着猜猜 `elif` 和 `else` 的作用是什么。
2. 改变 `cars`, `people`, 和 `trucks` 的数值, 然后追溯每一个 `if` 语句, 看看什么会被打印出来。
3. 试试一些更复杂的布尔表达式, 比如 `cars > people` 或者 `trucks < cars`。
4. 在每一行上面加上注释。

## 常见问题

如果多个 **`elif` 块都是 `True` 会发生什么**？Python 从顶部开始, 然后运行第一个是 `True` 的代码块, 也就是说, 它只会运行第一个。

## 练习 31 做决定

在这本书的前半部分你主要学习了调用函数、打印东西, 但是这些基本都是直线运行下来的。你的脚本从上面开始运行, 然后到底部结束。如果你用了一个函数, 你可以随后再运行它, 但是仍然不会有分叉需要你做决定的情况。现在你学习了 `if`, `else`, 以及 `elif`, 你就可以让脚本来做决定了。

在上个脚本中你写出了一个简单的问问题的测试集。在这个练习中你将问用户一些问题, 并基于他们的回答做决定。写下这个脚本, 然后多玩几遍, 把它弄明白。

[ex31.py](#)

```

1     print("""You enter a dark room with two doors.
2     Do you go through door #1 or door #2?""")
3
4     door = input("> ")
5
6     if door == "1":
7         print("There's a giant bear here eating a cheese cake.")
8         print("What do you do?")
9         print("1. Take the cake.")
10        print("2. Scream at the bear.")
11
12        bear = input("> ")
13
14        if bear == "1":
15            print("The bear eats your face off. Good job!")
16        elif bear == "2":
17            print("The bear eats your legs off. Good job!")
18        else:
19            print(f"Well, doing {bear} is probably better.")
20            print("Bear runs away.")
21
22    elif door == "2":
23        print("You stare into the endless abyss at Cthulhu's retina.")
24        print("1. Blueberries.")
25        print("2. Yellow jacket clothespins.")
26        print("3. Understanding revolvers yelling melodies.")
27
28        insanity = input("> ")
29
30        if insanity == "1" or insanity == "2":
31            print("Your body survives powered by a mind of jello.")
32            print("Good job!")
33        else:
34            print("The insanity rots your eyes into a pool of muck.")
35            print("Good job!")
36
37    else:
38        print("You stumble around and fall on a knife and die. Good job!")

```

这里很关键的一点是你现在在 if 语句里面又放了一个 if 语句。这在创建“嵌套”（nested）决定的时候非常有用，每一个分支指向另一个选择。

确保你理解了在 if 语句中嵌套 if 语句的理念。你可以通过做附加练习来真正掌握它。

# 你会看到

这是我玩这个冒险小游戏的结果，我可能玩儿得没那么好。

## 练习 31 会话

```
$ python3.6 ex31.py
You enter a dark room with two doors.
Do you go through door #1 or door #2?
> 1
There's a giant bear here eating a cheese cake.
What do you do?
1. Take the cake.
2. Scream at the bear.
> 2
The bear eats your legs off. Good job!
```

## 附加练习

1. 给这个游戏加一些新内容，同时改变用户可以做的决定。尽可能地扩展这个游戏，直到它变得很搞笑。
2. 写一个完全不同的新游戏。可能你不喜欢我的这个，你可以做一个你自己的。

## 常见问题

**我能用一系列的 if 语句来代替 elif 吗？** 在某些情况下可以，但是取决于每个 if/else 是怎么写的。如果这样的话还意味着 Python 将会检查每一个 if-else 组合，而不是像 if-elif-else 组合那样只会检查第一个是 false 的。你可以多试几次，感受一下区别。

**我如何表示一个数字的区间？** 有两种方式：一种是  $0 < x < 10$  或者  $1 \leq x < 10$  这种传统表示方法，另一种是 x 的区间是 (1, 10)。

**如果我想在 if-elif-else 代码块中放更多的选择怎么办？** 为每种可能的选择增加更多的 elif 块。

# 练习 32 循环和列表

你现在可以做一些更有意思的程序了。如果你一直跟着我们的节奏，你应该可以把所有学过的东西用 if 语句和布尔表达式结合起来，让你的程序做一些好玩的事情。

不过，程序仍然需要快速做一些重复的事情。我们要在这个练习中用一个 for-loop 来创建和打印各种列表。你在做练习的时候，得想想它们是什么。我不会立马告诉你，你得自己去想。

在你能够用一个 for-loop 之前，你需要一种方法来把这些循环的结果储存在某处。最好的办法就是用列表。列表顾名思义就是一个按顺序从头到尾组成的某种东西的容器。它并不复杂：你只需要学习一个新的语法。首先，你可以这样创建列表：

```
hairs = ['brown', 'blond', 'red']
eyes = ['brown', 'blue', 'green'] weights = [1, 2, 3, 4]
```

以左方括号（[）开始打开列表，然后把你想要的条目用逗号隔开放进去，有点类似于函数的参数。最后，用右方括号（]）来表明列表的结束。Python 会选取这个列表以及它的所有内容并把它们分配到变量里。

## 警告！

这块内容对于不会编程的人来说有点难理解，因为你的大脑一直以来都被训练成平面的了。还记得上个练习中的嵌套语句吗？这可能让你伤脑筋了，因为大多数人不理解如何在一个东西里面嵌套一个东西。在编程中，嵌套 if 语句的函数，这个 if 语句中又有一个包含列表的列表。如果你看到一个类似的结构无法理解，拿出一根笔和一张纸，然后手动模拟一下。

我们现在要用 for-loops 来创建一些列表，然后把它们打印出来。

[ex32.py](#)

```
1     the_count = [1, 2, 3, 4, 5]
2     fruits = ['apples', 'oranges', 'pears', 'apricots']
3     change = [1, 'pennies', 2, 'dimes', 3, 'quarters']
4
5     # this first kind of for-loop goes through a list
6     for number in the_count:
7         print(f"This is count {number}")
8
9     # same as above
10    for fruit in fruits:
11        print(f"A fruit of type: {fruit}")
12
13    # also we can go through mixed lists too
14    # notice we have to use {} since we don't know what's in it
15    for i in change:
16        print(f"I got {i}")
17
18    # we can also build lists, first start with an empty one
19    elements = []
20
21    # then use the range function to do 0 to 5 counts
22    for i in range(0, 6):
23        print(f"Adding {i} to the list.")
24    # append is a function that lists understand
25    elements.append(i)
26
27    # now we can print them out too
28    for i in elements:
29        print(f"Element was: {i}")
```

## 你会看到

### 练习 32 会话



```
$ python3.6 ex32.py
This is count 1
This is count 2
This is count 3
This is count 4
This is count 5
A fruit of type: apples
A fruit of type: oranges
A fruit of type: pears
A fruit of type: apricots
I got 1
I got pennies
I got 2
I got dimes
I got 3
I got quarters
Adding 0 to the list.
Adding 1 to the list.
Adding 2 to the list.
Adding 3 to the list.
Adding 4 to the list.
Adding 5 to the list.
Element was: 0
Element was: 1
Element was: 2
Element was: 3
Element was: 4
Element was: 5
```

## 附加练习

1. 看看你是如何使用 range 的。查阅上面的 range 函数并理解掌握。
2. 你能在第 22 行不使用 for-loop，而是直接把 range(0, 6) 赋给 elements 吗？
3. 找到 Python 文档关于列表的部分，然后读一读。看看除了 append，你还能对列表做哪些操作？

## 常见问题

如何创建一个二维列表？可以用这种列表中的列表：`[[1,2,3],[4,5,6]]`

**列表（lists）和数组（arrays）难道不是一个东西吗？** 这取决于语言以及实现方法。在传统术

语中，列表和数组的实现方式不同。在 Ruby 中都叫做 arrays，在 python 中都叫做 lists。所以我们就把这些叫做列表吧。

**为什么 `for-loop` 可以用一个没有被定义的变量？** 变量在 `for-loop` 开始的时候就被定义了，它被初始化到了每一次 loop 迭代时的当前元素中。

**为什么 `range(1, 3)` 中的 `i` 只循环了两次而不是三次？** `range()` 函数只处理从第一个到最后一个数，但不包括最后一个数，所以它在 2 就结束了。这是这类循环的通用做法。

**`element.append()` 的作用是什么？** 它只是把东西追加到列表的末尾。打开 Python shell 然后创建一个新列表。任何时候当你遇到类似的用法，试着多玩几次，去体会它们的作用。

## 练习 33 While 循环

现在我们来查看一个新的循环：`while-loop`。只要一个布尔表达式是 `True`，`while-loop` 就会一直执行它下面的代码块。

等等，你应该能理解这些术语吧？如果我们写一行以：结尾的代码，它就会告诉 Python 开始一个新的代码块。我们用这种方式来结构化你的程序，以便 Python 明白你的意图。如果你还没有掌握这块内容，先回去复习一下，再做一些 `if` 语句、函数以及 `for-loop`，直到你掌握为止。

之后我们会做一些练习来训练你的大脑读取这些结构，就像我们训练你掌握布尔表达式一样。

回到 `while-loop`，它所做的只是像 `if` 语句一样的测试，但是它不是只运行一次代码块，而是在 `while` 是对的方地方回到顶部再重复，直到表达式为 `False`。

但是 `while-loop` 有个问题：有时候它们停不下来。如果你的目的是让程序一直运行直到宇宙的终结，那这样的确很屌。但大多数情况下，你肯定是需要你的循环最终能停下来的。

为了避免这些问题，你得遵守一些规则：

1. 保守使用 `while-loop`，通常用 `for-loop` 更好一些。
2. 检查一下你的 `while` 语句，确保布尔测试最终会在某个点结果为 `False`。
3. 当遇到问题的时候，把你的 `while-loop` 开头和结尾的测试变量打印出来，看看它们在做什么。

在这个练习中，你要通过以下三个检查来学习 `while-loop`：

ex33.py

```
1     i = 0
2     numbers = []
3
4     while i < 6:
5         print(f"At the top i is {i}")
6         numbers.append(i)
7
8         i = i + 1
9         print("Numbers now: ", numbers)
10        print(f"At the bottom i is {i}")
11
12
13    print("The numbers: ")
14
15    for num in numbers:
16        print(num)
```

## 你会看到

练习 33 会话

```
$ python3.6 ex33.py
At the top i is 0
Numbers now: [0]
At the bottom i is 1
At the top i is 1
Numbers now:      [0, 1]
At the bottom i is 2
At the top i is 2
Numbers now:      [0, 1, 2]
At the bottom i is 3
At the top i is 3
Numbers now:      [0, 1, 2, 3]
At the bottom i is 4
At the top i is 4
Numbers now:      [0, 1, 2, 3, 4]
At the bottom i is 5
At the top i is 5
Numbers now:      [0, 1, 2, 3, 4, 5]
At the bottom i is 6
The numbers:
0
1
2
3
4
5
```

## 附加练习

1. 把这个 while-loop 转换成一个你可以调用的函数，然后用一个变量替代  $i < 6$  里面的 6。
2. 用这个函数重新写一下这个脚本，试试不同的数值。
3. 再增加一个变量给这个函数的参数，然后改变第 8 行的 +1，让它增加的值与之前不同。
4. 用这个函数重新写这个脚本，看看会产生什么样的效果。
5. 用 for-loop 和 range 写这个脚本。你还需要中间的增加值吗？如果不去掉这个增加值会发生什么？

任何时候你在运行程序的时候它失控了，只用按下 **CTRL-C**，程序就会终止。

# 常见问题

**for-loop 和 while-loop 的区别是什么？** for-loop 只能迭代（循环）一些东西的集合，而 while-loop 能够迭代（循环）任何类型的东西。不过，while-loop 很难用对，而你通常能够用 for-loop 完成很多事情。

**循环好难，我应该如何理解它们？** 人们不理解循环的主要原因是他们跟不上代码的运行。当一个循环运行的时候，它会过一遍代码块，到结尾之后再跳到顶部。为了直观表现这个过程，你可以用 print 打印出循环的整个过程，把 print 行写在循环的前面、顶部、中间、结尾。研究一下输出的内容，试着理解它是如何运行的。

## 练习 34 获取列表元素

列表（list）真的非常有用，前提是你要能获取到它们里面的内容。你已经能够按顺序遍历列表中的元素，但是如果你要取其中的第5个元素，你该怎么操操做？你需要知道如何获取一个列表里面的元素。下面是如何获取列表中第一个元素的方法：

```
animals = ['bear', 'tiger', 'penguin', 'zebra']
bear = animals[0]
```

你创建了一个动物列表，然后用 0 来取列表的第一个元素？！为什么呢？因为数学就是这样的，Python 列表的第一个元素是从序号 0 开始，而不是从 1 开始。这样虽然看起来有点奇怪，但是好处多多。

最好的解释可能是它反映了人使用数字和程序使用数字的区别。

想象一下你正在观察列表中的四个动物（熊，老虎，企鹅，斑马）赛跑，它们纷纷冲过了终点线，我们也得到了它们的次序。比赛非常激烈，因为最终它们都没有吃掉彼此。你的一个朋友来晚了，他想知道哪个动物胜出了。他肯定会说“谁得了第一名？”而不是“谁是第零个？”

这是因为动物们的次序非常重要。你不能在没有第一名的情况下就有第二名，同理没有第二名也不可能会有第三名。而第零名毫无意义，因为零意味着什么都没有。你怎么可能在异常比赛里面什么都没有呢？这说不通。我们把这些能排序的数字叫做序数（ordinal numbers），因为它们能代表一定的顺序。

然而，程序不会这么想。它们能从一个列表中任意取出一个元素来。对程序而言，动物们的列表

更像是一叠卡片。如果它们想要老虎，就直接去拿。如果想要斑马，也能直接去拿。这就需要这些元素能有一个恒定的地址（address），或者索引（index），以便程序能够以一种随机的方式把它们从列表中拿出来。最好的办法就是让指标（indices）从 0 开始。相信我，这样在数学上更为便捷。这种数字叫做基数（cardinal number），它意味着你可以随机取数，所以必须要有一个 0 元素。

**ai酱注：**这里感觉老肖没太解释清楚，在百度知道上看到一个回答，可以供大家参考：

<https://zhidao.baidu.com/question/1693009495708807428.html>

不知道以上解释能否帮助你理解列表？很简单，每次你对自己说，“我要第 3 个动物，”的时候，把 3 这个序数通过 -1 转换成 2 这个基数就行了。第 3 个动物就是索引为 2 的企鹅。你一生都在使用序数，现在你需要用基数来思考，只用减去 1 就行，没那么难。

记住：序数 == 排序，第一；基数 == 随机卡片，0。

( ordinal == ordered, 1st; cardinal == cards at random, 0. )

让我们练习一下。用如下动物列表，跟着我列出来的序数或基数要求，写出你从列表中取到的动物。记住，如果我说“第1个”（1st）、“第2个”（2nd），那我就是在用序数，直接减1就可以了。如果我说“第1位”（at 1），那我就是在用基数，直接按这个数字取就行。

```
animals = ['bear', 'python3.6', 'peacock', 'kangaroo', 'whale', ' '
```

（注：为了不造成混淆，以下内容保留英文原文）

1. The animal at 1.
2. The third (3rd) animal.
3. The first (1st) animal.
4. The animal at 3.
5. The fifth (5th) animal.
6. The animal at 2.
7. The sixth (6th) animal.
8. The animal at 4.

使用完整的表述格式进行回答，例如：“The first (1st) animal is at 0 and is a bear.” 然后反过来说一遍：“The animal at 0 is the 1st animal and is a bear.”

用 python 验证你的答案。

## 附加练习

1. 基于你所学的不同类型数字之间的区别, 你能解释为什么“2010年1月1日”中的2010年真的是2010年而不是2009年? (提示: 你不能随机去取年份)
2. 多写一些列表, 搞明白列表元素的索引, 知道你能够准确掌握。
3. 用 python 验证你的答案。

## 练习 35 分支和函数

目前为止你已经了解了 if 语句, 函数以及列表。现在是时候深入学习一下了。照例输入如下代码, 看看你能否明白程序在做什么。

[ex35.py](#)

```

1      from sys import exit
2
3      def gold_room():
4          print("This room is full of gold. How much do you take?")
5
6          choice = input("> ")
7          if "0" in choice or "1" in choice:
8              how_much = int(choice)
9          else:
10             dead("Man, learn to type a number.")
11
12             if how_much < 50:
13                 print("Nice, you're not greedy, you win!")
14                 exit(0)
15             else:
16                 dead("You greedy bastard!")
17
18
19     def bear_room():
20         print("There is a bear here.")
21         print("The bear has a bunch of honey.")
22         print("The fat bear is in front of another door.")
23         print("How are you going to move the bear?")
24         bear_moved = False
25
26         while True:
27             choice = input("> ")
28
29             if choice == "take honey":
30                 dead("The bear looks at you then slaps your face")
31             elif choice == "taunt bear" and not bear_moved:
32                 print("The bear has moved from the door.")
33                 print("You can go through it now.")
34                 bear_moved = True
35             elif choice == "taunt bear" and bear_moved:
36                 dead("The bear gets pissed off and chews your leg.")
37             elif choice == "open door" and bear_moved:
38                 gold_room()
39             else:
40                 print("I got no idea what that means.")
41
42
43     def cthulhu_room():
44         print("Here you see the great evil Cthulhu.")
45         print("He, it, whatever stares at you and you go insane.")
46         print("Do you flee for your life or eat your head?")

```



```
47
48     choice = input("> ")
49
50     if "flee" in choice:
51         start()
52     elif "head" in choice:
53         dead("Well that was tasty!")
54     else:
55         cthulhu_room()
56
57
58 def dead(why):
59     print(why, "Good job!")
60     exit(0)
61
62 def start():
63     print("You are in a dark room.")
64     print("There is a door to your right and left.")
65     print("Which one do you take?")
66
67     choice = input("> ")
68
69     if choice == "left":
70         bear_room()
71     elif choice == "right":
72         cthulhu_room()
73     else:
74         dead("You stumble around the room until you starve.")
75
76
77     start()
```

## 你会看到

以下是我玩这个游戏的结果：

练习 35 会话

```
$ python3.6 ex35.py
You are in a dark room.
There is a door to your right and left. Which one do you take?
> left
There is a bear here.
The bear has a bunch of honey.
The fat bear is in front of another door. How are you going to move the bear?
> taunt bear
The bear has moved from the door. You can go through it now.
> open door
This room is full of gold.          How much do you take?
> 1000
You greedy bastard! Good job!
```

## 附加练习

1. 画一个这个游戏的流程图，并指出它是如何运转的。
2. 修正你的错误，包括拼写和语法错误。
3. 为你不理解的函数写上注释。
4. 为游戏增加一些功能，同时使代码更加简化。
5. 这个 `gold_room` 让你输入数字的方式有点奇怪。这样做有哪些 bug？你能改善我的代码吗？可以查查看 `int()` 的相关知识。

## 常见问题

**救命！这个程序是怎么工作的！？** 当你遇到不理解的代码时，不要着急，只要在每行代码下面写下注释，弄清楚这一行是做什么的，就很容易明白。确保你的注释和代码一样简洁。然后要么画图，要么写一段话来描述代码是如何运行的。这样你就会理解其背后的原理。

**为什么你要用 `while True`？** 这样可以构建一个无限循环。

**`exit(0)` 是干什么用的？** 在很多操作系统中，一个程序可以用 `exit(0)` 来结束，其中传入的数字代表是否有错误。如果你用 `exit(1)` 代表有 1 个错误，`exit(0)` 则代表程序正常退出。它不同于通常的布尔逻辑 (`0==False`)，因为你可以用不同的数字来表示不同的错误结果。你可以用 `exit(100)` 来表示与 `exit(2)` 或者 `exit(1)` 不同的错误结果。

**为什么 `input()` 有时会被写成 `input('> ')`？** `input` 的参数是一个字符串，所以要在获取用户输入的内容前面加一个提示符。（ai酱注：这里 `>` 也可以换成想要提示用户的文字。）

# 练习 36 设计和调试

现在你已经非常了解 if 语句了，我会再教你一些 for 循环和 while 循环的规则，以免日后你遇到麻烦。我还会教你一些调试的小技巧，以便你能发现自己程序的问题。最后，你将需要设计一个和上节类似的小游戏，不过内容略有更改。

## if 语句的规则：

1. 每一个“if 语句”必须包含一个 else。
2. 如果这个 else 永远都不应该被执行到，因为它本身没有任何意义，那你必须在 else 语句后面使用一个叫做 die 的函数，让它打印出错误信息并且死给你看，就像我们上节课做的那样，按照这个思路你可以找到很多错误。
3. “if 语句”的嵌套不要超过 2 层，最好尽量保持只有 1 层。
4. 把“if 语句”当做段落来对待，其中的每一个 `if, elif, else` 就跟段落中的句子一样。在每句前后留一个空行以作区分。
5. 你的布尔测试应该很简单，如果它们很复杂的话，你需要将它们的运算事先放到一个变量里，并且为变量取一个好名字。

如果你遵循上面的规则，你就会写出比大多数程序员都好的代码来。回到上一个练习中，看看我给出的代码有没有遵循这些规则，如果没有的话，就将其改正过来。

**警告！**

在日常编程中不要成为这些规则的奴隶。在训练中，你需要通过应用这些规则来巩固你学到的知识，而在

## 循环的规则

1. 只有在循环永不停止时使用“while循环”，这意味着你可能永远都用不到。这条只有 Python 中成立，其他的语言另当别论。
2. 其他类型的循环都使用“for循环”，尤其是在循环的对象数量固定或者有限的情况下。

# 调试建议

1. 不要使用调试器（“debugger”）。调试器所做的相当于对病人的全身扫描。你并不会得到某方面的有用信息，而且你会发现它输出的信息太多，而且大部分没有用，或者只会让你更加困惑。
2. 最好的调试程序的方法是用 print 在每个你想要检查的关键环节将关键变量打印出来，从而检查那里是否有错。
3. 让程序一部分一部分地运行起来。不要等一个很长的脚本写完后才去运行它。写一点，运行一点，再修改一点。

# 课后作业

写一个和上节练习类似的游戏，同类的任何题材的游戏都可以。你可以花一个星期时间让它尽可能有趣一些。作为附加练习，你可以尽量多使用列表、函数、以及模组（还记得习题 13 吗？），而且尽量多写一些新的 Python 代码让你的游戏跑起来。

在写代码之前，你应该先把游戏的设计图画出来，包括玩家会碰到的房间、怪物、以及陷阱等环节。

一旦你搞定了地图，就可以开始写代码了。如果你发现地图有问题，就调整一下地图，让代码和地图互相符合。

写软件最好的方式是把它分解成很多小任务来完成：

1. 在一张纸或者索引卡片上，写上要完成这个软件你需要做的任务列表，这就是你的待办事项。
2. 从任务列表上找到对你来说最容易的一个任务。
3. 在你的源文件上用注释的方式写下你要用代码实现它的思路指南。
4. 在你的注释下面写代码。
5. 写完就执行脚本看看你的写的代码能否正常运行。
6. 坚持这样的过程：写代码、运行测试、修改代码，知道它能正常运行。
7. 完成后给这项任务打勾，然后找下一个最容易的任务，重复上述步骤。

这个过程能够帮助你以一种系统化和连贯的方式来写软件。在工作的过程中，通过移除不必要的任务以及添加新任务来更新你的任务列表。

# 练习 37. 复习各种符号

现在该复习你学过的符号和 python 关键字了，而且在接下来的几节里你还会学到一些新的东西。我已经把所有需要重点掌握的 Python 符号和关键字列出来了。

在这节课中，看到一个关键字，回忆并写下它的作用，然后上网搜它真正的用处。这里可能对你来说有些困难，因为有些内容真的很难收到，但是不管怎么样，还是要试一试。

如果你发现你的记忆有误，就在索引卡片上写下正确的定义，试着将自己的记忆纠正过来。

最后，在一个小的 Python 程序里使用每一个符号和关键字，或者你也可以尽量多写一些程序来练习。我们的目标是要明白各个符号的作用，确认自己没搞错，如果搞错了就纠正过来，然后将其用在程序里，通过这样的方式来巩固自己的记忆。

## 关键词

关键词	描述	示例
and	逻辑上的“和”	True and False == False
as	<i>with-as</i> 语句的一部分	with X as Y: pass
assert	断言某个表达式为 true（如果为 false，则会触发异常）	assert False. "Error!"
break	立即停止循环	while True: break
continue	不运行循环的剩余部分，重新开始循环	while True: continue
def	定义一个函数	def X(): pass
del	从字典中删除	del X[Y]
elif	else if 条件	if: X; elif: Y; else: J
else	else 条件	if: X; elif: Y; else: J
except	如果例外发生，就执行该语句	except ValueError, e: print(e)

关键词	描述	示例
exec	把字符串作为 python 运行	exec 'print("Hello")'
finally	不管是否发生例外，都执行该语句	finally: pass
for	遍历循环集合中的元素	for X in Y: pass
from	从模块中导入特定部分。	from X import Y
global	声明一个全局变量	global X
if	if 条件	if: X; elif: Y; else: J
import	导入一个模块来使用	import os
in	for 循环的一部分。也用于测试 X 是否在 Y 中。	for X in Y: pass 或 1 in [1] == True
is	相当于 <code>==</code> ，测试相等性	1 is 1 == True
lambda	创建一个短的匿名函数	s = lambda y: y ** y; s(3) 注1
not	逻辑上的“非”	not True == False
or	逻辑上的“或”	True or False == True
pass	该代码块为空	def empty(): pass
print	打印这个字符串	print("this string")
raise	当程序出错，抛出一个指定异常信息	raise ValueError("No")
return	返回一个值同时退出函数	def X(): return Y
try	尝试执行这个代码块，如果遇到例外，执行 except 语句	try: pass
while	while 循环	while X: pass
with	把表达式作为一个变量来用	with X as Y: pass 注2
yield	Pause here and return to caller.	def X(): yield Y; X().next()

ai酱注：

注1：输入匿名短函数：`s = lambda y: y ** y`，执行该函数 `s(3)`，输出结果为 27，`**` 为平方运算。lambda 简化了函数定义的书写形式，使代码更为简洁。

注2：with X as Y，X 是一个要执行的表达式，Y 是变量，它存储的是表达式执行返回的结果。一般用于文件的读写和存储。

## 数据类型

针对每一种数据类型，都举出一些例子来，例如对于 string，写下你如何创建一个字符串，对于 number，写出一些不同类型的数字。

类型	描述	示例
True	True 布尔值	True or False == True
False	False 布尔值	False and True == False
None	代表“无”或者“没有”值	x = None
bytes	存储字节，可以是文本、PNG、文件等	x = b"hello"
strings	存储文本信息	x = "hello"
numbers	存储整数	i = 100
floats	存储小数	i = 10.389
lists	存储一系列元素	j = [1,2,3,4]
dicts	存储一系列“键=值”的元素	e = {'x': 1, 'y': 2}

## 字符串转义序列（Escape Sequences）

对于字符串转义序列，你需要在字符串中应用它们，确保自己清楚地知道它们的功能。

转义字符	描述
\	反斜杠

转义字符	描述
'	单引号
"	双引号
\a	响铃
\b	退格
\f	换页符
\n	换行符
\r	回车符
\t	Tab 制表符
\v	垂直制表符

## 老式字符串格式化

一样的，在字符串中使用它们，确认它们的功能。

转义字符	描述	示例
%d	十进制整数（不含浮点数）	"%d" % 45 == '45'
%i	同 %d	"%i" % 45 == '45'
%o	八进制数	"%o" % 1000 == '1750'
%u	无符号十进制整数	"%u" % -1000 == '-1000'
%x	十六进制数小写	"%x" % 1000 == '3e8'
%X	十六进制数大写	"%X" % 1000 == '3E8'
%e	指数计数法，小写 'e'	"%e" % 1000 == '1.000000e+03'
%E	指数计数法，大写 'E'	"%E" % 1000 == '1.000000E+03'
%f	浮点数	"%f" % 10.34 == '10.340000'



转义字符	描述	示例
%F	同 %f	"%F" % 10.34 == '10.340000'
%g	%f 或 %e 更短者	"%g" % 10.34 == '10.34'
%G	同 %g 但是大写	"%G" % 10.34 == '10.34'
%c	符号格式化	"%c" % 34 == ' "' 注1
%r	Repr 格式化（调试格式化）	"%r" % int == '<type 'int'>' 注2
%s	字符串格式化	"%s there" % 'hi' == 'hi there'
%%	百分号	"%g%" % 10.34 == '10.34%'

**ai酱注：**

**注1：** %c 是把 34 转换为 ASCII 码，其对应的 ASCII 码为 " "。

**注2：** %r 打印时能够重现它所代表的对象。

Python 2 的代码使用这些格式化字符来实现 Python 3 中 f 的功能，你也可以试试这些替代方法。

# 运算符

有些操作符号你可能还不熟悉，不过还是逐一看过去，研究一下它们的功能，如果你研究不出来也没关系，记录下来日后再去解决。

运算符	描述	示例
+	加	2 + 4 == 6
-	减	2 - 4 == -2
*	乘	2 * 4 == 8
**	乘方	2 ** 4 == 16
/	除	2 / 4 == 0.5

运算符	描述	示例
//	地板除法（商向下取整）	<code>2 // 4 = 0</code>
%	字符串插入符；取模	<code>2 % 4 == 2</code>
<	小于	<code>4 &lt; 4 == False</code>
>	大于	<code>4 &gt; 4 == False</code>
<=	小于等于	<code>4 &lt;= 4 == True</code>
>=	大于等于	<code>4 &gt;= 4 == True</code>
==	等于	<code>4 == 5 == False</code>
!=	不等于	<code>4 != 5 == True</code>
()	括号	<code>len('hi') == 2</code>
[]	列表中括号	<code>[1,3,4]</code>
{}	字典大括号	<code>{'x': 5, 'y': 10}</code>
@	At (修饰符)	<code>@classmethod</code>
,	逗号	<code>range(0, 10)</code>
:	冒号	<code>def X():</code>
.	点	<code>self.x = 10</code>
=	赋值等号	<code>x = 10</code>
;	分号	<code>print("hi"); print("there")</code>
+=	加赋值	<code>x = 1; x += 2</code>
-=	减赋值	<code>x = 1; x -= 2</code>
*=	乘赋值	<code>x = 1; x *= 2</code>
/=	除赋值	<code>x = 1; x /= 2</code>
//=	地板除赋值	<code>x = 1; x //= 2</code>

运算符	描述	示例
<code>%=</code>	取模赋值	<code>x = 1; x %= 2</code>
<code>**=</code>	乘方赋值	<code>x = 1; x **= 2</code>

**ai酱注：** `x += 2` 相当于 `x = x + 2`，其他同类运算同理。

花一个星期来学习这些东西，如果你能提前完成的话更好。我们的目的是覆盖到所有的符号类型，确认你已经牢牢记住它们。另外很重要的一点是这样可以帮助你发现还没掌握的内容，以便日后学习巩固。

## 阅读代码

现在去找一些 Python 代码阅读一下。你需要尽可能地阅读你能找到的任何代码，然后从中学习一些东西。目前为止你学到的东西已经足够让你看懂一些代码了，但你可能还无法理解这些代码的功能。这节课我要教给你的是：如何运用你学到的东西去理解别人的代码。

首先把你想要理解的代码打印到纸上。没错，你需要打印出来，因为和屏幕相比，你的眼睛和大脑更习惯于纸质内容。一次最多打印几页就可以了。

然后通读你打印出来的代码并做好标记，标记的内容包括以下几个方面：

1. 函数以及函数的功能。
2. 每个变量的初始赋值。
3. 每个在程序中多次出现的变量。它们以后可能会给你带来麻烦。
4. 任何不包含 `else` 的 `if` 语句。它们是正确的吗？
5. 任何可能没有结束的 `while` 循环。
6. 代码中任何你看不懂的部分。

第三步，你需要用注释向自己解释代码的含义。解释各个函数的使用方法，各个变量的用途，以及任何其它方面的内容，只要能帮助你理解代码即可。

最后，对于代码中比较难的部分，逐行或者逐个函数地去跟踪变量值。你可以再打印一份出来，在空白处写出你要“追踪”的每个变量的值。

一旦你基本理解了代码的功能，回到电脑面前，在屏幕再重读一次，看看能不能找到新的问题点。然后继续找新的代码，用上述方法去阅读和理解，直到你不再需要纸质打印为止。

## 附加练习

1. 研究一下什么是“流程图”（flow chart），并试着画一画。
2. 如果你在读代码的时候发现了错误，试着把它们改对，然后把修改内容发给作者。
3. 如果不使用纸质打印，你可以使用注解符号 `#` 在程序中加入笔记。这些笔记会对后来的读代码的人有很大的帮助。

## 常见问题

我如何在网上搜索这些东西？只要在任何你要搜的内容前面加上“python3.6”就行了。比如你要搜 `yield`，就输入 `python3.6 yield`。

## 练习 38. 操作列表

你已经学过了列表。在你学习“while循环”的时候，你对列表进行过“追加(append)”操作——把数字追加到列表结尾并把它们打印了出来。另外你应该还在附加练习里研究过 Python 文档，看了列表支持的其他操作。不过距离之前的学习已经过去了一段时间，所以如果你不记得了的话，就回到本书的前面再复习一遍吧。

找到了吗？还记得吗？很好。那时候你对一个列表执行了 `append` 函数。不过，你也许还没有真正明白发生了什么，所以我们再来看看还可以对列表进行哪些操作。

当你看到像 `mystuff.append('hello')` 这样的代码时，你事实上已经在 Python 内部激发了一个连锁反应。以下是它的工作原理：

1. Python 看到你用到了 `mystuff`，于是就去找到这个变量。也许它需要倒着检查你有没有在哪里用 `=` 创建过这个变量，或者检查它是不是一个函数参数，或者看它是不是一个全局变量。不管哪种方式，它得先找到 `mystuff` 这个变量才行。
2. 一旦它找到了 `mystuff`，就轮到处理句点 `.` (period) 这个操作符，而且开始查看 `mystuff` 内部的一些变量了。由于 `mystuff` 是一个列表，Python 知道 `mystuff` 支持一些函数。
3. 接下来轮到处理 `append`。Python 会将 `append` 和 `mystuff` 支持的所有函数的名称——对比，如果确实其中有一个叫 `append` 的函数，那么 Python 就会去使用这个函数。

4. 接下来 Python 看到了括号 `(`，并且意识到，“噢，原来这是一个函数”，到了这里，它就正常会调用这个函数了，不过这里的函数还要多一个参数才行。
5. 这个额外的参数其实是..... `mystuff`！你会觉得很奇怪对不对？不过这就是 Python 的工作原理，记住它就行了。所以到最后真正发生的事情其实是 `append(mystuff, 'hello')`，不过你所看到的 `mystuff.append('hello')`。

大部分时候你不需要知道这些细节，不过如果你看到如下的 Python 错误信息，上面的细节就对你有用了：

```
$ python3.6
>>> class Thing(object):
...     def test(message):
...         print(message)
...
>>> a = Thing()
>>> a.test("hello")
Traceback (most recent call last): File "<stdin>", line 1 , in <module>
TypeError : test() takes exactly 1 argument (2 given)
>>>
```

这些是什么呢？这是我在 Python 命令行下展示给你的一点魔法。你还没有见过 `class`，不过后面很快就要碰到了。现在你看到 Python 说 `test()takes exactly 1 argument (2 given)` (`test()` 只可以接受 1 个参数，实际上给了 2 个)。它意味着 python 把 `a.test("hello")` 改成了 `test(a, "hello")`，而有人在某个地方弄错了，没有为 `a` 添加这个参数。

一下子要消化这么多可能有点难度，不过我们会做几个练习让你加深印象。下面的练习将字符串和列表混在一起，看看你能不能在里边找出点乐趣来

[ex38.py](#)

```

1     ten_things = "Apples Oranges Crows Telephone Light Sugar"
2
3     print("Wait there are not 10 things in that list. Let's fix it.")
4
5     stuff = ten_things.split(' ')
6     more_stuff = ["Day", "Night", "Song", "Frisbee", "Corn", "Banana", "Girl", "Boy"]
7
8     while len(stuff) != 10:
9         next_one = more_stuff.pop()
10        print("Adding: ", next_one)
11        stuff.append(next_one)
12        print(f"There are {len(stuff)} items now.")
13
14    print("There we go: ", stuff)
15
16    print("Let's do some things with stuff.")
17
18    print(stuff[1])
19    print(stuff[-1]) # whoa! fancy
20    print(stuff.pop())
21    print(' '.join(stuff)) # what? cool!
22    print('#'.join(stuff[3:5])) # super stellar!

```

## 你会看到

```

Wait there are not 10 things in that list. Let's fix that. Adding:      Boy
There are 7 items now. Adding:      Girl
There are 8 items now. Adding:      Banana
There are 9 items now. Adding:      Corn
There are 10 items now.
There we go:      ['Apples', 'Oranges', 'Crows', 'Telephone', 'Light' 'Sugar', 'Boy', 'Girl', 'B
Let's do some things with stuff. Oranges
Corn Corn
Apples Oranges Crows Telephone Light Sugar Boy Girl Banana Telephone#Light

```

## 列表能做什么

假设你想基于“Go Fish”来创建一个计算机游戏，如果你不知道“Go Fish”是什么，可以花些时间去网上查查。要做这个游戏你必须要有“一副卡牌”（“deck of cards”）的概念，并且把它运用到你的 Python 程序里。然后你需要去写 Python 代码来实现这个想象中的卡牌游戏，并且让玩游戏的人认为它是真的，即使它不是。你需要的是“一副牌”的思维框架，程序员们将此称之为—

种“数据结构”。

什么是数据结构？如果你仔细想想，“数据结构”就是用一种正式的方式来组织一些数据（事实），就是这么简单。尽管一些数据结构会异常复杂，它们终究还是一种把事实（facts）存储在一个程序里的方式，你可以用不同的方式访问这些数据。数据结构使数据形成体系。

我会在下一个练习里深入讲解这一点，但是列表是程序员们使用最多的数据结构之一，**它们把你想要存储的内容以一种简单、有序列表方式存储起来，并且可以通过索引（index）来随机（randomly）或线性（linearly）地获取到。**列表不会因为程序员解释说“列表就是列表”而比现实世界中已经存在的列表更复杂。让我们以一副卡牌为例来理解一下列表：

1. 你有一副包含值（value）的卡牌。
2. 这些卡牌被从上到下放成一摞。
3. 你可以从上面取，从下面取，或者从中间随机抽取。
4. 如果你想找到一张特定的卡牌，你就得把整副牌拿起来，一张一张翻找。

让我们看看我刚才所说的定义：

“一个有序列表”，是的，一副卡牌是有序的，有第一张，有最后一张。

“有你想要存储的东西”是的，卡牌就是我想要存储的东西。

“可以随机获取”是的，我可以从这副卡牌里随机抽取一张。

“或者线性获取”是的，如果我想找特定的一张卡牌，我可以从头开始按顺序找。

“通过一个索引”基本上，你拿着一副牌，如果我让你取出第 19 张，你肯定得一张一张数到第 19。在我们的 Python 列表里，程序可以直接跳到任何你指定的索引。

以上就是一个列表所能做的所有事情。这些应该能够帮助你弄明白编程中的思想。编程中的每一种思想通常都与现实世界有着某种联系。至少那些有用的思想都是这样。如果你能弄明白它们在现实世界中的类似情形，你就能据此理解数据结构的作用。

## 何时使用列表

当你有能够匹配列表数据结构有用特性的东西时，你就可以使用列表：

1. 如果你需要保持次序。记住，是列出的顺序，而不是分类的顺序，列表不会为你做分类工作。

2. 如果你需要通过一个数字随机获取到内容。记住，是使用从 0 开始的基数。
3. 如果你需要线性地遍历这些内容 (从头到尾)。记住，这就是 for 循环的作用。

那么，你就可以使用列表。

## 附加练习

1. 将每一个被调用的函数以前面提到的方式翻译成 Python 实际执行的动作。例如：`more_stuff.pop()` 其实是 `pop(more_stuff)`。
2. 将这两种调用函数的方式翻译为自然语言。例如，`more_stuff.pop()` 可以翻译成“在 `more_stuff` 上调用 `pop`”，而 `pop(more_stuff)` 是指“调用 `pop`，参数为 `more_stuff`”。想想它们为什么是同一件事情。
3. 上网阅读一些关于“面向对象编程(Object Oriented Programming)”的资料。是不是有点困惑？我以前也是。别担心，后面还有更难的，慢慢学吧。
4. 查一下 Python 中的“class”是什么东西。不要去看其他语言的“class”用法，会把你搞晕的。
5. 如果你不知道我在说什么，别担心。程序员为了显得自己聪明，于是就发明了“面向对象编程”这种东西，简称为 OOP，然后他们就开始滥用这个东西。如果你觉得这东西太难，你可以先学一下“函数编程(functional programming)”。
6. 在现实世界中找 10 个可以适用于列表的东西。试着写一些脚本来操作一下。

## 常见问题

你不是说不能用 **while 循环**吗？我是说过，所以你要记住，如果你有足够好的理由，你完全可以打破规则。只有傻瓜才会永远做规则的奴隶。

为什么不能用 `join(' ', stuff)`？用这种方式使用 `join` 是行不通的。`join` 是一种可以调用的方法，它能够把字符串放在列表的元素之间，把它们连接起来。你需要这样用它：`' '.join(stuff)`。

你为什么要用 **while 循环**？你可以试试用 **for 循环**，看会不会容易点儿。

`stuff[3:5]` 的作用是什么？它可以从 `stuff` 列表里抽取一个从元素 3 到元素 4 的切片。也就是它并不包含 5，有点类似于 `range(3,5)`。



# 练习 39. 字典， 可爱的字典

现在你要学习 Python 中的另一种数据结构——字典（Dictionary）。字典（也叫 dict）是一种和列表类似的数据存储方式。但是不同于列表只能用数字获取数据，字典可以用任何东西来获取。你可以把字典当成是一个存储和组织数据的数据库。

让我们比较一下列表和字典的作用。你看，列表可以让你做这些事情：

## 练习 39 Python 会话

```
>>> things = ['a', 'b', 'c', 'd']
>>> print(things[1])
b
>>> things[1] = 'z'
>>> print(things[1])
z
>>> things
['a', 'z', 'c', 'd']
```

你可以用数字来索引列表，找到列表里面有些什么。到现在你应该能够理解这一点。但是你还要确保自己明白，你只能用数字来取出列表中的元素。

相比之下，字典能让你用几乎所有的东西，而不只是数字。是的，字典能够把一个东西和另一个东西关联起来，不管它们是什么类型。我们来看看：

## 练习 39 Python 会话

```
>>> stuff = {'name': 'Zed', 'age': 39, 'height': 6 * 12 + 2}
>>> print(stuff['name'])
Zed
>>> print(stuff['age'])
39
>>> print(stuff['height'])
74
>>> stuff['city'] = "SF"
>>> print(stuff['city'])
SF
```

你会看到我们用了字符串（而不是数字）来从 stuff 字典中取出了我们想要的东西。我们也可以使用字符串来给字典添加新的东西。而且，也可以不用字符串，我们可以这样做：

## 练习 39 Python 会话

```
>>> stuff[1] = "Wow"
>>> stuff[2] = "Neato"
>>> print(stuff[1])
Wow
>>> print(stuff[2])
Neato
```

在这一段代码中我用了数字，所以你看，我在打印字典的时候既可以用数字也可以用字符串来作为键。我可以用任何东西。好吧，大多数东西，不过你现在就假装能够用任何东西吧。

当然，如果一个字典只能放东西那就太蠢了。下面是如何用 'del' 关键词来删除其中的东西：

## 练习 39 Python 会话

```
>>> del stuff['city']
>>> del stuff[1]
>>> del stuff[2]
>>> stuff
{'name': 'Zed', 'age': 39, 'height': 74}
```

# 一个字典示例

接下来我们要做一个练习，你必须非常仔细，我要求你将这个练习写下来，然后试着弄懂它做了些什么。当你把东西放进字典、随意取出、以及做其他操作的时候记得做一下笔记。

注意一下这个例子是如何把州名和它们的缩写以及州的缩写和城市映射（mapping）起来的，记住，“映射”或者说“关联”（associate）是字典的核心理念。

[ex39.py](#)

```

1      # create a mapping of state to abbreviation
2      states = {
3          'Oregon': 'OR',
4          'Florida': 'FL',
5          'California': 'CA',
6          'New York': 'NY',
7          'Michigan': 'MI'
8      }
9
10     # create a basic set of states and some cities in them
11     cities = {
12         'CA': 'San Francisco',
13         'MI': 'Detroit',
14         'FL': 'Jacksonville'
15     }
16
17     # add some more cities
18     cities['NY'] = 'New York'
19     cities['OR'] = 'Portland'
20
21     # print out some cities
22     print('-' * 10)
23     print("NY State has: ", cities['NY'])
24     print("OR State has: ", cities['OR'])
25
26     # print some states
27     print('-' * 10)
28     print("Michigan's abbreviation is: ", states['Michigan'])
29     print("Florida's abbreviation is: ", states['Florida'])
30
31     # do it by using the state then cities dict
32     print('-' * 10)
33     print("Michigan has: ", cities[states['Michigan']])
34     print("Florida has: ", cities[states['Florida']])
35
36     # print every state abbreviation
37     print('-' * 10)
38     for state, abbrev in list(states.items()):
39         print(f"{state} is abbreviated {abbrev}")
40
41     # print every city in state
42     print('-' * 10)
43     for abbrev, city in list(cities.items()):
44         print(f"{abbrev} has the city {city}")
45
46     # now do both at the same time

```

```
47     print('-' * 10)
48     for state, abbrev in list(states.items()):
49         print(f"{state} state is abbreviated {abbrev}")
50         print(f"and has city {cities[abbrev]}")
51
52     print('-' * 10)
53     # safely get a abbreviation by state that might not be there
54     state = states.get('Texas')
55
56     if not state:
57         print("Sorry, no Texas.")
58
59     # get a city with a default value
60     city = cities.get('TX', 'Does Not Exist')
61     print(f"The city for the state 'TX' is: {city}")
```

## 你会看到

### 练习 39 会话

```

$ python3.6 ex39.py
-----
NY State has:      New York
OR State has:      Portland
-----
Michigan's abbreviation is: MI
Florida's abbreviation is:      FL
-----
Michigan has:      Detroit
Florida has: Jacksonville
-----
Oregon is abbreviated OR
Florida is abbreviated FL
California is abbreviated CA
New York is abbreviated NY
Michigan is abbreviated MI
-----
CA has the city San Francisco
MI has the city Detroit
FL has the city Jacksonville
NY has the city New York
OR has the city Portland
-----
Oregon state is abbreviated OR
and has city Portland
Florida state is abbreviated FL
and has city Jacksonville
California state is abbreviated CA
and has city San Francisco
New York state is abbreviated NY
and has city New York
Michigan state is abbreviated MI
and has city Detroit
-----
Sorry, no Texas.
The city for the state 'TX' is: Does Not Exist

```

## 字典能做什么

字典是另一种数据结构的例子，在编程中和列表一样常用。字典是用来把你想要存储的东西映射或关联到一些键（keys），以便你能够获取到它们。同样的，程序员们不会用“字典”这个词来指代跟现实生活中的字典没有关系的东西。所以，让我们来看一个现实世界的例子。

假设你想查查“Honorificabilitudinitatibus”这个词的意思。现在你只需要把这个词复制粘贴到一个

搜索引擎就能得到答案。可以说搜索引擎就像一个超级复杂版本的牛津英语字典。在没有搜索引擎之前，你可能会这么做：

1. 去图书馆找到一本字典，让我们假设你要找牛津字典。
2. 你知道“honorificabilitudinitatibus”以字母 H 开头，所以你在书的侧边找那个小小的 H。
3. 然后你在 H 部分翻页寻找直到接近以“hon”开头的单词。
4. 继续翻了几页之后你终于找到了“honorificabilitudinitatibus”，或者你已经翻到以“hp”开头的单词了，你才意识到牛津字典里根本没有这个词。
5. 一旦你找到了这个词，你会看看它的定义，以弄明白它的意思。

这个过程基本上就是字典的工作原理，你把“honorificabilitudinitatibus”这个词映射到它的定义上。而 Python 中的字典和现实世界中的牛津字典非常类似。

## 附加练习

1. 给你所在的国家或其他国家建立城市和州/地区的类似映射。
2. 找找 Python 的字典文档，试试多进行一些操作。
3. 看看你不能用字典做什么。很重要的一点是它们没有次序，试着玩玩看。

## 常见问题

**列表和字典有哪些区别？** 列表是元素的有序排列。而字典是把一些元素（称为“键”， keys）和另一些元素（称为“值”， values）匹配起来。

**我能用字典做什么？** 当你需要用这个东西去查另一个东西的时候。其实，你可以把字典称为“查询表”（look up tables）。

**我能用列表做什么？** 可以用于任何有序排列的东西，同时你只需要用数字索引来查找它们。

**如果我需要一个字典，但我想让它们有序排列怎么办呢？** 去看看 Python 中的 `collections.OrderedDict` 数据结构。可以在网上找找相关文档。

# 练习 40. 模块、类和对象

Python 是一门“面向对象的编程语言”（Object Oriented Programming）。这是指 Python 中有一个叫做 类（class）的结构，能够让你用一种特定的方式结构化你的软件。通过使用类，你可以让你的程序保持连贯性，使用起来更清晰。至少理论上是这样。

我现在要教你一些面向对象编程的初级知识——类和对象，就用你已经学过的列表和模块来解释。我知道面向对象编程（OOP）这个说法听起来有点抽象，但是你必须试着去理解它，老老实实敲代码，我会在下一个练习中深入讲解。

让我们开始吧。

## 模块就像字典

你知道一个字典是如何被创建、使用以及将一个东西映射到另一个东西上。也就是说，如果你有一个字典的键是“apple”，你可以这样做来获取它：

[ex40a.py](#)

```
1     mystuff = {'apple': "I AM APPLES!"}
2     print(mystuff['apple'])
```

记住这种 "get X from Y" 的方式。现在想想模块。你已经用过一些了，应该知道它们是：

1. 一个包含函数和变量的 Python 文件 ..
2. 你导入这个文件。
3. 你用 `.` 运算符来获取这个模块中的函数或变量。

假设我有一个名为 `mystuff.py` 的模块，我在其中放了一个叫做 `apple` 的函数。以下是 `mystuff.py` 这个模块的内容：

[mystuff.py](#)

```
1     def apple():
2         print("I AM APPLES!")
```

**ai酱注：**新建一个 `mystuff.py` 文件来输入。

一旦有了这些代码，我就可以通过导入来使用这个模块，然后获取其中的 apple 函数：

ex40b.py

```
1 import mystuff
2 mystuff.apple()
```

**ai酱注：**新建一个 `ex40b.py` 文件来输入。

我还可以创建一个名为 `tangerine` 的变量：

mystuff.py

```
1 def apple():
2     print("I AM APPLES!")
3
4     # this is just a variable
5     tangerine = "Living reflection of a dream"
```

**ai酱注：**继续在 `mystuff.py` 文件中输入第 4-5 行。

我可以用同样的方式来访问这个变量：

ex40b.py

```
1 import mystuff
2
3 mystuff.apple()
4 print(mystuff.tangerine)
```

**ai酱注：**继续在 `ex40b.py` 文件中输入第 4 行。

说回字典，你应该已经意识到了上述模块的使用与字典非常类似，但是语法有些不一样，让我们对比一下：

```
1 mystuff['apple'] # get apple from dict
2 mystuff.apple() # get apple from the module
3 mystuff.tangerine # same thing, it's just a variable
```

这表明 Python 中有一个非常通用的模式：



1. 用一个键=值（key=value）形式的容器。
2. 通过键的名称来从中获取内容。

在字典中，键是一个字符串，语法是：`[key]`。而在模块中，键是一个识别符，语法是`.key`，除此之外它们几乎是同一种东西。

### 40.1.1 类就像模块

你可以把模块想象成一个可以储存 Python 代码并且可以用 `.` 运算符获取它的特定字典。Python 还有一种类似功能的结构叫做类（class）。**类是一种整合一组函数和数据的方式，它将函数和数据放在一个容器内以便你能通过 `.` 运算符进行访问。**

如果我要创建一个类似 `mystuff` 模块的类，我会这么做：

[ex40c.py](#)

```
1 class MyStuff(object):
2
3     def __init__(self):
4         self.tangerine = "And now a thousand years between"
5
6     def apple(self):
7         print("I AM CLASSY APPLES!")
```

**ai酱注：**新建一个 [ex40c.py](#) 文件来输入。

和模块比起来这看起来有些复杂，而且不同点很多，但是你应该能够看出来它就像一个小型的 `MyStuff` 模块，包含了一个 `apple()` 函数。可能让你困惑的是这个 `__init__()` 函数以及设置实例变量使用的 `self.tangerine`。

使用类而不用模块的原因有：你可以用这个 `MyStuff` 类复制很多个，如果你想的话，一次几百万个都行，并且它们之间不会相互干涉。但是当你导入一个模块，对整个程序来说只有一个，除非你用一些黑客技术。

不过在你理解这些之前，你需要先知道什么是“对象”，以及如何和 `MyStuff` 类一起使用，就像你使用 `mystuff.py` 模块一样。

### 40.1.2 对象就像导入（import）

如果说类是一个小型模块，那么应该要有一个概念和 `import` 类似。这个概念就叫做“实例

化”（instantiate），你可以理解为它是“创造”（create）一词的华而不实、令人讨厌、自以为是的说法。当你实例化一个类，你得到的东西就叫做对象。

你通过像函数一样调用这个类来实例化（创造）一个对象，就像这样：

[ex40c.py](#)

```
1     thing = MyStuff()
2     thing.apple()
3     print(thing.tangerine)
```

**ai酱注：**继续在 [ex40c.py](#) 文件中输入。

第一行就是实例化操作，它很像调用一个函数。不过，Python 在屏幕后面为你协调了一系列事件，我会为你过一下这些步骤：

1. Python 找了一下 `MyStuff()` 然后发现它是你定义的一个类。
2. Python 创建了一个空对象，以及你在类中用 `def` 指定的所有函数。
3. 然后 Python 会看你会不会用一个神奇的 `__init__` 函数，来初始化你新创建的空对象。
4. 在 `MyStuff` 类的 `__init__` 函数中，还有一个变量 `self`，这是 Python 为我创建的空对象，我可以在里面设置变量，就像在模块、字典或其他对象里一样。
5. 在这个例子中，我设置了一个 `self.tangerine` 变量，并给它赋了一句歌词，然后我就完成了对这个对象的初始化。
6. 现在 Python 可以把这个新打造的对象赋给 `thing` 变量来供我使用。

这就是当你像调用函数一样调用一个类的时候，Python 所做的类似于“小型导入”的基本过程。记住，这里不是把这个类给你，而是以这个类为蓝本，创建一个同类型的副本。

你要记住，我告诉你的是一些不太准确的概念，主要是希望能基于已经学过的模块来帮助你理解类。但事实上，类和对象会在这一点上与模块产生偏离。如果我更坦诚一点，我应该这样说：

- 类就像创建新的小型模块的蓝本或定义。
- 实例化是你如何创建这些小型模块并同时导入它们。“实例化”的意思就是从类那里创建一个对象。
- 你所创建的小型模块的结果被称作对象，然后你把它赋给一个变量来使用。

至此，对象和模块开始变得截然不同，以上只能是帮助你理解类和对象的一个桥梁。

## 40.1.3 获取数据

我现在有三种从获取数据的方式：

```
1      # dict style
2      mystuff['apples']
3
4      # module style
5      mystuff.apples()
6      print(mystuff.tangerine)
7
8      # class style
9      thing = MyStuff()
10     thing.apples()
11     print(thing.tangerine)
```

## 40.1.4 第一个类的例子

你应该已经注意到了这三种键值对（键=值）容器的相似之处，并且可能还有一大堆问题要问。先别着急问，因为下个练习会向你硬性灌输一些“面向对象的术语”。在这个练习中，我只希望你敲敲代码，让程序正常运行，这样你能在继续学习之前获得一些切身体验。

[ex40.py](#)

```
1      class Song(object):
2
3          def __init__(self, lyrics):
4              self.lyrics = lyrics
5
6          def sing_me_a_song(self):
7              for line in self.lyrics:
8                  print(line)
9
10     happy_bday = Song(["Happy birthday to you",
11                        "I don't want to get sued",
12                        "So I'll stop right there"])
13
14     bulls_on_parade = Song(["They rally around tha family",
15                             "With pockets full of shells"])
16
17     happy_bday.sing_me_a_song()
18
19     bulls_on_parade.sing_me_a_song()
```

# 你会看到

## 练习 40 会话

```
$ python3.6 ex40.py
Happy birthday to you
I don't want to get sued
So I'll stop right there
They rally around tha family
With pockets full of shells
```

## 附加练习

1. 用这个方法再写一些歌，确保你明白你正在用字符列表来传歌词。
2. 把歌词放在一个单独的变量里，然后把这个变量放在类里面来使用。
3. 如果你能搞定这些，可以用它来做更多的事情。要是你现在没什么想法也别担心，就试试看会发生什么。然后把它们掰开、揉碎、反复研究。
4. 在网上搜搜“面向对象的编程”，然后填满你的大脑。别担心你看不懂，因为几乎一半的东西我也看不懂。

## 常见问题

为什么我在类下面用 `__init__` 函数或者其他函数的时候要用 `self` ？如果你不用 `self`，那么像 `cheese = 'Frank'` 这样的代码就会很含糊，计算机不知道你是指实例的 `cheese` 属性还是一个叫做 `cheese` 的局部变量。而用 `self.cheese = 'Frank'` 的话就会很清晰，你是指实例的属性 `self.cheese`。

## 练习 41. 学着去说面向对象

在这个练习中，我要教你如何去说“面向对象”。我所做的就是给你一些你需要了解的词和定义。然后我会给出一些需要填空的句子让你去理解。最后，你要完成一个大练习，从而在大脑中巩固这些句子。

# 词汇训练

（注：为了方便理解，定义保留英文原文。）

**类（class）**：告诉 Python 创建一个新类型的东西（Tell Python to make a new type of thing）。

**对象（object）** 两种含义：最基本类型的东西，任何实例。（the most basic type of thing, and any instance of something.）

**实例（instance）**：当你告诉 Python 创建一个类的时候你所得到的东西。（What you get when you tell Python to create a class.）

**def**：你如何在类里面定义一个函数。（How you define a function inside a class.）

**self**：在一个类的函数里面，self 是被访问的实例/对象的一个变量。（Inside the functions in a class, self is a variable for the instance/object being accessed.）

**继承（inheritance）**：关于一个类能从另一个类那里继承它的特征的概念，很像你和你的父母。（The concept that one class can inherit traits from another class, much like you and your parents.）

**组合（composition）**：关于一个类可以由其他一些类构成的概念，很像一辆车包含几个轮子。（The concept that a class can be composed of other classes as parts, much like how a car has wheels.）

**属性（attribute）**：类所拥有的从组合那里得到的特性，通常是变量。（A property classes have that are from composition and are usually variables.）

**is-a**：一种用来表达某物继承自一种东西的表述，就像“三文鱼是一种鱼”。（A phrase to say that something inherits from another, as in a “salmon” is a “fish.”）

**has-a**：一种用来表达某物是由一些东西组成或具有某种特性的表述，就像“三文鱼有一个嘴巴”。（A phrase to say that something is composed of other things or has a trait, as in “a salmon has-a mouth.”）

花点时间为这些术语做一些闪词卡（flash cards）并记住它们，虽然在你完成这个练习之前单纯的记忆没有任何意义，但你必须要先了解这些基础的词汇。

# 短语训练

接下来是一些 Python 代码片段以及右边的解释。

```
class X(Y) :
```

创建一个名为 X 并继承自 Y 的类。

(“Make a class named X that is-a Y.”)

```
class X(object): def __init__(self, J)
```

类 X 有一个带有 self 和 J 参数的 `__init__` 函数。

(“class X has-a `__init__` that takes self and J parameters.”)

```
class X(object): def M(self, J) :
```

类 X 有一个带有 self 和 J 参数的 M 函数。

(“class X has-a function named M that takes self and J parameters.”)

```
foo = X() :
```

设 foo 为类 X 的一个实例。

(“Set foo to an instance of class X.”)

```
foo.M(J)
```

从 foo 那里获取 M 函数，并用 self 和 J 参数来调用它。

(“From foo, get the M function, and call it with parameters self, J.”)

```
foo.K = Q
```

从 foo 那里获取 K 属性，并设它为 Q。

(“From foo, get the K attribute, and set it to Q.”)

在上述每一句中，当你看到 X, Y, M, J, K, Q, 以及 foo, 你可以把它们当空格，比如，我还可以把这些句子写成：

1. “Make a class named ??? that is-a Y.”  
(创建一个名为 ??? 的类，它继承自 Y。)
2. “class ??? has-a `__init__` that takes self and ??? parameters.”  
(类 ??? 有一个带了 self 和 ??? 参数的 `__init__`。)
3. “class ??? has-a function named ??? that takes self and ??? parameters.”  
(类 ??? 有一个名为 ??? 的函数，这个函数带有 self 和 ??? 两个参数。)
4. “Set foo to an instance of class ???.”

- (设 foo 为类 ??? 的一个实例。)
5. “From foo, get the ??? function, and call it with self=??? and parameters ???.”  
(从 foo 那里获取 ??? 函数, 并用 `self=???` 以及参数 ??? 来调用它。)
  6. “From foo, get the ??? attribute, and set it to ???.”  
(从 foo 那里获取 ??? 属性, 把它设为 ???。)

同样地, 把这些短语写到一些闪词卡上, 然后记一记。把 Python 代码片段放在正面, 解释的句子放在背面, 你必须每次都正确说出每一个短语的意思。不是说得类似就行, 而是要一模一样。

## 综合训练

最后一项准备工作是把词汇训练和短语训练结合在一起, 以下是训练内容:

1. 做一个短语卡然后练习记忆。
2. 把它翻过来, 读句子, 如果在句子中看到词汇训练中的词汇, 就找到相应的词汇卡片。
3. 练习记忆这些词汇卡片。
4. 坚持练习, 要是你感到有些累, 就休息一下再继续。

## 一个阅读测试

现在我有一个小的 Python 脚本来帮助你掌握这些词汇和短语, 并且能够无限运行。这段脚本很简单, 你应该能够看明白, 它所做的事情就是用一个叫做 `urllib` 的图书馆来下载一系列单词。以下是脚本代码, 你需要输入到 `oop_test.py` 这个文件里来使用:

[ex41.py](#)

```

1  import random
2  from urllib.request import urlopen
3  import sys
4
5  WORD_URL = "http://learncodethehardway.org/words.txt"
6  WORDS = []
7
8  PHRASES = {
9      "class %%(%%)":":
10         "Make a class named %% that is-a %%.",
11         "class %%(object):\n\tdef __init__(self, ***)" :
12             "class %% has-a __init__ that takes self and *** params.",
13         "class %%(object):\n\tdef ***(self, @@@)":
14             "class %% has-a function *** that takes self and @@@ params.",
15         "*** = %%()":":
16             "Set *** to an instance of class %%. ",
17         "****.***(@@@)":
18             "From *** get the *** function, call it with params self @@@.",
19         "****.*** = '***'":
20             "From *** get the *** attribute and set it to '***'."
21     }
22
23     # do they want to drill phrases first
24     if len(sys.argv) == 2 and sys.argv[1] == "english":
25         PHRASE_FIRST = True
26     else:
27         PHRASE_FIRST = False
28
29     # load up the words from the website
30     for word in urlopen(WORD_URL).readlines():
31         WORDS.append(str(word.strip(), encoding="utf-8"))
32
33
34     def convert(snippet, phrase):
35         class_names = [w.capitalize() for w in
36             random.sample(WORDS, snippet.count("%%"))]
37         other_names = random.sample(WORDS, snippet.count("***"))
38         results = []
39         param_names = []
40
41         for i in range(0, snippet.count("@@@")):
42             param_count = random.randint(1,3)
43             param_names.append(', '.join(
44                 random.sample(WORDS, param_count)))
45
46         for sentence in snippet, phrase:

```



```

47         result = sentence[:]
48
49         # fake class names
50         for word in class_names:
51             result = result.replace("%%%", word, 1)
52
53         # fake other names
54         for word in other_names:
55             result = result.replace("****", word, 1)
56
57         # fake parameter lists
58         for word in param_names:
59             result = result.replace("@@@", word, 1)
60
61         results.append(result)
62
63     return results
64
65
66 # keep going until they hit CTRL-D
67 try:
68     while True:
69         snippets = list(PHRASES.keys())
70         random.shuffle(snippets)
71
72         for snippet in snippets:
73             phrase = PHRASES[snippet]
74             question, answer = convert(snippet, phrase)
75             if PHRASE_FIRST:
76                 question, answer = answer, question
77
78             print(question)
79
80             input("> ")
81             print(f"ANSWER: {answer}\n\n")
82 except EOFError:
83     print("\nBye")

```

运行这个脚本，试着用“面向对象的短语”来把它翻译成自然语言，你应该能看到短语字典有两种形式，你只用输入正确的那个就行。

## 练习从自然语言到代码

接下来你应该选择用“english”选项来运行这段代码，然后用相反的方式来练习：

```
$ python oop_test.py english
```

记住，这些短语在用一些废话，学习阅读这些代码的一部分原因就是试着不再去给这些变量和类的名字赋予这么多意义。通常当人们看到像“cork”（软木塞）这样的词时，会对它的含义感到很困惑。在上述例子中，“cork”只是一个随机选取的类的名字。别给它赋予太多含义，而是试着用我教你的方式来对待它。

## 读更多代码

你现在需要继续读更多的代码，并在这些代码中复习你之前学过的短语。试着找到尽可能多的包含类的文件，然后跟着如下要求去做：

1. 给出每个类的名字，以及其他的类从它那里继承了什么。
2. 在每个类下面，列出它所拥有的函数以及它们的参数。
3. 列出所有它用 self 使用的属性。
4. 对于每个属性，给出它继承自哪个类。

这些练习的目的是过一遍真实的代码，并试着把你学过的短语和它们的用法匹配和关联起来。如果你做足了训练，你会开始看到这些匹配模式（match patterns）呼之欲出，而不再是一些你不明白的空格或字符。

## 常见问题

`result = sentence[:]` 是干什么用的？这是 Python 复制一个列表的方式。它用的是列表的切片（slice）语法 `[:]`，能够很快地创建一个从第一个元素到最后一个元素的列表切片。

**这个脚本好难运行！** 到目前为止你应该能够让它正常运行。虽然它确实有几个小地方比较烦人，但是并不复杂。试着用你目前为止所学过的东西来调试它。把每一行输入进去，并且确保和我的一模一样，然后遇到不明白的地方就在网上查查。

**还是很难！** 你可以这样做。慢点敲，一个字符一个字符地敲，但是要保证准确，然后弄明白每个词的意思。

# 练习 42. Is-A, Has-A, 对象和类

你必须理解类和对象的区别，这是一个很重要的概念。不过问题是，类和对象之间没有什么真正的区别。它们在不同的时间点其实是同一种东西，我会用禅宗（Zen koan）来解释这一点：

## 鱼和三文鱼的区别是什么？

这个问题会让你困惑吗？坐下来认真想一分钟，我是说，鱼和三文鱼的确是有区别的，但是它们是同一种东西，对吧？三文鱼是鱼的一种，所以我说它们没什么区别。但是同时，三文鱼只是一种特定种类的鱼，它肯定不同于其他种类的鱼。三文鱼是三文鱼，而不是比目鱼。所以三文鱼和鱼是同一种东西，但是又有区别。

这个问题很令人困惑，因为大多数人不会这么去思考真实的东西，但是大家直觉上又能理解。你不需要去想鱼和三文鱼的具体区别是什么，因为你知道它们是相关的。你知道三文鱼是一种鱼，而且还有其他种类的鱼我们不用去理解。

让我们更进一步。假设你有一个水桶装了三条三文鱼，由于你是一个好人，你决定给它们三个起个名字，分别叫 Frank、Joe 和 Mary。现在想想这个问题：

## Mary 和三文鱼的区别是什么？

这也是一个很奇怪的问题。但是它好像比鱼和三文鱼的问题要简单一点。你知道 Mary 是一条三文鱼，所以她真的不一样，她只是三文鱼的一个“实例”。Joe 和 Frank 也是三文鱼的实例。当我说“实例”的时候我指的是什么？我是指它们由其他三文鱼创造而来，但是现在代表了一个具有三文鱼属性的真实存在的东西。

现在回到这个让人伤脑筋的问题：鱼是一个类，三文鱼也是一个类，而 Mary 是一个对象。想几秒钟，让我们拆开来讲，看你是否理解了。

鱼是一个类，意味着它不是一个真正的东西，而是一个我们用来给具有相似属性的实例归类的词，理解了吗？比如有鳍，有鳃，生活在水里，好吧，那可能是鱼。

可能会有位 Ph.D. 跑过来说，“不，年轻人，这鱼其实是大西洋鲑，人们喜欢叫它三文鱼。”这位教授只是更详细地澄清了一下，同时创建了一个叫做“三文鱼”的新类，它有一些更特别的属性。鼻子很长，肉呈淡红色，体型大，生活在淡水里，很好吃？那可能是三文鱼。

最后，一位厨师跑过来告诉这位 Ph.D.，“不，你看这条三文鱼，我叫她 Mary，我等会儿要用她做一道很好吃的生鱼片。”现在你有了一个叫做 Mary 的三文鱼实例（也是鱼的实例），她是真实存在的，能填饱你的肚子。她已经变成了一个对象。

现在你明白了：Mary 是一种三文鱼，三文鱼是一种鱼。对象一种类，类是另一种类。

## 代码怎么写

这是个很奇怪的概念，不过说实话，你只用在创建新类和使用类的时候才用担心它。我会教你两个识别一个东西是类还是对象的小技巧。

首先，你需要学习两个信号词：“is-a”（是...）和“has-a”（有...）。当你表达对象和类的相互关系时，你用“is-a”。当你指对象和类相互引用时，你用“has-a”。

现在，过一遍这些代码，然后把 `###??` 替换为注释，说明下一行代表了 is-a 还是 has-a 的关系，以及是什么关系。我在代码最开始已经列出了一些示例，你需要完成剩余的部分。

记住，is-a 指的是鱼和三文鱼之间的关系，has-a 指的是三文鱼和鳃的关系。

[ex42.py](#)

```

1      ## Animal is-a object (yes, sort of confusing) look at the extra credit (附加练习)
2      class Animal(object):
3          pass
4
5      ## ??
6      class Dog(Animal):
7
8          def __init__(self, name):
9              ## ??
10             self.name = name
11
12         ## ??
13         class Cat(Animal):
14
15             def __init__(self, name):
16                 ## ??
17                 self.name = name
18
19         ## ??
20         class Person(object):
21
22             def __init__(self, name):
23                 ## ??
24                 self.name = name
25
26             ## Person has-a pet of some kind
27             self.pet = None
28
29         ## ??
30         class Employee(Person):
31
32             def __init__(self, name, salary):
33                 ## ?? hmm what is this strange magic?
34                 super(Employee, self).__init__(name)
35                 ## ??
36                 self.salary = salary
37
38         ## ??
39         class Fish(object):
40             pass
41
42         ## ??
43         class Salmon(Fish):
44             pass
45
46         ## ??

```

```
47     class Halibut(Fish):
48         pass
49
50
51     ## rover is-a Dog
52     rover = Dog("Rover")
53
54     ## ??
55     satan = Cat("Satan")
56
57     ## ??
58     mary = Person("Mary")
59
60     ## ??
61     mary.pet = satan
62
63     ## ??
64     frank = Employee("Frank", 120000)
65
66     ## ??
67     frank.pet = rover
68
69     ## ??
70     flipper = Fish()
71
72     ## ??
73     crouse = Salmon()
74
75     ## ??
76     harry = Halibut()
```

## 关于 类名(object)

我一直强迫你使用 `类名(object)`，但一直没跟你解释为什么要这样用。刚才你已经学了类和对象的区别，现在我可以告诉你原因了。因为如果我早告诉你的话，你可能会晕掉，也就学不会这门技术了。

真正的原因是在 Python 早期，它对于类的定义在很多方面都是严重有问题的。当他们承认这一点的时候已经太迟了，所以逼不得已，他们需要支持这种有问题的类。为了解决已有的问题，他们需要引入一种“新类”，这样的话“旧类”还能继续使用，而你也有一个新的正确的类可以使用了。

这就用到了“类即是对象”的概念。他们决定用小写的“object”这个词作为一个类，让你在创建新类时从它继承下来。有点晕吧？一个类继承自另一个类，而后者虽然是个类，名字却叫“object”。不过在定义类的时候，别忘记要从 object 继承就好了。

的确如此。一个词的不同就让这个概念变得更难理解，让我不得不现在才讲给你。现在你可以试着去理解“一个是对象的类”这个概念了，如果你感兴趣的话。

不过我还是建议你别去理解了，干脆完全忘记旧格式和新格式类的区别吧，就假设 Python 的类永远都要求你加上 (object) 好了，你的脑力要留着思考更重要的问题。

## 附加练习

1. 研究一下为什么 Python 添加了这个奇怪的叫做 object 的类，它究竟有什么含义呢？
2. 有没有可能把一个类当作对象来使用呢？
3. 在习题中为 animals、fish、还有 people 添加一些函数，让它们做一些事情。看看当函数在 Animal 这样的“基类(base class)”里和在 Dog 里有什么区别。
4. 找找别人的代码，弄明白里面的 is-a 和 has-a 的关系。
5. 使用列表和字典创建一些新的一对多的“has-many”的关系。
6. 你认为会有一种“has-many”的关系吗？阅读一下关于“多重继承(multiple inheritance)”的资料，然后尽量避免这种用法。

## 常见问题

**这些 ## ?? 注释是做什么的？** 这些是一些注释的填空，你需要在那里填上正确的“is-a”或“has-a”的概念。把这个练习再读一遍，看看其他的注释，你就明白我的意思了。

`self.pet = None` 有什么意义？这样确保 `self.pet` 这个类的属性被设为默认的 None。

`super(Employee, self).__init__(name)` 是干什么的？这是你运行父类的 `init` 方法的一种可靠方式。搜索一下“python3.6 super”，读读那些对你有利有弊的各种建议。

## 练习 43. 面向对象的分析和设计基础

这个练习我想说一下当你想要用 Python 创建一个东西，尤其是面向对象编程的时候，过程是怎

样的。我说的“过程”指的是我会给出一些有序的步骤，但你也不用生搬硬套，因为它们也不一定适用每一个问题。它们只不过是给很多编程问题提供一个很好的开端，而不是解决这类问题的唯一方法，只是你可以参考的其中一种方法。

过程如下：

1. 把问题写或者划下来。
2. 提炼出关键概念，并进行研究。
3. 为这些概念创建一个类的层级和对象关系图。
4. 写下这些类的代码，并测试运行。
5. 重复和改进。

这是一种“自上而下”的方式，它从非常抽象、松散的想法开始，然后慢慢提炼，直到想法变得具体，可以通过代码来实现。

我会先从写下问题开始，尽可能地写下我所能想到的点。可能我还会画一两张图表、地图之类的，甚至会给我自己写一系列邮件来阐述这个问题。这样能让我针对这个问题把一些关键的概念表达出来，并且探索出关于该问题我可能已经掌握的东西。

然后我会过一遍这些笔记、图表以及描述，从其中抽象出关键概念。这里有一个小技巧：把你所写所画的东西里面所有的名词和动词列一个表出来，然后写下它们之间是如何相互关联的。这种方法让我得到了一个关于下一步的类、对象和函数名的列表。我拿着这个概念列表，研究其中我不明白的点，如果我需要的话，对其进行改进。

一旦我有了这个概念列表，我就创造了一个简单的概念框架，以及它们作为类是如何相互关联的。你可以经常列你的名词表，然后问自己“这个跟其他的概念名词类似吗？也就是说，它们有共同的父类吗？有的话应该叫什么？”重复这个过程直到你得到一个类的层级结构，可能就是一个简单的树状图或者示意图。然后把所有的动词挑出来，看看它们能不能作为每个类的函数名，然后把它们放到你的树状图里面。

等类的层级结构梳理清楚之后，我会坐下来，写一些基本的代码框架，只是一些类和它们的函数，没有其他东西。然后我会写一些测试代码，跑一下，看这些类有没有意义以及能不能正常运行。有时我会先写测试代码，有时候就是一小段测试，一小段代码，再一小段测试，以此类推，直到我把整个程序构建起来。

最后，我会重复这个过程，并且在运行的过程中不断精简，在添加更多应用之前让代码更简洁明了。如果我在某个特定环节因为一个概念或者我没有预料到的问题而卡壳，我会坐下来，只运行这一部分，直到把问题弄明白之后再继续。



我现在要通过一个游戏引擎和一个游戏练习来过一遍这个过程。

## 一个简单的游戏引擎分析

我要制作的这个游戏叫做“来自25号行星的哥顿人”（Gothons from Planet Percal #25），它是一个小型太空冒险游戏。因为我满脑子都是这个概念，我就去探索这个想法，然后思考如何把这个游戏做出来。

### 43.1.1 写或画出这个问题

我会写一小段关于这个游戏的文字：

“外星人入侵了一艘宇宙飞船，我们的英雄必须穿过迷宫般的房间打败他们，这样他才能逃到逃生舱去到下面的星球。游戏更像是 Zork 之类的文字冒险游戏，并且有着很有意思的死亡方式。这款游戏的引擎会运行一张满是房间或场景的地图。当玩家进入游戏时，每个房间都会打印自己的描述，然后告诉引擎下一步该运行地图中的哪个房间。”

这时我有了一个关于这个游戏以及它如何运行的好想法，所以现在我要描述一下每个场景：

**死亡（Death）**：玩家死的时候，会非常有意思。

**中央走廊（Central Corridor）**：这是起点，已经有一个哥顿人站在那里，在继续之前，玩家必须用一个笑话来击败他。

**激光武器军械库（Laser Weapon Armory）**：这是英雄在到达逃生舱之前用中子弹炸毁飞船的地方。这里有一个键盘，英雄必须猜出数字。

**桥（The Bridge）**：另一个和哥顿人战斗的场景，英雄在这里放置了炸弹。

**逃生舱（Escape Pod）**：英雄逃脱的地方，前提是他猜出正确的逃生舱。

到这一步我可能会画一幅映射图，或者为每个房间写更多的描述——反正就是当我探究这个问题的时候，任何我脑子里冒出的想法。

### 43.1.2 抽取关键概念并予以研究

我现在有足够的信息来提取其中的名词，并分析他们的类层级结构。首先，我会做一个所有名词的列表：

- Alien（外星人）
- Player（玩家）

- Ship（飞船）
- Maze（迷宫）
- Room（房间）
- Scene（场景）
- Gothon（哥特人）
- Escape Pod（逃生舱）
- Planet（行星）
- Map（地图）
- Engine（引擎）
- Death（死亡）
- Central Corridor（中央走廊）
- Laser Weapon Armory（激光武器军械库）
- The Bridge（桥）

我可能还会浏览一遍所有的动词，看它们适不合作为函数名，但是我会先暂时跳过这一步。

现在你可能也会研究一下每个概念以及任何你不明白的东西。比如，我会玩几个同类型的游戏，确保我知道它们是如何工作的。我可能还会研究船是如何设计的或者炸弹是怎么用的。还有一些技术性问题，比如如何把游戏状态储存在数据库中。当我完成这些研究，我可能会基于这些新信息从第一步开始，重新写我的描述，并做概念提取。

### 43.1.3 为这些概念创建类的层级结构和对象地图

我通过询问“什么与其他东西类似？”、“什么基本上就是另一个东西的另一个词？”来把我已经有的东西转换成类的层级结构。

很快我就发现“房间”（“Room”）和“场景”（“Scene”）基本上是同一种东西，取决于我想用它们来做什么。在这个游戏中我选择用“场景”。然后我意识到所有特定的房间比如“中央走廊”其实就是“场景”。我还发现“死亡”（“Death”）也可以说是场景，这确认了我选择“场景”而不是“房间”的正

确性，因为你可以说“死亡”是一种场景，但如果说它是一个“房间”就有点奇怪了。“迷宫”（“Maze”）和“地图”（“Map”）也基本上是同一种东西，我会选择用“地图”，因为我更常用它。我不想做一个战斗系统，所以我会暂时忽略“外星人”（“Alien”）和“玩家”（“Player”）这两个东西，先保存起来以备后用。“行星”（“Planet”）也可以是另一种场景，而不是其他特定的东西。

经过上述思考过程，我开始创建一个看起来像这样的类的层级结构：

- Map
- Engine
- Scene
  - Death
  - Central Corridor
  - Laser Weapon Armory
  - The Bridge
  - Escape Pod

然后我会浏览一遍，基于我描述里面的东西，想想看每个类下面需要些什么动作。例如，我从描述里知道，我需要一种方式来“运行”这个引擎，从地图“到达下一个场景”，到达“开场”，并“进入”一个场景，我会像这样把这些动作加上：

- Map
  - next\_scene
  - opening\_scene
- Engine
  - play
- Scene
  - enter
    - Death
    - Central Corridor
    - Laser Weapon Armory
    - The Bridge
    - Escape Pod

注意我只把“enter”放在了“场景”下面，所有“场景”下面的东西都会继承这个动作，需要随后再重写。

## 43.1.4 编写类代码并通过测试来运行

一旦我有了这个类和函数的树状图，我在我的编辑器里面打开一个源文件，试着写它们的代码。通常我就是把树状图里的东西复制粘贴到源文件里，然后把它们编辑成类。下面是它们最开始的样子，文件最后放了一个小测试：

ex43\_classes.py

```
1     class Scene(object):
2
3         def enter(self):
4             pass
5
6
7     class Engine(object):
8
9         def __init__(self, scene_map):
10             pass
11
12         def play(self):
13             pass
14
15     class Death(Scene):
16
17         def enter(self):
18             pass
19
20     class CentralCorridor(Scene):
21
22         def enter(self):
23             pass
24
25     class LaserWeaponArmory(Scene):
26
27         def enter(self):
28             pass
29
30     class TheBridge(Scene):
31
32         def enter(self):
33             pass
34
35     class EscapePod(Scene):
36
37         def enter(self):
38             pass
39
40
41     class Map(object):
42
43         def __init__(self, start_scene):
44             pass
45
46         def next_scene(self, scene_name):
```

```
47         pass
48
49     def opening_scene(self):
50         pass
51
52
53     a_map = Map('central_corridor')
54     a_game = Engine(a_map)
55     a_game.play()
```

在这个文件中你可以看到，我只是复制了层级结构中我想要的东西，并在最后加上了一些测试代码来运行，看这个基本结构能不能成立。在这个练习后面的几部分，你会填上剩余的代码，让它像游戏描述中那样运行。

## 43.1.5 重复和改进

过程的最后一步准确来说不是一个步骤，而像是一个 while 循环。你不可能一次完成这个过程。相反，你会再次回顾整个过程，并根据你从后续步骤中学到的信息对其进行改进。有时我会进入第三步，然后意识到我需要再回到第一步和第二步，那我就会停下来，回到前面去做。有时我会灵光一闪，跳到最后，把脑子里的解决方案代码敲出来，然后再回过头来做前面的步骤，以确保我涵盖了所有可能的情况。

在这个过程中你需要注意的另一个问题是，它不仅仅是你在一个单层面上做的事，而是当你遇到一个特定的问题时，你可以在每个层面上做的事情。假设我不知道怎么写 `Engine.play` 这个方法。我可以停下来，把整个过程专注在这一个函数上来弄明白代码应该怎么写。

## 自上而下 vs 自下而上

这个过程通常被称为“自上而下”，因为它从最抽象的概念（上）开始，然后一直向下到实际的应用。我希望能从现在开始分析这本书里遇到的问题时使用我刚才描述的这个过程，但是你应该知道编程中还有另一种解决问题的方式，那就是，从写代码开始，然后逐渐“上升”到抽象的概念，这种方式被称为“自下而上”。它的步骤大致如下：

1. 从问题中拿出一小部分，开始写简单的能运行的代码。
2. 然后用类和自动化测试来把代码改进地更正式一些。
3. 抽象出你所使用的关键概念，试着探究一下它们。
4. 针对正在发生的事情写一段描述。
5. 回过头去继续改进代码，也可能把之前写的删掉重新开始。

6. 转到这个问题的其他部分，然后重复以上步骤。

这个过程只有在你对编程已经比较熟练并且面对问题能够自然使用编程思维的情况下才会更好，同时，当你知道整个工程的一小部分、却对全局概念的信息掌握不全的时候，这个方法也很好用。你可以将整个过程拆解成很多小块，然后边写代码边探索，这样可以帮助你一点一点地钻研这个问题，直到整个问题都得到解决。但是，请记住，你的解决方案很可能会曲折而怪异，所以我才把回顾、研究以及基于你所学到的东西对代码进行改进和清理这些步骤加入到我的过程描述中。

## “来自25号行星的哥顿人”游戏代码

停！接下来我要向你展示针对之前的问题我最终的解决方，但是我想让你直接跳进去开始敲代码，我希望你自己先基于描述粗略地写出代码框架，然后试着让它运行，一旦你有了你的解决方案，你再回来看我是怎么做的。

我会把最终的 `ex43.py` 拆成几个部分分别解释，而不是直接把所有代码一次全部给你。

`ex43.py`

```
1      from sys import exit
2      from random import randint
3      from textwrap import dedent
```

这是游戏所需库的基本引入。唯一的新东西是从 `textwrap` 模块导入 `dedent` 函数。这个函数将帮助我们使用 `"""`（三引号）字符串来编写我们的房间描述。它就是简单地从字符串的行首删除空白。如果没有这个函数，使用 `"""` 样式字符串就会失败，因为它们在屏幕上缩进的程度与 Python 代码相同。

`ex43.py`

```
1      class Scene(object):
2
3          def enter(self):
4              print("This scene is not yet configured.")
5              print("Subclass it and implement enter().")
6              exit(1)
```

正如你在框架代码中看到的，我有一个基类 `Scene`，它具有所有场景都具有的公共功能。在这

个简单的程序中，它们不会做太多的工作，只是向你演示如何创建基类。

#### ex43.py

```
1      class Engine(object):
2
3          def __init__(self, scene_map):
4              self.scene_map = scene_map
5
6          def play(self):
7              current_scene = self.scene_map.opening_scene()
8              last_scene = self.scene_map.next_scene('finished')
9
10             while current_scene != last_scene:
11                 next_scene_name = current_scene.enter()
12                 current_scene = self.scene_map.next_scene(next_scene_name)
13
14             # be sure to print out the last scene
15             current_scene.enter()
```

我还有 Engine 类。你可以看到我已经在使用 `Map.opening_scene` 和 `Map.next_scene` 这两个方法了。因为我已经提前计划好了，所以可以在写出 Map 类之前就把这些方法写下来并用起来。

#### ex43.py

```
1      class Death(Scene):
2
3          quips = [
4              "You died. You kinda suck at this.",
5              "Your Mom would be proud...if she were smarter.",
6              "Such a luser.",
7              "I have a small puppy that's better at this.",
8              "You're worse than your Dad's jokes."
9          ]
10
11
12          def enter(self):
13              print(Death.quips[randint(0, len(self.quips)-1)])
14              exit(1)
```

我的第一个场景很反常地设置为了 Death，主要是想向你展示你可以写的最简单的场景。



```

1 class CentralCorridor(Scene):
2
3     def enter(self):
4         print(dedent("""
5             The Gothons of Planet Percal #25 have invaded your ship and
6             destroyed your entire crew. You are the last surviving
7             member and your last mission is to get the neutron destruct
8             bomb from the Weapons Armory, put it in the bridge, and
9             blow the ship up after getting into an escape pod.
10
11             You're running down the central corridor to the Weapons
12             Armory when a Gothon jumps out, red scaly skin, dark grimy
13             teeth, and evil clown costume flowing around his hate
14             filled body. He's blocking the door to the Armory and
15             about to pull a weapon to blast you.
16             """))
17
18         action = input("> ")
19
20         if action == "shoot!":
21             print(dedent("""
22                 Quick on the draw you yank out your blaster and fire
23                 it at the Gothon. His clown costume is flowing and
24                 moving around his body, which throws off your aim.
25                 Your laser hits his costume but misses him entirely.
26                 This completely ruins his brand new costume his mother
27                 bought him, which makes him fly into an insane rage
28                 and blast you repeatedly in the face until you are
29                 dead. Then he eats you.
30                 """))
31             return 'death'
32
33         elif action == "dodge!":
34             print(dedent("""
35                 Like a world class boxer you dodge, weave, slip and
36                 slide right as the Gothon's blaster cranks a laser
37                 past your head. In the middle of your artful dodge
38                 your foot slips and you bang your head on the metal
39                 wall and pass out. You wake up shortly after only to
40                 die as the Gothon stomps on your head and eats you.
41                 """))
42             return 'death'
43
44         elif action == "tell a joke":
45             print(dedent("""
46                 Lucky for you they made you learn Gothon insults in

```

```
47         the academy. You tell the one Gothon joke you know:
48         Lbhe zbgure vf fb sng, jura fur fvgf nebhaq gur ubhfr,
49         fur fvgf nebhaq gur ubhfr. The Gothon stops, tries
50         not to laugh, then busts out laughing and can't move.
51         While he's laughing you run up and shoot him square in
52         the head putting him down, then jump through the
53         Weapon Armory door.
54         """))
55     return 'laser_weapon_armory'
56
57 else:
58     print("DOES NOT COMPUTE!")
59     return 'central_corridor'
```

然后我创建了中央走廊，这是游戏的开始。我把游戏的场景放在 Map 之前，是因为我需要在随后引用它们。你应该也看到了我在第4行用了 `dedent` 函数。稍后你可以尝试删除它，看看它会做什么。

[ex43.py](#)

```

1 class LaserWeaponArmory(Scene):
2
3     def enter(self):
4         print(dedent("""
5             You do a dive roll into the Weapon Armory, crouch and scan
6             the room for more Gothons that might be hiding. It's dead
7             quiet, too quiet. You stand up and run to the far side of
8             the room and find the neutron bomb in its container.
9             There's a keypad lock on the box and you need the code to
10            get the bomb out. If you get the code wrong 10 times then
11            the lock closes forever and you can't get the bomb. The
12            code is 3 digits.
13            """))
14
15         code = f"{randint(1,9)}{randint(1,9)}{randint(1,9)}"
16         guess = input("[keypad]> ")
17         guesses = 0
18
19         while guess != code and guesses < 10:
20             print("BZZZZEDDD!")
21             guesses += 1
22             guess = input("[keypad]> ")
23
24         if guess == code:
25             print(dedent("""
26                 The container clicks open and the seal breaks, letting
27                 gas out. You grab the neutron bomb and run as fast as
28                 you can to the bridge where you must place it in the
29                 right spot.
30                 """))
31             return 'the_bridge'
32         else:
33             print(dedent("""
34                 The lock buzzes one last time and then you hear a
35                 sickening melting sound as the mechanism is fused
36                 together. You decide to sit there, and finally the
37                 Gothons blow up the ship from their ship and you die.
38                 """))
39             return 'death'
40
41
42
43 class TheBridge(Scene):
44
45     def enter(self):
46         print(dedent("""

```

```

47         You burst onto the Bridge with the netron destruct bomb
48         under your arm and surprise 5 Gothons who are trying to
49         take control of the ship. Each of them has an even uglier
50         clown costume than the last. They haven't pulled their
51         weapons out yet, as they see the active bomb under your
52         arm and don't want to set it off.
53         """))
54
55     action = input("> ")
56
57     if action == "throw the bomb":
58         print(dedent("""
59             In a panic you throw the bomb at the group of Gothons
60             and make a leap for the door. Right as you drop it a
61             Gothon shoots you right in the back killing you. As
62             you die you see another Gothon frantically try to
63             disarm the bomb. You die knowing they will probably
64             blow up when it goes off.
65             """))
66         return 'death'
67
68     elif action == "slowly place the bomb":
69         print(dedent("""
70             You point your blaster at the bomb under your arm and
71             the Gothons put their hands up and start to sweat.
72             You inch backward to the door, open it, and then
73             carefully place the bomb on the floor, pointing your
74             blaster at it. You then jump back through the door,
75             punch the close button and blast the lock so the
76             Gothons can't get out. Now that the bomb is placed
77             you run to the escape pod to get off this tin can.
78             """))
79
80         return 'escape_pod'
81     else:
82         print("DOES NOT COMPUTE!")
83         return "the_bridge"
84
85
86 class EscapePod(Scene):
87
88     def enter(self):
89         print(dedent("""
90             You rush through the ship desperately trying to make it
91             the escape pod before the whole ship explodes. It seems
92             like hardly any Gothons are on the ship, so your run is

```

```

93         clear of interference. You get to the chamber with the
94         escape pods, and now need to pick one to take. Some of
95         them could be damaged but you don't have time to look.
96         There's 5 pods, which one do you take?
97         """)
98
99     good_pod = randint(1,5)
100    guess = input("[pod #]> ")
101
102
103    if int(guess) != good_pod:
104        print(dedent("""
105            You jump into pod {guess} and hit the eject button.
106            The pod escapes out into the void of space, then
107            implodes as the hull ruptures, crushing your body into
108            jam jelly.
109            """))
110        return 'death'
111    else:
112        print(dedent("""
113            You jump into pod {guess} and hit the eject button.
114            The pod easily slides out into space heading to the
115            planet below. As it flies to the planet, you look
116            back and see your ship implode then explode like a
117            bright star, taking out the Gothon ship at the same
118            time. You won!
119            """))
120
121        return 'finished'
122
123 class Finished(Scene):
124
125     def enter(self):
126         print("You won! Good job.")
127         return 'finished'

```

这是游戏的剩余场景，因为我知道我需要它们，并且已经想过它们之间如何流转，所以我能直接把代码写出来。

顺便说一句，我不会把所有这些代码都输入进去。还记得我说过要循序渐进，一点一点来。现在我只给你们看最后的结果。

[ex43.py](#)

```

1      class Map(object):
2
3          scenes = {
4              'central_corridor': CentralCorridor(),
5              'laser_weapon_armory': LaserWeaponArmory(),
6              'the_bridge': TheBridge(),
7              'escape_pod': EscapePod(),
8              'death': Death(),
9              'finished': Finished(),
10         }
11
12         def __init__(self, start_scene):
13             self.start_scene = start_scene
14
15         def next_scene(self, scene_name):
16             val = Map.scenes.get(scene_name)
17             return val
18
19         def opening_scene(self):
20             return self.next_scene(self.start_scene)

```

然后是我的 Map 类，你可以看到它把每个场景的名字存储在一个字典里，然后我用 Map.scenes 引用那个字典。这也是为什么地图出现在场景之后的原因，因为字典必须引用场景，所以场景必须先存在。

### ex43.py

```

1      a_map = Map('central_corridor')
2      a_game = Engine(a_map)
3      a_game.play()

```

最后是我通过制作地图来运行游戏的代码，在调用 play 使游戏工作之前，我把地图交给了引擎。

## 你会看到

确保你理解了这个游戏，并且你先试图自己去解决它。如果你被难住了，你可以通过阅读我的代码来作弊，然后继续尝试自己解决它。

以下是我运行我的游戏的结果：

## 练习 43 会话

```
$ python3.6 ex43.py
```

```
The Gothons of Planet Percal #25 have invaded your ship and destroyed your entire crew. You are th
```

```
You're running down the central corridor to the Weapons Armory when a Gothon jumps out, red scaly
```

```
> dodge!
```

```
Like a world class boxer you dodge, weave, slip and slide right as the Gothon's blaster cranks a l
```

```
You're worse than your Dad's jokes.
```

## 附加练习

1. 改变它！也许你不喜欢这个游戏，因为太暴力了，也可能你对科幻不感兴趣。先让游戏运行起来，然后把它变成你喜欢的样子。这是你的电脑，你可以让它做你想做的。
2. 我这段代码有一个bug。为什么门锁猜了11次？
3. 解释一下如何返回隔壁房间。
4. 在游戏中添加作弊代码，这样你就可以通过比较难的房间。我可以通过在一行写两个字来实现这一点。
5. 回到我的描述和分析，然后尝试为这个英雄和他遇到的各种哥特人建立一个小的战斗系统。
6. 这实际上是“有限状态机”（finite state machine）的一个小版本。读读相关的内容，你可能看不懂，但无论如何都要试一试。

## 常见问题

我在哪里可以找到我自己的游戏故事？你可以自己编，就像你给朋友讲故事一样。或者你可以从你喜欢的书或电影中选取一些简单的场景。

## 练习 44. 继承和组合

在英雄打败坏人的童话故事中，总有一个类似黑暗森林的东西。它可能是一个洞穴、一片森林、

另一个星球，或者只是一些每个人都知道的英雄不应该去的地方。当然，在介绍了坏人之后不久，你就会发现，是的，英雄不得不去那个愚蠢的森林杀死坏人。似乎英雄总是不停地陷入需要他冒着生命危险待在邪恶森林里这个情境。

你很少能读到一些英雄足够聪明、能完全避免这种境地的童话故事，你永远不会听到一个英雄说：“等等，如果我离开去公海发财，离开我的宝贝儿，我会死的，然后她就得嫁给一个叫汉珀丁克（Humperdink）的丑王子。汉珀丁克！我想我会留在这里，做一个经营租赁业务的农场男孩。”如果他这么做了，就不会有火灾沼泽、死亡、复活、剑战、巨人，或者任何类型的故事。正因为如此，这些故事中的森林似乎就像个黑洞一样存在着，不管主人公做什么，它都会把他拖进去。

在面向对象编程中，继承（inheritance）就是邪恶的森林。有经验的程序员知道要避免这种邪恶，因为他们知道在“继承”这个黑暗森林的深处，是邪恶的女王“多重继承”（Multiple Inheritance）。她喜欢吃软件和程序员，用她那巨大的复杂的牙齿，咀嚼堕落者的血肉。但是森林是如此强大、如此诱人，以至于几乎每个程序员都必须深入其中，在他们可以自称为真正的程序员之前，尝试带着邪恶皇后的人头活着出来。你根本无法抗拒继承森林的吸引力，所以你会不可避免地进去。在这次冒险之后，你要学会离开那座愚蠢的森林，如果你被迫再次进入森林，你需要带一支军队。

这是一种用来讲授“继承”的非常有趣的方式。你们需要小心使用继承。目前正在森林里与女王搏斗的程序员可能会告诉你必须进去。他们这样说是因为他们需要你的帮助，因为他们所创造的可能对他们来说太多而难以驾驭。但你应该永远记住这一点：继承的大多数用法都可以用组合（composition）来简化或替换。并且无论如何都要避免多重继承。

## 什么是继承？

继承用来表明一个类将从其父类那里获得大多数或所有特性。无论何时你写 `class Foo(Bar)`，继承都会隐式地发生，它的意思是“创建一个继承自 Bar 的 Foo 类”。当你这样做的时候，编程语言会让你对 Foo 的实例所做的任何动作都能像对 Bar 的实例所做的那样生效。这样做可以让你把公共函数功能放在 Bar 类中，然后在 Foo 类中按需将该功能专门化。

当你在做这种专门化时，有三种父类和子类可以交互的方法：

1. 对子类的行为意味着对父类的行为。
2. 子类上的操作会覆盖父类上的操作。
3. 子类上的操作会更改父类上的操作。



现在，我会依次演示这些方法以及它们的代码。

## 44.1.1 隐式继承（Implicit Inheritance）

首先，我会向你展示当你在父类而不是子类中定义一个函数时发生的隐式动作。

[ex44a.py](#)

```
1      class Parent(object):
2
3          def implicit(self):
4              print("PARENT implicit()")
5
6      class Child(Parent):
7          pass
8
9      dad = Parent()
10     son = Child()
11
12     dad.implicit()
13     son.implicit()
```

在子类 Child 下面使用 pass 是告诉 Python 你需要一个空块的方式。这样就创建了一个名为 Child 的类，但是并没有什么新的内容需要定义。相反，它将继承父类的所有行为。当你运行这个代码，你会得到：

Exercise 44a 会话

```
$ python3.6 ex44a.py
PARENT implicit()
PARENT implicit()
```

注意，即使我在第 16 行调用了 `son.implicit()`，并且 Child 里面也没有定义一个隐式函数，它仍然可以正常运行，它调用了在 Parent 中定义的那个函数。这表明如果将函数放在基类中(比如 Parent)，然后所有子类（比如 Child）会自动获得这些特性。对于需要写很多重复代码的类来说非常方便。

## 44.1.2 显式继承（Override Explicitly）

隐式调用函数的问题在于，有时你希望子类的行为有所不同。在本例中，你希望覆盖子类中的函

数，从而有效地替换功能。为此，你只需要在 Child 中定义一个同名函数。例如：

ex44b.py

```
1      class Parent(object):
2
3          def override(self):
4              print("PARENT override()")
5
6      class Child(Parent):
7
8          def override(self):
9              print("CHILD override()")
10
11      dad = Parent()
12      son = Child()
13
14      dad.override()
15      son.override()
```

在这个例子中，两个类都有一个名为 `override` 的函数，让我们看看当你运行它的时候会发生什么：

Exercise 44b 会话

```
$ python3.6 ex44b.py
PARENT override()
CHILD override()
```

如你所见，当第 14 行运行时，它运行了 `Parent.override` 函数，因为那个变量（`dad`）是一个 `Parent`。但是当第 15 行运行时，它打印了 `Child.override` 信息。因为 `son` 是 `Child` 的一个实例，`Child` 通过定义它自己的版本来重写了那个函数。

现在休息一下，在继续之前尝试研究一下这两个概念。

### 44.1.3 修改前后

第三种使用继承的方式是覆盖的一种特殊情况，你希望在父类的版本运行之前或之后更改行为。你首先像上一个示例那样覆盖该函数，然后使用名为 `super` 的 Python 内置函数调用父类版本。

下面是一个示例，帮助你理解这个描述：

## ex44c.py

```
1      class Parent(object):
2
3          def altered(self):
4              print("PARENT altered()")
5
6      class Child(Parent):
7
8          def altered(self):
9              print("CHILD, BEFORE PARENT altered()")
10             super(Child, self).altered()
11             print("CHILD, AFTER PARENT altered()")
12
13      dad = Parent()
14      son = Child()
15
16      dad.altered()
17      son.altered()
```

这里比较重要的是 9-11 行，在 Child 中，当调用 `son.altered()` 时，我其实做了以下事情：

1. 因为在 `Child.altered` 版本运行时，我就重写了 `Parent.altered`。第 9 行就按照你的预期执行了。
2. 在这个例子中，我想做一个之前和之后的对比，所以在第 9 行之后，我想使用 `super` 来获得 `Parent.altered` 版本。
3. 在第 10 行，我调用 `super(Child, self).altered()`，它意识到需要继承，并会为你获取 `Parent` 类。你应该能够把这个理解为“使用参数 `Child` 和 `self` 来调用 `super`，然后在它返回的任何地方调用 `altered` 函数”。
4. 此时，`Parent.altered` 版本的函数运行，并打印出 `Parent` 的信息。
5. 最后，它从 `Parent.altered` 返回。`Child.altered` 函数继续打印出之后的信息。

如果你运行这段代码，你会看到：

### Exercise 44c 会话

```
$ python3.6 ex44c.py

PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

## 44.1.4 三者结合

为了解释以上所有情况，我有一个最终版本，用一个文件来说明继承的每种交互情况：

[ex44d.py](#)

```
1      class Parent(object):
2
3          def override(self):
4              print("PARENT override()")
5
6          def implicit(self):
7              print("PARENT implicit()")
8
9          def altered(self):
10             print("PARENT altered()")
11
12     class Child(Parent):
13
14         def override(self):
15             print("CHILD override()")
16
17         def altered(self):
18             print("CHILD, BEFORE PARENT altered()")
19             super(Child, self).altered()
20             print("CHILD, AFTER PARENT altered()")
21
22     dad = Parent()
23     son = Child()
24
25     dad.implicit()
26     son.implicit()
27
28     dad.override()
29     son.override()
30
31     dad.altered()
32     son.altered()
```

过一遍这段代码的每一行，然后给每一行加上注释，说明它的作用，以及它是否进行了重写，然后运行它，看是否与你的预期相符：

Exercise 44d 会话

```
$ python3.6 ex44d.py
PARENT implicit()
PARENT implicit()
PARENT override()
CHILD override()
PARENT altered()
CHILD, BEFORE PARENT altered()
PARENT altered()
CHILD, AFTER PARENT altered()
```

## 用 `super()` 的理由

这似乎是常识，但是我们遇到了多重继承的麻烦。多重继承是指你定义了一个继承自一个或多个类的类，就像这样：

```
class SuperFun (Child, BadStuff):
    pass
```

这就像是在说“创建一个名为 `SuperFun` 的类，它同时继承自 `Child` 类和 `BadStuff` 类。”

在这种情况下，每当你任何 `SuperFun` 的实例执行隐式操作时，Python 都必须在 `Child` 类和 `BadStuff` 类的层级结构中查找可能的函数，不过它需要以一致的顺序来执行这项操作。为了做到这一点，Python 使用了“方法解析顺序”(method resolution order, MRO)和一种被称为 C3 的算法。

因为 MRO 非常复杂，并且使用了定义良好的算法，所以 Python 不能让你自己来处理 MRO。而是提供了 `super()` 函数，它可以在你需要更改类似动作的地方为你解决这个问题，就像我在 `Child.altered` 中所做的那样。使用 `super()`，你不用担心是否正确，Python 会为你找到正确的函数。

### 44.2.1 用 `__init__` 来使用 `super()`

`super()` 最常用的用法其实是在基类中使用 `__init__` 函数。这通常是你在一个子类中唯一需要做一些操作，然后在父类中完成初始化的地方。下面是一个在用在子类上的简单例子：

```
class Child (Parent):

    def __init__(self, stuff):
        self.stuff = stuff
        super(Child, self).__init__( )
```

这和上面例子中的 `Child.altered` 很像，除了我在用 `Parent.__init__` 给 `Parent` 做初始化之前在 `__init__` 里面设置了一些参数。

## 组合

继承很有用，但是还有一种能实现相同效果的方法，就是使用其他类和模块，而不是依赖于隐式继承。如果你看看使用继承的三种方法，其中两种都涉及编写新代码来替换或更改函数功能。这很容易通过调用模块中的函数来复制。例如：

[ex44e.py](#)

```
1      class Other(object):
2
3          def override(self):
4              print("OTHER override()")
5
6          def implicit(self):
7              print("OTHER implicit()")
8
9          def altered(self):
10             print("OTHER altered()")
11
12     class Child(object):
13
14         def __init__(self):
15             self.other = Other()
16
17         def implicit(self):
18             self.other.implicit()
19
20         def override(self):
21             print("CHILD override()")
22
23         def altered(self):
24             print("CHILD, BEFORE OTHER altered()")
25             self.other.altered()
26             print("CHILD, AFTER OTHER altered()")
27
28     son = Child()
29
30     son.implicit()
31     son.override()
32     son.altered()
```

在这段代码中，我没有使用 Parent 这个名字，因为不存在父子 is-a 关系，而是一个 has-a 关系，其中 Child 有一个（has-a）Other 来完成它的工作。当我运行这段代码，会得到以下输出：

#### Exercise 44e 会话

```
$ python3.6 ex44e.py
OTHER implicit()
CHILD override()
CHILD, BEFORE OTHER altered()
OTHER altered()
CHILD, AFTER OTHER altered()
```

可以看到，Child 和 Other 中的大多数代码都是相同的，可以完成相同的事情。唯一的区别是我必须定义一个 `Child.implicit` 函数来完成这个动作。然后我可以问自己是否需要这个 Other 作为一个类，我是否可以将其放入一个名为 `Other.py` 的模块中？

## 何时使用继承或组合

“继承与组合”的问题可以归结为试图解决可复用代码的问题。你不希望在你的软件中到处都有重复的代码，因为这不够简洁和高效。继承通过在基类中创建隐含特性的机制来解决这个问题。组合通过提供模块以及调用其他类中的函数来解决这个问题。

如果这两个解决方案都解决了复用问题，那么在哪种情况下用哪个方案比较合适呢？答案非常主观，但我会给你我的三个指导方针来帮你做选择：

1. 无论如何都要避免多重继承，因为它太复杂而且不可靠。如果你被它困住了，那么要准备好了解一下类的层次结构，并花时间找出所有内容的来源。
2. 使用组合将代码打包到模块中，这些模块可以用于许多不同的、不相关的地方和情境。
3. 只有当存在明显相关的可复用代码片段，并且这些代码片段符合单个通用概念，或者由于你使用了某些东西而别无选择时，你才可以使用继承。

不要成为这些规则的奴隶。关于面向对象编程，需要记住的一点是，它完全是程序员为了打包和共享代码而创建的一种社会约定。因为这是一种社会惯例，并且在 Python 中已经形成了这种惯例，你可能会因为与你一起工作的人而被迫绕过这些规则。在这种情况下，弄明白他们是如何使用每一种东西，然后努力适应这种情况。

## 附加练习

这个练习只有一个附加练习，因为这节课是一个大练习。请阅读 <http://www.python.org/dev/peps/pep-0008/> 并开始在你的代码中使用它。你会注意到其中一些和你在本书中所学的不太一样，但是现在你应该能够理解它们的建议并且用在自己的代码中。本书中的其余代码是否遵循这些指导原则，取决于它是否会让代码变得更加混乱。我建议你也这样做，因为理解比你把深奥的规则强加给别人更重要。

## 常见问题

**如何才能更好地解决我之前没有遇到过的问题？** 要想更好地解决问题，唯一的方法就是自己解决尽可能多的问题。通常情况下，人们遇到一个难题，就会冲出去寻找答案。当你着急完成工作的时候，这没问题。但是如果你有时间自己解决，那就花时间去解决。停下来，尽可能长时间地思考这个问题，尝试所有可能的方案，直到你把它解决或者放弃。在这之后，你找到的答案会更令人满意，你最终也会更擅长解决问题。

**对象不就是类的拷贝吗？** 在某些语言中（如 JavaScript），这是对的。这些被称为原型语言，除了用法之外，对象和类之间没有太多区别。然而，在 Python 中，类充当“铸造”（mint）新对象的模板，类似于使用模具（die）来铸造硬币的概念。

## 练习 45. 你来做一个游戏

你要开始学会自食其力了。通过阅读这本书，你应该已经知道，你需要的所有的信息网上都有，你只要去搜索就能找到。唯一困扰你的就是如何使用正确的词汇进行搜索。学到现在，你在挑选搜索关键字方面应该已经有些感觉了。现在是时候尝试写一个大项目，并让它运行起来。

要求如下：

1. 创建一个不同于我之前那个的游戏。
2. 使用多个文件，并使用 import 来调用它们。确保你知道 import 的用法。
3. 每个房间使用一个类，并给出符合其用途的类名（比如 GoldRoom、KoiPondRoom）。
4. 你的执行器需要了解这些房间，所以创建一个运行并了解它们的类。有很多方法可以做到这一点，但是你要考虑如何让每个房间返回到下一个房间，或者设置一个下



一个房间是什么的变量。

除此之外，就交给你了。花一周时间来做这件事，让它成为你能做的最好的游戏。使用类、函数、字典、列表，任何你可以使它变得更好的东西。本课的目的是教你如何在其他文件中构造需要其他类的类。

记住，我不会告诉你具体怎么做，因为你必须自己做、去弄清楚。编程就是解决问题的过程，这个过程意味着你要不断尝试、试验、失败、放弃你之前所做的，然后再试一次。当你遇到困难时，可以向别人寻求帮助，并给他们看你的代码。如果他们对你很刻薄，就别理他们，把注意力放在那些不刻薄并且愿意帮你的人身上。持续修改和简化你的代码，直到它完整可执行为止，然后再研究一下还有没有可以被改进的地方。

祝你好运，一周后再见。

## 评估你的游戏

在这个练习中你会评估你所完成的游戏。也许你只完成了一半，卡在哪里没有进行下去，也许你勉强做出来了。不管怎样，我们会串一下你应该弄明白的一些东西，并确认你的游戏里有使用到这些内容。我们还会学习如何用正确的格式构建类以及使用类的一些通用习惯，另外还有很多“书本知识”让你学习。

为什么我会让你先行尝试，然后才告诉你正确的做法呢？因为从现在开始你要学会“自给自足”，以前我一直牵着你往前走，以后就得靠你自己了。后面的习题我只会告诉你你的任务，你需要自己去完成，在你完成后我再告诉你如何改进你的作业。

一开始你可能会觉得很困难并且很不习惯，但是只要你坚持下去，就会培养出自己解决问题的能力。你还会找出创新的方法来解决问题，这比从课本中复制粘贴强多了。

## 函数的风格

以前我教过的如何写好函数的方法同样适用，不过这里要再添加几条：

- 由于各种各样的原因，程序员将类里边的函数称作方法（method）。很大程度上这只是个宣传策略。不过每次你把它们称作“函数”，就会有啰嗦的人跳出来纠正你应该叫“方法”。你要是觉得他们太烦了，可以让他们从数学角度演示一下“函数”和“方法”究竟有什么不同，这样他们很快就会闭嘴。
- 在你使用类的过程中，很大一部分时间是告诉你的类如何“做事情”。给这些函数命名

的时候，与其用函数的功能来命名函数，不如把它当作是给类的命令来命名。就像列表的 `pop` 函数一样，它相当于说：“嘿，列表，把这东西给我 `pop` 出去。”它的名字不是 `remove_from_end_of_list`，即使它的功能的确是这样，这一串字符也不是针对一个列表的命令。

- 让你的函数保持小而简洁。不知道为什么，有些人开始学习类之后就会忘了这一点。

## 类的风格

- 你的类应该使用“驼峰式大小写”（camel case），比如你应该用 `SuperGoldFactory` 而不是 `super_gold_factory`。
- 你的 `__init__` 函数不应该做太多的事情，这会让类变得难以使用。
- 你的其它函数应该使用“下划线隔词”（underscore format），所以你可以写 `my_awesome_hair`，而不是 `myawesomhair` 或者 `MyAwesomeHair`。
- 用一致的方式组织函数的参数。如果你的类需要处理 `users`、`dogs` 和 `cats`，就保持这个次序（特殊情况除外）。如果一个函数的参数是（`dog`, `cat`, `user`），另一个的是（`user`, `cat`, `dog`），这会让函数使用起来很困难。
- 尽量不要使用来自模块或者全局的变量，它们应该是相互独立的。
- 愚蠢地保持一致性是思维狭隘的表现。一致性是好事，但是无脑地遵循一些白痴口号是一种很不好的作为。你应该好好为自己着想。
- 永远永远都要使用 `类名(object)` 的方式定义类，否则你会碰到大麻烦。

## 代码的风格

- 为了以方便他人阅读，为自己的代码字符之间留一些空行。你会看到一些很差的程序员，他们写的代码还算通顺，但字符之间没有任何空行。这种风格在任何编程语言中都是坏习惯，因为人的眼睛和大脑会通过空白和垂直对齐的位置来扫描和区分视觉元素，如果你的代码里没有任何空白，这相当于为你的代码刷了一层伪装涂料。
- 如果一段代码你无法朗读出来，那么这段代码的可读性可能就有问题。如果你无法让一段代码简单易用，试着大声朗读出来。这样不仅会强迫你放慢速度，真正去仔细阅读，还会帮你找到难

读的段落，从而知道哪些代码的易读性需要作出改进。

- 学着模仿别人的风格写 Python 程序，直到有一天你也能找到自己的风格。
- 一旦你有了自己的风格，也别把它太当回事。程序员工作的一部分就是和别人的代码打交道，有的人审美就是很差。相信我，你的审美某一方面一定也很差，只是你从未意识到而已。
- 如果你发现有人写的代码风格你很喜欢，那就模仿他们的风格。

## 好的注释

- 程序员可能会告诉你，你的代码需要有足够的可读性，所以不用写注释。他们会以自己接近官腔的声音说“所以你永远都不应该写注释或文档。证明完毕（QED）”。这些程序员要么是做顾问的，如果别人无法使用他们的代码，就会付更多钱给他们让他们解决问题。要么他们能力不足，从来没有跟别人合作过。别理会这些人，好好写你的注解。
- 当你写注解的时候，要描述清楚为什么要这样做以及你在做什么。代码只会告诉你“这样实现”，但是“为什么要这样实现”更为重要。
- 当你为函数写文档注解的时候，记得为别的代码使用者也写些东西。你不需要狂写一大堆，但一两句话说明一下这个函数的用法还是很有用的。
- 虽然注释是好东西，但是太多的注释就不见得是了。而且注释也是需要维护的，所以你要尽量让注释简短切题，如果你对代码做了更改，记得检查并更新相关的注释，确保它们还是正确的。

## 评估你的游戏

现在我要你假装成是我，板起脸来，把你的代码打印出来，然后拿一支红笔，把代码中所有的错误都标出来，包括你从这个练习中或者从目前为止你读到的其他指南中发现的。等你批改完了，我要求你把所有的错误改正过来。这个过程你要多重复几次，争取找到更多可以改进的地方。使用我前面教过的方法，把代码分解成最小的单元，一一进行分析。

这个练习的目的就是训练你对于细节的关注。等你检查完自己的代码，再找一段别人的代码用这种方法检查一遍。把一部分代码打印出来，检查出所有关于代码和风格方面的错误，然后试着在不改坏别人代码的前提下把它们修改正确。

这周我要求你的事情就是批改和纠错，包含你自己的代码和别人的代码。这个练习难度很大，不

过一旦你完成了任务，你学过的东西就会牢牢记在脑中。

## 练习 46. 一个项目骨架

这个练习你将学习如何创建一个好的项目“骨架”（skeleton）目录。这个骨架目录具备让项目跑起来的所有基本内容。它里边会包含你的项目文件布局、自动化测试代码、模块，以及安装脚本。当你建立一个新项目的时候，只要把这个目录复制过去，改改目录的名字，再编辑里边的文件就行了。

### macOS/Linux 设置

在开始这个练习之前，你需要用一个叫做 pip3.6（或者 pip）的工具为 Python 安装一些新的模块。python3.6 的安装中已经包含了 `pip3.6` 命令。你可以通过如下命令来验证一下：

```
$ pip3.6 list
pip (9.0.1)
setuptools (28.8.0)
$
```

如果看到任何弃用警告，可以忽略它。您可能还会看到安装了其他工具，但是基本的应该是 pip 和 setuptools。一旦你验证了这一点，你就可以安装 virtualenv:

```
$ sudo pip3.6 install virtualenv Password:
Collecting virtualenv
Downloading virtualenv-15.1.0-py2.py3-none-any.whl (1.8MB) 100% |||||
Installing collected packages: virtualenv Successfully installed virtualenv- 15.1.0
$
```

这是用于 Linux 或者 macOS 系统的，如果你用的是 Linux/macOS，你可以运行如下命令来确保你安装了正确的 virtualenv:

```
$ whereis virtualenv
/Library/Frameworks/Python.framework/Versions/3.6/bin/virtualenv
```

您应该能在 macOS 上看到类似上面的内容，但是 Linux 上可能不一样。在 Linux 上，你可能有一个实际的 `virtualenv3.6` 命令，或者你最好从你的包管理系统（package management

system) 中为它安装一个包 (package)。

一旦安装了 virtualenv, 你就可以用它来创建一个“伪” Python 安装, 从而更容易管理不同项目的包版本。首先, 运行如下命令, 我等会儿会解释它的作用:

```
$ mkdir ~/.venvs
$ virtualenv --system-site-packages ~/.venvs/lpthw
$ . ~/.venvs/lpthw/bin/activate
(lpthw) $
```

每一行发生的事情如下:

1. 你在你的 HOME `~/` 地址下创建了一个叫做 `.venvs` 的目录, 用来存储你的虚拟环境。
2. 你运行了 `virtualenv` 然后告诉它要包含 `system site packages` (`-- system-site-packages`), 然后指导它在 `~/.venvs/lpthw` 中创建 `virtualenv`。
3. 然后你在 bash 中用 `.` 运算符来获得 `lpthw` 虚拟环境, 后面跟着 `~/.venvs/lpthw/bin/activate` 脚本。
4. 最后, 你的提示变成了包含 `(lpthw)`, 这样你就知道了你正在使用虚拟环境。

现在你可以看到东西被安装在哪里:

```
(lpthw) $ which python
/Users/zedshaw/.venvs/lpthw/bin/python
(lpthw) $ python
Python 3.6.0rc2 (v3.6.0rc2:800a67f7806d, Dec 16 2016, 14:12:21)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> quit()
(lpthw) $
```

你可以看到运行中的 python 被安装在 `/Users/zedshaw/.venvs/lpthw/bin/python` 目录, 而不是原来的地址。这样还免去了要输入 `python3.6` 的麻烦, 因为它把二者都安装了:

```
$ which python3.6
/Users/zedshaw/.venvs/lpthw/bin/python3.6
(lpthw) $
```

你会发现 `virtualenv` 和 `pip` 命令也一样。这个设置的最后一步是安装 `nose`, 一个我们要在练习中用到的测试框架。

```
$ pip install nose
Collecting nose
  Downloading nose-1.3.7-py3-none-any.whl (154kB)
    100% |#####| 163kB 3.2mb/s Installing collected packages...
Successfully installed nose -1.3.7
(lpthw) $
```

## Windows 10 设置

Windows 10 的安装会比 Linux 或者 macOS 简单一些，但是前提是你只安装了一个版本的 Python。如果你有两个版本：Python 3.6 和 Python 2.7，那你自己靠自己了，因为搞多重安装太复杂了。如果你一直跟着这本书学的话，你应该只有 Python 3.6，然后你可以这样做。

首先，切换到你的 home 目录，然后确保你正在运行正确版本的 Python：

```
> cd ~
> python
Python 3.6.0 (v3.6.0:41 df79263a11 , Dec 23 2016, 08:06:12) [MSC v.1900 64 bit (AMD64)] d
Type "help", "copyright", "credits" or "license" for more informa
>>> quit()
```

然后运行 `pip` 确保你已经做了基本的安装：

```
> pip list
pip (9.0.1)
setuptools (28.8.0)
```

你可以安全地忽略任何弃用警告，如果你安装了其他的包也没有关系。接着，安装 `virtualenv` 来为这本书接下来的内容设置一个虚拟环境：

```
> pip install virtualenv
Collecting virtualenv
  Using cached virtualenv-15.1.0-py2.py3-none-any.whl
Installing collected packages:
Successfully installed virtualenv-15.1.0
```

安装好 `virtualenv` 之后，你需要创建一个 `.venvs` 目录，并填入一个虚拟环境：

```
> mkdir .venvs
> virtualenv --system-site --packages.venvs/lpthw
Using base prefix
  'c:\users\zedsh\appdata\local\programs\python\python36
New python executable in
  C:\Users\zedshaw\.venvs\lpthw\Scripts\python.exe
Installing setuptools, pip, wheel ... done.
```

这两行命令创建了一个 `.venvs` 文件夹来存储不同的虚拟环境，然后又创建了你的第一个虚拟环境 `lpthw`。一个虚拟环境（virtualenv）是一个用来运行软件的虚构的地方，这样你就有了针对每个项目包的不同版本。设置好 `virtualenv` 之后你需要激活它：

```
> .\.venvs\lpthw\Scripts\activate
```

这个命令会让 PowerShell 运行 `activate` 脚本，这个脚本会为你当前的 shell 配置 `lpthw` 虚拟环境。每次你想用你在这本书里的软件，你都要运行这个命令。你会看到 PowerShell 中的下一行命令提示符前面已经有了一个 `(lpthw)`，这表明了你正在使用的虚拟环境。最后，你只需要安装 `nose` 来运行随后的测试：

```
(lpthw) > pip install nose
Collecting nose
  Downloading nose-1.3.7-py3-none-any.whl (154kB)
    100% | | i63kB i.2mb/s Installing collected packages: nos
Successfully installed nose-1.3.7
(lpthw) >
```

你会看到，这样就安装了 `nose`，不过 `pip` 会把它安装在你的 `.venvs\lpthw` 虚拟环境中，而不是主系统的目录。这可以让你为每个项目安装相互冲突的 Python 包版本，而不会影响主系统的配置。

## 创建项目骨架目录

首先，用以下这些命令创建你的项目骨架结构：

```
$ mkdir projects
$ cd projects/
$ mkdir skeleton
$ cd skeleton
$ mkdir bin, NAME, tests, docs
```

**ai酱注：**这里原文是 `mkdir bin Name tests docs` 无法正常运行，作者本意是创建平行文件夹，所以用 `,` 隔开。

我用一个叫做 `projects` 的目录来存储所有我正在使用的变量。在这个目录下，我创建了 `skeleton` 目录，并把我项目的一些基础文件放了进去。这个 `NAME` 可以被重命名为任何你想给你项目的主模块取的名字。

接着，我们需要设置一些初始化文件，以下是 Linux/macOS 系统上的操作：

```
$ touch NAME/__init__.py
$ touch tests/__init__.py
```

以下是 Windows PowerShell 上的操作：

```
$ new-item -type file NAME/__init__.py
$ new-item -type file tests/__init__.py
```

这样就创建了一个空的 Python 模块目录，我们可以把我们的代码放进去。然后我们需要创建一个 `setup.py` 以便在之后需要的时候来安装我们的项目：

**ai酱注：**该文件创建在当前 `skeleton` 目录下，可以参考前述创建文件的命令来创建。

[setup.py](#)



```

1      try:
2          from setuptools import setup
3      except ImportError:
4          from distutils.core import setup
5
6      config = {
7          'description': 'My Project',
8          'author': 'My Name',
9          'url': 'URL to get it at.',
10         'download_url': 'Where to download it.',
11         'author_email': 'My email.',
12         'version': '0.1',
13         'install_requires': ['nose'],
14         'packages': ['NAME'],
15         'scripts': [],
16         'name': 'projectname'
17     }
18
19     setup(**config)

```

编辑这个文件，在其中填上你的联系信息，并且保证当你复制该文件的时候它能正常运行。

最后，你需要一个简单的骨架文件来测试，文件名为：`tests/NAME_tests.py`：

NAME\_tests.py

```

1      from nose.tools import *
2      import NAME
3
4      def setup():
5          print("SETUP!")
6
7      def teardown():
8          print("TEAR DOWN!")
9
10     def test_basic():
11         print("I RAN!")

```

## 46.3.1 最终目录结构

当你完成以上所有设置，你的目录应该像下面这样：

```
skeleton /  
  
NAME/  
    __init__.py  
bin/  
docs/  
setup.py  
tests/  
    NAME_tests.py  
    __init__.py
```

从现在开始，你应该从这个目录中运行你的命令。如果你看不到，可以输入 `ls -R`，如果你没看到同样的结构，那应该是搞错了当前目录。比如，人们通常在 `tests/` 下运行文件，这肯定行不通。要运行你的应用的测试，你需要处在 `tests/` 目录的上一层，所以如果你这样：

```
$ cd tests/          # WRONG! WRONG! WRONG!  
$ nosetests  
-----  
Ran 0 tests in 0.000 s OK
```

那就大错特错了，你得在 `tests` 的上一层目录。要是你犯了这个错误，你可以这样改正：

```
$ cd ..             # get out of tests/  
$ ls                # CORRECT! you are now in the right spot  
NAME      bin      docs      setup.py  
$ nosetests  
.  
-----  
Ran 1 test in 0.004s OK
```

记住这一点，因为人们经常犯这样的错误。

## 测试你的 Setup

当你安装好了所有东西之后，你应该可以运行这个：

```
$ nosetests  
.  
-----  
Ran 1 test in 0.007s OK
```

我会在下个练习中给你解释 `nosetests` 是做什么的，但是现在，你如果没看到这个，那你可能哪个地方搞错了。确保你把 `__init__.py` 文件放在了你的 `NAME` 目录和 `tests` 目录下面，并且确保你把 `tests/NAME_tests.py` 放在了正确的位置。

## 使用这个骨架

现在你已经完成了一连串的动作。任何时候当你想要开始一个新项目，只需要这样做：

1. 创建一个骨架目录的副本，以你的新项目名称命名。
2. 用你的新项目名重命名 `NAME` 目录，或者其他你想用的名字，来调用你的根模块。
3. 编辑你的 `setup.py` 文件，为你的新项目填入相应的信息。
4. 重命名 `tests/NAME_tests.py` 文件，跟你的模块文件保持一致。
5. 再次用 `nosetests` 确保所有文件都能正常运行。
6. 开始编写代码。

## 课后测试

这个练习没有附加练习，但是你必须完成一个测试：

1. 读一读如何使用你所安装的所有东西。
2. 读一读 `setup.py` 文件及其内容。警告：它不是一个写得很好的软件，所以可能会很难用。
3. 创建一个项目，并且开始把代码放入模块，然后让这个模块运行起来。
4. 在 `bin` 目录中放一个可以运行的脚本。读一读你如何能让一个 Python 脚本在你的系统中正常运行。
5. 在你的 `setup.py` 文件中加上你所创建的 `bin` 脚本，以使其得到安装。
6. 用你的 `setup.py` 来安装你自己的模块，确保它能正常运行，然后使用 `pip` 来卸载它。

## 常见问题

**这些指导适用于 Windows 吗？** 适用的，不过还取决于你的 Windows 版本。有些版本可能会让你在安装的时候遇到一些麻烦。你可以通过搜索来解决，或者找一个对 Python+Windows 比较有经验的朋友来帮你解决。

我应该在我的 `setup.py` 配置文件中放些什么呢？确保你阅读了该链接中的发布工具（distutils）文档：<http://docs.python3.6.org/distutils/setupscript.html>.

我没办法引入 `NAME` 模块，总是收到报错信息 `ImportError`。确保你创建了 `NAME/__init__.py` 文件。如果你用的是 Windows，确保你没有不小心把它命名成了 `NAME/__init__.py.txt`，某些编辑器会有这样的默认设置。

为什么我们需要一个 `bin/` 文件夹呢？这是一个用于存放在命令行运行的脚本的标准地方，它不是一个存放模块的地方。

我的 `nosetests` 运行结果只显示了一个 `test` 被运行。这是正确的吗？是的，我的输出结果也是这样的。

## 练习 47. 自动化测试

为了确认游戏功能是否正常，你需要一遍一遍地在你的游戏中输入命令。这个过程非常枯燥。如果能写一小段代码来测试你的代码岂不是更好？一旦你对程序做了任何修改，或者添加了什么新东西，你只要“跑一下你的测试”，这些测试就能确保程序依然能正常运行。这些自动测试不会捕捉到所有 bug，但是可以让你无需重复输入命令来运行你的代码，从而为你节约很多时间。

从这一节开始，以后每个练习将不再有“你会看到”这一部分，取而代之的是“你应该测试”（What You Should Test）部分。从现在开始，你需要为自己写的所有代码写自动化测试，这会让你成为一个更好的程序员。

我不会试图解释为什么你需要写自动化测试。我要告诉你的是，你想要成为一个程序员，而程序的作用是让无聊冗繁的工作自动化，测试软件毫无疑问是无聊冗繁的，所以你还是写点代码让它为你来做测试工作比较好。

这应该是你需要的所有的解释了。因为你写单元测试的原因是让你的大脑更加强健。你读了这本书，写了很多代码让它们实现一些事情。现在你将更进一步，写出能读懂你写的其他代码的代码。这个写代码来测试你写的其他代码的过程将强迫你清楚地理解你之前写的代码。同时清晰地了解这些代码实现的功能及其原理，从而让你对细节的注意更上一个台阶。

## 写一个测试用例（test case）

我们会拿一段非常简单的代码为例，写一个简单的测试，这个测试将建立在上节我们创建的项目

骨架上面。

首先，从你的项目骨架创建一个叫做 ex47 的项目。以下是你要遵循的步骤，我会通过语言描述来告诉你，而不是直接给你代码，这样可以给你思考的机会：

1. 把项目骨架复制到 ex47。
2. 把所有的 NAME 文件重命名为 ex47。
3. 把所有文件中的 NAME 单词替换为 ex47。
4. 最后，删除所有 \*.pyc 文件来确保你的文件夹是干净的。

如果你卡住了，可以回去查阅练习 46，如果没办法完成这些步骤，那你可能需要多练习几次。

**警告！**

记住，你通过运行 `nosetests` 命令来运行测试。你可以直接输入 `python3.6 ex47_tests.py` 来运行，但是没那么容易跑通，而且你需要为每个测试文件执行一次这个命令。

接下来，创建一个简单的文件 `ex47/game.py`，你可以把代码放入其中进行测试。这是一个非常小的类，其代码如下：

`game.py`

```
1      class Room(object):
2
3          def __init__(self, name, description):
4              self.name = name
5              self.description = description
6              self.paths = {}
7
8          def go(self, direction):
9              return self.paths.get(direction, None)
10
11          def add_paths(self, paths):
12              self.paths.update(paths)
```

一旦你有了这个文件，把单元测试骨架改成这样：

`ex47_tests.py`

```

1     from nose.tools import *
2     from ex47.game import Room
3
4
5     def test_room():
6         gold = Room("GoldRoom",
7                     """This room has gold in it you can grab. There's a
8                     door to the north.""")
9         assert_equal(gold.name, "GoldRoom")
10        assert_equal(gold.paths, {})
11
12    def test_room_paths():
13        center = Room("Center", "Test room in the center.")
14        north = Room("North", "Test room in the north.")
15        south = Room("South", "Test room in the south.")
16
17        center.add_paths({'north': north, 'south': south})
18        assert_equal(center.go('north'), north)
19        assert_equal(center.go('south'), south)
20
21    def test_map():
22        start = Room("Start", "You can go west and down a hole.")
23        west = Room("Trees", "There are trees here, you can go east.")
24        down = Room("Dungeon", "It's dark down here, you can go up.")
25
26        start.add_paths({'west': west, 'down': down})
27        west.add_paths({'east': start})
28        down.add_paths({'up': start})
29
30        assert_equal(start.go('west'), west)
31        assert_equal(start.go('west').go('east'), start)
32        assert_equal(start.go('down').go('up'), start)

```

这个文件引入了你在 `ex47.game` 模块中的 `Room` 类，这样你就能在这上面进行测试。然后是一系列以 `test_` 开头的函数来进行的测试。在每一个测试用例中都有一小段代码，它们会创建一个或多个房间，然后去确认房间的功能和你期望的是否一样。它先测试了基本的房间功能，然后测试了路径，最后测试了整个地图。

这里最重要的函数是 `assert_equal`，它保证了你设置的变量，以及你在 `Room` 里设置的路径和你的期望相符。如果你得到错误的结果，`nose` 将会打印出一个错误信息，这样你就可以找到出错的地方并修正过来。

# 测试指南

在测试时，你可以照着下面这些不是很严格的指南来做：

1. 测试脚本要放到 `tests/` 目录下，并且命名为 `BLAH_tests.py`，否则 `nosetests` 就不会执行你的测试脚本了。这样做还有一个好处就是防止测试代码和别的代码互相混淆。  
**ai酱注：**这里的 `BLAH_tests.py` 是一种调皮的写法（BLAH是废话的意思），你应该把 `BLAH` 替换为你的 `NAME`，在这个练习中就是 `ex47`。
2. 为你创建的每个模块写一个测试。
3. 测试用例（函数）尽量保持简短，但如果看上去不怎么整齐也没关系，测试用例一般都有点乱。
4. 就算测试用例有些乱，也要试着让他们保持整洁，把里边重复的代码删掉。创建一些辅助函数来避免重复的代码。当你下次在改完代码需要改测试的时候，你会感谢我这一条建议的。重复的代码会让修改测试变得很难操作。
5. 最后一条是别太把测试当回事。有时候，更好的方法是把代码和测试全部删掉，然后重新设计代码。

## 你会看到

Exercise 47 会话

```
$ nosetests
...
-----
Ran 3 tests in 0.008s OK
```

如果一切正常的话你应该会看到这个。试着搞一个错误，看看输出结果是什么，然后再把代码修改正确。

## 附加练习

1. 阅读 `nosetests` 相关文档，再去了解一下其他替代方案。
2. 了解一下 Python 的“doc tests”，看看你是不是更喜欢这种测试方式。
3. 改进你游戏里的 `Room`，然后用它重建你的游戏，这次重写，你需要一边写代码，

一边把单元测试写出来。

## 常见问题

我运行 `nosetests` 的时候收到了一个**语法错误 (syntax error)**。如果你收到这样的提示，看一下错误提示是怎么说的，然后修改出错的哪一行或者之前的一行。像 `nosetests` 这样的工具是运行你的代码和测试代码，所以它们会在运行 Python 的同时发现语法错误。

我为什么没办法引入 `ex47.game`？确认你创建了 `ex47/__init__.py` 文件，回到练习 46 看看如何创建。如果问题不是出在这儿，那么你可以这样做：

macOS/Linux 系统：

```
export PYTHONPATH=.
```

Windows 系统：

```
$env :PYTHONPATH = "$env :PYTHONPATH ; . "
```

最后，确保你是用 `nosetests` 来进行测试，而不是在用 Python。

我运行 `nosetests` 的时候看到了 `UserWarning`。你可能装了两个版本的 Python，或者你不是用的 `distribute`，回去跟着练习 46 安装一下 `distribute` 或者 `pip` 就可以了。

## 练习 48. 更复杂的用户输入

在之前的游戏中，你通过设置特定的字符串来控制用户的输入。比如，只有用户输入“run”，而且得是精确的“run”，游戏才能正常运行。他们要是输入了类似的短语，比如“run fast”，程序都会报错。但我们需要的是一个能让用户通过多种方式来输入的设备，同时我们可以把用户输入的内容转换成计算机能理解的语言。比如，我们可以让下面这些短语同样生效：

- open door
- open the door
- go THROUGH the door
- punch bear
- Punch The Bear in the FACE



我们应该允许用户在游戏中输入一些自然语言，并且要让游戏能读懂这些语言的含义。要做到这一点，我们需要写一个专门的模块。这个模块会有几个类一起工作来处理用户输入，这些输入会被转换成你的游戏可以可靠处理的东西。

一个简化版本的英文语言会包含以下要素：

- 用空格隔开的单词。
- 由单词构成的句子。
- 组织句子并形成含义的语法。

这意味着最先解决的问题应该是如何从用户那里获得单词，并判断这些单词是什么类型。

## 我们的游戏词汇表（lexicon）

我们需要为这个游戏创建一系列可以被接受的单词，我们把它称为“词汇表”（lexicon）：

- 方向词: north, south, east, west, down, up, left, right, back
- 动词: go, stop, kill, eat
- 停用词（stop word）: the, in, of, from, at, it
- 名词: door, bear, princess, cabinet
- 数词: any string of 0 through 9 characters（0-9字符的任何字符串）

名词方面有一个小问题，因为每个房间都可以有几组不同的名词。所以我们先选一小组词，随后再进行改进。

### 48.1.1 拆解句子

有了词汇表之后，我们就需要找一种方法来拆解句子，这样我们才能知道它们是什么。在这个例子中，我们已经定义了句子由“被空格分隔的单词”所组成。所以，我们只需要这样做：

```
stuff = input('> ')
words = stuff.split()
```

这就是我们现在为止要搞定的所有东西，不过这些也能管用好长一段时间。

### 48.1.2 词汇表元组（Lexicon Tuples）

知道了如何把一个句子分解成单词之后，我们只需要遍历这一列单词，并搞清楚它们是什么类型

即可。要做到这一点我们需要用到一个非常有用的小 Python 结构：元组（tuple）。元组是一个你不能修改的列表。把数据放进两个 `()` 中，并像列表一样用逗号隔开，就能创建一个元组：

```
first_word = ( 'verb ' , 'go ' )
second_word = ( ' direction ' , ' north ' )
third_word = ( ' direct ion ' , ' west ' )
sentence = [ first_word , second_word , third_word ]
```

这样就创建了一对 (类型, 单词)，你可以看着单词来进行操作。

这只是一个例子，不过基本上也是最终结果。你从用户那里获得原始输入，将其分割成单词，再分析这些单词以确定它们的类型，最后，将它们组成一个句子。

### 48.1.3 扫描输入

现在你可以开始写你的扫描器（scanner）了。这个扫描器会从用户那里获取一个原始输入字符串，并返回一个由一系列 (TOKEN, WORD) 元组对组成的句子。如果一个单词不是词汇表的一部分，那么它应该仍然返回该单词，但是将 TOKEN 设置为错误标记，从而告诉用户他们搞错了。

这块很有意思，但我不会告诉你怎么做。相反，我会写一个“单元测试”，你来写扫描器，来保证单元测试能够正常运行。

### 48.1.4 异常和数字

有个小地方我要先帮一下你，那就是数字的转换。不过，要做到这一点，我们需要先使用一下欺骗（cheat）和异常（exceptions）。异常是你运行一些函数的时候收到的错误情况。或者说，就是当函数遇到错误的时候，它会“抛出”（raise）一个异常，然后你需要处理这个异常。比如，如果你在 Python 里输入这个，你就会收到一个异常：

练习 48 Python 会话

```
Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwi Type "help", "copyright", "credi
>>> int("hell")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hell'
```

这个 `ValueError` 就是 `int()` 函数抛出来的一个异常，因为你放进 `int()` 里面的不是一个数

字。这个 `int()` 函数本来应该给你返回一个值，告诉你它遇到了一个错误。但是，因为它只能返回整数，所以它很难直接告诉你。它不能返回 -1，因为这是一个数字。所以，与其绞尽脑汁地思考遇到错误的时候应该返回什么，`int()` 函数直接抛出了一个 `ValueError` 异常让你来处理。

你可以通过使用 `try` 和 `except` 关键词来处理异常：

ex48\_convert.py

```
1     def convert_number(s):
2         try:
3             return int(s)
4         except ValueError:
5             return None
```

你把你想要“try”的代码放在 `try` 区域，然后把出现错误后要运行的代码放在 `except` 区域。在这个例子中，我们想要尝试对某个数字调用 `int()` 函数，如果出错了，我们“捕获”（catch）这个错误，然后返回 `None`。

在你写的扫描器里，你可以用这个函数来测试一个东西是不是数字，你还应该在声明一个单词是错误的之前，把这个作为最后一道检验来执行一下。

## 一个测试优先挑战

测试优先（Test first）是编写自动化测试时用到的一个编程策略，这个策略先假装你的代码能正常运行，然后你再去写代码，从而让这个测试能真正运行。当你无法可视化代码的实现过程，但是又能够想象出自己会如何做的时候，这种方法会很有效。比如，如果你知道如何在另一个模块中使用一个新类，但是你还不太知道如何实现这个类，那么你可以先写测试代码。

接下来，你要用我提供给你的一个测试来写代码，并让它正常运行。要完成这个练习，你需要遵循如下步骤：

1. 先完成我给你的测试中的一小部分。
2. 看它会正常运行还是会报错，这样你就能知道这个测试其实就是在确认一个特性能否正常运行。
3. 在你的源文件 `lexicon.py` 里写代码，让这个测试能够运行通过。
4. 重复这个过程直到你把测试中的所有东西都完全实现。

当你到第三步的时候，也可以结合我们写代码的其他方法：

1. 如果你需要的话，创建一些“骨架”函数或者类。
2. 在其中写一些注释，解释函数是如何运行的。
3. 把注释描述的东西用代码写出来。
4. 移除和代码重复的注释。

这种写代码的方法被称为“伪代码”（psuedo code），如果你不知道如何实现某些东西，但是可以用自己的话来描述它，那么这种方法就非常有效。

把“测试优先”和“伪代码”策略相结合，我们就有了这个编程的简要步骤：

1. 写一些会失败的测试代码。
2. 编写测试所需要的骨架函数/模块/类。
3. 用自己的话通过注释来填充这些骨架，解释它是如何运行的。
4. 用代码来替换注释，直到测试能运行通过。
5. 重复。

在这个练习中，你会通过我给你的测试，实现 `lexicon.py` 模块，来练习这个方法。

## 你需要测试

以下是你需要用到的测试用例 `tests/lexicon_tests.py`，但是先别输入：

`lexicon_tests.py`

```

1  from nose.tools import *
2  from ex48 import lexicon
3
4
5  def test_directions():
6      assert_equal(lexicon.scan("north"), [('direction', 'north')])
7      result = lexicon.scan("north south east")
8      assert_equal(result, [('direction', 'north'),
9                             ('direction', 'south'),
10                            ('direction', 'east')])
11
12  def test_verbs():
13      assert_equal(lexicon.scan("go"), [('verb', 'go')])
14      result = lexicon.scan("go kill eat")
15      assert_equal(result, [('verb', 'go'),
16                             ('verb', 'kill'),
17                             ('verb', 'eat')])
18
19
20  def test_stops():
21      assert_equal(lexicon.scan("the"), [('stop', 'the')])
22      result = lexicon.scan("the in of")
23      assert_equal(result, [('stop', 'the'),
24                             ('stop', 'in'),
25                             ('stop', 'of')])
26
27
28  def test_nouns():
29      assert_equal(lexicon.scan("bear"), [('noun', 'bear')])
30      result = lexicon.scan("bear princess")
31      assert_equal(result, [('noun', 'bear'),
32                             ('noun', 'princess')])
33
34  def test_numbers():
35      assert_equal(lexicon.scan("1234"), [('number', 1234)])
36      result = lexicon.scan("3 91234")
37      assert_equal(result, [('number', 3),
38                             ('number', 91234)])
39
40
41  def test_errors():
42      assert_equal(lexicon.scan("ASDFADFASDF"),
43                  [('error', 'ASDFADFASDF')])
44      result = lexicon.scan("bear IAS princess")
45      assert_equal(result, [('noun', 'bear'),
46                             ('error', 'IAS')],

```

你可能想用这个项目骨架来创建一个新项目，就像练习 47 中一样。那么你需要创建这个测试用例，以及它所用的 `lexicon.py` 文件。看看这个测试用例的最上面，它是如何引入模块以及作何用途的。

接下来，按照我给你的步骤，编写这个测试的一部分。比如这是我所做的：

1. 写出最上面的 `import` 部分，让它运行。
2. 创建第一个测试用例 `test_directions` 的空版本，确保它能正常运行。
3. 写出 `test_directions` 测试用例的第一行，让它运行失败。
4. 然后去 `lexicon.py` 文件，创建一个空的 `scan` 函数。
5. 运行测试，确保至少 `scan` 能成功运行，哪怕整体运行失败。
6. 填入伪代码注释，解释 `scan` 如何工作，并让 `test_directions` 运行通过。
7. 撰写与注释匹配的代码，直到 `test_directions` 运行通过。
8. 回到 `test_directions` 并撰写余下各行。
9. 回到 `lexicon.py` 的 `scan` 函数，撰写其内容，并让新的测试代码通过。
10. 完成之后，你就有了你的第一个运行通过的测试，之后你就可以移步下一个测试。

只要你依照这个步骤依次完成一小块，你就可以成功地把一个大问题分解成很多可以解决的小问题。这个过程就像把攀登一座高峰转化为跨越一座座小山。

## 附加练习

1. 改进单元测试，确保你能测试更多的词汇表。
2. 丰富词汇表，再更新单元测试。
3. 确保你的扫描器能处理用户输入的大小写，更新测试，确保这一点能成功运行。
4. 找找其他可以转化数字的方法。
5. 我的解决方案有 37 行，你的更长还是更短呢？

## 常见问题

为什么我一直收到 `ImportErrors` ？ `ImportErrors` 通常由四种情况造成：

1. 你没有在包含模块的目录中创建 `__init__.py`。
2. 你处在错误的目录下。

3. 你因为拼写错误引入了错误的模块。
4. 你的 PYTHONPATH 没有设置到 `.`，所以你无法从当前目录加载模块。

`try-except` 和 `if-else` 的区别是什么？`try-except` 仅用于处理异常，绝对不要将它作为 `if-else` 使用。

有没有办法让游戏在等待用户输入的时候不间断地运行？我猜想你是想把游戏做得更高级，当用户反应过慢就被怪物杀死之类的。这个是可以做到，不过需要用到更高级的模块和编程技巧，本书不会涉及这些内容。

## 练习 49. 创建句子

从我们这个小游戏的词汇扫描器中，我们应该可以得到类似下面的列表：

Exercise 49 Python 会话

```
Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin Type "help", "copyright", "credits()" or "quit()" for more
>>> from ex48 import lexicon
>>> lexicon.scan("go north")
[('verb', 'go'), ('direction', 'north')]
>>> lexicon.scan("kill the princess")
[('verb', 'kill'), ('stop', 'the'), ('noun', 'princess')]
>>> lexicon.scan("eat the bear")
[('verb', 'eat'), ('stop', 'the'), ('noun', 'bear')]
```

以上对更长的句子也管用，比如：

如：`lexicon.scan("open the door and smack the bear in the nose")`。

现在让我们把这个转换成游戏可以使用的东西，比如句子类（Sentence class）。不知你是否还记得小学时候学过一个句子的简单结构：

主语(Subject) + 谓语(动词 Verb) + 宾语(Object)

显然，实际的句子比这个复杂，你可能已经在英语语法课上被搞得头大。我们的目的，是将上面的元组列表转换为一个 Sentence 对象，而这个对象又包含主谓宾各个要素。

# 匹配和窥探（Peek）

为此我们需要四样工具：

1. 循环访问元组列表的方法，这挺简单的。
2. 匹配我们的主谓宾设置中不同种类元组的方法。
3. 一个“窥视”潜在元组的方法，以便做决定时用到。
4. 跳过(skip)我们不在乎的内容的方法，比如停用词（stop word）。
5. 一个用以存放结果的句子类。

我们要把这些函数放到一个叫做 `ex48.parser` 模块中（将该文件命名为 `ex48/parser.py`），以方便对其进行测试。我们使用 `peek` 函数来执行“查看元组列表中的下一个元素，然后做匹配、取出来并进行处理”这一系列动作。

## 句子语法

在写代码之前，你需要先理解一下英语句子的基本语法。在我们的语法解析器（parser）中，我们想要产生一个包含三种属性的句子对象：

`Sentence.subject` 这是任何句子的主语，但是大多数时候可以默认为“玩家”（player），因为比如“run north”其实就是“player run north”。这应该是一个名词。

`Sentence.verb` 这是句子的动作。在“run north”中，就是“run”。这是一个动词。

`Sentence.object` 这是另一个名词，指的是动作所作用的对象（即宾语，object）。在我们的游戏中，我们所分的方向就是宾语。所以在“run north”里面，这个“north”就是宾语。在“hit bear”里面，“bear”就是宾语。

然后，我们的解析器需要使用我们所描述的函数，给出的扫描过的句子，把它转换成一系列句子对象来和输入内容进行匹配。

## 关于异常

你已经简单学过一些关于异常的东西，但还没学过怎样“抛出”(raise)异常。这节的代码就演示了如何抛出前面定义的 `ParserError`。注意，系统用类来赋予异常的类型。另外还要注意我们是如何使用 `raise` 这个关键字来抛出异常的。



你的测试代码也应该要测试到这些异常，我随后会演示给你看如何实现。

## 解析器代码（The Parser Code）

如果你想要额外的挑战，现在就停下来，试着根据我的描述来写。如果遇到问题，你可以回来看看我是如何做的，但是尝试自己实现解析器是很好的实践。现在我会过一遍代码，以便你可以将其输入到 `ex48/parser.py` 中。我们以一个解析错误异常来开始我们的解析器：

`parser.py`

```
1 class ParserError(Exception):
2     pass
```

这也是你如何创建你自己的 `ParserError` exception 类的方法。下面，我们需要创建 Sentence object：

`parser.py`

```
1 class Sentence(object):
2
3     def __init__(self, subject, verb, obj):
4         # remember we take ('noun','princess') tuples and convert them.
5         self.subject = subject[1]
6         self.verb = verb[1]
7         self.object = obj[1]
```

这些代码目前为止没什么特别的。你只是在创建简单的类。

**ai酱注：**接下来的这些函数不需要缩进，它们不是 Sentence 类下面的函数，而是独立的函数！

在我们的问题描述中，我们需要一个能够“窥探”一系列单词并返回其类型的函数：

`parser.py`

```
1 def peek(word_list):
2     if word_list:
3         word = word_list[0]
4         return word[0]
5     else:
6         return None
```

我们之所以需要这个函数，是因为我们得基于下一个词是什么来判断我们正在处理的句子是什么类型。然后我们可以调用另一个函数来消灭（consume）那个字并往下进行。

要消灭一个单词，我们要用到 `match` 函数，这个函数可以确认当前单词是不是正确的类型，是的话就把它从列表中拿出来，然后返回这个单词。

[parser.py](#)

```
1     def match(word_list, expecting):
2         if word_list:
3             word = word_list.pop(0)
4
5             if word[0] == expecting:
6                 return word
7             else:
8                 return None
9         else:
10            return None
```

同样的，这个也非常简单，但是你要确保你能理解这些代码。还要确保你能理解我为什么要用这种方式来实现它。我需要窥探列表中的单词来决定我正在处理的句子是什么类型，然后我需要匹配这些单词来创建我的 Sentence。

我需要的最后一个东西是跳过对句子无用的单词的方法。这些单词被标记为“stop words” (type 'stop')，比如“the”、“and”、和“a”等。

[parser.py](#)

```
1     def skip(word_list, word_type):
2         while peek(word_list) == word_type:
3             match(word_list, word_type)
```

记住，skip 不只跳过一个单词，它会跳过所有它所找到的那个类型的单词。比如，如果有人输入 “scream at the bear”，你只会得到“scream”和“bear”这两个词。

这是我们解析函数的基本设定，有了这个函数，我们就可以解析任何我们想要解析的文本。这个解析器非常简单，所以剩余的函数也很简短。

首先，我们可以试着解析一个动词：

[parser.py](#)

```

1     def parse_verb(word_list):
2         skip(word_list, 'stop')
3
4         if peek(word_list) == 'verb':
5             return match(word_list, 'verb')
6         else:
7             raise ParserError("Expected a verb next.")

```

我们跳过了任何的 stop words，然后提前进行了窥探，确保下一个单词是“verb”（动词）类型。如果不是，就会抛出 `ParserError` 并说明原因。如果是“verb”，那就进行匹配，并把它从列表中拿出来。处理宾语的函数同理：

[parser.py](#)

```

1     def parse_object(word_list):
2         skip(word_list, 'stop')
3         next_word = peek(word_list)
4
5         if next_word == 'noun':
6             return match(word_list, 'noun')
7         elif next_word == 'direction':
8             return match(word_list, 'direction')
9         else:
10            raise ParserError("Expected a noun or direction next.")

```

同样地，跳过 stop words，先窥探，然后基于内容决定句子是否正确。尽管在 `parse_object` 函数中，我们需要同时处理“noun”（名词）和“direction words”（方向词）作为可能的宾语。主语也是一样，但是因为我们想要用隐含的“player”名词，所以我们要这样用 peek：

[parser.py](#)

```

1     def parse_subject(word_list):
2         skip(word_list, 'stop')
3         next_word = peek(word_list)
4
5         if next_word == 'noun':
6             return match(word_list, 'noun')
7         elif next_word == 'verb':
8             return ('noun', 'player')
9         else:
10            raise ParserError("Expected a verb next.")

```

这些都准备好了以后，我们最终的 `parse_sentence` 函数会非常简单：

`parser.py`

```
1 def parse_sentence(word_list):
2     subj = parse_subject(word_list)
3     verb = parse_verb(word_list)
4     obj = parse_object(word_list)
5
6     return Sentence(subj, verb, obj)
```

## 玩一玩解析器

要看这个如何运行，你可以这样做：

练习 49a Python 会话

```
Python 3.6.0 (default, Feb 2 2017, 12:48:29)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwi Type "help", "copyright", "credi
>>> from ex48.parser import *
>>> x = parse_sentence([('verb', 'run'), ('direction', 'north')])
>>> x.subject
'player'
>>> x.verb
'run'
>>> x.object
'north'
>>> x = parse_sentence([('noun', 'bear'), ('verb', 'eat'), ('stop', 'the'),
...                     ('noun', 'honey')])
>>> x.subject
'bear'
>>> x.verb
'eat'
>>> x.object
'honey'
```

**ai酱注：**这里要先切换到 `skeleton` 目录，在运行 `python`，因为引入模块那里是从 `ex48.parser` 导入的，说明不能在 `ex48` 这个目录下运行。

试着把句子映射成句子中正确的对，比如，你会怎么说“the bear run south”？

# 你需要测试

对于练习 49，编写一个完整的测试，以确认代码中的所有内容都是有效的。把测试放在 `tests/parser_tests.py` 中，就像上个练习中的测试文件那样。还要试着给解析器错误的句子来产生异常。

通过使用 nose 文档中的 `assert_raise` 函数来检查异常。学习如何使用它，这样你就可以编写预期会失败的测试，这在测试中是非常重要的。通过阅读 nose 文档来了解这个功能（以及其他功能）。

完成之后，你应该知道这段代码是如何工作的，以及如何为其他人的代码写测试，即使他们不希望你这样做。相信我，这是一个非常有用的技能。

## 附加练习

1. 改变 `parse_methods`，试着把它们放到一个类中，而不是只当做方法来用。你更喜欢哪种设计？
2. 提高 parser 对于错误输入的抵御能力，这样即使用户输入了你预定义语汇之外的词语，你的程序也能正常运行下去。
3. 改进语法，让它可以处理更多的东西，例如数字。
4. 想想在游戏里你的 `Sentence` 类可以对用户输入做哪些有趣的事情。

## 常见问题

`assert_raises` 老是弄不对。确认你写成了 `assert_raises(exception, callable, parameters)` 而不是 `assert_raises(exception, callable(parameters))`。注意第二个格式，它所做的其实是将函数的返回值作为参数传到 `assert_raises` 中，这样做是错误的。你必须把函数和它的参数分别传入 `assert_raises` 中。

## 练习 50. 你的第一个网站

最后这三个练习会非常难，你可能得多花点时间。第一个练习是要你给你的游戏创建一个简单的网页。在你尝试这个练习之前，你必须已经成功完成了练习 46，正确安装了 pip，并且学会了

如何安装软件包以及如何创建项目框架。如果你不记得这些内容，就回到《习题 46》重新复习一遍。

## 安装 Flask

在创建你的第一个网页应用程序之前，你需要安装一个叫做 flask 的“Web 框架”（web framework），所谓的“框架”通常是指“让某件事情做起来更容易的软件包”。在网页应用的世界里，人们创建了各种各样的“网页框架”，用来解决他们在创建网站时碰到的问题，然后把这些解决方案用软件包的方式发布出来，这样你就可以利用它们引导创建你自己的项目了。

在这个练习中，我们会使用 flask 框架，但是可选的框架类型有很多很多，不过在这里我们会使用 flask 框架。你可以先学会它，等到差不多的时候再去接触其它的框架（也可以一直用 flask，因为它真的很好用。）

用 pip 安装 flask:

```
$ sudo pip install flask
[ sudo ] password for zedshaw: Downloading/unpacking flask
Running setup.py egg_info for package flask

Installing collected packages : flask Running setup.py install for flask

Successfully installed flask Cleaning up...
```

这是 Linux 和 macOS 电脑上的操作，如果是 windows，把 `sudo` 去掉，直接输入 `pip install flask` 即可。如果不行的话，回到练习 46，确保你把之前的步骤都做好了。

## 创建一个简单的“Hello World”项目

现在你要创建一个非常简单的“Hello World”网页应用，并且会用 flask 来创建项目目录。首先，创建你的项目目录：

```
$ cd projects
$ mkdir gothonweb
$ cd gothonweb
$ mkdir bin gothonweb tests docs templates
$ touch gothonweb/__init__.py
$ touch tests/__init__.py
```

**ai酱注：** 这里是 Linux/macOS 的命令，Windows 要换成 `new-item` 来创建新文件，具体可以参考练习 46。

你可以把练习 43 的游戏拿过来，把它做成一个网页应用，这也是为什么我们这个项目叫做 gothonweb。不过在你做之前，我们需要先创建好 flask 的基本应用。把下面这些代码输入到 `app.py` 中：

`ex50.py`

```
1     from flask import Flask
2     app = Flask(__name__)
3
4     @app.route('/')
5     def hello_world():
6         return 'Hello, World!'
7
8     if __name__ == "__main__":
9         app.run()
```

然后像这样运行这个应用：

```
(lpthw) $ python3.6 app.py
*       Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

最后，用你的浏览器打开 <http://localhost:5000/> 这个网址，你就可以看到两样东西。第一个是在你的浏览器中，你会看到 Hello, World!，第二个是在你的终端，你会看到新的输出内容：

```
(lpthw) $ python3.6 app.py
*       Running on http://127.0.0.1:5000/ (Press CTRL+C to quit) 127.0.0.1 -- [22/Feb/2017 14:28:
127.0.0.1 -- [22/Feb/2017 14:28:50] " GET /favicon.ico HTTP/1.1" 404-
127.0.0.1 -- [22/Feb/2017 14:28:50] " GET /favicon.ico HTTP/1.1" 404-
```

这些是 flask 为你打印的日志消息，这样你就能看到服务器在正常工作，以及浏览器在屏幕后面都做了什么。这些日志信息可以帮助你调试代码，告诉你哪里出错了。比如，它会说，你的浏览器尝试获取了 `/favicon.ico`，但是这个文件不存在，所以它返回了 `404 Not Found` 状态码。

我还没解释这种网页运行的原理，因为我想先让你做好准备，这样我可以在之后的两个练习中更好地给你解释。要实现这一点，我得让你用几种不同的方式来把你的 flask 应用拆解开来，然后再重组，这样你就能明白它是如何建立起来的了。

# 发生了什么？

当你的浏览器触发了你的应用，发生了这些事情：

1. 浏览器通过网络连接到你自己的电脑，它的名字叫做 localhost，这是一个标准称谓，表示不管我的计算机在网络上叫什么名字，都可以用 localhost 来访问。它用到的网络端口是 5000。
2. 连接成功以后，浏览器对 `app.py` 这个应用程序发出了 HTTP 请求(request)，要求访问 `/ URL`，这通常是一个网站的第一个 URL。
3. 在 `app.py` 里有一个列表，里边包含了 URL 和类的匹配关系。我们这里只定义了一组匹配，那就是 `/, 'index'` 的匹配。它的含义是：如果有人用浏览器访问 `/` 这一级目录，flask 就会找到这个 `def index`，然后用它来处理这个浏览器请求。
4. flask 找到 `def index` 以后，就会调用它来实际处理这个请求。函数运行之后会返回一个字符串，以供 flask 发送给浏览器。
5. 最终，flask 处理了这个请求，并将这个响应发送给了浏览器，正如你所看到的那样。

确保你真正理解了以上这些，画一个流程图，展示一下信息是如何从浏览器去到 flask，再到 `def index`，最后又返回到你的浏览器的。

---

**ai酱注：** 以上这 5 条解释建议大家忽略，因为老肖的解释跟上面 `app.py` 里面的代码有点对不上，这些解释是旧版书中的解释，因为旧版书的这一节用了不同的方法，所以代码完全不同，但是老肖估计忘了修改这一块内容了。推荐大家去 flask 官网学习一下 [Flask 官方文档](#)，里面有对这段代码的解释：

1. 首先，我们导入了 Flask 类。这个类的实例将会是我们的 WSGI（Web服务器网关接口，Python Web Server Gateway Interface，缩写为WSGI）应用程序。
2. 接下来，我们创建一个该类的实例，第一个参数是应用模块或者包的名称。如果你使用单一的模块（如本例），你应该使用 `__name__`，因为模块的名称将会因其作为单独应用启动还是作为模块导入而有不同（也即是 `'__main__'` 或实际的导入名）。这是必须的，这样 Flask 才知道到哪去找模板、静态文件等等。详情见 Flask 的文档。
3. 然后，我们使用 `route()` 装饰器告诉 Flask 什么样的 URL 能触发我们的函数。  
(装饰器可以参考廖雪峰老师的讲解：[装饰器](#))



4. 这个函数的名字也在生成 URL 时被特定的函数采用，这个函数返回我们想要显示在浏览器中的信息。
5. 最后我们用 `run()` 函数来让应用运行在本地服务器上。其中 `if __name__ == '__main__':` 确保服务器只会在该脚本被 Python 解释器直接执行的时候才会运行，而不是作为模块导入的时候。

## 修正错误

首先，把第 8 行的 `greeting` 变量赋值删掉，然后刷新浏览器。再用 `CTRL-C` 关掉 flask 再重启，当它再次运行的时候，刷新你的浏览器，你应该会看到一个“Internal Server Error”。回到终端，你会看到这个（`[VENV]` 是你的 `.venvs/` 目录的路径。）：

**ai酱注：**上面的代码中并没有 `greeting` 变量，我在网上找到了另一个版本的 [LP3THW](#)，其 `app.py` 的代码跟我们在用的这一版略有区别，`def hello_world()` 下面是两行内容，第 1 行：`greeting = "World"`，第 2 行：`return f'Hello, {greeting}!'`。大家可以修改一下再按照上面一段内容进行操作。

```
(lpthw) $ p`ython3.6 app.py
*       Running on http://127.0.0.1 :5000/ (Press CTRL+C to quit) [2017-02-22 14:35:54,256] ERROR
File "[VENV]/site-packages/flask/app.py", line 1982, in wsgi_app
response = self.full_dispatch_request() File "[VENV]/site-packages/flask/app.py",
line 1614, in full_dispatch_request rv = self . handle_user_exception(e)
File      "[VENV]/site-packages/flask/app.py", line 1517, in handle_user_exception reraise(exc_t
File "[VENV]/site-packages/flask/_compat.py", line 33, in reraise
raise value
File "[VENV]/site-packages/flask/app.py", line 1612, in full_dispatch_request
rv = self.dispatch_request()
File "[VENV]/site-packages/flask/app.py", line 1598, in dispatch_request
return self.view_functions[rule.endpoint](**req.view_args)
File "app.py", line 8, in index
return render_template("index.html", greeting=greeting) NameError: name 'greeting'
127.0.0.1--[22/Feb/2017 14:35:54] "GET / HTTP/1.1" 500-
```

这样可以正常运行，不过你也可以在“调试模式”（debugger mode）下运行 Flask。你会得到一个更好的错误页面，以及更多的有用信息。不过，调试模式的问题是它在互联网上运行不够安全，因此，你必须像这样打开它：

```
(lpthw) $ export FLASK_DEBUG=1
(lpthw) $ python3.6 app.py
*      Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
*      Restarting with stat
*      Debugger is      active!
*      Debugger pin code: 222-752-342
```

**ai酱注：**上面第一行是 linux/macOS 下的命令，windows 系统输入 `set FLASK_DEBUG=1`。（可是我运行之后页面跟原来的一样，并没有多出什么信息来（摊手））

然后，刷新你的浏览器，你就可以得到一个更详细的页面，上面有可以用来调试这个应用的信息，还有一个实时控制台来帮你查找更多的信息。

**警告！**

正是 Flask

实时控制台和升级版的输出信息才让调试模式在网上变得危险。有了这些信息，攻击者可以完全远程控制 `FLASK_DEBUG` 变得很难。当然，你也很容易为了在开发中省掉一个步骤而黑掉这个启动选项，但是这会作服务器，从而使它变成一个真正的入侵，而不只是你在疲惫夜晚的一个犯懒之举。

## 创建基本的模板

你可以把你的 flask 应用拆解开，但是不知你是否注意到。“Hello World”并不是一个很好的 HTML 页面？这是一个网页应用，因此它需要一个适当的 HTML 响应。要实现这一点，你需要创建一个简单的模板，让“Hello World”以一个大的绿色字体呈现。

第一步是创建一个如下的 `templates/index.html` 文件：

index.html

```

<html>
  <head>
    <title>Gothons Of Planet Percal #25</title>
  </head>
  <body>

    {% if greeting %}
    I just wanted to say
    <em style="color: green; font-size: 2em;">{{ greeting }}</em>
    {% else %}
    <em>Hello</em>, world!
    {% endif %}

  </body>
</html>

```

如果你知道 HTML 是什么，那这些代码你应该看起来很熟悉。如果不知道的话，搜一下 HTML，试着亲手写一些网页，这样你就能知道它是如何运行的。这个 HTML 文件只是一个模板，这意味着 flask 会将你传给它的变量放进模板中的这些“洞”（holes）里。每个 `$greeting` 所在的地方都代表一个变量，你可以传给这个模板来改变它的内容。

要让你的 `app.py` 做到这些，你需要添加一些代码来告诉 flask 从哪里加载这个模板以及如何渲染它。把 `app.py` 拿过来，做如下改动：

### app.py

```

1      from flask import Flask
2      from flask import render_template
3
4      app = Flask(__name__)
5
6      @app.route("/")
7      def index():
8          greeting = "Hello World"
9          return render_template("index.html", greeting=greeting)
10
11     if __name__ == "__main__":
12         app.run()

```

注意一下新的 `render` 变量。以及我如何改变 `index.GET` 的最后一行，使得它能返回 `render.index()`，并传入你的 `greeting` 变量。

有了这些之后，在浏览器中重新加载 web 页面，你应该会看到一条绿色的消息。你还应该能够

在浏览器的页面上“查看源代码”，以查看它是否是有效的 HTML。

这些可能对你来说有点太快了，所以我会给你解释一下模板的原理：

1. 在 `app.py` 中，你在最开始引入了一个名为 `render_template` 的新函数。
2. 这个 `render_template` 知道如何从 `templates/` 目录中加载 `.html` 文件，因为这是一个 Flask 应用的默认设置。
3. 在你后面的代码中，当浏览器触发了 `def index` 之后，它没有再返回简单的 `greeting` 字符串，取而代之的是调用了 `render_template`，并且将问候语句作为一个变量传递给它。
4. 然后，这个 `render_template` 方法加载了 `templates/index.html` 文件（虽然你没有明确说它是模板）并进行了处理。
5. 在这个 `templates/index.html` 文件中，你有了看起来像普通 HTML 的文件，但是还有代码放在两种标记中。一种是 `{% %}`，代表了“可执行代码”（executable code）（if 语句、for 循环等）；另一个是 `{{ }}`，代表了要被转化为文本并放到 HTML 输出结果中的变量。`{% %}` 可执行代码不会显示在 HTML 中。要了解更多关于模板语言的内容，可以阅读 [Jinja2 文档](#)。

要深入理解这个过程，你可以修改 `greeting` 变量以及 HTML 模板的内容，看看会有什么效果。然后创建一个叫做 `templates/foo.html` 的模板，并用前面讲到的方法来渲染它。

## 附加练习

1. 阅读这个文档：<http://flask.pocoo.org/docs/0.12/>，它和 flask 项目是一样的。（**ai酱注**：中文版传送门：[Flask官方文档](#)）
2. 实验一下你在上述网站看到的所有东西，包括里边的示例代码。
3. 阅读一下 HTML5 和 CSS3 相关的东西，自己练习写几个 `.html` 和 `.css` 文件。
4. 如果你有一个懂 Django 的朋友，并且愿意帮你的话，你可以试着使用 Django 完成一下习题  
50、51、52，看看结果会是什么样子。

## 常见问题

我好像无法连接 <http://localhost:5000/>。试一下  
<http://127.0.0.1:5000/> 这个网址。

**flask 和 web.py 的区别是什么？** 没有区别。我只是“锁定”（locked）了一个特定版本的 web.py，以使它对学生来说保持一致，然后把它叫做 flask。之后的 web.py 版本可能跟这一版不一样。

我找不到 index.html（以及其他相关的东西）。你可能是先做了 cd bin/，然后才运行这个项目。别这样做，所有的命令和指导都假设你处于 bin/ 的上一级目录中，所以如果你不能输入 python3.6 app.py，那你就是处在错误的目录下。

**为什么调用 template 时要写 greeting=greeting ？** 这一句并不是赋值给 greeting，而是将一个命名参数传到模板中。这也算是一种赋值，不过只会在模板函数的调用中生效。

我无法使用 5000 端口。可能是哪个杀毒软件占用了这个端口，那就换一个端口。

**安装 flask 时出现 ImportError "No module named web"。** 很有可能是你在系统中安装了多个版本的 Python，而在这里你用了错误的一个。或者由于 pip 版本太旧导致安装没有正确完成。试着卸载并重装 flask。如果还不行，那就再仔细检查确认自己用了正确版本的 Python。

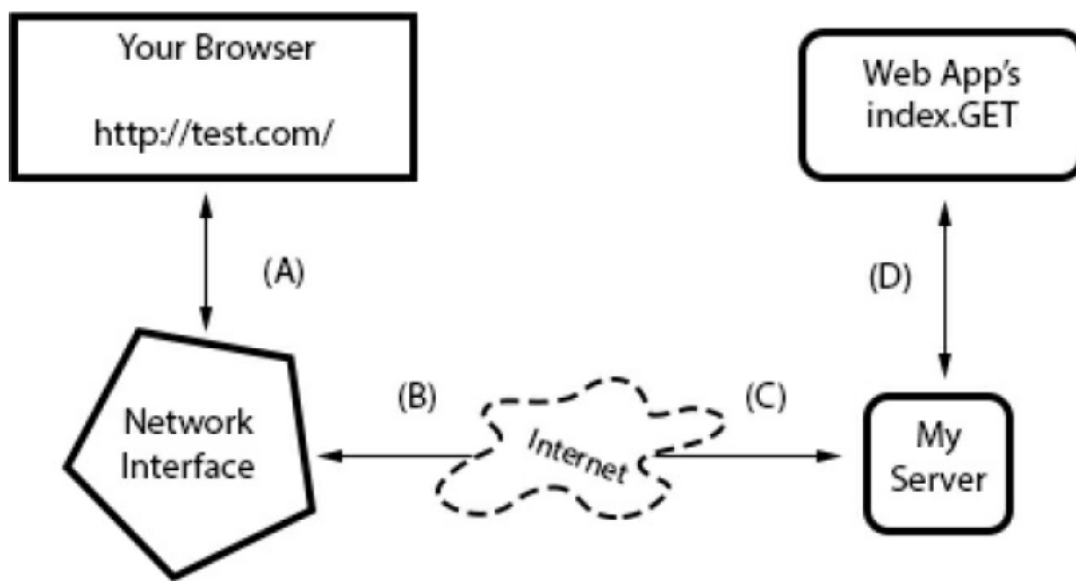
## 练习 51. 从浏览器获取输入

虽然能让浏览器显示“Hello World”是件很激动人心的事情，但是如果能让用户通过表单(form)向你的应用程序提交文本，那就更令人兴奋了。在这个练习中，我们会使用 form 改进你的 web 程序，并且将用户相关的信息保存到他们的“会话(session)”中。

## Web 是如何工作的？

该学点无趣的东西了。在创建 form 前你需要先多学一点关于 web 的工作原理。这里的描述并不完整，但是相当准确，在你的程序出错时，它会帮你找到出错的原因。另外，如果你理解了 form 的应用，那么创建 form 对你来说就会更容易。

我会从一个简单的图示讲起，它向你展示了 web 请求的不同部分，以及信息传递的大致流程：为了方便讲述一个常规请求（request）的流程，我在每条线上面加了字母标签以作区别：



1. 你在浏览器输入网址 <http://test.com/>，它会通过你电脑的网络设备发送请求（线路 A）。
2. 你的请求被传送到互联网（线路 B），然后再抵达远程服务器（线路 C），然后我的服务器会接受这个请求。  
**ai酱注：** 这里之所以用“my server”是因为旧版书中，作者举例用的链接是 <http://learnpythonthehardway.org/>，这是作者自己的网站，所以对应也会指向他的服务器。在新版书中，虽然更换了链接，但是作者并没有对这里的表述加以更正。
3. 我的服务器接受请求后，我的 web 应用程序就会去处理这个请求（线路 D），然后我的 Python 代码会去运行 `index.GET` 这个“处理程序(handler)”。
4. 在代码 `return` 的时候，我的 Python 服务器就会发出响应(response)，这个响应会再通过线路 D 传递到你的浏览器。
5. 运行这个网站的服务器会从线路 D 获得响应，然后服务器将这个网站通过线路 C 传回至互联网。
6. 响应通过互联网由线路 B 传至你的计算机，计算机的网卡再通过线路 A 将响应传给你的浏览器。
7. 最后，你的浏览器显示了这个响应的内容。

这段描述中有几个术语需要你了解一下，以便你在谈论 web 应用时能够明白并应用它们：

**浏览器 (browser)** 这是你几乎每天都会用到的软件。大部分人并不知道它真正的原理，他们只会把它叫作“网” (the Internet)。它的作用其实是接收你输入到地址栏网址(例如 <http://learnpythonthehardway.org/>)，然后使用该信息向该网址对应的服务器提出请求。

**地址 (address)** 通常这是一个像 <http://test.com/> 一样的 URL (Uniform Resource Locator, 统一资源定位器), 它告诉浏览器该打开哪个网站。前面的 http 指出了你要使用的协议 (protocol), 这里我们用的是“超文本传输协议(Hyper-Text Transport Protocol)”。你还可以试试 <ftp://ibiblio.org/>, 这是一个“FTP 文件传输协议(File Transport Protocol)”的例子。test.com 这部分是“主机名(hostname)”, 也就是一个便于人阅读和记忆的地址, 主机名会被匹配到一串叫作“IP 地址”的数字上面, 这个“IP 地址”就相当于网络中一台计算机的电话号码, 通过这个号码可以访问到这台计算机。最后, URL 后面还可以跟一个路径, 就像 <http://test.com/book/> 中的 /book/ 部分, 它对应的是服务器上的某个文件或者某些资源, 通过访问这样的网址, 你可以向服务器发出请求, 然后获得这些资源。网站地址还有很多别的组成部分, 不过这些是最主要的。

**连接 (connection)** 一旦浏览器知道了你想用的协议(http)、你想访问的服务器 (<http://test.com/>)、以及该服务器需要获取的资源, 它就要创建一个连接。浏览器会让操作系统 (Operating System, OS)打开计算机的一个“端口(port)” (通常是 80 端口), 端口准备好以后, 操作系统会回传给你的程序一个类似文件的东西, 它所做的事情就是通过网络传输和接收数据, 让你的计算机和 <http://test.com/> 这个网站所属的服务器之间实现数据交换。当你使用 <http://localhost:8080/> 访问你自己的站点时, 发生的事情其实是一样的, 只不过这次你告诉了浏览器要访问的是你自己的计算机(localhost), 要使用的端口不是默认的 80, 而是 8080。你还可以直接访问 <http://test.com:80/>, 这和不输入端口效果一样, 因为 HTTP 的默认端口本来就是 80。

**请求 (request)** 你的浏览器通过你提供的地址建立了连接, 现在它需要从远端服务器要到它 (或你) 想要的资源。如果你在 URL 的结尾加了 /book/, 那你想要的就是 /book/ 对应的文件或资源, 大部分的服务器会直接为你调用 /book/index.html 这个文件, 不过我们就假装它不存在好了。浏览器为了获得服务器上的资源, 它需要向服务器发送一个“请求”。这里我就不讲细节了, 你只需要明白, 为了得到服务器上的内容, 它必须先向服务器发送一个请求才行。有意思的是, “资源”不一定非要是文件。例如当浏览器向你的应用程序提出请求的时候, 服务器返回的其实是你的 Python 代码生成的一些东西。

**服务器 (server)** 服务器指的是浏览器另一端连接的计算机, 它知道如何回应浏览器请求的文件和资源。大部分的 web 服务器只要发送文件就可以了, 这也是服务器流量的主要部分。不过你学的是使用 Python 组建一个服务器, 这个服务器知道如何接受请求, 然后返回用 Python 处理过的字符串。当你使用这种处理方式时, 你其实是假装把文件发给了浏览器, 其实你用的都只是代码而已。就像你在《练习 50》中看到的, 要构建一个“响应”其实也不需要多少代码。

**响应 (response)** 这就是你的服务器回复你的请求, 发回至浏览器的 HTML (包括 css、javascript 或 images)。以文件响应为例, 服务器只要从磁盘读取文件, 发送给浏览器就可以了, 不过它还要将这些内容包在一个特别定义的“头部信息(header)”中, 这样浏览器就会知道它

获取的是什么类型的内容。以你的 web 应用程序为例，你发送的其实还是一样的东西，包括 header 也一样，只不过这些数据是你用 Python 代码即时生成的。

这可以算是你能在网上找到的关于浏览器如何访问网站的最快的快速课程了。这个课程应该可以帮你更容易地理解本节的练习，如果你还是不明白，就找找资料多多了解这方面的信息，直到你明白为止。有一个很好的方法，就是你对照着上面的图示，把你在《练习 50》中创建的 web 程序中的内容分成几个部分，让其中的各部分对应到上面的图示中。如果你可以正确地将程序的各部分对应到这个图示，那你就大致明白它的工作原理了。

## 表单（forms）是如何工作的

熟悉“表单”最好的方法就是写一个可以接收表单数据的程序出来，然后看你可以对它做些什么。先将你的 `app.py` 文件修改成下面的样子：

form\_test.py

```
1     from flask import Flask
2     from flask import render_template
3     from flask import request
4
5     app = Flask(__name__)
6
7     @app.route("/hello")
8     def index():
9         name = request.args.get('name', 'Nobody')
10
11         if name:
12             greeting = f"Hello, {name}"
13         else:
14             greeting = "Hello World"
15
16         return render_template("index.html", greeting=greeting)
17
18     if __name__ == "__main__":
19         app.run()
```

重启 flask（按 CTRL + C，然后再次运行）确保它再次加载，然后用浏览器访问 <http://localhost:5000/hello>，应该会显示 “I just wanted to say Hello, Nobody.” 接着，把浏览器中的 URL 改为 <http://localhost:5000/hello?name=Frank>，你会看到 “Hello, Frank.” 最后，把 `name=Frank` 这里改成你的名字，它就会对你说 Hello。



让我们拆解一下脚本中的这些变更：

1. 我们没有直接为 `greeting` 赋值，而是使用了 `request.args` 从浏览器获取数据。这是一个用键值对（key=value pairs）来包含表单值的简单字典。
2. 然后我用新的 `name` 构建 `greeting`，这句你应该已经很熟悉了。
3. 其他的内容和以前是一样的，我们就不再分析了。

URL 中还可以包含多个参数。将本例的两个变量改成这样：<http://localhost:5000/hello?name=Frank&greet=Hola>。然后修改代码，让它像这样获取 `name` 和 `greet`：

```
greet = request.args.get( ' greet ' , ' Hello ' )  
greeting = f"{greet}, {name}"
```

你还应该试着不在 URL 上给出 `greet` 和 `name` 参数，只让浏览器访问 <http://localhost:5000/hello>，然后你会看到，`name` 会默认为“Nobody”，`greet` 会默认为“Hello”。

## 创建 HTML 表单

在 URL 上传递参数也可以，但就是有点丑，而且对普通用户来说有点难用。你真正想要的是一个“发送表单”（POST form），这是一个特殊的 HTML 文件，里面有一个 `<form>` 标签。这个表单会从用户那里收集信息，然后发送给你的网站，就像你之前做的那样。

让我们来快速创建一个，从中你可以看出它的工作原理。你需要创建一个新的 HTML 文件 `templates/hello_form.html`：

hello\_form.html

```
<html>
  <head>
    <title>Sample Web Form</title>
  </head>
  <body>

  <h1>Fill Out This Form</h1>

  <form action="/hello" method="POST">
    A Greeting: <input type="text" name="greet">
    <br/>
    Your Name: <input type="text" name="name">
    <br/>
    <input type="submit">
  </form>

  </body>
</html>
```

然后你需要把 [app.py](#) 改成这样：

[app.py](#)

```

1     from flask import Flask
2     from flask import render_template
3     from flask import request
4
5     app = Flask(__name__)
6
7     @app.route("/hello", methods=['POST', 'GET'])
8     def index():
9         greeting = "Hello World"
10
11         if request.method == "POST":
12             name = request.form['name']
13             greet = request.form['greet']
14             greeting = f"{greet}, {name}"
15             return render_template("index.html", greeting=greeting)
16         else:
17             return render_template("hello_form.html")
18
19
20     if __name__ == "__main__":
21         app.run()

```

改完之后，再次重启 web 应用，像之前一样刷新浏览器。

这次你会看到一个表单，向你获取“A Greeting”和“Your Name.”。当你点击表单上的提交（Submit）按钮时，它会给你跟之前一样的问候。不过这次，浏览器上面的 URL 还是 <http://localhost:5000/hello>，哪怕你已经传递了参数。

让这个发挥作用的是 `hello_form.html` 文件中的这一行：`<form action="/hello" method="POST">`。这告诉浏览器：

1. 从表单中的各个栏位收集用户输入的数据。
2. 使用一种 POST 类型的请求，将这些数据发送给服务器。这是另外一种浏览器请求，它会表单栏位“隐藏”起来。
3. 将这个请求发送至 `/hello` URL，这是由 `action="/hello"` 这部分内容告诉浏览器的。

你可以看到这两个 `<input>` 标签是如何和你新代码中的变量名相匹配的。还要注意一下，在 `class index` 里面，我没有用 GET 方法，而是使用了 POST 方法。这个新程序的工作原理如下：

1. 你的新请求像之前一样去到了 `index()`，不过现在有一个 if 语句来检查

`request.method` 是 "POST" 还是 "GET" 方法。这样浏览器就能告诉 `app.py` 一个请求是表单提交还是 URL 参数。

2. 如果 `request.method` 是 "POST", 程序就会对表单填写和提交的内容进行处理, 并返回合适的问候语。
3. 如果 `request.method` 是其他东西, 那你只要返回 `hello_form.html` 让用户来填写。

作为练习, 在 `templates/index.html` 中添加一个链接, 让它指向 `/hello`, 这样你可以反复填写、提交表单并查看结果。

确认你可以解释清楚这个链接的工作原理, 以及它是如何让你实现在

`templates/index.html` 和 `templates/hello_form.html` 之间循环跳转的, 还有就是要明白你新修改过的 Python 代码中, 运行的是哪一部分代码。

## 创建布局模板(layout template)

在你下一节练习创建游戏的过程中, 你需要创建很多的小 HTML 页面。如果你每次都写一个完整的网页, 你会很快感觉到厌烦的。幸运的是你可以创建一个“布局模板”, 也就是一种提供了通用的头文件 (headers) 和脚注 (footers) 的外壳模板, 你可以用它将你所有的其他网页包裹起来。好程序员会尽可能减少重复动作, 所以要做一个好程序员, 使用布局模板是很重要的。

将 `templates/index.html` 修改为这样:

`index_laid_out.html`

```
{% extends "layout.html" %}

{% block content %}
{% if greeting %}
    I just wanted to say
    <em style="color: green; font-size: 2em;">{{ greeting }}</em>.
{% else %}
    <em>Hello</em>, world!
{% endif %}

{% endblock %}
```

然后将 `templates/hello_form.html` 修改为这样:

## hello\_form\_laid\_out.html

```
{% extends "layout.html" %}

{% block content %}

<h1>Fill Out This Form</h1>

<form action="/hello" method="POST">
    A Greeting: <input type="text" name="greet">
    <br/>
    Your Name: <input type="text" name="name">
    <br/>
    <input type="submit">
</form>

{% endblock %}
```

我们所做的就是将每一个页面顶部和底部反复用到的“boilerplate”（样板）代码去掉。这些被去掉的代码会被放到一个单独的 `templates/layout.html` 文件中，之后，这些反复用到的代码就由 `layout.html` 来提供了。

修改好之后，创建一个 `templates/layout.html` 文件，内容如下：

## layout.html

```
<html>
<head>
    <title>Gothons From Planet Percal #25</title>
</head>
<body>

{% block content %}

{% endblock %}
</body>
</html>
```

这个文件和普通的模板文件类似，不过它会收到其它模板传递的内容，并将它们“包裹”起来。任何写在这里的内容都无需写在别的模板中了。你的其他 HTML 模板会被插入到

`{% block content %}` 中。Flask 知道要把 `layout.html` 文件用作布局，因为你在模板的顶部放了 `{% extends "layout.html" %}`。

# 为表单撰写自动测试代码

使用浏览器测试 web 程序是很容易的，只要点刷新按钮就可以了。不过毕竟我们是程序员嘛，如果我们可以写一些代码来测试我们的程序，为什么还要重复手动测试呢？接下来你要做的，就是为你的 web 程序写一个小测试。这会用到你在《练习 47》学过的一些东西，如果你不记得的话，可以回去复习一下。

创建一个新文件，并命名为 tests/app\_tests.py，其内容如下：

app\_tests.py

```
1     from nose.tools import *
2     from app import app
3
4     app.config['TESTING'] = True
5     web = app.test_client()
6
7     def test_index():
8         rv = web.get('/', follow_redirects=True)
9         assert_equal(rv.status_code, 404)
10
11         rv = web.get('/hello', follow_redirects=True)
12         assert_equal(rv.status_code, 200)
13         assert_in(b"Fill Out This Form", rv.data)
14
15         data = {'name': 'Zed', 'greet': 'Hola'}
16         rv = web.post('/hello', follow_redirects=True, data=data)
17         assert_in(b"Zed", rv.data)
18         assert_in(b"Hola", rv.data)
```

最后，用 `nosetests` 运行这个测试程序，来测试你的 web 应用：

```
$ nosetests
.
-----
Ran 1 test in 0.059s OK
```

我在这儿其实是把整个应用都从 `app.py` 模块中引入进来了，然后手动运行它。flask 框架有一个非常简单用来处理请求的 API，它看起来像这样：

```
data = {'name': 'Zed', 'greet': 'Hola'}
rv = web.post('/hello', follow_redirects=True, data=data)
```

这意味着你可以用 `post()` 方法发送一个 POST 请求，然后把表单数据作为字典传给它。其他都和测试 `web.get()` 请求一模一样。

在 `tests/app_tests.py` 自动测试脚本中，我首先确认 `/` 返回了一个“404 Not Found”响应，因为这个 URL 其实是不存在的。然后我检查了 `/hello` 在 GET 和 POST 两种请求的情况下都能正常工作。就算你没有弄明白测试的原理，这些测试代码应该是很好读懂的。

花些时间研究一下这个最新版的 web 程序，重点研究一下自动测试的工作原理。确保你理解了将 `app.py` 做为一个模块导入，然后进行自动化测试的流程。这是一个很重要的技巧，它会引导你学到更多东西。

## 附加练习

1. 阅读和 HTML 相关的更多资料，然后为你的表单设计一个更好的输出格式。你可以先在纸上设计出来，然后用 HTML 去实现它。
2. 这是一道难题，试着研究一下如何进行文件上传，通过网页上传一张图像，然后将其保存到磁盘中。
3. 更难的难题，找到 HTTP RFC 文件（讲述 HTTP 工作原理的技术文件），然后努力阅读一下。这是一篇很无趣的文档，不过偶尔你也会用到里边的一些知识。
4. 又是一道难题，找人帮你设置一个 web 服务器，例如 Apache、Nginx 或者 `thttpd`。试着让服务器 `serve` 一下你创建的 `.html` 和 `.css` 文件。如果失败了也没关系，web 服务器本来就都有点烂。
5. 完成上面的任务后休息一下，然后试着多创建一些 web 程序出来。

## 拆解

这里很适合讲一下如何拆解 web 应用。你应该这样做：

1. 打开 `FLASK_DEBUG` 会造成多大的损害？注意做这个的时候别把自己电脑搞垮了。
2. 假设你没有为表单设置默认参数，哪里会出错？
3. 你先检查 `POST` 然后是“其他东西”。你可以用 `curl` 命令行工具生成不同的请求类型。看看会发生什么？

# 练习 52. 创建你的 web 游戏

这本书马上就要结束了。这节练习对你来说是个真正的挑战。当你完成以后，你就可以算是一个能力不错的 Python 初学者了。为了进一步学习，你还需要多读一些书，多写一些程序，不过你已经具备进一步学习的能力了。接下来的学习就只是时间、动力、以及资源的问题了。

在本节练习中，我们不会去创建一个完整的游戏，而是要为《练习 47》中的游戏创建一个“引擎 (engine)”，让这个游戏能够在浏览器中运行起来。这会涉及到将《习题 43》中的游戏“重构 (refactor)”，将《习题 47》中的架构混合进来，添加自动测试代码，最后创建一个可以运行游戏的 web 引擎。

这个练习会非常庞大。我预测你要花一周到一个月时间才能完成它。你最好一点一点来，每天晚上完成一点，在进行下一步之前确保上一步已经正确完成。

## 重构《练习 43》的游戏

你已经在两个练习中修改了 gothonweb 项目，这节习题中你会再修改一次。这种修改的技术叫做“重构 (refactoring)”，或者用我喜欢的讲法来说，叫“修修补补 (fixing stuff)”。重构是一个编程术语，它指的是清理旧代码或者为旧代码添加新功能的过程。你其实已经做过这样的事情了，只不过不知道这个术语而已。这是写软件过程的第二个自然属性。

你在本节中要做的，是将《习题 47》中的可以测试的房间地图，以及《习题 43》中的游戏这两样东西归并到一起，创建一个新的游戏架构。游戏的内容不会发生变化，只不过我们会通过“重构”让它有一个更好的架构而已。

第一步是将 `ex47/game.py` 的内容复制到 `gothonweb/planisphere.py` 中，然后将 `tests/ex47_tests.py` 的内容复制到 `tests/planisphere_tests.py` 中，然后再次运行 `nosetests`，确保他们还能正常工作。“planisphere”这个词是地图的同义词，用这个名字是为了避免 Python 内置的 `map` 函数。同义词典 (Thesaurus) 是个好东西，要善于利用它。

### 警告！

从现在开始，我不会再向你展示我运行测试的输出结果了。我假设你会自己去做测试，所以测试是个前提

当你把《练习 47》的代码复制好之后，你就该开始重构它了，让它包含《习题 43》中的地图。我一开始会把基本架构为你准备好，然后你需要去完成 `planisphere.py` 和



`planisphere_tests.py` 这两个文件里边的内容。

首先要做的是使用 Room 类来构建基本的地图架构：

[planisphere.py](#)

```

1      class Room(object):
2
3          def __init__(self, name, description):
4              self.name = name
5              self.description = description
6              self.paths = {}
7
8          def go(self, direction):
9              return self.paths.get(direction, None)
10
11         def add_paths(self, paths):
12             self.paths.update(paths)
13
14
15         central_corridor = Room("Central Corridor",
16             """
17             The Gothons of Planet Percal #25 have invaded your ship and destroyed
18             your entire crew. You are the last surviving member and your last
19             mission is to get the neutron destruct bomb from the Weapons Armory, put
20             it in the bridge, and blow the ship up after getting into an escape pod.
21
22             You're running down the central corridor to the Weapons Armory when a
23             Gothon jumps out, red scaly skin, dark grimy teeth, and evil clown
24             costume flowing around his hate filled body. He's blocking the door to
25             the Armory and about to pull a weapon to blast you.
26             """)
27
28
29         laser_weapon_armory = Room("Laser Weapon Armory",
30             """
31             Lucky for you they made you learn Gothon insults in the academy. You
32             tell the one Gothon joke you know: Lbhe zbgure vf fb sng, jura fur fvgf
33             nebhaq gur ubhfr, fur fvgf nebhaq gur ubhfr. The Gothon stops, tries
34             not to laugh, then busts out laughing and can't move. While he's
35             laughing you run up and shoot him square in the head putting him down,
36             then jump through the Weapon Armory door.
37
38             You do a dive roll into the Weapon Armory, crouch and scan the room for
39             more Gothons that might be hiding. It's dead quiet, too quiet. You
40             stand up and run to the far side of the room and find the neutron bomb
41             in its container. There's a keypad lock on the box and you need the
42             code to get the bomb out. If you get the code wrong 10 times then the
43             lock closes forever and you can't get the bomb. The code is 3 digits.
44             """)
45
46

```

```

47     the_bridge = Room("The Bridge",
48         """
49         The container clicks open and the seal breaks, letting gas out. You
50         grab the neutron bomb and run as fast as you can to the bridge where you
51         must place it in the right spot.
52
53         You burst onto the Bridge with the netron destruct bomb under your arm
54         and surprise 5 Gothons who are trying to take control of the ship. Each
55         of them has an even uglier clown costume than the last. They haven't
56         pulled their weapons out yet, as they see the active bomb under your arm
57         and don't want to set it off.
58         """)
59
60
61     escape_pod = Room("Escape Pod",
62         """
63         You point your blaster at the bomb under your arm and the Gothons put
64         their hands up and start to sweat. You inch backward to the door, open
65         it, and then carefully place the bomb on the floor, pointing your
66         blaster at it. You then jump back through the door, punch the close
67         button and blast the lock so the Gothons can't get out. Now that the
68         bomb is placed you run to the escape pod to get off this tin can.
69
70         You rush through the ship desperately trying to make it to the escape
71         pod before the whole ship explodes. It seems like hardly any Gothons
72         are on the ship, so your run is clear of interference. You get to the
73         chamber with the escape pods, and now need to pick one to take. Some of
74         them could be damaged but you don't have time to look. There's 5 pods,
75         which one do you take?
76         """)
77
78
79     the_end_winner = Room("The End",
80         """
81         You jump into pod 2 and hit the eject button. The pod easily slides out
82         into space heading to the planet below. As it flies to the planet, you
83         look back and see your ship implode then explode like a bright star,
84         taking out the Gothon ship at the same time. You won!
85         """)
86
87
88     the_end_loser = Room("The End",
89         """
90         You jump into a random pod and hit the eject button. The pod escapes
91         out into the void of space, then implodes as the hull ruptures, crushing
92         your body into jam jelly.

```

```

93     """
94     )
95
96     escape_pod.add_paths({
97         '2': the_end_winner,
98         '*': the_end_loser
99     })
100
101     generic_death = Room("death", "You died.")
102
103     the_bridge.add_paths({
104         'throw the bomb': generic_death,
105         'slowly place the bomb': escape_pod
106     })
107
108     laser_weapon_armory.add_paths({
109         '0132': the_bridge,
110         '*': generic_death
111     })
112
113     central_corridor.add_paths({
114         'shoot!': generic_death,
115         'dodge!': generic_death,
116         'tell a joke': laser_weapon_armory
117     })
118
119     START = 'central_corridor'
120
121     def load_room(name):
122         """
123         There is a potential security problem here.
124         Who gets to set name? Can that expose a variable?
125         """
126         return globals().get(name)
127
128     def name_room(room):
129         """
130         Same possible security problem. Can you trust room?
131         What's a better solution than this globals lookup?
132         """
133         for key, value in globals().items():
134             if value == room:
135                 return key

```

你会发现我们的 Room 类和地图有一些问题：

1. 我们必须把放在 if-else 语句中的文本在进入一个房间之前打印出来，作为每个房间的一部分。这就意味着你不能把 planisphere 打乱，这很好。你要在这个练习中慢慢修复它。
2. 原版游戏中我们使用了专门的代码来生成一些内容，例如炸弹的激活键码，舰舱的选择等，这次我们做游戏时就先使用默认值好了，不过后面的附加练习里，我会要求你把这些功能再加入到游戏中。
3. 我为游戏中的所有失败结尾写了一个 `generic_death`，你需要去补全这个函数。你需要把原版游戏中所有的失败结尾都加进去，并确保代码能正确运行。
4. 我添加了一种新的转换模式，以 "\*" 为标记，用来在游戏引擎中实现“catch-all”动作。

等你把上面的代码基本写好以后，接下来就是引导你继续写下去的自动测试的内容

`tests/planisphere_test.py` :

`planisphere_tests.py`

```

1     from nose.tools import *
2     from gothonweb.planisphere import *
3
4     def test_room():
5         gold = Room("GoldRoom",
6             """This room has gold in it you can grab. There's a
7             door to the north.""")
8         assert_equal(gold.name, "GoldRoom")
9         assert_equal(gold.paths, {})
10
11     def test_room_paths():
12         center = Room("Center", "Test room in the center.")
13         north = Room("North", "Test room in the north.")
14         south = Room("South", "Test room in the south.")
15
16         center.add_paths({'north': north, 'south': south})
17         assert_equal(center.go('north'), north)
18         assert_equal(center.go('south'), south)
19
20     def test_map():
21         start = Room("Start", "You can go west and down a hole.")
22         west = Room("Trees", "There are trees here, you can go east.")
23         down = Room("Dungeon", "It's dark down here, you can go up.")
24
25         start.add_paths({'west': west, 'down': down})
26         west.add_paths({'east': start})
27         down.add_paths({'up': start})
28
29         assert_equal(start.go('west'), west)
30         assert_equal(start.go('west').go('east'), start)
31         assert_equal(start.go('down').go('up'), start)
32
33     def test_gothon_game_map():
34         start_room = load_room(START)
35         assert_equal(start_room.go('shoot!'), generic_death)
36         assert_equal(start_room.go('dodge!'), generic_death)
37
38         room = start_room.go('tell a joke')
39         assert_equal(room, laser_weapon_armory)

```

你在这部分练习中的任务是完成这个地图，并且让自动测试可以完整地检查过整个地图。这包括将所有的 `generic_death` 对象修正为游戏中实际的失败结尾。让你的代码成功运行起来，并让你的测试越全面越好。后面我们会对地图做一些修改，到时候这些测试将保证修改后的代码还可以正常工作。

# 创建一个引擎

你应该让你的游戏地图正常运行，并对它进行良好的单元测试。我现在想让你做一个简单的小游戏引擎，它将运行房间、收集来自玩家的输入，并跟踪玩家在游戏的位置。我们将使用你刚刚学会的会话来创建一个简单的游戏引擎，这个引擎会做这些事情：

1. 为新用户开启一个新游戏。
2. 为用户展示房间。
3. 从用户获取输入。
4. 通过游戏运行用户的输入。
5. 呈现结果，并继续运行，直至用户挂掉。

要做到这些，你需要使用你一直在写的可靠的 [app.py](#)，来创建一个运行良好的、基于会话的游戏引擎。问题是，我需要做一个非常简单的基本 HTML 文件，它将由你来完成它。这是基础引擎：

[app.py](#)

```

1     from flask import Flask, session, redirect, url_for, escape, request
2     from flask import render_template
3     from gothonweb import planisphere
4
5     app = Flask(__name__)
6
7     @app.route("/")
8     def index():
9         # this is used to "setup" the session with starting value
10        session['room_name'] = planisphere.START
11        return redirect(url_for("game"))
12
13    @app.route("/game", methods=['GET', 'POST'])
14    def game():
15        room_name = session.get('room_name')
16
17        if request.method == "GET":
18            if room_name:
19                room = planisphere.load_room(room_name)
20                return render_template("show_room.html", room=room)
21            else:
22                # why is there here? do you need it?
23                return render_template("you_died.html")
24        else:
25            action = request.form.get('action')
26
27            if room_name and action:
28                room = planisphere.load_room(room_name)
29                next_room = room.go(action)
30
31                if not next_room:
32                    session['room_name'] = planisphere.name_room
33                else:
34                    session['room_name'] = planisphere.name_room
35
36            return redirect(url_for("game"))
37
38
39    # YOU SHOULD CHANGE THIS IF YOU PUT ON THE INTERNET
40    app.secret_key = 'A0Zr98j/3yX R~XHH!jmN]LWX/,?RT'
41
42    if __name__ == "__main__":
43        app.run()

```

这个脚本中有更多的新东西，但神奇的是，这个小文件是一个完全基于 web 的游戏引擎。在运行 `app.py` 之前，需要更改 `PYTHONPATH` 环境变量。不知道那是什么？我知道这有点枯燥，



但你必须学习这是什么来运行基本的 Python 程序，没办法，用 Python 的人就喜欢这样。

在你的终端输入：

```
export PYTHONPATH=$PYTHONPATH:.
```

在 Windows 的 PowerShell 中输入：

```
$env:PYTHONPATH = "$env:PYTHONPATH;."
```

你只要针对每一个命令行会话界面输入一次就可以了，不过如果你运行 Python 代码时看到了 `import error`，或者你输入错误，那就需要再去执行一下上面的命令。

接下来你需要删掉 `templates/hello_form.html` 和 `templates/index.html`，并创建两个前面代码中提到的模板。这是一个非常简单的 `templates/show_room.html`：

`show_room.html`

```
{% extends "layout.html" %}

{% block content %}

<h1> {{ room.name }}          </h1>

<pre>
{{ room.description }}
</pre>

{% if room.name in ["death", "The End"] %}
    <p><a href="/">Play Again?</a></p>
{% else %}
    <p>
        <form action="/game" method="POST">
            - <input type="text" name="action"> <input type="SUBMIT">
        </form>
    </p>
{% endif %}

{% endblock %}
```

这是在游戏中显示房间的模板。接下来你需要一个模板来告诉用户他们已经死了，以防他们意外地去到地图的结尾，也就是 `templates/you_die.html`：

you\_died.html

```
<h1>You Died!</h1>

<p>Looks like you bit the dust.</p>
<p><a href="/">Play Again</a></p>
```

这些都弄好了之后，你可以这样做：

1. 让 `tests/app_tests.py` 再次运行来测试这个游戏。因为有会话，所以你只需要在游戏里点几下就行。不过，你应该能做一些基本操作。
2. 运行 `python3.6 app.py` 脚本来玩一下这个游戏。

你需要和往常一样刷新和修正你的游戏，慢慢修改游戏的 HTML 文件和引擎，直到你实现游戏需要的所有功能为止。

## 你的期末考试

你有没有觉着我一下子给了你超多的信息呢？那就对了，我想要你在学习技能的同时可以有一些可以用来鼓捣的东西。为了完成这节习题，我会给你最后一套需要你自己完成的练习。你应该注意到，到目前为止你写的游戏并不是很好，这只是你的第一版代码而已。你现在的任务是让游戏更加完善，实现下面的这些功能：

1. 修正代码中所有我提到和没提到的 bug，如果你发现了新的 bug，可以告诉我。
2. 改进所有的自动测试，让你可以测试更多的内容，直到你可以不用浏览器就能测到所有的内容为止。
3. 让 HTML 页面看上去更美观一些。
4. 研究一下网页登录系统，为这个程序创建一个登录界面，这样人们就可以登录这个游戏，并且可以保存游戏高分。
5. 完成游戏地图，尽可能地把游戏做大，功能做全。
6. 给用户一个“帮助系统”，让他们可以查询每个房间里可以执行哪些命令。
7. 为你的游戏添加任何你能想到的新功能。
8. 创建多个地图，让用户可以选择他们想要玩的一张来进行游戏。你的 `app.py` 应该可以运行提供给它任意的地图，这样你的引擎就可以支持多个不同的游戏。
9. 最后，使用你在练习 48 和 49 中学到的东西来创建一个更好的输入处理器。你手头已经有了大部分必要的代码，你只需要改进语法，让它和你的输入表单以及游戏引擎挂钩即可。

祝你好运！

## 常见问题

我在游戏中用了 **session**，但不能用 **nosetests** 测试。阅读 Flask 测试文档（Flask Testing Documentation）中的“其他测试技巧”（Other Testing Tricks），了解关于在游戏中创建“假会话”（fake sessions）的信息。

我收到了一个 `ImportError`。可能是以下情况中的一种或几种：错误的目录，错误的 Python 版本，没有设置 PYTHON-PATH，没有 `init.py` 文件，以及（或者）import 中存在拼写错误。

## 练习 53. 接下来的步骤

你还不是一个程序员。我会把这本书看作是你的“编程黑带”。你已经知道了足够多的东西，可以开始写另一本关于编程的书了，并且可以写得很好。这本书应该已经给了你阅读大多数 Python 书籍并实际学习一些东西的心智工具和态度。它甚至可能让这件事变得更容易。

我建议你看看这些项目，并尝试用它们来创建一些东西：

- **Learn Ruby The Hard Way** 当你学习更多的编程语言时，你会学到更多关于编程的知识，所以试着学习 Ruby 吧。
- **Django 教程** 尝试使用 Django web 框架构建一个 web 应用程序。
- **SciPy** 如果你对科学、数学和工程感兴趣，那就去学 scipy 吧。
- **PyGame** 看看你能否制作一个带有图像和声音的游戏。
- **Pandas** 用于数据操作和分析的 Pandas。
- **Natural Language Tool Kit** 用于分析文本、编写垃圾邮件过滤器以及聊天机器人等内容的自然语言工具包。还有用于机器学习和可视化的 TensorFlow。
- **Requests** 学习 web 和 HTTP 客户端。
- **ScraPy** 尝试爬取一些网站并获取信息。
- **Kivy** 在台式机和移动平台上做用户界面。
- **Learn C The Hard Way** 在你熟悉 Python 之后，尝试用我的另一本书学习 C 语言和算法。慢慢来，C 语言与众不同，但它很值得学习。

从以上这些参考资料中选一个，阅读它们提供的任何教程和文档。在查看带有代码的文档时，输入所有代码并使其工作。我就是这么做的。每个程序员都是这么做的。光阅读编程文档是学不会

的，你必须去做。在你看完教程和他们的其他文档之后，做点什么。任何东西都可以，即使是别人已经写过的东西。

你要明白你写的东西可能会很烂。我刚开始使用一种编程语言的时候也都写得很糟糕，这没什么。没有人在初学者的时候就能写出完美的东西，任何说他们一开始就完美的人都是大骗子。

## 如何学习任何编程语言

我将教你如何学习大多数你将来可能想学的编程语言。本书的组织是基于我和许多其他程序员学习新语言的方式。我通常遵循的流程是：

1. 找一本关于这门语言的书或一些介绍性文档。
2. 阅读这本书，输入所有的代码，让它运行起来。
3. 一边写代码一边看书，同时做笔记。
4. 使用该语言去实现一些你用其他熟悉的语言写的小程序。
5. 阅读其他人的代码，并尝试复制他们的模式。

在这本书中，我让你非常缓慢地、小块地完成这个过程。其他书的组织方式可能会不同，所以你必须根据我告诉你的方式来推断它们的内容是如何组织的。

要做到这一点，最好的方法是轻松地阅读这本书，并列出所有主要代码部分的列表。把这个列表变成一组基于章节的练习，然后按顺序一次做一个。

上述过程也适用于新技术，假设有相关的书籍供你阅读。对于任何没有书的内容，你可以使用在线文档或源代码，然后按照上述过程进行学习。

你学习的每一种新语言都会让你成为一个更好的程序员，而且随着你学习得越来越多，学起来会更容易。当你学到第三或第四种语言时，你应该能在一周内学会类似的语言，而学习陌生的语言则需要更长的时间。现在你已经了解了 Python，那么你应该可以很快地学会 Ruby 和 JavaScript。因为很多语言都具有相似的概念，一旦你学习了一种语言中的概念，它们也适用于其他语言。

学习一门新语言要记住的最后一件事是：不要做一个愚蠢的游客。愚蠢的游客是指一个人去了另一个国家，然后抱怨那里的食物和国内的不一样。“为什么我在这个该死的国家吃不到好吃的汉堡！？”当你学习一门新的语言时，要假设它做的事情并不愚蠢，它只是不同而已。接受它，这样你才能学习它。”

在你学习了一门语言之后，不要成为这种语言做事方式的奴隶。有时候，人们使用一种语言来做

一些非常愚蠢的事情，不为别的，只是因为“我们一直都是这样做的”。如果你更喜欢自己的风格，而且你知道其他人是怎么做的，那么，如果你的风格使事情得到了改善，你可以随意打破他们的规则。

我真的很喜欢学习新的编程语言。我常常自诩为一个“程序员人类学家”，并且认为这些编程语言代表了那些使用它们的程序员群体的一些洞见。我正在学习一门通过电脑互相交流的语言，真的非常有趣。再说一次，只有当你真正想学的时候，你再去学编程语言。

享受这个过程吧！其乐无穷。

## 练习 54. 来自老程序员的建议

你已经读完了这本书，并决定继续编程。也许它会成为你的职业，也许它会成为你的爱好。你需要一些建议来确保你继续走在正确的道路上，并从你新选择的活动中获得最大的乐趣。

我已经编程很长时间了，时间长到甚至让我觉得有些无聊。在我写这本书的时候，我了解 20 多种编程语言，并且可以在一天到一周内学会新的语言，时间取决于它们的怪异程度。虽然最后都会变得无聊，使我没办法再保持兴趣。但这并不意味着我认为编程是无聊的，或者你会认为它是无聊的，只是在我的旅程中我发现它在这一点上是无趣的。

在这次学习之旅后，我发现重要的不是语言本身，而是你如何使用它们。事实上，我一直都知道这一点，但我会因为语言而分心，然后周期性地忘记它。现在我永远都不会忘记，你也不应该忘记。

你学习和使用哪种编程语言并不重要。不要陷入编程语言的宗教信仰中，因为那会让你看不到掌握它们的真正目的，那就是用它们来做有趣的事情。

编程作为一种智力活动是唯一允许你创造互动艺术的艺术形式。你可以创建其他人可以玩的项目，你可以和他们间接交谈。没有其他的艺术形式能这样互动。电影朝着一个方向流向观众。画不会动。而代码是双向的。

作为一种职业，编程只是比较有趣。它可能是一份不错的工作，但是经营一家快餐店可以让你赚同样多的钱，同时也更快乐。你最好在其他职业中使用代码作为你的秘密武器。

在科技公司里，能写代码的人多如牛毛，他们得不到尊重。而能够在生物学、医学、政府、社会学、物理学、历史学和数学领域编写代码的人才才会得到尊重，他们能够做出令人惊叹的事情来推动这些学科的发展。

当然，所有这些建议都是毫无意义的。如果你喜欢用这本书学习写软件，你应该试着用它来改善你的生活。走出去，探索这个奇怪的、美妙的、新奇的智力追求，在过去 50 年里，很少有人有机会去探索，趁你还有机会，尽情享受这个过程吧。

最后，我要说的是学习创建软件会改变你，使你与众不同。不是更好或更坏，只是不同。你可能会发现，因为你会开发软件，因为你可以剖析人们讨厌和你争论的逻辑，人们就会对你很苛刻，可能会用“书呆子”这样的词。你甚至会发现，仅仅知道电脑是如何工作的，就会让他们觉得你很讨厌、很奇怪。

对于这一点，我只有一个建议：让他们去死吧。这个世界需要更多奇怪的人，他们知道事情是如何运转的，他们喜欢搞清楚一切。当他们这样对待你时，记住这是你的旅程，不是他们的。与众不同不是罪，如果有人说它是，那他们只是嫉妒你获得了他们做梦都想不到的技能。

你会编程，但他们不会。这真是太酷了。

## 附录 A：命令行速成教程

该附录是一个命令行的超级速成教程，它主要是为了让你在一两天内快速上手命令行，而不是教你一些高级的 Shell 用法。

### 介绍：别说话，开始用 Shell

该附录是一个命令行的速成教程，命令行可以让你的计算机执行任务。作为一个速成教程，它不会像我其他的书一样教得很详细。它只是为了让你能够像一个真正的程序员一样使用你的电脑。当你学完这个附录，你将能够掌握每个使用 Shell 的人每天进行的最基本的一些操作。你还会明白目录和其他一些概念的基础知识。

我唯一要给你的建议就是：什么也别说了，开始用 Shell 吧。如果你对命令行感到恐惧，唯一克服的方法就是去学习和攻克它。编程语言就是用更高级的方式去控制你的计算机，而命令行就是编程语言的雏形。一旦你掌握了命令行，你就能够更轻松地学习编程语言。

#### 55.1.1 如何使用附录

使用附录最好的方式如下：

- 准备一个本子和一根笔。

- 从附录最开始做好每一个练习。
- 当你遇到任何不懂的地方时，把它记在你的本子上。留一些空白以便之后补充答案。
- 当你完成一个练习，回头来看你的本子，并重新审视你的问题。试着通过上网搜索或者请教别人来回答这些问题。（**如果实在搞不懂欢迎去微信公众号“学习癌”相应章节下面查看大家讨论或者在后台留言，你会很快得到答案。**）
- 每个练习都按这样的步骤来，写下你的问题，然后回头去找答案，当你做完这些，你会比你想象的更了解命令行的使用。

## 55.1.2 你需要记东西

在命令行的学习中我会要求你记东西，这是掌握知识最快的方法。我知道对某些人来说记东西非常痛苦，但是你得克服它，让自己记住。记忆是学习知识的不二法门，你必须克服这种恐惧。

以下是记忆方法：

- 告诉自己你可以做到，别试着找捷径，坐下来认真去记。
- 把你要记的东西写在一些索引卡片上，然后一半一半分成两堆。
- 每天花 15-30 分钟时间去记忆这些卡片，试着回忆每一张上面的内容。把没记清楚的放一块儿重点记忆，直到烂熟于心。然后再全部过一遍，检查自己有没有全部记住。
- 在你晚上睡觉之前，把你之前没记住的卡片拿出来复习 5 分钟再睡。

还有其他一些技巧，比如你可以把你学的东西写在一张纸上，然后贴到你经常能看到的墙上，当你看到墙的时候就顺便复习一遍。

如果你每天都坚持这样做，你应该能记住我让你记的大多数内容。一旦你这样做了，基本上其他任何东西都会变得更简单和更直觉（intuitive），这也是记忆的目的。它不仅是为了教会你抽象的概念，更是为了让你不用想就知道，这也是你去学习更难的知识所必需的基础。

## 附录练习 1 环境配置

在该附录中，你将需要做以下三件事情：

- 用你的 Shell（命令行、Terminal、Powershell）做一些操作。
- 学习你做的这些操作。
- 自己去做更多的操作。

在最开始的这个练习中，你需要打开你的 Terminal，并让它正常运行，以便去做接下来的练习。

## 55.2.1 跟我做

### macOS

用 macOS 的童鞋可以这样做：

- 按住 Command 键+空格。
- 右上角会出现搜索框。
- 输入 terminal。
- 点击 Terminal 以打开，它看起来像个黑盒子。
- 把 Terminal 放在 Dock（右键点击下面的 Terminal 图标，在“选项”中勾选“在Dock中保留”）

现在你已经打开了 Terminal，并把它放在了 Dock 以便快速访问。

### Linux

我假设如果你使用 Linux 你已经知道如何找到并打开 Terminal 了。

### Windows

在 Windows 系统下我们要用 Powershell。有些人习惯用一个叫 `cmd.exe` 的程序来工作，但是它没有 Powershell 好用。如果你用的是 Windows 7 以及以上的版本，可以这样做：

- 点击开始
- 在“搜索程序和文件”中输入 Powershell（Windows 10 可以直接在左下角搜索框输入）
- 点击回车

如果你用的不是 Windows 7 或者以上版本，你真的该考虑升级了。如果你不想或者没办法升级，可以去微软官网下载适合你系统版本的 Powershell。因为我没用过 XP，所以不知道整个过程是不是一样，但愿如此吧。

## 55.2.2 你学到的

你学到了如何打开 Terminal，以进行附录中后面的练习。



# 55.2.3 附加练习

这一节有一个庞大的附加练习，其他节都没有这么多。记忆这些内容能让你的大脑准备好去学习后面的东西，相信我，这会让你后面的学习更加轻松和顺畅。

## Linux/macOS

把以下命令符列表写在卡片上，名字在左，含义在右，随着附录的学习每天复习。

命令符	含义
pwd	打印工作目录
hostname	计算机网络运营商名称
mkdir	创建目录
cd	切换目录
ls	列示目录
rmdir	移除目录
pushd	前往新目录地址
popd	返回原目录地址
cp	复制文件或目录
mv	移动文件或目录
less	在文件中翻页
cat	打印整个文件
xargs	执行参数值
find	查找文件
grep	在文件中查找内容
man	打开帮助手册
apropos	查找合适的帮助内容

命令符	含义
<b>env</b>	查看环境
<b>echo</b>	打印参数值
<b>export</b>	输出/设置新环境变量
<b>exit</b>	退出 shell
<b>sudo</b>	<b>危险！</b> 获得 root 权限 <b>慎用！</b>

## Windows

如果你用的是 Windows，以下是你的命令符列表：

命令符	含义
<b>pwd</b>	打印工作目录
<b>hostname</b>	计算机网络运营商名称
<b>mkdir</b>	创建目录
<b>cd</b>	切换目录
<b>ls</b>	列示目录
<b>rmdir</b>	移除目录
<b>pushd</b>	前往新目录地址
<b>popd</b>	返回原目录地址
<b>cp</b>	复制文件或目录
<b>robocopy</b>	超强复制
<b>mv</b>	移动文件或目录
<b>more</b>	在文件中翻页
<b>type</b>	打印整个文件

命令符	含义
<b>forfiles</b>	在多个文件执行命令
<b>dir -r</b>	查找文件
<b>select-string</b>	在文件中查找内容
<b>help</b>	打开帮助手册
<b>helpctr</b>	查找合适的帮助内容
<b>env</b>	查看环境
<b>echo</b>	打印参数值
<b>set</b>	输出/设置新环境变量
<b>exit</b>	退出 shell
<b>runas</b>	<b>危险！</b> 获得 root 权限 <b>慎用！</b>

**练习练习练习，记忆记忆记忆**，直到你能够对这些命令符脱口而出，而且记忆必须是双向的，你得能够看着命令符说出它的作用，也得知道要执行某个操作需要哪个命令符。通过这种方式，你可以逐步构建起自己的计算机语言词汇，但是也不要花费太多时间，如果你感到厌倦了就往下进行，在学习中强化记忆。

## 附录练习2 路径，文件夹，目录 (pwd)

在这个练习中你将学习如何用 `pwd` 命令打印当前正在工作的目录。

### 55.3.1 跟我做

我会教你如何阅读我展示给你的会话（session）。你不用输入我列出来的所有内容，只用输入其中一部分：

- 不用输入 `$`（Unix 系统）或者 `>`（Windows 系统）。那只是我用来说明我的会话中得到的输出结果。
- 你输入 `$` 或者 `>` 后面的内容，然后回车。比如如果我写的是 `$ pwd`，你就只用输入 `pwd` 然后回车就行。

- 然后你就可以在 `$` 或者 `>` 之后我得到的输出结果。

让我们先做个简单的练习，你就明白了：

## Linux/macOS

### 练习 2 会话

```
$ pwd
/Users/zedshaw
$
```

## Windows

### 练习 2 Windows 会话

```
PS C:\Users\zed> pwd Path
----
C:\Users\zed

PS C:\Users\zed>
```

### 警告！

在附录部分我需要节省空间以至于你能专注在命令行的重要细节上。为此，我将去掉 `>` 之前的内容，这也意味着你的呈现结果可能会跟我的不太一样，不过没关系，记住 `>` 之后的内容是你输入的，在 Unix 系统下是 `$`。

## 55.3.2 你学到的

你的提示符可能跟我的不太一样，你的 `$` 前面可能是你用户名和电脑名。Windows 系统下可能也会不一样。不过最重要的是你看到的也是如下的模式：

- 有一个提示符。
- 你在提示符后面输入命令，在本练习中是 `pwd`。
- 它打印了一些东西。
- 重复。

你已经学习了 `pwd` 的作用，即“打印工作目录”。什么是目录？目录就是文件夹，它们是同一个

东西。当你打开你电脑的文件查看器去寻找文件的时候，你就是在文件夹中穿梭，这些文件夹就是所说的“目录”。

### 55.3.3 附加练习

- 输入 20 遍 `pwd`，边打边说“打印工作目录”。
- 写下这个命令输出的文件路径，用你的文件查看器找到这个文件。
- 我是认真的，输 20 遍，大声说出它的意思。

## 附录练习3 如果你迷路了

在学习上个练习的时候你可能会有点迷路，不知道你自己在哪儿，或者不知道文件在哪儿，也不知道怎么继续。要解决这个问题，我会教你停止迷路的命令。

不管你什么时候迷的路，很大可能是因为你输入命令的时候不知道你停在哪儿。你要做的就是输入 `pwd` 以查看你当前所在的目录，这将会告诉你你现在在哪儿。

接下来你需要回到你想回去的地方——你的 home，你需要输入 `cd ~`，然后你就能回到你的 home。也就是说，任何时候只要你迷路了，你都可以先输入 `pwd`，再输入 `cd ~`，前者让你知道你现在在哪，后者让你回到 home 以便重新开始。

### 55.4.1 跟我做

现在用 `pwd` 弄明白你在哪儿，然后用 `cd ~` 回到 home，这样可以确保你总是在正确的地方。

### 55.4.2 你学到的

如果你迷路了，如何返回 home。

## 附录练习4 创建目录（mkdir）

在这个练习中，你将学习如何用 `mkdir` 命令创建新目录。

### 55.5.1 跟我做

记住！在进行这个练习之前，你需要先用 `pwd` 和 `cd ~` 回到 home！在做附录之后的每个练习

前，都要先回到 home !

## Linux/macOS

### 练习 4 会话

```
$ pwd
$ cd ~

$ mkdir temp
$ mkdir temp/stuff
$ mkdir temp/stuff/things
$ mkdir -p temp/stuff/things/orange/apple/pear/grape
$
```

## Windows

### 练习 4 Windows 会话

```
> pwd
> cd ~
> mkdir temp
```

Directory: C:\Users\zed

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:02 AM		temp

```
> mkdir temp/stuff
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:02 AM		stuff

```
> mkdir temp/stuff/things
```

Directory: C:\Users\zed\temp\stuff

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:03 AM		things

```
> mkdir temp/stuff/things/orange/apple/pear/grape
```

Directory: C:\Users\zed\temp\stuff\things\orange\apple\pear

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:03 AM		grape

>

`pwd` 和 `cd ~` 命令我只列这一次，但是记住，做每个练习之前你都要做这个操作。

## 55.5.2 你学到的

现在我们开始输入多行命令了，这些是你使用 `mkdir` 的多种不同方式。`mkdir` 命令是用来做什么的？他是用来创建目录的。如果你问出了这个问题，那么你需要回过头去复习一下命令表了，再好好记记你做的卡片吧。

创建新目录是什么意思？就是新建文件夹。以上练习中你做的事情就是在目录中创建多层目录。这就叫做“路径”（path），它是一种描述“temp 文件夹下的 stuff 文件夹下的 things 文件夹”的方式。它是你想在计算机的文件夹树中放入某些东西时的路径指向，它构成了你计算机的硬盘。

### 警告！

在这个附录中，我将用 `/` 来表示路径，因为它适用于所有的电脑。然而，Windows 用户需要知道，你们也可以用 `\`。

## 55.5.3 附加练习

- “路径”的概念可能一开始会让你感到困惑。别担心，我们之后会多次用到这个概念，你会慢慢明白的。
- 在 temp 目录中再创建 20 个不同层级的目录。在图形界面的文件查看器中查看这些文件夹。
- 创建一个名称用 “ ” 括起来的目录：`mkdir "I Have Fun"`
- 如果临时文件夹已经存在了你的电脑就会报错。用 `cd` 切换到一个你能控制的工作目录下，然后再试。Windows 桌面是一个很好的选择。

## 附录练习 5 切换目录 (cd)

在这个练习中，你将学习如何使用 `cd` 命令从一个目录切换到另一个目录。

### 55.6.1 跟我做

在这部分练习中我会再给你一次指导说明：



- 不用输入 `$`（Unix 系统）或者 `>`（Windows 系统）。
- 你输入 `$` 或者 `>` 后面的内容，然后回车。比如如果我写的是 `$ cd`，你就只用输入 `cd` 然后回车就行。
- 回车之后你会在 `$` 或者 `>` 之后看到你的输出结果。
- 每次练习之前要先用 `pwd` 和 `cd ~` 回到 home，回到你最开始的地方。

## Linux/macOS

### 练习 5 会话

```
$ cd temp
$ pwd
~/temp
$ cd stuff
$ pwd
~/temp/stuff
$ cd things
$ pwd
~/temp/stuff/things
$ cd orange/
$ pwd
~/temp/stuff/things/orange
$ cd apple/
$ pwd
~/temp/stuff/things/orange/apple
$ cd pear/
$ pwd
~/temp/stuff/things/orange/apple/pear
$ cd grape/
$ pwd
~/temp/stuff/things/orange/apple/pear/grape
$ cd ..
$ cd ..
$ pwd
~/temp/stuff/things/orange/apple
$ cd ..
$ cd ..
$ pwd
~/temp/stuff/things
$ cd ../../..
$ pwd
~/
$ cd temp/stuff/things/orange/apple/pear/grape
$ pwd
~/temp/stuff/things/orange/apple/pear/grape
$ cd ../../../../../../../../../../
$ pwd
~/
$
```

## Windows

### 练习 5 Windows 会话

```
> cd temp
> pwd

Path
----
C:\Users\zed\temp

> cd stuff
> pwd

Path
----
C:\Users\zed\temp\stuff

> cd things
> pwd

Path
----
C:\Users\zed\temp\stuff\things

> cd orange
> pwd

Path
----
C:\Users\zed\temp\stuff\things\orange

> cd apple
> pwd

Path
----
C:\Users\zed\temp\stuff\things\orange\apple

> cd pear
> pwd

Path
----
C:\Users\zed\temp\stuff\things\orange\apple\pear
```

```
> cd grape
> pwd

Path
----
C:\Users\zed\temp\stuff\things\orange\apple\pear\grape

> cd ..
> cd ..
> cd ..
> pwd

Path
----
C:\Users\zed\temp\stuff\things\orange

> cd ../../..
> pwd

Path
----
C:\Users\zed\temp\stuff

> cd ..
> cd ..
> cd temp/stuff/things/orange/apple/pear/grape
> cd ../../../../../../../
> pwd

Path
----
C:\Users\zed

>
```

## 55.6.2 你学到的

你已经在上一个练习中创建了以上这些目录，你刚才只是用 `cd` 命令在这些目录之间来回移动，同时在练习中我还用了 `pwd` 命令来看自己当前所处的位置，所以别把 `pwd` 输出的内容当作命令输入进去。例如，在第三行，你看到 `~/temp`，但那只是 `pwd` 命令的输出结果，不要把

它作为你要输入的内容。

你还应该看到我如何使用 `..` 命令来沿着路径向上。

### 55.6.3 附加练习

在一个拥有图形用户界面（graphical user interface, GUI）的电脑上学习命令行界面（command line interface, CLI）的一个非常重要的事情就是要明白它们是如何一起工作的。我最早开始使用计算机的时候还没有 GUI，我们在 DOS 界面上进行所有的操作。后来，当计算机变成强大的图形界面时，我很容易就能把一些 CLI 的目录和 GUI 上面的目录和 GUI 的窗口和文件夹对应上。

然而如今大多数人对 CLI、路径和目录毫无概念。事实上，也很难教会他们。唯一可能的办法就是持续地去用 CLI，直到有一天你用起 CLI 来会跟 GUI 一样自然流畅。

这就需要你花时间去寻找 GUI 下文件查看器里的目录，然后在 CLI 下切换到这些目录。以下是你接下来要做的：

- 用一个命令切换到 `apple` 目录下。
- 用一个命令切换回 `temp` 目录，但不是续着上一步来做。
- 试试如何用一个命令切换到你的“home 目录”。
- 切换到你的 Document 目录下，然后用 GUI 下的文件查看器找到它。（MacOS 下是 Finder，Windows 下是文件资源管理器，即“我的电脑”或“计算机”）
- 切换到你的 Downloads 目录，然后用你的文件浏览器找到它。
- 用你的文件浏览器找到其他目录，然后在 CLI 下切换到该目录。
- 还记得你给目录名加过引号吗？你也可以在命令中加入引号，比如，如果你有一个目录是 `I Have Fun`，然后你可以输入：`cd "I Have Fun"`。

## 附录练习 6 列示目录 (ls)

在这个练习中你将学习如何用 `ls` 命令列示一个目录中的内容。

### 55.7.1 跟我做

在你开始之前，确保你回到 `temp` 的上一层目录。如果你不知道你在哪儿，用 `pwd` 来查看，然后切换到要求的地方。

## Linux/macOS

### 练习 6 会话

```
$ cd temp
$ ls stuff
$ cd stuff
$ ls things
$ cd things
$ ls orange
$ cd orange
$ ls apple
$ cd apple
$ ls pear
$ cd pear
$ ls
$ cd grape
$ ls
$ cd ..
$ ls grape
$ cd ../../../
$ ls orange
$ cd ../../
$ ls stuff

$
```

## Windows

### 练习 6 Windows 会话

```
> cd temp
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:03 AM		stuff

```
> cd stuff
> ls
```

Directory: C:\Users\zed\temp\stuff

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:03 AM		things

```
> cd things
> ls
```

Directory: C:\Users\zed\temp\stuff\things

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:03 AM		orange

```
> cd orange
> ls
```

Directory: C:\Users\zed\temp\stuff\things\orange

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:03 AM		apple

```
> cd apple
> ls
```

Directory: C:\Users\zed\temp\stuff\things\orange\apple

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:03 AM		pear

```
> cd pear
> ls
```

Directory: C:\Users\zed\temp\stuff\things\orange\apple\pear

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:03 AM		grape

```
> cd grape
> ls
> cd ..
> ls
```

Directory: C:\Users\zed\temp\stuff\things\orange\apple\pear

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:03 AM		grape

```
> cd ..
> ls
```

Directory: C:\Users\zed\temp\stuff\things\orange\apple



```
Mode                LastWriteTime         Length      Name
----                -
d-----          12/17/2011  9:03 AM              pear
```

```
> cd ../../..
> ls
```

Directory: C:\Users\zed\temp\stuff

```
Mode                LastWriteTime         Length      Name
----                -
d-----          12/17/2011  9:03 AM      things
```

```
> cd ..
> ls
```

Directory: C:\Users\zed\temp

```
Mode                LastWriteTime         Length      Name
----                -
d-----          12/17/2011  9:03 AM      stuff
```

```
>
```

## 55.7.2 你学到的

`ls` 命令列示出了你当前所在目录的内容。你能看到我使用 `cd` 命令在不同目录之间切换，然后列示出它们里面有些什么内容，然后让我决定接下来要去哪个目录。

`ls` 命令有很多选项，我们会在学习 `help` 命令时学习如何获取帮助。

## 55.7.3 附加练习

- 把每一个命令都输一遍，你必须通过输入来学习这些命令，只是读它们是不够的。

- 在 Unix 下，让你在 temp 目录下，试试 `ls -lR` 命令。
- 在 Windows 系统下，用 `dir -R` 做同样的操作。
- 用 `cd` 去到你电脑上的其他目录，然后用 `ls` 看看它们里面有什么。
- 把新的问题添加到你的本子上。我知道你可能会有一些，因为关于这个命令的内容我没有全讲到。
- 记住如果你迷路了，用 `ls` 和 `pwd` 命令查看你在哪儿，然后用 `cd` 命令去到你应该去的地方。

## 附录练习 7 移除目录 (rmdir)

在这个练习中，你将学习如何移除一个空目录。

### 55.8.1 跟我做

#### Linux/macOS

##### 练习 7 会话

```
$ cd temp
$ ls stuff
$ cd stuff/things/orange/apple/pear/grape/
$ cd ..
$ rmdir grape
$ cd ..
$ rmdir pear
$ cd ..
$ ls apple
$ rmdir apple
$ cd ..
$ ls orange
$ rmdir orange
$ cd ..
$ ls things
$ rmdir things
$ cd ..
$ ls stuff
$ rmdir stuff
$ pwd
~/temp
$
```

### 警告！

如果你在 MacOS 系统下尝试用 `rmdir` 命令，  
但是系统拒绝移除这个目录，即使你百分百确定它是空的，事实上的确有个文件在里面，叫做 `.DS_Store` 。遇到这种情况，输入 `rm -rf <dir>` （将 `<dir>` 替换成你要移除的目录名）。

## Windows

### 练习 7 Windows 会话

```
> cd temp
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:03 AM		stuff

```
> cd stuff/things/orange/apple/pear/grape/
> cd ..
> rmdir grape
> cd ..

> rmdir pear
> cd ..
> rmdir apple
> cd ..
> rmdir orange
> cd ..
> ls
```

Directory: C:\Users\zed\temp\stuff

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:14 AM		things

```
> rmdir things
> cd ..
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/17/2011 9:14 AM		stuff

```
> rmdir stuff
> pwd

Path
----
C:\Users\zed\temp

> cd ..
>
```

## 55.8.2 你学到的

我现在开始把这些目录混在一起用了，所以你一定要专心，确保自己都输对了。如果你犯错了，只能说明你不专心。如果你发现自己犯了很多错，休息一下，或者干脆今天就不学了，明天再继续。

在这个例子中，你学会了如何移除一个目录，非常简单。你只需要去到它的上层目录，然后输入 `rmdir <dir>`，用你要移除的目录名替换掉 `<dir>` 即可。

## 55.8.3 附加练习

- 创建 20 个目录，然后移除它们。
- 创建一个 10 层路径的目录，然后一次移除一个，就像我之前做的那样。
- 如果你试着移除一个有内容的目录，你会收到报错。我会在后面的练习中教你如何移除它们。

## 附录练习 8 来回移动（pushd, popd）

在这个练习中，你将学习如何用 `pushd` 命令保存你当前的位置然后去到一个新的位置，以及如何用 `popd` 命令返回之前保存的位置。

## 55.9.1 跟我做

### Linux/macOS

#### 练习 8 会话

```
$ cd temp
$ mkdir i/like/icecream
$ pushd i/like/icecream ~/temp
~/temp/i/like/icecream ~/temp
$ popd
~/temp
$ pwd
~/temp
$ pushd i/like
~/temp/i/like ~/temp
$ pwd
~/temp/i/like
$ pushd icecream
~/temp/i/like/icecream ~/temp/i/like ~/temp
$ pwd
~/temp/i/like/icecream
$ popd
~/temp/i/like ~/temp
$ pwd
~/temp/i/like
$ popd
~/temp
$ pushd i/like/icecream
~/temp/i/like/icecream ~/temp
$ pushd
~/temp ~/temp/i/like/icecream
$ pwd
~/temp
$ pushd
~/temp/i/like/icecream ~/temp
$ pwd
~/temp/i/like/icecream
$
```

## Windows

### 练习 8 Windows 会话

```
> cd temp
> mkdir i/like/icecream
```

Directory: C:\Users\zed\temp\i\like

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/20/2011 11:05 AM		icecream

```
> pushd i/like/icecream
> popd
> pwd
```

Path  
----  
C:\Users\zed\temp

```
> pushd i/like
> pwd
```

Path  
----  
C:\Users\zed\temp\i\like

```
> pushd icecream
> pwd
```

Path  
----  
C:\Users\zed\temp\i\like\icecream

```
> popd
> pwd
```

Path  
----  
C:\Users\zed\temp\i\like

```
> popd
```

>

### 警告！

在 Windows 系统下，你一般不用像 Linux 系统那样用 `-p`，但是我想这应该是最近的更新，如果你用老的 Windows 系统的 Powershell，应该还是需要 `-p` 的，所以每个人的情况可能不太一样，你可以试试看。

## 55.9.2 你学到的

你正在通过这些命令进入程序员的世界，这些命令很常用，所以我必须要教给你们。它们能让你暂时地去到别的目录，然后再回来，并在两者之间随意切换。

`pushd` 命令会把你当前的目录“push”到一个列表里，然后它会切换到另外一个目录，就好像在说：“保存我现在的位置，然后去到那儿”。

`popd` 命令则是把你从你之前去到的目录那里拉回来。

最后，在 Unix 下使用 `pushd` 的话，如果你后面不加任何东西，它会在你的当前目录和你之前保存的目录之间来回切换。但是在 Powershell 下这个就不适用了。

## 55.9.3 附加练习

- 用这些命令在你电脑的目录之间来回移动。
- 移除 `i/like/icecream` 目录，然后自己创建一些，并在它们中间来回切换。
- 跟自己解释 `pushd` 和 `popd` 的输入结果，注意它和堆栈的概念很类似。
- 虽然你已经知道了，但是要记住 `mkdir -p`（在 Linux/MacOS 下）会创建一个完整的路径，即使所有的目录都不存在。这也就是我在本练习开头所做的。
- 记住 Window 也会创建一个完整的路径，并且不需要 `-p`。

## 附录练习 9 创建空文件 (Touch, New-Item)

在这个练习中你将学习如何使用 `touch`（MacOS）或者 `new-item`（Windows）命令来创建空文件。



## 55.10.1 跟我做

### Linux/macOS

#### 练习 9 会话

```
$ cd temp
$ touch iamcool.txt
$ ls iamcool.txt
$
```

### Windows

#### 练习 9 Windows 会话

```
> cd temp
> New-Item iamcool.txt -type file
> ls

Directory: C:\Users\zed\temp


Mode                LastWriteTime         Length      Name
----                -
a---             12/17/2011 9:03 AM             iamcool.txt

>
```

## 55.10.2 你学到的

你学习了如何创建空文件。在 Unix 系统下用 `touch`，在 Windows 系统下用 `New-Item`。

## 55.10.3 附加练习

- Unix：创建一个目录，切换到该目录下，然后在它里面创建一个文件，然后再切换到该目录的上一次，用 `rmdir` 命令移除该目录。你会收到报错，试着理解一下为什么。
- Windows：做同样的事情，但是你不会收到报错，你会收到一个提示符问你是否真的要移除这个目录。

# 附录练习 10 复制文件 (cp)

在这个练习中，你将学习如何用 `cp` 命令把一个文件从一个地址复制到另一个地址。

## 55.11.1 跟我做

### Linux/macOS

练习 10 会话

```
$ cd temp
$ cp iamcool.txt neat.txt
$ ls
iamcool.txt neat.txt
$ cp neat.txt awesome.txt
$ ls
awesome.txt iamcool.txt neat.txt
$ cp awesome.txt thefourthfile.txt
$ ls
awesome.txt iamcool.txt neat.txt thefourthfile.txt
$ mkdir something
$ cp awesome.txt something/
$ ls
awesome.txt iamcool.txt neat.txt something thefourthfile.txt
$ ls something/ awesome.txt
$ cp -r something newplace
$ ls newplace/ awesome.txt
$
```

### Windows

练习 10 Windows 会话

```
> cd temp
> cp iamcool.txt neat.txt
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/22/2011 4:49 PM	0	iamcool.txt
-a---	12/22/2011 4:49 PM	0	neat.txt

```
> cp neat.txt awesome.txt
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/22/2011	4:49	PM 0 awesome.txt
-a---	12/22/2011	4:49	PM 0 iamcool.txt
-a---	12/22/2011	4:49	PM 0 neat.txt

```
> cp awesome.txt thefourthfile.txt
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/22/2011	4:49	PM 0 awesome.txt
-a---	12/22/2011	4:49	PM 0 iamcool.txt
-a---	12/22/2011	4:49	PM 0 neat.txt
-a---	12/22/2011	4:49	PM 0 thefourthfile.txt

```
> mkdir something
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/22/2011 4:52 PM		something

```
> cp awesome.txt something/  
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/22/2011	4:52	PM something
-a---	12/22/2011	4:49	PM 0 awesome.txt
-a---	12/22/2011	4:49	PM 0 iamcool.txt
-a---	12/22/2011	4:49	PM 0 neat.txt
-a---	12/22/2011	4:49	PM 0 thefourthfile.txt

```
> ls something
```

Directory: C:\Users\zed\temp\something

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/22/2011 4:49 PM	0	awesome.txt

```
> cp -recurse something newplace  
> ls newplace
```

Directory: C:\Users\zed\temp\newplace

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	12/22/2011 4:49 PM	0	awesome.txt

## 55.11.2 你学到的

现在你会复制文件了，它很简单。在这个练习中，我还创建了一个新目录，并且把一个文件复制到了那个新目录中。

我要告诉你一个关于程序员和系统管理员的秘密。他们很懒，我也很懒，我的朋友同样很懒。这也正是为什么我们要用计算机。我们热衷于让计算机为我们做无聊的事情。这个练习到目前为止你已经输入了很多重复的命令来学习它们，但是实际情况不是这样的，通常如果你发现自己在做一些无聊和重复的事情，就已经有一个程序员在想办法如何让这件事情变得简单，你只是不知道而已。

关于程序员的另一件事情就是，他们可能没你想的那么聪明。如果你觉得他们输入的东西有多么高深莫测，那你就错了。在你做练习的时候，你可以先试着先想想这些命令的名字和含义，然后再输入。一般你会想到一个名字或者一些缩写。如果你还是想象不出来，就回过头复习一下或者在网上搜一搜。

## 5.11.3 附加练习

- 用 `cp -r` 命令复制更多包含文件的目录。
- 把一个文件复制到你的 home 目录或者桌面。
- 在图形用户界面找到这些文件，然后用文本编辑器打开它们。
- 注意我有时候会放一个 `/` 在目录结尾，这样是为了确保这是一个目录，如果不是的话，我会收到报错。

## 附录练习 11 移动文件（mv）

在这个练习中，你将会学习如何使用 `mv` 命令把一个文件从一个地方移动到另一个地方。

### 55.12.1 跟我做

#### Linux/macOS

##### 练习 11 会话

```
$ cd temp
$ mv awesome.txt uncool.txt
$ ls
newplace uncool.txt
$ mv newplace oldplace
$ ls
oldplace uncool.txt
$ mv oldplace newplace
$ ls
newplace uncool.txt
$
```

## Windows

### 练习 11 Windows 会话

```
> cd temp
> mv awesome.txt uncool.txt
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/22/2011	4:52	PM newplace
d----	12/22/2011	4:52	PM something
-a---	12/22/2011	4:49	PM 0 iamcool.txt
-a---	12/22/2011	4:49	PM 0 neat.txt
-a---	12/22/2011	4:49	PM 0 thefourthfile.txt
-a---	12/22/2011	4:49	PM 0 uncool.txt

```
> mv newplace oldplace
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
----	-----	-----	----
d----	12/22/2011	4:52	PM oldplace
d----	12/22/2011	4:52	PM something
-a---	12/22/2011	4:49	PM 0 iamcool.txt
-a---	12/22/2011	4:49	PM 0 neat.txt
-a---	12/22/2011	4:49	PM 0 thefourthfile.txt
-a---	12/22/2011	4:49	PM 0 uncool.txt

```
> mv oldplace newplace
> ls newplace
```

Directory: C:\Users\zed\temp\newplace

Mode	LastWriteTime	Length	Name
----	-----	-----	----

```
-a---      12/22/2011  4:49 PM      0      awesome.txt

> ls

Directory: C:\Users\zed\temp

Mode                LastWriteTime         Length      Name
----                -
d----      12/22/2011         4:52      PM      newplace
d----      12/22/2011         4:52      PM      something
-a---      12/22/2011         4:49      PM      0      iamcool.txt
-a---      12/22/2011         4:49      PM      0      neat.txt
-a---      12/22/2011         4:49      PM      0      thefourthfile.txt
-a---      12/22/2011         4:49      PM      0      uncool.txt

>
```

## 55.12.2 你学到的

移动文件，或者重命名，很简单：给出原来的名字和新的名字即可。

## 55.12.3 附加练习

- 将 newplace 目录下的一个文件移动到另一个目录下，然后再移动回来。

## 附录练习 12 浏览文件（less, MORE）

做这个练习需要用到目前为止已经学过的一些命令。你还需要一个能创建文本文档（.txt）的文本编辑器，以下是你要做的：

- 打开你的文本编辑器，在新文件中输入一些东西。在 macOS 下，你可以用 TextWrangler，在 Windows 系统下你可以用 Notepad++，在 Linux 下可以用 Gedit。其他任何文本编辑器也都可以。
- 把这个文件保存到桌面，然后命名为 test.txt。
- 在 Shell 中用你学到的命令把这个文件复制到当前的工作目录——temp 目录下。



做完这些，再完成下面的练习。

## 55.13.1 跟我做

### Linux/macOS

练习 12 会话

```
$ less test.txt [displays file here]
$
```

就是这些，输入 `q` 即可退出 `less` 浏览模式。

### Windows

练习 12 Windows 会话

```
> more test.txt [displays file here]
>
```

#### 警告！

在前面的练习结果中，我用了 `[displays file here]` 来指代程序的输出结果，因为有些输出结果比较复杂。你要知道你的输出结果不是这个。

## 55.13.2 你学到的

这只是查看文件内容的一种方法。它很有用，因为当文件有很多行的时候，它可以翻页。在附加练习部分你会做更多的操作。

## 55.13.3 附加练习

- 再次打开你的文本文件，通过复制粘贴的方法把内容扩充到 50-100 行。
- 再把它复制到 `temp` 目录下。
- 现在再做一遍练习，这一遍可以翻页，Unix 系统可以用 `空格键` 和 `w` 来上下翻页，Windows 系统直接用 `空格键` 即可。
- 再看看你创建的其他一些空文件。
- `cp` 命令会覆盖一些已经存在的文件，所以要复制的时候要小心。

## 附录练习 13 Stream 文件（cat）

在做这个练习之前你需要再多做一些准备工作，以便在练习中使用。用编辑器创建另一个名为 `test2.txt` 的文件，但是这次直接把它保存在 `temp` 目录下。

### 55.14.1 跟我做

#### Linux/macOS

练习 13 会话

```
$ less test2.txt [displays file here]
$ cat test2.txt I am a fun guy.
Don't you know why? Because I make poems, that make babies cry.
$ cat test.txt
Hi there this is cool.
$
```

#### Windows

练习 13 Windows 会话

```
> more test2.txt [displays file here]

> cat test2.txt I am a fun guy.
Don't you know why? Because I make poems, that make babies cry.
> cat test.txt
Hi there this is cool.
>
```

### 55.14.2 你学到的

你已经学习了第一个命令，这个命令只是为了让你检查一下那个文件确实在。然后你把这个文件 `cat` 到屏幕，`cat` 命令是把整个文件内容全部呈现到屏幕上，没有翻页或者停止。

### 55.14.3 附加练习

- 再创建几个文件并使用 `cat` 命令。
- Unix：试试 `cat test.txt test2.txt`，看看会发生什么。

- Windows: 试试 `cat test.txt,test2.txt` , 看看会发生什么。

## 附录练习 14 移除文件 (rm)

在这个练习中你将学习如何用 `rm` 命令移除（删除）一个文件。

### 55.15.1 跟我做

#### Linux

##### 练习 14 会话

```
$ cd temp
$ ls
uncool.txt iamcool.txt neat.txt something thefourthfile.txt
$ rm uncool.txt
$ ls
iamcool.txt neat.txt something thefourthfile.txt
$ rm iamcool.txt neat.txt thefourthfile.txt
$ ls something
$ cp -r something newplace
$
$ rm something/awesome.txt
$ rmdir something
$ rm -rf newplace
$ ls
$
```

#### Windows

##### 练习 14 Windows 会话

```
> cd temp
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
d----	12/22/2011	4:52	PM
d----	12/22/2011	4:52	PM
-a---	12/22/2011	4:49	PM 0
-a---	12/22/2011	4:49	PM 0
-a---	12/22/2011	4:49	PM 0
-a---	12/22/2011	4:49	PM 0

```
> rm uncool.txt
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
d----	12/22/2011	4:52	PM
d----	12/22/2011	4:52	PM
-a---	12/22/2011	4:49	PM 0
-a---	12/22/2011	4:49	PM 0
-a---	12/22/2011	4:49	PM 0

```
> rm iamcool.txt
> rm neat.txt
> rm thefourthfile.txt
> ls
```

Directory: C:\Users\zed\temp

Mode	LastWriteTime	Length	Name
d----	12/22/2011 4:52 PM		newplace
d----	12/22/2011 4:52 PM		something

```
> cp -r something newplace
> rm something/awesome.txt
> rmdir something
> rm -r newplace
> ls
```

```
>
```

## 55.15.2 你学到的

这个练习我们学习了如何删除文件。还记得之前我让你们用 `rmdir` 命令移除包含内容的目录时失败了吗？是因为你不能用 `rmdir` 移除包含内容的目录。要移除这个目录，首先要删除它里面的文件，这也正是这个练习中所学到的。

## 55.15.3 附加练习

- 删除 temp 目录中目前为止所有的练习文件。
- 当你对文件进行递归删除的时候要千万小心。（译者注：递归删除就是你想删一个文件夹,而这个文件夹下还有其它的东西,它就会先把其它的东西删掉,再删这个文件夹）

# 附录练习 15 退出 Terminal（exit）

## 55.16.1 跟我做

### Linux/macOS

练习 15 会话

```
$ exit
```

### Windows

练习 15 Windows 会话

```
> exit
```

## 55.16.2 你学到的

最后一个练习是如何退出 Terminal，非常简单，但是我需要你再做一些练习。

## 55.16.3 附加练习

在本速成课的最后，我想让你用一下帮助系统，看看以下这些命令的解释和用法，学习如何使用它们。

以下是 Unix 系统下你要查询的命令列表：

- xargs
- sudo
- chmod
- chown

以下是 Windows 系统下你要查询的命令列表：

- forfiles
- runas
- attrib
- icacls

弄明白这些是什么，试试用用这些命令，然后把它们添加到你的索引卡片上。

## 命令行后续

你已经完成了命令行速成教程，基本掌握了一些基础命令的用法。但其实还有很多的技巧和键序列你没有见过。我会在这个教程的最后引导你去搜索和了解它们。

## 55.17.1 Unix Bash References

在 Unix 系统下，你使用的 Shell 叫做 Bash。它不是最好的 shell，但它无处不在。以下是一些关于 Bash 的列表：

- **Bash Cheat Sheet** [https://learncodethehardway.org/unix/bash\\_cheat\\_sheet.pdf](https://learncodethehardway.org/unix/bash_cheat_sheet.pdf)  
created by Raphael and CC licensed.
- **Reference Manual** <http://www.gnu.org/software/bash/manual/bashref.html>

## 55.17.2 PowerShell References

在 Windows 系统下只有 Powershell。以下是一些关于 Powershell 的列表；

- **Owner's Manual**  
<http://technet.microsoft.com/en-us/library/ee221100.aspx>
- **Cheat Sheet**  
[https://download.microsoft.com/download/2/1/2/2122F0B9-0EE6-4E6D-BFD6-F9DCD27C07F9/WS12\\_QuickRef\\_Download\\_Files/PowerShell\\_LangRef\\_v](https://download.microsoft.com/download/2/1/2/2122F0B9-0EE6-4E6D-BFD6-F9DCD27C07F9/WS12_QuickRef_Download_Files/PowerShell_LangRef_v)
- **Master PowerShell** <http://powershell.com/cs/blogs/ebook/default.aspx>