中国科学院大学
University of Chinese Academy of Sciences

计算机与控制学院
School of Computer and Control Engineering

移动互联网技术

第三章　互联网数据获取技术

Pyhon编程基础（二）

王文杰
wangwj@ucas.ac.cn

本节主要内容

- 概述
- Python安装和使用
- Python的编码规范
- 变量和常量
- 数据类型
- 运算符与表达式
- 流程控制
- 函数
- Python常用内置函数
- **文件处理**
- 目录的常见操作
- 面向对象编程
- 异常
- 数据库编程

Python文件处理放的主要模块

- Python提供了os、os.path、shutil等模块处理文件。
  - os模块是python标准库中的一个用于访问操作系统功能的模块
    - https://docs.python.org/3/library/os.html?highlight=os#module-os
  - os.path模块中包含的函数在不同平台上进行文件的操作
    - This module implements some useful functions on pathnames
    - https://docs.python.org/3/library/os.path.html?highlight=os#module-os.path
  - shutil模块提供高级文件操作，包括文件的删除、拷贝等,也支持文件压缩。
    - https://docs.python.org/3/library/shutil.html?highlight=shutil#module-shuti

文件处理

- 文件分为两类：
  - 文本文件：
  - 二进制文件：
- 无论是文本文件还是二进制文件，其操作流程基本都是一致的：
  1. 首先打开文件并创建文件对象
  2. 然后通过该文件对象对文件内容进行读取、写入、删除、修改等操作
  3. 最后关闭并保存文件内容。

文件的常见操作

- 文件的常见操作包括：
  - 打开文件
  - 读写文件
  - 复制文件
  - 删除文件

1、文件的创建

```
open(file, mode= ‘r’ , encoding=None, …… )
```

- ✓ file参数指定了被打开的文件名称。
- ✓ mode参数指定了打开文件后的处理方式，可见后续表1。
- ✓ encoding参数指定对文本进行编码和解码的方式，只适用于文本模式，可以使用Python支持的任何格式，如GBK、utf8、 CP936等等。

表1-文件的打开模式

参数	描述
r	以只读的方式打开文件
r+	以读写的方式打开文件
w	以写入的方式打开文件。先删除文件原有的内容，再重新写入新的内容。如果文件不存在，则创建一个新的文件。
w+	以读写的方式打开文件。先删除文件原有的内容，再重新写入新的内容。如果文件不存在，则创建一个新的文件。
a	以写入的方式打开文件，在文件末尾追加新的内容。如果文件不存在，则创建一个新文件
a+	以读写的方式打开文件，在文件末尾追加新的内容。如果文件不存在，则创建一个新文件
b	以二进制模式打开文件。可与r、w、a、+结合使用
U	支持所有的换行符号。如：’\r’、’\n’、’\r\n’

内置函数open()

- 如果执行正常，open()函数返回1个文件对象，通过该文件对象可以对文件进行读写操作。如果指定文件不存在、访问权限不够、磁盘空间不足或其他原因导致创建文件对象失败则抛出异常。

```
f1 = open( 'file1.txt', 'r' )      # 以读模式打开文件
f2 = open( ‘file2.txt’, ‘w’)       # 以写模式打开文件，
                                   #文件不存在，则先创建一个文件
```

- 当对文件内容操作完以后，一定要关闭文件对象，这样才能保证所做的任何修改都确实被保存到文件中。

```
f1.close()
```

2、文件的读取

（1）按行读取方式readline()

- readline()每次读取文件中的一行，需要使用永真表达式循环读取文件。
- 但当文件指针移动到文件的末尾时，依然使用readline()读取文件将出现错误。因此程序中需要添加1个判断语句，判断文件指针是否移动到文件的尾部，并且通过该语句中断循环。

例：使用readline()读文件

```
f=open("story.txt")
while True:
    line=f.readline()
    if line:
        print(line)
    else:
        break
f.close()
```

执行结果：

```
Marry had a little lamb,

and then she had some more.

haha!
```

（2）使用readlines()多行读取方式

- 函数readlines()可一次性读取文件中多行数据。
- 使用readlines()读取文件，需要通过循环访问readlines()返回的内容。

```
f=open('story.txt')
lines=f.readlines()
for line in lines:
    print (line)
f.close()
```

执行结果：

```
Marry had a little lamb,

and then she had some more.

haha!
```

（3）一次性读取方式

- 读取文件最简单的方法是使用read()，read()将从文件中一次性读出所有的内容，并赋值给一个字符串变量。

```
f=open('story.txt')
context=f.read()
print(context)
f.close()
```

执行结果：

```
Marry had a little lamb,
and then she had some more.
haha!
```

read([size])	从文本文件中读取size个字符（Python 3.x）的内容作为结果返回，或从二进制文件中读取指定数量的字节并返回，如果省略size则表示读取所有内容

例：使用read()返回指定字节的内容

```
f=open('story.txt')
context=f.read(5)   #读取文件前5个字节的内容
print(context)
print(f.tell())     #返回文件对象当前指针的位置

context=f.read(10) #继续读取10个字节的内容
print(context)
print(f.tell())     #输出文件当前的位置

f.close()
```

执行结果：

```
Marry
5
 had s lit
15
```

3、文件的写入

- 文件的写入同样有多种方法，可以使用write()、writelines()方法写入文件
  - write(str)的参数是一个字符串，就是你要写入文件的内容
  - writelines(sequence)的参数是字符序列，比如列表，也可以是字符串。

```
f1=open('hello1.txt','w+')
f2=open('hello2.txt','w+')

str='hello world!\nhello China!\n'
context=['hello world!\n','hello China!\n']
f1.writelines(context)
f2.write(str)
f1.close()
f2.close()
```

4、文件的删除

```
import os

open('hello.txt','w')
if os.path.exists('hello.txt'):
    os.remove('hello.txt')
```

5、文件的复制

- **file类并没有提供直接复制文件的方法，但可以使用read()、write()方法来实现复制文件的功能。**

```
#创建文件hello.txt
src=open('hello.txt','w')
context=['hello world\n', 'hello China\n']
src.writelines(context)
src.close()
#将hello.txt复制到hello2.txt
src=open('hello.txt', 'r')
dst=open('hello2.txt', 'w')
dst.write(src.read())
src.close()
dst.close()
```

利用shutil模块复制文件

- shutil模块是另一个文件、目录的管理接口，提供了一些用于复制文件、目录的函数。
- copyfile()函数可以实现文件的复制
  - copyfile(src, dst)
  - 其中，参数src表示源文件的路径，dst表示目标文件的路径，均为字符串类型。
- move()函数可以实现文件的剪切
  - move(src, dst)

例子：使用shutil模块实现文件的复制和移动

```
import shutil

#将hello.txt的内容复制给hello2.txt
shutil.copyfile('hello.txt', 'hello2.txt')
#将'hello.txt'移动到当前目录的父目录
shutil.move('hello.txt', '../')
#将hello2.txt移动到当前目录，命名为hello3.txt
shutil.move('hello2.txt', 'hello3.txt')
```

6、文件的重命名

- os模块的函数rename()可以对文件或目录进行重命名。
- 在实际应用中，经常需要将某一类文件修改为另一种类型，即修改文件的后缀名。可以通过函数rename()和字符串查找函数来实现。

```
import os

ls=os.listdir('.') #返回当前目录的文件列表
if 'hello3.txt' in ls:
    os.rename('hello3.txt', 'hello.txt')
```

例：修改后缀名

```
import os

files=os.listdir('.') #返回当前目录的文件列表
for filename in files: #获取当前目录中的每个文件名
    pos=filename.find('.') #查找'.'的所在位置
    if filename[pos+1:]=='html':   #判断文件的后缀是否为html
        newname=filename[:pos+1]+'htm' #若后缀名是html，则改为htm
        os.rename(filename,newname)   #重命名
```

7、处理二进制文件

- 数据库文件、图像文件、可执行文件、动态链接库文件、音频文件、视频文件、Office文档等均属于二进制文件
- Python中常用的处理二进制模块有struct、pickle、shelve、marshal。
- **对于pickle，可以使用pickle.dump将数据结构存储到磁盘，之后再用pickle.load从磁盘获取数据结构。**
- **pickle不能用于读写特殊格式的二进制文件，如GIF文件。对这种格式的文件，必须逐字节处理。**

例：二进制文件存取

```
import pickle

def make_picked_file():
    grades={'tom':[84,88,90,90],
            'jack':[77,87,97,88],
            'marry':[85,None,90,90],
            'alan':[100,88,90,95]}
    outfile=open('grades.dat', 'wb')
    pickle.dump(grades, outfile)

def get_pickle_data():
    infile=open('grades.dat','rb')
    grades=pickle.load(infile)
    return grades

make_picked_file()
print(get_pickle_data())
```

本节主要内容

- 概述
- Python安装和使用
- Python的编码规范
- 变量和常量
- 数据类型
- 运算符与表达式
- 流程控制
- 函数
- Python常用内置函数
- 文件处理
- **目录的常见操作**
- 面向对象编程
- 异常
- 数据库编程

os模块常用目录处理函数。

函数	描述
mkdir(path[,mode=0777])	创建path指定的一个目录
makedirs(name,mode=511)	创建多级目录，name表示为“path1\path2\…”
rmdir(path)	删除path指定的目录
removedirs(path)	删除path指定的多级目录
listdir(path)	返回path指定目录下的所有文件名
getcwd()	返回当前工作目录
chdir(path)	改变当前目录为path指定的目录
walk(op,topdown=True,onerror=None)	遍历目录树
path.isfile(path)	当path指定的是一个文件的名称时，返回True，否则返回False
path.isdir(path)	当path指定的是一个文件夹的名称时，返回True，否则返回False
stat(fname)	返回有关fname的信息，如大小（单位为字节）和最后一次修改时间。详细功能参加在线文档

1、创建目录和删除目录

```
>>> import os
>>> os.getcwd()                                  #返回当前工作目录
>>> os.mkdir(‘hello')                            #创建目录
>>> os.rmdir(‘hello')
>>> os.makedirs('hello\world')
>>> os.chdir(os.getcwd()+'\hello') #改变当前工作目录
>>> os.chdir(‘..’)
>>> os.mkdir(os.getcwd()+'\\test')
>>> os.listdir('.')
```

2、目录的遍历

- 目录的遍历有两种实现方法：递归函数、os.walk()。
- 用递归函数遍历目录d:\Python36\lib

```
import os

def VisitDir(path):
    ls=os.listdir(path)
    for p in ls:
        pathname=os.path.join(path,p)
        if not os.path.isfile(pathname):
            VisitDir(pathname)
        else:
            print(pathname)

if __name__=="__main__":
    path="C:\Python36\libs"
    VisitDir(path)
```

使用os.walk()遍历目录

```
import os

def VisitDir(path):
    for root,dirs,files in os.walk(path):
        for filepath in files:
            print(os.path.join(root,filepath))

if __name__=="__main__":
    path="C:\Python36\libs"
    VisitDir(path)
```

3、其他

```
>>> import os.path
>>> os.environ.get('path')                    #获取系统变量path的值
>>> import time
>>> time.strftime('%Y-%m-%d %H:%M:%S',  #查看文件创建时间
                  time.localtime(os.stat(‘hello').st_ctime))
>>> os.startfile('notepad.exe')                  #启动记事本程序
```

os.path模块

方法	功能说明
abspath(path)	返回给定路径的绝对路径
basename(path)	返回指定路径的最后一个组成部分
commonpath(paths)	返回给定的多个路径的最长公共路径
commonprefix(paths)	返回给定的多个路径的最长公共前缀
dirname(p)	返回给定路径的文件夹部分
exists(path)	判断文件是否存在
getatime(filename)	返回文件的最后访问时间
getctime(filename)	返回文件的创建时间
getmtime(filename)	返回文件的最后修改时间
getsize(filename)	返回文件的大小

os.path模块

方法	功能说明
isabs(path)	判断path是否为绝对路径
isdir(path)	判断path是否为文件夹
isfile(path)	判断path是否为文件
join(path, *paths)	连接两个或多个path
realpath(path)	返回给定路径的绝对路径
relpath(path)	返回给定路径的相对路径，不能跨越磁盘驱动器或分区
samefile(f1, f2)	测试f1和f2这两个路径是否引用的同一个文件
split(path)	以路径中的最后一个斜线为分隔符把路径分隔成两部分，以元组形式返回
splitext(path)	从路径中分隔文件的扩展名
splitdrive(path)	从路径中分隔驱动器的名称

os.path模块

```
>>> path='D:\\mypython_exp\\new_test.txt'
>>> os.path.dirname(path)                          #返回路径的文件夹名
'D:\\mypython_exp'
>>> os.path.basename(path)                         #返回路径的最后一个组成部分
'new_test.txt'
>>> os.path.split(path)                            #切分文件路径和文件名
('D:\\mypython_exp', 'new_test.txt')
>>> os.path.split('')                              #切分结果为空字符串
('', '')
>>> os.path.split('C:\\windows')                   #以最后一个斜线为分隔符
('C:\\', 'windows')
>>> os.path.split('C:\\windows\\')
('C:\\windows', '')
>>> os.path.splitdrive(path)                       #切分驱动器符号
('D:', '\\mypython_exp\\new_test.txt')
>>> os.path.splitext(path)                         #切分文件扩展名
('D:\\mypython_exp\\new_test', '.txt')
>>> os.path.realpath('tttt.py')                        #返回绝对路径
'C:\\Python36\\tttt.py'
>>> os.path.abspath('tttt.py')                         #返回绝对路径
'C:\\Python36\\tttt.py'
>>> os.path.relpath('C:\\windows\\notepad.exe')        #返回相对路径
```

本节主要内容

- 概述
- Python安装和使用
- Python的编码规范
- 变量和常量
- 数据类型
- 运算符与表达式
- 流程控制
- 函数
- Python常用内置函数
- 文件处理
- 目录的常见操作
- **面向对象编程**
- 异常
- 数据库编程

面向对象编程

1. 类和对象
2. 属性和方法
3. 继承

- Python是面向对象的解释型高级动态编程语言，完全支持面向对象的基本功能，如封装、继承、多态以及对基类方法的覆盖或重写。

- Python对面向对象的语法进行了简化，去掉了面向对象中许多复杂的特性。
  - 例如，类的属性和方法的限制符—public、private、protected。
  - Python提倡语法的简单、易用性，这些访问权限靠程序员自觉遵守，而不强制使用。

1. 类和对象

- 在面向对象程序设计中，程序员可以创建任何新的类型，这些类型可以描述每个对象包含的数据和特征，这种类型称为类。
- 类是一些对象的抽象，隐藏了对象内部复杂的结构和实现。
- 类由下面两部分组成：
  - 变量：称为成员变量
  - 函数：称为成员函数

类的定义

- Python使用class关键字定义一个类，类名首字符一般要大写。
- 格式：

```
class Class_name:
    …
```

```
class Person(object):
    """Class to represent a person
    """
    def __init__(self): #类的构造函数，用来初始化对象
        self.name=''
        self.age=0
    def display(self):  #类中定义的函数也称方法
        print("Person(%s, %d)" % (self.name, self.age))
```

- 说明：self是指向对象本身的变量，类似与C++的this指针

类的使用

- 定义了类之后，就可以用来实例化对象，并通过“对象名.成员”的方式来访问其中的数据成员或成员方法。

```
>>> p=Person()
>>> p.name='zhang'
>>> p.age=20
>>> p.display()
Person(zhang,20)
```

类的使用

- 在Python中，可以使用内置函数isinstance()来测试一个对象是否为某个类的实例，或者使用内置函数type()查看对象类型。

```
>>> isinstance(p, Person)
True
>>> isinstance(p, str)
False
>>> type(p)
<class '__main__.Person'>
```

说明

- 当定义一个类时，如果这个类没有任何父类，则将object设置为它的父类，用这种方式定义的类属于新式类。
- 如果定义的类没有设置任何父类，则这种方式定义的类属于经典类。

对象的创建

- 当一个对象被创建之后，包含3方面的特性：
- 对象的标识：
  - 对象的标识用于区分不同的对象，当对象被创建之后，该对象会获取一块存储空间，存储空间的地址即为对象的标识。
- 属性和方法。
  - 对象的属性和方法与类的成员变量和成员函数相对应。

对象的显示

- 可以定义一个方法display，用于显示对象的值。
- Python还提供了一些特殊方法，让我们能够定制对象的打印。如：
  - 特殊方法__str__，用于生成对象的字符串表示；
  - 特殊方法__repr__，返回对象的“官方”表示。
- 在大多数类中，方法__repr__都与__str__相同。

```
class Person(object):
    """Class to represent a person
    """
    def __init__(self): #类的构造函数，用来初始化对象
        self.name=''
        self.age=0
    def display(self):  #类中的定义的函数也称方法
        print("Person(%s, %d)" % (self.name, self.age))
    def __str__(self):
        return "Person(%s, %d)" % (self.name, self.age)
    def __repr__(self):
        return  "Person(%s, %d)" % (self.name, self.age)

if __name__=='__main__':
    p=Person()
    print(p)
    print(p.age)
    print(p.name)
    p.age=25
    p.name='Jack'
    p.display()
    print(str(p))
    print(repr(p))
    print(p)
```

执行结果：

```
Person(, 0)
0

Person(Jack, 25)
Person(Jack, 25)
Person(Jack, 25)
Person(Jack, 25)
```

2. 属性

- Python的构造函数、析构函数、私有属性或方法都是通过名称约定区分的。
- 此外，Python还提供了一些有用的内置方法，简化了类的实现。

（1）类的成员的属性

- Python的类的成员的属性一般分为私有属性和公有属性
  - Python默认情况下所有的属性都是“公有的”，对公有属性的访问没有任何限制，且都会被子类继承，也能从子类中进行访问。
  - 若不希望类中的属性在类外被直接访问，就要定义为私有属性。私有成员一般是在：
    - 类的内部进行访问和操作，或者
    - 在类的外部通过调用对象的公有成员方法来访问

- Python使用约定属性名称来划分属性类型。
  - 若属性的名字没有使用双下划线开始的表示公有属性。
  - 以下划线开头的变量名和方法名有特殊的含义：
    - **_xxx**：受保护成员；
    - **__xxx__**：系统定义的特殊成员；
    - **__xxx**：私有成员，只有类对象自己能访问，子类对象不能直接访问到这个成员，但在对象外部可以通过“对象名._类名__xxx”这样的特殊方式来访问。

❖ 注意：Python中不存在严格意义上的私有成员。

- 类的外部不能直接访问私有属性。
- 但是，Python并没有对私有成员提供严格的访问保护机制，Python提供了直接访问私有属性的方式，可用于程序的测试和调试
- 私有属性访问的格式：

  instance._classname__attribute
- 说明：
  - instance表示实例化对象；
  - classname表示类名；
  - attibute表示私有属性
- 注意：classname之前是单下划线，attibute之前是双下划线

- 虽然可以在外部程序中访问私有成员，但这会破坏类的封装性，不建议这样做。

```
>>> class A:
    def __init__(self, value1=0, value2=0):
        self._value1 = value1
        self.__value2 = value2
    def setValue(self, value1, value2):
        self._value1 = value1
        self.__value2 = value2
    def show(self):
        print(self._value1)
        print(self.__value2)
```

```
>>> a = A()
>>> a._value1
0
>>> a._A__value2 #在外部访问对象的私有数据成员
0
```

（2）数据成员

- 数据成员可以大致分为两类：
  - 属于对象的数据成员：
    - 一般在构造方法__init__()中定义，当然也可以在其他成员方法中定义
    - 在定义和在实例方法中访问数据成员时以self作为前缀，
    - 同一个类的不同对象（实例）的数据成员之间互不影响
    - 只能通过对象名访问
  - 属于类的数据成员：
    - 是该类所有对象共享的，不属于任何一个对象，
    - 在定义类时这类数据成员一般不在任何一个成员方法的定义中
    - 可以通过类名或对象名访问

- 利用类数据成员的共享性，可以实时获得该类的对象数量，并且可以控制该类可以创建的对象最大数量。例如：

```
>>> class Demo(object):
    total = 0
    def __new__(cls, *args, **kwargs):                    #该方法在__init__()之前被调用
        if cls.total >= 3:                                #最多允许创建3个对象
            raise Exception('最多只能创建3个对象')
        else:
            return object.__new__(cls)
    def __init__(self):
        Demo.total = Demo.total + 1


>>> t1 = Demo()
>>> t1
<__main__.Demo object at 0x00000000034A0278>
>>> t2 = Demo()
>>> t3 = Demo()
>>> t4 = Demo()
Exception: 最多只能创建3个对象
>>> t4
NameError: name 't4' is not defined
```

（3）Python的实例属性和静态属性：

- 实例属性是以self为前缀的属性，没有该前缀的属性是普通的局部变量。
- C++中有一类特殊的属性称为静态变量。
  - 静态变量可以被类直接调用
  - 当创建新的实例化对象后，静态变量并不会获取新的内存空间，而是使用类创建的内存空间。因此，静态变量能够被多个实例化对象共享。
  - 在Python中静态变量称为类变量，类变量可以在该类的所有实例中被共享。

```
class Fruit(object):
    price=0    #类属性

    def __init__(self):
        self.color='red'    #实例属性
        zone='China'        #局部变量

if __name__=='__main__':
    print(Fruit.price)      #使用类名调用类变量
    apple=Fruit()           #实例化apple
    print(apple.color)      #打印apple实例的颜色
    Fruit.price=Fruit.price+10   #将类变量加10
    print("apple's price:" + str(apple.price)) #打印apple实例的price
    banana=Fruit()
    banana.color ='yellow'
    print(banana.color )
    print("banana's price:" + str(banana.price)) #打印banana的price
```

执行结果：

```
0
red
apple's price:10
yellow
banana's price:10
```

3、类的方法

- 类的方法也分为公有方法和私有方法。
  - 私有方法不能被模块外的类或方法调用，
  - 私有方法也不能被外部的类或函数调用。
- C++中的静态方法使用关键字static声明，而Python使用函数staticmethod()或@staticmethod修饰器将普通的函数转换为静态方法。
- Python的静态方法并没有和类的实例进行名称绑定，要调用除了使用通常的方法，使用类名作为其前缀亦可。

```
#例9-7：类的方法和静态方法使用
class Fruit(object):
    price=0    #定义类变量

    def __init__(self):
        self.__color="red"   #定义私有变量

    def getColor(self):
        print(self.__color) #打印私有变量

    @staticmethod   #使用@staticmethod修饰器定义静态方法
    def getPrice():
        print(Fruit.price)

    def __getPrice():   #定义私有函数
        Fruit.price=Fruit.price+10
        print(Fruit.price)

    count=staticmethod(__getPrice) #使用staticmethod方法定义静态方法

if __name__=="__main__":
    apple=Fruit()      #实例化apple
    apple.getPrice()  #使用实例调用静态方法
    Fruit.count()      #使用类名调用静态方法
    banana=Fruit()
    Fruit.getPrice()
    Fruit.count()
```

执行结果：

```
0
10
10
20
```

(1) __init__方法

- 构造函数用于初始化类的内部状态，为类的属性设置默认值。
- **Python的构造函数名为__init__。**
  - __init__方法除了用于定义实例变量外，还用于程序的初始化。
  - __init__方法是可选的，若不提供__init__方法，Python将会给出一个默认的__init__方法。

```
class Person(object):
    def __init__(self,name='',age=0):
        self.name=name
        self.age=age
    def __str__(self):
        return "Person('%s', %d)" % (self.name, self.age)

if __name__=="__main__":
    p=Person('Jack', 25)
    print(p)
    p=Person() #可以创建空对象
    print(p)
```

执行结果：

```
Person('Jack', 25)
Person('', 0)
```

(2) __del__方法

- 析构函数用于释放对象占用的资源。
- Python提供了析构函数__del__()。
  - 析构函数也是可选的。若程序中不提供析构函数，Python会提供默认的析构函数。
  - 当对象不再被使用时，__del__方法运行，但是很难保证这个方法究竟什么时候运行，若想指明它的运行，就要显式地调用析构函数：del 对象名
- 由于Python中定义了__del__()的实例将无法被Python的循环垃圾收集器（gc）收集，所以建议只有需要时才定义__del__。

```
class Fruit(object):
    def __init__(self, color): #构造函数
        self.__color=color      #初始化属性color
        print(self.__color)

    def __del__(self):    #析构函数
        self.__color=""
        print("free...")

    def grow(self):
        print("grow...")

if __name__=="__main__":
    color="red"
    fruit=Fruit(color)
    fruit.grow()
    del fruit #显示调用析构函数，不然无法保证析构函数何时被运行
```

执行结果：

```
red
grow...
free...
```

3. 继承

- 在继承关系中，已有的、设计好的类称为父类或基类
- 新设计的类称为子类或派生类。
  - 派生类可以继承父类的公有成员，但是不能继承其私有成员。
  - 如果需要在派生类中调用基类的方法，可以使用内置函数super()或者通过“基类名.方法名()”的方式来实现这一目的。
- Python在类名后使用一对括号表示继承关系，括号中即为父类。
- Python支持多继承，如果父类中有相同的方法名，而在子类中使用时没有指定父类名，则Python解释器将从左向右按顺序进行搜索。

关于在单继承关系中的构造函数：

- python中如果子类有自己的构造函数，不会自动调用父类的构造函数
- 如果需要用到父类的构造函数，则需要在子类的构造函数中显式地调用。
- 如果子类需要扩展父类的行为，可以添加__init__方法的参数。
- 如果子类没有自己的构造函数，则会直接从父类继承构造函数。

使用继承

```
class Fruit(object):     #基类
    def __init__(self, color):
        self.color=color
        print("fruit's color: %s" % self.color)

    def grow(self):
        print("grow...")

class Apple(Fruit):                    #继承了Fruit类
    def __init__(self, color): #子类的构造函数
        Fruit.__init__(self, color) #显式地调用基类的构造函数
        print("apple's color: %s" % self.color)
```

```
class Banana(Fruit):                    #继承Fruit类
    def __init__(self, color): #子类的构造函数
        Fruit.__init__(self, color) #显式地调用父类的构造函数
        print("banana's color: %s" % self.color)

    def grow(self):
        print("banana grow...")

if __name__=="__main__":
    apple=Apple("red")
    apple.grow()
    banana=Banana("yellow")
    banana.grow()
```

执行结果：

```
fruit's color: red
apple's color: red
grow...
fruit's color: yellow
banana's color: yellow
banana grow...
```

- 在派生类中调用基类方法。

```
class Person(object):
    def __init__(self, name = '', age = 20, sex = 'man'):
        self.setName(name)
        self.setAge(age)
        self.setSex(sex)

    def setName(self, name):
        assert isinstance(name, str), 'name must be string.'
        self.__name = name

    def setAge(self, age):
        assert isinstance(age, int), 'age must be integer.'
        self.__age = age

    def setSex(self, sex):
        assert sex in ('man', 'woman'), 'sex must be "man" or "woman"'
        self.__sex = sex

    def show(self):
        print('Name:', self.__name)
        print('Age:', self.__age)
        print('Sex:', self.__sex)
```

```
class Teacher(Person):
    def __init__(self, name='', age = 30, sex = 'man', department = 'Compute
        super(Teacher, self).__init__(name, age, sex)
        # 也可以使用下面的形式对基类数据成员进行初始化
        #Person.__init__(self, name, age, sex)
        self.setDepartment(department)

    def setDepartment(self, department):
        assert isinstance(department, str), 'department must be a string.'
        self.__department = department

    def show(self):
        super(Teacher, self).show()
        print('Department:', self.__department)
```

```
if __name__ =='__main__':
    print('='*30)
    zhangsan = Person('Zhang San', 19, 'man')
    zhangsan.show()

    print('='*30)
    lisi = Teacher('Li Si',32, 'man', 'Math')
    lisi.show()
    print('='*30)
    lisi.setAge(40)
    lisi.show()
```

```
==============================
Name: Zhang San
Age: 19
Sex: man
==============================
Name: Li Si
Age: 32
Sex: man
Department: Math
==============================
Name: Li Si
Age: 40
Sex: man
Department: Math
```

4、多态性

- 所谓多态（polymorphism），是指基类的同一个方法在不同派生类对象中具有不同的表现和行为。派生类继承了基类行为和属性之后，还会增加某些特定的行为和属性，同时还可能会对继承来的某些行为进行一定的改变，这都是多态的表现形式。
- Python大多数运算符可以作用于多种不同类型的操作数，并且对于不同类型的操作数往往有不同的表现，这本身就是多态，是通过特殊方法与运算符重载实现的。

多态性应用1

- 例如，Apple、Banana类继承了Fruit类，因此Apple、Banana具有Fruit类的共性。Apple、Banana类的实例可以替代Fruit对象，同时又呈现出各自的特性。

```
class Fruit(object):
    def __init__(self, color=None):
        self.color=color

class Apple(Fruit):
    def __init__(self, color="red"):
        Fruit.__init__(self, color)

class Banana(Fruit):
    def __init__(self, color="yellow"):
        Fruit.__init__(self, color)
```

```
class FruitShop(object):
    def sellFruit(self, fruit):
        if isinstance(fruit, Apple):
            print("sell apple")
        if isinstance(fruit, Banana):
            print("sell banana")
        if isinstance(fruit, Fruit):
            print("sell fruit")


if __name__=="__main__":
    shop=FruitShop()
    apple=Apple()
    banana=Banana()
    shop.sellFruit(apple)
    shop.sellFruit(banana)
```

执行结果：

```
sell apple
sell fruit
sell banana
sell fruit
```

多态性应用2

```
>>> class Animal(object):       #定义基类
    def show(self):
        print('I am an animal.')
>>> class Cat(Animal):          #派生类，覆盖了基类的show()方法
    def show(self):
        print('I am a cat.')
>>> class Dog(Animal):          #派生类
    def show(self):
        print('I am a dog.')
>>> class Tiger(Animal):        #派生类
    def show(self):
        print('I am a tiger.')
>>> class Test(Animal):         #派生类，没有覆盖基类的show()方法
    pass
```

```
>>> x = [item() for item in (Animal, Cat, Dog, Tiger, Test)]
>>> for item in x:          #遍历基类和派生类对象并调用show()方法
    item.show()

I am an animal.
I am a cat.
I am a dog.
I am a tiger.
I am an animal.
```

本节主要内容

- 概述
- Python安装和使用
- Python的编码规范
- 变量和常量
- 数据类型
- 运算符与表达式
- 流程控制
- 函数
- Python常用内置函数
- 文件处理
- 目录的常见操作
- 面向对象编程
- **异常**
- 数据库编程

异常

- 为什么要使用异常？
- 如何自定义异常
- logging库使用

异常

- 异常是指程序运行时引发的错误，引发错误的原因有很多，例如除零、下标越界、文件不存在、网络异常、类型错误、名字错误、字典键错误、磁盘空间不足，等等。
- 当程序执行过程中出现错误时Python会自动引发异常，程序员也可以通过raise语句显式地引发异常。
- 异常处理是因为程序执行过程中由于输入不合法导致程序出错而在正常控制流之外采取的行为。
- 严格来说，语法错误和逻辑错误不属于异常，但有些语法错误往往会导致异常，例如由于大小写拼写错误而试图访问不存在的对象，或者试图访问不存在的文件，等等。当Python检测到一个错误时，解释器就会指出当前程序流已经无法再继续执行下去，这时候就出现了异常。

异常

- 大部分由程序错误而产生的错误和异常，一般由Python虚拟机自动抛出
- 如果判断某种错误情况，则可以创建相应的异常类的对象，并通过raise语句抛出

- Python虚拟机自动抛出异常示例

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
```

程序代码中通过raise语句抛出异常

```
>>> a=5
>>> if a<0:raise ValueError("数值不能为负数")

>>> a=-2
>>> if a<0:raise ValueError("数值不能为负数")

Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    if a<0:raise ValueError("数值不能为负数")
ValueError: 数值不能为负数
```

异常处理结构

- 异常处理结构主要包括下面几类：

（1）try...except...

（2）try...except...else...

（3）try...except...finally...

（4）可以捕捉多种异常的异常处理结构

（5）同时包含else子句、finally子句和多个except子句的异常处理结构

（1）try...except...

- 其中try子句中的代码块包含可能会引发异常的语句，而except子句则用来捕捉相应的异常。
- 如果try子句中的代码引发异常并被except子句捕捉，就执行except子句的代码块；
- 如果try中的代码块没有出现异常就继续往下执行异常处理结构后面的代码；
- 如果出现异常但没有被except捕获，继续往外层抛出，如果所有层都没有捕获并处理该异常，程序崩溃并将该异常呈现给最终用户。
- 该结构语法如下：

```
try:
    #可能会引发异常的代码，先执行一下试试
except Exception[ as reason]:
    #如果try中的代码抛出异常并被except捕捉，就执行这里的代码
```

（2）try...except...else...

- 如果try中的代码抛出了异常并且被except语句捕捉则执行相应的异常处理代码，这种情况下就不会执行else中的代码；
- 如果try中的代码没有引发异常，则执行else块的代码。
- 该结构的语法如下：

```
try:
    #可能会引发异常的代码
except Exception [ as reason]:
    #用来处理异常的代码
else:
    #如果try子句中的代码没有引发异常，就继续执行这里的代码
```

（3）try...except...finally...

- 在这种结构中，无论try中的代码是否发生异常，也不管抛出的异常有没有被except语句捕获，finally子句中的代码总是会得到执行。该结构语法为：

```
try:
    #可能会引发异常的代码
except Exception [ as reason]:
    #处理异常的代码
finally:
    #无论try子句中的代码是否引发异常，都会执行这里的代码
```

（4）可以捕捉多种异常的异常处理结构

- 一旦try子句中的代码抛出了异常，就按顺序依次检查与哪一个except子句匹配，如果某个except捕捉到了异常，其他的except子句将不会再尝试捕捉异常。该结构类似于多分支选择结构，语法格式为：

```
try:
    #可能会引发异常的代码
except Exception1:
    #处理异常类型1的代码
except Exception2:
    #处理异常类型2的代码
except Exception3:
    #处理异常类型3的代码
...
```

（5）同时包含else子句、finally子句和多个except子句的异常处理结构

```
>>> def div(x, y):
    try:
        print(x / y)
    except ZeroDivisionError:
        print('ZeroDivisionError')
    except TypeError:
        print('TypeError')
    else:
        print('No Error')
    finally:
        print("executing finally clause")
```

```
>>> div(3,5)
0.6
No Error
executing finally clause
>>> div('3',5)
TypeError
executing finally clause
>>> div(3,0)
ZeroDivisionError
executing finally clause
```

本节主要内容

- 概述
- Python安装和使用
- Python的编码规范
- 变量和常量
- 数据类型
- 运算符与表达式
- 流程控制
- 函数
- Python常用内置函数
- 文件处理
- 目录的常见操作
- 面向对象编程
- 异常
- **数据库编程**

数据库编程

- Python提供了一套数据库API（实现数据库编程的方法），它是一种通用的API，可以访问大多数数据库。

  http://wiki.python.org/moin/DatabaseInterfaces

  1. dbm持久字典
  2. SQLite数据库
  3. MongoDB数据库
  4. Redis
  5. MySQL数据库

1. dbm持久字典

- 可以使用持久字典来存储键/值对，持久字典可被保存在磁盘上。
- 若将数据存储到一个dbm支持的词典中，则当下一次启动该程序时，一旦加载了dbm文件，即可再次读取存储在指定键下的值。
- 这种字典与之前介绍的标准Python字典类似，主要区别在于数据是在磁盘上写入和读取的。
- 说明：
  - dbm—是database manager的缩写
  - dbm持久字典与普通字典的另一个不同之处是键和值都必须是字符串类型

（1）创建持久字典

- dbm模块支持使用open函数创建一个新的dbm对象。
- 一旦成功打开，便可以在字典中存储数据、读取数据、关闭dbm对象（以及相关联的数据文件）、移除项和检查字典中是否存在某个键等。

```
import dbm
db=dbm.open('websites','c') #第一个参数是数据库名，第二个参数是打开方式

db['www.python.org']='Pyhton home page' #写入一个数据

print(db['www.python.org'])  #访问一个数据

db.close()  #关闭并保存
```

创建dbm持久字典常用的文件打开模式：

参数	描述
c	打开数据文件以便对其进行读写，必要时创建该文件
n	打开文件以便对其进行读写，但总是创建一个新的空文件。若该文件已经存在，则将被覆盖，已有的内容将会丢失。
w	打开文件以便对其进行读写，但若该文件不存在，则不会创建它。

（2）访问持久字典

- 使用dbm模块时，可将从open函数返回的对象视为一个字典对象。
- 设置字典中的值：

```
db['key']='value'
```

- 获取字典中的值：

```
value=db['key']
```

- 删除字典中的值：

```
del db['key']
```

- 返回包含所有键的一个列表（使用keys方法）：

```
for key in db.keys():
  #   do something…
```

2. SQLite数据库

- SQLite是一个开源的嵌入式关系数据库引擎，体积小巧，可以嵌入到应用程序中，并提供了SQL接口访问数据。
- SQLite支持的数据类型包括：NULL、INTEGER、REAL、TEXT和BLOB，分别对应Python的数据类型：None、int、float、str和bytes
- Python3已经自带了sqlite3模块，sqlite3模块提供访问和操作数据库sqlite的各种功能
- 可从http://www.sqlite.org下载SQLite数据库

SQLite数据库使用

- **访问和操作SQLite数据时，需要首先导入sqlite3模块，然后创建一个与数据库关联的Connection对象：**
  - 创建数据库和表。创建数据库sales，并在其中创建表region，表中包含两个列：id和name，其中id为主码（primary key）

```
import sqlite3

con = sqlite3.connect("sales1.db")  #打开SQLite数据库：sales.db

cur = con.execute("select id, name from region")
                    #查询数据库表的记录内容

for row in cur:     #循环输出结果
    print(row)
con.close()
```

SQLite数据库使用

- 查询数据表中的记录信息
- 数据库表记录的插入、更新和删除操作

3. MongoDB数据库

- MongoDB是一个基于分布式文件存储的文档数据库，可以说是非关系型（NoSQL，Not Only SQL）数据库中比较像关系型数据库的一个
- 具有免费、操作简单、面向文档存储、自动分片、可扩展性强、查询功能强大等特点，对大数据处理支持较好，旨在为 WEB 应用提供可扩展的高性能数据存储解决方案。
- MongoDB 将数据存储为一个文档，数据结构由键值(key=>value)对组成。MongoDB 文档类似于 JSON 对象。字段值可以包含其他文档，数组及文档数
- 爬虫得到的数据很多都是非结构化的，可存储于该数据库中

MongoDB数据库下载和安装

- 下载：Https://www.mongodb.com/
- MongoDB安装成功后，在bin同级建立data\db目录，用于保存数据和配置
- 启动mongodb
  - 在安装目录的bin下，运行：

    mongod --dbpath C:\MongoDB\Server\3.4\data\db
- 验证启动是否成功
  - 启动浏览器，输入地址：http://localhost:27017/，显示下面信息，表示成功运行：

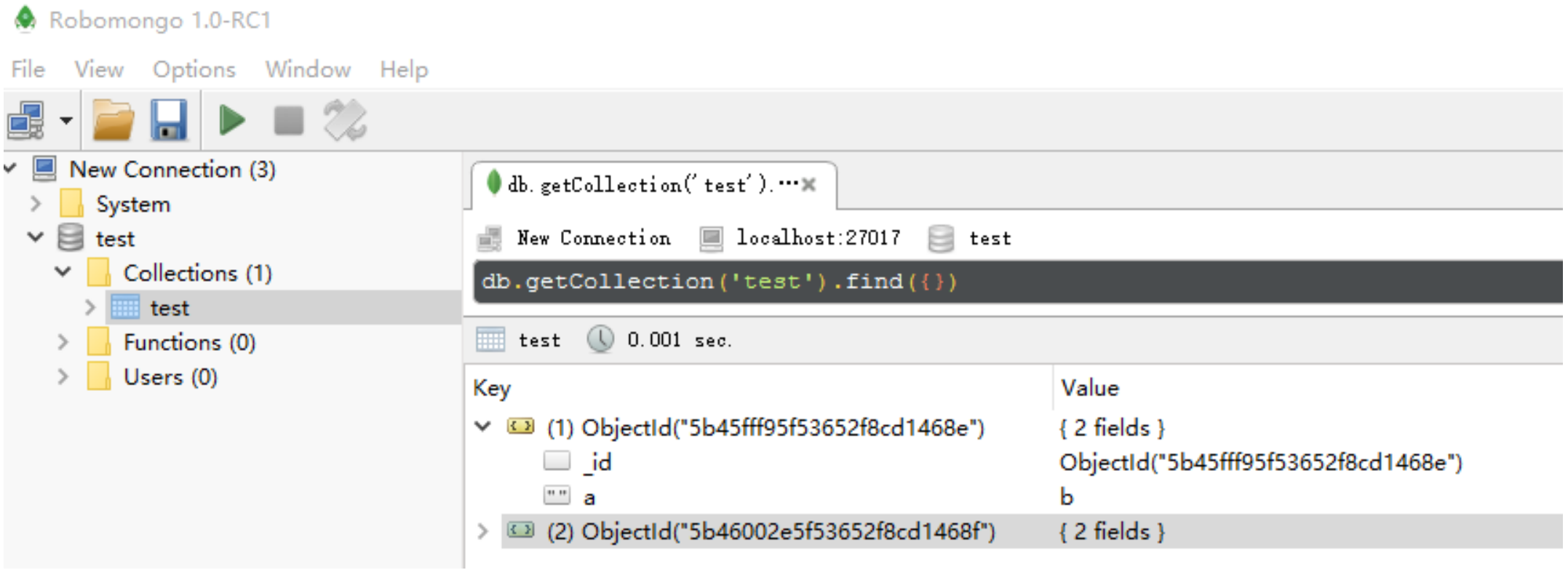
MongoDB数据库

- 以管理员方式启动cmd后，来到bin目录下
- 在data目录下创建一个日志logs目录，然后在logs目录下新建一个文件：mongo.log。即配置了mongoDB的日志文件
- 在管理员cmd窗口，运行下面命令：

```
mongod --bind_ip 0.0.0.0 --logpath C:\MongoDB\Server\3.4\data\logs\mongo.log --logappend --dbpath C:\MongoDB\Server\3.4\data\db --port 27017 --serviceName "MongoDB" --serviceDisplayName "MongoDB" –install
```

可视化工具

- 可视化查看MongoDB的数据
  - 可使用robomongo软件



操作MongoDb数据库

- 打开或创建数据库students

```
> use students
```

- 在数据库中插入数据

```
> zhangsan = {'name':'Zhangsan', 'age':18, 'sex':'male'}
> db.students.insert(zhangsan)
> lisi = {'name':'Lisi', 'age':19, 'sex':'male'}
> db.students.insert(lisi)
```

- 查询数据库中的记录

```
> db.students.find()
```

- 查看系统中所有数据库名称

```
> show dbs
```

操作MongoDb数据库

- Python扩展库pymongo完美支持MongoDB数据的操作

```
>>> import pymongo                                          #导入模块
>>> client = pymongo.MongoClient('localhost', 27017)        #连接数据库，27017是默认端口
>>> db = client.students                                    #获取数据库
>>> db.collection_names()                                   #查看数据集合名称列表
['students']
>>> students = db.students                                  #获取数据集合
>>> students.find()
<pymongo.cursor.Cursor object at 0x0000000030934A8>
>>>) for item in students.find():                           #遍历数据
    print(item)
{'age': 18.0, 'sex': 'male', '_id': ObjectId('5722cbcfeadfb295b4a52e23'), 'name':
  'Zhangsan'}
{'age': 19.0, 'sex': 'male', '_id': ObjectId('5722cc6eeadfb295b4a52e24'), 'name':
  'Lisi'}
```

操作MongoDb数据库

```
>>> wangwu = {'name':'Wangwu', 'age':20, 'sex':'male'}
>>> students.insert(wangwu)                                    #插入一条记录
ObjectId('5723137346bf3d1804b5f4cc')
>>> for item in students.find({'name':'Wangwu'}):              #指定查询条件
    print(item)
{'age': 20, '_id': ObjectId('5723137346bf3d1804b5f4cc'), 'sex': 'male',
'name': 'Wangwu'}
>>> students.find_one()                                        #获取一条记录
{'age': 18.0, 'sex': 'male', '_id': ObjectId('5722cbcfeadfb295b4a52e23'),
'name': 'Zhangsan'}
>>> students.find_one({'name':'Wangwu'})
{'age': 20, '_id': ObjectId('5723137346bf3d1804b5f4cc'), 'sex': 'male',
'name': 'Wangwu'}
>>> students.find().count()                                    #记录总数
3
```

操作MongoDb数据库

```
>>> students.remove({'name':'Wangwu'})                    #删除一条记录
{'ok': 1, 'n': 1}
>>> for item in students.find():
    print(item)
{'name': 'Zhangsan', '_id': ObjectId('5722cbcfeadfb295b4a52e23'), 'sex':
'male', 'age': 18.0}
{'name': 'Lisi', '_id': ObjectId('5722cc6eeadfb295b4a52e24'), 'sex': 'male',
'age': 19.0}
>>> students.find().count()
2
>>> students.create_index([('name', pymongo.ASCENDING)]) #创建索引
'name_1'
>>> students.update({'name':'Zhangsan'},{'$set':{'age':25}})   #更新数据库
{'nModified': 1, 'ok': 1, 'updatedExisting': True, 'n': 1}
>>> students.update({'age':25},{'$set':{'sex':'Female'}})      #更新数据库
{'nModified': 1, 'ok': 1, 'updatedExisting': True, 'n': 1}
>>> students.remove()                                         #清空数据库
{'ok': 1, 'n': 2}
```

操作MongoDb数据库

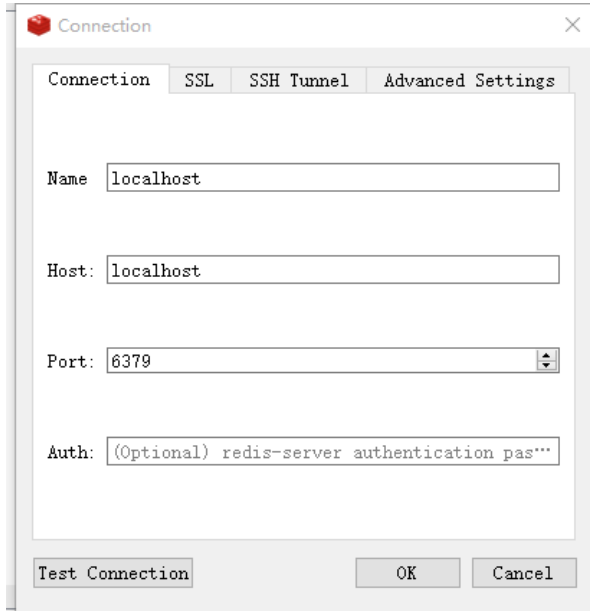
```
>>> Zhangsan = {'name':'Zhangsan', 'age':20, 'sex':'Male'}
>>> Lisi = {'name':'Lisi', 'age':21, 'sex':'Male'}
>>> Wangwu = {'name':'Wangwu', 'age':22, 'sex':'Female'}
>>> students.insert_many([Zhangsan, Lisi, Wangwu])                    #插入多条数据
<pymongo.results.InsertManyResult object at 0x0000000003762750>
>>> for item in students.find().sort('name',pymongo.ASCENDING):       #对查询结果排序
    print(item)
{'name': 'Lisi', '_id': ObjectId('57240d3f46bf3d118ce5bbe4'), 'sex': 'Male', 'age': 21}
{'name': 'Wangwu', '_id': ObjectId('57240d3f46bf3d118ce5bbe5'), 'sex': 'Female', 'age': 22}
{'name': 'Zhangsan', '_id': ObjectId('57240d3f46bf3d118ce5bbe3'), 'sex': 'Male', 'age': 20}
>>> for item in students.find().sort([('sex',pymongo.DESCENDING),('name',pymongo.ASCENDING)]):
    print(item)
{'name': 'Lisi', '_id': ObjectId('57240d3f46bf3d118ce5bbe4'), 'sex': 'Male', 'age': 21}
{'name': 'Zhangsan', '_id': ObjectId('57240d3f46bf3d118ce5bbe3'), 'sex': 'Male', 'age': 20}
{'name': 'Wangwu', '_id': ObjectId('57240d3f46bf3d118ce5bbe5'), 'sex': 'Female', 'age': 22}
```

4. Redis

- redis是一个key-value存储系统
- 它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。
  - 这些数据类型都 支持push/pop、add/remove及取交集并集和差集及更丰富的操作
  - 操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
  - 速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)
- 在分布式爬虫中有比较广泛的应用，可方便维护爬虫序列

4. Redis

- 安装可视化工具：
  - **redis desktop manager**



连接数据库服务器：

MySQL数据库

- 关系型数据库，社区版是免费的
  - mysql-installer-community-5.7.22.1.msi（安装版）
- 安装MySQLDb

  http://trac.edgewall.org/wiki/MySqlDb

操作MySQL数据库

- MySQLDb模块的主要方法：

✓ commit()：提交事务。

✓ rollback()：回滚事务。

✓ 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。

操作MySQL数据库

- ✓ nextset(self)：移动到下一个结果集。
- ✓ 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。。

操作MySQL数据库

- 查询记录

```
import MySQLdb
try:

   conn=MySQLdb.connect(host='localhost',user='root',passwd='root',d
   b='test',port=3306)
     cur=conn.cursor()
     cur.execute('select * from user')
     cur.close()
     conn.close()
except MySQLdb.Error,e:
      print("Mysql Error %d: %s" % (e.args[0], e.args[1]))
```

操作MySQL数据库

- 插入数据

```
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]))
```

技术不会停下脚步，学习永无止境。