Python数据处理库pandas入门教程

Posted on Feb 15, 2018

AI (http://qiangbo.space/category/#AI) Python (http://qiangbo.space/tags/#Python)
MachineLearning (http://qiangbo.space/tags/#MachineLearning) pandas (http://qiangbo.space/tags/#pandas)

---

pandas是一个Python语言的软件包，在我们使用Python语言进行机器学习编程的时候，这是一个非常常用的基础编程库。本文是对它的一个入门教程。

pandas提供了快速，灵活和富有表现力的数据结构，目的是使“关系”或“标记”数据的工作既简单又直观。它旨在成为在Python中进行实际数据分析的高级构建块。

入门介绍

pandas适合于许多不同类型的数据，包括：

- 具有异构类型列的表格数据，例如SQL表格或Excel数据
- 有序和无序（不一定是固定频率）时间序列数据。
- 具有行列标签的任意矩阵数据（均匀类型或不同类型）
- 任何其他形式的观测/统计数据集。

由于这是一个Python语言的软件包，因此需要你的机器上首先需要具备Python语言的环境。关于这一点，请自行在网络上搜索获取方法。

关于如何获取pandas请参阅官网上的说明：pandas Installation (http://pandas.pydata.org/pandas-docs/stable/install.html)。

通常情况下，我们可以通过 `pip` 来执行安装：

```
sudo pip3 install pandas
```

或者通过conda (http://pandas.pydata.org/pandas-docs/stable/install.html#installing-pandas-with-anaconda) 来安装pandas：

```
conda install pandas
```

目前（2018年2月）pandas的最新版本是v0.22.0 (http://pandas.pydata.org/pandas-docs/stable/whatsnew.html#v0-22-0-december-29-2017)（发布时间：2017年12月29日）。

我已经将本文的源码和测试数据放到Github上： pandas_tutorial (https://github.com/paulQuei/pandas_tutorial)，读者可以前往获取。

另外，pandas常常和NumPy (http://numpy.org)一起使用，本文中的源码中也会用到NumPy (http://numpy.org)。

建议读者先对NumPy (http://numpy.org)有一定的熟悉再来学习pandas，我之前也写过一个NumPy的基础教程，参见这里：Python 机器学习库 NumPy 教程 (/2018-01-06/AI_NumPy_Tutorial/)

核心数据结构

pandas最核心的就是 `Series` 和 `DataFrame` 两个数据结构。

这两种类型的数据结构对比如下：

名称	维度	说明
Series	1维	带有标签的同构类型数组
DataFrame	2维	表格结构，带有标签，大小可变，且可以包含异构的数据列

DataFrame可以看做是Series的容器，即：一个DataFrame中可以包含若干个Series。

> 注：在0.20.0版本之前，还有一个三维的数据结构，名称为Panel。这也是pandas库取名的原因：**pan**el-**da**ta-**s**。但这种数据结构由于很少被使用到，因此已经被废弃了。

Series

由于Series是一维结构的数据，我们可以直接通过数组来创建这种数据，像这样：

```
# data_structure.py

import pandas as pd
import numpy as np

series1 = pd.Series([1, 2, 3, 4])
print("series1:\n{}\n".format(series1))
```

这段代码输出如下：

```
series1:
0    1
1    2
2    3
3    4
dtype: int64
```

这段输出说明如下：

- 输出的最后一行是Series中数据的类型，这里的数据都是 `int64` 类型的。
- 数据在第二列输出，第一列是数据的索引，在pandas中称之为 `Index` 。

我们可以分别打印出Series中的数据和索引：

```
# data_structure.py

print("series1.values: {}\n".format(series1.values))

print("series1.index: {}\n".format(series1.index))
```

这两行代码输出如下：

```
series1.values: [1 2 3 4]

series1.index: RangeIndex(start=0, stop=4, step=1)
```

如果不指定（像上面这样），索引是[1, N-1]的形式。不过我们也可以在创建Series的时候指定索引。索引未必一定需要是整数，可以是任何类型的数据，例如字符串。例如我们以七个字母来映射七个音符。索引的目的是可以通过它来获取对应的数据，例如下面这样：

```
# data_structure.py

series2 = pd.Series([1, 2, 3, 4, 5, 6, 7],
    index=["C", "D", "E", "F", "G", "A", "B"])
print("series2:\n{}\n".format(series2))
print("E is {}\n".format(series2["E"]))
```

这段代码输出如下：

```
series2:
C    1
D    2
E    3
F    4
G    5
A    6
B    7
dtype: int64

E is 3
```

DataFrame

下面我们来看一下DataFrame的创建。我们可以通过NumPy的接口来创建一个4x4的矩阵，以此来创建一个DataFrame，像这样：

```
# data_structure.py

df1 = pd.DataFrame(np.arange(16).reshape(4,4))
print("df1:\n{}\n".format(df1))
```

这段代码输出如下：

```
df1:
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
```

从这个输出我们可以看到，默认的索引和列名都是[0, N-1]的形式。

我们可以在创建DataFrame的时候指定列名和索引，像这样：

```
# data_structure.py

df2 = pd.DataFrame(np.arange(16).reshape(4,4),
    columns=["column1", "column2", "column3", "column4"],
    index=["a", "b", "c", "d"])
print("df2:\n{}\n".format(df2))
```

这段代码输出如下：

```
df2:
   column1  column2  column3  column4
a        0        1        2        3
b        4        5        6        7
c        8        9       10       11
d       12       13       14       15
```

我们也可以直接指定列数据来创建DataFrame：

```
# data_structure.py

df3 = pd.DataFrame({"note" : ["C", "D", "E", "F", "G", "A", "B"],
    "weekday": ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]})
print("df3:\n{}\n".format(df3))
```

这段代码输出如下：

```
df3:
  note weekday
0    C     Mon
1    D     Tue
2    E     Wed
3    F     Thu
4    G     Fri
5    A     Sat
6    B     Sun
```

请注意：

- DataFrame的不同列可以是不同的数据类型
- 如果以Series数组来创建DataFrame，每个Series将成为一行，而不是一列

例如：

```
# data_structure.py

noteSeries = pd.Series(["C", "D", "E", "F", "G", "A", "B"],
    index=[1, 2, 3, 4, 5, 6, 7])
weekdaySeries = pd.Series(["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"],
    index=[1, 2, 3, 4, 5, 6, 7])
df4 = pd.DataFrame([noteSeries, weekdaySeries])
print("df4:\n{}\n".format(df4))
```

df4的输出如下：

```
df4:
     1    2    3    4    5    6    7
0    C    D    E    F    G    A    B
1  Mon  Tue  Wed  Thu  Fri  Sat  Sun
```

我们可以通过下面的形式给DataFrame添加或者删除列数据：

```
# data_structure.py

df3["No."] = pd.Series([1, 2, 3, 4, 5, 6, 7])
print("df3:\n{}\n".format(df3))

del df3["weekday"]
print("df3:\n{}\n".format(df3))
```

这段代码输出如下：

```
df3:
  note weekday  No.
0    C     Mon    1
1    D     Tue    2
2    E     Wed    3
3    F     Thu    4
4    G     Fri    5
5    A     Sat    6
6    B     Sun    7

df3:
  note  No.
0    C    1
1    D    2
2    E    3
3    F    4
4    G    5
5    A    6
6    B    7
```

Index对象与数据访问

pandas的Index对象包含了描述轴的元数据信息。当创建Series或者DataFrame的时候，标签的数组或者序列会被转换成Index。可以通过下面的方式获取到DataFrame的列和行的Index对象：

```
# data_structure.py

print("df3.columns\n{}\n".format(df3.columns))
print("df3.index\n{}\n".format(df3.index))
```

这两行代码输出如下：

```
df3.columns
Index(['note', 'No.'], dtype='object')

df3.index
RangeIndex(start=0, stop=7, step=1)
```

请注意：

- Index并非集合，因此其中可以包含重复的数据

- Index对象的值是不可以改变，因此可以通过它安全的访问数据

DataFrame提供了下面两个操作符来访问其中的数据：

- `loc`： 通过行和列的索引来访问数据
- `iloc`： 通过行和列的下标来访问数据

例如这样：

```
# data_structure.py

print("Note C, D is:\n{}\n".format(df3.loc[[0, 1], "note"]))
print("Note C, D is:\n{}\n".format(df3.iloc[[0, 1], 0]))
```

第一行代码访问了行索引为0和1，列索引为“note”的元素。第二行代码访问了行下标为0和1（对于df3来说，行索引和行下标刚好是一样的，所以这里都是0和1，但它们却是不同的含义），列下标为0的元素。

这两行代码输出如下：

```
Note C, D is:
0    C
1    D
Name: note, dtype: object

Note C, D is:
0    C
1    D
Name: note, dtype: object
```

文件操作

pandas库提供了一系列的 `read_` 函数来读取各种格式的文件，它们如下所示：

- read_csv
- read_table
- read_fwf
- read_clipboard
- read_excel
- read_hdf
- read_html
- read_json
- read_msgpack
- read_pickle
- read_sas
- read_sql
- read_stata
- read_feather

读取Excel文件

> 注：要读取Excel文件，还需要安装另外一个库： `xlrd`

通过pip可以这样完成安装：

```
sudo pip3 install xlrd
```

安装完之后可以通过pip查看这个库的信息：

```
$  pip3 show xlrd
Name: xlrd
Version: 1.1.0
Summary: Library for developers to extract data from Microsoft Excel (tm) spreadsheet files
Home-page: http://www.python-excel.org/
Author: John Machin
Author-email: sjmachin@lexicon.net
License: BSD
Location: /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages
Requires:
```

接下来我们看一个读取Excel的简单的例子：

```
# file_operation.py

import pandas as pd
import numpy as np

df1 = pd.read_excel("data/test.xlsx")
print("df1:\n{}\n".format(df1))
```

这个Excel的内容如下：

```
df1:
   C  Mon
0  D  Tue
1  E  Wed
2  F  Thu
3  G  Fri
4  A  Sat
5  B  Sun
```

> 注：本文的代码和数据文件可以通过文章开头提到的Github仓库获取。

读取CSV文件

下面，我们再来看读取CSV文件的例子。

第一个CSV文件内容如下：

```
$ cat test1.csv
C,Mon
D,Tue
E,Wed
F,Thu
G,Fri
A,Sat
```

读取的方式也很简单：

```
# file_operation.py

df2 = pd.read_csv("data/test1.csv")
print("df2:\n{}\n".format(df2))
```

我们再来看第2个例子，这个文件的内容如下：

```
$ cat test2.csv
C|Mon
D|Tue
E|Wed
F|Thu
G|Fri
A|Sat
```

严格的来说，这并不是一个CSV文件了，因为它的数据并不是通过逗号分隔的。在这种情况下，我们可以通过指定分隔符的方式来读取这个文件，像这样：

```
# file_operation.py

df3 = pd.read_csv("data/test2.csv", sep="|")
print("df3:\n{}\n".format(df3))
```

实际上， `read_csv` 支持非常多的参数用来调整读取的参数，如下表所示：

参数	说明
path	文件路径
sep或者delimiter	字段分隔符
header	列名的行数，默认是0（第一行）
index_col	列号或名称用作结果中的行索引
names	结果的列名称列表
skiprows	从起始位置跳过的行数
na_values	代替 `NA` 的值序列
comment	以行结尾分隔注释的字符
parse_dates	尝试将数据解析为 `datetime` 。默认为 `False`
keep_date_col	如果将列连接到解析日期，保留连接的列。默认为 `False` 。
converters	列的转换器
dayfirst	当解析可以造成歧义的日期时，以内部形式存储。默认为 `False`
data_parser	用来解析日期的函数
nrows	从文件开始读取的行数
iterator	返回一个TextParser对象，用于读取部分内容
chunksize	指定读取块的大小
skip_footer	文件末尾需要忽略的行数
verbose	输出各种解析输出的信息
encoding	文件编码
squeeze	如果解析的数据只包含一列，则返回一个 `Series`
thousands	千数量的分隔符

详细的read_csv函数说明请参见这里：pandas.read_csv (http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html?highlight=read_csv#pandas.read_csv)

处理无效值

现实世界并非完美，我们读取到的数据常常会带有一些无效值。如果没有处理好这些无效值，将对程序造成很大的干扰。

对待无效值，主要有两种处理方法：直接忽略这些无效值；或者将无效值替换成有效值。

下面我先创建一个包含无效值的数据结构。然后通过 pandas.isna 函数来确认哪些值是无效的：

```
# process_na.py

import pandas as pd
import numpy as np

df = pd.DataFrame([[1.0, np.nan, 3.0, 4.0],
                   [5.0, np.nan, np.nan, 8.0],
                   [9.0, np.nan, np.nan, 12.0],
                   [13.0, np.nan, 15.0, 16.0]])

print("df:\n{}\n".format(df));
print("df:\n{}\n".format(pd.isna(df)));****
```

这段代码输出如下：

```
df:
      0   1     2     3
0   1.0 NaN   3.0   4.0
1   5.0 NaN   NaN   8.0
2   9.0 NaN   NaN  12.0
3  13.0 NaN  15.0  16.0

df:
       0     1      2      3
0  False  True  False  False
1  False  True   True  False
2  False  True   True  False
3  False  True  False  False
```

忽略无效值

我们可以通过 pandas.DataFrame.dropna 函数抛弃无效值：

```
# process_na.py

print("df.dropna():\n{}\n".format(df.dropna()));
```

> 注： dropna 默认不会改变原先的数据结构，而是返回了一个新的数据结构。如果想要直接更改数据本身，可以在调用这个函数的时候传递参数 inplace = True 。

对于原先的结构，当无效值全部被抛弃之后，将不再是一个有效的DataFrame，因此这行代码输出如下：

```
df.dropna():
Empty DataFrame
Columns: [0, 1, 2, 3]
Index: []
```

我们也可以选择抛弃整列都是无效值的那一列：

```
# process_na.py

print("df.dropna(axis=1, how='all'):\n{}\n".format(df.dropna(axis=1, how='all')));
```

> 注： `axis=1` 表示列的轴。how可以取值'any'或者'all'，默认是前者。

这行代码输出如下：

```
df.dropna(axis=1, how='all'):
      0     2     3
0   1.0   3.0   4.0
1   5.0   NaN   8.0
2   9.0   NaN  12.0
3  13.0  15.0  16.0
```

替换无效值

我们也可以通过 `fillna` 函数将无效值替换成为有效值。像这样：

```
# process_na.py

print("df.fillna(1):\n{}\n".format(df.fillna(1)));
```

这段代码输出如下：

```
df.fillna(1):
      0    1     2     3
0   1.0  1.0   3.0   4.0
1   5.0  1.0   1.0   8.0
2   9.0  1.0   1.0  12.0
3  13.0  1.0  15.0  16.0
```

将无效值全部替换成同样的数据可能意义不大，因此我们可以指定不同的数据来进行填充。为了便于操作，在填充之前，我们可以先通过 `rename` 方法修改行和列的名称：

```
# process_na.py

df.rename(index={0: 'index1', 1: 'index2', 2: 'index3', 3: 'index4'},
          columns={0: 'col1', 1: 'col2', 2: 'col3', 3: 'col4'},
          inplace=True);
df.fillna(value={'col2': 2}, inplace=True)
df.fillna(value={'col3': 7}, inplace=True)
print("df:\n{}\n".format(df));
```

这段代码输出如下：

```
df:
        col1  col2  col3  col4
index1   1.0   2.0   3.0   4.0
index2   5.0   2.0   7.0   8.0
index3   9.0   2.0   7.0  12.0
index4  13.0   2.0  15.0  16.0
```

处理字符串

数据中常常牵涉到字符串的处理，接下来我们就看看pandas对于字符串操作。

Series 的 str 字段包含了一系列的函数用来处理字符串。并且，这些函数会自动处理无效值。

下面是一些实例，在第一组数据中，我们故意设置了一些包含空格字符串：

```
# process_string.py

import pandas as pd

s1 = pd.Series([' 1', '2 ', ' 3 ', '4', '5']);
print("s1.str.rstrip():\n{}\n".format(s1.str.lstrip()))
print("s1.str.strip():\n{}\n".format(s1.str.strip()))
print("s1.str.isdigit():\n{}\n".format(s1.str.isdigit()))
```

在这个实例中我们看到了对于字符串 strip 的处理以及判断字符串本身是否是数字，这段代码输出如下：

```
s1.str.rstrip():
0     1
1    2
2    3
3     4
4     5
dtype: object

s1.str.strip():
0    1
1    2
2    3
3    4
4    5
dtype: object

s1.str.isdigit():
0    False
1    False
2    False
3     True
4     True
dtype: bool
```

下面是另外一些示例，展示了对于字符串大写，小写以及字符串长度的处理：

```
# process_string.py

s2 = pd.Series(['Stairway to Heaven', 'Eruption', 'Freebird',
                    'Comfortably Numb', 'All Along the Watchtower'])
print("s2.str.lower():\n{}\n".format(s2.str.lower()))
print("s2.str.upper():\n{}\n".format(s2.str.upper()))
print("s2.str.len():\n{}\n".format(s2.str.len()))
```

该段代码输出如下：

```
s2.str.lower():
0          stairway to heaven
1                    eruption
2                    freebird
3            comfortably numb
4    all along the watchtower
dtype: object

s2.str.upper():
0          STAIRWAY TO HEAVEN
1                    ERUPTION
2                    FREEBIRD
3            COMFORTABLY NUMB
4    ALL ALONG THE WATCHTOWER
dtype: object

s2.str.len():
0    18
1     8
2     8
3    16
4    24
dtype: int64
```

结束语

本文是pandas的入门教程，因此我们只介绍了最基本的操作。对于

- MultiIndex/Advanced Indexing
- Merge, join, concatenate
- Computational tools

之类的高级功能，以后有机会我们再来一起学习。

读者也可以根据下面的链接获取更多的知识。

参考资料与推荐读物

- pandas官方网站 (https://pandas.pydata.org)
- Python for Data Analysis (https://www.amazon.com/Python-Data-Analysis-Wrangling-IPython/dp/1491957662/)
- Pandas Tutorial: Data analysis with Python: Part 1 (https://www.dataquest.io/blog/pandas-python-tutorial/)

Contents