数据科学之路：02-3 Pandas(上)

Pandas

Pandas是基于Numpy的专门用于数据分析的开源Python库，该库是为了解决数据分析任务而创建的，它能够以最简单的方式进行数据拼接、数据抽取和数据聚合等操作，并且提供了高效地操作大型数据集所需的工具。

- pandas提供了使我们能够快速便捷地处理结构化数据的大量数据结构和函数
- pandas兼具NumPy高性能的数组计算功能以及电子表格和关系型数据库灵活的数据处理功能
- 对于金融行业的用户，pandas提供了大量适合于金融数据的高性能时间序列功能和工具
- 学统计的人会对R语言比较熟悉，R提供的data.frame对象功能仅仅是pandas的DataFrame所提供的功能的一个子集

```
# 版本
import numpy as np
import pandas as pd
pd.__version__
```

数据结构

Pandas 有两种自己独有的基本数据结构：Series 和 DataFrame，这两种结构使得Pandas在处理表格数据非常高效。

Series

Series是具有灵活索引的一维数组，它可以通过列表或数组创建。

```
# 创建Series
x = pd.Series([1,2,3,4,5,6])
print(x)
print('-----------------')

# 输出元素
```

```
print(x.values)
print('------------------')

# 输出索引
print(x.index)
```

索引

```
# 设置索引
y = pd.Series([1,2,3,4],
              index = ['a', 'b', 'c', 'd'])
print(y)
print('------------------')
print(y['c'])

# 通过字典生成Series
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
print(population)
print('------------------')
print(population['California':'Illinois'])
```

DateFrame

DataFrame是具有灵活行索引和灵活列名称的二维数组。

```
# 利用Series生成DataFrame
data = pd.DataFrame({'population': population})
print(data)
print('------------------')
print(data.index)
print('------------------')
print(data.columns)
print('------------------')
```

```
# 通过DataFrame()方法
print(pd.DataFrame(population, columns=['population']))
print('-----------------')
```

索引和选择

索引

```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

索引指的是列，**切片**指的是行。

```
# 标签索引
print(data['area'])
print('-----------------')

print(data.area)
print('-----------------')

print(data['Florida':'Illinois'])
print('-----------------')

print(data[1:3])
print('-----------------')

print(data[data.area > 200000])
```

选择

- loc[]
- iloc[]
- ix[]

```
# loc[]方法
print(data.loc[:'Illinois', :'pop'])
print('-----------------')

# iloc[]方法
print(data.iloc[:3, :2])
print('-----------------')

# ix[]方法
print(data.ix[:3, :'pop'])
print('-----------------')

# 限制选择
print(data.loc[data.area > 200000, ['area', 'pop']])
```

选择数组

```
print(data.values)
print('-----------------')
print(data.T)
```

导入数据

常用方法

- pd.read_csv():从文件、URL、文件型对象中加载带分隔符的数据，默认分隔符为逗号。
- pd.read_table():从文件、URL、文件型对象中加载带分隔符的数据，默认分隔符为制表符“\t”。
- pd.read_excel():从EXCLE的xls文件中加载数据。

常用参数

- path ： 表示文件系统位置、URL、文件型对象的字符串
- sep/delimiter：用于对行中个字段进行拆分的字符序列或正则表达式
- header：用做列名的行号。默认为0（第一行），若无header行，设为None
- names：用于结果的列名列表，结合header=None
- skiprows: 需要忽略的行数
- na_values：一组用于替换NA的值
- nrows：需要读取的行数（从文件开始处算起）
- verbose：打印各种解析器信息，比如“非数值列中缺失值的数量”

- encoding：用于unicode的文本格式编码。例如，“utf-8”表示用UTF-8 编码的文本

```
# 导入iris.csv文件
df1 = pd.read_csv('./data/iris.csv')
print(df1.head())
print('----------------------------------------------------')

df2 = pd.read_table('./data/iris.csv', sep = ',')
print(df2.head())
```

```
# 导入brain_body.txt文件
df3 = pd.read_csv('./data/brain_body.txt', sep = '\t')
print(df3.head())
print('----------------------------------------------------')
df4 = pd.read_table('./data/brain_body.txt')
print(df4.head())
```

注意：载入xls或xlsx文件可能会提示你需要安装xlrd包，不然无法载入（在命令提示符窗口输入conda install xlrd或pip install xlrd安装）

```
# 导入IRIS.xls文件
data1 = pd.read_excel('./data/IRIS.xlsx', sheet_name= 'Sheet1')
print(data1.head())
print('----------------------------------------------------')
data2 = pd.read_excel('./data/IRIS.xlsx', sheet_name= 'Sheet2')
print(data2.head())
```

中文编码问题

```
# 导入datacsv文件
data3 = pd.read_csv('./data/data.csv')
```

```
data4 = pd.read_csv('./data/data.csv', encoding = 'gb2312')
data4.head()
```

查看和检测数据的方法

查看、检查数据

- df.head(n)：查看DataFrame对象的前n行
- df.tail(n)：查看DataFrame对象的最后n行
- df.shape()：查看行数和列数
- http://df.info()：查看索引、数据类型和内存信息
- df.describe()：查看数值型列的汇总统计
- s.value_counts(dropna=False)：查看Series对象的唯一值和计数
- df.apply(pd.Series.value_counts)：查看DataFrame对象中每一列的唯一值和计数

创建测试对象·

创建测试对象

- pd.DataFrame(np.random.rand(20,5))：创建20行5列的随机数组成的DataFrame对象
- pd.Series(my_list)：从可迭代对象my_list创建一个Series对象
- df.index = pd.date_range('1900/1/30', periods=df.shape[0])：增加一个日期索引

导入数据

导入数据

- pd.read_csv(filename)：从CSV文件导入数据
- pd.read_table(filename)：从限定分隔符的文本文件导入数据
- pd.read_excel(filename)：从Excel文件导入数据
- pd.read_sql(query, connection_object)：从SQL表/库导入数据
- pd.read_json(json_string)：从JSON格式的字符串导入数据
- pd.read_html(url)：解析URL、字符串或者HTML文件，抽取其中的tables表格
- pd.read_clipboard()：从你的粘贴板获取内容，并传给read_table()
- pd.DataFrame(dict)：从字典对象导入数据，Key是列名，Value是数据

输出数据

导出数据

- df.to_csv(filename)：导出数据到CSV文件
- df.to_excel(filename)：导出数据到Excel文件
- df.to_sql(table_name, connection_object)：导出数据到SQL表
- df.to_json(filename)：以Json格式导出数据到文本文件

分层索引

到目前为止，我们主要关注一维和二维数据，分别存储在Pandas的Series和DataFrame对象中。通常超越这一点并存储更高维度的数据（即由多于一个或两个键索引的数据）是有用的。虽然熊猫确实提供Panel和Panel4D原生地处理三维和四维数据（见对象旁白：面板数据），在实践中更为常见的模式是利用分层索引（也被称为多索引）并入多个索引级别在单个索引内。通过这种方式，高维数据可以在熟悉的一维内紧凑地表示Series和二维DataFrame物体。

```
# 追踪城市在两个不同年份的人口数据
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
print(pop)
print('------------------------------')

#索引
print(pop[('California', 2010):('Texas', 2000)])
print('------------------------------')

pop[[i for i in pop.index if i[1] == 2010]]
```

Pandas MultiIndex：创建多索引

```
index = pd.MultiIndex.from_tuples(index)
print(index)
```

```
pop = pop.reindex(index)
print(pop)
print('--------------------------------')
# 索引
print(pop[:, 2010])
```

unstack():将多重索引的Series转换为传统索引的DataFrame

```
pop_df = pop.unstack()
pop_df
```

stack():将传统索引的DataFram转换为多重索引的Series

```
pop_df.stack()
```

缺失数据

在Python中，表示为NaN的数据为缺失数据，有着几种表达方法：

- None
- np.nan

```
pd.Series([1, np.nan, 2, None])
```

空值操作

- isnull():生成指示缺失值的布尔编码
- notnull()：isnull()的反面
- dropna():删除空值
- fillna():填充空值
- isnull():返回数据的过滤版本

- fillna()：返回填充或估算缺失值的数据副本

```
# 检测空值
data = pd.Series([1, np.nan, 'hello', None])
data.isnull()

#data.notnull()
```

```
# 删除空值
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
print(df)
print('------------------------')
print(df.dropna())
print('------------------------')
print(df.dropna(axis='columns'))
print('------------------------')
df[3] = np.nan
print(df)
print('------------------------')
print(df.dropna(axis='columns', how='all'))
print('------------------------')
print(df.dropna(axis='index', thresh=3))
```

```
# 填充空值
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
print(data)
print('------------------------')
print(data.fillna(0))   # 将缺失值全部替换成0
print('------------------------')
data.fillna(method='ffill')   # 将缺失值替换成前一个元素(forward-fill
print('------------------------')
data.fillna(method='bfill')   # 将缺失值替换成后一个元素(back-fill)
print('------------------------')

print(df)
print('------------------------')
print(df.fillna(method='ffill', axis=1))# DataFrame替换指定行列
```

数据集连接

方法

- pd.concat()
- pd.append()

```
# numpy的合并
x = [1, 2, 3]
y = [4, 5, 6]
z = [7, 8, 9]
print(np.concatenate([x, y, z]))
print('-----------------')
x = [[1, 2],
     [3, 4]]
print(np.concatenate([x, x], axis=1))
```

pd.concat()：使用可选的设置逻辑将pandas对象沿特定轴连接起来

pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, copy=True)

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
print(pd.concat([ser1, ser2]))
print('------------------------')
print(pd.concat([ser1, ser2], axis = 1))
print('------------------------')
print(pd.concat([ser1, ser2], ignore_index=True))    #创建一个新的整数索引
print('------------------------')
print(pd.concat([ser1, ser2], keys = ['ser1', 'ser2']))    #产生分层索引
```

pd.append():直接连接

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
print(ser1.append(ser2))
```

数据集合并

方法

- pd.merge()
- pd.join()

pd.merge()：可以实现多中类型的连接：一对一，多对一和多对多合并

```
import pandas as pd
import numpy as np

class display(object):

    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

```
# 一对一
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering',
'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
df3 = pd.merge(df1, df2)
display('df1', 'df2', 'df3')
```

```
# 多对一
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

```
# 多对多
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

源代码