www.freeggache.com

freeggache

python

basic

目录

1 序言

作为一只编程菜鸟，每次学一门语言，我都要反反复复，对自己的记忆力感到无比崩溃。于是乎，蠢爆了的我只能仔仔细细老老实实的专研（本人实在是找不到任何捷径）。幸运的是，菜鸟做久了，也琢磨出点道道来了，在这里，想将自己的一些心得和学习笔记分享与大家，但想强调一下，我写的这些渣文仅适用于和我一样初学的菜鸟（承认吧，孩子），谢谢。

2 我学 Python 用来干什么？

《Python 学习手册（第 4 版）》一书中对 Python 的优缺点做了很多详细的说明，在基础篇里我们不太关心，因为道行还不够深，所以只关心学了 Python 后可以干些啥，在提高篇里再对其进行补充。

- 系统编程

Python 对操作系统服务的内置接口，使其成为编写可移植的维护操作系统的管理工具和部件（有时也称 shell 工具）的理想工具。

- 用户图形接口

Python 的简洁以及快速的开发周期十分适合开发 GUI 程序。Python 内置了 TKinter 的标准面向对象接口 Tk GUI API，使 Python 程序可以生成可移植的本地观感的 GUI。

- Internet 脚本

Python 提供了标准的 Internet 模块，它使得 Python 程序能够广泛地在多种网络任务中发挥作用，无论是在服务器端还是在客户端都是如此。

- 组件集成

Python 可以通过 C/C++系统进行扩展，并能够嵌套 C/C++系统的特性，使其能够作为一种灵活的黏合语言，可以脚本化处理其他系统和组件的行为。

- 数据库编程

对于传统的数据库需求，Python 提供了对所有主流关系数据库系统的接口。

- 快速原型

由于使用 Python 或 C 编写的组件看起来一样，所以可以在一开始利用 Python 做系统原型，之后再讲组件移植到 C 或 C++这样的变异语言上。

- 数值计算和科学计算编程
- 游戏、图像、人工智能、XML、机器人等

最后两点，这里暂时不提，感兴趣可以私下研究。

貌似有些复杂，好了，总之就是能干很多事了，刚开始学，大概了解下就行，知道些名词能吹吹牛就好。（没前途…）但有一个优点对于我们这些懒癌患者相当重要，那就是 Python 编程所需代码比 Java、C++少太多了！

在 java 中输出字符串代码如下：

```
public class A
{
    public static void main(String args[]){
        System.out.print(“haha”);
    }
}
```

C++中输出字符串代码如下：

```
#include<iostream>
using namespace std;
int main(){
    cout<<”haha”;
```

```
}
```

然而在 Python 中，代码仅仅如下：

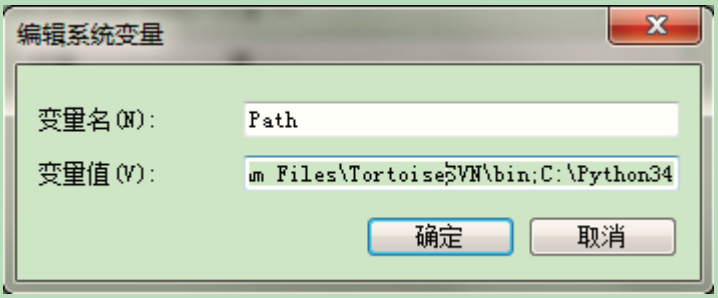
```
print(“haha”)
```

是不是格外的简单？

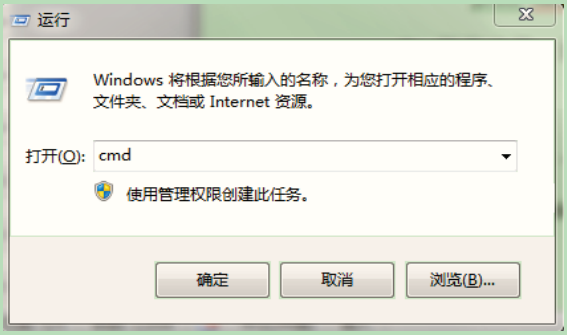
3 编写运行“Hello, world!”

无论学什么语言，永远的第一个程序都是输出“Hello, world!”，之所以如此，无非是让我们去了解该语言是如何运行程序的。所以，按照国际惯例。我们也来学习一下 Python 的“Hello, world!”程序该如何编写（基础篇暂时只研究 Windows）。

- 首先，要安装 Python。
  - 下载 Python（这里就以当前最新版 3.4.2 为例）
    https://www.python.org/ftp/python/3.4.2/python-3.4.2.amd64.msi
  - 安装
    按照默认路径，一直点确认即可。
  - 修改系统 Path
    右击“我的电脑”——属性——高级系统设置——环境变量——双击系统变量“Path”——在变量值的最后加上“;C:\Python34”如下图：



  - 进入命令控制行窗口（windows 键+R，输入 cmd）

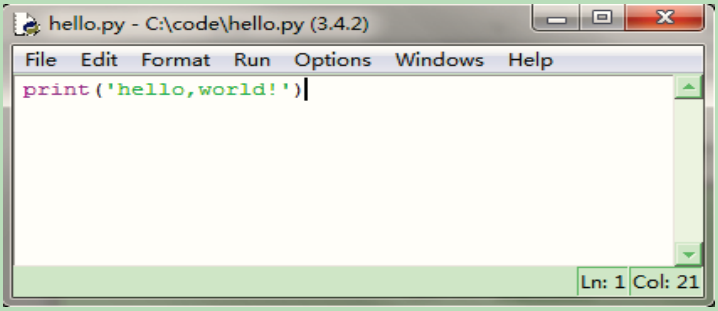


  - 检验 python 是否安装好
    输入：python
    如有显示以下画面说明已经安装好

```
c:\code>python
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  6 2014, 22:16:31) [MSC v.1600 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- 编写“Hello，world”程序
  - 首先在 C 盘下新建一个 code 文件夹，接着在 code 文件夹中新建文本文档 hello.txt，然后手动把文件名改成 hello.py，如下图：

- 选中该文件，然后右击选择 Edit with IDLE，在文本框内输入：print（‘hello,world!’）



- 保存文件，进入命令控制行，切换目录
  输入：cd c:\code

```
C:\Users\hu_xiaodan>cd c:\code

c:\code>
```

- 运行 hello.py
  输入：python hello.py

```
c:\code>python hello.py
hello,world!
```

好了，“hello,world!”就这样完成了！

像这样，每次编写代码（可以不用 python 自带的 IDLE，而是使用一些工具如 UltraEdit 或其他）都要保存为.py 的格式，然后用 Python 自带的 python.exe 去执行他。

除了上述方法，还有一种交互地运行代码的方式，可以启动 IDLE 得主窗口，或通过从 Python 的 Start 按钮菜单的菜单选项中选择“Python(command line)”来开始类似的交互会话。这两种方式都会产生一个具有同样功能的 Python 交互式命令提示符，而不必输入一条 shell 命令。

像这样在交互模式下工作，想输入多少 Python 命令就输入多少，每一个命令在输入回车后都会立即运行。此外，由于交互是对话自动打印输入表达式的结果，在这个提示模式下，往往不需要每次都刻意地输入“print”：

现在，我们对这次绘画中的代码并不是特别了解：仅仅是输入一些 Python 的打印语句和变量赋值的语句，以及一些表达式，这些我们都会在稍后进行深入的学习。

4 Python 对象类型

在典型的 Python 程序中，部署内存结构、管理内存分配、实现搜索和读取例程这些令人头痛的大部分工作都消失了。因为 Python 提供了强大的对象类型作为语言的组成部分，在你开始解决问题之前往往没有必要编写对象的实现。事实上，除非你有内置类型无法提供的特殊对象要处理，最好总是使用内置对象而不是使用自己的实现。

Python 的核心数据类型如下：

对象类型	例子 常量/创建
数字	1234,3.1415,3+4j,Decimal,Fraction
字符串	‘aa’,” ff”
列表	[1,[2,’ three’],4]
字典	{ ‘food’ :’ spam’ ,’ taste’ :’ yum’ }
元组	(1,’ aa’ ,4,’ U’)
文件	myfile = open(‘eggs’ ,’ r’)
集合	set(‘abc’),{ ‘a’ ,’ b’ ,’ c’ }
其他类型	类型、None、布尔型
编程单元类型	函数、模块、类
与实现相关的类型	编译的代码堆栈跟踪

4.1 数字

Python 的核心对象集合包括常规的类型：整数、浮点数以及更为少见的类型（有虚部的复数、固定精度的十进制数、带份子和分母的有理分数以及集合等）。Python 中的数字支持一般的运算。例如，加号（+）代表加法，星号（*）表示乘法，双星号（**）表示乘方。

```
>>>123 + 222
345
>>>1.5 * 4
6.0
```

```
>>>2 ** 100
1267650600228229401496703205376
```

注意，当需要的时候，Python3.4 的整数类型会自动提供额外的精度，以用于较大的数值。例如，你可以计算 2 的 1,000,000 次幂，但打印的话得要好一会了。

```
>>>len(str( 2** 1000000))          # How many digits in a really BIG number
301030
```

但当你刚接触浮点数时，却会遇到些奇怪的现象：

```
>>>3.1415 * 2
6.2830000000000004
>>>print(3.1415 * 2)
6.283
```

第一个结果并不是 Bug，而是现实的问题。这里体现了两种打印方式：全精度（第一个结果显示的那样）以及用户友好的方式（第二个）。如果以后遇到有些东西显示的很奇怪，试试使用打印语句显示他们。

4.2 字符串

字符串是用来记录文本信息的。在 Python 中他们以序列的形式储存。序列中的元素包含了一个从左到右的顺序——序列中的元素根据他们的相对位置进行存储和读取，从严格意义上来说，字符串是由单个字符组成的序列，其他类型的序列还包括列表和元组（后面介绍）。

4.2.1 序列的操作

如果我们有一个含有 4 个字符的字符串，我们可以通过内置的 len 函数验证他的长度并通过索引取得各个元素。

```
>>>s = 'ABCD'
>>>len(s)                    # Length
4
>>>s[0]                      # The first item in s,indexing by zero-base position
'A'
>>>s[1]                      # The second item from the left
'B'
```

在 Python 中，索引也是从 0 开始，第二项索引为 1，以此类推。另外，我们还可以进行反向索引，从最后一个开始

```
>>>s[-1]                     # The last item in s
'D'
>>>s[-2]                     # The second to last item from the end
```

```
'C'
```

除了简单地从位置进行索引，序列也支持一种所谓切片（slice）的操作。

```
>>>s                          # A 4-character string
'ABCD'
>>>s[1:3]                     # Slice of from offsets 1 through 2(not 3)
'BC'
```

认识切片最简单的办法就是把他们看作从一个字符串中一步就提取出一部分的方法。他们的一般形式为 X[I:J]，表示“从 X 中取出偏移 I 开始，直到但不包括 J 的部分”。结果返回一个新的对象。

在一个切片中，左边界默认为 0，并且右边界默认为切片序列的长度，于是演化出以下的用法。

```
>>>s[1:]                       # Everything past the first(1:len(s))
'BCD'
>>>s[0:3]                  # Everything but the last
'ABC'
>>>s[:3]                       # Same as s[0:3]
'ABC'
>>>s[:-1]                  # Everything but the last again,but simpler(0:-1)
'ABC'
>>>s[:]                        # All of S as a top-level copy(0:len(S))
'ABCD'
```

最后，作为一个序列，字符串也支持使用加号进行合并（将两个字符串合并成为一个新的字符串），或者复制。

```
>>>s
'ABCD'
>>>s + 'xyz'                   # Concatenating
'ABCDxyz'
>>>s                       # S is unchanged
'ABCD'
>>>s * 4                       # Repetition
'ABCDABCDABCDABCD'
```

注意，（+）对于不同的对象有不同的意义：对于数字为加法，对于字符串为合并。这个特性叫多态，在后面会提到。

4.2.2 不可变性

在之前的例子中，无论什么样的操作，都不会对原始的字符串进行改变，而是生成新的字符串来保存结果，这是因为字符串在 Python 中具有不可变性——在创建后就不能修改。

```
>>>s
```

```
'ABCD'
>>>s[0] = 'Z'                    # Immutable objects cannot be changed
...error text omitted
TypeError: 'str' object does not support item assignment
>>>s = 'Z' + s[1:]               # But we can run expressions to make new objects
>>>s
'ZBCD'
```

在 Python 中每个对象都具有部可变性或者可变性。在核心类型中，数字、字符串、元组是不可变的，而列表和字典是可变的。

4.2.3 字符串的特定操作

上述操作在其他序列中也可使用，而下面这些操作是字符串所特有的：

例如，find 方法是对子字符串进行查找的操作（他返回所找到的子字符串的偏移量，如没找到则返回-1），replace 方法将会对全局进行搜索和替换。

```
>>>s
'ABCD'
>>>s.find('BC')                  # find the offset of a substring
>>>1
>>>s.replace('BC' , 'XYZ')       # Replace occurrences of a substring with another
'AXYZD'
>>>s
'ABCD'
```

尽管字符串方法的命名有改变的含义，但却不会改变原始的字符串，而是会创建一个新的字符串作为结果——因为字符串具有不可变性。下面是字符串的一些常用方法 split()、upper()、isalpha()、rstrip()：

```
>>>line = 'aaa,bbb,ccccc,dd'
>>>line.split(',')                # Split on a delimiter into a list of substrings
['aaa', 'bbb', 'ccccc', 'dd']
>>>s = 'abcd'
>>>s.upper()                      # Upper-and lowercase conversions
>>>'ABCD'
>>>s.isalpha()                # Content tests:isalpha,isdigit,etc
True
>>>line = 'aaa,bbb,cccc,dd\n'
>>>line = line.rstrip()           # Remove whitespace characters on the right side
>>>line
'aaa,bbb,ccccc,dd'
```

4.2.4 转义字符

```
>>>s = 'A\nB\tC'                    # \n is end-of-line, \t is tab
>>>len(s)                    # Each stands for just one character
5
>>>ord('\n')                        # \n is a byte with the binary value 10 in ASCII
10
>>>s='A\0B\0C'                      # \0. a binary zero byte,does not terminate string
>>>len(s)
5
```

在 Python 中'xx '和"xxx" 表示方法相等，它还允许在三个引号（但引号或双引号）中包含多行字符串常量。这种表示方法会将所有的行都合并在一起，并在每一行的末尾增加换行符。

```
>>>msg = """aaa
bbb'''bbb""bbb'bbb
ccc"""
>>>msg
'\naaa\nbbb\'\'\'bbb""bbb\'bbb\nccc'
```

4.3 列表

Python 的列表对象是这个语言提供的最通用的序列，他代表任意类型的对象的有序集合，他没有固定的大小，是可变的。

4.3.1 序列操作

列表支持所有我们对字符串所进行过的序列操作。唯一的区别就是其结果往往是列表而不是字符串。

```
>>>L = [123, 'ABC' , 1.23]          # A list of three different-type objects
>>>len(L)
3
>>>L[0]                             # Indexing by position
123
>>>L[:-1]                    # Slicing a list returns a new list too
[123, 'ABC']
>>>L+[4,5,6]                        # Concatenation makes a new list too
[123, 'ABC, 1.23 ,4 ,5 ,6]
>>>L                                # We're not changing the original list
[123 , ABC' ,1.23]
```

4.3.2 特定类型的操作

Python 的列表与其他语言中的数组有些类似，但是列表要强大得多。其中一个方面就是，列表没有固定类型的约束。例如，上个例子中接触到的列表，包含了三个完全不同类型的对象。此外，列表没有固定大小，也就是说能够按照需要增加或减小列表大小，来响应其特定的操作：

```
>>>L.append('Suen')          # Growing: add object at end of list
>>>L
[123, 'ABC', 1.23, 'Suen']
>>>L.pop(2)                  # Shrinking: delete an item in the middle
1.23
>>>L                         # "del L[2]" deletes from a list too
[123, 'ABC' , 'Suen']
```

这里，append()扩充了列表的大小并在列表尾部插入一项,pop()方法(或者等效的 del 语句)移除给定偏移量的一项，从而让列表减小。其他的列表方法可以在任意位置插入（insert）元素，按照值移除（remove）元素等。因为列表是可变的，大多数列表方法都会直接改变列表对象，而不是创建一个新的列表：

```
>>>M = ['bb', 'aa', 'cc']
>>>M.sort()
>>>M
['aa', 'bb', 'cc']
>>>M.reverse()
>>>M
['cc', 'bb', 'aa']
```

4.3.3 边界检测

尽管列表没有固定的大小，Python 仍不允许引用不存在的元素。超出列表末尾之外的索引总是会导致错误，对列表末尾范围之外赋值也是如此。

```
>>>L
>>>[123, 'ABC' , 'Suen']
>>>L[99]
…error text omitted…
IndexError: list index out of range
>>>L[99] = 1
…error text omitted…
IndexError: list index out of range
```

这是有意为之的，去给一个列表边界外的元素赋值，往往会得到一个错误。在 Python 中，并不是默默地增大列表作为响应，而是会提示错误。为了让一个列表增大，我们可以调用 append 这样的列表方法。

4.3.4 嵌套

Python 核心数据类型的一个优秀特性就是它们支持任意的嵌套。能够以任意的组合对其进行嵌套，并可以多个层次进行嵌套。这种特性的一个直接的应用就是实现矩阵，或者 Python 中的“多维数组”。

```
>>>M=[ [1, 2, 3 ],                    # A 3*3 matrix, as nested lists
       [4, 5, 6 ],
       [7, 8, 9 ]]
>>>M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

这里，我们编写了一个包含 3 个其他列表的列表。这样的结构可以通过多种方法获取元素。

```
>>>M[1]                             # Get row 2
>>>[4, 5, 6]
>>>M[1][2]                          # Get row 2,then get item 3 within the row
```

4.3.5 列表的解析

列表解析表达式是 Python 所提供的一种处理像矩阵这样结构的强大的工具。例如，假设我们需从列举的矩阵里提取出第二列，由于矩阵是按照行进行存储的，所以通过简单的索引即可获取行，而使用列表可以同样简单的获得列。

```
>>>col2 = [row[1] for row in M]        # Collect the items in column2
>>>col2
[2, 5, 8]
>>>M                                          # The matrix is unchanged
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

这里的列表解释表达的是“把矩阵 M 的每个 row 中的 row[1]，放在一个新的列表中”。

4.4 字典

Python 中的字典是完全不同的东西，它们不是序列，而是一种映射。映射是一个其他对象的集合，它们是通过键而不是相对位置来存储的。映射并没有任何可靠的从左至右的顺序，而是简单地将键映射值。字典是 Python 核心对象集合中的唯一的一种映射类型，也具有可变性。

4.4.1 映射操作

作为常量编写时，字典编写在大括号中，并包含一系列的“键：值”对。在

我们需要将键与一系列值相关联（例如，为了表示某物的属性）的时候，字典是很有用的。如下例（键分别为“food”、“quantity”、“color”）：

```
>>>D = {‘food’ : ’beef’, ‘quantity’ : 4, ’color’ : ’pink’}
```

我们可以通过键对这个字典进行索引来读取或改变键所关联的值。

```
>>>D[‘food’]                    # Fetch value of key ‘food’
>>>’beef’
>>>D[‘quantity’] +=1            # Add 1 to ‘quantity’ value
>>>D
{‘food’ : ’beef’, ‘quantity’ : 5, ’color’ : ’pink’}
```

尽管可以使用大括号这种常量形式，最好还是见识一下不同的创建字典的方法。例如，下面开始时是一个空的字典，然后每次以一个键来填写它。与列表中禁止边界外的赋值不同，对一个新的字典的键赋值会创建该键：

```
>>>D = { }
>>>D[‘name’] = ‘Bob’            # Create keys by assignment
>>>D[‘job’] = ‘ engineer‘
>>>D
{‘name’ : ‘Bob’, ‘job’ : ‘engineer’}
>>>print(D[‘name’])
Bob
```

字典也可以用来执行搜索。通过键索引一个字典往往是Python中编写搜索的最快方法。

同样重要的是，在底层语言中，当我们不再需要该对象时，必须小心地去释放掉所有对象空间。在Python中，当最后一次引用对象后（例如，将这个变量用其他的值进行赋值），这个对象所占用的内存空间将会自动清理掉，从技术来说，Python具有一种叫做垃圾收集的特性，在程序运行时可以清理不再使用的内存，并将你从必须管理代码中这样的细节中解放出来。在Python中，一旦一个对象的最后一次引用被移除，空间将会立即回收。我们将会在后边学习这是如何工作的。目前，知道能够自由地使用对象就足够了，不需要为创建它们的空间或不再使用时清理空间而担心。

4.4.2 键的排序：for 循环

之前提到过，因为字典不是序列，它们并不包含任何可靠的从左至右的顺序。这意味着如果我们建立一个字典，并将它打印出来，它的键也许会以与我们输入时不同的顺序出现：

```
>>>D = {‘a’ : 1, ‘b’ : 2, ‘c’ : 3}
>>>D
>>>{‘b’ : 2, ‘c’ : 3, ‘a’ : 1}
```

那么，如果在一个字典的元素中，我们确实需要强调某种顺序的时候，应该

怎样做呢？一个常用的解决办法就是通过字典的 keys()方法收集一个键的列表，使用列表的 sort()方法对其进行排序，然后使用 Python 的 for 循环逐个进行显示结果（确保在循环的代码下面两次按下 Enter 键，交互提示模式中的一个空行意味着“执行”，某些接口中提示符是“...”）：

```
>>>K = D.keys()                    # Get D.keys
>>>K
dict_keys(['b', 'c', 'a'])
>>>K = list(K)                     # Unordered keys list
>>>K
['b', 'c', 'a']
>>>K.sort()                        # Sorted keys list
>>>K
['a' ,'b' ,'c']
>>>for key in K:                   # Iterate through sorted keys
        print(key,'=>,D[key])      # < == press Enter twice here
a => 1
b => 2
c => 3
```

然而，在最新的版本中，通过使用最新的 sorted 内置函数可以一步完成。sorted 调用返回结果并对各种对象类型进行排序。

```
>>>D
{'b' : 2, 'c' : 3, 'a' : 1}
>>>for key in sorted(D)
        print(key, '=>', D[key]
a => 1
b => 2
c => 3
c => 3
```

4.4.3 迭代和优化

for 循环和之前的列表解析表达式一样，都是通用的迭代工具，它们都遵守迭代协议，迭代协议是 Python 中重要的一个概念，在迭代操作情况下，每次都会产生一个 next 的后续对象。这就是为什么前面一节所介绍的 sorted()方法可以直接工作于字典之上，我们不必调用 keys()方法来得到一个序列，因为字典是可迭代的对象，可以用一个 next()返回后续的键。

这也意味着像下面这样的任何列表解析表达式都可以计算一列数字的平方

```
>>>squares = [x ** 2 for x in [1, 2, 3, 4, 5]]
>>>squares
[1, 4, 9, 16, 25 ]
```

关于字典还有另一个要点：尽管我们能够通过给新的键赋值来扩展字典，但是获取一个不存在的键值仍然是一个错误。

```
>>>D
{ ‘a’: 1, ‘c’: 3, ‘b’ : 2}
>>>D[‘e’] = 99                    #Assigning new keys grows dictionaries
>>>D
{ ‘a’ : 1, ‘c’ : 3, ‘b’ : 2, ‘e’ : 99}
>>>D[‘f’]                         #Referencing a nonexistent key is an error
... error tex t omitted ...
KeyError :
```

这就是我们所想要的：在取一个并不存在的东西往往是一个程序错误。但是，在一些通用程序中，我们编写程序时并不是总知道当前存在什么键。在这种情况下，我们如何处理并避免错误发生呢？一个技巧就是首先进行测试。in 关系表达式允许我们查询字典中一个键是否存在，并可以通过使用 Python 的 if 语句对结果进行分支处理

```
>>>‘f’ in D
False
>>>if not ‘f’ in D:
        print(' missing')
missing
```

4.5 元组

元组对象（tuple）基本上就像一个不可以改变的列表。就像列表一样，元组是序列，但是他具有不可变性，和字符串类似。从语法上讲，它们编写在圆括号中而不是方括号中，它们支持任意类型、任意嵌套以及常见的序列操作：

```
>>>T = (1, 2, 3, 4)
>>> len(T)
4
>>>T + (5, 6)
(1, 2, 3, 4, 5, 6)
>>>T[0]
1
```

在 Python 3.0 中，元组还有两个专有的可调用方法 ，但它的专有方法不像列表所拥有的那么多：

```
>>>T.index(4)              # Tuple methods: 4 appears at offset 3
3
>>>T.count (4)                    # 4 appears once
1
```

元组的真正的不同之处就在于一旦创建后就不能再改变。也就是说，元组是

不可变的序列：

```
>>>T[0] = 2                          # Tuples are immutable
…error text omitted…
TypeError: 'tuple' object does not support item assignment
```

与列表和字典一样，元组支持混合的类型和嵌套，但是不能增长或缩短，因为它们是不可变的：

```
>>>T = ('spam’, 3.0, [11, 22, 33])
>>>T[l]
3.0
>>>T[2][1]
22
>>>T.append(4)
AttributeError: 'tuple' object has no attribute ’ append ’
```

4.6 其他核心类型

Python 最近添加了一些新的数值类型：十进制数（固定精度浮点数）和分数（有一个分子和一个分母的有理数）。它们都用来解决浮点数学的局限性和内在地不精确性：

```
>>>1/3                                    # Floating-point (use .0 in Python 2.6)
0. 33333333333333331
>>>(2/3) + (1/ 2)
1.1666666666666665

>>>from decimal import *                  # Decimals: fixed precision
>>>d = decimal. Decimal( ' 3 .141')
>>>d + 1
Decimal('4.141')
>>>decimal. getcontext () • prec = 2
>>>decimal .Decimal('1.00') / decimal.Decimal(' 3.00')
Decimal('0.33')

>>>from fractions import Fraction      # Fractions: numerator + denominator
>>>f = Fraction(2, 3)
>>>f + 1
Fraction(5, 3)
>>>f + Fraction (1, 2)
Fraction(7, 6)
```

还有长期以来一直支持的特殊占位符对象 None（它通常用来初始化名字和对象）：

```
>>>X = None                    # None placeholder
>>>print(x)
None
>>>L = [None] * 100            # Initialize a list of 100 Nones
>>>L
[None, None, None, None, None, None, None……, None]
```

5 动态类型简介

5.1 变量、对象和引用

与传统的静态语言不同，Python 语言的的变量规定如下：

- 变量创建

  一个变量（也就是变量名），比如 a，当代码第一次给它赋值时就创建了它。之后的赋值将会改变已创建的变量名的值。从技术上来讲，Python 在代码运行之前先检测变量名，可以当成是最初的赋值创建变量。

- 变量类型

  变量永远不会有任何的和它关联的类型信息或约束。类型的概念是存在于对象中而不是变量名中。变量原本是通用的，它只是在一个特定的时间点，简单地引用了一个特定的对象而已。

- 变量使用

  当变量出现在表达式中时，它会马上被当前引用的对象所代替，无论这个对象是什么类型。此外，所有的变量必须在其使用前明确地赋值，使用未赋值的变量会产生错误。

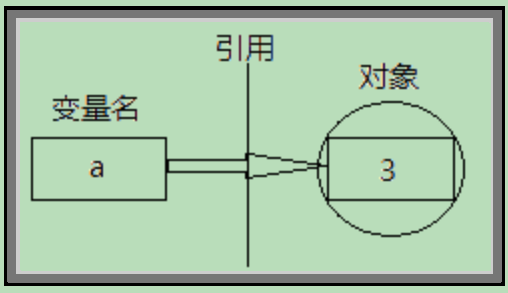
当我这们这样说时：

```
>>>a = 3
```

Python 将会执行三个不同的步骤去完成这个请求。遮羞步骤反映了 Python 语言中所有赋值的操作：

- 创建一个对象来代表值 3 。
- 创建一个变量 a ，如果它还没有创建的话。
- 3. 将变量与新的对象 3 相连接。

实际效果如下图：



在 Python 中从变量到对象的连接称作引用。也就是说，引用是一种关系，以内存中的指针的形式实现，一旦变量被使用（也就是说被引用），Python 自动跟随这个变量到对象的连接。以具体的术语来讲：

- 变量是一个系统表的元素，拥有指向对象的连接的空间。
- 对象是分配的一块内存，有足够的空间去表示它们所代表的值。
- 引用是自动形成的从变量到对象的指针。

注意 Python 中的类型是与对象相关联的，而不是和变量关联。每个对象都包含两个头部信息，一个是类型标识符，它用来表明该对象是什么类型，以便创建指向该对象的指针。另一个是引用计数器，下面我们来谈谈后者。

5.2 对象的垃圾收集

下面我们来思考，如果重新给变量 a 赋值，那么它前一个引用值会发生什么变化？例如，在下边的语句中，对象 3 发生了什么变化？

```
>>>a = 3
>>>a = ' spam '
```

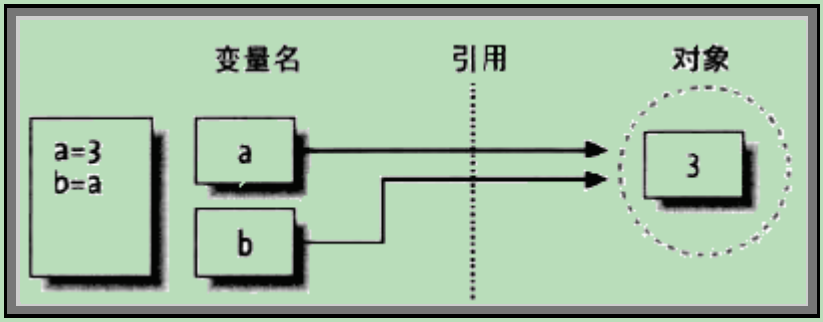
答案是，在 Python 中，每当一个变量名被赋予了一个新的对象，之前的那个对象占用的空间就会被回收（如果它没有被其他的变量名或对象所引用的话）。这种自动回收对象空间的技术叫做垃圾收集。垃圾收集最直接的、可感受到的好处就是，这意味着可以在脚本中任意使用对象而不需要考虑释放内存空间。

5.3 共享引用、

到现在为止，我们已经看到了单个变量被赋值引用了多个对象的情况。现在，在交互模式下，引入另一个变量，并看一下变量名和对象的变化：

```
>>>a = 3
>>>b = a
```

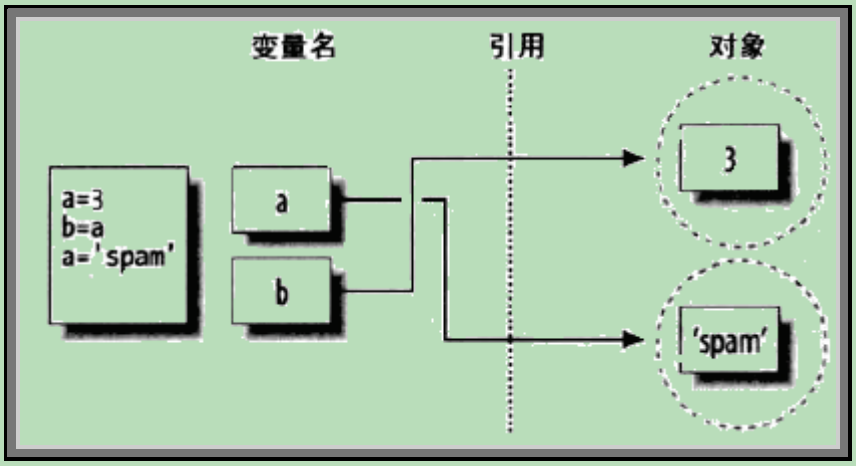
输入这两行语句后，第二行会使 Python 创建变量 b。变量 a 正在使用，并且它在这里没有被赋值，所以它被替换成其引用的对象 3，从而 b 也成为这个对象的一个引用。实际的效果就是变量 a 和 b 都引用了相同的对象（也就是说，指向了相同的内存空间）。这在 Python 中叫做共享引用——多个变量名引用了同一个对象。



下一步，假设运行另一个语句扩展了这样的情况：

```
>>>a = 3
>>>b = a
>>>a = ‘spam’
```

这条语句简单地创建了一个新的对象（代表字符串值‘Spam’）,并设置 a 对这个新的对象进行引用。尽管这样，这并不会改变 b 的值，b 仍然引用原始的对象——整数 3。最终的引用结构如图：



下面，让我们思考一下下面这 3 条语句：

```
>>>a = 3
>>>b = a
>>>a = a + 2
```

在这里，产生了同样的结呆： Python 让变量 a 引用对象 3 ，让 b 引用与 a 相同的对象，但对于 a=a+2，我们没有办法去改变对象 3 的值而是产生了新的对象 5，并且让 a 指向了 5，就像第 4 章所介绍过的，整数是不可变的，因此没有方法在原处修改它。

认识这种现象的一种方法就是，不像其他的一些语言，在 Python 中，变量总是一个指向对象的指针，而不是可改变的内存区域的标签。给一个变量赋一个新的值，并不是替换了原始的对象，而是让这个变量去引用完全不同的一个对象（仅限于不可变对象）。

5.4 共享引用和在原处修改

虽然不可变对象无法在原处修改，但可变变量却可以做到这一点。例如，在一个列表中对一个偏移进行赋值确实会改变这个列表对象，而不是生成一个新的列表对象。对于支持这种在原处修改的对象，共享引用时的确需要加倍的小心，因为对一个变量名的修改会影响其他的变量。

下面，让我们回顾一下之前所学过的列表：

```
>>>L1 = [2, 3, 4]
>>>L2 = L1
```

L1 是一个包含了对象 2、3 和 4 的列表。在列表中的元素是通过它们的位置进行读取的，所以 L1[0]引用对象 2 ，它是列表 L1 中的第一个元素。当然，列表自身也是对象，就像整数和字符串一样。在运行之前的两个赋值后，L1 和 L2 引用了相同的对象，就像我们之前例子中的 a 和 b 一样（如图 6-2 所示）。如果我们现在像下面这样去扩展这个交互：

```
>>>L1[0] = 24
>>>L1
[24, 3, 4]
```

```
>>>L2
[24, 3, 4]
```

这个时候，L2 也随着 L1 的改变而改变了！在这里，没有改变 L1，而是改变了 L1 所引用的对象的一个元素。这类修改会覆盖列表对象中的某部分。因为这个列表对象是与其他对象共享的（被其他对象引用），那么一个像这样在原处的改变不仅仅会对 L1 有影响。也就是说，必须意识到，当我们做了这样的修改，它会影响程序的其他部分。在这个例子中，也会对 L2 产生影响，因为它与 L1 都引用了相同的对象。另外，我们实际上并没有改变 L2 ，但是它的值将发生变化，因为它已经被修改了。

如果想避免这种情况，可以采用拷贝对象的办法，而不是去直接创建引用。如下：

```
>>>L1 = [2, 3, 4]
>>>L2 = L1[:]                    # Make a copy of L1
>>>L1[0] = 24
>>>L1
[24, 3, 4]
>>>L2                            # L2 is not changed
[2, 3, 4]
```

这里，对 L1 的修改不会影响 L2，因为 L2 引用的是 L1 所引用对象的一个拷贝。也就是说，两个变量指向了不同的内存区域。

我们需记住有些对象是可以在原处改变的（即可变的对象），这种对象往往对这些现象总是很开放。在 Python 中，这种对象包括了列表、字典以及一些通过 class 语句定义的对象。如果这不是你期望的现象，可以根据需要直接拷贝对象。

6 Python 语句简介

6.1Python 增加了什么

Python 中新的语法成分是冒号(:)。所有的 Python 复合语句（也就是语句中嵌套了语句）都有相同的一般形式，也就是首行以冒号结尾，首行下一行嵌套的代码往往按缩进的格式书写，如下所示：

```
Header line:
    Nested statement block
```

6.2Python 删除了什么

- 括号是可选的
  例如 if (x <y)在 Python 中可以写成 if x < y
- 终止行就是终止语句
  不需要添加分号‘;’
- 缩进的结束就是代码块的结束
  不需要花括号{ }

6.3 语句规则的特殊情况

虽然语句一般都是一行一个，但是 Python 中也有可能出现某一行挤进多个语句的情况，这时色们用分号隔开：

```
a = 1; b = 2; print(a + b)
```

这是 Python 中唯一需要分号的地方——作为语句界定符。语句的另一个特殊规则基本上是相反的一一可以让一个语句的范围棋跨多行。为了能实现这一操作，你只需要用一对括号把语句括起来就可以了：括号“（)”、方括号“[ ]”或者字典的大括号“{ }”。任何括在这些符号里的程序代码都可横跨好几行。语句将一直运行， 直到 Python 遇到包含闭合括号的那一行。例如，连续几行列表的常量：

```
mlist = [111,
         222,
         333]
```

6.4 简短实例：交互循环

6.4.1 一个简单的交互式循环

假设有人要你写个 Python 程序，要求在控制窗口与用户交互。也许你要把输入数据传送到数据库，或者读取将参与计算的数字。不管是什么目的，你需要写一个能够读取用户键盘输入数据的循环并打印每次读取的结果。换句话说，你需要写一个标准的“读取 / 计算 / 打印”的循环程序。

在 Python 中，这种交互式循环的典型模板代码可能会像这样。

```
While True:
    reply = input('Enter text:')
    if reply == 'stop' : break
    print(reply.upper())
```

这里需注意以下几点：

- 这个程序利用了 Python 的 while 循环，这个我们稍后会谈。
- input 是 Python 的内置函数，在这里用于通用控制台输出，他打印可选的参数字符串作为提示，并返回用户输入的回复字符串。
- Python 的 break 语句用于立即退出循环，也就是完全跳出循环语句而程序会继续循环之后的部分。

当程序运行时，我们从这个程序取得的某种程度上的交互：

```
Enter text: Hello
Hello
Enter text: 42
42
Enter text: stop
```

6.4.2 对用户输入数据做数学运算

脚本能够运行，但现在假设不是把文本字符 E 扭转换为大写字，而是想对数值的输入做些数学运算。例如，求平方。

```
>>>reply = '20'
>>>reply ** 2
... error text omitted ...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

然而这无先是在脚本中运行， 因为除非表边式里的对象类型都是数字，否则 Python 不会在表达式中自动转换对象类型，而来自于用户的输入返回脚本时一定是一个字符串。我们无法使用数字的字符串求幂，除非我们手动地把它转换为整数：

```
>> int(reply）**2
```

```
400
```

有了这个信息之后，现在我们可以重新编写循环来执行必要的数学运算。

```
while True:
    reply = input(' Enter text:')
    if reply == 'stop': break
    print(int(reply)**2)
print(' Bye')
```

这个脚本用了一个单行 if 语句在“stop”处退出，但是也能够转换输入来进行需要的数学运算。这个版本在底端加了一条结束信息。因为最后一行的 print 语句不像嵌套代码块那样缩进，不会看作是循环体的一部分，所以只能退出循环之后运行一次。

```
Enter text:2
4
Enter text: 40
1600
Enter text: stop
Bye
```

6.4.3 用测试输入数据来处理错误

但我们输入的数值无法转换成 int 型时，上面的程序会报错，为了使我们的脚本更加健全，可以事先用字符串对象的 isdigit()方法检查字符串的内容。

```
>>>S = ‘123’
>>>T = ‘xxx’
>>>S.isdigit(),T.isdigit()
(True ,False)
```

下面这个新版本的交互式脚本使用全方位的 if 语句来避免错误导致的异常。

```
While True:
    reply = input(‘Enter text:’)
    if reply == ‘stop’:
        break
    elif reply.isdigit():
        print(int(reply) ** 2)
    else:
        print(‘Bad!’ * 8)
print(‘Bye’)
```

当我们运行新脚本肘，程序会在错误发生前捕捉它，然后打印出（虽然不灵活）错误消息来进行说明。

```
Enter text : 5
```

```
25
Enter text : xyz
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:10
100
Enter text:stop
```

6.4.4 用 try 语句处理错误

在 Python 中，处理错误最通用的方式是使用 try 语句，用它来捕捉并完全复原错误。

```
While True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    try:
        num = int(reply)
    except:
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
print('Bye')
```

这个版本的运作方式和上一个版本的相同，但却把刻意进行错误检查的代码，换成了假设转换可工作的代码，然后把无法运作的情况，包含在异常处理器中。这个 try 语句的组成是：try 关键字后面跟代码主要代码块（我们尝试运行的代码），再跟 except 部分，给异常处理器代码，再接 else 部分，如果 try 部分没有引发异常，就执行这一部分的代码。Python 会先执行 try 部分，然后运行 except 部分（如果有异常发生）或 else 部分（如果没有异常发生）。

7 赋值、表达式和打印

7.1 赋值语句的特性

- 赋值语句建立对象引用值：
  赋值语句总是建立对象的引用值，而不是复制对象。因此，Python 变量更像是指针，而不是数据存储区域。
- 变量名在首次赋值时会被创建
- 变量名在引用前必须先赋值

7.2 赋值语句的形式

运算	解释
spam = ‘Spam’	基本形式
spam , ham = ‘yum’ , ‘YUM’	元组赋值运算（位置性）
[spam , ham] = [‘yum’ , ‘YUM’]	列表赋值运算（位置性）
a, b, c, d = ' spam '	序列赋值运算，通用性
a, *b = 'spam ’	扩展的序列解包（Python3.0）
spam= ham =’lunch’	多目标赋值运算
spams += 42	增强赋值运算（spams = spams + 42)