Chapter 1. 快速改造：基础知识

1.1. 安装 phthon

1.1.1. Windows

1.1.2. Linux 和 UNIX

1、绝大多数 Linux 和 UNIX 系统(包括 Mac OSX)只要安装完毕，Python 解释器已经默认存在。
2、在命令提示符下输入 python 可以查看版本信息等，按(ctrl+D 退出解释器,注意不是 command)

1.2. 交互式解释器

1、解释器的具体外观和错误信息都取决于所使用的版本
2、不同于一些计算机语言，Python 不用分号作为语句的结束，如果喜欢的话，可以加上分号，但是不会有任何作用
3、">>>"是提示符

1.3. 算法是什么

1、程序员像给小孩喂饭一样告诉计算机具体细节，并且使用计算机能够理解的语言--算法。
2、算法不过是步骤或者食谱的另外一种文绉绉的说法--对于如何做某事的一份详细描述

1.4. 数字和表达式

1、交互式 Python 解释器可以当做非常强大的计算器使用
2、整数与整数的除法是整数，如果想要进行浮点数的运算，将参数与除法运算的两个数其中一个变为浮点数即可(1 写成 1.0)
3、如果只希望 Python 只执行普通的除法，可以在程序前加上以下语句
- from __future__ import division (注意，future 前后两个连续的下划线，并与 from 与 import 都含有一个空格)

4、另外还有一个方法，通过命令行运行 Python，使用命令开关-Qnew
5、Python 提供了另外一个用于实现整数除的操作---双斜线
6、取模运算：%
7、幂运算符：**
- 注意，幂运算符比取反运算符要高级，所以-3**2 等同于-(3**2)

1.4.1. 长整数

1、Python 可以处理非常大的整数
2、~~普通整数的范围在[-2147483648,2147483647]，如果需要此范围外的整数，可~~

~~以使用长整数，在末尾加上 L(小写 l 也可以，但是太像 1，不推荐使用！)~~
3、**Python 中的 long 已经与 int 合并了，现在不能加 L 了**

1.4.2. 十六进制和八进制

1、十六进制:
- 前缀：0x(注意是零而不是欧)

2、八进制：
- 前缀：0(注意是零而不是欧)

1.5. 变量

1、变量基本上就是代表(或者引用)某值得名字
2、在使用变量之前，需要对其赋值
3、变量名可以包括字母数字和下划线，变量不能以数字开头

1.6. 语句

1、已经介绍的两类语句：print 语句和赋值语句
2、语句和表达式的区别
- **表达式就是：某件事情**
- **语句就是：做某件事情**
- **语句和表达式的区别在赋值时会表现的更加明显一点，因为语句不是表达式，所以没有值可以提供交互解释器打印出来(比如 x=3)**

3、定义语句的一般特征：它们改变了事物，如赋值语句改变了变量，print 语句改变了屏幕显示的内容
4、赋值语句可能使任何计算机程序设计语言中最重要的语句类型，尽管现在还难以说清它们的重要性。
5、变量就像临时的"存储器"，它的强大之处在于，在操作变量的时候并不需要知道它们存储了什么值。

1.7. 获取用户输入

1、input("The meaning of life: ")　//交互式解释器执行 input(...)语句，它打印出字符串"The meaning of life: "作为新的提示符
2、if 语句
- if 语句可以让程序在给定条件为真得情况下执行某些操作(执行另外的语句)
- 一类条件是使用相等运算符==进行的相等性测试

1.8. 函数

1、可以使用一个函数 pow 来代替幂运算符**
2、函数就像小型程序一样，用来实现特定的功能
3、可以自己定义函数，通常会把 pow 等标准函数称为内建函数

4、函数调用可以简单看作另外一类表达式
5、常见的内建函数：
- abs()：绝对值
- round()：四舍五入

1.9. 模块

1、可以把模块想象成导入 Python 以增强其功能的扩展，需要使用特殊的命令 import 来导入模块，例如，floor、ceil 函数在名为 math 的模块中
2、**用 import 导入模块，然后按照"模块.函数"的格式使用这个模块的函数(Java 中，可以直接使用"包名.函数"的格式来调用函数，如果不想写包名才用 import)**
3、在确定自己不会导入同名函数(从不同模块导入)的情况下，希望不要每次调用函数的时候都写上模块的名字，可以使用 import 命令的另一种形式：
- from math import sqrt
- 使用了"from 模块 import 函数"这种形式的命令后，可以直接使用函数，而不需要模块名作为前缀

1.9.1. cmath 和复数

1、sqrt(-1)会出现："math domain error"或者"nan"。nan 是 not a number 的含义
2、Python 本身就提供了对复数的支持
3、Python 中没有单独的虚数类型，它们看做实数部分为 0 的复数

1.9.2. 回到__future__

1、通过它可以导入那些在未来会成为标准的 Python 组成部分的新特性

1.10. 保存并执行程序

1、交互式解释器是 Python 的强项之一，它让人们能够实时检验解决方案并且用这门语言做一些实验。
2、在交互式解释器里输入的一切都会在它退出的时候丢失，而我们真正想要的是编写自己和他们都能运行的程序

1.10.1. 通过命令提示符运行 Python 脚本

- Windows 平台：C:\> python hello.py
- UNIX 平台：$ python hello.py
- 命令一样，仅仅系统提示符不同

1.10.2. 让脚本像普通程序一样运行

1、有时候希望像运行其他程序(比如 Web 浏览器，文本编辑器)一样运行 Python 程序(也叫作脚本)，而不需要显示使用 Python 解释器。
- UNIX 中的标准实现：
  - 在脚本首行前面加上#!(叫做 pound bang 或者 shebang)
  - 在其后机上用于解释脚本的程序的绝对路径(在这里，用于解释代码的程序是 Python，mac 上是"**#!/usr/bin/env python**")

- 在实际运行脚本之前，必须让脚本文件具有可执行的属性：
- chmod a+x hello.py
- ./hello.py **(hello.py 无法执行，原因暂时不明)**

1.10.3. 注释

1、井号"#"在 Python 中有些特殊，在代码中输入它的时候，它右边的一切内容都会被忽略

2、"**# -*- coding: utf-8 -*-**"：在 Python 最开始加上引号里面的内容，让注释中可以有中文存在，写在"**#!/usr/bin/env python**"之后即可

1.11. 字符串

1、字符串：即一串字符，字符串在所有真实可用的 Python 程序中都会存在，并且有多种用法，其中最主要的用法就是表示一些文本

1.11.1. 单引号字符串和转义引号

1、Python 打印字符串的时候，会根据字符串的形式自动选择使用''或""

2、我们在程序中可以使用单引号和双引号，没有区别！但注意以下情景：

- "Let's go"
- 'Let's go' 会导致 SyntaxError：invalid syntax，因为在这里字符串为 Let，Python 并不知道如何处理后面的 s
- '"Hello, world!" she said'

3、另外一个选择是使用反斜线对引号进行转义

- 'Let\'s go!'
- "\"Hello, world!\" she said"

1.11.2. 字符串拼接

1、一个接着另一个写了两个字符串，但是只是在同时写下两个字符串的时候才有效，而且要一个紧接着另一个

2、**使用加法，推荐这种**

1.11.3. 字符串表示，str 和 repr

1、通过 Python 打印的字符串还是被引用号括起来的，这是因为 Python 打印值的时候回保持该值在 Python 代码中的状态，而不是你希望用户所看到的状态，而使用 print 就不一样了

2、str 函数：把值转换为合理形式的字符串，以便用户可以理解

3、repr 函数：它会创建一个字符串，以合法的 Python 表达式的形式来表示值

4、repr(x)可以写作`x`（反引号），不建议，应该坚持用 repr(x)

5、Python 不支持将字符串和数字进行相加

1.11.4. input 和 raw_input 的比较

1、区别：

- input：会假设用户输入的是合法的 Python 表达式(或多或少有些与 repr 函

数相反的意思)。例如你想要表示你输入的是一个字符串，那么必须以""或''的形式进行输入

- raw_input：它会把所有的输入当做原始数据，然后将其放入字符串中，相当于将输入的数据填入""或''之中

2、除非有特殊需要，否则应该尽可能地使用 raw_input

1.11.5. 长字符串、原始字符串和 Unicode

1、长字符串

- 如果需要写一个非常非常长的字符串，它需要跨越许多行，那么，可以使用三个引号代替普通引号'''...'''　或者"""..."""，由于有这种特殊的引用方式，因此可以在字符串中使用单引号和双引号，而不需要使用反斜线进行转义
- 普通字符串也可以跨行，如果一行中最后一个字符是反斜线，那么换行符本身就"转义"了，也就是被忽略了

2、原始字符串

- 原始字符串对于反斜线并不会特殊对待，在某些情况下这个特性是很有用的。
- 在普通字符串中，反斜线是有特殊的作用，它会转义，让你在字符串中加入通常情况下不能直接加入的内容。例如换行符为\n
- **我们可以使用反斜线对反斜线本身进行转义，表明想要用一个反斜线而不是用其来进行转义**
- 另一种方法是使用原始字符串
  - 原始字符串以 r 开头，紧接字符串
  - 不能在原始字符串结尾输入反斜线，即原始字符串最后一个字符不能使反斜线，除非你对反斜线进行转义(用于转义的反斜线会成为字符串的一部分)
  - 可以在原始字符串中同时使用单双引号，甚至三引号字符串

3、Unicode 字符串

- 字符串常量的最后一种类型就是 Unicode 字符串(或者称为 Unicode 对象，与字符串并不是同一个类型)
- Python 中普通字符串在内部是以 8 位的 ASCII 码形式存储的，而 Unicode 字符串则存储为 16 位 Unicode 字符，这样就能够表示更多的字符集了，包括世界上大多数语言的特殊字符
- Unicode 字符串使用 u 前缀
- 在 Python3.0 中，所有字符串都是 Unicode 字符串

Chapter 2. 列表和元组

1、数据结构：数据结构是通过某种方式(例如对元素进行编号)组织在一起的数据元素的集合，这些数据元素可以是数字或者字符，甚至可以是其他数据结构
2、在 Python 中，最基本的数据结构是序列，序列中每个元素被分配一个序号，即元素的位置，也称为索引，第一个索引是 0，以此类推

2.1. 序列概览

1、Python 包含六种内建序列：
- 列表：类似于 Java 中的容器？
- 元组：类似于 Java 中的数组？
- 字符串
- Unicode 字符串
- buffer 对象
- xrange 对象

2、列表和元组的区别
- 列表可以被修改，元组不能。如果要根据要求来添加元素，那么列表可能会更好用，而处于某些原因，序列不能修改的时候，使用元组则更为合适，使用元组的理由通常是技术性的，它与 Python 内部运作方式有关，这也是内建函数会返回元组的原因。

3、序列可以包含其他的序列
4、容器：
- Python 中还有一种名为容器的数据结构，容器基本上是包含其他对象的任意对象。
- 序列(例如列表和元组)和映射(例如字典)是两类主要的容器
- 序列中的每个元素都有自己的编号
- 映射中的每个元素则有一个名字(也称为键)

2.2. 通用序列操作

1、所有序列都可以进行以下操作：
- 索引
- 分片
- 加
- 乘
- 检查某个元素是否属于序列的成员(成员资格)
- 计算长度
- 找出最大最小元素的内建函数

2.2.1. 索引

1、序列中的所有元素都是有编号的，从 0 开始递增，这些元素可以通过编号分别访问
2、字符串就是由一个字符组成的序列

3、字符串字面值能够直接使用索引，而不需要一个变量引用他们，例如"Hello"[0]
4、如果一个函数调用返回一个序列，那么可以直接对返回结果进行索引操作，例如 forth=raw_input('Year: ')[3]

2.2.2. 分片

1、与使用索引来访问单个元素类似，可以使用分片操作来访问一定范围内的元素，**分片通过冒号隔开两个索引来实现**
2、分片操作的实现需要提供两个索引作为边界，第一个索引的元素是包含在分片内的，第二个则不包含在分片内
3、负数索引的分片：

- numbers=[1,2,3,4,5,6,7,8,9,10]
- numbers[-3:-1]-->[8,9] //-3 代表倒数第三，-1 代表倒数第一
- 如果想要倒数第三个之后的元素
  - numbers[-3:0]会返回[]，因为-3 出现得比 0 晚，只要分片中最左边的索引比它右边的晚出现在序列中，结果就是一个空序列
  - numbers[-3:]会返回[8,9,10]
- 同样适用于前 n 个元素 numbers[:3]
- 如果需要复制整个序列，可以将两个索引都置空
  - x=[1,2,6,5]
  - y=x[:] **//这样是有意义的，y 会得到 x 的一个拷贝，如果 y=x 那么，名字 y 与名字 x 会指向同一个实际的对象**

4、更大的步长

- 进行分片的时候，分片的开始和结束需要制定进行(不管是间接还是直接)，
- 另一个参数(在 Python2.3 加入到内建类型)---步长---通常都是隐式设定的，在普通分片中，步长是 1
- 分片操作就是按照这个步长逐个遍历开始和结束之间的元素，然后返回这些元素
- numbers[::4]按步长 4 取出所有元素
- **步长不能是 0，但是可以是负数，此时从右到左提取元素(这种情况下，分片开始代表右边界(包含)，分片结束代表左边界(不包含)**

2.2.3. 序列相加

1、通过使用加运算符可以进行序列的连接操作
2、两种相同类型的序列才能进行连接操作

2.2.4. 乘法

1、用数字 a 乘以一个序列会生成新的序列，而在新的序列中，原来的序列将被重复 a 次
2、None、空列表和初始化

- 空列表可以简单地通过两个中括号来表示"[]"
- 如果想创建占用 10 个元素的空间，去不包括任何有用内容的列表，需要一个值来代替空置，None
- None 是一个 Python 的内建值，其含义是：这里什么也没有
- 例子：sequence=[None]*10

2.2.5. 成员资格

1、为了检查一个值是否在序列中，可以使用 in 运算符

2、in 这个运算符检查某个条件是否为真，然后返回相应的值：条件为真返回 True，条件为假返回 False，这样的运算符叫做布尔运算符，而返回的值叫做布尔值

3、例子：

- permissions="rw"
- 'w' in permissions

2.2.6. 长度、最小值和最大值

1、内建函数 len、min、和 max 非常有用

- len 函数：返回序列中所包含的元素的数量
- min 函数：返回序列中最小的元素
- max 函数：返回序列中最大的元素

2.3. 列表：Python 的"苦力"

1、列表不同于元组的地方：列表是可变的--可以改变列表的内容，而且列表有很多有用的专门的方法

2.3.1. list 函数

1、因为字符串不能像列表一样被修改，所以有时候根据字符串创建列表会很有用，list 函数可以实现这个功能：list('Hello')

2、将一个由字符组成的列表转换成字符串：''.join(somelist)

2.3.2. 基本的列表操作

1、列表可以使用所有适用于序列的标准操作，例如索引，分片，连接和乘法

2、列表是可以修改的：元素赋值，元素删除，分片赋值以及列表方法

3、元素赋值：

- 为特定索引的元素进行赋值：name[dex]=somevalue
- 不能为一个位置不存在的元素进行赋值

4、删除元素：

- 使用 del 语句来实现：del name[dex]

5、分片赋值：

- 利用分片赋值，程序可以一次为多个元素赋值
- 使用分片赋值时，可以使用与原来不等长的序列将分片替换
  - name=list('Perl')
  - name[1:]=list('ython')
- 分片赋值语句可以在不需要替换任何原有元素的情况下插入新的元素
  - numbers=[1,5] //这是列表!!!
  - numbers[1:1]=[2,3,4]
- 可以通过分片赋值来删除元素
  - **numbers=[1,2,3,4,5]**
  - numbers[1:4]=[]

2.3.3. 列表方法

1、方法是一个与某些对象有紧密联系的函数，对象可能是列表、数字、也可能是字符串或者其他类型的对象

2、方法调用形式：对象.方法(参数)

3、常用列表方法

- append：用于在列表末尾追加新的对象
- count：统计某个元素在列表中出现的次数
- extend：在列表末尾一次性追加另一个序列中的多个值，即可以用新列表扩展原有的列表，这个看起来与连接很像，但是连接会返回一个全新的列表，而 extend 是扩展了调用该方法的列表
- index：用于从列表中找出某个值第一个匹配项的索引位置，找不到时会抛出异常
- insert：用于将对象插入到列表中，需要指定插入的索引
- pop：移除列表中的一个元素需要提供被移除元素的索引，如果不提供索引，那么默认是最后一个，并且返回该元素的值。**pop 方法是唯一一个既能修改列表又返回元素值的列表方法**
- 使用 pop 可以实现一种常见的数据结构---栈；Python 中的入栈(push)用 append 代替
- 如果需要实现一个先进先出 FIFO 的队列，可以使用 insert(0,arg)来代替 append 方法；或者继续使用 append 方法，但是用 pop(0)来代替 pop()。更好的方案是 collection 模块中的 deque 对象
- remove：**用于移除列表中某个值的第一个匹配项，仅第一项**
- reverse：将列表中的元素反向存放
- sort：用于在原位置对列表进行排序，在原位置排序意味着改变原来的列表，从而让其中的元素能按一定的顺序排列，而不是简单地返回一个已排序的列表副本
- 高级排序，希望元素按照特定的方式进行排序
  - 通过 compare(x,y)的形式自定义比较函数，定义好该函数后，将其提供给 sort 作为参数，后面会介绍
  - sort 方法有另外两个可选参数--key 和 reverse，如果要使用它们，需要通过名字来指定(这叫做关键字参数)

2.4. 元组：不可变序列

1、元组与列表一样，也是一种序列，唯一的不同是元组不能修改，字符串也是如此

2、创建元组的语法：**用逗号分隔一些值，逗号必不可少**

- 1,2,3
- (1,2,3)
- ()

3、**如果该元组只有一个值，也必须有逗号**

- 42,
- (42,)

2.4.1. tuple 函数

1、tuple 函数的功能与 list 函数基本上是一样的，tuple 函数以一个序列作为参数并把它转换为元组，如果参数就是元组，那么该参数就会被原样返回

2.4.2. 元组的基本操作

1、访问，使用下标：name[dex]
2、分片：元组的分片还是元组

2.4.3. 元组的意义：

1、元组可以在映射(和集合的成员)中当做键使用--而列表不行
2、元组作为很多内建函数和方法的返回值存在，也就是说你必须对元组进行处理，只要不尝试修改元组，那么"处理元组"在绝大多数情况下就是把它们当做列表来进行操作(除非要使用一些元组没有的方法，例如 index 和 count)
3、一般来说列表可能更满足对序列的所有需求

Chapter 3. 使用字符串

3.1. 基本字符串操作

1、所有标准的序列操作(索引，分片，乘法，判断成员资格，求长度，取最小值和最大值)对字符串同样适用
2、**字符串是不可变的，因此分片赋值等都是不合法的**

3.2. 字符串格式化:精简版

1、**字符串格式化使用字符串格式化操作符**，即百分号%来实现
2、在%的左侧放置一个字符串(格式化字符串)，右侧放置希望被格式化的值，可以使用一个值，如一个字符串或者数字，也可以使用多个值得元组或者字典，一般情况下使用元组

- format="Hello, %s. %s enouth for ya?"
- values=('world','Hot')
- print format % values

3、**如果使用列表或者其他序列代替元组，那么序列会被解释为一个值，只有元组和字典可以格式化一个以上的值**
4、格式化字符串的%s 部分称为转换说明符，它们标记了需要插入转换值得为之。s 表示会被格式化为字符串---如果不是字符串，则会用 str 将其转换为字符串，这个方法对大多数值都有效
5、如果要在格式化字符串里面包括百分号，必须使用%%，这样 Python 就不会将百分号误认为是转换说明符了。
6、如果要格式化实数(浮点数)，可以使用 f 说明转换说明符的类型，同时提供所需要的精度：一个句点再加上希望保留的小数位数，格式化转换说明符总是以表示类型的字符结束，所以精度应该放在类型字符前面，如%.3f
7、模板字符串

- string 模块里提供另外一种格式化值得方法：模板字符串
- 它的工作方式类似于很多 UNIX Shell 里的变量替换
  - from string import Template
  - s=Template('$x, glorious $x!')
  - s.substitute(x='slurm')
- **如果替换字段是单词的一部分，那么参数名就必须用花括号括起来，从而准确指明结尾**
  - s=Template("It's ${x}tastic!")
- 可以使用$$插入美元符号
- 除了关键字参数(包含一个字符)之外，还可以使用字典变量提供值/名对，所以才有了如果替换字段是单词一部分，必须用花括号将参数括起来

3.3. 字符串格式化：完整版

1、格式化操作符的右侧操作数可以是任意类型，如果是元组或者映射类型(如字典)那么字符串格式化将会有所不同

2、如果右操作数是元组的话，则其中每一个元素都会被单独格式化，每个值都需要一个对应的转换说明符
3、如果需要转换的元组作为转换表达式的一部分存在，那么必须将它用圆括号括起来，以免出错，例如 '%s plus %s equals %s' % (1,1,2)
4、**基本的转换说明符(注意与字符串格式化操作符的区别)**

- **%字符**：标记转换说明符的开始
- **转换标志(可选)**：-表示左对齐，+表示在转换值之前要加上正负号，" "(空白字符)表示正数之前保留空格
- **最小字段宽度(可选)**：转换后的字符串至少应该具有该值指定的宽度，如果是*，则宽度会从值元组中读出
- **点(.)后跟精度值(可选)**：
  - 如果转换的是实数，精度值就表示出现在小数点后的位数
  - 如果转换的是字符串，那么该数字就表示最大字段宽度
  - 如果是*，那么精度将会从元组读出
- **转换类型：**
  - d,i：带符号的十进制整数
  - o：不带符号的八进制
  - u：不带符号的十进制
  - x,X：不带符号的十六进制
  - e,E：科学计数法表示的浮点数
  - f,F：十进制浮点数
  - g,G：如果指数大于-4 或小于精度值则和 eE 相同，其他情况与 fF 相同
  - C：单字符，接受整数或单字符字符串
  - r：字符串，使用 repr 转换任意 Python 对象
  - s：字符串，使用 str 转换任意 Python 对象

3.3.1. 简单转换

- 只需要写出转换类型

3.3.2. 字段宽度和精度

- 转换说明符可以包括字段宽度和精度
- 字段宽度是转换后的值锁保留的最小字符个数
- 精度对于数字转换来说则是结果中应该包含的小数位数
- 精度对于字符串转换来说是转换后的值所能包含的最大字符个数
- 这两个参数都是整数(首先是字段宽度，然后是精度)，通过点号(.)分隔，如果只给出精度，就必须包含点号
- 可以使用*(星号)作为字段宽度或者精度(或者两者都使用*)，此时数值会从元组参数中读出
  - 如果是"*.*"：元组的第一个值会作为字段宽度，第二个值会作为精度

3.3.3. 符号、对齐和用 0 填充

- 在字段宽度和精度值之前还可以放置一个"标志",该标志可以是'0','+','-',' '
- print('%+5d' %10)+'\n'+('%+5d' % -10)
- print('% 5d' %10)+'\n'+('% 5d' % -10)

3.4. 字符串方法

1、字符串从 string 模块中"继承"了很多方法，在早期的 Python 版本中，这些方法都是作为函数出现的

2、有用的字符串常量：

- string.digits：包含数字 0~9 的字符串
- string.letters：包含所有字母(大写或小写)的字符串
- string.lowercase：包含所有小写字符的字符串
- string.uppercase：包含所有大写字母的字符串
- string.printable：包含所有可打印字符的字符串
- string.punctuation：包含所有标点的字符串
- 字母字符串常量与地区有关(Python 所配置的语言有关)，如果可以确定使用的是 ASCII，那么可以在变量中使用 ascii_前缀，例如 string.ascii_letters

3.4.1. find

- find 方法可以在较长的字符串中查找子串，它返回子串所在位置的最左端索引，如果没有找到返回-1
- 这个方法还可以接受可选的起始点和结束点参数，表示的范围为[begin,end)
  - subject='$$$ Get rich now!!!$$$'
  - subject.find('$$$')
  - subject.find('$$$',1)
  - subject.find('!!!',0,16)

3.4.2. join

- **join 是非常重要的字符串方法，它是 split 方法的逆方法，用来连接序列中的元素**
- 被连接的序列元素必须是字符串，例如(1,2,3)不行，必须写成('1','2','3')
- seq=['1','2','3']
- sep='+'
- sep.join(seq)

3.4.3. lower

- lower 方法返回字符串的小写字母版
- 标题转换
  - title 方法：将字符串转换为标题---所有单词首字母大写，其他字母小写
  - capwords()**函数(string 模块)**：*string*.capwords('...')

3.4.4. replace

- replace 方法返回某字符串的所有匹配项均被替换之后得到的字符串

3.4.5. split

- **split 是一个非常重要的字符串方法，它是 join 的逆方法，用来将字符串分割成序列**

3.4.6. strip

- strip 方法返回去除两侧(不包括内部)空格的字符串
- 在 Java 中的类似方法是 String.trim()
- 也可以指定出去的字符，将他们列为参数即可
- strip(' *!')：会出去两侧所有的' ','*','!'

3.4.7. translate

- translate 方法和 replace 方法一样，可以替换字符串中某些部分，但是和 replace 不同的是，translate 方法只处理单个字符，它的优势在于可以同时进行多个替换，有时候比 replace 高效
- 使用 translate 之前需要先完成一张转换表，转换表是以某字符替换某字符的对应关系。因为这个表(事实上是字符串)有多达 256 项，因此手写是很费力的，可以使用 string 模块里面的 maketrans 函数即可
  - from string import maketrans
  - table=maketrans('cs','kz')：将初始对应关系(全部相同的对应关系)中的 c 对应成 k，s 对应成 z，其他不变
  - '.....'.translate(table)

Chapter 4. 字典：当索引不好用时

- 列表这种数据结构适合于将值组织到一个结构中，并且通过编号对其进行引用
- 映射：通过名字来引用值得数据结构，类似于 Map
- 字典是 Python 中唯一内建的映射类型，字典中的值并没有特殊的顺序，但是都存储在一个特定的键(Key)下。

4.1. 字典的使用

1、构造字典的目的，不管是实现中的字典还是在 Python 中的字典，都是为了可以轻松查找某个特定的词语(键)，从而找到它的定义(值)

2、字典适用条件：

- 表示一个游戏棋盘的状态，每个键都是由坐标值组成的元组
- 存储文件修改时间，用文件名作为键
- 数字电话/地址簿

4.2. 创建和使用字典

1、字典可以通过以下方式创建

- phonebook={'Alick':'2341','Beth':'9102','Cecil':'3258'}
- 字典由多个键以及与其对应的值构成键-值对组成，我们把键-值对称为项
- 每个键和值之间用冒号隔开，项之间用逗号隔开，整个字典用花括号括起来
- 空字典由两个大括号组成

4.2.1. dict 函数

1、可以使用 dict 函数，通过其他映射(比如其他字典)或者(键，值)对的序列建立字典

2、例子：

- items=[('name','Gumby'),('age',42)]
- d=dict(items)
-
- d=dict(name='Gumby',age=42)

3、不带参数的 dict 函数返回空字典

4.2.2. 基本字典操作

- len(d)：返回 d 中(键-值对)的数量
- d[k]：返回关联到键 k 上的值
- d[k]=v：将值 v 关联到键 k 上
- del d[k]：删除键为 k 的项
- k in d：检查 d 中是否含有键为 k 的项
- 键类型：字典的键不一定是整数类型(可以是)，键可以是任意的不可变类型，比如浮点型，字符串或者元组

- **自动添加：即使键初始在字典中并不存在，也可以为它赋值，这样字典就会建立新的项(跟 C++有点类似)**，对于列表则不能将值关联到列表范围之外的索引上
- **成员资格**：表达式 k in d(d 为字典)查找的是键而不是值，表达式 v in l(l 为列表)，用于查找值而不是索引
- 在字典中检查键成员资格比在列表中检查值得成员资格更加高效，数据结构的规模越大，两者的效率差距越明显

4.2.3. 字典的格式化字符串

1、如果使用字典而不是元组，会使字符串格式化更酷炫一点，在每个转换说明符中的%字符后面，可以加上键(用圆括号括起来)，后面再跟上其他说明元素
2、当以这种方式使用字典的时候，只要所有给出的键都能在字典中找到，就可以使用任意数量的转换说明符了，而使用元组的时候需要一一对应
3、"Cecil's phone number is %(Cecil)s." % phonebook

4.2.4. 字典方法

1、clear：清除字典中所有的额项，这是原址操作，无返回值
2、copy：返回一个具有相同键-值对的新字典，浅复制，值不是副本
   - 在副本替换值的时候，原字典不受影响
   - **副本修改值的时候，原字典会受影响**
   - 避免这个问题可以使用 copy 模块的 deepcopy 函数来完成操作
3、fromkeys：使用给定的键建立新的字典，每个键都对应一个默认的 None
   - {}.fromkeys(['name','age'])
   - **dict 是所有字典的类型**
   - dict.fromkeys(['name','age'])
   - 可以提供默认值，来代替 None
   - dict.fromkeys(['name','age'],('unknown'))
4、get：访问字典的方法
   - 当访问字典中不存在的项**不会出错**，而是返回 None
   - print d['不存在的键']会出错，但可以用来赋值 d['不存在的键']=something
5、has_key：检查字典中是否含有特定的键，相当于 k in d
6、items 和 iteritems
   - items 方法将字典所有的项以及列表方式返回，列表中的每一项都表示为(键，值)对的形式，但是在返回时并没有遵循特定的次序
     - d.items()
   - iteritems 方法作用大致相同，但是会返回一个迭代器对象而不是列表
     - it=d.iteritems()
     - list(it) <==将迭代器转换成列表？？？
7、keys 和 iterkeys
   - keys 方法将字典中的键以列表形式返回
   - iterkeys 返回针对键的迭代器
8、pop：用来获得对应于给定键的值，然后将这个键-值对删除
   - 与列表中的 pop 方法不同，没有默认无参数的 pop，因为对于字典没有"末尾元素(项)这一说法"

9、popitem：弹出随机的项，因为字典并没有末尾元素(项)这一说法，常用于一个接一个地移除并处理项，相当于：遍历然+调用 pop 方法

10、setdefault：该方法在某种程度上类似于 get 方法，能够获取与给定键相关联的值

- **可以设置第二个参数，表示当键不存在的时候，更新并返回的值，如果不设置，则默认为 None，此时就与 get 一样了**
- 当键不存在的时候，setdefault 返回默认值，并相应地更新字典
- 当键存在的时候，返回与其对应的值，不改变字典
- d.setdefault('name','N/A')　<=='N/A'就是当键不存在的时候，更新并返回的值

11、update：利用一个字典更新另外一个字典

- 提供的字典中的项会被添加到旧字典中，若有相同的键则会进行覆盖
-

12、values 和 itervalues

- values 方法以列表的形式返回字典中的值
- itervalues 方法返回值得迭代器

Chapter 5. 条件、循环和其他语句

5.1.1. print 和 import 的更多信息

5.1.2. 使用逗号输出

- print 打印表达式，无论是字符串还是其他类型，都会自动转换成字符串，跟 Java 的 System.out.println 有点类似(调用该类型的 toString()方法)
- 每个参数之间都带有空格
- print 的参数并不能构成元组
  - print 1,2,3 -->1 2 3
  - print (1, 2, 3)-->(1, 2, 3)
- Python3.0 中，print 不再是语句，而是函数

5.1.3. 把某件事作为另一件事导入

- import somemodule
- from somemodule import somefunction
- from somemodule import somefunction,anotherfunction,yetanotherfunction
- from somemodule imort * <==导入该模块所有的功能
- form somemodule import somefunction as anothername <==导入函数并赋予别名，可以避免多个模块的函数重复

5.2. 赋值魔法

5.2.1. 序列解包

1、多个赋值操作同时进行：
- x,y,z=1,2,3 <==丧心病狂，目前只有 Python 支持这样的赋值

2、交换变量：
- x,y=y,x <==更他妈丧心病狂！

3、序列解包(递归解包):将多个值的序列解开，然后放到变量的序列中
- 当函数或者方法返回元组(或者其他序列或可迭代对象)时，这个特性尤为有用，比如返回字典中的一个项(键值对)
- 解包的序列中的元素数量必须和放置在赋值符号=左边的变量数量完全一致，否则 Python 在赋值时将引发异常
- Python3.0 中另一个解包特性，可以想在函数的参数列表中一样使用星号运算符，例如 a,b,*rest=[1,2,3,4]，在 a 和 b 赋值后，其他参数都收集到 rest 中

5.2.2. 链式赋值

- 链式赋值：将同一个值赋值给多个变量的捷径
- x=y=1

5.2.3. 增量赋值(C++Java 中的复合赋值运算符)

- x+=1
- x*=2
- s+='foo'

5.3. 语句块：缩排的乐趣

1、语句块：

- 语句块并非一种语句
- 语句块是在条件为真(条件语句)时执行或执行多次(循环语句)的一组语句
- 在代码前放置空格来缩进语句即可创建语句块，没有花括号啦！

2、注意：

- 使用 tab 字符也可以缩进语句块，Python 将一个 tab 字符解释为到下一个 tab 字符位置的移动，而一个 tab 字符位置为 8 个空格
- 但是标准的方式只是用空格，尤其是在每个缩进需要 4 个空格的时候
- 块中的每行都应该缩进同样的量

3、Python 语句块的独特性：

- 很多语言都使用特殊字符或单词({或 begin)来表示一个语句块的开始，用另外的单词或者字符(}或 end)表示语句块的结束
- **Python 中，冒号(:)用来标识语句块的开始，块中每一个语句都是缩进的，当退回到和已经闭合的块一样的缩进量时，就表示当前块已经结束了(很多程序编辑器和集成开发环境都知道如何缩进语句块，可以帮助用户轻松把握缩进**

5.4. 条件和条件语句

- 真值(也叫作布尔值，这个名字根据在真值上做过大量研究的 George Boole)
- 以下值在作为布尔表达式的时候会被解释器看做假
  - False
  - None
  - 0
  - "" <==空字符串
  - () <==空元组
  - [] <==空列表
  - {} <==空字典
- bool 函数可以用来将其他类型的值转换为 bool 值，但是可以不用转换，直接用，因为 Python 会自动转换这些值

5.4.1. 条件执行和 if 语句

- if 语句可以实现条件执行，即如果条件判定为真，那么后面的语句块就会被执行，如果条件为假，就不执行

5.4.2. else 子句

- 对于 if 语句，可以用 else 子句增加一种选择

5.4.3. elif 子句

- 如果需要多个条件，就可以使用 elif

5.4.4. 嵌套代码块

- 注意缩进即可

5.4.5. 更复杂的条件

1、比较运算符：

- x==y
- x<y
- x>y
- x>=y
- x<=y
- x!=y (也可以写作 x<>y，不建议用，看到知道什么意思即可)
- x is y
- x is not y
- x in y
- x not in y
- Java 中的 x==y
  - 对于类类型是引用判断，即判断两个引用是否引用了同一个对象，而值判断要调用 x.equals(y)，Object 的方法
  - 如果是内置类型，则判断值是否相等

2、比较对象可以使用内建 cmp 函数

3、相等运算符==

4、同一运算符 is：is 运算符用来判断同一性而非相等性，(即 java 中的用于引用的==运算符，即判断是否引用了同一个对象)

5、**避免将 is 运算符用于比较类似数值和字符串这类不可变值**

6、成员资格运算符(类似 Java 中的 contains)

7、字符串和序列比较，按字典序比较

8、比较也适用于序列

9、短路特性

10、三元运算符：a if b else c <==如果 b 为真，返回 a，否则返回 c

5.4.6. 断言

- if not condition:
- crash program <==让程序崩溃
- 语句中使用关键字 assert
- 可以要求某些条件必须为真，否则就插入一个断点
- 条件后面可以添加字符串，用来解释断言
- assert condition, 'some explanation'

5.5. 循环

5.5.1. while 循环

- x=1
- while x<=100 :

- print x
- x+=1

5.5.2. for 循环

1、类似范围 for 语句，foreach 语句

- words=['this','is','an','ex','parrot']
- for word in words :
- print word

2、迭代某范围的数字(正常的 for 语句)

- for number in range(1,101):
- do something

3、如果能使用 for 循环，就尽量不用 while

4、range 函数的工作方式类似于分片，左闭右开的区域

5、xrange 函数的循环行为类似 range 函数，区别在于 range 一次创建整个序列，而 xrange 一次只创建一个数，当需要迭代一个巨大的序列时，xrange 会更高效

6、在 Python3.0 中，range 会转换成 xrange 风格的函数

5.5.3. 循环遍历字典元素

1、一个简单的 for 语句就能遍历字典的所有键

- for key in d:
- do something
- for key,value in d.items():
- do something
- 字典元素的顺序通常是没有定义的，换句话说，迭代的时候，字典中的键和值保证都能被处理，但是处理顺序不定

5.5.4. 一些迭代工具

1、并行迭代

- 内建的 zip 函数可以用来进行并行迭代，把两个或多个序列"压缩"在一起，然后返回一个元组的列表
- zip 函数可以作用任意多的序列
- zip 可以处理不等长的序列，当最短序列用完时就会停止

2、按索引迭代

- **enumerate 函数：在提供索引的地方迭代索引-值对**
- for index,string in enumerate(strings):
- do something

3、反转和排序迭代

- reversed、sorted：与列表的 reserse、sort 方法类似，但是作用域任何序列或者可迭代对象之上，不是原地修改对象，而是返回反转或排序后的版本

5.5.5. 跳出循环

1、break：跳出循环

2、continue：跳过当前剩余的循环体，开始下一次循环

3、while True/break 习语

- 避免设置哑值
- 避免重复

5.5.6. 循环中的 else 子句

1、语法：

- for n in range(99,81,-1):
- do something
- else: do anotherthing

2、else 子句：仅在没有调用 **break** 时执行

5.6. 列表推导式：轻量级循环

1、[x*x for x in range(10) if x%3==0]

2、[(x,y) for x in range(3) for y in range(3)]

5.7. 三人行

- pass、del、exec

5.7.1. 什么都没发生

1、pass：

- pass 在代码中做占位符使用
- 因为 **Python** 空代码块是非法的

5.7.2. 使用 del 删除

1、当引用被置位 None 后，对应就"漂"在内存里面了，因为没有任何名字(Java 叫做引用)绑定到它上面，Python 解释器直接删除了这个对象(这种行为被称为垃圾收集,与 Java 类似)

2、del 语句

- 不仅移除一个对象的引用，也会移除那个名字本身
- del 只负责删除引用关系以及名字(引用)本身，并不删除值(对象)，Python 解释器会负责值(对象，内存)的回收

5.7.3. 使用 exec 和 eval 执行和求值字符串

1、exec：

- exec 最有用的地方在于可以动态地创建代码字符串
- 为了安全起见，可以增加一个字典，起到命名空间的作用
  - from math import sqrt
  - scope={}
  - exec 'sqrt=1' in scope
  - sqrt(4)
  - scope['sqrt']

2、eval：

- 用于求值，是类似于 exec 的内建函数，exec 语句会执行一系列 Python 语句，而 eval 会计算 Python 表达式，并返回结果
- eval(raw_input('Enter an arithmetic expression: '))
- **跟 exec 一样，eval 也可以提供命名空间，尽管表达式几乎不像语句那样为变量重新赋值???什么意思**
- 事实上，可以给 eval 语句提供两个命名空间，一个全局一个局部，全局的必须是字典，局部的可以是任何形式的映射

3、**事实上 exec 和 eval 并不常用**

Chapter 6. 抽象

6.1. 懒惰即美德

6.2. 抽象和结构

- 抽象可以节省很多工作
- 抽象是计算机程序可以让人读懂的关键，计算机乐于处理精确和具体的指令

6.3. 创建函数

- 函数是可以调用的(可能带有参数，放在圆括号中的值)，它执行某种行为并且返回一个值
- 内建的 callable 函数可以用来判断函数是否可以调用
- callable 在 Python3.0 中不再可用，需要用表达式 hasattr(func,__call__)代替
- **使用 def 语句可以定义函数**

6.3.1. 文档化函数

- 如果在函数的开头写下字符串，他就会作为函数的一部分进行存储，这成为文档字符串
- func.__doc__ <==访问文档字符串
- 内建的 help 函数非常有用，在交互式解释器中使用它就可以得到关于函数，包括它的文档字符串的信息

6.3.2. 并非真正函数的函数

- 数学意义上的函数，总在计算其参数后返回点什么
- Python 的有些函数却并不返回任何东西
- 这类函数没有 return 语句，或者虽有 return 语句但是 return 后面不跟任何东西
- **所有的函数的确都返回了东西，当不需要他们返回值的时候，它们就返回 None**

6.4. 参数魔法

6.4.1. 值从哪里来

- 写在 def 语句中函数名后面的变量通常叫做函数的形参
- 调用函数时提供的值是实参，或者称为参数

6.4.2. 能改变参数吗

- **在函数内为参数赋予新值不会改变外部任何变量的值(具体是什么机制，值传递还是跟 Java 一样，在调用结束后会复原引用的指向???)**
  - 这句话是没有问题的，说的是赋值，就像 Java 一样，无论形参是内置类型或者是类类型，在函数内重新赋值(改变内置类型的值，或者将引用半丁到另一个对象上)，都不会对外部变量造成影响。

- 这句话不包含改变参数的值这个概念，下面会解释
- 总的来说，跟 Java 是一样的

➢ 在 Java 中
- 以值传递的方式传递内置类型，改变形参不会影响传入的参数
- 以引用传递的方式传递类类型，在函数内部改变其引用的对象后，在函数调用结束，仍然会将其绑定到传入时的对象上，但是可以改变对象的状态

➢ 参数存储在局部作用域内

➢ **字符串、以及数字和元组是不可变的，即无法被修改，也就是说只能用新的值覆盖(就像 Java 中 String，改变其值，相当于重新将名字引用到一个新的对象上，属于赋值的范畴)**

➢ 对于其他类型，参数传递是以引用的方式传递，即形参与外部变量引用的是同一个对象，函数内部通过形参改变了对象的状态，会影响到外部变量的状态，因为他们引用的是同一个对象

➢ 如果要避免共享同一个对象这种情况，可以利用切片制作副本，返回的切片总是一个副本

6.4.3. 关键字参数和默认值

➢ 目前我们使用的参数叫做位置参数，因为它们的位置很重要

➢ **在调用函数的时候可以提供形参的名字，这样即使乱序也没有关系(这个是 Python 特有的)，这类使用参数名提供的参数叫做关键字参数，它的主要作用是可以明确每个参数的作用**

➢ 使用关键字参数，多打了一些字，但是每个参数的含义变得更加清晰，就算弄乱了参数的顺序，对于程序的功能也没有任何影响

➢ 关键字参数可以在定义函数的时候给参数提供默认值(C++也可以提供默认值，但是 Java 好像不行???)

6.4.4. 收集参数

1、*
- ➢ 处理位置参数的"收集操作"
- ➢ def func(*arg): do something
- ➢ 参数前的星号将所有的值放在同一个元组中
- ➢ 星号的意思是收集其余的位置参数

2、**
- ➢ 处理关键字参数的"收集"操作
- ➢ 返回的是一个字典

6.4.5. 参数搜集的逆过程

➢ 调用方法的时候使用

➢ *对应的是位置参数

➢ **对应的是关键字参数

➢ 星号必须定义和调用配合使用，要么定义和调用全部使用，要么全部不用

6.5. 作用域

1、变量：

- **可以看做是值的名字，就像字典一样，键值引用，但是变量和所对应的值是个"不可见"的字典，但是这种说法已经和接近真实情况了**
- 内建的 vars 函数可以返回这个字典
  - x=1
  - scope=vars()
  - scope['x'] ==>1
  - scope['x']+=1
  - x =>>2

2、作用域

- 一般来说，vars 所返回的字典是不能修改的，因根据官方 Python 文档的说法，结果是未定义的，即可能得不到想要的结果
- **这类"不可见字典"叫做命名空间或者作用域**
- 除了全局作用域之外，每个函数调用都会创建一个新的作用域
- 函数参数工作的原理类似局部变量，因此用全局变量的名字作为参数名字没有问题
- 读取全局变量一般来说并不是问题，但是如果局部变量或者参数的名字和想要访问的全局变量名字相同的话，就不能直接访问了，因为全局变量的名字会被局部变量屏蔽
- **如果在函数内部将值赋予一个变量，它自动成为局部变量，除非告知 Python 将其声明为全局变量(global 关键字)**

3、嵌套作用域

- **Python 的函数是可以嵌套的(C++ Java 不可以!!!)**
- 嵌套一般来说并不是很有用，但它有一个很突出的应用，需要用一个函数创建另一个，**外层函数返回里层函数，也就是函数本身被返回了，但并没有被调用，重要的是返回的函数还可以访问它的定义所在的作用域，换句话说，它带着它的环境(和相关的局部变量)**

```
def multiplier(factor):
    def multiplyByFactor(number):
        return number*factor
    return multiplyByFactor
d=multiplier(2)
d(5)    ==> 10
multiplier(5)(4)    ==>20
```

- 类似 multiplyByFactor 函数存储子封闭作用域的行为叫做闭包

4、外部作用域的变量一般是不能进行重新绑定的，但是在 Python3.0 中，nonloal 关键字被引入，它和 global 关键字类似，可以让用户对外部作用域的变量进行赋值

6.6. 递归

1、有用的递归函数：

- 当函数直接返回值时有基本实例(最小可能性问题)
- 递归实例，包括一个或多个问题较小部分的递归调用

6.6.1. 两个经典：阶乘和幂

1、阶乘

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

2、幂函数

```
def power(x,n)
    if n==0:
        return 1
    else:
        return x*power(x,n-1)
```

Chapter 7. 更加抽象

- 创建自己的对象是 Python 的核心概念
- Python 被称为面向对象的语言(和 SmallTalk、C++、Java 以及其他语言一样)

7.1. 对象的魔力

1、在面向对象的程序设计中，术语对象基本上可以看做数据以及由一些列可以存取、操作这些数据的方法所组成的集合
2、使用对象代替全局变量和函数的原因：
- 多态(Polymorphism)：意味着可以对不同类的对象使用同样的操作
- 封装(Encapsulation)：对外部世界隐藏对象的工作细节
- 继承(Inheritance)：以通用的类为基础建立专门的类对象

7.1.1. 多态

1、多态意味着就算不知道变量所引用的对象类型是什么，还是能对它进行操作，而它也会根据对象(或类)类型的不同而表现出不同的行为
2、多态的多种形式，任何不知道对象到底是什么类型，但是又要对象"做点什么"的时候，都会用到多态
3、很多函数和运算符都是多态的---你写的绝大多数程序可能都是，即使你并非有意为之。
4、只要使用多态函数和运算符，就会与"多态"发生关联
5、事实上，唯一能够毁掉多态的就是使用函数显式地检查类型

7.1.2. 封装

1、封装是指向程序中的其他部分隐藏对象的具体实现细节的原则
2、多态可以让用户不知道是什么类(对象类型)的对象进行方法调用，而封装是可以不用关心对象是如何构建的而直接进行调用
3、基本上，需要将对象进行抽象，调用方法的时候不用关心其他东西，比如它是否干扰了全局变量，所以能将名字"封装"在对象内，可以将其作为特性存储

7.1.3. 继承

1、继承是另外一个懒惰的行为

7.2. 类和类型

7.2.1. 类到底是什么

1、可以将类或多或少视为种类或类型的同义词
2、对象为类的实例

7.2.2. 创建自己的类

1、例子

```
class Person
```

```
def setName(self,name):
    self.name=name
```

- 自动将自己作为第一个参数传入函数，形象的命名为 self，**注意 self 不是关键字，可以随意改。**(不同于 C++(常量指针)，Java 的 this(引用),this 是个关键字)
- 为什么 name 不用事先说明，直接就能用???

7.2.3. 特性、函数和方法

1、self 参数事实上正是方法和函数的区别

- 方法(更专业一点称为绑定方法)将它们的第一个参数绑定到所属的类型上，我们在调用时无需显式提供
- self 参数并不依赖于调用方法的方式
  - 将一个变量引用绑定到方法上面，对该变量进行类似函数的调用，也会对 self 参数进行访问 P119

2、再论私有化

- 默认情况下，程序可以从外部访问一个对象的特性
- 有些人认为这样破坏了封装的原则，它们认为对象的状态对于外部应该是完全隐藏的
- 关键在于方法可能会含有一些额外的操作，如果直接访问对象的特性会省略这些额外的操作，可能达到不同的效果，为了避免这种情况的发生，应该使用私有特性(就是访问权限的事情，说的真尼玛费劲)

3、Python 并不直接支持私有方式，而要靠程序员自己把握外部进行特性修改实际，**为了让方法或者特性变为私有(从外部无法访问)，只要在它的名字前面加上双下划线即可**

4、*在类的定义中，所有以双下划线开始的名字都被翻译为函数名字前面加上单下划线和类名的形式(__func==>_ClassName__func，在了解这些幕后之后，实际上还是能在类外访问这些私有方法，尽管不该这么做*

5、确保其他人不会访问对象的方法和特性是不可能的，但是这类"名称变化术"就是他们不该访问这些函数或者特性的强有力信号

6、**如果不需要使用这种方法，但是又想让其他对象不要访问内部数据(什么狗屁意思???)**，可以使用单下划线，这不过是个习惯，但的确有实际效果：前面有下划线的名字不会被带星号的 import 语句导入(from module import *)

7.2.4. 类的命名空间

1、所有位于 class 语句中的代码都在特殊的命名空间中执行---类命名空间，这个命名空间可由类内所有成员访问

2、*类的定义其实就是执行代码块，例如，在类的定义区并不只限定只能使用 def 语句*

3、在类作用域定义的变量可供所有成员(实例)访问，**有点像 C++，Java 中 static 变量的感觉**

4、有些语言支持多种层次的成员变量(特性)私有性，比如 Java 就支持四种级别，尽管单双下划线在某种程度上给出两个级别的私有性，但 Python 并没有真正的私有化支持

5、可以在实例中重绑定类作用域中的变量，使其变为对象的特性，该特性会屏蔽类范围内的同名变量(对象.名字==>指的就是特性)，但是可以使用"类名.变量名"来引用类作用域中的变量

6、**非常奇怪的一点，Python 中引用是没有具体类型的，可以绑定到任何东西上面，连方法都可以绑定，而且将一个东西作为特性存到对象中是不需要在其他地方说明的，只需一句赋值一句即可，Matlab 既视感**

7.2.5. 指定超类

1、子类可以扩展超类的定义

2、将其他类名卸载 class 语句后的圆括号内可以指定超类

3、类名可以继承方法

7.2.6. 检查继承

1、issubclass(SubClass,SuperClass)

2、ClassName.__bases__ #返回基类类型

3、obj.__class__ #返回对象的类型

4、如果使用__metaclass__=type 或从 object 继承的方式来定义新式类，那么可以使用 type(s)查看实例所属类型

7.2.7. 多个超类

1、__bases__暗示了它的基类可能会多于一个

2、Python 支持多重继承，是个非常有用的工具，除非特别熟悉多重继承，否则避免使用

3、当使用多重继承时，有个需要注意的地方，如果一个方法从多个超类继承(也就是说你有两个具有相同名字的不同方法)，那么必须注意一下超类的顺序

- 先继承的类中的方法会重写后继承的类中的方法
- 如果超类们共享一个超类，那么在查找给定方法或属性时访问超类的顺序称为 MRO(Method Resolution Order，方法判定顺序)

7.2.8. 接口和内省

1、接口的概念与多态有关，在处理多态对象时，只要关心它的接口(或称"协议")即可，也就是公开的方法和特性

2、在 Python 中，不用显式地指定对象必须包含哪些方法才能作为参数接收，例如**不用像 Java 一样显式地编写接口，可以在使用对象的时候假定它可以实现你所要求的行为，如果它不实现的话，程序就会失败，比 Java 更加动态**

3、检查方法是否存在 hasattr(obj,'func')或者 hasattr(obj,"func")

4、查看对象中所有存储的值：__dict__特性

7.3. 一些关于面向对象设计的思考

1、将属于一类的对象放在一起，如果一个函数操纵一个全局变量，那么两者最好放在类内作为特性和方法出现

2、不要让对象国语亲密，方法应该只关心自己实例的特性，让其他实例管理自己的状态

3、小心继承，尤其是多重继承，继承机制有时很有用，但也会在某些情况下让事情变得过于复杂。多继承难以正确使用，更难以调试
4、简单就好，让你的方法小巧，一般来说，多数方法都应能在 30 秒内被读完(以及理解),尽量将代码行数控制在一页或者一屏之内

7. 4. 小结

1、对象：对象包括特性和方法，特性只是作为对象的一部分的变量，方法则是存储在对象内的函数。(绑定)方法和其他函数的区别在于方法总是将对象作为自己的第一个参数，这个参数一般被称为 self
2、类：类代表对象的集合(或一类对象)，每个对象(实例)都有一个类，类的主要任务是定义它的实例会用到的方法
3、多态：多态是实现将不同类型和类的对象进行同样对待的特性--不需要知道对象属于哪个类就能调用方法
4、封装：对象可以将他们内部状态隐藏(或封装)起来

- 在一些语言中，这意味着对象的状态只能对自己的方法可用
- 在 Python 中，所有的特性都是公开可用的，但是程序员应该在直接访问对象状态时谨慎行事

5、继承：一个类可以是一个类或多个子类的子类
6、接口和内省：一般来说，对于对象不用探讨过深，程序员可以靠多态调用自己需要的方法，不过如果想要知道对象到底有什么方法和特性，可以利用某些函数

Chapter 8. 异常

8.1. 什么是异常

1、Python 用异常对象来表示异常情况。
2、如果异常对象未被处理或捕捉，程序就会用所谓的回溯终止执行
3、事实上，每个异常都是一些类的实例，这些实例可以被引发，并且可以用很多种方法进行捕捉，使得程序可以捉住错误并且对其进行处理，而不是让整个程序失效

8.2. 按自己的方式出错

1、异常可以在某些东西出错时自动引

8.2.1. raise 语句

1、为了引发异常，可以使用一个类或者实例参数调用 raise 语句

```
raise Exception
raise Exception("Explaintion")
```

2、使用类时，程序会自动创建类的一个实例
3、内建的异常都可以在 exception 模块(和内建的命名空间)中找到，使用 dir 函数列出模块的内容
4、一些内建异常

- Exception：所有异常的基类
- AttributeError：特性引用或赋值失败时引发
- IOError：试图打开不存在的文件(包括其他情况)时引发
- IndexError：使用序列中不存在的索引时引发
- KeyError：在使用映射中不存在的键时引发
- NameError：在找不到名字(变量)时引发
- SyntaxError：在代码为错误形式时引发
- TypeError：在内建操作或者函数应用于错误类型的对象时引发
- ValueError：在内建操作或者函数应用于正确类型的对象，但是该对象使用不适合的值时引发
- ZeroDivisionError：在除法或者模除操作的第二个参数为 0 时引发

8.2.2. 自定义异常类

1、只要确保从 Exception 类继承(不管是间接的或者直接的)，也就是说继承其他内建的异常类也是可以的

- class SomeCustomoException(Exception):pass

8.3. 捕捉异常

1、可以用 try/except 语句来实现

```
try:
    #do something1
```

```
except Exception:
    #do something2
```

2、如果没有捕捉异常，它就会被传播到调用的函数中，如果在那里依然没有被捕获，这些异常就会浮到程序的最顶层，也就是说你可以捕获到其他人的函数中所引发的异常

3、如果想要重新引发捕获到的异常，那么可以调用不带参数的 raise 语句

8.4. 不止一个 except 子句

```
try:
    #do something1
except ZeroDivisionError:
    #do something2
except TypeError:
    #do something3
```

8.5. 用一个块捕获两个异常

1、如过需要用一个块捕捉多个异常，可以将它们作为元组列出

```
try:
    #do something1
except (ZeroDivisionError,TypeError,NameError):
    #do something2
```

8.6. 捕捉对象

1、如果希望 except 子句中访问异常对象本身，可以使用两个参数(注意，就算要捕捉多个异常，也只需要向 except 子句提供一个参数，即一个元组)，比如，想让程序继续运行，但是又因为某种原因想记录下错误，这个功能就很有用

2、Python3.0 中，except 子句可以写作

```
except (ZeroDivisionError,TypeError) as e:
    print (e)
```

8.7. 真正的全捕获

1、就算程序能处理好几种类型的异常，但有些异常还是会从眼皮底下溜走

2、可以在 except 子句中忽略所有的异常类，来捕获所有的异常

3、捕获所有异常是危险的，因为它会隐藏所有程序员未想到并且未做好准备处理的错误，它同样会捕获用户终止执行的[Ctrl]+C 企图，以及用 sys.exit 函数终止程序的企图等等

4、通常用 except Exception,e 会更好

8.8. 万事大吉

1、有些情况中，**没有坏事发生时执行一段代码是很有用的**，可以像对条件和循环语句那样，给 try/except 加一个 else 子句
2、在执行 try 语句没有发生异常时，才会执行 else 语句

8.9. 最后

1、finally 子句，用来在可能的异常后进行清理，finally 子句肯定会被执行，无论 try 子句中是否有异常发生

```
try:
    #do something1
except ZeroDivisionError:
    #do something2
else:
    #do something3
finally:
    #do something4
```

8.10. 异常和函数

1、异常和函数能很自然地在一起工作，如果异常在函数内引发而不被处理，它就会传播到函数调用的地方，直到被捕获或者到达主程序，如果主程序没有异常处理程序，程序会带着栈跟踪终止

8.11. 异常之禅

1、异常处理并不是很复杂，如果直到某段代码可能会导致某种异常，而又不希望程序以堆栈跟踪的形式终止，那么久根据需要添加 try/except 语句进行处理
2、比如试着访问一个对象的某个特性或者方法，可以使用 try/except 语句
3、在很多情况下，使用 try/except 语句比 if/else 会更自然一些("Python 化")，应该养成尽可能使用 try/exception 语句的习惯

Chapter 9. 魔法方法、属性和迭代器

1、在 Python 中，有的名称会在前面和后面都加上两个下划线

- 这种写法表示名字有特殊含义，所以绝不要在自己的程序中使用这种名字
- 在 Python 中，由这些名字组成的集合所包含的方法为魔法(特殊)方法
- 如果对象实现了这些方法中的一种，那么这个方法会在特殊的情况下(确切的说是根据名字)被 Python 调用，而几乎没有直接调用它们的必要

9.1. 准备工作

1、为了确保类是新型的，应该把赋值语句__metaclass__=type 放在你的模块的最开始，或者(直接或间接)子类化内建类(实际上是类型)object(或其他一些新式类)

```
class A(object):pass    #新式类
class B:pass    #旧式类
```

- 如果文件以__metaclass__=type 开头，那么上述两个都是新式类

2、**在 python3.0 中没有旧式的类，也不需要显式地子类化 object 或者将元类设置为 type，所有的类都会隐式地成为 object 的子类，如果没有明确的超类的话，就会直接子类化，否则间接子类化(Java 中所有的类类型都是继承自 Object)**

9.2. 构造方法

1、构造方法是一个很奇特的名字，它代表着类似于以前例子中使用过的那种名为 init 的初始化方法

2、构造方法和其他普通方法不同的地方在于，当一个对象被创建后，会立即调用构造方法

3、在 Python 中创建一个构造方法很容易，只要把 init 方法的名字从简单的 init 修改为魔法版本__init__即可

4、在 Python 所有的魔法方法中，__init__是使用最多的一个

5、Python 中有一个魔法方法叫做__del__，也就是析构方法

9.2.1. 重写一般方法和特殊方法的构造方法

1、每个类都可能拥有一个或多个超类，它们从超类哪里继承行为方式

2、如果一个方法在某类的实例中被调用，但在该类中没有找到该方法，那么久会去它的超类里面找

3、子类可以重写父类的方法，重写是继承机制中一个重要内容，对于构造方法尤其重要

4、构造方法用来初始化新创建对象的状态，大多数子类不仅要拥有自己的初始化代码，还要拥有超类的初始化代码

5、如果一个类的构造方法被重写，那么久需要调用超类的构造方法，否则对象可能不会被正确地初始化

9.2.2. 调用未绑定的超类构造方法

1、这里介绍的是历史遗留问题

2、在目前的版本中，使用 super 函数会更为简单明了

3、调用超类的构造方法

- **SuperClass**.__init__(**self**,args)　#self 必须与子类构造方法中的第一个参数名字相同，未必要是 self，**SuperClass 是一个类名，而不是实例**
- 在调用一个实例的方法时，该方法的 self 参数会被自动绑定到实例上，如果直接调用类的方法，那么就没有实例会被绑定，这样就可以自由地提供需要的 self 参数
- 这样的方法称为未绑定的方法

9.2.3. 使用 super 函数

1、它只能在新式类中使用，无论如何，你都应该使用新式类

2、当前的类和对象可以作为 super 函数的参数使用，调用函数返回的对象的任何方法都是调用超类的方法，而不是当前类的方法

super(SubClass,self).func()　　**#SubClass 是当前类，self 是当前类的一个对象**

3、在 Pyton3.0 中，super 可以不带任何参数进行调用，功能仍然具有魔力

4、super 函数比在超类中直接调用未绑定方法更直观，super 是很智能的，即使类已经继承多个超类，也只需要一次 super 函数调用

5、实际上 super 返回一个对象，这个对象负责进行方法解析，当对其特性进行访问时，它会查找所有的超类，直到找到所需的特性为止(或者引发 AtrributeError 异常)

9.3. 成员访问

1、基本的序列和映射的规则很简单，但如果要实现它们全部功能就需要实现很多魔法函数

2、规则

- 规则这个词在 Python 中会常用，用来描述管理某种形式的行为的规则
- 规则与接口概念类似，规则说明了应该实现何种方法和这些方法应该做什么
- ***Python 中的多态性是基于对象的行为的(而不是基于祖先，例如它所属的类或超类等等)，在其他的语言中对象可能被要求属于某一个类，或者被要求实现某个接口，但在 Python 中，只是简单地要求它遵循几个给定的规则***

9.3.1. 基本的序列和映射规则

1、序列和映射是对象的集合，为了实现他们的行为(规则)，如果对象是不可变的，那么需要使用 2 个魔法方法，如果是可变的则需要使用 4 个

- __len__(self)：这个方法应该返回集合中所含项目的数量
  - 对于序列来说，就是元素的个数
  - 对于映射来说，则是键值对
  - 如果__len__返回 0，并且没有实现重写该行为的__nonzero__，对象会被当做一个布尔变量中的假值(空的列表，元组，字符串和字典也一样)进行处理
- __getitem__(self,key)：这个方法返回与所给键对应的值
  - 对于一个序列，键应该是一个 0~n-1 的整数，n 是序列长度

- 对于映射来说，可以使用任何种类的键
  - __setitem__(self,key,value)：这个方法应该按一定的方式存储和 key 相关的 value，该值随后可以使用__getitem__来获取
  - __delitem__(self,key)：这个方法在对一部分对象使用 del 语句时被调用，同时必须删除和键相关的值，这个方法也是为可修改的对象定义的

2、对以上方法的附加要求如下：
- 对于一个序列来说，如果键是负整数，要么从末尾开始计数，换句话说就是 x[-n]和 x[len(x)-n]是一样的
- 如果键是不合适的类型(例如，对序列使用字符串作为键)，会引发一个 TypeError 异常
- 如果序列的索引是正确的类型，但超出了范围，应该引发 IndexError 异常

3、分片操作也是可以模拟的，当支持__getitem__方法的实例进行分片操作时，分配对象作为键提供

4、**Python2.5 有一个方法叫做__index__，它允许你在分片中使用非整型限制**

5、P144-P146 的例子，体会魔法!!!

9.3.2. 子类化列表，字典和字符串

1、实现所有这些规则(魔法方法)是很繁重的工作

2、如果类的行为和默认的内建类的行为很接近，可以子类化内建类型

3、如果希望是实现一个和内建列表相似的序列，可以子类化 list 来实现

9.4. 更多魔力

9.5. 属性

1、访问器方法：将改变属性的操作封装成方法
- 名字如 getSize,setSize 的方法

2、为很多简单的特性写访问器方法是很不现实的

3、Python 能隐藏访问器方法，让所有特性看起来一样，这些通过访问器定义的特性被称为属性

4、实际上在 Python 中有两种创建属性的机制，主要讨论新的机制--只在新式类中使用的 property 函数

9.5.1. property 函数

```
class Rectangle:
    def __init__(self):
        self.width=0
        self.height=0
    def setSize(self,size):
        self.width,self.height=size
    def getSize(self):
        return self.width,self.height
    size=property(getSize,setSize)
```

1、随后 size 就可以作为特性进行访问了，实际上还是依赖于 getSize 和 setSize
2、实际上 property 函数可以用 0、1、2、3、4 个参数来调用
- 如果没有参数，产生的属性既不可读，也不可写
- 如果只有一个参数调用(一个取值方法)，产生的属性是只读的
- 第三个参数(可选)是一个用于删除特性的方法(它不要参数)
- 第四个参数(可选)是一个文档字符串
- property 的四个参数分别为 fget、fset、fdel、doc

3、在新式类中应该使用 property 函数而不是访问器方法
4、property 如何实现
- 实际上 property 不是一个真正的函数，它是其实例拥有很多特殊方法的类，也正是那些方法完成了所有的工作
- 涉及的方法是：
  - __get__
  - __set__
  - __delete__
  - 这三个方法合在一起，就定义了**描述符规则**，实现了任何一个方法的对象就叫**描述符**

9.5.2. 静态方法和类成员方法

1、使用@staticmethod 与@classmethod，在方法或函数的上方将装饰器列出，从而指定一个或多个装饰器

9.5.3. __getattr__、__setattr__和它的朋友们

1、拦截对象的所有特性访问时不可能的
2、为了在访问特性的时候可以执行代码，必须使用一些魔法方法
- __getattribute__(self,name)：当特性 name 被访问时自动被调用(只能在新式类中使用)
- __getattr__(self,name)：当特性 name 被访问且对象没有相应的特性时被自动调用
- __setattr__(self,name,value)：当试图给特性 name 赋值时会被自动调用
- __delattr__(self,name)：当试图删除特性 name 时被自动调用
- 尽管与使用 property 函数相比有点复杂，而且在某些方面效率更低，但是这些特殊方法是很强大的，因为可以对处理很多属性的方法进行再编码

3、__setattr__方法需要避免死循环，利用__dict__来代替普通特性的操作
4、**还有一个陷阱与__getattribute__有关：因为__getattribute__会拦截所有特性的访问(新式类中)，包括对__dict__的访问，因此，访问__getattribute__中与 self 相关的特性时，使用超类的__getattribute__方法(super 函数)是唯一安全的途径???**

9.6. 迭代器

1、魔法方法：__iter__

9.6.1. 迭代器规则

1、可以对任何实现了__iter__方法的对象进行迭代(有点类似于 Java 的接口的意思???)

2、__iter__方法返回一个迭代器

- 迭代器就是具有__next__方法的对象
- 在调用__next__方法时，迭代器会返回它的下一个值
- 如果__next__方法被调用，但迭代器没有值可以返回，就会引发 StopIteration 异常
- **在 Python3.0 中，迭代器对象应该实现__next__方法而不是 next**

3、为什么不使用列表？

- 因为列表的杀伤力太大，如果有很多值，列表就会占用太多的内存
- 使用迭代器更优雅，更简单

4、一个实现了__iter__方法的对象是可迭代的，一个实现了__next__方法的对象则是迭代器

5、**内建函数 iter 可以从可迭代的对象中获取迭代器**

- it=iter([1,2,3])
- it.__next__()　　==> 1

9.6.2. 从迭代器得到序列

1、除了在迭代器和可迭代对象进行迭代外，还能把它们转换为序列

2、在大部分能使用序列的情况下(除了在索引或分片等操作中)，都能使用迭代器(或者可迭代对象)替换

3、**使用 list 构造方法显式地将迭代器转化为列表(会调用迭代器遍历一遍，然后生成一个列表)**

- it=iter([1,2,3])
- list(it)　　==> [1,2,3]

9.7. 生成器

1、生成器是 Python 新引入的概念，由于历史原因，它也叫简单生成器

2、**生成器和迭代器是近几年来引入的最强大的两个特性，生成器的概念要更高级一些**

3、**生成器是一种用普通的函数语法定义的迭代器**

9.7.1. 创建迭代器

1、任何包含 yield 的语句的函数称为生成器

2、生成器的行为和普通函数有很大区别，在于它不像 return 那样返回值，而是每次产生多个值，每产生一个值，函数就会被冻结：即函数停在那点等待被重新唤醒(线程???)

3、**使用 list 构造方法显式地将生成器转化为列表**

4、循环生成器

- Python2.4 引入了列表推导式的概念 5.6
- 生成器推导式和列表推导式工作方式类似，只不过返回的不是列表而是生成器(并且不会立刻进行循环，允许你调用 next()方法)

- 与列表推导式不同，生成器推导式使用圆括号
- 更巧妙的地方在于，生成器推导式可以在当前的圆括号内使用
  - sum(i**2 for i in range(10))

9.7.2. 递归生成器

1、每层嵌套需要增加一个 for 循环，但是不知道有几层嵌套，可以加入递归，使得解决方案更灵活

```
def flatten(nested):
    try:
        for sublist int nested:
            for element in flatten(sublist):   #迭代情况
                yield element
    except TypeError:
        yield nested   #基本情况
```

- 当 flatten 被调用时，有两种可能性：基本情况和需要递归的情况
- 在基本情况中，函数被告知展开一个元素(比如一个数字)，这种情况下，for 循环会引发一个 TypeError 异常(因为试图对一个数字进行迭代)，生成器会产生一个元素

2、这样做只有一个问题，如果 nested 是一个类似于字符串对象(字符串，Unicode、UserString)等等，那么它就是一个序列，不会引发 TypeError

- 首先需要实现的是将类似于字符串的对象当成原子值，而不是当成被展开的序列
- 对这些类进行迭代实际上会导致无穷递归，因为一个字符串的第一个元素是另一个长度为 1 的字符串，而长度为 1 的字符串的第一个元素就是字符串本身(问题出在：字符串的元素仍然是字符串，维度没有下降，仍是一个可迭代的对象)

```
def flatten(nested):
    try:
        try:nexted+' '
        except TypeError:pass
        else:raise TypeError
        for sublist int nested:
            for element in flatten(sublist):   #迭代情况
                yield element
    except TypeError:
        yield nested   #基本情况
```

- 红色部分是检查一个字符串最简单最快速的方法

9.7.3. 通用生成器

1、生成器是一个包含 yield 关键字的函数，当它被调用时，在函数体中的代码不会被执行，而会返回一个迭代器，每次请求一个值，就会执行生成器中的代码，直到遇到一个 yield 或者 return 语句(线程的挂起与唤醒)，其中 return 语句意味着生成器要停止执行

2、生成器是由两部分组成的：生成器的函数和生成器的迭代器

- 生成器的函数是用 def 语句定义的，包含 yield 的部分
- 生成器的迭代器是这个函数的返回的部分
- 生成器的函数返回的迭代器可以像其他迭代器那样使用

9.7.4. 生成器方法???

1、生成器的新特征(Python2.5 中引入)是在开始运行后为生成器提供值的能力，表现为生成器和"外部世界"进行交流的取到

- 外部作用域访问生成器的 send 方法，就像访问 next 方法一样，只不过前者使用一个参数(要发送的"消息"--任意对象)
- 在内部则挂起生成器，yield 现在作为表达式而不是语句使用，当生成器重新运行的时候，yield 方法返回一个值，也就是外部通过 send 方法发送的值，如果 next 方法被使用，则 yield 方法返回 None
- 注意，使用 send 方法只有在生成器挂起之后才有意义，如果在此之前需要给生成器提供更多信息，那么只需要使用生成器函数的参数

9.7.5. 模拟生成器

1、生成器在旧版本中是不可用的，本节介绍模拟生成器的机制来写函数

2、步骤

- 在函数体的开始处放置：result=[]
- 将 result.append(some_expression)替换掉 yield some_expression
- 在末尾添加 return result

9.8. 八皇后的问题

9.8.1. 生成器和回溯

1、生成器是逐渐产生结果的复杂递归算法的理想实现工具

- 没有生成器的话，算法就需要一个作为额外参数传递的半成品方案，这样递归调用就可以在这个方案上建立起来
- 如果使用生成器，那么所有的递归调用只要创建自己的 yield 部分

9.8.2. 问题

1、一个棋盘和 8 个皇后，皇后之间不能形成威胁，必须把它们放置在每个皇后都不能吃掉其他皇后的状态

2、典型的做法就是回溯

Chapter 10. 自带电池

- Python 的标准安装中还包括一组模块，称为标准库

10.1. 模块

- import 从外部模块获取函数并为自己的程序所用

10.1.1. 模块是程序

1、任何 Python 程序都可以作为模块导入
   - import sys
   - sys.path.append('该 Python 程序的绝对路径')
   - import 'Python 程序名字'

2、在模块导入的时候，其中的代码被执行了，再次导入该模块，就什么都不会发生了

3、导入模块并不意味着导入时执行某些操作(如打印文本)，它们主要用于定义，比如变量，函数和类等。这些东西只需要定义一次，**导入模块多次和导入一次的效果是一样的**

4、为什么只导入一次：
   - 这种只导入一次的行为在大多数情况下是一种实质性优化，对于两个模块相互导入尤其重要(C++头文件相互包含是一种严重的设计错误)
   - 两个模块相互导入，如果不是只导入一次，那么会造成无限递归
   - 如果坚持要重新载入模块，可以使用内建的 reload 函数，它带有一个参数(要导入的模块)，并且返回重新载入的模块(Python3.0 中已经删除了 reload)

10.1.2. 模块用于定义

1、模块在第一次导入到程序中时被执行

2、在模块中定义函数
   - 函数在模块的作用域内被定义了，因此可以用 ModuleName.func(...)来调用函数
   - 可以通过同样的方式来使用任何在模块的全局作用域中定义的名称

3、把代码放在模块中是为了重用，避免在主程序中重写不必要的代码

4、在模块中增加测试代码
   - 模块被用来定义函数，类和其他一些内容，但有时候，在模块中添加一些检查模块本身是否正常工作的测试代码是很有用的
   - 但是又不想让测试代码在被其他程序导入的时候执行，需要借助__name__变量：
     - 在主程序(包括解释器的交互式提示符在内)中，变量__name__的值是'__main__'
     - 在导入的模块中，这个值就被设定为模块的名称
     - if __name__=='__main__':test()
     - 如果将该模块作为程序运行，那么 test 就会执行，而作为模块导入时，它的行为就会像普通模块一样

10.1.3. 让你的模块可用

1、我们可以通过手动改变 sys.path，其包含了(字符串组成的)一个目录列表，解释器在该列表中查找模块(差不多环境变量的意思)

2、我们还可以让 sys.path 一开始就包含正确的目录

- 其一是将模块放置在合适的位置
- 其二是告诉解释器去哪里查找需要的模块

3、将模块放在正确的位置

- 目录的列表可在 sys 模块中的 path 变量中找到
  - import sys,pprint
  - pprint.pprint(sys.path)
- 每个字符串都提供了一个放置模块的目录，解释器从这些目录中找到所需的模块，尽管这些目录都可以用，但是 **site-packages** 目录是最佳的，因为它就是用来做这些事情的

4、告诉编译器去哪里找

- 这个解决方案对于以下几种情况并不适用：
  - 不希望自己的模块填满 Python 解释器的目录
  - 没有在 Python 解释器目录中存储文件的权限
  - 想将模块放在其他地方
- 想将模块放在其他地方，就要告诉编译器去哪里找
  - 方法一：编辑 sys.path，但这不是通用方法，标准的实现方法是在 PYTHONPATH 环境变量中包含模块所在的目录
  - PYTHONPATH 的内容会因操作系统不同而有所差异

5、环境变量

- 环境变量并不是 Python 解释器的一部分，它们是操作系统的一部分，基本上，它们相当于 Python 变量，不过是在 Python 解释器外设置的
- 在 UNIX 系统中，主要在~/.barhrc 中设置
  - export PYTHONPATH=$PYTHONPATH:[模块放置的目录]
- 在 Window 就手动设置环境变量即可

6、命名模块

- 包含模块代码的文件的名字要和模块名一样，再加上.py 扩展名

10.1.4. 包

1、为了组织好模块，你可以将他们分组为包，包基本上就是另外一类模块，有趣的地方就是它们能包含其他模块

2、当模块存储在文件中时，包就是模块所在的目录，为了让 Python 将其作为包对待，它必须包含一个命名为__init__py 的文件(模块)

3、为了将模块放置在包内，直接把模块放在包目录内即可

4、简单的包布局(假设包名为 drawing)

文件/目录	描述
~/python/	PYTHONPATH 中的目录
~/python/drawing/	包目录
~/python/drawing/__init__.py	包代码
~/python/drawing/module1.py	module1 模块
~/python/drawing/module2.py	module2 模块

10.2. 探究模块

10.2.1. 模块中有什么

1、探究模块最直接的方式就是在 Python 解释器中研究它们，要做的第一件事就是导入它们

2、使用 dir

- 查看模块包含的内容可以使用 dir 函数，它会将对象的所有特性(以及模块的所有函数、类、变量等)列出
- [n for n in dir(copy) if not n.startswith('_')]

3、__all__变量

- 它是在模块内部被设置的，定义了模块的共有接口，它告诉解释器，从模块导入所有名字代表什么含义
- 使用 from copy import *只会导入__all__变量包含的名字
- 如果没有设定__all__，使用 import *会默认将导入模块中所有不以下划线开头的全局名称

10.2.2. 用 help 获取帮助

1、用法

- help(copy.copy)

10.2.3. 文档

1、__doc__

- func.__doc__查看模块或函数的文档字符串

2、并非每个模块和函数都有不错的文档字符串，有时候可能需要十分透彻地描述这些模块和函数是如何工作的，学习 Python 编程最有用的文档莫过于 **Python 库参考**，它对所有标准库中的模块都有描述，库参考可以在线浏览 **(http://python.org/doc/lib)**

10.2.4. 使用源代码

1、如何查找源代码

- func.__file__
- module.__file__

10.3. 标准库：一些最爱

10.3.1. sys

1、sys 这个模块让你能够访问 Python 解释器联系紧密的变量和函数

2、sys 模块中重要的函数和变量

- argv：命令行参数，包括脚本名称
- exit([arg])：退出当前的程序，可选参数为给定的返回值或者错误信息
- modules：映射模块名字到载入模块的字典
- path：查找模块所在目录的目录名列表
- platform：类似 sunos5 或者 win32 平台标识符
- stdin：标准输入流---一个类文件对象

- stdout：标准输出流---一个类文件对象
- stderr：标准错误流---一个类文件对象

10.3.2. os

1、os 模块提供了访问多个操作系统服务的功能
2、os 模块中重要的函数和变量
- environ：对环境变量进行映射
- system(command)：在子 shell 中执行操作系统命令
- sep：路径中的分隔符
- pathsep：分隔路径的分隔符
- linesep：行分隔符('\n','\r',or '\r\n')
- urandom(n)：返回 n 字节的加密强度随机数

3、对于 Windows 系统，有一个特有的函数 os.startfile：
- os.startfile(r'c:\Program Files\Mozilla Firefox\firefox.exe')来代替 os.system
- Windows 下必须使用反斜线哦

4、大多数情况下 os.system 很有用，但是对于启动浏览器这样特定的任务来说，还有更好的解决办法：webbrowser 模块
- import webbrowser
- webbrowser.open('http://www.python.org')

10.3.3. fileinput

1、fileinput 模块让你能够轻松遍历本文件的所有行，可以通过如下方式调用脚本
- python some_scripy.py file1.txt file2.txt file3.txt ...
- cat file.txt | python some_script.py

2、fileinput 模块中重要的函数
- input(files[,inplace[,backup]])：便于遍历多个输入流中的行
- filename()：返回当前文件的名称
- lineno()：返回当前(累计)的行数
- filelineno()：返回当前文件的行数
- isfirstline()：检查当前行是否是文件的第一行
- isstdin()：检查最后一行是否来自 sys.stdin
- nextfile()：关闭当前文件，移动到下一个文件
- close()：关闭序列

3、fileinput.input 是最重要的函数，他会返回能够用于 for 循环遍历的对象，如果不想使用默认行为(fileinput 查找需要循环遍历的文件)，那么可以给函数提供(序列形式的)一个或多个文件名，还能将 inplace 参数设置为真值(inplace=True)以进行原地处理

10.3.4. 集合、堆和双端队列

1、集合
- 集合在 Python2.3 才引入，尽管可以在现在的代码中创建 Set 实例，但是除非想要兼容以前的程序，否则没有必要这样做
- 在 Python2.3 中，集合通过 set 类型的实现称为了语言的一部分，这意味着不需要导入 sets 模块了，直接创建集合即可

- 集合主要用于检查成员资格
- 与字典一样，集合元素的顺序是随意的
- 对于集合可以使用'&'与'|'表示交集与并集，或者使用方法 a.union(b)或a.intersection(b)
- 基本方法：add 与 remove

2、堆

- 它是优先队列的一种，使用优先队列能够以任意顺序增加对象，并且能在任何时间找到最小元素，比列表的 min 方法有效的多
- Python 并没有独立的堆类型，只要一个包含一些堆操作函数的模块，这个模块叫做 heapq(q 是 queue 的缩写，即队列)
- heapq 模块中重要的函数
  - heappush(heap,x)：将 x 入堆
  - heappop(heap)：将堆中最小元素弹出
  - heapify(heap)：将 heap 属性强制应用到任意一个列表
  - heapreplace(heap,x)：将堆中最小元素弹出，同时将 x 入堆
  - nlargest(n,iter)：返回 iter 中第 n 大元素
  - nsmallest(n,iter)：返回 iter 中第 n 小元素

3、双端队列(以及其他集合类型)

- 双端队列在需要按照元素增加的顺序来移除元素时非常有用
- Python2.4 增加了 collections 模块，它包括 deque 类型
- 双端队列能够有效地在开头增加和弹出元素，这是列表中无法实现的
- 双端队列能够有效地旋转元素
- 双端队列对象有 extend 和 extendleft 方法，extendleft 类似于 appendleft

10.3.5. time

1、time 模块包括的函数能够实现：获取当前时间，操作时间和日期，从字符串读取时间以及格式化时间为字符串

2、日期可以用实数(从"新纪元"的 1 月 1 日 0 点开始计算到现在的秒数)，新纪元是一个与平台相关的年份，对 UNIX 来说是 1970 年)

3、time 模块中重要的函数

- asctime([tuple])：将时间元组转换为字符串
- localtime([secs])：将秒数转换为日期元组，以本地时间为准
- mktime(tuple)：将时间元组转换为本地时间
- sleep(secs)：休眠(不做任何事情)secs 秒
- strptime(string[,format])：将字符串解析为时间元组
- time()：当前时间(新纪元开始后的秒数，以 UTC 为准)

4、时间元组字段的意义：

索引	字段	值
0	年	2000，2016 等
1	月	1-12
2	日	1-31
3	时	0-23
4	分	0-59
5	秒	0-61

6	周	当周一为 0 时，0-6
7	儒历日	1-366
8	夏令时	0、1、-1

- 秒的范围 0-61 是为了应付闰秒与双闰秒
- 夏令时的数字是布尔值，如果使用了-1，mktime 就会工作正常

10.3.6. random

1、random 模块包括返回随机数的函数，可以用于模拟或者用于任何产生随机输出的程序

2、事实上，所产生的数字都是伪随机数，也就是说它们在有限时间内看起来是随机的，实际上，它们以一个可预测的系统作为基础

3、如果需要真的随机性，应该使用 os 模块的 urandom 函数，random 模块内的 SystemRandom 类也是基于同种功能，可以让数据接近真正的随机性

4、模块中一些重要的函数

- random()：返回 0<n≤1 之间的随机实数
- getrandbits(n)：以长整型形式返回 n 个随机位
- uniform(a,b)：返回随机实数 n，其中 a≤n<b
- randrange([start],stop,[step])：返回 range(start,stop,step)中的随机数
- choice(seq)：从序列 seq 中返回随意元素
- shuffle(seq[,random])：原地指定序列 seq
- sample(seq,n)：从序列 seq 中选择 n 个随机且独立的元素

10.3.7. shelve

1、如果只需要一个简单的存储方案，那么 shelve 模块可以满足你大部分的需要，你所要做的只是为它提供文件名

2、shelve 中唯一有趣的函数是 open，在调用它的时候(使用文件名作为参数)，它会返回一个 Shelf 对象，可以用它来存储内容，只需要把它当成普通的字典，键一定要是字符串，完成工作后调用 close 方法

3、潜在的陷阱

- shelve.open 函数返回的对象并不是普通的映射 P188

10.3.8. re

1、re 模块包含对正则表达式的支持

2、什么是正则表达式：

- 正则表达式是可以匹配文本片段的模式，最简单的正则表达式就是普通字符串，可以匹配自身
- 通配符：正则表达式可以匹配多余一个的字符，例如点号'.'可以匹配除了换行符之外的任何单个字符
- 对特殊字符进行转义：在正则表达式中如果将特殊字符作为普通字符使用是会遇到问题的
- 注意：为了获得 re 模块所需的单个反斜线，我们要在字符串中使用两个反斜线，1：通过解释器转义。2：通过 re 模块转义
  - 'python\\.org'

- r'python\.org'　<==使用原始字符串
- **字符集**：字符集可以匹配它所包括的任意字符
- **选择符和子模式**：利用管道符号'|'(选择符)，以及圆括号'()'(子模式)
  - python|perl
  - p(ython|erl)
- **可选项和重复子模式**：在子模式后面加上问号，它就变成了可选项，它可能出现在匹配字符串中，但并非必须的
- **字符串的开始和结尾**：行首(^)，行尾($)

3、re 模块的内容

- 重要函数
  - compile(pattern[,flags])：根据包含正则表达式的字符串创建模式对象
  - search(pattern,string[,flags])：在字符串中寻找模式
  - match(pattern,string[,flags])：在字符串的开始处匹配模式
  - split(pattern,string**[,maxsplit=0]**)：根据模式的匹配项来分隔字符串
  - findall(pattern,string)：列出字符串中模式的所有匹配项
  - sub(pat,repl,string[,count=0])：将字符串中所有 pat 的匹配项用 repl 替换
  - escape(string)：将字符串中所有特殊正则表达式字符转义

4、匹配对象和组

- 对于 re 模块中那些能够对字符串进行模式匹配的函数而言，当能找到匹配项的时候，它们都会返回 MatchObject 对象
- 这些对象包括匹配模式的子字符串的信息，它们还包含了哪个模式匹配了字符串哪部分的信息，这些部分叫做组
- 简而言之，组就是放置在圆括号之内的子模式，**组的序号取决于它左侧的括号数，组 0 就是整个模式**
  - 'There (was a (wee) (cooper)) who (lived in Fyfe)'
  - 0 There was a wee cooper who lived in Fyfe
  - 1 was a wee cooper
  - 2 wee
  - 3 cooper
  - 4 lived in Fyfe
- **一般来说，如果组中包含诸如通配符或者重复运算符之类的特殊字符，你可能会对是什么鱼给定组实现了匹配感兴趣**
- re 匹配对象的重要方法
  - group([group1,...])：获取给定子模式(组)的匹配项
  - start([group])：返回给定组的匹配项的开始位置
  - end([group])：返回给定组的匹配项的结束位置(和分片一样，不包括组的结束位置)
  - span([group])：返回一个组的开始和结束位置
  - **以上函数如果不指定组号，则默认为组 0**
  - 除了整体匹配外，我们只能使用 99 个组，1-99

5、作为替换的组号和函数

- 见证 re.sub 强大功能的最简单方式就是在替换字符串中使用组号，在替换内容中，以'\\n'(或 r'\n')形式出现的任何转义序列都会被模式中与组 n 匹配的字符串替换掉

- 例子：
  - emphasis_pattern=re.compile(r'''
    *
    (
    [^*]+
)
    *
    ''', **re.VERBOSE**)
  - re.sub(emphasis_pattern,r'**/1**','Hello, *world*!')
  - ==> 'Hello, world!'
  - **注意：让正则表达式变得更加易读的方式是在 re 函数中使用 VERBOSE 标志，它允许在模式中添加空白(空白字符，tab，换行符，等等)，re 则会忽略他们，除非将其放在字符类或者用反义斜线转义)**
- 重复运算符默认是贪婪的，它会尽可能包括所有的内容

Chapter 11. 文件和流

- 到目前为止，我们的程序与外部交互只是通过 input、raw_input 和 print 函数，与外部的交互很少

11.1. 打开文件

1、open 函数用来打开文件

open(name[,mode[,buffering]])

- open 函数使用一个**文件名作为唯一的强制参数**，然后返回一个文件对象，**模式**和**缓冲**参数都是可选的

11.1.1. 文件模式

1、如果 open 函数只带一个文件名参数，那么我们可以获得能读取文件内容的文件对象，如果要向文件内写入内容，则必须提供一个模式参数来显式声明

2、模式参数：

- r：读模式
- w：写模式
- a：追加模式
- b：二进制模式(可添加到其他模式中使用)
- +：读写模式(可添加到其他模式中使用，如'r+')

3、为什么使用二进制模式

- 用二进制模式读写一般文件的话，与文本模式没有太大的区别
- 由于在文本模式 Python 会对换行符进行自动地转换，以实现跨平台的功能，例如在 Windows 下换行符是'\r\n'，在 UNXI 平台下换行符是'\n'，Python 会自动对其进行转换
- 对于二进制文件，在文本模式下，可能会自动进行某些转换，因为那些字节恰好被翻译成换行符，而事实上，完全不是这样解析该二进制文件的，从而会产生出入，使用二进制模式，避免上述自动转换的发生

11.1.2. 缓冲

1、open 函数的第三个参数控制着文件的缓冲，如果参数是 0(或者是 False)，I/O 就是无缓冲的(所有的读写都是针对硬盘)，如果参数是 1(或者 True)，I/O 就是有缓冲的(意味着 Python 使用内存来代替硬盘，让程序更快，只有使用 flush 或者 close 时才会更新硬盘上的数据)

2、大于 1 的数字代表缓冲区的大小(单位字节)，-1(或者任何负数)代表使用默认的缓冲区大小

11.2. 基本的文件方法

1、类文件：类文件对象是支持一些 file 类方法的对象，最重要的是支持 read 方法或者 write 方法，或者两者兼有

2、三种标准的流：

- sys.stdin：数据输入流

- sys.stdout：数据输出流，能使用管道连接到其他程序的标准输入中
- sys.stderr：错误输出流

11.2.1. 读和写

1、文件(或流)最重要的能力就是提或者接受数据。
2、每次调用 file.writ()方法，所提供的参数 string 会被追加到文件中已存在部分的后面
3、完成了对一个文件操作时，调用 close
4、读取：file.read()方法，只要记得告诉流要读多少字符即可，不指定将会读取全部内容，转为 string，其中换行符存为 r'\n'(与 Java 中不同，Java 面向字符的 Reader.read()每次只能读取一个字符,InputStream.read()每次读取一个字节，InputStream.available()能检查还有多少个字节可读)

11.2.2. 管式输出

1、在 UNIX 的 shel 中，使用管道可以在一个命令后面续写其他的多个命令
2、管道符号'|'将一个命令的标准输出和下一个命令的标准输入连在一起
3、例如 cat somefile.txt | python3.5 somescript.py
4、随机访问：可以使用类文件对象的方法 seek 和 tell 直接访问感兴趣的部分
- seek(offset[,whence]):这个方法把当前位置(进行读和写的位置)移动到由 offset 和 whence 定义的位置
- offset 是一个字节(字符)数，表示偏移量
- whence 默认是 0，表示偏移量是从文件头开始计算的的(偏移量必须是非负的)，whence 可能被置为 1，此时 offset 可以是负的
- tell 方法返回当前文件的位置

11.2.3. 读写行

1、读行
- file.readline()读取单独的一行(从当前位置开始直到一个换行符的出现，也读取这个换行符)。不使用任何参数可以读取一整行，或者使用非负整数限定可以读取的字符(或字节)的最大值
- file.readlines()方法可以读取一个文件中的所有行并将其转为列表返回

2、写行
- writelines 方法和 readlines 方法相反，传给它一个字符串的列表(或可迭代对象)，它会把所有的字符串写入文件(连续写入，不会添加换行符的！)，
- 当然用 file.writelines(file.readlines())会得到原来的内容，**因为读入行的时候，连换行符也会读入(这一点与 Java 不同，Java 会读取换行符，但是转存为 Stirng 的时候，会将换行符去掉)**
- 没有 writeline 方法

11.2.4. 关闭文件

1、写入过的文件总是应该关闭的，因为 Python 可能会缓存写入的数据，如果程序因为某些原因崩溃了，那么数据根本不会被写入文件中，为了安全起见，要在使用完文件后关闭
2、如果想确保文件被关闭了，应该使用 try/finally 语句，并且在 finally 子句中调

用 close 方法
3、with 语句

```
with open("somefile.txt") as somefile:
    do_something(somefile)
```

- with 语句可以打开文件并且将其赋值到变量上，之后就可以将数据写入语体重的文件，文件在语句结束后会被自动关闭，即使是由异常引起的结束也是如此
- 在 Python2.5 以后，with 可以直接使用
- with 语句实际上是很通用的结构，允许使用所谓的上下文管理器，上下文管理器是一种支持__enter__和__exit__这两种方法的对象
- __enter__方法不带参数，它在进入 with 语句块的时候被调用，返回值绑定到 as 关键字之后的变量
- __exit__方法带有 3 个参数，异常类型，异常对象，和异常回溯，在离开方法(带有通过参数提供的，可以引发的异常)时，这个函数被调用，如果__eixt__返回 false，那么所有异常都不会被处理
- 文件可以被作用上下文管理器，它们的__enter__方法返回文件对象本身，__exit__方法关闭文件

11. 2. 5. 使用基本文件方法

1、基本方法

- file.read()
- file.readline()
- file.readlines()
- file.write()
- file.writelines()

11. 3. 对文件内容进行迭代

1、对文件内容进行迭代以及重复执行一些操作，是最常见的文件操作之一

11. 3. 1. 按字节处理

1、最常见的对文件内容进行迭代的方法是在 while 循环中使用 read()方法，例如对每个字符(字节)进行循环
2、可以将 file.read(1)所读取的字符串作为判断条件，如果到达了文件尾，则会返回一个空字符串，此时条件为假，其余时候为真

11. 3. 2. 按行操作

1、与字符处理一样的方式，用 while 循环中使用 readline()方法
2、可以将 file.readline()所读取的字符串作为判断条件，如果到达了文件尾，则会返回一个空字符串，此时条件为假，其余时候为真

11. 3. 3. 读取所有内容

11.3.4. 使用 fileinput 实现懒惰行迭代

1、在需要一个非常大的文件进行迭代操作时，readlines 会占用太多的内存，这个使用可以使用 while 循环和 readline 方法来代替

2、还可以使用 for 循环使用一个名为懒惰行迭代的方法：说它懒惰是因为它只是读取实际需要的文件部分???

11.3.5. 文件迭代器

1、Python 可以直接对文件进行迭代

```
for line in open(filename):
    process(line)
```

- 迭代是以行为单位的

2、可以对文件迭代器执行和普通迭代器相同的操作，例如 list(open(filename))，转化为一个行组成的列表

Chapter 12.图形用户界面

- 本章介绍最成熟的跨平台 PythonGUI 工具包

12.1. 丰富的平台

1、Tkinter
2、wxpython
3、PythonWin
4、Java Swing
5、PyGTK
6、PyQt

12.1.1. 下载和安装 wxPython

Chapter 13. 数据库支持

13.1. Python 数据库编程接口

13.1.1. 全局变量

1、任何支持 2.0 版本 DB API 的数据库模块都必须定义 3 个描述模块特性的全局变量

- apilevel：所使用的 Python DB API 版本
- threadsafety：模块的线程安全等级
- paramstyle：在 SQL 查询中使用的参数风格

2、API 级别(apilevel)

- 这是个字符串常量，提供正在使用的 API 版本号
- 对于 DB API 2.0 版本来说，其值可能是'1.0'和'2.0'
- 若这个变量不存在，那么该模块就不兼容 2.0 版本的 API，只能假定当前使用的是 1.0 版本的 API

3、线程安全性等级

- 这是个取值范围 0-3 的整数
- 0 表示线程完全不共享模块
- 3 表示是完全线程安全的
- 1 表示线程本身可以共享模块，但不对连接共享

4、参数风格(paramstyl)

- 表示在执行多次类似查询的时候，参数是如何被拼接到 SQL 查询中的
- 值'format'表示标准的字符串格式化(使用基本的格式代码)，可以在参数中进行拼接的地方插入%s
- 值'pyformat'表示扩展的格式代码，用于字典拼接中，比如%(foo)

13.1.2. 异常

1、API 中定义了一些异常类，以便尽可能进行错误处理

2、在 DB API 中使用的异常

异常	超类	描述
StandardError		所有异常的泛型基类
Warning	StandardError	在非致命错误发生时引发
Error	StandardError	所有错误条件的泛型超类
InterfaceError	Error	关于接口而非数据库的错误
DatabaseError	Error	与数据库相关的错误的基类
DataError	DatabaseError	与数据相关的问题，比如值超出范围
OperationalError	DatabaseError	数据库内部操作错误
IntegrityError	DatebaseError	关系完整性收到影响，比如键检查失败
InternalError	DatabaseError	数据库内部错误，比如非法游标
ProgrammingError	DatabaseError	用户编程错误，比如未找到表
NotSupportedError	DatabaseError	请求不支持的特性(比如回滚)

13. 1. 3.　　连接和游标

1、为了使用底层的数据库系统，首先必须连接到它，这个时候就需要在适当的环境下使用具名函数 connect，该函数有多个参数，而具体使用哪个参数取决于底层数据库的类型

2、connect 函数的常用参数

参数名	描述	是否可选
dsn	数据源名称	否
user	用户名	是
password	用户密码	是
host	主机名	是
database	数据库名	是

3、connect 函数返回连接对象，这个对象表示目前和数据库的会话

4、连接对象方法

方法名	描述
close()	关闭连接之后，连接对象和它的游标均不可用
commit()	如果支持的话就提交挂起的事务，否则不做任何事
rollback()	回滚挂起的事务(可能不可用)
cursor()	返回连接的游标对象

- rollback 方法可能不可用，因为不是所有数据库都支持事务(一系列动作)，如果可用，那么它就可以"撤销"所有未提交的事务
- commit 方法总是可用的，但是如果数据库不支持事务，他就没有任何作用，**如果关闭了连接，但是还有未提交的事务，它们会隐式地回滚**
- cursor 方法将我们引入另一个主题：**游标对象**，通过游标执行 SQL 查询并检查结果，**游标比连接支持更多方法**

5、游标对象方法

名称	描述
callproc(name[,params])	使用给定的名称和参数(可选)调用已命名的数据库过程
close()	关闭游标之后，游标不可用
execute(oper[,params])	执行一个 SQL 操作，可能带有参数
executemany(oper,pseq)	对序列中的每个参数集执行 SQL 操作
fetchone()	把查询的结果集中的下一行保存为序列，或者 None
fetchmany([size])	获取查询结果集中的多行，默认尺寸为 arraysize
fetchall()	将所有(剩余)的行作为序列的序列
nextset()	跳至下一个可用的结果集(可选)
setinputsizes(sizes)	为参数预先定义内存区域
setoutputsize(size[,col])	为获取的大数据值设定缓冲区尺寸

6、游标对象特性

名称	描述
description	结果列描述的序列，只读
rowcount	结果中的行数，只读
arraysize	fetchmany 中返回的行数，默认为 1

13.1.4. 类型

1、为了能与底层的 SQL 数据库更好的交互，需要插入到列中的值的类型做出各种要求和限制，DB API 定义了用于特殊类型和值的构造函数及常量(单例模式)。

2、DB API 构造函数和特殊值

名称	描述
Date(year,month,day)	创建保存日期值的对象
Time(hour,minute,second)	创建保存时间值的对象
Timestamp(y,mon,d,h,min,s)	创建保存时间戳值的对象
DateFromTicks(ticks)	创建保存自新纪元以来的秒数的对象
TimeFromTicks(ticks)	创建保存来自描述的时间值的对象
TimestampFromTicks(ticks)	创建保存来自描述的时间戳值的对象
Binary(string)	创建保存二进制字符串值的对象
STRING	描述基于字符串的列类型
BINARY	描述二进制序列
NUMBER	描述数字列
DATETIME	描述日期/时间列
ROWID	描述行 ID 列

13.2. SQLite 和 PySQLite

1、可用的 SQL 数据库引擎有很多，且都有相应的 Python 模块，多数数据库引擎都作为服务器程序运行，并且安装都需要管理员权限。

2、我们选择小型数据库引擎 SQLite 作为示例，它并不需要作为独立的服务器运行，并且不基于集中式数据库存储机制，而是直接在本地文件上运行

3、SQLite 的有时在于他的一个包装(PySQLite)已经被包括在标准库内，因此不用单独安装 PySQLite 和 SQLite 了

LeetCode 中遇到的语法问题

1. 想要在用 foreach 语句中获取索引号？
   - 利用函数 enumerate()
2. 逻辑表达式的与关系？
   - and 而不是&&
3. 如何定义一个指定长度的列表(当做其他语言的数组用)
   - a=[0 for i in range(10)]
4. Python 中如何实现类似 Java 中的 ary['b'-'a']等价于 ary[1]？换句话说，Python 有字符类型吗
   - 'a' in 'ab' ==> True
5. 获取包含 a-z 的字母的列表
   - print(''.join(map(chr,range(97,123))))
   - print(string.ascii_lowercase)
6. 字符(character)和整型(int)之间的转换：
   - ord(character) ==>将字符转为 int
   - chr(int) ==>将 int 转为字符
7. Python 放弃了自增运算符
   - Python 哲学的一句话：只用一种方式解决问题，所以你要的自增操作完全可以用 i+=1 完成，就不需要 i++了。
8. /默认执行整数的除法，若要执行浮点数，用 float 进行转换即可