Introduction to Pandas

Laszlo Tetenyi

New York University

March 25, 2016

Why use Pandas?

Loading and manipulating (large) datasets

- One can avoid any data manipulation by hand (or VBA) while maintaining easy visual handling (unlike R) - superior to Excel even as a standalone and much faster
- Many file types are supported (like csv, Stata or SAS)
- Different coding (utf8, latin1...) or even simply bad characters are easily translated or repaired
- Intelligent label-based slicing, fancy indexing, and subsetting of large data sets
- Flexible reshaping, stacking and pivoting of data sets
- Intuitive merging and joining data sets
- Time series-specific functionality

An application

To demonstrate the main features of Pandas we will load the dataset from the Survey on Consumer Finance (SCF 2013)

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import requests, zipfile, io
# So that we can download and unzip files

r = requests.get('... /scf/files/scfp2013excel.zip')
z = zipfile.ZipFile(io.BytesIO(r.content))
f = z.open('SCFP2013.xlsx')
table = pd.read_excel(f, sheetname='SCFP2013')
```

Good visualization

- To have a quick look at table use

```
table.head()
```

- or access the relevant rows using standard Python array slicing notation

```
table[0:5]
```

```
     YY1  Y1         WGT        HHSEX   AGE    AGECL   EDUC
0      1  11  3100.802441          1    54        3     11
1      1  12  3090.352195          1    54        3     11
2      1  13  3094.100275          1    54        3     11
3      1  14  3098.507516          1    54        3     11
4      1  15  3104.670102          1    54        3     11
[5 rows x 324 columns]
```

Operations

- table is a DataFrame - a multi-dimensional equivalent of Series (another Pandas object). As it has multiple columns, you can think of it as a matrix, where columns can be accessed by their 'names'. In fact, many operations can be performed on them (coming from numpy):

  ```
  table.max().max()
  ```

  ```
  Out : 1324540600.0
  ```

- But they know more than that - eg.: they have several built-in statistics (coming from statsmodel) :

  ```
  table.describe()
  ```

  returning averages etc.

Accessing and searching variables

- Try to access normalized income and net-worth variables - finding them could be a pain

```
table.dtypes.shape
```

```
Out : (324,)
```

- But this easy thanks to Pandas

```
[col for col in table.columns if 'NETWOR'  in col]
```

```
Out : ['NETWORTH']
```

- Of course you could access columns by their index, but variable names are sometimes more convenient

```
net_worth = table['NETWORTH']
```

Create a sub - dataframe

- Create a dataset only containing the variables of interest (income, net worth and their id-s)

```
keep = ['YY1','Y1', 'NORMINC', 'NETWORTH']
data = table[keep]
data.head()
```

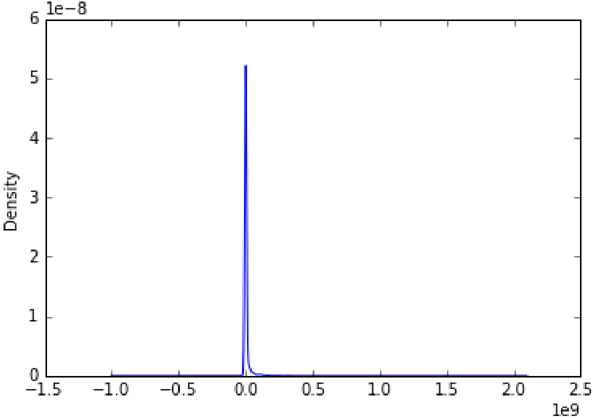
```
Out :      YY1  Y1       NORMINC  NETWORTH
      0      1  11  37537.663108      -400
      1      1  12  39566.725979      -400
      2      1  13  35508.600237      -400
      3      1  14  38552.194543      -400
      4      1  15  35508.600237      -400
```

- Rename the variable names

```
data.columns ='Household', 'Observation' , 'Income',
 \ ' Net Worth'
```

Plotting

- Plot the kernel density estimate:
```
data['Net Worth'].plot(kind='density')
plt.show()
```



Boolean operations

- Eliminate observations with net worth smaller than 1 million $:

```
data_trimmed = data[data['Net Worth'] > -1000000]
```

- The number observations decreased

```
data.shape[0] - data_trimmed.shape[0]
```

```
Out : 15
```

Some vector operations

- Suppose we want to substitute the built-in id provided by Pandas by the "Household" column. As a first step, create a non-unique identifier by transforming "Observations". This works but if you can, avoid it:

```
obs = data.loc[:,'Observation'] -
10 * data.loc[:,'Household']
```

```
data.loc[:,'Observation'] = obs
```

- Instead, use assign:

```
data = data.assign(Observations = (data['Observation'] -
10.0 * data['Household']).astype(int))
del data['Observation'] # delete the old column
data = data.rename(columns = {'Observations':'Observation'})
data = data[['Household', 'Observation' , 'Income'
, 'Net Worth']] # reinsert the column
```

Pivoting

- Now replace the id with the Household name - pivot

  ```
  p = data.pivot(index = 'Household' , columns
  = 'Observation' , values = 'Income' )
  ```

- now our p dataframe looks like this

  ```
  Observation \            1            2            3
  Household
  1              37537.663108  39566.725979  35508.600237
  2              22319.691578  22319.691578  22319.691578
  3              52755.634638  52755.634638  52755.634638
  4             125801.897980 125801.897980 125801.897980
  5              99424.080664 107540.332150 108554.863580
  ```

Stacking

- Use stacking to transform the data into a panel structure we are familiar with (and unstacking to go back to cross-section):

```
panel_data = p.stack()
```

```
Out :                       Income        Net Worth
Household Observation
1         1            37537.663108          -400
          2            39566.725979          -400
          3            35508.600237          -400
          4            38552.194543          -400
          5            35508.600237          -400
```

- Pandas has its own panel structure but it is neglected - very few functions are available

Conclusion

Advantages:

- Fast way to manipulate almost any data
- Easier to maintain the integrity of a project
- Very straightforward to learn

Disadvantages:

- Only makes sense to use if the rest of the codes is in Python (although exporting is easy)
- There is only one useful object - dataframe. Possibly not everything would fit in there
- It is so efficient that you can easily kill your computer