北京金源万博科技有限公司

金源万博培训中心

Ye Jiang

518@msn.cn

http://www.aipython.com

the 3nd Django presentation 2015

21st Nov 2015

1st December 2015

Outline

Introduction

- This is just a short example
- The comments in the LaTeX file are most important
- This is just the result after running pdflatex
- The style is based on the webpage `http://www.ru.nl/`

python 数字

- python 数字分为整形数及浮点数，整形数包含长整形数
  - 由阿拉伯数字组成不包含小数点的数字，都是整形数
  - 长整形数支持的数字长度比较长，在现在的 Python 的运算过程中，不用考虑 python 支持的数字的长度和空间
  - 浮点数，也就是通常成为的带有小数点的小数，在 python 称之为小数

数字运算

- 数字普通运算支持加、减、乘、除（+ 、-、*、/）操作
- python 的运算大多数情况下是在同类型的基础上进行

```
In [3]: 100+1-2
Out[3]: 99
In [4]: 50*3*0.5
Out[4]: 75.0
In [5]: 100/2
Out[5]: 50
In [6]: 100+'x'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last
<ipython-input-6-c4af87c2b9b9> in <module>()
----> 1 100+'x'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python 字符串

- 字符串是 python 中重要的数据对象
- python 字符串是以单引号、双引号、或者三个三单引号，三个双引号包含的任意的 python 数据对象都可以称为 python 字符串

```
In [7]: "strings"
Out[7]: 'strings'
In [8]: 'strings'
Out[8]: 'strings'
In [9]: "213212[1,2,3,4]{'a':a}"
Out[9]: "213212[1,2,3,4]{'a':a}"
In [10]: """hi
    ....: second line
    ....: last line"""
Out[10]: 'hi\nsecond line\nlast line'
```

转义符

- 在任何的语言里都有转义符这样的概念存在，转义符是让含有特殊意义的字符，失去特殊的意义，按照普通的形式打印

```
In [12]: print "The path is c:\windows\newdir\test\readfiles\views"
The path is c:\windows
eadfilesest
        iews
In [14]: print "The path is c:\\windows\\newdir\\test\\readfiles\\views"
The path is c:\windows\newdir\test\readfiles\views
In [15]: print 'He's name is harry'
  File "<ipython-input-15-b3938ffd1308>", line 1
    print 'He's name is harry'
             ^
SyntaxError: invalid syntax
In [16]: print 'He\'s name is harry'
He's name is harry
In [17]: print r"The path is c:\windows\newdir\test\readfiles\views"
The path is c:\windows\newdir\test\readfiles\views
```

Python 字符串方法

- 字符串方法是针对字符串的操作，方法已经定义并封装，使用简化操作和代码重写
- python 字符串常用方法
  - 字符串填充
  - 字符串删减
  - 字符串变形
  - 字符串切分
  - 字符串连接
  - 字符串判定
  - 字符串查找
  - 字符串替换
  - 字符串编码

字符串填充

- **center(width[,fillchar])**
  - center 居中
- **ljust(width[,fillchar])**
  - ljust() 左对齐
- **rjust(width[, fillchar])**
  - rjust() 右对齐
- **zfill(width)**
  - zfill() 即是以字符 0 进行填充，在输出数值时比较常用
- **expandtabs([tabsize])**
  - expandtabs() 的 tabsize 参数默认为 8。它的功能是把字符串中的制表符（tab）转换为适当数量的空格

Example

```
In [19]: "    aaa  aa      ".center(50)
Out[19]: '                    aaa  aa                    '
In [20]: "    aaa  aa      ".center(50,'#')
Out[20]: '################    aaa  aa      ################'
In [25]: "aaaaa".ljust(20)
Out[25]: 'aaaaa               '
In [26]: "aaaaa".rjust(20)
Out[26]: '               aaaaa'
In [27]: "aaaaa".zfill(20)
Out[27]: '000000000000000aaaaa'
```

字符串删减

- strip() 函数族用以去除字符串两端的空白符
- strip([chars])
- rstrip([chars])
- lstrip([chars])

Example

```
In [33]: "   this is python strings    ".strip()
Out[33]: 'this is python strings'

In [34]: "   this is python strings    ".lstrip()
Out[34]: 'this is python strings    '

In [35]: "   this is python strings    ".rstrip()
Out[35]: '   this is python strings'
```

字符串变形

- **lower()**
  - 将字符串转换为小写
- **upper()**
  - 将字符串转换为大写
- **capialize()**
  - 首字母大写
- **swapcase()**
  - 大小写之间转换
- **title()**
  - 单词首字母大写

Example

```
In [43]: a.capitalize()
Out[43]: 'This is python strings    '

In [44]: "this is python strings    ".capitalize()
Out[44]: 'This is python strings    '

In [45]: "this is python strings    ".swapcase()
Out[45]: 'THIS IS PYTHON STRINGS    '

In [46]: "this is python strings    ".title()
Out[46]: 'This Is Python Strings    '

In [47]: "this is python strings    ".lower()
Out[47]: 'this is python strings    '

In [48]: "this is python strings    ".upper()
Out[48]: 'THIS IS PYTHON STRINGS
```

字符串切分

- **splitlines([keepends])**
- **split([sep[,maxsplit]])**
- **rsplit([sep[,maxsplit]])**

Example

In [57]: "this is python strings ".split() Out[57]: ['this', 'is', 'python', 'strings']

In [58]: "this is python strings ".split(' ',2) Out[58]: ['this', 'is', 'python strings ']

In [59]: "this is python strings ".split(' ') Out[59]: ['this', 'is', 'python', 'strings', '', '', '', '']

In [60]: "this is python strings ".rsplit(' ') Out[60]: ['this', 'is', 'python', 'strings', '', '', '', '']

字符串连接

- **join(seq)**
- join() 函数的高效率（相对于循环相加而言），使它成为最值得关注的字符串方法之一

```
In [63]: "this is python strings    ".join('abc')
Out[63]: 'athis is python strings    bthis is python strings
```

字符串判断

- isalnum()
- isalpha()
- isdigit()
- islower()
- isupper()
- isspace()
- istitle()
- startswith(prefix[, start[, end]])
- endswith(suffix[,start[, end]])

字符串查找

- **count(sub[, start[, end]])**
- **find(sub[, start[, end]])**
- **index(sub[, start[, end]])**

字符串替换

- **replace(old, new[,count])**
  - replace() 函数的 count 参数用以指定最大替换次数
- **translate(table[,deletechars])**
  - translate() 的参数 table 可以由 string.maketrans(frm, to) 生成 translate() 对 unicode 对象的支持并不完备，建议不要使用。

字符串编码

- **encode([encoding[,errors]])**
- **decode([encoding[,errors]])**
- **这是一对互逆操作的方法，用以编码和解码字符串。因为 str 是平台相关的，它使用的内码依赖于操作系统环境，而 unicode 是平台无关的，是 Python 内部的字符串存储方式。unicode 可以通过编码（encode）成为特定编码的 str，而 str 也可以通过解码（decode）成为 unicode**

字符串分片操作

- 字符串是有序的序列
- 索引开始从 0 开始，结束为-1
- 分片操作包含索引开始值，但是不包含终止值，通常是终止值减 1

Example

```
In [68]: "python strings"[::]
Out[68]: 'python strings'

In [69]: "python strings"[0:3]
Out[69]: 'pyt'

In [70]: "python strings"[0:-1]
Out[70]: 'python string'

In [72]: "python strings"[0:-1:2]
Out[72]: 'pto tig'

In [73]: "1234567890"[0::2]
Out[73]: '13579'
```

Python 列表

- **python 列表是 python 内置的数据对象之一**
- **列表是用"[]" 包含，内有任意的数据对象，每一个数据对象以“,” 分割，每个数据对像称之为元素**
- **python 列表是一个有序的序列**
- **python 列表支持任意的嵌套，嵌套的层次深度没有限制**

Python 列表定义

- **[]** 空列表
- **["seq1","seq2",'keyman',808,3.1]**
- **[["seq1","seq2",'keyman',808,3.1],['a'].'b',['c']]** 嵌套的列表
- **list('strings')** list 函数创建列表

列表分片操作

- 列表是有序的序列，支持索引操作，或者按照元素寻找索引
- 列表的索引是从 0 开始，以此类推
- ['s', 't', 'r', 'i', 'n', 'g', 's'][0:-1:2] 操作方法和字符串的操作方法一样，返回的为列表
- ['s', 't', 'r', 'i', 'n', 'g', 's'][0] 根据索引的位置，查找索引对应的元素，返回的为字符串

列表方法

- append
- count
- extend
- index
- insert
- pop
- remove
- reverse
- sort

list.append(object)

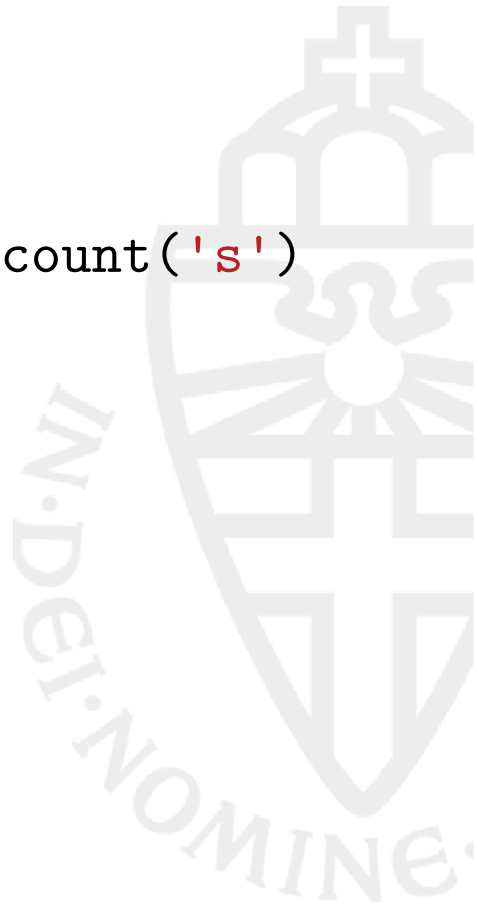
- **append** 是将新的元素附加到列表内，附加的位置在列表最后
- **append** 追加的元素只能是一个元素，不能进行多元素同时操作

```
In [15]: l
Out[15]: ['s', 't', 'r', 'i', 'n', 'g', 's']

In [16]: l.append([1,2,3,4,5])

In [17]: l
Out[17]: ['s', 't', 'r', 'i', 'n', 'g', 's', [1, 2, 3, 4, 5]]
```

list.count(value)

- 统计每个对象在列表的对象中出现的次数
- 返回值为 **number**

```
In [27]: ['s', 't', 'r', 'i', 'n', 'g', 's'].count('s')
Out[27]: 2
```



list.extend(iterable)

- 扩展列表，将可迭代的对象进行拆分，将其追加到列表
- **iterable** 是指可迭代的对象

```
In [30]: l=['s', 't', 'r']

In [31]: l.extend("123")

In [32]: l
Out[32]: ['s', 't', 'r','1', '2', '3']
```

list.index(value, [start, [stop]])

- 根据列表内的元素查找元素所对应的索引
- 返回值为 **number**
- 可选参数 **start stop** 指在什么范围内查找

```
In [36]: l.index('s')
Out[36]: 0

In [37]: l.index('s',0,len(l)-1)
Out[37]: 0

In [38]: l.index('s',1,len(l)-1)
Out[38]: 6
```

list.insert(index, object)

- 在给定的索引位置上，插入相对应的对象，对象可以是任意的
- **insert 方法相当于 list[6]=object**

```
In [41]: l.insert(3,'index3')

In [42]: l
Out[42]: ['s', 't', 'r', 'index3', 'i', 'n', 'g']
```

list.pop(index)

- 弹出列表内的元素，默认是最后一个
- **index** 参数指定弹出指定索引对应的元素

```
In [45]: l.pop(3)
Out[45]: 'index3'

In [46]: l
Out[46]: ['s', 't', 'r', 'i', 'n', 'g', 's', '1', '2]
```

list.remove(value)

- 删除指定的元素，元素应当是在列表中存在，若值不存在引发 **ValueError**

```
In [48]: l.remove('t')

In [49]: l
Out[49]: ['s', 'r', 'i', 'n', 'g', 's', '1', '2']
```

list.reverse()

- 将列表内元素顺序前后对调
- **reverse** 不是反排序

```
In [52]: l
Out[52]: ['s', 'r', 'i', 'n', 'g', 's', '1', '2', '3']

In [53]: l.reverse()

In [54]: l
Out[54]: ['3', '2', '1', 's', 'g', 'n', 'i', 'r', 's']
```

list.sort()

- 将列表的内的元素进行排序
- 排序的顺序为 0-9,A-Z,a-z
- reverse=True 反排序

```
In [66]: l=['6', '5', '4', '3', '2', '1', 's', 'g']

In [67]: l.sort()

In [68]: l
Out[68]: ['1', '2', '3', '4', '5', '6', 'g', 's']

In [69]: l.sort(reverse=True)

In [70]: l
Out[70]: ['s', 'g', '6', '5', '4', '3', '2', '1']
```

列表的其他方法

- list[index]=value
- del list[index]

```
In [72]: l
Out[72]: ['s', 100, '6', '5', '4', '3', '2', '1']

In [73]: del l[2]

In [74]: l
Out[74]: ['s', 100, '5', '4', '3', '2', '1']

In [75]: l[3::]="strings"

In [76]: l
Out[76]: ['s', 100, '5', 's', 't', 'r', 'i', 'n', 'g', 's']
```

什么是 Python 字典

- 用大括号 {} 包裹的以逗号分割的键值对，每个键与值之间使用冒号连接, 构成（key–values) 结构。
- 键和值可以是任意的数据对象，大多数情况还是以数字和字符串的方式构成
- 字典是无序的，键在字典中必须是唯一，在字典中取值的方式是以键寻找相对应的值
- 字典是 python 中的映射数据类型
- 字典不支持拼接 (concatenation) 和重复 (repetirion)

字典定义

- {}
  - 定义空字典，没有任何的键和值
- {'name':'keyman','age':25,'address':'BeiJing'}
  - 创建字典，键与值对应，以逗号分割，键与值以冒号连接
- {}.fromkeys(['key1','key2','key3'])
  - 创建字典, 使用字典方法，值创建键，值为 None，键来自列表的元素
- dict((['x', 1], ['y', 2])) / dict((['x', 1], ['y', 2]))
  - 使用字典函数创建字典
  - dict(zip(["key1",'key2'],(1,2)))
  - dict((['x', 1])) ??

字典访问

- 字典是无序的，定义字典的顺序与生成字典的顺序是不一致的
- 通过字典的键，取得字典键所对应的值
  - d=’name’:’keyman’,’age’:25,’address’:’BeiJing’,’nes’:”a”:1
  - d[’name’]
  - d[’nes’][’a’]
- 字典更新
  - d[’new_key’]=’new_value’
- 字典删除
  - del d[’new_key’]

字典方法 1

- dict.keys()
  - 获取字典内所有键，以列表返回

```
>>> d={"name":'keyman','age':20}
>>> d.keys()
['age', 'name']
```

- dict.values()
  - 获取字典内所有值，以列表返回

```
>>> d.values()
[20, 'keyman']
```

- dict.pop(k,[d])
  - 弹出指定键所对应的值，并且返回弹出的结果, 如果键没有找到，将返回指定 d

```
>>> d.pop('age')
20
>>> d
{'name': 'keyman'}
>>> d.pop('namexx','return value')
'return value'
```

字典方法 One

- dict.update()
  - 更新字典内键与值，键名即使是字符串，在更新时不需要加引号，更新是以赋值的方式进行。

```
>>> d.update(namexx='zhangsan')
>>> d
{'age': 20, 'name': 'keyman', 'namexx': 'zhangsan'}
```

- dict.get()
  - 获取字典内所有键对应的值，与 d['keyname'] 相同
- dict.items()
  - 将字典的键和值放在一个元组中，每一个键和值作为一个元组，放在列表中存储返回

```
>>> d.items()
[('age', 20), ('name', 'keyman')]
```

字典方法 Tow

- dict.popitem()
  - 随机的删除字典中的一个键值对，并且以元组的方式返回

```
>>> d.popitem()
('age', 20)
```

- dict.viewitems()、dict.keyitems()、dict.valueitems()
  - dict.viewitems() 返回字典 dict 的元素视图
  - dict.viewkeys() 返回字典 dict 的键视图
  - dict.viewvalues() 返回字典 dict 的值视图

```
>>> d.viewitems()
dict_items([('name', 'keyman'), ('namexx', 'zhangsan')])
>>> d.viewkeys()
dict_keys(['name', 'namexx'])
>>> d.viewvalues()
dict_values(['keyman', 'zhangsan'])
```

字典方法 Three

- dict.iteritems()、dict.iterkeys()、dict.itervalues()
  - dict.iteritems() 返回字典 dict 的迭代对象
  - dict.iterkeys() 返回字典 dict 的键的迭代对象
  - dict.itervalues() 返回字典 dict 的值迭代对象

```
>>> d.iteritems()
<dictionary-itemiterator object at 0x2aa9100>
>>> d.iterkeys()
<dictionary-keyiterator object at 0x2aa90a8>
>>> d.itervalues()
<dictionary-valueiterator object at 0x2aa9100>
```

字典方法 Four

- dict.setdefault(k，[d])
  - 如果键在字典中，返回字典键所对应的值，如果不在创建该键，值默认为 None，如果 d 给定值, 值为 d

```
>>> d.setdefault('name')
'keyman'
>>> d
{'name': 'keyman', 'namexx': 'zhangsan'}
>>> d.setdefault('name_1')
>>> d
{'name_1': None, 'name': 'keyman'}
```

- dict.copy ()
  - 产生字典副本，或者说是替身

```
>>> d1=d.copy()
>>> d is d1
False
>>> d1['name']='lisi'
>>> d1
{'name_1': None, 'name': 'lisi'}
>>> d
{'name_1': None, 'name': 'keyman'}
```

字典方法 Five

- dict.copy ()
  - 产生字典副本，或者说是替身

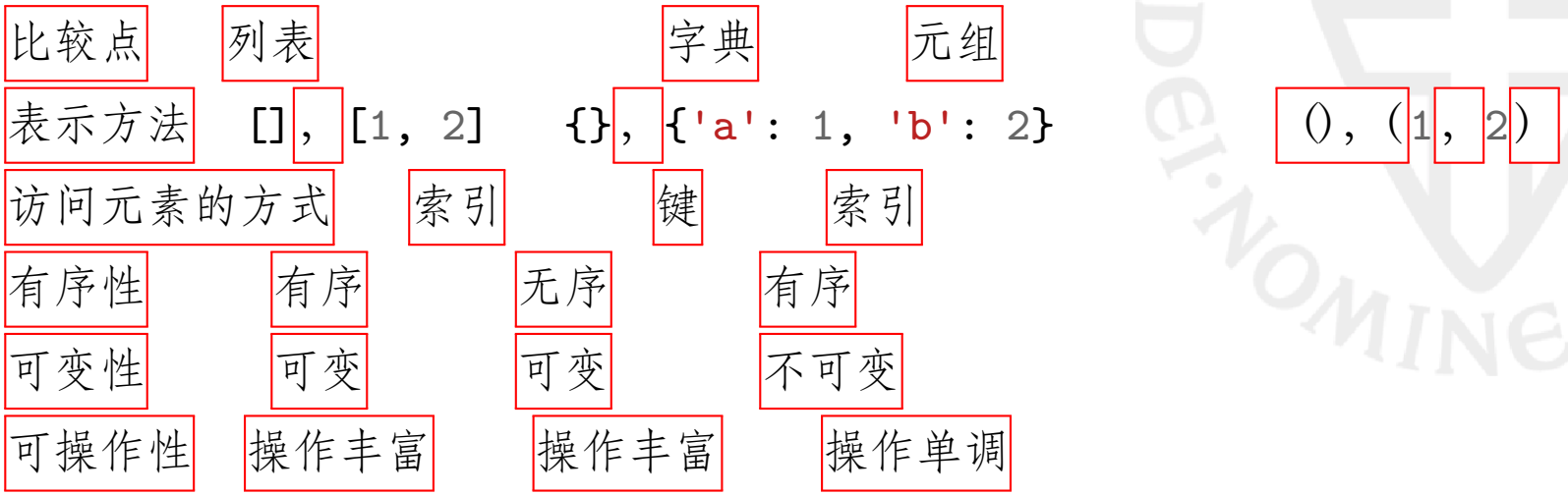
```
>>> d
{'name_1': None, 'name': 'keyman'}
>>> d.clear()
>>> d
{}
```

字典方法 Six

- dict.clear ()
  - 清除字典内的所有键、值

```
>>> d.has_key('name')
True
```

- dict.has_key ()
  - 成员关系检查，判断字典中是否存在指定的键
  - python3.0 has_key 将不在支持，被 in 替代



字典方法视图与迭代对象

- 字典视图
  - 从 2.7 版本开始，Python 中引入了字典视图（Dictionary views）。字典视图是字典的动态视图：它们会与字典保持同步，实时反应出字典的变化。
- 迭代对象
  - 其实就是一个迭代器，它会返回当前的数据, 然后自动的调用内置的 next() 方法来读取，自动的移动到下一个元素, 准备下次的读取, 到达对象末尾时.

python 元组

- Python 元组是 python 重要的数据对象
- 元组大多数情况下，用于系统的输入或者系统的返回，元组的特性是不可以修改，但是支持迭代
- 元组的方法相对来说，比较少，只有 count 和 index
- 元组的定义是以"()" 包含以"," 作为分割元素的对象，元素也可以是任何的数据对象

Example

```
In [82]: ('name','age','address').count('age')
Out[82]: 1
In [83]: ('name','age','address').index('name')
Out[83]: 0
In [84]: ('name','age','address')[1:2]
Out[84]: ('age',)
In [85]: ('name','age','address')[1]='change'
---------------------------------------------------------------------------
TypeError                                 Traceback (most rec
<ipython-input-85-eeaecee1042e> in <module>()
----> 1 ('name','age','address')[1]='change'
TypeError: 'tuple' object does not support item assignment
```

python 数据类型比较

- python 数据类型十分灵活, 分别有列表，字典，元组

python 模块使用

- python 模块分为系统内置的模块、第三方的模块和用户编写的模块
- 默认情况下,python 第三方的模块安装在 python 的安装目录下 site-packages 下，以文件或者目录的形式存放
- 用户模块，程序模块化对区分功能和结构，代码清晰度有很好的帮助
- 默认情况下，在 python 运行时只是加载了少数的系统内置的模块，可以使用 vars() 查看

python 程序

- python 程序可以作为模块运行，也可以作为模块被导入使用
- 如果调用 ___main___ 时就代表直接运行程序当前本身
- 被倒入文件命名要以.py 结尾，并且需要在 python 环境变量可以搜索到的位置

python 环境变量设定

- PYTHONPATH 环境变量设定 python 倒入模块的路径
- 在交互式的 DILE 或者命令行运行时，用户的 cwd 默认加入 pythonpath 变量中
- 临时性的修改 sys.path

```
>>> import sys
>>> sys.path
[ '/usr/lib64/python2.7/site-packages',
'/usr/lib64/python2.7/site-packages/gtk-2.0',
'/usr/lib/python2.7/site-packages']
>>> sys.path.append('/tmp')
>>> sys.path
['/usr/lib64/python2.7/site-packages',
'/usr/lib64/python2.7/site-packages/gtk-2.0',
'/usr/lib/python2.7/site-packages', '/tmp']
```

python 赋值语句

- 赋值语句，variables_name=variables_values
  - 在定义变量名是不得使用系统关键字和特殊符号，变量值可以是任意的数据对象
  - 变量名是未知的类型，变量名的类型取决于被赋予值的类型
- 链式赋值
  - a=b=c=100
  - 将数字 100 分别的赋予给 a,b,c
- 序列解包赋值
  - name,age,address=('keyman',27,'BeiJing')
  - 序列解包赋值变量名和变量值必须相等

import 语句

- import 语句是也是一个赋值语句
  - improt socket
  - 将 socket 模块倒入之后，赋值给 scoekt
- import 语句是将 python 模块倒入，模块应该放置在 pythonpath 目录下
- import module_name 或者 import module_name as alias_name
- from module_name import method 或者 from module_name import method as alias_name

```
import time
import time as t
print t.asctime(),time.asctime()

from math import sqrt
from math import sqrt as q
print sqrt(121),q(121)
```

python 关键字判断

- python 预留的关键字可以使用 keyword 模块检查
- keyword 模块下的 kwlist 方法可以打印系统中所有的被预留的关键字
- keyword 模块下的 iskeyword 可判断对象是否是关键字

```
name="keyman"
age=21
address='BeiJing'
print name,age,address

print "hello",
print "world"
```

python 打印语句

- 在 python2.6、2.7 中 print 是一个语句，在 python3.0 中 python，print 是一个函数
- print 连续打印多个对象，可以用',' 隔开
- 如果要使用多个 print 语句，但结果需要显示在一行，需要在 print 语句之后加上','

```
In [10]: print keyword.kwlist
['and', 'as', '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', 'with', 'yield']

In [11]: keyword.iskeyword('for')
Out[11]: True

In [12]: keyword.iskeyword('list')
Out[12]: False
```

exec 语句

- exec 主要用于执行字符串内包含有 python 的代码
- exec(”print 'hello world'”)

```
code=raw_input('please enter any python code here :')
exec code
```

eval 语句

- eval 主要用于执行字符串内包含有 python 的算术表达式
- exec(”print (1+100-20)/2”)

对象身份比较

- 对象身份的比较，是对值比较的补充
- value_a *is* value_b
- value_a *is* value_b 语句等同于 id(value_a)==id(value_b)
- value_a *is not* value_b

布尔类型

- 布尔逻辑运算符 and,or 和 not 都是 python 的关键字，
- 布尔逻辑运算符优先级
  - not 拥有最高的运算级别，只比所有运算符低一级
  - and 与运算次于 not
  - or 或运算符次于 and
  - not a ==" and b==2 or c==3
  - not a==”” or c==3
  - not a==”” or c==5

标准类型内建函数

- cmp(obj1,obj2)
  - 比较 OBJ1 和 OBJ2，根据比较结果返回整数 i
  - i 小于 0, if obj1 <obj2
  - i 大于 0, if obj1 >obj2
  - i 等于 0,if obj1 == obj2
- repr(obj) 或 ‘obj‘
  - 返回一个对象的字符串表示
- type()
  - 判断对象类型，根据对象返回 str,list,dict,tuple,int,float 等
- isinstance()
  - 判断对象是否是指定对象的实例

什么是函数

- 函数是对程序逻辑进行结构化或过程化的一种编程方法。能将整块代码巧妙地隔离成易于管理的小块, 把重复代码放到函数中而不是进行大量的拷贝–这样既能节省空间有助于保持一致性, 因为只需改变单个的拷贝而无须去寻找再修改大量复制代码的拷贝

函数与过程

- 函数与过程都是可以被调用的实体，可能携带参数或者不携带参数，经过一定的处理, 最后向调用者传回返回值
- 过程是简单, 特殊, 没有返回值的函数

函数与过程

- 函数与过程都是可以被调用的实体，可能携带参数或者不携带参数，经过一定的处理，最后向调用者传回返回值
- 过程是简单，特殊，没有返回值的函数

函数定义

- 函数定义使用 def 语句，def 语句也是一个复合语句
- 函数名应该尽量的体现函数的功能和作用
- 函数的定义时的参数，通常成为形参
- 在调用函数时，传递的参数为实参

```
def demo_function(x,y):
    z=x+y
    return  z
```

位置参数

- 位置参数在函数定义的时候，按照什么顺序定义的传递参数的时候就是按照位置排序
- 位置参数一般情况下都需要将完整的参数传入

```
def function_demo(x,y,z=0):
    z=x+y
    return z

print function_demo(100,200)
```

函数关键字参数

- 关键字参数仅针对函数的调用，这种理念是让函数的调用者通过函数的参数名来区分参数，这样允许参数缺失或者不按照顺序输入
- 也可以通过参数的定义的位置及传入的位置确定参数传入的值
- def function1(vars1,vars2) 当调用函数时，function1(100,200)，100 对应 vars1，200 对用 vars2, 如不按照顺序 function1(vars=200,vars1=100) 传入

函数默认参数

- 默认值参数值在定义函数的时候，将参数赋予了默认值
- 在定义函数时，默认值参数必须在非默认值参数之后
- 在调用时，如果没有传入完全的参数，那么传入的参数将是非默认参数

```
def function_demo1(*args):
  return args

def function_demo2(**args):
  return args

print function_demo1('name','age','address')
print function_demo2(name="keyman",age=27,address="BeiJing")
```

函数参数组

- Python 同样允许程序员执行一个没有显式定义参数的函数, 相应的方法是通过一个把元组 (非关键字参数) 或字典 (关键字参数) 作为参数组传递给函数
- *vars_name 表示的是，列表或者元组形式的参数
- **vars_name 表示的是字典类型的参数

```
def function5_demo(x,y):
  return "after return python code is stoped"
  z=x+y
  return z
  print z
```

函数分类

- 函数大体上可以分为两种, 运算型函数，返回型函数
- 运算型函数大体上可以理解为完成某项运算，一般不需要有返回结果
- 返回型函数，大多数情况下都需要返回一个结果给函数的调用者
- 有无结果返回可以理解为在函数体内部是否含有 return 语句

函数 return 语句

- return 语句可以出现在函数的任何地方
- 一但程序读到 return 是就将停止所有的函数体内部的操作
- 不一定每个函数都必须要有返回

```
def function_demo5(x,y):
    z=x+y
    return z

print callable(function_demo5)
```

函数判断

- callable 函数判断某对象是否会被当成函数调用
- 语法 callable(functions)

```
def function_demo5(x,y):
    z=x+y
    return z

print callable(function_demo5)
```

Python unit 3

- 作用域就是命名空间，使用变量或者函数的范围
- 变量是在赋值时产生的，在哪里赋值就意味存在在什么位置
- 变量的类型由值决定

vars 函数

- vars() 查看系统变量，vars() 得到的结果是一个字典
- scope=vars()
- scope['variables_name']

LEGB

- LEGB 是 python 搜索变量的顺序集规则，一旦搜到就会使用，否则就返回对象为定义
- L 本地作用域
- G 全局作用域
- E 嵌套作用域
- B 内置作用域

全局变量与函数变量

- LEGB 是 python 搜索变量的顺序集规则，一旦搜到就会使用，否则就返回对象为定义

```
In [29]: x=100
In [30]: def demo():
    ....:     x=200
    ....:     print x
In [31]: def demo1():
    ....:     print x

In [33]: demo()
200

In [34]: demo1()
100
```

global

- Global 将内部的变量声明为全局变量，影响全局变量

```
x=100
def change_global():
  global x
  x+=1
change_global()
print x
```

嵌套作用域

```
def func_external(any):
  def func_internal(number):
    return any* number
  return func_internal


x=func_external(9)
print x
print x(2)
```

python 包

- 默认情况下，python 的模块都是命名为 py 结尾的文件，存放在当前目录或者 sys.path 的目录下, 然后使用 import 倒入
- 如果模块比较繁多，传统的方法就会有很不方便，就需要使用 python 包这种概念

python 模块搜索顺序

- python 模块倒入搜索顺序
  - 首先在运行的当前的目录倒入
  - 如当前目录没有发现，按照 sys.path 的目录顺序搜索
  - sys.path 目录下没有发现，前往 python 的安装目录下搜索
- 按照搜索顺序如果在最近找到，就近生效，产生覆盖行为

python 包的组成

- 在 python 包当中必须包含一个 ___init___.py 的文件
- ___init___.py 可以为空，只要它存在，就表明此目录应被作为一个 package 处理
- ___init___.py 中也可以设置相应的内容
- 如果文件包含有多层子目录，每一级目录下都必须有一个 ___init___.py 的文件

python 包的组成

```
wisdom@localhost ~]$ tree  python_packagedir1/
python_packagedir1/
 __init__.py
 subdir1
   __init__.py
      subsubdir2
         __init__.py

In [1]: import python_packagedir1

In [2]: python_packagedir1.__file__
Out[2]: 'python_packagedir1/__init__.py'
```

python 包倒入

```
python_packagedir1/
  demo_test.py
  __init__.py
  subdir1
    __init__.py
      sub1.py
       subsubdir2
            __init__.py
          sub2.py
```

```
In [12]: import  python_packagedir1.subdir1 import sub1
In [13]: from python_packagedir1.subdir1 import *
In [14]: import python_packagedir1
In [15]: from python_packagedir1.demo_test import demo
```

__init__.py 配置

- __all__ = ["Module1", "Module2", "subPackage1", "subPackage2"]
- from module_name import *

面向过程与面向对象编程

- 什么是面向过程编程
  - 根据操作数据的函数或语句块来设计程序的。这被称为面向过程的编程
- 什么是面向对象编程
  - 把数据和功能结合起来，用称为对象的东西包裹起来组织程序的方法。这种方法称为面向对象的编程理念

面向对象术语

- 域
  - 对象可以使用普通的属于对象的变量存储数据。属于一个对象或类的变量被称为域
- 方法
  - 对象也可以使用属于类的函数来具有功能。这样的函数被称为类的方法
- 域和方法可以合称为类的属性，属于另一个对象的数据或者函数元素
- 域有两种类型属于每个实例类的对象或属于类本身。它们分别被称为实例变量和类变量

self 参数

- 类的方法与普通的函数只有一个特别的区别——它们必须有一个额外的第一个参数名称, 调用的时候却不需要为这个参数赋值，按照惯例它的名称是 self
- 参数任何名称，但是强烈建议使用 self 这个名称——其他名称都是不赞成使用。使用一个标准的名称有很多优点, 程序读者可以迅速识别它，如果使用 self 的话，还有在 IDE 集成开发环境会自动加载

self 参数应用

- self 工作原理
  - 一个类称为 MyClass 和这个类的一个实例 MyObject。当调用这个对象的方法 MyObject.method(arg1, arg2) 的时候，这会由 Python 自动转为 MyClass.method(MyObject, arg1, arg2)
  - 意味着如果有一个不需要参数的方法，还是得给这个方法定义一个 self 参数。

```
class class_demo:
    def  function1(self):
        pass

p=class_demo()
class_demo.function1(p)
```

Python 类名命名与方法命名

- 类名通常由大写字母打头, 标准惯例
- 类名使用混合记法 (mixedCase) 或骆驼记法 (camelCase), Python 规范推荐使用骆驼记法的下划线方式, 比如 “update_phone”
- 方法名应当指出对应对象或值的行为，动词加对象，方法应当表明程序员想要在对象进行什么操作

类的编写

- class 语句后跟类名，创建了一个新的类。这后面跟着一个缩进的语句块形成类体, 在类名之后冒号之前也可以有 () 表示继承类
- 使用 def 定义方法，方法必须要拥有 SELF 参数，方法可以有多个
- 也可以设定变量

```
class test_class():
    a=True
    def  fun1(self,x=1,y=2):
        self.x=x
        self.y=y
        return  self.x+self.y
    def func2(self):
         return self.fun1()
```

类与实例

- 类与实例相互关联着: 类是对象的定义, 而实例是" 真正的实物", 它存放了类中所定义的对象的具体信息。
- object 是 “所有类之母”。如果你的类没有继承任何其他父类,object 将作为默认的父类。它位于所有类继承结构的最上层

绑定

```
t=test_class()

print  t
<__main__.test_class instance at 0x1a60680>
```

- Python 严格要求, 没有实例, 方法是不能被调用, 方法必须绑定 (到一个实例) 才能直接被调用
- 实例名 = 类名（参数）
- 当打印实例名时 python 返回，已经在 ___main___ 模块中有了一个 test_class 类的实例, 存储对象的计算机内存地址也打印了出来
- 不同的实例在实例化的时候，都会有不同的内存地址, 实例之间是不会互相影响和依赖
- 非绑定的方法可能可以被调用, 但实例对象一定要明确给出, 才能确保调用成功

对象的方法

- 改进类的方式之一就是给类添加功能。类的功能有一个更通俗的名字叫方法
- 方法定义在类定义中, 只能被实例所调用, 调用一个方法的最终途径必须以下步骤
  - 定义类 (和方法)
  - 创建一个实例
  - 最后一步, 用这个实例调用方法
- 类/对象可以拥有像函数一样的方法，这些方法与函数的区别只是一个额外的 self 变量, 参数代表实例对象本身
- sayHi 方法没有任何参数，但仍然在函数定义时有 self

```
class Person:
    def sayHi(self):
        print 'Hello, how are you?'

p = Person()
p.sayHi()
```

如何决定类属性

- 决定类的属性有两种方法
  - dir() 内建函数
  - 通过访问类的字典属性 __dict__

如何决定类属性 Example

```
class MyClass(object):
    'MyClass class definition'
    myVersion='1.1'
    def showMyVersion(self):
        print MyClass.myVsersion

MyClass.__dict__
dir(MyClass)
```

特殊的类属性 ___doc___

- ___doc___ 是类的文档字符串, 与函数及模块的文档字符串相似, 必须紧随头行 (header line) 后的字符串。文档字符串不能被派生类继承, 派生类必须含有它们自己的文档字符串

特殊的类属性 ___dict___

- ___dict___ 属性包含一个字典, 由类的数据属性组成。访问一个类属性的时候,Python 解释器将会搜索字典以得到需要的属性。如果在 ___dict___ 中没有找到, 将会在基类的字典中进行搜索, 采用“深度优先搜索”顺序。基类集的搜索是按顺序的, 从左到右, 按其在类定义时, 定义父类参数时的顺序。对类的修改会仅影响到此类的字典; 基类的 ___dict___ 属性不会被改动的

特殊的类属性 ___name___

- ___name___ 是给定类的字符名字, 适用于那种只需要字符串(类对象的名字), 而非类对象本身的情况
- 类型对象是一个内建类型的例子, 它有 ___name___ 的属性

特殊的类属性 ___module___

- 在以前的版本中, 没有特殊属性 ___module___
- 一个类已是一种类型, 现在的 python 类和类型之间界线已经很模糊，经典类并不认同这种等价性 (一个经典类是一个类对象, 一个类型是一个类型对象)

___init___ 方法

- 实例化的第一步是创建实例对象。一旦对象创建了,Python 检查是否实现了 ___init___ 方法。
- 注意，这个名称的开始和结尾都是双下划线，不得随意的定义

___del___() 方法

- Python 具有垃圾对象回收机制 (靠引用计数)
- 析构函数直到该实例对象所有的引用都被清除掉后才会执行。

  Python 中的解构器是在实例释放前提供特殊处理功能的方法

析构函数与构造函数实例

```
class InsrCt(object):
        count=0
        def __init__(self):
            InsrCt.count+=1
        def __del__(self):
            InsrCt.count-=1
        def howMany(self):
            return InsrCt.count

a=InsrCt()
b=InsrCt()
a.howMany()
del b
a.howMany()
del a
InsrCt.count
```

”实例化”实例属性

- 在构造器中首先设置实例属性,__init__() 是实例创建后第一个被调用的方法
- 默认参数提供默认的实例安装

```
class MakeMoney():
    def __init__(self,hour,salary=15,tax=0.1):
        self.hour=hour
        self.salary=salary
        self.tax=tax
    def  calcTotal(self,hours=1):
        tax_of_one=round((self.hour*self.salary*self.tax),2)
        return float(hours)*tax_of_one
sfo=MakeMoney(1)
sfo.calcTotal(10)
```

实例属性 vs 类属性

- 类属性仅是与类的数据值相关，类属性和实例无关
- 静态成员引用，即使在多次实例化中调用类，它们的值都保持不变，静态成员不会因为实例而改变它们的值
- 实例属性与类属性的比较，类似于自动变量和静态变量，修改类属性会影响所有的实例

```
class Demo_Class():
  a=100
c=Demo_Class()
c.a ; Demo_Class.a
c.a=200 ; c.a
Demo_Class.a
del c.a ; c.a
Demo_Class.a=1000
```

继承

- 继承描述了基类的属性如何“遗传”给派生类。
- 一个子类可以继承它的基类的任何属性，不管是数据属性还是方法。

```
class a():
  pass
class b(a):
  pass
class c(a,b):
  pass
a.__bases__
b.__bases__
c.__bases__
```

Overriding 方法

```
class a():
    def __init__(self,name):
        self.name=name
        print "class a initiationing ........."
    def greeting(self):
        HI="Name is %s"%self.name
class b(a):
    def __init__(self,age):
        self.age=age
        a.__init__(self)
        report='Name is %s age is %s' \
    %(self.name,self.age)
```

私有化

- 默认情况下, 属性在 Python 中都是“public”，类所在模块和导入了类所在模块的其他模块的代码都可以访问到
- 双下划线 (__)
  - 由双下划线开始的属性在运行时被“混淆”，所以直接访问是不允许的

```
class Person():
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def __secret(self):
        a=("%s age is %s"%(self.name,self.age))

    def public(self):
        return self.__secret()
```

_module

- 模块名带有 _，如 _demo，不允许是用 from module_name import * 倒入

```
class Person():
    def __init__(self,name,age):
        self.name=name
        self.age=age
    def __secret(self):
        a=("%s age is %s"%(self.name,self.age))

    def public(self):
        return self.__secret()
```

类、实例和其他对象的内建函数

- issubclass(sub, sup)
- isinstance(obj1, obj2)
- hasattr(), getattr(),setattr(), delattr()
- dir()

什么是网络编程

- 网络编程我们通常也称作 socket 编程，也就使用套接字编程
- 套接字是为特定网络协议（例如 TCP/IP，ICMP/IP，UDP/IP 等）套件对上的网络应用程序提供者提供当前可移植标准的对象。它们允许程序接受并进行连接，如发送和接受数据。为了建立通信通道，网络通信的每个端点拥有一个套接字对象极为重要。
- 套接字为 BSD UNIX 系统核心的一部分，而且他们也被许多其他类似 UNIX 的操作系统包括 Linux 所采纳。许多非 BSD UNIX 系统（如 ms-dos，windows，os/2，mac os 及大部分主机环境）都以库形式提供对套接字的支持。

网络编程模块

- socket
  - socket 模块既可以完成客户端编程，也可以完成服务端编程，只是用法有所不同
- socket()
  - socket(socket_family,socket_type,protocol0)
  - socket_family 可以是 AF_UNIX 或是 AF_INET，socket_type 可以是 SOCK_STREAM 或是 SOCK_DGRAM，protocol 一般情况下是不填的，默认为 0.
  - 创建 TCP/IP 套接字 tcpSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  - 创建 UDP/IP 套接字 udpSock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
  - 通过调用 socket.socket() 创建一个套接字对象，之后的交互都可以通过调用这个套接字对象的方法来进行。

服务端套接字对象常用的方法

- s.bind()
  - 绑定地址到套接字对象，地址为主机、端口
  - 主机可以是 DNS 域名或者是 ip 地址
  - 段口号通常可以使用 0-65535 之间，但是 1024 以下是公共服务端口，通常使用大于 1024 的端口号
- s.listen() 监听端口
  - 监听端口
- s.accept()
  - 被动的阻塞式的接受连接

客户端套接字对象常用的方法

- s.connect()
  - 初始化连接
- s.connect_ex()
  - connect() 的扩展版本，出错时会返回错误码而不是抛出异常

共用套接字

- socket 编程共用套接字
  - s.recv() 接收 TCP 数据
  - s.send() 发送 TCP 数据
  - s.sendall() 完整发送 TCP 数据
  - s.recvfrom() 接收 UDP 数据
  - s.sendto() 发送 UDP 数据
  - s.getpeername() 连接到当前套接字的远端的地址
  - s.getsockname() 当前套接字的地址
  - s.getsockopt() 返回指定套接字的参数
  - s.setsockopt() 设置指定套接字的参数
  - s.close() 关闭套接字

服务端连接六步骤

- 创建 socket 对象。调用 socket 构造函数。
  socket=socket.socket(familly,type)
  - family 的值可以是 AF_UNIX(Unix 域，用于同一台机器上的进程间通讯)，也可以是 AF_INET（对于 IPV4 协议的 TCP 和 UDP)，至于 type 参数，SOCK_STREAM（流套接字）或者 SOCK_DGRAM（数据报文套接字）,SOCK_RAW（raw 套接字）。

服务端连接六步骤第二步

- 将 socket 绑定（指派）到指定地址上，socket.bind(address)
  - address 必须是一个双元素元组,((host,port)), 主机名或者 ip 地址 + 端口号。如果端口号正在被使用或者保留，或者主机名或 ip 地址错误，则引发 socke.error 异常。

服务端连接六步骤第三步

- 绑定后，必须准备好套接字，以便接受连接请求，socket.listen(backlog)
  - backlog 指定了最多连接数，至少为 1，接到连接请求后，这些请求必须排队，如果队列已满，则拒绝请求。

服务端连接六步骤第四步

- 服务器套接字通过 socket 的 accept 方法等待客户请求一个连接，connection,address=socket.accept()
  - 调用 accept 方法时，socket 会进入'waiting'（或阻塞）状态。客户请求连接时，方法建立连接并返回服务器。accept 方法返回一个含有俩个元素的元组，形如 (connection,address)。第一个元素（connection）是新的 socket 对象，服务器通过它与客户通信；第二个元素（address）是客户的 internet 地址

服务端连接六步骤第五步

- 处理阶段
  - 处理阶段，服务器和客户通过 send 和 recv 方法通信（传输数据）。服务器调用 send，并采用字符串形式向客户发送信息。send 方法返回已发送的字符个数。服务器使用 recv 方法从客户接受信息。调用 recv 时，必须指定一个整数来控制本次调用所接受的最大数据量。recv 方法在接受数据时会进入'blocket' 状态，最后返回一个字符串，用它来表示收到的数据。如果发送的量超过 recv 所允许，数据会被截断。多余的数据将缓冲于接受端。以后调用 recv 时，多余的数据会从缓冲区删除。

服务端连接六步骤第六步

- 传输结束
  - 传输结束，服务器调用 socket 的 close 方法以关闭连接。

客户端连四个接步骤

- 第 1 步，创建一个 socket 以连接服务器
  socket=socket.socket(family,type)
- 第 2 步，使用 socket 的 connect 方法连接服务器
  socket.connect((host,port))
- 第 3 步，客户和服务器通过 send 和 recv 方法通信。
- 第 4 步，结束后，客户通过调用 socket 的 close 方法来关闭连接。

基本 socket 编程服务端

```
from socket import *
host,port,bufsiz=['',9000,1024]
addr=(host,port)
s=socket(AF_INET,SOCK_STREAM)
s.bind(addr)
s.listen(10)
ss,addr=s.accept()
print  'got connected from ',addr
ss.send('byebye')
ra=ss.recv(512)
ss.close()
s.close()
```

基本 socket 编程客户端

```
#!/usr/bin/python
from socket import *
address =('127.0.0.1',9000)
s=socket(AF_INET,SOCK_STREAM)
s.connect(address)
data=s.recv(512)
print 'The data received is',data
s.send('hello')
s.close()
```

TCP 服务器示例

```
import socket
from time import ctime
HOST,PORT,BUFSIZ=["",21567,1024]
ADDR = (HOST,PORT)
tcpSerSock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
tcpSerSock.bind(ADDR)
tcpSerSock.listen(5)
while True:
    print 'Waiting for connection...'
    tcpCliSock,addr = tcpSerSock.accept()
    print 'connected from:',addr
    while True:
      data = tcpCliSock.recv(BUFSIZ)
      if not data:break
      tcpCliSock.send('[%s]%s' %(ctime(),data))
tcpCliSock.close()
tcpSerSock.close()
```

TCP 客户端示例

```
import socket

HOST,PORT,BUFSIZ =["",21567,1024]
ADDR = (HOST, PORT)
tcpCliSock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
tcpCliSock.connect(ADDR)
while True:
    data = raw_input('> ')
    if not data:
        break
    tcpCliSock.send(data)
    data = tcpCliSock.recv(BUFSIZ)
    if not data:
        break
    print data
tcpCliSock.close()
```

Python DB-API

- MySQLdb Python 标准的数据库接口的 Python DB-API（包括 Python 操作 MySQL）。大多数 Python 数据库接口使用标准
- 数据库驱动型程序，都支持很多的数据库，数据库也同样的支持给语言的连接
- MySQL 是一个开源数据库，在当今的开发环境和应用下十分的常见，也是比不可少
- Python 默认支持 sqlite, 在安装 python 程序时已经默认安装。
- MySQLdb 是用于 Python 链接 Mysql 数据库的接口，它实现了 Python 数据库 API 规范 V2.0，基于 MySQL C API 上建立的

Python 支持的事物型数据库

- 支持事物型数据库
  - IBM DB2
  - Firebird (and Interbase)
  - Informix
  - Ingres
  - MySQL
  - Oracle
  - PostgreSQL
  - SAP DB (also known as "MaxDB")
  - Microsoft SQL Server 、Microsoft Access
  - Sybase
- 具体情况可以在此查询
  - https://wiki.python.org/moin/DatabaseInterfaces

Python 支持的嵌入式数据库

- 支持嵌入式数据库
  - asql、GadFly、SQLite、ThinkSQL
- Record-based Databases
  - MetaKit、ZODB、BerkeleyDB、KirbyBase、Durus、atop、buzhug
- XML Databases
  - 4Suite server、Oracle/Sleepycat DB XML
- Graph Databases
  - Neo4j
- Native Python Databases
  - buzhug、SnakeSQL

MySQLdb 安装

- windows 下载网址及安装
  - 在装该插件之前需要安装 mysql 数据库
  - https://pypi.python.org/pypi/MySQL-python/1.2.4
- Linux 安装
  - Linux 需要先安装数据库 mysql-server
  - MySQLdb 使用 MySQL-python RPM 包安装

```
yum install mysql-server mysql MySQL-python
```

DB API 使用流程

- DB API 提供了与数据库协同调用工作，尽可能使用 Python 的结构和语法的成本最低。API 包括以下:
  - 导入 API 模块.
  - 获取与数据库的连接.
  - 发出 SQL 语句和存储过程.
  - 关闭连接

DB API 模块倒入

- API 模块 MySQLdb

```
>>> import MySQLdb
>>> MySQLdb.__file__
'/usr/lib64/python2.7/site-packages/MySQLdb/__init__.pyc'
```

- 如果没有 MySQLdb 模块，则会报错

```
Traceback (most recent call last):
  File "test.py", line 3, inimport MySQLdb
ImportError: No module named MySQLdb
```

数据库建立连接

- conn=MySQLdb.connect(host="localhost",user="root",passwd="sa",db="myta

  - 提供的 connect 方法用来和数据库建立连接, 接收数个参数, 返回连接对象
  - host: 数据库主机名. 默认是用本地主机.
  - user: 数据库登陆名. 默认是当前用户.
  - passwd: 数据库登陆的秘密. 默认为空.
  - db: 要使用的数据库名. 没有默认值.
  - port:MySQL 服务使用的 TCP 端口. 默认是 3306.

事务支持

- commit() 提交
- rollback() 回滚
- 事物操作是需要数据库同时支持，MYSQL 数据库有些引擎不支持事物操作

数据库游标

- 游标（cursor）是系统为用户开设的一个数据缓冲区，存放 SQL 语句的执行结果。每个游标区都有一个名字。用户可以用 SQL 语句逐一从游标中获取记录，并赋给主变量，交由主语言进一步处理。
- 在数据库中，游标是一个十分重要的概念。游标提供了一种对从表中检索出的数据进行操作的灵活手段，就本质而言，游标实际上是一种能从包括多条数据记录的结果集中每次提取一条记录的机制。
- 游标总是与一条 SQL 查询语句相关联因为游标由结果集（可以是零条、一条或由相关的选择语句检索出的多条记录）和结果集中指向特定记录的游标位置组成。

数据库游标 (续)

- 为何要使用游标
  - 使用游标 (cursor) 的一个主要的原因就是把集合操作转换成单个记录处理方式。用 SQL 语言从数据库中检索数据后，结果放在内存的一块区域中，且结果往往是一个含有多个记录的集合。游标机制允许用户在数据库内逐行地访问这些记录，按照用户自己的意愿来显示和处理这些记录。
- 如何使用游标
  - 声明游标。把游标与 T-SQL 语句的结果集联系起来
  - 打开游标
  - 使用游标操作数据
  - 关闭游标

cursor 用来执行命令的方法

- callproc(self, procname, args): 用来执行存储过程, 接收的参数为存储过程名和参数列表, 返回值为受影响的行数
- execute(self, query, args): 执行单条 sql 语句, 接收的参数为 sql 语句本身和使用的参数列表, 返回值为受影响的行数
- executemany(self, query, args): 执行单条 sql 语句, 但是重复执行参数列表里的参数, 返回值为受影响的行数
- nextset(self): 移动到下一个结果集

cursor 接收返回值的方法

- fetchall(self): 接收全部的返回结果行
- fetchmany(self, size=None): 接收 size 条返回结果行. 如果 size 的值大于返回的结果行的数量, 则会返回 cursor.arraysize 条数据
- fetchone(self): 返回一条结果行
- scroll(self, value, mode='relative'): 移动指针到某一行. 如果 mode='relative', 则表示从当前所在行移动 value 条, 如果 mode='absolute', 则表示从结果集的第一行移动 value 条

SQL 语句执行

```
>>> \tiny host,user,passwd,db=('127.0.0.1','testuser','testus
>>> conn=MySQLdb.connect(host=,user,passwd,db)
>>> conn.rollback()
>>> cursor=conn.cursor()
>>>\tiny  cursor.execute('create table test_t (id int, name c
>>> \tiny cursor.execute("insert into test_t values (1,'wisdo
>>> conn.commit();
>>> cursor.execute("select id,name,age from test_t")
```

语句执行

```
>>> param=((id,'name','age'),(3,'zhangsan',21))
>>> sql="insert into test_t values (%s,%s,%s)"
>>> conn.commit();
>>> cursor.executemany(sql,param)
2L
>>> cursor.execute('select * from test_t')
6L
>>> cursor.fetchmany(3)
((0L, 'k', 20L), (1L, 'w', 21L), (0L, 'n', 0L))
```

数据库连接示例

```
#!/usr/bin/python
import MySQLdb
\tiny host,user,passwd,db=('127.0.0.1','testuser1','testuser'
try:
    conn=MySQLdb.connect(host,user,passwd,db)
    cur=conn.cursor()
    cur.execute('select * from test_t')
    cur.close()
    conn.close()
except MySQLdb.Error,e:
    print "MysqlError %d:%s"%(e.args[0],e.args[1])
```

数据库更新及插入示例

```
import MySQLdb
try:
    conn=MySQLdb.connect(host='localhost',user='root',passwd='root',port=3306)
    cur=conn.cursor()
    cur.execute('create database if not exists python')
    conn.select_db('python')
    cur.execute('create table test(id int,info varchar(20))')
    value=[1,'hi rollen']
    cur.execute('insert into test values(%s,%s)',value)
    values=[]
    for i in range(20):
        values.append((i,'hi rollen'+str(i)))
    cur.executemany('insert into test values(%s,%s)',values)
    cur.execute('update test set info="I am rollen" where id=3')
    conn.commit()
    cur.close()
    conn.close()
except MySQLdb.Error,e:
     print "Mysql Error %d: %s" % (e.args[0], e.args[1])
```

Python 配置文件

- 什么是 python 的配置文件
  - python 程序的获得自己的配置信息
  - 增强软件扩展性
  - 软件智能化，便于操作
- 可以为 XML 格式、ini 或者普通的文本格式

Python 模块

- ConfigParser
  - import ConfigParser
  - python 默认安装的模块
  - Linux 默认安装位置/usr/lib64/python2.7/ConfigParser.py

ConfigParser 使用方法

- ConfigParser.ConfigParser 是 ConfigParser 模块下的一个类
- ConfigParser 下方法有
  - ConfigParser.ConfigParser.read 读取配置文件
  - ConfigParser.ConfigParser.sections 获取配置文件章节的名
  - ConfigParser.ConfigParser.options 获取章节名下的选项名
  - ConfigParser.ConfigParser.get 获取章节选项对应的值
  - ConfigParser.ConfigParser.items 获取某章节内所有选项和值以列表返回，每个选项和值是列表中的一个元组

ConfigParser 删除添加方法

- ConfigParser 下删除添加方法，都是修改获取到的内容，不会修改源文件，只有 WRITE 方法会修改。
  - ConfigParser.ConfigParser.remove_section 删除配置文件的章节
  - ConfigParser.ConfigParser.remove_option 删除配置文件选项
  - ConfigParser.ConfigParser.add_section 添加章节
  - ConfigParser.ConfigParser.set 添加指定章节选项的值
  - ConfigParser.ConfigParser.write(self,fp) 将修好的配置文件存入配置文件，fp 为权限 + 文件名

条件判断

- ConfigParser.ConfigParser.has_section 检查是否存在指定的章节
- ConfigParser.ConfigParser.has_option 检查指定的选项是否存在

ConfigParser 示例

- 通过网卡的配置文件检查网络的连通性

```
import ConfigParser
import os
config=ConfigParser.ConfigParser()
config.read('/tmp/nic_back.config')
section=config.sections()[0]
if config.has_option('nic_config','GATEWAY0'):
  ip=config.get('nic_config','GATEWAY0')
  os.system('ping -c 2 -w 1 %s'%ip)
```

python 配置文件

- Python 配置文件使用 [section] 声明章节
- 在 [section] 下配置选项，使用选项名 = 值或者选项名: 值
  - option_name=value
  - option_name:value

```
[section1]
hostname = station120.example.com
domain_name= example.com
ip address= 192.168.0.120
os_type= linux

[section2]
parameter=1
```

python 日志

- python 日志模块
  - logging
- logging.basicConfig 参数
  - level 日志等级，logging.INFO
  - filename 日志文件的位置
  - format 日志存储的格式，"%(levelname)-10s %(asctime)s %(message)s"

日志级别

- 日志级别及对应的数值
- CRITICAL 50
- ERROR 40
- WARNING 30
- INFO 20
- DEBUG 10
- NOTSET 0

python 日志示例

- python 日志配置案例

```
import logging
logging.basicConfig(level=40,
  filename='myPythonlog.log',
  format="%(levelname)-10s %(asctime)s %(message)s")
logging.error("Starting program")
logging.info("Trying to divide 1 by 0")
try:
    print 1/0
except Exception,e:
    logging.critical(e)
logging.info("The division succeeded")
logging.info("Ending program")
```