Python编程

0基础到入门

逻辑教育 ◎著

前腾讯高级开发工程师
推荐入门书籍

目　录

视频教程及版权声明

《Python编程：零基础到入门》本书配套 视频教程 和 源码 ，请用微信扫描下方二维码免费开通学习权限！

第1章 Python环境搭建

1.1-Python介绍

1.1.1 Python来源(了解)

Python翻译成汉语是蟒蛇的意思，并且Python的logo也是两条缠绕在一起的蟒蛇的样子，然而Python语言和蟒蛇实际上并没有一毛钱关系。

Python语言是由荷兰程序员Guido van Rossum，江湖人称“龟叔”，独立开发完成初版的。“龟叔”曾供职于google，现任职于dropbox。1989年圣诞节期间，在阿姆斯特丹，为了打发圣诞节的无趣，决心开发一个新的脚本解释语言，作为ABC语言的一种继承，然后他就这么做了，并实现了（大神的能力）。之所以选中Python作为该编程语言的名字，是因为他是一个叫Monty Python喜剧团体的爱好者，其本意并不是想选条蟒蛇。

1.1.2 Python语言的特点(熟悉)

① 简单易学、明确优雅、开发速度快

- 简单易学：与 `C` 和 `Java` 比，Python的学习成本和难度曲线不是低一点，更适合新手入门，自底向上的技术攀爬路线。先订个小目标爬个小山，然后再往更高的山峰前进。而不像C和JAVA光语言学习本身，对于很多人来说就像珠穆朗玛峰一样高不可攀。
- 明确优雅：Python的语法非常简洁，代码量少，非常容易编写，代码的测试、重构、维护等都非常容易。一个小小的脚本，用C可能需要1000行，用JAVA可能几百行，但是用Python往往只需要几十行！
- 开发速度快：当前互联网企业的生命线是什么？产品开发速度！如果你的开发速度不够快，在你的产品推出之前别人家的产品已经上线了，你也就没有生存空间了，这里的真实例子数不胜数。那么，Python的开发速度说第二没人敢称第一!（不欢迎辩论^_^）

② 跨平台、可移植、可扩展、交互式、解释型、面向对象的动态语言

- 跨平台：Python支持Windows、Linux和MAC os等主流操作系统。
- 可移植：代码通常不需要多少改动就能移植到别的平台上使用。
- 可扩展：Python语言本身由C语言编写而成的，你完全可以在Python中嵌入C，从而提高代码的运行速度和效率。你也可以使用C语言重写Python的任何模块，从根本上改写Python，PyPy就是这么干的。
- 交互式：Python提供很好的人机交互界面，比如IDLE和IPython。可以从终端输入执行代码并获得结果，互动的测试和调试代码片断。
- 解释型：Python语言在执行过程中由解释器逐行分析，逐行运行并输出结果。
- 面向对象：Python语言具备所有的面向对象特性和功能，支持基于类的程序开发。
- 动态语言：在运行时可以改变其结构。例如新的函数、对象、甚至代码可以被引进，已有的函数可以被删除或是其他结构上的变化。动态语言非常具有活力。

③ “内置电池”，大量的标准库和第三方库

Python为我们提供了非常完善的基础库，覆盖了系统、网络、文件、GUI、数据库、文本处理等方方面面，这些

是随同解释器被默认安装的，各平台通用，你无需安装第三方支持就可以完成大多数工作，这一特点被形象地称作“内置电池（batteries included）”。

在程序员界，有一句话叫做“不要重复造轮子”。什么意思呢？就是说不要做重复的开发工作，如果对某个问题已经有开源的解决方案或者说第三方库，就不要自己去开发，直接用别人的就好。不要过分迷信自己的代码能力，要知道，能作为标准库被Python内置，必然在可靠性和算法效率上达到了目前最高水平，能被广泛使用的第三方库，必然也是经受了大量的应用考验。除非公司要求，不要自己去开发，请使用现成的库。那些造轮子的事情，就交给世界最顶尖的那一波程序员去干吧，没有极致的思维和数学能力，想创造好用的轮子是很难的。

④ 社区活跃，贡献者多，互帮互助

技术社区的存在就相当于程序员手中的指南针，没有指南针，很多时候，碰到了问题，就像无头的苍蝇只能到处乱飞，最终在茫茫的海洋中转晕致死。技术社区可以给我们对语言的学习和使用提供巨大的帮助，无论是前期的学习，还是日后的工作，只要有问题，技术社区的大牛都可以帮我们解决，有这些助力，可以帮我们更好地了解、学习和使用一门语言。技术社区同时还推动Python语言的发展方向，功能需求，促使公司企业更多的使用Python语言，招聘Python程序员。

然而，上面说的是国外。在国内，好像没有比较成熟，影响范围广的Python技术社区，还是说我见识浅薄不知道而已？据本人分析，有历史原因和Python流行过程中形成的习惯等因素，国外Python高手都喜欢用邮件列表、wiki等方式进行交流，而国内喜欢的论坛、bbs等没有形成规模，所以造成现在的状况。

因此，同学们，学好英语，去和世界范围的程序员交流吧！

⑤ 开源语言，发展动力巨大

Python是基于C语言编写的，并且使用GPL开源协议，你可以免费获取它的源代码，进行学习、研究甚至改进。众人拾柴火焰高，有更多的人参与Python的开发，促使它更好的发展，被更多的应用，形成良性循环。Python为什么会越来越火就是因为它的开放性，自由性，聚起了人气，形成了社区，有很多人在其中做贡献，用的人越来越多，自然就提高了市场占有率，企业、公司、厂家就不得不使用Python，提供的Python程序员岗位就越来越多，这就是开源的力量。

这里附带跟大家说一个代码封闭的问题。Python写的源代码通常是不加密的，如果要发布你的Python程序，实际上就是发布源代码，这一点跟C语言不同，C语言不用发布源代码，只需要把编译后的机器码（也就是你在Windows上常见的xxx.exe文件）发布出去。要从机器码反推出C代码基本是不可能的，所以，凡是编译型的语言，都没有这个问题，而解释型的语言，则必须把源码发布出去。如果你不想让别人看到或抄袭你写的python代码怎么办？使用类似py2exe的包装工具，将python源码转换成一个类似于exe可执行文件的形式，但这个也不是绝对保险，只是增加了反编译的门槛和难度，对于有经验的人而言，一样可以获得你的源代码。

你可能要问，我要通过写代码编软件卖出去挣钱怎么办？少年！目前的互联网时代，靠卖软件授权的商业模式越来越少了，靠网站服务和移动应用卖服务的模式越来越多了，这种模式不需要把源码给别人。再说了，现在如火如荼的开源运动和互联网自由开放的精神是一致的，互联网上有无数非常优秀的像Linux生态圈一样的开源项目，我们千万不要高估自己写的代码真的有非常大的“商业价值”。在Python的世界，开源是王道，不要纠结你的代码被抄袭模仿，而是尽量提高自己的水平和能力，这才是立身之本。

1.1.3 Python的应用方向(了解)

① 常规软件开发

② 科学计算

③ 自动化运维

④ 云计算

⑤ WEB开发

⑥ 网络爬虫

⑦ 数据分析

⑧ 人工智能

1.1.4 Python之禅(了解)

最后，让我们以Python的官方格言，也就是俗称的Python之禅来结束对Python的介绍。在Python的IDLE或者交互式解释器中，输入 `import this` ，你就会看到下面的一段话：

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

翻译过来就是：

```
优美胜于丑陋（Python 以编写优美的代码为目标）
明了胜于晦涩（优美的代码应当是明了的，命名规范，风格相似）
```

简洁胜于复杂（优美的代码应当是简洁的，不要有复杂的内部实现）
复杂胜于凌乱（如果复杂不可避免，那代码间也不能有难懂的关系，要保持接口简洁）
扁平胜于嵌套（优美的代码应当是扁平的，不能有太多的嵌套）
间隔胜于紧凑（优美的代码有适当的间隔，不要奢望一行代码解决问题）
可读性很重要（优美的代码是可读的）
即便假借特例的实用性之名，也不可违背这些规则（这些规则至高无上）
不要包容所有错误，除非你确定需要这样做（精准地捕获异常，不写 except:pass 风格的代码）
当存在多种可能，不要尝试去猜测而是尽量找一种，最好是唯一一种明显的解决方案（如果不确定，就用穷举法）
虽然这并不容易，因为你不是 Python 之父（这里的 Dutch 是指 Guido ）
做也许好过不做，但不假思索就动手还不如不做（动手之前要细思量）
如果你无法向人描述你的方案，那肯定不是一个好方案；反之亦然（方案测评标准）
命名空间是一种绝妙的理念，我们应当多加利用（倡导与号召）

1.2-编译器与解释器

编译器/解释器：高级语言与机器之间的翻译官

都是将代码翻译成机器可以执行的二进制机器码，只不过在运行原理和翻译过程有不同而已。

1.2.1 那么两者有什么区别呢？

编译器：先整体编译再执行

解释器：边解释边执行

用一个通俗的例子进行比喻：我们去饭馆吃饭，点了八菜一汤。编译器的方式就是厨师把所有的菜给你全做好了，一起给你端上来，至于你在哪吃，怎么吃，随便。解释器的方式就是厨师做好一个菜给你上一个菜，你就吃这个菜，而且必须在饭店里吃。

编译方式：运行速度快，但任何一个小改动都需要整体重新编译。可脱离编译环境运行。代表语言是C语言。

解释方式：运行速度慢，但部分改动不需要整体重新编译。不可脱离解释器环境运行。代表语言是Python语言。

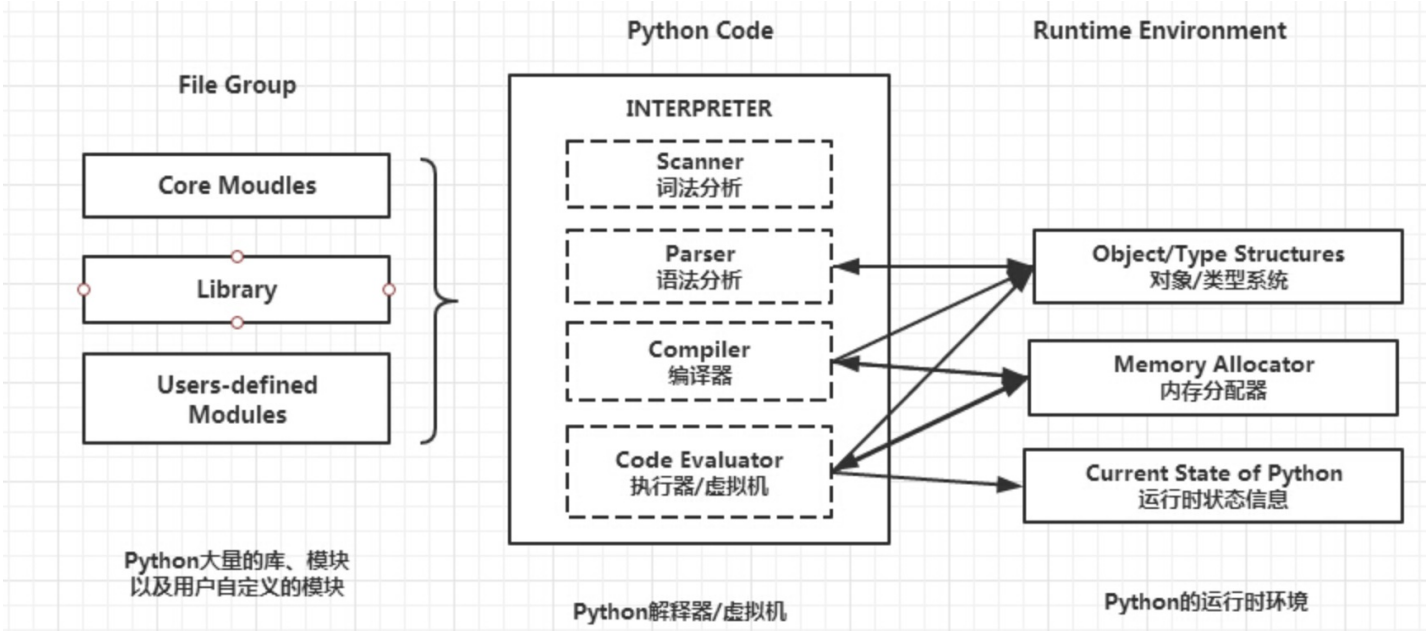
1.2.2 Python解释器种类

Python有好几种版本的解释器：

- CPython：官方版本的解释器。这个解释器是用C语言开发的，所以叫CPython。CPython是使用最广的Python解释器。我们通常说的、下载的、讨论的、使用的都是这个解释器。
- Ipython：基于CPython之上的一个交互式解释器，在交互方式上有所增强，执行Python代码的功能和CPython是完全一样的。CPython用>>>作为提示符，而IPython用In [序号]:作为提示符。
- PyPy：一个追求执行速度的Python解释器。采用JIT技术，对Python代码进行动态编译（注意，不是解释），可以显著提高Python代码的执行速度。绝大部分CPython代码都可以在PyPy下运行，但还是有一些不同的，这就导致相同的Python代码在两种解释器下执行可能会有不同的结果。
- Jython：运行在Java平台上的Python解释器，可以直接把Python代码编译成Java字节码执行。
- IronPython：和Jython类似，只不过IronPython是运行在微软.Net平台上的Python解释器，可以直接把Python代码编译成.Net的字节码。

1.2.3 Python的运行机制

Python作为动态解释性语言，其运行机制可参考下：



图中，在解释器与右边的对象/类型系统、内存分配器之间的箭头表示“使用”关系，而与运行时状态之间的箭头表示“修改”关系，即Python在执行的过程中会不断地修改当前解释器所处得状态，在不同的状态之间切换。

都说解释器慢，Python也有想办法提高一下运行速度的，那就是使用pyc文件。这点参考了JAVA的字节码做法，但并不完全类同。

我们编写的代码一般都会保存在以 `.py` 为后缀的文件中。在执行程序时，解释器逐行读取源代码并逐行解释运行。每执行一次，就重复一次这个过程，这其中耗费了大量的重复性的解释工作。为了减少这一重复性的解释工作，Python引入了 `pyc` 文件， `pyc` 文件是将 `py` 文件的解释结果保存下来的文件，这样，下次再运行的时候就不用再解释了，直接使用 `pyc` 文件就可以了，这无疑大大提高了程序运行速度。

1.3-Python环境搭建

Python是一个跨平台、可移植的编程语言，因此可在windows、Linux和Mac OS X系统中安装使用。

安装完成后，你会得到Python解释器环境，可以通过终端输入python命令查看本地是否已经按照python以及python版本。这里有一点需要注意的是，如果没有将python的安装目录添加到环境变量中，会报错（python不是内部命令或外部命令，也不是可执行程序）。需要把python安装环境添加到环境变量中。

1.3.1 Python下载

python官网：https://www.python.org/

`Source code`这里是源码安装的意思，用于`Linux`安装

python文档地址：https://www.python.org/doc/

建议大家下载和学习python 3版本，当然也可以下载python 2版本。有人会纠结到底选择python 2 还是python 3 ？

其实这里并不需要纠结，官方的解释在这里：https://wiki.python.org/moin/Python2orPython3， 其全文的意思基本上就是Python 3是语言的现在和未来，Python 2必将成为过去式。

1.3.2 各个版本之间的区别

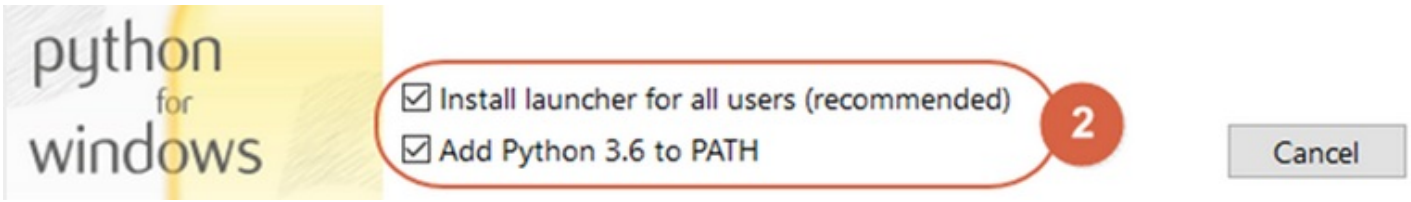
x86是32位，x86-64是64位：

- Download Windows x86-64 embeddable zip file
- Download Windows x86-64 executable installer
- Download Windows x86-64 web-based installer

- web-based installer-->基于web联网下载安装
- executable installer-->通过exe可执行文件进行安装
- embeddable zip file-->已经下载好的压缩文件可以嵌入其他应用中（免安装版本）

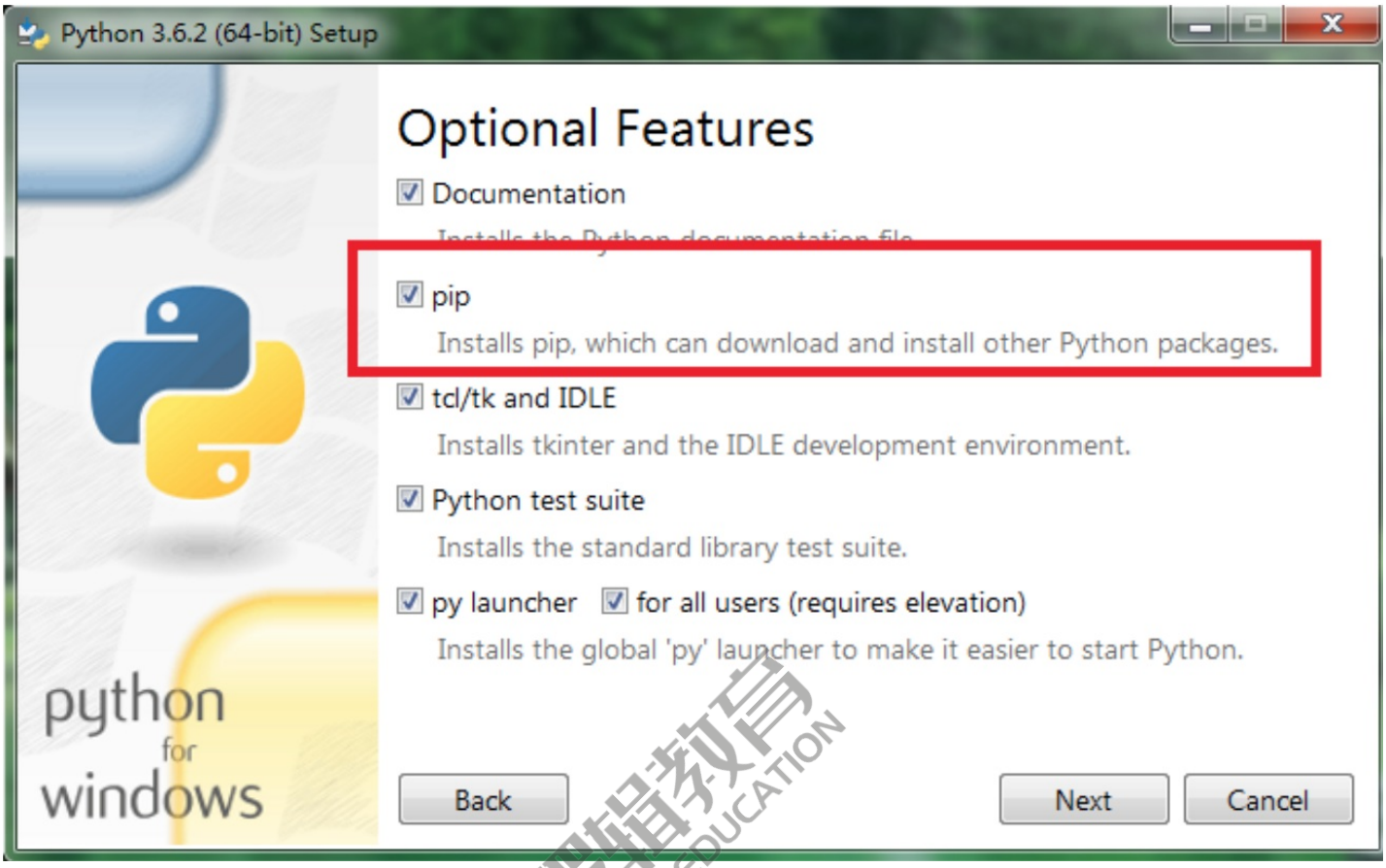
1.3.3 Windows下安装Python

到官网下载安装包，接下来基本就是点下一步就可以了。



在安装的时候，把这个选项勾选上。会直接添加环境变量。

1.3.4 添加pip



安装完成后，可以验证一下。在CMD界面中，输入python。如果可以看到下面的界面说明安装以及成功了。

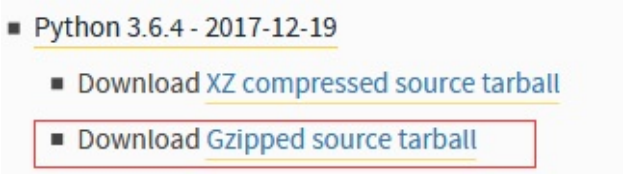
```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

1.3.5 Linux下安装Python

绝大多数的 `Linux` 发行版都默认安装python，但是默认安装的是python 2。如果想要安装python 3 版本需要自己来安装。

最好在安装前，在虚拟机中拍快照。

到官网上找到这个下载：https://www.python.org/downloads/source/



接下来到到Linux系统中，解压下载下来的文件

- 安装python3.6可能使用的依赖

```
yum install openssl-devel bzip2-devel expat-devel gdbm-devel readline-devel sql
```

```
ite-devel

yum -y install gcc*
```

- 到python官网找到下载路径, 用wget下载

```
wget https://www.python.org/ftp/python/3.6.4/Python-3.6.4.tgz
```

- 解压tgz包

```
tar -zxvf Python-3.6.4.tgz
```

- 把python移到/usr/local下面

```
mv Python-3.6.4 /usr/local
```

- 删除旧版本的python依赖

```
ll /usr/bin | grep python

rm -rf /usr/bin/python
```


- 进入python目录

```
cd /usr/local/Python-3.6.4/
```

- 配置

```
./configure --prefix=/usr/local/python3.6  (这里一定要指定目录, 要不然后面很麻烦)
```

- 编译 make

```
make
```

- 编译，安装

```
make install
```

- 删除旧的软链接，创建新的软链接到最新的python

```
rm -rf /usr/bin/python

ln -s /usr/local/bin/python3.6 /usr/bin/python

ln -s /usr/local/bin/python3.6/bin/pip3.6    /usr/bin/pip3

python -V
```

1.3.6 可能会出现的问题

`zipimport.ZipImportError: can’t decompress data` 因为缺少zlib 的相关工具包导致的，知道了问题所在，那么我们只需要安装相关依赖包即可。

```
yum -y install zlib*
```

1.3.7 MAC下安装python

通常MAC系统自带的python是2.7版本的，要安装python 3版本去官网下载就可以了，这里就不详细的说明了。

1.4-pip的介绍和使用

我们都知道python有很多的第三方库或者说是模块。这些库针对不同的应用，发挥不同的作用。我们在实际的项目中肯定会用到这些模块。那如何将这些模块导入到自己的项目中呢？

Python官方的PyPi仓库为我们提供了一个统一的代码托管仓库，所有的第三方库，甚至你自己写的开源模块，都可以发布到这里，让全世界的人分享下载 。

python有两个著名的包管理工具easy_install和pip。在python 2中easy_install是默认安装的，而pip需要我们手动安装。随着Python版本的提高，easy_install已经逐渐被淘汰，但是一些比较老的第三方库，在现在仍然只能通过easy_install进行安装。目前，pip已经成为主流的安装工具，自Python 2 >=2.7.9或者Python 3.4以后默认都安装有pip

1.4.1 pip的基础使用

在命令行下，输入pip，回车可以看到帮助说明：

```
C:\Users\Administrator>pip

Usage:
  pip <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
  freeze                      Output installed packages in requirements format.
  list                        List installed packages.
  show                        Show information about installed packages.
  check                       Verify installed packages have compatible dependencies.
  config                      Manage local and global configuration.
  search                      Search PyPI for packages.
  wheel                       Build wheels from your requirements.
  hash                        Compute hashes of package archives.
  completion                  A helper command used for command completion.
  help                        Show help for commands.

General Options:
  -h, --help                  Show help.
  --isolated                  Run pip in an isolated mode, ignoring environment variables and user configuration.
  -v, --verbose               Give more output. Option is additive, and can be used up to 3 times.
  -V, --version               Show version and exit.
  -q, --quiet                 Give less output. Option is additive, and can be used up to 3 times (corresponding to WARNING, ERROR, and CRITICAL
                              logging levels).
  --log <path>                Path to a verbose appending log.
  --proxy <proxy>             Specify a proxy in the form [user:passwd@]proxy.server:port.
  --retries <retries>         Maximum number of retries each connection should attempt (default 5 times).
  --timeout <sec>             Set the socket timeout (default 15 seconds).
  --exists-action <action>    Default action when a path already exists: (s)witch, (i)gnore, (w)ipe, (b)ackup, (a)bort).
  --trusted-host <hostname>   Mark this host as trusted, even though it does not have valid or any HTTPS.
  --cert <path>               Path to alternate CA bundle.
  --client-cert <path>        Path to SSL client certificate, a single file containing the private key and the certificate in PEM format.
  --cache-dir <dir>           Store the cache data in <dir>.
  --no-cache-dir              Disable the cache.
  --disable-pip-version-check
                              Don't periodically check PyPI to determine whether a new version of pip is available for download. Implied with --no-
```

- 查看pip版本

```
pip -V
pip --version
```

```
chufeng:~ binbin$ pip -V
pip 19.0.3 from /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/
pip (python 3.6)
chufeng:~ binbin$ pip --version
pip 19.0.3 from /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/
pip (python 3.6)
```

- 普通安装

```
pip install requests
pip install ipython
package-->包名
```

- 指定版本安装

```
pip install robotframework==2.8.7
```

- 卸载已安装的库

```
pip uninstall requests
```

- 列出已经安装的库

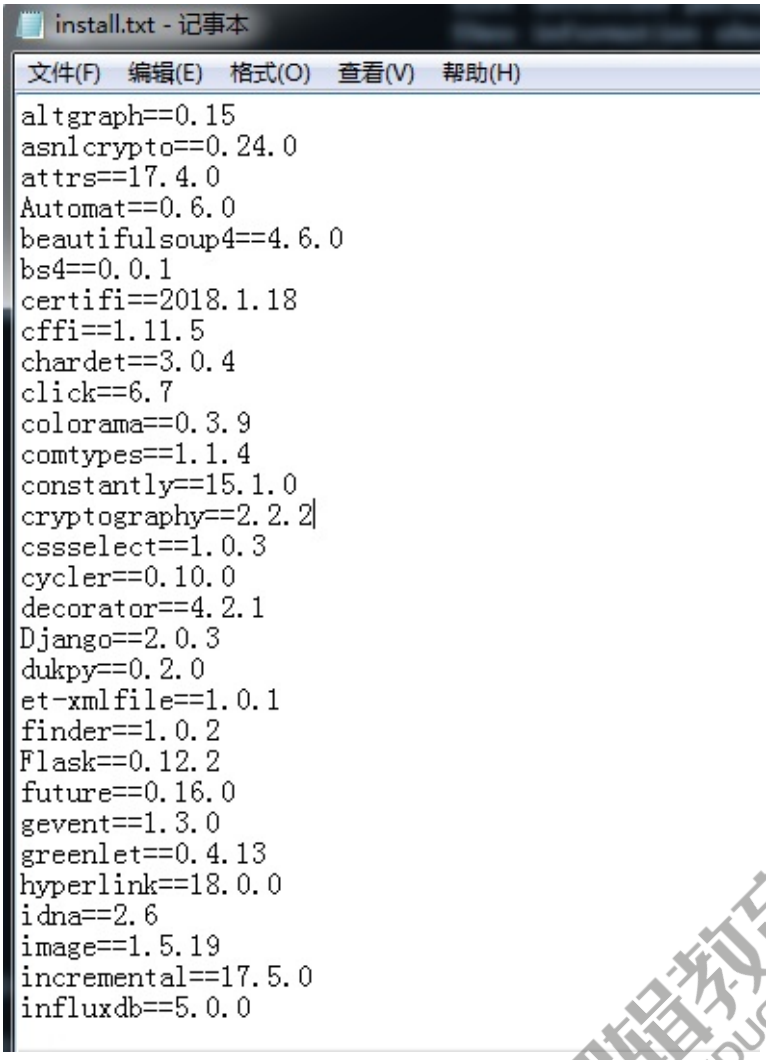
```
pip list
pip freeze
```


- 显示所安装包的信息

```
pip show package
pip show -f package
```

- 升级指定的包
  `pip install -U package`
- 查看可升级的包
  `pip list -o`
- 将已经安装的库列表保存到文本文件中

```
pip freeze > D:\桌面\install.txt
```



这个功能非常常用、好用！经常被用作项目环境依赖文件。

- 根据依赖文件批量安装库

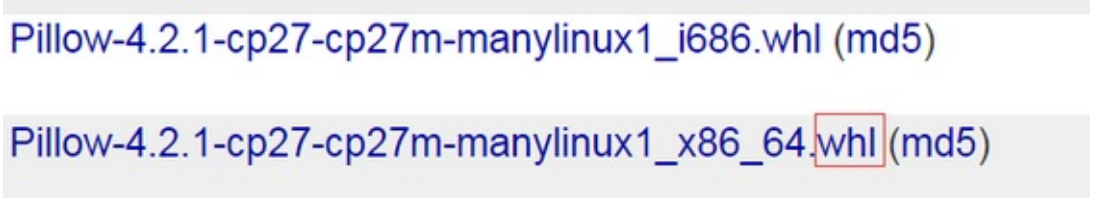
```
pip install -r install.txt
```

上面的txt文件，批量安装第三方库

- 使用wheel文件安装

除了使用上面的方式联网进行安装外，还可以将安装包也就是wheel格式的文件，下载到本地，然后使用pip进行安装。比如我在PYPI上提前下载的pillow库的wheel文件，后缀名为whl

地址：https://www.lfd.uci.edu/~gohlke/pythonlibs/



可以使用 `pip install pillow-4.2xxxxxxx.whl` 的方式离线进行安装

1.5-代码编辑器

Python解释器、pip工具箱和virtuanlenv虚拟环境都安装好了后，基本的Python环境就搭建好了，可以开始我们的“搬砖”之旅了。但是现在还缺一个好用的编辑器，这里推荐大家用pycharm。当然如果你有一些其他的编辑器也可以，比如sublime_text，notepad++，vscode，Anaconda等等。

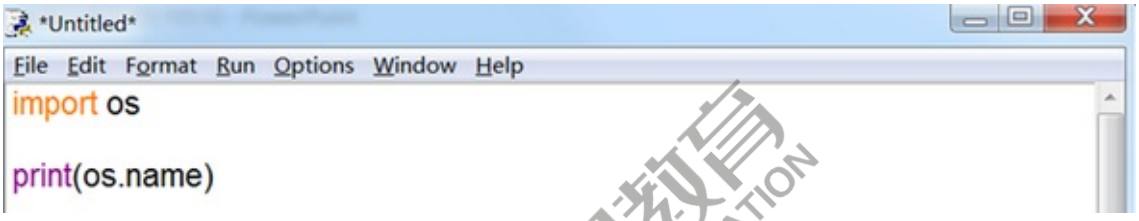
1.5.1 python官方IDLE集成开发环境

这是一个官方提供的交互式集成开发环境，也就是说你无须使用其它编辑器，Python自带！但是，通常我们只用它进行教学、展示、测试和调试代码，不建议用它进行实际的开发工作。

- 可以从开始菜单→所有程序→Python 3.6→IDLE (Python 3.6 64-bit)来启动IDLE。打开之后界面是这样的。

```
Type "copyright", "credits" or "license()" for more information.
>>>
```

- 当然你也可以在这里写代码，file→new file，就会看到如下界面：

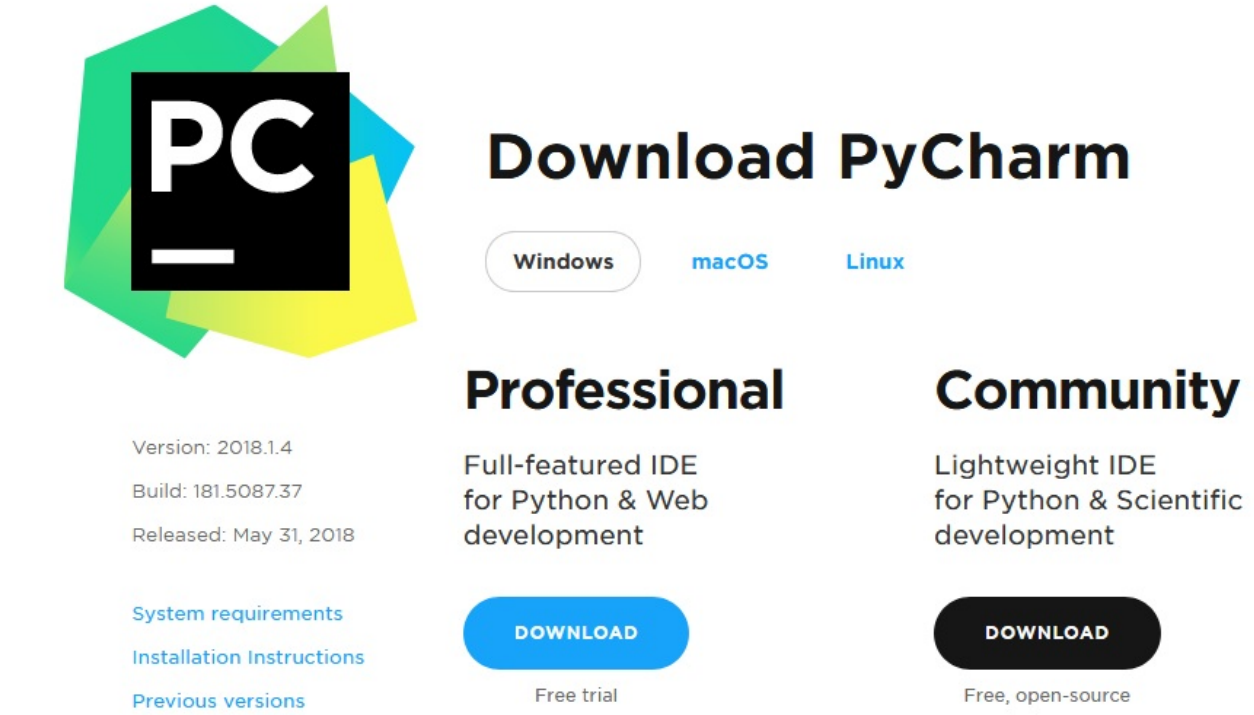


这个IDLE很少用到，这里就不做详细的介绍。

1.5.2 pycharm集成开发环境

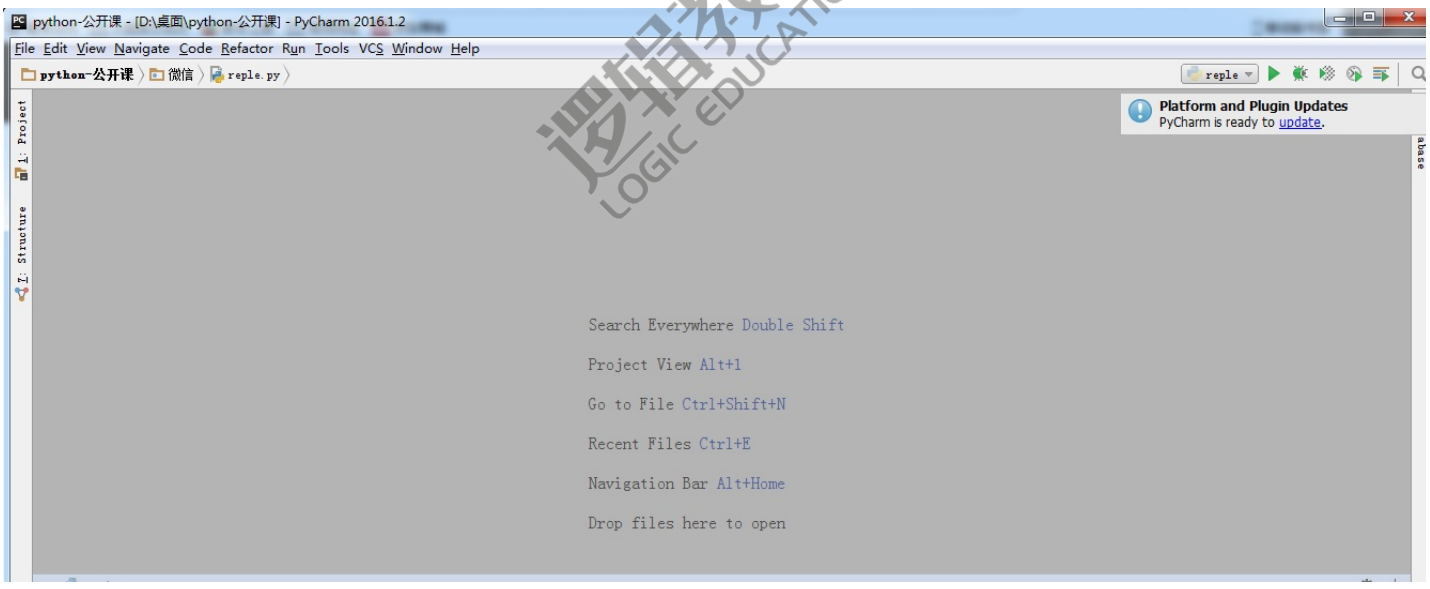
PyCharm是由JetBrains公司打造的一款 Python IDE，支持Windows、Linux、macOS系统。

下载地址：https://www.jetbrains.com/pycharm/download/#section=windows



pycharm分为专业版和社区版。专业版是全功能的python开发IDE包括web开发。免费试用，逾期需购买许可，价格略贵。而社区版则是轻量级的Python开发IDE，但是免费并且开源。如果负担得起，建议大家使用专业版。

pycharm工作界面：



在Pycharm中运行代码有好几种方式：

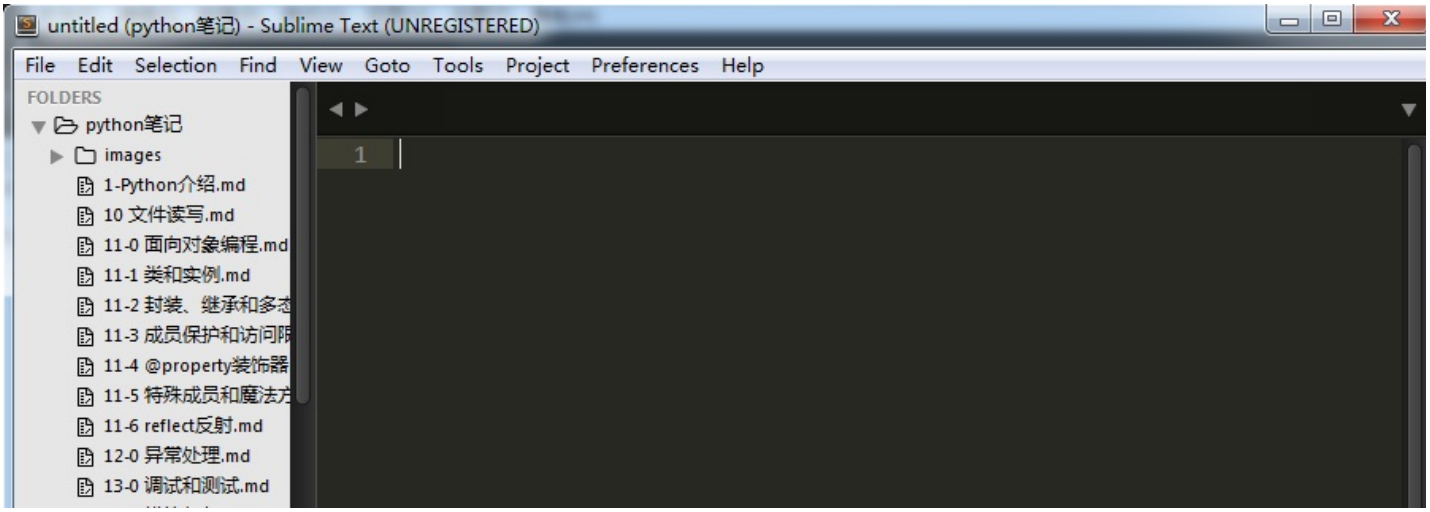
- 可以选中py文件，然后菜单 `Run->run`
- 可以右键py文件编辑窗口内部，然后run
- 可以点击工具栏中的绿色三角符号
- 可以点击下方Run的ToolBar中的绿色三角符号
- 还可以右击py文件的导航标签，然后run

总之，各种花样运行！但是，要小心了，也容易各种花样翻车，容易运行成别的py文件，因此，建议在py文件的

编辑窗口内部右键运行的方式，基本不会出错。

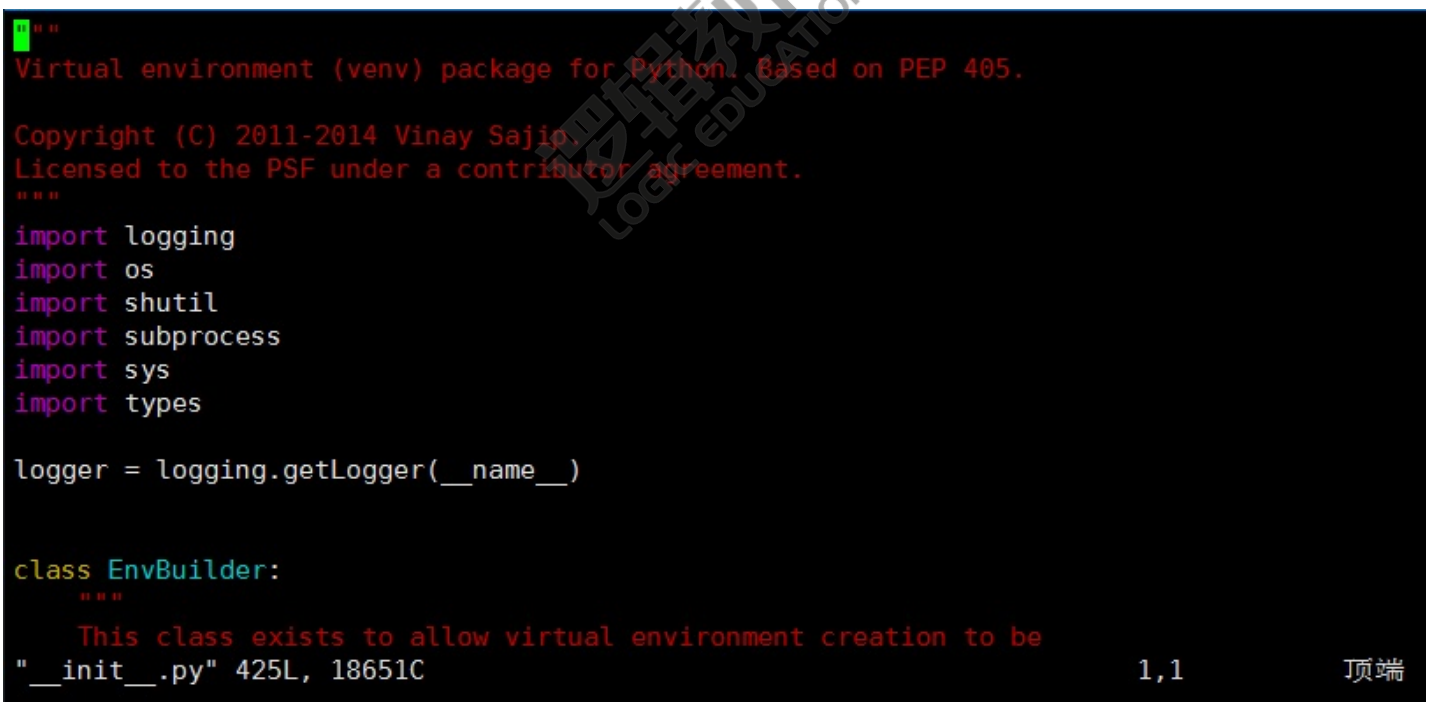
1.5.3 其他编辑器

sublime text界面，这是一个轻量级的编辑器，也非常好用。



Linux下的VIM

Linux下当之无愧的头号编辑器，界面如下：



但是VIM有很多快捷命令行，比如， `ls` `ll` `wq` `yyp` 等等。编辑器的话就给大家介绍到这里，编辑器不分那个好用那个不好用，主要看自己适合那个编辑器。

第2章 Python基础

2.1-基础语法

2.1.1 标识符

所谓的标识符就是对 `变量` 、 `常量` 、 `函数` 、 `类` 等对象起的名字。
首先必须说明的是，Python语言在任何场景都严格区分大小写！也就是说A和a代表的意义完全不同
python对于表示标识符的命名有如下规定：

> 第一个字符必须是字符表中的字符或者下划线

例如： `a` ， `abc` ， `_id` ，等都是可以的。但是例如： `$a` (以$开头的是PHP的变量语法)， `~abc` ， `123a` 都是不可以的。这一点一定要注意。
可能有人会问，中文可以作为标识符吗？答案是可以的

```
我 = "json", 打印出‘我’ 结果是json

虽然可以但是不建议大家这样做。
```

另外，以下划线开头的标识符通常都有特殊意义。以单下划线开头的变量，例如_foo代表禁止外部访问的类成员，需通过类提供的接口进行访问，不能用 `"from xxx import *"` 导入。而以双下划线开头的，例如 `__foo` ，代表类的私有成员 以双下划线开头和结尾的 `__foo__` 是python里特殊方法专用的标识。如 `__init__` 代表类的构造函数。这些我们后面会专门的讨论这里不做过多的解释。

标识符的其他的部分由字母、数字和下划线组成

标识符除了首字符不可以是数字外，其他部分还可以包含数字。这里需要注意特殊字符是不可以的。例如： `a123b` ， `bbc` ， `a_b_c_1` 这些都是可以的。但是 `a&b` ， `a-b-c` 这些都是不可以的。
另外要注意的是，由于l(小写的L）和数字1， 大小写的o与数字0在外观上的相似性，请尽量不要让它们相邻出现，保持语义的清晰性，确保不会发现错误认读的情况。
同样，英文中夹塞中文在语法上也是可以的，但绝对不要这么做！

```
a这都能行b就服你 = 100
a这都能行b就服你
100
```

标识符对大小写敏感

刚刚上面也给大家说过了，标识符ab和AB是完全不同的两个标识符

**变量名全部小写，常量名全部大写

这条不能算语法层面的要求，而是代码规范的要求。

可以用PI来表示一个变量，但通常我们都会认为这是代表圆周率的一个常量

函数和方法名用小写加下划线

这算是一个代码的规范，我们在定义一个方法的时候。尽量用get_images，count_apple之类的命名方式。当然也可以采用小驼峰的方式，getImages，countApple这种。

类名用大驼峰

同样也是代码规范，例如ThreadMinxIn，ButtonClick这种。就是每个单词的首字母大写，组合在一起就像是驼峰一样高低排列。

模块和包的名字用小写

请尽量小写模块和包的名字，并且不要和标准库以及著名的第三方库同名。如果同名程序运行会报错。
最后提醒大家，变量的命名不要用关键字和内置函数的名称！！

2.1.2 python保留字

Python保留字，也叫关键字，是Python语言官方确定的用作语法功能的专用标识符，不能把它们用作任何自定义标识符名称。关键字只包含小写字母。可以通过python提供的库输出这些关键字：

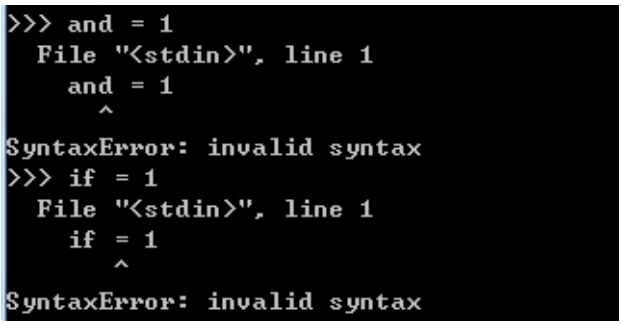
and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Python的标准库提供了一个 `keyword` 模块，可以输出当前版本的所有关键字

```
import keyword
```

```
keyword.kwlist
```

如果真的用关键字来当做变量，会怎么样呢？



系统会直接提示语法错误，所以这里一定要注意不要用关键字来当做变量。除了不能使用关键字作为标识符，内置的函数同样也是不可以的。sum是一个求和的函数。这里我给它定义成一个字符串看看会有什么结果？

```
>>> sum([1,2,3,4])
10
>>> sum = "错误标识符名"
>>> sum([1,2,3,4])
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    sum([1,2,3,4])
TypeError: 'str' object is not callable
```

2.1.3 注释

我们写的程序里，不光有代码，还要有很多注释。注释有说明性质的、帮助性质的，它们在代码执行过程中相当于不存在，透明的。

单行注释

Python中，以符号“#”为单行注释的开始，从它往后到本行的末尾，都是注释内容。

```
# 单行注释
```

多行注释

Python没有真正意义上的多行注释（块注释）语法

```
# 第一行注释
# 第二行注释
# 第三行注释
```

注释文档

在某些特定的位置，用三引号包括起来的部分，也被当做注释。

```
"""
    这个是函数的说明文档。
    :param a: 加数
    :param b: 加数
    :return: 和
```

```
"""
```

2.1.4 代码前两行是什么意思？

很多时候，我们在一些py脚本文件的开头都能看到类似的以#开头的这样两行代码，它们不是注释，是一些设定

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
```

第一行：用于指定运行该脚本的Python解释器，Linux专用，windows不需要。 `env` 方式下，系统会自动使用环境变量里指向的Python。还有一种方式， `#!/usr/bin/python3.6` ，这会强制要求使用系统中的 `python3.6` 解释器执行文件，这种方式不好，一旦你本地的Python3.6版本删除了，会出现找不到解释器的错误。无论两种方式的哪一种，都指的是在linux下使用。

第二行：代码的编码方式。不是程序要处理的数据的编码方式，而是程序自己本身的字符编码。在Python3中，全面支持Unicode，默认以UTF-8编码，我们不用再纠结中文的问题，乱码的问题，所以本行其实可以不需要。但在Python2中，对字符的编码是个非常令人头疼的问题，通常都需要指定这么一行。如果要自定义别的编码类型的话，可以像这样：# -- coding: cp-1252 --，但如果没有强制需求的话，不要自己作死，请坚持使用utf-8编码。

这里的-*-是什么意思呢？没意思，装饰美观好看而已

2.1.5语句与缩进

语句：在代码中，能够完整表达某个意思、操作或者逻辑的最短代码，被称为语句。

```
a = 321
a = 321
print("hello world")
list.append(item)
```

这里强调一下，python的标准语言不需要使用分号。简单的换行就表示语句已经结束。

代码块：为完成某一特定功能而联系在一起的一组语句构成一个代码块。有判断、循环、函数、类等各种代码块。代码块的首行通常以关键字开始，以冒号(：)结束。比如：

```
if expression:
    pass
elif expression :
    pass
else:
    pass
```

Python最具特色的语法就是 `使用缩进来表示代码块` ，不需要使用大括号

像 `PHP` 、 `JAVA` 等语言都是使用 `（{}）` 来表示代码块的。 `python` 一般用 `四个空格` 就是 `tab` 来缩进。在 `pycharm` 中tab自动回转成4个空格。在Linux环境中，如VIM编辑器，建议使用空格。

那么怎么才是正确的缩进方式呢？

1.所有的普通语句，顶左开始编写，不需要缩进

2.所有的语句块，首行不用缩进，从冒号结束后开始下一行，都要缩进

3.直到该语句块结束，就退回缩进，表示当前块已结束

4.语句块可以嵌套，所以缩进也可以嵌套

多行语句： 前面是多条语句在一行，但如果一条语句实在太长，也是可以占用多行的，可以使用反斜杠 `(\)` 来实现多行语句

```
string = "i love this country,"\
         +"because it is very beautiful!"\
```

不到迫不得已，不要使用这种，该换行就换行。

2.1.6 pass语句

pass语句是占位语句，它什么都不做，只是为了保证语法的正确性而写。以下场景中，可以使用pass语句：

- 当你不知道后面的代码怎么写的时候
- 当你不需要写代码细节的时候
- 当语法必须，又没有实际内容可写的时候
- 其它的一些你觉得需要的场景

```
def func(a,b):
    pass
```

2.1.7 字符串的表示形式

在后面的章节中，会进行更深入的介绍，这里作为一个前期的知识铺垫。

`abc` 可能是个变量，但是 `"abc"` 肯定是个字符串了。这里强调一下在编程语言中，所有的符号都是英文状态下的。

在python中单引号和双引号的作用完全相同。在其他语言中，单双引号还是有一定区别的。

原生字符串： 通过在字符串前加 `r` 或 `R` ，如 `r"this is test \n"` ，表示这个字符串里的斜杠不需要

转义，等同于自身。

2.2-变量与常量

变量：在程序运行过程中，值会发生变化的量

常量：在程序运行过程中，值不会发生变化的量

无论是变量还是常量，在创建时都会在内存中开辟一块空间，用于保存它的值。

这里有一点需要注意的是，在python中是不需要声明类型的。这是根据Python的动态语言特性而来。变量可以直接使用，而不需要提前声明类型。

2.2.1 变量的定义

Python 中的变量不需要声明类型

```
a = 4
b = "hello"
c = [1,2]
d = (1,2)
```

- 这些变量都是不需要声明它的类型的，在 `C` 和 `Java` 中是必须要声明的。这里的 `=` 是赋值而不是等于的意思。每个变量在使用前都必须赋值，变量赋值以后才会被创建。如果一个变量没有赋值，直接用的话，系统会报出错误。
- 这里的等号要理解并读作 `“赋值”` ，而不是 `“等于”` ， `“赋值”` 是对变量的操作，而 `“等于”` 是对两个变量进行比较。

每个变量在使用前都必须赋值，变量赋值以后才会被创建

新的变量通过赋值的动作，创建并开辟内存空间，保存值。如果没有赋值而直接使用会抛出赋值前引用的异常或者未命名异常

```
>>> a        # 孤单单一个a，什么也表示不了，只能报错
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a
NameError: name 'a' is not defined
>>> a = 1   # 这样就没问题了，解释器知道a是个新变量了
>>> c.append(1)      # c是个什么鬼？
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    c.append(1)
NameError: name 'c' is not defined
```

Python中，一切事物都是对象，变量引用的是对象或者说是对象在内存中的地址。

（后面我们详细来讲解）

在Python中，变量本身没有数据类型的概念

通常所说的 “变量类型” 是变量所引用的对象的类型，或者说是变量的值的类型

```
>>> a = 1
>>> a = "haha"
>>> a = [1, 2, 3]
>>> a = { "k1":"v1"}
```

例子中，变量a在创建的时候，赋予了值为1的整数类型，然后又被改成字符串“haha”，再又变成一个列表，最后是个字典。变量a在动态的改变，它的值分别是不同的数据类型，这是动态语言的特点。

“=”号这个赋值运算符是从右往左的计算顺序。

```
>>> a = 1
>>> b = 2
>>> c = a + b        # 先计算a+b的值，再赋给c
>>> c
3
```

Python允许同时为多个变量赋值。

例如：a = b = c = 2，a，b，c的值最终都是2

同样也可以为多个变量赋不同的值

a，b，c = 1，2，3 最终a = 1，b = 2 ，c = 3

当我们写：

a = 'ABC' 时，Python解释器干了两件事情：

1.在内存中创建了一个‘ABC’的字符串对象；

2.在内存中创建了一个名为a的变量，并把它指向'ABC'。

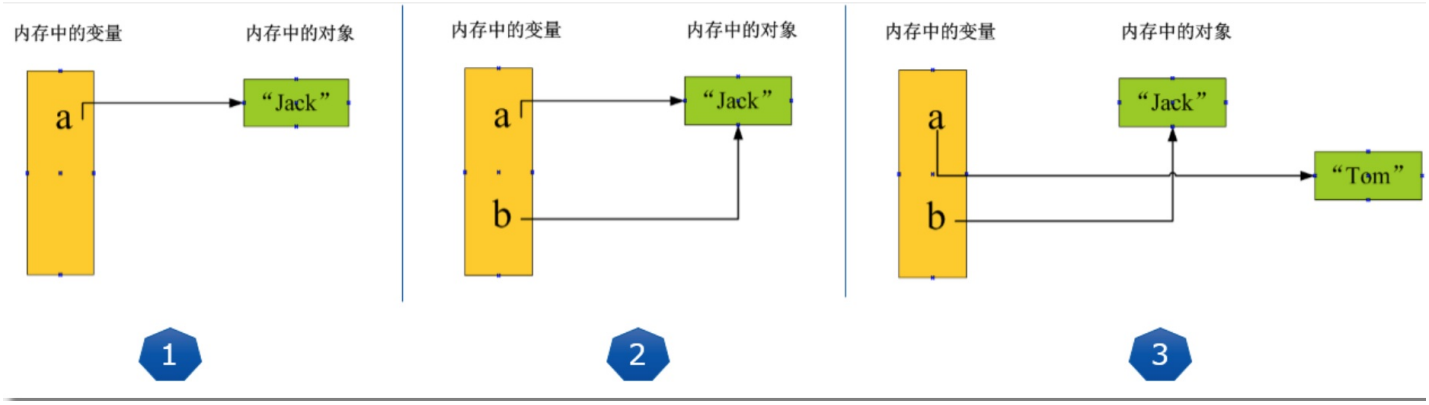
思考

```
a = 'Jack'
b = a
a = 'Tom'
print(b)
print(a)
```

执行 `a = ‘Jack’` ，解释器创建字符串 `‘Jack’` 对象和变量 `a` ，并把 `a` 指向 `‘Jack’` 对象；

执行 `b = a` ，解释器创建变量 `b` ，并且将其指向变量 `a` 指向的字符串 `‘Jack’` 对象；

执行 `a = ‘Tom’` ，解释器创建字符串 `‘Tom’` 对象，并把 `a` 改为指向 `‘Tom’` 对象，与 `b` 无关。



2.2.2 变量的使用

练习：制作名片管理系统

```
input()      字符串的输入
print()      打印
%s           打印字符串
%d           打印整数
```


2.2.3 常量

最后我们还要说下常量!

常量就是不变的变量，比如常用的数学常数圆周率就是一个常量。在Python中，通常用全部大写的变量名表示常量：

```
PI = 3.14159265359
```

但事实上，从Python语法角度看， `PI` 仍然是一个变量，因为Python根本没有任何机制保证 `PI` 不会被改变。你完全可以给 `PI` 赋值为 `10` ，不会弹出任何错误。

所以，用全部大写的变量名表示常量只是一个习惯上的用法。

2.3-输入和输出

2.3.1 input输入函数

input函数：获取用户输入，保存成一个字符串。重要的话，说两遍，input函数的返回值是一个 `字符串类型` 。哪怕你输入的是个数字1，返回给你的只会是字符串“1”，而不是 整数1 。

```
str = input("请输入你的姓名:")
请输入你的姓名:json
'json'
type(str)
<class 'str'>
age = input("请输入你的年龄:")
请输入你的年龄:18
'18'
type(age)
<class 'str'>

a = input("请输入一个字符串:")
如果输入的是一个空白字符串,输入的也是空白字符

a = input("请输入一个字符：")
如果输入的是前后都有空格的字符,输出的也是前后都有空格的字符串
```

从上面两个例子中，大家也可以发现了，我输入的值不管是什么，类型都是字符串。

`type` 是python内置的函数之一，作用是 `查看数据的类型` 。

比如将字符串转换成数字类型

```
age = input("请输入你的年龄：  ")

age = int(age)   # 将字符串转化为整数

if age > 18:
    print("你已经成年!")
else:
    print("还没断奶？")
```

前面我们在将字符串转化为整数用的是 `int()` 函数，这种方式有危险的

```
s = "123"
a = int(s)
a
123
```

```
s = "something"
a = int(s)

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    a = int(s)
ValueError: invalid literal for int() with base 10: 'something'
```

对于形如 “123” ， “283242” 的字符串，转化没问题，但是对于包含字符、特殊字符的字符串就没办法转化了，会弹出异常错误。所以在使用int函数之前，要先对输入进行判断

```
age = input("请输入你的年龄：  ")

if age.isdigit():   # 使用isdigit函数判断输入是否全是数字格式
    age = int(age)   # 将字符串转化为整数
    print("你的年龄是： ", age)
else:
    print("输入不合法！")
```

input函数有时可以巧妙地用于阻塞或暂停程序

```
print("程序前面部分执行完毕......")

input("请按回车继续......")       # 在这里程序会暂停，等待你的回车动作

print("继续执行程序的后面部分......")
```


此时的input函数不会将输入保存下来，只是用作暂停程序动作

2.3.2 print输入函数

print函数我们其实已经不陌生了，前面我们也已经用了很多次了。作用就是打印变量。

```
a = "json"
b = "teach"
print(a,b)
"json teach"    # 自动以空格分隔
print(a+"a"+"b")
"jsonab"        # 无分隔
```

我们看一下print函数的原型： `print(self, *args, sep=' ', end='\n', file=None)`

sep参数: 分隔的符号，默认是一个空格

end参数: 打印后的结束方式，默认为换行符 `\n` 。如果，设置 `end=''` ，则可以不换行，让print在一行内

连续打印。活用print的参数，可以实现灵活的打印控制。

```
a = "i am"
b = "teach"
print(a,"a",b,sep="*")
i am*a*teach
```

2.3.3 print格式化输出

在Python2.6以后，有两种格式化输出的方式。

一种就是类似C语言的printf的 `%` 百分号格式化输出，也是Python最基础最常用的格式化输出方式。另一种就是 `str.format()` 的方式。

这里先介绍一下传统的 `%` 格式化输出方式，例如：

```
print("我叫%s 今年%d岁"%('json',18))

首先构造了一个字符串"我叫%s 今年%d岁"
将其中需要用别的变量替换的部分，用%号加一个数据类型代号
前面有多少个%号，后面就要提供多少个参数
每个参数值之间用逗号隔开
每个参数与前面的%，相对应的。并且数据类型也要能够合法对应。
```

格式化符号:

符　号	描述
%c	格式化字符及其ASCII码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型
%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f和%e的简写
%G	%f 和 %E 的简写
%p	用十六进制数格式化变量的地址

需要特别说明的是，如果你想在print中打印一个%百分符号本身，那么你需要使用%%，两个百分符转义出一个百分符

```
age=18
print('%d%%'%age)
18%
```

2.4-运算符

运算符：以 `1 + 2` 为例， `1` 和 `2` 被称为操作数， `“+”` 称为运算符。

Python语言支持以下类型的运算符:

- 算术运算符
- 比较（关系）运算符
- 赋值运算符
- 逻辑运算符
- 位运算符
- 成员运算符
- 身份运算符
- 三目运算符

2.4.1 算术元运算符

以下假设变量： `a=10` ， `b=20` ：

运算符	描述	实例
+	加 - 两个对象相加	a + b 输出结果 30
-	减 - 得到负数或是一个数减去另一个数	a - b 输出结果 -10
*	乘 - 两个数相乘或是返回一个被重复若干次的字符串	a * b 输出结果 200
/	除 - x除以y	b / a 输出结果 2
%	取模 - 返回除法的余数	b % a 输出结果 0
**	幂 - 返回x的y次幂	a**b 为10的20次方， 输出结果 100000000000000000000
//	取整除 - 返回商的整数部分	9//2 输出结果 4 , 9.0//2.0 输出结果 4.0

python中，有三种除法，分别是

```
10/3    3.333333333
```

① 计算结果是浮点数，即使两个整数恰好整数，结果也是浮点数

```
9/3     3.0
```

② 只取整数部分，余数被抛弃

```
10//3   3
```

③ 余数运算，可以得到两个整数相除的余数

```
10%3     1
```

如果想同时得到商和余数，可以用这个方法

```
divmod(10,3)    (3,1)
```

因为浮点数精度的问题，Python还存在一些计算方面的小问题

```
0.1+0.1+0.1-0.3
5.551115123125783e-17
```

要解决这个问题，可以导入 `decimal` 模块

```
from decimal import Decimal
Decimal('0.1')+Decimal('0.1')+Decimal('0.1')-Decimal('0.3')
Decimal('0.0')
```


2.4.2 比较运算符

运算符	描述	实例
==	等于 - 比较对象是否相等	(a == b) 返回 False。
!=	不等于 - 比较两个对象是否不相等	(a != b) 返回 true.
<>	不等于 - 比较两个对象是否不相等	(a <> b) 返回 true。这个运算符类似 != 。
>	大于 - 返回x是否大于y	(a > b) 返回 False。
<	小于 - 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。	(a < b) 返回 true。
>=	大于等于 - 返回x是否大于等于y。	(a >= b) 返回 False。
<=	小于等于 - 返回x是否小于等于y。	(a <= b) 返回 true。

① 字符串大小比较

```
'abc' < 'xyz'
# 为什么字符串可以进行大小比较,这是因为电脑存储字符时，是以ASCII码值存储的
也就是A是65，z是90.你输入的字符也是一个对应的数字

(3,2) < ('a','b')
```

② 请分别说出下面各项的值：

```
True == 1
False == 0

3>2>1

3>2>2

(3>2)>1

(3>2)>2
```

对于连续比较，Python是按这种机制解释的：

`3>2>1 等于 (3>2) and (2>1)` ，相当于两个比较，然后用与&再结合起来。

2.4.3 赋值运算符

运算符	描述	实例
=	简单的赋值运算符	c = a + b 将 a + b 的运算结果赋值为 c
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a 等效于 c = c - a
*=	乘法赋值运算符	c *= a 等效于 c = c * a
/=	除法赋值运算符	c /= a 等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a 等效于 c = c ** a
//=	取整除赋值运算符	c //= a 等效于 c = c // a

注意对于 `a /= b` 之类的操作，等同于 `a = a / b` ，而不是 `= b /` ，方向一定不要搞反了。

```
a = 1
a++
print(a)
```

在python中是没有 `++` 这种写法的。

2.4.4 位运算符

按位运算符是把数字看作二进制来进行计算的。Python中的按位运算法则如下：

下表中变量 `a 为 60` ， `b 为 13` ，二进制格式如下：

```
a = 0011 1100
b = 0000 1101
-----------------
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a  = 1100 0011
```

运算符	描述	实例
&	按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0	(a & b) 输出结果 12 ，二进制解释： 0000 1100
|	按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。	(a | b) 输出结果 61 ，二进制解释： 0011 1101
^	按位异或运算符：当两对应的二进位相异时，结果为1	(a ^ b) 输出结果 49 ，二进制解释： 0011 0001
~	按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1。 `~x` 类似于 `-x-1`	(~a) 输出结果 -61 ，二进制解释： 1100 0011，在一个有符号二进制数的补码形式。
<<	左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数	a >> 2 输出结果 15 ，二进制解释： 0000 1111

2.4.5 逻辑运算符

Python语言支持逻辑运算符，但是没有其它语言中的 `&&` 和 `||` 语法，取而代之的是更加人性化的英文单词 `and or not` （全部都是小写字母）。

以下假设变量 `a 为 10` ， `b为 20` ：

运算符	逻辑表达式	描述	实例
and	x and y	布尔"与" - 如果 x 为 False，x and y 返回 False，否则它返回 y 的计算值。	(a and b) 返回 20。
or	x or y	布尔"或" - 如果 x 是非 0，它返回 x 的值，否则它返回 y 的计算值。	(a or b) 返回 10。
not	not x	布尔"非" - 如果 x 为 True，返回 False 。如果 x 为 False，它返回 True。	not(a and b) 返回 False

```
x = False
y = True

x and y
False

a = 10
b = 20
a and b
20
```

2.4.6 成员运算符

`与 not` 是Python独有的运算符（全部都是小写字母），用于判断对象是否某个集合的元素之一，非常好用，并且运行速度很快。返回的结果是布尔值类型的True或者False。

运算符	描述	实例
in	如果在指定的序列中找到值返回 True，否则返回 False。	x 在 y 序列中，如果 x 在 y 序列中返回 True。
not in	如果在指定的序列中没有找到值返回 True，否则返回 False。	x 不在 y 序列中，如果 x 不在 y 序列中返回 True。


```
list1 = [1, 2, 3, 4, 5]
a = 1

for i in list1:
    if i == a:
        print("a在list1中")
    else:
        print('a不在list1中')

flag = False
for i in list1:
    if i == a:
        flag = True
        break
if flag:
    print("a是list1的元素之一")
else:
    print("a不是list1的元素")
```

```
如果用in运算符来做的话，就不用这么麻烦了
list1 = [1, 2, 3, 4, 5]
a = 1
if a in list1:
    print("a是list1的元素之一")
else:
    print("a不是list1的元素")
```

2.4.7 身份运算符

这也是Python的特色语法（全部都是小写字母）。

运算符	描述	实例
is	is 是判断两个标识符是不是引用自一个对象	x is y, 类似 id(x) == id(y) , 如果引用的是同一个对象则返回 True，否则返回 False
is not	is not 是判断两个标识符是不是引用自不同对象	x is not y ， 类似 id(a) != id(b)。如果引用的不是同一个对象则返回结果 True，否则返回 False。

注意 `is` 与比较运算符“==”的区别，两者有根本上的区别，切记不可混用：

is用于判断两个变量的引用是否为同一个对象，而==用于判断变量引用的对象的值是否相等！


```
a = [1,2]
b = a
b is a
Ture
b == a
Ture
b = a[:]
b is a
False
b == a
Ture

a = 2
b = 2.0
b is a
a == b
```

2.4.8 三目运算符

python中的三目运算符不像其他语言一般的表示方法：判定条件 `?` 为真时的结果， `:` 为假时的结果。

在python中的格式为：判定条件 `if` 为真时的结果，判定条件 `else` 为假时的结果，例如：

```
a = 1
b = 2
h = ""
h = a-b if a>b else a+b
```

如果 `a > b` 执行 `a - b` ，如果 `a < b` 执行 `a + b`

2.4.9 运算符优先级

下表列出了从最高到最低优先级的所有运算符。优先级高的运算符优先计算或处理，同级别的按从左往右的顺序计算（赋值运算符除外，它是按从右往左的顺序）

运算符	描述
**	指数 (最高优先级)
~ + -	按位翻转, 一元加号和减号 (最后两个的方法名为 +@ 和 -@)
* / % //	乘，除，取模和取整除
+ -	加法减法
>> <<	右移，左移运算符
&	位 'AND'
^ |	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
is is not	身份运算符
in not in	成员运算符
not or and	逻辑运算符

第3章 Python数据类型

3.1-数据类型

在python这门语言中，数据类型分为两种。 内置的和自定义的。

内置的包括 `数字` 、 `字符串` 、 `布尔` 、 `列表` 、 `元组` 、 `字典` 、 `Bytes` 、 `集合` 这些常用的以及一些不太常用的数据类型。而自定义的，一般以类的形式，根据需要组合以上内置类型成为独特的数据类型。

数据类型是Python语言非常重要的部分（哪部分不重要？），尤其是不同数据类型所支持的原生操作，更是重中之重，需要熟练的背在脑海里。很多时候，写大型项目时，不需要你多复杂的技巧，只需要用这些数据操作方法就可以。

- 原因之一，更好的分配管理内存,节省不必要的开支。如果没有数据类型的区别，那么所有的对象都必须按体积最大的对象所必须大小的房子分配空间，也就是内存空间，这样的浪费太严重了。有了数据类型，计算机就可以根据类型预定义的空间需求分配大小，合理开支。内存节省精简了，还能提高读取速度和运行效率。
- 原因之二，方便统一管理，提供同样的API。这样，我们可以为同一数据类型，提供同样的操作，限制其它不允许的行为。也易于查找错误，定位问题。
- 原因之三，区分数据类型，更贴切人类对自然事物的分类管理习惯。我们人类对事物都进行了各种分类，植物是植物、动物是动物，书是书，笔是笔。分类了之后，我们很自然的知道书可以读，笔可以写。数据类型也一样，让我们对抽象的数据有了可分辨的行为和自然的记忆。

3.2-数字类型

数字类型是不可变类型。所谓的不可变类型，指的是类型的值一旦有不同了，那么它就是一个全新的对象。数字1和2分别代表两个不同的对象，对变量重新赋值一个数字类型，会新建一个数字对象。

还是要强调一下Python的变量和数据类型的关系，变量只是对某个对象的引用或者说代号、名字、调用等等，变量本身没有数据类型的概念 。只有 `1` ， `[1, 2]` ， `"hello"` 这一类对象才具有数据类型的概念。

Python 支持三种不同的数字类型，整数、浮点数和复数。

3.2.1 整数

通常被称为整型，数值为正或者负，不带小数点。python 3的整型可以当做Long类型使用，所以python 3 没有python 2的Long类型。

表示数字的时候，有时我们还会用八进制或十六进制来表示。

十六进制用 `0x` 前缀和 `0-9` ， `a-f` 表示，例如： `0xff00`

八进制用 `0o` 前缀和 `0-7` 表示，例如 `0o45`

python的整数长度为16、32位，并且通常是连续分配内存空间的。

```
id(-2)
505205760
id(-1)
505205776
```


从上面的空间地址看，地址之间正好差16。

小整数对象池

python初始化的时候会自动建立一个小整数对象池，方便我们调用，避免后期重复生成！这是一个包含 `262` 个指向整数对象的指针数组，范围是 `-5` 到 `256` 。

也就是说比如整数10，即使我们在程序里没有创建它，其实在Python后台已经悄悄为我们创建了。

为什么要这样呢？我们都知道，在程序运行时，包括Python后台自己的运行环境中，会频繁使用这一范围内的整数，如果每需要一个，你就创建一个，那么无疑会增加很多开销。创建一个一直存在，永不销毁，随用随拿的小整数对象池，无疑是个比较实惠的做法。

```
id(-6)
10114720
id(-5)
496751568
id(255)
496755728
```

从 `id(-6)` 和 `id(257)` 的地址，我们能看出小整数对象池的范围，正好是 `-5` 到 `256` 。

除了 `小整数对象池` ，Python还有 `整数缓冲区` 的概念，也就是刚被删除的整数，不会被真正立刻删除回收，而是在后台缓冲一段时间，等待下一次的可能调用。

```
a = 100
id(a)
503175776
del a # 删除变量a
b = 100
id(b)
503175776

a = 10;print(id(a));del a;b = 10;print(id(b))  # 在Python交互环境中不能实现。
```

上面，我给变量a赋值了整数100，看了一下它的内存地址。然后我把a删了，又创建个新变量b，依然赋值为100，再次看下b的内存地址，和以前a存在的是一样的。

3.2.2 浮点数

`浮点数` 也就是小数，如 `1.23` ， `3.14` ， `-9.01` 等等。但是对于很大或很小的浮点数，一般用科学计数法表示，把10用e替代， `1.23x10^9` 就是 `1.23e9` ，或者 `12.3e8` ， `0.000012` 可以写成 `1.2e-5` 等等。

3.2.3 复数（complex）

`复数` 由实数部分和虚数部分构成，可以用a + bj,或者complex(a,b)表示，复数的实部a和虚部b都是浮点。

数字类型转换

在某些特定的情况下，我们需要对数字的类型进行转换。

python为我们提供了内置的数据类型转换函数。

```
int(x)       将x转换为一个整数。如果x是一个浮点数，则截取小数部分
float(x)     将x转换成一个浮点数
complex(x)   将x转换到一个复数，实数部分为 x，虚数部分为 0。
complex(x, y)： 将 x 和 y 转换到一个复数，实数部分为 x，虚数部分为 y。
```

转换过程中如果出现无法转换的对象，则会抛出异常，比如 `int("haha")`

数学计算

对于数学计算，除了前面提到过的简单的加减乘除等等，更多的科学计算需要导入 `math` 这个库，它包含了绝大多数我们可能需要的科学计算函数，如下表

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如abs(-10) 返回 10
ceil(x)	返回数字的上入整数，如math.ceil(4.1) 返回 5
exp(x)	返回e的x次幂(ex),如math.exp(1) 返回2.718281828459045
fabs(x)	返回数字的绝对值，如math.fabs(-10) 返回10.0
floor(x)	返回数字的下舍整数，如math.floor(4.9)返回 4
log(x)	如math.log(math.e)返回1.0,math.log(100,10)返回2.0
log10(x)	返回以10为基数的x的对数，如math.log10(100)返回 2.0
max(x1, x2,...)	返回给定参数的最大值，参数可以为序列。
min(x1, x2,...)	返回给定参数的最小值，参数可以为序列。
modf(x)	返回x的整数部分与小数部分，两部分的数值符号与x相同，整数部分以浮点型表示。
pow(x, y)	x**y 运算后的值。
round(x [,n])	返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的位数。
sqrt(x)	返回数字x的平方根
acos(x)	返回x的反余弦弧度值。
asin(x)	返回x的反正弦弧度值。
atan(x)	返回x的反正切弧度值。
atan2(y, x)	返回给定的 X 及 Y 坐标值的反正切值。
cos(x)	返回x的弧度的余弦值。
hypot(x, y)	返回欧几里德范数 sqrt(xx + yy)
sin(x)	返回的x弧度的正弦值。
tan(x)	返回x弧度的正切值。
degrees(x)	将弧度转换为角度,如degrees(math.pi/2) , 返回90.0
radians(x)	将角度转换为弧度

3.3-布尔类型

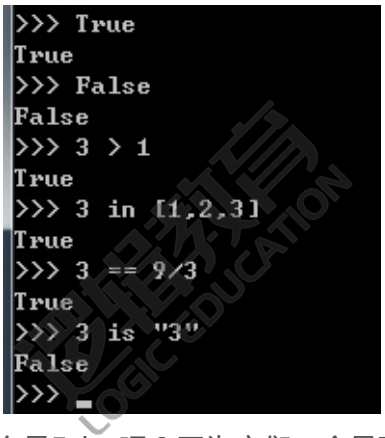
`对` 与 `错` 、 `0` 和 `1` 、 `正` 与 `反` ，都是传统意义上的布尔类型。

但在Python语言中，布尔类型只有两个值， **`True`** 与 **`False`** 。请注意，是英文单词的对与错，并且首字母要大写，不能其它花式变型。

布尔值通常用来判断条件是否成立。 例如：

```
a = 1
if a > 3:
    print("a是一个大于3的数字")
else:
    print("a不是一个大于3的数字")
```

Python内置的 `bool()` 函数可以用来测试一个表达式的布尔值结果。



这里需要注意的是， `3` is `"3"` 为什么是False呢？因为它们一个是整数一个是字符串。

is是运算符，比较的是对象，当然是错误的。

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool([])
False
>>> bool(())
False
>>> bool({})
False
>>> bool(-1)
True
>>> bool('')
False
>>> bool("False")
True
>>> bool("True")
```

```
True
>>> bool(0.0)
False
>>> bool(1.0)
True
>>> bool(-0.0)
False
```

0、0.0、-0.0、空字符串、空列表、空元组、空字典，这些都被判定为False。

而-1、"False"也被判断为True。

3.3.1 and 、 or 和 not 运算

and 运算是 与 运算，只有所有都为 True ， and 运算的结果才是 True ：

```
>>> True and True
True
>>> True and False
False
>>> False and False
False
>>> 5 > 3 and 3 > 1
True
```


or 运算是 或 运算，只要其中有一个为 True ， or 运算结果就是 True ：

```
>>> True or True
True
>>> True or False
True
>>> False or False
False
>>> 5 > 3 or 1 > 3
True
```

not 运算是 非 运算，它是单目运算符，把 True 变成 False ， False 变成 True ：

```
>>> not True
False
>>> not False
True
>>> not 1 > 2
True
```

布尔类型还能做别的运算吗？

```
>>> True > False
True
>>> True < False
False
>>> True >=False
True
>>> True -1
0
>>> True + 1
2
>>> True *3
3
>>> False -1
-1
```

由以上案例可以看出，在做四则运算的时候，明显把 `True` 看做 `1` ， `False` 看做 `0` 。

3.3.2 空值

空值不是布尔类型，严格的来说放在这里是不合适的，只不过和布尔关系比较紧密。

空值是Python里一个特殊的值，用 `None` 表示（首字母大写）。None不能理解为0，因为0是整数类型，而None是一个特殊的值。None也不是布尔类型，而是NoneType。


```
>>> bool(None)
False
>>> type(None)
<class 'NoneType'>
```

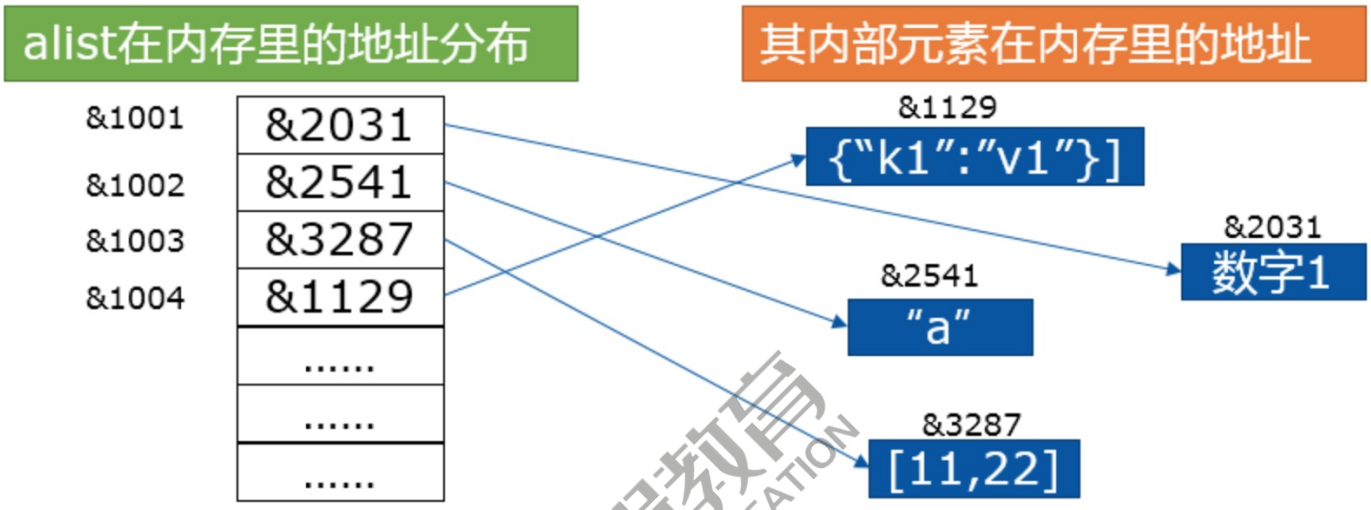
3.4-列表

`列表` 是Python中最基本也是最常用的数据结构之一。列表中的每个元素都被分配一个数字作为 `索引` ，用来表示该元素在列表内所排在的位置。第一个元素的索引是0，第二个索引是1，依此类推。

Python的列表是一个 `有序可重复的元素集合` ，可 `嵌套` 、 `迭代` 、 `修改` 、 `分片` 、 `追加` 、 `删除` ， `成员判断` 。

从数据结构角度看，Python的列表是一个可变长度的 `顺序存储结构` ，每一个位置存放的都是对象的指针。

对于这个列表 `alist = [1, “a”, [11,22], {“k1”:”v1”}]` ，其在内存内的存储方式是这样的：



3.4.1 创建方式

创建一个列表，只要把逗号分隔的不同的数据项使用 `方括号` 括起来即可

```
['1',2,'hello']  # 列表
('1',2,'hello')  # 元组
{'1',2,'hello'}  # 集合
{'name':'juran','age':18}

for i in ['1',2,'hello']:
    print(i)

# 通过查看源码来看创建的方法
l = list(['1',2,'hello'])
print(l)

l = [1, 2, 3]
l1 = [1, 'hello', 1.2]
print(l1)
----------------------------------------------
list = []       # 定义空列表
list1 = ['physics', 1997, 2000]
list2 = [1, 2, 3]
```

```
list3 = ["a", "b", "c"]
```

3.4.2 访问列表内的元素

列表从0开始为它的每一个元素顺序创建下标索引，直到总长度减一 。要访问它的某个元素，以方括号加下标值的方式即可。

注意要确保索引不越界，一旦访问的索引超过范围，会抛出异常。

所以，一定要记得最后一个元素的索引是 `len(list)-1` 。

```
>>> lis = ["a", "b", "c"]
>>> lis[0]
'a'
>>> lis[1]
'b'
>>> lis[2]
'c'
>>> lis[3]

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    lis[3]
IndexError: list index out of range
```


3.4.3 修改元素的值

直接对元素进行重新赋值。

```
>>> lis[0]
'a'
>>> lis[0] = "d"
>>> lis[0]
'd'
```

3.4.4 删除元素

使用 `del` 语句或者 `remove()` ， `pop()` 方法删除指定的元素。

```
>>> lis = ["a", "b", "c"]
>>> del lis[0]      # 根据索引删除
>>> lis
['b', 'c']
>>> lis.remove("b") # 直接根据值进行删除
>>> lis
```

```
['c']
>>> lis.pop()        # 弹出最后一个
'c'
>>> lis
[]

l1 = [1, 'hello', 1]

l2 = l1.pop(1)  # 根据索引来弹出 1为索引
print(l2)
```

3.4.5 列表的特殊操作

```
语句                                    结果                                    描述
[1, 2, 3] + [4, 5, 6]                   [1, 2, 3, 4, 5, 6]                      组合两个列表
l1 = [1, 2, 3]
l2 = [4, 5, 6]
print(l1.__add__(l2))  # 底层调用了__add__方法

['Hi!'] * 4                             ['Hi!', 'Hi!', 'Hi!', 'Hi!']            列表的乘法
l1 = [1, 2, 3]
print(l1.__mul__(3))

3 in [1, 2, 3]                          True                                    判断元素是否存在于
列表中
l1 = [1, 2, 3]
print(l1.__contains__(1))

for x in [1, 2, 3]: print x,    1 2 3                                           迭代列表中的每个元
素
```


3.4.6 列表的常用函数

```
l1 = [1, 2, 3]
函数            作用
len(list)   返回列表元素个数，也就是获取列表长度 l1.__len__()
max(list)   返回列表元素最大值  max(l1)  max(1,2,3,4)
min(list)   返回列表元素最小值
list(seq)   将序列转换为列表

>>> s = list((1, "a", "b", 2))
>>> s
[1, 'a', 'b', 2]
>>> max(s)          # 不能混合不同类型进行最大最小求值
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    max(s)
TypeError: '>' not supported between instances of 'str' and 'int'
```

3.4.7 排序和反转

```
list.reverse()  将列表反转 修改本身 *IN PLACE*
list.sort()     排序，默认升序
如果列表内的元素全为数字，或者字母排序没有问题。
如果字母和数字都有的情况下，排序会报错
list.sort(reverse=True)     降序排列
```

3.4.8 切片(截取)

切片指的是对序列进行截取，选取序列中的某一段。

切片的语法是： `list[start:end]`

以 `冒号` 分割索引， `start` 代表 `起点索引` ， `end` 代表 `结束点索引` 。

省略 `start` 表示以 `0` 开始，省略 `end` 表示到列表的结尾。注意，区间是 `左闭右开` 的！

如果提供的是负整数下标，则从列表的最后开始往头部查找。例如-1表示最后一个元素，-3表示倒数第三个元素。

切片过程中还可以设置 `步长` ，以第2个冒号分割，例如 `list[3:9:2]` ，表示每隔 `2` 距离取一个元素。


```
l3 = ['a','b','c','d','e','f']
print(l3[1])
print(l3[2])
print(l3[1:3])
print(l3[:3])
print(l3[2:])
print(l3[:])        # 复制列表
print(l3[::2])      # 隔一个一取


a[-1]       8
a[-5:]      [4,5,6,7,8]
a[::-1]     想当于逆序输出 [8, 7, 6, 5, 4, 3, 2, 1]
a[1:5:2]    [2,4]
```

3.4.9 多维列表（嵌套列表）

```
a = [[1,2,3],[4,5,6],[7,8,9]]
a[0][1]
2
a = [[1,2,3],[4,5,6],{"k1":"v1"}]
a[2]["k1"]
v1
```

3.4.10 列表的遍历

列表有多种遍历方式：

```
a = [1,2,3,4,5,6]
for i in a:                 # 遍历每一个元素本身
    print(i)
for i in range(len(a)):      # 遍历列表的下标，通过下标取值
    print(i, a[i])

x = 9
if x in a:       # 进行是否属于列表成员的判断。该运算速度非常快。
    print("True")
else:
    print("False")
```

3.4.11 列表的内置方法

```
方法                  作用
append(obj)           在列表末尾添加新的对象
count(obj)            统计某个元素在列表中出现的次数
extend(seq)           在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）
index(obj)            从列表中找出某个值第一个匹配项的索引位置
insert(index, obj)    将对象插入列表
pop(obj=list[-1])     移除列表中的一个元素（默认最后一个元素），并且返回该元素的值
remove(obj)           移除列表中某个值的第一个匹配项
reverse()             反向列表中元素
sort([func])          对原列表进行排序
copy()                复制列表
clear()               清空列表，等于del lis[:]
```


注意：其中的类似 `append` ， `insert` ， `remove` 等方法会修改列表本身，并且没有返回值（严格的说是返回None）。

3.5-元组

我们知道，用方括号括起来的是列表，那么用圆括号括起来的是什么呢？是 `元组` 。
元组也是序列结构，但是是一种 `不可变序列` ，你可以简单的理解为内容不可变的列表。.
除了在内部元素不可修改的区别外，元组和列表的用法差不多。

元组与列表相同的操作：

- 使用方括号加下标访问元素
- 切片（形成新元组对象）
- count()/index()
- len()/max()/min()/tuple()

元组中不允许的操作，确切的说是元组没有的功能

- 修改、新增元素
- 删除某个元素（但可以删除整个元组）
- 所有会对元组内部元素发生修改动作的方法。例如，元组没有remove，append，pop等方法

看一些实例：

```
tup1 = ()           # 创建空元组
tup1 = (40, )       # 创建只包含一个元素的元组时，要在元素的后面跟个逗号
tup = (1, 2, 3, 4)
tup[2]
3
tup[3] = "a"
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    tup[3] = "a"
TypeError: 'tuple' object does not support item assignment
```

元组只保证它的一级子元素不可变，对于嵌套的元素内部，不保证不可变！

```
tup = ('a', 'b', ['A', 'B'])
tup[2][0] = 'a'
tup[2][1] = 'b'
tup
('a', 'b', ['a', 'b'])
```

列表和元组的转换

- 使用list函数可以把元组转换成列表
- 使用tuple函数可以把列表转换成元组

3.6-字典

Python的字典数据类型是基于 `hash散列算法` 实现的，采用 `键值对(key:value)` 的形式，根据 `key` 的值计算 `value` 的地址，具有非常快的查取和插入速度。但它是无序的，包含的元素个数不限，值的类型也可以是其它任何数据类型！

字典的key必须是不可变的对象，例如 `整数` 、 `字符串` 、 `bytes` 和 `元组` ，但使用最多的还是字符串。列表、字典、集合等就不可以作为key。同时，同一个字典内的key必须是唯一的，但值则不必。

字典可精确描述为不定长、可变、无序、散列的集合类型

字典的每个 `键值对` 用 `冒号(:)` 分割，每个 `对` 之间用 `逗号(,)` 分割，整个 `字典` 包括在 `花括号({})` 中 ，例如：

```
d = {key1 : value1, key2 : value2 }
```

3.6.1 创建字典

`dict()` 函数是Python内置的创建字典的方法。

```
test = {}       # 创建空字典
test = {"a":'123','b':'2',"c":3}
dict([('name', 'juran'), ('age', 18), ('addr', 'cs')])
{'name': 'juran', 'age': 18, 'addr': 'cs'}

dict(a=1, b=2, jack=4098)
{'a':1,'b':2}
```


3.6.2 访问字典

字典是集合类型，不是序列类型，因此没有索引下标的概念，更没有切片的说法。但是，与list类似，字典采用把相应的键放入方括号内获取对应值的方式取值。

```
dic = {'Name': 'json','Age': 18}
print ("dic['Name']: ", dic['Name'])
print ("dic['Age']: ", dic['Age'])

# 如果访问字典里没有的键，会抛出异常：
dic['address']

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    dic["address"]
KeyError: 'address'
```

3.6.3 增加和修改

增加就是往字典插入新的键值对，修改就是给原有的键赋予新的值。由于一个key只能对应一个值，所以，多次对一个key赋值，后面的值会把前面的值冲掉。

```
dic = {'Name': 'json','Age': 18}
dic["address"] = "beijing"
dic["address"] = "Shanghai"
dic["Age"] = 20
dic
{'Name': 'json', 'Age': 20, 'address': 'Shanghai'}
-------------------------------------
# 要统计字典内键的个数，可以使用Python内置的len()函数：
len(dic)
3
```

3.6.4 删除字典元素、清空字典和删除字典

使用 `del` 关键字删除字典元素或者字典本身，使用字典的 `clear()` 方法清空字典

```
dic
{'Name': 'json', 'Age': 20, 'address': 'Shanghai'}
del dic['Name']     # 删除指定的键
dic
{'Age': 20, 'address': 'Shanghai'}
-------------------------------------
a = dic.pop('Age')        # 弹出并返回指定的键。必须提供参数！

20
dic.clear()        # 清空字典
del dic            # 删除字典本身
```

3.6.5 字典的重要方法

```
get(key)     返回指定键的值，如果值不在字典中，则返回default值
items()      以列表返回可遍历的(键, 值) 元组对
keys()   以列表返回字典所有的键
values()     以列表返回字典所有的值
```

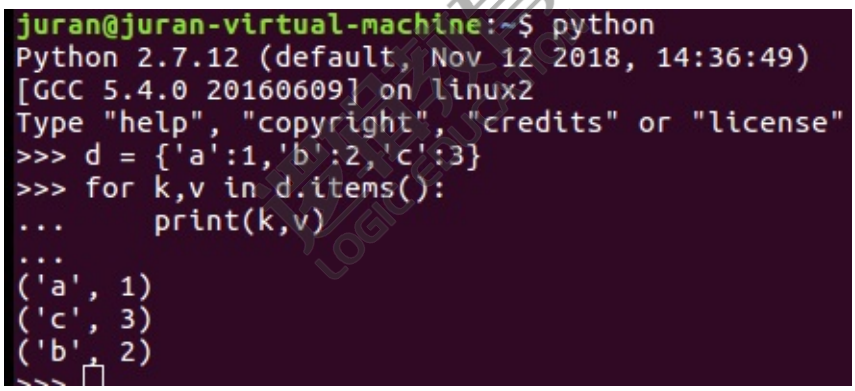
接下来我们看看例子：

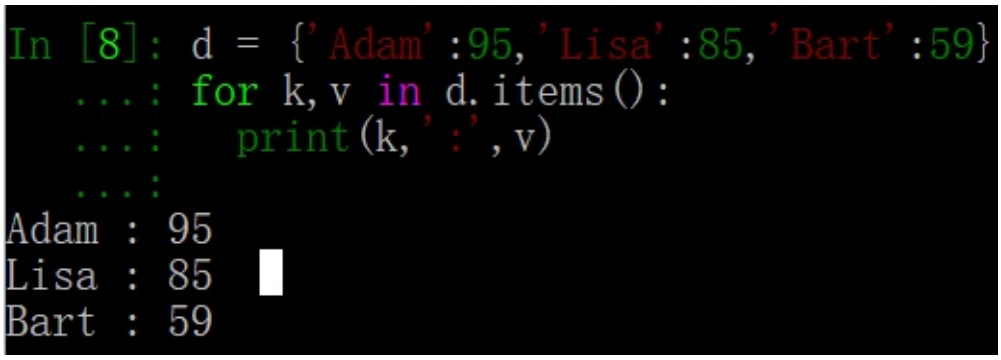
```
dic = {'Name': 'json', 'Age': 20, 'address': 'Shanghai'}
dic.get("sex")           # 访问不存在的key，没有报错
dic['sex']               # 访问不存在的key，报错
```

```
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    dic["sex"]
KeyError: 'sex'
-------------------------------------
dic.items()
dict_items([('Name', 'json'), ('Age', 20), ('address', 'Shanghai')])
-------------------------------------
dic.values()
dict_values(['json', 20, 'Shanghai'])
-------------------------------------
dic.keys()
dict_keys(['Name', 'Age', 'address'])
-------------------------------------
dic.pop('Name')
json
```

3.6.6 遍历字典

从Python3.6开始遍历字典获得的键值对是有序的






```
dic = {'Name': 'Jack', 'Age': 7, 'Class': 'First'}

# 1  直接遍历字典获取键，根据键取值
for key in dic:
    print(key, dic[key])

# 2  利用items方法获取键值，速度很慢，少用！
```

```python
for key,value in dic.items():
    print(key,value)

# 3  利用keys方法获取键
for key in dic.keys():
    print(key, dic[key])

# 4  利用values方法获取值，但无法获取对应的键。
for value in dic.values():
    print(value)
```

3.7-bytes

在Python3以后，字符串和bytes类型彻底分开了。字符串是以字符为单位进行处理的，bytes类型是以字节为单位处理的。

bytes数据类型在所有的操作和使用甚至内置方法上和字符串数据类型基本一样，也是不可变的序列对象。

Python3中，bytes通常用于网络数据传输、二进制图片和文件的保存等等。

可以通过调用 `bytes()` 生成bytes实例，其值形式为 `b'xxxxx'` ，对于同一个字符串如果采用不同的编码方式生成bytes对象，就会形成不同的值。

```
b = b''                                 # 创建一个空的bytes
b = byte()                              # 创建一个空的bytes
b = b'hello'                            # 直接指定这个hello是bytes类型
b = bytes('string',encoding='编码类型')    # 利用内置bytes方法，将字符串转换为指定编码的bytes
b = str.encode('编码类型')                 # 利用字符串的encode方法编码成bytes，默认为utf-8类型
bytes.decode('编码类型')                  # 将bytes对象解码成字符串，默认使用utf-8进行解码。

# 当然也有简单的使用方法
string = b'xxxxxx'.decode()             # 直接以默认的utf-8编码解码bytes成string
b = string.encode()                     # 直接以默认的utf-8编码string为bytes
```

```
In [38]: s = 'abc'

In [39]: s.encode('utf8')
Out[39]: b'abc'

In [40]: s1 = b'abc'

In [41]: s1.decode()
Out[41]: 'abc'
```

3.8-集合

`set集合` 是一个 `无序不重复` 元素的集，基本功能包括关系测试和消除重复元素。

集合使用大括号 `({})` 框定元素，并以 `逗号` 进行分隔。

但是注意：如果要创建一个空集合，必须用 `set()` 而不是 `{}` ，因为后者创建的是一个 `空字典` 。

集合数据类型的核心在于 `自动去重` 。

```
s = set([1,1,2,3,3,4])
s
{1, 2, 3, 4}                    # 自动去重
-------------------------------------
>>> set("this is test")      # 对于字符串，集合会把它一个一个拆开，然后去重,空格是空格去重
{'t', ' ', 's', 'h', 'e', 'i'}
```

通过 `add(key)` 方法可以添加元素到 `set` 中，可以重复添加，但不会有效果：

```
s = {1,2,3,4}
s.add(5)
s
{1,2,3,4,5}

s.add(5)
{1,2,3,4,5}
```


可以通过 `update()` 方法，将另一个对象更新到已有的集合中，这一过程同样会进行去重。

```
>>> s
{1, 2, 3, 4, 5}
>>> s.update("json")
>>> s
{1, 2, 3, 4, 5, 'j', 's', 'n', 'o'}
```

通过 `remove(key)` 方法删除指定元素，或者使用 `pop()` 方法。

注意，集合的 `pop` 方法无法设置参数，删除指定的元素：

```
s
{1, 2, 3, 4, 5, 'j', 's', 'n', 'o'}

s.remove("n")
{1, 2, 3, 4, 5, 'j', 's', 'o'}
```

```
s.pop()         # 弹出第一个元素
1

s.pop(3)

Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    s.pop(3)
TypeError: pop() takes no arguments (1 given)
```

需要注意的是，集合 `不能取出某个元素` ，因为集合既 `不支持下标索引` 也 `不支持` 字典那样的 `通过键值对` 获取。

除了 `add` 、 `clear` 、 `copy` 、 `pop` 、 `remove` 、 `update` 等集合常规操作，剩下的全是数学意义上的集合操作，交并差等等

对集合进行交并差等，既可以使用 `union` 一类的英文方法名，也可以更方便的使用 `减号` 表示 `差集` ， `“&”` 表示 `交集` ， `“|”` 表示 `并集` 。

```
x = set('runoob')
y = set('google')
x, y
(set(['b', 'r', 'u', 'o', 'n']), set(['e', 'o', 'g', 'l']))   # 重复的被删除
x & y         # 交集
set(['o'])
x | y         # 并集
set(['b', 'e', 'g', 'l', 'o', 'n', 'r', 'u'])
x - y         # 差集
set(['r', 'b', 'u', 'n'])
```

集合数据类型属于Python内置的数据类型，但不被重视，在很多书籍中甚至都看不到一点介绍。

其实，集合是一种非常有用的数据结构，它的去重和集合运算是其它内置类型都不具备的功能，在很多场合有着非常重要的作用，比如网络爬虫。

第4章 Python流程控制

流程控制指的是代码运行逻辑、分支走向、循环控制，是真正体现我们程序执行顺序的操作。流程控制一般分为顺序执行、条件判断和循环控制。

4.1-顺序执行

Python代码在执行过程中，遵循下面的基本原则：

- 普通语句，直接执行；
- 碰到函数，将函数体载入内存，并不直接执行
- 碰到类，执行类内部的普通语句，但是类的方法只载入，不执行
- 碰到if、for等控制语句，按相应控制流程执行
- 碰到@，break，continue等，按规定语法执行
- 碰到函数、方法调用等，转而执行函数内部代码，执行完毕继续执行原有顺序代码

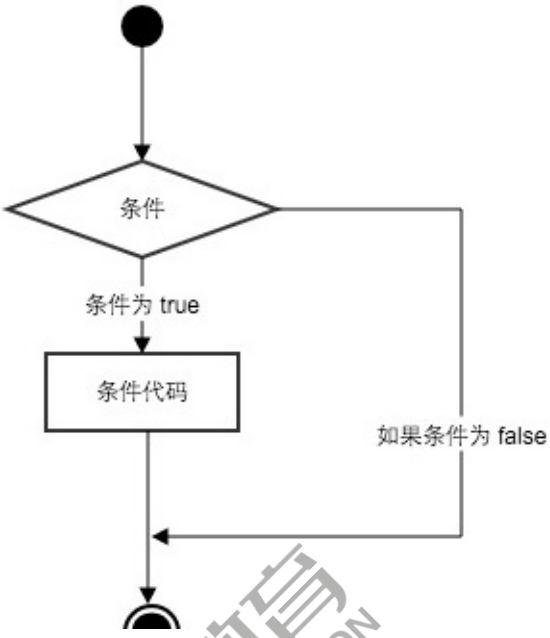
if name == 'main':

很多时候，我们经常在python程序中看到这么一行语句，这里简要解释一下：
首先， `__name__` 是所有模块都会有的一个内置属性，一个模块的 `__name__` 值取决于你如何调用模块。
假如你有一个 `test.py` 文件，如果在 `a.py` 文件中使用import导入这个模块 `import test.py` ，那么 `test.py` 模块的 `__name__` 属性的值就是 `test` ，不带路径或者文件扩展名。
其实顺序执行，简单来说就是代码按照从上到下的顺序来执行。

4.2-条件判断

条件判断是通过一条或多条判断语句的执行结果（True或者False）来决定执行的代码块。

在Python语法中，使用 `if` 、 `elif` 和 `else` 三个关键字来进行条件判断。



if语句的一般形式如下所示

```
if condition1:                # condition1为True 执行statement_block_1
    statement_block_1
elif condition2:              # condition2为True 执行statement_block_2
    statement_block_2
else:                         # 都不是的话执行 statement_block_3
    statement_block_3
```

条件判断的使用原则：

- 每个条件后面要使用冒号（:）作为判断行的结尾，表示接下来是满足条件（结果为True）后要执行的语句块。
- 除了if分支必须有，elif和else分支都可以根据情况省略。
- 使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。
- 顺序判断每一个分支，任何一个分支首先被命中并执行，则其后面的所有分支被忽略，直接跳过！
- 在Python中没有switch – case语句。

接下来看一个案例：

```
number = 20
print("猜数字")
```

```
while True:
        guess = int(input("请输入你猜的数字:"))
        if guess == number:
                print("恭喜, 你猜对了")
                break
        elif guess<number:
                print("猜的数字太小了")
        elif guess>number:
                print("猜的数字太大了")
```

`if/else` 语句可以嵌套，也就是把 `if...elif...else` 结构放在另外一个 `if...elif...else` 结构中。

形成如下的结构：

```
var = 100
if var < 200:
    print("比200小！")
    if var == 150:
        print('这是150')
    elif var == 100:
        print('这是100')
    elif var == 50:
        print('这是50')
elif var < 50:
    print("比50小！")
else:
    print("无法判断正确的值！")

print("Good bye!")
```

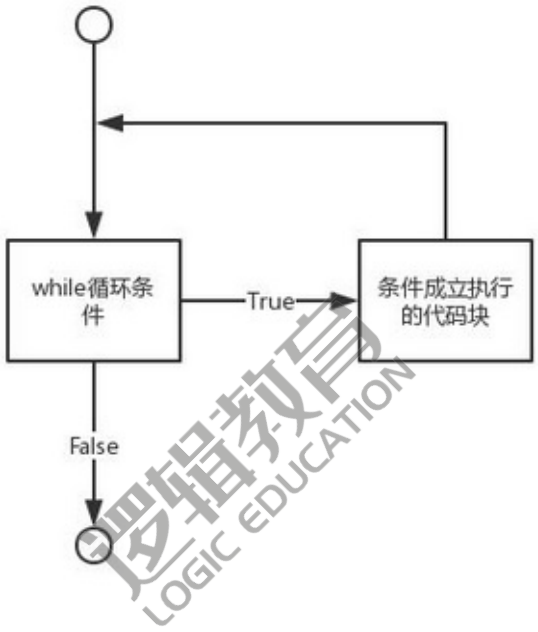
4.3-循环控制

循环控制，就是让程序循环运行某一段代码直到满足退出的条件，才退出循环。

Python用关键字for和while来进行循环控制，但是没有其它语言的do...while语句（在Java和PHP中都有do while）。

4.3.1 while循环

while循环语句的控制结构图如下所示：




```
while 判断表达式：
    内部代码块

# 接下来看一个具体的案例，求1-100之间的总和
n = 100
sum = 0
counter = 1
while counter <= n:
    sum = sum + counter
    counter += 1

print("1 到 %d 之和为: %d" % (n,sum))
----------------------------------------
# 通常在循环条件中，会设置退出条件，防止程序死循环。
while True:
    s = input("what's you name:")
    print("you name is %s"%s)
print("bye")
这个程序永远也不会打印bye。
```

while的else从句：

while循环还可以增加一个else从句。

当while循环正常执行完毕，会执行else语句。注意else与while平级的缩进方式！

```
number = 10
i = 0
# i = 11
while i < number:
    print(i)
    i += 1
else:
    print("执行完毕！")
```

如果是被break等机制强制提前终止的循环，不会执行else语句。

```
number = 10
i = 0
while i < number:
    print(i)
    i += 1
    if i == 7:
        break
else:
    print("执行完毕！")
```


4.3.2 for循环

虽然与while一样都是循环的关键字，但for循环通常用来遍历可迭代的对象，如一个列表或者一个字典。其一般格式如下：

```
for <variable> in <sequence>:
    <statements>
for ... in ....:属于固定套路
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print(sum)
```

4.3.3 循环的嵌套

if判断可以嵌套，while和for当然也可以嵌套。但是建议大家不要嵌套3层以上，那样的效率会很低。

```
# 这是一个判断质数的程序
# 一个数，如果只有1和它本身两个因数，这样的数叫做质数（或素数）

for n in range(2, 100):
    for x in range(2, n):
        if n % x == 0:
            print(n, '等于', x, '*', n//x)
            break
    else:
        # 循环中没有找到元素
        print(n, ' 是质数')
        # print(n,end = ', ')
```

4.3.4 break语句

想在循环过程中退出循环，怎么办？用break语句！

break只能用于循环体内。其效果是直接结束并退出当前循环，剩下的未循环的工作全部被忽略和取消。

Python的break只能退出一层循环，对于多层嵌套循环，不能全部退出。

```
for letter in 'Hello world':     # 第一个实例
   if letter == 'r':
      break
   print ('当前字母为 :', letter)
------------------------------------------
var = 10                         # 第二个实例
while var > 0:
   print ('当前变量值为 :', var)
   var -= 1
   if var == 5:
      break
```


4.3.5 continue语句

与break不同，continue语句用于跳过当前循环的剩余部分代码，直接开始下一轮循环。它不会退出和终止循环，只是提前结束当前轮次的循环。同样的，continue语句只能用在循环内。

```
for letter in 'Hello world':     # 第一个实例
   if letter == 'o':             # 字母为 o 时跳过输出
      continue
   print ('当前字母 :', letter)
 ------------------------------------------------------------
var = 10                      # 第二个实例
while var > 0:
   var -= 1
```

```
    if var == 5:             # 变量为 5 时跳过输出
       continue
    print ('当前变量值 :', var)
```

前面跟大家说过，python中的break只能跳出当前层的循环，无法全部跳出。那如果有这个需求怎么办？

```
# 设置flag

flag = False              # 用于控制外层循环的标志
for i in range(10):
    if flag:              # 当flag被内层循环设置为True的时候，跳出外层循环
        break
    for j in range(10):
        if j==7:
            flag = True
            break
        print(i,j)
```

第5章 Python函数

5.1-range函数

在其他语言中，如果想要循环一个变量从1到100，要怎么写呢？

```
for(i = 1,i<=100,i++){}
```

python怎么实现这个功能呢？python设计了range()函数，直接实现了上面的功能。range是内置函数，无须导入。在任何地方都可以直接使用它。

```
for i in range(5):
    print(i)
0
1
2
3
4

# 从上面的执行结果，也可以看出来range函数的范围是从0-4。
----------------------------------------------------------------------------
# 遵守左闭右开的原则。range默认是从0开始的。当然也可以指定遍历的区间。
for i in range(1,5):
    print(i)
1
2
3
4
----------------------------------------
# 还可以指定步长，就像切片一样。
for i in range(1, 12, 2):
    print(i)
1
3
5
7
9
11
----------------------------------------
# 很多的时候是结合range()和len()函数，遍历一个序列的索引
a = ['Google', 'Baidu', 'Huawei', 'Taobao', 'QQ']
for i in range(len(a)):
    print(i, a[i])
```

5.2-匿名函数

当我们在创建函数时，有些时候，不需要显式地定义函数，直接传入匿名函数更方便。这省去了我们挖空心思为函数命名的麻烦，也能少写不少代码，很多编程语言都提供这一特性。

Python语言使用 `lambda` 关键字来创建匿名函数。

所谓匿名，即不再使用 `def` 语句这样标准的形式定义一个函数。

- lambda只是一个表达式,而不是一个代码块，函数体比def简单很多。
- l仅仅能在lambda表达式中封装有限的逻辑。
- llambda 函数拥有自己的命名空间。

例如： `lambda x: x * x` 。它相当于下面的函数：

```
def f(x):
    return x*x

# 关键字lambda表示匿名函数，冒号前面的x表示函数参数，x*x是执行代码
```

匿名函数只能有一个表达式，不用也不能写return语句，表达式的结果就是其返回值。 匿名函数没有函数名字，不必担心函数名冲突，节省字义空间。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：


```
f = lambda x: x * x
f
<function <lambda> at 0x3216fef44>
f(6)
36

# 也可以把匿名函数作为别的函数的返回值返回
def count(i,j):
    return lambda : i*j

f = count(6,7)
print(f())
```

5.2.1 匿名函数的应用

对列表中的字典进行排序

```
lis = [1,-2,4,-3]
lis.sort(key=abs)
print(lis)
```

```
infors = [{'name':'cangls','age':18},{'name':'bols','age':20},{'name':'jtls','age':25}]

infors.sort(key = lambda x:x['name'])

print(infors)
```

5.2.2 匿名函数当做实参

```
def test(a,b,func):
    result = func(a,b)
    return result

nums = test(11,22,lambda x,y:x+y)
print(nums)
```

5.3-推导式

Python语言有一种独特的推导式语法，相当于语法糖的存在，可以帮你在某些场合写出比较精简酷炫的代码。

5.3.1 列表推导式

列表推导式是一种快速生成列表的方式。其形式是用方括号括起来的一段语句

```
lis = [x * x for x in range(1, 10)]
print(lis)
-----------------------------------
# 结果：
[1, 4, 9, 16, 25, 36, 49, 64, 81]
-----------------------------------------
# 如果不用推导式，如何完成这个功能
lis = []
for i in range(10):
    lis.append(i*i)
print(lis)

# 打印结果：
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
-----------------------------------------
# 列表推导式，还可以增加一些条件，其核心语法是用中括号[]将生成逻辑封装起来
lis = [i*i for i in range(10) if i%2 == 0]
print(lis)

# 打印结果：
[0, 4, 16, 36, 64]
-----------------------------------------
# 多重循环
lis = [a+b for a in '123' for b in 'abc']
print(lis)

# 同时循环a和b两个变量，在进行拼接
# 打印结果：
['1a', '1b', '1c', '2a', '2b', '2c', '3a', '3b', '3c']
-----------------------------------------
# 更多用法
dic = {"k1":'v1','k2':'v2'}
lis = [k+":"+v for k,v in dic.items()]
print(lis)

# 打印结果：
['k1:v1', 'k2:v2']
```

5.3.2 字典推导式

有列表推导式，那有没有字典推导式呢？答案是有的，使用{}就可以定义了。

```
dic = {i:i**3 for i in range(5)}
print(dic)

# i的三次方
# 打印结果：
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64}
```

5.3.3 集合推导式

大括号除了能用作字典推导式，还可以用作集合推导式，两者仅仅在细微处有差别。

```
s = {i for i in 'abasrdfsdfa' if i not in 'abc'}
print(s)

# 把abc排除剩余字符串
# 打印结果：
{'r', 'f', 's', 'd'}
```


有同学会说还有元组推导式，但是在python中元组被用作生成器了。是没有元组推导式的

```
tup = (x for x in range(9))
print(tup)
print(type(tup))

# 打印结果：
<generator object <genexpr> at 0x000000000255DA98>
<class 'generator'>
----------------------------------------------------------
tup = tuple(x for x in range(9))
print(tup)
print(type(tup))

# 打印结果：
(0, 1, 2, 3, 4, 5, 6, 7, 8)
<class 'tuple'>
```

5.3.4 面试真题

看下面代码回答输出的结果是什么?为什么？

```
result = [lambda x: x + i for i in range(10)]
print(result[0](10))
```

分析：

```
result = [lambda x: x + i,lambda x: x + i,lambda x: x + i,....]
print((lambda x: x + i)(10))
f = lambda x: x + i
f(10)
```

5.4-迭代器

在介绍迭代器之前，先说明下迭代的概念：

迭代：通过for循环遍历对象的每一个元素的过程。

Python的for语法功能非常强大，可以遍历任何可迭代的对象。

在Python中，`list/tuple/string/dict/set/bytes` 都是可以迭代的数据类型。

可以通过collections模块的 `Iterable` 类型来判断一个对象是否可迭代：

```
from collections import Iterable
isinstance('abc', Iterable)          # str是否可迭代
True
isinstance([1,2,3], Iterable)        # list是否可迭代
True
isinstance(123, Iterable)            # 整数是否可迭代
False
```

5.4.1 迭代器是什么

迭代器是一种可以被遍历的对象，并且能作用于next()函数。迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。 迭代器只能往后遍历不能回溯，不像列表，你随时可以取后面的数据，也可以返回头取前面的数据。迭代器通常要实现两个基本的方法： `iter()` 和 `next()` 。

字符串，列表或元组对象，甚至自定义对象都可用于创建迭代器：


```
from collections import Iterable,Iterator

print(isinstance([],Iterable))       # 是否可迭代
print(isinstance([],Iterator))       # 是否是迭代器

lis=[1,2,3,4]
it = iter(lis)       # 使用Python内置的iter()方法创建迭代器对象
next(it)             # 使用next()方法获取迭代器的下一个元素
1
next(it)
2
next(it)
3
next(it)
4
next(it)             # 当后面没有元素可以next的时候，弹出错误

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    next(it)
```

```
StopIteration
----------------------------------------------------
# 或者使用for循环遍历迭代器：
lis = [1,2,3,4]
it = iter(lis)          # 创建迭代器对象
for x in it:            # 使用for循环遍历迭代对象
    print (x, end=" ")
```

总结：Python的迭代器表示的是一个元素流，可以被next()函数调用并不断返回下一个元素，直到没有元素时抛出StopIteration错误。可以把这个元素流看做是一个有序序列，但却不能提前知道序列的长度，只能不断通过next()函数得到下一个元素，所以迭代器可以节省内存和空间。

5.4.2迭代器(Iterator)和可迭代(Iterable)的区别

1. 凡是可作用于for循环的对象都是可迭代类型；
2. 凡是可作用于next()函数的对象都是迭代器类型；
3. list、dict、str等是可迭代的但不是迭代器，因为next()函数无法调用它们。可以通过iter()函数将它们转换成迭代器。
4. Python的for循环本质上就是通过不断调用next()函数实现的。

5.5-生成器

有时候，序列或集合内的元素的个数非常巨大，如果全制造出来并放入内存，对计算机的压力是非常大的。比如，假设需要获取一个10**20次方如此巨大的数据序列，把每一个数都生成出来，并放在一个内存的列表内，这是粗暴的方式，有如此大的内存么？如果元素可以按照某种算法推算出来，需要就计算到哪个，就可以在循环的过程中不断推算出后续的元素，而不必创建完整的元素集合，从而节省大量的空间。在Python中，这种一边循环一边计算出元素的机制，称为生成器：generator。

生成生成器：

```
g = (x * x for x in range(1, 4))
g
<generator object <genexpr> at 0x1022ef630>
```

可以通过 `next()` 函数获得 `generator` 的下一个返回值，这点和迭代器非常相似：

```
next(g)
1
next(g)
4
next(g)
9
next(g)
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    next(g)
StopIteration
-------------------------------------------------
# 但更多情况下，我们使用for循环。
for i in g:
    print(i)
```

除了使用生成器推导式，我们还可以使用 `yield` 关键字。

```
def createNums():
    print("----func start------")
    a,b = 0,1
    for i in range(5):
        # print(b)
        print("--1--")
        yield b
        print("--2--")
        a,b = b,a+b    # a,b = 1, 1     a,b = 1,2
        print("--3--")
```

```
    print("----func end------")
g= createNums()
next(g)          # 如果想得到yield的值,可以打印next(g)
```

在 Python中，使用 `yield` 返回的函数会变成一个生成器（ `generator` ）。 在调用生成器的过程中，每次遇到yield时函数会暂停并保存当前所有的运行信息，返回yield的值。并在下一次执行next()方法时从当前位置继续运行。

```
# 斐波那契函数
def fibonacci(n):
    a = 0
    b = 1
    counter = 0
    while True:
        if counter > n:
            return
        yield a              # yield让该函数变成一个生成器
        a, b = b, a + b
        counter += 1

fib = fibonacci(10)            # fib是一个生成器
print(type(fib))
for i in fib:
    print(i, end=" ")
```


生成器是可以循环的,相比next来说,for循环更友好

```
a = createNums()
这两种取值方式是一样的！！！
a.__next__()
next(a)

for i in a:
    print(i)
```

5.5.1 send

```
def test():
    i = 0
    while i<5:
        temp = yield i
        print(temp)
        i+=1

t = test()
```

```
next(t)
next(t)
t.send("juran")
next(t)
-------------------------------------------
t = test()
t.send("juran")
Traceback (most recent call last):
  File "/Users/binbin/Desktop/Python/demo.py", line 179, in <module>
    t.send("juran")
TypeError: can't send non-None value to a just-started generator

# 如何解决这个错误?
> next(t)
  t.send("juran")

> send(None)
```

5.5.2 生成器的应用

实现多任务


```
def test1():
    while True:
        print("--1--")
        yield None

def test2():
    while True:
        print("--2--")
        yield None

t1 = test1()
t2 = test2()
while True:
    next(t1)
    next(t2)
```

5.6-装饰器

装饰器（Decorator）：从字面上理解，就是装饰对象的器件。可以在不修改原有代码的情况下，为被装饰的对象增加新的功能或者附加限制条件或者帮助输出。装饰器有很多种，有函数的装饰器，也有类的装饰器。装饰器在很多语言中的名字也不尽相同，它体现的是设计模式中的装饰模式，强调的是开放封闭原则。装饰器的语法是将@装饰器名，放在被装饰对象上面。

```
@dec
def func():
    pass
```

在介绍装饰器之前，先要了解几个知识点。

重名函数会怎么样？

```
def test():
    print('1')
def test():
    print('2')
test()
# 这个时候会输出什么？
# 会输出2，后面的会覆盖前面的函数
--------------------------------
def foo():
    print('test')
foo = lambda x:x+1
foo()
# 此时foo()的执行结果是什么？
# foo会执行匿名函数的内容，而不会执行函数foo，打印test
```


接下来就是一个比较经典的案例了，有一个大公司，下属的基础平台部负责内部应用程序及API的开发。另外还有上百个业务部门负责不同的业务，这些业务部门各自调用基础平台部提供的不同函数，也就是API处理自己的业务，情况如下：

基础平台部门提供的功能如下：

```
def f1():
    print("业务部门1的数据接口.....")
def f2():
    print("业务部门2的数据接口.....")
def f3():
    print("业务部门3的数据接口.....")
def f100():
    print("业务部门100的数据接口.....")
```

```
# 各部门分别调用基础平台提供的功能
f1()
f2()
f3()
f100()
```

目前公司发展壮大，但是以前没有考虑到验证相关的问题，即：基础平台提供的功能可以被任何人使用。现在需要对基础平台的所有功能进行重构，为平台提供的功能添加验证机制，执行功能前，先进行验证 老大把工作交给A，他是这么做的： 跟每个业务部门交涉，每个业务部门自己写代码，调用基础平台的功能之前先验证。这样一来基础平台就不需要做任何修改了。 当前A被老大开除了 老大又把工作交给B，他是这么做的：

```
def f1():
    # 验证1
    # 验证2
    # 验证3
def f2():
    # 验证1
    # 验证2
    # 验证3
过了一周B也被开除了。。。。。。。。。。
```

老大又把工作交给C，他是这么做的：


```
def check_login()
    # 验证1
    # 验证2
    # 验证3
def f1():
    check_login()
def f2():
    check_login()
```

老大说写代码要遵循开放封闭的原则，虽然在这个原则是用的面向对象开发，但是也适用于函数式编程，简单来说，已经实现的功能不允许修改，但可以被扩展 。

封闭：已实现的功能代码块

开放：对扩展开发

```
老大给出了方案，
def foo():
    def inner():
        验证1
        验证2
        验证3
        result = func()
```

```
        return result
return inner

@foo
def f1():
    pass
@foo
def f2():
    pass
```

看不懂？没关系，接下来写一个简单点的

```
def w1(func):
    def inner():
        print('正在验证权限')
        func()
    return inner

def f1():
    print('f1')

innerFunc = w1(f1)          # innerFunc = inner
innerFunc()                 # inner()

f1 = w1(f1)
f1()
```



```
def outer(func):
    def inner():
        print("认证成功！")
        result = func()
        print("日志添加成功")
    return inner

@outer                  # f1 = outer(f1) = inner
def f1():
    print("业务部门1数据接口.....")

f1()
```

1. 程序开始运行，从上往下解释，读到def outer(func):的时候，发现这是个“一等公民”函数，于是把函数体加载到内存里，然后过。
2. 读到@outer的时候，程序被@这个语法糖吸引住了，知道这是个装饰器，按规矩要立即执行的，于是程序开

始运行@后面那个名字outer所定义的函数。

3. 程序返回到outer函数，开始执行装饰器的语法规则。

规则是：被装饰的函数的名字会被当作参数传递给装饰函数。装饰函数执行它自己内部的代码后，会将它的返回值赋值给被装饰的函数。原来的f1函数被当做参数传递给了func，而f1这个函数名之后会指向inner函数

5.6.1 两个装饰器

```
def makeBold(fn):
    def wrapped():
        return "<b1>" + fn() + "</b1>"
    return wrapped

def makeItalic(fn):
    def wrapped():
        return "<b2>" + fn() + "</b2>"
    return wrapped



@makeBold       # test3 = makeBold(test3) = wrapped
@makeItalic     # test3 = makeItalic(test3) = wrapped
def test3():
    return "hello world"

print(test3())
```


5.6.2 装饰器的执行时间

```
def w1(func):
    print("--正在装饰--")
    def inner():
        print("--正在权限验证--")
        func()
    return inner

# 只要Python解释器执行到了这个代码，那么就会自动的进行装饰，而不是等到调用的时候才进行装饰
@w1     # f1 = w1(f1)
def f1():
    print("--f1--")

# 在调用f1前已经装饰了
f1()
```

练习

```
def w1(func):
    print("正在装饰1")
    def inner():
        print("正在验证权限1")
        func()
    return inner()

def w2(func):
    print("正在装饰2")
    def inner():
        print("正在验证权限2")
        func()
    return inner()

@w1
@w2
def f1():
    print("--f1--")
f1()
```

5.7-内置函数

前面使用过一些函数，有的同学会疑问我没有导入这个函数，为什么可以直接使用？

因为这些函数都是一个叫做 `builtins` 模块中定义的函数，而 `builtins` 模块默认在Python环境启动的时候就自动导入，所以你可以直接使用这些函数。

我们可以在IDLE 进行输出

```
globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<class '_frozen_importlib.BuiltinImporter'>, '__spec__': None, '__annotations__'
: {},
 '__builtins__': <module 'builtins' (built-in)>}

dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'Block
ingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'Connect
ionAbortedError',
'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'Deprecati
onWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsEr
ror',
'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'I
OError',
'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedE
rror',
'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryErr
or',
'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplement
ed',
'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning',

'PermissionError', 'ProcessLookupError', 'RecursionError', 'ReferenceError',
'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'Sto
pIteration',
'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'Timeo
utError',
'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeE
rror',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'Valu
eError',
'Warning', 'WindowsError', 'ZeroDivisionError', '_', '__build_class__', '__debu
g__',
'__doc__', '__import__', '__loader__', '__name__', '__package__', '__spec__', '
abs', 'all',
'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classm
```

```
ethod',
'compile', 'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod'
, 'enumerate',
'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr', 'g
lobals',
'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubcla
ss', 'iter',
'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed',
'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', '
type', 'vars',
'zip']
```

builtins模块里有接近80个内置函数，60多个内置异常，还有几个内置常数，特殊名称以及模块相关的属性。

接下来给大家介绍一些工作中常用的一些内置函数：

5.7.1 abs()

绝对值函数。

以abs()函数为例，展示两个特性。一是内置函数是可以被赋值给其他变量的，同样也可以将其他对象赋值给内置函数，这时就完全变了。

所以，内置函数不是Python关键字，要注意对它们的保护，不要使用和内置函数重名的变量名，这会让代码混乱，容易发生难以排查的错误。

如 `abs(-1) = 1`


```
abs(-10)
10
f = abs
f(-1)
1

abs = id
abs(1)
1869788224
```

5.7.2 all()

接收一个可迭代对象，如果对象里的所有元素的bool运算值都是True，那么返回True，否则False

```
all([1,1,1])
True
```

```
all([1,1,0])
False
```

5.7.3 any()

接收一个可迭代对象，如果迭代对象里有一个元素的bool运算值是True，那么返回True，否则False。与all()是一对兄弟。

```
any([0,0,1])
True
any([0,0,0])
False
```

5.7.4 bin()、oct()、hex()

三个函数是将十进制数分别转换为2/8/16进制。

```
i = 10
bin(i)
'0b1010'        # 0b表示2进制
oct(i)
'0o12'          # 0o表示8进制
hex(i)
'0xa'           # 0x表示16进制
```


5.7.5 bool()

测试一个对象或表达式的执行结果是True还是False。

```
bool(1==2)
False
bool(abs(-1))
True
bool(None)
False
```

5.7.6 bytes()

将对象转换成字节类型。例如： `s = '张三';m = bytes(s,encoding='utf-8')`

```
i=2
bytes(i)
```

```
b'\x00\x00'
s = 'haha'
bytes(s)

Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    bytes(s)
TypeError: string argument without an encoding
------------------------------
bytes(s, encoding="utf-8")
b'haha'
bytes(s, encoding="GBK")
b'haha'
s = "juran"
s.encode()                  # 将字符串转换成bytes
```

5.7.7 str()

将对象转换成字符串类型，同样也可以指定编码方式。例如： `str(bytes对象, encoding='utf-8')`

```
i =  2
str(i)
'2'
b = b"haha"
str(b)      # 注意！
"b'haha'"
str(b,encoding="gb2312")
'haha'
str([1,2,3,])
'[1, 2, 3]'

b = b'juran'
b.decode()              # 将bytes转换成str
# Bytes和string之间的互相转换，更多使用的是encode()和decode()方法。
```


5.7.8 chr()

返回某个十进制数对应的ASCII字符，例如： `chr(99) = 'c'` 。 它可以配合 `random.randint(65, 90)` 随机方法，生成随机字符，用于生产随机验证码。

```
import random
for i in range(10):
    a = random.randint(65,90)           # 65-90是ASCII表中A-Z
    c = chr(a)
    print(c,end='')
print('')
```

5.7.9 ord()

与chr()相反，返回某个ASCII字符对应的十进制数，例如， `ord('A') = 65`

```
ord("A")
65
ord("\n")
10
```

5.7.10 compile()

将字符串编译成Python能识别或执行的代码。

```
s  = "print('helloworld')"
r = compile(s,"<string>","exec")
r
<code object <module> at 0x000001B23E6BE660, file "<string>", line 1>
r()

Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    r()
TypeError: 'code' object is not callable

exec(r)             # 执行的话需要用exec
helloworld
eval(r)             # eval也可以
helloworld
```


5.7.11 complex()

通过数字或字符串生成复数类型对象。

```
complex(1,2)
(1+2j)
```

5.7.12 dir()

显示对象所有的属性和方法。

```
dir([1,2,])
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__'
, '__doc__',

'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__',

'__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__len__',

 '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__re
pr__',

'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str_
_',

 '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'in
sert', 'pop',

 'remove', 'reverse', 'sort']
```

5.7.13 divmod()

除法，同时返回商和余数的元组。

```
divmod(10,3)
(3, 1)
divmod(11,4)
(2, 3)
```


5.7.14 format()

执行format()，其实就是调用该对象所属类的 `__format__` 方法 。

```
format("324324")
'324324'
format([1,2,3])
'[1, 2, 3]'
```

5.7.15 globals()

列出当前环境下所有的全局变量。注意要与global关键字区分！

5.7.16 hash()

为不可变对象，例如字符串生成哈希值的函数！

```
hash("i am jack")
5602200374213231465
hash(1)
1
hash(100000)
100000
hash([1,2,3,])
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    hash([1,2,3,])
TypeError: unhashable type: 'list'
hash((1,2,3))
2528502973977326415
```

5.7.17 id()

返回对象的内存地址,常用来查看变量引用的变化，对象是否相同等。常用功能之一！

```
id(0)
1456845856
id(True)
1456365792
a = "Python"
id(a)
37116704
```


5.7.18 iter()

制造一个迭代器，使其具备next()能力。

```
lis = [1, 2, 3]
next(lis)

Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    next(lis)
TypeError: 'list' object is not an iterator

i = iter(lis)
i
<list_iterator object at 0x0000000002B4A128>
next(i)
1
```

5.7.19 len()

返回对象的长度。不能再常用的函数之一了。

5.7.20 max min

返回给定集合里的最大或者最小的元素。可以指定排序的方法！

```
lis = ['abcaaaa','abcd','abcde']
max(lis)
指定按照长度返回
max(lis,key=len)
```

5.7.21 next()

通过调用迭代器的 `__next__()` 方法，获取下一个元素。

5.7.22 open()

打开文件的方法。在Python2里，还有一个file()方法，Python3中被废弃了 。

5.7.23 pow()

幂函数。


```
>>> pow(3, 2)
9
```

5.7.24 reversed()

反转，逆序对象

```
>>> reversed            # reversed本身是个类
<class 'reversed'>
>>> reversed([1,2,3,4,5])   # 获得一个列表反转器
<list_reverseiterator object at 0x0000022E322B5128>
>>> a = reversed([1,2,3,4,5])
>>> a
<list_reverseiterator object at 0x0000022E32359668>
>>> list(a)         # 使用list方法将它转换为一个列表
[5, 4, 3, 2, 1]
```

5.7.25 round()

四舍五入 .

```
round(1.5)
2
round(1.4)
1
```

5.7.26 sum()

求和 .

```
>>> sum(1,2,3)          # 需要传入一个可迭代的对象
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    sum(1,2,3)
TypeError: sum expected at most 2 arguments, got 3
>>> sum([1,2,3])            # 传入一个列表
6
>>> sum({1:1,2:2})          # 突发奇想，作死传入一个字典
3
```

5.7.27 type()

显示对象所属的数据类型。常用方法！前面已经展示过。

5.7.28 filter()

过滤器，用法和map类似。在函数中设定过滤的条件，逐一循环对象中的元素，将返回值为True时的元素留下（注意，不是留下返回值！），形成一个filter类型的迭代器。

```
def f1(x):
    if x > 3:
        return True
    else:
        return False
li = [1,2,3,4,5]
data = filter(f1,li)
print(type(data))
print(list(data))
----------------------------
# 运行结果：
<class 'filter'>
[4, 5]
```

5.7.29 zip()

组合对象。将对象逐一配对。

```
list_1 = [1,2,3]
list_2 = ['a','b','c']
s = zip(list_1,list_2)
print(list(s))
--------------------------------
运行结果：

[(1, 'a'), (2, 'b'), (3, 'c')]
```

组合3个对象：

```
list_1 = [1, 2, 3, 4]
list_2 = ['a', 'b', 'c', "d"]
list_3 = ['aa', 'bb', 'cc', "dd"]
s = zip(list_1, list_2, list_3)
print(list(s))

运行结果：
[(1, 'a', 'aa'), (2, 'b', 'bb'), (3, 'c', 'cc'), (4, 'd', 'dd')]
```

那么如果对象的长度不一致呢？多余的会被抛弃！以最短的为基础！

```
list_1 = [1,2,3]
list_2 = ['a','b','c',"d"]
s = zip(list_1,list_2)
print(list(s))
--------------------------------
运行结果：
[(1, 'a'), (2, 'b'), (3, 'c')]
```

5.7.30 sorted()

排序方法。有 `key` 和 `reverse` 两个重要参数。

基础用法: 直接对序列进行排序

```
>>> sorted([36, 5, -12, 9, -21])
[-21, -12, 5, 9, 36]
```

指定排序的关键字。关键字必须是一个可调用的对象。例如下面的例子，规则是谁的绝对值大，谁就排在后面。

```
>>> sorted([36, 5, -12, 9, -21], key=abs)
[5, 9, -12, -21, 36]
```

指定按反序排列。下面的例子，首先按忽略大小写的字母顺序排序，然后倒序排列。

```
>>> sorted(['bob', 'about', 'Zoo', 'Credit'], key=str.lower, reverse=True)
['Zoo', 'Credit', 'bob', 'about']
```

第6章 Python文件读写

6.1-文件类型

文件的作用

就是把一些存储存放起来，可以让程序下一次执行的时候直接使用，而不必重新制作一份，省时省力

6.2-文件的基础操作

6.2.1 打开文件

Python内置了一个 `open()` 方法，用于对文件进行读写操作。

使用open()方法操作文件就像把大象塞进冰箱一样，可以分三步走，一是打开文件，二是操作文件，三是关闭文件。

`open()` 方法的返回值是一个file对象，可以将它赋值给一个变量（文件句柄）。其基本语法格式为:

```
f = open(filename, mode)
```

`filename` 文件名称

`mode` 打开模式

打开模式常用的有 `r(读模式,文件必须存在)` ， `w(写模式)` ，当然还有一些其他方式。

问模式	说明
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

6.2.2 文件的关闭

```
f.close()
```

6.2.3 编码问题

要读取非UTF-8编码的文件，需要给open()函数传入encoding参数，例如，读取GBK编码的文件：

```
>>> f = open('gbk.txt', 'r', encoding='gbk')
>>> f.read()
'GBK'
```

遇到有些编码不规范的文件，可能会抛出 `UnicodeDecodeError` 异常，这表示在文件中可能夹杂了一些非法编码的字符。遇到这种情况，可以提供errors参数，表示如果遇到编码错误后如何处理。

```
f = open('gbk.txt', 'r', encoding='gbk', errors='ignore')
```

6.3-文件对象操作

每当我们用open方法打开一个文件时，将返回一个文件对象。这个对象内置了很多操作方法。下面假设，已经打开了一个f文件对象。

6.3.1 f.read(size)

读取一定大小的数据, 然后作为字符串或字节对象返回。size是一个可选的数字类型的参数，用于指定读取的数据量。当size被忽略了或者为负值，那么该文件的所有内容都将被读取并且返回。

```
f = open("1.txt", "r")
str = f.read()
print(str)

str = f.read()
print(str)

str = f.read(1)
print(str)

f.close()
```


6.3.2 f.readline()

从文件中读取一行n内容。换行符为'\n'。如果返回一个空字符串，说明已经已经读取到最后一行。这种方法，通常是读一行，处理一行，并且不能回头，只能前进，读过的行不能再读了。

```
f = open("1.txt", "r")
str = f.readline()
print(str)
f.close()
```

6.3.3 f.readlines()

将文件中所有的行，一行一行全部读入一个列表内，按顺序一个一个作为列表的元素，并返回这个列表。readlines方法会一次性将文件全部读入内存，所以也存在一定的风险。但是它有个好处，每行都保存在列表里，可以随意存取。

```
f = open("1.txt", "r")
a = f.readlines()
print(a)
f.close()
```

6.3.4 f.write()

将字符串或bytes类型的数据写入文件内。write()动作可以多次重复进行，其实都是在内存中的操作，并不会立刻写回硬盘，直到执行close()方法后，才会将所有的写入操作反映到硬盘上

```
# 打开一个文件
f = open("foo.txt", "w")
f.write("人生苦短我用Python!\n")

# 关闭打开的文件
f.close()
```

练习

创建两个文件，一个文件中放入'人生苦短 我用Python'，一个文件读取刚才创建的文件内容。

6.3.5 读取大文件的处理方式

比如一个文件很大，比如5G，怎么把文件的数据读取到内存然后进行处理呢？


```
while True:
    content = filename.read(1024)          每次读取1024个字节
    if len(content)==0:                    如果读取内容长度等于0，意味着文件读取完毕
    break
```

6.3.6 文件的定位读写- f.seek()

```
f = open(filename)
第一个参数     开始的偏移量，也就是代表需要移动偏移的字节数
第二个参数     0 从文件开始读取     1 从当前位置去读     2 从文件末尾开始读取
f.seek(2,0)
cont = f.readline()      打印出来的结果是从filename第二个字节开始的
print(cont)
con = f.read()
print(con)               打印的剩下的所有内容
```

6.3.7 f.tell()

返回文件读写指针当前所处的位置,它是从文件开头开始算起的字节数。一定要注意了，是字节数，不是字符数。

6.3.8 f.close()

关闭文件对象。当处理完一个文件后，调用f.close()来关闭文件并释放系统的资源。文件关闭后，如果尝试再次调用该文件对象，则会抛出异常。

6.3.9 with关键字

with关键字用于Python的上下文管理器机制。为了防止诸如open这一类文件打开方法在操作过程出现异常或错误，或者最后忘了执行close方法，文件非正常关闭等可能导致文件泄露、破坏的问题 。Python提供了with这个上下文管理器机制，保证文件会被正常关闭。在它的管理下，不需要再写close语句。注意缩进。

```
with open('test.txt', 'w') as f:
    f.write('Hello, world!')
with支持同时打开多个文件：
with open('log1') as obj1, open('log2','w') as obj2:
    s=obj1.read()
    obj2.write(s)
```

6.3.10 文件的相关操作


```
修改文件名称
import os
os.rename(filename,newfilename)
删除文件
os.remove(filename)
创建文件夹
os.mkdir(dirname)
获取当前目录
os.getcwd()
改变默认路径
os.chdir("../")
删除文件夹
os.rmdir(dirname)
列出当前目录下的文件
os.listdir()
```

案例1：

制作文件的备份

```
# 思路：
    获取要复制的文件名
    打开这个文件
    新建一个文件
    读取旧文件
```

```
    往新文件写
    关闭两个文件

filename = input("请输入要复制的文件名称:")
f = open(filename,'r')
content = f.read()
position = filename.rfind(".")
newfile = filename[:position]+ '(复件)' + filename[position:]
r = open(newfile,'w')
r.write(content)
f.close()
r.close()
```

案例2：

批量重命名文件名称

```
import os
# 获取重命名的文件夹 名称
dir_name = input("请输入要重命名的文件夹:")
# 获取文件夹中的所有文件名称
filenames = os.listdir(dir_name)
# 切换目录       如果不切换目录要在下面连接
os.chdir(dir_name)
for name in filenames:
    # os.rename(dir_name+'/'+name,dir_name+'/'+"[居然]"+name)
    os.rename(name,"[居然]"+name)
```

第7章 面向对象编程

面向对象编程： `Object Oriented Programming` ，简称 `OOP` ，是一种程序设计方法。

面向对象和面向过程的区别


```
完成自我介绍功能，面向过程完成功能
stu_a = {
    "name":"A"
    "age":18,
    "hometown":"东北"
}
stu_b = {
    "name":"B"
    "age":19,
    "hometown":"山东"
}
stu_c = {
    "name":"C"
    "age":20,
    "hometown":"河北"
}
def stu_info(stu):
    # 自我介绍
    for key,value in stu.items():
        print("key=%s,value=%d"%(key,value))

stu_info(stu_a)
stu_info(stu_b)
stu_info(stu_c)

# 接下来我们用面向对象的思想来完成功能
```

```
stu_a = Student(个人信息)
stu_b = Student(个人信息)
stu_c = Student(个人信息)
stu_a.info()
stu_b.info()
stu_c.info()

# 文件读写
f = open(path,'r')
f.read()
f.close()
```

面向过程：根据业务逻辑从上到下写代码,要面面俱到都要思考到

面向对象：将数据与函数绑定到一起，进行封装。减少重复代码的重写过程,找一个能完成这个功能的哥们来完成

面向对象和面向过程都是解决问题的一种思路而已。

概念及术语

- 类(Class)： 用来描述具有相同属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。其中的对象被称作类的实例。
- 实例： 也称对象。通过类定义的初始化方法，赋予具体的值，成为一个"有血有肉的实体"。
- 实例化： 创建类的实例的过程或操作。
- 实例变量： 定义在实例中的变量，只作用于当前实例。
- 类变量： 类变量是所有实例公有的变量。类变量定义在类中，但在方法体之外。
- 数据成员： 类变量、实例变量、方法、类方法、静态方法和属性等的统称。
- 方法： 类中定义的函数。
- 静态方法： 不需要实例化就可以由类执行的方法
- 类方法： 类方法是将类本身作为对象进行操作的方法。
- 方法重写： 如果从父类继承的方法不能满足子类的需求，可以对父类的方法进行改写，这个过程也称override。
- 封装： 将内部实现包裹起来，对外透明，提供api接口进行调用的机制
- 继承： 即一个派生类（derived class）继承父类（base class）的变量和方法。
- 多态： 根据对象类型的不同以不同的方式进行处理。

7.1-类和实例

7.1.1 类

类是抽象的模板，用来描述具有相同属性和方法的对象的集合，比如 `Animal类` 。 类名通常采用 `驼峰式` 命名方式，尽量让字面意思体现出类的作用。
Python使用 `class` 关键字来定义类，其基本结构如下：

```
class 类名(父类列表):
    pass
```

类名通常采用驼峰式命名方式，尽量让字面意思体现出类的作用。Python采用多继承机制，一个类可以同时继承多个父类（也叫基类、超类），继承的基类有先后顺序，写在类名后的圆括号里。继承的父类列表可以为空，此时的圆括号可以省略。但在Python3中，即使你采用类似 `class Student:pass` 的方法没有显式继承任何父类的定义了一个类，它也默认继承 `object` 类。因为， `object` 是Python3中所有类的基类。

```
class Student:
    room = '102'
    address = 'changsha'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def print_age(self):
        print('%s: %s' % (self.name, self.age))
```


可以通过调用类的实例化方法（有的语言中也叫初始化方法或构造函数）来创建一个类的实例。默认情况下，使用类似 `obj=Student()` 的方式就可以生成一个类的实例。但是，通常每个类的实例都会有自己的实例变量，例如这里的name和age，为了在实例化的时候体现实例的不同，Python提供了一个 `def __init__(self):` 的实例化机制。任何一个类中，名字为 `__init__` 的方法就是类的实例化方法，具有 `__init__` 方法的类在实例化的时候，会自动调用该方法，并传递对应的参数。比如：

```
li = Student("juran", 24)
zhang = Student("tony", 23)
```

7.1.2 实例变量和类变量

实例变量：

实例变量指的是实例本身拥有的变量。每个实例的变量在内存中都不一样。Student类中 `__init__` 方法里的name和age就是两个实例变量。通过实例名加圆点的方式调用实例变量。

类变量：

定义在类中，方法之外的变量，称作类变量。类变量是所有实例公有的变量，每一个实例都可以访问、修改类变量。在Student类中，classroom和address两个变量就是类变量。可以通过类名或者实例名加圆点的方式访问类变量，比如：

```
Student.room
Student.address
li.room
zhang.address
```

在使用实例变量和类变量的时候一定要注意，使用类似zhang.name访问变量的时候，实例会先在自己的实例变量列表里查找是否有这个实例变量，如果没有，那么它就会去类变量列表里找，如果还没有，弹出异常。

7.1.3 类的方法：

Python的类中包含实例方法、静态方法和类方法三种方法。这些方法无论是在代码编排中还是内存中都归属于类，区别在于传入的参数和调用方式不同。在类的内部，使用 `def` 关键字来定义一个方法。

实例方法

类的实例方法由实例调用，至少包含一个self参数，且为第一个参数。执行实例方法时，会自动将调用该方法的实例赋值给self。 `self` 代表的是类的实例，而非类本身。 `self` 不是关键字，而是Python约定成俗的命名，你完全可以取别的名字，但不建议这么做。

```
def print_age(self):
        print('%s: %s' % (self.name, self.age))

# 调用方法
li.print_age()
zhang.print_age()
```

静态方法

静态方法由类调用，无默认参数。将实例方法参数中的self去掉，然后在方法定义上方加上@staticmethod，就成为静态方法。它属于类，和实例无关。建议只使用类名.静态方法的调用方式。（虽然也可以使用实例名.静态方法的方式调用）

```
class Foo:
```

```
    @staticmethod
    def static_method():
        pass

#调用方法
Foo.static_method()
```

类方法

类方法由类调用，采用@classmethod装饰，至少传入一个cls（代指类本身，类似self）参数。执行类方法时，自动将调用该方法的类赋值给cls。建议只使用类名.类方法的调用方式。（虽然也可以使用实例名.类方法的方式调用）

```
class Foo:

    @classmethod
    def class_method(cls):
        pass

Foo.class_method()
```

7.2-封装、继承和多态

封装、继承和多态。

7.2.1 封装

`封装` 是指将 `数据` 与 `具体操作` 的实现代码放在某个对象内部，使这些代码的实现细节不被外界发现，外界只能通过接口使用该对象，而不能通过任何形式修改对象内部实现。

正是由于封装机制，程序在使用某一对象时不需要关心该对象的数据结构细节及实现操作的方法。

使用封装能隐藏对象实现细节，使代码更易维护，同时因为不能直接调用、修改对象内部的私有信息，在一定程度上保证了系统安全性。类通过将函数和变量封装在内部，实现了比函数更高一级的封装。

```
class Student:
    room = '101'
    address = 'changsha'

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def print_age(self):
        print('%s: %s' % (self.name, self.age))


# 以下是错误的用法
# 类将它内部的变量和方法封装起来，阻止外部的直接访问
print(room)
print(adress)
print_age()
```

7.2.2 继承

`继承` 来源于现实世界，一个最简单的例子就是孩子会具有父母的一些特征，即 `每个孩子` 都会 `继承父亲或者母亲` 的某些特征，当然这只是最基本的继承关系，现实世界中还存在着更复杂的继承。

继承机制实现了代码的复用，多个类公用的代码部分可以只在一个类中提供，而其他类只需要继承这个类即可。

继承最大的好处是子类获得了父类的全部变量和方法的同时，又可以根据需要进行修改、拓展。其语法结构如下：

```
class Foo(superA, superB,superC....):
class DerivedClassName(modname.BaseClassName):  ## 当父类定义在另外的模块时
```

Python支持多父类的继承机制，所以需要注意圆括号中基类的顺序，若是基类中有相同的方法名，并且在子类使用时未指定，Python会从左至右搜索基类中是否包含该方法。一旦查找到则直接调用，后面不再继续查找。

```
# 父类定义
class people:

    def __init__(self, name, age, weight):
        self.name = name
        self.age = age
        self.__weight = weight

    def speak(self):
        print("%s 说: 我 %d 岁。" % (self.name, self.age))

# 单继承示例
class student(people):

    def __init__(self, name, age, weight, grade):
        # 调用父类的实例化方法
        people.__init__(self, name, age, weight)
        self.grade = grade

    # 重写父类的speak方法
    def speak(self):
        print("%s 说: 我 %d 岁了, 我在读 %d 年级" % (self.name, self.age, self.gra
de))

s = student('ken', 10, 30, 3)
s.speak()
```


Python3的继承机制

Python3的继承机制不同于Python2。其核心原则是下面两条，请谨记！

- 子类在调用某个方法或变量的时候，首先在自己内部查找，如果没有找到，则开始根据继承机制在父类里查找。
- 根据父类定义中的顺序，以深度优先的方式逐一查找父类！

super()函数

我们都知道，在子类中如果有与父类同名的成员，那就会覆盖掉父类里的成员。那如果你想强制调用父类的成员呢？使用super()函数！这是一个非常重要的函数，最常见的就是通过super调用父类的实例化方法 `__init__` ！

语法： `super(子类名, self).方法名()` ，需要传入的是子类名和self，调用的是父类里的方法，按父类的

方法需要传入参数。

```
class A:
    def __init__(self, name):
        self.name = name
        print("父类的__init__方法被执行了！")
    def show(self):
        print("父类的show方法被执行了！")

class B(A):
    def __init__(self, name, age):
        super(B, self).__init__(name=name)
        self.age = age

    def show(self):
        super(B, self).show()

obj = B("jack", 18)
obj.show()
```

7.2.3 多态

先看下面的代码：


```
class Animal:

    def kind(self):
        print("i am animal")


class Dog(Animal):

    def kind(self):
        print("i am a dog")


class Cat(Animal):

    def kind(self):
        print("i am a cat")


class Pig(Animal):

    def kind(self):
        print("i am a pig")
```

```
# 这个函数接收一个animal参数，并调用它的kind方法
def show_kind(animal):
    animal.kind()


d = Dog()
c = Cat()
p = Pig()

show_kind(d)
show_kind(c)
show_kind(p)

------------------
# 打印结果：

i am a dog
i am a cat
i am a pig
```

`狗、猫、猪` 都继承了动物类，并各自重写了 `kind` 方法。`show_kind()` 函数接收一个 `animal` 参数，并调用它的 `kind` 方法。可以看出，无论我们给 `animal` 传递的是 `狗、猫还是猪` ，都能正确的调用相应的方法，打印对应的信息。这就是多态。

实际上，由于Python的 `动态语言` 特性，传递给函数 `show_kind()` 的参数 `animal` 可以是任何的类型，只要它有一个 `kind()` 的方法即可。

动态语言调用实例方法时不检查类型，只要方法存在，参数正确，就可以调用。

这就是动态语言的 `“鸭子类型”` ，它并不要求严格的继承体系，一个对象只要“看起来像鸭子，走起路来像鸭子”，那它就可以被看做是鸭子。

7.3-成员保护和访问限制

在类的内部，有各种变量和方法。这些数据成员，可以在类的外部通过实例或者类名进行调用，例如：

```
class People:
    title = "人类"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def print_age(self):
        print('%s: %s' % (self.name, self.age))

obj = People("jack", 12)
obj.age = 18
obj.print_age()
print(People.title)
```

上面的调用方式是我们大多数情况下都需要的，但是往往我们也不希望所有的变量和方法能被外部访问，需要针对性地保护某些成员，限制对这些成员的访问。这样的程序才是健壮、可靠的，也符合业务的逻辑。

在类似JAVA的语言中，有private关键字，可以将某些变量和方法设为私有，阻止外部访问。但是，Python没有这个机制，Python利用变量和方法名字的变化，实现这一功能。

在Python中，如果要让内部成员不被外部访问，可以在成员的名字前加上两个下划线 `__` ，这个成员就变成了一个私有成员（private）。私有成员只能在类的内部访问，外部无法访问。

```
class People:
    title = "人类"

    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def print_age(self):
        print('%s: %s' % (self.__name, self.__age))

obj = People("jack", 18)
obj.__name

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

```
  File "F:/Python/pycharm/201705/1.py", line 68, in <module>
    obj.__name
AttributeError: 'People' object has no attribute '__name'
```

那外部如果要对 `__name` 和 `__age` 进行访问和修改呢？在类的内部创建外部可以访问的get和set方法！

```
class People:
    title = "人类"

    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def print_age(self):
        print('%s: %s' % (self.__name, self.__age))

    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    def set_name(self, name):
        self.__name = name

    def set_age(self, age):
        self.__age = age

obj = People("jack", 18)
obj.get_name()
obj.set_name("tom")
```


这样做，不但对数据进行了保护的同时也提供了外部访问的接口，而且在 `get_name` ， `set_name` 这些方法中，可以额外添加对数据进行检测、处理、加工、包裹等等各种操作，作用巨大！

类的成员与下划线总结：

- `_name` 、 `_name_` 、 `_name__` :建议性的私有成员，不要在外部访问。
- `__name` 、 `__name_` :强制的私有成员，但是你依然可以蛮横地在外部危险访问。
- `__name__` :特殊成员，与私有性质无关，例如 `__doc__` 。
- `name_` 、 `name__` :没有任何特殊性，普通的标识符，但最好不要这么起名。

7.4-特殊成员和魔法方法

Python中有大量类似 `__doc__` 这种以双下划线开头和结尾的特殊成员及“魔法方法”，它们有着非常重要的地位和作用，也是Python语言独具特色的语法之一！

```
__init__ :        构造函数，在生成对象时调用
__del__ :         析构函数，释放对象时使用
__repr__ :        打印，转换
__setitem__ :     按照索引赋值
__getitem__:      按照索引获取值
__len__:          获得长度
__cmp__:          比较运算
__call__:         调用
__add__:          加运算
__sub__:          减运算
__mul__:          乘运算
__div__:          除运算
__mod__:          求余运算
__pow__:          幂
```

`__doc__`

说明性文档和信息。Python自建，无需自定义。


```
class Foo:
    """ 描述类信息，可被自动收集 """

    def func(self):
        pass

# 打印类的说明文档
print(Foo.__doc__)
```

`__init__()`

实例化方法，通过类创建实例时，自动触发执行。

```
class Foo:

    def __init__(self, name):
        self.name = name
        self.age = 18

obj = Foo(jack') # 自动执行类中的 __init__ 方法
```

`__module__` 和 `__class__`

`__module__` 表示当前操作的对象在属于哪个模块。

`__class__` 表示当前操作的对象属于哪个类。

这两者也是Python内建，无需自定义。

```
class Foo:
    pass

obj = Foo()
print(obj.__module__)
print(obj.__class__)

------------
运行结果：
__main__
<class '__main__.Foo'>
```

`__del__()`

析构方法，当对象在内存中被释放时，自动触发此方法。

注：此方法一般无须自定义，因为Python自带内存分配和释放机制，除非你需要在释放的时候指定做一些动作。

析构函数的调用是由解释器在进行垃圾回收时自动触发执行的。


```
class Foo:

    def __del__(self):
        print("我被回收了！")


obj = Foo()

del obj
```

`__call__()`

如果为一个类编写了该方法，那么在该类的实例后面加括号，可会调用这个方法。

注：构造方法的执行是由类加括号执行的，即： `对象 = 类名()` ，而对于 `__call__()` 方法，是由对象后加括号触发的，即： `对象()` 或者 `类()()`

```
class Foo:

    def __init__(self):
        pass
```

```
    def __call__(self, *args, **kwargs):

        print('__call__')


obj = Foo()     # 执行 __init__
obj()       # 执行 __call__
```

`__dict__`

列出类或对象中的所有成员！非常重要和有用的一个属性，Python自建，无需用户自己定义。

```
class Province:
    country = 'China'
    def __init__(self, name, count):
        self.name = name
        self.count = count

    def func(self, *args, **kwargs):
        print ('func')

# 获取类的成员
print(Province.__dict__)

# 获取 对象obj1 的成员
obj1 = Province('HeBei',10000)
print(obj1.__dict__)

# 获取 对象obj2 的成员
obj2 = Province('HeNan', 3888)
print(obj2.__dict__)
```


`__str__()`

如果一个类中定义了 `__str__()` 方法，那么在打印对象时，默认输出该方法的返回值。这也是一个非常重要的方法，需要用户自己定义。

下面的类，没有定义 `__str__()` 方法，打印结果是：

```
<__main__.Foo object at 0x000000000210A358>
```

```
class Foo:
    pass

obj = Foo()
print(obj)
```

定义了 `__str__()` 方法后，打印结果是：'jack'。

```
class Foo:

    def __str__(self):
        return 'jack'

obj = Foo()
print(obj)
```

`__getitem__()` 、 `__setitem__()` 、 `__delitem__()`

取值、赋值、删除这“三剑客”的套路，在Python中，我们已经见过很多次了，比如前面的 `@property` 装饰器。

Python中，标识符后面加圆括号，通常代表执行或调用方法的意思。而在标识符后面加中括号[]，通常代表取值的意思。Python设计了 `__getitem__()` 、 `__setitem__()` 、 `__delitem__()` 这三个特殊成员，用于执行与中括号有关的动作。它们分别表示取值、赋值、删除数据。

也就是如下的操作：

```
a = 标识符[]   :     执行__getitem__方法
标识符[] = a   :     执行__setitem__方法
del 标识符[]   :     执行__delitem__方法
```


如果有一个类同时定义了这三个魔法方法，那么这个类的实例的行为看起来就像一个字典一样，如下例所示：

```
class Foo:

    def __getitem__(self, key):
        print('__getitem__',key)

    def __setitem__(self, key, value):
        print('__setitem__',key,value)

    def __delitem__(self, key):
        print('__delitem__',key)


obj = Foo()

result = obj['k1']      # 自动触发执行 __getitem__
obj['k2'] = 'jack'      # 自动触发执行 __setitem__
del obj['k1']             # 自动触发执行 __delitem__
```

`__iter__()`

这是迭代器方法！列表、字典、元组之所以可以进行for循环，是因为其内部定义了 `__iter__()` 这个方法。如果用户想让自定义的类的对象可以被迭代，那么就需要在类中定义这个方法，并且让该方法的返回值是一个可迭代的对象。当在代码中利用for循环遍历对象时，就会调用类的这个 `__iter__()` 方法。

普通的类：

```
class Foo:
    pass


obj = Foo()

for i in obj:
    print(i)

# 报错：TypeError: 'Foo' object is not iterable<br># 原因是Foo对象不可迭代
```

添加一个 `__iter__()` ，但什么都不返回：

```
class Foo:

    def __iter__(self):
        pass

obj = Foo()

for i in obj:
    print(i)

# 报错：TypeError: iter() returned non-iterator of type 'NoneType'

#原因是 __iter__方法没有返回一个可迭代的对象
```

返回一个个迭代对象：

```
class Foo:

    def __init__(self, sq):
        self.sq = sq

    def __iter__(self):
```

```
        return iter(self.sq)

obj = Foo([11,22,33,44])

for i in obj:
    print(i)

# 这下没问题了！
```

最好的方法是使用生成器：

```
class Foo:
    def __init__(self):
        pass

    def __iter__(self):
        yield 1
        yield 2
        yield 3

obj = Foo()
for i in obj:
    print(i)
```


`__len__()`

在Python中，如果你调用内置的len()函数试图获取一个对象的长度，在后台，其实是去调用该对象的 `__len__()` 方法，所以，下面的代码是等价的：

```
>>> len('ABC')
3
>>> 'ABC'.__len__()
3
```

Python的list、dict、str等内置数据类型都实现了该方法，但是你自定义的类要实现len方法需要好好设计。

`__repr__()`

这个方法的作用和 `__str__()` 很像，两者的区别是 `__str__()` 返回用户看到的字符串，而 `__repr__()` 返回程序开发者看到的字符串，也就是说， `__repr__()` 是为调试服务的。通常两者代码一样。

```
class Foo:

    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "this is %s" % self.name

    __repr__ = __str__
```

运算方法

```
__add__:      加运算

__sub__:      减运算

__mul__:      乘运算

__div__:      除运算

__mod__:      求余运算

__pow__:      幂运算
```

这些都是算术运算方法，需要你自己为类设计具体运算代码。有些Python内置数据类型，比如int就带有这些方法。Python支持运算符的重载，也就是重写。

```
class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print (v1 + v2)
```

__author__

`__author__` 代表作者信息！类似的特殊成员还有很多，就不罗列了。

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

"""
a test module
"""
__author__ = "Jack"

def show():
    print(__author__)

show()
```

`__slots__`

Python作为一种动态语言，可以在类定义完成和实例化后，给类或者对象继续添加随意个数或者任意类型的变量或方法，这是动态语言的特性。例如：


```
def print_doc(self):
    print("haha")

class Foo:
    pass

obj1 = Foo()
obj2 = Foo()
# 动态添加实例变量
obj1.name = "jack"
obj2.age = 18
# 动态的给类添加实例方法
Foo.show = print_doc
obj1.show()
obj2.show()
```

但是！如果我想限制实例可以添加的变量怎么办？可以使 `__slots__` 限制实例的变量，比如，只允许Foo的实例添加 `name` 和 `age` 属性。

```
def print_doc(self):
    print("haha")

class Foo:
    __slots__ = ("name", "age")
```

```
    pass


obj1 = Foo()
obj2 = Foo()

# 动态添加实例变量
obj1.name = "jack"
obj2.age = 18
obj1.sex = "male"        # 这一句会弹出错误

# 但是无法限制给类添加方法
Foo.show = print_doc
obj1.show()
obj2.show()
```

由于'sex'不在 `__slots__` 的列表中，所以不能绑定sex属性，试图绑定sex将得到AttributeError的错误。

```
Traceback (most recent call last):
  File "F:/Python/pycharm/201705/1.py", line 14, in <module>
    obj1.sex = "male"
AttributeError: 'Foo' object has no attribute 'sex'
```

需要提醒的是， `__slots__` 定义的属性仅对当前类的实例起作用，对继承了它的子类是不起作用的。想想也是这个道理，如果你继承一个父类，却莫名其妙发现有些变量无法定义，那不是大问题么？如果非要子类也被限制，除非在子类中也定义 `__slots__` ，这样，子类实例允许定义的属性就是自身的 `__slots__` 加上父类的 `__slots__`

作业一

打印皮卡丘

`字符串` 是python中常用的数据类型，它是使用(' 或 ")括起来的任意文本， `''` 或 `""` 本身只是一种表示方式，不是字符串的一部分（这里的引号都是英文引号）

用多行字符串完整打印出下面的皮卡丘吧

```
     へ          /|
   /＼7        ∠_/
  /   |      /   /
 |   Z _,<   /      /`ヽ
 |           ヽ     /    〉
  Y            `   /    /
 ｲ●  、 ●     ⊂⊃〈    /
 ()    へ        |   ＼〈
   >ｰ 、_    ィ    | ／／
  / へ      /  ﾉ<|  ＼＼
   ヽ_ﾉ     (_/    |／／
    7               |／
     >―r￣￣`ｰ―＿＿|
```

作业二

破解谍报密码

现在我的手头有这样一份谍报密码，请你按照提示完成密码的破解，下面，让我们来开始吧！

```
report = [
  'u', 'b', '1', 't', 's', '0', '3', '9', 'k', 'b',
  '4', 'n', ' ', '7', 'b', 'f', 'h', 'r', '3', '6',
  's', 'v', 'f', ' ', '-', 'z', 'e', 'b', '8', '5',
  'ə', 'j', 'u', '2', 'o', 'l', '8', 'b', 'i', 'q',
  'b', '7', '9', 'b', 'm', 'i', 's', '3', 'i', '8',
  '$', 'u', '0', 't', '9', ';', 'q', 'w', ' ', '!',
]

#取出列表 report 中第 12 到 13 个元素（包含 12 和 13），存放在到变量里，比如 secret 变量里；
#列表 secret 尾部追加 report 列表中最中间的一个元素；
#列表 secret 尾部追加 report 列表中元素'b'出现的次数
#列表 secret 尾部追加 report 列表中最后的两个元素；
#将字符串'ʌo'插入到列表索引 secret[3]的位置
#使用print(''.join(secret))将列表 secret 转为字符串并打印到屏幕上。
```

现在请按照上面的提示来破解这封谍报，看看这份谍报写了什么信息？

作业三

用python写个自动选择加油站的小程序

- 输入车子的剩余油量，赋值给gasoline(英文汽油的意思)，为了计算简单，将剩余油量设定为整数
- 已知加满一箱油需要50L
- 在你的附近分别有两家距离一样的加油站，但是他们的油价不一样

加油站	≤20升 的油价	> 20升 的部分油价
1	6	5.5元/L
2	5.5	6元/L

请用python编写一个根据剩余油量，自动选择最优惠加油站的小程序。

作业四

用python做个彩票统计分析工具

假如我们一共有5组连续的彩票开奖号码，他们分别是：

期数	开奖号码
1	'1', '5' , '9' , '3' , '7'
2	'4', '2' , '1', '3' , '6'
3	'2', '3' , '8' , '4' , '9'
4	'9', '3' , '2' , '4' , '5'
5	'5', '3' , '6' , '8' , '1'

我们需要统计一下这5期彩票中，每个号码出现的次数，并将次数作为号码的值，以字典的形式打印出来！

作业五

用python设计一个简单的猜数字游戏

设计要求：

- 随机输入一个0~99之间的整数num，请你的朋友输入一个guess_num
- 如果guess_num 等于 num，打印出提示：恭喜你猜对了
- 如果guess_num 大于 num，打印出提示：不好意思，你猜大了
- 如果guess_num 小于 num，打印出提示：不好意思，你猜小了
- 注意，设置一个变量统计你朋友猜的次数，当次数超过3次时，打印提示：你的机会已经用完了

作业六

存款利息计算器

- 假如你每月存2000到银行，存1年的年利率是2.6%，存3年的年利率为3.8%，存5年的年利率为4.2%
- 这种存款方式属于零存整取，利息计算公式为：

  利息 = 月存金额×累计月积数×月利率

  累计月积数 = （存入次数 + 1）÷2×存入次数

  `假如存3年，那么：利息=2000×累计月积数×（3.8%÷12），  累计月积数=（36+1）÷2×36`

请根据提示，设计一个零存整取的利息计算器。

作业七

奇异博士大战灭霸

灭霸为了减轻宇宙的负担，开始了他的救(mie)世(shi)计划，并最终成功的集齐所有宝石，用无限手套随机杀死了宇宙的一半人。
现在给你一个回到过去的机会，请你帮助奇异博士守住无限宝石，阻止灭霸！**
这一天，灭霸来到地球准备夺取奇异博士手中的时间宝石，在你的帮助下奇异博士能否守住手中的时间宝石呢？
我们先来设置几个条件：

- 奇异博士和灭霸的血量（HP），都为500
- 他们两人都是真正的君子，决定采用回合制来展开攻击，每一次攻击，对方随机掉血（10-50）
- 由于你帮助，奇异博士有15%的几率可以防住灭霸的攻击，伤害为0
- 当有一人血量 ≤ 0 时，结束战斗，对方获胜

让我们用面向对象的编程思维来分析这次对战：

- 1. 奇异博士和灭霸是两个不同的对象
- 2. 他们的HP都是500，这是属性
- 3. 他们的攻击都是随机掉血，这是方法
- 4. 当有一人血量 ≤ 0 时，结束战斗，这也是方法

接下来，让我们开始打造这个小游戏吧