# Python基础

# 第一章python概述

## 1.1 什么是 Python？

Python 是一门优雅而健壮的编程语言，它继承了传统编译语言的强大性和通用性，同时也借鉴了简单脚本和解释语言的易用性。它可以帮你完成工作，而且一段时间以后，你还能看明白自己写的这段代码。你会对自己如此快地学会它和它强大的功能感到十分的惊讶，更不用提你已经完成的工作了！只有你想不到，没有Python 做不到。

## 1.2 起源

贵铎·范·罗萨姆（Guido van Rossum）于1989 年底始创了Python，那时，他还在荷兰的CWI（Centrum voor Wiskunde en Informatica，国家数学和计算机科学研究院）。1991 年

初，Python 发布了第一个公开发行版。这一切究竟是如何开始的呢？像C、C++、Lisp、Java

和Perl 一样，Python 来自于某个研究项目，项目中的那些程序员利用手边现有的工具辛苦的工作着，他们设想并开发出了更好的解决办法。

## 1.3 Python语言的特性

**高级：**

汇编语言是上帝献给那些挣扎在机器代码中的人的礼物，后来有了Fortran、 C 和 Pascal 语言，它们将计算提升到了崭新的高度，并且开创了软件开发行业。伴随着C 语言诞生了更多的像C++、Java 这样的现代编译语言。我们没有止步于此，于是有了强大的、可以进行系统调用的解释型脚本语言，例如Tcl、Perl 和Python。

这些语言都有高级的数据结构，这样就减少了以前“框架”开发需要的时间。像Python 中的列表（**大小可变的数组**）和字典（**哈希表**）就是内建于语言本身的。。在核心语言中提供这些重要的构建单元，可以鼓励人们使用它们，缩短开发时间与代码量，产生出可读性更好的代码。

**面向对象：**

**面向对象**编程为数据和逻辑相分离的结构化和过程化编程添加了新的活力。面向对象 编程支持将特定的行为、特性以及和/或功能与它们要处理或所代表的数据结合在一起。Python 的面向对象的特性是与生俱来的。然而，Python 绝不想Java 或Ruby 仅仅是一门面向对象语言，事实上它融汇了多种编程风格。例如，它甚至借鉴了一些像Lisp 和Haskell 这样的**函数语言**的特性。

**可扩展：**

就算你的项目中有大量的Python 代码，你也依旧可以有条不紊地通过将其分离为多个文件或模块加以组织管理。而且你可以从一个模块中选取代码，而从另一个模块中读取属性。更棒的是，对于所有模块，Python 的访问语法都是相同的。不管这个模块是Python 标准库中的还是你一分钟之前创造的，哪怕是你用其他语言写的扩展都没问题！借助这些特点，你会感觉自己根据需要“扩展”了这门语言，而且你已经这么做了。

代码中的瓶颈，可能是在性能分析中总排在前面的那些热门或者一些特别强调性能的地方，可以作为 Python 扩展用 C 重写。需要重申的是，这些接口和纯Python 模块的接口是一模一样的，乃至代码和对象的访问方法也是相同的。唯一不同的是，这些代码为性能带来了显著的提升。

**可移植性：**

在各种不同的系统上可以看到Python 的身影，这是由于在今天的计算机领域，Python 取得了持续快速的成长。因为Python 是用C 写的，又由于C 的可移植性，使得Python 可以运行在任何带有ANSI C 编译器的平台上。尽管有一些针对不同平台开发的特有模块，但是在任何一个平台上用Python 开发的通用软件都可以稍事修改或者原封不动的在其他平台上运行。这种可移植性既适用于不同的架构，也适用于不同的操作系统。

**易学：**

Python 关键字少、结构简单、语法清晰。这样就使得学习者可以在相对更短的时间内轻松上手。对初学者而言，可能感觉比较新鲜的东西可能就是Python 的面向对象特点了。那些还未能全部精通OOP（Object Oriented Programming, 面向对象的程序设计）的人对径直使用Python还是有所顾忌的，但是OOP 并非必须或者强制的。入门也是很简单的，你可以先稍加涉猎，等到有所准备之后才开始使用。

**易维护：**

源代码维护是软件开发生命周期的组成部分。只要不被其他软件取代或者被放弃使用，你的软件通常会保持继续的再开发。这通常可比一个程序员在一家公司的在职时间要长得多了。Python 项目的成功很大程度上要归功于其源代码的易于维护，当然这也要视代码长度和复杂度而定。然而，得出这个结论并不难，因为Python 本身就是易于学习和阅读的。Python 另外一个激动人心的优势就是，当你在阅读自己半年之前写的脚本程序的时候，不会把自己搞得一头雾水，也不需要借助参考手册才能读懂自己的软件。

**健壮性：**

没有什么能够比允许程序员在错误发生的时候根据出错条件提供处理机制更有效的了。针对错误，Python 提供了“安全合理”的退出机制，让程序员能掌控局面。一旦你的Python 由于错误崩溃，解释程序就会转出一个“堆栈跟踪”，那里面有可用到的全部信息，包括你程序崩溃的原因以及是那段代码（文件名、行数、行数调用等等）出错了。这些错误被称为异常。如果在运行时发生这样的错误，Python 使你能够监控这些错误并进行处理。

这些异常处理可以采取相应的措施，例如解决问题、重定向程序流、执行清除或维护步骤、正常关闭应用程序、亦或干脆忽略掉。无论如何，这都可以有效的缩减开发周期中的调试环节。Python 的健壮性对软件设计师和用户而言都是大有助益的。一旦某些错误处理不当，Python 也还能提供一些信息，作为某个错误结果而产生的堆栈追踪不仅可以描述错误的类型和位置，还能指出代码所在模块。

**内存管理器：**

C 或者C++最大的弊病在于内存管理是由开发者负责的。所以哪怕是对于一个很少访问、修改和管理内存的应用程序，程序员也必须在执行了基本任务之外履行这些职责。这些加诸在开发者身上的没有必要的负担和责任常常会分散精力。

在Python 中，由于内存管理是由Python 解释器负责的，所以开发人员就可以从内存事务中解放出来，全神贯注于最直接的目标，仅仅致力于开发计划中首要的应用程序。这会使错误更少、程序更健壮、开发周期更短。

**解释性和（字节）编译性：**

Python 是一种解释型语言，这意味着开发过程中没有了编译这个环节。一般来说，由于不是以本地机器码运行，纯粹的解释型语言通常比编译型语言运行的慢。然而，类似于Java，Python实际上是字节编译的，其结果就是可以生成一种近似机器语言的中间形式。这不仅改善了Python的性能，还同时使它保持了解释型语言的优点。

注意：Python 源文件通常用.py 扩展名。当源文件被解释器加载或者显式地进行字节码编译的时候会被编译成字节码。由于调用解释器的方式不同，源文件会被编译成带有.pyc 或.pyo 扩展名的文。

## 1.4 Python环境搭建：

Python下载地址：<https://www.python.org/downloads/windows/>

Anaconda下载地址：<https://www.continuum.io/downloads/>

其中有两个不同的版本：

1. x & 3.x

适用于多个系统：

Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, 等等。)

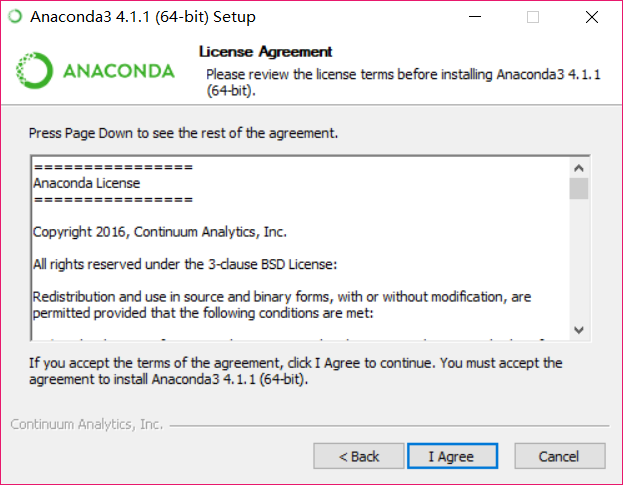
Windows

Mac

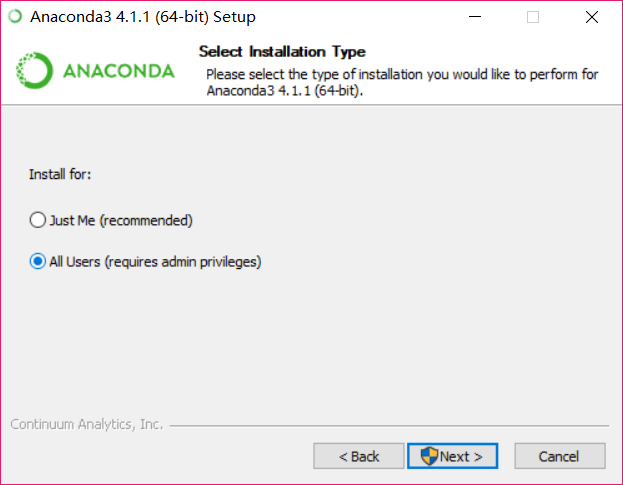
### 1.4.1 Anaconda的安装



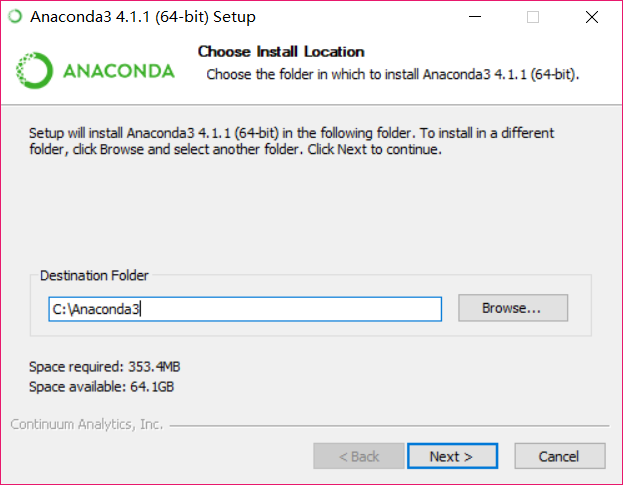
第二步：



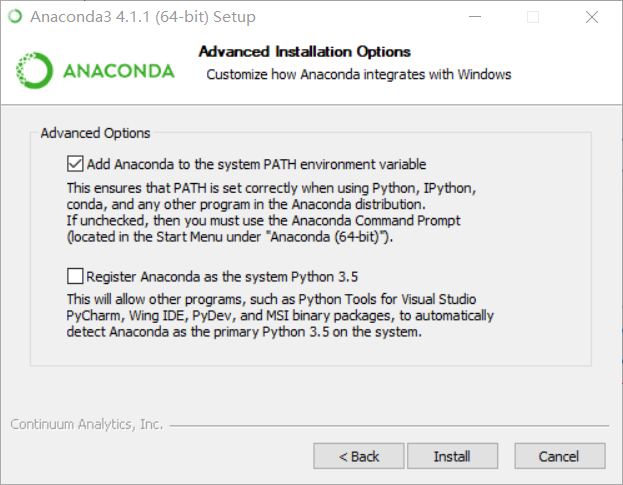
第三步：



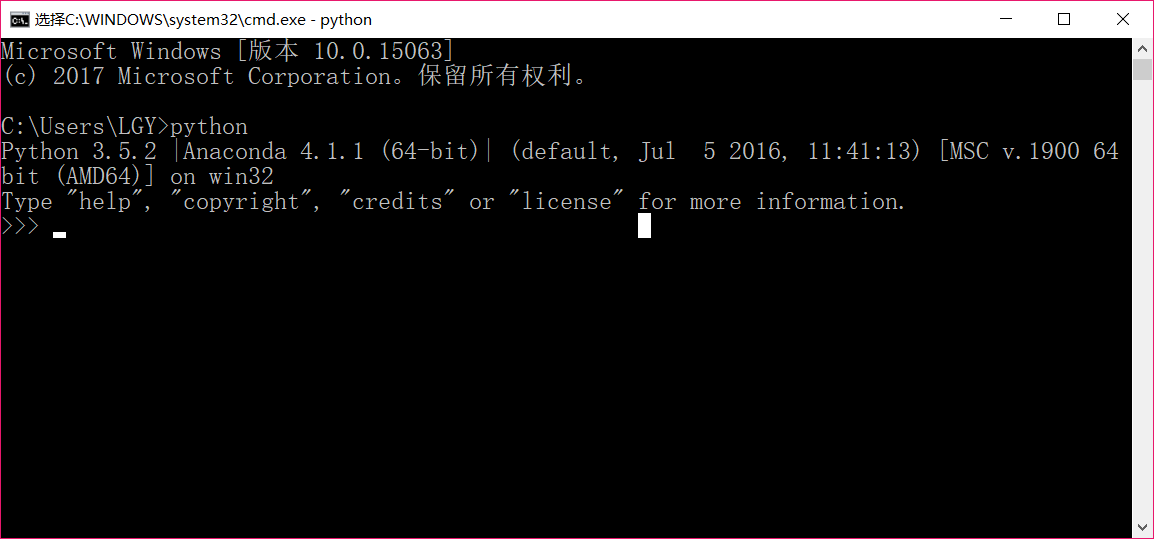
第四步：



第五步：

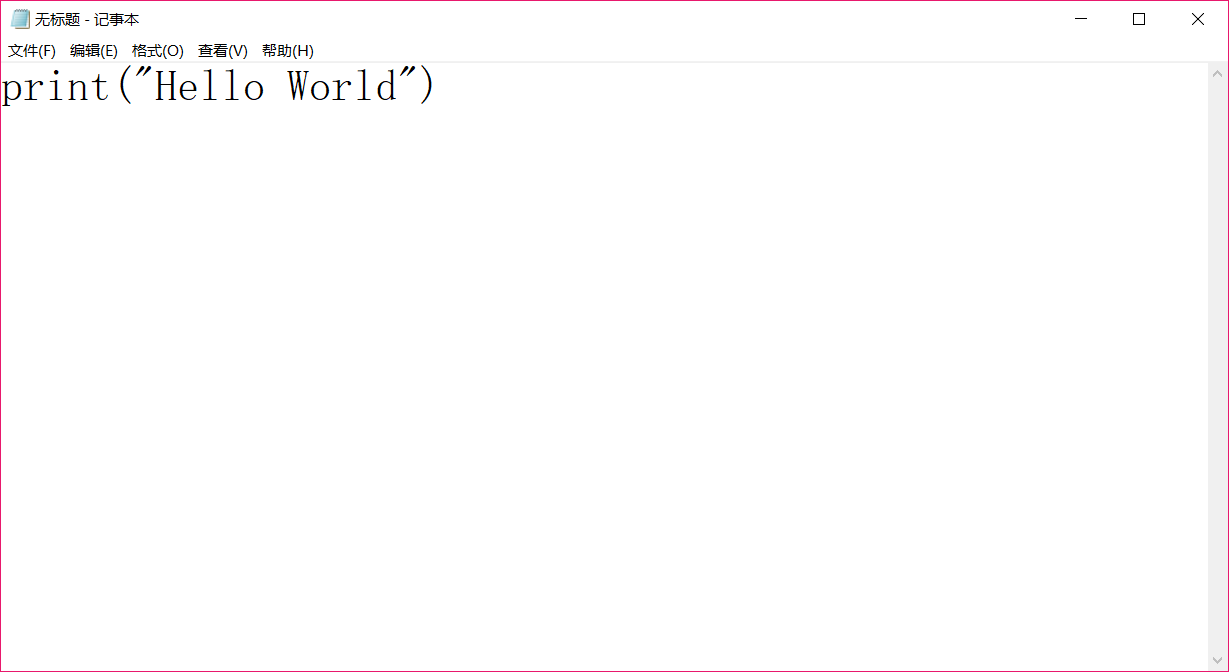


等待安装完成。。。。。。



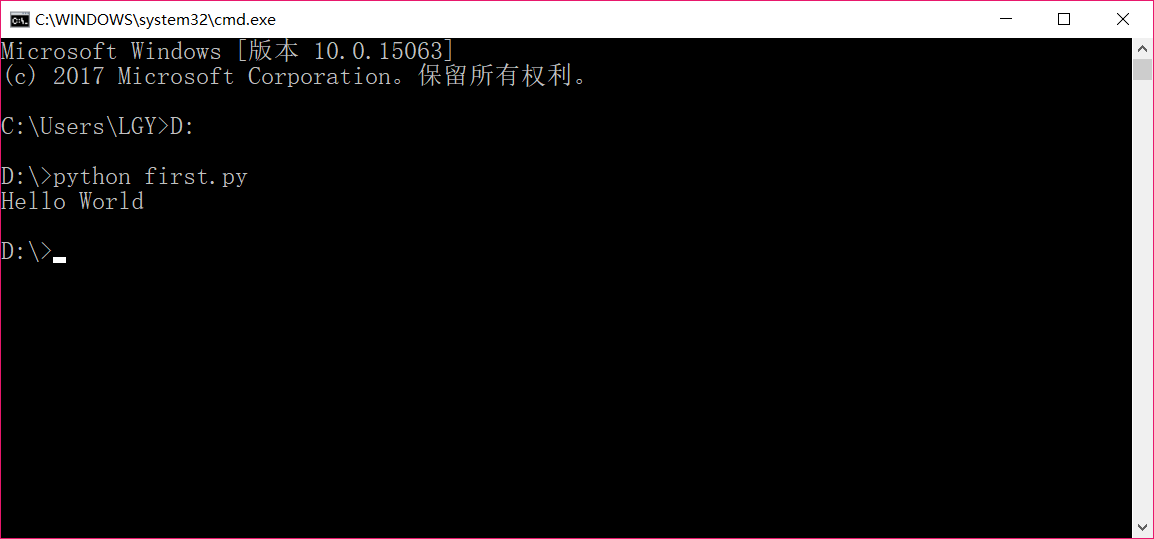
安装完成后呢！你点击Windows

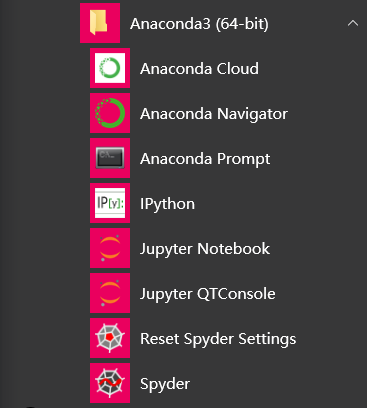
### 1.4.2 HelloWorld



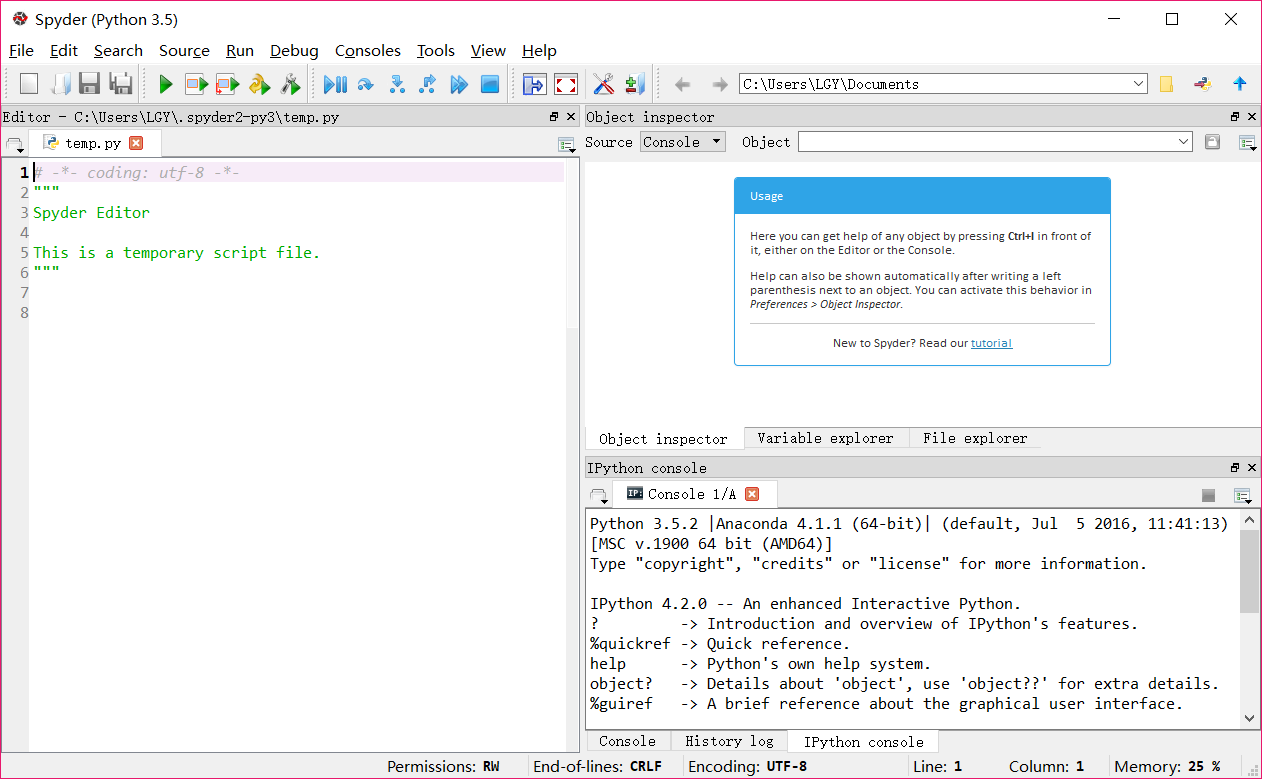


运行.py文件

1.4.3 Spyder（IDE）

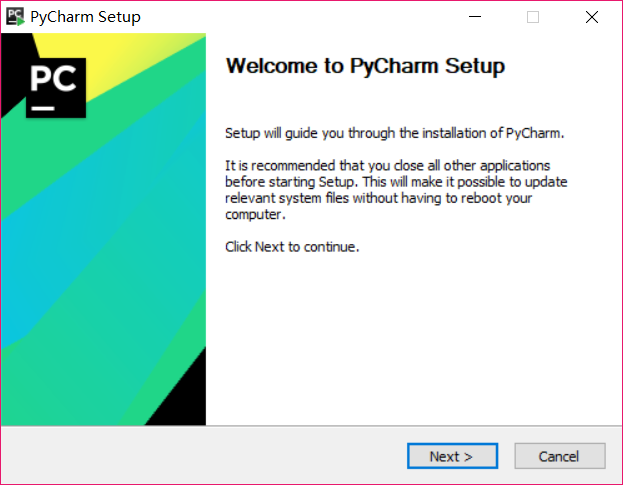


Spyder的最大优点就是模仿MATLAB的“工作空间”。

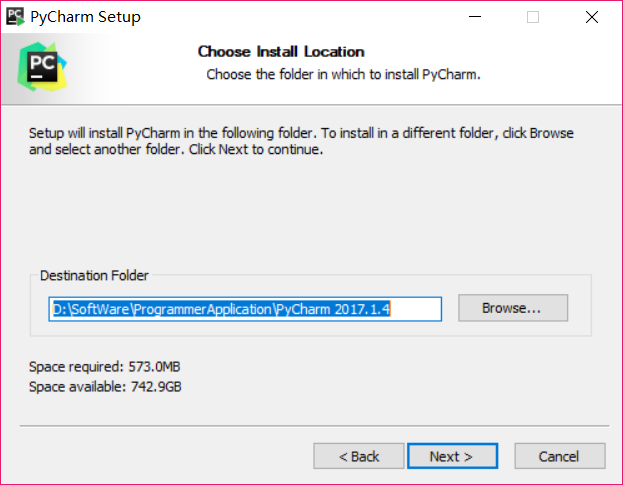


### 1.4.3 Pycharm的安装

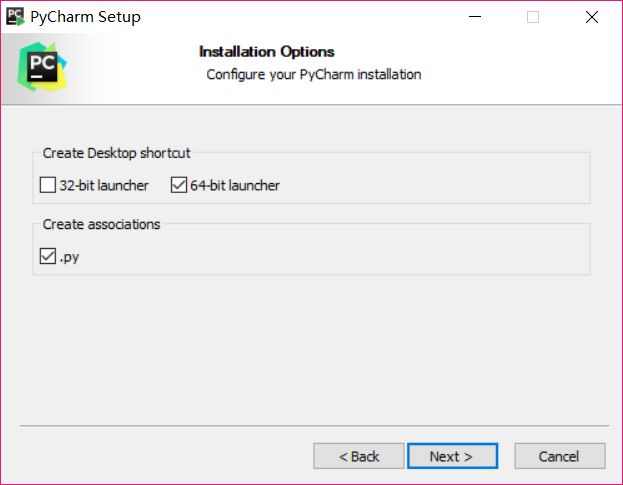
第一步：



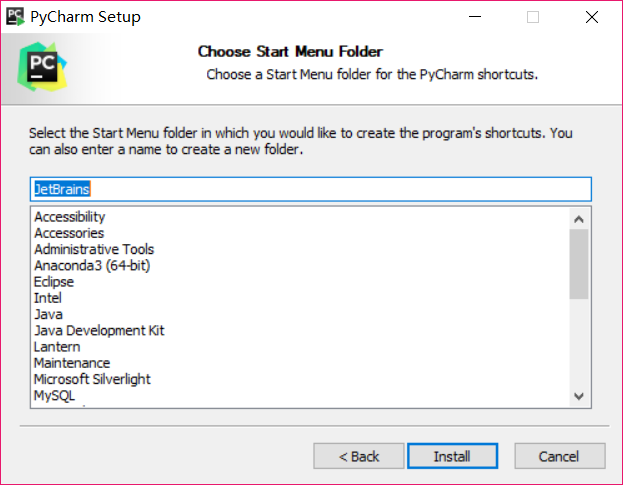
第二步：



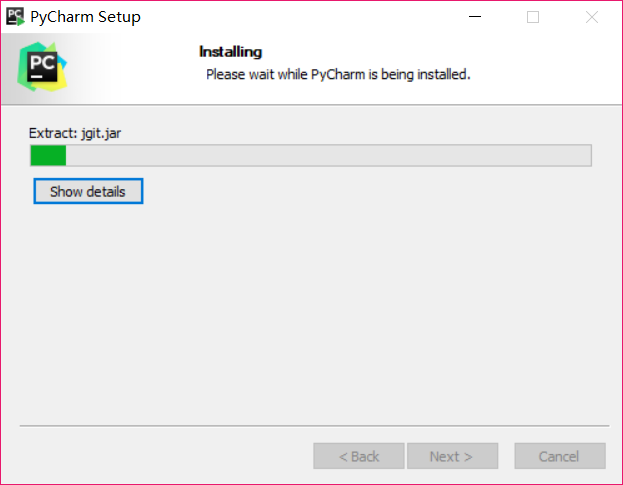
第三步：



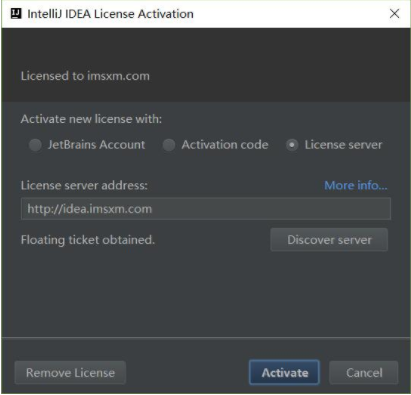
第四步：



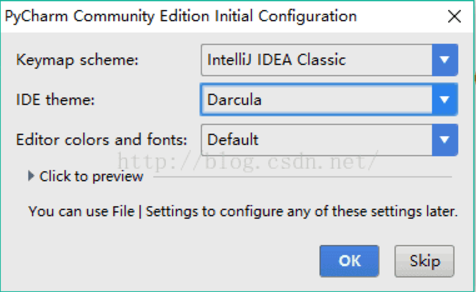
最后



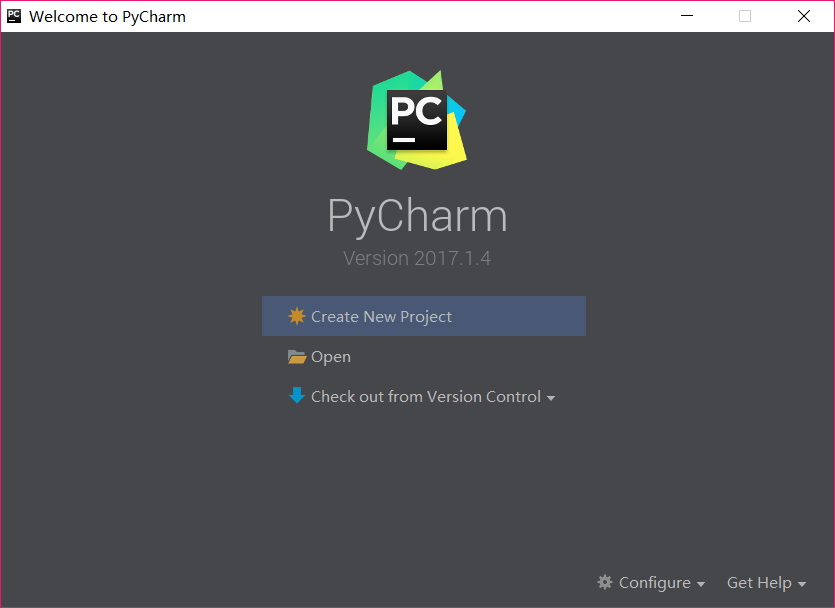
Pycharm激活码：[http://idea.imsxm.com](http://idea..com/)



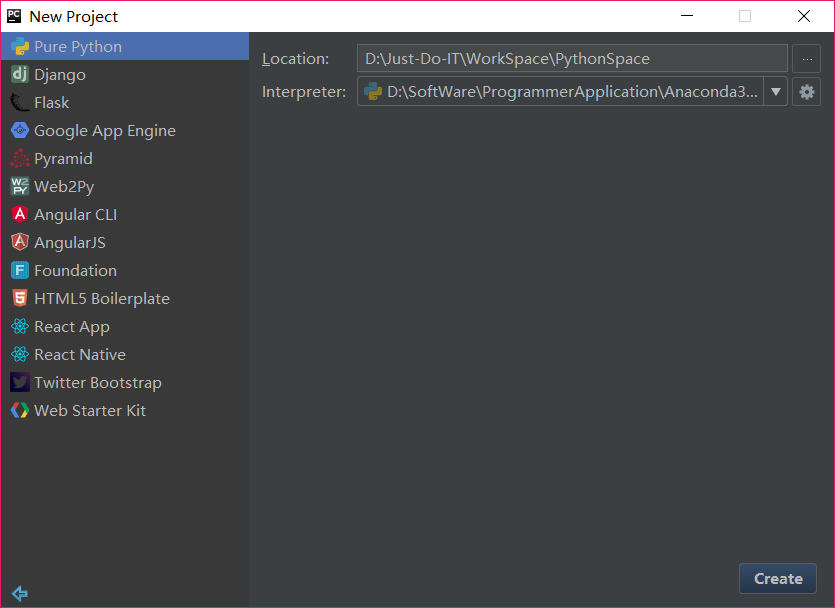
我们点击OK



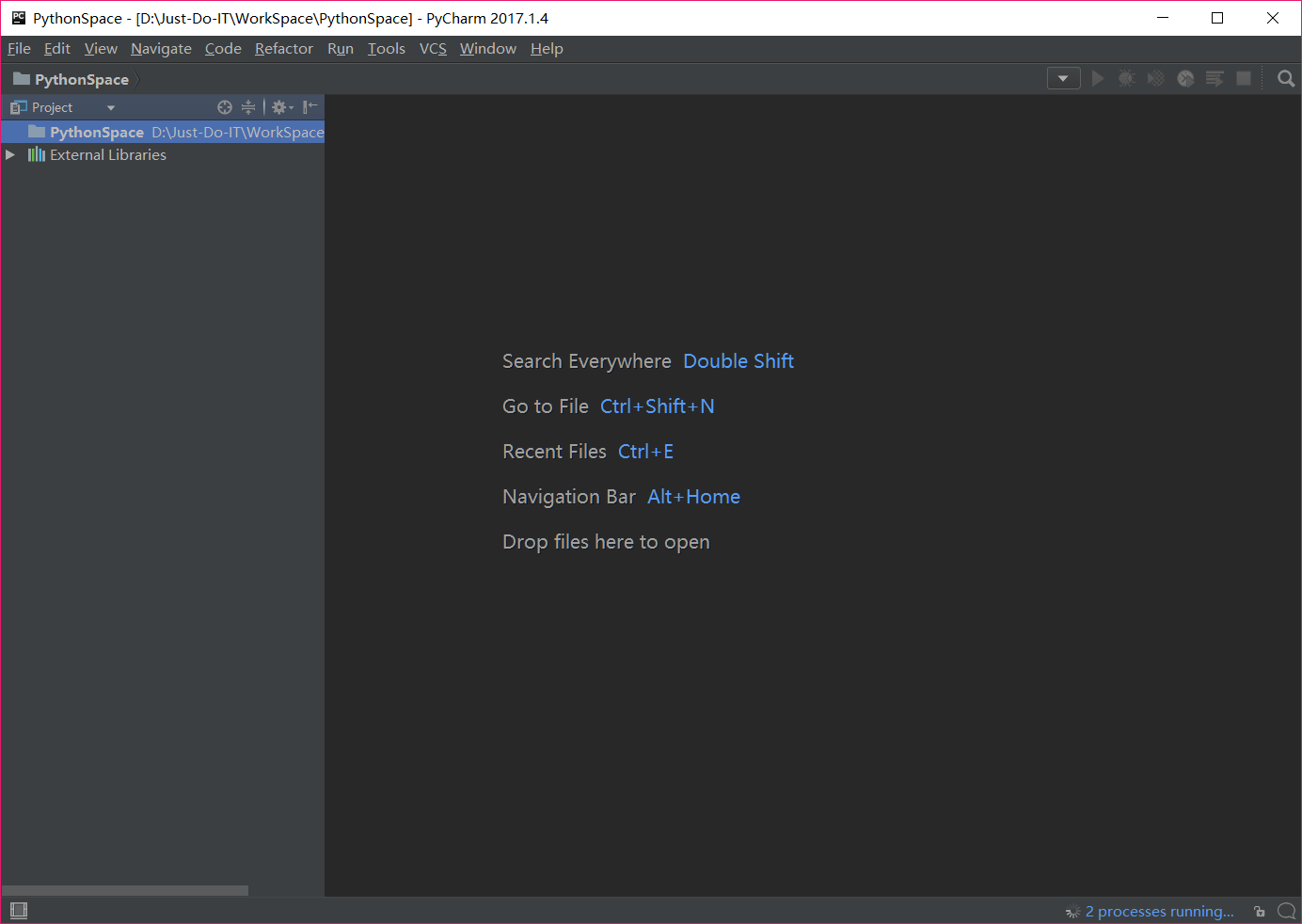
出现



点击Create New Project，进入如下图的界面，图中的Location是你要创建python的位置，选择好后，点击create。



创建完成



# 第二章Python3的基础语法

## 2.1编码

默认情况下，Python 3 源码文件以 UTF-8 编码，所有字符串都是 unicode 字符串。 当然你也可以为源码文件指定不同的编码：

*# -\*- coding:utf-8 -\*-*

## 2.2标识符

1、第一个字符必须是字母表中字母或下划线'\_'。

2、标识符的其他的部分有字母、数字和下划线组成。

4、标识符对大小写敏感。

## 2.3 保留字

保留字即关键字，我们不能把它们用作任何标识符名称。Python 的标准库提供了一个 keyword 模块，可以输出当前版本的所有关键字：

*# -\*- coding:utf-8 -\*-***import** keyword  
  
print(keyword.kwlist)

运行结果：

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

## 2.4 注释

1、Python中单行注释以 # 开头，实例如下：

*# -\*- coding:utf-8 -\*-  
  
#第一个python程序*print(**"Hello,Python"**) *#打印结果*

2、批量、多行注释符号

在python中也会有注释有很多行的时候，这种情况下就需要批量多行注释符了。多行注释是用三引号'''   '''或””” “””包含的，例如：

**'''  
该方法调用go方法  
需要返回一个字符串类型  
'''**

**或**

**"""  
该方法调用go方法  
需要返回一个字符串类型  
"""**

## 2.5行与缩进

python最具特色的就是使用缩进来表示代码块，不需要使用大括号({})。

缩进的空格数是可变的，但是同一个代码块的语句必须包含相同的缩进空格数。

**if True** :  
 print(**"is True"**)  
**else** :  
 print(**"is False"**)

如果出现缩进不一致的情况会出现，直接报错

**if True** :  
 print(**"is True"**)  
**else** :  
 print(**"is False"**)  
 print(**"Error"**)

File "D:/Just-Do-IT/WorkSpace/PythonSpace/test/first.py", line 15

print("00000000")

^

IndentationError: unexpected indent

## 2.6 多行语句

Python 通常是一行写完一条语句，但如果语句很长，我们可以使用反斜杠(\)来实现多行语句，例如：

a = 10  
b = 20  
c = 30  
  
sum = a + \  
 + b \  
 + c  
print(sum)

在 [], {}, 或 () 中的多行语句，不需要使用反斜杠(\)，例如：

sum = [a,  
 b , c]  
print(sum)

## 2.7 空行

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。

空行与代码缩进不同，空行并不是Python语法的一部分。书写时不插入空行，Python解释器运行也不会出错。但是空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。

记住：空行也是程序代码的一部分。

## 2.8 等待用户输入input

line = input(**"请输入,按下enter键后退出:"**)  
print(line)  
print(type(line))

以上代码中 ，打印**"请输入,按下enter键后退出:"**。一旦用户按下键时，程序将退出。

**注意：**在以前的2.x版本中有raw\_input()这个函数，而在现在的3.x版本中，因为版本进行了升级所以raw\_put()已经被废弃掉了。这两个函数在2.x的版本中是有区别的。

# 数据类型与运算符

## 3.1基本数据类型

Python 中的变量不需要声明。每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。

在 Python 中，变量就是变量，它没有类型，我们所说的"类型"是变量所指的内存中对象的类型。

等号（=）用来给变量赋值。

等号（=）运算符左边是一个变量名,等号（=）运算符右边是存储在变量中的值。例如：

*#-\*- coding:utf-8 -\*-*count = 1 *#整型变量*flag = **True** *#布尔型*name = **"LGY"** *#字符串类型*fnum = 12.5 *#浮点型*print(type(count))  
print(type(flag))  
print(type(name))  
print(type(fnum))

运行结果：

<class 'int'>

<class 'bool'>

<class 'str'>

<class 'float'>

### 如何为多个变量赋值？

同时为多个变量赋一个值

a = b = c = 1

您也可以为多个对象指定多个变量。

a , b , c = 1 , 2 , **"Python"**print(a,b,c)

运行结果：

1 2 Python

两个整型对象 1 和 2 的分配给变量 a 和 b，字符串对象 "python" 分配给变量 c。

### Python基本数据类型所占用的内存空间的大小

*#-\*- coding:utf-8 -\*-***import** sys  
  
count = 1 *#整型变量*flag = **True** *#布尔型*name = **"LGY"** *#字符串类型*fnum = 12.5 *#浮点型*print(**"%s Size %d"** %(type(count),sys.getsizeof(count)))  
print(**"%s Size %d"** %(type(flag),sys.getsizeof(flag)))  
print(**"%s Size %d"** %(type(name),sys.getsizeof(name)))  
print(**"%s Size %d"** %(type(fnum),sys.getsizeof(fnum)))

## 3.2 标准数据类型

Python3 中有六个标准的数据类型：

1、Number（数字）

2、String（字符串）

3、List（列表）

4、Tuple（元组）

5、Sets（集合）

6、Dictionary（字典）

### 3.2.1 Number（数字）

Python3 支持 int、float、bool、complex（复数）。

在Python 3里，只有一种整数类型 int，表示为长整型，没有 python2 中的 Long。像大多数语言一样，数值类型的赋值和计算都是很直观的。内置的 type() 函数可以用来查询变量所指的对象类型。

a , b ,c , d = 20 , 5.5 , **True** , 4 + 3j  
print(type(a),type(b),type(c),type(d))

<class 'int'> <class 'float'> <class 'bool'> <class 'complex'>

此外还可以用 isinstance 来判断：

a = 1  
data\_type1 = isinstance(a,int)  
data\_type2 = isinstance(a,bool)  
print(data\_type1,data\_type2)

运行结果：

True False

isinstance 和 type 的区别在于：

**class** A(object):  
 **pass  
  
class** B(A):  
 **pass**print(isinstance(A(),A),type(A())==A)  
print(isinstance(B(),A),type(B())==A)

运行结果：

True True

True False

区别就是:

type()不会认为子类是一种父类类型。

isinstance()会认为子类是一种父类类型。

**注意：**在 Python2 中是没有布尔型的，它用数字 0 表示 False，用 1 表示 True。到 Python3 中，把 True 和 False 定义成关键字了，但它们的值还是 1 和 0，它们可以和数字相加。

当你指定一个值时，Number 对象就会被创建：

var1 = 20;  
var2 = 21;

还可以使用del语句删除一些对象引用。

del语句的语法是：

del var1[,var2[,var3[....,varN]]]]

var1 = 20;  
var2 = 21;  
  
**del** var1 , var2

**数值运算实例：**

print(5 + 4) *#加法*print(4.3 - 2) *#减法*print(3 \* 7) *#乘法*print(2 / 4) *#除法，得到一个浮点数*print(2 // 4) *#除法，得到整数*print(17 % 3) *#取余*print(2 \*\* 6) *#乘方*

注意：

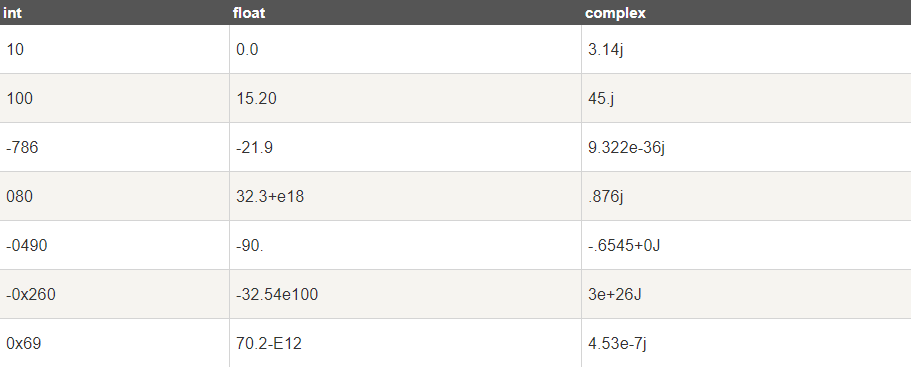
1、Python可以同时为多个变量赋值，如a, b = 1, 2。

2、一个变量可以通过赋值指向不同类型的对象。

3、数值的除法（/）总是返回一个浮点数，要获取整数使用//操作符。

4、在混合计算时，Python会把整型转换成为浮点数。

**数值类型实例**



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

### 3.2.2 String（字符串）

Python中的字符串用单引号(')或双引号(")括起来，同时使用反斜杠(\)转义特殊字符。

字符串的截取的语法格式如下：

变量[头下标:尾下标]

索引值以 0 为开始值，-1 为从末尾的开始位置。

加号 (+) 是字符串的连接符， 星号 (\*) 表示复制当前字符串，紧跟的数字为复制的次数。

**例:**

str = **"python"***#打印字符串*print(str)  
*#输出第一个到倒数第二个的所有字符*print(str[0:-1])  
*# 输出字符串第一个字符*print(str[0])  
*# 输出从第1个开始到第3个的字符*print(str[0:4])  
*#输出从第1个往后的全部字符串*print(str[1:])  
*#输出当前字符串两次*print((str+**" "**) \* 2)  
others = **"Tensorflow"**print(str + others)

运行结果：

python

pytho

p

pyth

ython

python python

pythonTensorflow

Python 使用反斜杠(\)转义特殊字符，如果你不想让反斜杠发生转义，可以在字符串前面添加一个 r/R，表示原始字符串。

*#加上\转义特殊字符*print(**"Te\nsorflow"**)  
*#加上r/R表示原始字符串*print(**r"Te\nsorflow"**)

另外，反斜杠(\)可以作为续行符，表示下一行是上一行的延续。也可以使用 """...""" 或者 '''...''' 跨越多行。

注意：Python 没有单独的字符类型，一个字符就是长度为1的字符串。

与 C 字符串不同的是，Python 字符串不能被改变。向一个索引位置赋值，比如word[0] = 'm'会导致错误。

如下：

word = **"python"**word[0] = **"a"**

**如何迭代一个字符串？**

words = **"python"  
for** word **in** words:  
 print(word)

**如何判断一个字符串是否包含(不包含)另一个字符串?**

python提供了in和not in关键字

a = **"python"**b = **"Hello python"**print(a **in** b)  
print(a **not in** b)

运行结果：

True

False

注意：

1、反斜杠可以用来转义，使用r可以让反斜杠不发生转义。

2、字符串可以用+运算符连接在一起，用\*运算符重复。

3、Python中的字符串有两种索引方式，从左往右以0开始，从右往左以-1开始。

4、Python中的字符串不能改变。

**字符串格式化**

Python 支持格式化字符串的输出 。尽管这样可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符 %s 的字符串中。

在 Python 中，字符串格式化使用与 C 中 printf 函数一样的语法。

**例:**

print(**"人生苦短，我用%s"**%(**"python。"**))  
print(**"1 + 1 = %s 或 %d"**%(**"田"**,2))

运行结果：

人生苦短，我用python。

1 + 1 = 田 或 2

**python字符串格式化符号:**



**Unicode 字符串**

在Python2中，普通字符串是以8位ASCII码进行存储的，而Unicode字符串则存储为16位unicode字符串，这样能够表示更多的字符集。使用的语法是在字符串前面加上前缀 u。

在Python3中，所有的字符串都是Unicode字符串。

**字符串内建函数**

1. capitalize()：将字符串的第一个字符转换为大写。

s = **"python"**print(str.capitalize(s))  
print(s.capitalize())

1. center(width, fillchar)：返回一个指定的宽度 width 居中的字符串，fillchar 为填充的字符，默认为空格。

s = **"python"**print(str.center(s,20,**"\*"**),s.center(20,**"\*"**))

1. count(str, beg= 0,end=len(string))：返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数

s = **"python"**print(s.count(**"t"**,1,len(s)))

1. bytes.decode(encoding="utf-8", errors="strict")：Python3 中没有 decode 方法，但我们可以使用 bytes 对象的 decode() 方法来解码给定的 bytes 对象，这个 bytes 对象可以由 str.encode() 来编码返回。errors -- 设置不同错误的处理方案。默认为 'strict',意为编码错误引起一个UnicodeError。 其他可能得值有 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' 以及通过 codecs.register\_error() 注册的任何值。

s = **"人生苦短，我用python。"**s\_utf = s.encode(**"UTF-8"**)  
s\_gbk = s.encode(**"GBK"**)  
print(s\_utf)  
print(s\_gbk)

print(s\_utf.decode(**"utf-8"**))  
print(s\_gbk.decode(**"gbk"**))

1. endswith(suffix, beg=0, end=len(string))：检查字符串是否以 obj 结束，如果beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False。

s = **"人生苦短，我用python。"**print(s.endswith(**"python。"** , 0 , len(s)))

1. expandtabs(tabsize=8)：把字符串 string 中的 tab 符号转为空格，tab 符号默认的空格数是 8 。tabsize -- 指定转换字符串中的 tab 符号('\t')转为空格的字符数。

s = **"人生苦\t短，我\t用python。"**print(s)  
print(s.expandtabs(1))

1. find(str, beg=0 end=len(string))：检测 str 是否包含在字符串中 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1。

s = **"人生苦短，我用python。"**print(s.find(**"苦"** , 1 , len(s)))

1. index(str, beg=0, end=len(string))：跟find()方法一样，只不过如果str不在字符串中会报一个异常。

s = **"人生苦短，我用python。"**print(s.index(**"苦"** , 0 , len(s)))

没有字符串抛出异常：

ValueError: substring not found

1. isalnum()：如果字符串至少有一个字符并且所有字符都是字母或数字则返回 True,否则返回 False。(方法检测字符串是否由字母和数字组成)

s1 = **"2017python"**print(s1.isalnum())  
s2 = **"python."**print(s2.isalnum())

1. isalpha():如果字符串至少有一个字符并且所有字符都是字母则返回 True, 否则返回 False。

s1 = **"python"**print(s1.isalpha())  
s2 = **"python2017"**print(s2.isalpha())

1. isdigit()：如果字符串只包含数字则返回 True 否则返回 False。

s = **"2017"**print(s.isdigit())

1. islower():如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False。

s = **"人生苦短，我用python。"**print(s.islower())

1. isnumeric()：如果字符串中只包含数字字符，则返回 True，否则返回 False。

s = **"2017"**print(s.isnumeric())

1. isspace()：如果字符串中只包含空格，则返回 True，否则返回 False。

s = **" "**print(s.isspace())

1. istitle()：如果字符串是标题化的(见 title())则返回 True，否则返回 False。(istitle() 方法检测字符串中所有的单词拼写首字母是否为大写，且其他字母为小写。)

s = **"Life Is Short, I Use Python."**print(s.istitle())

1. isupper():如果字符串中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False。

s = **"人生苦短，我用PYTHON。"**print(s.isupper())

1. join(seq):以指定字符串作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串。

s = **"-"**seq = (**"p"**,**"y"**,**"t"**,**"h"**,**"o"**,**"n"**)  
print(s.join(seq))

1. len(string)：判断字符串的长度。

s = **"人生苦短，我用PYTHON。"**print(len(s))

1. lower()\upper()：转换字符串中所有大\小写字符为小\大写。

s = **"人生苦短，我用PYTHON。"**print(s.lower())

s = **"Life Is Short, I Use Python."**print(s.upper())

1. lstrip()、rstrip()、strip([chars])：截掉字符串左边的空格。

s = **" 人生苦短，我用PYTHON。 ...."**print(s.lstrip())

s = **"pyt hon "**print(s.rstrip())

1. maketrans()：创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。

intab = **"abc"**outtab = **"123"**s1 = str.maketrans(intab,outtab)  
print(s1)

1. max(str)\min(str)：返回字符串 str 中最大\小的字母。

s = **"abcdr"**print(max(s))  
print(min(s))

1. replace(old, new [, max])：把将字符串中的 str1 替换成 str2,如果 max 指定，则替换不超过 max次。

s = **"abcadar"**print(s.replace(**"a"**,**"b"**,2))

1. rfind(str, beg=0,end=len(string))：类似于 find()函数，不过是从右边开始查找.

s = **"abcadar"**print(s.rfind(**"a"**,0,-1))

1. rindex( str, beg=0, end=len(string))：类似于 index()，不过是从右边开始.

s = **"abcadar"**print(s.rindex(**"a"**,0,-1))

没有找到会报错

1. rjust(width,[, fillchar])：返回一个原字符串右对齐,并使用fillchar(默认空格）填充至长度 width 的新字符串

s = **"python"**print(s.rjust(20,**"\*"**))

s = **"python"**print(s.ljust(20,**"\*"**))

1. split(str="", num=string.count(str))：num=string.count(str)) 以 str 为分隔符截取字符串，如果 num 有指定值，则仅截取 num 个子字符串。

s = **"p y t h o n"**print(s.split(**" "**))  
print(s.split(**" "**,3))

s = **"python\ndu"**print(s.split(**"\n"**))

1. splitlines([keepends])：按照行('\r', '\r\n', \n')分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符。

s = **"ab c\n\nde fg\rkl\r\n"**print(s.splitlines())  
print(s.splitlines(**True**))

1. startswith(str, beg=0,end=len(string))：检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查。

s = **"人生苦短，我用python。"**print(s.startswith(**"人"**,0,-1))

1. swapcase():将字符串中大写转换为小写，小写转换为大写。

s = **"PYTHON人生苦短，我用python。"**print(s.swapcase())

1. translate(table, deletechars="")：根据 str 给出的表(包含 256 个字符)转换 string 的字符, 要过滤掉的字符放到 deletechars 参数中。

table -- 翻译表，翻译表是通过maketrans方法转换而来。

deletechars -- 字符串中要过滤的字符列表。

s1 = **"abc123"**s2 = **"456abc"**m = str.maketrans(s1,s2)  
print(m)  
s3 = **"abcder13"**.translate(m)  
print(s3)

1. zfill (width)：返回长度为 width 的字符串，原字符串右对齐，前面填充0。

s = **"python"**print(s.zfill(20))

1. isdecimal()：检查字符串是否只包含十进制字符，如果是返回 true，否则返回 false。

s1 = **"abcd"**s2 = **"2017"**print(s1.isdecimal())  
print(s2.isdecimal())

### 3.2.3 List（列表）

List（列表） 是 Python 中使用最频繁的数据类型。

列表可以完成大多数集合类的数据结构实现。列表中元素的类型可以不相同，它支持数字，字符串甚至可以包含列表（所谓嵌套）。

列表是写在方括号([])之间、用逗号分隔开的元素列表。

和字符串一样，列表同样可以被索引和截取，列表被截取后返回一个包含所需元素的新列表。

列表截取的语法格式如下：

变量[头下标:尾下标]

Python有6个序列的内置类型，但最常见的是列表和元组。

**如何创建一个列表：**

l1 = [1,2,3,4,5]  
l2 = [**"python"**,**"java"**,**"google"**]  
print(l1,l2)  
print(l1[0:1])

与字符串的索引一样，列表索引从0开始。列表可以进行截取、组合等。

**访问列表中的值：**

使用下标索引来访问列表中的值，同样你也可以使用方括号的形式截取字符，如下所示：

l1 = [1,2,3,4,5]  
l2 = [**"python"**,**"java"**,**"google"**]  
*#获取l1中的第三个元素*print(l1[2])  
*#获取l2中第2个到最后一个元素*print(l2[1:])

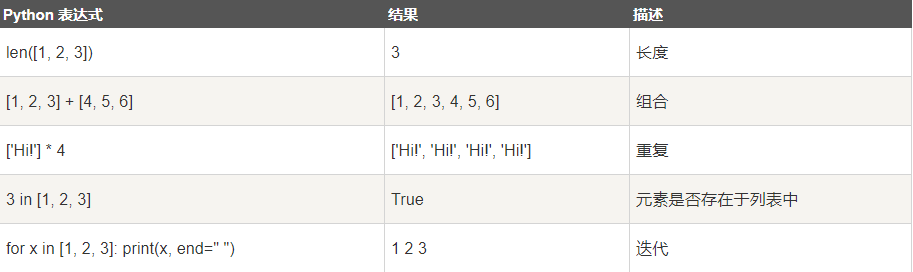
**更新列表：**

你可以对列表的数据项进行修改或更新，你也可以使用append()方法来添加列表项

l1 = [1,2,3,4,5]  
l2 = [**"python"**,**"java"**,**"google"**]  
l2[1] = **"C"**print(l2)  
*#一次批量更新*l1[0:3] = [0,0,0]  
print(l1)

**Python列表脚本操作符：**

列表对 + 和 \* 的操作符与字符串相似。+ 号用于组合列表，\* 号用于重复列表。



**嵌套列表：**

l = [[1,2,3,4,5],[**"A"**,**"B"**,**"C"**]]  
print(l)  
print(l[0][2:4])  
print(l[1][2])

**Python列表函数：**

1、len()：

l = [1,2,3,4,5]  
print(len(l))  
**for** x **in** l : print(x)

1. max(list)\min(list)：

l = [1,2,3,4,5]  
print(max(l))  
l1 = [**"A"**,**"b"**,**"C"**]  
print(max(l1))  
print(min(l1))

1. list(seq)：将元组转换为列表。

t = (**"python"**,**"java"**,(**"C"**,**"C++"**,**"Ruby"**))  
l = list(**"python"**)  
print(list(t))  
print(l)

1. list.append(obj)：在列表末尾添加新的对象。

l = [**"python"**,**"google"**,**"c"**]  
l.append(100)  
l.append(**"你好"**)  
print(l)

1. list.count(obj)：统计某个元素在列表中出现的次数。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
print(l.count(**"python"**))

1. list.extend(seq)：在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
print(l.extend(**"c++"**))  
print(l)  
l1 = list(range(5))  
l.extend(l1)  
print(l)

1. list.index(obj)：将对象插入列表。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
index = l.index(**"python"**)  
print(index)

1. list.insert(index, obj)：将对象插入列表。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l.insert(1,**"java"**)  
print(l)

9、 list.pop(obj=list[-1])：移除列表中的一个元素（默认最后一个元素），并且返回该元素的值。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l.pop()  
print(l)  
l.pop(1)  
print(l)

1. list.remove(obj)：移除列表中某个值的第一个匹配项。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l.remove(**"python"**)  
print(l)

1. list.reverse()：反向列表中元素。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l.reverse()  
print(l)

1. list.sort([func])：对原列表进行排序，func为可选函数。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l.sort()  
print(l)

1. list.clear()：清空列表。

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
print(l)  
l.clear()  
print(l)

1. list.copy()：复制列表

l = [**"python"**,**"google"**,**"c"**,**"python"**]  
l1 = l.copy()  
print(l1)  
l1.append(**"java"**)  
print(**"原："**+str(l))  
print(**"新："**+str(l1))

### 3.2.4 Tuple(元组)

Python 的元组与列表类似，不同之处在于元组的元素不能修改。

元组使用小括号，列表使用方括号。

元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

创建元祖

t = (**"python"**,**"google"**,**"c"**,**"python"**)  
print(t)

创建一个空元祖

t = ()  
print(t)

元组中只包含一个元素时，需要在元素后面添加逗号。

t = (**"python"**,)  
print(t)

元组与字符串类似，下标索引从0开始，可以进行截取，组合等。

**访问元组：**

t1 = (1,2,3,4,5,6,)  
t2 = (**"python"**,**"google"**,**"java"**,**"A"**)  
print(t1[1])  
print(t2[2])  
print(t1[3:60])

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合。

t1 = (1,2,3,4,5,6,)  
t2 = (**"python"**,**"google"**,**"java"**,**"A"**)  
print(t1 + t2)

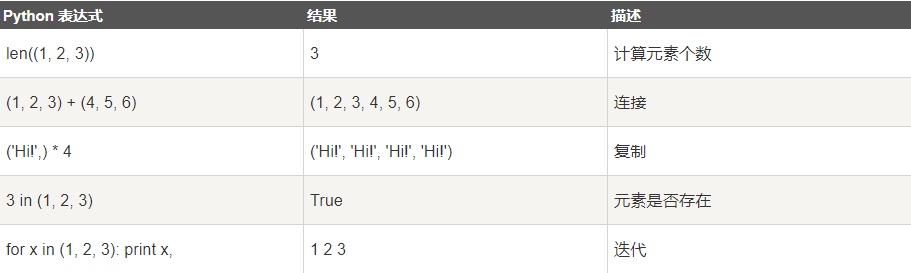
**删除元组：**

元组中的元素值是不允许删除的，但我们可以使用del语句来删除整个元组。

t1 = (1,2,3,4,5,6,)  
print(t1)  
**del** t1  
print(t1)

**元组运算符：**

与字符串一样，元组之间可以使用 + 号和 \* 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的元组。



**元组内置函数：**

1. len(tuple)：计算元组的长度。
2. max(tuple)：查找元组中最大的元素。
3. min(tuple)：查找元组中最小的元素。
4. tuple(seq)：将列表转换为元组。

### 3.2.5 Dict(字典)

字典是另一种可变容器模型，且可存储任意类型对象。

字典的每个键值(key=>value)对用冒号(:)分割，每个对之间用逗号(,)分割，整个字典包括在花括号({})中。

键必须是唯一的，但值则不必。

值可以取任何数据类型，但键必须是不可变的，如字符串，数字或元组。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
print(d1)  
d2 = {1:**"A"**,**"B"**:2,(1,2,3):**"c"**}  
print(d2)

**访问字典里的值：**

print(d1[**"Java"**])  
d2 = {1:**"A"**,**"B"**:2,(1,2,3):**"c"**}  
print(d2[(1,2,3)])

**修改字典：**

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
d1[**"C++"**] = 7000  
d1[**"Ruby"**] = 3000  
print(**"添加数据："**,d1)  
**del** d1[**"Java"**]  
print(**"删除数据："**,d1)  
**del** d1  
print(d1)

**字典键的特性：**

字典值可以没有限制地取任何python对象，既可以是标准的对象，也可以是用户定义的，但键不行。

需要注意以下两点：

1）不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住。

2）键必须不可变，所以可以用数字，字符串或元组充当，而用列表就不行。

**字典内置函数：**

1. len(dict)：计算字典元素个数，即键的总数。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
print(len(d1))

1. str(dict)：输出字典，以可打印的字符串表示。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
print(str(d1))

1. type(variable)：返回输入的变量类型，如果变量是字典就返回字典类型。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
print(type(d1))

1. dict.clear()：删除字典内所有元素。

d1 = {**"Java"**:9907,**"Python"**:9010,**"C++"**:6001}  
print(d1)  
d1.clear()  
print(d1)

1. dict.copy()：返回字典的浅复制。

直接赋值：其实就是对象的引用（别名）。

浅拷贝(copy)：拷贝父对象，不会拷贝对象的内部的子对象。

深拷贝(deepcopy)： copy 模块的 deepcopy 方法，完全拷贝了父对象及其子对象。

浅copy()

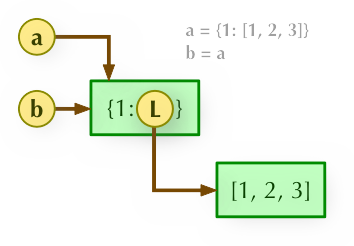
a = {1:[1,2,3]}  
b = a.copy()  
a[1].remove(1)  
print(a,b)

深copy()

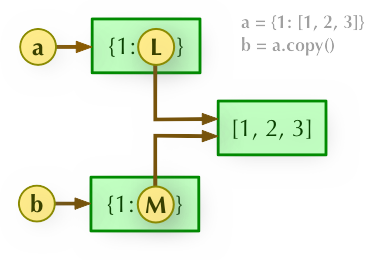
**import** copy  
  
a = {1:[1,2,3]}  
b = copy.deepcopy(a)  
a[1].append(4)  
print(a,b)

**解析：**

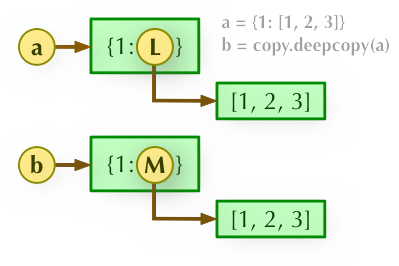
b = a: 赋值引用，a 和 b 都指向同一个对象。



b = a.copy(): 浅拷贝, a 和 b 是一个独立的对象，但他们的子对象还是指向统一对象（是引用）。



b = copy.deepcopy(a): 深度拷贝, a 和 b 完全拷贝了父对象及其子对象，两者是完全独立的。



1. dict.fromkeys()：创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应的初始值。

d = dict.fromkeys((1,2,3,4),[1,2,3,4])  
print(d)

1. dict.get(key, default=None)：返回指定键的值，如果值不在字典中返回default值。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
print(d1.get(1,**"没有"**))

1. key in dict：如果键在字典dict里返回true，否则返回false。
2. dict.items()：以列表返回可遍历的(键, 值) 元组数组。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
**for** item **in** d1.items():  
 print(item)

10、dict.keys()：以列表返回一个字典所有的键。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
keys = d1.keys()  
**for** key **in** keys:  
 print(d1.get(key))

1. dict.setdefault(key, default=None)：和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
d1.setdefault(4,**"D"**)  
print(d1)

1. dict.update(dict2)：把字典dict2的键/值对更新到dict里。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
d2 = {4:**"D"**}  
d3 = {1:**"B"**,2:**"A"**}  
d1.update(d2)  
print(d1)  
d1.update(d3)  
print(d1)

1. dict.values()：以列表返回字典中的所有值。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
print(d1.values())

1. pop(key[,default])：删除字典给定键 key 所对应的值，返回值为被删除的值。key值必须给出。 否则，返回default值。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
print(d1.pop(1))  
print(d1)

1. popitem()：随机返回并删除字典中的一对键和值。

d1 = {1:**"A"**,2:**"B"**,3:**"C"**}  
print(d1.popitem())  
print(d1)  
print(d1.popitem())

## 3.3运算符

**什么是运算符？**

1+3 = 4 其中的1和3是操作数，其中的 “+”就是运算符。

Python语言支持以下类型的运算符:

1、算术运算符

2、比较（关系）运算符

3、赋值运算符

4、逻辑运算符

5、位运算符

6、成员运算符

7、身份运算符

8、运算符优先级

**Python算术运算符：**



**Python比较运算符：**



**Python赋值运算符：**



**Python位运算符：**



变量 a 为 60，b 为 13二进制格式

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

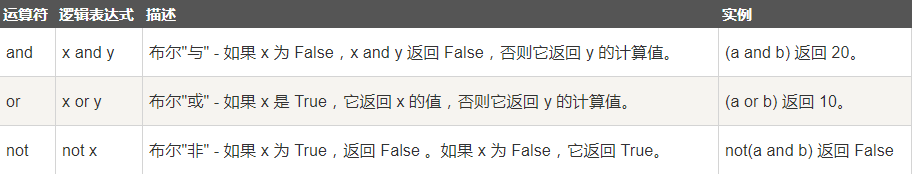
a^b = 0011 0001

~a = 1100 0011

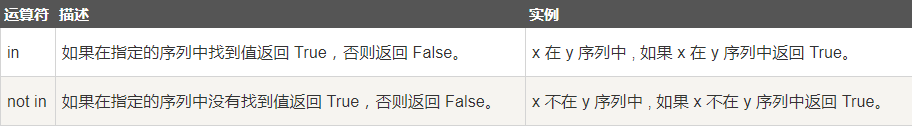
例：

a = 60 *# 60 = 0011 1100*b = 13 *# 13 = 0000 1101*c = 0  
  
c = a & b; *# 12 = 0000 1100*print(**"1 - c 的值为："**, c)  
  
c = a | b; *# 61 = 0011 1101*print(**"2 - c 的值为："**, c)  
  
c = a ^ b; *# 49 = 0011 0001*print(**"3 - c 的值为："**, c)  
  
c = ~a; *# -61 = 1100 0011*print(**"4 - c 的值为："**, c)  
  
c = a << 2; *# 240 = 1111 0000*print(**"5 - c 的值为："**, c)  
  
c = a >> 2; *# 15 = 0000 1111*print(**"6 - c 的值为："**, c)

**Python逻辑运算符：**

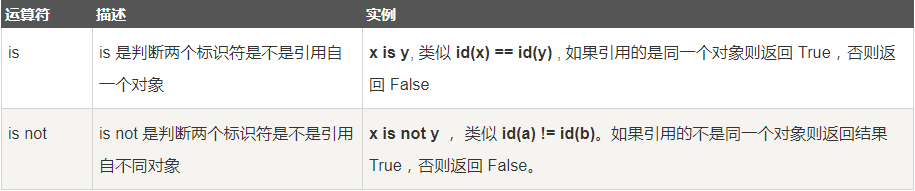


**Python成员运算符：**



**Python身份运算符：**

用于比较两个对象的存储单元



id() 函数用于获取对象内存地址。

is 与 == 区别：

is 用于判断两个变量引用对象是否为同一个， == 用于判断引用变量的值是否相等。

**Python运算符优先级：**

从最高到最低优先级的所有运算符：



# 函数

## 4.1什么是函数？

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数能提高应用的模块性，和代码的重复利用率。你已经知道Python提供了许多内建函数，比如print()。但你也可以自己创建函数，这被叫做用户自定义函数。

## 4.2 如何定义一个函数？

你可以定义一个由自己想要功能的函数，以下是简单的规则：

1、函数代码块以 def 关键词开头，后接函数标识符名称和圆括号()。

2、任何传入参数和自变量必须放在圆括号中间。圆括号之间可以用于定义参数。

3、函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。

4、函数内容以冒号起始，并且缩进。

5、return [表达式] 结束函数，选择性地返回一个值给调用方。不带表达式的return相当于返回 None。

*#-\*-coding:utf-8-\*-***def** func():  
 print(**"定义第一个函数！！！！"**)

定义一个有返回值的函数

**def** func():  
 print(**"定义第一个函数！！！！"**)  
 **return "我是函数"**,20  
  
par1,par2 = func()  
print(par1)  
print(par2)

## 4.3参数传递

*#-\*-coding:utf-8-\*-*a = [1,2,3]  
a = **"tensorflow"**

[1,2,3]是List类型而**"tensorflow"**是String类型，这里的a是没有类型的，他仅仅是一个对象的引用(一个指针)，他既可以指向List类型的对象，也可以指向String类型的对象。

**可更改(mutable)与不可更改(immutable)对象**

在 python中，String, tuple,和number是不可更改的对象，而 list,dict 等则是可以修改的对象。

这里会具体解释：

**不可变类型**：变量赋值a=5后再赋值a=10，这里实际是新生成一个int值对象10，再让a指向它，而5被丢弃，不是改变a的值，相当于新生成了a。

**可变类型**：变量赋值 l=[1,2,3,4] 后再赋值 l[2]=5 则是将 list l的第三个元素值更改，本身l没有动，只是其内部的一部分值被修改了。

**python 函数的参数传递：**

**不可变类型：**类似 c++ 的值传递，如整数、字符串、元组。如fun（a），传递的只是a的值，没有影响a对象本身。比如在 fun（a）内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。

**可变类型：**类似 c++ 的引用传递，如 列表，字典。如 fun（l），则是将l真正的传过去，修改后fun外部的l也会受影响。

python中一切都是对象，严格意义我们不能说值传递还是引用传递，我们应该说传不可变对象和传可变对象。

**python 传不可变对象实例**

*#-\*-coding:utf-8-\*-***def** ChangeInt(a):  
 a = 2  
b = 1  
ChangeInt(b)  
print(b)

实例中有 int 对象 1，指向它的变量是 b，在传递给 ChangeInt 函数时，按传值的方式复制了变量 b，a 和 b 都指向了同一个 Int 对象，在 a=2 时，则新生成一个 int 值对象 2，并让 a 指向它。

**Python传可变对象实例**

*#-\*-coding:utf-8-\*-***def** noChange(l):  
 l.append(**"A"**)  
 print(l)  
  
l = [1,2,3]  
noChange(l)  
print(l)

**不定长参数**

有时可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，声明时不会命名。

**def** printDemo(args,\*data):  
 print(args)  
 **for** word **in** data:  
 print(word)  
  
printDemo(**"hello"**,1,2,3,4)

# 第五章 条件控制与循环 迭代器与生成器

Python条件语句是通过一条或多条语句的执行结果（True或者False）来决定执行的代码块。

可以通过下图来简单了解条件语句的执行过程:

## 5.1 if 语句

Python中if语句的一般形式如下所示：

if condition\_1: statement\_block\_1

elif condition\_2: statement\_block\_2

else: statement\_block\_3

如果 "condition\_1" 为 True 将执行 "statement\_block\_1" 块语句

如果 "condition\_1" 为False，将判断 "condition\_2"

如果"condition\_2" 为 True 将执行 "statement\_block\_2" 块语句

如果 "condition\_2" 为False，将执行"statement\_block\_3"块语句

Python 中用 elif 代替了 else if，所以if语句的关键字为：if – elif – else。

**注意：**

1、每个条件后面要使用冒号（:），表示接下来是满足条件后要执行的语句块。

2、使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。

3、在Python中没有switch – case语句。

例1：

a = 100  
b = 0  
**if** a:  
 print(**"a条件为True"**)  
  
**if** b:  
 print(**"b条件为True"**)

这时候呢if b不执行

例2：

判断年龄

age = int(input(**"请输入年龄:"**))  
  
**if** age < 18:  
 print(**"未成年人"**)  
**elif** age > 18 **and** age < 30:  
 print(**"年轻人"**)  
**elif** age >=30 **and** age < 50:  
 print(**"中年人"**)  
**else**:  
 print(**"老年人"**)

if 嵌套：

if 表达式1:

语句

if 表达式2:

语句

elif 表达式3:

语句

else:

语句

elif 表达式4:

语句

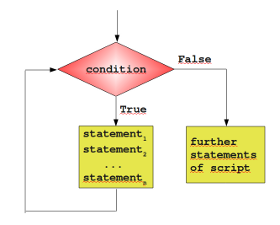
else:

语句

## 5.2 循环语句

Python中的循环语句有 for 和 while。

Python循环语句的控制结构图如下所示：



while 循环：

while 判断条件：

语句

同样需要注意冒号和缩进。另外，在Python中没有do..while循环。

以下实例使用了 while 来计算 1 到 100 的总和：

count = 100  
sum = 0;  
**while** count:  
 sum += count  
 count -= 1  
  
  
print(sum)

无限循环

我们可以通过设置条件表达式永远不为 false 来实现无限循环。

**while** 1 == 1:  
 s = input(**"请输入内容："**)  
 print(s)  
 **if** s == **"exit"**:  
 **break**print(**"Good Bye"**)

while 循环使用 else 语句：

在 while … else 在条件语句为 false 时执行 else 的语句块：

count = 1  
**while** count < 5:  
 print(str(count)+**"小于5"**)  
 count += 1  
**else**:  
 print(str(count) +**"大于等于5"**)

简单语句组：

类似if语句的语法，如果你的while循环体中只有一条语句，你可以将该语句与while写在同一行中。

flag = **True  
  
while** flag: print(**"人生苦短，我用python！"**)  
print(**"Good Bye"**)

for 语句

Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

for循环的一般格式如下：

for <variable> in <sequence>:

<statements>

else: <statements>

items = [1,2,3,4,5,6]  
  
**for** item **in** items:  
 print(item)

range()函数

**for** i **in** range(10):  
 print(i)  
  
print(**"==============================="**)  
**for** j **in** range(5,10):  
 print(j)

可以使range以指定数字开始并指定不同的增量(甚至可以是负数，有时这也叫做'步长'):

**for** i **in** range(1,20,2):  
 print(i)

**for** i **in** range(20,1,-1):  
 print(i)

break和continue语句及循环中的else子句：

break 语句可以跳出 for 和 while 的循环体。如果你从 for 或 while 循环中终止，任何对应的循环 else 块将不执行。

**for** i **in** range(20,1,-1):  
 **if** i == 6:  
 **break** print(i)  
print(**"Break"**)

continue语句被用来告诉Python跳过当前循环块中的剩余语句，然后继续进行下一轮循环。

**for** i **in** range(20,1,-1):  
 **if** i == 6:  
 **continue** print(i)  
print(**"continue"**)

pass 语句：

Python pass是空语句，是为了保持程序结构的完整性。

pass 不做任何事情，一般用做占位语句。

例：

**if True**:  
 **pass  
  
for** i **in** range(6):  
 **pass  
  
while True**:  
 **pass  
  
  
  
class** Pass(object):  
 **pass  
  
  
def** go():  
 **pass**

## 5.3迭代器

**什么是迭代？**

可以直接作用于for循环的对象统称为可迭代对象(Iterable)。

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)。

所有的Iterable均可以通过内置函数iter()来转变为Iterator。

迭代是Python最强大的功能之一，是访问集合元素的一种方式。

迭代器是一个可以记住遍历的位置的对象。

迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：iter() 和 next()。

字符串，列表或元组对象都可用于创建迭代器：

l = [1,2,3,4,5,6,7]  
it = iter(l)  
  
print(next(it))  
print(next(it))  
print(next(it))  
print(next(it))

迭代器对象可以使用常规for语句进行遍历：

l = [1,2,3,4,5,6,7]  
it = iter(l)  
  
**for** i **in** it:  
 print(i,end=**"\n"**)

使用next()函数：

**import** sys  
  
l = [1,2,3,4,5,6,7]  
it = iter(l)  
  
**while True**:  
 **try**:  
 print(next(it))  
 **except** StopIteration:  
 sys.exit()

题外话：

1. 内置函数iter()仅仅是调用了对象的\_\_iter()方法，所以list对象内部一定存在方法iter\_\_()
2. 内置函数next()仅仅是调用了对象的\_\_next()方法，所以list对象内部一定不存在方法next\_\_()，但是Itrator中一定存在这个方法。
3. for循环内部事实上就是先调用iter()把Iterable变成Iterator在进行循环迭代的。

## 5.4 生成器

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。

例：我们要生成要生成list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

l = range(1,11)  
print(l)

要生成[1x1, 2x2, 3x3, ..., 10x10]怎么做？

方法一：

l = []  
  
  
**for** i **in** range(1,11):  
 l.append(i \* i)  
  
print(l)

方法二：

我们来用列表生成式来做：

l = [i \* i **for** i **in** range(1,11)]  
print(l)

还可以使用两层循环，可以生成全排列：

l = [a + b **for** a **in "ABC" for** b **in "ZYX"**]  
print(l)

列出当前目录下的所有文件和目录名：

**import** os  
  
l = [file **for** file **in** os.listdir(**"D:/"**)]  
print(l)

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 = [i \* i **for** i **in** range(1,11)]  
print(type(l))  
  
print(l)  
l = (i \* i **for** i **in** range(1,11))  
print(type(l))  
print(l)  
print(next(l))

我们讲过，generator保存的是算法，每次调用next(l)，就计算出l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(l)实在是（变态），正确的方法是使用for循环，因为generator也是可迭代对象。

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,end=**" "**)  
 a , b = b , a + b  
 n = n + 1  
  
fib(6)

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  
  
**for** i **in** fib(6):  
 print(i)

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** pop():  
 print(**"step1"**)  
 **yield** 1  
 print(**"step2"**)  
 **yield** 3  
 print(**"step3"**)  
 **yield** 5  
  
p = pop()  
  
print(next(p))  
print(next(p))  
print(next(p))

# 第六章 函数式编程

**介绍：**

函数是Python内建支持的一种封装，我们通过把大段代码拆成函数，通过一层一层的函数调用，就可以把复杂任务分解成简单的任务，这种分解可以称之为面向过程的程序设计。函数就是面向过程的程序设计的基本单元。

而函数式编程，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

这里我们首先要搞明白一个概念：

计算机（Computer）和计算（Compute）的概念。

在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

Python对函数式编程提供部分支持。由于Python允许使用变量，因此，Python不是纯函数式编程语言。

## 6.1 高阶函数

### 6.1.1 什么是高阶函数？

这里我们结合案例一步一步来看，最后总结出概念。

**变量可以指向函数**

以Python内置的求绝对值的函数abs()为例：

print(abs(-1))  
print(abs)

运行结果？

1

<built-in function abs>

可见，abs(-10)是函数调用，而abs是函数本身。

要获得函数调用结果，我们可以把结果赋值给变量。

但是，如果把函数本身赋值给变量呢？

f = abs  
print(f)  
print(f(-1))

说明变量f现在已经指向了abs函数本身。直接调用abs()函数和调用变量f()完全相同。

### 6.1.2 函数名也是变量

那么函数名是什么呢？函数名其实就是指向函数的变量！对于abs()这个函数，完全可以把函数名abs看成变量，它指向一个可以计算绝对值的函数！

abs = 1  
abs(1)

Traceback (most recent call last):

File "D:/Just-Do-IT/WorkSpace/PythonSpace/test/test/\_\_init\_\_.py", line 15, in <module>

abs(1)

TypeError: 'int' object is not callable

把abs指向10后，就无法通过abs(-10)调用该函数了！因为abs这个变量已经不指向求绝对值函数而是指向一个整数10！

当然实际代码绝对不能这么写，这里是为了说明函数名也是变量。要恢复abs函数，请重启Python交互环境(这里我们用的IDE所以不用重启)。

注：由于abs函数实际上是定义在import builtins模块中的，所以要让修改abs变量的指向在其它模块也生效，要用import builtins; builtins.abs = 10。

### 6.1.3 传入函数

既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

**def** add(x, y, f):  
 **return** f(x) + f(y)

当我们调用add(-5, 6, abs)时，参数x，y和f分别接收-5，6和abs，根据函数定义，我们可以推导计算过程为：

x = -5

y = 6

f = abs

f(x) + f(y) ==> abs(-5) + abs(6) ==> 11

return 11

print(add(-5, 6, abs))

编写高阶函数，就是让函数的参数能够接收别的函数。把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

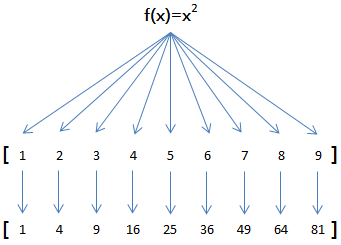
### 6.1.4 map()/reduce()

Python内建了map()和reduce()函数

如果熟悉Google大名鼎鼎的论文：“MapReduce: Simplified Data Processing on Large Clusters”，你就能大概明白map/reduce的概念。当然没听过也没事，我们接下来会慢慢讲解。

Map：map()函数接收两个参数，一个是函数，一个是Iterable，map将传入的函数依次作用到序列的每个元素，并把结果作为新的Iterator返回。

例：比如我们有一个函数f(x)=x^2，要把这个函数作用在一个list [1, 2, 3, 4, 5, 6, 7, 8, 9]上，就可以用map()实现如下：



**def** f(x):  
 **return** x \* x  
l = [1,2,3,4,5,6,7,8,9]  
results = map(f,l)  
print(list(results))

map()传入的第一个参数是f，即函数对象本身。由于结果r是一个Iterator，Iterator是惰性序列，因此通过list()函数让它把整个序列都计算出来并返回一个list。

当然你可能会想为什么我们要用map呢？不需要map()函数，写一个循环，也可以计算出结果：

**def** f(x):  
 **return** x \* x  
l = [1,2,3,4,5,6,7,8,9]  
  
  
L = []  
**for** n **in** l:  
 L.append(f(n))  
print(L)

的确可以，但是，从上面的循环代码，能一眼看明白“把f(x)作用在list的每一个元素并把结果生成一个新的list”吗？

所以，map()作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的f(x)=x2，还可以计算任意复杂的函数，比如，把这个list所有数字转为字符串：

l = [1,2,3,4,5,6,7,8,9]  
  
print(list(map(str, l)))

Reduce：reduce把一个函数作用在一个序列[x1, x2, x3, ...]上，这个函数必须接收两个参数，reduce把结果继续和序列的下一个元素做累积计算，其效果就是：

reduce(f, [x1, x2, x3, x4]) = f(f(f(x1, x2), x3), x4)

比方说对一个序列求和，就可以用reduce实现：

**from** functools **import** reduce  
  
**def** add(x,y):  
 **return** x + y  
  
print(reduce(add,[1,2,3,4,5,6,7,8]))

求和运算可以直接用Python内建函数sum()，没必要动用reduce。

但是如果要把序列[1, 3, 5, 7, 9]变换成整数13579，reduce就可以派上用场：

**from** functools **import** reduce  
  
**def** fn(x , y):  
 **return** x \* 10 + y  
  
  
print(reduce(fn,[1,3,5,7,9]))

例题：

1、如果考虑到字符串str也是一个序列，对上面的例子稍加改动，配合map()，我们就可以写出把str转换为int的函数。

2、利用map()函数，把用户输入的不规范的英文名字，变为首字母大写，其他小写的规范名字。输入：['adam', 'LISA', 'barT']，输出：['Adam', 'Lisa', 'Bart']：

3、利用map和reduce编写一个str2float函数，把字符串'123.456'转换成浮点数123.456。

### 6.1.4 filter

Python内建的filter()函数用于过滤序列。

类似于map()，filter()也是接受一个函数和一个序列。和map()不同的是，filter()把传入的函数依次作用于每一个元素，然后根据返回值是True和False决定保留还是丢弃该元素。

*#删除列表中的所有偶数***def** remove(i):  
 **return** i % 2 == 1  
  
print(list(filter(remove,[1,2,3,4,5,6])))

把一个序列中的空字符串删掉。

**def** not\_empty(s):  
 **return** s **and** s.strip();  
  
print(list(filter(not\_empty,[**"A"**,**""**,**"B"**,**"C"**,**"d"**])))

可以看出用filter()这个高阶函数，关键在于正确实现一个“筛选”函数。

注意到filter()函数返回的是一个Iterator，也就是一个惰性序列，所以要强迫filter()完成计算结果，需要用list()函数获得所有结果并返回list。

## 6.2 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：

# 模块

## 7.1 模块简介

在日常程序的开发过程中，随着程序代码量越来越多，在一个文件里代码就会越来越长，最后会慢慢变得越来越不容易维护。

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例如：Java、C#、C++等。在Python中，一个.py文件就称之为一个模块（Module）。

**模块对于我们的日常开发有什么好处呢？**

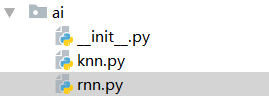
最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。

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。

大家可能还会有些以问，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）。

例：一个knn.py的文件就是一个名叫knn的模块，一个rnn.py的文件就是一个名叫rnn的模块。

假设我们的knn和rnn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ai，按照如下目录存放：



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，knn.py模块的名字就变成了ai.knn，类似的，rnn.py的模块名变成了ai.rnn。

**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ai。

**\_\_init\_\_.py的作用**

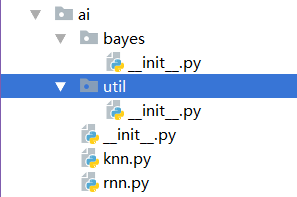
每个目录下都有\_\_init\_\_.py文件，这个是初始化模块，from-import语句导入子包时需要它，可以在里面做一些初始化工作，也可以是空文件。

注意：\_\_init\_\_.py定义的属性直接使用 顶层包.子包 的方式导入，如在目录a的\_\_init\_\_.py文件中定义init\_db()方法，调用如下：

from util import app

app.init\_db()

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：



## 7.2 模块的使用

### 7.2.1 使用import语句导入模块

有下面两种方式：

第一种：

import module1

import module2

import module3

第二种：

import module1,module2,module3

这两种方式的效果是一样的，但是第一种可读性比第二种好，推荐按照下面的顺序导入模块，并且一般在文件首部导入所有的模块：

* python标准库
* 第三方模块
* 应用程序自定义模块

也可以在函数内部导入模块，这样被导入的模块作用域是局部的

### 7.2.2 使用from-import语句导入模块的属性

单行导入

**from** ai.knn **import** demo

多行导入

**from** ai.knn **import** demo1,\  
 demo2

导入全部属性（由于容易覆盖当前名称空间中现有的名字，所以一般不推荐使用，适合模块中变量名很长并且变量很多的情况）

from module import \*

如果你不想某个模块的属性被以上方法导入，可以给该属性名称前加一个下划线(\_test)，如果需要取消隐藏，可以显示的导入该属性（from module import \_test）

## 7.3 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：name，age，PI等一系列的属性。

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** \_\_privateDemo1():  
 **return "张三"  
  
def** \_\_privateDemo2():  
 **return "李四"  
  
  
def** getting(count):  
 **if** count == 1:  
 **return** \_\_privateDemo1()  
 **else**:  
 **return** \_\_privateDemo2()

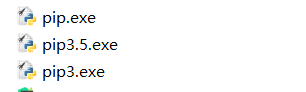
我们在模块里公开getting()函数，而把内部逻辑用private函数隐藏起来了，这样，调用getting()函数不用关心内部的private函数细节，这也是一种非常有用的代码封装和抽象的方法，即：

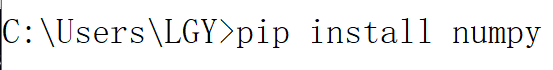
外部不需要引用的函数全部定义成private，只有外部需要引用的函数才定义为public。

## 7.4 第三方模块的安装

在Python中，安装第三方模块，是通过包管理工具pip来完成。

如果是Mac或Linux，安装pip（包管理工具）本身这个步骤就可以跳过了。





pip install 第三方模块名称。

# 面向对象

## 8.1简介

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信息，为了表示一个学生的信息，面向过程的程序可以用一个dict表示：

stu1 = {**"name"**:**"张三"**,**"age"**:20,**"sex"**:**"男"**}  
stu2 = {**"name"**:**"李四"**,**"age"**:20,**"sex"**:**"男"**}

而处理学生信息可以通过函数实现，比如打印学生的具体信息：

**def** printMsg(stu):  
 print(**"name: %s,age: %d,sex: %s"** %(stu[**"name"**],stu[**"age"**],stu[**"sex"**]))  
  
printMsg(stu1)

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、age和sex这三个属性（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printMsg消息，让对象自己把自己的数据打印出来。

**class** Student:  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.name = name  
 self.age = age  
 self.sex = sex  
  
 **def** printMsg(self):  
 print(**"name: %s,age: %d,sex: %s"** %(self.name,self.age,self.sex))  
  
stu = Student(**"晓宇"**,22,**"女"**)  
stu.printMsg()

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。

## 8.2 类和实例

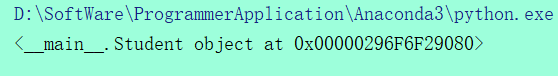
面向对象最重要的概念就是类（Class）和实例（Instance），必须牢记类是抽象的模板，我们拿Student类为例，而实例是根据类创建出来的一个个具体的“对象”，每个对象都拥有相同的方法，但各自的数据可能不同。

**class** Student(object):  
 **pass**

class后面紧跟类名，即Student，类名通常是大写开头的单词，紧接着是(object)，表示该类是从哪个类继承下来（继承会在后面部分讲解），通常，如果没有合适的继承类，就使用object类，这是所有类最终都会继承的类。

如何创建实例：

stu = Student(**"晓宇"**,22,**"女"**)  
print(stu)



可以看到，变量stu指向的就是一个Student的实例，后面的0x00000296F6F29080是内存地址，每个实例的地址都不一样，而Student本身则是一个类。

可以自由地给一个实例变量绑定属性，比如，给实例stu绑定一个score属性：

stu.score = 98  
print(stu.score)

由于类可以起到模板的作用，因此，可以在创建实例的时候，把一些我们认为必须绑定的属性强制填写进去。通过定义一个特殊的\_\_init\_\_方法，在创建实例的时候，就把name，age，sex等属性绑上去：

**class** Student(object):  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.name = name  
 self.age = age  
 self.sex = sex

注意到\_\_init\_\_方法的第一个参数永远是self，表示创建的实例本身，因此，在\_\_init\_\_方法内部，就可以把各种属性绑定到self，因为self就指向创建的实例本身。

有了\_\_init\_\_方法，在创建实例的时候，就不能传入空的参数了，必须传入与\_\_init\_\_方法匹配的参数，但self不需要传，Python解释器自己会把实例变量传进去。

和普通的函数相比，在类中定义的函数只有一点不同，就是第一个参数永远是实例变量self，并且，调用时，不用传递该参数。除此之外，类的方法和普通函数没有什么区别，所以，你仍然可以用默认参数、可变参数、关键字参数和命名关键字参数。

**数据封装**

面向对象编程的一个重要特点就是数据封装。在上面的Student类中，每个实例就拥有各自的name、age和sex这些数据。我们可以通过函数来访问这些数据，比如打印一个学生的信息：

**def** printMsg(self):  
 print(**"name: %s,age: %d,sex: %s"** %(self.name,self.age,self.sex))

但是，既然Student实例本身就拥有这些数据，要访问这些数据，就没有必要从外面的函数去访问，可以直接在Student类的内部定义访问数据的函数，这样，就把“数据”给封装起来了。这些封装数据的函数是和Student类本身是关联起来的，我们称之为类的方法：

**class** Student(object):  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.name = name  
 self.age = age  
 self.sex = sex  
  
 **def** printMsg(self):  
 print(**"name: %s,age: %d,sex: %s"** %(self.name,self.age,self.sex))

要定义一个方法，除了第一个参数是self外，其他和普通函数一样。要调用一个方法，只需要在实例变量上直接调用，除了self不用传递，其他参数正常传入。

这样一来，我们从外部看Student类，就只需要知道，创建实例需要给出name和age、sex，而如何打印，都是在Student类的内部定义的，这些数据和逻辑被“封装”起来了，调用很容易，但却不用知道内部实现的细节。

封装的另一个好处是可以给Student类增加新的方法，比如get\_lastname：

**class** Student(object):  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.name = name  
 self.age = age  
 self.sex = sex  
  
 **def** printMsg(self):  
 print(**"name: %s,age: %d,sex: %s"** %(self.name,self.age,self.sex))  
  
 **def** get\_lastname(self):  
 **return** self.name[0]  
  
stu = Student(**"张晓宇"**,22,**"女"**)  
print(stu.get\_lastname())

同样get\_lastname()方法可以直接在实例变量上调用，不需要知道内部实现细节。

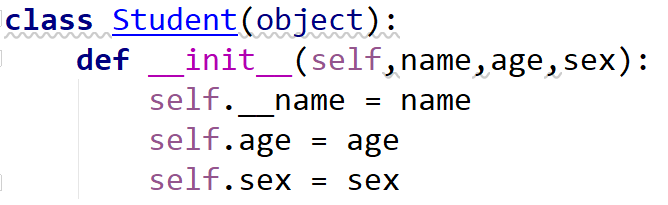
## 8.3 访问限制

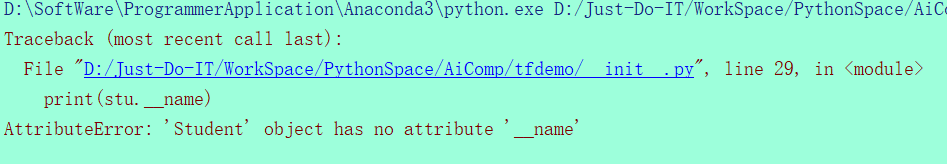
在class内部，可以有属性和方法，而外部代码可以通过直接调用实例变量的方法来操作数据，这样，就隐藏了内部的复杂逻辑。

但是，从前面Student类的定义来看，外部代码还是可以自由地修改一个实例的name、age、sex属性：

stu = Student(**"张晓宇"**,22,**"女"**)  
print(stu.name)  
print(stu.age)  
print(stu.sex)

如果要让内部属性不被外部访问，可以把属性的名称前加上两个下划线\_\_，在Python中，实例的变量名如果以\_\_开头，就变成了一个私有变量（private），只有内部可以访问，外部不能访问，所以，我们把Student类改一改：





这样就确保了外部代码不能随意修改对象内部的状态，这样通过访问限制的保护，代码更加健壮。

但是如果外部代码要获取name怎么办？可以给Student类增加get\_name这样的方法。

**def** get\_name(self):  
 **return** self.\_\_name

如果这时候想修改name属性值,那么增加set\_name（）方法即可。

**def** set\_name(self,name):  
 self.\_\_name = name

你也许会问，原先那种直接通过stu.age = 99也可以修改啊，为什么要定义一个方法大费周折？因为在方法中，可以对参数做检查，避免传入无效的参数。

例如：人类目前的年龄不能可能大于200岁。

注意：

在Python中，变量名类似\_\_xxx\_\_的，也就是以双下划线开头，并且以双下划线结尾的，是特殊变量，特殊变量是可以直接访问的，不是private变量，所以，不能用\_\_name\_\_、\_\_age\_\_这样的变量名。

有些时候，你会看到以一个下划线开头的实例变量名，比如\_name，这样的实例变量外部是可以访问的，但是，按照约定俗成的规定，当你看到这样的变量时，意思就是，“虽然我可以被访问，但是，请把我视为私有变量，不要随意访问”。

双下划线开头的实例变量是不是一定不能从外部访问呢？其实也不是。不能直接访问\_\_name是因为Python解释器对外把\_\_name变量改成了\_Student\_\_name，所以，仍然可以通过\_Student\_\_name来访问\_\_name变量：

stu = Student(**"张晓宇"**,22,**"女"**)  
print(stu.\_Student\_\_name)  
print(stu.age)  
print(stu.sex)

但是强烈建议你不要这么干，因为不同版本的Python解释器可能会把\_\_name改成不同的变量名。

总的来说就是，Python本身没有任何机制阻止你干坏事，一切全靠自觉。

最后注意下面的这种错误写法：

stu = Student(**"张晓宇"**,22,**"女"**)  
print(stu.get\_name())  
stu.\_\_name = **"小臭鱼"**print(stu.get\_name())

表面上看，外部代码“成功”地设置了\_\_name变量，但实际上这个\_\_name变量和class内部的\_\_name变量不是一个变量！内部的\_\_name变量已经被Python解释器自动改成了\_Student\_\_name，而外部代码给stu新增了一个\_\_name变量。不信试试：

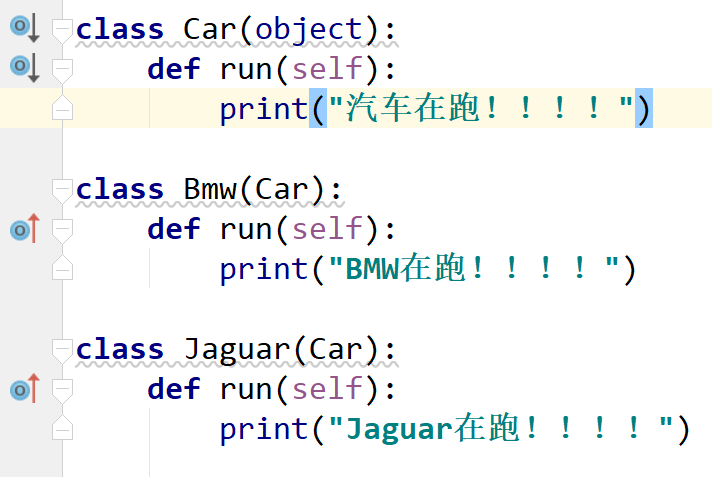
## 8.4 继承和多态

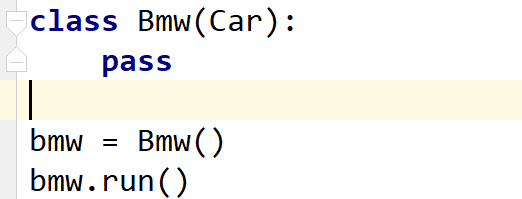
在OOP程序设计中，当我们定义一个class的时候，可以从某个现有的class继承，新的class称为子类（Subclass），而被继承的class称为基类、父类或超类（Base class、Super class）。

比如，我们已经编写了一个名为Cat的class，有一个run()方法可以直接打印：

**class** Car(object):  
 **def** run(self):  
 print(**"汽车在跑！！！！"**)

当我们需要编写Bmw和Jaguar类时，就可以直接从Cat类继承：





这时候实际是调用的Car类的run函数。

当子类和父类都存在相同的run()方法时，我们说，子类的run()覆盖了父类的run()，在代码运行的时候，总是会调用子类的run()。这样，我们就获得了继承的另一个好处：多态。

要理解什么是多态，我们首先要对数据类型再作一点说明。当我们定义一个class的时候，我们实际上就定义了一种数据类型。我们定义的数据类型和Python自带的数据类型，比如str、list、dict没什么两样：

a = list()  
b = Car()  
c = Bmw()  
  
print(isinstance(a,list))  
print(isinstance(b,Car))  
print(isinstance(c,Bmw))  
**'''可是尝试一下'''**print(isinstance(c,Car))

True

True

True

True

看来c不仅仅是Bmw，c还是Car。

不过仔细想想，这是有道理的，因为Bmw是从Car继承下来的，当我们创建了一个Bmw的实例c时，我们认为c的数据类型是Bmw没错，但c同时也是Car也没错，Bmw本来就是Car的一种！

所以，在继承关系中，如果一个实例的数据类型是某个子类，那它的数据类型也可以被看做是父类。但是，反过来就不行：

print(isinstance(b,Bmw))

False

Bmw可以看成Car，但Car不可以看成Bmw。

要理解多态的好处，我们还需要再编写一个函数，这个函数接受一个Car类型的变量：

**class** Car(object):  
 **def** run(self):  
 print(**"汽车在跑！！！！"**)  
  
**class** Bmw(Car):  
 **def** run(self):  
 print(**"BMW在跑！！！！"**)  
  
**class** Jaguar(Car):  
 **def** run(self):  
 print(**"Jaguar在跑！！！！"**)  
  
  
**def** run\_demo(car):  
 car.run()  
 car.run()  
  
run\_demo(Car())  
run\_demo(Bmw())  
run\_demo(Jaguar())

运行结果：

汽车在跑！！！！

汽车在跑！！！！

BMW在跑！！！！

BMW在跑！！！！

Jaguar在跑！！！！

Jaguar在跑！！！！

看上去没啥意思，但是仔细想想，现在，如果我们再定义一个Benz类型，也从Car派生。

你会发现，新增一个Car的子类，不必对run\_demo()做任何修改，实际上，任何依赖Car作为参数的函数或者方法都可以不加修改地正常运行，原因就在于多态。

多态的好处就是，当我们需要传入Bmw、Jaguar、Tortoise……时，我们只需要接收Car类型就可以了，因为Bmw、Jaguar、Benz……都是Car类型，然后，按照Car类型进行操作即可。由于Car类型有run()方法，因此，传入的任意类型，只要是Car类或者子类，就会自动调用实际类型的run()方法，这就是多态的意思：

对于一个变量，我们只需要知道它是Car类型，无需确切地知道它的子类型，就可以放心地调用run()方法，而具体调用的run()方法是作用在Car、Bmw、Jaguar还是Benz对象上，由运行时该对象的确切类型决定，这就是多态真正的威力：调用方只管调用，不管细节，而当我们新增一种Car的子类时，只要确保run()方法编写正确，不用管原来的代码是如何调用的。这就是著名的“开闭”原则：

对扩展开放：允许新增Car子类；

对修改封闭：不需要修改依赖Car类型的run\_demo()等函数。

继承还可以一级一级地继承下来，就好比从爷爷到爸爸、再到儿子这样的关系。而任何类，最终都可以追溯到根类object，这些继承关系看上去就像一颗倒着的树。比如如下的继承树：

图略。。。。。。。。。。。。。。。。

**静态语言 & 动态语言**

对于静态语言（例如Java、C++、C）来说，如果需要传入Car类型，则传入的对象必须是Car类型或者它的子类，否则，将无法调用run()方法，也可以说是在编译器就要确定类型。而对于Python这样的动态语言来说，则不一定需要传入Car类型。我们只需要保证传入的对象有一个run()方法就可以了：

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

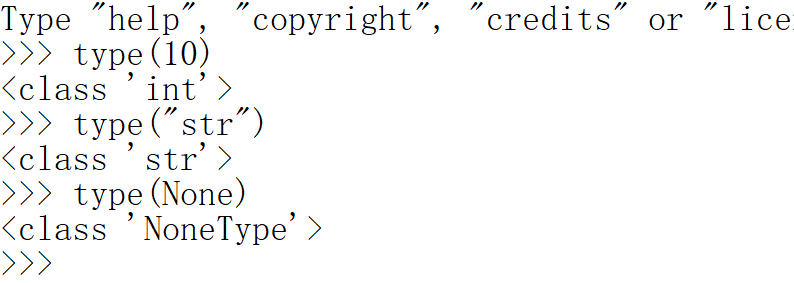
Python的“file-like object“就是一种鸭子类型。对真正的文件对象，它有一个read()方法，返回其内容。但是，许多对象，只要有read()方法，都被视为“file-like object“。许多函数接收的参数就是“file-like object“，你不一定要传入真正的文件对象，完全可以传入任何实现了read()方法的对象。

## 8.5 获取对象信息

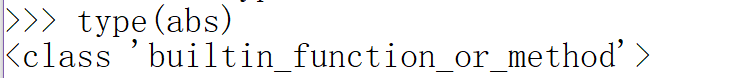
当我们拿到一个对象的引用时，如何知道这个对象是什么类型、有哪些方法呢？

### type()函数：

基本类型都可以用type()判断：



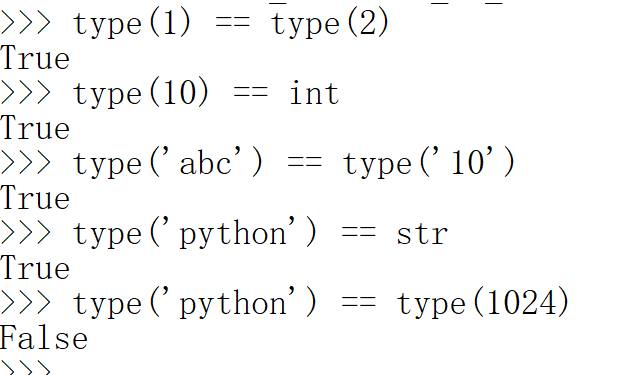
如果一个变量指向函数或者类，也可以用type()判断：



**class** Student(object):  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.\_\_name = name  
 self.age = age  
 self.sex = sex  
 **def** get\_name(self):  
 **return** self.\_\_name  
 **def** set\_name(self,name):  
 self.\_\_name = name  
 **def** printMsg(self):  
 print(**"name: %s,age: %d,sex: %s"** %(self.name,self.age,self.sex))  
  
 **def** get\_lastname(self):  
 **return** self.name[0]  
  
stu = Student(**"张晓宇"**,22,**"女"**)  
print(type(stu))

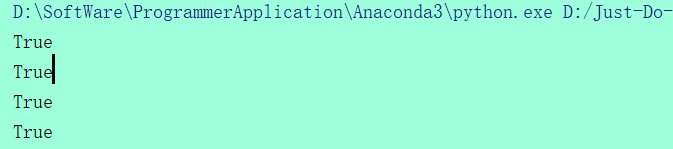
<class '\_\_main\_\_.Student'>

但是type()函数返回的是什么类型呢？它返回对应的Class类型。如果我们要在if语句中判断，就需要比较两个变量的type类型是否相同：



如果想判断基本数据类型可以直接写int，str等，但如果要判断一个对象是否是函数怎么办？可以使用types模块中定义的常量：

**import** types  
  
**def** func():  
 **pass**print(type(func) == types.FunctionType)  
print(type(abs) == types.BuiltinFunctionType)  
print(type(**lambda** x: x) == types.LambdaType)  
print(type((x **for** x **in** range(10))) == types.GeneratorType)



### isinstance()函数的使用

对于class的继承关系来说，使用type()就很不方便。我们要判断class的类型，可以使用isinstance()函数。

我们回顾上次的例子，如果继承关系是：

object -> Car -> Volkswagen -> Porsche

那么，isinstance()就可以告诉我们，一个对象是否是某种类型。先创建3种类型的对象：

a = Car()  
d = Volkswagen()  
c = Porsche()

isinstance(c, Porsche)  
isinstance(c, Volkswagen)  
isinstance(c, Car)

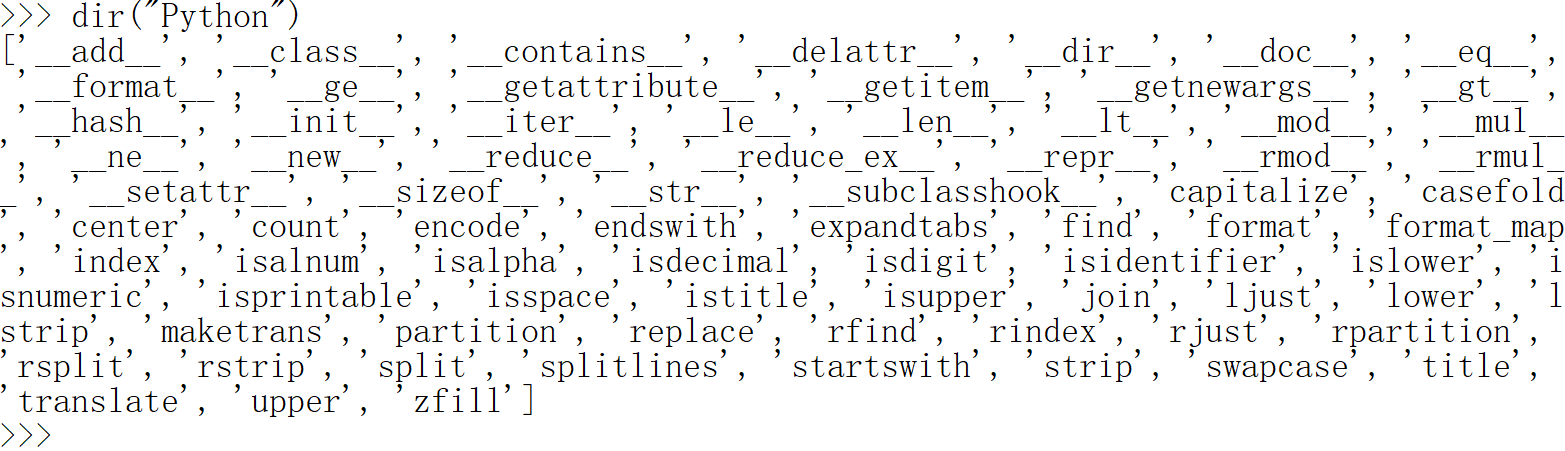
运行结果为：True True True

并且还可以判断一个变量是否是某些类型中的一种，比如下面的代码就可以判断是否是list或者tuple：

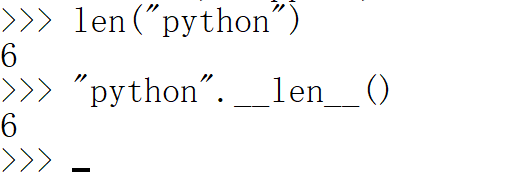
print(isinstance([1, 2, 3], (list, tuple)))  
print(isinstance((1, 2, 3), (list, tuple)))

### dir()函数

如果要获得一个对象的所有属性和方法，可以使用dir()函数，它返回一个包含字符串的list，比如，获得一个str对象的所有属性和方法：



类似\_\_xxx\_\_的属性和方法在Python中都是有特殊用途的，比如\_\_len\_\_方法返回长度。在Python中，如果你调用len()函数试图获取一个对象的长度，实际上，在len()函数内部，它自动去调用该对象的\_\_len\_\_()方法，所以，下面的代码是等价的：



我们自己写的类，如果也想用len(obj)的话，就自己写一个\_\_len\_\_()方法：

**class** MyObj(object):  
 **def** \_\_len\_\_(self):  
 **return** 10  
  
obj = MyObj()  
print(len(obj))

仅仅把属性和方法列出来是不够的，配合getattr()、setattr()以及hasattr()，我们可以直接操作一个对象的状态：

**class** MyObj(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"晓宇"  
 def** \_\_len\_\_(self):  
 **return** 10  
  
obj = MyObj()  
print(hasattr(obj,**"name"**))  
**if** hasattr(obj,**"age"**):  
 print(obj.age)  
**else**:  
 setattr(obj,**"age"**,20)  
print(getattr(obj,**"age"**,**"没有该属性"**)) //这里最好添加个默认值，否则没有该属性会报错。

也可以获得对象的方法：

**class** MyObj(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"晓宇"  
 def** \_\_len\_\_(self):  
 **return** 10  
 **def** run(self,content):  
 print(content)  
  
  
obj = MyObj()  
func = getattr(obj,**"run"**)  
func(**"获取方法！！！"**)

这样该方法就能得到调用。

## 8.6 实例属性和类属性

由于Python是动态语言，根据类创建的实例可以任意绑定属性。

给实例绑定属性的方法是通过实例变量，或者通过self变量：

例：

**class** Student(object):  
 **def** \_\_init\_\_(self,name,age,sex):  
 self.\_\_name = name  
 self.age = age  
 self.sex = sex

stu = Student(**"张晓宇"**,22,**"女"**)  
stu.score = 100

但是，如果Student类本身需要绑定一个属性呢？可以直接在class中定义属性，这种属性是类属性，归Student类所有：

**class** Student(object):  
 name = **"张三"**

**当我们定义了一个类属性后，这个属性虽然归类所有，但类的所有实例都可以访问到。**

stu = Student()  
print(stu.name)# 打印name属性，因为实例并没有name属性，所以会继续查找class的name属性  
print(Student.name)

从上面的例子可以看出，在编写程序的时候，千万不要对实例属性和类属性使用相同的名字，因为相同名称的实例属性将屏蔽掉类属性，但是当你删除实例属性后，再使用相同的名称，访问到的将是类属性。

**class** Student(object):  
 name = **"张三"  
 def** \_\_init\_\_(self):  
 self.name = **"李四"**stu = Student()  
print(Student.name)  
print(stu.name)# 由于实例属性优先级比类属性高，因此，它会屏蔽掉类的name属性  
**del** stu.name  
print(stu.name)

# 第九章 面向对象高级编程

数据封装、继承和多态只是面向对象程序设计中最基础的3个概念。在Python中，面向对象还有很多高级特性，允许我们写出非常强大的功能。

我们会讨论多重继承、定制类、元类等概念。

## 9.1 使用\_\_slots\_\_

### 简介

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**

**创建实例，给实例绑定一个属性和方法。**

**from** types **import** MethodType  
  
**class** Student(object):  
 **pass**stu = Student()  
stu.name = **"张三"  
def** set\_age(self,age):  
 self.age = age  
stu.set\_age = MethodType(set\_age,stu)  
stu.set\_age(20)  
print(stu.age)

但是，给一个实例绑定的方法，对另一个实例是不起作用的：

stuTmp = Student()  
stuTmp.set\_age(20)

这时候就会运行出现错误。

现在我们想给所有的实例都绑定方法，那么我们就可以给class绑定方法。

**class** Student(object):  
 **pass**stu = Student()  
stu.name = **"张三"  
def** set\_age(self,age):  
 self.age = age  
Student.set\_age = set\_age  
stu.set\_age(19)  
print(stu.age)  
stuTmp = Student()  
stuTmp.set\_age(20)  
print(stuTmp.age)

通常情况下，上面的set\_age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。

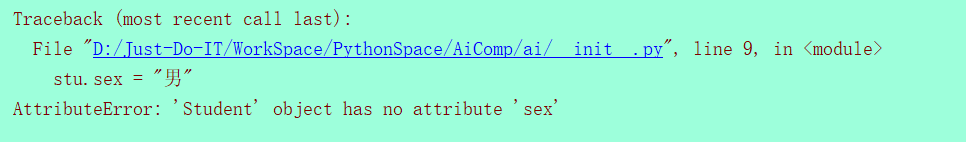
### \_\_slots\_\_的使用

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：

*#-\*-coding:utf-8-\*-***class** Student(object):  
 \_\_slots\_\_ = (**'name'**, **'age'**) *# 用tuple定义允许绑定的属性名称*

stu = Student()  
stu.name = **"Tom"**stu.age = 20  
stu.sex = **"男"**



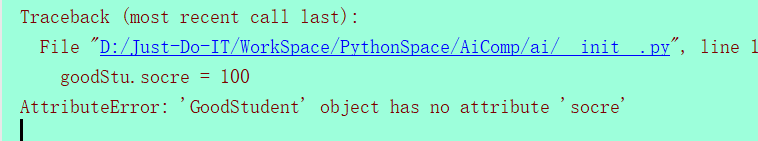
由于sex没有被放到\_\_slots\_\_中，所以不能绑定sex属性，试图绑定sex将得到AttributeError的错误。

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** GoodStudent(Student):  
 **pass**goodStu = GoodStudent()  
goodStu.name = **"张三"**goodStu.age = 20  
goodStu.sex = **"男"**

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\_\_。

**class** GoodStudent(Student):  
 \_\_slots\_\_ = (**"sex"**)  
 **pass**goodStu = GoodStudent()  
goodStu.name = **"张三"**goodStu.age = 20  
goodStu.sex = **"男"**goodStu.socre = 100



## 9.2 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，导致可以把年龄随便改：

**class** Student(object):  
 **def** \_\_init\_\_(self):  
 self.name = **None** self.age = **None** self.sex = **None**stu = Student()  
stu.name = **"张三"**stu.age = 2000  
stu.sex = **"男"**

这显然不合逻辑。为了限制age的范围，可以通过一个set\_age()方法来设置成绩，再通过一个get\_age()来获取成绩，这样，在set\_age()方法里，就可以检查参数：

**class** Student(object):  
 **def** \_\_init\_\_(self):  
 self.name = **None** self.age = **None** self.sex = **None  
 def** set\_age(self,age):  
 **if not** isinstance(age,int):  
 **raise** ValueError(**"年龄是一个int类型的值"**)  
 **if** age < 0 **or** age > 120:  
 **raise** ValueError(**"年龄在0 - 120之间"**)  
 self.age = age  
   
stu = Student()  
stu.name = **"张三"**stu.age = 2000  
stu.sex = **"男"**

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。

@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@age.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

**class** Student(object):  
 @property  
 **def** age(self):  
 **return** self.\_age  
 @age.setter  
 **def** age(self,value):  
 **if not** isinstance(value,int):  
 **raise** ValueError(**"年龄是一个int类型的值"**)  
 **if** value < 0 **or** value > 120:  
 **raise** ValueError(**"年龄在0 - 120之间"**)  
 self.\_age = value  
  
stu = Student()  
stu.age = 20  
print(stu.age)

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：

*#-\*-coding:utf-8-\*-***class** Student(object):  
 @property  
 **def** age(self):  
 **return** 2012 - self.\_birth  
 @property  
 **def** birth(self):  
 **return** self.\_birth  
 @birth.setter  
 **def** birth(self,value):  
 self.\_birth = value  
stu = Student()  
stu.birth = 2000  
print(stu.age)

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。

## 9.3 多继承

继承是面向对象编程的一个重要的方式，因为通过继承，子类就可以扩展父类的功能。

回忆一下Car类的设计，假设我们现在有一下四种汽车：

Benz-奔驰

Jaguar-捷豹

Bmw-宝马

Audi-奥迪

如果按照哺乳动物和鸟类归类，我们可以设计出这样的类的层次：

图片略。。。。

## 9.4 定制类

看到类似\_\_slots\_\_这种形如\_\_xxx\_\_的变量或者函数名就要注意，这些在Python中是有特殊用途的。

\_\_slots\_\_我们已经知道怎么用了，\_\_len\_\_()方法我们也知道是为了能让class作用于len()函数。

除此之外，Python的class中还有许多这样有特殊用途的函数，可以帮助我们定制类。

### \_\_str\_\_

我们先定义一个Student类，打印一个实例：

**class** Student(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"张三"**stu = Student()  
print(stu)



打印出了一堆东西，这个不用多解释。

怎样打印出我们想要的东西，只需要定义\_\_str\_\_()方法，返回一个漂亮的字符串就可以了。

**class** Student(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"张三"  
 def** \_\_str\_\_(self):  
 **return "Student object name (%s) "** %self.name  
stu = Student()  
print(stu)

这是因为直接显示变量调用的不是\_\_str\_\_()，而是\_\_repr\_\_()，两者的区别是\_\_str\_\_()返回用户看到的字符串，而\_\_repr\_\_()返回程序开发者看到的字符串，也就是说，\_\_repr\_\_()是为调试服务的。

解决办法是再定义一个\_\_repr\_\_()。但是通常\_\_str\_\_()和\_\_repr\_\_()代码都是一样的，所以，有个偷懒的写法：

**class** Student(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"张三"  
 def** \_\_str\_\_(self):  
 **return "Student object name (%s) "** %self.name  
 \_\_repr\_\_ = \_\_str\_\_  
stu = Student()  
print(stu.\_\_repr\_\_)

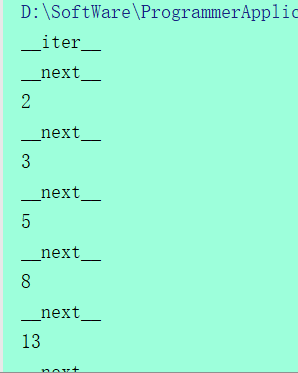
### \_\_iter\_\_

如果一个类想被用于for ... in循环，类似list或tuple那样，就必须实现一个\_\_iter\_\_()方法，该方法返回一个迭代对象，然后，Python的for循环就会不断调用该迭代对象的\_\_next\_\_()方法拿到循环的下一个值，直到遇到StopIteration错误时退出循环。

我们以斐波那契数列为例，写一个Fib类，可以作用于for循环：

**class** Fib(object):  
 **def** \_\_init\_\_(self):  
 self.a,self.b = 1,2  
 **def** \_\_iter\_\_(self):  
 print(**"\_\_iter\_\_"**)  
 **return** self  
 **def** \_\_next\_\_(self):  
 print(**"\_\_next\_\_"**)  
 self.a,self.b = self.b,self.a + self.b  
 **if** self.a > 100:  
 **raise** StopIteration()  
 **return** self.a  
**for** n **in** Fib():  
 print(n)

运行结果如下：



### \_\_getitem\_\_

Fib实例虽然能作用于for循环，看起来和list有点像，但是，把它当成list来使用还是不行，比如，取第5个元素：

print(Fib()[100])

要表现得像list那样按照下标取出元素，需要实现\_\_getitem\_\_()方法：

**class** Fib(object):  
 **def** \_\_getitem\_\_(self, item):  
 a,b = 1,1  
 **for** x **in** range(item):  
 a,b = b,a+b  
 **return** a  
  
print(Fib()[2])  
print(Fib()[10])

但是list有个神奇的切片方法：

对于Fib却报错。原因是\_\_getitem\_\_()传入的参数可能是一个int，也可能是一个切片对象slice，所以要做判断：

**class** Fib(object):  
 **def** \_\_getitem\_\_(self, n):  
 **if** isinstance(n, int): *# n是索引* a, b = 1, 1  
 **for** x **in** range(n):  
 a, b = b, a + b  
 **return** a  
 **if** isinstance(n, slice): *# n是切片* start = n.start  
 stop = n.stop  
 **if** start **is None**:  
 start = 0  
 a, b = 1, 1  
 L = []  
 **for** x **in** range(stop):  
 **if** x >= start:  
 L.append(a)  
 a, b = b, a + b  
 **return** L

但是没有对step参数作处理：

也没有对负数作处理，所以，要正确实现一个\_\_getitem\_\_()还是有很多工作要做的。

此外，如果把对象看成dict，\_\_getitem\_\_()的参数也可能是一个可以作key的object，例如str。

与之对应的是\_\_setitem\_\_()方法，把对象视作list或dict来对集合赋值。最后，还有一个\_\_delitem\_\_()方法，用于删除某个元素。

总之，通过上面的方法，我们自己定义的类表现得和Python自带的list、tuple、dict没什么区别，这完全归功于动态语言的“鸭子类型”，不需要强制继承某个接口。

### \_\_getattr\_\_

正常情况下，当我们调用类的方法或属性时，如果不存在，就会报错。比如定义Student类：

**class** Student(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"张三"**stu = Student()  
print(stu.name)  
print(stu.score)

调用name属性，没问题，但是，调用不存在的score属性，就有问题了：

错误信息很清楚地告诉我们，没有找到score这个attribute。

要避免这个错误，除了可以加上一个score属性外，Python还有另一个机制，那就是写一个\_\_getattr\_\_()方法，动态返回一个属性。修改如下：

**class** Student(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"张三"  
 def** \_\_getattr\_\_(self, item):  
 **if** item == **"score"**:  
 **return** 100  
  
stu = Student()  
print(stu.name)  
print(stu.score)

这里也可以返回函数，只是调用方式有点变化。

**class** Student(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"张三"  
 def** \_\_getattr\_\_(self, item):  
 **if** item == **"score"**:  
 **return lambda** : 100  
  
stu = Student()  
print(stu.name)  
print(stu.score())

注意：

只有在没有找到属性的情况下，才调用\_\_getattr\_\_，已有的属性，比如name，不会在\_\_getattr\_\_中查找。

此外，注意到任意调用如s.abc都会返回None，这是因为我们定义的\_\_getattr\_\_默认返回就是None。要让class只响应特定的几个属性，我们就要按照约定，抛出AttributeError的错误：

**class** Student(object):  
 **def** \_\_init\_\_(self):  
 self.name = **"张三"  
 def** \_\_getattr\_\_(self, item):  
 **if** item == **"score"**:  
 **return lambda** : 100  
 **raise** AttributeError(**"Student has not %s attr"**%item)  
stu = Student()  
print(stu.name)  
print(stu.age())

这实际上可以把一个类的所有属性和方法调用全部动态化处理了，不需要任何特殊手段。

这种完全动态调用的特性有什么实际作用呢？作用就是，可以针对完全动态的情况作调用。

举个例子：

现在很多网站都搞REST API，比如新浪微博、豆瓣啥的，调用API的URL类似：

<http://api.server/user/friends>

<http://api.server/user/timeline/list>

如果要写SDK，给每个URL对应的API都写一个方法，那得累死，而且，API一旦改动，SDK也要改。

利用完全动态的\_\_getattr\_\_，我们可以写出一个链式调用：

**class** Chain(object):  
  
 **def** \_\_init\_\_(self, path=**''**):  
 self.\_path = path  
  
 **def** \_\_getattr\_\_(self, path):  
 **return** Chain(**'%s/%s'** % (self.\_path, path))  
  
 **def** \_\_str\_\_(self):  
 **return** self.\_path  
  
 \_\_repr\_\_ = \_\_str\_\_  
  
print(Chain().status.user.timeline.firends)

这样，无论API怎么变，SDK都可以根据URL实现完全动态的调用，而且，不随API的增加而改变！

### \_\_call\_\_

一个对象实例可以有自己的属性和方法，当我们调用实例方法时，我们用instance.method()来调用。能不能直接在实例本身上调用呢？在Python中，答案是肯定的。

任何类，只需要定义一个\_\_call\_\_()方法，就可以直接对实例进行调用。请看示例：

**class** Student(object):  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
  
 **def** \_\_call\_\_(self):  
 print(**'My name is %s .'** % self.name)  
  
s = Student(**"张三"**)  
s()*# self参数不要传入*

\_\_call\_\_()还可以定义参数。对实例进行直接调用就好比对一个函数进行调用一样，所以你完全可以把对象看成函数，把函数看成对象，因为这两者之间本来就没啥根本的区别。

如果你把对象看成函数，那么函数本身其实也可以在运行期动态创建出来，因为类的实例都是运行期创建出来的，这么一来，我们就模糊了对象和函数的界限。

那么，怎么判断一个变量是对象还是函数呢？其实，更多的时候，我们需要判断一个对象是否能被调用，能被调用的对象就是一个Callable对象，比如函数和我们上面定义的带有\_\_call\_\_()的类实例：

print(callable(Student()))  
print(callable([1, 2, 3]))

通过callable()函数，我们就可以判断一个对象是否是“可调用”对象。

## 9.5 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，例如一周几天、月份：

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'**))  
Date = Enum(**"Date"**,(**"Monday"**,**"Tuesday"**,**"Wednesday"**,**"Thursday"**,**"Friday"**,**"Saturday"**,**"Sunday"**))

这样我们就获得了Month、Date类型的枚举类，可以直接使用Month.Jan、Date.Monday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：

print(Weekday.Fri)  
print(Weekday[**'Fri'**])  
print(Weekday.Fri.value)  
print(Weekday(5))  
**for** name, member **in** Weekday.\_\_members\_\_.items():  
 print(name, **'=>'**, member)

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。

## 9.6 元类

### type()

动态语言和静态语言最大的不同，就是函数和类的定义，不是编译时定义的，而是运行时动态创建的。

比方说我们要定义一个Hello的class，就写一个hello.py模块：

**class** Hello(object):  
 **def** hello(self, name=**'world'**):  
 print(**'Hello, %s.'** % name)

当Python解释器载入hello模块时，就会依次执行该模块的所有语句，执行结果就是动态创建出一个Hello的class对象

print(type(Hello))  
print(type(Hello()))

<class 'type'>

<class '\_\_main\_\_.Hello'>

type()函数可以查看一个类型或变量的类型，Hello是一个class，它的类型就是type，而h是一个实例，它的类型就是class Hello。

我们说class的定义是运行时动态创建的，而创建class的方法就是使用type()函数。

type()函数既可以返回一个对象的类型，又可以创建出新的类型，比如，我们可以通过type()函数创建出Hello类，而无需通过class Hello(object)...的定义。

要创建一个class对象，type()函数依次传入3个参数：

class的名称；

继承的父类集合，注意Python支持多重继承，如果只有一个父类，别忘了tuple的单元素写法；

class的方法名称与函数绑定，这里我们把函数fn绑定到方法名hello上。

通过type()函数创建的类和直接写class是完全一样的，因为Python解释器遇到class定义时，仅仅是扫描一下class定义的语法，然后调用type()函数创建出class。

正常情况下，我们都用class Xxx...来定义类，但是，type()函数也允许我们动态创建出类来，也就是说，动态语言本身支持运行期动态创建类，这和静态语言有非常大的不同，要在静态语言运行期创建类，必须构造源代码字符串再调用编译器，或者借助一些工具生成字节码实现，本质上都是动态编译，会非常复杂。

### metaclass

除了使用type()动态创建类以外，要控制类的创建行为，还可以使用metaclass。

metaclass，直译为元类，简单的解释就是：

当我们定义了类以后，就可以根据这个类创建出实例，所以：先定义类，然后创建实例。

但是如果我们想创建出类呢？那就必须根据metaclass创建出类，所以：先定义metaclass，然后创建类。

连接起来就是：先定义metaclass，就可以创建类，最后创建实例。

所以，metaclass允许你创建类或者修改类。换句话说，你可以把类看成是metaclass创建出来的“实例”。

metaclass是Python面向对象里最难理解，也是最难使用的魔术代码。正常情况下，你不会碰到需要使用metaclass的情况，所以，以下内容看不懂也没关系，因为基本上你不会用到。

我们先看一个简单的例子，这个metaclass可以给我们自定义的MyList增加一个add方法：

定义ListMetaclass，按照默认习惯，metaclass的类名总是以Metaclass结尾，以便清楚地表示这是一个metaclass：

*# metaclass是类的模板，所以必须从`type`类型派生：***class** ListMetaclass(type):  
 **def** \_\_new\_\_(cls, name, bases, attrs):  
 attrs[**'add'**] = **lambda** self, value: self.append(value)  
 **return** type.\_\_new\_\_(cls, name, bases, attrs)

有了ListMetaclass，我们在定义类的时候还要指示使用ListMetaclass来定制类，传入关键字参数metaclass：

**class** MyList(list, metaclass=ListMetaclass):  
 **pass**

当我们传入关键字参数metaclass时，魔术就生效了，它指示Python解释器在创建MyList时，要通过ListMetaclass.\_\_new\_\_()来创建，在此，我们可以修改类的定义，比如，加上新的方法，然后，返回修改后的定义。

\_\_new\_\_()方法接收到的参数依次是：

1、当前准备创建的类的对象。

2、类的名字。

3、类继承的父类集合。

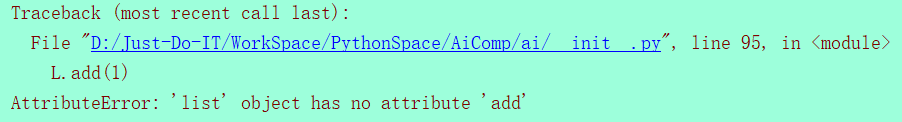
4、类的方法集合。

测试一下MyList是否可以调用add()方法：

*# metaclass是类的模板，所以必须从`type`类型派生：***class** ListMetaclass(type):  
 **def** \_\_new\_\_(cls, name, bases, attrs):  
 attrs[**'add'**] = **lambda** self, value: self.append(value)  
 **return** type.\_\_new\_\_(cls, name, bases, attrs)  
   
**class** MyList(list, metaclass=ListMetaclass):  
 **pass**L = MyList()  
L.add(1)  
print(L)

而普通的list没有add()方法：

L = list()  
L.add(1)  
print(L)



动态修改有什么意义？直接在MyList定义中写上add()方法不是更简单吗？正常情况下，确实应该直接写，通过metaclass修改纯属变态。

但是，总会遇到需要通过metaclass修改类定义的。ORM就是一个典型的例子。

ORM全称“Object Relational Mapping”，即对象-关系映射，就是把关系数据库的一行映射为一个对象，也就是一个类对应一个表，这样，写代码更简单，不用直接操作SQL语句。

要编写一个ORM框架，所有的类都只能动态定义，因为只有使用者才能根据表的结构定义出对应的类来。

让我们来尝试编写一个ORM框架。

编写底层模块的第一步，就是先把调用接口写出来。比如，使用者如果使用这个ORM框架，想定义一个User类来操作对应的数据库表User，我们期待他写出这样的代码：

**class** User(Model):  
 *# 定义类的属性到列的映射：* id = IntegerField(**'id'**)  
 name = StringField(**'username'**)  
 email = StringField(**'email'**)  
 password = StringField(**'password'**)  
  
*# 创建一个实例：*u = User(id=12345, name=**'Michael'**, email=**'test@orm.org'**, password=**'my-pwd'**)  
*# 保存到数据库：*u.save()

其中，父类Model和属性类型StringField、IntegerField是由ORM框架提供的，剩下的魔术方法比如save()全部由metaclass自动完成。虽然metaclass的编写会比较复杂，但ORM的使用者用起来却异常简单。

现在，我们就按上面的接口来实现该ORM。

首先来定义Field类，它负责保存数据库表的字段名和字段类型：

**class** Field(object):  
 **def** \_\_init\_\_(self, name, column\_type):  
 self.name = name  
 self.column\_type = column\_type  
  
 **def** \_\_str\_\_(self):  
 **return '<%s:%s>'** % (self.\_\_class\_\_.\_\_name\_\_, self.name)

在Field的基础上，进一步定义各种类型的Field，比如StringField，IntegerField等等：

**class** StringField(Field):  
  
 **def** \_\_init\_\_(self, name):  
 super(StringField, self).\_\_init\_\_(name, **'varchar(100)'**)

下一步，就是编写最复杂的ModelMetaclass了：

**class** ModelMetaclass(type):  
  
 **def** \_\_new\_\_(cls, name, bases, attrs):  
 **if** name==**'Model'**:  
 **return** type.\_\_new\_\_(cls, name, bases, attrs)  
 print(**'Found model: %s'** % name)  
 mappings = dict()  
 **for** k, v **in** attrs.items():  
 **if** isinstance(v, Field):  
 print(**'Found mapping: %s ==> %s'** % (k, v))  
 mappings[k] = v  
 **for** k **in** mappings.keys():  
 attrs.pop(k)  
 attrs[**'\_\_mappings\_\_'**] = mappings *# 保存属性和列的映射关系* attrs[**'\_\_table\_\_'**] = name *# 假设表名和类名一致* **return** type.\_\_new\_\_(cls, name, bases, attrs)

以及基类Model：

**class** Model(dict, metaclass=ModelMetaclass):  
  
 **def** \_\_init\_\_(self, \*\*kw):  
 super(Model, self).\_\_init\_\_(\*\*kw)  
  
 **def** \_\_getattr\_\_(self, key):  
 **try**:  
 **return** self[key]  
 **except** KeyError:  
 **raise** AttributeError(**r"'Model' object has no attribute '%s'"** % key)  
  
 **def** \_\_setattr\_\_(self, key, value):  
 self[key] = value  
  
 **def** save(self):  
 fields = []  
 params = []  
 args = []  
 **for** k, v **in** self.\_\_mappings\_\_.items():  
 fields.append(v.name)  
 params.append(**'?'**)  
 args.append(getattr(self, k, **None**))  
 sql = **'insert into %s (%s) values (%s)'** % (self.\_\_table\_\_, **','**.join(fields), **','**.join(params))  
 print(**'SQL: %s'** % sql)  
 print(**'ARGS: %s'** % str(args))

当用户定义一个class User(Model)时，Python解释器首先在当前类User的定义中查找metaclass，如果没有找到，就继续在父类Model中查找metaclass，找到了，就使用Model中定义的metaclass的ModelMetaclass来创建User类，也就是说，metaclass可以隐式地继承到子类，但子类自己却感觉不到。

在ModelMetaclass中，一共做了几件事情：

排除掉对Model类的修改；

在当前类（比如User）中查找定义的类的所有属性，如果找到一个Field属性，就把它保存到一个\_\_mappings\_\_的dict中，同时从类属性中删除该Field属性，否则，容易造成运行时错误（实例的属性会遮盖类的同名属性）；

把表名保存到\_\_table\_\_中，这里简化为表名默认为类名。

在Model类中，就可以定义各种操作数据库的方法，比如save()，delete()，find()，update等等。

我们实现了save()方法，把一个实例保存到数据库中。因为有表名，属性到字段的映射和属性值的集合，就可以构造出INSERT语句。

编写代码试试：

*# 创建一个实例：*u = User(id=12345, name=**'Michael'**, email=**'test@orm.org'**, password=**'my-pwd'**)  
*# 保存到数据库：*u.save()

可以看到，save()方法已经打印出了可执行的SQL语句，以及参数列表，只需要真正连接到数据库，执行该SQL语句，就可以完成真正的功能。

不到100行代码，我们就通过metaclass实现了一个精简的ORM框架，是不是非常简单？

# 文件IO

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

比如你打开浏览器，访问新浪首页，浏览器这个程序就需要通过网络IO获取新浪的网页。浏览器首先会发送数据给新浪服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后新浪服务器把网页发过来，这个动作是从外面接收数据，叫Input。所以，通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。

IO编程中，Stream（流）是一个很重要的概念，可以把流想象成一个水管，数据就是水管里的水，但是只能单向流动。Input Stream就是数据从外面（磁盘、网络）流进内存，Output Stream就是数据从内存流到外面去。对于浏览网页来说，浏览器和新浪服务器之间至少需要建立两根水管，才可以既能发数据，又能收数据。

由于CPU和内存的速度远远高于外设的速度，所以，在IO编程中，就存在速度严重不匹配的问题。举个例子来说，比如要把100M的数据写入磁盘，CPU输出100M的数据只需要0.01秒，可是磁盘要接收这100M数据可能需要10秒，怎么办呢？有两种办法：

第一种是CPU等着，也就是程序暂停执行后续代码，等100M的数据在10秒后写入磁盘，再接着往下执行，这种模式称为同步IO；

另一种方法是CPU不等待，只是告诉磁盘，“您老慢慢写，不着急，我接着干别的事去了”，于是，后续代码可以立刻接着执行，这种模式称为异步IO。

同步和异步的区别就在于是否等待IO执行的结果。好比你去麦当劳点餐，你说“来个汉堡”，服务员告诉你，对不起，汉堡要现做，需要等5分钟，于是你站在收银台前面等了5分钟，拿到汉堡再去逛商场，这是同步IO。

你说“来个汉堡”，服务员告诉你，汉堡需要等5分钟，你可以先去逛商场，等做好了，我们再通知你，这样你可以立刻去干别的事情（逛商场），这是异步IO。

很明显，使用异步IO来编写程序性能会远远高于同步IO，但是异步IO的缺点是编程模型复杂。想想看，你得知道什么时候通知你“汉堡做好了”，而通知你的方法也各不相同。如果是服务员跑过来找到你，这是回调模式，如果服务员发短信通知你，你就得不停地检查手机，这是轮询模式。总之，异步IO的复杂度远远高于同步IO。

操作IO的能力都是由操作系统提供的，每一种编程语言都会把操作系统提供的低级C接口封装起来方便使用，Python也不例外。我们后面会详细讨论Python的IO编程接口。

注意，本章的IO编程都是同步模式，异步IO由于复杂度太高，后续涉及到服务器端程序开发时我们再讨论。

## 10.1文件读写

Python内置了读写文件的函数，用法和C语言是兼容的。

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）。

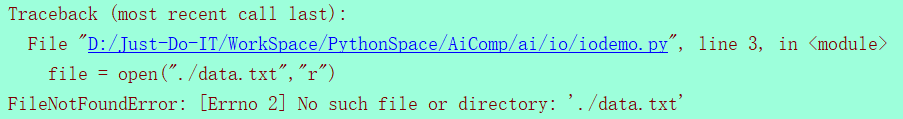
**读文件：**

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：

file = open(**"./data.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：



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：

file = open(**"./data.txt"**,**"r"**)  
str = file.read()  
print(str)

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：

file.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**:  
 file = open(**"./data.txt"**,**"r"**)  
 str = file.read()  
 print(str)  
**finally**:  
 **if** file:  
 file.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(**'./data.txt'**, **'r'**) **as** file:  
 print(file.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：

file = open(**"./data.txt"**,**"r"**)  
**for** line **in** file.readlines():  
 print(line.strip())

**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。

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：

file = open(**"1.png"**,**"rb"**)  
print(file.read())*# 十六进制表示的字节*

**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：

file = open(**"data.txt"**,**"r"**,encoding=**"gbk"**)  
print(file.read())

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：

file = open(**"data.txt"**,**"r"**,encoding=**"gbk"**,errors=**"ignore"**)  
print(file.read())

**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：

file = open(**"data.txt"**,**"w"**)  
file.write(**"Tensorflow"**)  
file.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(**"data.txt"**,**"a"**) **as** file:  
 file.write(**"tensorflow"**)

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。

细心的童鞋会发现，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。

**open mode**

|  |  |
| --- | --- |
| w | 以写方式打开文件，可向文件写入信息。如文件存在，则清空该文件，再写入新内容 |
| a | 以追加模式打开文件（即一打开文件，文件指针自动移到文件末尾），如果文件不存在则创建 |
| r+ | 以读写方式打开文件，可对文件进行读和写操作。 |
| w+ | 消除文件内容，然后以读写方式打开文件。 |
| a+ | 以读写方式打开文件，并把文件指针移到文件尾。 |
| b | 以二进制模式打开文件，而不是以文本模式。该模式只对Windows或Dos有效，类Unix的文件是用二进制模式进行操作的。 |

**操作文件对象方法**

|  |  |
| --- | --- |
| f.close() | 关闭文件，记住用open()打开文件后一定要记得关闭它，否则会占用系统的可打开文件句柄数。 |
| f.fileno() | 获得文件描述符，是一个数字 |
| f.flush() | 刷新输出缓存 |
| f.isatty() | 如果文件是一个交互终端，则返回True，否则返回False。 |
| f.read([count]) | 读出文件，如果有count，则读出count个字节。 |
| f.readline() | 读出一行信息。 |
| f.readlines() | 读出所有行，也就是读出整个文件的信息。 |
| f.seek(offset[,where]) | 把文件指针移动到相对于where的offset位置。where为0表示文件开始处，这是默认值 ；1表示当前位置；2表示文件结尾。 |
| f.tell() | 获得文件指针位置。 |
| f.truncate([size]) | 截取文件，使文件的大小为size。 |
| f.write(string) | 把string字符串写入文件。 |
| f.writelines(list) | 把list中的字符串一行一行地写入文件，是连续写入文件，没有换行。 |

## 10.2 StringIO和BytesIO

#### 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  
  
file = StringIO(**"Hello \n Python"**)  
**for** line **in** file.readlines():  
 print(line.strip())

**from** io **import** StringIO  
  
str = StringIO()  
str.write(**"Hello\n"**)  
str.write(**"Python"**)  
str.flush()  
print(str.getvalue())  
str.close()

#### 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  
  
bt = BytesIO()  
bt.write(**"大家好python"**.encode(**"utf-8"**))  
bt.flush()  
print(bt.getvalue())  
bt.close()

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  
  
bt = BytesIO(**b"\xe5\xa4\xa7\xe5\xae\xb6\xe5\xa5\xbdpython"**)  
print(bt.read().decode(**"utf-8"**))

## 10.3 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  
  
print(os.name)*#操作系统类型*

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()

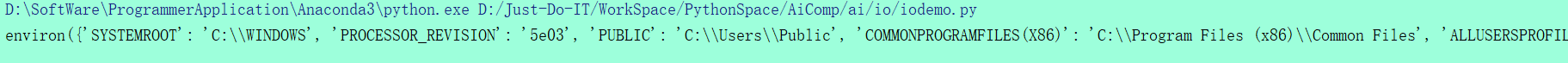
注意：

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。

**环境变量**

print(os.environ)

运行结果太多不再copy！！！！



要获取某个环境变量的值：

print(os.environ.get(**"PATH"**))

### 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：

**import** os  
  
*# 查看当前目录的绝对路径*print(os.path.abspath(**'./'**))  
*# 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*print(os.path.join(**"D:\Just-Do-IT"**,**"python"**))  
*#创建一个目录*os.mkdir(**"D:/Just-Do-IT/python"**)  
*# 删掉一个目录*os.rmdir(**"D:/Just-Do-IT/python"**)

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：

print(os.path.split(**"D:/Just-Do-IT/python"**))

os.path.splitext()可以直接让你得到文件扩展名，很多时候非常方便：

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data.txt文件：

重命名与删除文件

os.rename(**"data.txt"**,**"data.py"**)

os.remove(**"data.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：

l = [x **for** x **in** os.listdir(**'d:/'**) **if** os.path.isdir(x)]  
print(l)

[x **for** x **in** os.listdir(**'.'**) **if** os.path.isfile(x) **and** os.path.splitext(x)[1] ==**'.py'**]

## 10.4 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=**"Tom"**,age=20,sex=**"男"**)

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'Tom'。

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，在其他语言中也被称之为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=**"Tom"**,age=20,sex=**"男"**)  
print(pickle.dumps(d))

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：

**import** pickle  
  
d = dict(name=**"Tom"**,age=20,sex=**"男"**)  
file = open(**"dump.txt"**,**"wb"**)  
pickle.dump(d,file)  
file.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：

file = open(**"dump.txt"**,**"rb"**)  
d = pickle.load(file)  
file.close()  
print(d)

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。

### 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/flaot |
| 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=**"Tom"**,age=20,sex=**"男"**)  
print(json.dumps(d))

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：

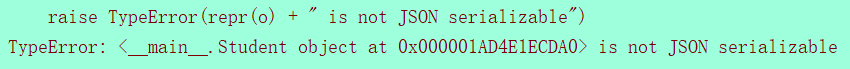
str = **'{"age": 20, "sex": "男", "name": "Tom"}'**print(json.loads(str))

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。

### 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(**'Tom'**, 20, 88)  
print(json.dumps(s))



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。

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：

**class** Student(object):  
 **def** \_\_init\_\_(self, name, age, score):  
 self.name = name  
 self.age = age  
 self.score = score  
  
**def** student2dict(std):  
 **return** {  
 **'name'**: std.name,  
 **'age'**: std.age,  
 **'score'**: std.score  
 }  
  
s = Student(**'Tom'**, 20, 88)  
print(json.dumps(s,default=student2dict))

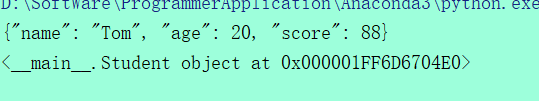
不过，下次如果遇到一个Teacher类的实例，照样无法序列化为JSON。我们可以偷个懒，把任意class的实例变为dict：

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'**])  
json\_str = **'{"age": 20, "score": 88, "name": "Bob"}'**print(json.loads(json\_str, object\_hook=dict2student))



# 线程与进程

单线程的程序往往功能非常有限，例如，我们开发一个简单的服务器程序，这个服务器程序需要向不同的客户端提供服务，不同的客户端之间应该互不干扰。对于多线程可以这样理解：单线程的程序只有一个顺序执行流，多线程的程序则可以包括多个顺序执行流，多个顺序流之间互不干扰。同样也可以这样理解：单线程的程序如同只雇佣一个服务员的餐厅，他必须做完一件事情后，才可以做下一件事情；多线程类似于雇佣多个服务员的餐厅，他们可以同时做多个事情。

## 11.1 线程概述

目前我们所熟知的操作系统Windows、Unix、Linux系统都支持同时运行多个任务，一个任务通常就是一个程序，每个运行中的程序就是一个进程。当一个程序运行时，内部可能包含了多个顺序执行流，每个顺序执行流就是一个线程。

#### 11.1.1 线程与进程

几乎所有的操作系统都支持进程的概念，所有运行中的任务通常对应一个进程。当一个程序进入内存时就变成了一个进程。进程就是处于运行过程中的程序，并且具有一定的独立能力，进程是系统进行资源分配和调度的一个独立单位。

进程包含如下3个特征：

独立性：进程是系统中独立存在的实体，它可以拥有自己独立的资源，每一个进程都拥有自己私有的地址空间。在没有经过进程本身允许的情况下，一个用户进程不可以直接访问其他进程的地址空间。

动态性：进程与程序的区别在于，程序只是一个静态的指令集合，而进程是一个正在系统中活动的指令集合。在进程中加入了时间的概念。进程具有自己的生命周期和各种不同的状态，这些概念在程序中都是不具备的。

并发性：多个进程可以在单个处理器上并发执行，多个进程之间不会互相影响。

大部分操作系统都支持多进程并发运行，现代的操作系统几乎都支持同时运行多个任务。例如：我们可以一边开着开发工具写程序，一边开着参考手册备查，同时还使用播放工具听着音乐……除此之外，每台电脑运行时还有大量底层的支撑性程序在运行……这些进程看上去像是在同时工作。

但事实的真相是，对于一个CPU而言，它在某个时间点只能执行一个程序，也就是说，只能运行一个进程，CPU不断地在这些进程之间轮换执行。那么为什么我们感觉不到任何中断现象呢？这是因为CPU的执行速度相对我们的感觉来说实在是太快了，所以虽然CPU在多个进程间轮换执行，但是我们感觉到好像有多个进程在同时进行。

多线程则扩展了多进程的概念，使得一个进程可以同时并发处理多个任务。线程（Thread）也被称作轻量级进程，线程是进程的执行单元。类似于进程在操作系统中的地位一样，线程在程序中是独立的、并发的执行流。当进程被初始化后，主线程就被创建了。对于绝大多数的应用程序来说，通常仅要求有一个主线程，但也可以在该进程内创建多条顺序执行流，这些顺序执行流就是线程，每个线程也是相互独立的。

线程是进程的组成部分，一个进程可以拥有多个线程，一个线程必须有一个父进程。线程可以拥有自己的堆栈、自己的程序计数器和自己的局部变量，但不拥有系统资源，它与父进程的其他线程共享该进程所有拥有的全部资源。因为多个线程共享父进程里的全部资源，因此编程更加方便，但必须要注意的一点是：确保线程不会妨碍同一进程里的其他线程。

线程可以完成一定的任务，可以与其他线程共享父进程中的共享变量及部分环境，相互之间协同来完成进程所要完成的任务。

线程是独立运行的，它并不知道进程中是否存在其他的线程存在。线程的执行是抢占式的，也就是当前运行的线程在任何时候都有可能被挂起，以便另外一个线程可以运行。

一个线程可以创建和撤销另一个线程，同一个进程中的多个线程之间可以并发执行。

从逻辑角度来看，多线程存在于一个应用程序中，让一个应用程序中可以有多个执行部分同时执行，但操作系统无需将多个线程看成多个独立的应用，对多线程实现调度和管理以及资源分配。因为线程的调度和管理是由进程本身负责完成。

简而言之，一个程序运行后至少有一个进程，一个进程可以包含多个线程，但至少要包含一个线程。

归纳如下：操作系统可以同时执行多个任务，每个任务就是进程；进程可以同时执行多个任务，每个任务就是线程。

Python既支持多进程，又支持多线程，我们会讨论如何编写这两种多任务程序。

## 11.2 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：

*#-\*-coding:utf-8-\*-***import** os  
  
print(**"Process %s starting "**%os.getpid())  
*#这个只有在Linux系统上有*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))

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。

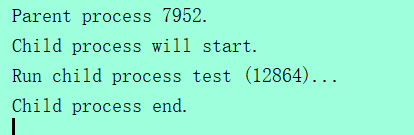
#### Multiprocessing

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：

*#-\*-coding:utf-8-\*-***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.'**)



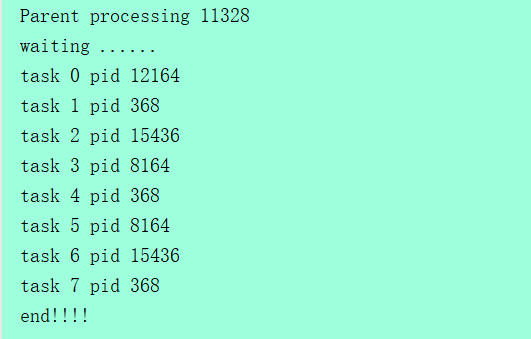
创建子进程时，只需要传入一个执行函数和函数的参数，创建一个Process实例，用start()方法启动，这样创建进程比fork()还要简单。

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。

#### 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(**"task %s pid %s"**%(name,os.getpid()))  
 start = time.time()  
 time.sleep(random.random() \* 3)  
 end = time.time()  
 print(**"task run time is %s"**%(name,(end - start)))  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 print(**"Parent processing %s"**%os.getpid())  
 p = Pool(4)  
 **for** i **in** range(8):  
 p.apply\_async(long\_time\_task,args=(i,))  
 print(**"waiting ......"**)  
 p.close()  
 p.join()  
 print(**"end!!!!"**)



代码详解：

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8，因此，最多同时执行8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(8)

就可以同时跑8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。

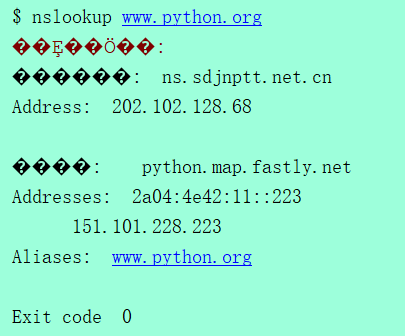
#### 3、子进程

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)



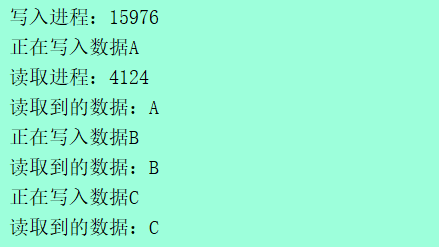
如果子进程还需要输入，则可以通过communicate()方法输入：

#### 4、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(queue):  
 print(**"写入进程：%s"**%os.getpid())  
 **for** v **in** [**'A'**,**'B'**,**'C'**]:  
 print(**"正在写入数据"**+v)  
 queue.put(v)  
 time.sleep(random.random())  
  
*#读数据进程执行的代码***def** read(queue):  
 print(**"读取进程：%s"**%os.getpid())  
 **while True**:  
 v = queue.get(**True**)  
 print(**"读取到的数据："**+v)  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 queue = Queue()  
 pw = Process(target=write,args=(queue,))  
 pr = Process(target=read,args=(queue,))  
 *#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()



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。

## 11.3多线程

#### 1、简介

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。

我们前面提到了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：

例：

**import** threading,time,random  
  
**def** pr\_data(str):  
 **while True**:  
 **for** s **in** str:  
 time.sleep(random.random()\*0.2)  
 print(s,end=**""**)  
  
 print()  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 th1 = threading.Thread(target=pr\_data,args=(**"AAAAAAAAA"**,))  
 th2 = threading.Thread(target=pr\_data,args=(**"BBBBBBBBB"**,))  
 th1.start()  
 th2.start()

例：

**import** threading,time  
  
**def** music(args):  
 **for** i **in** range(2):   
 print(threading.current\_thread().name+**"正在听音乐"**+args)  
 time.sleep(1)  
  
**def** game(args):  
 **for** i **in** range(2):  
 print(threading.current\_thread().name+**"正在玩游戏"**+args)  
 time.sleep(1)  
  
  
th1 = threading.Thread(target=music,args=(**"简单爱"**,))  
th2 = threading.Thread(target=game,args=(**"英雄联盟"**,))  
ths = []  
ths.append(th1)  
ths.append(th2)  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 **for** th **in** ths:  
 th.start()

#### 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（即线程安全），把内容给改乱了。

**import** threading,time,random  
  
lock = threading.Lock()  
  
**def** pr\_data(str):  
 lock.acquire()  
 *# 先要获取锁* **try**:  
 **for** s **in** str:  
 time.sleep(random.random() \* 0.2)  
 print(s, end=**""**)  
 print()  
 **finally**:  
 *# 改完了一定要释放锁* lock.release()  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 th1 = threading.Thread(target=pr\_data,args=(**"AAAAAAAAA"**,))  
 th2 = threading.Thread(target=pr\_data,args=(**"BBBBBBBBB"**,))  
 th1.start()  
 th2.start()

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。

**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。

## 11.4 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。

## 11.5 进程 vs. 线程

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。

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。

### 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。

### 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。

## 11.6 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：

*#-\*-coding:utf-8-\*-***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=(**''**, 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。

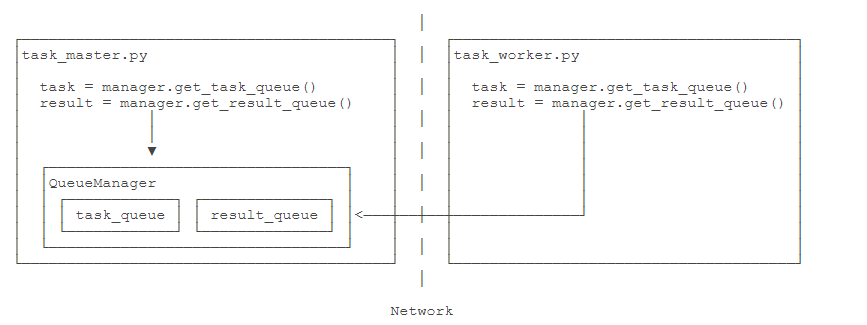
然后，在另一台机器上启动任务进程（本机上启动也可以）：

**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.'**)

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。

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。

# 标准库

## 12.1 datetime

datetime是Python处理日期和时间的标准库。

#### 获取当前日期和时间：

**from** datetime **import** datetime  
  
date = datetime.now()  
print(date)

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。

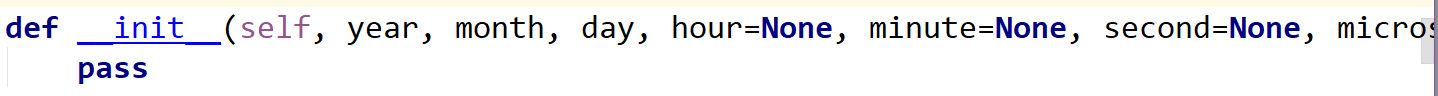
如果仅导入import datetime，则必须引用全名datetime.datetime。

datetime.now()返回当前日期和时间，其类型是datetime。

#### 获取指定日期和时间

**from** datetime **import** datetime  
  
date = datetime(2018,3,29,13,21,1)  
print(date)

这里还可以具体到秒、毫秒等。



#### datetime转换为timestamp

在计算机中，时间实际上是用数字表示的。我们把1970年1月1日 00:00:00 UTC+00:00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。

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  
  
date = datetime(2018,3,29,13,21,1)  
print(date.timestamp())



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。

#### 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  
  
date = datetime(2018,3,29,13,21,1)  
print(date.timestamp())  
print(date.fromtimestamp(date.timestamp()))

*#timestamp也可以直接被转换到UTC标准时区的时间*print(date.utcfromtimestamp(date.timestamp()))

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。

#### 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(**'2018-3-29 13:47:20'**, **'%Y-%m-%d %H:%M:%S'**)  
print(cday)

字符串'%Y-%m-%d %H:%M:%S'规定了日期和时间部分的格式。详细介绍可以参照：

<https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior>

#### 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'**))

#### 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()  
print(datetime(2018, 5, 18, 16, 57, 3, 540997))  
now + timedelta(hours=10)  
print(datetime(2018, 5, 18, 16, 57, 3, 540997))  
now - timedelta(days=1)  
print(datetime(2018, 5, 18, 16, 57, 3, 540997))  
now + timedelta(days=2, hours=12)  
print(datetime(2018, 5, 18, 16, 57, 3, 540997))

## 12.2 collections

collections是Python内建的一个集合模块，提供了许多有用的集合类。

#### 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 = (10,20)

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(10,20)  
print(p.x)  
print(p.y)

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：

print(isinstance(p,Point))  
print(isinstance(p,tuple))



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'**])

#### 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'**)  
deque([**'y'**, **'a'**, **'b'**, **'c'**, **'x'**])  
  
print(q)

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。

#### 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'**print(dd[**'key1'**]) *# key1存在*print(dd[**'key2'**]) *# key2不存在，返回默认值*

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。

#### 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)])  
*# dict的Key是无序的*print(d)  
od = OrderedDict([(**'a'**, 1), (**'b'**, 2), (**'c'**, 3)])  
*# OrderedDict的Key是有序的*print(od)

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)

#### Counter

Counter是一个简单的计数器，例如，统计字符出现的个数：

*#-\*-coding:utf-8-\*-***from** collections **import** Counter  
c = Counter()  
**for** ch **in 'programming'**:  
 c[ch] = c[ch] + 1  
print(c)



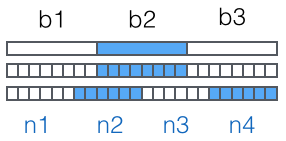
## 12.3 base64

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：

然后，对二进制数据进行处理，每3个字节一组，一共是3x8=24bit，划为4组，每组正好6个bit：



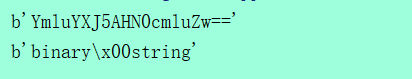
这样我们得到4个数字作为索引，然后查表，获得相应的4个字符，就是编码后的字符串。

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。

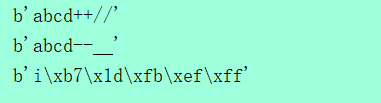
Python内置的base64可以直接进行base64的编解码：

**import** base64  
print(base64.b64encode(**b'binary\x00string'**))  
print(base64.b64decode(**b'YmluYXJ5AHN0cmluZw=='**))



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\_：

**import** base64  
  
print(base64.b64encode(**b'i\xb7\x1d\xfb\xef\xff'**))  
print(base64.urlsafe\_b64encode(**b'i\xb7\x1d\xfb\xef\xff'**))  
print(base64.urlsafe\_b64decode(**'abcd--\_\_'**))



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：

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。

## 12.4 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 & 0xff0000) >> 16  
b3 = (n & 0xff00) >> 8  
b4 = n & 0xff  
bs = bytes([b1, b2, b3, b4])  
print(bs)

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  
print(struct.pack(**'>I'**, 10240099))

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：

print(struct.unpack(**'>IH'**, **b'\xf0\xf0\xf0\xf0\x80\x80'**))

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\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：

print(struct.unpack(**'<ccIIIIIIHH'**, s))

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。

请编写一个bmpinfo.py，可以检查任意文件是否是位图文件，如果是，打印出图片大小和颜色数。

## 12.5 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，根据不通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)*print(h.hexdigest())



# 网络编程

自从互联网诞生以来，现在基本上所有的程序都是网络程序，很少有单机版的程序了。

计算机网络就是把各个计算机连接到一起，让网络中的计算机可以互相通信。网络编程就是如何在程序中实现两台计算机的通信。

举个例子，当你使用浏览器访问新浪网时，你的计算机就和新浪的某台服务器通过互联网连接起来了，然后，新浪的服务器把网页内容作为数据通过互联网传输到你的电脑上。

由于你的电脑上可能不止浏览器，还有QQ、Skype、Dropbox、邮件客户端等，不同的程序连接的别的计算机也会不同，所以，更确切地说，网络通信是两台计算机上的两个进程之间的通信。比如，浏览器进程和新浪服务器上的某个Web服务进程在通信，而QQ进程是和腾讯的某个服务器上的某个进程在通信。

Python 提供了两个级别访问的网络服务。：

低级别的网络服务支持基本的 Socket，它提供了标准的 BSD Sockets API，可以访问底层操作系统Socket接口的全部方法。

高级别的网络服务模块 SocketServer， 它提供了服务器中心类，可以简化网络服务器的开发。

## 13.1 什么是 Socket?

Socket又称"套接字"，应用程序通常通过"套接字"向网络发出请求或者应答网络请求，使主机间或者一台计算机上的进程间可以通讯。

建立网络通信连接至少要一对端口号(socket)。socket本质是编程接口(API)，对TCP/IP的封装，TCP/IP也要提供可供程序员做网络开发所用的接口，这就是Socket编程接口；HTTP是轿车，提供了封装或者显示数据的具体形式；Socket是发动机，提供了网络通信的能力。

## 13.2 TCP/IP简介

虽然大家现在对互联网很熟悉，但是计算机网络的出现比互联网要早很多。

计算机为了联网，就必须规定通信协议，早期的计算机网络，都是由各厂商自己规定一套协议，IBM、Apple和Microsoft都有各自的网络协议，互不兼容，这就好比一群人有的说英语，有的说中文，有的说德语，说同一种语言的人可以交流，不同的语言之间就不行了。

为了把全世界的所有不同类型的计算机都连接起来，就必须规定一套全球通用的协议，为了实现互联网这个目标，互联网协议簇（Internet Protocol Suite）就是通用协议标准。Internet是由inter和net两个单词组合起来的，原意就是连接“网络”的网络，有了Internet，任何私有网络，只要支持这个协议，就可以联入互联网。

因为互联网协议包含了上百种协议标准，但是最重要的两个协议是TCP和IP协议，所以，大家把互联网的协议简称TCP/IP协议。

通信的时候，双方必须知道对方的标识，好比发邮件必须知道对方的邮件地址。互联网上每个计算机的唯一标识就是IP地址，类似123.123.123.123。如果一台计算机同时接入到两个或更多的网络，比如路由器，它就会有两个或多个IP地址，所以，IP地址对应的实际上是计算机的网络接口，通常是网卡。

IP协议负责把数据从一台计算机通过网络发送到另一台计算机。数据被分割成一小块一小块，然后通过IP包发送出去。由于互联网链路复杂，两台计算机之间经常有多条线路，因此，路由器就负责决定如何把一个IP包转发出去。IP包的特点是按块发送，途径多个路由，但不保证能到达，也不保证顺序到达。

internet-computers

IP地址实际上是一个32位整数（称为IPv4），以字符串表示的IP地址如192.168.0.1实际上是把32位整数按8位分组后的数字表示，目的是便于阅读。

IPv6地址实际上是一个128位整数，它是目前使用的IPv4的升级版，以字符串表示类似于2001:0db8:85a3:0042:1000:8a2e:0370:7334。

TCP协议则是建立在IP协议之上的。TCP协议负责在两台计算机之间建立可靠连接，保证数据包按顺序到达。TCP协议会通过握手建立连接，然后，对每个IP包编号，确保对方按顺序收到，如果包丢掉了，就自动重发。

许多常用的更高级的协议都是建立在TCP协议基础上的，比如用于浏览器的HTTP协议、发送邮件的SMTP协议等。

一个TCP报文除了包含要传输的数据外，还包含源IP地址和目标IP地址，源端口和目标端口。

端口有什么作用？在两台计算机通信时，只发IP地址是不够的，因为同一台计算机上跑着多个网络程序。一个TCP报文来了之后，到底是交给浏览器还是QQ，就需要端口号来区分。每个网络程序都向操作系统申请唯一的端口号，这样，两个进程在两台计算机之间建立网络连接就需要各自的IP地址和各自的端口号。

一个进程也可能同时与多个计算机建立链接，因此它会申请很多端口。

了解了TCP/IP协议的基本概念，IP地址和端口的概念，我们就可以开始进行网络编程了。