

Intro to Python

Jiatong

为什么学？

- 当今最为流行的编程语言
- Python = 大数据分析、大数据分析利器？
- 直译式、交谈式 (Interpreted, Interactive)
- 与多种语言接口
- 易学易用 (Easy Learning)
- 大量的开源代码(Open Source)

Python语言基础

1. 简介与安装:
2. 数据类型: Number、String、List、Dictionary、Tuple
3. 控制语句: 赋值、表达式、Print、If、While、For
4. 函数: 作用域、变量传递、return、lambda、map
5. 模块与包: 概念、import(from)、reload、__name__
6. 对象与类: 运算符重载

python?

- 快速学习基本语法
- 主要学习方式
 - 「依样画葫芦」
 - 「应用于日常生活」
- 实践是王道

1. 简介与安装

软件安装

- Python安装
- Pip安装包
- PyCharm安装
- Anaconda安装
- Jupyter Notebook

pip

- 1 pip下载: <https://pypi.python.org>
- 2 pip安装:
 - tar -xzvf pip-1.5.4.tar.gz
 - python setup.py install
- 3. 安装包:
 - pip install SomePackage
 - pip install wheel
 - pip install VTK-5.10.1+qt486-cp27-none-win32.whl
- 4. 查看包: pip show --files SomePackage
- 5. 检查更新: pip list --outdated
- 6. 升级包: pip install --upgrade SomePackage
- 7. 卸载包: pip uninstall SomePackage

pip

- 从哪里下
 - 官网: <http://e.pypi.python.org>
 - 豆瓣: <http://pypi.douban.com/simple/>
 - 清华: <https://pypi.tuna.tsinghua.edu.cn/simple>
- 怎么下
 - `pip install -i https://pypi.tuna.tsinghua.edu.cn/simple gevent`
 - windows下, 直接在user目录中创建一个pip目录, 如: C:\Users\xx\pip, 新建文件**pip.ini**, 内容如下
 - `[global] index-url = https://pypi.tuna.tsinghua.edu.cn/simple`
- 下在哪里
 - Lib-Sitepackage

2. 数据类型

- Number
- String
- List
- Dictionary
- Tuple

Python的程序架构

- Python语言的组成系统, 可以分解成模块(Module), 叙述(Statement), 和物件(Object)
- 程序由模块文件组成
- 模组文件包含了叙述
- 叙述建构了对象并处理对象

内部对象形态

对象形态		范例
数值	Number	3.1415, 1234, 999L, 3+4j
字符串	String	'LaRC' , "NTHU" s"
序列	List	[1, [2, 'three'], 4]
字典	Dictionary	{ 'food' : 'spam' , 'taste' : 'yum' }
元组	Tuple	(1, 'LaRC' , 4, 'U')
文件	File	text = open('eggs' , 'r').read()

3.1 Number

Number-Introduction

- 整数, and 浮点数两大类
- 与C语言相较, 多了复数and精确度无限的长整数
- 常见的数值常数

常数	说明
1234, -24, 0	一般整数
999999999999999L	长整数
1.23, 3.14e-10, 4E210, 4.0e+210	浮点数
0177, 0x9ff	8进位和16进位常数
3+4j, 3.0+4.0j, 3J	复数常数

Number-运算符以及优先权(1/3)

- 越下面的项目优先权越高
- 下列的运算符并不是所有的都跟数值有关

运算符	说明
x or y,	or的逻辑运算,
x and y	and的逻辑运算
not x	否定的逻辑运算
<, <=, >, >=, ==, <>, !=	比较运算符
is, is not	对等测试
in, not in	序列的成员关系

Number-运算符以及优先权(2/3)

运算符	说明
$x + y, x - y$	加法, 减法
$x * y, x / y, x \% y$	乘法, 除法, 余数运算
$-x, +x, \sim x$	变号, 本身, 补码的位运算
$x[i], x[i:j],$ $x.y, x(...)$	索引, 切片, 名称评定用法, 函数调用
$(...), [...], \{...\},$ \backslash, \dots	Tuple, 序列, 字典, 转换成字符串

Number-实例操演(1/3)

- 基本运算 加减乘除没问题的啦~

%Python

```
>>> a = 3                      #建立名称
```

```
>>> b = 4
```

```
>>> b / 2 + a                  #等于 ((4/2)+3)
5
```

```
>>> b / (2.0 + a)              #等于 (4/(2.0+3))
0.8
```

- 运作方式
 - 先算 $2.0 + a$, 看到 2.0 , 将 a 的 $3 \Rightarrow 3.0$, 得到 5.0
 - 再算 $b / 5.0$, 看到 5.0 , 将 b 的 $4 \Rightarrow 4.0$, 得到 0.8
 - If 使用 2 来计算, 则 $4/5$ 会得到 0

Number-实例操演(3/3)

- 复数

```
>>> 1j * 1j  
(-1 + 0j)
```

```
>>> 2 + 1j * 3  
(2+3j)
```

```
>>> (2 + 1j) * 3  
(6+3j)
```

- 其他的工具

- 你可以在math模块里面找到

```
>>> import math                #类似C中的#include<xxx.h>
```

```
>>> math.pi                    #  $\pi$   
3.14159265359
```

```
>>> abs(-42), 2**4, pow(2,4)  
(42, 16, 16)
```

3.2 String

String-Introduction

- 字符的集合
- Immutable sequence 不可变更序列
 - 内容于指定后不可被更换
 - 其实是Lists里的特例
- 常见的字符串常数
 - `S1 = ''` 空字符串
 - `S2 = "spam's"` 双引号
- 双引号 or 单引号 ?? 都可以的啦 !!

String-常见的字符串运算

- 基本运算

- S1 + S2 串接
- S2 * 3 重复串接
- for x in S2 循环的迭代
- ' m ' in S2 成员关系

- 索引参考和切片运算

- S2[i] 索引参考
- S2[i : j] 切片运算
- len(S2) 求字符串长

- 内容变更与书写格式

- " a %s parrot " % ' dead ' 字符串的输出方式

String-基本运算(1/2)

- 长度：字符个数

```
>>> len ( ' LaRC ' )  
4
```

- 串接：形成新字符串

```
>>> ' LaRC ' + ' EE '  
' LaRCEE '
```

- 重复串接：等于 ' LaRC ' + ' LaRC ' + ' LaRC '

```
>>> ' LaRC ' * 3  
' LaRCLaRCLaRC '
```

- Python 看不懂的东西：

```
- ' LaRC ' + 3          #字符串与数字混用
```

String-基本运算(2/2)

- 有没有包括：判断成员关系是否成立

```
>>> mylab = "LaRC"
```

```
>>> "R" in mylab
```

```
1
```

```
# means true
```

```
>>> for letter in mylab: #循环的迭代
```

```
...     print letter
```

```
L
```

```
#一次print一个字母
```

```
a
```

```
R
```

```
C
```

String-索引|参考&切片运算(1/3)

- 基本上与C类似, 都从 $[0] \sim [n-1]$
- 增加了负值的表示方法 $[-n] \sim [-1]$

– $\text{Offset}_p = \text{Offset}_n + n$

```
>>> mylab = " LaRC "
```

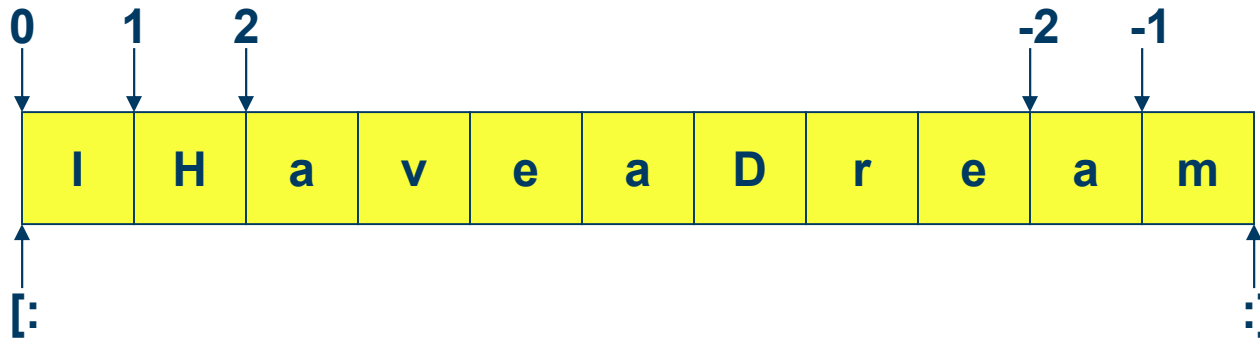
```
>>> mylab[0], mylab[-2]    #前面&后面索引  
( ' L ' , ' R ' )        #tuple
```

```
>>> mylab[1:3]             #切片运算  
' aR '
```

- 预设边界值 $[(\text{lower bound}), :] [: (\text{upper bound})]$

```
>>> mylab[1:], mylab[:-1]  #省略写法  
( ' aRC ' , ' LaR ' )    #tuple
```

String-索引参考&切片运算(2/3)



- 索引参考 (S[i])
 - 以偏移量把字符读出来
 - 负索引值是从字符串的尾端倒数回来
 - $S[-2] = S[\text{len}(S)-2]$
- 切片运算 (S[i:j])
 - 从序列中把某一片断的节区抽取出来
 - 切片的分界值预设是0和 $\text{len}(S)$
 - $S[:-1]$ 会把除了最后一个字符以外的都包含进来

String-索引|参考&切片运算(3/3)

- 切片的实务用法1：自变量的取出

echo.py

```
Import sys                                #类似C中的#include<xxx.h>
Print sys.argv
Print sys.argv[1:]
```

```
>>> python echo.py -a -b -c
[ 'echo.py', '-a', '-b', '-c' ]
[ '-a', '-b', '-c' ]
```

- 切片的实务用法2：清理输入一系列文字的结尾字符

```
>>> LaRC = "Lab for Reliable Computing\n"
>>> LaRC[:-1]
"Lab for Reliable Computing"
```

String-内容变更

- 不可变更?!

```
>>> mylab = 'LaRC'
```

```
>>> mylab[3] = "K"
```

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

```
File "<pyshell#9>", line 1, in <code>-
```

```
TypeError: object doesn't support item assignment
```

- 山不转, 路转 !!

— 转变成新的字符串不就得了~

```
>>> mylab = mylab[:-1] + "K!"
```

```
>>> mylab  
LaRK!
```

```
>>> mylab = mylab[:4] + " is a bird " + mylab[-1]  
LaRK is a bird!
```

String-书写格式(1/3)

- >>> 'That is %d %s!' % (1, 'bird')

– 'That is 1 bird!'

- 格式转换符号:

%s 字符串	%X 16进位整数(大写)
%c 字符	%e 浮点数科学符号(小写)
%d 10进位整数	%E 浮点数科学符号(大写)
%x 16进位整数	%f 浮点数固定位数
%u 无号整数	%% 印出%字符
%o 8进位整数	

String-书写格式(2/3)

```
>>> Mylab = " LaRC "  
>>> " I am in %s now! " % Mylab  
    ' I am in LaRC now! '  
>>> " %d %s %d = ? " % (1, '+', 1)  
    ' 1 + 1 = ? '  
>>> " %s - %s - %s " % (33, 3.1415926, [1, 2, 3])  
    ' 33 - 3.1415926 - [1, 2, 3] '
```

- %s这个符号会将所有的对象型态都转成字符串

```
>>> l = 3.1415926  
>>> l = "%s" % l  
    ' 3.1415926 '
```

- %永远传回一个新的字符串

额外的说-常用的字符串工具

```
>>>import string
>>>S = " immediaTely "
>>>string.upper(S)          ' IMMEDIATELY '
>>>string.lower(S)          ' immediately '
>>>string.find(S, 'mm')      #传回'mm'在string里的索引
1
>>>string.atoi("151"), `151` #非常常用的对象型态转换方式
( 151, ' 151 ' )
>>> "LaRC" + `151`, string.atoi("151") + 2
(LaRC151, 153)
>>>string.join('abc','xyz')
'axyzbxyzc'
>>>string.join(string.split(S, 'mm'), 'll') #以mm分割, 再用ll合并起来
'illediaTely'
```

String-书写格式(3/3)

- 几个变体

- >>> SARS = "We" "are" "the" " World"
- >>> SARS
 - 'WearetheWorld'
- >>> Big = """ " Oh, what is this ?
- ... It is too big !!!" " "
- >>> Big
 - 'Oh, what is this ? \n It is too big !!!'

\n	换行字符		
\\	倒斜线	\v	竖向跳格
\'	单引号	\t	横向跳格
\"	双引号	\r	返回字符
\b	倒退字符	\0XX	8进位数值
\000	空字符	\xXX	16进位数值

3.3 List

List-Introduction

- 最富弹性的对象型态
 - 字串只能文字, 序列什么都能存(数值, 文字, 甚至序列)
- 经由索引值来参考
 - 次序由左至右排列, 有前后关系
 - 与字串相同, 可透过索引值运算=>可切片运算,串接
- 长度可变, 无限巢状扩展
 - 内容可增可减, 长度无限制
 - 可以玩 “序列的序列的序列” 的复杂结构
- 属于可变更序列之一

List-常见的序列运算(1/2)

- 基本运算

- L1 = [] #空字符串
- L2 = [0, 1, 2, 3] #4个项目的序列
- L3 = ['abc' , ['def' , 'ghi']] #巢状序列
- len (L2) #长度
- L1 + L2 #串接
- L2 * 3 #重复串接
- For x in L2 #循环反复
- 3 in L2 #成员关系成立与否

- 索引值参考和切片运算

- L2[i], L3[i][j] #索引值参考
- L2[i:j] #切片运算

List-常见的序列运算(2/2)

- 变更序列内容 <~~~~~不同于字符串的
 - L2.append(4) #项目增加
 - L2.sort() #L2变更
 - L2.index(1) #搜寻
 - L2.reverse() #反向
 - del L2[k] #项目减少
 - L2[i : j] = []
 - L2[i] = 1 #索引值指定
 - L2[i : j] = [4, 5, 6] #切片指定
 - range(4) #建立整数的序列

List-基本运算(1/2)

- 长度

- ```
>>> len([1, 2, 3])
- 3
```

- 串接

- ```
>>> [ 1, 2, 3 ] + [ 4, 5, 6 ]
- [ 1, 2, 3, 4, 5, 6 ]
```

- 重复串接

- ```
>>> ['Oh~'] * 4
- ['Oh~', 'Oh~', 'Oh~', 'Oh~']
```

- 循环迭代

- ```
>>> for x in [ 1, 2 ]:
...     print x
- 1
- 2
```

List-基本运算(2/2)

- 字符串与序列
- Ans. 请归化异族= = +
 - 想归化字符串, 请用``
 - 想归化序列, 请用list()

```
>>> `[ 1, 2 ]` + "34"          #等于 "[1,2]" + "34"  
      '[ 1, 2 ]34'
```

```
>>> [ 1, 2 ] + list( " 34 " ) #等于 [1, 2] + [ "3" , "4" ]  
      [ 1, 2, '3' , '4' ]
```

List-索引值参考和切片运算

- 跟字符串的运算方法一样

```
>>> mylab = [ 'larc' , 'Larc?' , 'LaRC!' ]
```

```
>>> mylab[ 2 ]  
'LaRC!'
```

```
>>> mylab[ -2 ]  
'Larc?'
```

```
>>> mylab[ 1: ]           #切片运算取出片段内容  
[ 'Larc?' , 'LaRC!' ]
```

- 切片运算的结果, 传回来的都是新的序列

List – 变更序列内容(1/3)

- 以新的对象参考地址来替代原先的

```
>>> mylab = [ 'larc' , 'Larc?' , 'LaRC!' ]  
>>> mylab[ 1 ] = 'is'           #索引式的指定运算  
>>> mylab  
[ 'larc' , 'is' , 'LaRC!' ]
```

- 切片指定运算: 删除对象+新增物件+插入新参考地址

```
>>> mylab[ 0 : 2 ] = [ "l m" , "a" ]  
>>> mylab           #第0,1项以更换新值  
[ "l m" , 'a' , 'LaRC!' ]
```

- 也可以用多个取代一个

```
>>> mylab[ 1 : 2 ] = [ "student" , "in" ]  
>>> mylab  
[ "l m" , 'student' , 'in' , 'LaRC!' ]
```

List-变更序列内容(2/3)

- 调用成员函数
- **项目增加**
 - 跟串接很像, 差在期望的是单一物件, 不是序列

```
>>> mylab.append( 'nthu' )
```

```
>>> mylab
```

```
[ "I" m" , 'student' , 'in' , 'LaRC!' , 'nthu' ]
```

- 结果等于 `mylab = mylab + ['nthu']`
 - 也等于 `mylab[len(mylab):] = ['nthu']`
 - 不等于 `mylab[len(mylab)] = 'nthu'`

List-变更序列内容(3/3)

- **排序**

- 会产生新序列, 且一定存回原序列

```
>>> mylab.sort( )           #依照ASCII来排序
```

```
>>> mylab                    → [ "I" m" , 'LaRC!' , 'in' ,  
    'nthu' , 'student' ]
```

- **删去**

```
>>> del mylab[-1]           #删去一个项目
```

```
>>> mylab                    → [ "I" m" , 'LaRC!' , 'in' ,  
    'nthu' ]
```

```
>>> del mylab[:-1]          #删去整个片段
```

```
>>> mylab                    → [ 'nthu' ]    #等于mylab[:-1] =  
    [ ]
```


3.4 Dictionary

Dictionary-Introduction

- 任意对象皆可, 无前后次序的关系
- 长度可变, 异质性, 无线巢状延伸
- 可变更序列
 - 为独特的第三大类(对映), 使用专有的运算方式
 - 无法使用切片, 串接等利用偏移量的运算

Dictionary-常见的字典运算

- 基本运算

- D1 = { } #空字典
- D2 = { 'LaRC' :1, 'NTHU' :2 } #内含两个项目的字典
- D3 = { 'TW' :{ 'LaRC' :1, 'NTHU' :2}} #巢状字典
- D2['NTHU'], D3['TW']['LaRC'] #以key参考
- D2.has_key('NTHU') #成员关系测试
- D2.keys() #key列表
- D2.values() #列出字典的数值项目
- len(D2) #资料项数

- 变更字典内容

- D2[key] = new #增加/变更项目内容
- Del D2[key] #删除

Dictionary-基本运算

- 取出key的数值

```
>>> D2 = { 'LaRC' :1, 'NTHU' :2, 'TW' :3 }  
>>> D2[ 'NTHU' ]  
2
```

- 字典数据项的数量

```
>>> len(D2)  
3
```

- Key的关系测试

```
>>> D2.has_key( 'TW' )  
1
```

- 列出key (以序列的形式)

```
>>> D2.keys()  
[ 'LaRC' , 'NTHU' , 'TW' ]
```

Dictionary-变更字典内容

- 只要锁定key即可删改, 找不到的key就会新增值

- 改变资料项

```
>>> D2[ 'NTHU' ] = [ 'EE' , 'CS' ]
```

```
>>> D2
{ 'LaRC' :1, 'NTHU' :{ 'EE' , 'CS' }, 'TW' :3 }
```

- 删除数据项

```
>>> del D2[ 'TW' ]
```

```
>>> D2
{ 'LaRC' :1, 'NTHU' :{ 'EE' , 'CS' } }
```

- 新增资料项

```
>>> D2[8]= 'floor'      #key并不是都是字符串
```

```
>>> D2
{ 8: 'Floor' , 'LaRC' :1, 'NTHU' :{ 'EE' , 'CS' } }
```

Dictionary-一个小例子

```
>>> table = { 'Python' : 'Guido van Rossum' ,  
              'Perl' : 'Larry Wall' ,  
              'Tcl' : 'John Ousterhout' }
```

```
>>> language = 'Python'
```

```
>>> creator = table[language]
```

```
>>> creator  
'Guido van Rossum'
```

```
>>> for lang in table.keys():  
    print lang, '\t' , table[lang]
```

```
Perl      Larry Wall  
TclJohn Ousterhout  
Python    Guido van Rossum
```

3.5 Tuple

Tuple-Introduction

- 非常类似序列
- 差别---不可变更
 - 不能变更某个项目的内容
 - 无法自行增长项目or删减项目
 - 没有提供成员函数
- 不可变更可以提供某种程度上的保证
- 使用于function上面的传递

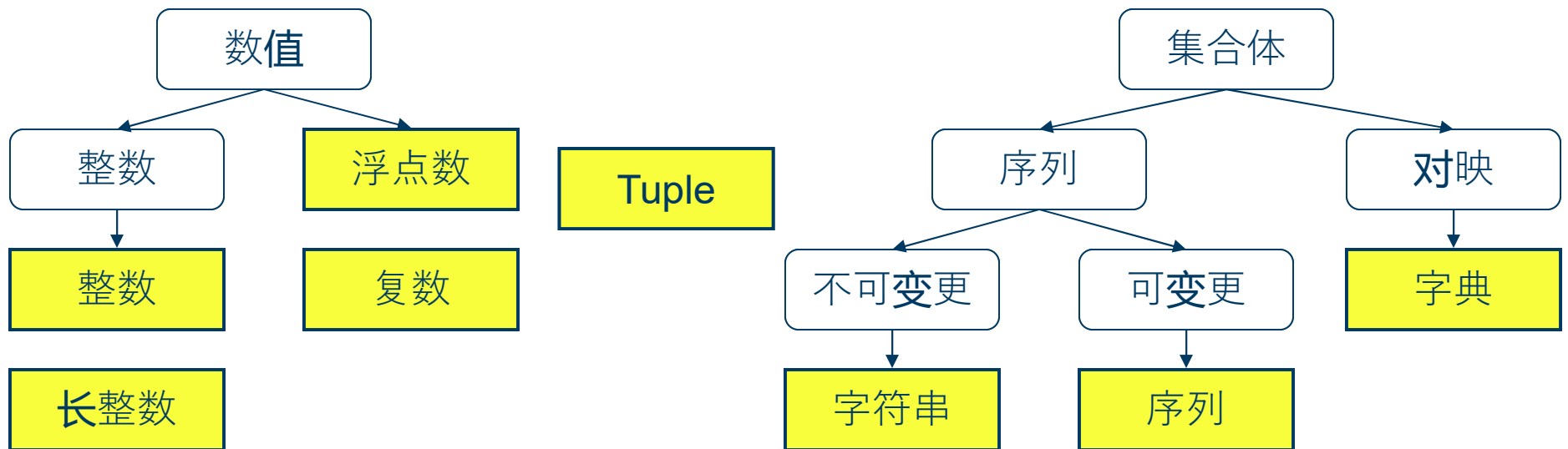
Tuple-基本tuple运算

- `()` #空tuple
- `T1 = (0,)` #1个项目的tuple(与表达式区别)
- `T2 = (0, 1, 2, 3)` #4个项目的tuple
- `T3 = 0, 1, 2, 3` #4个项目的tuple(与上式相同)
- `T4 = ('abc' , ('def' , 'ghi'))` #巢状tuples
- `T1[i], T3[i][j]` #索引值参考
- `T1[i:j], len(t1)` #切片运算, 长度
- `T1 + T2` #串接
- `T2 * 3` #重复串接
- `for x in T2` #循环反复
- `3 in T2` #成员关系

3.6 总结

对象的共通特性(1/3)

- 型态的分类



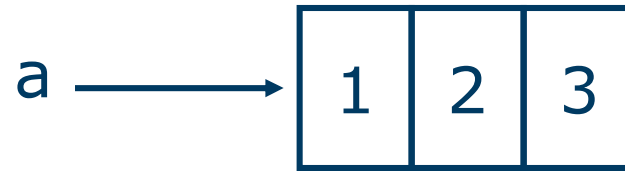
对象的共通特性(2/3)

- 参考地址共享
- Assignment manipulates references
 - * `x = y` **does not make a copy** of `y`
 - * `x = y` makes `x` **reference** the object `y` references
- 非常好用, 但是也要非常小心
- Example:

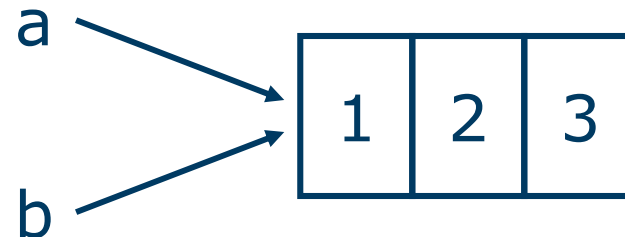
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

参考地址的例子1

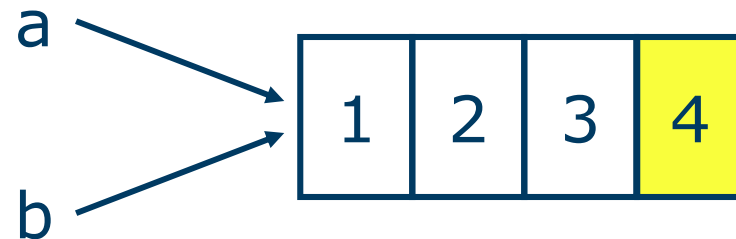
`a = [1, 2, 3]`



`b = a`

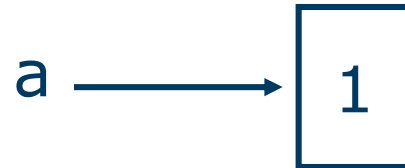


`a.append(4)`

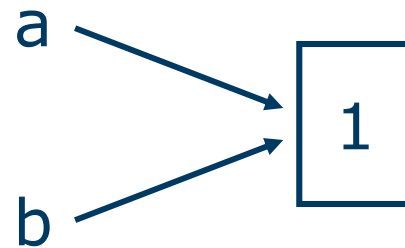


参考地址的例子2

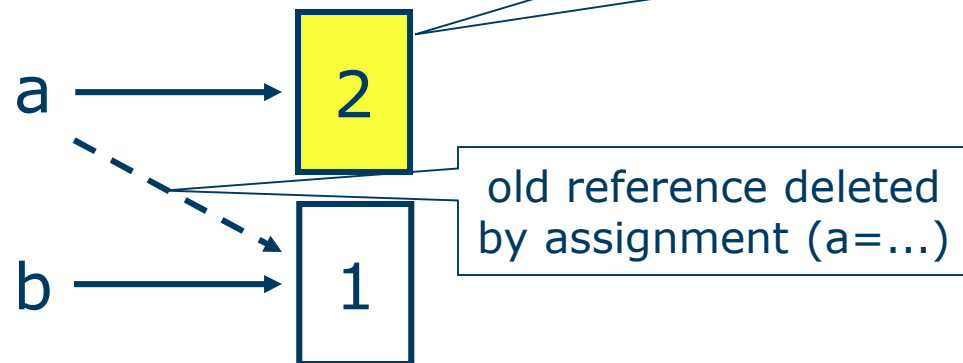
`a = 1`



`b = a`



`a = a+1`



对象的共通特性(3/3)

- ```
>>> L1 = [1, 2]
```
- ```
>>> L2 = [1, 2]
```
- ```
>>> L1 == L2, L1 is L2
```

  - (1, 0)
- 等同
  - == 运算符用来测试等同性
  - is 运算符用来测试对象的本体
- 真假值
  - 真: "xx" , 1
  - 假: " " , [ ], { }, 0.0, None #等于C语言的NULL
- 比较
  - 数值利用相对大小比较
  - 字符串根据辞理, 一个字符一个字符比对
  - 序列和Tuple由左到右来进行比较
  - 字典由排序过后来比较

## 恐怖的陷阱(1/2)

- 赋值语句(=)给的是地址

```
>>> L = [1, 2, 3]
```

```
>>> M = ['X' ,L, 'Y'] #M = ['X' , L[:], 'Y']
```

```
>>> M → ['X' , [1, 2, 3], 'Y']
```

```
>>> L[1] = 0
```

```
>>> M → ['X' , [1, 0, 3], 'Y']
```

- 循环数据结构

```
>>> L = [1]; L.append(L)
```

```
>>> L → [1, [...]]
```



## 恐怖的陷阱(2/2)

- 重复增加深度
  - 尽量避免, 或者用产生新序列的方式错开

```
>>> L = [4, 5, 6]
```

```
>>> X = L * 3
```

```
>>> Y = [L] * 3 # [L] + [L] + [L] = [L, L, L]
```

```
>>> X → [4, 5, 6, 4, 5, 6, 4, 5, 6]
```

```
>>> Y → [[4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

```
>>> L[1] = 0
```

```
>>> X → [4, 5, 6, 4, 5, 6, 4, 5, 6]
```

```
>>> Y → [[4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

# Summary

- 我们谈了
  - Python的简介
  - 基本对象型态
    - \* 数值, 字符串, 序列, 字典, tuple
- What' s in next week ?
  - 基本的控制结构
  - 函数

## 4. 控制语句

- 赋值运算
- 表达式
- Print
- If
- While
- For

# Python的程序架构

- Python语言的组成系统, 可以分解成模块(Module), 语句(Statement), 和物件(Object)
- 程序由模块文档组成
- 模组文档包含了语句
- 语句建构了对象并处理对象

# Python的基本语句

| 语句                                           | 角色   | 范例                                                       |
|----------------------------------------------|------|----------------------------------------------------------|
| 赋值                                           | 传地址  | <code>ee, larc = 'dept' , 'lab'</code>                   |
| 调用                                           | 执行函数 | <code>stdout.write( "larc is not a bird!!\n" )</code>    |
| <code>print</code>                           | 输出内容 | <code>print 'always Coca-Cola' , joke</code>             |
| <code>if/elif/else</code>                    | 选择   | <code>if "larc" in text: print text</code>               |
| <code>for/else</code>                        | 序列反复 | <code>for x in list: print x</code>                      |
| <code>while/else</code>                      | 循环   | <code>while 1: print 'hi'</code>                         |
| <code>pass</code>                            | 空语句  | <code>while 1: pass</code>                               |
| <code>break,</code><br><code>continue</code> | 循环跳跃 | <code>while 1:</code><br><code>if not line: break</code> |

# Python的进阶语句

| 语句                     | 角色     | 范例                                            |
|------------------------|--------|-----------------------------------------------|
| try/except<br>finally/ | 捕捉例外事件 | try: action()<br>except: print 'action error' |
| raise                  | 引致例外事件 | raise endSearch, location                     |
| import,<br>from        | 模块的存取  | import sys; from sys import<br>stdin          |
| def, return            | 函数的构体  | def f(a, b=1, *c): return a+b+c[0]            |
| class                  | 对象的建立  | class myclass: data = []                      |
| global                 | 名称空间   | def function(): global x, y;<br>x= 'new'      |
| del                    | 删除     | del data[k], del data[i:j], del obj.x         |

# 赋值运算(=)-Introduction

- 基本形式
  - 目标物件 = 赋值对象
- 赋值运算所赋值的是对象参考地址
  - Python的名称 = C的指标
  - 若要变成分开的两个物件则用+来解决
    - \* `num2 = num1 + 0`
    - \* `string2 = string1 + ''`
    - \* `list = list + []`
- 隐性赋值语句
  - 导入模块, 函数, 定义类别, for循环变量, 函数自变量
  - 只在程序运行时间才进行名称与对象参考地址链接

# 赋值运算(=)-型式

| 运算                              | 说明         |
|---------------------------------|------------|
| lab = 'LaRC'                    | 基本型        |
| dept, lab = 'EE' , 'LaRC'       | Tuple的赋值运算 |
| [dept, lab] = [ 'EE' , 'LaRC' ] | 序列的赋值运算    |
| dept = lab = 'mine'             | 多个目标物      |

- 序列和tuple的赋值运算
  - 都一样的啦!!完全没有差
  - 由左到右依序赋值
- 多个目标物的赋值运算
  - 会将右边的对象参考地址指给左边的每一个名称



# 赋值运算(=)-一些范例

- >>> X = 1
- >>> Y = 2
- >>> A, B = X, Y                      #tuple
- >>> A, B  
    – (1, 2)
- >>> [C, D] = [X, Y]                  #序列
- >>> C, D  
    – (1, 2)
- >>> **X, Y = Y, X**                      #tuple 交换内容
- >>> X, Y                              #同于 temp=X, X=Y,  
    Y=temp  
    – (2, 1)

# 赋值运算(=)-变量名称命名规则

- 语法
  - (底线or字母)+(任意数量的字母or数字or底线)
  - 可以: `_larc`, `larc`, `larc_1`
  - 不可以: `1_larc`, `larc$#!?`
- 大小写有差 `LaRC`≠`larc`
- 保留字的限制

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

# 表达式

- 通常表达式的结果并不会储存起来(除非赋值)
- 但是有两种情况会把表达式当语句用:
  - 调用函数和成员函数
    - \* `function(x, y)` #函数调用
    - \* `function.subfunc(x, y)` #成员函数调用
  - 在交互方式下印出数值
    - \* `x` #交互方式打印
    - \* `x < y and x != z` #复合表达式
    - \* `x < y < z` #范围测试
- 运算式不可嵌入赋值语句
  - 避免`=`和`==`的混淆

# Print

- Print将物件写到sys模块下的stdout()

```
>>> print 'Hello python'
Hello python
```

```
>>> import sys
```

```
>>> sys.stdout.write('Hello python\n')
Hello python
```

- 同时输出两个

```
>>> print "Hello" , "python"
Hello python
```

```
print 'Hello',
print 'python'
print 'YA! '
```

```
Hello python
YA!
```

# Print的相反

- `raw_input()`
  - 内部函数的一种 (不是语句欧~)
- ```
>>> raw_input( 'Who are you ?' )
```

Who are you ? 输入: I am king of the world !!
"I am king of the world !!"
- ```
>>> print "What is your lab ?" ;
```
- ```
>>> lab = raw_input( '-->' )
```

What is your lab ?
--> LaRC
- ```
>>> print lab
```

LaRC

# if-introduction

- 有if, elif, else三种
  - elif, else两者可有可无
- 书写格式
  - if <test1>: #if 测试
  - <statements1> #关联的区段
  - elif <test2>: #可有可无的elif
  - <statements2>
  - else: #可有可无的else
  - <statements3>

# if-一些范例

- if的出现

- ```
>>> if 1:  
...     print 'true'  
...  
- true
```

- else的出现

- ```
>>> if not 1:
... print 'true'
... else
... print 'false'
...
- false
```

- elif的出现

- ```
>>> x = 'Bill'  
...  
>>> if x == 'John':  
...     print 'Fine'  
... elif x == 'David':  
...     print 'OK'  
... else:  
...     print 'Run!'  
...  
- Run!
```

if-case?

- Python里面没有case
- 可以用if, elif, else取代

```
>>> choice = 'ham'
>>> if choice == 'spam':
...     print 1.25
... elif choice == 'bacon':
...     print 1.10
... elif choice == 'eggs':
...     print 0.99
... elif choice == 'ham':
...     print 1.99
... else:
...     print 'Bad choice'
...
- 1.99
```

- 可以用字典取代

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


语法规则(1/2)

- 程序逻辑区段以缩排来区别
 - 以 space 或 tab 进行缩排

#狂重要!!

- ```
>>> x = 1
```
- ```
>>> if x:
```
- ```
... y = 2
```
- ```
...     if y:
```
- ```
... print 'block2'
```
- ```
...     print 'block1'
```
- ```
... print 'block0'
```
- ```
...
```

 - block2
 - block1
 - block0

巢状的区段程序代码



语法规则(2/2)

- 可利用 \ 符号至下行接续程序内容。
 - 只要在括号内的皆可无限制换行

```
>>> print 123,\
```

```
    456
```

```
123 456
```

- 可利用 ; 在同一行内写下两个语句

```
>>> print 123; print 456
```

```
123
```

```
456
```

if-真值测试(1/2)

- 真: 不为0的数值or不为空的物件
- 假: 数字0, 空物件, None
- 真值测试有两种
 - 比较和等同运算:
 - * $>$, $==$, $<$, $>=$, $<=$
 - 布尔运算符:
 - * X and Y : 如果X, Y都为真, 表达式就为真 (等于 $\&\&$ in C)
 - * X or Y : X或Y其中一个为真, 表达式就为真 (等于 $||$ in C)
 - * not X : 如果X为假, 表达式就为真 (等于 $!$ in C)
- 比较和等同运算传回的是1 or 0
- $>>> 2<3, 3<2$ #传回1或0
 - (1, 0)

if-真值测试(2/2)

- 布尔运算符传回的是真or假的操作数物件(函数也可)
- 短线评断 (很重要)
 - 只要or左边为真, 就为真
 - 只要and左边为假, 就为假

- `>>> 2 or 3, 3 or 2` #如果为真, 就传回左边的操作数
 – `(2, 3)` #否则就传回右边的操作数
- `>>> [] or 3, [] or {}` #不论结果的真or假
 – `(3, {})`
- `>>> 2 and 3, 3 and 2` #如果为假, 就传回左边的操作数
 – `(3, 2)` #否则就传回右边的操作数
- `>>> [] and {}, 3 and []` #不论结果的真or假
 – `([], [])`

while回圈-Introduction

- 书写格式

- while <test>: #循环测试
- <statements1> #循环主体
- else: #可有可无的else
- <statements2> #如果没有用break跳离
- #就会执行此区段

```
count = 0
```

```
while count < 5:
```

```
    print count, " is less than 5"
```

```
    count = count + 1
```

```
else:
```

```
    print count, " is not less than 5"
```

while循环-一些范例

```
>>> x = 'LaRC'
```

```
>>> while x:
```

```
...     print x,
```

```
...     x = x[1:]
```

#抽掉x的第一个字符

```
...
```

```
LaRC, aRC, RC, C
```

```
>>> a = 0; b = 10
```

```
>>> while a < b:
```

#for循环的另一种写法

```
...     print a,
```

```
...     a = a + 1
```

```
...
```

```
0 1 2 3 4 5 6 7 8 9
```

while循环-break, continue, pass, else

- break语句
- continue语句
- pass语句
- 循环的else区段
 - 如果循环正常跳离(没有break搅局), 就会执行此区段
- 整体书写格式
 - while <test>:
 - <statements>
 - if <test> : break #跳离循环, 略过 else
 - if <test> : continue #回到循环的顶端
 - <statements> #continue判断式若成立, 就不执行
 - else:
 - <statements> #没用到break跳出, 就会执行

for循环-Introduction

- 书写格式

```
for <target> in <object>:    #赋值object给target
    <statements>
    if <test>: break          #跳离循环
    if <test>: continue       #回到循环顶端
    <statements>             # continue判断式成立, 就不执行
else:                         #如果没break, 就会执行
    <statements>
```

- 循环运算

- 停止条件: 所有序列内的对象都循序跑过一次
- object内的对象一定要是序列型态(string, list, tuple)

for循环-一些范例

- 累加器

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

  - 10

- 累乘器

- ```
>>> prod = 1
```
- ```
>>> for x in [1, 2, 3, 4]:
```
- ```
...     prod = prod * x
```
- ```
...
```
- ```
>>> prod
```

 - 24

- 也可使用字符串,tuple,分解tuple

- ```
S, T = "LaRC" , ("I" , "am")
```
- ```
for x in S: print x
```

 - L a R C
- ```
for x in T: print x
```

  - I am
- ```
T = [(1, 2), (3, 4)]
```
- ```
for (a, b) in T:
```
- ```
    print a, b
```

 - 1, 2
 - 3, 4

for循环-in也算一种for循环

```
>>> items = [ "aaa" , 111, (4, 5)]
>>> tests = [(4, 5), 3.14]
>>> for key in tests:
...     for item in items:
...         if item == key:
...             print key, "was found"
...             break
...     else:
...         print key, "not found!"
...
(4, 5) was found
3.14 not found!
```

for循环-range()

```
>>> range(5)
```

```
[0, 1, 2, 3, 4]
```

```
>>> range(2, 5)
```

```
[2, 3, 4]
```

```
>>> range(0, 10, 2)
```

```
[0, 2, 4, 6, 8]
```

```
>>> for x in range(4):
```

```
...     print x, "little,",
```

```
... else:
```

```
...     print "Indians!!"
```

```
1 little, 2 little, 3 little, Indians!!
```

恐怖的陷阱

- 冒号
 - 记得复合语句的首列结尾一定要加：
- 空白列
 - 文档中忽略
 - 对话模式中拿来当复合语句的结尾
 - * 千万别在...的提示符号下乱按Enter键
- 缩排一致
 - 千万别把space跟tab混着用
- 调用函数和导入模块
 - 调用函数后面一定要加小括号()
 - * `function()` `#Error: function`
 - 导入模块时不能在后面加扩展名
 - * `import math` `#Error: import math.py`

5.Function

函数的基本概念

- def用来建立函数并赋值函数名称
 - def后面的文字赋值给函数做为函数名称使用
- return会回传结果给原来的调用程序
- global用来声明模块层次的变量
 - 函数内部的名称属于该函数所有,别人看不到
 - 只有函数执行时才存在
 - 要给整个模块看就要加global
- 自变量是经由赋值运算而传递的
 - 传递的是对象地址, 而非变量本身名称
- 引数, 传回对象的型态和变量都不用声明
 - 同一个函数可以适用不同型态对象的操作

书写格式和定义调用

```
def <name> (arg1, arg2, ... argN):
```

```
    <statements>
```

```
    return <value>                #可有可无
```

- 如果不加return,函数会自动传回None物件

```
>>> def times(x, y):              #建立函数并做决定
```

```
...     return x * y              #回传x*y的值
```

```
...
```

```
>>> times(2, 4)                   #小括号里的是自变量
```

```
8
```

```
>>> times("No", 4)                #自变量没有型態之分
```

```
'NoNoNoNo'
```


函数的范围法则

- 每调用一次函数就产生一个新的区域名称空间
- 除非以global声明, 函数内的名称皆属于区域名称空间
- 其他的名称则为广域名称or内建名称

LGB法则

- 搜寻方式:
 - 先区域(Local), 再广域(Global), 最后内建(Built-in)
 - 只要发现符合名称, 就立刻停止搜寻
 - * 可以拥有相同名称, 区域的永远优先

内建 (Python)
预先定义的名称: open, len等等

广域 (模块)
在模块的最高层次赋值的名称
或以Global语句声明的名称

区域 (函数)
在函数语句底下赋值的名称

LGB法则-范例

```
>>> X = 99          #X广域范围内赋值
>>> def func(Y):     #func()广域范围内赋值
...     #区域范围    #Y, Z于区域范围内赋值
...     Z = X + Y     #X无赋值, 所以是广域X的内容
...     return Z
...
...
func(1)
100
```

- 广域名称: X, func
- 区域名称: Y, Z

global

- 唯一类似声明语法的语句
- 意指位于模块文档的最高层次
- 函数内若想要成为广域名称, 必须以global声明
- 函数内是可以直接使用别处的广域名称无须声明的

- ```
>>> y, z = 1, 2
```

- ```
>>> def all_global():
```

- ```
... global x
```

- ```
...     x = y + z
```

- ```
...
```

- 除非必要, 不要使用!!

- ```
>>> x
```

- Traceback (...):

- line 1, in ?

- NameError: name 'x'

- is not defined

- ```
>>> all_global()
```

- ```
>>> x
```

- 3

传递自变量(函数后面的值)

- 自变量的传递其实是把对象赋值给区域名称
- 引数没有别名的关系, 不会害外面的变量受影响
 - 例外: 若为可变更对象(list, dictionary)就有影响

```
>>> def change(x, y):  
...     x = 2                #只改变局部变量  
...     y[0] = 'LaRC'        #改变了共享对象的内容  
...  
>>> X = 1  
>>> L = [1, 2]  
>>> change(X, L)             #传递不可变更和可变更对象  
>>> X, L  
(1, [ 'LaRC' , 2])
```

return

- 也是可以用tuple方式回传的喔~~
 - 所以可以回传不只一个对象

```
>>> def multiple(x, y)
```

```
...     x = 2
```

#只改变区域名称

```
...     y = [3, 4]
```

```
...     return x, y
```

#传回新值(tuple)

```
...
```

```
>>> X = 1
```

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

```
>>> X, L = multiple(X, L)
```

#结果回传给自己

```
>>> X, L  
(2, [3, 4])
```

自变量比对模式

- 正常定义: `def function(name1in, ...)`
- 刚刚都是直接由左至右的比对, 调用时使用
- 关键词比对
 - 由自变量名称来比对, 调用时使用
 - `function(name1in = name1out, ...)`
- 默认值比对
 - 若未被赋值到, 则其数值为默认值, 定义时使用
 - `def function(name1in = default1, ...)`

自变量比对模式-范例(1/2)

```
>>> def func(larc, ee, nthu=0, tw=0): #前面两个一定要
...     print (larc, ee, nthu, tw)
>>> func(1, 2, 3, 4)
1, 2, 3, 4
>>> func(1, 2)
1, 2, 0, 0
>>> func(1, tw=1, ee=0)
1, 0, 0, 1
>>> func(larc=1, ee=0)
1, 0, 0, 0
>>> func(nthu=1, ee=2, larc=3)
3, 2, 1, 0
```


自变量比对模式-范例(2/2)

- 交集

- ```
>>> s1, s2, s3 = "larc", "Larc", "LaRC"
```

- ```
>>> def intersect(*args):
```
- ```
... res = []
```
- ```
...     for x in args[0]:
```
- ```
... for other in args[1:]:
```
- ```
...             if x not in other: break
```
- ```
... else:
```
- ```
...             res.append(x)
```
- ```
... return res
```
- ```
...
```
- ```
>>> intersect(s1, s2)
```

  - ```
– ['a', 'r', 'c']
```
- ```
>>> intersect(s1, s2, s3)
```

  - ```
– ['a']
```

- 联集

- ```
>>> def union(*args):
```
- ```
...     res = [ ]
```
- ```
... for seq in args:
```
- ```
...         for x in seq:
```
- ```
... if not x in res:
```
- ```
...                 res.append(x)
```
- ```
... return res
```
- ```
...
```
- ```
>>> union(s1, s2)
```

  - ```
– ['l', 'a', 'r', 'c', 'L']
```
- ```
>>> union(s1, s2, s3)
```

  - ```
– ['l', 'a', 'r', 'c', 'L', 'R', 'C']
```

lambda

- 书写格式
 - name = **lambda** arg1, ... argN : Expr. using args
- 与def的比较
- lambda是表达式, 不是语句
 - lambda可以内嵌在主体程序内
 - 回传一个函数给赋值的函数名称
- lambda是一行的表达式, 不是区段的语句
 - 类似return的结构
 - 远不如def复杂
- 小函数的必备良药

lambda-一些范例

- def 与 lamda的比较

```
>>> def func(x, y, z): return x+y+z
```

```
>>> func(2, 3, 4)    → 9
```

```
>>> func = lamda(x, y, z): x+y+z
```

```
>>> func(2, 3, 4)    → 9
```

- 妙用：短小精悍的程序

```
>>> L = [lambda x: x**2, lambda y: y**3, lambda z: z**4]
```

```
>>> for f in L:
```

```
...     print f(2),
```

```
    4  8 16
```

```
>>> print L[0](3)    → 9
```

map()

- 算是for循环的函数版

- ```
>>> counts = [1, 2, 3, 4]
```

- for循环写法

- ```
>>> new = []
```
- ```
>>> for x in counts:
```
- ```
...     new.append(x+10)
```
- ```
...
```
- ```
>>> new
```

 - ```
[11, 12, 13, 14]
```

- map写法

- ```
>>> def inc(x):
```
- ```
... return x + 10
```
- ```
...
```
- ```
>>> map(inc, counts)
```

  - ```
[11, 12, 13, 14]
```

- 不甘寂寞的lambda应用

- ```
>>> map((lambda x:
```

  - ```
x+10), counts)
```
 - ```
[11, 12, 13, 14]
```

# 恐怖的陷阱-名称空间的侦测(1/3)

- 区域名称空间的侦测属静态侦测

```
>>> X = 99
```

```
>>> def selector():
```

```
... print X # 其实还不存在
```

```
... X = 88 # X被分类为区域名称
```

```
... # 其他的赋值语句也会(def, lambda..)
```

```
>>> selector()
```

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

```
File "<input>", line 1, in ?
```

```
File "<input>", line 2, in selector
```

```
UnboundLocalError: local variable 'X' referenced before
assignment
```

## 恐怖的陷阱-名称空间的侦测(2/3)

- 方法一：使用global强迫X变成广域变数
- >>> X = 99
- >>> def selector():
- ...      global X                      #强迫X成为广域变量
- ...      print X
- ...      X = 88                      #会改到广域的X
- ...
- >>> selector()
- 99
- >>> X
- 88
- 缺点：接下来的运算也都会影响广域变量 X

## 恐怖的陷阱-名称空间的侦测(3/3)

- 方法二：使用加载\_\_main\_\_模块

```
>>> def selector():
... import __main__ #导入模块
... print __main__.X #以评定方法取得
... X = 88 #没有评定, 视为局部变量
... print X #打印局部变量
...
```

- >>> selector():

99

88

- 优点：两者皆可以随意运用, 但是!! 绝对不是一种好的写法
- 方法三：直接使用不同名称的变量 = = .....

# 恐怖的陷阱-默认值和可变更对象

- ```
>>> def saver(x = []):
```
- ```
... x.append(1)
```
- ```
...     print x
```
- ```
...
```
- ```
>>> saver([2]) #未用到默认值
```

 - ```
[2, 1]
```
- ```
>>> saver() #用到默认值
```

 - ```
[1]
```
- ```
>>> saver() #每次调用同一个
```

 - ```
[1, 1]
```
- ```
>>> saver()
```

 - ```
[1, 1, 1]
```

- 解决办法：每次都用新序列

- ```
>>> def saver(x = None):
```
- ```
... if x is None:
```
- ```
...         x = [] #新序列
```
- ```
... x.append(1) #改变新序列
```
- ```
...     print x
```
- ```
...
```
- ```
>>> saver([2])
```

 - ```
[2, 1]
```
- ```
>>> saver()
```

 - ```
[1]
```
- ```
>>> saver() #没有成长
```

 - ```
[1]
```



## 6. 模块与包

# Why Modules?

- 程序代码的再使用
  - 对话模式如果结束Python, 程序代码也会不见
  - 模组可以让我们把程序代码储存在文档里
  - 还可以reload
- 系统空间名称分割
  - 模组是Python里最高层次的单元
  - 是组织系统组件的工具
- 实作共享服务或共享数据
  - 若有许多函数会使用相同的数据结构, 可以将此数据结构写在module里面, 再由各个函数自行导入即可

# 基本概念

- 建立模块：
  - 只要将程序代码写到文档里, 存成.py即可
  - 也可以是C延伸文档
- 使用模块：
  - Import: 得到一整个完整的模块文档
  - From: 可以从某个模块文档里取出几个特定的名称
  - Reload: 可重载模块程序代码而毋须跳离解译程序
  - 另外也可以直接于系统下执行
- 模组搜寻路径：
  - PYTHONPATH环境参数中的目录路径
  - `sys.path`
  - `sys.path.append`

# 基本范例

- 模块可以使用任何一种文书编辑软件来编辑
- 请一定要以.py作为结尾
- 文档的名称一样受到命名规则的规范
  - 无法导入if.py的模块
- Ex. 有一个名为‘larc.py’的模块文档
  - `def printer(x):`            `#模块属性`
  - `print x`
- `%python`                    `#进入python`

# 基本范例

- `>>> import larc` #得到模块
- `>>> larc.printer('Hello python!')` #以评定用法取出名称  
– Hello python!
- `>>> from larc import printer` #得到一次释出
- `>>> printer('Hello python!')` #不需要评定取出名称  
– Hello python!
- `>>> from larc import *` #得到全部的释出
- `>>> printer('Hello python!')`  
– Hello python!

# 模块文档==名称空间

- 模块的语句在第一次导入时执行
  - python会建立一个空的模块对象
  - 从头到尾执行一遍
- 最高层次的指定运算建立模块属性
  - 由原先模块里最高层次的=, def来建立
  - 由=建立成员变量, def建立成员函数
  - 指定的名称会存在模块的名称空间里
- 模组名称空间: `__dict__`
  - 模块导入时建立的名称空间是字典
- 模组是独立的范围
  - 使用必需使用评定方式
  - 载入之后一直可以使用
  - 与函数只有在执行时存在名称空间不一样

## 基本范例二

- Ex. 有一个名为'larc.py'的模块文档

```
– import sys
– name = 42
– def func(): pass
– print 'done loading.'
```

- >>> import larc.py  
– done loading.

- >>> larc.sys  
– <module 'sys'>

- >>> larc.name  
– 42

- >>> larc.func  
– <function func at 765f20>

- >>> larc.\_\_dict\_\_.keys()  
– ['\_\_file\_\_', 'name', '\_\_name\_\_', #\_\_file\_\_:模块的文件名  
– 'sys', '\_\_doc\_\_', '\_\_builtins\_\_', 'func'] #\_\_name\_\_:模块名称

# 名称评定用法

- 简单变量
  - “X”意指在当前的范围里搜寻名称X(LGB法则)
- 评定用法
  - “X.Y”意指在X对象中搜寻属性Y(not范围)
- 评定用法路径
  - “X.Y.Z”意指在X对象中搜寻Y, 然后在对象X.Y中搜寻Z
- 跟之前的范围法则毫无关系
- LGB法则只对无评定用法的名称生效而已
- 可以将它想象成一个连结到另一个名称空间的指标
- 评定用法适用于所有有属性的对象
  - 模组, 类别皆适用
  - 也算是类别里继承(inheritance)的实作



# 导入模式

- **导入只发生一次**
  - 模組在第一次使用import或from时会载入
  - 执行模块的程序代码, 会使模块建立其名称空间
  - 之后任何的import和from只会从已加载的模块存取东西
- `## simple.py`
- `spam = 1`
- `## python`
- `>>> import simple`
- `>>> simple.spam`
  - 1
- `>>> simple.spam = 2`
- `>>> import simple`
- `>>> simple.spam`
  - 2

# import和from

- 和def一样, import和from都是可执行语句, 所以都可以写在if语句内, or 函数内部
- import和from也都是隐性的指定运算
  - import会把整个模块指定给一个名称
  - from会把名称指定给另一个模块中同样名称的对象
- ~~from module import \*~~等同于下列程序
- ```
>>> Import module
```
- ```
>>> name1 = module.name1
```
- ```
>>> name2 = module.name2
```
- ```
>>> ...
```
- ```
>>> del module
```

导入包

- Graphics/
 __init__.py
 Primitive/
 __init__.py
 lines.py
 fill.py
 text.py
 ...
 Graph2d/
 __init__.py
 plot2d.py
 ...
 Graph3d/
 __init__.py
 plot3d.py
 ...
 Formats/
 __init__.py
 gif.py
 png.py
 tiff.py
 jpeg.py

- import Graphics.Primitive.fill
 - Graphics.Primitive.fill.floodfill(img,x,y,color)
- from Graphics.Primitive import fill
 - fill.floodfill(img,x,y,color)
- from Graphics.Primitive.fill import floodfill
 - floodfill(img,x,y,color)
- 下边这个语句具有歧义:
from Graphics.Primitive import *
- `__all__ = ["lines","text","fill",...]`

- ******下面这个语句只会执行Graphics目录下的__init__.py文件，而不会导入任何模块：**

```
import Graphics
```

```
Graphics.Primitive.fill.floodfill(img,x,y,color) # 失败!
```

- **不过既然 import Graphics 语句会运行 Graphics 目录下的 __init__.py文件,我们就可以采取下面的手段来解决这个问题：**

```
# Graphics/__init__.py
```

```
import Primitive, Graph2d, Graph3d
```

- **# Graphics/Primitive/__init__.py**
import lines, fill, text, ...

指定会碰到的问题

- **## small.py**
- `x = 1`
- `y = [1,2]`
- **## python**
- `>>> from small import x, y` #复制两个名称出来
- `>>> x = 42` #只改变区域的x
- `>>> y[0] = 42` #改变共享的可变更对象
- `>>> import small` #得到模块名称
- `>>> small.x` #small的x不是区域的x
 - 1
- `>>> small.y` #与区域共享一个可变更对象
 - [42,2]

reload()-introduction

- import只会在第一次导入模块时执行模块的程序代码
 - 稍后的import只会使用已加载的模块对象
 - reload函数会强迫已经加载的模块程序代码重新执行
- reload是函数不是语句
- 可以容许程序不中断
 - 并变更部分的程序代码
- 书写格式
 - import module #导入模块
 - ... #在此改变模块的程序代码
 - reload(module) #得到更新后的结果
 - ...

reload()-详细说明

- reload会在模块的名称空间里重新执行模块的程序代码
 - 会覆写模块已存在的名称空间
 - 不是先删除旧的名称空间, 再载入一次
- 最高层次的指定运算会替换对映名称的内容
- 会影响所有以import来取出模块名称内容的程序
 - reload会替代所有之前位于模块名称空间里的数值
- reload只会影响重载之后的from语句
 - 若使用from来取出模块的属性, 则reload并不会影响原先持有的数值内容
 - 但是reload之后还有from语句发生, 则程序得到的是新的内容

reload()-范例(1/2)

- # changer.py
- message = "1st version"
- def printer():
- print message
- 先调用一次
- >>> import changer
- >>> changer.printer()
 - 1st version
- 新开一个窗口编辑changer.py
- %cat changer.py
- message = "2nd version"

- 回到Python解释程序
- >>> import changer
- >>> changer.printer()
 - 1st version #没有影响
- 重载模块
- >>> reload(changer)
- <module 'changer'>
- >>> changer.printer()
 - 2nd version #新的结果

模块编译程序

- python系统称为解译程序
 - 其实较类似java,介于编译程序与解译程序之间
 - 拥有中介形式：位码(bytecode)
 - 拥有一个Visual Machine来执行bytecode
- bytecode程序储存为.pyc扩展名
 - 在import过后就会产生
- 对一个模块M来说
 - 如果M.py在M.pyc储存之后没有任何的变更, 则python将改载入M.pyc而不是M.py
 - 如果M.py有更动的话, 则会载入M.py并产生新的M.pyc覆盖之前的版本
- 也可以将.pyc视为python程序的封装版本
 - 可拥有资料的隐密性

`__name__`和`__main__`

- 特殊情况：
 - 名称开头有底线字符就不会被导入
 - 减少名称空间多余的导入
- 每一个模块都有一个内建的属性：`__name__`
 - 如果文档当作程序来执行, `__name__`会设定成“`__main__`”字符串
 - 如果文档当作模块来导入, `__name__`会设定成模块的名称
- `__main__`在自身测试上最常出现

```
● %cat tester.py
● if __name__=='__main__':
●     print "This is a program"
● else:
●     print "This is a module"
```

```
● %python
● >>> import tester
●     - This is a module
● %python test.py
● This is a program
```

变更模块搜寻路径

- 模块搜寻路径存在PYTHONPATH的目录里
- 使用sys.path来改变搜寻路径
- ```
>>> import string
```
- ```
>>> sys.path
```

 - `['.', 'c:\\python\\lib', 'c:\\python\\lib\\tkinter']`
- ```
>>> sys.path = ['.', 'c:\\book\\examples'] #改变搜寻路径
```
- ```
>>> sys.path
```

 - `['.', 'c:\\book\\examples']`
- ```
>>> import string
```

  - `Traceback (innermost last): #因为将python\\lib的路径删了`
  - `File "<stdin>", line 1 , in ?`
  - `ImportError: No module named string`
- 不要乱删除python原始的目录路径

# 模块设计的概念

- 到处都是模块
  - 只要有程序代码, 就有模块
  - 事实上, 交互方式也是建构在模块上, 档名为 `__main__`
- 减少模块之间的对偶关系(coupling)
  - 如同函数, 模块写的越独立, 运作的越好
  - 若模块在另一个模块里, 要做到不受该模块的广域变量影响比较好
- 模组应该尽量不要更改别的模块的变量内容
  - 也就是少发生goto的情形

# 恐怖的陷阱-from只复制名称

- `%cat module1.py`
- `X = 99`
- `def printer(): print X`
- `%python`
- `>>> from module1 import X, printer`
- `>>> X= 88`
- `>>> printer()`  
– 99

- 解决方法:
- 使用import方法, 不要用from

- `>>> import module1`
- `>>> module1.X = 88`
- `>>> module1.printer()`  
– 88

# 恐怖的陷阱-程序代码的先后次序

- 由于模块第一次导入时, 就会从头到尾执行一次, 所以会有几个问题:
  - 位于模块最高层次的程序代码(即不在函数or巢状语句里的部份)在导入时就会立刻被执行, 因此这些程序代码绝对不能参考到文档后面才指定的名称
  - 位于函数主体的程序代码只有当函数被调用时才会执行, 所以在调用前其名称空间并不存在, 通常可以参考文档里的任何名称

- ```
>>> func1()          #错误: “func1”尚未指定
```
- ```
>>> def func1():
```
- ```
...     print func2():  #没问题: “func2”稍后才会搜寻
```
- ```
>>> func1() #错误: “func2”尚未指定
```
- ```
>>> def func2():
```
- ```
... return “Hello”
```
- ```
>>> func1()          #没问题: “func1”和“func1”都指定了
```

- 解决方法:
 - 请把def语句放在文档最前头, 把最高层次的程序代码放在文档的尾部

恐怖的陷阱-from的递归导入

- 导入必须从头到尾执行一次
 - 模组之间彼此互相导入, 会有问题
 - 若模块在导入另一个模块时, 模块还没有执行完全, 使得有一些名称没有指定到
- %cat recur1.py
- X = 1 #正确:有执行
- import recur2 #错误:recur2不存在
- Y = 2 #未执行
- %cat recur2.py
- from recur1 import X #正确:“X”已经指定了
- from recur1 import Y #错误:“Y”尚未指定
- 解决方法:
 - 歹路不可行阿 ~ ~ ~
 - 尽量减少模块循环的次数, 降低模块依存的关系
 - 如果一定要, 请推迟模块名称的存取, 或是摆到函数里执行

恐怖的陷阱-reload会败给from

- 万恶的渊薮：from
 - 由于from只会将名称复制过去, 等于此名称和原来的模块失去了联系
 - reload的重载对from完全无影响力
- 算是module状态的goto
 - 如果你的module常常变动需要reload, 请不要用from
- from使用的时机:
 - 只使用一个大模块里的小函数(节省内存)
 - 小module, 且不常变动(方便不需评定)

恐怖的陷阱-reload只有一次

- 当reload时候, python只会重载那个指定的模块文档, 如果内部还有导入模块的语句, python会当作没看到
- %cat A.py
- import B
- import C
- %python
- >>> ...
- >>> reload A
- 只会reload A, B和C不会重载一次
- 解决方法:
 - ㄟ.....最好不要这样搞...
 - 目前学的撇步里面没这招!!(还是有解的拉)

三、模块

- 模块实际上是将一组函数放在一起共享公共的主题
- 将这些函数存储于一个.py文件中;
- 使用import命令导入。

1、模块的创建及导入

- **创建模块，即创建一个.py文件，在其中包含用于完成任务的变量、类和函数，不包括main函数。**
- **模块使用之前要导入该模块，导入方法之前已做过介绍。**
- **例5-11：创建模块，用于在屏幕上打印各种形状。**

定义的模块shapes及使用模块的源程序：

```
#shapes.py
"""A collection of fuctions
   for printing basic shapes.
"""
CHAR='*'
def rectangle(height,width):
    """Prints a rectangle."""
    for row in range(height):
        for col in range(width):
            print(CHAR,end='')
        print()
def square(side):
    """Prints a square."""
    rectangle(side,side)
def triangle(height):
    """Prints aright triangle."""
    for row in range(height):
        for col in range(1,row+2):
            print(CHAR,end='')
        print()
```

#例5-11：在屏幕打印各种形状

```
import shapes

print(dir(shapes))

print(shapes.__doc__)
print()

print(shapes.CHAR)
print()

shapes.square(5)
print()

shapes.triangle(3)
print()

shapes.rectangle(3,8)
```

执行结果:

```
['CHAR', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'rectangle', 'square', 'triangle']
```

```
A collection of fuctions  
    for printing basic shapes.
```

```
*
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*
```

```
**
```

```
***
```

```
*****
```

```
*****
```

```
*****
```

例：创建一个求圆面积、圆周长、圆表面积和圆体积的模块

```
#circle.py
```

```
pi = 3.14159
```

```
def area(radius):  
    return pi*(radius**2)
```

```
def circumference(radius):  
    return 2*pi*radius
```

```
def sphereSurface(radius):  
    return 4.0*area(radius)
```

```
def sphereVolume(radius):  
    return (4.0/3.0)*pi*(radius**3)
```



确认将此代码保存为名称为circle.py的文件

调用方式一：

```
>>> import circle
>>> pi = 3.0
>>> print (pi)
3.0
>>> print (circle.pi)
3.14159
>>> print (circle.area(3))
28.27431
>>> print (circle.circumference(3))
18.849539999999998
```

调用方式二:

```
>>> from circle import *  
>>> pi = 0.0  
>>> print (pi)  
0.0  
>>> print (area(3))  
28.27431  
>>> print (circumference(3))  
18.849539999999998
```


2、模块的属性

- 模块有一些内置属性，用于完成特定的任务。
- 例5-11中的`dir(shapes)`，就输出了模块`shapes`的属性。

```
['CHAR', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'rectangle', 'square', 'triangle']
```

- 如：
 - `__doc__`：模块中用于描述的文档字符串
 - `__name__`：模块名
 - `__file__`：模块保存的路径

3、模块的内置函数

- Python提供了一个内联模块**buildin**。该模块定义了一些常用函数，利用这些函数可以实现数据类型的转换、数据的计算、序列的处理等功能。

(1) filter()

- **声明:**

Class filter(object)

filter(function or None, iterable)-->filter object

- **功能: filter()可以对某个序列做过滤处理, 根据自定义函数返回的结果是否为真来过滤, 并一次性返回处理结果。返回结果是filter对象。**

例5-13: filter()函数应用

#例5-13: filter()函数应用

```
def func(x):
```

```
    if x>0:
```

```
        return x
```

```
print(filter(func, range(-9,10)))
```

#调用filter函数返回的是filter对象

```
print(list(filter(func, range(-9,10))))
```

#将filter对象转换为列表

执行结果：

```
<filter object at 0x0229A350>
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

(2) reduce()

- `reduce(func, sequence[,initail]) -> value`
- 功能：对序列中的元素进行连续操作。例如：可对某个序列中的元素进行累加、累乘和阶乘等操作。

```
def sum(x,y):  
    return x+y  
from functools import reduce #引入reduce  
print(reduce(sum,range(0,10))) #对0-9累加  
print(reduce(sum,range(0,10),10)) #对0-9累加，再加10  
print(reduce(sum,range(0,0),10)) #结果为10
```

执行结果：

```
45  
55  
10
```

(3) map()

- 声明:
- Class map(object)
map(func, *iterables)-->map object
- 功能: 对多个序列的每个元素都执行相同的操作, 并返回一个map对象。

例：map()函数应用。求列表中数字的幂运算

```
def power(x):  
    return x**x  
print(map(power, range(1, 5)))    #打印map对象  
print(list(map(power, range(1, 5)))) #转换为列表输出  
def power2(x, y):  
    return x**y  
print(map(power2, range(1, 5), range(5, 1, -1)))    #打印map对象  
print(list(map(power2, range(1, 5), range(5, 1, -1)))) #转换为列表输出。map函数提供了两个参数，  
                                                         #结果为1^5, 2^3, 3^2, 4^1
```

执行结果：

```
<map object at 0x0229A350>  
[1, 4, 27, 256]  
<map object at 0x02C24C50>  
[1, 16, 27, 16]
```

常用内置模块函数（一）：

函 数	描 述
<code>abs(x)</code>	返回x的绝对值
<code>bool([x])</code>	将一个值或表达式转换为bool类型。如果表达式x为真返回True，否则返回False
<code>delattr(obj.name)</code>	等价于 <code>del obj.name</code>
<code>eval(s[,globals,locals])</code>	计算表达式的值
<code>float(x)</code>	将数字或字符串转换为float类型
<code>hash(object)</code>	返回一个对象的hash值
<code>help([object])</code>	返回内置函数的帮助说明
<code>id(x)</code>	返回一个对象的标识
<code>input([prompt])</code>	接收控制台的输入，并将输入的值转换为数字
<code>int(x)</code>	将数字或字符串转换为整型

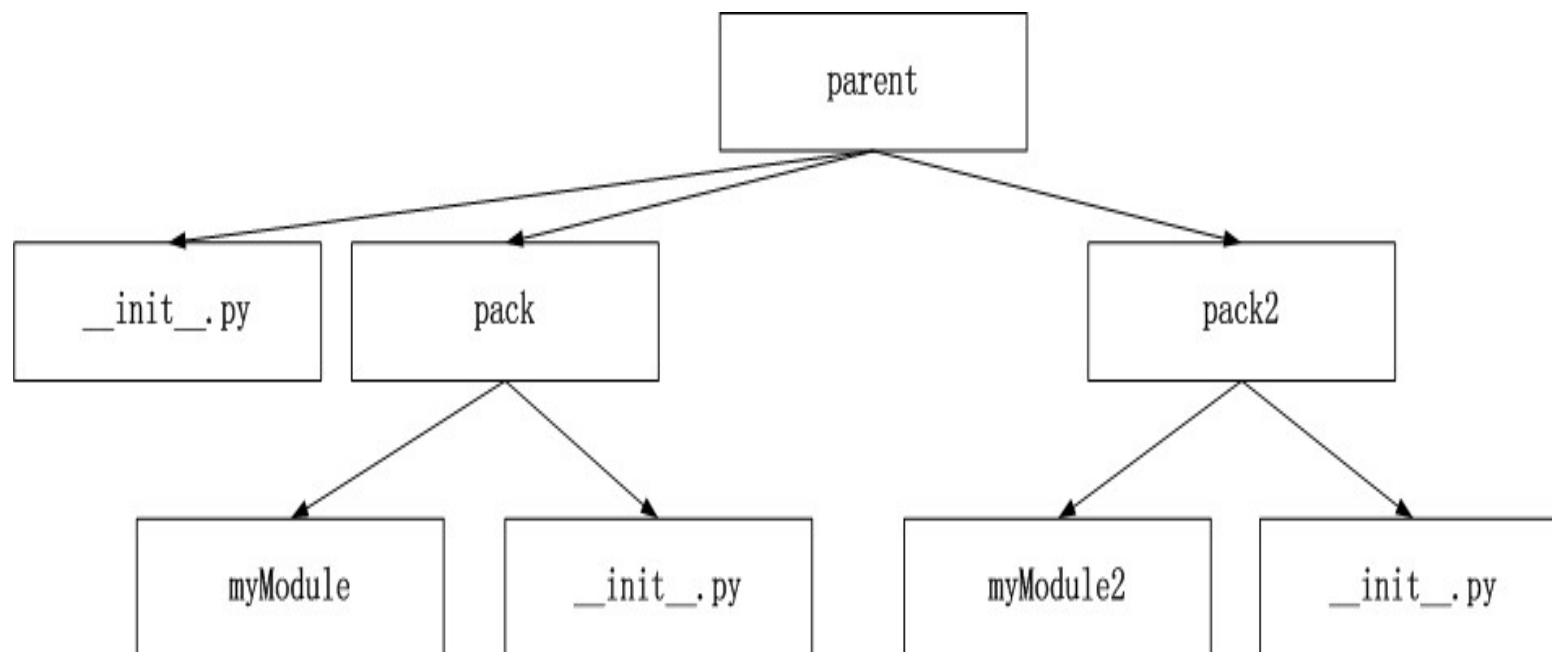
常用内置模块函数（二）：

函 数	描 述
<code>len(obj)</code>	对象包含的元素个数
<code>range([start,]end[,step])</code>	生成一个列表并返回
<code>reduce(func,sequence[,initial])</code>	对序列的值进行累计计算
<code>round(x,n=0)</code>	四舍五入函数
<code>set([iterable])</code>	返回一个set集合
<code>sorted(iterable[,cmp[,key[,reverse]]])</code>	返回一个排序后的列表
<code>sum(iterable[,start=0])</code>	返回一个序列的和
<code>type(obj)</code>	返回一个对象的类型
<code>zip(iter1[,iter2[...]])</code>	将n个序列作为列表的元素返回

4、自定义包

- 除了系统自带的包之外，还可以自定义包。
- 之前已经介绍过，包至少含有一个__init__.py文件。__init__.py文件的内容可以为空，它用于标识当前文件夹是一个包。
- 包的作用是为了程序的重用，把实现一些特定功能的代码组合到一个包中，调用包提供的功能从而实现重用。

例：一个包与模块的树形关系



- 定义一个包**parent**。在**parent**包中创建两个子包**pack**和**pack2**。
- **Pack**包中定义了一个模块**myModule**，**pack2**包中定义了一个模块**myModule2**。
- 最后在包**parent**中定义一个模块**main**，调用包**pack**和**pack2**。

包pack的初始化程序及myModule模块：

```
#包pack的__init__.py程序
if __name__ == '__main__':
    print('作为主程序运行')
else:
    print('pack初始化')    #当包pack被其他模块调用时，将输出"pack初始化"

#包pack的myModule模块
def func():
    print("pack.myModule.func()")

if __name__ == '__main__':
    print('myModule作为主程序运行')
else:
    print('myModule被另一个模块调用')
```

包pack2的初始化程序及myModule2模块：

```
#包pack2的__init__.py程序
if __name__ == '__main__':
    print('作为主程序运行')
else:
    print('pack2初始化')    #当包pack2被其他模块调用时，将输出"pack2初始化"
```

```
#包pack2的myModule2模块
def func2():
    print("pack2.myModule2.func2()")

if __name__ == '__main__':
    print('myModule2作为主程序运行')
else:
    print('myModule2被另一个模块调用')
```

包parent中的main模块及执行结果：

```
#包parent中的main模块，调用了pack、pack2中的函数
from pack import myModule
from pack2 import myModule2

myModule.func()
myModule2.func2()
```

main模块执行结果：

```
pack初始化
myModule被另一个模块调用
pack2初始化
myModule2被另一个模块调用
pack.myModule.func()
pack2.myModule2.func2()
```

4、第三方模块的导入

(1) 单文件模块

- 直接把文件拷贝到 `$python_dir/Lib`



(2) 多文件模块, 带setup.py

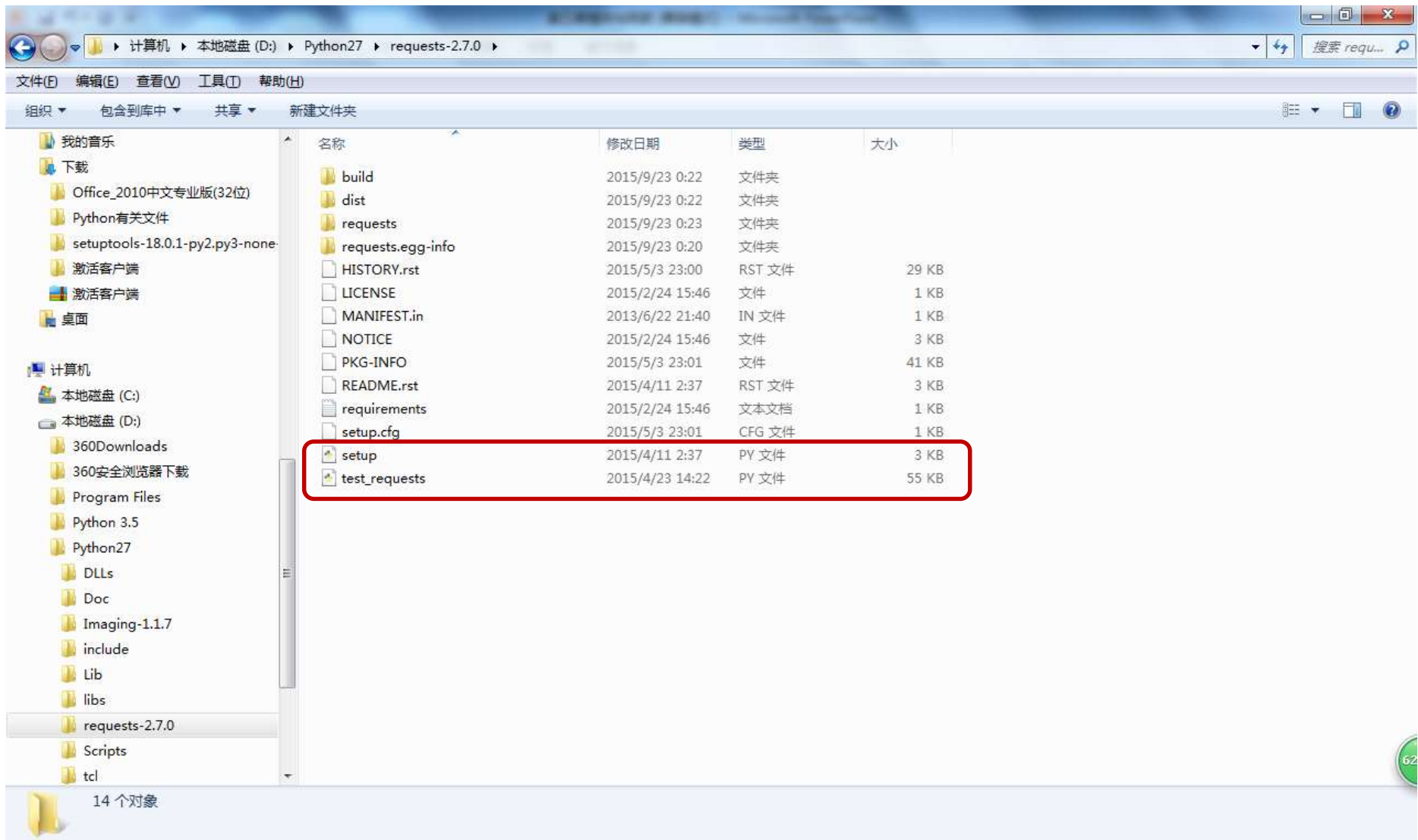
- `python setup.py install`

例5-16：导入第三方模块requests

- **requests简介：**requests是python的一个HTTP客户端库。支持 HTTP 连接保持和连接池，支持使用 cookie 保持会话，支持文件上传，支持自动确定响应内容的编码，支持国际化的 URL 和 POST 数据自动编码。

步骤1:

- 去第三方库的网站（ <https://pypi.python.org> ）
下载安装包，解压在python的安装目录。
- 注意第三方库的文件夹的位置以及setup.py的位置。
- 本例在Python 2.7下安装的。



步骤2:

- 运行cmd，进入命令行。利用cd命令进入第三方库文件夹的位置。



步骤3:



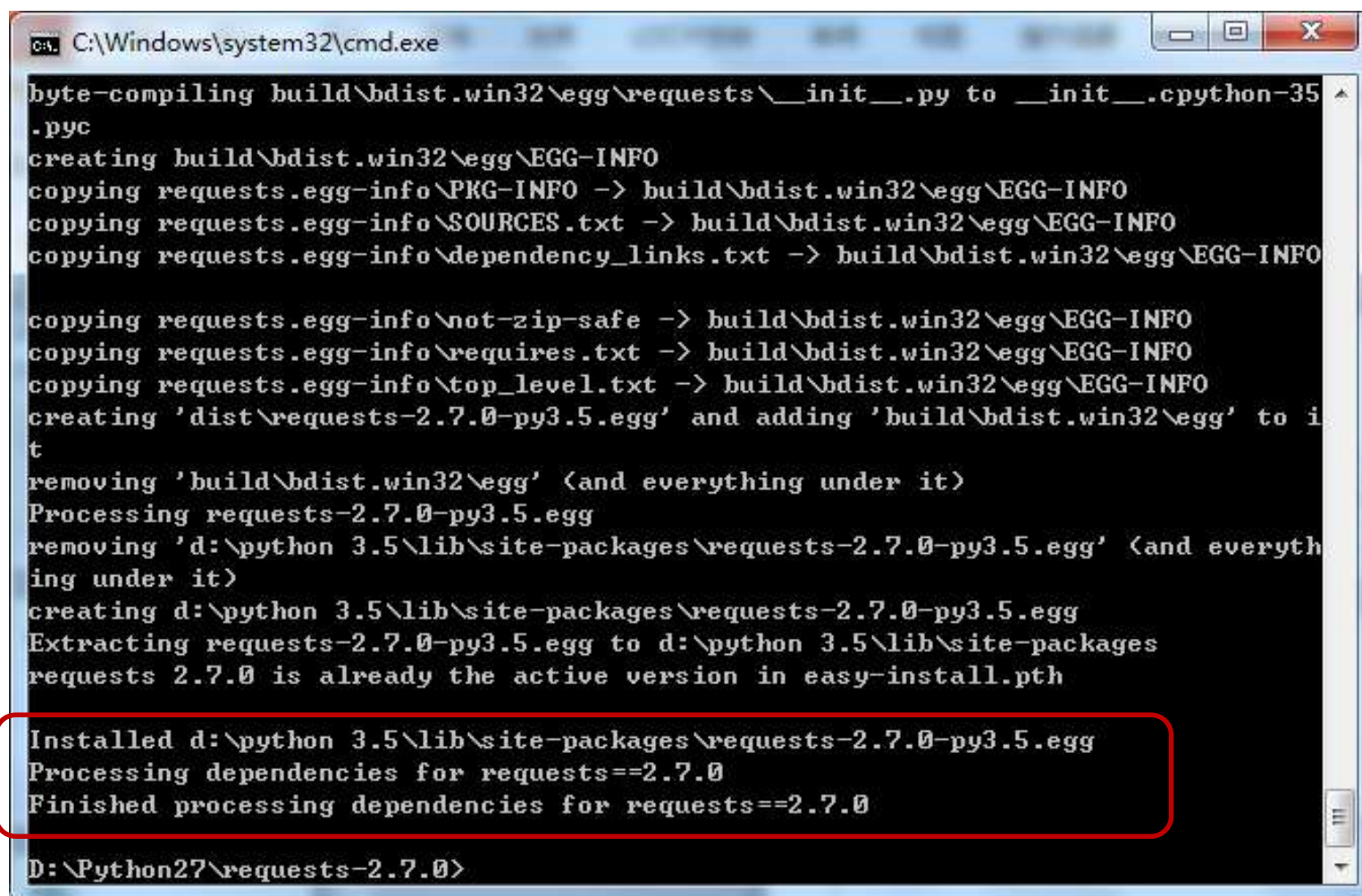
```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\zhangjl>d:

D:\>cd python27\requests-2.7.0

D:\Python27\requests-2.7.0>python setup.py install
```

安装完成



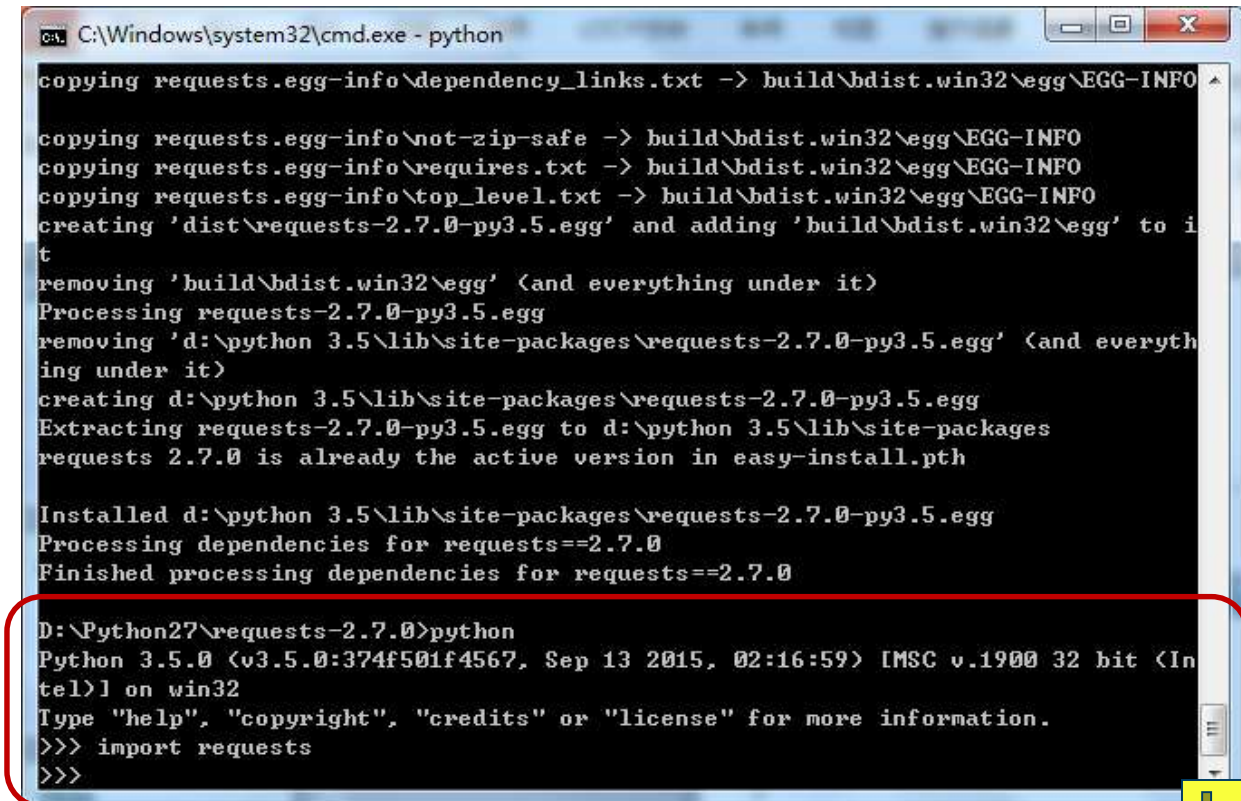
```
C:\Windows\system32\cmd.exe
byte-compiling build\bdist.win32\egg\requests\__init__.py to __init__.cpython-35
.pyc
creating build\bdist.win32\egg\EGG-INFO
copying requests.egg-info\PKG-INFO -> build\bdist.win32\egg\EGG-INFO
copying requests.egg-info\SOURCES.txt -> build\bdist.win32\egg\EGG-INFO
copying requests.egg-info\dependency_links.txt -> build\bdist.win32\egg\EGG-INFO
copying requests.egg-info\not-zip-safe -> build\bdist.win32\egg\EGG-INFO
copying requests.egg-info\requires.txt -> build\bdist.win32\egg\EGG-INFO
copying requests.egg-info\top_level.txt -> build\bdist.win32\egg\EGG-INFO
creating 'dist\requests-2.7.0-py3.5.egg' and adding 'build\bdist.win32\egg' to it
removing 'build\bdist.win32\egg' (and everything under it)
Processing requests-2.7.0-py3.5.egg
removing 'd:\python 3.5\lib\site-packages\requests-2.7.0-py3.5.egg' (and everyth
ing under it)
creating d:\python 3.5\lib\site-packages\requests-2.7.0-py3.5.egg
Extracting requests-2.7.0-py3.5.egg to d:\python 3.5\lib\site-packages
requests 2.7.0 is already the active version in easy-install.pth

Installed d:\python 3.5\lib\site-packages\requests-2.7.0-py3.5.egg
Processing dependencies for requests==2.7.0
Finished processing dependencies for requests==2.7.0

D:\Python27\requests-2.7.0>
```

步骤4:

- 最后进入命令行，import库名称，观察第三方库是否安装成功。



```
C:\Windows\system32\cmd.exe - python
copying requests.egg-info\dependency_links.txt -> build\bdist.win32\egg\EGG-INFO
copying requests.egg-info\not-zip-safe -> build\bdist.win32\egg\EGG-INFO
copying requests.egg-info\requires.txt -> build\bdist.win32\egg\EGG-INFO
copying requests.egg-info\top_level.txt -> build\bdist.win32\egg\EGG-INFO
creating 'dist\requests-2.7.0-py3.5.egg' and adding 'build\bdist.win32\egg' to it
removing 'build\bdist.win32\egg' (and everything under it)
Processing requests-2.7.0-py3.5.egg
removing 'd:\python 3.5\lib\site-packages\requests-2.7.0-py3.5.egg' (and everything under it)
creating d:\python 3.5\lib\site-packages\requests-2.7.0-py3.5.egg
Extracting requests-2.7.0-py3.5.egg to d:\python 3.5\lib\site-packages
requests 2.7.0 is already the active version in easy-install.pth

Installed d:\python 3.5\lib\site-packages\requests-2.7.0-py3.5.egg
Processing dependencies for requests==2.7.0
Finished processing dependencies for requests==2.7.0

D:\Python27\requests-2.7.0>python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 13 2015, 02:16:59) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>>
```



8. Classes

Why Classes?

- 类别：定义新的对象
 - 内建物件：数值, 字符串, 序列, 字典, tuple
 - 继承(inheritance)
 - 组合(composition)
- 与模块的比较
 - 多个实体对象
 - * 每个新的对象都拥有独立的名称空间
 - * 从同一个类别出来的对象都可以存取该类别的属性
 - 个别化(customization)
 - * 类别支持继承, 又可以覆盖
 - * 名称空间的阶层架构

基本概念-多个实体对象(1/3)

- 物件有两种：
 - 类别物件：提供对象默认的行为举止, 实体对象的来源(list)
 - 实体对象：实际处理的对象($x = []$)
 - * 拥有独立的名称空间
 - * 保留了和建造它的类别之间的存取关系
- 类别物件提供预设行为
 - 类别叙述建立类别对象, 并指定给类别对象一个名称
 - * 如同def一样, class也是可执行的叙述
 - * 执行时会建立一个新的类别对象, 并在首列指定名称
 - 位在类别叙述中的自变量将成为类别的属性
 - * 如同module一样, 所有叙述里的自变量都成为类别的属性
 - * 也是透过名称评定的方法来存取(object.name)
 - 类别属性是出类别的状态与行为
 - * 属性会分享给所有由此类别对象而生的实体对象
 - * 如果有def叙述, 则会形成该物件的成员函数

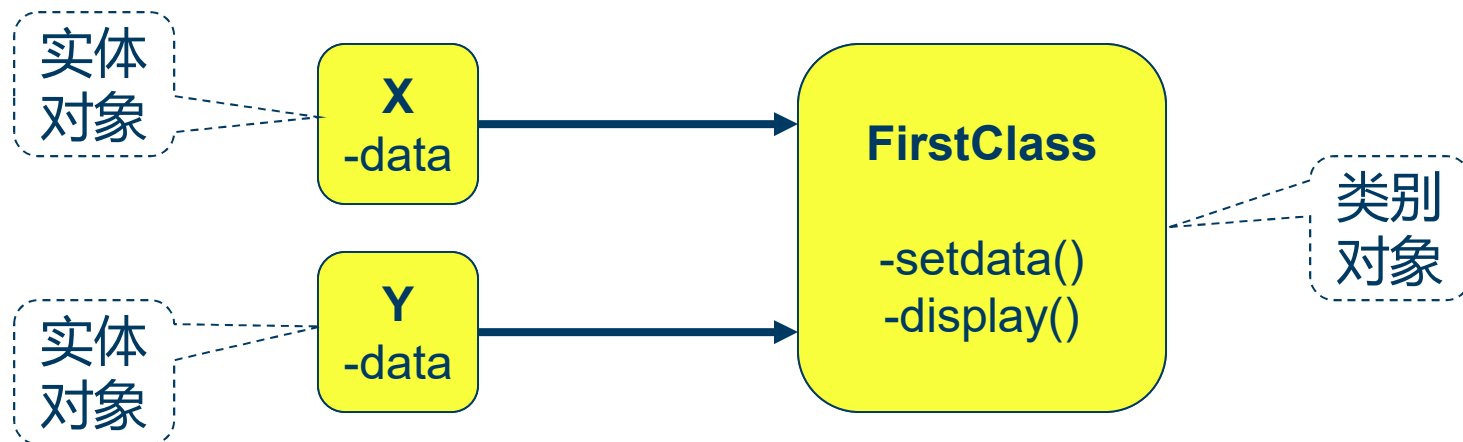
基本概念-多个实体对象(2/3)

- 实体对象从类别而来
 - 调用类别对象会生成新的实体对象
 - * 只要调用, 就会产生新的实体对象
 - 每个实体对象都会继承类别的属性并得到独立的名称空间
 - * 拥有自己新的名称空间; 一开始是空的
 - * 会继承类别对象的属性
 - 传递给self的自变量会改变实体对象的属性
 - * 类别的成员函数第一个自变量会指向正在处理的实体对象
 - * 传递self的自变量会建立或改变实体对象的数据
 - * 但是绝不会改变类别里的属性

```
>>> class FirstClass:           #定义类别对象
...     def setdata(self, value): #定义类别成员函数
...         self.data = value     #self是实体
...     def display(self):
...         print self.data
```

基本概念-多个实体对象(3/3)

```
>>> x = FirstClass()           #两个实体对象
>>> y = FirstClass()           #每一个都是新的名称空间
>>> x.setdata("King Arthur")   #调用成员函数, self = x
>>> y.setdata(3.14159)         #执行FirstClass.setdata(y, 3.14159)
>>> x.display(), y, display()  #self.data两个都不同
    ('King Arthur', 3.14159)
>>> x.data = "New value"       #也可以使用名称评定的方式设定
>>> x.display()
    New value
```



基本概念-个别化与继承(1/2)

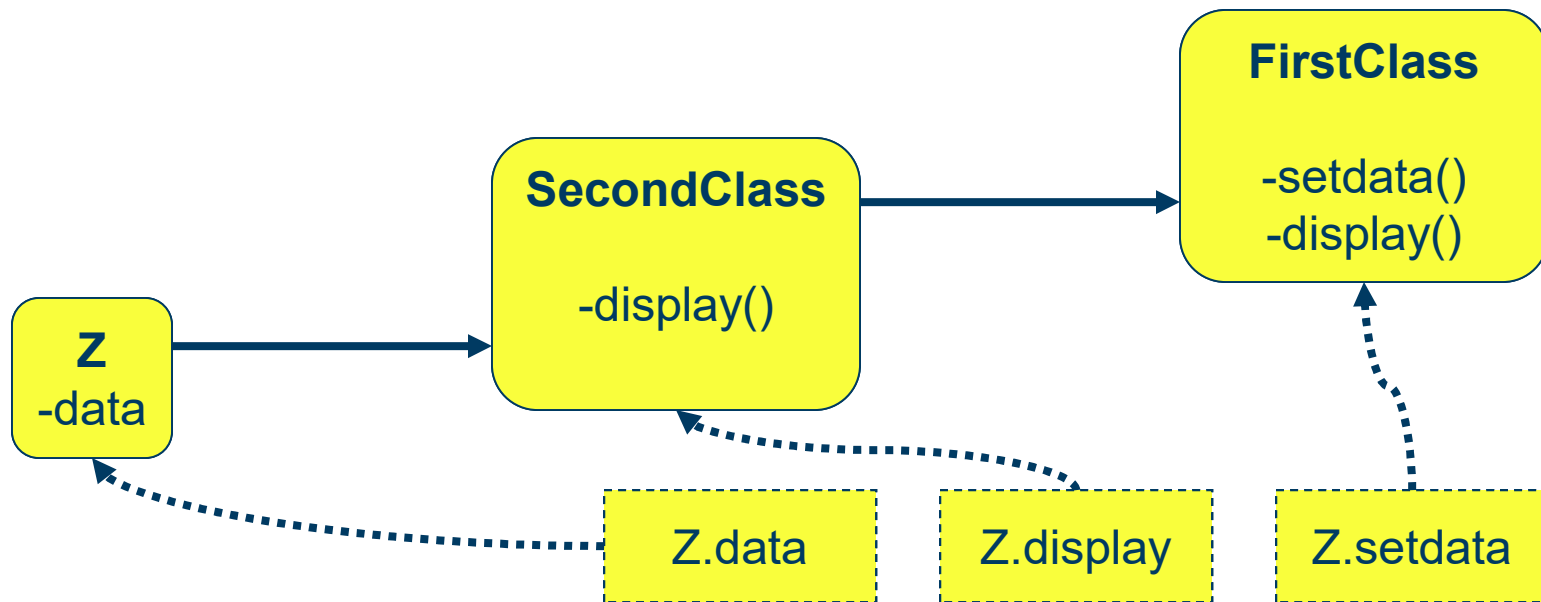
- 母类别(superclass)要列在类别首列的小括号中
 - 继承属性的类别称为子类别(subclass)
- 类别会从母类别继承属性
 - 如同实体对象, 将得到母类别名称空间内的所有属性
 - 评定名称时, 如果无法在子类别找到, 会自动往母类别上找
- 实体物件继承的属性不限于产出它的类别
 - 延伸出实体对象时, 会将所有包括母类别的属性全部继承下来, 评定名称时, 先检视实体对象, 然后类别, 再来所有的母类别
- 逻辑内容的变更始于子类别
 - 如果子类别里有重新定义母类别的名称, 子类别会覆盖掉原先继承下来的属性

- ```
>>> class SecondClass(FirstClass): #继承setdata
```
- ```
...     def display(self):           #变更display
```
- ```
... print 'Current value = "%s"' % self.data
```

[范例复习](#)

## 基本概念-个别化与继承(2/2)

- >>> z = SecondClass()
- >>> z.setdata(42) #setdata在第一类找到
- >>> z.display() #display在SecondClass被替代掉
  - Current value = “42”
- >>> x.display()
  - New value #FirstClass里的display并没有被影响



# 基本概念-运算符重载(1/2)

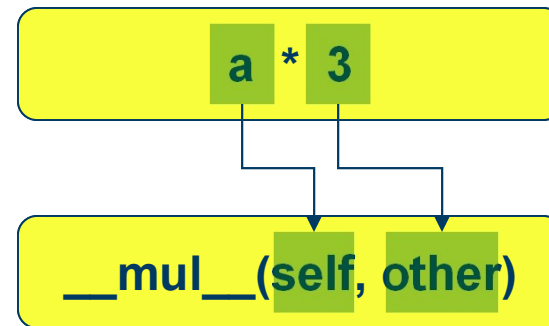
- 写成\_\_X\_\_形式的成员函数, 即可拦截
  - Python提供特别命名的函数进行拦截运算
- Python评断运算符时, 就会自动调用
  - 当物件出现某个+运算, \_\_add\_\_就会被调用
- 类别可以覆盖内建型态的运算
  - 所有内建型态的运算皆有对应的函数可以覆盖
- 运算符允许类别与Python的对象模式整合
  - 藉由过载型态运算, 可以自行定义出内部对象

```
>>> class ThirdClass(SecondClass): # 继承 SecondClass
... def __init__(self, value): # ThirdClass(Value)
... self.data = value
... def __add__(self, value): # self + other
... return ThirdClass(self.data + other)
... def __mul__(self, value): # self * other
... self.data = self.data * other
```

[范例复习](#)

## 基本概念-运算符重载(2/2)

- `>>> a = ThirdClass("abc")`
- `>>> a.display()`
  - Current value = "abc"
- `>>> b = a + 'xyz'`
- `>>> b.display()`
  - Current value = "abcxyz"
  - #制造出新的对象
- `>>> a * 3`
- `>>> a.display()`
  - Current value = "abcabcab"
  - #直接修改对象内容



- ThirdClass是一个SecondClass, 所以display有继承过来
- 调用时, 多加了自变量'abc', 会传递给建构子\_\_init\_\_, 并指定给data
- 使用 + 和 \* 时, python会将运算符的左边传递给self, 右边传递给other
- 这些重载也可以由子类别or实体对象继承

# 基本概念-范例复习

- >>> class FirstClass: #定义类别对象
- ... def setdata(self, value): #定义类别成员函数
- ... self.data = value #self是实体
- ... def display(self):
- ... print self.data

[返回](#)

- >>> class SecondClass(FirstClass): #继承setdata
- ... def display(self): #变更display
- ... print 'Current value = "%s"' % self.data

[返回](#)



# 类别的叙述

- 书写格式
  - `class <name>(superclass, ...):` #指定给name
  - `data = value` #共享类别数据
  - `def method(self, ...):` #成员函数
  - `self.member = value` #实体对象数据
- 和模块一样：
  - Python会先将所有的主体叙述, 从头到尾执行一遍
  - 指定的名称会变成类别对象的属性

|                                                   |                                             |
|---------------------------------------------------|---------------------------------------------|
| ● <code>&gt;&gt;&gt; class Subclass:</code>       | ● <code>&gt;&gt;&gt; x = Subclass(1)</code> |
| ● <code>... data = 'little'</code>                | ● <code>&gt;&gt;&gt; y = Subclass(2)</code> |
| ● <code>... def __init__(self, value):</code>     | ● <code>&gt;&gt;&gt; x.display()</code>     |
| ● <code>... self.data = value</code>              | – 1 little                                  |
| ● <code>... def display(self):</code>             | ● <code>&gt;&gt;&gt; y.display()</code>     |
| ● <code>... print self.data, Subclass.data</code> | – 2 little                                  |

self.data不同, 但是  
Subclass.data相同

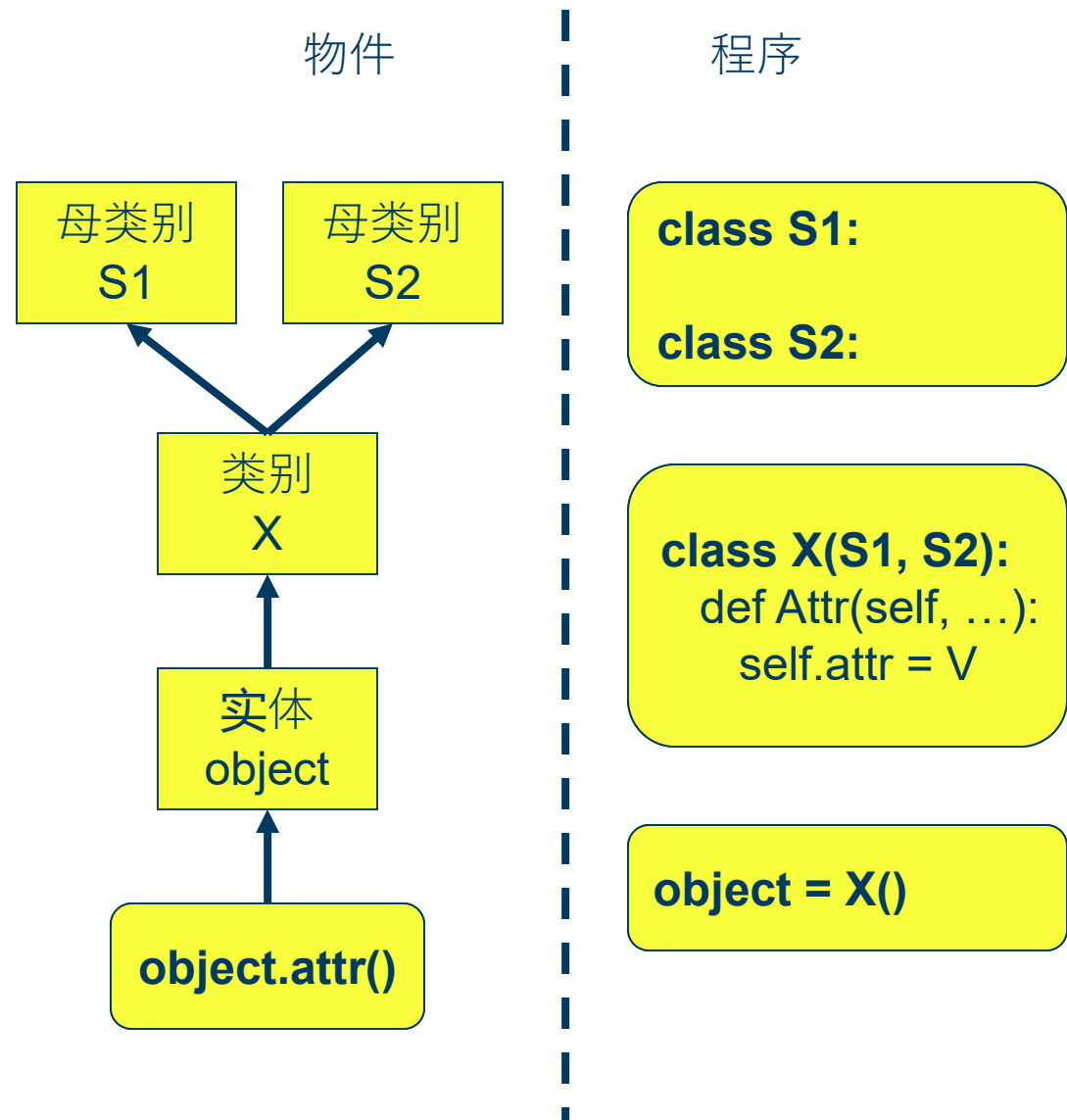
# 类别成员函数

- 与函数其实很像
  - 差别在于成员函数的第一个自变量
    - \* 接收的永远是实体对象
  - 也就是python会自行把实体对象的成员函数调用转换成类别对象的函数调用
  - `instance.method(args...) → class.method(instance, args...)`
  - `self`类似C++里面的this指标

- `>>> class NextClass:` #定义类别
- `... def printer(self, text):` #定义成员函数
- `... print text`
- `>>> x = NextClass()` #做出实体
- `>>> x.printer('Hello python!')` #调用成员函数
  - Hello python!
- `>>> NextClass.printer(x, 'Hello python!')` #类别成员函数
  - Hello python!

# 继承-搜寻名称空间树

- 实体对象的属性是经由指定成员函数中的 **self** 属性而产生的
- 类别的属性是经由 **class** 叙述中的指定叙述而产生的
- 把母类别置放在 **class** 叙述首列的小括号里, 就可以达到母类别与子类别的**联系**
- 结果将构成名称空间树, 每次评定时, 就会由**实体对象**的位置开始向上搜寻属性, 直到最高点的母类别为止



# 继承-成员函数的演化(1/2)

- 继承的树搜寻模式变成演化系统的最佳方式
- 由于会先从子类别的名称开始继承, 因此子类别可以替代掉预设的行为, 只要重新定义母类别的属性即可
- 过载继承名称的观念导致演化的发展
  - 提供一个和母类别相同的名称即可替代继承过来的名称
  - 也可以延伸母类别的功能

- ```
>>> class Super:
```
- ```
... def method(self):
```
- ```
...         print 'in Super.method'
```
- ```
>>> class Sub(Super):
```
- ```
...     def method(self):
```

 #覆盖成员函数
- ```
... print 'starting Sub.method'
```

 #多加的动作
- ```
...         Super.method(self):
```

 #执行预设的动作
- ```
... print 'ending Sub.method'
```

 #多加的动作

## 继承-成员函数的演化(2/2)

```
>>> x = Super() #做一个Super的实体
>>> x.method() #执行Super.method
 in Super.method
>>> x = Sub() #做一个Sub的实体
>>> x.method() #执行Sub.method
 starting Sub.method #会在调用Super.method
 in Super.method
 ending Sub.method
```

- Sub延伸了method的功能
  - 利用call back母类别所释出的版本
- 这种延伸很常见于\_\_init\_\_里面
  - 可以继承所有母类别建构时所得到的属性
  - Super.\_\_init\_\_(self, ...)

# 运算符重载

- 让类别中途拦截正常的Python运算
- 类别可以重载所有Python表达式的运算符
- 类别也可以重载对象的运算：
  - 列印, 调用, 评定用法等等
- 重载使得类别实体更能像内建型态般运作
- 重载的实作方法
  - 透过提供特殊成员函数名称而达成
  - 皆以\_\_X\_\_的方式存在

# 运算符重载-常见的成员函数

| 成员函数                      | 过载      | 用法                                          |
|---------------------------|---------|---------------------------------------------|
| <code>__init__</code>     | 建构子     | 建立对象 <code>class()</code>                   |
| <code>__del__</code>      | 解构子     | 释放对象                                        |
| <code>__add__</code>      | 运算符‘+’  | <code>X + Y</code>                          |
| <code>__or__</code>       | 运算符‘ ’  | <code>X   Y</code>                          |
| <code>__repr__</code>     | 打印,转换型态 | <code>Print X, `X`</code>                   |
| <code>__getattr__</code>  | 名称评定用法  | <code>X.未定义</code>                          |
| <code>__getitem__</code>  | 索引值参考   | <code>X[key]</code> , for loops, in tests   |
| <code>__setitem__</code>  | 索引值指定运算 | <code>X[key] = value</code>                 |
| <code>__getslice__</code> | 切片运算    | <code>X[low:high]</code>                    |
| <code>__len__</code>      | 长度      | <code>len(X)</code> , truth tests           |
| <code>__cmp__</code>      | 比较      | <code>X == Y</code> , <code>X &lt; Y</code> |

# 运算符重载-\_\_getitem\_\_

- `__getitem__` 会拦截实体对象索引值参考的运算
  - 会将物件当作第一个自变量
  - 括号内的索引值当成第二个自变量
- 范例：传回索引值的平方值

```
>>> class indexer:
... def __getitem__(self, index):
... return index ** 2
...
>>> X = indexer()
>>> for index in range(5):
... print X[index], #调用__getitem__(X, index)
...
0 1 4 9 16
```



# 运算符重载-\_\_getitem\_\_

- 由于for循环的运作方式(使用索引值0~N)使得\_\_getitem\_\_会被调用
- \_\_getitem\_\_成为过载迭代和成员关系测试的方法

```
>>> class stepper:
... def __getitem__(self, i):
... return self.data[i]
...
>>> X = stepper() # X是stepper物件
>>> X.data = 'larc'
>>> for item in X: # for循环调用__getitem__
... print item, # for的索引值是 0...3
...
- l a r c
>>> 'a' in X # in运算符也会调用__getitem__
- 1
```

# 运算符过载-\_\_getattr\_\_

- `__getattr__` 会拦截未定义的属性
  - 若以评定用法调用一个不存在的属性名称, 就会被调用
  - 且该属性名称将会以字符串形式传递给`__getattr__`
  - 如果在继承树的搜寻程序内找到的属性名称, 就不会被调用
- ```
>>> class empty:
```
- ```
... def __getattr__(self, attr):
```
- ```
...         if attr == 'age':
```
- ```
... return 36
```
- ```
...         else:
```
- ```
... return "undefined value"
```
- ```
...
```
- ```
>>> X = empty()
```
- ```
>>> X.age
```

 - 36
- ```
>>> X.name
```

  - 'undefined value'

# 运算符重载-\_\_repr\_\_

- \_\_repr\_\_是用来传回字符串的
  - 当print或是倒引号``被用到时, 将会自动被调用
- 范例：顺便复习一下\_\_init\_\_, \_\_add\_\_

```
>>> class adder:
... def __init__(self, value=0): #预设value值
... self.data = value #建构设定
... def __add__(self, other):
... self.data = self.data + other #加上other
... def __repr__(self):
... return `self.data` #转换成字符串
...
>>> X = adder(1) #__init__
>>> X + 2; X + 2 #__add__
>>> X #__repr__
— 5
```

# 多重继承(1/2)

- 搜寻的程序是
  - 先从深度发展
  - 再由左至右横跨多重继承

```
• >>> class Lister:
• def __repr__(self):
• return (“<Instance of %s, address %s:>\n%s” %
• (self.__class__.__name__, id(self), self.attrnames()))
• def attrnames(self):
• result = ‘ ’
• for attr in self.__dict__.keys(): #扫描实体名称空间字典
• if attr[:2] == ‘__’:
• result = result + “\tname %s=<built-in>\n” % attr
• else:
• result = result + “\tname %s=%s\n” % (attr, self.__dict__[attr])
• return result
```

## 多重继承(2/2)

```
class Super:
... def __init__(self): #无__repr__
... self.data1 = 'larc'
class Sub(Super, Lister): #混入__repr__
... def __init__(self): #Lester可以存取self
... Super.__init__(self)
... self.data2 = 123 #实体属性

●>>> Y = Super()
●>>> print Y
 - <Super instance at 87f1b0> #预设格式: 类别, 地址

●>>> X = Sub()
●>>> print X
 - <Instance of Sub, address 97833392>:
 - name data2 = 123
 - name data1 = larc
```

# 成员函数的边界

- 无界类别成员函数：无self
  - 必须提供一个实体对象作为函数的第一个自变量
- 有界实体成员函数：self+函数
  - Python会自动把实体对象包装起来

```
>>> class larc:
```

```
... def ok(self, message):
```

```
... print message
```

```
>>> object1 = larc()
```

```
>>> x = object1.ok
```

#有界成员函数物件

```
>>> x('hello')
```

#隐含了实体

```
>>> t = larc.ok
```

#无界成员函数物件

```
>>> t(object1, 'hello')
```

#必须传递实体

# 类别观念补遗-私有属性

- python的类别属性
  - 都类似C++的public和virtual
  - 有没有类似C++的private?
- 只要C++有, 哪敢没有
  - 于ver 1.5以后, 有了名称破坏(name mangling)
  - 如果名称开头是两个底线字符(结尾不是), 名称的开头就会自动包含类别的名称
- 例如 :
  - 在class类别里有一个\_\_X的名称
  - 将自动变更成\_class\_\_X
  - 必须使用class.\_class\_\_X才能调用出来

# 类别观念补遗-\_\_doc\_\_(1/2)

- 我们使用批注都是以#开头来说明程序代码
  - 可是当程序执行时, 这些批注完全都看不到
- python提供了\_\_doc\_\_来放置你想要说明的字符串(函数, 模块, 类别皆有)
  - 只要开头的部份是字符串而非叙述即可
  - python会将该字符串存到\_\_doc\_\_里(只有储存第一行)

- 范例：有一个module名字叫docstr
- >>> "I am: docstr.\_\_doc\_\_"
- >>> class larc:
- ... "I am: larc.\_\_doc\_\_ or docstr.larc.\_\_doc\_\_"
- ... def method(self, arg):
- ... "I am: larc.method.\_\_doc\_\_ or self.method.\_\_doc\_\_"
- >>> def func(args):
- ... "I am: docstr.func.\_\_doc\_\_"



## 类别观念补遗- `__doc__` (2/2)

- `>>> import docstr`
- `>>> docstr.__doc__`
  - `'I am: docstr.__doc__'`
- `>>> docstr.larc.__doc__`
  - `'I am: larc.__doc__ or docstr.larc.__doc__'`
- `>>> docstr.larc.method.__doc__`
  - `'I am: larc.method.__doc__ or self.method.__doc__'`
- `>>> docstr.func.__doc__`
  - `'I am: docstr.func.__doc__'`
- 你可以直接在交互方式寻找组件的说明文件
- 目前并没有广泛的被使用
  - 比批注没有弹性
  - 总之不管批注还是说明字符串, 写详细都是好事

# 恐怖的陷阱-变更类别属性(1/2)

- 与其说是类别的陷阱, 还不如说是名称空间的陷阱
- 类别物件与**实体对象**都算是可**变更对象**的一种
  - 一切的梦魇又再发生一次(会引起**边际效应**)

- ```
>>> class X:
```
- ```
... a = 1 #类别属性
```
- ```
>>> l = X()
```
- ```
>>> l.a #实体继承
```

  - 1
- ```
>>> X.a = 2          #受影响的不只是X
```
- ```
>>> l.a #l也改变
```

  - 2
- ```
>>> J = X()          #J也会继承改变
```
- ```
>>> J.a
```

  - 2

## 恐怖的陷阱-变更类别属性(2/2)

- Why not?
    - 也是有好处低
    - 不过最好的作法是只修改到**实体对象**的**层级**就好
    - 受影响的物件越少越好
    - 免得**难以**管理(忘记哪里有改过之类的...)
  - 拿它来模拟 C++ 的 `struct, record`
- ```
>>> class Record: pass
>>> X = Record()
>>> X.name = 'bob'
>>> X.job = 'Pizza maker'
```

恐怖的陷阱-多重继承

- 使用多重继承必须十分小心
 - 名称的意义必需靠母类别的次序来解释, 容易混淆
 - 除非必要, 不然别用

```
● >>> class Lister:
●     def __repr__(self): ...
●     def other(self): ...
● >>> class Super:
●     def __repr__(self): ...
●     def other(self): ...
● >>> class Sub(Super, Lister): #会使用Super的__repr__
●     other = Lister.other      #在此指明用Lister的other
●     def __init__(self): ...
```