Pandas Basics

Rui Leite

FEP

Oct 2023

Pandas Basics

Why Pandas is Usefull - Key Features of Pandas

- quick and efficient data manipulation and analysis.
- handle a large number of data file formats and access to databases
- a rich number of indexing, slicing (e.g. label based) and subsetting operations available
- easy to merge, join, pivoting and reshaping of datasets
- have functions to handle missing values
- provides time-series functionality
- easy operations to perform grouping of data and aggregation
- work fine together with MatPlotLib and Numpy to achieve data visualization and numerical tasks

Pandas - Install and Import

Install **Pandas** is easy!

Open your terminal (shell ou cmd) and use one of the following commands:

```
$ conda install pandas
```

OR

```
$ pip install pandas
```

In jupyter notebook use the command

```
!pip install pandas
```

In spyder console use the command

```
pip install pandas
```

Import **Pandas**

To import pandas using the most used alias (short name) do

```
import pandas as pd
```

Fundamental Objects in **Pandas**

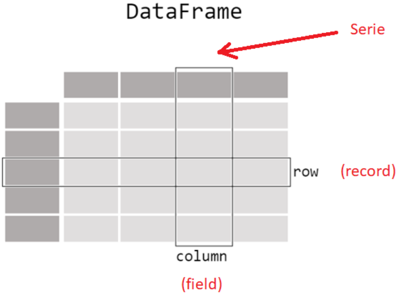
Pandas has two main objects: **Series** and **Dataframes**

Series is an indexed 1D array (one type) that support label indices.

```
In [1]: import pandas as pd

In [2]: d=pd.Series([0.3, 0.7, 1.2, 5.2])

In [3]: d
Out[3]:
0    0.3
1    0.7
2    1.2
3    5.2
dtype: float64

In [4]: d[2]
Out[4]: 1.2

In [5]: d=pd.Series([0.3, 0.7, 1.2, 5.2],index=['a','b','c','d'])

In [6]: d['a']
Out[6]: 0.3

In [7]: d[['a','c']]
Out[7]:
a    0.3
c    1.2
dtype: float64
```

Fundamental Objects in **Pandas**

Dataframes is collection of Series (with the same size) [1].
It works like a dictionary of Series and basically define data tables.
Each column is a Serie.



[1] If the series have different sizes the smaller will be shifted down adding missing values on the first rows

Fundamental Objects in **Pandas**

DataFrame object constructor

syntax: pandas.DataFrame(data, index , columns , dtype , copy)

index are the row labels (label indices)

columns are the column names

Defining Dataframes (several examples)

- using list of lists to provide the data
  `d=pd.DataFrame([[1,2],[3,4]])`
- using a dictionary where the elements are Series
  `d=pd.DataFrame({'A':SerieA,'B':SerieB})`
- using a **Numpy** 2D array
  `d=pd.DataFrame(anArray)`
- defining also **index** and **columns**
  `d=pd.DataFrame([[1,2],[3,4]],index=['first','second'], columns=['fieldA','fieldB'])`

Fundamental Objects in **Pandas**

```
In [1]: import pandas as pd
   ...: d=pd.DataFrame([[1,2],[3,4]])

In [2]: d
Out[2]:
   0  1
0  1  2
1  3  4

In [3]: # in the previous we didn't define index, and columns
   ...: SerieA=pd.Series([3,4])
   ...: SerieB=pd.Series(["p1","p2"])
   ...: d=pd.DataFrame({'A':SerieA,'B':SerieB})

In [4]: d
Out[4]:
   A   B
0  3  p1
1  4  p2

In [5]: import numpy as np
   ...: anArray=np.array([[1.7,2,3],[10,2.1,3.5]])
   ...: d=pd.DataFrame(anArray,["firstRow","secondRow"],["fld1","fld2","fld3"])

In [6]: d
Out[6]:
           fld1  fld2  fld3
firstRow    1.7   2.0   3.0
secondRow  10.0   2.1   3.5
```

Pandas Indexing (Series and Dataframes)

We can index a Serie with the same procedure that we do with lists and some additional ones that uses indices

Example:

S=pd.Series([2,5,3],["a","b","c"])
S[0] # normal
S[0:2] # slicing
S[[0,2]] # multiple elements

S["b"] # using index - identify one element
S["a":"b"] # slicing on indices - includes the last element of slice
S[["a","c"]] # multiple elements using indices

S[S>2] # indexing with logical indices - selects those where the corresponding logical expression is True

Pandas Indexing (Series and Dataframes)

We can index a dataframe using the same procedure that we do with lists and some additional ones that uses indices

Example:

d=pd.DataFrame([[2,5],[3,4],[1,2]],["a","b","c"],["f1","f2"])
d[0] # means select column 0 (first field) but only works if we didn't define column names (not here that we have "f1" and "f2")
d["f2"] # means select column with the name "f2"
d[[["f1","f2"]] # multiple elements - only for column indices
d[0:2] # slicing on the rows - here selects row 0 and row 1
d["a":"b"] # indices slicing on the rows - here selects row "a" until (including) row "b"
d["a":"b"] # indices slicing on the rows - here selects row "a" until (including) row "b"
d[d["f2"]>1.4] # indexing with logical indices - selects those rows where the corresponding logical expression is True

Pandas Indexing using iloc

We can index a dataframe using functions **iloc** and **loc**.
iloc is used for integer indexing and slicing and **loc** is used for indices indexing (the labels)

Examples with .iloc:

d=pd.DataFrame([[2,5,1],[3,4,2],[1,2,7]],["a","b","c"],["f1","f2","f3"])

d.iloc[0,2] # means select element located on row 0 and column 2
d.iloc[[0,2],1] # means select elements located on row 0 and 2 and that are on column 2
d.iloc[[1,2],-1::-1] # means select elements on rows 1 and 2 and columns presented reversed (appear according to the slice)
d.iloc[0,:]=0 # means change to zero all the elements of the row 0 and every column
d.iloc[1,:]=3*d.iloc[1,:] # means multiply by 3 all the elements of row 1

Pandas Indexing using **loc**

As mention in the last slide **loc** is used for indexing using indices [2]

Examples with iloc:

- d.loc["a",:] # selection just the row with index "a"
- d.loc[["a","c"],:] # multiselection of rows with index "a" and "b"
- d.loc["a","f2"] # selection the cell with row "a" and column "f2"
- d.loc[["a","c"],["f1","f3"]] # selection of the block of cells whose rows are "a" and "b" and columns are "f1" and "f3"
- d.loc["b":,"f3"::-1] # selection of the block of cells whose rows are given by the slice of indices that starts at "b" and the columns are identified with the slice "f3"::-1

[2] It can be defined indices (labels) that identify each observation in a **Serie** and in a **Dataframe**. The fiels of **Dataframe** are also identified by labels (indices).

Getting Information from Dataframes

method **info()** shows information about the number of rows and also the column names and types.

```
In [2]: d.info()
<class 'pandas.core.frame.DataFrame'>
Index: 3 entries, a to c
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   f1      3 non-null      int64
 1   f2      3 non-null      int64
 2   f3      3 non-null      int64
dtypes: int64(3)
memory usage: 204.0+ bytes
```

method **describe()** shows some statistics of each column (field).

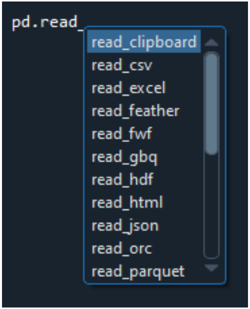
```
In [4]: d.describe()
Out[4]:
             f1    f2        f3
count  3.000000   3.0  3.000000
mean   3.333333   7.0  4.000000
std    2.516611   7.0  2.645751
min    1.000000   2.0  2.000000
25%    2.000000   3.0  2.500000
50%    3.000000   4.0  3.000000
75%    4.500000   9.5  5.000000
max    6.000000  15.0  7.000000
```

method **head()** (and **tail()**) shows the top 5 (bottom 5) rows of a **Dataframe**. We can use **head(<n>)** to see the top <n> rows.

```
In [8]: d.head(2)
Out[8]:
   f1  f2  f3
a   6  15   3
b   3   4   2
```

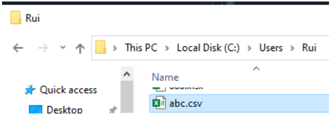
Read Dataframes - Many File Formats

There are several **Pandas** methods for reading **Dataframes**
If we use text completion in Spyder we can see many options by typing **pd.read** and press de **TAB** key.



- read_csv(filePath) read the Dataframe in **c**omma **s**eparated **v**alues. Accept a regular file path or an URL
- read_clipboard() read the Dataframe from a text segment copied to the clipboard (CTRL+C). It is interpreted as CSV text
- read_json(filePath) read the Dataframe in JSON format
- ...
- read_excel() , ..., read_sas, read_spss, read_sql, ...

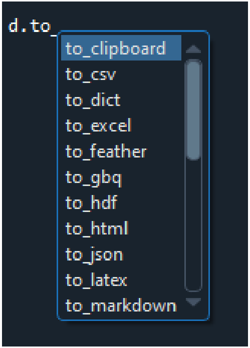
Read Dataframes - Examples




```
In [4]: d=pd.read_csv("/Users/Rui/abc.csv")

In [5]: d
Out[5]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9

In [6]: countries=pd.read_csv("https://raw.githubusercontent.com/cs109/2014_data/
master/countries.csv")

In [7]: countries
Out[7]:
       Country         Region
0      Algeria         AFRICA
1       Angola         AFRICA
2        Benin         AFRICA
3     Botswana         AFRICA
4      Burkina         AFRICA
..         ...            ...
189   Paraguay  SOUTH AMERICA
190       Peru  SOUTH AMERICA
191   Suriname  SOUTH AMERICA
192    Uruguay  SOUTH AMERICA
193  Venezuela  SOUTH AMERICA

[194 rows x 2 columns]
```

Save Dataframes - Many File Formats

Consider **d** a Dataframe. To save the data of the Dataframe **d** we can invoke its methods that starts with **to_**.
If we use text completion in Spyder we can see many options by typing **d.to_** and press de **TAB** key.



- to_csv(filePath) save the Dataframe in **c**omma **s**eparated **v**alues. Accept a regular file path or an URL
- to_clipboard() copy the dataframe to the clipboard.
- to_json(filePath) read the Dataframe in JSON format
- ...
- to_excel() , ..., to_stata, to_sql, ...

Save Dataframes - Examples

```
In [1]: import pandas as pd

In [2]: countries=pd.read_csv("https://raw.githubusercontent.com/cs109/2014_data/
master/countries.csv")

In [3]: countries
Out[3]:
       Country         Region
0      Algeria         AFRICA
1       Angola         AFRICA
2        Benin         AFRICA
3     Botswana         AFRICA
4      Burkina         AFRICA
..         ...            ...
189   Paraguay  SOUTH AMERICA
190       Peru  SOUTH AMERICA
191   Suriname  SOUTH AMERICA
192    Uruguay  SOUTH AMERICA
193  Venezuela  SOUTH AMERICA

[194 rows x 2 columns]

In [4]: countries.to_excel("/Users/Rui/countries.xlsx")

In [5]: countries.to_csv("/Users/Rui/countries.csv")
```

Operations on Dataframes - add columns and rows

Consider the Dataframes **df1** and **df2**

df1=pd.DataFrame([[3,4,5],[4,1,6],[7,3,4]],columns=["A","B","C"])

df2=pd.DataFrame([[3,5,2],[3,7,6]],columns=["A","B","C"])

- add a new column to a Dataframe

  we just need to index a column with the new column name and use it in left side of an assignment

  example: df["D"]=pd.Series([8,7,9])

  other example:df["D"]=df["A"]+df["B"] # here the new column is defined as the sum of other columns

- add 1 (multiple) new row(s) to a Dataframe

  we can add 1 row using the **loc** method and providing the index value

  df.loc[3]=[6,3,1] # the row with the values 6, 3, 1 is inserted with index '3'

  to append multiple rows we can use the **concat** method of **Pandas**

  pd.concat([df1,df2],ignore_index=True)

Operations on Dataframes - sort

Consider in the examples the Dataframes **df** and **df2**

df=pd.DataFrame([[3,4,5],[4,1,6],[7,3,4]],index=["one","two","three"], columns=["A","B","C"])

df2=pd.DataFrame([[4,6],[1,10]],columns=["B","D"])

- **sort Dataframes** - methods **sort_index** and **sort_values**
  We can sort a Dataframe by the index (row labels) or by the values of a specified column
  examples

```
In [5]: df.sort_index()
Out[5]:
       A  B  C
one    3  4  5
three  7  3  4
two    4  1  6
```

```
In [6]: df.sort_values(by="B")
Out[6]:
       A  B  C
two    4  1  6
three  7  3  4
one    3  4  5
```

Operations on Dataframes - merge

Consider in the examples the Dataframes **df** and **df2**

df=pd.DataFrame([[3,4,5],[4,1,6],[7,3,4]],index=["one","two","three"], columns=["A","B","C"])

df2=pd.DataFrame([[4,6],[1,10]],columns=["B","D"])

- **merge two Dataframes**
  examples

```
In [10]: pd.merge(df,df2,left_on="B",right_on="B")
Out[10]:
   A  B  C   D
0  3  4  5   6
1  4  1  6  10

In [11]: pd.merge(df,df2,left_on="B",right_on="B",how="left")
Out[11]:
   A  B  C     D
0  3  4  5   6.0
1  4  1  6  10.0
2  7  3  4   NaN
```

Dataframe methods

Some methods for **arrays** of **numpy** are also available in **pandas**. In the case of methods that represent summary functions the axis parameter is also available

Some Examples (explore and try other methods)

```
In [12]: df.mean()
Out[12]:
A    4.666667
B    2.666667
C    5.000000
dtype: float64

In [13]: df.mean(axis=1)
Out[13]:
one      4.000000
two      3.666667
three    4.666667
dtype: float64

In [14]: df["C"].sum()
Out[14]: 15

In [15]: df.corr()
Out[15]:
          A         B         C
A  1.000000 -0.052414 -0.720577
B -0.052414  1.000000 -0.654654
C -0.720577 -0.654654  1.000000

In [16]: df.max(axis=1)
Out[16]:
one      5
two      6
three    7
dtype: int64
```

```
In [21]: df.T  # transpose
Out[21]:
   one  two  three
A    3    4      7
B    4    1      3
C    5    6      4

In [22]: df.std() # standard deviation
Out[22]:
A    2.081666
B    1.527525
C    1.000000
dtype: float64

In [23]: df.cumsum()
Out[23]:
        A  B   C
one     3  4   5
two     7  5  11
three  14  8  15

In [24]: df.apply(func=lambda x:sum(x)/len(x))
Out[24]:
A    4.666667
B    2.666667
C    5.000000
dtype: float64
```

Numpy Methods Applied to **Pandas** Dataframes

We can use **numpy** methodselement-wise to each cell of a **pandas** Dataframe

Examples

```
In [61]: np.exp(df)
Out[61]:
             A          B           C
0    20.085537  54.598150  148.413159
1    54.598150   2.718282  403.428793
2  1096.633158  20.085537   54.598150

In [62]: np.round(np.exp(df),0)
Out[62]:
        A     B      C
0    20.0  55.0  148.0
1    55.0   3.0  403.0
2  1097.0  20.0   55.0

In [63]: np.sqrt(df)
Out[63]:
          A         B         C
0  1.732051  2.000000  2.236068
1  2.000000  1.000000  2.449490
2  2.645751  1.732051  2.000000

In [64]: np.sin(df)
Out[64]:
          A         B         C
0  0.141120 -0.756802 -0.958924
1 -0.756802  0.841471 -0.279415
2  0.656987  0.141120 -0.756802
```

Expressions with Dataframes as Operands

Examples

```
In [70]: df ** 2
Out[70]:
    A   B   C
0   9  16  25
1  16   1  36
2  49   9  16

In [71]: df+df
Out[71]:
    A  B   C
0   6  8  10
1   8  2  12
2  14  6   8

In [72]: 4*df
Out[72]:
    A   B   C
0  12  16  20
1  16   4  24
2  28  12  16

In [73]: df>4
Out[73]:
       A      B      C
0  False  False   True
1  False  False   True
2   True  False  False

In [74]: df**2<15
Out[74]:
       A      B      C
0   True  False  False
1  False   True  False
2  False   True  False
```

Dataframe methods for Data Cleaning

- **find and remove duplicated rows**
  To identify the duplicates use df.duplicated()
  To remove duplicated rows use df.drop_duplicates()
- **deal with missing values**
  The missing values can be null (value None), NaN (values non numeric) and NaT (invalid date/time data)
  - identify missing values df.isnull()
  - drop the rows (or columns, defined by axis parameter) with missing values df.dropna()
  - fill missing with a value (say 10) df.fillna(10)
  - fill missings with interpolated values df.interpolated()

Group rows of a Dataframe and apply a summary to each group

Consider that we want to organize the rows of a Dataframe by forming groups defined by the value of a field (or fields) and finally apply a summary function to each group. We can use the Dataframe method **group_by** to achieve this.

Example

```
In [104]: sales
Out[104]:
  product  qtd  value
0   prodA    3      2
1   prodB    5      3
2   prodB    2      4
3   prodC    5      2
4   prodA    7      3

In [105]: sales["total"]=sales["qtd"]*sales["value"]

In [106]: sales.groupby("product")["qtd"].sum()
Out[106]:
product
prodA    10
prodB     7
prodC     5
Name: qtd, dtype: int64

In [107]: sales.groupby("product")["qtd"].count()
Out[107]:
product
prodA    2
prodB    2
prodC    1
Name: qtd, dtype: int64

In [108]: sales.groupby("product")["total"].sum()
Out[108]:
product
prodA    27
prodB    23
prodC    10
Name: total, dtype: int64
```