Because learning changes everything.®

TITLE: Python Programming: Problem Solving, Packages and Libraries

Edition

Lecture PPT Chapter pandas: Open Source Data Analysis and Manipulation Tool

Pandas - - Learning Objectives

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- **LO 1** Understand the three basic data structures in pandas.
- **LO 2** Understand in detail the signature of the DataFrame class and its various methods.
- **LO 3** Use pandas to open csv files.
- **LO 4** Use html reading capabilities of pandas to read html files.
- **LO 5** Read/Write to JSON files.

Need for a library like Pandas

- pandas is streamlined for data representation. This is especially true for data stored in files or available on a web resource.
- pandas has a number of methods for input/output operations. It can directly load data in various formats like csv, html and json and therefore can easily do data analysis on data in various formats.
- pandas also has a number of built-in features for data filtering.
- The pandas library is basically Python's answer to the statistical analysis capabilities of the R programming language.
- If you want to pursue a course in data science or machine learning, then pandas is a very helpful library.

Various data structures in Pandas

There are basically three types of data structures in pandas: Series, DataFrame and Panel.

- Series is a one-dimensional array with a label. It can hold data of almost any type including strings, numbers and Python objects. You can visualize Series as a column of data, such as observations on a single variable.
- DataFrame is a two-dimensional (2D) data structure whose axes have labels. The two dimensions can hold data of different types. DataFrame can be thought of as similar to an Excel spread sheet. DataFrame is the most commonly used pandas object.
- Panel is a container for three-dimensional data. It can have heterogeneous data. A panel can be thought of as a container for DataFrames. A panel has three axis. Axis0 (also called items) represents the individual dataframes contained in the panel. Axis1 (also called major_axis) represents the row index of each individual dataframe and Axis2 (Also called minor_axis) represents the columns for each of the individual dataframes.

Series

The following script shows how to create two Series objects s1 and s2 on Jupyter notebook:

```
import pandas as pd
s1 = pd.Series([1, 'cat', (2,3)])
s2 = pd.Series(['Ant', 'bat', 'cat'], index = ['A', 'B', 'C'])
print('s1->', s1)
print('s2->', s2)
```

```
#Output
s1->01
1       cat
2    (2, 3)
dtype: object
s2-> A    Ant
B    bat
C    cat
dtype: object
```

DataFrame - - 1

- **DataFrame is the most widely used data structure of pandas.**
- **This is because for reading tables and csv files, one generally needs a 2D object.**
- **Further, all the spread sheets and text files are read as DataFrame, therefore it is a very important data structure of pandas.**
- **One can understand a DataFrame by looking at Table 20.1.**

TABLE 20.1 Some data about four planets

Planet	Mass (10^{24} kg)	Diameter (km)	Length of day (Hours)	Distance from Sun (10^6 km)
Mercury	.330	4879	4222.6	57.9
Venus	4.87	12,104	2802.0	108.2
Earth	5.97	12,756	24.0	149.6
Mars	0.642	6792	24.7	227.9

Now in Table 20.1, each row can be thought of as a list. For example, the data for Earth could be represented as [Earth, 5.97, 12756, 24.0, 149.6]. Or the data for Earth could also be represented as Earth → [5.97, 12756, 24.0, 149.6] where Earth is the index of the list. Further you could represent the data as five columns, i.e., [Planet, Mass, Dia, Len_day, Dist_from_sun]. But in case you want to use the first column as index of each row then you need only four columns, i.e., [Mass, Dia, Len_day, Dist_from_sun].

DataFrame - - 2

You can represent the entire table above as a list of lists. Table 20.2 shows two tables giving each of the two possible formats. In the first table, the index is automatically generated by pandas and starts from 0. In the second table, you have to explicitly provide the names of each item in the index.

TABLE 20.2 Two different ways in which index may be used in a DataFrame object. In the first table, the index is a number, while in the second table, it is the planet name

Index		List of lists
0	→	[[Mercury, 0.330, 4879, 4222.6, 57.9],
1	→	[Venus, 4.87, 12104, 2802.0, 108.2],
2	→	[Earth, 5.97, 12756, 24.0, 149.6],
3	→	[Mars, 0.642, 6792, 24.7, 227.9]]

Index		List of lists
Mercury	→	[[0.330, 4879, 4222.6, 57.9],
Venus	→	[4.87, 12104, 2802.0, 108.2],
Earth	→	[5.97, 12756, 24.0, 149.6],
Mars	→	[0.642, 6792, 24.7, 227.9]]

You may use the above data to create DataFrame. You can do it in two ways:

- **First: You may create a data frame where the name of the planet will be a part of the list and the index of each row then will be 0 to 3.**
- **Second: You may create a list where the planet name will not be part of the list but will be the index of each list.**

Signature of the DataFrame class - - 1

```
import pandas as pd
?pd.DataFrame
```

```
Init signature: pd.DataFrame(data=None, index=None, columns=None,
dtype=None, copy=False)
Docstring: Two-dimensional size-mutable, potentially heterogeneous
tabular data
structure with labeled axes (rows and columns). Arithmetic operations
align on both row and column labels. Can be thought of as a dict-like
container for Series objects. The primary pandas data structure
Parameters
(a)data : numpy ndarray (structured or homogeneous), dict, or DataFrame.
Dict can contain Series, arrays, constants, or list-like objects
(b)index : Index or array-like. Index to use for resulting frame. Will
default to np.arange(n) if no indexing information part of input data
and no index provided
(c)columns : Index or array-like. Column labels to use for resulting
frame. Will default to np.arange(n) if no column labels are provided
(d)dtype : dtype, default None. Data type to force, otherwise infer
(e)copy : boolean, default False. Copy data from inputs. Only affects
DataFrame / 2d ndarray input
```

Signature of the DataFrame class - - 2

[- - - Some notable points about the signature of the DataFrame class given on previous slide are given below]

For the present purpose, the following are relevant:

- **The parameter `data` can contain list, series or list like objects.**
- **Parameter `index` if not provided will give np.arange(n) where n is the number of items. Note: arrange(n) will generate numbers from 0 to n-1.**
- **Similarly, `columns` if not provided will be from 0 to m-1 where m is the number of columns.**
- **Notice all the parameters have default values, so a DataFrame object will be created even if you give no parameters to the method.**

Example code for creating a DataFrame object - - 1

Method 1:-

- **Here a DataFrame object is created which has a “list of lists” of 4 planets.**
- **The point to note here is that the index of each inner list is from 0 to 3**

```
import numpy as np
import pandas as pd
# my_data is a list of lists
# Each inner list represents one row
my_data = [['Mercury', 0.330, 4879, 4222.6, 57.9],
           ['Venus', 4.87, 12104, 2802.0, 108.2],
           ['Earth', 5.97, 12756, 24.0, 149.6],
           ['Mars', 0.642, 6792, 24.7, 227.9]]
df1 = pd.DataFrame(data = my_data)
print(df1)
```

```
# OUTPUT---
         0      1      2       3      4
0  Mercury  0.330   4879  4222.6   57.9
1    Venus  4.870  12104  2802.0  108.2
2    Earth  5.970  12756    24.0  149.6
3     Mars  0.642   6792    24.7  227.9
```

Example code for creating a DataFrame object - - 2

Method 2:- You can modify the above script as follows:

- **Make the names of planets as index. So remove the names of planets from the list of lists.**
- **Make a list of names of names of planets. Use this list as index to the DataFrame object.**
- **Make a list of headers for each column and use this list for columns.**
- **The modified script is as follows:**

```
import numpy as np
import pandas as pd
# my_data is a list of lists
# Each inner list represents one row
my_data = [[0.330, 4879, 4222.6, 57.9],
           [4.87, 12104, 2802.0, 108.2],
           [5.97, 12756, 24.0, 149.6],
           [0.642, 6792, 24.7, 227.9]]
# Give data for index
my_idx = ['Mercury', 'Venus', 'Earth', 'Mars']
# Give data for columns
my_col = ['Mass', 'Dia', 'Day_len', 'Dist_sun']

df1 = pd.DataFrame(data = my_data, index = my_idx, columns = my_col)
print(df1)
```

```
# OUTPUT---
          Mass    Dia  Day_len  Dist_sun
Mercury  0.330   4879   4222.6      57.9
Venus    4.870  12104   2802.0     108.2
Earth    5.970  12756     24.0     149.6
Mars     0.642   6792     24.7     227.9
```

Selecting a particular “row” or “column” of a dataFrame object

You can always select and print a particular row, i.e., index or column of a DataFrame object. The following script selects column “Mass” and row “Venus”:

```
#print column Mass
print(df1['Mass'])
# Print row Venus
print(df1.loc['Venus'])
```

```
# OUTPUT---
Column Mass->
Mercury    0.330
Venus      4.870
Earth      5.970
Mars       0.642
Name: Mass, dtype: float64
Row Venus->
Mass            4.87
Dia         12104.00
Day_len      2802.00
Dist_sun      108.20
Name: Venus, dtype: float64
```

Using Boolean conditions for items in a column of a DataFrame object

You can test for data in a particular column, row wise using Boolean selection.

For example the following script uses the condition `df1.Mass > 1` to check whether the Mass of a particular planet is greater than 1 or not.

```
# Test for a condition on a given column of the DataFrame object
print(df1.Mass > 1)
# OUTPUT---
Mercury    False
Venus       True
Earth       True
Mars       False
Name: Mass, dtype: bool
```

Using Boolean conditions for items in a row of a DataFrame object

- You can also select rows which meet a particular condition in a given column.
- Suppose you want only those rows (i.e., those planets) whose `Mass > 1`, then you can write the code as follows:

```
# Select only those rows where Mass > 1
print(df1[df1.Mass > 1])
# OUTPUT---
       Mass    Dia  Day_len  Dist_sun
Venus  4.87  12104   2802.0     108.2
Earth  5.97  12756     24.0     149.6
```

Creating a DataFrame from list or from list of lists

You can easily create a DataFrame object either from a list or from a list of lists (i.e., a nested list). An example script is as follows:

```
import pandas as pd
L1 = ['a', 'b', 'c']
L2 = [['Row1', 'a', 1], ['Row2', 'b', 2], ['Row3', 'c', 3]]
# Use list to create a 1D DataFrame object
df1 = pd.DataFrame(data = L1, index = [1, 2, 3], columns = ['Letters'])
print(df1)
# Use list of lists to create 2D DataFrame object
df2 = pd.DataFrame(data = L2, columns = ['RowNumb', 'char', 'number'])
print(df2)
```

```
# OUTPUT---
  Letters
1       a
2       b
3       c
  RowNumb char  number
0    Row1    a       1
1    Row2    b       2
2    Row3    c       3
```

Using the key: value pair of a Dictionary to create a DataFrame object

- **In the previous example, you had provided a list of lists to the parameter data to create a DataFrame object.**
- **You can also create a DataFrame by using a dictionary of equal-length list.**
- **The key of the dictionary becomes the column names and the value of the dictionary must be a list and each of the list item becomes an entry in the column.**
- **This is shown in the following code:**

```
import pandas as pd
some_data = {'Name': ['Anil', 'Babita', 'Charu', 'Dimple'],
             'Age': [20, 21, 22, 23],
             'Sex': ['M', 'F', 'F', 'F']}
my_df = pd.DataFrame(some_data)
print(my_df)
```

```
# OUTPUT---
     Name  Age Sex
0    Anil   20   M
1  Babita   21   F
2   Charu   22   F
3  Dimple   23   F
```

Adding new column to a DataFrame object

You can always add a new column to the DataFrame. This can be done by using the following format:

```
df_object['new_column_name'] = [list_of_data]
```

So you can add a new column say Marks to the previous DataFrame. The complete code is shown as follows:

```
import pandas as pd
some_data = {'Name': ['Anil', 'Babita', 'Charu', 'Dimple'],
             'Age': [20, 21, 22, 23],
             'Sex': ['M', 'F', 'F', 'F']}
my_df = pd.DataFrame(some_data)
# print(my_df)
my_df['Marks'] = [70, 75, 80, 85]
print(my_df)
```

```
# OUTPUT---
     Name  Age Sex  Marks
0    Anil   20   M     70
1  Babita   21   F     75
2   Charu   22   F     80
3  Dimple   23   F     85
```

Changing the “index” of a DataFrame object

- You can make any of the columns as index by using the `set_index([column_name])` method of the DataFrame.
- So if you add the following line and then give a print command as shown, then output is as follows:

```
my_df = my_df.set_index(['Name'])
print(my_df)
```

```
# OUTPUT---
        Age Sex  Marks
Name
Anil     20   M     70
Babita   21   F     75
Charu    22   F     80
Dimple   23   F     85
```

Panel - - 1

Panel is not a very important data structure in pandas. You can easily skip the discussion on Panel.)

The signature for `pd.Panel()` is:

```
Init signature: pd.Panel(data=None, items=None, major_axis=None,
minor_axis=None, copy=False, dtype=None)
Docstring:
 Represents wide format panel data, stored as 3-dimensional array
 Parameters
 ----------
 data : ndarray (items x major x minor), or dict of DataFrames
 items : Index or array-like axis=0
 major_axis : Index or array-like axis=1
 minor_axis : Index or array-like axis=2
 dtype : dtype, default None. Data type to force, otherwise infer
 copy : boolean, default False. Copy data from inputs. Only affects
DataFrame / 2d ndarray input
```

Panel - - 2

A panel is a 3D container of data.

A panel container in pandas consists of three axes whose details are as follows:

- **items: axis 0, each item is a DataFrame object contained inside**
- **major_axis: axis 1, it represents the index (rows) of each of the DataFrame object.**
- **minor_axis: axis 2, it represents the columns of each of the DataFrame object.**

Suppose you had a 3D data like say sale of goods of types say ['G1', 'G2', 'G3', 'G4'] in say six years from 2012 to 2017. Further suppose you have two shops say S1 and S2. (The three dimensions are: (1) Good types, (2) Years and (3) the two shops.). Then a random sample of data from each shop could be as shown in Figure 20.1.

shop1 sale data of goods G1 to G4 over years 2012 to 2017

	G1	G2	G3	G4
2012–12–31	0	2	10	0
2013–12–31	7	1	8	5
2014–12–31	2	1	4	8
2015–12–31	10	9	3	8
2016–12–31	6	2	8	1
2017–12–31	7	3	5	9

shop2 sale data of goods G1 to G4 over years 2012 to 2017

	G1	G2	G3	G4
2012–12–31	3	3	4	4
2013–12–31	7	1	3	10
2014–12–31	8	8	5	1
2015–12–31	7	1	5	6
2016–12–31	5	10	8	6
2017–12–31	3	4	3	10

FIGURE 20.1 Panel with three axes, one each for (1) Good types (2) Years and (3) the two shops

Panel with three axes, one each for (1) Good types (2) Years and (3) the two shops

The following script uses pandas to create this 3D data shown in previous table:

```python
import pandas as pd
import numpy as np
wp = pd.Panel(np.random.randint(1, 11, (2,6,4)), items=['shop1', 'shop2'],
              major_axis=pd.date_range('1/1/2012', periods=6, freq = 'A'),
              minor_axis=['G1', 'G2', 'G3', 'G4'])
print(wp['shop1'])
print(wp['shop2'])
```

Creating a Panel as a dictionary of DataFrames

You can also create a Panel as a dictionary of DataFrames as shown in the following code:

```
import pandas as pd
import numpy as np
my_data = np.random.randint(1, 11, (2,6,4))
my_items = ['shop1', 'shop2']
my_major_axis = pd.date_range('1/1/2012', periods=6, freq = 'A')
my_minor_axis = ['G1', 'G2', 'G3', 'G4']
wp = pd.Panel(data = my_data, items = my_items,
              major_axis = my_major_axis,
              minor_axis = my_minor_axis)

print(wp['shop1'])  # You can get data for shop2 also
```

```
# OUTPUT---
            G1  G2  G3  G4
2012-12-31   3  10  10   6
2013-12-31   6   2   2   8
2014-12-31   4   7   5   3
2015-12-31   5   7   5  10
2016-12-31   1   3   2   8
2017-12-31   8   6   2   9
```

Using pandas to open csv files
read_csv() method - - 1

The pandas library has a method read_csv(). This method can accept a number of parameters. The signature of the method is as follows:

```
Signature: pd.read_csv(filepath_or_buffer, sep=',', delimiter=None,
header='infer', names=None, index_col=None, usecols=None, squeeze=False,
prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None,
true_values=None, false_values=None, skipinitialspace=False, skiprows=None,
nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False,
skip_blank_lines=True, parse_dates=False, infer_datetime_format=False,
keep_date_col=False, date_parser=None, dayfirst=False, iterator=False,
chunksize=None, compression='infer', thousands=None, decimal=b'.',
lineterminator=None, quotechar='"', quoting=0, escapechar=None, comment=None,
encoding=None, dialect=None, tupleize_cols=False, error_bad_lines=True,
warn_bad_lines=True, skipfooter=0, skip_footer=0, doublequote=True,
delim_whitespace=False, as_recarray=False, compact_ints=False,
use_unsigned=False, low_memory=True, buffer_lines=None, memory_map=False,
float_precision=None)
Docstring: Read CSV (comma-separated) file into DataFrame
Returns:- DataFrame or TextParser
```

read_csv() method - - 2

There are too many parameters to this method `read_csv()`. Only the important ones are discussed here. They are as follows:

- `filepath_or_buffer`: You need to provide a file name here.
- `sep`: This stands for the seperator. The default is comma, i.e., “,”.
- `header`: The default behaviour is to infer Row number(s) to use as the column names. However, you may specify the row number (as an integer) to be used as header.
- `index_col`: Here you can specify the column to be used as row label of the DataFrame.
- `nrows`: Here you can specify (as an integer), the number of rows to read (useful for reading few lines of a very large file).
- `Encoding`: Encoding to use for UTF when reading/writing

Using read_csv() method to read a file from disk - - Example

You may first create a csv file (you may create your own csv file). A Notepad++ is used here. The data is as follows:

```
Letter, Fruit, Animal, Place
A, Apple, Ant, Amritsar
B, Berry, Bat, Bangalore
C, Cherry, Cat, California
```

The name of the file is letters.csv and the absolute path is:
“`C:\Users\DS\Documents\planets.csv`”
The code on Jupyter is as follows:

```
import pandas as pd
myData = pd.read_csv("C:\Temp\letters.csv", encoding = "ISO-8859-1")
print(myData)
```

It has a very large number of parameters out of which only the first, i.e., filepath has to be compulsorily provided.
The two common encoding schemes are:

- **utf-8(For encoding in Unicode.)**
- **latin1/ *ISO-8859-1*=(ISO/IEC 8859 is divided into several parts. The most widely used part is latin-1 and it covers most western European Languages.)**

Using read_csv() on Jupyter

The script along with its output on Jupyter is shown in Figure 20.3.

```
In [2]:  import pandas as pd
         myData = pd.read_csv("C:\Temp\letters.csv", encoding = "latin1")
         print(myData)

           Letter    Fruit  Animal       Place
         0      A    Apple     Ant    Amritsar
         1      B    Berry     Bat    Banglore
         2      C   Cherry     Cat  California
```

FIGURE 20.3 Screenshot of output of previous script on Jupyter

Using pandas to read html files

You can use the read_html() method of pandas to get a list of dataframes. Consider the following code:

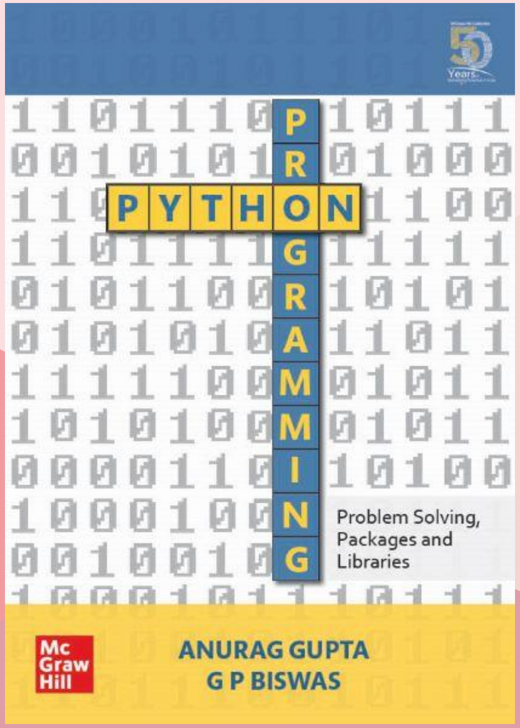
```
1  import pandas as pd
2  
3  def getData(url):
4      try:
5          dat = pd.read_html(url)
6          print(dat)
7          print(type(dat))
8      except:
9          print("Something went wrong")
10 #Valid url
11 u1 = 'https://nssdc.gsfc.nasa.gov/planetary/factsheet/'
12 #Invalid url
13 u2 = 'https://xxxxyyyyzzzz'
14 getData(u1)
15 getData(u2)
```

Line 3–9: You have created a function getData(url). Inside the function you can check for possible error. Whenever you are scraping data from the Internet, it is a good practice to handle possible errors because a number of things can go wrong on the Internet.
Line 11: u1 is a valid url which gives planets data.
Line 13: This is an invalid url and will therefore throw an exception which will be caught in line 8-9.

Reading/Writing to JSON files

In the following script, a JSON object from a DataFrame object is created. For this you may use a to_json('path') method, specifying the path to the file where the json object is to be saved.

```
import pandas as pd
import json
# Create a pandas dataframe from the planet data used earlier.
df = pd.DataFrame([[0.330, 4879, 4222.6, 57.9],
                   [4.87, 12104, 2802.0, 108.2],
                   [5.97, 12756, 24.0, 149.6],
                   [0.642, 6792, 24.7, 227.9]],
                  index = ['Mercury', 'Venus', 'Earth', 'Mars'],
                   columns = ['Mass', 'Dia', 'Day_len', 'Dist_sun'])
#Save the dataframe as a JSON object to file testJ.json
df.to_json(r'C:\Temp\testJ.json')
# You can open the testJ.json file and load data to a JSON object called
jData
with open(r'C:\Temp\testJ.json', 'r') as jFile:
    jData = json.load(jFile)
print(jData)
print('Convert json data into key:value pair')
for key, val in jData.items():
    print(str(key) + ':' + str(val))
```


PYTHON
PROGRAMMING
Problem Solving, Packages and Libraries
Mc Graw Hill
ANURAG GUPTA
G P BISWAS

Because learning changes everything.®

Thank You!

For any queries or feedback contact us at:

support.india@mheducation.com

1800-103-5875

www.mheducation.co.in