# Python基础

# 本节目录

- Python语言数据类型、运算符和表达式
- Python的数据结构
- Python的流程控制
- Python函数
- Python模块
- Python的输入、输出
- 异常处理

# Python语言文件类型、运算符和表达式

- 文件类型
- Python程序基本概念
- Python运算符和表达式

# 文件类型

- 在交互模式下，想输入多少Python命令，就输入多少；每个命令在输入回车后都立即运行。
- 只要不重新开启新的解释器，都在同一个会话中运行，因此，前面定义的变量，后面的语句都可以使用。
- 一旦关闭解释器，会话中的所有变量和敲入的语句将不复存在。

# 文件类型

- 为了能够永久保存程序，并且能够被重复执行，必须要将代码保存在文件中，因此，就需要用编辑器来进行代码的编写，和其他编程语言一样，Python的源代码可以直接执行而不需要像编译型语言一样编译成二进制代码。

- Python源代码文件就是普通的文本文件，只要是能编辑文本文件的编辑器都可以用来编写Python程序，如notepad/word等。

# 文件类型

Python的文件类型分为三种：源代码、字节代码、优化代码。

- 源代码

Python源代码文件，即py脚本文件，由 python.exe 解释，可在控制台下运行。

pyw脚本文件是图形用户接口(Graphical user interface) 的源文件，专门用来开发图形界面，由 pythonw.exe 解释运行。

!python test.py（run test.py）

# 文件类型

- 字节代码

  Python源文件经过编译后生成的pyc文件，即字节文件。它与平台无关，所以可以移植到其他系统上。下面这段脚本可以把 example.py 编译为 example.pyc

```
#compile py to pyc
import py_compile
py_compile.compile('example.py')
```

运行此脚本即可得到example.pyc

# 文件类型

不能在python的交互界面中运行。只能够在DOS屏幕上运行：python example.pyc
在程序中调用可以用：
os.system ("python example.pyc ")

- 优化代码

经过优化的源文件生成扩展名为pyo的文件，即优化文件。下面步骤可以把 example.py 编译为 example.pyo

- 启动命令行窗口，进入example.py所在目录：
  D:　cd D:\path\examples

# 文件类型

- 在DOS命令行中输入

python -O -m py_compile example.py

参数 -O 表示生成优化代码
参数 -m 表示导入的 py_compile 模块作为脚本运行。编译 example.pyo 需要调用 py_compile 模块中的 compile() 方法
参数 example.py 是待编译的文件名。

能够在DOS屏幕上运行：python example.pyo

# 文件类型

- 当程序比较大的时候，可以将程序划分成多个模块编写，每个模块用一个文件保存。
- 模块之间可以通过导入互相调用（import）。
- 模块也可以导入库中的其他模块。

Python是以模块进行重用的，模块中可以包括类、函数、变量等。

# 编码风格

- 以“#”号开头的内容为注释，python解释器会忽略该行内容。
- 在Python中是以缩进(indent)来区分程序功能块的，缩进的长度不受限制，但就一个功能块来讲，最好保持一致的缩进量。
  - 可以使用空格、Tab键等，但是最好保持一致
- 如果一行中有多条语句，语句间要以分号（;）分隔。

# Python程序基本概念

- 常量
  - 一个字面意义上的常量的例子是如同5、1.23、9.25e-3这样的数，或者如同‘This is a string’、“It‘s a string!”这样的字符串。
  - 它们被称作字面意义上的，因为它们具备*字面*的意义——按照它们的字面意义使用它们的值。数2总是代表它自己，而不会是别的什么东西——它是一个常量，因为不能改变它的值。因此，所有这些都被称为字面意义上的常量。

# Python程序基本概念

## 数

在Python中有4种类型的数——整数、长整数、浮点数和复数。

- 2是一个整数的例子。 >>> type(2)
- 长整数不过是大一些的整数。type(2L)

  基本上int是32位的。long是无限精度的。

- 3.23和52.3E-4是浮点数的例子。E标记表示10的幂。在这里，52.3E-4表示52.3 * 10-4。
- (-5+4j)和(2.3-4.6j)是复数的例子

# Python程序基本概念

## 字符串

字符串是*字符的序列*。 C语言中用字符数组表示，如char str[20] = “hello”.

Python中的字符串可以如下表示：

- **使用单引号（‘）**：可以用单引号指示字符串，就如同‘ Hello world ‘这样。所有的空白，即空格和制表符都照原样保留。
- **使用双引号（“）**：在双引号中的字符串与单引号中的字符串的使用完全相同，例如”What‘s your name?“。

# Python程序基本概念

- 使用三引号（'''或"""）：利用三引号，可以指示一个多行的字符串，可以在三引号中自由的使用单引号和双引号，如：

```
'''这是一个多行的
字符串，你可以写入
任意字符，甚至是
单引号'和双引号"
'''
```

# Python程序基本概念

## 变量

- 仅仅使用字面意义上的常量很快就会不能满足我们的需求——需要一种既可以储存信息又可以对它们进行操作（改变它的内容）的方法。这是为什么要引入 *变量* 。
- 变量的值可以*变化*，即可以使用变量存储任何东西。变量只是计算机中存储信息的一部分内存。与字面意义上的常量不同，需要一些能够访问这些变量的方法，因此要给变量命名

# Python程序基本概念

## 标识符的命名

变量是标识符的例子。*标识符* 是用来标识 *某样东西* 的名字。在命名标识符的时候，要遵循这些规则：

- 标识符的第一个字符必须是字母表中的字母（大写或小写）或者一个下划线（‘_’）。
- 标识符名称的其他部分可以由字母（大写或小写）、下划线（‘_’）或数字（0-9）组成。

# Python程序基本概念

- 标识符名称是对大小写敏感的。例如，myname和myName不是一个标识符。
- *有效* 标识符名称的例子有i、__my_name、name_23和a1b2_c3。
- *无效* 标识符名称的例子有2things、this is spaced out和my-name。

# Python程序基本概念

## 标识符的命名-关键字

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

# Python程序基本概念

- **标识符的命名-类保留（下划线的使用）**

  普通标识符为小写字母表示，类变量使用大写开头的字符串。以下划线开头的标识符是有特殊意义的。

  - _*：私有类名称，不能用'from module import *'导入
  - __*__：系统定义的名字， python里特殊方法专用的标识
  - __*：类的私有变量或方法。

# Python程序基本概念

## 数据类型

- 每个变量都有自己的类型，可以处理不同类型的值，称为**数据类型**。
- 基本的类型是数和字符串，在后面的章节里面还可以见到怎么用*类*创造自己的类型。
- Python中一切都是对象，包括字符串和数。

# Python程序基本概念

- 对象

Python把在程序中用到的任何东西都称为*对象*。Python是完全面向对象的语言，任何变量都是对象，甚至包括执行的代码：函数。>>>run var

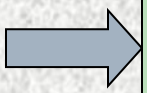
```
# Filename : var.py
i = 5
print(i)
i = i + 1
print(i)

s = '''This is a multi-line string.
This is the second line.'''
print(s)
```

# Python程序基本概念

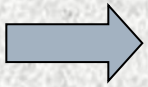
## 逻辑行与物理行

- 物理行是在编写程序时所 *看见* 的。逻辑行是Python *看见* 的单个语句。Python假定每个 *物 理行* 对应一个 *逻辑行*
- Python希望每行都只使用一个语句，这样使得代码更加易读
- 如果想要在一个物理行中使用多于一个逻辑行，那么需要使用分号（;）来特别地标明这种用法。分号表示一个逻辑行/语句的结束。例如：

```
i=5
print i
```



```
i=5;
print i;
```



```
i=5;print i
```

# Python程序基本概念

- **强烈建议坚持在每个物理行只写一句逻辑行**。
- 仅仅当逻辑行太长的时候，再多于一个物理行写一个逻辑行。这些都是为了尽可能避免使用分号，从而让代码更加易读。
- 下面是一个在多个物理行中写一个逻辑行的例子。它被称为明确的行连接。

```
s = 'This is a string. \
This continues the string.'
print s
```

```
This is a string. This continues the string.
```

# Python程序基本概念

- 有一种暗示的假设，可以不需要使用反斜杠。这种情况出现在逻辑行中使用了圆括号、方括号或波形括号的时候。这被称为暗示的行连接。

```
a = [100,
     200]
print a
```

# Python程序基本概念

- 缩进
  - 空白在Python中是重要的。事实上**行首的空白是重要的**。它称为**缩进**。在逻辑行首的空白（空格和制表符）用来决定逻辑行的缩进层次，从而用来决定语句的分组。
  - 这意味着同一层次的语句**必须**有相同的缩进。每一组这样的语句称为一个**块**。错误的缩进会引发错误
  - 不同于C/C++、Java用的是{}

```
i = 5
   print 'value is', i # Error! Notice a single space at the start of the line
print 'I repeat, the value is', i
```

# Python程序基本概念

- 如何缩进

  **不要**混合使用制表符和空格来缩进，因为这在跨越不同的平台的时候，无法正常工作。**强烈建议** 在每个缩进层次使用单个制表符或两个或四个空格 。

  选择这三种缩进风格之一。更加重要的是，选择一种风格，然后**一贯地**使用它，即只使用这一种风格。

# Python程序基本概念

- Python迫使程序员写成统一、整齐并且具有可读性的程序，这就意味着必须根据程序的逻辑结构，以垂直对齐的方式来组织程序代码，结果就是让程序更一致，并具有可读性，因而具备了重用性和可维护性，对自己和他人都是如此。

```
if (x)
    if (y)
        statements;
else
    statements;
```

```
if x:
    if y:
        statements
else:
    statements
```

# Python程序基本概念

- Python程序结构

1. 程序由模块构成　2.模块包含语句
3. 语句包含表达式　4.表达式建立并处理对象

（python中没有类型声明，运行的表达式决定了建立对象的类型，这点有点儿像matlab。）

Python语法实质上是有语句和表达式组成的。表达式处理对象并嵌套在语句中。语句编程实现程序操作中更大的逻辑关系。此外，语句还是对象生成的地方，有些语句会生成新的对象类型（函数、类等）。语句总是存在于模块中，而模块本身则又是由语句来管理的。

# Python程序基本概念

## Python语法

```
if (x > y) {
    x = 1;
    y = 2;
}
```

```
if x > y:
    x = 1
    y = 2
```

- Python增加了什么

  新的语法成分冒号（:）。所有的复合语句（语句中嵌套了语句）都有相同的一般形式，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。

- Python删除了什么
  - 括号是可选的
  - 终止行就是终止语句（分号）
  - 缩进的结束就是代码块的结束（}）

# 运算符及其用法

| 运算符 | 名称 | 说明 | 例子 |
| --- | --- | --- | --- |
| + | 加 | 两个对象相加 | 3 + 5得到8。'a' + 'b'得到'ab'。 |
| - | 减 | 得到负数或是一个数减去另一个数 | -5.2得到一个负数。50 - 24得到26。 |
| * | 乘 | 两个数相乘或是返回一个被重复若干次的字符串 | 2 * 3得到6。'la' * 3得到'lalala'。 |
| ** | 幂 | 返回x的y次幂 | 3 ** 4得到81（即3 * 3 * 3 * 3） |
| / | 除 | x除以y | 4/3得到1（整数的除法得到整数结果）。4.0/3或4/3.0得到1.3333333333333333 |
| // | 取整除 | 返回商的整数部分 | 4 // 3.0得到1.0 |
| % | 取模 | 返回除法的余数 | 8%3得到2。-25.5%2.25得到1.5 |
| << | 左移 | 把一个数的比特向左移一定数目（每个数在内存中都表示为比特或二进制数字，即0和1） | 2 << 2得到8。——2按比特表示为10 |
| >> | 右移 | 把一个数的比特向右移一定数目 | 11 >> 1得到5。——11按比特表示为1011，向右移动1比特后得到101，即十进制的5。 |

# 运算符及其用法（位运算符）

| 运算符 | 名称 | 说明 | 例子 |
| --- | --- | --- | --- |
| & | 按位与 | 数的按位与 | 5 & 3得到1。 |
| \| | 按位或 | 数的按位或 | 5 \| 3得到7。 |
| ^ | 按位异或 | 数的按位异或 | 5 ^ 3得到6 |
| ~ | 按位翻转 | x的按位翻转是-(x+1) | ~5得到-6。 |

# 运算符及其用法

| 运算符 | 名称 | 说明 | 例子 |
| --- | --- | --- | --- |
| < | 小于 | 返回x是否小于y。所有比较运算符返回1表示真，返回0表示假。这分别与特殊的变量True和False等价。注意，这些变量名的大写。 | 5 < 3返回0（即False）而3 < 5返回1（即True）。比较可以被任意连接：3 < 5 < 7返回True。 |
| > | 大于 | 返回x是否大于y | 5 > 3返回True。如果两个操作数都是数字，它们首先被转换为一个共同的类型。否则，它总是返回False。type(5)>type(3L)返回False。 |
| <= | 小于等于 | 返回x是否小于等于y | x = 3; y = 6; x <= y返回True。 |
| >= | 大于等于 | 返回x是否大于等于y | x = 4; y = 3; x >= y返回True。 |
| == | 等于 | 比较对象是否相等 | x = 2; y = 2; x == y返回True。x = 'str'; y = 'stR'; x == y返回False。x = 'str'; y = 'str'; x == y返回True。 |
| != | 不等于 | 比较两个对象是否不相等 | x = 2; y = 3; x != y返回True。 |

# 运算符及其用法

| 运算符 | 名称 | 说明 | 例子 |
| --- | --- | --- | --- |
| not | 布尔“非” | 如果x为True，返回False。如果x为False，它返回True。 | x = True; not x返回False。 |
| and | 布尔“与” | 如果x为False，x and y返回False，否则它返回y的计算值。 | x = False; y = True; x and y，由于x是False，返回False。在这里，Python不会计算y，因为它知道这个表达式的值肯定是False（因为x是False）。这个现象 称为短路计算。 |
| or | 布尔“或” | 如果x是True，它返回True，否则它返回y的计算值。 | x = True; y = False; x or y返回True。 |

# 运算符优先级—由高向低

| 运算符 | 描述 |
| --- | --- |
| 'expr' | 字符串转换 |
| {key:expr,...} | 字典 |
| [expr1,expr2...] | 列表 |
| (expr1,expr2,...) | 元组 |
| function(expr,...) | 函数调用 |
| x[index:index] | 切片 |
| x[index] | 下标索引取值 |
| x.attribute | 属性引用 |
| ~x | 按位取反 |
| +x，-x | 正，负 |
| x**y | 幂 |
| x*y，x/y，x%y | 乘，除，取模 |

| 运算符 | 描述 |
| --- | --- |
| x+y，x-y | 加，减 |
| x<<y，x>>y | 移位 |
| x&y | 按位与 |
| x^y | 按位异或 |
| x\|y | 按位或 |
| x<y，x<=y，x==y，x!=y，x>=y，x>y | 比较 |
| x is y，x is not y | 等同测试 |
| x in y，x not in y | 成员判断 |
| not x | 逻辑否 |
| x and y | 逻辑与 |
| x or y | 逻辑或 |
| lambda arg,...:expr | Lambda匿名函数 |

# 真值表

| 对象/常量 | 值 |
| --- | --- |
| "" | 假 |
| "string" | 真 |
| 0 | 假 |
| >=1 | 真 |
| <=-1 | 真 |
| ()空元组 | 假 |
| []空列表 | 假 |
| {}空字典 | 假 |
| None | 假 |

真值表在判断、循环等语句中应用广泛。

复合表达式

- and
  - 当计算a and b时，python会计算a，如果a为假，则取a值，如果a为真，则python会计算b且整个表达式会取b值。
- or
  - 当计算a or b时，python会计算a，如果a为真，则整个表达式取a值，如果a为假，表达式将取b值。
- not
  - 如果表达式为真，not为返回假，如为表达式为假，not为返回真。

# 给变量赋值

- 简单赋值，Variable(变量)=Value(值)。

- 多变量赋值，
  Variable1,variable2,...=Value1,Value2,...
  - a,b,c=1,2,3
  - a=[1,2,3];b,c,d=a
  - a=(1,2,3);b,c,d=a

# 给变量赋值

- 多变量赋值也可用于变量交换
  - a,b=b,a
- 多目标赋值，a=b=variable
- 自变赋值，如+=，-=，*=等。在自变赋值中，python仅计算一次，而普通写法需计算两次；自变赋值会修改原始对象，而不是创建一个新对象。

# 变量和基本的表达式

变量就是用来记录程序中的信息，它的特点：

- 变量如对象一样不需要声明。
- 变量在第一次赋值时创建。
- 变量在表达式中使用将被替换为他们的值。
- 变量在表达式中使用以前必须已经赋值。

# 内置数学工具和扩展

- 表达式操作符

  +、-、*、/、**

- 内置数学函数

  pow、abs

- 公用模块

  random、math等

- 专业扩展NumPy

  矩阵、向量处理等

# 数学内置函数和内置模块

- math模块-普通数学函数
- cmath模块-处理复数的模块

| | |
|---|---|
| 'acos', | 'log', |
| 'acosh', | 'log10', |
| 'asin', | 'phase', |
| 'asinh', | 'pi', |
| 'atan', | 'polar', |
| 'atanh', | 'rect', |
| 'cos', | 'sin', |
| 'cosh', | 'sinh', |
| 'e', | 'sqrt', |
| 'exp', | 'tan', |
| 'isinf', | 'tanh' |
| 'isnan', | |

| | |
|---|---|
| 'acos', | 'fsum', |
| 'acosh', | 'hypot', |
| 'asin', | 'isinf', |
| 'asinh', | 'isnan', |
| 'atan', | 'ldexp', |
| 'atan2', | 'log', |
| 'atanh', | 'log10', |
| 'ceil', | 'log1p', |
| 'copysign', | 'modf', |
| 'cos', | 'pi', |
| 'cosh', | 'pow', |
| 'degrees', | 'radians', |
| 'e', | 'sin', |
| 'exp', | 'sinh', |
| 'fabs', | 'sqrt', |
| 'factorial', | 'tan', |
| 'floor', | 'tanh', |
| 'fmod', | 'trunc' |
| 'frexp', | |

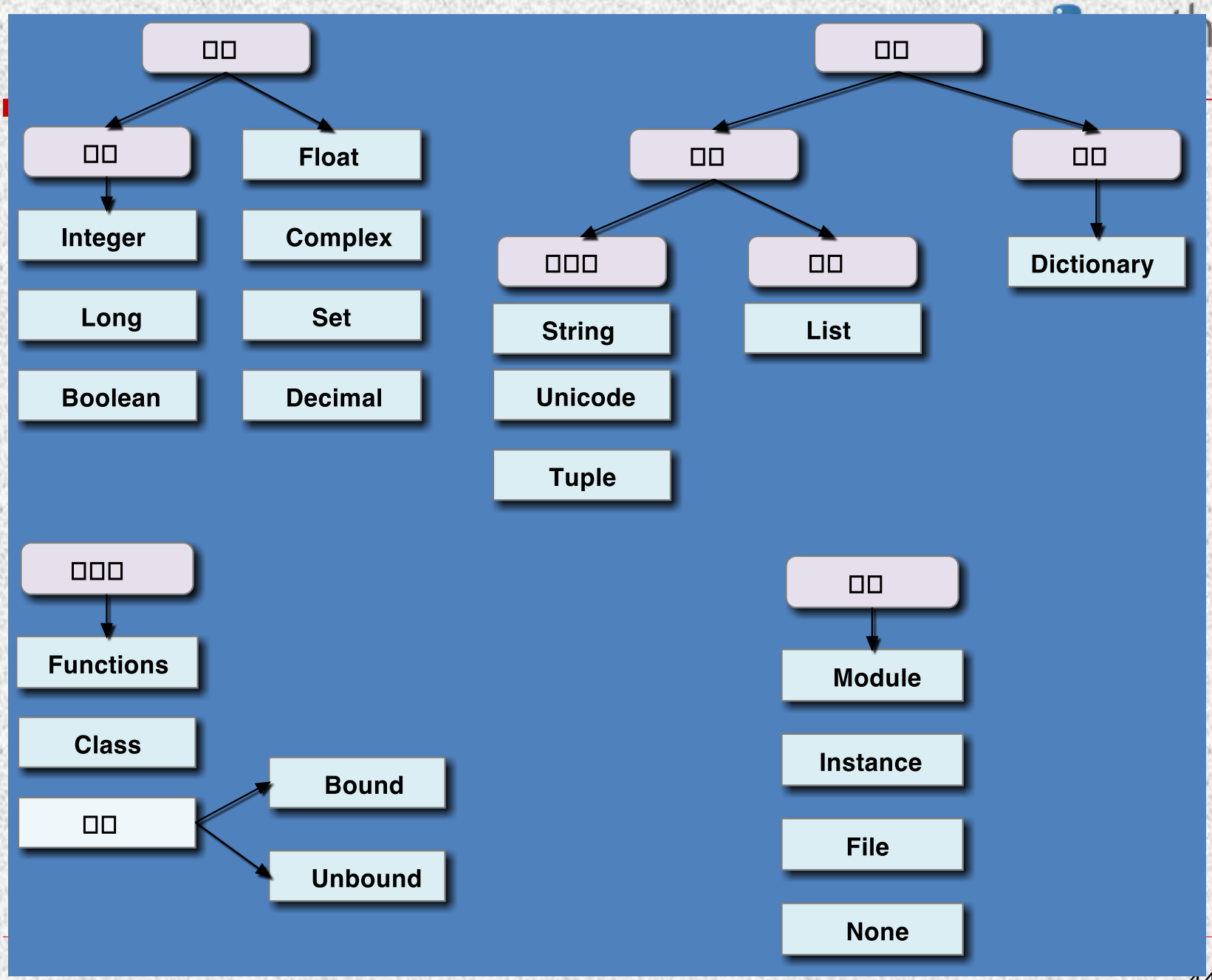
# 数学内置函数和内置模块

- random模块

  用于产生随机数

```
>>> import random
>>> random.random()
0.33452758558893336
>>> random.randint(1, 10)
5
>>> random.choice(['a', 'b', 'c'])
'c'
```

- Python的数据结构
  - 数字
  - 字符串
  - 列表
  - 元组
  - 字典


□□
□□
Float
Integer
Complex
Long
Set
Boolean
Decimal
□□
□□
□□
□□□
□□
Dictionary
String
List
Unicode
Tuple
□□□
□□
Functions
Module
Class
Instance
□□
Bound
File
Unbound
None

# 数字

- Python提供了常用的数字类型：整数、浮点数以及与之相关的语法和操作。
- 允许使用八进制、十六进制常量。
- 提供了复数类型。

```
>>> import sys
>>> print sys.maxint
2147483647
```

- 提供了无穷精度的长度类型（只要内存空间允许，可以增长成为任意位数的整数）。

# 数字常量

| 数字 | 常量 |
| --- | --- |
| 1234，-24，0 | 一般整数（c语言长整型） |
| 99999999999999999999999999L<br>98888888888888881 | 长整型数（无限大小） |
| 1.23，3.14e-10，4E210，4.0e+210 | 浮点数（C语言双精度浮点数） |
| 0177，0x9ff | 八进制、十六进制 |
| 3+4j，3.0+4.0j，3J | 复数常量 |

# 字符串的定义

- 字符串在python被看成是单个字符的序列，具有序列对象的特殊功能，字符串是固定的，不可变的。
- 可在字符串中使用单引号和双引号，注意要搭配。如‘boy’, ”girl”等。
- 字符串内部的一个反斜杠“\”可允许把字符串放于多行
- 也可以使用三个’或”使字符串跨行。
- 使用“*”号重复字符串，如：'hello'*3➔hellohellohello

```
>>> a = '12345\
... 67890'
>>> print a
1234567890
```

```
>>> a="""123456
... 7890"""
>>> print a
123456
7890
```

```
>>> a='hello'*3
>>> print a
hellohellohello
```

# 转义符

| 转义字符 | 描述 |
| --- | --- |
| \\ | 反斜杠符号 |
| \’ | 单引号 |
| \″ | 双引号 |
| \a | 响铃 |
| \b | 退格(Backspace) |
| \e | 转义 |
| \000 | 空 |
| \v | 纵向制表符 |
| \t | 横向制表符 |
| \r | 回车 |

| 转义字符 | 描述 |
| --- | --- |
| \n | 换行 |
| \(在行尾时) | 续行符 |
| \f | 换页 |
| \oyy | 八进制数yy代表的字符，例如：\o12代表换行 |
| \xyy | 十进制数yy代表的字符，例如：\x0a代表换行 |
| \other | 其它的字符以普通格式输出 |

不想让转义字符生效时，用r和R来定义原始字符串。
如：print r'\t\r' → \t\r

# 字符串基本操作

- +字符串合并
- *字符串重复

```
>>> len('abc')
3
>>> 'abc'+'def'
'abcdef'
>>> 'abc' 'def'
'abcdef'
>>> 'hello'*4
'hellohellohellohello‘
>>> 'abc'+9
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

# 字符串基本操作

- 可以用for语句在一个字符串中进行迭代，并使用in表达式操作符进行成员关系的测试，这实际上是一种搜索。

```
>>> s = 'hello'
>>> for c in s:
...     print c
...
h e l l o
>>> "h" in s
True
>>> "b" in s
False
```

for循环指派了一个变量去获取一个序列其中的元素，并对每一个元素执行一个或多个语句，变量c相当于在的指字符串中步进针。

# 字符串索引和分片

- 字符串是字符的有序集合，能够通过其位置来获得他们的元素
- Python中字符串中的字符是通过索引提取的
- 索引从0开始，但不同于C语言是可以取负值，表示从末尾提取，最后一个是-1，前一个是-2，依次类推，认为是从结束处反向计数.

```
>>> s = 'spam'
>>> s[0]
's'
>>> s[1]
'p'
>>> s[-1]
'm'
>>> s[-2]
'a'
```

# 字符串索引和分片

- 分片：从字符串中分离提取了一部分内容（子字符串）。
- 当使用一对以冒号分隔的偏移索引字符串这样的序列对象时，Python就返回一个新的对象，其中包含了以这对偏移所标识的连续的内容。
- 左边的偏移被取作是下边界（包含下边界在内），而右边的偏移被认为是上边界（不包括上边界在内）。
- 如果被省略上下边界的默认值分别对应为0和分片对象的长度。

```
>>> s = 'spam'
>>> s[1:3]
'pa'
>>> s[1:]
'pam'
>>> s[:-1]
'spa'
>>> s[:]
'spam'
```

# 分片的扩展形式

- 在Python2.3后，分片表达式增加了一个可选的第三个索引，用作步进选取
- 完整形式为：X[I:J:K],这表示：索引（获取）对象X中元素，从偏移为I直到J-1，每隔K元素索引一次
- K默认为1，这就是通常在切片中从左至右提取每个元素的原因
- 步进为负数表示将会从右至左进行而不是从左至右

# 分片的扩展形式

x[1:10:2]会取出x中，偏移量1-9之间，间隔一个元素的元素，即获取偏移量为1、3、5、7、9

```
>>> s = 'abcdefghijklmnop'
>>> s[1:10:2]
'bdfhj'
>>> s[::2]
'acegikmo'
```

```
>>> s = '0123456'
>>> s[::]
'0123456'
>>> s[::-1]
'6543210'
>>> s[::-2]
'6420'
>>> s[1:5:-1]
''
>>> s[5:1:-1]
'5432'
>>> s[9::-1]
'6543210'
>>> s[6:-1:-1]
''
>>> s[6:-2:-1]
'6'
```

# 字符串转化

- Python不允许字符串和数字直接相加。

```
>>> "15" + 1
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

- 这是有意设计的，因为+既能够进行加法运算也能够进行合并运算，这样的语法会变得模棱两可，因此，Python将其作为错误处理，在Python中，如果让操作变得复杂或含糊，就会避免这样的语法。

# 字符串转化

- 如果用户从文件或用户界面得到一个作为字符串的数字，怎么把这个字符串变为数字型呢？这就用到类型的转换函数

```
>>> s = '42'
>>> type(s)
<type 'str'>
>>> i = int(s)
>>> type(i)
<type 'int'>
>>> s1 = str(i)
>>> type(s1)
<type 'str'>
```

```
>>> s = '15'
>>> s + 1
Traceback (most recent call
last):
  File "<interactive input>",
line 1, in <module>
TypeError: cannot concatenate
'str' and 'int' objects
>>> int(s) + 1
16
```

通过明确的手动类型转换再进行+操作

# 字符串转化

- 常用的类型转换还有字符串到浮点型的转换。

```
>>> s = '15.0'
>>> float(s)
15.0
```

- 内置的eval函数，用于运行一个包含了Python表达式代码的字符串。(将字符串str当成有效的表达式来求值并返回计算结果。)

```
>>> eval('12')
12
>>> eval('12 + 3')
15
```

# 字符串代码转换

- 单个字符可以通过ord函数转换为对应的ASCII数值（整数）。
- chr函数相反，可以将一个整数转换为对应的字符。

```
>>> ord('a')
97
>>> chr(97)
'a'
```

# 修改字符串

- 字符串对象是“不可变序列”，不可变的意思是不能实地的修改一个字符串。

```
>>> s = 'spam'
>>> s[0] = 'x'
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- 那如何改变一个字符串呢？这就要利用合并、分片这样的工具来建立并赋值给一个新的字符串；必要的话，可以将结果赋值给字符串最初的变量名。

```
>>> s = 'spam'
>>> s = s + 'SPAM'
>>> s
'spamSPAM'
>>> s = s[:4] + 'OK!' + s[-1]
>>> s
'spamOK!M'
```

# 修改字符串

- 每修改一次字符串就生成一个新的字符串对象，这看起来好像会造成效率下降，其实，在Python内部会自动对不再使用的字符串进行垃圾回收，所以，新的对象重用了前面已有字符串的空间。
- Python的效率比我们想象的要好。

# 字符串格式化

- Python可以用%操作符编写格式化的字符串
- 格式化字符串：
  - 1、在%操作符左侧放置一个需要进行格式化的字符串，这个字符串带有一个或多个嵌入的转换目标，都以%开头，如%d、%f等
  - 2、在%操作符右侧放置一个对象（或多个，在括号内），这些对象会被插入到左侧格式化字符串的转换目标的位置上

```
>>> bookcount = 10
>>> "there are %d books"%bookcount
'there are 10 books'

"That is %d %s bird!" % (1, 'dead')
'That is 1 dead bird!'
```

# List列表

- 列表是Python中最具灵活性的有序集合对象类型。和字符串不同的是，列表可以包含任何种类的对象：数字、字符串、自定义对象甚至其他列表。
- 与其他高级语言的数组列表相似，Python中的列表是可变对象。列表支持在原处修改，可以通过指定的偏移值和分片、列表方法调用、删除语句等方法实现。

# 列表的主要性质

- 任意对象的有序集合

  从功能是看，列表就是收集其他对象的地方，可以他们**看成数组**；同时，列表所包含的每一项都保持了从左到右的位置顺序（也就是说，它们**是序列**）。

- 通过偏移读取

  和字符串一样，可以通过列表对象的偏移对其进行索引，从而读取对象的一部分内容。当然也可以执行诸如分片和合并之类的操作。

# 列表的主要性质

- 可变长度、异构以及任意嵌套

  和字符串不同，列表可以根据需要增长或缩短（长度可变），并且可以包含任何类型的对象，并支持任意的嵌套。

- 可变序列

  列表支持在原处的修改，也可以响应所有针对字符串序列的操作，如索引、分片以及合并。实际上，序列操作在列表与字符串中工作方式相同。唯一区别是：当合并或分片应用于列表时，返回新的列表而不是新的字符串。当然，支持某些字符串不支持的操作(如：删除和索引赋值操作，即在原处修改列表）。

# 常用列表常量和操作

| 操作 | 解释 |
| --- | --- |
| L1=[] | 一个空的列表 |
| L2 = [0, 1, 2, 3] | 四元素列表 |
| L3 = ['abc',10,['def', 'ghi']] | 嵌套列表 |
| L2[i] | 索引 |
| L3[i][j] | 索引的索引 |
| L2[i:j] | 分片 |
| len(L2) | 求长度 |
| L1 + L2 | 合并 |
| L2 * 3 | 重复 |

# 列表的方法

- append(x)

  把一个元素添加到列表的结尾，相当于a[len(a)] = [x]

- extend(L)

  通过添加指定列表的所有元素来扩充列表，相当于a[len(a):]=L

- insert(i,x)

  在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如a.insert(0,x)会插入到整个链表之前，而a.insert(len(a), x)相当于a.append(x)

# 列表的方法

- remove(x)

  删除链表中值为x的第一个元素。如果没有这样的元素，就会返回一个错误。

- pop([i])

  从链表的指定位置删除元素，并将其返回。如果没有指定索引，a.pop()返回最后一个元素。元素随即从链表中被删除。（方法中i两边的方括号表示这个参数是可选的，而不是要求输入一对方括号，会经常在Python库参考手册中遇到这样的标记。）

# 列表的方法

- index(x)

  返回链表中第一个值为x的元素的索引。如果没有匹配的元素就会返回一个错误。

- count(x)

  返回x在链表中出现的次数。

- sort()

  对链表中的元素进行适当的排序。

- reverse()

  倒排链表中的元素。

# 列表的方法

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

# List对象的操作

| 方法 | 描述 |
| --- | --- |
| append(x) | 在列表尾部追加单个对象x。使用多个参数会引起异常。 |
| count(x) | 返回对象x在列表中出现的次数。 |
| extend(L) | 将列表L中的表项添加到列表中。返回None。 |
| index(x) | 返回列表中匹配对象x的第一个列表项的索引。无匹配元素时产生异常。 |
| insert(i,x) | 在索引为i的元素前插入对象x。如list.insert(0,x)在第一项前插入对象。返回None。 |
| pop(x) | 删除列表中索引为x的表项，并返回该表项的值。若未指定索引，pop返回列表最后一项。 |
| sort() | 对列表排序，返回none。bisect模块可用于排序列表项的添加和删除。 |
| remove(x) | 删除列表中匹配对象x的第一个元素。匹配元素时产生异常。返回None。 |
| reverse() | 颠倒列表元素的顺序。 |

```
help(list)
help(list.count)
```

# 把列表当作堆栈使用

- 列表方法使得列表可以很方便的做为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放（后进先出）。

  用append() 方法可以把一个元素添加到堆栈顶。用不指定索引的pop()方法可以把一个元素从堆栈顶释放出来。

- 举例：

```
>>> st = [3, 4, 5]
>>> st.append(6)
>>> st.append(7)
>>> st
[3, 4, 5, 6, 7]
>>> st.pop()
7
>>> st
[3, 4, 5, 6]
>>> st.pop()
6
>>> st.pop()
5
>>> st
[3, 4]
```

# 把列表当作队列使用

- 也可以把列表当做队列使用，队列作为特定的数据结构，最先进入的元素最先释放（先进先出）。使用append()方法可以把元素添加到队列最后，以0为参数调用pop()方法可以把最先进入的元素释放出来。

```
>>> queue = ['a', 'b', 'c']
>>> queue.append('d')
>>> queue.append('e')
>>> queue
['a','b', 'c', 'd', 'e']
>>> queue.pop(0)
'a'
>>> queue
['b', 'c', 'd', 'e']
>>> queue.pop(0)
'b'
>>> queue
['c', 'd', 'e']
```

# 删除列表元素

- 可以用del进行

  可以删除某个索引的元素或切片元素

```
>>> lst = [1, 2, 3]
>>> lst
[1, 2, 3]
>>> lst[1]=5
>>> lst
[1, 5, 3]
>>> del lst[1]  # lst.pop(1)
>>> lst
[1, 3]
>>> lst.append(4)
>>> lst
[1, 3, 4]
>>> del lst[0:]
>>> lst
[]
```

# Tuple元组

列表和字符串有很多通用的属性，例如索引和切片操作。它们是序列类型中的两种。因为Python是一个在不断进化的语言，也会加入其它的序列类型，另一种标准序列类型：元组。

# 元组简介

- 一个元组由数个逗号分隔的值组成，例如：

```
>>> t = 12345, 54321, 'hello'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello')
>>> u = t, (1, 2, 3)
>>> u
((12345, 54321, 'hello'), (1, 2, 3))
```

如上所示，元组在输出时总是有括号的，以便于正确表达嵌套结构。

在输入时，有或没有括号都可以，不过经常括号都是必须的（如果元组是一个更大的表达式的一部分）。

# 元组

- 元组有很多用途。例如(x, y)坐标点，数据库中的员工记录等等。
- 元组就像字符串，不可改变：不能给元组的一个独立的元素赋值（尽管可以通过联接和切片来模仿）
- 可以通过包含可变对象来创建元组，例如列表。

```
>>> lst = [1, 2, 3]
>>> t = tuple(lst)
>>> t
(1, 2, 3)
```

# 元组

- 一个特殊的问题是构造包含零个或一个元素的元组：为了适应这种情况，语法上有一些额外的改变。一对空的括号可以创建空元组；要创建一个单元素元组可以在值后面跟一个逗号（在括号中放入一个单值是不够的）。丑陋，但是有效。

```
>>> emp = ()
>>> emp
()
>>> single = 'ab',  #<-- note trailing
comma
>>> len(emp)
0
>>> len(single)
1
>>> single
('ab',)
```

# 元组封装和解封

- 语句t = 12345, 54321, ’hello!’ 是元组封装（sequence packing）的一个例子：值12345，54321 和’hello!’ 被封装进元组。其逆操作可能是这样：

```
>>> t = 12345, 54321, 'hello!'
>>> print t
  (12345, 54321, 'hello!')
>>> x,y,z=t
>>> print x,y,z
12345 54321 hello!
```

这个调用被称为序列拆封非常合适。序列拆封要求左侧的变量数目与序列的元素个数相同。

# 元组封装和解封

- 拆封和封装有一点不对称：封装多重参数通常会创建一个元组，而拆封操作可以作用于任何序列。

```
>>> t = [1, 2, 3]
>>> x, y, z = t
>>> print x, y, z
1 2 3

>>> s = "123"
>>> x,y,z = s
>>> print x, y, z
1 2 3
```

help(tuple)
help(tuple.count)

# 序列对象

- 字符串、列表和元组的对象类型均属于称为序列的Python对象,一种可使用数字化索引进行访问其中元素的对象 。
- 可用算术运算符联接或重复序列。 (+/*)
- 比较运算符(<，<=，>，>=，!=，==)也可用于序列 。
  - a="123"; b="456"　a和b的比较关系

# 序列对象

- 可通过下标，切片和解包来访问序列的某部份。
  - a=“123456”
  - print a[1],a[3:],a[:3],a[2:4]
- in运算符可判断当有对象是否序列对象成员。
  - '1' in a
- 也可通过循环运算符对序列对象进行迭代操作。
  - for x in range(1,10):

# Dictionary字典

- 另一个非常有用的Python内建数据类型是字典。字典在某些语言中可能称为“联合内存”（“associative memories”）或“联合数组”（“associative arrays”）。
- 字典类似于通过联系人名字查找地址和联系人详细情况的地址簿，即：我们把**键**（名字）和**值**（详细情况）联系在一起。注意，键必须是唯一的，就像如果有两个人恰巧同名的话，将无法找到正确的信息。

# 字典

- 序列是以连续的整数为索引，与此不同的是，字典以关键字为索引。
- 关键字可以是任意不可变类型，通常用字符串或数值。如果元组中只包含字符串和数字，它可以做为关键字，如果它直接或间接的包含了可变对象，就不能当做关键字。
- 不能用列表做关键字，因为列表可以用它们的append()和extend()方法，或者用切片、或者通过检索变量来即时改变。
- 基本说来，应该只使用简单的对象作为键。

# 字典

- 理解字典的最佳方式是把它看做无序的(关键字:值)对集合，关键字必须是互不相同的（在同一个字典之内）。
- 一对大括号创建一个空的字典：{}。
- 字典的主要操作是依据关键字来存储和析取值。也可以用del来删除(关键字:值)对。
- 如果使用一个已经存在的关键字存储新的值或对象，以前为该关键字分配的值就会被遗忘。
- 试图析取从一个不存在的关键字中读取值会导致错误。

# 字典

- 字典的keys()方法返回由所有关键字组成的列表，该列表的顺序不定（如果需要它有序，只能调用返回列表的sort()方法）。
- 使用字典的has_key()方法可以检查字典中是否存在某一关键字。
- 字典的values()方法返回字典内所有的值。
- 字典的get()方法可以根据关键字返回值，如果不存在输入的关键字，返回None。

# 字典例子

```
>>> tel = {'jack':4098, 'shy':4139}
>>> tel['gree'] = 4127
>>> tel
{'gree': 4127, 'jack': 4098, 'shy': 4139}
>>> tel['jack']
4098
>>> del tel['shy']
>>> tel
{'gree': 4127, 'jack': 4098}
>>> tel['irv'] = 4127
>>> tel
{'gree': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['jack', 'irv', 'gree']
>>> tel.has_key('gree')
True
>>> tel.has_key('lee')
False
>>> tel.values()
[4098, 4127, 4127]
>>> tel.get('irv')    # tel['irv']
4127
```

# 字典

- 字典的update(*anothordict*)方法类似于合并，它把一个字典的关键字和值合并到另一个，盲目的覆盖相同键的值。

```
>>> tel
{'gree': 4127, 'irv': 4127, 'jack': 4098}
>>> tel1 = {'gree':5127, 'pang':6008}
>>> tel.update(tel1)
>>> tel
{'gree': 5127, 'irv': 4127, 'jack': 4098, 'pang':
6008}
```

# 字典

- 字典的pop()方法能够从字典中删除一个关键字并返回它的值，类似于列表的pop方法，只不过删除的是通过一个关键字而不是位置。

```
>>> tel
{'gree': 5127, 'irv': 4127, 'jack': 4098, 'pang':
6008}
>>> tel.pop('gree')
5127
>>> tel
{'irv': 4127, 'jack': 4098, 'pang': 6008}
>>> tel.pop('li')
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
KeyError: 'li'
```

# dictionary对象的操作

| 方法 | 描述 |
| --- | --- |
| has_key(x) | 如果字典中有键x，则返回真。 |
| keys() | 返回字典中键的列表 |
| values() | 返回字典中值的列表。 |
| items() | 返回tuples的列表。每个tuple由字典的键和相应值组成。 |
| clear() | 删除字典的所有条目。 |
| copy() | 返回字典高层结构的一个拷贝，但不复制嵌入结构，而只复制对那些结构的引用。 |
| update(x) | 用字典x中的键值对更新字典内容。 |
| get(x[,y]) | 返回键x的值，若未找到该键返回none，若提供y，则未找到x时返回y。 |

help(dict)

- 流程控制的语句
  - if
  - while
  - for
  - break
  - continue

# 控制流简介

在Python中有三种控制流语句——if、for和while。

## If语句

if语句是选取要执行的操作，是Python主要的选择工具，代表Python程序所拥有的大多数逻辑。

if语句是复合语句，同其他复合语句一样，if语句可以包含其他语句

```
if <test1>:
    <statements1>
elif <test2>:
    <statements2>
else:
    <statements3>
```

# if 的例子

```
#coding:utf-8
number = 23
guess = int(raw_input('Enter an integer : '))
if guess == number:
    print 'Congratulations, you guessed it.' # New block
starts here
    print "(but you do not win any prizes!)" # New block
ends here
elif guess < number:
    print 'No, it is a little lower than that' # Another
block
    # You can do whatever you want in a block ...
else:
    print 'No, it is a little higher than that'
    # you must have guess > number to reach here
print 'Done'
# This last statement is always executed, after the if
statement is executed
```

# if

- Python中没有switch、case语句
- 可以用多个if实现，或者对字典进行索引运算或搜索列表，因为字典和列表可在运行时创建，有时会比硬编码的if逻辑更有灵活性。

# 字典实现switch

```
choice = 'ham'
dic =  {'spam': 1.25,
        'ham': 1.99,
        'eggs': 0.99,
        'bacon': 1.10}
print dic[choice]
```

字典适用于将值和键相关联，值也可以是函数，因此可以用于更多灵活的处理。

```
if choice == 'spam':
    print 1.25
elif choice == 'ham':
    print 1.99
elif choice == 'eggs':
    print 0.99
elif choice == 'bacon':
    print 1.10
else:
    print 'bad choice'
```

# while、for

- while、for用于提供循环的控制功能
- while一般格式：

```
while <test>:          #Loop test
      <statements1>    #Loop body
else:                  #Optional else
      <statements2>    #Run if didn't exit loop with break
```

```
a = 0
b = 10
while a<b:
      print a
      a = a + 1
```

# while例子

```
number = 23
running = True
while running:
    guess = int(raw_input('Enter an integer : '))
    if guess == number:
        print 'Congratulations, you guessed it.'
        running = False # this causes the while loop to
stop
    elif guess < number:
        print 'No, it is a little lower than that'
    else:
        print 'No, it is a little higher than that'
else:
    print 'The while loop is over.'
    # Do anything else you want to do here
print 'Done'
```

# 中断循环

- 在循环进行中，如果满足一定条件而中断整个循环或本次循环，可以使用break或continue。
- break语句是用来 终止 循环语句的，哪怕循环条件没有称为False或序列还没有被完全递归，也停止执行循环语句。

- 注意的是：如果从for或while循环中终止，任何对应的循环else块将不执行。

# break的例子

```
while True:
    s = raw_input('Enter
something : ')
    if s == 'quit':
        break
    print 'Length of the string
is', len(s)
else:
    print 'The while loop is over.'
print 'Done'
```

# continue

- 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

```
while True:
    s = raw_input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print 'Input is of sufficient
length'
        continue
        print '54321'
    # Do other kinds of processing here...
```

# for

- for循环在Python中是一个通用的序列迭代器：可以遍历任何有序的序列对象内的元素。
- for语句可用于字符串、列表、元组、其他内置可迭代对象，以及用户通过类创建的新对象。

# for一般格式

- for循环首行定义一个赋值目标，以及想遍历的对象；首行后面是想重复的语句块。

```
for <target> in <object>:#Assign object items to target
      <statements>  #Repeated loop body:use target
else:
      <statements>  #If we didn't hit a 'break'
```

- 运行for循环时，会逐个将序列对象中的元素赋值给目标，然后为每个元素执行循环主体。循环主体一般使用赋值目标来引用序列中当前元素。

# for完整格式

```
for <target> in <object>:#Assign object
items to target
     <statements>    #Repeated loop
body:use target
     if <test>: break #Exit loop now,
skip else
     if <test>: continue #Go to top of
loop now
else:
     <statements>    #If we didn't hit a
'break'
```

# for例子

```
>>> for x in ['a', 'b', 'c']:
...     print x
...
a
b
c
```

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
```

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

# for例子

```
def text_create():
    path = 'E:/python/'        #给变量path赋值为路径；
    for text_name in range(1,11): # 将1-10范围内的每个数字依次装入变量text_name中，每次命名一个文件；
        with open (path + str(text_name) + '.txt','w') as text:      # 打开位于路径的txt文件，并给每一个text执行写入操作；
            text.write(str(text_name+9)) #给每个文件依次写入；
            text.close()
            print('Done')

text_create()
```

# for例子

在**Python**中，“**with...as**”语法是用来代替传统的“**try...finally**”的。

```
file = open('E:/python/1.txt')
try:
    data = file.read()
finally:
    file.close()

print data
```

```
with open('E:/python/1.txt') as file:
    data = file.read()
    file.close()

print data
```

# range

- range用来产生整数列表
- range([start], stop[, step])

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```