



开课吧2018年官方最新Python3教程

目录

Python基础课件	1.1
Python介绍	1.2
Python简介	1.2.1
Python的历史	1.2.2
Python的优缺点	1.2.3
Python的安装	1.3
安装Python	1.3.1
开发工具pyCharm	1.3.2
Python基础知识	1.4
输入和输出	1.4.1
数据类型和变量	1.4.2
数据类型的转换	1.4.3
标识符和关键字	1.4.4
运算符	1.4.5
注释	1.4.6
判断语句和循环语句	1.4.7
字符串	1.5
字符串的输入和输出	1.5.1
下标和切片	1.5.2
字符串的常用操作	1.5.3
列表	1.6
列表的介绍	1.6.1
列表的遍历	1.6.2
列表的常见操作	1.6.3
列表的嵌套	1.6.4
元组	1.6.5

字典	1.7
字典的介绍	1.7.1
字典的遍历	1.7.2
字典的常见操作	1.7.3
公共方法和内置函数	1.8
公共方法	1.8.1
内置函数	1.8.2
函数	1.9
函数的简介	1.9.1
函数的文档说明	1.9.2
函数的参数一	1.9.3
函数的参数二	1.9.4
递归函数	1.9.5
函数返回值	1.9.6
局部变量和全局变量	1.10
局部变量	1.10.1
全局变量	1.10.2
多函数程序的基本使用流程	1.10.3
Python语法之高级特性	1.11
切片	1.11.1
迭代	1.11.2
列表生成式	1.11.3
生成器	1.11.4
迭代器	1.11.5
函数式编程	1.12
返回函数	1.12.1
闭包	1.12.2
匿名函数	1.12.3
装饰器	1.12.4

偏函数	1.12.5
模块	1.13
模块简介	1.13.1
使用模块	1.13.2
安装第三方模块	1.13.3
面向对象一	1.14
面向对象编程	1.14.1
类和对象	1.14.2
对象的属性方法	1.14.3
init()方法	1.14.4
str()方法	1.14.5
del()方法	1.14.6
面向对象二	1.15
程序中的继承	1.15.1
单继承	1.15.2
多继承	1.15.3
子类重写父类的同名属性和方法	1.15.4
子类调用父类同名属性和方法	1.15.5
多继承	1.15.6
super()的使用	1.15.7
多态	1.15.8
面向对象三	1.16
对象的属性和方法	1.16.1
对象的私有权限	1.16.2
单例模式	1.16.3
面向对象高级编程	1.17
使用slots	1.17.1
使用@property	1.17.2
使用枚举类	1.17.3

错误、测试和调试	1.18
错误处理	1.18.1
调试	1.18.2
单元测试	1.18.3
文档测试	1.18.4
IO编程	1.19
文件读写	1.19.1
StringIO和BytesIO	1.19.2
操作文件和目录	1.19.3
序列化	1.19.4
进程和线程	1.20
多进程	1.20.1
多线程	1.20.2
ThreadLocal	1.20.3
进程VS线程	1.20.4
分布式进程	1.20.5
常用内建模块	1.21
datetime	1.21.1
collections	1.21.2
base64	1.21.3
struct	1.21.4
hashlib	1.21.5
hmac	1.21.6
itertools	1.21.7
contextlib	1.21.8
urllib	1.21.9
XML	1.21.10
HTMLParser	1.21.11
Introduction	1.22

Python简介

学一项新的语言之前,我们要先搞清楚这门语言的特性,及其它应用与发展趋势.

(1) 在介绍python语言之前我们要先了解什么叫做编程语言?

编程语言是用来定义 计算机程序 的形式语言。我们通过编程语言来编写程序代码，再通过语言处理程序执行向计算机发送指令，让计算机完成对应的工作。简而言之,编程语言就是人类指挥计算进行工作的语言,也就是人类与计算机沟通的语言

(2) 那什么是python语言呢?

Python也是编程语言的一种,并且是高级的编程语言 Python语言可能是第一种即简单又功能强大的编程语言。它不仅适合于初学者，也适合于专业 人员使用，更加重要的是，用Python编程是一种愉快的事。本身将帮助你学习这个奇妙的语言，并且向你展示如何即快捷又方便地完成任务——真正意义上“为编程问题提供的完美解决方案!”

Python的历史

Python的作者，**Guido von Rossum**，荷兰人。1982年，Guido从阿姆斯特丹大学获得了数学和计算机硕士学位。然而，尽管他算得上是一位数学家，但他更加享受计算机带来的乐趣。用他的话说，尽管拥有数学和计算机双料资质，他总趋向于做计算机相关的工作，并热衷于做任何和编程相关的活儿。

在那个时候，Guido接触并使用过诸如 `Pascal`、`C`、`Fortran` 等语言。这些语言的基本设计原则是让机器能更快运行。在80年代，虽然IBM和苹果已经掀起了个人电脑浪潮，但这些个人电脑的配置很低。比如早期的Macintosh，只有8MHz的CPU主频和128KB的RAM，一个大的数组就能占满内存。所有的编译器的核心是做优化，以便让程序能够运行。为了增进效率，语言也迫使程序员像计算机一样思考，以便能写出更符合机器口味的程序。在那个时代，程序员恨不得用手榨取计算机每一寸的能力。有人甚至认为C语言的指针是在浪费内存。至于动态类型，内存自动管理，面向对象……别想了，那会让你的电脑陷入瘫痪。

这种编程方式让Guido感到苦恼。Guido知道如何用C语言写出一个功能，但整个编写过程需要耗费大量的时间，即使他已经准确的知道了如何实现。他的另一个选择是 `shell`。Bourne Shell作为UNIX系统的解释器已经长期存在。UNIX的管理员们常常用shell去写一些简单的脚本，以进行一些系统维护的工作，比如定期备份、文件系统管理等等。shell可以像胶水一样，将UNIX下的许多功能连接在一起。许多C语言下上百行的程序，在shell下只用几行就可以完成。然而，shell的本质是调用命令。它并不是一个真正的语言。比如说，shell没有数值型的数据类型，加法运算都很复杂。总之，shell不能全面的调动计算机的功能。

Guido希望有一种语言，这种语言能够像C语言那样，能够全面调用计算机的功能接口，又可以像shell那样，可以轻松的编程。ABC语言让Guido看到希望。ABC是由荷兰的数学和计算机研究所开发的。Guido在该研究所工作，并参与到ABC语言的开发。ABC语言以教学为目的。与当时的大部分语言不同，ABC语言的目标是“让用户感觉更好”。ABC语言希望让语言变得容易阅读，容易使用，容易记忆，容易学习，并以此来激发人们学习编程的兴趣。

一门语言的诞生

1991年，第一个Python编译器诞生。它是用C语言实现的，并能够调用C语言的库文件。从一出生，Python已经具有了：类，函数，异常处理，包含表和词典在内的核心数据类型，以及模块为基础的拓展系统。Python语法很多来自C，但又受到ABC语言的强烈影响。来自ABC语言的一些规定直到今天还富有争议，比如强制缩进。但这些语法规规定让Python容易读。另一方面，Python聪明的选择服从一些惯例，特别是C语言的惯例，比如回归等号赋值。Guido认为，如果“常识”上确立的东西，没有必要过度纠结。Python从一开始就特别在意可拓展性。Python可以在多个层次上拓展。从高层上，你可以直接引入.py文件。在底层，你可以引用C语言的库。Python程序员可以快速的使用Python写.py文件作为拓展模块。但当性能是考虑的重要因素时，Python程序员可以深入底层，写C程序，编译为.so文件引入到Python中使用。Python就好像是使用钢构建房一样，先规定好大的框架。而程序员可以在此框架下相当自由的拓展或更改。最初的Python完全由Guido本人开发。Python得到Guido同事的欢迎。他们迅速的反馈使用意见，并参与到Python的改进。Guido和一些同事构成Python的核心团队。他们将自己大部分的业余时间用于hack Python。随后，Python拓展到研究所之外。Python将许多机器层面上的细节隐藏，交给编译器处理，并凸显出逻辑层面的编程思考。Python程序员可以花更多的时间用于思考程序的逻辑，而不是具体的实现细节。这一特征吸引了广大的程序员。Python开始流行。

伟大的哲学家鲁迅曾说过！



Python的优缺点

万物皆有定律,python也不例外,我们来看看python的优缺点

优点:

1. 简单——Python是一种代表简单主义思想的语言。阅读一个良好的Python程序就感觉像是在读英语一样, 尽管这个英语的要求非常严格! Python的这种伪代码本质是它最大的优点之一。它使你能够专注于解决问题而不是去搞明白语言本身。
2. 易学——就如同你即将看到的一样, Python极其容易上手。前面已经提到了, Python有极其简单的语法。
3. 免费、开源——Python是FLOSS (自由/开放源码软件) 之一。简单地说, 你可以自由地发布这个软件的拷贝、阅读它的源代码、对它做改动、把它的一部分用于新的自由软件中。FLOSS是基于一个团体分享知识的概念。这是为什么Python如此优秀的原因之一——它是由一群希望看到一个更加优秀的Python的人创造并经常改进着的。
4. 层语言——当你用Python语言编写程序的时候, 你无需考虑诸如如何管理你的程序使用的内存一类的底层细节。
5. 可移植性——由于它的开源本质, Python已经被移植在许多平台上 (经过改动使它能够工作在不同平台上)。如果你小心地避免使用依赖于系统的特性, 那么你的所有Python程序无需修改就可以在下述任何平台上面运行。这些平台包括Linux、Windows、FreeBSD、Macintosh、Solaris、OS/2、Amiga、AROS、AS/400、BeOS、OS/390、z/OS、Palm OS、QNX、VMS、Psion、Acom RISC OS、VxWorks、PlayStation、Sharp Zaurus、Windows CE甚至还有PocketPC、Symbian以及Google基于linux开发的Android平台!
6. 解释性——这一点需要一些解释。一个用编译性语言比如C或C++写的程序可以从源文件 (即C或C++语言) 转换到一个你的计算机使用的语言 (二进制代码, 即0和1)。这个过程通过编译器和不同的标记、选项完成。当你运行你的程序的时候, 连接/转载器软件把你的程序从硬盘复制到内存中并且运行。而

Python语言写的程序不需要编译成二进制代码。你可以直接从源代码运行程序。在计算机内部，Python解释器把源代码转换成称为字节码的中间形式，然后再把它翻译成计算机使用的机器语言并运行。事实上，由于你不再需要担心如何编译程序，如何确保连接转载正确的库等等，所有这一切使得使用Python更加简单。由于你只需要把你的Python程序拷贝到另外一台计算机上，它就可以工作了，这也使得你的Python程序更加易于移植。

7. 面向对象———Python既支持面向过程的编程也支持面向对象的编程。在“面向过程”的语言中，程序是由过程或仅仅是可重用代码的函数构建起来的。在“面向对象”的语言中，程序是由数据和功能组合而成的对象构建起来的。与其他主要的语言如C++和Java相比，Python以一种非常强大又简单的方式实现面向对象编程。

8. 可扩展性———如果你需要你的一段关键代码运行得更快或者希望某些算法不公开，你可以把你的部分程序用C或C++编写，然后在你的Python程序中使用它们。

9. 丰富的库———Python标准库确实很庞大。它可以帮助你处理各种工作，包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、CGI、FTP、电子邮件、XML、XML-RPC、HTML、WAV文件、密码系统、GUI（图形用户界面）、Tk和其他与系统有关的操作。记住，只要安装了Python，所有这些功能都是可用的。这被称作Python的“功能齐全”理念。除了标准库以外，还有许多其他高质量的库，如wxPython、Twisted和Python图像库等等。

规范的代码———Python采用强制缩进的方式使得代码具有极佳的可读性。

缺点：

运行速度，有速度要求的话，用C++改写关键部分吧。

国内市场较小（国内以python来做主要开发的，目前只有一些web2.0公司）。但时间推移，目前很多国内软件公司，尤其是游戏公司，也开始规模使用他。

中文资料匮乏（好的python中文资料屈指可数）。托社区的福，有几本优秀的教材已经被翻译了，但入门级教材多，高级内容还是只能看英语版。

构架选择太多（没有像C#这样的官方.net构架，也没有像ruby由于历史较短，构架开发的相对集中。Ruby on Rails 构架开发中小型web程序天下无敌）。不过这也从另一个侧面说明，python比较优秀，吸引的人才多

安装python

我们的教程提供两种方式来下载Python,一种是较为普及的方式,直接Python官网下载对应的Python的版本,Python官网由于是国外网站,下载起来很慢,也可以移步国内镜像文件来下载,在这里我为大家也提供了国内镜像(<https://pan.baidu.com/s/1kU5OCOB#list/path=%2Fpub%2Fpython>).

无论是Windows可以使用国内镜像,mac直接上官网下载

第二种是安装 anaconda 的方式来下载python并部署气搭载环境.我们的课程也是搭配 anaconda 的方式来授课,所以使用第二种方式来配置python环境.

注:

每种安装方式都需要熟练掌握!下面我用图片教程来教大家怎样一步一步把Python安装到你的设备上:

1. 第一种安装方式

(1) 直接安装,我们先直接进入python鼠标上浮到红色箭头所选的Downloads上,就可以看见所有设备所对应的

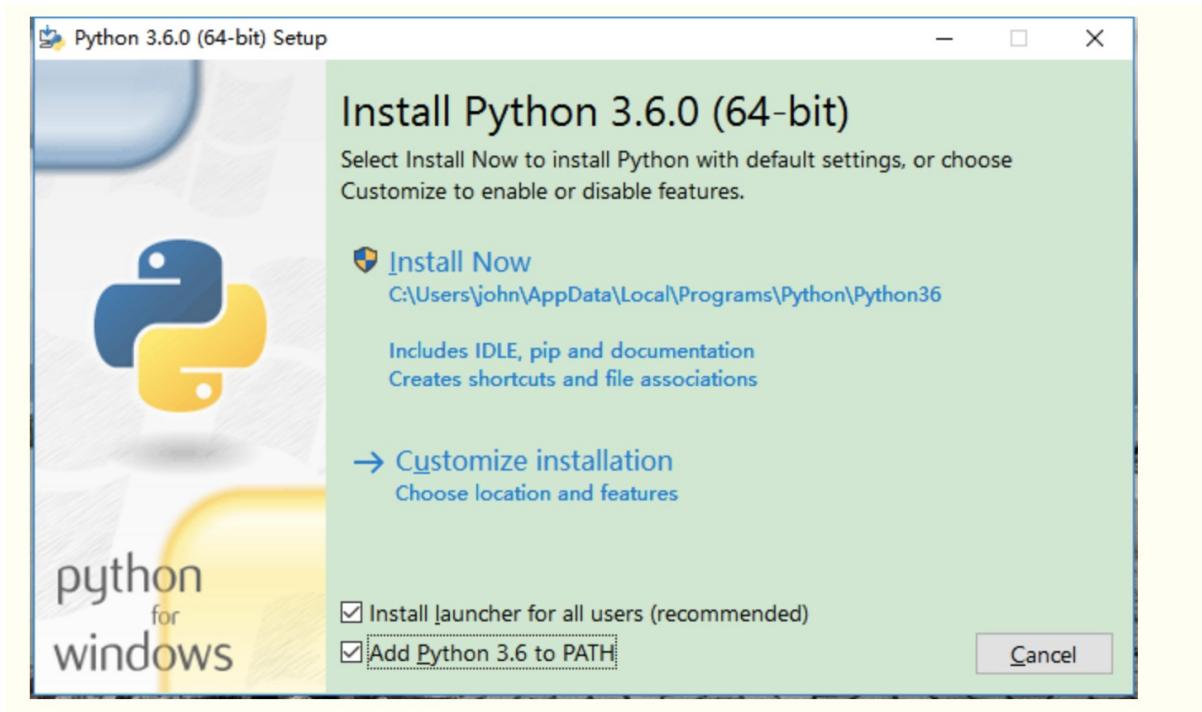


(2) 选中后进入下载界面吗选择对应的python版本号,需注意 x86-64,里面的64对应你的操作系统位数,如果你是64位操作系统,请对应下载

Python Releases for Windows

- [Latest Python 3 Release - Python 3.7.0](#)
- [Latest Python 2 Release - Python 2.7.15](#)
- [Python 3.7.0 - 2018-06-27](#)
 - Download [Windows x86 web-based installer](#)
 - Download [Windows x86 executable installer](#)
 - Download [Windows x86 embeddable zip file](#)
 - Download [Windows x86-64 web-based installer](#)
 - Download [Windows x86-64 executable installer](#)
 - Download [Windows x86-64 embeddable zip file](#)
 - Download [Windows help file](#)

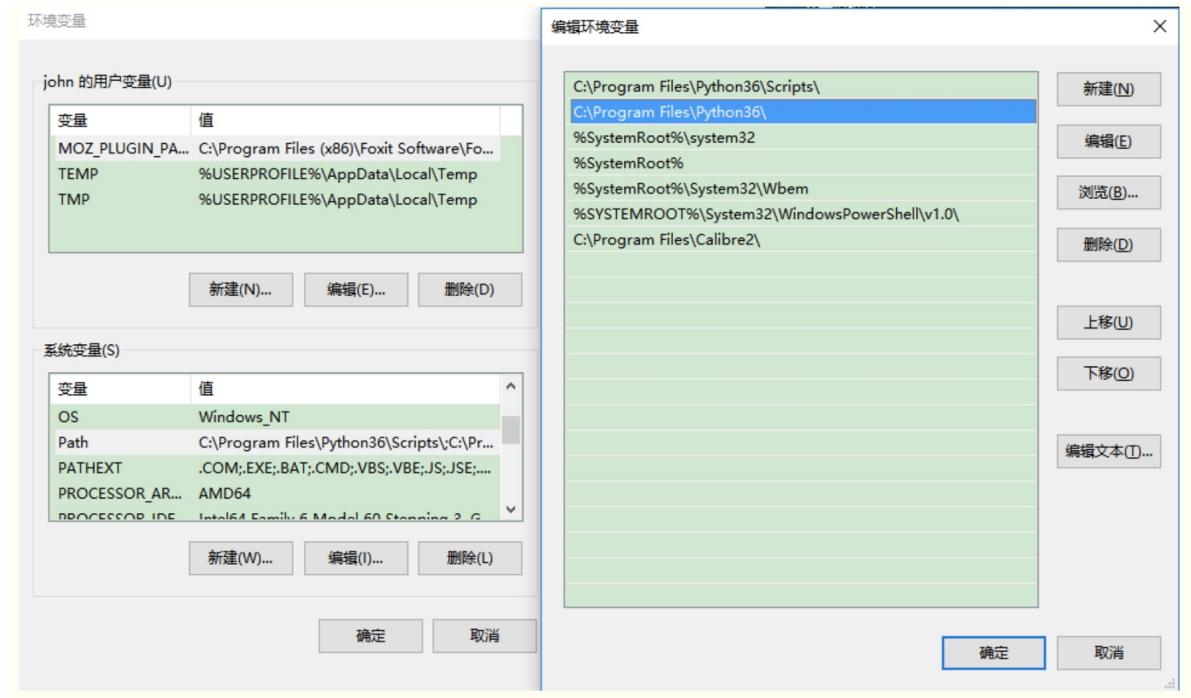
(3) 下载完成后，双击运行安装，一直next,直至安装完成 (安装时可以选择 add python 3.6 to path)



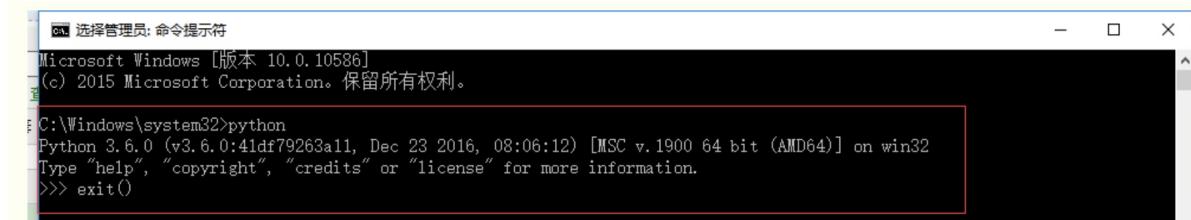
(4) 如果安装的时候选择 `add python to path`, 则会自动完成环境变量配置。

如果忘记勾选, 手动配置环境变量, 在“Path”行, 添加python安装在win下面的路径即可, 本人python安装在 `C:\Program Files\Python36\`, 所以在后面, 添加该路径即可。

windows7下: 路径直接用分号“;”隔开！ windows10下: 新建添加

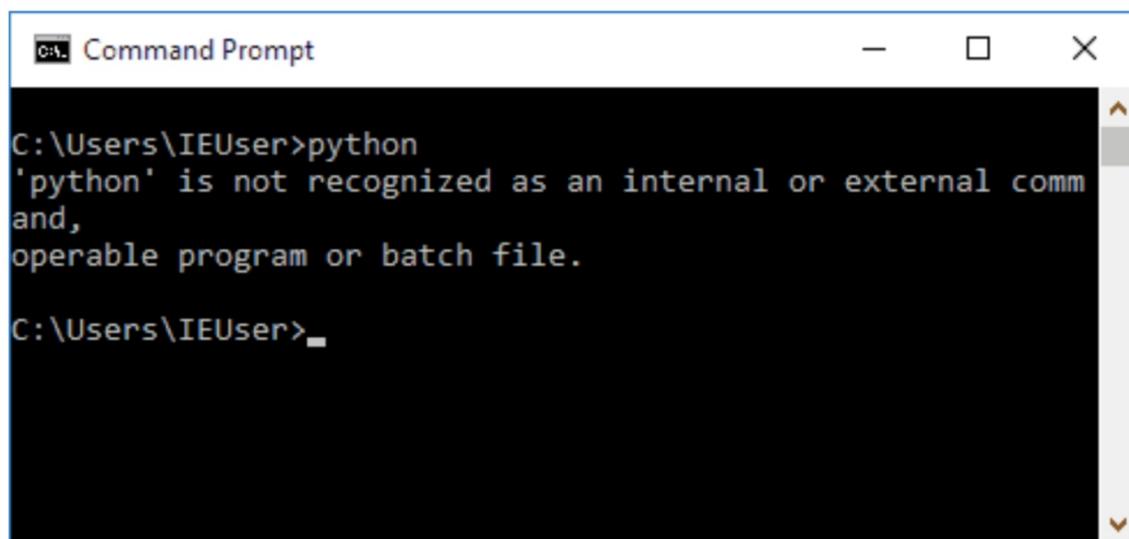


(5) 测试安装是否成功,打开你的cmd(中文翻译过来叫做命令提示窗口),对应的mac端叫做终端,输入python,如果如下图显示则表示安装成功了



也可能出现情况2:

python 不是内部或外部命令,也不是可运行的程序或批处理文件.



那是因为你忘记勾选 `add python to path`, 请手动配置环境变量.

2. 使用anconda方式安装python环境

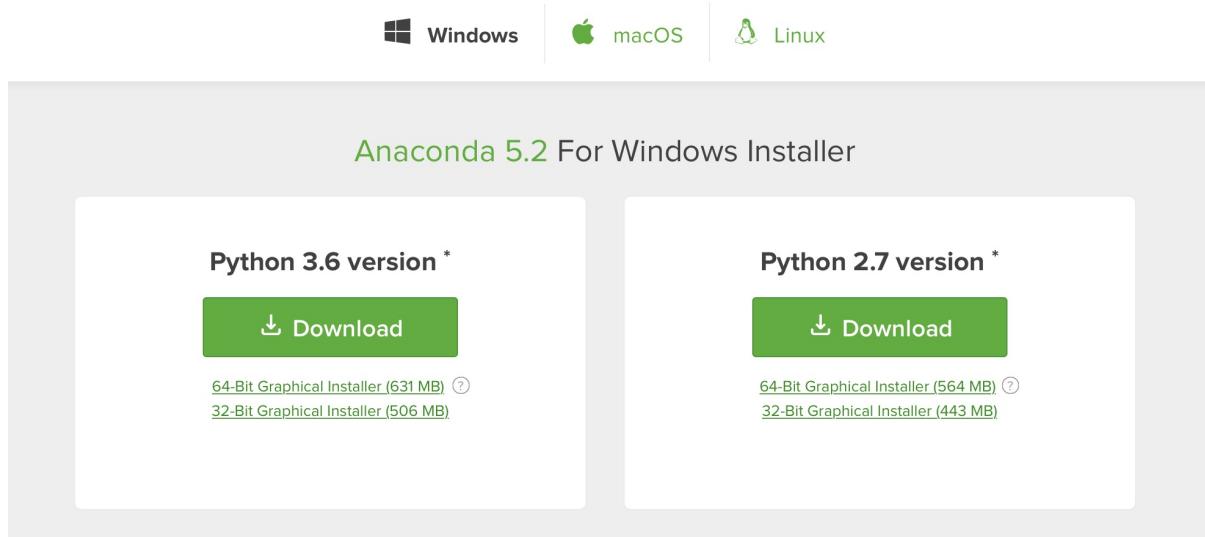
首先我们先介绍一下anaconda, 什么是anaconda, 为什么要使用这种方式安装, 他给我们带来什么样的好处:

Anaconda指的是一个开源的Python发行版本, 其包含了conda、Python等180多个科学包及其依赖项。[1] 因为包含了大量的科学包, Anaconda 的下载文件比较大(约 600 MB), 如果只需要某些包, 或者需要节省带宽或存储空间, 也可以使用**Miniconda**这个较小的发行版(仅包含conda和 Python)。

这是百度百科上面的解释, 随着我们课程的深入讲解, 我会一一对照讲解anconda的使用, 现在我们暂时需要了解的就是, 他可以帮助我们安装Python环境, 十分搭配我们的Python开发语言.

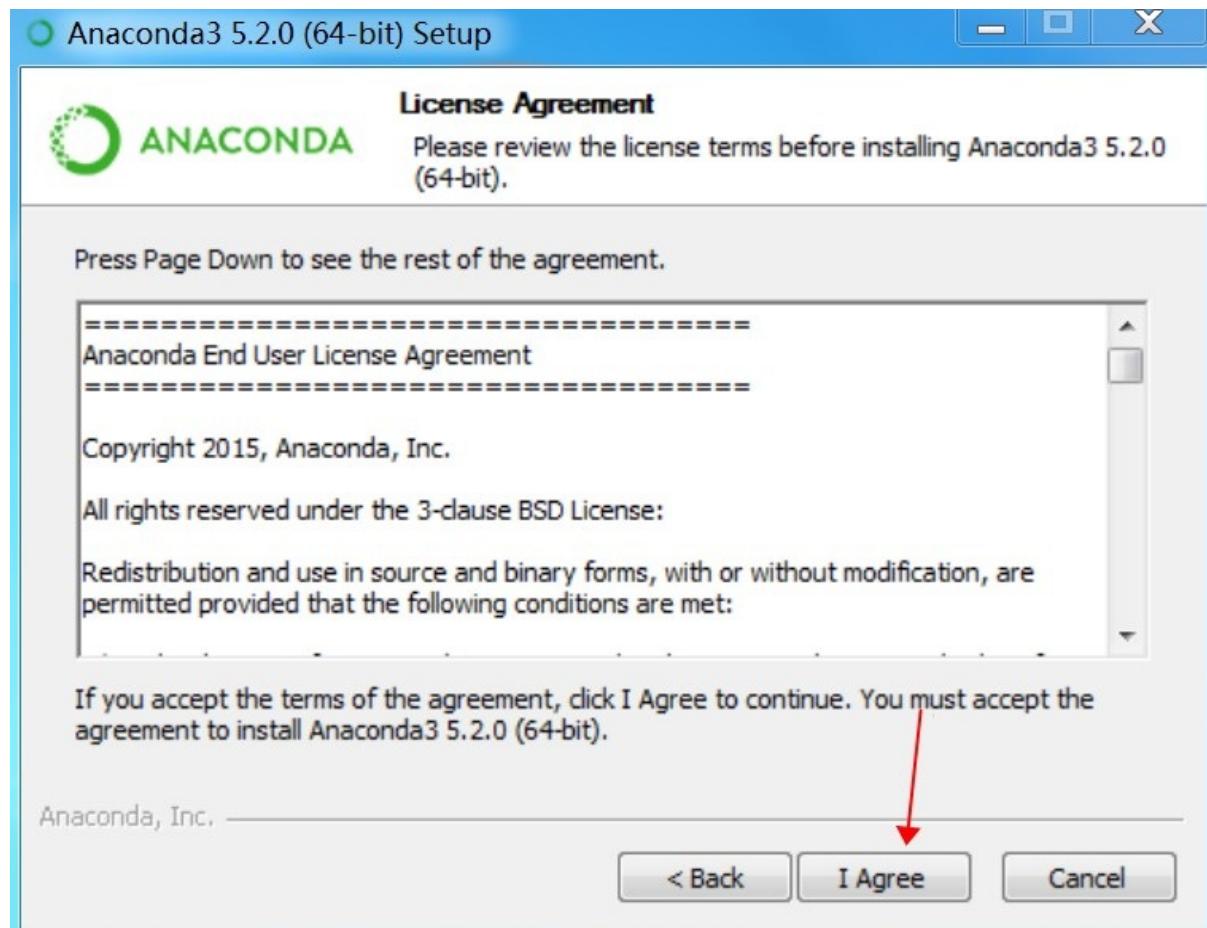
安装anconda

(1) 同样的去anconda官网下载anconda, anconda也提供了两种Python环境供我们选择, 对应我们学习的内容请选择Python3.6版本(官方下载地址<https://www.continuum.io/downloads>)

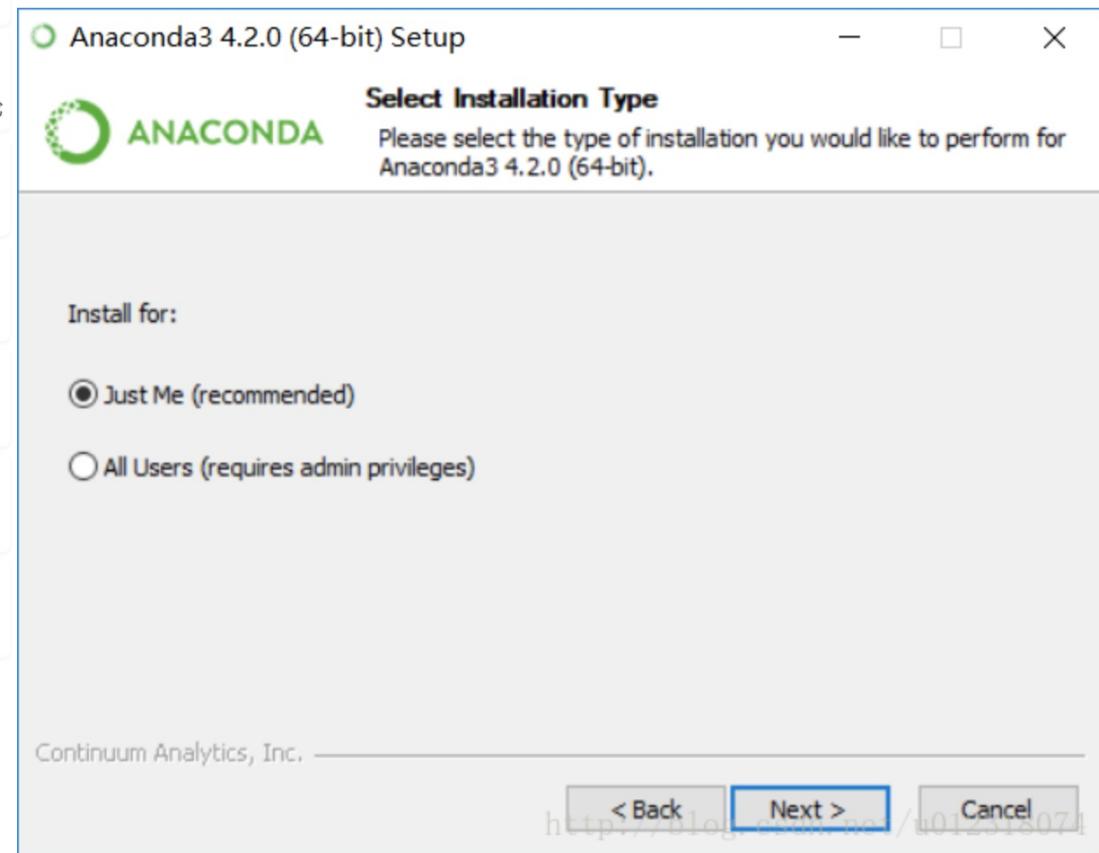


(2) 安装较为简单, 基本都是下一步, 为了避免不必要的麻烦, 最后默认安装路径, 具体安装过程为: 双击安装文件, 启动安装程序

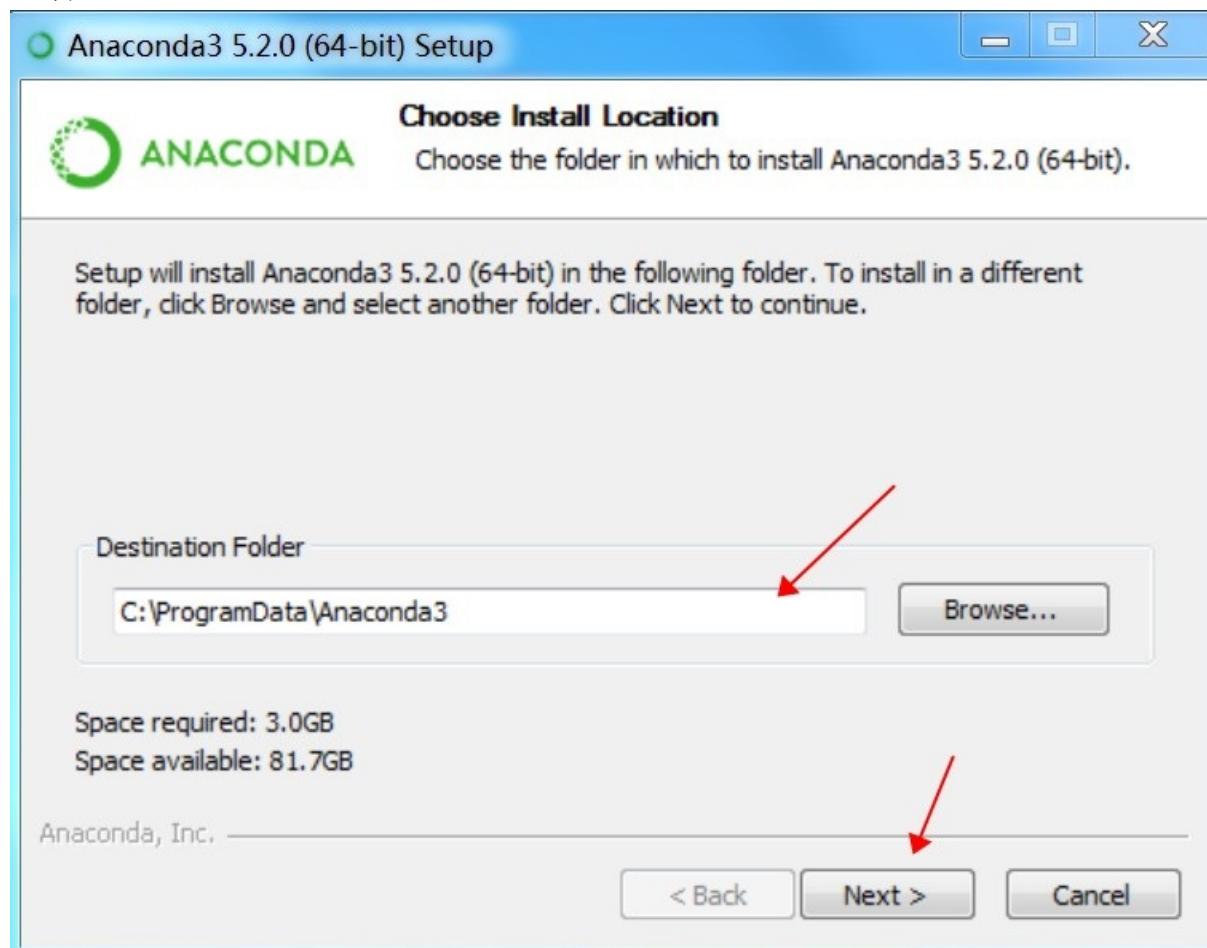




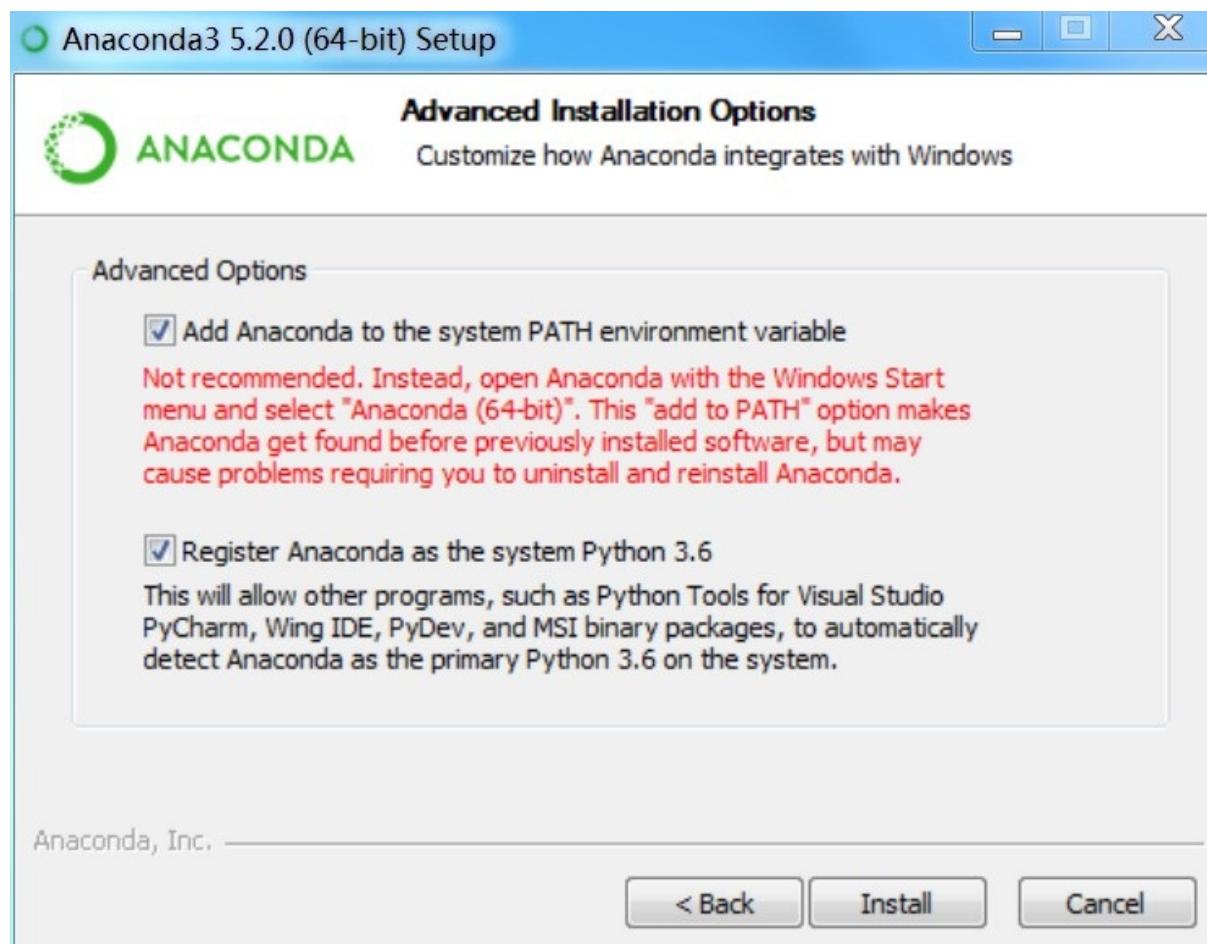
注意!如果系统只有一个用户选择默认的第一个即可,如果有多个用户而且都要用到 Anaconda 则选择第二个选项。



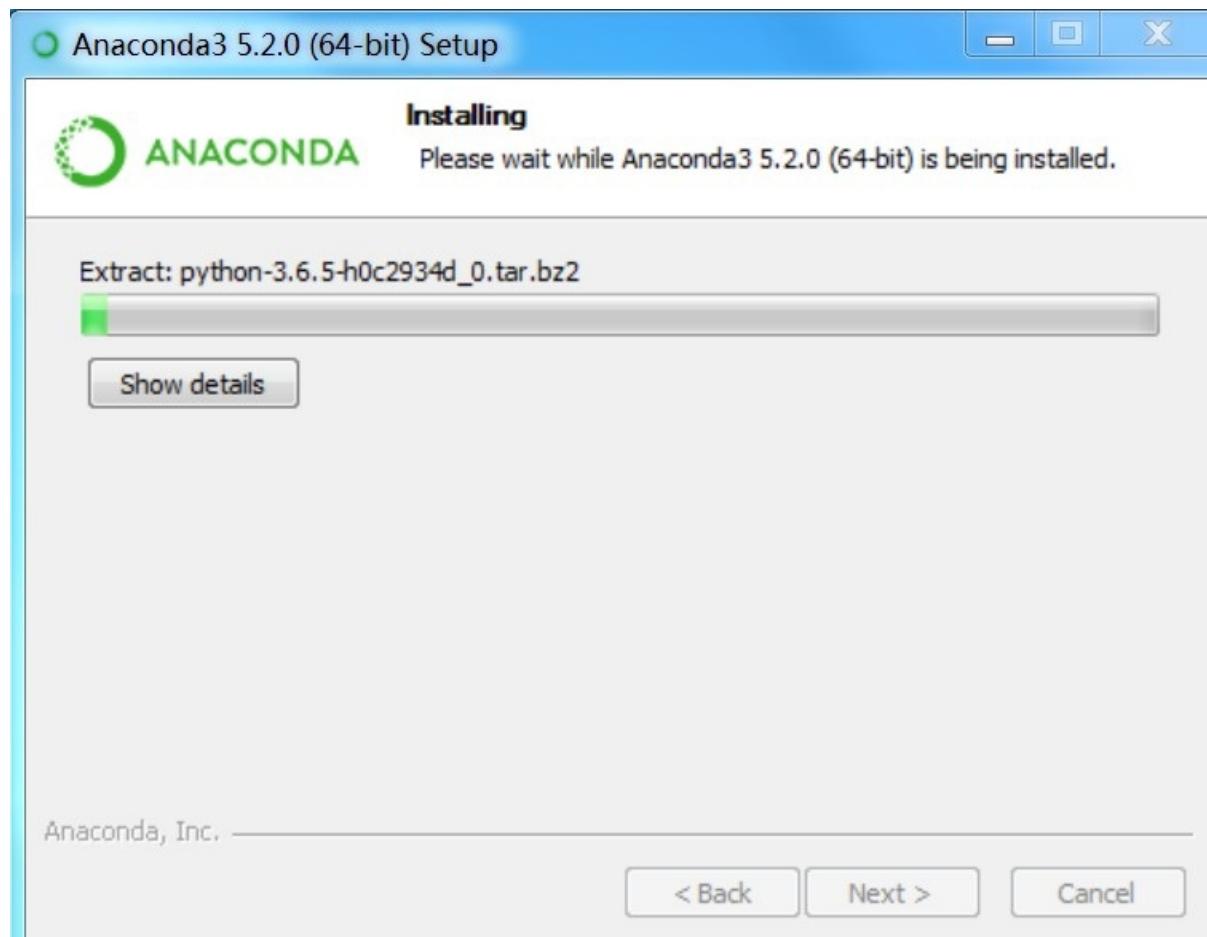
为了避免之后不必要的麻烦，建议默认路径安装即可，需要占用空间大约 1.8 G 左右。



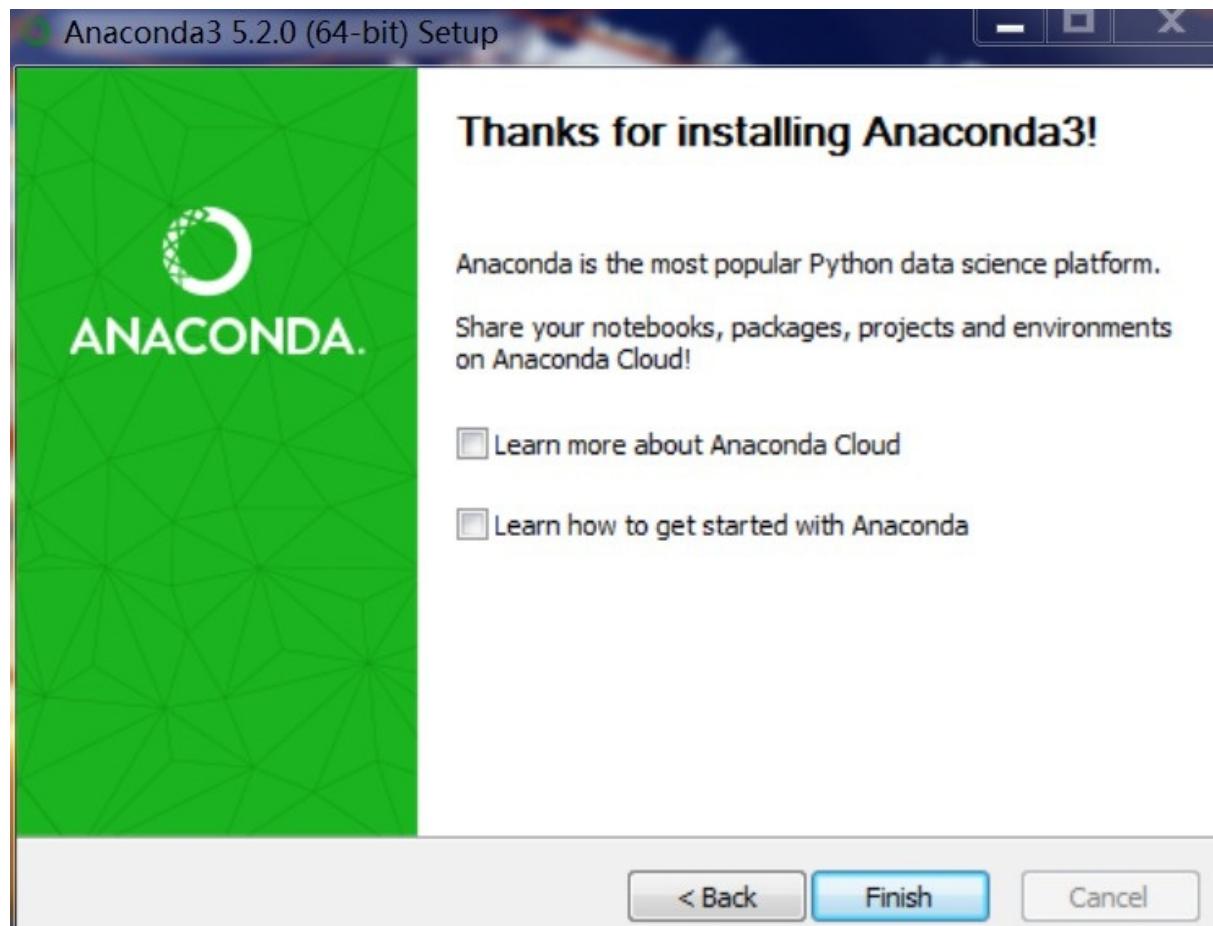
两个默认就好，第一个是加入环境变量，第二个是默认使用 Python 3.5。(最新版本为3.6)



安装需要一段时间，等待安装完成即可。



到这里就安装完成了，可以将“Learn more about Aanaconda Cloud”前的对号去掉，然后点击“Finish”即可。



这样你就安装完毕了anconda,它自带python3.6环境不需要自己再次去官网下载python了.

安装PyCharm开发工具

开始写Python代码之前,我们需要确定我们在哪里去写代码,你可以在记事本里敲打你的代码,市面上也提供了很多Python书写代码的工具,接下来我们就要介绍一下市面上最流行的开发工具——**PyCharm**.

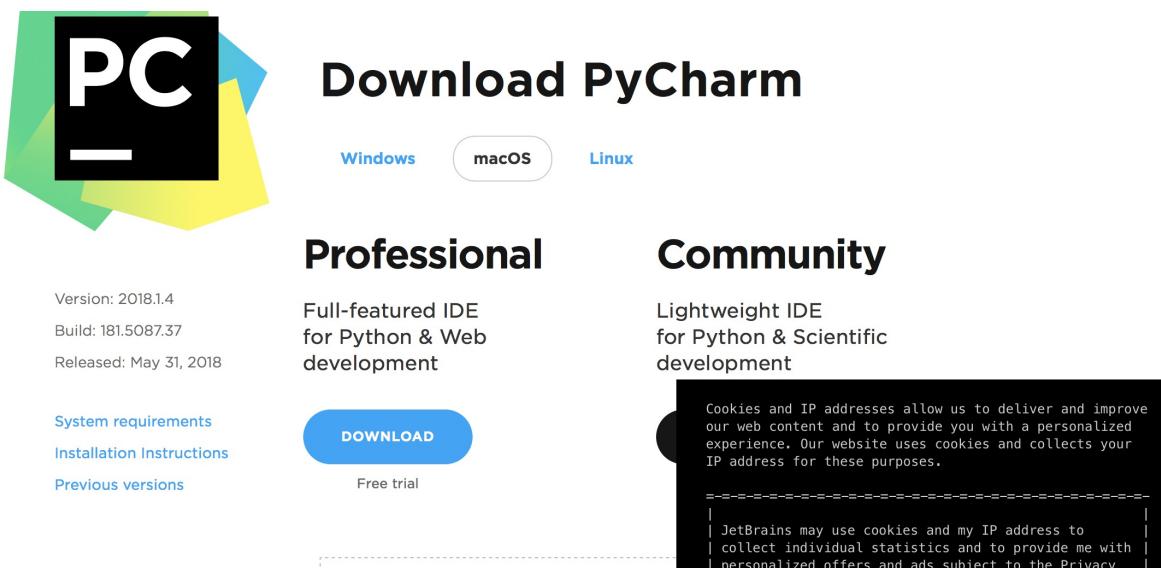
PyCharm是一 Python IDE , 带有一整套可以帮助用户在使用Python语言开发时提高其效率的工具, 比如调试、语法高亮、Project管理、代码跳转、智能提示、自动完成、单元测试、版本控制。此外, 该IDE提供了一些高级功能, 以用于支持Django框架下的专业Web开发。

既然**PyCharm**这么强大,接下来我们来一起下载它:

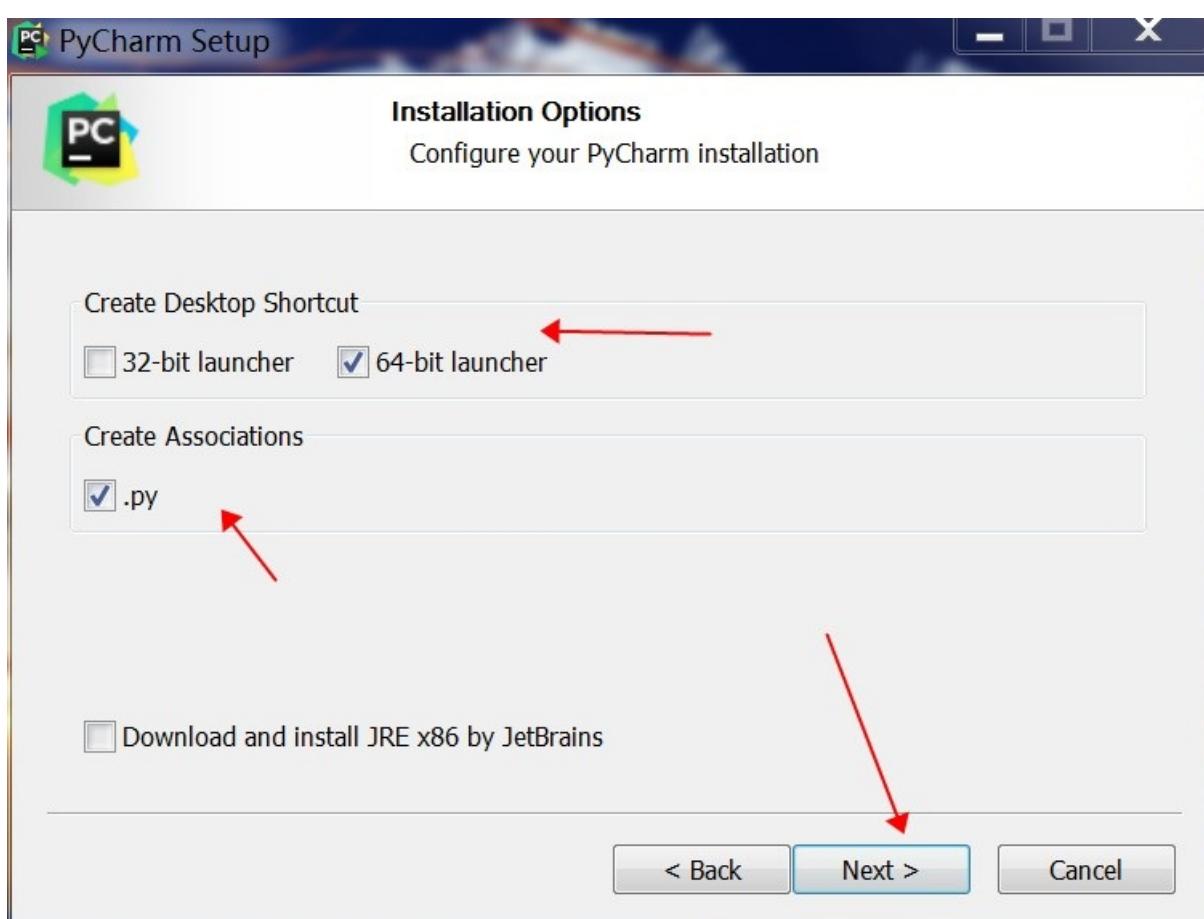
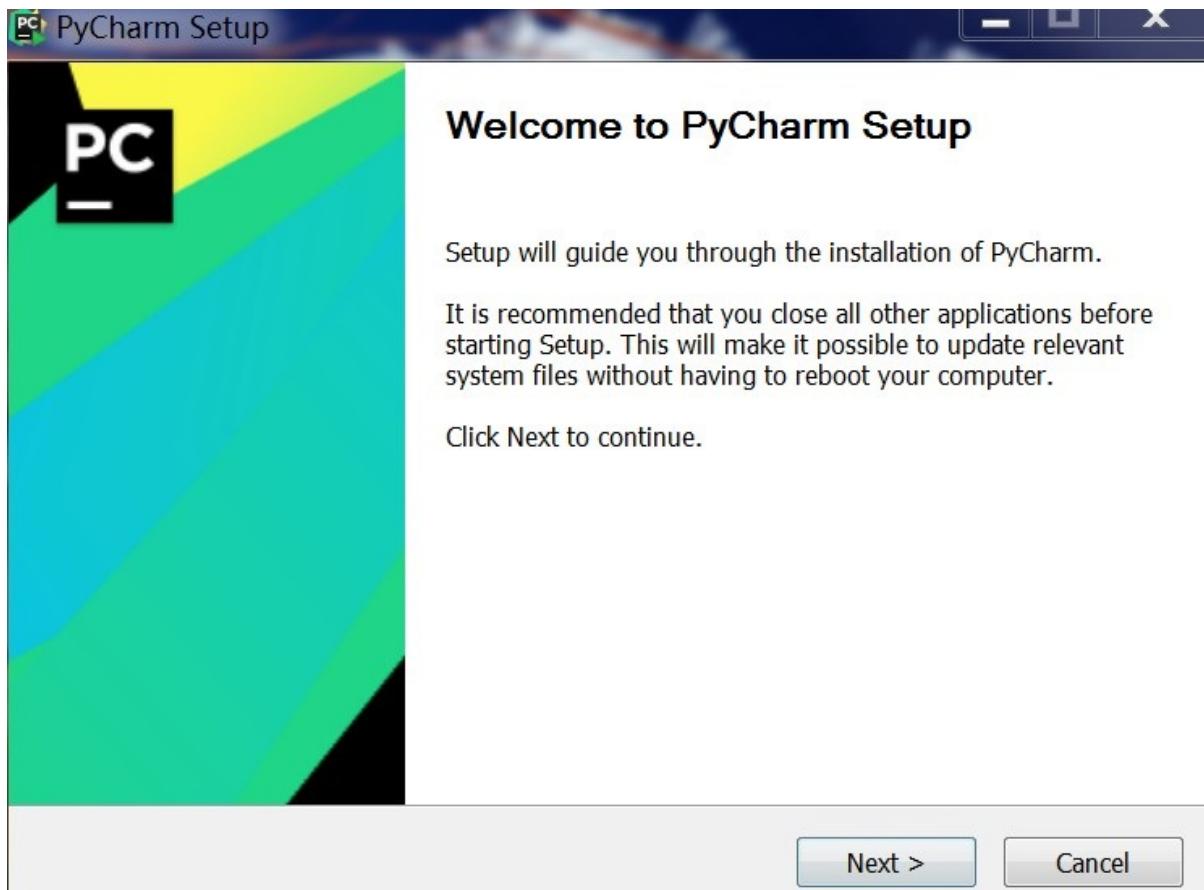
(1) 同样的可以去PyCharm官网下载,也可以移步国内镜像,我在给你们提供python下载的镜像里面,可以查询到对应PyCharm开发工具.

由于大部分开发网站都是国外网站,所以用国内的网络运行下载速度都会很慢,同学们可以进行vpn翻墙等操作来进行国外网站的浏览.

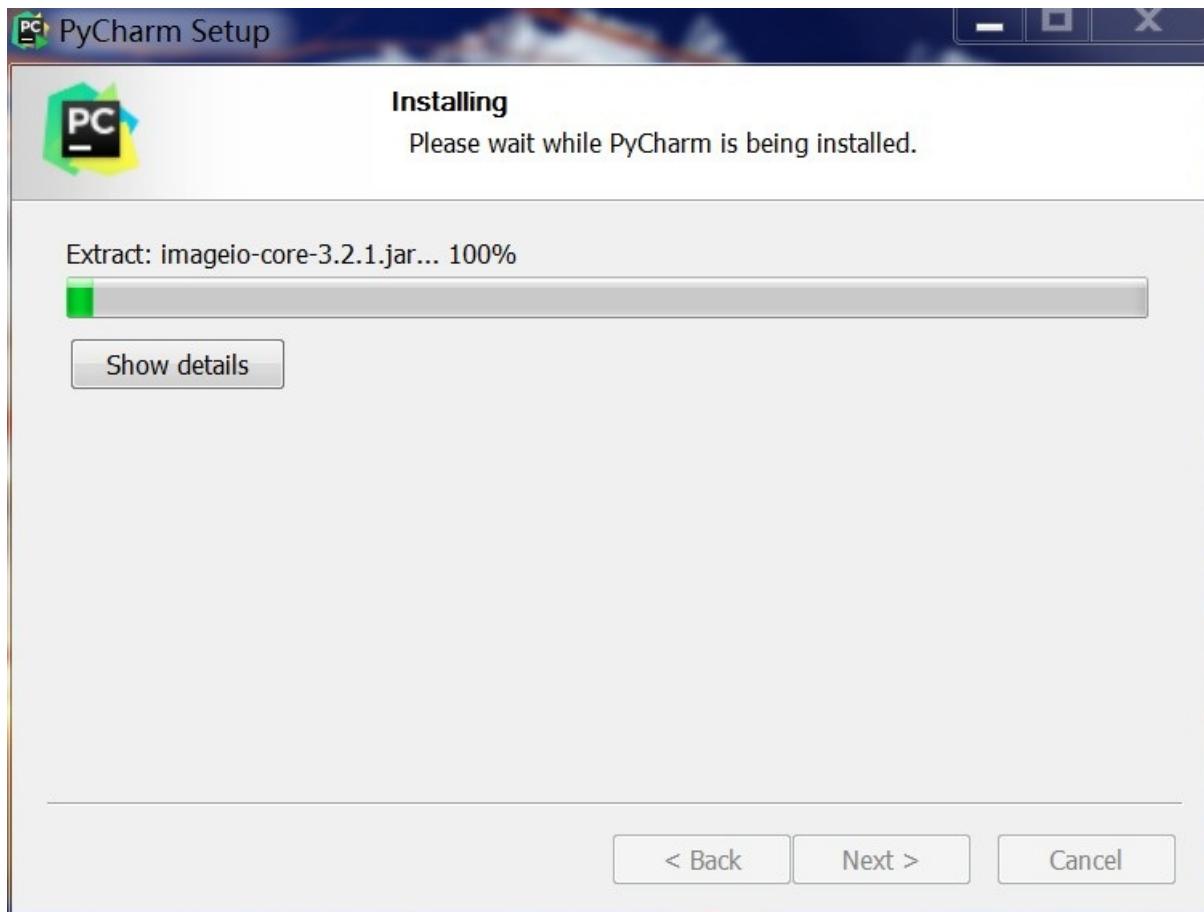
(2) 我们直接选中下载(选择**Professional**版本)



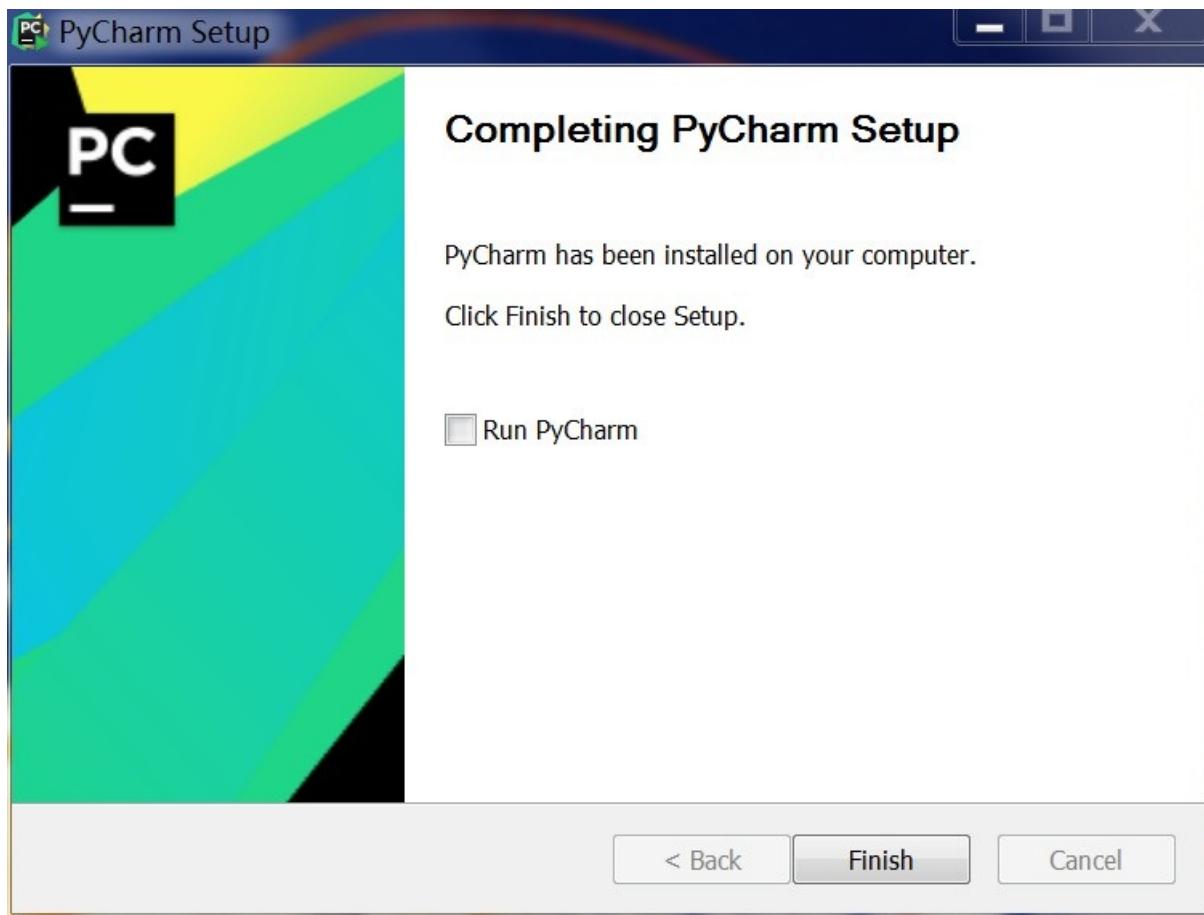
(3) 下载完毕,点击next进入下一步,然后选择对应的操作系统位数,关联py文件



点击Install进行安装：



安装完成后出现下图界面，点级Finish结束安装：



输入和输出

生活中无时无刻其实都存在输入和输出的例子,我们先举例看一下生活中有哪些输入输出的情况,在来对比Python中的输入输出和生活中的有什么相同和不同之处

1. 生活中的输出:

我们经常会去电影院看电影,我们从电影院屏幕中获取影片的内容来观看影片,而影片的放映室经过投影仪""照射""到大屏幕,同样的也可以说是投影仪把影片输出到大荧幕上来供我们观看,这就是生活中最简单的输出方式



2. Python中的输出

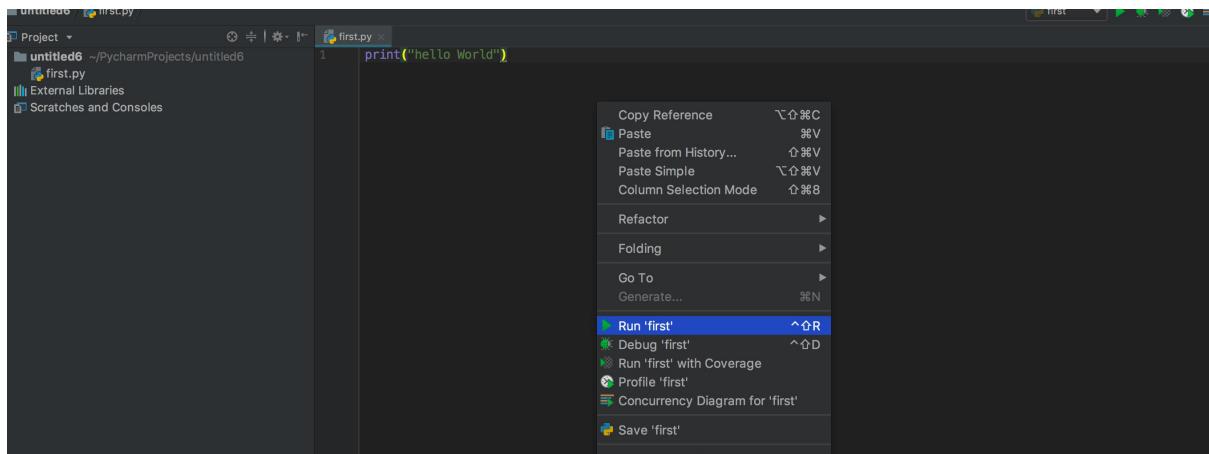
python中的输出和生活中的输出是一样的原理,只不过Python中的输出,特指是在在控制台中输出,或者是将你准备要输出的内容相应的输出到你的设备上,如你在手机上看到的文字,图片,视频等数据,其实本质上也是我们敲打代码输出到手机上的.那我们先来看一下第一种,如何将你想要输出文字输出到控制台.

例如我们将一段文字"Hello World"输出到控制台

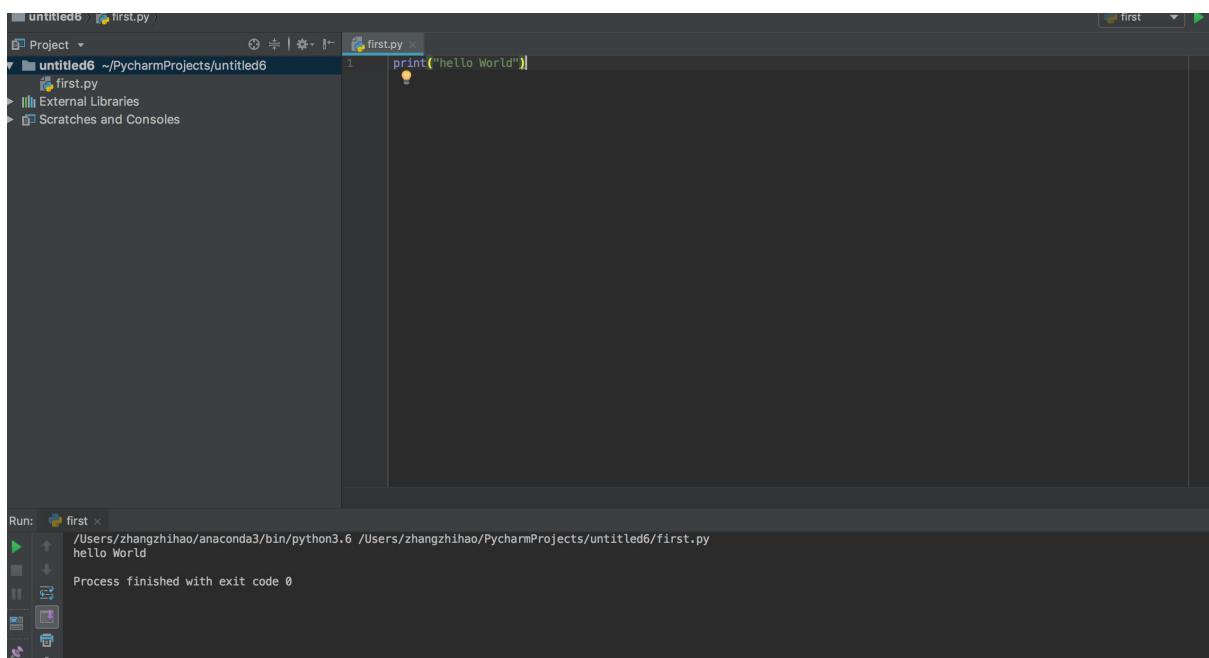
这里面利用的 `print()` 函数进行输出,以后我们会讲到函数的概念,先暂时知道 `print()`可以帮助我们进行输出

```
print('Hello World')
```

右键点击选择 `Run` 和你对应的要运行的文件名字就可以运行刚刚的输出代码



我们在图片最下方看见的输出结果为 `Hello World` 输出的位置就是控制台



我们可以试试输出`100+100`会是什么结果

```
print(100 +100)
```

3. 生活中的输入

生活中的输入无处不在,例如你需要在各种软件中输入的账号密码,去ATM机器取钱也同样需要输入密码.



4. Python中的输入

和输出同理我们也可用到控制台来记录输入结果,同样用到一个函数 `input()`

我们可以利用这段代码在控制台输入,然后在利用刚刚学的 `print()` 函数把你输入的结果在输出出来.

注:当你运行完毕 `name = input()` 代码并将鼠标光标移动到控制台,Python交互式命令就在等你的输入了,你可以输入任意字符,然后按回车完成输入.

```
name = input()
print(name)
```

数据类型和变量

1. 什么是变量?

请回忆初中数学所学的代数基础知识,这里的变量和初中变量概念基本一样.

例如我们现在需要对两条数据求和,就可以利用变量

```
a = 100 // a就是变量,本意就是将100赋值给变量a  
b = 100 //这里注意 '=' 的意思并不是b等于100,而是将100的值赋值给变量b  
result = a + b //定义变量result,将a + b的值赋值给它
```

注: 在Python中, 存储一个数据, 需要一个叫做 变量 的东西,

所谓变量, 可以理解为 菜篮子 , 如果需要存储多个数据, 最简单的方式是有很多个变量, 当然了也可以使用一个

书写形式: 变量名 = 值

变量要先赋值, 后使用

程序就是用来处理数据的, 而变量就是用来存储数据的

2. 数据类型

计算机顾名思义就是做数据计算的及其,因此计算机程序理所当然可以处理各种数值.但是,计算机能处理的远不止数值,还可以处理文本、图形、音频、视频、网页等各种数据,不同的数据,需要定义不同的数据类型.在Python中,能够直接处理的数据类型有以下几种:

(1) 整数(整型)

英文为 int 又名整数

Python 的整数没有大小限制,包括负数,表达方式和数学上的写法一样,例如:

0, 199 , -9999, 等等.

根据不同位数操作系统取值范围也有不同 (32: $-2^{31} \sim 2^{31}-1$ 64: $-2^{63} \sim 2^{63}-1$) \wedge 代表平方, 我们基本不会用到如此大的数值.

(2) 浮点数

浮点数就是小数,之所以称为浮点数是因为按照科学计数法表示的,例如 1.23×100 和 12.3×10^1 是一样的.

浮点数可以用数学法书写,例如:

```
3.14    1.56    8.3333333
```

但是很大或者很小的浮点数就需要用科学计算法例如 1.23×10^1 可以书写 $1.23e1$.

0.000012 , 可以写成 $1.2e-5$.

注: 整数和浮点数在计算机内部的存储方式是不同的, 整数计算永远精准(包括除法), 而浮点数运算有可能会有四舍五入的误差.

(3) 字符串

字符串是以单引号 ' 或双引号 " 或者 """ 括起来的任意文本, 比如 'abc' , "xyz" 等等. 请注意, ' 或 " 本身只是一种表示方式, 不是字符串的一部分, 因此, 字符串 'abc' 只有 a , b , c 这3个字符. 如果 ' 本身也是一个字符, 那就可以用 "" 括起来, 比如 "I'm OK" 包含的字符是 I , ' , m , 空格, O , K 这6个字符。

如果字符串内部既包含 ' 又包含 " 怎么办? 可以用转义字符 \ 来标识, 比如:

```
'I\'m \"OK\"!'
```

表示的字符串内容是:

```
I'm "OK"!
```

(4) BOOL类型

布尔值和布尔代数的表示完全一致，本意是真假的意思，在Python中非真即假，一个布尔值只有 `True`、`False` 两种值，要么是 `True`，要么是 `False`，在 Python中，可以直接用 `True`、`False` 表示布尔值（请注意大小写），也可以通过布尔运算计算出来：

```
>>> True
True
>>> False
False
>>> 3 > 2
True
>>> 3 > 5
False
```

(5) 空值

空值是Python里一个特殊的值，用 `None` 表示。`None` 不能理解为 `0`，因为 `0` 是有意义的，而 `None` 是一个特殊的空值。

此外，Python还提供了列表、字典等多种数据类型，还允许创建自定义数据类型，我们后面会继续讲到。

这就是我们常用的基本数据类型，那么他们之间可以转换么？这个问题是必然的。我们就按来说一下数据类型的转换。

数据类型的转换

1. 字符串--整形

```
a = '123'  
int(a)  
//这样就很简单的讲字符串转换成整数类型了,但需注意,字符串需要兼容数字才可以转换  
a = '说的说的'  
int(a)//这样就属于不兼容  
print(type(a))//type()函数可以帮助我们查看当前数据的类型
```

2. 整型—字符串

```
a=123  
str(a)
```

3. 字符串—浮点型:

```
a = 1.23  
str(a)  
eval(): 将字符串形式的数据, 转换为原本的类型
```

4. 常用的数据类型转换

函数	说明
int(x [,base])	将x转换为一个整数
float(x)	将x转换为一个浮点数
complex(real [,imag])	创建一个复数, real为实部, imag为虚部
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串

eval(str)	用来计算在字符串中的有效Python表达式,并返回一个对象
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
chr(x)	将一个整数转换为一个Unicode字符
ord(x)	将一个字符转换为它的ASCII整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串
bin(x)	将一个整数转换为一个二进制字符串

标识符和关键字

1. 标识符

(1) 标识符定义:

标识符是开发人员在程序中自定义的一些符号和名称，标识符是自己定义的，如变量名、函数名等。并且标识符由字母、下划线和数字组成，且数字不能开头这样就确立一下标识符的命名规则。

(2) Python的命名规则:

首先定义一个变量或者函数名字的时候，尽量定义成一个有意义的名字，尽量做到看一眼就知道是什么意思，这样会提高代码的质量，方便我们自己阅读代码，这就是所谓的见名知意，例如定义年龄变量用age.

驼峰命名法

由于这种命名方法标识名字高低不同，像极了骆驼的驼峰所以有驼峰命名的叫法。

- 小驼峰命名法 (**lower camel case**) : 第一个单词以小写字母开始；第二个单词的首字母大写，例如： myName 、 aDog
- 大驼峰式命名法 (**lower camel case**) : 每一个单字的首字母都采用大写字母，例如： FirstName 、 LastName
- 还有一种命名法是用下划线“_”来连接所有的单词，比如 my_name 、 a_dog

2. 关键字

python一些具有特殊功能的标识符，这就是所谓的关键字

关键字，是python已经使用的了，所以不允许开发者自己定义和关键字相同的名字的标识符。

我们可以通过PyChram敲代码来看一下Python已经使用那些关键字

```
import keyword

kw = keyword.kwlist
print(kw)
```

```
[False, None, True, and, as, assert, break, class, continue, def, del, elif, else, except, finally, for, from, global,
Process finished with exit code 0
```

关键字的学习以及使用，这些关键字不需要大家死记硬背,会在今后的课程中一一学习。

运算符

1. 算数运算符

Python支持以下几种运算符:

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	b / a 输出结果 2
//	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
%	取余	返回除法的余数 $b \% a$ 输出结果 0
**	指数	$a^{**}b$ 为10的20次方, 输出结果 1000000000000000000000000

注意: 混合运算时, 优先级顺序为: `**` 高于 `*` `/` `%` `//` 高于 `+` `-`, 为了避免歧义, 建议使用 `()` 来处理运算符优先级。

并且, 不同类型的数字在进行混合运算时, 整数将会转换成浮点数进行运算。

2. 赋值运算符

运算符	描述	实例
=	赋值运算符	把 = 号右边的结果 赋给 左边的变量, 如 <code>num = 1 + 2 * 3</code> , 结果 <code>num</code> 的值为 7

```
# 单个变量赋值
num = 10
```

```
# 多个变量赋值
```

```
num1, num2, f1, str1 = 100, 200, 3.14, "hello"
```

3. 复合赋值运算符

运算符	描述	实例
<code>+=</code>	加法赋值运算符	<code>c += a</code> 等效于 <code>c = c + a</code>
<code>-=</code>	减法赋值运算符	<code>c -= a</code> 等效于 <code>c = c - a</code>
<code>*=</code>	乘法赋值运算符	<code>c *= a</code> 等效于 <code>c = c * a</code>
<code>/=</code>	除法赋值运算符	<code>c /= a</code> 等效于 <code>c = c / a</code>
<code>%=</code>	取模赋值运算符	<code>c %= a</code> 等效于 <code>c = c % a</code>
<code>**=</code>	幂赋值运算符	<code>c **= a</code> 等效于 <code>c = c ** a</code>
<code>//=</code>	取整除赋值运算符	<code>c //= a</code> 等效于 <code>c = c // a</code>

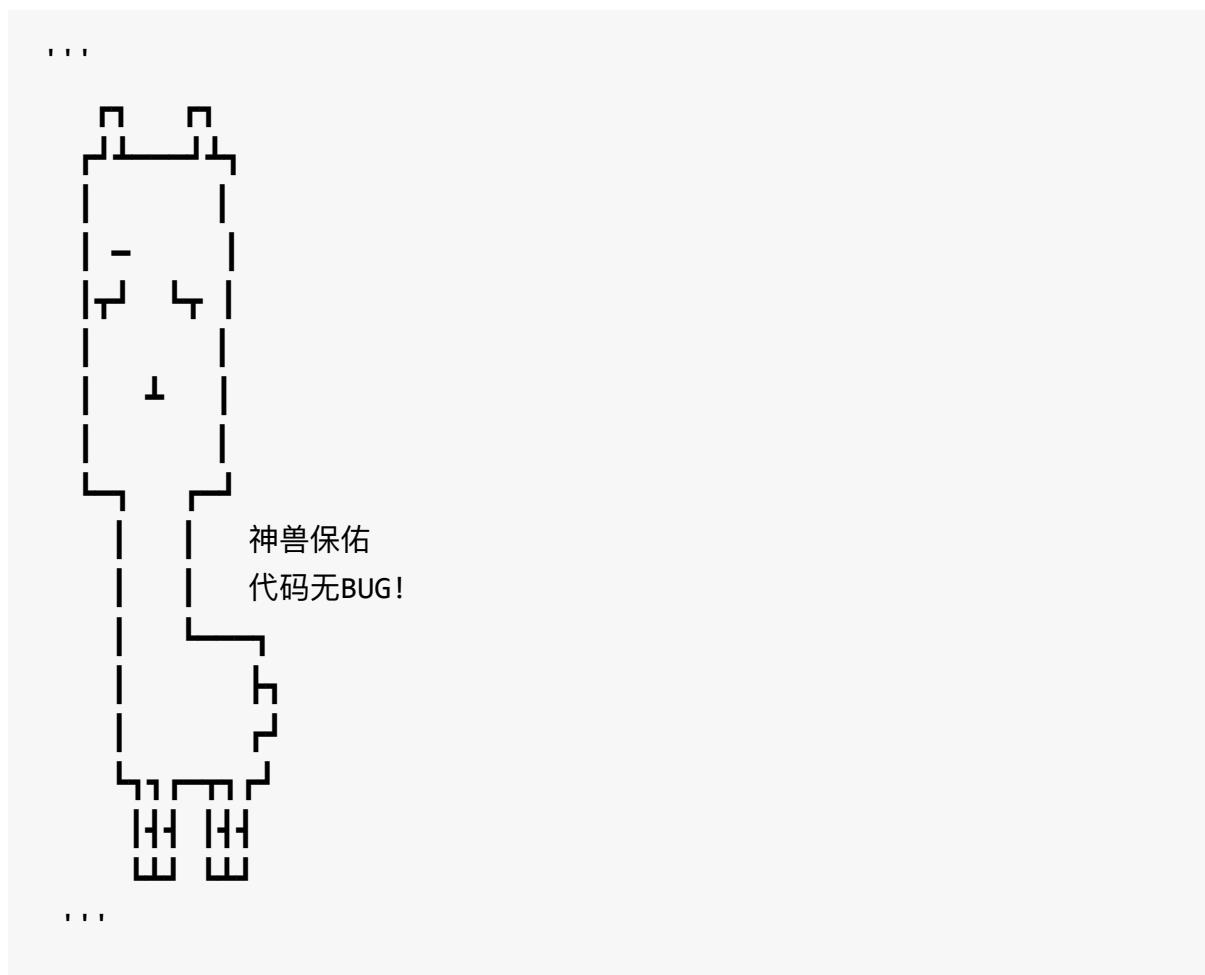
注释

注释又名代码注释,是为了帮助我们提高代码阅读量的存在,例如:

```
#在控制台输出 hello world
printf('hello world')
```

如果我们等程序只有一行代码,可能注释的概念并不重要,但我们的实际项目都是成千上万的代码,所以非常有必要对我们代码进行必要的注释.

并且Python是支持多行注释的,刚刚列举的是单行注释,但有可能一行注释标注不明白下面的代码,我们就可以用到多行注释了



注:单行注释用 # ,多行注释 '...',三个单引号围起来'....'

判断语句和循环语句

1. if语句

计算机之所以能做很多自动化的任务，因为它可以自己做条件判断。

比如，输入用户年龄，根据年龄打印不同的内容，在Python程序中，用 `if` 语句实现：

```
age = 20
if age >= 18:
    print('your age is', age)
    print('adult')
```

根据Python的缩进规则，如果 `if` 语句判断是 `True`，就把缩进的两行`print`语句执行了，否则，什么也不做。

也可以给 `if` 添加一个 `else` 语句，意思是，如果 `if` 判断是 `False`，不要执行 `if` 的内容，去把 `else` 执行了：

```
age = 3
if age >= 18:
    print('your age is', age)
    print('adult')
else:
    print('your age is', age)
    print('teenager')
```

注意不要少写了冒号 `:`。

当然上面的判断是很粗略的，完全可以用 `elif` 做更细致的判断：

```
age = 3
if age >= 18:
    print('adult')
```

```
elif age >= 6:  
    print('teenager')  
else:  
    print('kid')
```

`elif` 是 `else if` 的缩写，完全可以有多个 `elif`，所以 `if` 语句的完整形式就是：

```
if <条件判断1>:  
    <执行1>  
elif <条件判断2>:  
    <执行2>  
elif <条件判断3>:  
    <执行3>  
else:  
    <执行4>
```

`if` 语句执行有个特点，它是从上往下判断，如果在某个判断上是 `True`，把该判断对应的语句执行后，就忽略掉剩下的 `elif` 和 `else`，所以，请测试并解释为什么下面的程序打印的是 `teenager`：

```
age = 20  
if age >= 6:  
    print('teenager')  
elif age >= 18:  
    print('adult')  
else:  
    print('kid')
```

再议`input`

最后看一个有问题的条件判断。很多同学会用 `input()` 读取用户的输入，这样可以自己输入，程序运行得更有意思：

```
birth = input('birth: ')  
if birth < 2000:
```

```
    print('00前')
else:
    print('00后')
```

输入 1982，结果报错：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
```

这是因为 `input()` 返回的数据类型是 `str`，`str` 不能直接和整数比较，必须先把 `str` 转换成整数。Python 提供了 `int()` 函数来完成这件事情：

```
s = input('birth: ')
birth = int(s)
if birth < 2000:
    print('00前')
else:
    print('00后')
```

再次运行，就可以得到正确地结果。但是，如果输入 `abc` 呢？又会得到一个错误信息：

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'abc'
```

原来 `int()` 函数发现一个字符串并不是合法的数字时就会报错，程序就退出了。

拓展：

```
str_1 = "123"
str_2 = "Abc"
str_3 = "123Abc"
```

```
#用isdigit函数判断是否数字
print(str_1.isdigit())
Ture
print(str_2.isdigit())
False
print(str_3.isdigit())
False

#用isalpha判断是否字母
print(str_1.isalpha())
False
print(str_2.isalpha())
Ture
print(str_3.isalpha())
False

#isalnum判断是否数字和字母的组合
print(str_1.isalpha())
False
print(str_2.isalpha())
False
print(str_1.isalpha())
Ture
```

2. 循环

我们来试想这样一种情况,现在让你们在控制台输出100条 `hello world`,本质上,我们写一百条 `print` 函数输出就可以了,但是如果一千条一万条呢.这就要用到循环语句了.

(1) while 循环

while循环语句语法

```
while 条件:
    条件满足时, 做的事情1
    条件满足时, 做的事情2
```

条件满足时，做的事情3

...(省略)...

例如输出100条 helloWorld

```
i = 100
while i <=100
    print('hello world')
    i + 1
```

相对应，在while循环语句里面，每执行一次循环语句，`i` 就会加1，直到 `i` 等于101时不满足

`i<=100` 的条件，循环就结束了

(2) for 循环

for循环和while一样同样可以进行循环，并且是运用最多的循环方式，而且它有一项非常厉害的功能——遍历，在Python中 `for` 循环可以遍历任何序列项目，如字符串，或者今后会学到的列表，例如我们遍历字符串，就特指将字符串的所有字符全部访问一遍

```
names = ['Michael', 'Bob', 'Tracy']
for name in names:
    print(name)
```

执行这段代码，会依次打印 `names` 的每一个元素：

```
Michael
Bob
Tracy
```

所以 `for x in ...` 循环就是把每个元素代入变量 `x`，然后执行缩进块的语句。

再比如我们想计算1-10的整数之和，可以用一个 `sum` 变量做累加：

```
sum = 0
for x in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    sum = sum + x
print(sum)
```

Python还提供了一个`range()`函数,可以配合我们的`for`循环使用,例如:

```
for i in range(5):
    print(i)

#效果等同于 while 循环的:

i = 0
while i < 5:
    print(i)
    i += 1
```

我们在来学习两个关键字 `break` `continue`,这两个关键字是我们循环语句的好帮手

break

在循环中，`break` 语句可以提前退出循环。例如，本来要循环打印1~100的数字：

```
n = 1
while n <= 100:
    print(n)
    n = n + 1
print('END')
```

上面的代码可以打印出1~100。

如果要提前结束循环，可以用 `break` 语句：

```
n = 1
while n <= 100:
    if n > 10: # 当n = 11时, 条件满足, 执行break语句
        break # break语句会结束当前循环
    print(n)
    n = n + 1
print('END')
```

执行上面的代码可以看到, 打印出1~10后, 紧接着打印 END , 程序结束。

可见 break 的作用是提前结束循环。

continue

在循环过程中, 也可以通过 continue 语句, 跳过当前的这次循环, 直接开始下一次循环。

```
n = 0
while n < 10:
    n = n + 1
    print(n)
```

上面的程序可以打印出1~10。但是, 如果我们想只打印奇数, 可以用 continue 语句跳过某些循环:

```
n = 0
while n < 10:
    n = n + 1
    if n % 2 == 0: # 如果n是偶数, 执行continue语句
        continue # continue语句会直接继续下一轮循环, 后续的print()语句
    不会执行
    print(n)
```

执行上面的代码可以看到, 打印的不再是1~10, 而是1, 3, 5, 7, 9。

可见 continue 的作用是提前结束本轮循环, 并直接开始下一轮循环。

字符串的输入和输出

我们在介绍数据类型的时候,简单介绍了一下字符串类型.因为字符串是Python语言中特别重要的概念(不仅仅是Python,在其他语言中也有着举重若轻的位置),我们详细的讲解一下字符串的用法。

我们已经知道了,单引号,双引号,包括三引号包围的字符组,就是字符串,例如

```
str = 'hello'#定义字符串变量
str = "he1o"#定义字符串变量
str = """he1o
he1o""""#定义多行字符串变量
```

字符串的输入和输出

```
name = 'Zhang'
position = '讲师'
address = '北京市'

print('-----')
print("姓名: %s" % name)
print("职位: %s" % position)
print("公司地址: %s" % address)
print('-----')
```

结果:

```
-----
姓名: Zhang
职位: 讲师
公司地址: 北京市
-----
```

之前在学习 `input` 的时候，通过它能够完成从键盘获取数据，然后保存到指定的变量中；

注意：`input` 获取的数据，都以字符串的方式进行保存，即使输入的是数字，那么也是以字符串方式保存

demo:

```
userName = input('请输入用户名：')
print("用户名为：%s" % userName)

password = input('请输入密码：')
print("密码为：%s" % password)
```

结果：（根据输入的不同结果也不同）

```
请输入用户名:123456
用户名为: 123456
请输入密码:112233
密码为: 112233
```

下标和切片

1. 下标索引

下标在Python中的概念就是编号的意思, 字符串 元组 列表 都会经常用到下标的概念,我们可以根据下标找到它们所对应的元素.就好像生活中你要准备去看电影,电影票上的座位号找到对应的位置.

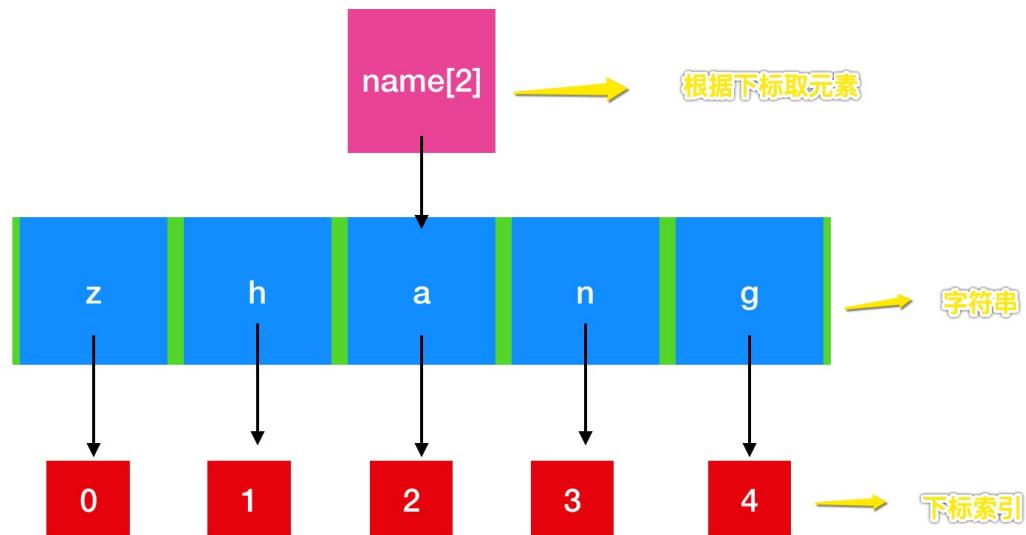
我们现在考虑这样的一个问题,例如我们创建了一个字符串 `name = zhang`,我现在想取到名为 `name` 字符串里面的a字符.如何去取呢?,其实我们可以通过我们讲过的for循环来遍历这个字符串,通过这种方法来取出字符串里的字符,但是 Python 给我们提供了更为简便的方法,我们就可以用下标来取出 a 字符

```
name = 'zhangsan'  
  
print(name[2])
```

运行结果:

```
a
```

这里注意,下标是从零开始的



2. 切片

我们可以利用下表索引取到字符串里面对应的一个元素,但如果想要截取一段元素就要用到切片片.

切片是指对操作的对象截取其中一部分的操作。字符串、列表、元组都支持切片操作。

切片的语法: [起始:结束:步长]

我们以字符串为例讲解。

如果取出一部分, 则可以在中括号 [] 中, 使用:

```
name = 'abcdef'  
  
print(name[0:3]) # 取 下标0~2 的字符
```

运行结果:

abc

```
name = 'abcdef'  
  
print(name[3:5]) # 取 下标为3、4 的字符
```

下标和切片

运行结果：

de

```
name = 'abcdef'  
  
print(name[2:]) # 取 下标为2开始到最后的字符
```

运行结果：

cdef

支持负数

```
name = 'abcdef'  
  
print(name[1:-1]) # 取 下标为1开始 到 最后第2个 之间的字符
```

运行结果：

bcde

```
a = "abcdef"  
a[:3]  
'abc'  
a[::-2]  
'ace'  
a[5:1:2]  
''  
a[1:5:2]  
'bd'  
a[::-2]  
'fdb'  
a[5:1:-2]  
'fd'
```

字符串的常用操作

如有字符串 `mystr = 'hello world itcast and itcastcpp'`，以下是常见的操作

1. find

检测 `str` 是否包含在 `mystr` 中，如果是返回开始的索引值，否则返回-1

```
mystr.find(str, start=0, end=len(mystr))
```

例如：

```
mystr = 'hello world kkb'  
mystr.find("kkb")  
运行结果为:12
```

```
mystr = 'hello world kkb'  
mystr.find("kkb",0,10)#在mystr字符串0-10下标范围查询  
运行结果:-1
```

2. index

跟 `find()` 方法一样，只不过如果 `str` 不在 `mystr` 中会报一个异常。

```
mystr.index(str, start=0, end=len(mystr))
```

例如：

```
mystr = 'hello world kkb'  
mystr.index("ab")
```

运行结果:控制台会直接报错(ValueError: substring not found)

3. count

返回 str 在 start 和 end 之间在 mystr 里面出现的次数

```
mystr.count(str, start=0, end=len(mystr))
```

例如:

```
mystr = 'hello world kkb and kkb'  
mystr.count('kkb')  
运行结果: 2
```

4. replace

把 mystr 中的 str1 替换成 str2 ,如果 count 指定, 则替换不超过 count 次.

```
mystr.replace(str1, str2, mystr.count(str1))
```

5. split

以 str 为分隔符切片 mystr , 如果 maxsplit 有指定值, 则仅分隔 maxsplit 个子字符串

```
mystr.split(str=" ", 2)
```

6、capitalize

把字符串的第一个字符大写

```
mystr.capitalize()
```

7. title

```
a = "hello kkb"  
a.title()  
运行结果  
'Hello Kkb'
```

8. startswith

检查字符串是否是以 `hello` 开头, 是则返回 `True` , 否则返回 `False`

```
mystr.startswith(hello)
```

9. endswith

检查字符串是否以 `obj` 结束, 如果是返回 `True` , 否则返回 `False` .

```
mystr.endswith(obj)
```

10. lower

转换 `mystr` 中所有大写字符为小写

```
mystr.lower()
```

11. upper

转换 `mystr` 中的小写字母为大写

```
mystr.upper()
```

12. ljust

返回一个原字符串左对齐,并使用空格填充至长度 `width` 的新字符串

```
mystr.ljust(width)
```

13. rjust

返回一个原字符串右对齐,并使用空格填充至长度 `width` 的新字符串

```
mystr.rjust(width)
```

14. center

返回一个原字符串居中,并使用空格填充至长度 `width` 的新字符串

```
mystr.center(width)
```

15. lstrip

删除 `mystr` 左边的空白字符

```
mystr.lstrip()
```

16. rstrip

删除 `mystr` 字符串末尾的空白字符

```
mystr.rstrip()
```

17. strip

删除 `mystr` 字符串两端的空白字符

```
a = "\n\t kkb \t\n"
```

```
a.strip()
```

运行结果:

```
'kkb'
```

18. rfind

类似于 `find()` 函数，不过是从右边开始查找。

```
mystr.rfind(str, start=0,end=len(mystr) )
```

19. rindex

类似于 `index()`，不过是从右边开始。

```
mystr.rindex( str, start=0,end=len(mystr))
```

20. partition

把 `mystr` 以 `str` 分割成三部分, `str` 前, `str` 和 `str` 后

```
mystr.partition(str)
```

21. rpartition

类似于 `partition()` 函数, 不过是从右边开始。

```
mystr.rpartition(str)
```

22. splitlines

按照行分隔，返回一个包含各行作为元素的列表

```
mystr.splitlines()
```

23、isalpha

如果 `mystr` 所有字符都是字母则返回 `True`, 否则返回 `False`

```
mystr.isalpha()
```

24. `isdigit`

如果 `mystr` 只包含数字则返回 `True` 否则返回 `False`.

```
mystr.isdigit()
```

25. `isalnum`

如果 `mystr` 所有字符都是字母或数字则返回 `True`, 否则返回 `False`

```
mystr.isalnum()
```

26、`isspace`

如果 `mystr` 中只包含空格, 则返回 `True` , 否则返回 `False` .

```
mystr.isspace()
```

27. `join`

`mystr` 中每个元素后面插入 `str` ,构造出一个新的字符串

```
mystr.join(str)
```

列表介绍

Python内置的一种数据类型是列表： list 。 list 是一种有序的集合，可以随时添加和删除其中的元素。比如，列出班里所有同学的名字，就可以用一个list 表示：

```
classmates = ['Michael', 'Bob', 'Tracy']
结果:
['Michael', 'Bob', 'Tracy']
```

1. 列表的格式

变量A的类型为列表

```
A = ['xiaoWang', 'xiaoZhang', 'xiaoHua']
```

比C语言的数组强大的地方在于列表中的元素可以是不同类型的

```
B = [1, 'a']
```

2. 打印列表

```
namesList = ['xiaoWang', 'xiaoZhang', 'xiaoHua']
print(namesList[0])
print(namesList[1])
print(namesList[2])
结果:
xiaoWang
xiaoZhang
xiaoHua
```

列表的遍历

1. 使用for循环

为了更有效率的输出列表的每个数据，可以使用循环来完成

```
namesList = ['xiaoWang', 'xiaoZhang', 'xiaoHua']
for name in namesList:
    print(name)
```

结果：

```
xiaoWang
xiaoZhang
xiaoHua
```

2. 使用while循环

为了更有效率的输出列表的每个数据，可以使用循环来完成

```
namesList = ['xiaoWang', 'xiaoZhang', 'xiaoHua']

length = len(namesList)

i = 0

while i < length:
    print(namesList[i])
    i+=1
```

结果：

```
xiaoWang
xiaoZhang
xiaoHua
```

列表的常见操作

1. 列表的长度

```
#用len()函数可以获得list元素的个数:  
namesList = ['xiaoWang','xiaoZhang','xiaoHua']  
len(namesList)
```

2. 列表的访问

用索引来访问 `list` 中每一个位置的元素，记得索引是从0开始的：

```
namesList = ['xiaoWang','xiaoZhang','xiaoHua']  
print(namesList[0])  
print(namesList[1])  
print(namesList[2])  
print(namesList[3])  
结果：  
    xiaoWang  
    xiaoZhang  
    xiaoHua  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    IndexError: list index out of range
```

当索引超出了范围时，Python会报一个 `IndexError` 错误，所以，要确保索引不要越界，记得最后一个元素的索引是 `len(classmates) - 1`。

如果要取最后一个元素，除了计算索引位置外，还可以用-1做索引，直接获取最后一个元素：

```
namesList = ['xiaoWang','xiaoZhang','xiaoHua']  
print(namesList[-1])
```

结果：

```
xiaoHua
```

以此类推，可以获取倒数第2个、倒数第3个：

```
namesList = ['xiaoWang', 'xiaoZhang', 'xiaoHua']
print(namesList[-1])
print(namesList[-2])
print(namesList[-3])
结果：
xiaoHua
xiaoZhang
xiaoWang
```

3. 添加元素(`append` , `extend` , `insert`)

通过 `append` 可以向列表添加元素

```
#定义变量A, 默认有3个元素
A = ['xiaoWang', 'xiaoZhang', 'xiaoHua']

print("----添加之前, 列表A的数据----")
for tempName in A:
    print(tempName)

#提示、并添加元素
temp = input('请输入要添加的学生姓名：')
A.append(temp)

print("----添加之后, 列表A的数据----")
for tempName in A:
    print(tempName)
```

通过 `extend` 可以将另一个集合中的元素逐一添加到列表中

```
>>> a = [1, 2]
```

```
>>> b = [3, 4]
>>> a.append(b)
>>> a
[1, 2, [3, 4]]
>>> a.extend(b)
>>> a
[1, 2, [3, 4], 3, 4]
```

`insert(index, object)` 在指定位置 `index` 前插入元素 `object`

```
>>> a = [0, 1, 2]
>>> a.insert(1, 3)
>>> a
[0, 3, 1, 2]
```

4. 修改元素

修改元素的时候，要通过下标来确定要修改的是哪个元素，然后才能进行修改

```
#定义变量A, 默认有3个元素
A = ['xiaoWang','xiaoZhang','xiaoHua']

print("----修改之前, 列表A的数据----")
for tempName in A:
    print(tempName)

#修改元素
A[1] = 'xiaoLu'

print("----修改之后, 列表A的数据----")
for tempName in A:
    print(tempName)
```

结果：

```
----修改之前, 列表A的数据----
xiaoWang
xiaoZhang
```

```
xiaoHua
-----修改之后，列表A的数据-----
xiaoWang
xiaoLu
xiaoHua
```

5. 查找元素

所谓的查找，就是看看指定的元素是否存在。

python中查找的常用方法为：

- `in` (存在) ,如果存在那么结果为 `true` , 否则为 `false`
- `not in` (不存在) , 如果不存在那么结果为 `true` , 否则 `false`

```
#待查找的列表
nameList = ['xiaoWang','xiaoZhang','xiaoHua']

#获取用户要查找的名字
findName = input('请输入要查找的姓名：')

#查找是否存在
if findName in nameList:
    print('在字典中找到了相同的名字')
else:
    print('没有找到')
```

`index` 和 `count` 与字符串中的用法相同

```
>>> a = ['a', 'b', 'c', 'a', 'b']
>>> a.index('a', 1, 3) # 注意是左闭右开区间
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'a' is not in list
>>> a.index('a', 1, 4)
3
>>> a.count('b')
```

```
2
>>> a.count('d')
0
```

6. 删除元素

列表元素的常用删除方法有：

- del：根据下标进行删除
- pop：删除最后一个元素
- remove：根据元素的值进行删除

(1) del

```
list1 = ['a', 'b', 'c', 'd', 'e', 'f']

print('-----删除之前-----')
for tempName in list1:
    print(tempName)

del movieName[2]

print('-----删除之后-----')
for tempName in list1:
    print(tempName)
```

结果：

```
-----删除之前-----
a
b
c
d
e
f
-----删除之后-----
a
b
d
```

```
e  
f
```

(2) pop

```
list2 = ['a', 'b', 'c', 'd', 'e', 'f']

print('-----删除之前-----')
for tempName in list2:
    print(tempName)

movieName.pop()

print('-----删除之后-----')
for tempName in list2:
    print(tempName)
```

结果：

```
-----删除之前-----
a
b
c
d
e
f
-----删除之后-----
a
b
c
d
e
```

(3) remove

```
list3 = ['a', 'b', 'c', 'd', 'e', 'f']

print('-----删除之前-----')
```

```
for tempName in list3:  
    print(tempName)  
  
movieName.remove('e')  
  
print('-----删除之后-----')  
for tempName in list3:  
    print(tempName)
```

结果：

```
-----删除之前-----  
a  
b  
c  
d  
e  
f  
-----删除之后-----  
a  
b  
c  
d  
f
```

(4) 排序

`sort` 方法是将 `list` 按特定顺序重新排列，默认为由小到大，参数 `reverse=True` 可改为倒序，由大到小。

`reverse` 方法是将 `list` 逆置。

```
>>> a = [1, 4, 2, 3]  
>>> a  
[1, 4, 2, 3]  
>>> a.reverse()  
>>> a  
[3, 2, 4, 1]  
>>> a.sort()
```

```
>>> a
[1, 2, 3, 4]
>>> a.sort(reverse=True)
>>> a
[4, 3, 2, 1]
```

列表的嵌套

类似 while 循环的嵌套，列表也是支持嵌套的一个列表中的元素又是一个列表，那么这就是列表的嵌套

```
schoolNames = [[ '北京大学', '清华大学'],
                [ '南开大学', '天津大学', '天津师范大学'],
                [ '山东大学', '中国海洋大学']]
```

例如：有3个班级，现在有8位学生等待分配，请编写程序，完成随机的分配

```
#encoding=utf-8

import random

# 定义一个列表用来保存3个班级
class = [[],[],[]]

# 定义一个列表用来存储8个学生的名字
names = [ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']

i = 0
for name in names:
    index = random.randint(0,2)
    offices[index].append(name)

i = 1
for tempNames in class:
    print('班级%d的人数为:%d'%(i,len(tempNames)))
    i+=1
    for name in tempNames:
        print("%s"%name,end=' ')
    print("\n")
    print("-"*20)
```

元组 (tuple)

另一种有序列表叫元组： tuple 。 tuple 和 list 非常类似，但是 tuple 一旦初始化就不能修改，比如同样是列出同学的名字：

```
>>> classmates = ('Michael', 'Bob', 'Tracy')
```

现在， classmates 这个 tuple 不能变了，它也没有 append() ， insert() 这样的方法。其他获取元素的方法和list是一样的，你可以正常地使用 classmates[0] ， classmates[-1] ，但不能赋值成另外的元素。

不可变的tuple有什么意义？因为tuple不可变，所以代码更安全。如果可能，能用 tuple 代替 list 就尽量用 tuple 。

如果要定义一个空的 tuple ，可以写成 () :

```
>>> t = ()  
>>> t  
()
```

但是，要定义一个只有1个元素的 tuple ，如果你这么定义：

```
>>> t = (1)  
>>> t  
1
```

定义的不是tuple，是1这个数！这是因为括号()既可以表示tuple，又可以表示数学公式中的小括号，这就产生了歧义，因此，Python规定，这种情况下，按小括号进行计算，计算结果自然是1。

所以，只有1个元素的tuple定义时必须加一个逗号，来消除歧义：

```
>>> t = (1,)
```

```
>>> t  
(1,)
```

Python在显示只有1个元素的 `tuple` 时，也会加一个逗号，以免你误解成数学计算意义上的括号。

最后来看一个“可变的”tuple：

```
>>> t = ('a', 'b', ['A', 'B'])  
>>> t[2][0] = 'X'  
>>> t[2][1] = 'Y'  
>>> t  
('a', 'b', ['X', 'Y'])
```

字典的介绍

Python内置了字典：`dict` 的支持，`dict` 全称 `dictionary`，在其他语言中也称为 `map`，使用键-值（key-value）存储，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用 `list` 实现，需要两个 `list`：

```
names = ['Michael', 'Bob', 'Tracy']
scores = [95, 75, 85]
```

给定一个名字，要查找对应的成绩，就先要在 `names` 中找到对应的位置，再从 `scores` 取出对应的成绩，`list` 越长，耗时越长。

如果用 `dict` 实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。用Python写一个 `dict` 如下：

```
>>> d = {'Michael': 95, 'Bob': 75, 'Tracy': 85}
>>> d['Michael']
95
```

由于一个 `key` 只能对应一个 `value`，所以，多次对一个 `key` 放入 `value`，后面的值会把前面的值冲掉：

```
>>> d['Jack'] = 90
>>> d['Jack']
90
>>> d['Jack'] = 88
>>> d['Jack']
88
```

如果 `key` 不存在，`dict` 就会报错：

```
>>> d['Thomas']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Thomas'
```

字典的遍历

通过 `for ... in ...` 我们可以遍历字符串、列表、元组、字典等

1. 遍历字典的key

```
d1 = {'name':'abc', 'age':'18', 'class':'cnh'}
```

```
for key in d1:  
    print(key)
```

结果:

```
name  
age  
class
```

2. 遍历字典的value

```
d1 = {'name':'abc', 'age':'18', 'class':'cnh'}
```

```
for value in d1.values():  
    print(value)
```

结果:

```
abc  
18  
cnh
```

3. 遍历字典的项

```
d1 = {'name':'abc', 'age':'18', 'class':'cnh'}
```

```
for item in d1.items():  
    print(item)
```

结果:

```
('name', 'abc')
('age', '18')
('class', 'cnh')
```

4. 遍历字典的key-value

```
for key,value in d1.items():
    print('key=%s,value=%s'%(key,value))
```

结果:

```
key=name,value=abc
key=age,value=18
key=class,value=cnh
```

`enumerate()` 函数用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，一般用在 `for` 循环当中。

```
>>> chars = ['a', 'b', 'c', 'd']
>>> for i, chr in enumerate(chars):
    print i, chr
```

字典的常见操作

1. 修改元素

字典的每个元素中的数据是可以修改的，只要通过 key 找到，即可修改

```
info = {'name': 'kkb', 'id': 100, 'sex': 'f', 'address': '中国北京'}
```

```
new_id = input('请输入新的学号：')
```

```
info['id'] = int(new_id)
```

```
print('修改之后的id为: %d' % info['id'])
```

2. 添加元素

访问不存在的元素

```
info = {'name': 'kkb', 'sex': 'f', 'address': '中国北京'}
```

```
print('id为:%d' % info['id'])
```

结果：

```
>>> info = {'name': 'kkb', 'sex': 'f', 'address': '中国北京'}
```

```
>>>
```

```
>>> print('id为:%d' % info['id'])
```

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

```
  File "<stdin>", line 1, in <module>
```

```
KeyError: 'id'
```

如果在使用 变量名['键'] = 数据 时，这个“键”在字典中，不存在，那么就会新增这个元素。

添加新的元素

```
info = {'name': 'kkb', 'sex': 'f', 'address': '中国北京'}

# print('id为:%d'%info['id'])#程序会终端运行，因为访问了不存在的键

newId = input('请输入新的学号：')

info['id'] = newId

print('添加之后的id为:%d' % info['id'])
```

结果：

请输入新的学号： 188
添加之后的id为： 188

3. 删除元素

对字典进行删除操作，有一下几种：

- `del`
- `clear()`

`del` 删除指定的元素

```
info = {'name': 'kkb', 'sex': 'f', 'address': '中国北京'}

print('删除前,%s' % info['name'])

del info['name']

print('删除后,%s' % info['name'])
```

结果

```
>>> info = {'name': 'kkb', 'sex': 'f', 'address': '中国北京'}
>>> 
>>> print('删除前,%s' % info['name'])
删除前,kkb
>>> 
>>> del info['name']
>>>
```

```
>>> print('删除后,%s' % info['name'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'name'
```

del 删除整个字典

```
info = {'name':'monitor', 'sex':'f', 'address':'China'}

print('删除前,%s' % info)

del info

print('删除后,%s' % info)
```

clear 清空整个字典

```
info = {'name':'monitor', 'sex':'f', 'address':'China'}

print('清空前,%s' % info)

info.clear()

print('清空后,%s' % info)
```

4. len()

测量字典中，键值对的个数

```
d1 = {'name':'abc','age':'18', 'class':'cnh'}
print(len(d1))
结果:
3
```

5. keys

返回一个包含字典所有 `key` 的列表

```
d1 = {'name':'abc', 'age':'18', 'class':'cnh'}  
print(list(d1.keys()))
```

结果:

```
['name', 'age', 'class']
```

6. values

返回一个包含字典所有 `value` 的列表

```
d1 = {'name':'abc', 'age':'18', 'class':'cnh'}  
print(list(d1.values()))
```

结果:

```
['abc', '18', 'cnh']
```

7. items

返回一个包含所有（键，值）元祖的列表

```
d1 = {'name':'abc', 'age':'18', 'class':'cnh'}  
print(list(d1.items()))
```

结果:

```
[('name', 'abc'), ('age', '18'), ('class', 'cnh')]
```

8. has_key (Python3 已取消)

`dict.has_key(key)` 如果key在字典中，返回 `True`，否则返回 `False`

公共方法

运算符	Python 表达式	结果	描述	支持的数据类型	
+	[1, 2] + [3, 4]	[1, 2, 3, 4]	合并	字符串、列表、元组	
*	['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	['Hi!', 'Hi!', 'Hi!', 'Hi!']	复制	字符串、列表、元组
in	3 in (1, 2, 3)	True	元素是否存在	字符串、列表、元组、字典	
not in	4 not in (1, 2, 3)	True	元素是否不存在	字符串、列表、元组、字典	

+

```
>>> "hello " + "kkb"
'hello kkb'
>>> [1, 2] + [3, 4]
[1, 2, 3, 4]
>>> ('a', 'b') + ('c', 'd')
('a', 'b', 'c', 'd')
```

*

```
>>> 'ab' * 4
'ababab'
>>> [1, 2] * 4
[1, 2, 1, 2, 1, 2, 1, 2]
>>> ('a', 'b') * 4
('a', 'b', 'a', 'b', 'a', 'b', 'a', 'b')
```

in

```
>>> 'kk' in 'hello kkb'  
True  
>>> 3 in [1, 2]  
False  
>>> 4 in (1, 2, 3, 4)  
True  
>>> "name" in {"name":"kkb", "age":24}  
True
```

注意，`in` 在对字典操作时，判断的是字典的键

Python内置函数

Python包含了以下内置函数:

序号	方法	描述
1	cmp(item1, item2)	比较两个值
2	len(item)	计算容器中元素个数
3	max(item)	返回容器中元素最大值
4	min(item)	返回容器中元素最小值
5	del(item)	删除变量

cmp

```
print ("cmp(80, 100) : ", cmp(80, 100))
print ("cmp(180, 100) : ", cmp(180, 100))
print ("cmp(-80, 100) : ", cmp(-80, 100))
print ("cmp(80, -100) : ", cmp(80, -100))
print ("cmp(80, 80) : ", cmp(80, 80))
```

以上实例运行后输出结果为:

```
cmp(80, 100) : -1
cmp(180, 100) : 1
cmp(-80, 100) : -1
cmp(80, -100) : 1
cmp(80, 80)) : 0
-1
```

注意: `cmp` 在比较字典数据时, 先比较键, 再比较值。

```
>>> cmp({"a":1}, {"b":1})
-1
>>> cmp({"a":2}, {"a":1})
1
```

```
>>> cmp({"a":2}, {"a":2, "b":1})  
-1
```

len

```
>>> len("hello")  
5  
>>> len([1, 2, 3, 4])  
4  
>>> len((3,4))  
2  
>>> len({"a":1, "b":2})  
2
```

注意：len在操作字典数据时，返回的是键值对个数。

max

```
>>> max("hello kkb")  
'o'  
>>> max([1,4,522,3,4])  
522  
>>> max({"a":1, "b":2})  
'b'  
>>> max({"a":10, "b":2})  
'b'  
>>> max({"c":10, "b":2})  
'c'
```

del

del有两种用法，一种是del加空格，另一种是del()

```
>>> a = 1  
>>> a  
1  
>>> del a
```

```
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>> a = ['a', 'b']
>>> del a[0]
>>> a
['b']
>>> del(a)
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

多维列表/元祖访问的示例

```
>>> tuple1 = [(2,3),(4,5)]
>>> tuple1[0]
(2, 3)
>>> tuple1[0][0]
2
>>> tuple1[0][2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

```
>>> tuple1[0][1]
3
>>> tuple1[2][2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> tuple2 = tuple1+[(3)]
>>> tuple2
[(2, 3), (4, 5), 3]
```

```
>>> tuple2[2]
3
>>> tuple2[2][0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not subscriptable
```

函数的简介

1. 简介

如果在开发程序时，需要某块代码多次，但是为了提高编写的效率以及代码的重用，所以把具有独立功能的代码块组织为一个小模块，这就是函数。

例如：我们知道圆的面积计算公式为： $S = \pi r^2$

当我们知道半径r的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积：

```
r1 = 12.34
r2 = 9.08
r3 = 73.1
s1 = 3.14 * r1 * r1
s2 = 3.14 * r2 * r2
s3 = 3.14 * r3 * r3
```

当代码出现有规律的重复的时候，你就需要当心了，每次写 `3.14 * x * x` 不仅很麻烦，而且，如果要把`3.14`改成 `3.14159265359` 的时候，得全部替换。

有了函数，我们就不再每次写 `s = 3.14 * x * x`，而是写成更有意义的函数调用 `s = area_of_circle(x)`，而函数 `area_of_circle` 本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，Python也不例外。Python不但能非常灵活地定义函数，而且本身内置了很多有用的函数，可以直接调用。

2. 函数定义和调用

(1) 定义函数

定义函数的格式如下：

```
def 函数名():
    代码
```

```
# 定义一个函数，能够完成打印信息的功能
def printInfo():
    print('-----')
    print('      人生苦短，我用Python')
    print('-----')
```

(2) 调用函数

定义了函数之后，就相当于有了一个具有某些功能的代码，想要让这些代码能够执行，需要调用它

调用函数很简单的，通过 `函数名()` 即可完成调用

```
# 定义完函数后，函数是不会自动执行的，需要调用它才可以
printInfo()
```

(3) 注意：

- 每次调用函数时，函数都会从头开始执行，当这个函数中的代码执行完毕后，意味着调用结束了
- 当然了如果函数中执行到了`return`也会结束函数

函数的文档说明

```
>>> def test(a,b):
...     "用来完成对2个数求和"
...     print("%d"%(a+b))
...
>>>
>>> test(11,22)
33
```

如果执行，以下代码

```
>>> help(test)
```

能够看到test函数的相关说明

```
Help on function test in module __main__:

test(a, b)
    用来完成对2个数求和
(END)
```

还可以用 `test.__doc__` 直接查看文档说明

```
>>> def test(a,b):
...     "用来完成对2个数求和"
...     print("%d"%(a+b))
...
>>>
>>> print(test.__doc__)
用来完成对2个数求和
```

函数的参数

Python的函数定义非常简单，但灵活度却非常大。除了正常定义的必选参数外，还可以使用默认参数、可变参数和关键字参数，使得函数定义出来的接口，不但能处理复杂的参数，还可以简化调用者的代码。

1. 位置参数

我们先写一个计算 x^2 的函数：

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

对于 `power(x)` 函数，参数x就是一个位置参数。

当我们调用 `power` 函数时，必须传入有且仅有的一个参数x：

```
>>> power(5)  
25  
>>> power(15)  
225
```

现在，如果我们要计算 x^3 怎么办？可以再定义一个 `power3` 函数，但是如果要计算 x^4 、 x^5 怎么办？我们不可能定义无限多个函数。

你也许想到了，可以把 `power(x)` 修改为 `power(x, n)`，用来计算 x^n ，说干就干：

```
def power(x, n):  
    s = 1  
    while n > 0:  
        n = n - 1  
        s = s * x
```

```
return s
```

对于这个修改后的 `power(x, n)` 函数，可以计算任意 `n` 次方：

```
>>> power(5, 2)
25
>>> power(5, 3)
125
```

修改后的 `power(x, n)` 函数有两个参数：`x` 和 `n`，这两个参数都是位置参数，调用函数时，传入的两个值按照位置顺序依次赋给参数 `x` 和 `n`。

2. 默认参数

新的 `power(x, n)` 函数定义没有问题，但是，旧的调用代码失败了，原因是我们在增加了一个参数，导致旧的代码因为缺少一个参数而无法正常调用：

```
>>> power(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: power() missing 1 required positional argument: 'n'
```

Python 的错误信息很明确：调用函数 `power()` 缺少了一个位置参数 `n`。

这个时候，默认参数就排上用场了。由于我们经常计算 `x^2`，所以，完全可以把第二个参数 `n` 的默认值设定为 2：

```
def power(x, n=2):
    s = 1
    while n > 0:
        n = n - 1
        s = s * x
    return s
```

这样，当我们调用 `power(5)` 时，相当于调用 `power(5, 2)`：

```
>>> power(5)
25
>>> power(5, 2)
25
```

而对于 `n > 2` 的其他情况，就必须明确地传入 `n`，比如 `power(5, 3)`。

从上面的例子可以看出，默认参数可以简化函数的调用。设置默认参数时，有几点要注意：

- 一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；
- 二是当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

3. 使用默认参数有什么好处？

举个例子，我们写个一年级小学生注册的函数，需要传入 `name` 和 `gender` 两个参数：

```
def enroll(name, gender):
    print('name:', name)
    print('gender:', gender)
```

这样，调用 `enroll()` 函数只需要传入两个参数：

```
>>> enroll('Sarah', 'F')
name: Sarah
gender: F
```

如果要继续传入年龄、城市等信息怎么办？这样会使得调用函数的复杂度大大增加。

我们可以把年龄和城市设为默认参数：

```
def enroll(name, gender, age=6, city='Beijing'):  
    print('name:', name)  
    print('gender:', gender)  
    print('age:', age)  
    print('city:', city)
```

这样，大多数学生注册时不需要提供年龄和城市，只提供必须的两个参数：

```
>>> enroll('Sarah', 'F')  
name: Sarah  
gender: F  
age: 6  
city: Beijing
```

只有与默认参数不符的学生才需要提供额外的信息：

```
enroll('Bob', 'M', 7)  
enroll('Adam', 'M', city='Tianjin')
```

可见，默认参数降低了函数调用的难度，而一旦需要更复杂的调用时，又可以传递更多的参数来实现。无论是简单调用还是复杂调用，函数只需要定义一个。

有多个默认参数时，调用的时候，既可以按顺序提供默认参数，比如调用 `enroll('Bob', 'M', 7)`，意思是，除了 `name`，`gender` 这两个参数外，最后1个参数应用在参数 `age` 上，`city` 参数由于没有提供，仍然使用默认值。

也可以不按顺序提供部分默认参数。当不按顺序提供部分默认参数时，需要把参数名写上。比如调用 `enroll('Adam', 'M', city='Tianjin')`，意思是，`city` 参数用传进去的值，其他默认参数继续使用默认值。

注意：默认参数有个最大的坑，演示如下：

先定义一个函数，传入一个list，添加一个 END 再返回：

```
def add_end(L=[]):
    L.append('END')
    return L
```

当你正常调用时，结果似乎不错：

```
>>> add_end([1, 2, 3])
[1, 2, 3, 'END']
>>> add_end(['x', 'y', 'z'])
['x', 'y', 'z', 'END']
```

当你使用默认参数调用时，一开始结果也是对的：

```
>>> add_end()
['END']
```

但是，再次调用 `add_end()` 时，结果就不对了：

```
>>> add_end()
['END', 'END']
>>> add_end()
['END', 'END', 'END']
```

很多初学者很疑惑，默认参数是 `[]`，但是函数似乎每次都“记住了”上次添加了 `'END'` 后的list。

原因解释如下：

Python函数在定义的时候，默认参数 `L` 的值就被计算出来了，即 `[]`，因为默认参数 `L` 也是一个变量，它指向对象 `[]`，每次调用该函数，如果改变了 `L` 的内容，则下次调用时，默认参数的内容就变了，不再是函数定义时的 `[]` 了。

定义默认参数要牢记一点：默认参数必须指向不变对象！

要修改上面的例子，我们可以用 `None` 这个不变对象来实现：

```
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L
```

现在，无论调用多少次，都不会有问题：

```
>>> add_end()
['END']
>>> add_end()
['END']
```

为什么要设计 `str`、`None` 这样的不变对象呢？因为不变对象一旦创建，对象内部的数据就不能修改，这样就减少了由于修改数据导致的错误。此外，由于对象不变，多任务环境下同时读取对象不需要加锁，同时读一点问题都没有。我们在编写程序时，如果可以设计一个不变对象，那就尽量设计成不变对象。

函数的参数

1. 可变参数

在Python函数中，还可以定义可变参数。顾名思义，可变参数就是传入的参数个数是可变的，可以是1个、2个到任意个，还可以是0个。

我们以数学题为例子，给定一组数字 $a, b, c\dots$ ，请计算 $a^2 + b^2 + c^2 + \dots$ 。

要定义出这个函数，我们必须确定输入的参数。由于参数个数不确定，我们首先想到可以把 $a, b, c\dots$ 作为一个 `list` 或 `tuple` 传进来，这样，函数可以定义如下：

```
def calc(numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

但是调用的时候，需要先组装出一个 `list` 或 `tuple`：

```
>>> calc([1, 2, 3])
14
>>> calc((1, 3, 5, 7))
84
```

如果利用可变参数，调用函数的方式可以简化成这样：

```
>>> calc(1, 2, 3)
14
>>> calc(1, 3, 5, 7)
84
```

所以，我们把函数的参数改为可变参数：

```
def calc(*numbers):
    sum = 0
    for n in numbers:
        sum = sum + n * n
    return sum
```

定义可变参数和定义一个 `list` 或 `tuple` 参数相比，仅仅在参数前面加了一个 `*` 号。在函数内部，参数 `numbers` 接收到的是一个 `tuple`，因此，函数代码完全不变。但是，调用该函数时，可以传入任意个参数，包括0个参数：

```
>>> calc(1, 2)
5
>>> calc()
0
```

如果已经有一个`list`或者`tuple`，要调用一个可变参数怎么办？可以这样做：

```
>>> nums = [1, 2, 3]
>>> calc(nums[0], nums[1], nums[2])
14
```

这种写法当然是可行的，问题是太繁琐，所以Python允许你在`list`或`tuple`前面加一个 `*` 号，把`list`或`tuple`的元素变成可变参数传进去：

```
>>> nums = [1, 2, 3]
>>> calc(*nums)
14
```

`*nums` 表示把 `nums` 这个`list`的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

2. 命名关键字参数

对于关键字参数，函数的调用者可以传入任意不受限制的关键字参数。至于到底传入了哪些，就需要在函数内部通过 `kw` 检查。

仍以 `person()` 函数为例，我们希望检查是否有 `city` 和 `job` 参数：

```
def person(name, age, **kw):
    if 'city' in kw:
        # 有city参数
        pass
    if 'job' in kw:
        # 有job参数
        pass
    print('name:', name, 'age:', age, 'other:', kw)
```

但是调用者仍可以传入不受限制的关键字参数：

```
>>> person('Jack', 24, city='Beijing', addr='Chaoyang', zipcode=123
456)
```

如果要限制关键字参数的名字，就可以用命名关键字参数，例如，只接收 `city` 和 `job` 作为关键字参数。这种方式定义的函数如下：

```
def person(name, age, *, city, job):
    print(name, age, city, job)
```

和关键字参数 `**kw` 不同，命名关键字参数需要一个特殊分隔符 `*`，`*` 后面的参数被视为命名关键字参数。

调用方式如下：

```
>>> person('Jack', 24, city='Beijing', job='Engineer')
Jack 24 Beijing Engineer
```

如果函数定义中已经有了一个可变参数，后面跟着的命名关键字参数就不再需要一个特殊分隔符 `*` 了：

```
def person(name, age, *args, city, job):
    print(name, age, args, city, job)
```

命名关键字参数必须传入参数名，这和位置参数不同。如果没有传入参数名，调用将报错：

```
>>> person('Jack', 24, 'Beijing', 'Engineer')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: person() takes 2 positional arguments but 4 were given
```

由于调用时缺少参数名 `city` 和 `job`，Python解释器把这4个参数均视为位置参数，但 `person()` 函数仅接受2个位置参数。

命名关键字参数可以有缺省值，从而简化调用：

```
def person(name, age, *, city='Beijing', job):
    print(name, age, city, job)
```

由于命名关键字参数 `city` 具有默认值，调用时，可不传入 `city` 参数：

```
>>> person('Jack', 24, job='Engineer')
Jack 24 Beijing Engineer
```

使用命名关键字参数时，要特别注意，如果没有可变参数，就必须加一个 `*` 作为特殊分隔符。如果缺少 `*`，Python解释器将无法识别位置参数和命名关键字参数：

```
def person(name, age, city, job):
    # 缺少 *, city和job被视为位置参数
    pass
```

3. 参数组合

在Python中定义函数，可以用必选参数、默认参数、可变参数、关键字参数和命名关键字参数，这5种参数都可以组合使用。但是请注意，参数定义的顺序必须是：必选参数、默认参数、可变参数、命名关键字参数和关键字参数。

比如定义一个函数，包含上述若干种参数：

```
def f1(a, b, c=0, *args, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)

def f2(a, b, c=0, *, d, **kw):
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

在函数调用的时候，Python解释器自动按照参数位置和参数名把对应的参数传进去。

```
>>> f1(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> f1(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> f1(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> f1(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
>>> f2(1, 2, d=99, ext=None)
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

最神奇的是通过一个 `tuple` 和 `dict`，你也可以调用上述函数：

```
>>> args = (1, 2, 3, 4)
>>> kw = {'d': 99, 'x': '#'}
>>> f1(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}
>>> args = (1, 2, 3)
>>> kw = {'d': 88, 'x': '#'}
>>> f2(*args, **kw)
a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

所以，对于任意函数，都可以通过类似 `func(*args, **kw)` 的形式调用它，无论它的参数是如何定义的。

虽然可以组合多达5种参数，但不要同时使用太多的组合，否则函数接口的可理解性很差。

递归函数

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。

举个例子，我们来计算阶乘 $n! = 1 \times 2 \times 3 \times \dots \times n$ ，用函数 `fact(n)` 表示，可以看出：

```
fact(n) = n! = 1 × 2 × 3 × ... × (n-1) × n = (n-1)! × n = fact(n-1) × n
```

所以，`fact(n)` 可以表示为 `n × fact(n-1)`，只有 $n=1$ 时需要特殊处理。

于是，`fact(n)` 用递归的方式写出来就是：

```
def fact(n):
    if n==1:
        return 1
    return n * fact(n - 1)
```

上面就是一个递归函数。可以试试：

```
>>> fact(1)
1
>>> fact(5)
120
>>> fact(100)
9332621544394415268169923885626670049071596826438162146859296389521
7599993229915608941463976156518286253697920827223758251185210916864
0000000000000000000000000000000000
```

如果我们计算 `fact(5)`，可以根据函数定义看到计算过程如下：

```
====> fact(5)
```

```
====> 5 * fact(4)
====> 5 * (4 * fact(3))
====> 5 * (4 * (3 * fact(2)))
====> 5 * (4 * (3 * (2 * fact(1))))
====> 5 * (4 * (3 * (2 * 1)))
====> 5 * (4 * (3 * 2))
====> 5 * (4 * 6)
====> 5 * 24
====> 120
```

递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。

使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。可以试试 `fact(1000)`：

```
>>> fact(1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in fact
...
  File "<stdin>", line 4, in fact
RuntimeError: maximum recursion depth exceeded in comparison
```

解决递归调用栈溢出的方法是通过尾递归优化，事实上尾递归和循环的效果是一样的，所以，把循环看成是一种特殊的尾递归函数也是可以的。

尾递归是指，在函数返回的时候，调用自身本身，并且，`return` 语句不能包含表达式。这样，编译器或者解释器就可以把尾递归做优化，使递归本身无论调用多少次，都只占用一个栈帧，不会出现栈溢出的情况。

上面的 `fact(n)` 函数由于 `return n * fact(n - 1)` 引入了乘法表达式，所以就不是尾递归了。要改成尾递归方式，需要多一点代码，主要是要把每一步的乘积传入到递归函数中：

```
def fact(n):
    return fact_iter(n, 1)

def fact_iter(num, product):
    if num == 1:
        return product
    return fact_iter(num - 1, num * product)
```

可以看到，`return fact_iter(num - 1, num * product)` 仅返回递归函数本身，`num - 1` 和 `num * product` 在函数调用前就会被计算，不影响函数调用。

`fact(5)` 对应的 `fact_iter(5, 1)` 的调用如下：

```
====> fact_iter(5, 1)
====> fact_iter(4, 5)
====> fact_iter(3, 20)
====> fact_iter(2, 60)
====> fact_iter(1, 120)
====> 120
```

尾递归调用时，如果做了优化，栈不会增长，因此，无论多少次调用也不会导致栈溢出。

遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的 `fact(n)` 函数改成尾递归方式，也会导致栈溢出。

小结

- 使用递归函数的优点是逻辑简单清晰，缺点是过深的调用会导致栈溢出。
- 针对尾递归优化的语言可以通过尾递归防止栈溢出。尾递归事实上和循环是等价的，没有循环语句的编程语言只能通过尾递归实现循环。
- Python标准的解释器没有针对尾递归做优化，任何递归函数都存在栈溢出的问题。

函数返回值

1. “返回值”介绍

现实生活中的场景:

我给儿子10块钱，让他给我买包烟。这个例子中，10块钱是我给儿子的，就相当于调用函数时传递到参数，让儿子买烟这个事情最终的目标是，让他把烟给你带回来然后给你对么，此时烟就是返回值

开发中的场景:

定义了一个函数，完成了获取室内温度，想一想是不是应该把这个结果给调用者，只有调用者拥有了这个返回值，才能够根据当前的温度做适当的调整

所谓“返回值”，就是程序中函数完成一件事情后，最后给调用者的结果

2. 带有返回值的函数

想要在函数中把结果返回给调用者，需要在函数中使用 `return`

如下示例:

```
def add2num(a, b):  
    c = a+b  
    return c
```

或者

```
def add2num(a, b):  
    return a+b
```

3. 保存函数的返回值

在“买烟”的例子中，最后儿子给你烟时，你一定是从儿子手中接过来 对么，程序也是如此，如果一个函数返回了一个数据，那么想要用这个数据，那么就需要保存。

保存函数的返回值示例如下：

```
#定义函数
def add2num(a, b):
    return a+b

#调用函数，顺便保存函数的返回值
result = add2num(100,98)

#因为result已经保存了add2num的返回值，所以接下来就可以使用了
print(result)
结果：

198
```

4. 在python中我们怎样返回多个值？

(1) 多个return?

```
def create_nums():
    print("----1---")
    return 1 # 函数中下面的代码不会被执行，因为return除了能够将数据返回之外，还有一个隐藏的功能：结束函数
    print("----2---")
    return 2
    print("----3---")
```

总结1：一个函数中可以有多个 `return` 语句，但是只要有一个 `return` 语句被执行到，那么这个函数就会结束了，因此后面的 `return` 没有什么用处。如果程序设计为如下，是可以的因为不同的场景下执行不同的 `return`

```
def create_nums(num):
```

```
print("---1---")
if num == 100:
    print("---2---")
    return num+1 # 函数中下面的代码不会被执行，因为return除了能够将数据返回之外，还有一个隐藏的功能：结束函数
else:
    print("---3---")
    return num+2
print("---4---")

result1 = create_nums(100)
print(result1) # 打印101
result2 = create_nums(200)
print(result2) # 打印202
```

(2) 一个函数返回多个数据的方式

```
def divid(a, b):
    shang = a//b
    yushu = a%b
    return shang, yushu #默认是元组

result = divid(5, 2)
print(result) # 输出(2, 1)
```

总结2： `return` 后面可以是元组、列表、字典等，只要是能够存储多个数据的类型，就可以一次性返回多个数据

```
def function():
    # return [1, 2, 3]
    # return (1, 2, 3)
    return {"num1": 1, "num2": 2, "num3": 3}
```

如果 `return` 后面有多个数据，那么默认是元组

```
In [1]: a = 1, 2
```

```
In [2]: a  
Out[2]: (1, 2)
```

```
In [3]:  
In [3]: b = (1, 2)  
In [4]: b  
Out[4]: (1, 2)
```

```
In [5]:
```

局部变量

1. 什么是局部变量

如图所示：

The screenshot shows a Sublime Text editor window titled 'test.py' and a terminal window. The code in the editor is:

```
test.py - Sublime Text
test.py
1 #coding=utf-8
2
3 def test1():
4     a = 300
5     print('----test1--修改前--a=%d'%a)
6     a = 200
7     print('----test1--修改后--a=%d'%a)
8
9 def test2():
10    a = 400
11    print('----test2----a=%d'%a)
12
13
14 # 调用函数
15 test1()
16 test2()
```

The variable 'a' in the first assignment of the `test1()` function is highlighted with a red box. A red arrow points from this highlighted code to the text '局部变量' (Local Variable) located in the center of the editor area.

In the terminal window, the output of running `python3 test.py` is shown:

```
python@ubuntu:~/Desktop$ python3 test.py
----test1--修改前--a=300
----test1--修改后--a=200
----test2----a=400
python@ubuntu:~/Desktop$
```

2. 总结

- 局部变量，就是在函数内部定义的变量。
- 作用范围是这个函数内部，即只能在这个函数中使用，在函数的外部是不能使用的。因为其作用范围只是在自己的函数内部，所以不同的函数可以定义相同名字的局部变量（打个比方，把你、我是当做成函数，把局部变量理解为每个人手里的手机，你可有个iPhone8，我当然也可以有个iPhone8了，互不相关）
- 局部变量的作用，为了临时保存数据需要在函数中定义变量来进行存储。
- 当函数调用时，局部变量被创建，当函数调用完成后这个变量就不能够使用了。

全局变量

1. 什么是全局变量

如果一个变量，既能在一个函数中使用，也能在其他的函数中使用，这样的变量就是全局变量。

例如：有2个兄弟各自都有手机，各自有自己的小秘密在手机里，不让另外一方使用（可以理解为局部变量）；但是家里的电话是2个兄弟都可以随便使用的（可以理解为全局变量）

```
# 定义全局变量
a = 100

def test1():
    print(a) # 虽然没有定义变量a但是依然可以获取其数据

def test2():
    print(a) # 虽然没有定义变量a但是依然可以获取其数据
```

```
# 调用函数
test1()
test2()
```

运行结果：

```
100
100
```

总结：

- 在函数外边定义的变量叫做 全局变量
- 全局变量能够在所有的函数中进行访问

2. 全局变量和局部变量名字相同问题

代码：

```
# 定义全局变量
a = 100

def test1():
    # 定义局部变量
    a = 300
    print('---test1---%d'%a)

    #修改
    a = 200
    print('修改后的%d'%a)

def test2():
    print('a = %d'%a)

test1()
test2()
```

结果：

```
---test1---300
修改后的200
a = 100
```

总结：

- 当函数内出现局部变量和全局变量相同名字时，函数内部中的 `变量名 = 数` 据 此时理解为定义了一个局部变量，而不是修改全局变量的值

3. 修改全局变量

函数中进行使用全局变量时可否进行修改呢？

```
# 定义全局变量
```

```
a = 100

def test1():
    # 定义局部变量
    global a
    print('修改之前: %d'%a)

#修改
a = 200
print('修改后的%d'%a)

def test2():
    print('a = %d'%a)

test1()
test2()
```

结果：

修改之前: 100

修改后的200

a = 200

多函数程序的基本使用流程

一般在实际开发过程中，一个程序往往由多个函数组成，并且多个函数共享某些数据，这种场景是经常出现的，因此下面来总结下，多个函数中共享数据的几种方式

1. 使用全局变量

```
g_num = 0

def test1():
    global g_num
    # 将处理结果存储到全局变量g_num中.....
    g_num = 100

def test2():
    # 通过获取全局变量g_num的值，从而获取test1函数处理之后的结果
    print(g_num)

# 1. 先调用test1得到数据并且存到全局变量中
test1()

# 2. 再调用test2，处理test1函数执行之后的这个值
test2()
```

2. 使用函数的返回值、参数

```
def test1():
    # 通过return将一个数据结果返回
    return 50

def test2(num):
```

```
# 通过形参的方式保存传递过来的数据，就可以处理了
print(num)

# 1. 先调用test1得到数据并且存到变量result中
result = test1()

# 2. 调用test2时，将result的值传递到test2中，从而让这个函数对其进行处理
test2(result)
```

3. 函数嵌套调用

```
def test1():
    # 通过return将一个数据结果返回
    return 20

def test2():
    # 1. 先调用test1并且把结果返回来
    result = test1()
    # 2. 对result进行处理
    print(result)

# 调用test2时，完成所有的处理
test2()
```

切片

取一个 `list` 或 `tuple` 的部分元素是非常常见的操作。比如，一个 `list` 如下：

```
>>> L = ['Michael', 'Sarah', 'Tracy', 'Bob', 'Jack']
```

取前3个元素，应该怎么做？

笨办法：

```
>>> [L[0], L[1], L[2]]  
['Michael', 'Sarah', 'Tracy']
```

取前N个元素，也就是索引为0-(N-1)的元素，可以用循环：

```
>>> r = []  
>>> n = 3  
>>> for i in range(n):  
...     r.append(L[i])  
...  
>>> r  
['Michael', 'Sarah', 'Tracy']
```

对这种经常取指定索引范围的操作，用循环十分繁琐，因此，Python提供了切片 `Slice` 操作符，能大大简化这种操作。

对应上面的问题，取前3个元素，用一行代码就可以完成切片：

```
>>> L[0:3]  
['Michael', 'Sarah', 'Tracy']
```

`L[0:3]` 表示，从索引 `0` 开始取，直到索引 `3` 为止，但不包括索引 `3`。即索引 `0`, `1`, `2`，正好是3个元素。

如果第一个索引是 `0`，还可以省略：

```
>>> L[:3]
['Michael', 'Sarah', 'Tracy']
```

也可以从索引1开始，取出2个元素出来：

```
>>> L[1:3]
['Sarah', 'Tracy']
```

类似的，既然Python支持 `L[-1]` 取倒数第一个元素，那么它同样支持倒数切片，试试：

```
>>> L[-2:]
['Bob', 'Jack']
>>> L[-2:-1]
['Bob']
```

记住倒数第一个元素的索引是 `-1`。

切片操作十分有用。我们先创建一个0-99的数列：

```
>>> L = list(range(100))
>>> L
[0, 1, 2, 3, ..., 99]
```

可以通过切片轻松取出某一段数列。比如前10个数：

```
>>> L[:10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

后10个数：

```
>>> L[-10:]  
[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

前11-20个数：

```
>>> L[10:20]  
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

前10个数，每两个取一个：

```
>>> L[:10:2]  
[0, 2, 4, 6, 8]
```

所有数，每5个取一个：

```
>>> L[::-5]  
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80,  
85, 90, 95]
```

甚至什么都不写，只写 `[:]` 就可以原样复制一个list：

```
>>> L[:]  
[0, 1, 2, 3, ..., 99]
```

`tuple` 也是一种 `list`，唯一区别是 `tuple` 不可变。因此，`tuple` 也可以用切片操作，只是操作的结果仍是 `tuple`：

```
>>> (0, 1, 2, 3, 4, 5)[:3]  
(0, 1, 2)
```

字符串 'xxx' 也可以看成是一种 list，每个元素就是一个字符。因此，字符串也可以用切片操作，只是操作结果仍是字符串：

```
>>> 'ABCDEFG'[:3]  
'ABC'  
>>> 'ABCDEFG'[::2]  
'ACEG'
```

在很多编程语言中，针对字符串提供了很多各种截取函数（例如，`substring`），其实目的就是对字符串切片。**Python**没有针对字符串的截取函数，只需要切片一个操作就可以完成，非常简单。

迭代

如果给定一个 `list` 或 `tuple`，我们可以通过 `for` 循环来遍历这个 `list` 或 `tuple`，这种遍历我们称为迭代 `Iteration`。

在Python中，迭代是通过 `for ... in` 来完成的，而很多语言比如C语言，迭代 `list` 是通过下标完成的，比如Java代码：

```
for (i=0; i<list.length; i++) {  
    n = list[i];  
}
```

可以看出，Python的 `for` 循环抽象程度要高于C的 `for` 循环，因为Python的 `for` 循环不仅可以用在`list`或`tuple`上，还可以作用在其他可迭代对象上。

`list` 这种数据类型虽然有下标，但很多其他数据类型是没有下标的，但是，只要是可迭代对象，无论有无下标，都可以迭代，比如`dict`就可以迭代：

```
>>> d = {'a': 1, 'b': 2, 'c': 3}  
>>> for key in d:  
...     print(key)  
...  
a  
c  
b
```

因为 `dict` 的存储不是按照`list`的方式顺序排列，所以，迭代出的结果顺序很可能不一样。

默认情况下，`dict` 迭代的是 `key`。如果要迭代 `value`，可以用 `for value in d.values()`，如果要同时迭代 `key` 和 `value`，可以用 `for k, v in d.items()`。

由于字符串也是可迭代对象，因此，也可以作用于 `for` 循环：

```
>>> for ch in 'ABC':  
...     print(ch)  
...  
A  
B  
C
```

所以，当我们使用 `for` 循环时，只要作用于一个可迭代对象，`for` 循环就可以正常运行，而我们不太关心该对象究竟是list还是其他数据类型。

那么，如何判断一个对象是可迭代对象呢？方法是通过**collections**模块的**Iterable**类型判断：

```
>>> 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
```

最后一个小小问题，如果要对list实现类似Java那样的下标循环怎么办？Python内置的 `enumerate` 函数可以把一个list变成索引-元素对，这样就可以在 `for` 循环中同时迭代索引和元素本身：

```
>>> for i, value in enumerate(['A', 'B', 'C']):  
...     print(i, value)  
...  
0 A  
1 B  
2 C
```

上面的 `for` 循环里，同时引用了两个变量，在Python里是很常见的，比如下面的代码：

```
>>> for x, y in [(1, 1), (2, 4), (3, 9)]:  
...     print(x, y)  
...  
1 1  
2 4  
3 9
```

列表生成式

列表生成式即 List Comprehensions，是Python内置的非常简单却强大的可以用来创建 list 的生成式。

举个例子，要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] 可以用 list(range(1, 11))：

```
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

但如果要生成 [1x1, 2x2, 3x3, …, 10x10] 怎么做？方法一是循环：

```
>>> L = []
>>> for x in range(1, 11):
...     L.append(x * x)
...
>>> L
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

但是循环太繁琐，而列表生成式则可以用一行语句代替循环生成上面的 list：

```
>>> [x * x for x in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

写列表生成式时，把要生成的元素 `x * x` 放到前面，后面跟 `for` 循环，就可以把list创建出来，十分有用，多写几次，很快就可以熟悉这种语法。

`for`循环后面还可以加上`if`判断，这样我们就可以筛选出仅偶数的平方：

```
>>> [x * x for x in range(1, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

还可以使用两层循环，可以生成全排列：

```
>>> [m + n for m in 'ABC' for n in 'XYZ']
['AX', 'AY', 'AZ', 'BX', 'BY', 'BZ', 'CX', 'CY', 'CZ']
```

三层和三层以上的循环就很少用到了。

运用列表生成式，可以写出非常简洁的代码。例如，列出当前目录下的所有文件和目录名，可以通过一行代码实现：

```
>>> import os # 导入os模块，模块的概念后面讲到
>>> [d for d in os.listdir('.')] # os.listdir可以列出文件和目录
['.emacs.d', '.ssh', '.Trash', 'Adlm', 'Applications', 'Desktop',
'Documents', 'Downloads', 'Library', 'Movies', 'Music', 'Pictures',
'Public', 'VirtualBox VMs', 'Workspace', 'XCode']
```

`for` 循环其实可以同时使用两个甚至多个变量，比如 `dict` 的 `items()` 可以同时迭代 `key` 和 `value`：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> for k, v in d.items():
...     print(k, '=', v)
...
y = B
x = A
z = C
```

因此，列表生成式也可以使用两个变量来生成 `list`：

```
>>> d = {'x': 'A', 'y': 'B', 'z': 'C' }
>>> [k + '=' + v for k, v in d.items()]
['y=B', 'x=A', 'z=C']
```

最后把一个list中所有的字符串变成小写：

```
>>> L = ['Hello', 'World', 'IBM', 'Apple']
>>> [s.lower() for s in L]
['hello', 'world', 'ibm', 'apple']
```

练习

如果 `list` 中既包含字符串，又包含整数，由于非字符串类型没有 `lower()` 方法，所以列表生成式会报错：

```
>>> L = ['Hello', 'World', 18, 'Apple', None]
>>> [s.lower() for s in L]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 1, in <listcomp>
      AttributeError: 'int' object has no attribute 'lower'
```

使用内建的 `isinstance` 函数可以判断一个变量是不是字符串：

```
>>> x = 'abc'
>>> y = 123
>>> isinstance(x, str)
True
>>> isinstance(y, str)
False
```

生成器

通过列表生成式，我们可以直接创建一个列表。但是，受到内存限制，列表容量肯定是有限的。而且，创建一个包含100万个元素的列表，不仅占用很大的存储空间，如果我们仅仅需要访问前面几个元素，那后面绝大多数元素占用的空间都白白浪费了。

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器： generator 。

要创建一个 generator ，有很多种方法。第一种方法很简单，只要把一个列表生成式的 [] 改成 () ，就创建了一个 generator :

```
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x * x for x in range(10))
>>> g
<generator object <genexpr> at 0x1022ef630>
```

创建 L 和 g 的区别仅在于最外层的 [] 和 () ， L 是一个list，而 g 是一个 generator 。

我们可以直接打印出 list 的每一个元素，但我们怎么打印出 generator 的每一个元素呢？如果要一个一个打印出来，可以通过 next() 函数获得 generator 的下一个返回值：

```
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
```

```
>>> next(g)
9
>>> next(g)
16
>>> next(g)
25
>>> next(g)
36
>>> next(g)
49
>>> next(g)
64
>>> next(g)
81
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

我们讲过，`generator` 保存的是算法，每次调用 `next(g)`，就计算出 `g` 的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出 `StopIteration` 的错误。

当然，上面这种不断调用 `next(g)` 实在是太变态了，正确的方法是使用 `for` 循环，因为 `generator` 也是可迭代对象：

```
>>> g = (x * x for x in range(10))
>>> for n in g:
...     print(n)
...
0
1
4
9
16
25
36
```

49
64
81

所以，我们创建了一个 `generator` 后，基本上永远不会调用 `next()`，而是通过 `for` 循环来迭代它，并且不需要关心 `StopIteration` 的错误。

`generator` 非常强大。如果推算的算法比较复杂，用类似列表生成式的 `for` 循环无法实现的时候，还可以用函数来实现。

比如，著名的斐波拉契数列（Fibonacci），除第一个和第二个数外，任意一个数都可由前两个数相加得到：

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

斐波拉契数列用列表生成式写不出来，但是，用函数把它打印出来却很容易：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
    return 'done'
```

注意，赋值语句：

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

相当于：

```
t = (b, a + b) # t是一个tuple
a = t[0]
b = t[1]
```

但不必显式写出临时变量t就可以赋值。

上面的函数可以输出斐波那契数列的前N个数：

```
>>> fib(6)
1
1
2
3
5
8
'done'
```

仔细观察，可以看出，`fib` 函数实际上是定义了斐波拉契数列的推算规则，可以从第一个元素开始，推算出后续任意的元素，这种逻辑其实非常类似 `generator`。

也就是说，上面的函数和 `generator` 仅一步之遥。要把 `fib` 函数变成 `generator`，只需要把 `print(b)` 改为 `yield b` 就可以了：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'
```

这就是定义 `generator` 的另一种方法。如果一个函数定义中包含 `yield` 关键字，那么这个函数就不再是一个普通函数，而是一个 `generator`：

```
>>> f = fib(6)
>>> f
<generator object fib at 0x104feaaa0>
```

这里，最难理解的就是generator和函数的执行流程不一样。函数是顺序执行，遇到 `return` 语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用 `next()` 的时候执行，遇到 `yield` 语句返回，再次执行时从上次返回的 `yield` 语句处继续执行。

举个简单的例子，定义一个 `generator`，依次返回数字 1, 3, 5：

```
def odd():
    print('step 1')
    yield 1
    print('step 2')
    yield(3)
    print('step 3')
    yield(5)
```

调用该 `generator` 时，首先要生成一个 `generator` 对象，然后用 `next()` 函数不断获得下一个返回值：

```
>>> o = odd()
>>> next(o)
step 1
1
>>> next(o)
step 2
3
>>> next(o)
step 3
5
>>> next(o)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

可以看到，`odd` 不是普通函数，而是 `generator`，在执行过程中，遇到 `yield` 就中断，下次又继续执行。执行3次 `yield` 后，已经没有 `yield` 可以执行了，所以，第4次调用 `next(o)` 就报错。

回到 `fib` 的例子，我们在循环过程中不断调用 `yield`，就会不断中断。当然要给循环设置一个条件来退出循环，不然就会产生一个无限数列出来。

同样的，把函数改成generator后，我们基本上从来不会用 `next()` 来获取下一个返回值，而是直接使用 `for` 循环来迭代：

```
>>> for n in fib(6):
...     print(n)
...
1
1
2
3
5
8
```

但是用 `for` 循环调用 `generator` 时，发现拿不到 `generator` 的 `return` 语句的返回值。如果想要拿到返回值，必须捕获 `StopIteration` 错误，返回值包含在 `StopIteration` 的 `value` 中：

```
>>> g = fib(6)
>>> while True:
...     try:
...         x = next(g)
...         print('g:', x)
...     except StopIteration as e:
...         print('Generator return value:', e.value)
...         break
...
g: 1
g: 1
g: 2
g: 3
g: 5
g: 8
Generator return value: done
```

关于如何捕获错误，后面的错误处理还会详细讲解。

迭代器

我们已经知道，可以直接作用于 `for` 循环的数据类型有以下几种：

一类是集合数据类型，如 `list`、`tuple`、`dict`、`set`、`str` 等；

一类是 `generator`，包括生成器和带 `yield` 的 `generator function`。

这些可以直接作用于 `for` 循环的对象统称为可迭代对象：`Iterable`。

可以使用 `isinstance()` 判断一个对象是否是 `Iterable` 对象：

```
>>> from collections import Iterable
>>> isinstance([], Iterable)
True
>>> isinstance({}, Iterable)
True
>>> isinstance('abc', Iterable)
True
>>> isinstance((x for x in range(10)), Iterable)
True
>>> isinstance(100, Iterable)
False
```

而生成器不但可以作用于 `for` 循环，还可以被 `next()` 函数不断调用并返回下一个值，直到最后抛出 `StopIteration` 错误表示无法继续返回下一个值了。

可以被 `next()` 函数调用并不断返回下一个值的对象称为迭代器：`Iterator`。

可以使用 `isinstance()` 判断一个对象是否是 `Iterator` 对象：

```
>>> from collections import Iterator
>>> isinstance((x for x in range(10)), Iterator)
True
>>> isinstance([], Iterator)
False
```

```
>>> isinstance({}, Iterator)
False
>>> isinstance('abc', Iterator)
False
```

生成器都是 `Iterator` 对象，但 `list`、`dict`、`str` 虽然是 `Iterable`，却不是 `Iterator`。

把 `list`、`dict`、`str` 等 `Iterable` 变成 `Iterator` 可以使用 `iter()` 函数：

```
>>> isinstance(iter([]), Iterator)
True
>>> isinstance(iter('abc'), Iterator)
True
```

你可能会问，为什么 `list`、`dict`、`str` 等数据类型不是 `Iterator`？

这是因为Python的 `Iterator` 对象表示的是一个数据流，`Iterator` 对象可以被 `next()` 函数调用并不断返回下一个数据，直到没有数据时抛出 `StopIteration` 错误。可以把这个数据流看做是一个有序序列，但我们却不能提前知道序列的长度，只能不断通过 `next()` 函数实现按需计算下一个数据，所以 `Iterator` 的计算是惰性的，只有在需要返回下一个数据时它才会计算。

`Iterator` 甚至可以表示一个无限大的数据流，例如全体自然数。而使用 `list` 是永远不可能存储全体自然数的。

小结

凡是可作用于 `for` 循环的对象都是 `Iterable` 类型；

凡是可作用于 `next()` 函数的对象都是 `Iterator` 类型，它们表示一个惰性计算的序列；

集合数据类型如 `list`、`dict`、`str` 等是 `Iterable` 但不是 `Iterator`，不过可以通过 `iter()` 函数获得一个 `Iterator` 对象。

返回函数

1. 函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个可变参数的求和。通常情况下，求和的函数是这样定义的：

```
def calc_sum(*args):
    ax = 0
    for n in args:
        ax = ax + n
    return ax
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数：

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for n in args:
            ax = ax + n
        return ax
    return sum
```

当我们调用 `lazy_sum()` 时，返回的并不是求和结果，而是求和函数：

```
>>> f = lazy_sum(1, 3, 5, 7, 9)
>>> f
<function lazy_sum.<locals>.sum at 0x101c6ed90>
```

调用函数 `f` 时，才真正计算求和的结果：

```
>>> f()  
25
```

在这个例子中，我们在函数 `lazy_sum` 中又定义了函数 `sum`，并且，内部函数 `sum` 可以引用外部函数 `lazy_sum` 的参数和局部变量，当 `lazy_sum` 返回函数 `sum` 时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用 `lazy_sum()` 时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
>>> f1 = lazy_sum(1, 3, 5, 7, 9)  
>>> f2 = lazy_sum(1, 3, 5, 7, 9)  
>>> f1==f2  
False
```

`f1()` 和 `f2()` 的调用结果互不影响。

闭包

注意到返回的函数在其定义内部引用了局部变量 `args`，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了 `f()` 才执行。我们来看一个例子：

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
            return i*i
        fs.append(f)
    return fs

f1, f2, f3 = count()
```

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都返回了。

你可能认为调用 `f1()`，`f2()` 和 `f3()` 结果应该是 `1`，`4`，`9`，但实际结果是：

```
>>> f1()
9
>>> f2()
9
>>> f3()
9
```

全部都是 9！原因就在于返回的函数引用了变量 i，但它并非立刻执行。等到3个函数都返回时，它们所引用的变量 i 已经变成了 3，因此最终结果为 9。

返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
def count():
    def f(j):
        def g():
            return j*j
        return g
    fs = []
    for i in range(1, 4):
        fs.append(f(i)) # f(i)立刻被执行，因此i的当前值被传入f()
    return fs
```

再看看结果：

```
>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
4
>>> f3()
9
```

匿名函数

当我们在传入函数时，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。

在Python中，对匿名函数提供了有限支持。还是以 `map()` 函数为例，计算 $f(x)=x^2$ 时，除了定义一个 $f(x)$ 的函数外，还可以直接传入匿名函数：

```
>>> list(map(lambda x: x * x, [1, 2, 3, 4, 5, 6, 7, 8, 9]))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

通过对比可以看出，匿名函数 `lambda x: x * x` 实际上就是：

```
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。

匿名函数有个限制，就是只能有一个表达式，不用写 `return`，返回值就是该表达式的结果。

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。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 0x101c6ef28>
>>> f(5)
25
```

同样，也可以把匿名函数作为返回值返回，比如：

```
def build(x, y):
    return lambda: x * x + y * y
```

装饰器

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

```
>>> def now():
...     print('2018-3-25')
...
>>> f = now
>>> f()
2018-3-25
```

函数对象有一个 `__name__` 属性，可以拿到函数的名字：

```
>>> now.__name__
'now'
>>> f.__name__
'now'
```

现在，假设我们要增强 `now()` 函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改 `now()` 函数的定义，这种在代码运行期间动态增加功能的方式，称之为“装饰器” `Decorator`。

本质上，`decorator` 就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的 `decorator`，可以定义如下：

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

观察上面的 `log`，因为它是一个 `decorator`，所以接受一个函数作为参数，并返回一个函数。我们要借助Python的@语法，把 `decorator` 置于函数的定义处：

```
@log  
def now():  
    print('2018-3-25')
```

调用 `now()` 函数，不仅会运行 `now()` 函数本身，还会在运行 `now()` 函数前打印一行日志：

```
>>> now()  
call now():  
2018-3-25
```

把 `@log` 放到 `now()` 函数的定义处，相当于执行了语句：

```
now = log(now)
```

由于 `log()` 是一个 `decorator`，返回一个函数，所以，原来的 `now()` 函数仍然存在，只是现在同名的 `now` 变量指向了新的函数，于是调用 `now()` 将执行新函数，即在 `log()` 函数中返回的 `wrapper()` 函数。

`wrapper()` 函数的参数定义是 `(*args, **kw)`，因此，`wrapper()` 函数可以接受任意参数的调用。在 `wrapper()` 函数内，首先打印日志，再紧接着调用原始函数。

如果 `decorator` 本身需要传入参数，那就需要编写一个返回 `decorator` 的高阶函数，写出来会更复杂。比如，要自定义log的文本：

```
def log(text):  
    def decorator(func):  
        def wrapper(*args, **kw):  
            print('%s %s():' % (text, func.__name__))  
            return func(*args, **kw)
```

```
    return wrapper
return decorator
```

这个3层嵌套的 `decorator` 用法如下：

```
@log('execute')
def now():
    print('2018-3-25')
```

执行结果如下：

```
>>> now()
execute now():
2018-3-25
```

和两层嵌套的 `decorator` 相比，3层嵌套的效果是这样的：

```
>>> now = log('execute')(now)
```

我们来剖析上面的语句，首先执行 `log('execute')`，返回的是 `decorator` 函数，再调用返回的函数，参数是 `now` 函数，返回值最终是 `wrapper` 函数。

以上两种 `decorator` 的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有 `__name__` 等属性，但你去看经过 `decorator` 装饰之后的函数，它们的 `__name__` 已经从原来的 '`now`' 变成了 '`wrapper`'：

```
>>> now.__name__
'wrapper'
```

因为返回的那个 `wrapper()` 函数名字就是 '`wrapper`'，所以，需要把原始函数的 `__name__` 等属性复制到 `wrapper()` 函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写 `wrapper.__name__ = func.__name__` 这样的代码，Python内置的 `functools.wraps` 就是干这个事的，所以，一个完整的 `decorator` 的写法如下：

```
import functools

def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

或者针对带参数的 `decorator`：

```
import functools

def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

`import functools` 是导入 `functools` 模块。模块的概念稍候讲解。现在，只需记住在定义 `wrapper()` 的前面加上 `@functools.wraps(func)` 即可。

偏函数

Python的 `functools` 模块提供了很多有用的功能，其中一个就是偏函数（**Partial function**）。要注意，这里的偏函数和数学意义上的偏函数不一样。

在介绍函数参数的时候，我们讲到，通过设定参数的默认值，可以降低函数调用的难度。而偏函数也可以做到这一点。举例如下：

`int()` 函数可以把字符串转换为整数，当仅传入字符串时，`int()` 函数默认按十进制转换：

```
>>> int('12345')
12345
```

但 `int()` 函数还提供额外的 `base` 参数，默认值为 `10`。如果传入 `base` 参数，就可以做N进制的转换：

```
>>> int('12345', base=8)
5349
>>> int('12345', 16)
74565
```

假设要转换大量的二进制字符串，每次都传入 `int(x, base=2)` 非常麻烦，于是，我们想到，可以定义一个 `int2()` 的函数，默认把 `base=2` 传进去：

```
def int2(x, base=2):
    return int(x, base)
```

这样，我们转换二进制就非常方便了：

```
>>> int2('1000000')
64
```

```
>>> int2('1010101')
85
```

`functools.partial` 就是帮助我们创建一个偏函数的，不需要我们自己定义 `int2()`，可以直接使用下面的代码创建一个新的函数 `int2`：

```
>>> import functools
>>> int2 = functools.partial(int, base=2)
>>> int2('1000000')
64
>>> int2('1010101')
85
```

所以，简单总结 `functools.partial` 的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单。

注意到上面的新的 `int2` 函数，仅仅是把 `base` 参数重新设定默认值为 `2`，但也可以在函数调用时传入其他值：

```
>>> int2('1000000', base=10)
1000000
```

最后，创建偏函数时，实际上可以接收函数对象、`*args` 和 `**kw` 这3个参数，当传入：

```
int2 = functools.partial(int, base=2)
```

实际上固定了 `int()` 函数的关键字参数 `base`，也就是：

```
int2('10010')
```

相当于：

```
kw = { 'base': 2 }
```

```
int('10010', **kw)
```

当传入：

```
max2 = functools.partial(max, 10)
```

实际上会把 10 作为 *args 的一部分自动加到左边，也就是：

```
max2(5, 6, 7)
```

相当于：

```
args = (10, 5, 6, 7)
max(*args)
```

结果为 10。

小结：

当函数的参数个数太多，需要简化时，使用 `functools.partial` 可以创建一个新的函数，这个新函数可以固定住原函数的部分参数，从而在调用时更简单。

模块

1. 定义

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Python中，一个 `.py` 文件就称之为一个模块（Module）。

2. 使用模块有什么好处？

最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。但是也要注意，尽量不要与内置函数名字冲突。点[这里](#)查看Python的所有内置函数。

你也许还想到，如果不同的人编写的模块名相同怎么办？为了避免模块名冲突，Python又引入了按目录来组织模块的方法，称为包（Package）。

举个例子，一个 `abc.py` 的文件就是一个名字叫 `abc` 的模块，一个 `xyz.py` 的文件就是一个名字叫 `xyz` 的模块。

现在，假设我们的 `abc` 和 `xyz` 这两个模块名字与其他模块冲突了，于是我们可以通过包来组织模块，避免冲突。方法是选择一个顶层包名，比如 `mycompany`，按照如下目录存放：

```
mycompany
├── __init__.py
└── abc.py
```

```
└ xyz.py
```

引入了包以后，只要顶层的包名不与别人冲突，那所有模块都不会与别人冲突。现在，`abc.py` 模块的名字就变成了 `mycompany.abc`，类似的，`xyz.py` 的模块名变成了 `mycompany.xyz`。

请注意，每一个包目录下面都会有一个 `__init__.py` 的文件，这个文件是必须存在的，否则，Python 就把这个目录当成普通目录，而不是一个包。`__init__.py` 可以是空文件，也可以有 Python 代码，因为 `__init__.py` 本身就是一个模块，而它的模块名就是 `mycompany`。

类似的，可以有多级目录，组成多级层次的包结构。比如如下的目录结构：

```
mycompany
├── web
│   ├── __init__.py
│   ├── utils.py
│   └── www.py
├── __init__.py
└── abc.py
└── xyz.py
```

文件 `www.py` 的模块名就是 `mycompany.web.www`，两个文件 `utils.py` 的模块名分别是 `mycompany.utils` 和 `mycompany.web.utils`。

自己创建模块时要注意命名，不能和 Python 自带的模块名称冲突。例如，系统自带了 `sys` 模块，自己的模块就不可命名为 `sys.py`，否则将无法导入系统自带的 `sys` 模块。

`mycompany.web` 也是一个模块，请指出该模块对应的.py 文件。

总结：

模块是一组 Python 代码的集合，可以使用其他模块，也可以被其他模块使用。

创建自己的模块时，要注意：

- 模块名要遵循 Python 变量命名规范，不要使用中文、特殊字符；

- 模块名不要和系统模块名冲突，最好先查看系统是否已存在该模块，检查方法是在Python交互环境执行 `import abc`，若成功则说明系统存在此模块。

使用模块

1. Python模块的标准文件模板

Python本身就内置了很多非常有用的模块，只要安装完毕，这些模块就可以立刻使用。

我们以内建的 `sys` 模块为例，编写一个 `hello` 的模块：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

' a test module '

__author__ = 'Michael Liao'

import sys

def test():
    args = sys.argv
    if len(args)==1:
        print('Hello, world!')
    elif len(args)==2:
        print('Hello, %s!' % args[1])
    else:
        print('Too many arguments!')

if __name__=='__main__':
    test()
```

第1行和第2行是标准注释，第1行注释可以让这个 `hello.py` 文件直接在 Unix/Linux/Mac 上运行，第2行注释表示 `.py` 文件本身使用标准 `UTF-8` 编码；

第4行是一个字符串，表示模块的文档注释，任何模块代码的第一个字符串都被视为模块的文档注释；

第6行使用 `__author__` 变量把作者写进去，这样当你公开源代码后别人就可以瞻仰你的大名；

以上就是Python模块的标准文件模板，当然也可以全部删掉不写，但是，按标准办事肯定没错。

2. if name=='main'

你可能注意到了，使用 `sys` 模块的第一步，就是导入该模块：

```
import sys
```

导入 `sys` 模块后，我们就有了变量 `sys` 指向该模块，利用 `sys` 这个变量，就可以访问 `sys` 模块的所有功能。

`sys` 模块有一个 `argv` 变量，用 `list` 存储了命令行的所有参数。`argv` 至少有一个元素，因为第一个参数永远是该 `.py` 文件的名称，例如：

运行 `python3 hello.py` 获得的 `sys.argv` 就是 `['hello.py']`；

运行 `python3 hello.py Michael` 获得的 `sys.argv` 就是 `['hello.py', 'Michael']`。

最后，注意到这两行代码：

```
if __name__=='__main__':
    test()
```

当我们在命令行运行 `hello` 模块文件时，Python解释器把一个特殊变量 `__name__` 置为 `__main__`，而如果在其他地方导入该 `hello` 模块时，`if` 判断将失败，因此，这种 `if` 测试可以让一个模块通过命令行运行时执行一些额外的代码，最常见的就是运行测试。

我们可以用命令行运行 `hello.py` 看看效果：

```
$ python3 hello.py
Hello, world!
```

```
$ python hello.py Michael  
Hello, Michael!
```

如果启动Python交互环境，再导入 `hello` 模块：

```
$ python3  
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 23 2015, 02:52:03)  
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> import hello  
>>>
```

导入时，没有打印 `Hello, word!`，因为没有执行 `test()` 函数。

调用 `hello.test()` 时，才能打印出 `Hello, word!`：

```
>>> hello.test()  
Hello, world!
```

3. 作用域

在一个模块中，我们可能会定义很多函数和变量，但有的函数和变量我们希望给别人使用，有的函数和变量我们希望仅仅在模块内部使用。在Python中，是通过 `_` 前缀来实现的。

正常的函数和变量名是公开的（public），可以被直接引用，比如：`abc`，`x123`，`PI` 等；

类似 `__xxx__` 这样的变量是特殊变量，可以被直接引用，但是有特殊用途，比如上面的 `__author__`，`__name__` 就是特殊变量，`hello` 模块定义的文档注释也可以用特殊变量 `__doc__` 访问，我们自己的变量一般不要用这种变量名；

类似 `_xxx` 和 `__xxx` 这样的函数或变量就是非公开的（private），不应该被直接引用，比如 `_abc`，`__abc` 等；

之所以我们说，`private` 函数和变量“不应该”被直接引用，而不是“不能”被直接引用，是因为Python并没有一种方法可以完全限制访问`private`函数或变量，但是，从编程习惯上不应该引用`private` 函数或变量。

`private` 函数或变量不应该被别人引用，那它们有什么用呢？请看例子：

```
def _private_1(name):
    return 'Hello, %s' % name

def _private_2(name):
    return 'Hi, %s' % name

def greeting(name):
    if len(name) > 3:
        return _private_1(name)
    else:
        return _private_2(name)
```

我们在模块里公开`greeting()` 函数，而把内部逻辑用`private` 函数隐藏起来了，这样，调用`greeting()` 函数不用关心内部的`private` 函数细节，这也是一种非常有用的代码封装和抽象的方法。即：外部不需要引用的函数全部定义成`private`，只有外部需要引用的函数才定义为`public`。

安装第三方模块

1. pip的使用

在Python中，安装第三方模块，是通过包管理工具pip完成的。

如果你正在使用 Mac 或 Linux，安装pip本身这个步骤就可以跳过了。

如果你正在使用 Windows，请参考[安装Python](#)一节的内容，确保安装时勾选了 pip 和 Add python.exe to Path。

在命令提示符窗口下尝试运行 pip，如果 Windows 提示未找到命令，可以重新运行安装程序添加 pip。

注意：Mac或Linux上有可能并存 Python 3.x 和 Python 2.x，因此对应的 pip 命令是 pip3。

例如，我们要安装一个第三方库—— Python Imaging Library，这是Python下非常强大的处理图像的工具库。不过，PIL目前只支持到 Python 2.7，并且有年头没有更新了，因此，基于 PIL 的 Pillow 项目开发非常活跃，并且支持最新的 Python 3。

一般来说，第三方库都会在Python官方的pypi.python.org网站注册，要安装一个第三方库，必须先知道该库的名称，可以在官网或者pypi上搜索，比如Pillow的名称叫[Pillow](#)，因此，安装 Pillow 的命令就是：

```
pip install Pillow
```

耐心等待下载并安装后，就可以使用 Pillow 了。

且慢！有更好的方法！



2. 安装常用模块

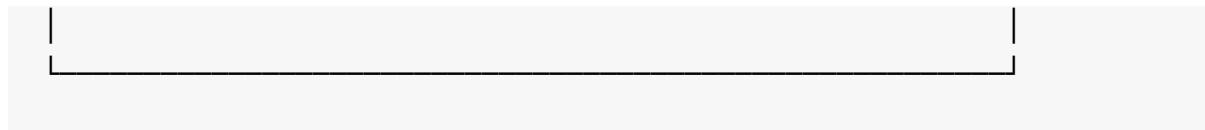
在使用Python时，我们经常需要用到很多第三方库，例如，上面提到的 `Pillow`，以及 `MySQL` 驱动程序，Web框架 `Flask`，科学计算 `Numpy` 等。用 `pip` 一个一个安装费时费力，还需要考虑兼容性。我们推荐直接使用 [Anaconda](#)，这是一个基于Python的数据处理和科学计算平台，它已经内置了许多非常有用第三方库，我们装上 `Anaconda`，就相当于把数十个第三方模块自动安装好了，非常简单易用。

可以从[Anaconda官网](#)下载GUI安装包，安装包有500~600M，所以需要耐心等待下载。网速慢的同学请移步[国内镜像](#)。下载后直接安装，Anaconda会把系统Path中的python指向自己自带的Python，并且，Anaconda安装的第三方模块会安装在Anaconda自己的路径下，不影响系统已安装的Python目录。

安装好Anaconda后，重新打开命令行窗口，输入python，可以看到Anaconda的信息：

```
Command Prompt - python - □ x
Microsoft Windows [Version 10.0.0]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\> python
Python 3.6.3 |Anaconda, Inc.| ... on win32
Type "help", ... for more information.
>>> import numpy
>>> _
```



可以尝试直接 `import numpy` 等已安装的第三方模块。

3. 模块搜索路径

当我们试图加载一个模块时，Python会在指定的路径下搜索对应的 `.py` 文件，如果找不到，就会报错：

```
>>> import mymodule
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named mymodule
```

默认情况下，Python解释器会搜索当前目录、所有已安装的内置模块和第三方模块，搜索路径存放在 `sys` 模块的 `path` 变量中：

```
>>> import sys
>>> sys.path
 ['', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6.zip', '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6', ..., '/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages']
```

如果我们要添加自己的搜索目录，有两种方法：

- 一是直接修改 `sys.path`，添加要搜索的目录：

```
>>> import sys
>>> sys.path.append('/Users/michael/my_py_scripts')
```

这种方法是在运行时修改，运行结束后失效。

- 第二种方法是设置环境变量 `PYTHONPATH`，该环境变量的内容会被自动添

加到模块搜索路径中。设置方式与设置Path环境变量类似。注意只需要添加你自己的搜索路径，Python自己本身的搜索路径不受影响。

面向对象编程

面向对象编程——Object Oriented Programming，简称**OOP**，是一种程序设计思想。OOP把对象作为程序的基本单元，一个对象包含了数据和操作数据的函数。

面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

而面向对象的程序设计把计算机程序视为一组对象的集合，而每个对象都可以接收其他对象发过来的消息，并处理这些消息，计算机程序的执行就是一系列消息在各个对象之间传递。

在Python中，所有数据类型都可以视为对象，当然也可以自定义对象。自定义的对象数据类型就是面向对象中的类（Class）的概念。

我们以一个例子来说明面向过程和面向对象在程序流程上的不同之处。

假设我们要处理学生的成绩表，为了表示一个学生的成绩，面向过程的程序可以用一个 `dict` 表示：

```
std1 = { 'name': 'Michael', 'score': 98 }
std2 = { 'name': 'Bob', 'score': 81 }
```

而处理学生成绩可以通过函数实现，比如打印学生的成绩：

```
def print_score(std):
    print('%s: %s' % (std['name'], std['score']))
```

如果采用面向对象的程序设计思想，我们首选思考的不是程序的执行流程，而是 `Student` 这种数据类型应该被视为一个对象，这个对象拥有 `name` 和 `score` 这两个属性（Property）。如果要打印一个学生的成绩，首先

必须创建出这个学生对应的对象，然后，给对象发一个 `print_score` 消息，让对象自己把自己的数据打印出来。

```
class Student(object):

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

    def print_score(self):
        print('%s: %s' % (self.name, self.score))
```

给对象发消息实际上就是调用对象对应的关联函数，我们称之为对象的方法（Method）。面向对象的程序写出来就像这样：

```
bart = Student('Bart Simpson', 59)
lisa = Student('Lisa Simpson', 87)
bart.print_score()
lisa.print_score()
```

面向对象的设计思想是从自然界中来的，因为在自然界中，类（Class）和实例（Instance）的概念是很自然的。Class是一种抽象概念，比如我们定义的 `class Student`，是指学生这个概念，而实例（Instance）则是一个个具体的 `Student`，比如，`Bart Simpson` 和 `Lisa Simpson` 是两个具体的 `Student`。

所以，面向对象的设计思想是抽象出Class，根据Class创建 Instance。

面向对象的抽象程度又比函数要高，因为一个Class既包含数据，又包含操作数据的方法。

小结

数据封装、继承和多态是面向对象的三大特点，我们后面会详细讲解。

类和对象

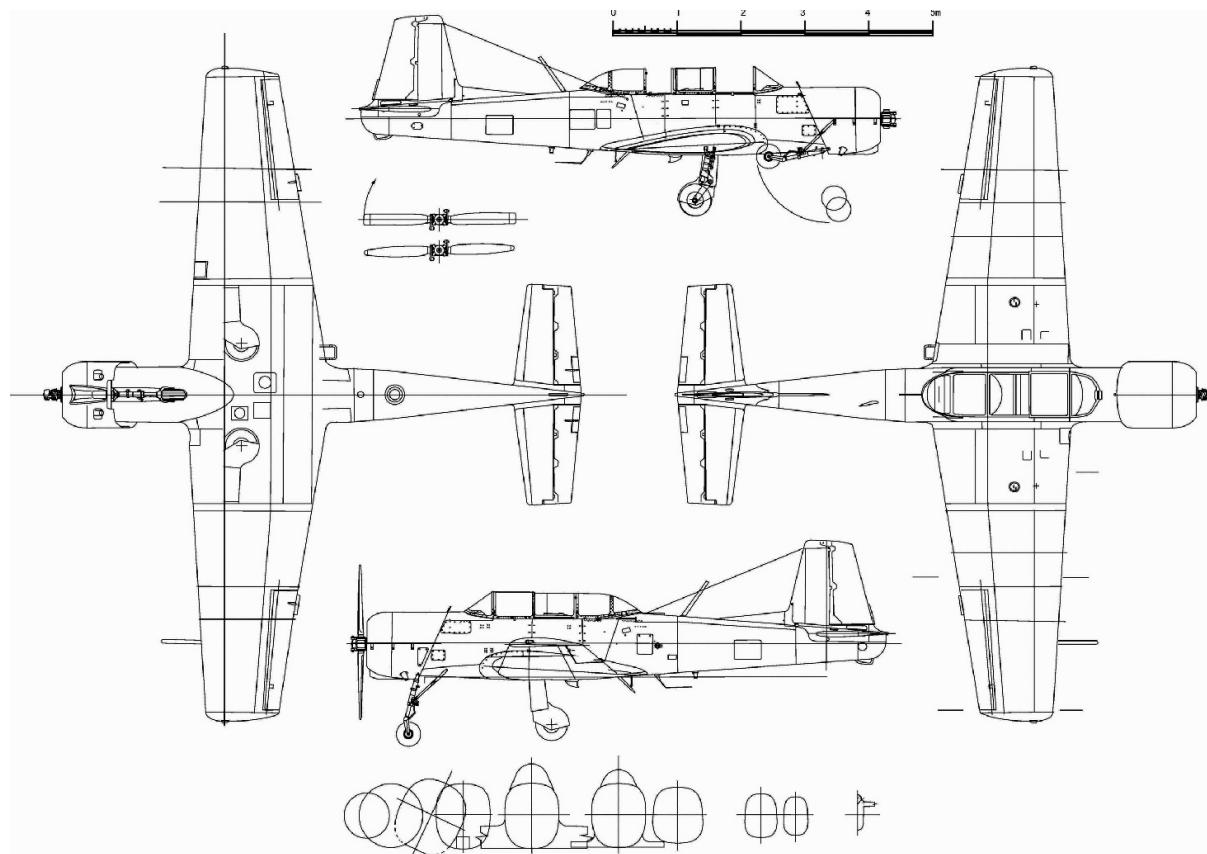
面向对象编程的2个非常重要的概念：类和对象

对象是面向对象编程的核心，在使用对象的过程中，为了将具有共同特征和行为的一组对象抽象定义，提出了另外一个新的概念——类

类就相当于制造飞机时的图纸，用它来进行创建的飞机就相当于对象。

1. 类

人以类聚 物以群分。 具有相似内部状态和运动规律的实体的集合(或统称为抽象)。 具有相同属性和行为事物的统称 类是抽象的,在使用的时候通常会找到这个类的一个具体的存在,使用这个具体的存在。一个类可以找到多个对象

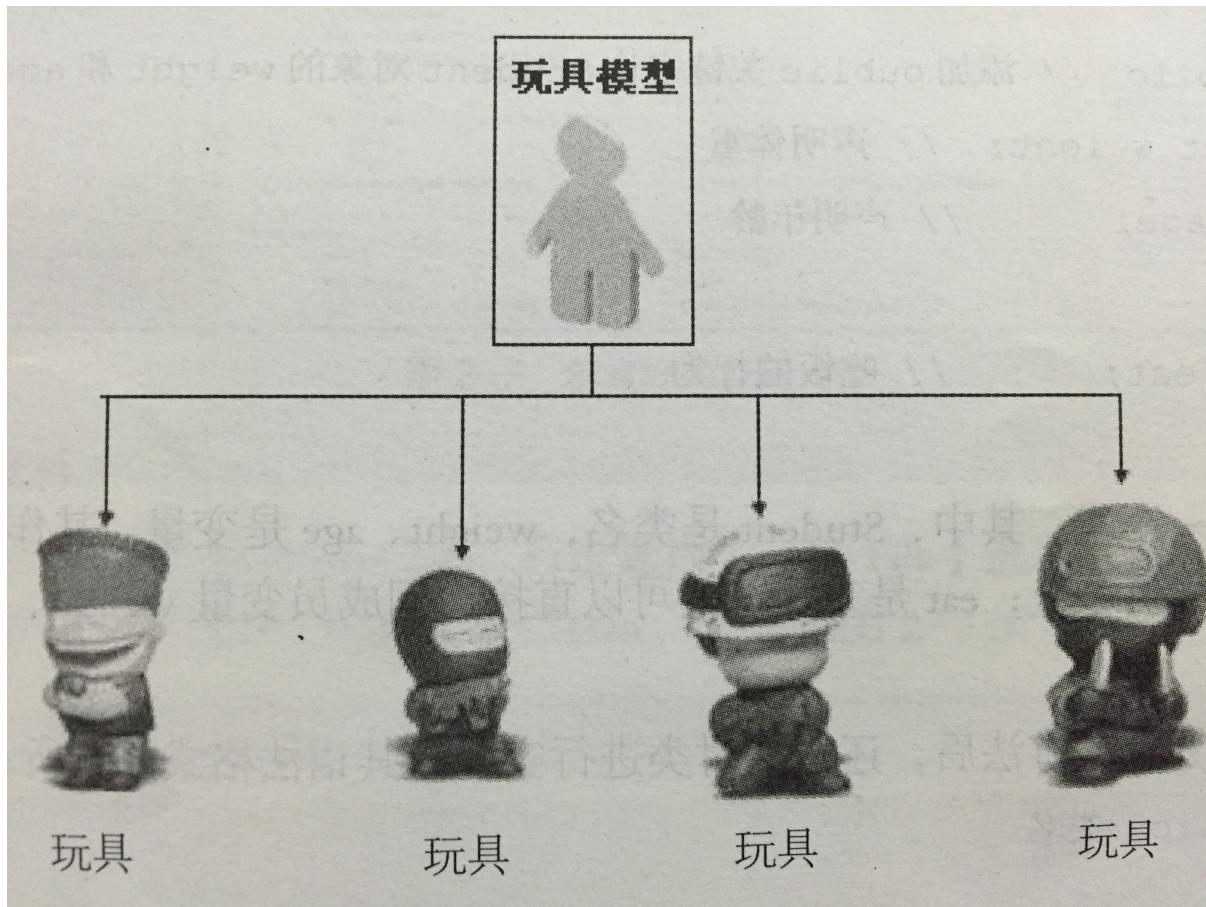


2. 对象

某一个具体事物的存在，在现实世界中可以是看得见摸得着的。可以是直接使用的



3. 类和对象之间的关系



总结：类就是创建对象的模板

4. 定义类和创建对象

定义一个类，格式如下：

```
class 类名:  
    方法列表  
  
# class Hero: # 经典类（旧式类）定义形式  
# class Hero():  
  
class Hero(object): # 新式类定义形式  
    def info(self):  
        print("hero")
```

说明：

- 定义类时有2种形式：新式类和经典类，上面代码中的Hero为新式类，前两行注释部分则为经典类；
- `object` 是Python 里所有类的最顶级父类；
- 类名 的命名规则按照"大驼峰命名法"；
- `info` 是一个实例方法，第一个参数一般是 `self`，表示实例对象本身，当然了可以将self换为其它的名字，其作用是一个变量 这个变量指向了实例对象。
- python中，可以根据已经定义的类去创建出一个或多个对象。

创建对象的格式为：

```
对象名1 = 类名()
对象名2 = 类名()
对象名3 = 类名()
```

```
class Hero(object): # 新式类定义形式
    """info 是一个实例方法，类对象可以调用实例方法，实例方法的第一个参数一定是self"""
    def info(self):
        """当对象调用实例方法时，Python会自动将对象本身的引用做为参数，传递到实例方法的第一个参数self里"""
        print(self)
        print("self各不同，对象是出处。")

# Hero这个类 实例化了一个对象
hero = Hero()

# 对象调用实例方法info()，执行info()里的代码
# . 表示选择属性或者方法
hero.info()

print(hero) # 打印对象，则默认打印对象在内存的地址，结果等同于info里的
print(self)
```

```
print(id(hero)) # id(hero) 则是内存地址的十进制形式表示
```

对象的属性和方法

1. 添加和获取对象的属性

```
class Hero(object):
    """定义了一个英雄类，可以移动和攻击"""
    def move(self):
        """实例方法"""
        print("正在前往事发地点...")

    def attack(self):
        """实例方法"""
        print("发出了一招强力的普通攻击...")

# 实例化了一个英雄对象
hero = Hero()

# 给对象添加属性，以及对应的属性值
hero.name = "德玛西亚" # 姓名
hero.hp = 2600 # 生命值
hero.atk = 450 # 攻击力
hero.armor = 200 # 护甲值

# 通过.成员选择运算符，获取对象的属性值
print("英雄 %s 的生命值 :%d" % (self.name, self.hp))
print("英雄 %s 的攻击力 :%d" % (self.name, self.atk))
print("英雄 %s 的护甲值 :%d" % (self.name, self.armor))

# 通过.成员选择运算符，获取对象的实例方法
taidamier.move()
taidamier.attack()
```

2. 通过self获取对象属性

```
class Hero(object):
    """定义了一个英雄类，可以移动和攻击"""
    def move(self):
        """实例方法"""
        print("正在前往事发地点...")

    def attack(self):
        """实例方法"""
        print("发出了一招强力的普通攻击...")

    def info(self):
        """在类的实例方法中，通过self获取该对象的属性"""
        print("英雄 %s 的生命值 :%d" % (self.name, self.hp))
        print("英雄 %s 的攻击力 :%d" % (self.name, self.atk))
        print("英雄 %s 的护甲值 :%d" % (self.name, self.armor))

# 实例化了一个英雄对象
hero = Hero()

# 给对象添加属性，以及对应的属性值
hero.name = "德玛西亚" # 姓名
hero.hp = 2600 # 生命值
hero.atk = 450 # 攻击力
hero.armor = 200 # 护甲值

# 通过.成员选择运算符，获取对象的实例方法
hero.info() # 只需要调用实例方法info()，即可获取英雄的属性
hero.move()
hero.attack()
```

init()

1. init方法

```
class Hero(object):
    """定义了一个英雄类，可以移动和攻击"""
    # Python 的类里提供的，两个下划线开始，两个下划线结束的方法，就是魔法方法，__init__()就是一个魔法方法，通常用来做属性初始化 或 赋值 操作。
    # 如果类面没有写__init__方法，Python会自动创建，但是不执行任何操作，

    # 如果为了能够在完成自己想要的功能，可以自己定义__init__方法，
    # 所以一个类里无论自己是否编写__init__方法 一定有__init__方法。

    def __init__(self):
        """ 方法，用来做变量初始化 或 赋值 操作，在类实例化对象的时候，会被自动调用"""
        self.name = "hero" # 姓名
        self.hp = 2600 # 生命值
        self.atk = 450 # 攻击力
        self.armor = 200 # 护甲值

    def move(self):
        """实例方法"""
        print("正在前往事发地点...")

    def attack(self):
        """实例方法"""
        print("发出了三招强力的普通攻击...")

# 实例化了一个英雄对象，并自动调用__init__()方法
hero = Hero()

# 通过.成员选择运算符，获取对象的实例方法
hero.info() # 只需要调用实例方法info()，即可获取英雄的属性
```

```
hero.move()  
hero.attack()
```



总结：

- `__init__()` 方法，在创建一个对象时默认被调用，不需要手动调用
- `__init__(self)` 中的 `self` 参数，不需要开发者传递，python 解释器会自动把当前的对象引用传递过去。

2. 有参数的init()方法

```
class Hero(object):  
    """定义了一个英雄类，可以移动和攻击"""  
  
    def __init__(self, name, skill, hp, atk, armor):  
        """ __init__() 方法，用来做变量初始化 或 赋值 操作 """  
        # 英雄名  
        self.name = name  
        # 技能  
        self.skill = skill  
        # 生命值：  
        self.hp = hp  
        # 攻击力  
        self.atk = atk  
        # 护甲值  
        self.armor = armor  
  
    def move(self):  
        """实例方法"""  
        print("%s 正在前往事发地点..." % self.name)  
  
    def attack(self):  
        """实例方法"""  
        print("发出了一招强力的%s..." % self.skill)  
  
    def info(self):  
        print("英雄 %s 的生命值 :%d" % (self.name, self.hp))
```

```

        print("英雄 %s 的攻击力 :%d" % (self.name, self.atk))
        print("英雄 %s 的护甲值 :%d" % (self.name, self.armor))

# 实例化英雄对象时，参数会传递到对象的__init__()方法里
blind = Hero("瞎哥", "回旋踢", 2600, 450, 200)
gailun = Hero("盖伦", "大宝剑", 4200, 260, 400)

# 打印 gailun 和 blind 对象
# print(gailun)
# print(blind)

# 不同对象的属性值的单独保存
print(id(blind.name))
print(id(gailun.name))

# 同一个类的不同对象，实例方法共享
print(id(blind.move()))
print(id(gailun.move()))

```

注意：

- 通过一个类，可以创建多个对象，就好比通过一个模具创建多个实体一样
- `__init__(self)` 中，默认有1个参数名字为self，如果在创建对象时传递了2个实参，那么 `__init__(self)` 中除了self作为第一个形参外还需要2个形参，例如 `__init__(self,x,y)`
- 在类内部获取 属性 和 实例方法，通过self获取；
- 在类外部获取 属性 和 实例方法，通过对对象名获取。
- 如果一个类有多个对象，每个对象的属性是各自保存的，都有各自独立的地址；
- 但是实例方法是所有对象共享的，只占用一份内存空间。类会通过self来判断是哪个对象调用了实例方法。

__str__() 方法

```

class Hero(object):
    """定义了一个英雄类，可以移动和攻击"""

    def __init__(self, name, skill, hp, atk, armor):
        """ __init__() 方法，用来做变量初始化 或 赋值 操作"""
        # 英雄名
        self.name = name # 实例变量
        # 技能
        self.skill = skill
        # 生命值:
        self.hp = hp # 实例变量
        # 攻击力
        self.atk = atk
        # 护甲值
        self.armor = armor

    def move(self):
        """实例方法"""
        print("%s 正在前往事发地点..." % self.name)

    def attack(self):
        """实例方法"""
        print("发出了-招强力的%s..." % self.skill)

    # def info(self):
    #     print("英雄 %s 的生命值 :%d" % (self.name, self.hp))
    #     print("英雄 %s 的攻击力 :%d" % (self.name, self.atk))
    #     print("英雄 %s 的护甲值 :%d" % (self.name, self.armor))

def __str__(self):
    """
    这个方法是一个魔法方法 (Magic Method)，用来显示信息
    """

```

该方法需要 `return` 一个数据，并且只有`self`一个参数，当在类的外部 `print(对象)` 则打印这个数据

```
"""
return "英雄 <%s> 数据: 生命值 %d, 攻击力 %d, 护甲值 %d" % (
    self.name, self.hp, self.atk, self.armor)

blind = Hero("瞎哥", "回旋踢", 2600, 450, 200)
gailun = Hero("盖伦", "大宝剑", 4200, 260, 400)

# 如果没有__str__ 则默认打印 对象在内存的地址。
# 当类的实例化对象 拥有 __str__ 方法后，那么打印对象则打印 __str__ 的返回值。
print(blind)
print(gailun)

# 查看类的文档说明，也就是类的注释
print(Hero.__doc__)
```

说明：

- 在python中方法名如果是`xxxx()`的，那么就有特殊的功能，因此叫做“魔法”方法
- 当使用`print`输出对象的时候，默认打印对象的内存地址。如果类定义了`str(self)`方法，那么就会打印从在这个方法中`return`的数据
- `str`方法通常返回一个字符串，作为这个对象的描述信息

__del__() 方法

创建对象后， python解释器默认调用 __init__() 方法；

当删除对象时， python解释器也会默认调用一个方法，这个方法为 __del__() 方法

```
class Hero(object):

    # 初始化方法
    # 创建完对象后会自动被调用
    def __init__(self, name):
        print('__init__方法被调用')
        self.name = name

    # 当对象被删除时，会自动被调用
    def __del__(self):
        print("__del__方法被调用")
        print("%s 被 GM 干掉了..." % self.name)

# 创建对象
hero = Hero("hero")

# 删除对象
print("%d 被删除1次" % id(hero))
del(hero)

print("—" * 10)

gailun = Hero("盖伦")
gailun1 = gailun
gailun2 = gailun

print("%d 被删除1次" % id(gailun))
del(gailun)
```

```
print("%d 被删除1次" % id(gailun1))
del(gailun1)

print("%d 被删除1次" % id(gailun2))
del(gailun2)
```

总结

- 当有变量保存了一个对象的引用时，此对象的引用计数就会加1；
- 当使用 `del()` 删除变量指向的对象时，则会减少对象的引用计数。如果对象的引用计数不为1，那么会让这个对象的引用计数减1，当对象的引用计数为0的时候，则对象才会被真正删除（内存被回收）。

程序中的继承

- 在程序中，继承描述的是多个类之间的所属关系。
- 如果一个类A里面的属性和方法可以复用，则可以通过继承的方式，传递到类B里。
- 那么类A就是基类，也叫做父类；类B就是派生类，也叫做子类。

```
# 父类
class A(object):
    def __init__(self):
        self.num = 10

    def print_num(self):
        print(self.num + 10)

# 子类
class B(A):
    pass

b = B()
print(b.num)
b.print_num()
```

单继承(子类只继承一个父类)

```
# 定义一个Master类
class Master(object):
    def __init__(self):
        # 属性
        self.kongfu = "古法煎饼果子配方"

    # 实例方法
    def make_cake(self):
        print("按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

# 定义Prentice类, 继承了 Master, 则Prentice是子类, Master是父类。
class Prentice(Master):
    # 子类可以继承父类所有的属性和方法, 哪怕子类没有自己的属性和方法, 也可以使用父类的属性和方法。
    pass

    # laoli = Master()
    # print(laoli.kongfu)
    # laoli.make_cake()

    damao = Prentice() # 创建子类实例对象
    print(damao.kongfu) # 子类对象可以直接使用父类的属性
    damao.make_cake() # 子类对象可以直接使用父类的方法
```

总结:

- 虽然子类没有定义 `__init__` 方法初始化属性, 也没有定义实例方法, 但是父类有。所以只要创建子类的对象, 就默认执行了那个继承过来的 `__init__` 方法
- 子类在继承的时候, 在定义类时, 小括号()中为父类的名字

- 父类的属性、方法，会被继承给子类

多继承(子类继承多个父类)

```
class Master(object):
    def __init__(self):
        self.kongfu = "古法煎饼果子配方" # 实例变量, 属性

    def make_cake(self): # 实例方法, 方法
        print("[古法] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

    def dayandai(self):
        print("师傅的大烟袋..")

class School(object):
    def __init__(self):
        self.kongfu = "现代煎饼果子配方"

    def make_cake(self):
        print("[现代] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

    def xiaoyandai(self):
        print("学校的小烟袋..")

# class Prentice(School, Master): # 多继承, 继承了多个父类 (School在前)
#     pass

# damao = Prentice()
# print(damao.kongfu)
# damao.make_cake()
# damao.dayandai()
# damao.xiaoyandai()

class Prentice(Master, School): # 多继承, 继承了多个父类 (Master在前)
```

```
pass

damao = Prentice()
print(damao.kongfu) # 执行Master的属性
damao.make_cake() # 执行Master的实例方法

# 子类的魔法属性__mro__决定了属性和方法的查找顺序
print(Prentice.__mro__)

damao.dayandai() # 不重名不受影响
damao.xiaoyandai()
```

结论：

- 多继承可以继承多个父类，也继承了所有父类的属性和方法
- 注意：如果多个父类中有同名的 属性和方法，则默认使用第一个父类的属性和方法（根据类的魔法属性 `mro` 的顺序来查找）
- 多个父类中，不重名的属性和方法，不会有影响。

子类重写父类的同名属性和方法

```
class Master(object):
    def __init__(self):
        self.kongfu = "古法煎饼果子配方"

    def make_cake(self):
        print("[古法] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

class School(object):
    def __init__(self):
        self.kongfu = "现代煎饼果子配方"

    def make_cake(self):
        print("[现代] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

class Prentice(School, Master): # 多继承, 继承了多个父类
    def __init__(self):
        self.kongfu = "猫氏煎饼果子配方"

    def make_cake(self):
        print("[猫氏] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

# 如果子类和父类的方法名和属性名相同, 则默认使用子类的
# 叫 子类重写父类的同名方法和属性
damao = Prentice()
print(damao.kongfu) # 子类和父类有同名属性, 则默认使用子类的
damao.make_cake() # 子类和父类有同名方法, 则默认使用子类的

# 子类的魔法属性__mro__决定了属性和方法的查找顺序
print(Prentice.__mro__)
```

子类调用父类同名属性和方法

```

class Master(object):
    def __init__(self):
        self.kongfu = "古法煎饼果子配方" # 实例变量，属性

    def make_cake(self): # 实例方法，方法
        print("[古法] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

class School(object):
    def __init__(self):
        self.kongfu = "现代煎饼果子配方"

    def make_cake(self):
        print("[现代] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

class Prentice(School, Master): # 多继承，继承了多个父类
    def __init__(self):
        self.kongfu = "猫氏煎饼果子配方"

    def make_cake(self):
        print("执行子类的__init__方法前, self.kongfu属性: %s" % self.
kongfu)
        self.__init__() # 执行本类的__init__方法，做属性初始化 self.k
ongfu = "猫氏...."
        print("执行子类的__init__方法前, self.kongfu属性: %s" % self.
kongfu)
        print("[猫氏] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

# 调用父类方法格式：父类类名.父类方法(self)
def make_old_cake(self):
    # 不推荐这样访问父类的实例属性，相当于创建了一个新的父类对象

```

```
# print("直接调用Master类的kongfu属性: %s" % Master().kongfu)

# 可以通过执行Master类的__init__方法，来修改self的属性值
print("执行Master类的__init__方法前, self.kongfu属性: %s" % s
elf.kongfu)
    Master.__init__(self) # 调用了父类Master的__init__方法 self.
kongfu = "古法...."
    print("执行Master类的__init__方法后, self.kongfu属性: %s" % s
elf.kongfu)
    Master.make_cake(self) # 调用父类Master的实例方法


def make_new_cake(self):
    # 不推荐这样访问类的实例属性，相当于创建了一个新的父类对象
    # print("直接调用School类的kongfu属性: %s" % School().kongfu)

# 可以通过执行School类的__init__方法，来修改self的属性值
print("执行School类的__init__方法前, self.kongfu属性: %s" % s
elf.kongfu)
    School.__init__(self) # 调用了父类School的__init__方法 self.
kongfu = "现代...."
    print("执行School类的__init__方法后, self.kongfu属性: %s" % s
elf.kongfu)
    School.make_cake(self) # 调用父类School的实例方法

# 实例化对象，自动执行子类的__init__方法
damao = Prentice()

damao.make_cake() # 调用子类的方法（默认重写了父类的同名方法）

print("—" * 10)
damao.make_old_cake() # 进入实例方法去调用父类Master的方法

print("—" * 10)
damao.make_new_cake() # 进入实例方法去调用父类School的方法
```

```
print("—" * 10)
damao.make_cake() # 调用本类的实例方法
```



执行结果：

```
执行子类的__init__方法前, self.kongfu属性: 猫氏煎饼果子配方
执行子类的__init__方法前, self.kongfu属性: 猫氏煎饼果子配方
[猫氏] 按照 <猫氏煎饼果子配方> 制作了一份煎饼果子...
```

```
-----
执行Master类的__init__方法前, self.kongfu属性: 猫氏煎饼果子配方
执行Master类的__init__方法后, self.kongfu属性: 古法煎饼果子配方
[古法] 按照 <古法煎饼果子配方> 制作了一份煎饼果子...
```

```
-----
执行School类的__init__方法前, self.kongfu属性: 古法煎饼果子配方
执行School类的__init__方法后, self.kongfu属性: 现代煎饼果子配方
[现代] 按照 <现代煎饼果子配方> 制作了一份煎饼果子...
```

```
-----
执行子类的__init__方法前, self.kongfu属性: 现代煎饼果子配方
执行子类的__init__方法前, self.kongfu属性: 猫氏煎饼果子配方
[猫氏] 按照 <猫氏煎饼果子配方> 制作了一份煎饼果子...
```

重点：无论何时何地，`self` 都表示是子类的对象。在调用父类方法时，通过传递 `self` 参数，来控制方法和属性的访问修改。

多层继承

```
class Master(object):
    def __init__(self):
        self.kongfu = "古法煎饼果子配方"

    def make_cake(self):
        print("[古法] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

class School(object):
    def __init__(self):
        self.kongfu = "现代煎饼果子配方"

    def make_cake(self):
        print("[现代] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

class Prentice(School, Master): # 多继承, 继承了多个父类
    def __init__(self):
        self.kongfu = "猫氏煎饼果子配方"
        self.money = 10000 # 亿美金

    def make_cake(self):
        self.__init__() # 执行本类的__init__方法, 做属性初始化 self.kongfu = "猫氏...."
        print("[猫氏] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

# 调用父类方法格式: 父类类名.父类方法(self)
def make_old_cake(self):
    Master.__init__(self) # 调用了父类Master的__init__方法 self.kongfu = "古法...."
    Master.make_cake(self) # 调用了父类Master的实例方法
```

```
def make_new_cake(self):
    School.__init__(self) # 调用了父类School的__init__方法 self.
    kongfu = "现代...."
    School.make_cake(self) # 调用父类School的实例方法,

class PrenticePrentice(Prentice): # 多层继承
    pass

pp = PrenticePrentice()
pp.make_cake() # 调用父类的实例方法
pp.make_new_cake()
pp.make_old_cake()

print(pp.money)
```

super()的使用

```
class Master(object):
    def __init__(self):
        self.kongfu = "古法煎饼果子配方" # 实例变量，属性

    def make_cake(self): # 实例方法，方法
        print("[古法] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

# 父类是 Master类
class School(Master):
    def __init__(self):
        self.kongfu = "现代煎饼果子配方"

    def make_cake(self):
        print("[现代] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)
        super().__init__() # 执行父类的构造方法
        super().make_cake() # 执行父类的实例方法

# 父类是 School 和 Master
class Prentice(School, Master): # 多继承，继承了多个父类
    def __init__(self):
        self.kongfu = "猫氏煎饼果子配方"

    def make_cake(self):
        self.__init__() # 执行本类的__init__方法，做属性初始化 self.
kongfu = "猫氏...."
        print("[猫氏] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

    def make_all_cake(self):
        # 方式1. 指定执行父类的方法（代码臃肿）
        # School.__init__(self)
        # School.make_cake(self)
```

```
#  
# Master.__init__(self)  
# Master.make_cake(self)  
#  
# self.__init__()  
# self.make_cake()  
  
# 方法2. super() 带参数版本, 只支持新式类  
# super(Prentice, self).__init__() # 执行父类的 __init__方法  
  
# super(Prentice, self).make_cake()  
# self.make_cake()  
  
# 方法3. super()的简化版, 只支持新式类  
super().__init__() # 执行父类的 __init__方法  
super().make_cake() # 执行父类的 实例方法  
self.make_cake() # 执行本类的实例方法  
  
damao = Prentice()  
damao.make_cake()  
damao.make_all_cake()  
  
# print(Prentice.__mro__)
```

总结：

- 子类继承了多个父类，如果父类类名修改了，那么子类也要涉及多次修改。而且需要重复写多次调用，显得代码臃肿。
- 使用 `super()` 可以逐一调用所有的父类方法，并且只执行一次。调用顺序遵循 `mro` 类属性的顺序。
- 注意：如果继承了多个父类，且父类都有同名方法，则默认只执行第一个父类的(同名方法只执行一次，目前 `super()` 不支持执行多个父类的同名方法)。
- `super ()` 在Python2.3之后才有的机制，用于通常单继承的多层继承。

多态

所谓多态：定义时的类型和运行时的类型不一样，此时就成为多态，多态的概念是应用于 Java 和 C# 这一类强类型语言中，而Python崇尚“鸭子类型”。

鸭子类型：虽然我想要一只“鸭子”，但是你给了我一只鸟。但是只要这只鸟走路像鸭子，叫起来像鸭子，游泳也像鸭子，我就认为这是鸭子。

Python的多态，就是弱化类型，重点在于对象参数是否有指定的属性和方法，如果有就认定合适，而不关心对象的类型是否正确。

```
class F1(object):
    def show(self):
        print('F1.show')

class S1(F1):
    def show(self):
        print('S1.show')

class S2(F1):
    def show(self):
        print('S2.show')

# 由于在Java或C#中定义函数参数时，必须指定参数的类型
# 为了让Func函数既可以执行S1对象的show方法，又可以执行S2对象的show方法，
# 所以在def Func的形参中obj的类型是 S1和S2的父类即F1
#
# 而实际传入的参数是：S1对象和S2对象

def Func(F1 obj):
    """Func函数需要接收一个F1类型或者F1子类的类型"""

    print(obj.show())

s1_obj = S1()
```

```
Func(s1_obj) # 在Func函数中传入S1类的对象 s1_obj, 执行 S1 的show方法,  
结果: S1.show
```

```
s2_obj = S2()  
Func(s2_obj) # 在Func函数中传入Ss类的对象 ss_obj, 执行 Ss 的show方法,  
结果: S2.show
```

理解：定义 `obj` 这个变量是说的类型是：F1的类型，但是在真正调用 `Func` 函数时给其传递的不一定是F1类的实例对象，有可能是其子类的实例对象，这种情况就是所谓的多态

```
Python “鸭子类型”  
class F1(object):  
    def show(self):  
        print('F1.show')  
  
class S1(F1):  
    def show(self):  
        print('S1.show')  
  
class S2(F1):  
    def show(self):  
        print('S2.show')  
  
def Func(obj):  
    # python是弱类型，即无论传递过来的是什么，obj变量都能够指向它，这也  
    就没有所谓的多态了（弱化了这个概念）  
    print(obj.show())  
  
s1_obj = S1()  
Func(s1_obj)  
  
s2_obj = S2()  
Func(s2_obj)
```

属性和方法

1. 类属性和实例属性

类属性就是类对象所拥有的属性，它被所有类对象的实例对象所共有，在内存中只存在一个副本，这个和C++中类的静态成员变量有点类似。对于公有的类属性，在类外可以通过类对象和实例对象访问

```
class People(object):
    name = 'Tom'  # 公有的类属性
    __age = 12    # 私有的类属性

p = People()

print(p.name)  # 正确
print(People.name)  # 正确
print(p.__age)  # 错误，不能在类外通过实例对象访问私有的类属性
print(People.__age) # 错误，不能在类外通过类对象访问私有的类属性
实例属性(对象属性)
class People(object):
    address = '山东'  # 类属性
    def __init__(self):
        self.name = 'xiaowang'  # 实例属性
        self.age = 20  # 实例属性

p = People()
p.age = 12  # 实例属性
print(p.address)  # 正确
print(p.name)  # 正确
print(p.age)  # 正确

print(People.address)  # 正确
print(People.name)  # 错误
print(People.age)  # 错误
```

```
#通过实例(对象)去修改类属性
class People(object):
    country = 'china' #类属性

print(People.country)
p = People()
print(p.country)
p.country = 'japan'
print(p.country) # 实例属性会屏蔽掉同名的类属性
print(People.country)
del p.country # 删除实例属性
print(p.country)
```

总结

如果需要在类外修改类属性，必须通过类对象去引用然后进行修改。如果通过实例对象去引用，会产生一个同名的实例属性，这种方式修改的是实例属性，不会影响到类属性，并且之后如果通过实例对象去引用该名称的属性，实例属性会强制屏蔽掉类属性，即引用的是实例属性，除非删除了该实例属性。

2. 静态方法和类方法

(1) 类方法

是类对象所拥有的方法，需要用修饰器@classmethod来标识其为类方法，对于类方法，第一个参数必须是类对象，一般以cls作为第一个参数（当然可以用其他名称的变量作为其第一个参数，但是大部分人都习惯以'cls'作为第一个参数的名字，就最好用'cls'了），能够通过实例对象和类对象去访问。

```
class People(object):
    country = 'china'

#类方法，用classmethod来进行修饰
@classmethod
def get_country(cls):
    return cls.country
```

```
p = People()
print(p.get_country())      #可以用过实例对象引用
print(People.get_country())  #可以通过类对象引用
类方法还有一个用途就是可以对类属性进行修改：

class People(object):
    country = 'china'

    #类方法，用classmethod来进行修饰
    @classmethod
    def get_country(cls):
        return cls.country

    @classmethod
    def set_country(cls,country):
        cls.country = country

p = People()
print(p.get_country())      #可以用过实例对象访问
print(People.get_country())  #可以通过类访问

p.set_country('japan')

print(p.get_country())
print(People.get_country())

#结果显示在用类方法对类属性修改之后，通过类对象和实例对象访问都发生了改变
```

(2) 静态方法

需要通过修饰器 `@staticmethod` 来进行修饰，静态方法不需要多定义参数，可以通过对象和类来访问。

```
class People(object):
    country = 'china'
```

```
@staticmethod  
#静态方法  
def get_country():  
    return People.country  
  
p = People()  
# 通过对象访问静态方法  
p.get_country()  
  
# 通过类访问静态方法  
print(People.get_country())
```

总结

从类方法和实例方法以及静态方法的定义形式就可以看出来，类方法的第一个参数是类对象cls，那么通过cls引用的必定是类对象的属性和方法；实例方法的第一个参数是实例对象self，那么通过self引用的可能是类属性、也有可能是实例属性（这个需要具体分析），不过在存在相同名称的类属性和实例属性的情况下，实例属性优先级更高。静态方法中不需要额外定义参数，因此在静态方法中引用类属性的话，必须通过类实例对象来引用

3. new方法

new和init的作用

```
class A(object):  
    def __init__(self):  
        print("这是 init 方法")  
  
    def __new__(cls):  
        print("这是 new 方法")  
        return object.__new__(cls)  
  
A()
```

总结

- **new**至少要有一个参数cls，代表要实例化的类，此参数在实例化时由Python解释器自动提供
- **new**必须要有返回值，返回实例化出来的实例，这点在自己实现**new**时要特别注意，可以return父类**new**出来的实例，或者直接是object的**new**出来的实例
- **init**有一个参数self，就是这个**new**返回的实例，**init**在**new**的基础上可以完成一些其它初始化的动作，**init**不需要返回值

对象的私有权限

面向对象三大特性：封装、继承、多态

封装的意义：

将属性和方法放到一起做为一个整体，然后通过实例化对象来处理；隐藏内部实现细节，只需要和对象及其属性和方法交互就可以了；对类的属性和方法增加访问权限控制。

1. 私有权限

在属性名和方法名前面加上两个下划线 __ 类的私有属性和私有方法，都不能通过对象直接访问，但是可以在本类内部访问；类的私有属性和私有方法，都不会被子类继承，子类也无法访问；私有属性和私有方法往往用来处理类的内部事情，不通过对象处理，起到安全作用。

```
class Master(object):
    def __init__(self):
        self.kongfu = "古法煎饼果子配方"
    def make_cake(self):
        print("[古法] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

class School(object):
    def __init__(self):
        self.kongfu = "现代煎饼果子配方"

    def make_cake(self):
        print("[现代] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

class Prentice(School, Master):
    def __init__(self):
        self.kongfu = "猫氏煎饼果子配方"
        # 私有属性，可以在类内部通过self调用，但不能通过对象访问
        self.__money = 10000
```

```
# 私有方法，可以在类内部通过self调用，但不能通过对象访问
def __print_info(self):
    print(self.kongfu)
    print(self.__money)

def make_cake(self):
    self.__init__()
    print("[猫氏] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

def make_old_cake(self):
    Master.__init__(self)
    Master.make_cake(self)

def make_new_cake(self):
    School.__init__(self)
    School.make_cake(self)

class PrenticePrentice(Prentice):
    pass

damao = Prentice()
# 对象不能访问私有权限的属性和方法
# print(damao.__money)
# damao.__print_info()

pp = PrenticePrentice()
# 子类不能继承父类私有权限的属性和方法
print(pp.__money)
pp.__print_info()
```

总结

- Python中没有像C++中 `public` 和 `private` 这些关键字来区别公有属性和私有属性。

- Python是以属性命名方式来区分，如果在属性和方法名前面加了2个下划线'__'，则表明该属性和方法是私有权限，否则为公有权限。

2. 修改私有属性的值

如果需要修改一个对象的属性值，通常有2种方法

| 对象名.属性名 = 数据 ----> 直接修改 对象名.方法名() ----> 间接修改

私有属性不能直接访问，所以无法通过第一种方式修改，一般的通过第二种方式修改私有属性的值：定义一个可以调用的公有方法，在这个公有方法内访问修改。

```
class Master(object):
    def __init__(self):
        self.kongfu = "古法煎饼果子配方"
    def make_cake(self):
        print("[古法] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

class School(object):
    def __init__(self):
        self.kongfu = "现代煎饼果子配方"

    def make_cake(self):
        print("[现代] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

class Prentice(School, Master):
    def __init__(self):
        self.kongfu = "猫氏煎饼果子配方"
        # 私有属性，可以在类内部通过self调用，但不能通过对象访问
        self.__money = 10000

    # 现代软件开发中，通常会定义get_xxx()方法和set_xxx()方法来获取和修改私有属性值。
    # 返回私有属性的值
    def get_money(self):
```

```
return self.__money

# 接收参数，修改私有属性的值
def set_money(self, num):
    self.__money = num


def make_cake(self):
    self.__init__()
    print("[猫氏] 按照 <%s> 制作了一份煎饼果子..." % self.kongfu)

def make_old_cake(self):
    Master.__init__(self)
    Master.make_cake(self)

def make_new_cake(self):
    School.__init__(self)
    School.make_cake(self)

class PrenticePrentice(Prentice):
    pass


damao = Prentice()
# 对象不能访问私有权限的属性和方法
# print(damao.__money)
# damao.__print_info()

# 可以通过访问公有方法set_money()来修改私有属性的值
damao.set_money(100)

# 可以通过访问公有方法get_money()来获取私有属性的值
print(damao.get_money())
```

单例模式

1. 单例是什么

举个常见的单例模式例子，我们日常使用的电脑上都有一个回收站，在整个操作系统中，回收站只能有一个实例，整个系统都使用这个唯一的实例，而且回收站自行提供自己的实例。因此回收站是单例模式的应用。

确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，单例模式是一种对象创建型模式。

2. 创建单例-保证只有1个对象

```
# 实例化一个单例
class Singleton(object):
    __instance = None

    def __new__(cls, age, name):
        #如果类属性__instance的值为None,
        #那么就创建一个对象，并且赋值为这个对象的引用，保证下次调用这个
方法时
        #能够知道之前已经创建过对象了，这样就保证了只有1个对象
        if not cls.__instance:
            cls.__instance = object.__new__(cls)
        return cls.__instance

a = Singleton(18, "dongGe")
b = Singleton(8, "dongGe")

print(id(a))
print(id(b))

a.age = 19 #给a指向的对象添加一个属性
print(b.age) #获取b指向的对象的age属性
运行结果：
```

```
In [12]: class Singleton(object):
...:     __instance = None
...:
...:     def __new__(cls, age, name):
...:         if not cls.__instance:
...:             cls.__instance = object.__new__(cls)
...:         return cls.__instance
...:
...: a = Singleton(18, "dongGe")
...: b = Singleton(8, "dongGe")
...:
...: print(id(a))
...: print(id(b))
...:
...: a.age = 19
...: print(b.age)
...:

4391023224
4391023224
19
```

3. 创建单例时，只执行1次init方法

```
# 实例化一个单例
class Singleton(object):
    __instance = None
    __is_first = True

    def __new__(cls, age, name):
        if not cls.__instance:
            cls.__instance = object.__new__(cls)
        return cls.__instance

    def __init__(self, age, name):
        if self.__is_first:
            self.age = age
```

```
self.name = name
Singleton. __is_first = False

a = Singleton(18, "习大大")
b = Singleton(28, "习大大")

print(id(a))
print(id(b))

print(a.age)
print(b.age)

a.age = 19
print(b.age)
```

使用slots

正常情况下，当我们定义了一个 `class`，创建了一个 `class` 的实例后，我们可以给该实例绑定任何属性和方法，这就是动态语言的灵活性。

先定义 `class`：

```
class Student(object):
    pass
```

然后，尝试给实例绑定一个属性：

```
>>> s = Student()
>>> s.name = 'Michael' # 动态给实例绑定一个属性
>>> print(s.name)
Michael
```

还可以尝试给实例绑定一个方法：

```
>>> def set_age(self, age): # 定义一个函数作为实例方法
...     self.age = age
...
>>> from types import MethodType
>>> s.set_age = MethodType(set_age, s) # 给实例绑定一个方法
>>> s.set_age(25) # 调用实例方法
>>> s.age # 测试结果
25
```

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

```
>>> s2 = Student() # 创建新的实例
>>> s2.set_age(25) # 尝试调用方法
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
AttributeError: 'Student' object has no attribute 'set_age'
```

为了给所有实例都绑定方法，可以给class绑定方法：

```
>>> def set_score(self, score):
...     self.score = score
...
>>> Student.set_score = set_score
```

给class绑定方法后，所有实例均可调用：

```
>>> s.set_score(100)
>>> s.score
100
>>> s2.set_score(99)
>>> s2.score
99
```

通常情况下，上面的 `set_score` 方法可以直接定义在class中，但动态绑定允许我们在程序运行的过程中动态给 `class` 加上功能，这在静态语言中很难实现。

但是，如果我们想要限制实例的属性怎么办？比如，只允许对 `Student` 实例添加 `name` 和 `age` 属性。

为了达到限制的目的，Python允许在定义class的时候，定义一个特殊的 `__slots__` 变量，来限制该class实例能添加的属性：

```
class Student(object):
    __slots__ = ('name', 'age') # 用tuple定义允许绑定的属性名称
```

然后，我们试试：

```
>>> s = Student() # 创建新的实例
>>> s.name = 'Michael' # 绑定属性'age'
```

```
>>> s.age = 25 # 绑定属性'age'  
>>> s.score = 99 # 绑定属性'score'  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
AttributeError: 'Student' object has no attribute 'score'
```

由于 'score' 没有被放到 `__slots__` 中，所以不能绑定 `score` 属性，试图绑定 `score` 将得到 `AttributeError` 的错误。

使用 `__slots__` 要注意，`__slots__` 定义的属性仅对当前类实例起作用，对继承的子类是不起作用的：

```
>>> class GraduateStudent(Student):  
...     pass  
...  
>>> g = GraduateStudent()  
>>> g.score = 9999
```

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__`。

使用@property

在绑定属性时，如果我们直接把属性暴露出去，虽然写起来很简单，但是，没办法检查参数，导致可以把成绩随便改：

```
s = Student()  
s.score = 9999
```

这显然不合逻辑。为了限制score的范围，可以通过一个 `set_score()` 方法来设置成绩，再通过一个 `get_score()` 来获取成绩，这样，在 `set_score()` 方法里，就可以检查参数：

```
class Student(object):  
  
    def get_score(self):  
        return self._score  
  
    def set_score(self, value):  
        if not isinstance(value, int):  
            raise ValueError('score must be an integer!')  
        if value < 0 or value > 100:  
            raise ValueError('score must between 0 ~ 100!')  
        self._score = value
```

现在，对任意的Student实例进行操作，就不能随心所欲地设置score了：

```
>>> s = Student()  
>>> s.set_score(60) # ok!  
>>> s.get_score()  
60  
>>> s.set_score(9999)  
Traceback (most recent call last):  
...  
...
```

```
ValueError: score must between 0 ~ 100!
```

但是，上面的调用方法又略显复杂，没有直接用属性这么直接简单。

有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的Python程序员来说，这是必须要做到的！

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python内置的 `@property` 装饰器就是负责把一个方法变成属性调用的：

```
class Student(object):

    @property
    def score(self):
        return self._score

    @score.setter
    def score(self, value):
        if not isinstance(value, int):
            raise ValueError('score must be an integer!')
        if value < 0 or value > 100:
            raise ValueError('score must between 0 ~ 100!')
        self._score = value
```

`@property` 的实现比较复杂，我们先考察如何使用。把一个getter方法变成属性，只需要加上 `@property` 就可以了，此时，`@property` 本身又创建了另一个装饰器 `@score.setter`，负责把一个setter方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

```
>>> s = Student()
>>> s.score = 60 # OK, 实际转化为s.set_score(60)
>>> s.score # OK, 实际转化为s.get_score()
60
>>> s.score = 9999
Traceback (most recent call last):
...
...
```

```
ValueError: score must between 0 ~ 100!
```

注意到这个神奇的 `@property`，我们在对实例属性操作的时候，就知道该属性很可能不是直接暴露的，而是通过getter和setter方法来实现的。

还可以定义只读属性，只定义getter方法，不定义setter方法就是一个只读属性：

```
class Student(object):

    @property
    def birth(self):
        return self._birth

    @birth.setter
    def birth(self, value):
        self._birth = value

    @property
    def age(self):
        return 2015 - self._birth
```

上面的 `birth` 是可读写属性，而 `age` 就是一个只读属性，因为 `age` 可以根据 `birth` 和当前时间计算出来。

小结：

`@property` 广泛应用在类的定义中，可以让调用者写出简短的代码，同时保证对参数进行必要的检查，这样，程序运行时就减少了出错的可能性。

使用枚举类

当我们需要定义常量时，一个办法是用大写变量通过整数来定义，例如月份：

```
JAN = 1
FEB = 2
MAR = 3
...
NOV = 11
DEC = 12
```

好处是简单，缺点是类型是 `int`，并且仍然是变量。

更好的方法是为这样的枚举类型定义一个`class`类型，然后，每个常量都是`class`的一个唯一实例。Python提供了 `Enum` 类来实现这个功能：

```
from enum import Enum

Month = Enum('Month', ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))
```

这样我们就获得了 `Month` 类型的枚举类，可以直接使用 `Month.Jan` 来引用一个常量，或者枚举它的所有成员：

```
for name, member in Month.__members__.items():
    print(name, '=>', member, ', ', member.value)
```

`value` 属性则是自动赋给成员的 `int` 常量，默认从 1 开始计数。

如果需要更精确地控制枚举类型，可以从 `Enum` 派生出自定义类：

```
from enum import Enum, unique
```

```
@unique
class Weekday(Enum):
    Sun = 0 # Sun的value被设定为0
    Mon = 1
    Tue = 2
    Wed = 3
    Thu = 4
    Fri = 5
    Sat = 6
```

`@unique` 装饰器可以帮助我们检查保证没有重复值。

访问这些枚举类型可以有若干种方法：

```
>>> day1 = Weekday.Mon
>>> print(day1)
Weekday.Mon
>>> print(Weekday.Tue)
Weekday.Tue
>>> print(Weekday['Tue'])
Weekday.Tue
>>> print(Weekday.Tue.value)
2
>>> print(day1 == Weekday.Mon)
True
>>> print(day1 == Weekday.Tue)
False
>>> print(Weekday(1))
Weekday.Mon
>>> print(day1 == Weekday(1))
True
>>> Weekday(7)
Traceback (most recent call last):
...
ValueError: 7 is not a valid Weekday
>>> for name, member in Weekday.__members__.items():
...     print(name, '=>', member)
...
```

```
Sun => Weekday.Sun
Mon => Weekday.Mon
Tue => Weekday.Tue
Wed => Weekday.Wed
Thu => Weekday.Thu
Fri => Weekday.Fri
Sat => Weekday.Sat
```

可见，既可以用成员名称引用枚举常量，又可以直接根据value的值获得枚举常量。

错误处理

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见。比如打开文件的函数 `open()`，成功时返回文件描述符（就是一个整数），出错时返回 `-1`。

用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错：

```
def foo():
    r = some_function()
    if r==(-1):
        return (-1)
    # do something
    return r

def bar():
    r = foo()
    if r==(-1):
        print('Error')
    else:
        pass
```

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套 `try...except...finally...` 的错误处理机制，Python也不例外。

1. try

让我们用一个例子来看看 `try` 的机制：

```
try:  
    print('try...')  
    r = 10 / 0  
    print('result:', r)  
except ZeroDivisionError as e:  
    print('except:', e)  
finally:  
    print('finally...')  
print('END')
```

当我们认为某些代码可能会出错时，就可以用 `try` 来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即 `except` 语句块，执行完 `except` 后，如果有 `finally` 语句块，则执行 `finally` 语句块，至此，执行完毕。

上面的代码在计算 `10 / 0` 时会产生一个除法运算错误：

```
try...  
except: division by zero  
finally...  
END
```

从输出可以看到，当错误发生时，后续语句 `print('result:', r)` 不会被执行，`except` 由于捕获到 `ZeroDivisionError`，因此被执行。最后，`finally` 语句被执行。然后，程序继续按照流程往下走。

如果把除数 `0` 改成 `2`，则执行结果如下：

```
try...  
result: 5  
finally...  
END
```

由于没有错误发生，所以 `except` 语句块不会被执行，但是 `finally` 如果有，则一定会被执行（可以没有 `finally` 语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的 `except` 语句块处理。没错，可以有多个 `except` 来捕获不同类型的错误：

```
try:  
    print('try...')  
    r = 10 / int('a')  
    print('result:', r)  
except ValueError as e:  
    print('ValueError:', e)  
except ZeroDivisionError as e:  
    print('ZeroDivisionError:', e)  
finally:  
    print('finally...')  
print('END')
```

`int()` 函数可能会抛出 `ValueError`，所以我们用一个 `except` 捕获 `ValueError`，用另一个 `except` 捕获 `ZeroDivisionError`。

此外，如果没有错误发生，可以在 `except` 语句块后面加一个 `else`，当没有错误发生时，会自动执行 `else` 语句：

```
try:  
    print('try...')  
    r = 10 / int('2')  
    print('result:', r)  
except ValueError as e:  
    print('ValueError:', e)  
except ZeroDivisionError as e:  
    print('ZeroDivisionError:', e)  
else:  
    print('no error!')  
finally:  
    print('finally...')  
print('END')
```

Python的错误其实也是 `class`，所有的错误类型都继承自 `BaseException`，所以在使用 `except` 时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:  
    foo()  
except ValueError as e:  
    print('ValueError')  
except UnicodeError as e:  
    print('UnicodeError')
```

第二个 `except` 永远也捕获不到 `UnicodeError`，因为 `UnicodeError` 是 `ValueError` 的子类，如果有，也被第一个 `except` 给捕获了。

Python所有的错误都是从 `BaseException` 类派生的，常见的错误类型和继承关系看这里：

<https://docs.python.org/3/library/exceptions.html#exception-hierarchy>

使用 `try...except` 捕获错误还有一个巨大的好处，就是可以跨多层调用，比如函数 `main()` 调用 `foo()`，`foo()` 调用 `bar()`，结果 `bar()` 出错了，这时，只要 `main()` 捕获到了，就可以处理：

```
def foo(s):  
    return 10 / int(s)  
  
def bar(s):  
    return foo(s) * 2  
  
def main():  
    try:  
        bar('0')  
    except Exception as e:  
        print('Error:', e)  
    finally:  
        print('finally...')
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写 `try...except...finally` 的麻烦。

2. 调用栈

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。来看看 `err.py`：

```
# err.py:  
def foo(s):  
    return 10 / int(s)  
  
def bar(s):  
    return foo(s) * 2  
  
def main():  
    bar('0')  
  
main()
```

执行，结果如下：

```
$ python3 err.py  
Traceback (most recent call last):  
  File "err.py", line 11, in <module>  
    main()  
  File "err.py", line 9, in main  
    bar('0')  
  File "err.py", line 6, in bar  
    return foo(s) * 2  
  File "err.py", line 3, in foo  
    return 10 / int(s)  
ZeroDivisionError: division by zero
```

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：

错误信息第1行：

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

告诉我们这是错误的跟踪信息。

第2~3行：

```
  File "err.py", line 11, in <module>
    main()
```

调用 `main()` 出错了，在代码文件 `err.py` 的第11行代码，但原因是第9行：

```
  File "err.py", line 9, in main
    bar('0')
```

调用 `bar('0')` 出错了，在代码文件 `err.py` 的第9行代码，但原因是第6行：

```
  File "err.py", line 6, in bar
    return foo(s) * 2
```

原因是 `return foo(s) * 2` 这个语句出错了，但这还不是最终原因，继续往下看：

```
  File "err.py", line 3, in foo
    return 10 / int(s)
```

原因是 `return 10 / int(s)` 这个语句出错了，这是错误产生的源头，因为下面打印了：

```
ZeroDivisionError: integer division or modulo by zero
```

根据错误类型 `ZeroDivisionError`，我们判断，`int(s)` 本身并没有出错，但是 `int(s)` 返回 `0`，在计算 `10 / 0` 时出错，至此，找到错误源头。

出错的时候，一定要分析错误的调用栈信息，才能定位错误的位置。



3. 记录错误

如果不捕获错误，自然可以让Python解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python内置的 `logging` 模块可以非常容易地记录错误信息：

```
# err_logging.py

import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except Exception as e:
        logging.exception(e)
```

```
main()
print('END')
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python3 err_logging.py
ERROR:root:division by zero
Traceback (most recent call last):
  File "err_logging.py", line 13, in main
    bar('0')
  File "err_logging.py", line 9, in bar
    return foo(s) * 2
  File "err_logging.py", line 6, in foo
    return 10 / int(s)
ZeroDivisionError: division by zero
END
```

通过配置，`logging` 还可以把错误记录到日志文件里，方便事后排查。

4. 抛出错误

因为错误是 `class`，捕获一个错误就是捕获到该 `class` 的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。

如果要抛出错误，首先根据需要，可以定义一个错误的 `class`，选择好继承关系，然后，用 `raise` 语句抛出一个错误的实例：

```
# err_raise.py
class FooError(ValueError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
```

```
return 10 / n

foo('0')
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python3 err_raise.py
Traceback (most recent call last):
  File "err_throw.py", line 11, in <module>
    foo('0')
  File "err_throw.py", line 8, in foo
    raise FooError('invalid value: %s' % s)
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择Python已有的内置的错误类型（比如 `ValueError` , `TypeError` ），尽量使用Python内置的错误类型。

最后，我们来看另一种错误处理的方式：

```
# err_reraise.py

def foo(s):
    n = int(s)
    if n==0:
        raise ValueError('invalid value: %s' % s)
    return 10 / n

def bar():
    try:
        foo('0')
    except ValueError as e:
        print('ValueError!')
        raise

bar()
```

在 `bar()` 函数中，我们明明已经捕获了错误，但是，打印一个 `ValueError!` 后，又把错误通过 `raise` 语句抛出去了，这不有病么？

其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。好比一个员工处理不了一个问题时，就把问题抛给他的老板，如果他的老板也处理不了，就一直往上抛，最终会抛给CEO去处理。

`raise` 语句如果不带参数，就会把当前错误原样抛出。此外，在 `except` 中 `raise` 一个Error，还可以把一种类型的错误转化成另一种类型：

```
try:  
    10 / 0  
except ZeroDivisionError:  
    raise ValueError('input error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个 `IOError` 转换成毫不相干的 `ValueError`。

调试

程序能一次写完并正常运行的概率很小，基本不超过1%。总会有各种各样的bug需要修正。有的bug很简单，看看错误信息就知道，有的bug很复杂，我们需要知道出错时，哪些变量的值是正确的，哪些变量的值是错误的，因此，需要一整套调试程序的手段来修复bug。

1. print()

第一种方法简单直接粗暴有效，就是用 `print()` 把可能有问题的变量打印出来看看：

```
def foo(s):
    n = int(s)
    print('>>> n = %d' % n)
    return 10 / n

def main():
    foo('0')

main()
```

执行后在输出中查找打印的变量值：

```
$ python err.py
>>> n = 0
Traceback (most recent call last):
...
ZeroDivisionError: integer division or modulo by zero
```

用 `print()` 最大的坏处是将来还得删掉它，想想程序里到处都是 `print()`，运行结果也会包含很多垃圾信息。所以，我们又有第二种方法。

2. 断言

凡是用 `print()` 来辅助查看的地方，都可以用断言（`assert`）来替代：

```
def foo(s):
    n = int(s)
    assert n != 0, 'n is zero!'
    return 10 / n

def main():
    foo('0')
```

`assert` 的意思是，表达式 `n != 0` 应该是 `True`，否则，根据程序运行的逻辑，后面的代码肯定会出现错误。

如果断言失败，`assert` 语句本身就会抛出 `AssertionError`：

```
$ python err.py
Traceback (most recent call last):
...
AssertionError: n is zero!
```

程序中如果到处充斥着 `assert`，和 `print()` 相比也好不到哪去。不过，启动 Python 解释器时可以用 `-O` 参数来关闭 `assert`：

```
$ python -O err.py
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

关闭后，你可以把所有的 `assert` 语句当成 `pass` 来看。

3. logging

把 `print()` 替换为 `logging` 是第3种方式，和 `assert` 比，`logging` 不会抛出错误，而且可以输出到文件：

```
import logging

s = '0'
n = int(s)
logging.info('n = %d' % n)
print(10 / n)
```

`logging.info()` 就可以输出一段文本。运行，发现除了 `ZeroDivisionError`，没有任何信息。怎么回事？

别急，在 `import logging` 之后添加一行配置再试试：

```
import logging
logging.basicConfig(level=logging.INFO)
```

看到输出了：

```
$ python err.py
INFO:root:n = 0
Traceback (most recent call last):
  File "err.py", line 8, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这就是 `logging` 的好处，它允许你指定记录信息的级别，有 `debug`，`info`，`warning`，`error` 等几个级别，当我们指定 `level=INFO` 时，`logging.debug` 就不起作用了。同理，指定 `level=WARNING` 后，`debug` 和 `info` 就不起作用了。这样一来，你可以放心地输出不同级别的信息，也不用删除，最后统一控制输出哪个级别的信息。

`logging` 的另一个好处是通过简单的配置，一条语句可以同时输出到不同的地方，比如 `console` 和文件。

4. pdb

第4种方式是启动Python的调试器 `pdb`，让程序以单步方式运行，可以随时查看运行状态。我们先准备好程序：

```
# err.py
s = '0'
n = int(s)
print(10 / n)
```

然后启动：

```
$ python -m pdb err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(2)<module>()
-> s = '0'
```

以参数 `-m pdb` 启动后，`pdb`定位到下一步要执行的代码 `-> s = '0'`。输入命令 `l` 来查看代码：

```
(Pdb) l
1      # err.py
2 -> s = '0'
3      n = int(s)
4      print(10 / n)
```

输入命令 `n` 可以单步执行代码：

```
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(3)<module>()
-> n = int(s)
(Pdb) n
> /Users/michael/Github/learn-python3/samples/debug/err.py(4)<module>()
-> print(10 / n)
```

任何时候都可以输入命令 `p 变量名` 来查看变量：

```
(Pdb) p s
'0'
(Pdb) p n
0
```

输入命令 `q` 结束调试，退出程序：

```
(Pdb) q
```

这种通过pdb在命令行调试的方法理论上是万能的，但实在是太麻烦了，如果有上千行代码，要运行到第999行得敲多少命令啊。还好，我们还有另一种调试方法。

5. `pdb.set_trace()`

这个方法也是用pdb，但是不需要单步执行，我们只需要 `import pdb`，然后，在可能出错的地方放一个 `pdb.set_trace()`，就可以设置一个断点：

```
# err.py
import pdb

s = '0'
n = int(s)
pdb.set_trace() # 运行到这里会自动暂停
print(10 / n)
```

运行代码，程序会自动在 `pdb.set_trace()` 暂停并进入pdb调试环境，可以用命令 `p` 查看变量，或者用命令 `c` 继续运行：

```
$ python err.py
> /Users/michael/Github/learn-python3/samples/debug/err.py(7)<module
e>()
-> print(10 / n)
```

```
(Pdb) p n
0
(Pdb) c
Traceback (most recent call last):
  File "err.py", line 7, in <module>
    print(10 / n)
ZeroDivisionError: division by zero
```

这种方式比直接启动pdb单步调试效率要高很多，但也高不到哪去。

6. IDE

如果要比较爽地设置断点、单步执行，就需要一个支持调试功能的IDE。目前比较好的Python IDE有：

Visual Studio Code: <https://code.visualstudio.com/>，需要安装Python插件。

PyCharm: <http://www.jetbrains.com/pycharm/>

另外，[Eclipse](#)加上[pydev](#)插件也可以调试Python程序。

小结

- 写程序最痛苦的事情莫过于调试，程序往往会以你意想不到的流程来运行，你期待执行的语句其实根本没有执行，这时候，就需要调试了。
- 虽然用IDE调试起来比较方便，但是最后你会发现，logging才是终极武器。

单元测试

1. 单元测试

如果你听说过“测试驱动开发”（TDD：Test-Driven Development），单元测试就不陌生。

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数 `abs()`，我们可以编写出以下几个测试用例：

1. 输入正数，比如 `1`、`1.2`、`0.99`，期待返回值与输入相同；
2. 输入负数，比如 `-1`、`-1.2`、`-0.99`，期待返回值与输入相反；
3. 输入 `0`，期待返回 `0`；
4. 输入非数值类型，比如 `None`、`[]`、`{}`，期待抛出 `TypeError`。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对 `abs()` 函数代码做了修改，只需要再跑一遍单元测试，如果通过，说明我们的修改不会对 `abs()` 函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

我们来编写一个 `Dict` 类，这个类的行为和 `dict` 一致，但是可以通过属性来访问，用起来就像下面这样：

```
>>> d = Dict(a=1, b=2)
>>> d['a']
1
>>> d.a
1
```

`mydict.py` 代码如下：

```
class Dict(dict):

    def __init__(self, **kw):
        super().__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value
```

为了编写单元测试，我们需要引入Python自带的 `unittest` 模块，编写 `mydict_test.py` 如下：

```
import unittest

from mydict import Dict

class TestDict(unittest.TestCase):

    def test_init(self):
        d = Dict(a=1, b='test')
        self.assertEqual(d.a, 1)
        self.assertEqual(d.b, 'test')
```

```
self.assertTrue(isinstance(d, dict))

def test_key(self):
    d = Dict()
    d['key'] = 'value'
    self.assertEqual(d.key, 'value')

def test_attr(self):
    d = Dict()
    d.key = 'value'
    self.assertTrue('key' in d)
    self.assertEqual(d['key'], 'value')

def test_keyerror(self):
    d = Dict()
    with self.assertRaises(KeyError):
        value = d['empty']

def test_attrerror(self):
    d = Dict()
    with self.assertRaises(AttributeError):
        value = d.empty
```

编写单元测试时，我们需要编写一个测试类，从 `unittest.TestCase` 继承。

以 `test` 开头的方法就是测试方法，不以 `test` 开头的方法不被认为是测试方法，测试的时候不会被执行。

对每一类测试都需要编写一个 `test_xxx()` 方法。由于 `unittest.TestCase` 提供了很多内置的条件判断，我们只需要调用这些方法就可以断言输出是否是我们所期望的。最常用的断言就是 `assertEqual()`：

```
self.assertEqual(abs(-1), 1) # 断言函数返回的结果与1相等
```

另一种重要的断言就是期待抛出指定类型的Error，比如通过 `d['empty']` 访问不存在的key时，断言会抛出 `KeyError`：

```
with self.assertRaises(KeyError):
    value = d['empty']
```

而通过 `d.empty` 访问不存在的key时，我们期待抛出 `AttributeError`：

```
with self.assertRaises(AttributeError):
    value = d.empty
```

2. 运行单元测试

一旦编写好单元测试，我们就可以运行单元测试。最简单的运行方式是在 `mydict_test.py` 的最后加上两行代码：

```
if __name__ == '__main__':
    unittest.main()
```

这样就可以把 `mydict_test.py` 当做正常的python脚本运行：

```
$ python mydict_test.py
```

另一种方法是在命令行通过参数 `-m unittest` 直接运行单元测试：

```
$ python -m unittest mydict_test
.....
-----
---
Ran 5 tests in 0.000s

OK
```

这是推荐的做法，因为这样可以一次批量运行很多单元测试，并且，有很多工具可以自动来运行这些单元测试。

3. setUp与tearDown

可以在单元测试中编写两个特殊的 `setUp()` 和 `tearDown()` 方法。这两个方法会分别在每调用一个测试方法的前后分别被执行。

`setUp()` 和 `tearDown()` 方法有什么用呢？设想你的测试需要启动一个数据库，这时，就可以在 `setUp()` 方法中连接数据库，在 `tearDown()` 方法中关闭数据库，这样，不必在每个测试方法中重复相同的代码：

```
class TestDict(unittest.TestCase):

    def setUp(self):
        print('setUp...')

    def tearDown(self):
        print('tearDown...')
```

可以再次运行测试看看每个测试方法调用前后是否会打印出 `setUp...` 和 `tearDown...`。

文档测试

如果你经常阅读Python的官方文档，可以看到很多文档都有示例代码。比如[re模块](#)就带了很多示例代码：

```
>>> import re
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
```

可以把这些示例代码在Python的交互式环境下输入并执行，结果与文档中的示例代码显示的一致。

这些代码与其他说明可以写在注释中，然后，由一些工具来自动生成文档。既然这些代码本身就可以粘贴出来直接运行，那么，可不可以自动执行写在注释中的这些代码呢？

答案是肯定的。

当我们编写注释时，如果写上这样的注释：

```
def abs(n):
    """
    Function to get absolute value of number.

    Example:

    >>> abs(1)
    1
    >>> abs(-1)
    1
    >>> abs(0)
    0
    ...

    return n if n >= 0 else (-n)
```

无疑更明确地告诉函数的调用者该函数的期望输入和输出。

并且，Python内置的“文档测试”（doctest）模块可以直接提取注释中的代码并执行测试。

doctest 严格按照Python交互式命令行的输入和输出来判断测试结果是否正确。只有测试异常的时候，可以用 ... 表示中间一大段烦人的输出。

让我们用 doctest 来测试上次编写的 Dict 类：

```
# mydict2.py
class Dict(dict):
    ...
    Simple dict but also support access as x.y style.

    >>> d1 = Dict()
    >>> d1['x'] = 100
    >>> d1.x
    100
    >>> d1.y = 200
    >>> d1['y']
    200
    >>> d2 = Dict(a=1, b=2, c='3')
    >>> d2.c
    '3'
    >>> d2['empty']
    Traceback (most recent call last):
    ...
    KeyError: 'empty'
    >>> d2.empty
    Traceback (most recent call last):
    ...
    AttributeError: 'Dict' object has no attribute 'empty'
    ...
    def __init__(self, **kw):
        super(Dict, self).__init__(**kw)
```

```
def __getattr__(self, key):
    try:
        return self[key]
    except KeyError:
        raise AttributeError(r"'Dict' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

运行 `python mydict2.py` :

```
$ python mydict2.py
```

什么输出也没有。这说明我们编写的 `doctest` 运行都是正确的。如果程序有问题，比如把 `__getattr__()` 方法注释掉，再运行就会报错：

```
$ python mydict2.py
*****
***  
File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py"
, line 10, in __main__.Dict
Failed example:  
    d1.x
Exception raised:  
    Traceback (most recent call last):  
    ...
AttributeError: 'Dict' object has no attribute 'x'  

*****
***  
File "/Users/michael/Github/learn-python3/samples/debug/mydict2.py"
, line 16, in __main__.Dict
Failed example:
```

```
d2.c
Exception raised:
  Traceback (most recent call last):
    ...
AttributeError: 'Dict' object has no attribute 'c'
*****
*** 1 items had failures:
  2 of  9 in __main__.Dict
***Test Failed*** 2 failures.
```

注意到最后3行代码。当模块正常导入时，`doctest` 不会被执行。只有在命令行直接运行时，才执行 `doctest`。所以，不必担心 `doctest` 会在非测试环境下执行。

文件读写

读写文件是最常见的IO操作。Python内置了读写文件的函数，用法和C是兼容的。

读写文件前，我们先必须了解一下，在磁盘上读写文件的功能都是由操作系统提供的，现代操作系统不允许普通的程序直接操作磁盘，所以，读写文件就是请求操作系统打开一个文件对象（通常称为文件描述符），然后，通过操作系统提供的接口从这个文件对象中读取数据（读文件），或者把数据写入这个文件对象（写文件）。

1. 读文件

要以读文件的模式打开一个文件对象，使用Python内置的 `open()` 函数，传入文件名和标示符：

```
>>> f = open('/Users/michael/test.txt', 'r')
```

标示符'r'表示读，这样，我们就成功地打开了一个文件。

如果文件不存在，`open()` 函数就会抛出一个 `IOError` 的错误，并且给出错误码和详细的信息告诉你文件不存在：

```
>>> f=open('/Users/michael/notfound.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '/Users/michael/notfound.txt'
```

如果文件打开成功，接下来，调用 `read()` 方法可以一次读取文件的全部内容，Python把内容读到内存，用一个 `str` 对象表示：

```
>>> f.read()
```

```
'Hello, world!'
```

最后一步是调用 `close()` 方法关闭文件。文件使用完毕后必须关闭，因为文件对象会占用操作系统的资源，并且操作系统同一时间能打开的文件数量也是有限的：

```
>>> f.close()
```

由于文件读写时都有可能产生 `IOError`，一旦出错，后面的 `f.close()` 就不会调用。所以，为了保证无论是否出错都能正确地关闭文件，我们可以使用 `try ... finally` 来实现：

```
try:  
    f = open('/path/to/file', 'r')  
    print(f.read())  
finally:  
    if f:  
        f.close()
```

但是每次都这么写实在太繁琐，所以，Python引入了 `with` 语句来自动帮我们调用 `close()` 方法：

```
with open('/path/to/file', 'r') as f:  
    print(f.read())
```

这和前面的 `try ... finally` 是一样的，但是代码更佳简洁，并且不必调用 `f.close()` 方法。

调用 `read()` 会一次性读取文件的全部内容，如果文件有10G，内存就爆了，所以，要保险起见，可以反复调用 `read(size)` 方法，每次最多读取size个字节的内容。另外，调用 `readline()` 可以每次读取一行内容，调用 `readlines()` 一次读取所有内容并按行返回 `list`。因此，要根据需要决定怎么调用。

如果文件很小，`read()` 一次性读取最方便；如果不能确定文件大小，反复调用 `read(size)` 比较保险；如果是配置文件，调用 `readlines()` 最方便：

```
for line in f.readlines():
    print(line.strip()) # 把末尾的'\n'删掉
```

2. file-like Object

像 `open()` 函数返回的这种有个 `read()` 方法的对象，在Python中统称为 `file-like Object`。除了file外，还可以是内存的字节流，网络流，自定义流等等。`file-like Object` 不要求从特定类继承，只要写个 `read()` 方法就行。

`StringIO` 就是在内存中创建的 `file-like Object`，常用作临时缓冲。

3. 二进制文件

前面讲的默认都是读取文本文件，并且是UTF-8编码的文本文件。要读取二进制文件，比如图片、视频等等，用 '`rb`' 模式打开文件即可：

```
>>> f = open('/Users/michael/test.jpg', 'rb')
>>> f.read()
b'\xff\xd8\xff\xe1\x00\x18Exif\x00\x00...' # 十六进制表示的字节
```

4. 字符编码

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

```
>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk')
>>> f.read()
'测试'
```

遇到有些编码不规范的文件，你可能会遇到 `UnicodeDecodeError`，因为在文本文件中可能夹杂了一些非法编码的字符。遇到这种情况，`open()` 函数还接收一个 `errors` 参数，表示如果遇到编码错误后如何处理。最简单的方式是直接

忽略：

```
>>> f = open('/Users/michael/gbk.txt', 'r', encoding='gbk', errors='ignore')
```

5. 写文件

写文件和读文件是一样的，唯一区别是调用 `open()` 函数时，传入标识符 '`w`' 或者 '`wb`' 表示写文本文件或写二进制文件：

```
>>> f = open('/Users/michael/test.txt', 'w')
>>> f.write('Hello, world!')
>>> f.close()
```

你可以反复调用 `write()` 来写入文件，但是务必要调用 `f.close()` 来关闭文件。当我们写文件时，操作系统往往不会立刻把数据写入磁盘，而是放到内存缓存起来，空闲的时候再慢慢写入。只有调用 `close()` 方法时，操作系统才保证把没有写入的数据全部写入磁盘。忘记调用 `close()` 的后果是数据可能只写了一部分到磁盘，剩下的丢失了。所以，还是用 `with` 语句来得保险：

```
with open('/Users/michael/test.txt', 'w') as f:
    f.write('Hello, world!')
```

要写入特定编码的文本文件，请给 `open()` 函数传入 `encoding` 参数，将字符串自动转换成指定编码。

注意：以 '`w`' 模式写入文件时，如果文件已存在，会直接覆盖（相当于删掉后新写入一个文件）。如果我们希望追加到文件末尾怎么办？可以传入 '`a`' 以追加（append）模式写入。

StringIO和BytesIO

1. StringIO

很多时候，数据读写不一定是文件，也可以在内存中读写。

StringIO顾名思义就是在内存中读写 str。

要把 str 写入 StringIO，我们需要先创建一个 StringIO，然后，像文件一样写入即可：

```
>>> from io import StringIO
>>> f = StringIO()
>>> f.write('hello')
5
>>> f.write(' ')
1
>>> f.write('world!')
6
>>> print(f.getvalue())
hello world!
```

getvalue() 方法用于获得写入后的 str。

要读取 StringIO，可以用一个 str 初始化 StringIO，然后，像读文件一样读取：

```
>>> from io import StringIO
>>> f = StringIO('Hello!\nHi!\nGoodbye!')
>>> while True:
...     s = f.readline()
...     if s == '':
...         break
...     print(s.strip())
...
```

```
Hello!  
Hi!  
Goodbye!
```

2. BytesIO

`StringIO` 操作的只能是 `str`，如果要操作二进制数据，就需要使用 `BytesIO`。

`BytesIO` 实现了在内存中读写 `bytes`，我们创建一个 `BytesIO`，然后写入一些 `bytes`：

```
>>> from io import BytesIO  
>>> f = BytesIO()  
>>> f.write('中文'.encode('utf-8'))  
6  
>>> print(f.getvalue())  
b'\xe4\xb8\xad\xe6\x96\x87'
```

请注意，写入的不是 `str`，而是经过UTF-8编码的 `bytes`。

和 `StringIO` 类似，可以用一个 `bytes` 初始化 `BytesIO`，然后，像读文件一样读取：

```
>>> from io import BytesIO  
>>> f = BytesIO(b'\xe4\xb8\xad\xe6\x96\x87')  
>>> f.read()  
b'\xe4\xb8\xad\xe6\x96\x87'
```

小结

`StringIO`和`BytesIO`是在内存中操作 `str` 和 `bytes` 的方法，使得和读写文件具有一致的接口。

操作文件和目录

如果我们要操作文件、目录，可以在命令行下面输入操作系统提供的各种命令来完成。比如 `dir` 、 `cp` 等命令。

如果要在Python程序中执行这些目录和文件的操作怎么办？其实操作系统提供的命令只是简单地调用了操作系统提供的接口函数，Python内置的 `os` 模块也可以直接调用操作系统提供的接口函数。

打开Python交互式命令行，我们来看看如何使用 `os` 模块的基本功能：

```
>>> import os  
>>> os.name # 操作系统类型  
'posix'
```

如果是 `posix`，说明系统是 `Linux` 、 `Unix` 或 `Mac OS X`，如果是 `nt`，就是 `Windows` 系统。

要获取详细的系统信息，可以调用 `uname()` 函数：

```
>>> os.uname()  
posix.uname_result(sysname='Darwin', nodename='MichaelMacPro.local',  
, release='14.3.0', version='Darwin Kernel Version 14.3.0: Mon Mar  
23 11:59:05 PDT 2015; root:xnu-2782.20.48~5/RELEASE_X86_64', machin  
e='x86_64')
```

注意 `uname()` 函数在Windows上不提供，也就是说，`os` 模块的某些函数是跟操作系统相关的。

1. 环境变量

在操作系统中定义的环境变量，全部保存在 `os.environ` 这个变量中，可以直接查看：

```
>>> os.environ  
environ({'VERSIONER_PYTHON_PREFER_32_BIT': 'no', 'TERM_PROGRAM_VERS  
ION': '326', 'LOGNAME': 'michael', 'USER': 'michael', 'PATH': '/usr  
/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/local/my  
sql/bin', ...})
```

要获取某个环境变量的值，可以调用 `os.environ.get('key')`：

```
>>> os.environ.get('PATH')  
'/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/opt/X11/bin:/usr/loc  
al/mysql/bin'  
>>> os.environ.get('x', 'default')  
'default'
```

2. 操作文件和目录

操作文件和目录的函数一部分放在 `os` 模块中，一部分放在 `os.path` 模块中，这一点要注意一下。查看、创建和删除目录可以这么调用：

```
# 查看当前目录的绝对路径:  
>>> os.path.abspath('.').  
'/Users/michael'  
# 在某个目录下创建一个新目录，首先把新目录的完整路径表示出来:  
>>> os.path.join('/Users/michael', 'testdir')  

```

把两个路径合成一个时，不要直接拼字符串，而要通过 `os.path.join()` 函数，这样可以正确处理不同操作系统的路径分隔符。在Linux/Unix/Mac下，`os.path.join()` 返回这样的字符串：

part-1/part-2

而Windows下会返回这样的字符串：

```
part-1\part-2
```

同样的道理，要拆分路径时，也不要直接去拆字符串，而要通过 `os.path.split()` 函数，这样可以把一个路径拆分为两部分，后一部分总是最后级别的目录或文件名：

```
>>> os.path.split('/Users/michael/testdir/file.txt')
('/Users/michael/testdir', 'file.txt')
```

`os.path.splitext()` 可以直接让你得到文件扩展名，很多时候非常方便：

```
>>> os.path.splitext('/path/to/file.txt')
('/path/to/file', '.txt')
```

这些合并、拆分路径的函数并不要求目录和文件要真实存在，它们只对字符串进行操作。

文件操作使用下面的函数。假定当前目录下有一个 `test.txt` 文件：

```
# 对文件重命名：
>>> os.rename('test.txt', 'test.py')
# 删掉文件：
>>> os.remove('test.py')
```

但是复制文件的函数居然在 `os` 模块中不存在！原因是复制文件并非由操作系统提供的系统调用。理论上讲，我们通过上一节的读写文件可以完成文件复制，只不过要多写很多代码。

幸运的是 `shutil` 模块提供了 `copyfile()` 的函数，你还可以在 `shutil` 模块中找到很多实用函数，它们可以看做是 `os` 模块的补充。

最后看看如何利用Python的特性来过滤文件。比如我们要列出当前目录下的所有目录，只需要一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isdir(x)]
['.lein', '.local', '.m2', '.npm', '.ssh', '.Trash', '.vim', 'Applications', 'Desktop', ...]
```

要列出所有的 `.py` 文件，也只需一行代码：

```
>>> [x for x in os.listdir('.') if os.path.isfile(x) and os.path.splitext(x)[1]=='.py']
['apis.py', 'config.py', 'models.py', 'pymonitor.py', 'test_db.py',
 'urls.py', 'wsgiapp.py']
```

小结

Python的 `os` 模块封装了操作系统的目录和文件操作，要注意这些函数有的在 `os` 模块中，有的在 `os.path` 模块中。

序列化

1. 序列化

在程序运行的过程中，所有的变量都是在内存中，比如，定义一个 `dict`：

```
d = dict(name='Bob', age=20, score=88)
```

可以随时修改变量，比如把 `name` 改成 `'Bill'`，但是一旦程序结束，变量所占用的内存就被操作系统全部回收。如果没有把修改后的 `'Bill'` 存储到磁盘上，下次重新运行程序，变量又被初始化为 `'Bob'`。

我们把变量从内存中变成可存储或传输的过程称之为序列化，在Python中叫 `pickling`，在其他语言中也称之为 `serialization, marshalling, flattening` 等等，都是一个意思。

序列化之后，就可以把序列化后的内容写入磁盘，或者通过网络传输到别的机器上。

反过来，把变量内容从序列化的对象重新读到内存里称之为反序列化，即 `unpickling`。

Python提供了 `pickle` 模块来实现序列化。

首先，我们尝试把一个对象序列化并写入文件：

```
>>> import pickle
>>> d = dict(name='Bob', age=20, score=88)
>>> pickle.dumps(d)
b'\x80\x03}q\x00(X\x03\x00\x00\x00ageq\x01K\x14X\x05\x00\x00\x00scoreq\x02KXX\x04\x00\x00\x00nameq\x03X\x03\x00\x00\x00Bobq\x04u.'
```

`pickle.dumps()` 方法把任意对象序列化成一个 `bytes`，然后，就可以把这个 `bytes` 写入文件。或者用另一个方法 `pickle.dump()` 直接把对象序列化后写入一个 `file-like Object`：

```
>>> f = open('dump.txt', 'wb')
>>> pickle.dump(d, f)
>>> f.close()
```

看看写入的 `dump.txt` 文件，一堆乱七八糟的内容，这些都是Python保存的对象内部信息。

当我们要把对象从磁盘读到内存时，可以先把内容读到一个 `bytes`，然后用 `pickle.loads()` 方法反序列化出对象，也可以直接用 `pickle.load()` 方法从一个 `file-like Object` 中直接反序列化出对象。我们打开另一个Python命令行来反序列化刚才保存的对象：

```
>>> f = open('dump.txt', 'rb')
>>> d = pickle.load(f)
>>> f.close()
>>> d
{'age': 20, 'score': 88, 'name': 'Bob'}
```

变量的内容又回来了！

当然，这个变量和原来的变量是完全不相干的对象，它们只是内容相同而已。

`Pickle` 的问题和所有其他编程语言特有的序列化问题一样，就是它只能用于 Python，并且可能不同版本的Python彼此都不兼容，因此，只能用Pickle保存那些不重要的数据，不能成功地反序列化也没关系。

2. JSON

如果我们要在不同的编程语言之间传递对象，就必须把对象序列化为标准格式，比如XML，但更好的方法是序列化为JSON，因为JSON表示出来就是一个字符串，可以被所有语言读取，也可以方便地存储到磁盘或者通过网络传输。

JSON不仅是标准格式，并且比XML更快，而且可以直接在Web页面中读取，非常方便。

JSON表示的对象就是标准的JavaScript语言的对象，JSON和Python内置的数据类型对应如下：

JSON类型	Python类型
{}	dict
[]	list
"string"	str
1234.56	int或float
true/false	True/False
null	None

Python内置的 `json` 模块提供了非常完善的Python对象到JSON格式的转换。我们先看看如何把Python对象变成一个JSON：

```
>>> import json
>>> d = dict(name='Bob', age=20, score=88)
>>> json.dumps(d)
'{"age": 20, "score": 88, "name": "Bob"}'
```

`dumps()` 方法返回一个 `str`，内容就是标准的JSON。类似的，`dump()` 方法可以直接把JSON写入一个 `file-like Object`。

要把JSON反序列化为Python对象，用 `loads()` 或者对应的 `load()` 方法，前者把JSON的字符串反序列化，后者从 `file-like Object` 中读取字符串并反序列化：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'
>>> json.loads(json_str)
{'age': 20, 'score': 88, 'name': 'Bob'}
```

由于JSON标准规定JSON编码是UTF-8，所以我们总是能正确地在Python的 `str` 与JSON的字符串之间转换。

3. JSON进阶

Python的 `dict` 对象可以直接序列化为JSON的 `{}`，不过，很多时候，我们更喜欢用 `class` 表示对象，比如定义 `Student` 类，然后序列化：

```
import json

class Student(object):
    def __init__(self, name, age, score):
        self.name = name
        self.age = age
        self.score = score

s = Student('Bob', 20, 88)
print(json.dumps(s))
```

运行代码，毫不留情地得到一个 `TypeError`：

```
Traceback (most recent call last):
...
TypeError: <__main__.Student object at 0x10603cc50> is not JSON serializable
```

错误的原因是 `Student` 对象不是一个可序列化为JSON的对象。

如果连 `class` 的实例对象都无法序列化为JSON，这肯定不合理！

别急，我们仔细看看 `dumps()` 方法的参数列表，可以发现，除了第一个必须的 `obj` 参数外，`dumps()` 方法还提供了一大堆的可选参数：

<https://docs.python.org/3/library/json.html#json.dumps>

这些可选参数就是让我们来定制JSON序列化。前面的代码之所以无法把 `Student` 类实例序列化为JSON，是因为默认情况下，`dumps()` 方法不知道如何将 `Student` 实例变为一个JSON的 {} 对象。

可选参数 `default` 就是把任意一个对象变成一个可序列化为JSON的对象，我们只需要为 `Student` 专门写一个转换函数，再把函数传进去即可：

```
def student2dict(std):
    return {
        'name': std.name,
        'age': std.age,
        'score': std.score
    }
```

这样，`Student` 实例首先被 `student2dict()` 函数转换成 `dict`，然后再被顺利序列化为JSON：

```
>>> print(json.dumps(s, default=student2dict))
{"age": 20, "name": "Bob", "score": 88}
```

不过，下次如果遇到一个 `Teacher` 类的实例，照样无法序列化为JSON。我们可以偷个懒，把任意 `class` 的实例变为 `dict`：

```
print(json.dumps(s, default=lambda obj: obj.__dict__))
```

因为通常 `class` 的实例都有一个 `__dict__` 属性，它就是一个 `dict`，用来存储实例变量。也有少数例外，比如定义了 `__slots__` 的 `class`。

同样的道理，如果我们要把JSON反序列化为一个 `Student` 对象实例，`loads()` 方法首先转换出一个 `dict` 对象，然后，我们传入的 `object_hook` 函数负责把 `dict` 转换为 `Student` 实例：

```
def dict2student(d):
    return Student(d['name'], d['age'], d['score'])
```

运行结果如下：

```
>>> json_str = '{"age": 20, "score": 88, "name": "Bob"}'  
>>> print(json.loads(json_str, object_hook=dict2student))  
<__main__.Student object at 0x10cd3c190>
```

打印出的是反序列化的 `Student` 实例对象。

多进程

1. 多进程

要让Python程序实现多进程（multiprocessing），我们先了解操作系统的相关知识。

Unix/Linux操作系统提供了一个 `fork()` 系统调用，它非常特殊。普通的函数调用，调用一次，返回一次，但是 `fork()` 调用一次，返回两次，因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后，分别在父进程和子进程内返回。

子进程永远返回 `0`，而父进程返回子进程的ID。这样做的理由是，一个父进程可以fork出很多子进程，所以，父进程要记下每个子进程的ID，而子进程只需要调用 `getppid()` 就可以拿到父进程的ID。

Python的 `os` 模块封装了常见的系统调用，其中就包括 `fork`，可以在Python程序中轻松创建子进程：

```
import os

print('Process (%s) start...' % os.getpid())
# Only works on Unix/Linux/Mac:
pid = os.fork()
if pid == 0:
    print('I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid()))
else:
    print('I (%s) just created a child process (%s).' % (os.getpid(), pid))
```

运行结果如下：

```
Process (876) start...
```

```
I (876) just created a child process (877).
I am child process (877) and my parent is 876.
```

由于Windows没有 `fork` 调用，上面的代码在Windows上无法运行。由于Mac系统是基于**BSD**（Unix的一种）内核，所以，在Mac下运行是没有问题的，推荐大家用Mac学Python！

有了 `fork` 调用，一个进程在接到新任务时就可以复制出一个子进程来处理新任务，常见的**Apache**服务器就是由父进程监听端口，每当有新的http请求时，就 `fork` 出子进程来处理新的http请求。

2. multiprocessing

如果你打算编写多进程的服务程序，Unix/Linux无疑是正确的选择。由于 Windows没有 `fork` 调用，难道在Windows上无法用Python编写多进程的程序？

由于Python是跨平台的，自然也应该提供一个跨平台的多进程支持。`multiprocessing` 模块就是跨平台版本的多进程模块。

`multiprocessing` 模块提供了一个 `Process` 类来代表一个进程对象，下面的例子演示了启动一个子进程并等待其结束：

```
from multiprocessing import Process
import os

# 子进程要执行的代码
def run_proc(name):
    print('Run child process %s (%s)...' % (name, os.getpid()))

if __name__=='__main__':
    print('Parent process %s.' % os.getpid())
    p = Process(target=run_proc, args=(('test',)))
    print('Child process will start.')
    p.start()
    p.join()
    print('Child process end.')
```

执行结果如下：

```
Parent process 928.  
Process will start.  
Run child process test (929)...  
Process end.
```

创建子进程时，只需要传入一个执行函数和函数的参数，创建一个 `Process` 实例，用 `start()` 方法启动，这样创建进程比 `fork()` 还要简单。

`join()` 方法可以等待子进程结束后再继续往下运行，通常用于进程间的同步。

3. Pool

如果要启动大量的子进程，可以用进程池的方式批量创建子进程：

```
from multiprocessing import Pool  
import os, time, random  
  
def long_time_task(name):  
    print('Run task %s (%s)...' % (name, os.getpid()))  
    start = time.time()  
    time.sleep(random.random() * 3)  
    end = time.time()  
    print('Task %s runs %0.2f seconds.' % (name, (end - start)))  
  
if __name__ == '__main__':  
    print('Parent process %s.' % os.getpid())  
    p = Pool(4)  
    for i in range(5):  
        p.apply_async(long_time_task, args=(i,))  
    print('Waiting for all subprocesses done...')  
    p.close()  
    p.join()  
    print('All subprocesses done.')
```

执行结果如下：

```
Parent process 669.  
Waiting for all subprocesses done...  
Run task 0 (671)...  
Run task 1 (672)...  
Run task 2 (673)...  
Run task 3 (674)...  
Task 2 runs 0.14 seconds.  
Run task 4 (673)...  
Task 1 runs 0.27 seconds.  
Task 3 runs 0.86 seconds.  
Task 0 runs 1.41 seconds.  
Task 4 runs 1.91 seconds.  
All subprocesses done.
```

代码解读：

对 `Pool` 对象调用 `join()` 方法会等待所有子进程执行完毕，调用 `join()` 之前必须先调用 `close()`，调用 `close()` 之后就不能继续添加新的 `Process` 了。

请注意输出的结果，`task 0`，`1`，`2`，`3` 是立刻执行的，而`task 4` 要等待前面某个task完成后才执行，这是因为 `Pool` 的默认大小在我的电脑上是4，因此，最多同时执行4个进程。这是 `Pool` 有意设计的限制，并不是操作系统的限制。如果改成：

```
p = Pool(5)
```

就可以同时跑5个进程。

由于 `Pool` 的默认大小是CPU的核数，如果你不幸拥有8核CPU，你要提交至少9个子进程才能看到上面的等待效果。

4. 子进程

很多时候，子进程并不是自身，而是一个外部进程。我们创建了子进程后，还需要控制子进程的输入和输出。

`subprocess` 模块可以让我们非常方便地启动一个子进程，然后控制其输入和输出。

下面的例子演示了如何在Python代码中运行命令 `nslookup www.python.org`，这和命令行直接运行的效果是一样的：

```
import subprocess

print('$ nslookup www.python.org')
r = subprocess.call(['nslookup', 'www.python.org'])
print('Exit code:', r)
```

运行结果：

```
$ nslookup www.python.org
Server:      192.168.19.4
Address:    192.168.19.4#53

Non-authoritative answer:
www.python.org      canonical name = python.map.fastly.net.
Name:      python.map.fastly.net
Address: 199.27.79.223

Exit code: 0
```

如果子进程还需要输入，则可以通过 `communicate()` 方法输入：

```
import subprocess

print('$ nslookup')
p = subprocess.Popen(['nslookup'], stdin=subprocess.PIPE, stdout=su
bprocess.PIPE, stderr=subprocess.PIPE)
output, err = p.communicate(b'set q=mx\npython.org\nexit\n')
print(output.decode('utf-8'))
print('Exit code:', p.returncode)
```

上面的代码相当于在命令行执行命令 `nslookup`，然后手动输入：

```
set q=mx
python.org
exit
```

运行结果如下：

```
$ nslookup
Server:      192.168.19.4
Address:     192.168.19.4#53

Non-authoritative answer:
python.org      mail exchanger = 50 mail.python.org.

Authoritative answers can be found from:
mail.python.org      internet address = 82.94.164.166
mail.python.org      has AAAA address 2001:888:2000:d::a6

Exit code: 0
```

5. 进程间通信

`Process` 之间肯定是需要通信的，操作系统提供了很多机制来实现进程间的通信。Python的 `multiprocessing` 模块包装了底层的机制，提供了 `Queue`、`Pipes` 等多种方式来交换数据。

我们以 `Queue` 为例，在父进程中创建两个子进程，一个往 `Queue` 里写数据，一个从 `Queue` 里读数据：

```
from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码:
def write(q):
    print('Process to write: %s' % os.getpid())
```

```
for value in ['A', 'B', 'C']:
    print('Put %s to queue...' % value)
    q.put(value)
    time.sleep(random.random())

# 读数据进程执行的代码:
def read(q):
    print('Process to read: %s' % os.getpid())
    while True:
        value = q.get(True)
        print('Get %s from queue.' % value)

if __name__=='__main__':
    # 父进程创建Queue，并传给各个子进程:
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    # 启动子进程pw，写入:
    pw.start()
    # 启动子进程pr，读取:
    pr.start()
    # 等待pw结束:
    pw.join()
    # pr进程里是死循环，无法等待其结束，只能强行终止:
    pr.terminate()
```

运行结果如下：

```
Process to write: 50563
Put A to queue...
Process to read: 50564
Get A from queue.
Put B to queue...
Get B from queue.
Put C to queue...
Get C from queue.
```

在 Unix/Linux 下，`multiprocessing` 模块封装了 `fork()` 调用，使我们不需要关注 `fork()` 的细节。由于 Windows 没有 `fork` 调用，因此，`multiprocessing` 需要“模拟”出 `fork` 的效果，父进程所有 Python 对象都必须通过 `pickle` 序列化再传到子进程去，所有，如果 `multiprocessing` 在 Windows 下调用失败了，要先考虑是不是 `pickle` 失败了。

小结

- 在 Unix/Linux 下，可以使用 `fork()` 调用实现多进程。
- 要实现跨平台的多进程，可以使用 `multiprocessing` 模块。
- 进程间通信是通过 `Queue`、`Pipes` 等实现的。

多线程

1. 多线程

多任务可以由多进程完成，也可以由一个进程内的多线程完成。

进程是由若干线程组成的，一个进程至少有一个线程。

由于线程是操作系统直接支持的执行单元，因此，高级语言通常都内置多线程的支持，Python也不例外，并且，Python的线程是真正的Posix Thread，而不是模拟出来的线程。

Python的标准库提供了两个模块：`_thread` 和 `threading`，`_thread` 是低级模块，`threading` 是高级模块，对 `_thread` 进行了封装。绝大多数情况下，我们只需要使用 `threading` 这个高级模块。

启动一个线程就是把一个函数传入并创建 `Thread` 实例，然后调用 `start()` 开始执行：

```
import time, threading

# 新线程执行的代码:
def loop():
    print('thread %s is running...' % threading.current_thread().name)
    n = 0
    while n < 5:
        n = n + 1
        print('thread %s >>> %s' % (threading.current_thread().name,
                                         n))
        time.sleep(1)
    print('thread %s ended.' % threading.current_thread().name)

print('thread %s is running...' % threading.current_thread().name)
t = threading.Thread(target=loop, name='LoopThread')
t.start()
```

```
t.join()
print('thread %s ended.' % threading.current_thread().name)
```

执行结果如下：

```
thread MainThread is running...
thread LoopThread is running...
thread LoopThread >>> 1
thread LoopThread >>> 2
thread LoopThread >>> 3
thread LoopThread >>> 4
thread LoopThread >>> 5
thread LoopThread ended.
thread MainThread ended.
```

由于任何进程默认就会启动一个线程，我们把该线程称为主线程，主线程又可以启动新的线程，Python的 `threading` 模块有个 `current_thread()` 函数，它永远返回当前线程的实例。主线程实例的名字叫 `MainThread`，子线程的名字在创建时指定，我们用 `LoopThread` 命名子线程。名字仅仅在打印时用来显示，完全没有其他意义，如果不起名字Python就自动给线程命名为 `Thread-1`，`Thread-2`

2. Lock

多线程和多进程最大的不同在于，多进程中，同一个变量，各自有一份拷贝存在于每个进程中，互不影响，而多线程中，所有变量都由所有线程共享，所以，任何一个变量都可以被任何一个线程修改，因此，线程之间共享数据最大的危险在于多个线程同时改一个变量，把内容给改乱了。

来看看多个线程同时操作一个变量怎么把内容给改乱了：

```
import time, threading

# 假定这是你的银行存款:
balance = 0
```

```
def change_it(n):
    # 先存后取，结果应该为0：
    global balance
    balance = balance + n
    balance = balance - n

def run_thread(n):
    for i in range(100000):
        change_it(n)

t1 = threading.Thread(target=run_thread, args=(5,))
t2 = threading.Thread(target=run_thread, args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
print(balance)
```

我们定义了一个共享变量 `balance`，初始值为 `0`，并且启动两个线程，先存后取，理论上结果应该为 `0`，但是，由于线程的调度是由操作系统决定的，当 `t1`、`t2` 交替执行时，只要循环次数足够多，`balance` 的结果就不一定是 `0` 了。

原因是因为高级语言的一条语句在CPU执行时是若干条语句，即使一个简单的计算：

```
balance = balance + n
```

也分两步：

1. 计算 `balance + n`，存入临时变量中；
2. 将临时变量的值赋给 `balance`。

也就是可以看成：

```
x = balance + n
balance = x
```

由于x是局部变量，两个线程各自都有自己的x，当代码正常执行时：

```
初始值 balance = 0

t1: x1 = balance + 5 # x1 = 0 + 5 = 5
t1: balance = x1      # balance = 5
t1: x1 = balance - 5 # x1 = 5 - 5 = 0
t1: balance = x1      # balance = 0

t2: x2 = balance + 8 # x2 = 0 + 8 = 8
t2: balance = x2      # balance = 8
t2: x2 = balance - 8 # x2 = 8 - 8 = 0
t2: balance = x2      # balance = 0

结果 balance = 0
```

但是t1和t2是交替运行的，如果操作系统以下面的顺序执行t1、t2：

```
初始值 balance = 0

t1: x1 = balance + 5 # x1 = 0 + 5 = 5

t2: x2 = balance + 8 # x2 = 0 + 8 = 8
t2: balance = x2      # balance = 8

t1: balance = x1      # balance = 5
t1: x1 = balance - 5 # x1 = 5 - 5 = 0
t1: balance = x1      # balance = 0

t2: x2 = balance - 8 # x2 = 0 - 8 = -8
t2: balance = x2      # balance = -8

结果 balance = -8
```

究其原因，是因为修改 balance 需要多条语句，而执行这几条语句时，线程可能中断，从而导致多个线程把同一个对象的内容改乱了。

两个线程同时一存一取，就可能导致余额不对，你肯定不希望你的银行存款莫名其妙地变成了负数，所以，我们必须确保一个线程在修改 `balance` 的时候，别的线程一定不能改。

如果我们要确保 `balance` 计算正确，就要给 `change_it()` 上一把锁，当某个线程开始执行 `change_it()` 时，我们说，该线程因为获得了锁，因此其他线程不能同时执行 `change_it()`，只能等待，直到锁被释放后，获得该锁以后才能改。由于锁只有一个，无论多少线程，同一时刻最多只有一个线程持有该锁，所以，不会造成修改的冲突。创建一个锁就是通过 `threading.Lock()` 来实现：

```
balance = 0
lock = threading.Lock()

def run_thread(n):
    for i in range(100000):
        # 先要获取锁:
        lock.acquire()
        try:
            # 放心地改吧:
            change_it(n)
        finally:
            # 改完了一定要释放锁:
            lock.release()
```

当多个线程同时执行 `lock.acquire()` 时，只有一个线程能成功地获取锁，然后继续执行代码，其他线程就继续等待直到获得锁为止。

获得锁的线程用完后一定要释放锁，否则那些苦苦等待锁的线程将永远等待下去，成为死线程。所以我们用 `try...finally` 来确保锁一定会被释放。

锁的好处就是确保了某段关键代码只能由一个线程从头到尾完整地执行，坏处当然也很多，首先是阻止了多线程并发执行，包含锁的某段代码实际上只能以单线程模式执行，效率就大大地下降了。其次，由于可以存在多个锁，不同的线程持有不同的锁，并试图获取对方持有的锁时，可能会造成死锁，导致多个线程全部挂起，既不能执行，也无法结束，只能靠操作系统强制终止。

3. 多核CPU

如果你不幸拥有一个[多核CPU](#)，你肯定在想，多核应该可以同时执行多个线程。

如果写一个死循环的话，会出现什么情况呢？

打开Mac OS X的Activity Monitor，或者Windows的Task Manager，都可以监控某个进程的CPU使用率。

我们可以监控到一个死循环线程会100%占用一个CPU。

如果有两个死循环线程，在多核CPU中，可以监控到会占用200%的CPU，也就是占用两个CPU核心。

要想把N核CPU的核心全部跑满，就必须启动N个死循环线程。

试试用Python写个死循环：

```
import threading, multiprocessing

def loop():
    x = 0
    while True:
        x = x ^ 1

    for i in range(multiprocessing.cpu_count()):
        t = threading.Thread(target=loop)
        t.start()
```

启动与CPU核心数量相同的N个线程，在4核CPU上可以监控到CPU占用率仅有102%，也就是仅使用了一核。

但是用C、C++或Java来改写相同的死循环，直接可以把全部核心跑满，4核就跑到400%，8核就跑到800%，为什么Python不行呢？

因为Python的线程虽然是真正的线程，但解释器执行代码时，有一个**GIL锁Global Interpreter Lock**，任何Python线程执行前，必须先获得GIL锁，然后，每执行100条字节码，解释器就自动释放GIL锁，让别的线程有机会执行。

这个GIL全局锁实际上把所有线程的执行代码都给上了锁，所以，多线程在Python中只能交替执行，即使100个线程跑在100核CPU上，也只能用到1个核。

GIL是Python解释器设计的历史遗留问题，通常我们用的解释器是官方实现的CPython，要真正利用多核，除非重写一个不带GIL的解释器。

所以，在Python中，可以使用多线程，但不要指望能有效利用多核。如果一定要通过多线程利用多核，那只能通过C扩展来实现，不过这样就失去了Python简单易用的特点。

不过，也不用过于担心，**Python**虽然不能利用多线程实现多核任务，但可以通过多进程实现多核任务。多个Python进程有各自独立的GIL锁，互不影响。

小结

- 多线程编程，模型复杂，容易发生冲突，必须用锁加以隔离，同时，又要小心死锁的发生。
- Python解释器由于设计时有GIL全局锁，导致了多线程无法利用多核。多线程的并发在Python中就是一个美丽的梦。

ThreadLocal

在多线程环境下，每个线程都有自己的数据。一个线程使用自己的局部变量比使用全局变量好，因为局部变量只有线程自己能看见，不会影响其他线程，而全局变量的修改必须加锁。

但是局部变量也有问题，就是在函数调用的时候，传递起来很麻烦：

```
def process_student(name):
    std = Student(name)
    # std是局部变量，但是每个函数都要用它，因此必须传进去：
    do_task_1(std)
    do_task_2(std)

def do_task_1(std):
    do_subtask_1(std)
    do_subtask_2(std)

def do_task_2(std):
    do_subtask_2(std)
    do_subtask_2(std)
```

每个函数一层一层调用都这么传参数那还得了？用全局变量？也不行，因为每个线程处理不同的 `Student` 对象，不能共享。

如果用一个全局 `dict` 存放所有的 `Student` 对象，然后以 `thread` 自身作为 `key` 获得线程对应的 `Student` 对象如何？

```
global_dict = {}

def std_thread(name):
    std = Student(name)
    # 把std放到全局变量global_dict中：
    global_dict[threading.current_thread()] = std
```

```
do_task_1()
do_task_2()

def do_task_1():
    # 不传入std, 而是根据当前线程查找:
    std = global_dict[threading.current_thread()]
    ...

def do_task_2():
    # 任何函数都可以查找出当前线程的std变量:
    std = global_dict[threading.current_thread()]
    ...
```

这种方式理论上是可行的，它最大的优点是消除了 `std` 对象在每层函数中的传递问题，但是，每个函数获取 `std` 的代码有点丑。

有没有更简单的方式？

`ThreadLocal` 应运而生，不用查找 `dict`，`ThreadLocal` 帮你自动做这件事：

```
import threading

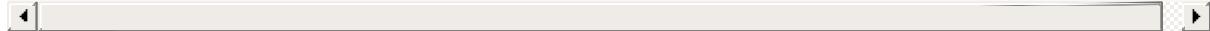
# 创建全局ThreadLocal对象:
local_school = threading.local()

def process_student():
    # 获取当前线程关联的student:
    std = local_school.student
    print('Hello, %s (in %s)' % (std, threading.current_thread().name))

def process_thread(name):
    # 绑定ThreadLocal的student:
    local_school.student = name
    process_student()

t1 = threading.Thread(target= process_thread, args=( 'Alice',), name=
'Thread-A')
```

```
t2 = threading.Thread(target= process_thread, args=( 'Bob' ,), name=' Thread-B ')
t1.start()
t2.start()
t1.join()
t2.join()
```



执行结果：

```
Hello, Alice (in Thread-A)
Hello, Bob (in Thread-B)
```

全局变量 `local_school` 就是一个 `ThreadLocal` 对象，每个 `Thread` 对它都可以读写 `student` 属性，但互不影响。你可以把 `local_school` 看成全局变量，但每个属性如 `local_school.student` 都是线程的局部变量，可以任意读写而互不干扰，也不用管理锁的问题，`ThreadLocal` 内部会处理。

可以理解为全局变量 `local_school` 是一个 `dict`，不但可以用 `local_school.student`，还可以绑定其他变量，如 `local_school.teacher` 等等。

`ThreadLocal` 最常用的地方就是为每个线程绑定一个数据库连接，HTTP请求，用户身份信息等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

小结

一个 `ThreadLocal` 变量虽然是全局变量，但每个线程都只能读写自己线程的独立副本，互不干扰。`ThreadLocal` 解决了参数在一个线程中各个函数之间互相传递的问题。

进程 vs. 线程

1. 多进程和多线程的优缺点

我们介绍了多进程和多线程，这是实现多任务最常用的两种方式。现在，我们来讨论一下这两种方式的优缺点。

首先，要实现多任务，通常我们会设计Master-Worker模式，Master负责分配任务，Worker负责执行任务，因此，多任务环境下，通常是一个Master，多个Worker。

如果用多进程实现Master-Worker，主进程就是Master，其他进程就是Worker。

如果用多线程实现Master-Worker，主线程就是Master，其他线程就是Worker。

多进程模式最大的优点就是稳定性高，因为一个子进程崩溃了，不会影响主进程和其他子进程。（当然主进程挂了所有进程就全挂了，但是Master进程只负责分配任务，挂掉的概率低）著名的Apache最早就是采用多进程模式。

多进程模式的缺点是创建进程的代价大，在Unix/Linux系统下，用 `fork` 调用还行，在Windows下创建进程开销巨大。另外，操作系统能同时运行的进程数也是有限的，在内存和CPU的限制下，如果有几千个进程同时运行，操作系统连调度都会成问题。

多线程模式通常比多进程快一点，但是也快不到哪去，而且，多线程模式致命的缺点就是任何一个线程挂掉都可能直接造成整个进程崩溃，因为所有线程共享进程的内存。在Windows上，如果一个线程执行的代码出了问题，你经常可以看到这样的提示：“该程序执行了非法操作，即将关闭”，其实往往是某个线程出了问题，但是操作系统会强制结束整个进程。

在Windows下，多线程的效率比多进程要高，所以微软的IIS服务器默认采用多线程模式。由于多线程存在稳定性的问题，IIS的稳定性就不如Apache。为了缓解这个问题，IIS和Apache现在又有多进程+多线程的混合模式，真是把问题越搞越复杂。

2. 线程切换

无论是多进程还是多线程，只要数量一多，效率肯定上不去，为什么呢？

我们打个比方，假设你不幸正在准备中考，每天晚上需要做语文、数学、英语、物理、化学这5科的作业，每项作业耗时1小时。

如果你先花1小时做语文作业，做完了，再花1小时做数学作业，这样，依次全部做完，一共花5小时，这种方式称为单任务模型，或者批处理任务模型。

假设你打算切换到多任务模型，可以先做1分钟语文，再切换到数学作业，做1分钟，再切换到英语，以此类推，只要切换速度足够快，这种方式就和单核CPU执行多任务是一样的了，以幼儿园小朋友的眼光来看，你就正在同时写5科作业。

但是，切换作业是有代价的，比如从语文切到数学，要先收拾桌子上的语文书本、钢笔（这叫保存现场），然后，打开数学课本、找出圆规直尺（这叫准备新环境），才能开始做数学作业。操作系统在切换进程或者线程时也是一样的，它需要先保存当前执行的现场环境（CPU寄存器状态、内存页等），然后，把新任务的执行环境准备好（恢复上次的寄存器状态，切换内存页等），才能开始执行。这个切换过程虽然很快，但是也需要耗费时间。如果有几千个任务同时进行，操作系统可能就主要忙着切换任务，根本没有多少时间去执行任务了，这种情况最常见的就是硬盘狂响，点窗口无反应，系统处于假死状态。

所以，多任务一旦多到一个限度，就会消耗掉系统所有的资源，结果效率急剧下降，所有任务都做不好。

3. 计算密集型 vs. IO密集型

是否采用多任务的第二个考虑是任务的类型。我们可以把任务分为计算密集型和IO密集型。

计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力。这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低，所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数。

计算密集型任务由于主要消耗CPU资源，因此，代码运行效率至关重要。Python这样的脚本语言运行效率很低，完全不适合计算密集型任务。对于计算密集型任务，最好用C语言编写。

第二种任务的类型是IO密集型，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度）。对于IO密集型任务，任务越多，CPU效率越高，但也有一个限度。常见的大部分任务都是IO密集型任务，比如Web应用。

IO密集型任务执行期间，99%的时间都花在IO上，花在CPU上的时间很少，因此，用运行速度极快的C语言替换用Python这样运行速度极低的脚本语言，完全无法提升运行效率。对于IO密集型任务，最合适的语言就是开发效率最高（代码量最少）的语言，脚本语言是首选，C语言最差。

4. 异步IO

考虑到CPU和IO之间巨大的速度差异，一个任务在执行的过程中大部分时间都在等待IO操作，单进程单线程模型会导致别的任务无法并行执行，因此，我们才需要多进程模型或者多线程模型来支持多任务并发执行。

现代操作系统对IO操作已经做了巨大的改进，最大的特点就是支持异步IO。如果充分利用操作系统提供的异步IO支持，就可以用单进程单线程模型来执行多任务，这种全新的模型称为事件驱动模型，Nginx就是支持异步IO的Web服务器，它在单核CPU上采用单进程模型就可以高效地支持多任务。在多核CPU上，可以运行多个进程（数量与CPU核心数相同），充分利用多核CPU。由于系统总的进程数量十分有限，因此操作系统调度非常高效。用异步IO编程模型来实现多任务是一个主要的趋势。

对应到Python语言，单线程的异步编程模型称为协程，有了协程的支持，就可以基于事件驱动编写高效的多任务程序。我们会在后面讨论如何编写协程。

分布式进程

在Thread和Process中，应当优选Process，因为Process更稳定，而且，Process可以分布到多台机器上，而Thread最多只能分布到同一台机器的多个CPU上。

Python的 `multiprocessing` 模块不但支持多进程，其中 `managers` 子模块还支持把多进程分布到多台机器上。一个服务进程可以作为调度者，将任务分布到其他多个进程中，依靠网络通信。由于 `managers` 模块封装很好，不必了解网络通信的细节，就可以很容易地编写分布式多进程程序。

举个例子：如果我们已经有一个通过 `Queue` 通信的多进程程序在同一台机器上运行，现在，由于处理任务的进程任务繁重，希望把发送任务的进程和处理任务的进程分布到两台机器上。怎么用分布式进程实现？

原有的 `Queue` 可以继续使用，但是，通过 `managers` 模块把 `Queue` 通过网络暴露出去，就可以让其他机器的进程访问 `Queue` 了。

我们先看服务进程，服务进程负责启动 `Queue`，把 `Queue` 注册到网络上，然后往 `Queue` 里面写入任务：

```
# task_master.py

import random, time, queue
from multiprocessing.managers import BaseManager

# 发送任务的队列:
task_queue = queue.Queue()
# 接收结果的队列:
result_queue = queue.Queue()

# 从BaseManager继承的QueueManager:
class QueueManager(BaseManager):
    pass

# 把两个Queue都注册到网络上, callable参数关联了Queue对象:
```

```

QueueManager.register('get_task_queue', callable=lambda: task_queue)
QueueManager.register('get_result_queue', callable=lambda: result_queue)
# 绑定端口5000, 设置验证码'abc':
manager = QueueManager(address=':', port=5000, authkey=b'abc')
# 启动Queue:
manager.start()
# 获得通过网络访问的Queue对象:
task = manager.get_task_queue()
result = manager.get_result_queue()
# 放几个任务进去:
for i in range(10):
    n = random.randint(0, 10000)
    print('Put task %d...' % n)
    task.put(n)
# 从result队列读取结果:
print('Try get results...')
for i in range(10):
    r = result.get(timeout=10)
    print('Result: %s' % r)
# 关闭:
manager.shutdown()
print('master exit.')

```

请注意，当我们在一台机器上写多进程程序时，创建的 `Queue` 可以直接拿来用，但是，在分布式多进程环境下，添加任务到 `Queue` 不可以直接对原始的 `task_queue` 进行操作，那样就绕过了 `QueueManager` 的封装，必须通过 `manager.get_task_queue()` 获得的 `Queue` 接口添加。

然后，在另一台机器上启动任务进程（本机上启动也可以）：

```

# task_worker.py

import time, sys, queue
from multiprocessing.managers import BaseManager

```

```
# 创建类似的QueueManager:
class QueueManager(BaseManager):
    pass

# 由于这个QueueManager只从网络上获取Queue，所以注册时只提供名字:
QueueManager.register('get_task_queue')
QueueManager.register('get_result_queue')

# 连接到服务器，也就是运行task_master.py的机器:
server_addr = '127.0.0.1'
print('Connect to server %s...' % server_addr)
# 端口和验证码注意保持与task_master.py设置的完全一致:
m = QueueManager(address=(server_addr, 5000), authkey=b'abc')
# 从网络连接:
m.connect()
# 获取Queue的对象:
task = m.get_task_queue()
result = m.get_result_queue()
# 从task队列取任务，并把结果写入result队列:
for i in range(10):
    try:
        n = task.get(timeout=1)
        print('run task %d * %d...' % (n, n))
        r = '%d * %d = %d' % (n, n, n*n)
        time.sleep(1)
        result.put(r)
    except Queue.Empty:
        print('task queue is empty.')
# 处理结束:
print('worker exit.')
```

任务进程要通过网络连接到服务进程，所以要指定服务进程的IP。

现在，可以试试分布式进程的工作效果了。先启动 `task_master.py` 服务进程：

```
$ python3 task_master.py
Put task 3411...
Put task 1605...
```

```
Put task 1398...
Put task 4729...
Put task 5300...
Put task 7471...
Put task 68...
Put task 4219...
Put task 339...
Put task 7866...
Try get results...
```

`task_master.py` 进程发送完任务后，开始等待 `result` 队列的结果。现在启动 `task_worker.py` 进程：

```
$ python3 task_worker.py
Connect to server 127.0.0.1...
run task 3411 * 3411...
run task 1605 * 1605...
run task 1398 * 1398...
run task 4729 * 4729...
run task 5300 * 5300...
run task 7471 * 7471...
run task 68 * 68...
run task 4219 * 4219...
run task 339 * 339...
run task 7866 * 7866...
worker exit.
```

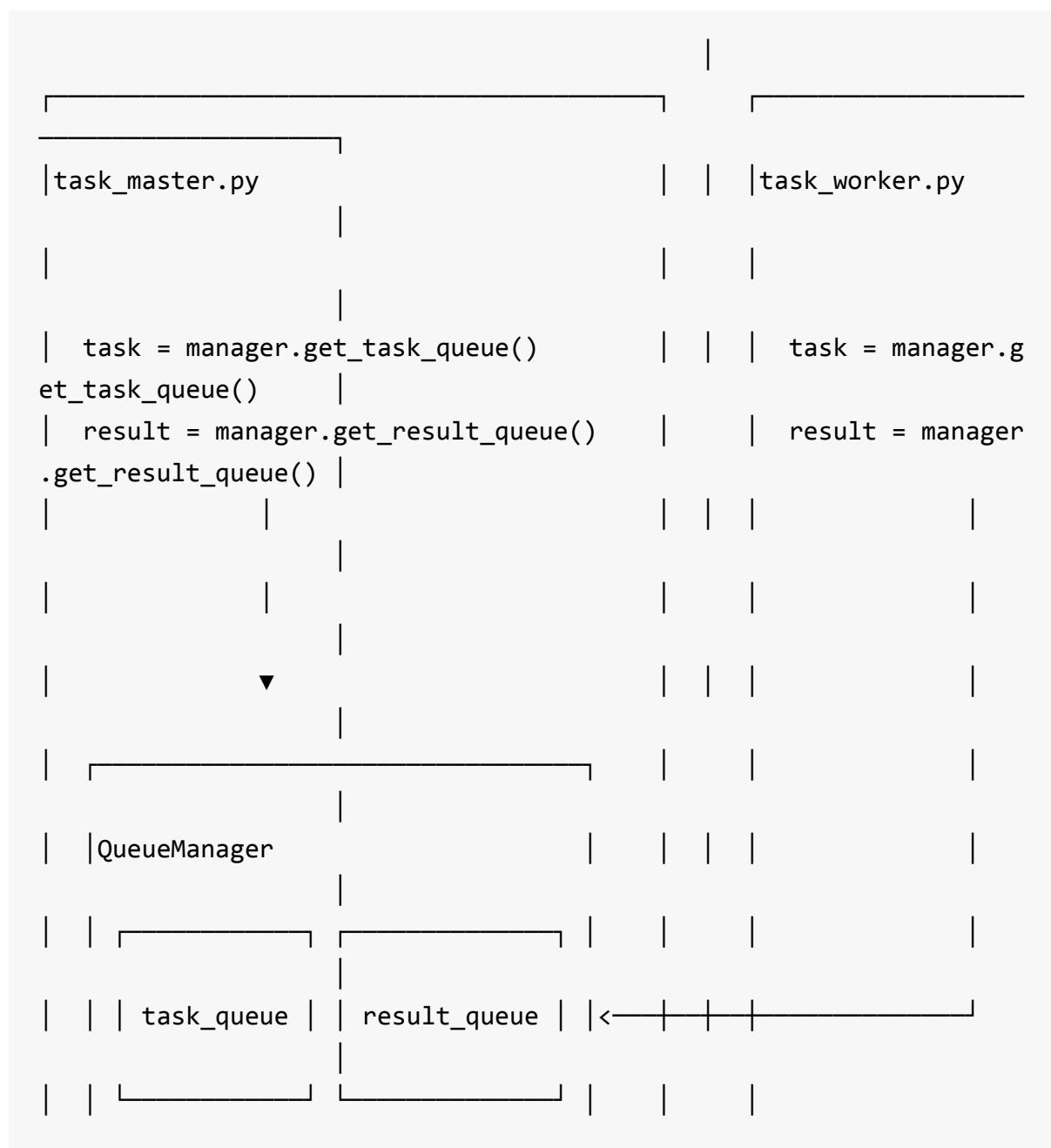
`task_worker.py` 进程结束，在 `task_master.py` 进程中会继续打印出结果：

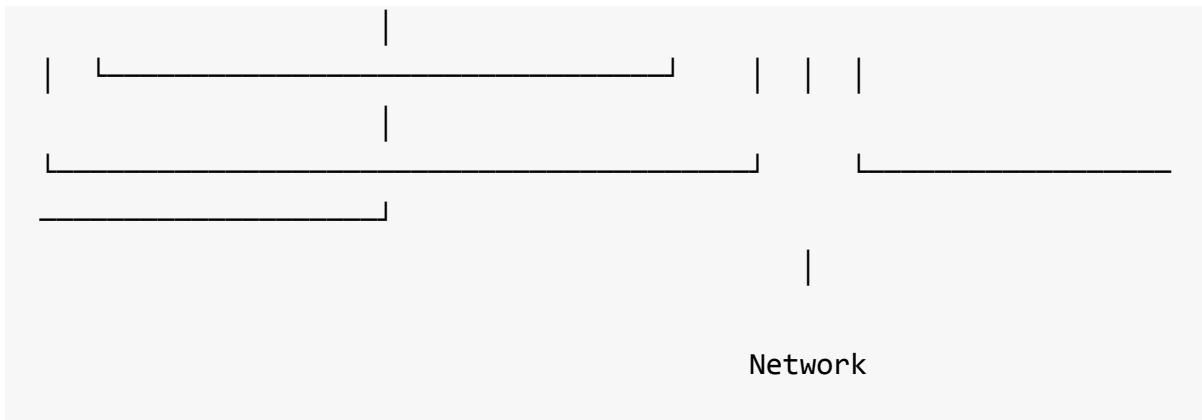
```
Result: 3411 * 3411 = 11634921
Result: 1605 * 1605 = 2576025
Result: 1398 * 1398 = 1954404
Result: 4729 * 4729 = 22363441
Result: 5300 * 5300 = 28090000
Result: 7471 * 7471 = 55815841
Result: 68 * 68 = 4624
Result: 4219 * 4219 = 17799961
```

```
Result: 339 * 339 = 114921
Result: 7866 * 7866 = 61873956
```

这个简单的Master/Worker模型有什么用？其实这就是一个简单但真正的分布式计算，把代码稍加改造，启动多个worker，就可以把任务分布到几台甚至几十台机器上，比如把计算 $n*n$ 的代码换成发送邮件，就实现了邮件队列的异步发送。

Queue对象存储在哪？注意到 `task_worker.py` 中根本没有创建Queue的代码，所以，Queue对象存储在 `task_master.py` 进程中：





而 Queue 之所以能通过网络访问，就是通过 QueueManager 实现的。由于 QueueManager 管理的不止一个 Queue，所以，要给每个 Queue 的网络调用接口起个名字，比如 `get_task_queue`。

`authkey` 有什么用？这是为了保证两台机器正常通信，不被其他机器恶意干扰。如果 `task_worker.py` 的 `authkey` 和 `task_master.py` 的 `authkey` 不一致，肯定连接不上。

小结

- Python的分布式进程接口简单，封装良好，适合需要把繁重任务分布到多台机器的环境下。
- 注意Queue的作用是用来传递任务和接收结果，每个任务的描述数据量要尽量小。比如发送一个处理日志文件的任务，就不要发送几百兆的日志文件本身，而是发送日志文件存放的完整路径，由Worker进程再去共享的磁盘上读取文件。

datetime

`datetime` 是Python处理日期和时间的标准库。

1. 获取当前日期和时间

我们先看如何获取当前日期和时间：

```
>>> from datetime import datetime
>>> now = datetime.now() # 获取当前datetime
>>> print(now)
2015-05-18 16:28:07.198690
>>> print(type(now))
<class 'datetime.datetime'>
```

注意到 `datetime` 是模块，`datetime` 模块还包含一个 `datetime` 类，通过 `from datetime import datetime` 导入的才是 `datetime` 这个类。

如果仅导入 `import datetime`，则必须引用全名 `datetime.datetime`。

`datetime.now()` 返回当前日期和时间，其类型是 `datetime`。

2. 获取指定日期和时间

要指定某个日期和时间，我们直接用参数构造一个 `datetime`：

```
>>> from datetime import datetime
>>> dt = datetime(2015, 4, 19, 12, 20) # 用指定日期时间创建datetime
>>> print(dt)
2015-04-19 12:20:00
```

3. datetime转换为timestamp

在计算机中，时间实际上是由数字表示的。我们把1970年1月1日 00:00:00 UTC+0:00时区的时刻称为epoch time，记为 0（1970年以前的时间timestamp为负数），当前时间就是相对于epoch time的秒数，称为timestamp。

你可以认为：

```
timestamp = 0 = 1970-1-1 00:00:00 UTC+0:00
```

对应的北京时间是：

```
timestamp = 0 = 1970-1-1 08:00:00 UTC+8:00
```

可见timestamp的值与时区毫无关系，因为timestamp一旦确定，其UTC时间就确定了，转换到任意时区的时间也是完全确定的，这就是为什么计算机存储的当前时间是以timestamp表示的，因为全球各地的计算机在任意时刻的timestamp都是完全相同的（假定时间已校准）。

把一个 `datetime` 类型转换为timestamp只需要简单调用 `timestamp()` 方法：

```
>>> from datetime import datetime  
>>> dt = datetime(2015, 4, 19, 12, 20) # 用指定日期时间创建datetime  
>>> dt.timestamp() # 把datetime转换为timestamp  
1429417200.0
```

注意Python的timestamp是一个浮点数。如果有小数位，小数位表示毫秒数。

某些编程语言（如Java和JavaScript）的timestamp使用整数表示毫秒数，这种情况下只需要把timestamp除以1000就得到Python的浮点表示方法。

4. timestamp转换为datetime

要把timestamp转换为 `datetime`，使用 `datetime` 提供的 `fromtimestamp()` 方法：

```
>>> from datetime import datetime  
>>> t = 1429417200.0
```

```
>>> print(datetime.fromtimestamp(t))
2015-04-19 12:20:00
```

注意到timestamp是一个浮点数，它没有时区的概念，而datetime是有时区的。上述转换是在timestamp和本地时间做转换。

本地时间是指当前操作系统设定的时区。例如北京时区是东8区，则本地时间：

```
2015-04-19 12:20:00
```

实际上就是UTC+8:00时区的时间：

```
2015-04-19 12:20:00 UTC+8:00
```

而此刻的格林威治标准时间与北京时间差了8小时，也就是UTC+0:00时区的时间应该是：

```
2015-04-19 04:20:00 UTC+0:00
```

timestamp也可以直接被转换到UTC标准时区的时间：

```
>>> from datetime import datetime
>>> t = 1429417200.0
>>> print(datetime.fromtimestamp(t)) # 本地时间
2015-04-19 12:20:00
>>> print(datetime.utcfromtimestamp(t)) # UTC时间
2015-04-19 04:20:00
```

5. str转换为datetime

很多时候，用户输入的日期和时间是字符串，要处理日期和时间，首先必须把str转换为datetime。转换方法是通过 `datetime.strptime()` 实现，需要一个日期和时间的格式化字符串：

```
>>> from datetime import datetime
>>> cday = datetime.strptime('2015-6-1 18:19:59', '%Y-%m-%d %H:%M:%S')
>>> print(cday)
2015-06-01 18:19:59
```

字符串 `'%Y-%m-%d %H:%M:%S'` 规定了日期和时间部分的格式。详细的说明请参考[Python文档](#)。

注意转换后的datetime是没有时区信息的。

6. datetime转换为str

如果已经有了datetime对象，要把它格式化为字符串显示给用户，就需要转换为str，转换方法是通过 `strftime()` 实现的，同样需要一个日期和时间的格式化字符串：

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> print(now.strftime('%a, %b %d %H:%M'))
Mon, May 05 16:28
```

7. datetime加减

对日期和时间进行加减实际上就是把datetime往后或往前计算，得到新的datetime。加减可以直接用 `+` 和 `-` 运算符，不过需要导入 `timedelta` 这个类：

```
>>> from datetime import datetime, timedelta
>>> now = datetime.now()
>>> now
datetime.datetime(2015, 5, 18, 16, 57, 3, 540997)
>>> now + timedelta(hours=10)
datetime.datetime(2015, 5, 19, 2, 57, 3, 540997)
>>> now - timedelta(days=1)
datetime.datetime(2015, 5, 17, 16, 57, 3, 540997)
>>> now + timedelta(days=2, hours=12)
```

```
datetime.datetime(2015, 5, 21, 4, 57, 3, 540997)
```

可见，使用 `timedelta` 你可以很容易地算出前几天和后几天的时刻。

8. 本地时间转换为UTC时间

本地时间是指系统设定时区的时间，例如北京时间是UTC+8:00时区的时间，而UTC时间指UTC+0:00时区的时间。

一个 `datetime` 类型有一个时区属性 `tzinfo`，但是默认为 `None`，所以无法区分这个 `datetime` 到底是哪个时区，除非强行给 `datetime` 设置一个时区：

```
>>> from datetime import datetime, timedelta, timezone
>>> tz_utc_8 = timezone(timedelta(hours=8)) # 创建时区UTC+8:00
>>> now = datetime.now()
>>> now
datetime.datetime(2015, 5, 18, 17, 2, 10, 871012)
>>> dt = now.replace(tzinfo=tz_utc_8) # 强制设置为UTC+8:00
>>> dt
datetime.datetime(2015, 5, 18, 17, 2, 10, 871012, tzinfo=datetime.timezone(datetime.timedelta(0, 28800)))
```

如果系统时区恰好是UTC+8:00，那么上述代码就是正确的，否则，不能强制设置为UTC+8:00时区。

9. 时区转换

我们可以先通过 `utcnow()` 拿到当前的UTC时间，再转换为任意时区的时间：

```
# 拿到UTC时间，并强制设置时区为UTC+0:00:
>>> utc_dt = datetime.utcnow().replace(tzinfo=timezone.utc)
>>> print(utc_dt)
2015-05-18 09:05:12.377316+00:00
# astimezone()将转换时区为北京时间：
>>> bj_dt = utc_dt.astimezone(timezone(timedelta(hours=8)))
>>> print(bj_dt)
2015-05-18 17:05:12.377316+08:00
```

```
# astimezone()将转换时区为东京时间:  
>>> tokyo_dt = utc_dt.astimezone(timezone(timedelta(hours=9)))  
>>> print(tokyo_dt)  
2015-05-18 18:05:12.377316+09:00  
# astimezone()将bj_dt转换时区为东京时间:  
>>> tokyo_dt2 = bj_dt.astimezone(timezone(timedelta(hours=9)))  
>>> print(tokyo_dt2)  
2015-05-18 18:05:12.377316+09:00
```

时区转换的关键在于，拿到一个 `datetime` 时，要获知其正确的时区，然后强制设置时区，作为基准时间。

利用带时区的 `datetime`，通过 `astimezone()` 方法，可以转换到任意时区。

注：不是必须从UTC+0:00时区转换到其他时区，任何带时区的 `datetime` 都可以正确转换，例如上述 `bj_dt` 到 `tokyo_dt` 的转换。

小结

- `datetime` 表示的时间需要时区信息才能确定一个特定的时间，否则只能视为本地时间。
- 如果要存储 `datetime`，最佳方法是将其转换为 `timestamp` 再存储，因为 `timestamp` 的值与时区完全无关。

collections

`collections` 是Python内建的一个集合模块，提供了许多有用的集合类。

1. namedtuple

我们知道 `tuple` 可以表示不变集合，例如，一个点的二维坐标就可以表示成：

```
>>> p = (1, 2)
```

但是，看到 `(1, 2)`，很难看出这个 `tuple` 是用来表示一个坐标的。

定义一个class又小题大做了，这时，`namedtuple` 就派上了用场：

```
>>> from collections import namedtuple  
>>> Point = namedtuple('Point', ['x', 'y'])  
>>> p = Point(1, 2)  
>>> p.x  
1  
>>> p.y  
2
```

`namedtuple` 是一个函数，它用来创建一个自定义的 `tuple` 对象，并且规定了 `tuple` 元素的个数，并可以用属性而不是索引来引用 `tuple` 的某个元素。

这样一来，我们用 `namedtuple` 可以很方便地定义一种数据类型，它具备`tuple`的不变性，又可以根据属性来引用，使用十分方便。

可以验证创建的 `Point` 对象是 `tuple` 的一种子类：

```
>>> isinstance(p, Point)  
True  
>>> isinstance(p, tuple)  
True
```

类似的，如果要用坐标和半径表示一个圆，也可以用 `namedtuple` 定义：

```
# namedtuple('名称', [属性list]):
Circle = namedtuple('Circle', ['x', 'y', 'r'])
```

2. deque

使用 `list` 存储数据时，按索引访问元素很快，但是插入和删除元素就很慢了，因为 `list` 是线性存储，数据量大的时候，插入和删除效率很低。

`deque`是为了高效实现插入和删除操作的双向列表，适合用于队列和栈：

```
>>> from collections import deque
>>> q = deque(['a', 'b', 'c'])
>>> q.append('x')
>>> q.appendleft('y')
>>> q
deque(['y', 'a', 'b', 'c', 'x'])
```

`deque`除了实现`list`的 `append()` 和 `pop()` 外，还支持 `appendleft()` 和 `popleft()`，这样就可以非常高效地往头部添加或删除元素。

3. defaultdict

使用 `dict` 时，如果引用的Key不存在，就会抛出 `KeyError`。如果希望key不存在时，返回一个默认值，就可以用 `defaultdict`：

```
>>> from collections import defaultdict
>>> dd = defaultdict(lambda: 'N/A')
>>> dd['key1'] = 'abc'
>>> dd['key1'] # key1存在
'abc'
>>> dd['key2'] # key2不存在，返回默认值
'N/A'
```

注意默认值是调用函数返回的，而函数在创建 `defaultdict` 对象时传入。

除了在Key不存在时返回默认值，`defaultdict` 的其他行为跟 `dict` 是完全一样的。

4. OrderedDict

使用 `dict` 时，Key是无序的。在对 `dict` 做迭代时，我们无法确定Key的顺序。

如果要保持Key的顺序，可以用 `OrderedDict`：

```
>>> from collections import OrderedDict
>>> d = dict([('a', 1), ('b', 2), ('c', 3)])
>>> d # dict的Key是无序的
{'a': 1, 'c': 3, 'b': 2}
>>> od = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
>>> od # OrderedDict的Key是有序的
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

注意，`OrderedDict` 的Key会按照插入的顺序排列，不是Key本身排序：

```
>>> od = OrderedDict()
>>> od['z'] = 1
>>> od['y'] = 2
>>> od['x'] = 3
>>> list(od.keys()) # 按照插入的Key的顺序返回
['z', 'y', 'x']
```

`OrderedDict` 可以实现一个FIFO（先进先出）的dict，当容量超出限制时，先删除最早添加的Key：

```
from collections import OrderedDict

class LastUpdatedOrderedDict(OrderedDict):
```

```

def __init__(self, capacity):
    super(LastUpdatedOrderedDict, self).__init__()
    self._capacity = capacity

def __setitem__(self, key, value):
    containsKey = 1 if key in self else 0
    if len(self) - containsKey >= self._capacity:
        last = self.popitem(last=False)
        print('remove:', last)
    if containsKey:
        del self[key]
        print('set:', (key, value))
    else:
        print('add:', (key, value))
    OrderedDict.__setitem__(self, key, value)

```

5. Counter

`Counter` 是一个简单的计数器，例如，统计字符出现的个数：

```

>>> from collections import Counter
>>> c = Counter()
>>> for ch in 'programming':
...     c[ch] = c[ch] + 1
...
>>> c
Counter({'g': 2, 'm': 2, 'r': 2, 'a': 1, 'i': 1, 'o': 1, 'n': 1, 'p': 1})

```

`Counter` 实际上也是 `dict` 的一个子类，上面的结果可以看出，字符 'g'、'm'、'r' 各出现了两次，其他字符各出现了一次。

小结

`collections` 模块提供了一些有用的集合类，可以根据需要选用。

base64

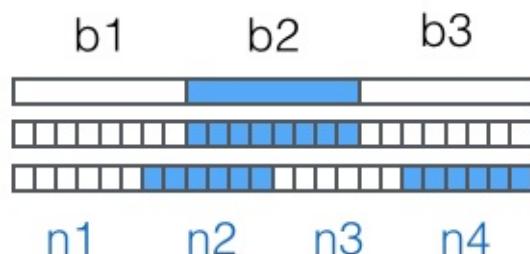
Base64 是一种用64个字符来表示任意二进制数据的方法。

用记事本打开 `exe`、`jpg`、`pdf` 这些文件时，我们都会看到一大堆乱码，因为二进制文件包含很多无法显示和打印的字符，所以，如果要让记事本这样的文本处理软件能处理二进制数据，就需要一个二进制到字符串的转换方法。Base64是一种最常见的二进制编码方法。

Base64的原理很简单，首先，准备一个包含64个字符的数组：

```
[ 'A', 'B', 'C', ... 'a', 'b', 'c', ... '0', '1', ... '+', '/' ]
```

然后，对二进制数据进行处理，每3个字节一组，一共是 $3 \times 8 = 24$ bit，划为4组，每组正好6个bit：



这样我们得到4个数字作为索引，然后查表，获得相应的4个字符，就是编码后的字符串。

所以，**Base64**编码会把3字节的二进制数据编码为4字节的文本数据，长度增加33%，好处是编码后的文本数据可以在邮件正文、网页等直接显示。

如果要编码的二进制数据不是3的倍数，最后会剩下1个或2个字节怎么办？Base64用 `\x00` 字节在末尾补足后，再在编码的末尾加上1个或2个 = 号，表示补了多少字节，解码的时候，会自动去掉。

Python内置的 `base64` 可以直接进行base64的编解码:

```
>>> import base64
```

```
>>> base64.b64encode(b'binary\x00string')
b'YmluYXJ5AHN0cmLuZw=='
>>> base64.b64decode(b'YmluYXJ5AHN0cmLuZw==')
b'binary\x00string'
```

由于标准的Base64编码后可能出现字符 + 和 /，在URL中就不能直接作为参数，所以又有一种"url safe"的base64编码，其实就是把字符 + 和 / 分别变成 - 和 _：

```
>>> base64.b64encode(b'i\xb7\x1d\xfb\xef\xff')
b'abcd++//'
>>> base64.urlsafe_b64encode(b'i\xb7\x1d\xfb\xef\xff')
b'abcd--_'
>>> base64.urlsafe_b64decode('abcd--_')
b'i\xb7\x1d\xfb\xef\xff'
```

还可以自己定义64个字符的排列顺序，这样就可以自定义Base64编码，不过，通常情况下完全没有必要。

Base64是一种通过查表的编码方法，不能用于加密，即使使用自定义的编码表也不行。

Base64适用于小段内容的编码，比如数字证书签名、Cookie的内容等。

由于 = 字符也可能出现在Base64编码中，但 = 用在URL、Cookie里面会造成歧义，所以，很多Base64编码后会把 = 去掉：

```
# 标准Base64:
'abcd' -> 'YWJjZA=='
# 自动去掉=:
'abcd' -> 'YWJjZA'
```

去掉 = 后怎么解码呢？因为Base64是把3个字节变为4个字节，所以，Base64编码的长度永远是4的倍数，因此，需要加上 = 把Base64字符串的长度变为4的倍数，就可以正常解码了。

小结

Base64是一种任意二进制到文本字符串的编码方法，常用于在URL、Cookie、网页中传输少量二进制数据。

struct

准确地讲，Python没有专门处理字节的数据类型。但由于 `b'str'` 可以表示字节，所以，字节数组 = 二进制str。而在C语言中，我们可以很方便地用 `struct` 、 `union` 来处理字节，以及字节和 `int` ， `float` 的转换。

在Python中，比方说要把一个32位无符号整数变成字节，也就是4个长度的 `bytes`，你得配合位运算符这么写：

```
>>> n = 10240099
>>> b1 = (n & 0xff000000) >> 24
>>> b2 = (n & 0xffff0000) >> 16
>>> b3 = (n & 0xff00) >> 8
>>> b4 = n & 0xff
>>> bs = bytes([b1, b2, b3, b4])
>>> bs
b'\x00\x9c@c'
```

非常麻烦。如果换成浮点数就无能为力了。

好在Python提供了一个 `struct` 模块来解决 `bytes` 和其他二进制数据类型的转换。

`struct` 的 `pack` 函数把任意数据类型变成 `bytes`：

```
>>> import struct
>>> struct.pack('>I', 10240099)
b'\x00\x9c@c'
```

`pack` 的第一个参数是处理指令， '`>I`' 的意思是：

`>` 表示字节顺序是 `big-endian`，也就是网络序， `I` 表示4字节无符号整数。

后面的参数个数要和处理指令一致。

`unpack` 把 `bytes` 变成相应的数据类型：

```
>>> struct.unpack('>IH', b'\xf0\xf0\xf0\xf0\x80\x80')
(4042322160, 32896)
```

根据 `>IH` 的说明，后面的 `bytes` 依次变为 `I`：4字节无符号整数和 `H`：2字节无符号整数。

所以，尽管Python不适合编写底层操作字节流的代码，但在对性能要求不高的地方，利用 `struct` 就方便多了。

`struct` 模块定义的数据类型可以参考Python官方文档：

<https://docs.python.org/3/library/struct.html#format-characters>

Windows的位图文件（.bmp）是一种非常简单的文件格式，我们来用 `struct` 分析一下。

首先找一个bmp文件，没有的话用“画图”画一个。

读入前30个字节来分析：

```
>>> s = b'\x42\x4d\x38\x8c\x0a\x00\x00\x00\x00\x00\x36\x00\x00\x00\x00\x28\x00\x00\x00\x80\x02\x00\x00\x68\x01\x00\x00\x01\x00\x18\x00'
```

BMP格式采用小端方式存储数据，文件头的结构按顺序如下：

两个字节：'BM' 表示Windows位图，'BA' 表示OS/2位图；一个4字节整数：表示位图大小；一个4字节整数：保留位，始终为0；一个4字节整数：实际图像的偏移量；一个4字节整数：Header的字节数；一个4字节整数：图像宽度；一个4字节整数：图像高度；一个2字节整数：始终为1；一个2字节整数：颜色数。

所以，组合起来用 `unpack` 读取：

```
>>> struct.unpack('<ccIIIIIIHH', s)
(b'B', b'M', 691256, 0, 54, 40, 640, 360, 1, 24)
```

结果显示，`b'B'`、`b'M'` 说明是Windows位图，位图大小为640x360，颜色数为24。

hashlib

1. 摘要算法简介

Python的 `hashlib` 提供了常见的摘要算法，如MD5，SHA1等等。

什么是摘要算法呢？摘要算法又称哈希算法、散列算法。它通过一个函数，把任意长度的数据转换为一个长度固定的数据串（通常用16进制的字符串表示）。

举个例子，你写了一篇文章，内容是一个字符串 `'how to use python hashlib - by Michael'`，并附上这篇文章的摘要

是 `'2d73d4f15c0db7f5ecb321b6a65e5d6d'`。如果有人篡改了你的文章，并发表为 `'how to use python hashlib - by Bob'`，你可以一下子指出Bob篡改了你的文章，因为根据 `'how to use python hashlib - by Bob'` 计算出的摘要不同于原始文章的摘要。

可见，摘要算法就是通过摘要函数 `f()` 对任意长度的数据 `data` 计算出固定长度的摘要 `digest`，目的是为了发现原始数据是否被人篡改过。

摘要算法之所以能指出数据是否被篡改过，就是因为摘要函数是一个单向函数，计算 `f(data)` 很容易，但通过 `digest` 反推 `data` 却非常困难。而且，对原始数据做一个bit的修改，都会导致计算出的摘要完全不同。

我们以常见的摘要算法MD5为例，计算出一个字符串的MD5值：

```
import hashlib

md5 = hashlib.md5()
md5.update('how to use md5 in python hashlib?'.encode('utf-8'))
print(md5.hexdigest())
```

计算结果如下：

```
d26a53750bc40b38b65a520292f69306
```

如果数据量很大，可以分块多次调用 `update()`，最后计算的结果是一样的：

```
import hashlib

md5 = hashlib.md5()
md5.update('how to use md5 in '.encode('utf-8'))
md5.update('python hashlib?'.encode('utf-8'))
print(md5.hexdigest())
```

试试改动一个字母，看看计算的结果是否完全不同。

MD5是最常见的摘要算法，速度很快，生成结果是固定的128 bit字节，通常用一个32位的16进制字符串表示。

另一种常见的摘要算法是SHA1，调用SHA1和调用MD5完全类似：

```
import hashlib

sha1 = hashlib.sha1()
sha1.update('how to use sha1 in '.encode('utf-8'))
sha1.update('python hashlib?'.encode('utf-8'))
print(sha1.hexdigest())
```

SHA1的结果是160 bit字节，通常用一个40位的16进制字符串表示。

比SHA1更安全的算法是SHA256和SHA512，不过越安全的算法不仅越慢，而且摘要长度更长。

有没有可能两个不同的数据通过某个摘要算法得到了相同的摘要？完全有可能，因为任何摘要算法都是把无限多的数据集合映射到一个有限的集合中。这种情况称为碰撞，比如Bob试图根据你的摘要反推出一篇文章 '`how to learn hashlib in python - by Bob`'，并且这篇文章的摘要恰好和你的文章完全一致，这种情况也并非不可能出现，但是非常非常困难。

2. 摘要算法应用

摘要算法能应用到什么地方？举个常用例子：

任何允许用户登录的网站都会存储用户登录的用户名和口令。如何存储用户名和口令呢？方法是存到数据库表中：

name	password
michael	123456
bob	abc999
alice	alice2008

如果以明文保存用户口令，如果数据库泄露，所有用户的口令就落入黑客的手里。此外，网站运维人员是可以访问数据库的，也就是能获取到所有用户的口令。

正确的保存口令的方式是不存储用户的明文口令，而是存储用户口令的摘要，比如MD5：

username	password
michael	e10adc3949ba59abbe56e057f20f883e
bob	878ef96e86145580c38c87f0410ad153
alice	99b1c2188db85afee403b1536010c2c9

当用户登录时，首先计算用户输入的明文口令的MD5，然后和数据库存储的MD5对比，如果一致，说明口令输入正确，如果不一致，口令肯定错误。

hmac

通过哈希算法，我们可以验证一段数据是否有效，方法就是对比该数据的哈希值，例如，判断用户口令是否正确，我们用保存在数据库中的 `password_md5` 对比计算 `md5(password)` 的结果，如果一致，用户输入的口令就是正确的。

为了防止黑客通过彩虹表根据哈希值反推原始口令，在计算哈希的时候，不能仅针对原始输入计算，需要增加一个salt来使得相同的输入也能得到不同的哈希，这样，大大增加了黑客破解的难度。

如果salt是我们自己随机生成的，通常我们计算MD5时采用 `md5(message + salt)`。但实际上，把salt看做一个“口令”，加salt的哈希就是：计算一段 `message` 的哈希时，根据不同口令计算出不同的哈希。要验证哈希值，必须同时提供正确的口令。

这实际上就是Hmac算法：Keyed-Hashing for Message Authentication。它通过一个标准算法，在计算哈希的过程中，把key混入计算过程中。

和我们自定义的加salt算法不同，Hmac算法针对所有哈希算法都通用，无论是MD5还是SHA-1。采用Hmac替代我们自己的salt算法，可以使程序算法更标准化，也更安全。

Python自带的hmac模块实现了标准的Hmac算法。我们来看看如何使用hmac实现带key的哈希。

我们首先需要准备待计算的原始消息 `message`，随机key，哈希算法，这里采用MD5，使用hmac的代码如下：

```
>>> import hmac
>>> message = b'Hello, world!'
>>> key = b'secret'
>>> h = hmac.new(key, message, digestmod='MD5')
>>> # 如果消息很长，可以多次调用h.update(msg)
>>> h.hexdigest()
```

```
'fa4ee7d173f2d97ee79022d1a7355bcf'
```

可见使用hmac和普通hash算法非常类似。hmac输出的长度和原始哈希算法的长度一致。需要注意传入的key和message都是 `bytes` 类型，`str` 类型需要首先编码为 `bytes`。

itertools

Python的内建模块 `itertools` 提供了非常有用的用于操作迭代对象的函数。

首先，我们看看 `itertools` 提供的几个“无限”迭代器：

```
>>> import itertools
>>> naturals = itertools.count(1)
>>> for n in naturals:
...     print(n)
...
1
2
3
...
```

因为 `count()` 会创建一个无限的迭代器，所以上述代码会打印出自然数序列，根本停不下来，只能按 `Ctrl+C` 退出。

`cycle()` 会把传入的一个序列无限重复下去：

```
>>> import itertools
>>> cs = itertools.cycle('ABC') # 注意字符串也是序列的一种
>>> for c in cs:
...     print(c)
...
'A'
'B'
'C'
'A'
'B'
'C'
...
```

同样停不下来。

`repeat()` 负责把一个元素无限重复下去，不过如果提供第二个参数就可以限定重复次数：

```
>>> ns = itertools.repeat('A', 3)
>>> for n in ns:
...     print(n)
...
A
A
A
```

无限序列只有在 `for` 迭代时才会无限地迭代下去，如果只是创建了一个迭代对象，它不会事先把无限个元素生成出来，事实上也不可能在内存中创建无限多个元素。

无限序列虽然可以无限迭代下去，但是通常我们会通过 `takewhile()` 等函数根据条件判断来截取出一个有限的序列：

```
>>> naturals = itertools.count(1)
>>> ns = itertools.takewhile(lambda x: x <= 10, naturals)
>>> list(ns)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`itertools` 提供的几个迭代器操作函数更加有用：

1. `chain()`

`chain()` 可以把一组迭代对象串联起来，形成一个更大的迭代器：

```
>>> for c in itertools.chain('ABC', 'XYZ'):
...     print(c)
# 迭代效果: 'A' 'B' 'C' 'X' 'Y' 'Z'
```

2. `groupby()`

`groupby()` 把迭代器中相邻的重复元素挑出来放在一起：

```
>>> for key, group in itertools.groupby('AAABBBCCAAA'):
...     print(key, list(group))
...
A ['A', 'A', 'A']
B ['B', 'B', 'B']
C ['C', 'C']
A ['A', 'A', 'A']
```

实际上挑选规则是通过函数完成的，只要作用于函数的两个元素返回的值相等，这两个元素就被认为是在一组的，而函数返回值作为组的key。如果我们要忽略大小写分组，就可以让元素 '`'A'`' 和 '`'a'`' 都返回相同的key：

```
>>> for key, group in itertools.groupby('AaaBBbcCAAa', lambda c: c.
...                                         upper()):
...     print(key, list(group))
...
A ['A', 'a', 'a']
B ['B', 'B', 'b']
C ['c', 'C']
A ['A', 'A', 'a']
```

contextlib

在Python中，读写文件这样的资源要特别注意，必须在使用完毕后正确关闭它们。正确关闭文件资源的一个方法是使用 `try...finally` :

```
try:  
    f = open('/path/to/file', 'r')  
    f.read()  
finally:  
    if f:  
        f.close()
```

写 `try...finally` 非常繁琐。Python的 `with` 语句允许我们非常方便地使用资源，而不必担心资源没有关闭，所以上面的代码可以简化为：

```
with open('/path/to/file', 'r') as f:  
    f.read()
```

并不是只有 `open()` 函数返回的fp对象才能使用 `with` 语句。实际上，任何对象，只要正确实现了上下文管理，就可以用于 `with` 语句。

实现上下文管理是通过 `__enter__` 和 `__exit__` 这两个方法实现的。例如，下面的class实现了这两个方法：

```
class Query(object):  
  
    def __init__(self, name):  
        self.name = name  
  
    def __enter__(self):  
        print('Begin')  
        return self
```

```

def __exit__(self, exc_type, exc_value, traceback):
    if exc_type:
        print('Error')
    else:
        print('End')

def query(self):
    print('Query info about %s...' % self.name)

```

这样我们就可以把自己写的资源对象用于 `with` 语句：

```

with Query('Bob') as q:
    q.query()

```

`@contextmanager`

编写 `__enter__` 和 `__exit__` 仍然很繁琐，因此Python的标准库 `contextlib` 提供了更简单的写法，上面的代码可以改写如下：

```

from contextlib import contextmanager

class Query(object):

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

    def query(self):
        print('Query info about %s...' % self.name)

@contextmanager
def create_query(name):
    print('Begin')
    q = Query(name)
    yield q
    print('End')

```

`@contextmanager` 这个decorator接受一个generator，用 `yield` 语句把 `with ... as var` 把变量输出出去，然后，`with` 语句就可以正常地工作了：

```
with create_query('Bob') as q:
    q.query()
```

很多时候，我们希望在某段代码执行前后自动执行特定代码，也可以用 `@contextmanager` 实现。例如：

```
@contextmanager
def tag(name):
    print("<%s>" % name)
    yield
    print("</%s>" % name)

with tag("h1"):
    print("hello")
    print("world")
```

上述代码执行结果为：

```
<h1>
hello
world
</h1>
```

代码的执行顺序是：

1. `with` 语句首先执行 `yield` 之前的语句，因此打印出 `<h1>`；
2. `yield` 调用会执行 `with` 语句内部的所有语句，因此打印出 `hello` 和 `world`；
3. 最后执行 `yield` 之后的语句，打印出 `</h1>`。

因此，`@contextmanager` 让我们通过编写generator来简化上下文管理。

`@closing`

如果一个对象没有实现上下文，我们就不能把它用于 `with` 语句。这个时候，可以用 `closing()` 来把该对象变为上下文对象。例如，用 `with` 语句使用 `urlopen()`：

```
from contextlib import closing
from urllib.request import urlopen

with closing(urlopen('https://www.python.org')) as page:
    for line in page:
        print(line)
```

`closing` 也是一个经过`@contextmanager`装饰的generator，这个generator编写起来其实非常简单：

```
@contextmanager
def closing(thing):
    try:
        yield thing
    finally:
        thing.close()
```

它的作用就是把任意对象变为上下文对象，并支持 `with` 语句。

`@contextlib` 还有一些其他decorator，便于我们编写更简洁的代码。

urllib

`urllib` 提供了一系列用于操作URL的功能。

1. Get

`urllib` 的 `request` 模块可以非常方便地抓取URL内容，也就是发送一个GET请求到指定的页面，然后返回HTTP的响应：

例如，对豆瓣的一个URL `https://api.douban.com/v2/book/2129650` 进行抓取，并返回响应：

```
from urllib import request

with request.urlopen('https://api.douban.com/v2/book/2129650') as f:
    data = f.read()
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', data.decode('utf-8'))
```

可以看到HTTP响应的头和JSON数据：

```
Status: 200 OK
Server: nginx
Date: Tue, 26 May 2015 10:02:27 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 2049
Connection: close
Expires: Sun, 1 Jan 2006 01:00:00 GMT
Pragma: no-cache
Cache-Control: must-revalidate, no-cache, private
X-DAE-Node: pid11
```

```
Data: {"rating": {"max": 10, "numRaters": 16, "average": "7.4", "min": 0}, "subtitle": "", "author": ["廖雪峰编著"], "pubdate": "2007-6", ...}
```

如果我们要想模拟浏览器发送GET请求，就需要使用 `Request` 对象，通过往 `Request` 对象添加HTTP头，我们就可以把请求伪装成浏览器。例如，模拟 iPhone 6去请求豆瓣首页：

```
from urllib import request

req = request.Request('http://www.douban.com/')
req.add_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')
with request.urlopen(req) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', f.read().decode('utf-8'))
```

这样豆瓣会返回适合iPhone的移动版网页：

```
...
<meta name="viewport" content="width=device-width, user-scalable=no, initial-scale=1.0, minimum-scale=1.0, maximum-scale=1.0">
<meta name="format-detection" content="telephone=no">
<link rel="apple-touch-icon" sizes="57x57" href="http://img4.douban.com/pics/cardkit/launcher/57.png" />
...
```

2. Post

如果要以POST发送一个请求，只需要把参数 `data` 以bytes形式传入。

我们模拟一个微博登录，先读取登录的邮箱和口令，然后按照weibo.cn的登录页的格式以 `username=xxx&password=xxx` 的编码传入：

```

from urllib import request, parse

print('Login to weibo.cn...')
email = input('Email: ')
passwd = input('Password: ')
login_data = parse.urlencode([
    ('username', email),
    ('password', passwd),
    ('entry', 'mweibo'),
    ('client_id', ''),
    ('savestate', '1'),
    ('ec', ''),
    ('pagerefer', 'https://passport.weibo.cn/signin/welcome?entry=mweibo&r=http%3A%2F%2Fm.weibo.cn%2F')
])

req = request.Request('https://passport.weibo.cn/sso/login')
req.add_header('Origin', 'https://passport.weibo.cn')
req.add_header('User-Agent', 'Mozilla/6.0 (iPhone; CPU iPhone OS 8_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/8.0 Mobile/10A5376e Safari/8536.25')
req.add_header('Referer', 'https://passport.weibo.cn/signin/login?entry=mweibo&res=wel&wm=3349&r=http%3A%2F%2Fm.weibo.cn%2F')

with request.urlopen(req, data=login_data.encode('utf-8')) as f:
    print('Status:', f.status, f.reason)
    for k, v in f.getheaders():
        print('%s: %s' % (k, v))
    print('Data:', f.read().decode('utf-8'))

```

如果登录成功，我们获得的响应如下：

```

Status: 200 OK
Server: nginx/1.2.0
...
Set-Cookie: SSOLoginState=1432620126; path=/; domain=weibo.cn
...

```

```
Data: {"retcode":20000000,"msg":"","data":{...,"uid":"1658384301"}}
```

如果登录失败，我们获得的响应如下：

```
...
Data: {"retcode":50011015,"msg":"\u7528\u6237\u540d\u6216\u5bc6\u7801\u9519\u8bef","data":{"username":"example@python.org","errline":536}}
```

3. Handler

如果还需要更复杂的控制，比如通过一个Proxy去访问网站，我们需要利用 `ProxyHandler` 来处理，示例代码如下：

```
proxy_handler = urllib.request.ProxyHandler({'http': 'http://www.example.com:3128/'})
proxy_auth_handler = urllib.request.ProxyBasicAuthHandler()
proxy_auth_handler.add_password('realm', 'host', 'username', 'password')
opener = urllib.request.build_opener(proxy_handler, proxy_auth_handler)
with opener.open('http://www.example.com/login.html') as f:
    pass
```

小结

`urllib` 提供的功能就是利用程序去执行各种HTTP请求。如果要模拟浏览器完成特定功能，需要把请求伪装成浏览器。伪装的方法是先监控浏览器发出的请求，再根据浏览器的请求头来伪装，`User-Agent` 头就是用来标识浏览器的。

XML

XML虽然比JSON复杂，在Web中应用也不如以前多了，不过仍有很多地方在用，所以，有必要了解如何操作XML。

DOM vs SAX

操作XML有两种方法：DOM和SAX。DOM会把整个XML读入内存，解析为树，因此占用内存大，解析慢，优点是可以任意遍历树的节点。SAX是流模式，边读边解析，占用内存小，解析快，缺点是我们需要自己处理事件。

正常情况下，优先考虑SAX，因为DOM实在太占内存。

在Python中使用SAX解析XML非常简洁，通常我们关心的事件是 `start_element`，`end_element` 和 `char_data`，准备好这3个函数，然后就可以解析xml了。

举个例子，当SAX解析器读到一个节点时：

```
<a href="/">python</a>
```

会产生3个事件：

1. `start_element`事件，在读取 `` 时；
2. `char_data`事件，在读取 `python` 时；
3. `end_element`事件，在读取 `` 时。

用代码实验一下：

```
from xml.parsers.expat import ParserCreate

class DefaultSaxHandler(object):
    def start_element(self, name, attrs):
        print('sax:start_element: %s, attrs: %s' % (name, str(attrs)))
```

```

)))
def end_element(self, name):
    print('sax:end_element: %s' % name)

def char_data(self, text):
    print('sax:char_data: %s' % text)

xml = r'''<?xml version="1.0"?>
<ol>
    <li><a href="/python">Python</a></li>
    <li><a href="/ruby">Ruby</a></li>
</ol>
...
'''

handler = DefaultSaxHandler()
parser = ParserCreate()
parser.StartElementHandler = handler.start_element
parser.EndElementHandler = handler.end_element
parser.CharacterDataHandler = handler.char_data
parser.Parse(xml)

```

需要注意的是读取一大段字符串时，`CharacterDataHandler` 可能被多次调用，所以需要自己保存起来，在 `EndElementHandler` 里面再合并。

除了解析XML外，如何生成XML呢？99%的情况下需要生成的XML结构都是非常简单的，因此，最简单也是最有效的生成XML的方法是拼接字符串：

```

L = []
L.append(r'<?xml version="1.0"?>')
L.append(r'<root>')
L.append(encode('some & data'))
L.append(r'</root>')
return ''.join(L)

```

如果要生成复杂的XML呢？建议你不要用XML，改成JSON。

小结

解析XML时，注意找出自己感兴趣的节点，响应事件时，把节点数据保存起来。解析完毕后，就可以处理数据。

HTMLParser

如果我们要编写一个搜索引擎，第一步是用爬虫把目标网站的页面抓下来，第二步就是解析该HTML页面，看看里面的内容到底是新闻、图片还是视频。

假设第一步已经完成了，第二步应该如何解析HTML呢？

HTML本质上是XML的子集，但是HTML的语法没有XML那么严格，所以不能用标准的DOM或SAX来解析HTML。

好在Python提供了HTMLParser来非常方便地解析HTML，只需简单几行代码：

```
from html.parser import HTMLParser
from html.entities import name2codepoint

class MyHTMLParser(HTMLParser):

    def handle_starttag(self, tag, attrs):
        print('<%s>' % tag)

    def handle_endtag(self, tag):
        print('</%s>' % tag)

    def handle_startendtag(self, tag, attrs):
        print('<%s/>' % tag)

    def handle_data(self, data):
        print(data)

    def handle_comment(self, data):
        print('<!--', data, '-->')

    def handle_entityref(self, name):
        print('&%s;' % name)

    def handle_charref(self, name):
```

```
print('"%s;" % name)

parser = MyHTMLParser()
parser.feed('''<html>
<head></head>
<body>
<!-- test html parser -->
<p>Some <a href="#">html</a> HTML tutorial...<br>END</p>
</body></html>'''')
```

`feed()` 方法可以多次调用，也就是不一定一次把整个HTML字符串都塞进去，可以一部分一部分塞进去。

特殊字符有两种，一种是英文表示的 ，一种是数字表示的 `Ӓ`，这两种字符都可以通过Parser解析出来。

小结

利用 `HTMLParser`，可以把网页中的文本、图像等解析出来。