

# Python基础

---

路线参考：

<https://github.com/MoRan1607/BigDataGuide>

<https://github.com/heibaiying/BigData-Notes>

## 数据类型转换

- `int(x)` 转整数
- `float(x)` 转浮点数
- `str(x)` 转字符串, `x`必须是数字

```
for x in 数组:
```

```
...
```

`range`函数：

```
range(5)      [0,1,2,3,4]
```

```
range(5,10)   [5,6,7,8,9]
```

```
range(5,10,2) [5,7,9] 元素相隔为2
```

## 字符串格式化

```
name
```

```
num=18
```

```
"我的年纪是%s"%num
```

```
"我的名字是%s，年纪是%s"%(name,num)
```

或者：

```
print(f"我的名字是{name}，年纪是{num}")
```

## 函数

```
def 函数名 (参数):
```

```
    函数体
```

```
    return 返回值
```

## None类型

如果一个函数没有return，那么它默认返回一个None类型的字面量。即为空，**在if判断中None等同于False。**

### 作用域：

在函数体定义的变量是局部变量，函数结束消失。

而**全局变量，函数可以直接调用（不需要通过参数，和java不同）**

**注意：和java相同的是 函数不可以改变全局变量。（可以用global关键字解决问题）**

例如：

```
num=100          #全局变量

def testA():
    num=500      #这里的num是局部变量，不会影响全局变量的num

print(num)       输出100

def testB():
    global num   #这里的global声明了这个变量是全局变量。这个时候就会影响全局变量
    num=600

print(num)       输出600
```

## 数据容器（支持for循环）

- 列表

名字=[x,y,...]      可嵌套: [[...],[...]]

### 内部函数：

```
A=["x","y","z"]

B=A.index("x")          #返回元素索引=0，若不存在则报错

A[0]="c1x"              #修改元素

A.insert(插入位置,元素)  #插入元素

A.append(元素)           #尾部追加元素

A.extend(数据容器)       #在尾部追加多个元素，如另一个list

A.pop(索引)              #删除元素（通过索引）

A.remove(元素)           #删除元素（通过值）注意：相同值的元素remove只能删一个

A.clear()                #清空列表
```

A.count(元素)                      #统计该值在该列表的元素个数

len(A)                              #列表长度

### 遍历:

```
for 临时变量 in 数据容器:
    处理
```

- 元组

同列表相同，但是只要被定义就不能修改。**可以理解为只读的列表**

```
名字=(x,y,...)
```

特殊点：如果元组只有一个元素，后边要有逗号，如：A=(1, )

### 内部方法:

```
A= ("x","y","z")
```

B=A.index("x")                      #返回元素索引=0，若不存在则报错

A.count(元素)                      #统计该值在该元组的元素个数

len(A)                              #元组长度

### 注意点:

```
A=("x","y",[1,2,3])
```

```
A[2][0]=5
```

```
print(A[2][0])                      #输出为5
```

**元组元素的确不可变，但是！元组内的容器是可变的！**

- 字符串

字符串是一个**不可修改的数字容器**（和java一样可更换但无法消失）

如：

```
a="abc"
a[0]="d"
结果a指向"dbc"但是"abc"没有消失
```

### 内部方法：

<code>B=A.index("x")</code>	#返回元素索引=0，若不存在则报错
<code>A.replace(字符串1,字符串2)</code>	#将A里的字符串1替换为字符串2.得到“新字符串”
<code>A.strip("MM")</code>	#去除字符串里的“MM”字符，若无参数默认去除首尾空格
<code>A.count(字符串1)</code>	#返回A中字符串1的出现次数
<code>len(A)</code>	#统计字符串A中的字符个数，即长度

## 数据容器切片操作

### 语法：

序列[起始下标:结束下标:步长]

- 集合

可修改与列表的不同点在于**元素不可重复且无序**。

变量名={}

注意：

集合是{}    列表是[]    元组是()

### 内部方法：

<code>set()</code>	定义一个空集合
<code>A.add(元素)</code>	添加元素
<code>A.remove(元素)</code>	溢出指定元素
<code>A.pop()</code>	从集合随机取一个元素
<code>A.clear()</code>	清空集合
<code>A.difference(B)</code>	返回集合A和集合B的差集，返回也是一个集合
<code>A.difference_update(B)</code>	集合A删除与集合B重复的元素
<code>A.union(B)</code>	取A和B的并集，返回也是一个集合
<code>len(A)</code>	集合元素个数

- 字典

## key不可重复

```
变量名={  
key:val,  
key:val  
}
```

用法:变量名{key}

## 内置方法:

变量名["key"]=val	更改/新增
变量名.pop(key)	移除
A.clear()	清空字典
A.keys()	获得全部的key
len(A)	元素个数

# 函数进阶

## 函数多返回值

```
def 方法名():  
    return 返回值1,返回值2...
```

接收方式:

x,y=方法名()     如果返回三个,左边要多加变量

## 传参方式(看4和5,6即可)

1. 常见的根据位置, a对应1b对应2

add(a,b)            A=add(1,2)

2. 关键字参数

add(a,b)            A=add(b=1,a=2)

3. 缺省参数 (就是默认值)

add(a,b=6)            A=add(1)

4. 不定长参数（即参数数量不固定）

`add(*参数名)`      加了\*后你传参的参数就可以是任意个数  
但是这些参数都会组合为元组

`add(**参数名)`    如果是\*\*那么则组合为字典

注意传参形式要是`key=val`，如：`add('age'=6, 'name'='c1x')`

5. 函数作为参数

```
def add(compute):  
    a=compute(1,2)  
    return a
```

6. lambda匿名函数

\*注意：

- （1）匿名函数没有名字
- （2）匿名函数只能使用一次

定义：

`lambda 参数:函数体`    （直接返回函数体内容）

例如：

```
def add(compute):  
    a=compute(1,2)  
    return a
```

用匿名函数传参：

```
add(lambda x,y:x+y)
```

## 文件读取和写入

### 文件打开/创建

```
f=open(name,mode,encoding)
```

```
f.close()
```

 关闭文件

你也可以用以下写法自动就可以关闭文件

```
with open as f:  
    ...对文件的操作
```

- name:文件具体路径
- mode:打开文件的模式（访问模式）读取【r】？写入【w】？追加【a】？
- encoding: 编码格式（如UTF-8）
- f: 是文件对象，拥有属性和方法

## 读取文件内容

```
f=open(name,r,encoding)
f.read(num)
f.readlines()
f.readline()
```

- f: 文件对象
- num: 读取文件中的字节长度，如果没有传参默认所有数据
- readlines函数，**返回列表**，每行数据为一个元素
- readline函数一次只能读取一行数据

**注意点：（这里举个例子）**

```
f.read(2)
f.read()
f.readlines()
```

如果这三行代码同时存在，假设f文件对象内容是：

"我是clx" 第一行代码读取内容为"我是" 第二行则是"clx" 第三行得到空列表[]、

为什么呢？**这是因为读取就相当于光标的移动，每次读取光标不会回归原点，所以这一次的读取将影响下一次读取。**

## 写入文件

```
f=open(name,w或者a,encoding)           #不存在则创建，存在则打开

f.write("内容")      #将内容写入内存缓冲区，可用\n表示换行
f.flush()            #刷新内容，将数据从内存写入文件。
```

**注意：**

**1.write()+flush()内容才会真正写入文件。**

**2.参数为w，删除以前内容重新写入。参数为a在文件末尾追加内容**

## 异常处理

```
try:
    可能异常的代码
except:
    出现异常执行代码
else:    #可选
    没有异常，处理的代码
finally: #可选
    无论有无异常都会执行
```

例如:

```
try:
    f=open('c\lx.txt','r')    #读取异常,即没能找到该文件
except:
    f=open('c\lx.txt','w')    #创造该文件
```

**捕获指定异常:**

```
try:
    异常代码
except 异常名字 as e:        #其余异常无法捕获
    处理代码
```

**捕获多种异常:**

```
try:
    异常代码
except (异常1, 异常2...) as e:    #其余异常无法捕获
    处理代码
```

**捕获所有异常: (只要是异常都捕获)**

```
try:
    异常代码
except Exception as e:        #其余异常无法捕获
    处理代码
```

## 模块

模块导入方式:

```
import 模块名

from 模块名 import 类/变量/方法

from 模块名 import *

import 模块名 as 别名

from 模块名 as import 功能名字 as 别名
```



# 自定义模块

1.新建python文件（模块名就是python文件名）

2.定义函数即可

**注意点1:**

```
from 模块1 import test
from 模块2 import test

test(1,2)
```

如上面的代码，如果两个模块的方法重名。后引入的方法会覆盖前面的方法！

**注意点2:**

```
if __name__=='__main__':
    代码。。。
```

上图代码作用相当于主函数，当然我们知道一般我们不写他程序也会执行。下面我们演示一下它的作用：

```
#模块clx:
def test(a,b):
    print(a+b)
print(test(1,2))

#其他模块:
from clx import test

print(test(2,3))
#输出将会是:3
    #5

#这是因为我们导入clx时    print(test(1,2))被执行了，程序从clx模块开始执行
如果我们这样写:
from clx import test
if __name__=='__main__':
    print(test(2,3))
#输出只有5，因为我规定了程序的执行入口是当前模块
```

**注意点3:**

我们知道from 模块名 import \*表示引入模块的所有资源，这个“所有”我们是可以指定的

```
#clx模块
__all__=['test1']

def test1():
    print('A')
```

```
def test2():
    print('B')

#其他模块
from clx import *
test1() #可以执行
test2() #不可以执行, __all__指定了可以导入的资源
```

## 自定义python包

本质就是包含多个python文件/模块的文件夹

导入包:

```
import 包名.模块名
from 包名 import 模块名
from 包名.模块名 import 方法名
```

## 第三方python包

- 数据分析: pandas包
- 大数据计算: pyspark, apach-flink
- 图形可视化: matplotlib, pyecharts

## pyecharts模块

官方文档: <https://pyecharts.org/#/zh-cn/>

各种可视化图表源代码: <https://gallery.pyecharts.org/#/README>

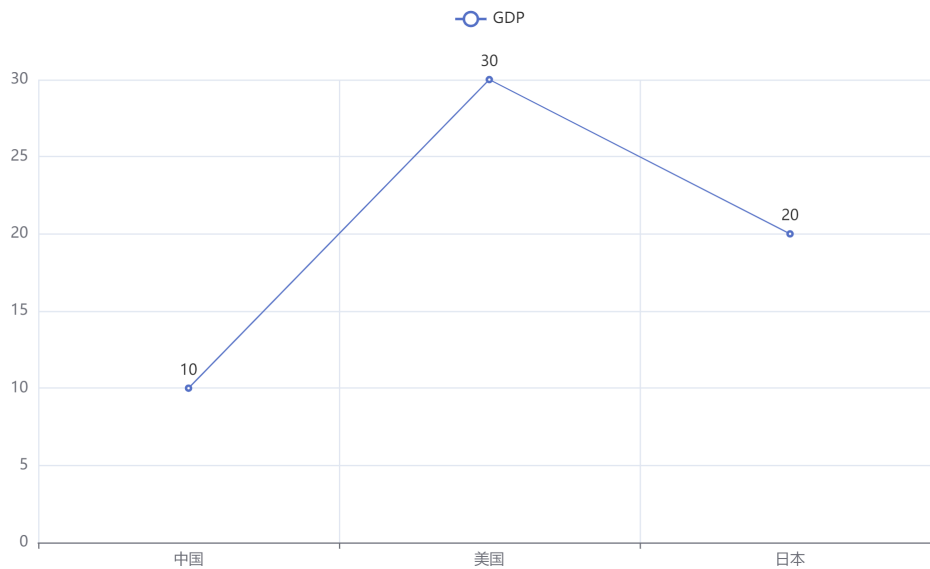
- 折线图

```
#导入Line功能构造折线图
from pyecharts.charts import Line

#1. 获得折线图对象
line=Line()
#2. 添加X,Y轴数据
line.add_xaxis(["中国", "美国", "日本"])
line.add_yaxis("GDP", [10, 30, 20])

#3. 生成图表
line.render() #将图像变成文件, 并不会直接展示图像
```

用谷歌浏览器打开生成的html文件:



## 全局配置和系列配置

全局配置就是整体配置如标题等，系列配置就是x,y轴这些

常用的全局配置：

```
#引入标题,图例,工具箱,视觉映射选项
from pyecharts.options import TitleOpts, LegendOpts, ToolboxOpts, VisualMapOpts
#设置全局配置,用set_global_opts方法
line.set_global_opts(
    #1. 设置标题
    title_opts=TitleOpts(title="GDP折线图展示", pos_left="center", pos_bottom="1%"),
    #参数表示: title标题, pos_left左右位置 (这里选择center表示居中), pos_bottom表示距离底部的百分比

    #2. 设置图例
    legend_opts=LegendOpts(is_show=True),
    #参数表示: 是否展示图例

    #3. 设置工具箱
    toolbox_opts=ToolboxOpts(is_show=True),
    #参数表示: 是否展示工具箱

    #4. 设置视觉映射
    visualmap_opts=VisualMapOpts(is_show=True)
)
```



## 数据处理(这里只处理了美国文件，其他国家过程一样不演示)

```
import json
#处理数据
file=open("美国.txt","r",encoding="UTF-8")
f_data=file.read()#读取所有内容

#删除不符合JSON格式的开头(替换操作)
f_data=f_data.replace("json_1629344292311_69436(", "")
#去除不符合JSON格式的结尾(切片操作)
f_data=f_data[:-2]

#将JSON格式换成字典格式(用json包)
f_dict=json.loads(f_data)

#获得字典内部的列表
trend_data=f_dict['data'][0]['trend']

#获取日期作为X轴(这里只取314号之前的数据)
x_data=trend_data['updateDate'][:314]

#取确诊人数作为Y轴
y_data=trend_data['list'][0]['data'][:314]
```

## 下面是生成图表:

```
#导入Line功能构造折线图
from pyecharts.charts import Line

#1. 获得折线图对象
line=Line()

#2. 引入标题,图例,工具箱,视觉映射选项
from pyecharts.options import TitleOpts, LegendOpts, ToolboxOpts, VisualMapOpts
#设置全局配置,用set_global_opts方法
```

```

line.set_global_opts(
    #1. 设置标题
    title_opts=TitleOpts(title="三国疫情确诊情
况", pos_left="center", pos_bottom="1%"),
    #参数表示: title标题, pos_left左右位置 (这里选择center表示居中), pos_bottom表示距离底部
    的百分比

    #2. 设置图例
    legend_opts=LegendOpts(is_show=True),
    #参数表示: 是否展示图例

    #3. 设置工具箱
    toolbox_opts=ToolboxOpts(is_show=True),
    #参数表示: 是否展示工具箱

    #4. 设置视觉映射
    visualmap_opts=VisualMapOpts(is_show=True)
)

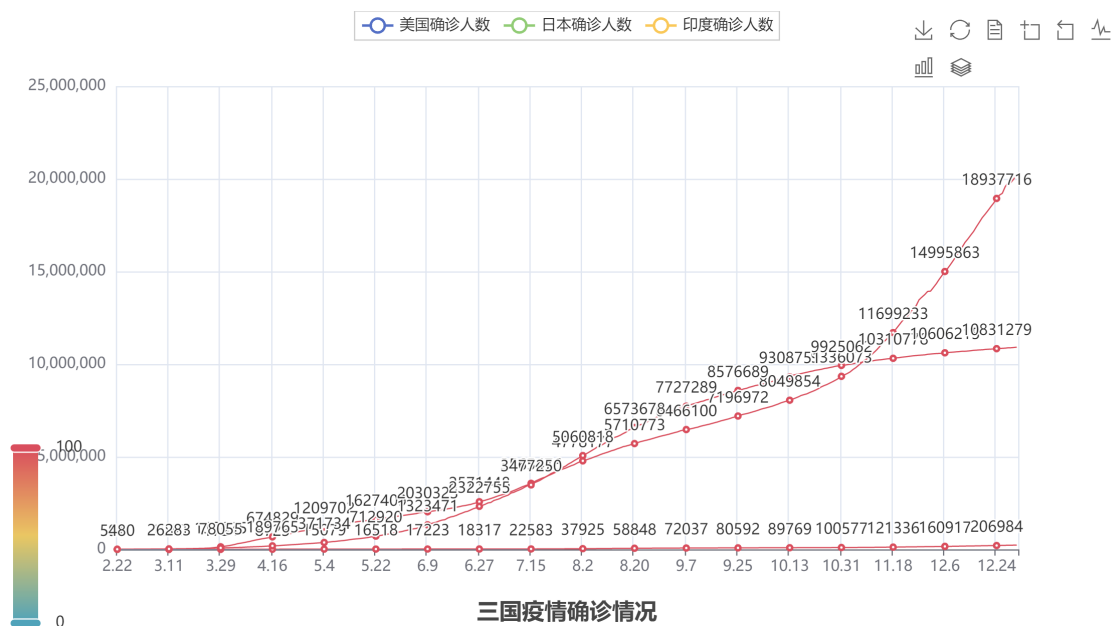
#3. 添加X,Y数据
line.add_xaxis(x_data)#X轴是公用的, 所以添加一个即可
line.add_yaxis("美国确诊人数", y_data)
line.add_yaxis("日本确诊人数", y_data2)
line.add_yaxis("印度确诊人数", y_data3)

#4. 生成图表
line.render()    #将图像变成文件, 并不会直接展示图像

file.close()
file2.close()
file3.close()

```

运行结果:



完整代码见: "D:\课外辅导\大数据开发学习\笔记\python\案例数据集\折线图案例\test.py"

# 对象

```
class 类名:
    name=None
    age=None
    __salary=80000#前面__的变量表示私有变量
#构造函数
    __init__(self,name,age):
        self.name=name
        self.age=age
        print("构造函数")
#自定义函数
    def move(self):
        print(f"打印成员变量:{self.name}")
        ...
#前面加__的表示私有方法
    def __show(self):
        print(self.__salary)

#内置函数:
    def __str__(self):#相当于java 的toString
        return f"显示{self.name}"

    def __lt__(self,other):#用于小于符号"<"的比较(使相同类对象之间能够比较)
        return self.age<other.age

    def __le__(self,other):#逻辑和__lt__一样，但是可以比较"<="
        return self.age<=other.age

    def __eq__(self,other):#和java的equal()+hashmap()一样
        return self.age==other.age

#对象初始化
A=对象名("clx",21)
pay=A.__salary #无法获得私有成员变量，报错
A.__salary =30000 #给私有成员变量赋值，无效，但不报错
```

## 注意点：

### 1.成员方法的self参数必须写

- self用来表示类对象自身
- 被调用时self会自动被传入（传参时忽视）
- **方法内部访问类的成员变量，必须用self**

### 2.私有成员变量/方法在内部是可调用的

# 继承和多态

## 单继承

```
class 子类(父类):  
    ...
```

## 多继承

```
class 子类(类1,类2...):  
    ...
```

## pass关键字

```
class 子类(类1,类2...):  
    pass
```

由于已经继承了很多类拥有了许多功能，不需要写其他东西了。这时候可以用pass关键字代替

### 注意点：

1.多继承中若父类有同名方法/属性

### 先继承的优先级大于后继承的

当然你也可以自己重写方法。

2.子类调用父类成员/方法：

```
方式1:  
父类名.变量/方法  
  
方式2:  
super().变量/方法
```

## 类型注解

用于申明方法参数，传参数据类型。（主要是方便看，写错了也不会影响运行）

```
#数据类型注解  
变量名: 类型=...  
#如:  
age: int=10  
stu: student=student()  
my_list :list=[1,2,3]  
my_list :list[int]=[1,2,3] #list[int]类似于泛型  
  
#函数类型注解(-> int表示返回类型是int)  
def func(x:int,y:int) -> int:  
    ...
```

## 多态

和java有点不同，java要父类类型指向子类对象，python更加简单。

```
def animal_speak(animal:Animal):
    animal.speak()
#Dog和Cat都是Animal的子类
dog=Dog()
cat=Cat()
#直接传参就行了（调用的都是自己的speak方法）
animal_speak(dog)
animal_speak(cat)
```

## 数据库

连接方式：

```
from pymysql import Connection
#获取MySQL的连接对象
conn=Connection(
    host='localhost',    #主机名字（本地）
    port=3306,           #端口
    user='root',
    password='c1x1586869556'
)

#获得数据库信息
print(conn.get_server_info())

#关闭数据库
conn.close()
```

执行非查询语句：

```
#1.通过连接对象得到游标对象
cursor=conn.cursor()
#2.选择数据库
conn.select_db("数据库名字")
#3.SQL操作
cursor.execute("执行的SQL语句")
#4.提交操作
conn.commit()
```

第四步提交操作你可能觉得麻烦，我们可以设置连接对象为自动提交：



```
conn=Connection(  
    host='localhost',    #主机名字（本地）  
    port=3306,           #端口,mysql默认3306  
    user='root',  
    password='clx1586869556',  
    autocommit=True      #设置自动提交  
)
```

### 执行查询语句：

操作基本相同，但是最后一步不同。

```
#1.通过连接对象得到游标对象  
cursor=conn.cursor()  
#2.选择数据库  
conn.select_db("数据库名字")  
#3.SQL操作  
cursor.execute("执行的SQL语句")  
#4.读取数据  
results: tuple=cursor.fetchall()    #results是元组类型（tuple）  
#其内容大概是：((行数据),(行数据)...) 
```

## Pyspark

spark是分布式计算框架，用于调用服务器集群。

### 编程模型

**SparkContext**类对象是，pyspark编程一切功能的入口。

pysspark编程一般有以下步骤：

- 读取数据，得到RDD对象 （数据可以来自JSON文件，文本文档，数据库数据等等，也可以是python的数据容器对象/字符串）
- 通过RDD对象完成数据计算的需求
- 将处理完后的RDD对象写出文件，转换为list等操作

### RDD

所谓RDD——弹性分布式数据集

在一系列操作中

- 数据存储 在RDD对象里
- 对数据计算的方法也是RDD的成员方法
- **所有计算方法的返回值还是RDD对象**

## 基本操作

```
from pyspark import SparkConf, SparkContext

#设置环境变量，可以通过控制台命令"where python"找到，python解释器位置
import os
os.environ['PYSPARK_PYTHON']='E:/python3.10.9/python.exe'      #最好是python10版本

#1. 创建SparkConf类对象
conf=SparkConf().setMaster("local[*]").setAppName("test_spark_app")
#setMaster("local[*]")表示集群位于本地，setAppName("test_spark_app")表示设置该Spark程序名字为：test_spark_app

#2. 通过conf对象创建SparkContext类对象
sc=SparkContext(conf=conf)
#打印当前spark版本
print(sc.version)

#停止Spak程序
sc.stop()
```

### 其中：

```
conf=SparkConf.setMaster("local[*]").setAppName("test_spark_app")
#等同于：
conf=SparkConf()
conf.setMaster("local[*]")
conf.setAppName("test_spark_app")
#这两个方法都返回SparkConf类对象
```

### 将Python数据容器对象的数据存入RDD对象：

```
rdd=sc.parallelize(数据容器对象)
#查看数据
print(rdd.collect())
```

### 读取本地文件到RDD对象：

```
rdd=sc.textFile(文件路径)
```

# 常用的计算方法（算子）

## 1.Map算子

将RDD的数据一条条处理，其参数是处理逻辑的方法。返回值是新的RDD

示例：

```
#1. 创建SparkConf类对象
conf=SparkConf().setMaster("local[*]").setAppName("test_spark_app")
#setMaster("local[*]")表示集群位于本地，setAppName("test_spark_app")表示设置该Spark程序名字为：test_spark_app

#2. 通过conf对象创建SparkContext类对象
sc=SparkContext(conf=conf)
#打印当前spark版本
print(sc.version)

rdd=sc.parallelize([1,2,3,4,5])
#将每个数据*10
#1. 我先定义逻辑函数
def func(data):
    return data*10
#2. 执行Map算子
rdd2=rdd.map(func)
#查看结果
print(rdd2.collect())
```

我们还可以通过lambda表达式使函数为匿名函数，更简单：

```
#2. 执行Map算子
rdd2=rdd.map(lambda x:x*10)
```

输出：

[10, 20, 30, 40, 50]

## 2.flatMap算子

和Map算子一样，都是对每个数据进行计算但是不同的是它有解除嵌套的操作。

例如：

Map算子能做到：（对每个数据进行分割操作）

```
["c1x 15", "hello 16"] --> [ ["c1x", "15"], ["hello", "16"] ]
```

可见Map算子操作后会出现嵌套列表，如果我们不希望出现嵌套如：

```
["c1x", "15", "hello", "16"]
```

则需要flatMap

示例：

```
rdd=sc.parallelize(["c1x 985 1000万/年", "barry hello ok", "good well"])
rdd2=rdd.map(lambda x:x.split(" "))
```

结果：

```
[['c1x', '985', '1000万/年'], ['barry', 'hello', 'ok'], ['good', 'well']]
```

```
rdd=sc.parallelize(["c1x 985 1000万/年", "barry hello ok", "good well"])
rdd2=rdd.flatMap(lambda x:x.split(" "))
```

结果：

```
['c1x', '985', '1000万/年', 'barry', 'hello', 'ok', 'good', 'well']
```

### 3.reduceByKey算子

针对KV型RDD（如二元组），按Key分组，根据提供的聚合逻辑，对数据进行聚合。

聚合逻辑函数注意事项：

- 参数要有2个
- 参数类型与返回值类型要一致

示例：

```
rdd=sc.parallelize([("c1x",140),("c1x",138),("barry",140),("barry",132)])
#求两个人的总成绩
rdd2=rdd.reduceByKey(lambda x,y : x+y)
#查看结果
print(rdd2.collect())
```

结果：

```
[('c1x', 278), ('barry', 272)]
```

# 案例

读取txt文件中各个单词出现的次数

```
from pyspark import SparkConf, SparkContext
#设置环境变量，可以通过控制台命令"where python"找到，python解释器位置
import os
os.environ['PYSPARK_PYTHON']="E:/python3.10.9/python.exe"

#1. 创建SparkConf类对象
conf=SparkConf().setMaster("local[*]").setAppName("test_spark_app")
#setMaster("local[*]")表示集群位于本地，setAppName("test_spark_app")表示设置该Spark程序名字为：test_spark_app

#2. 通过conf对象创建SparkContext类对象
sc=SparkContext(conf=conf)
#.....

#读取文件
rdd=sc.textFile("D:/课外辅导/大数据开发学习/笔记/python/案例数据集/读单词/hello.txt")
#每行一个字符串
#提取所有的单词（还要解开嵌套）
word_rdd=rdd.flatMap(lambda x:x.split(" "))
#已经得到单词列表，但是要技数，我们要想办法将每个单词编程（k,v）的二元数组形式所以：
KV_rdd=word_rdd.map(lambda x:(x,1))
#分组聚合
all_add=KV_rdd.reduceByKey(lambda a,b: a+b)
#展示结果
print(all_add.collect())

#.....
#停止Spak程序
sc.stop()
```

运行结果：

```
[('itcast', 4), ('python', 6), ('itheima', 7), ('spark', 4), ('pyspark', 3)]
```

## 4.fileter算子

用于过滤数据

```
rdd=sc.parallelize([1,2,3,4,5,6])
#对数据过滤(保留偶数)
rdd2=rdd.filter(lambda a : a%2==0)

print(rdd2.collect())
```

输出：

```
[2, 4, 6]
```

## 5.distinct算子

对RDD对象里的数据进行去重操作

```
rdd=sc.parallelize([1,2,3,4,5,6,1,2,3,5,4,6,2])
#对数据过滤(保留偶数)
rdd2=rdd.distinct()

print(rdd2.collect())
```

输出:

[1, 2, 3, 4, 5, 6]

## 6.sortBy算子

对RDD对象的数据进行排序（可自定义排序规则）

而sortbykey()就是按照KV里的key排序，和这个参数用法一样

```
rdd.sortBy(func,ascending=False,numPartitions=1)
```

- func是函数，返回排序依据变量
- ascending=False则降序，为True则升序
- numPartitions表示多少区排列（可忽略）

例如:

```
rdd=sc.parallelize([('A',2),('B',8),('C',1),('D',0),('E',9)])
#x表示列表里的每个元素【即元组】，x[1]指的就是元组的2号元素
rdd2=rdd.sortBy(lambda x:x[1],ascending=False,numPartitions=1)#ascending=False表示降序

print(rdd2.collect())
```

结果:

[('E', 9), ('B', 8), ('A', 2), ('C', 1), ('D', 0)]

## 7.union算子

### 2个RDD合并（合并但不去重）

```
rdd=sc.parallelize([1,2,3])
rdd2=sc.parallelize([4,5,6])
rdd3=rdd.union(rdd2)
```

结果：

```
[1, 2, 3, 4, 5, 6]
```

## 8.join算子

### 对两个RDD实现内/外连接（只能作用与二元元组）

```
rdd=sc.parallelize([(1001,'clx'),(1002,'barry'),(1003,'tom'),(1004,'bob')])
rdd2=sc.parallelize([(1001,'老板'),(1002,'科研')])

rdd3=rdd.join(rdd2)#内连接
rdd4=rdd.leftOuterJoin(rdd2)#左外连接
rdd5=rdd.rightOuterJoin(rdd2)#右外连接

print('内连接: '+rdd3.collect())
print('左外连接: '+rdd4.collect())
print('右外连接: '+rdd5.collect())
```

运算结果：

```
[(1001, ('clx', '老板')), (1002, ('barry', '科研'))]
```

```
[(1001, ('clx', '老板')), (1002, ('barry', '科研')), (1003, ('tom', None)), (1004, ('bob', None))]
```

```
[(1001, ('clx', '老板')), (1002, ('barry', '科研'))]
```

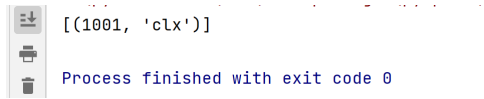
## 9.intersection算子

### 求两个RDD的交集

```
rdd=sc.parallelize([(1001,'clx'),(1002,'barry'),(1003,'tom'),(1004,'bob')])
rdd2=sc.parallelize([(1001,'老板'),(1002,'科研'),(1001,'clx')])
rdd3=rdd.intersection(rdd2)

print(rdd3.collect())
```

运行结果：



```
[(1001, 'clx')]
Process finished with exit code 0
```

## 10.groupbykey算子

针对KV型，按照key分组

```
rdd=sc.parallelize([('clx',140),('barry',135),('clx',120),('barry',140)])
rdd2=rdd.groupByKey()
print(rdd2.collect())
```

运行结果：

```
[('clx', <pyspark.resultiterable.ResultIterable object at 0x0000023F996EED00>), ('barry', <pyspark.resultiterable.ResultIterable object at 0x0000023F996EF2B0>)]
```

由于分组成的是一个迭代器对象，所以显示的是地址。

我们可以做以下处理变成list，这样好看一些：

```
print(rdd2.map(lambda x:(x[0],list(x[1]))).collect())
```

运行结果：

```
[('clx', [140, 120]), ('barry', [135, 140])]
```

## 11.常用的action算子

### (1)first

取出RDD中第一个数据，返回的类型根据第一个数据来定。

### (2)take算子

取前n个数据，组成list返回

### (3)top算子

对RDD数据集降序排序，取前n个

```
rdd=sc.parallelize([10,1,5,6,8,2])

print(rdd.first())
print(rdd.take(3))
print(rdd.top(3))
```

输出：



```
10  
[10, 1, 5]  
[10, 8, 6]
```

## 12.takeSample算子

```
rdd=sc.parallelize([10,1,5,6,8,2,89,100,101,520])  
print(rdd.takeSample(False,5,1))
```

- 参数1: true:允许取同一个数据。false:不允许取同一个数据
- 参数2: 取样要几个
- 随机数种子 (传一个数字就行了, 随便) 【若种子一样那么随机数无论取几次理论也一样】

运行结果:

```
[89, 101, 520, 100, 2]
```

## 13.foreach算子

和map算子功能相同, 但是它没有返回值, 独自输出

```
rdd=sc.parallelize([10,1,5,6,8,2,89,100,101,520])  
rdd.foreach(lambda x:print(x*10))
```

运行结果:

```
10  
20  
5200  
890  
60  
1010  
50  
100  
20
```

## RDD的输出

即将RDD转换为文件/python对象

## 转换为python对象

### 1. Collect算子

```
rdd.collect()
```

将rdd数据**形成一个list对象**

### 2.reduce算子

我们指代rdd的reducebykey(fun)是分组聚合，而这里的reduce(fun)是**按逻辑两两聚合**。

```
rdd.reduce(lambda a,b:a+b)
```

### 3.take算子

**取前N个数据，做成列表返回**

```
list=rdd.take(3)           #取前3个元素做成列表
```

### 4.count算子

返回RDD的**数据总条数**

**countbykey()就是针对KV元组类型计数(返回字典)**

```
num=rdd.count()

a=rdd.countbykey()  #a是字典
```

## 转换到文件中

### 1.saveAsTextFile算子

```
rdd=sc.parallelize([1,2,3,4,5,6,1,2,3,5,4,6,2])
rdd.saveAsTextFile(路径)
```

需要配置环境：

- 下载Hadoop
- 配置环境变量指定Hadoop位置

```
import os
os.environ['HADOOP_HOME']='Hadoop路径'
```

- 下载Winutils.exe并放入Hadoop下的bin目录
- 将hadoop.dll放入C:/windows/System32