Python programming — Pandas

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

October 5, 2013

Overview

Pandas?

Reading data

Summary statistics

Indexing

Merging, joining

Group-by and cross-tabulation

Statistical modeling

Pandas?

"Python Data Analysis Library"

Young library for data analysis

Developed from http://pandas.pydata.org/

Main author Wes McKinney has written a 2012 book (McKinney, 2012).

Why Pandas?

A better Numpy: keep track of variable names, better indexing, easier linear modeling.

A better R: Access to more general programming language.

Why not pandas?

R: Still primary language for statisticians, means most avanced tools are there.

NaN/NA (Not a number/Not available)

Support to third-party algorithms compared to Numpy? Numexpr? (NumExpr in 0.11)

Get some data from R

Get a standard dataset, *Pima*, from R:

```
$ R
> library(MASS)
> write.csv(Pima.te, "pima.csv")
```

`pima.csv` now contains comma-separated values:

```
"","npreg","glu","bp","skin","bmi","ped","age","type"
"1",6,148,72,35,33.6,0.627,50,"Yes"
"2",1,85,66,29,26.6,0.351,31,"No"
"3",1,89,66,23,28.1,0.167,21,"No"
"4",3,78,50,32,31,0.248,26,"Yes"
"5",2,197,70,45,30.5,0.158,53,"Yes"
"6",5,166,72,19,25.8,0.587,51,"Yes"
```

Read data with Pandas

Back in Python:

```
>>> import pandas as pd
>>> pima = pd.read_csv("pima.csv")
```

"`pima`" is now what Pandas call a *DataFrame* object. This object keeps track of both data (numerical as well as text), and column and row headers.

Lets use the first columns and the index column:

```
>>> import pandas as pd
>>> pima = pd.read_csv("pima.csv", index_col=0)
```

Summary statistics

```
>>> pima.describe()
       Unnamed: 0       npreg         glu          bp        skin         bmi  \
count  332.000000  332.000000  332.000000  332.000000  332.000000  332.000000
mean   166.500000    3.484940  119.259036   71.653614   29.162651   33.239759
std     95.984374    3.283634   30.501138   12.799307    9.748068    7.282901
min      1.000000    0.000000   65.000000   24.000000    7.000000   19.400000
25%     83.750000    1.000000   96.000000   64.000000   22.000000   28.175000
50%    166.500000    2.000000  112.000000   72.000000   29.000000   32.900000
75%    249.250000    5.000000  136.250000   80.000000   36.000000   37.200000
max    332.000000   17.000000  197.000000  110.000000   63.000000   67.100000

              ped         age
count  332.000000  332.000000
mean     0.528389   31.316265
std      0.363278   10.636225
min      0.085000   21.000000
25%      0.266000   23.000000
50%      0.440000   27.000000
75%      0.679250   37.000000
max      2.420000   81.000000
```

. . . Summary statistics

Other summary statistics (McKinney, 2012, around page 101):

pima.count() Count the number of rows

pima.mean(), pima.median(), pima.quantile()

pima.std(), pima.var()

pima.min(), pima.max()

Operation across columns instead, e.g., with the mean method:

```
pima.mean(axis=1)
```

Indexing the rows

For example, you can see the first two rows or the three last rows:

```
>>> pima[0:2]
   npreg  glu  bp  skin   bmi    ped  age type
1      6  148  72    35  33.6  0.627   50  Yes
2      1   85  66    29  26.6  0.351   31   No
>>> pima[-3:]
     npreg  glu  bp  skin   bmi    ped  age type
330     10  101  76    48  32.9  0.171   63   No
331      5  121  72    23  26.2  0.245   30   No
332      1   93  70    31  30.4  0.315   23   No
```

Notice that this is not an ordinary numerical matrix: We also got text (in the “type” column) within the “matrix”!

Indexing the columns

See a specific column, here 'bmi' (body-mass index):

```
>>> pima["bmi"]
1      33.6
2      26.6
3      28.1
4      31.0
[here I cut out several lines]
330     32.9
331     26.2
332     30.4
Name: bmi, Length: 332
```

The returned type is another of Pandas *Series* object, — another of the fundamental objects in the library:

```
>>> type(pima["bmi"])
<class 'pandas.core.series.Series'>
```

Conditional indexing

Get the fat people (those with BMI above 30):

```
>>> pima.shape
(332, 9)
>>> pima[pima["bmi"]>30].shape
(210, 9)
```

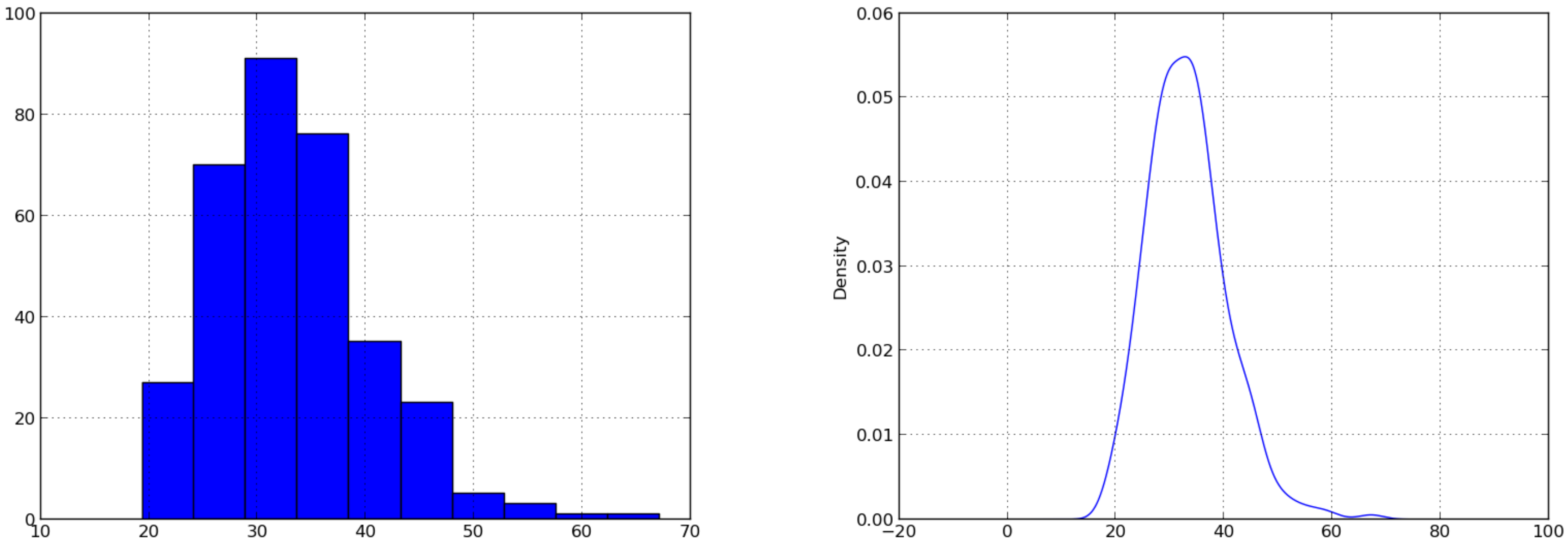
See histogram (with `from pylab import *`):

```
>>> pima["bmi"].hist()
>>> show()
```

Or kernel density estimation plot (McKinney, 2012, p 239)

```
>>> pima["bmi"].plot(kind="kde")
>>> show()
```

Plots



Histogram and kernel density estimate (KDE) of the “bmi” variable (body mass index) of the Pima data set.

Row and column conditional indexing

Example by David Marx in R:

```
A <- runif(10)
B <- runif(10)
C <- runif(10)
D <- runif(10)
E <- runif(10)

df <- data.frame(A,B,C,D,E)
sliced_df <- df[ , df[1,]<.5 ]
```

That is, select the columns in a dataframe where the values of the first row is below 0.5. Here with a 10-by-5 dataset with uniformly-distributed random numbers and columns indexed by letters.

. . . Row and column conditional indexing

Equivalent in Python

```
import pandas as pd
from pylab import *
df = pd.DataFrame(rand(10,5), columns=["A", "B", "C", "D", "E"])
df.ix[:, df.ix[0, :]<0.5]
```

These variations do not work

```
df[:, df[0]<0.5]
df[:, df[:1]<0.5]
df.ix[:, df[:1]<0.5]
```

Constructing a DataFrame

Constructing a DataFrame from a dictionary where the keys become the column names

```
>>> import pandas as pd
>>> import string

>>> spam_corpus = map(string.split, [ "buy viagra", "buy antibody" ])
>>> unique_words = set([ word for doc in spam_corpus for word in doc ])
>>> word_counts = [ (word, map(lambda doc: doc.count(word), spam_corpus))
                    for word in unique_words ]
>>> spam_bag_of_words = pd.DataFrame(dict(word_counts))
>>> print(spam_bag_of_words)
   antibody  buy  viagra
0         0    1       1
1         1    1       0
```

Concatenation

Another corpus and then concatenation with the previous dataset

```
>>> other_corpus = map(string.split, [ "buy time", "hello" ])
>>> unique_words = set([ word for doc in other_corpus for word in doc ])
>>> word_counts = [ (word, map(lambda doc: doc.count(word), other_corpus))
                    for word in unique_words ]
>>> other_bag_of_words = pd.DataFrame(dict(word_counts))
>>> print(other_bag_of_words)
   buy  hello  time
0    1      0     1
1    0      1     0

>>> pd.concat([spam_bag_of_words, other_bag_of_words], ignore_index=True)
   antibody  buy  hello  time  viagra
0         0    1    NaN   NaN       1
1         1    1    NaN   NaN       0
2       NaN    1      0     1     NaN
3       NaN    0      1     0     NaN
```

Filling in missing data

(McKinney, 2012, page 145+)

```
>>> pd.concat([spam_bag_of_words, other_bag_of_words], ignore_index=True)
   antibody  buy  hello  time  viagra
0         0    1    NaN   NaN       1
1         1    1    NaN   NaN       0
2       NaN    1      0     1     NaN
3       NaN    0      1     0     NaN

>>> pd.concat([spam_bag_of_words, other_bag_of_words], ignore_index=True).fillna(0)
   antibody  buy  hello  time  viagra
0         0    1      0     0       1
1         1    1      0     0       0
2         0    1      0     1       0
3         0    0      1     0       0
```

Combining datasets

See http://pandas.pydata.org/pandas-docs/dev/merging.html for other Pandas operations:

`concat`

`join`

`merge`

`combine_first`

Join example

Two data sets with partially overlapping rows (as not all students answer each questionnaire) where the columns should be concatenated (i.e., scores for individual questionnaires)

```
import pandas as pd

xl = pd.ExcelFile("E13_1_Resultater-2013-10-02.xlsx")
df1 = xl.parse("Resultater", index_col=[0, 1, 2], header=3)
df1.columns = map(lambda colname: unicode(colname) + "_1", df1.columns)

xl = pd.ExcelFile("E13_2_Resultater-2013-10-02.xlsx")
df2 = xl.parse("Resultater", index_col=[0, 1, 2], header=3)
df2.columns = map(lambda colname: unicode(colname) + "_2", df2.columns)

df = pd.DataFrame().join([df1, df2], how="outer")
df[["Score_1", "Score_2"]].corr()                     # Score correlation
```

Processing after join

```
>>> df.ix[:5,["Score_1", "Score_2"]]
                                              Score_1   Score_2
Bruger   Fornavn           Efternavn
(faan)   Finn Årup         Nielsen           1.000000  1.000000
s06...   ...                                 0.409467       NaN
s07..    ...                                      NaN  0.870900
s07..    ...                                 0.576568  0.741800
s07..    ...                                 0.686347  0.569666
```

(edited)

Note that the second user ("s06...") did not solve the second assignment. The joining operation by default adds a NaN to the missing element, — indicating a missing value (not available, NA).

The Groupby

Groupby method (McKinney, 2012, chapter 9): splits the dataset based on a key, e.g., a DataFrame column name.

Think of SQL's `GROUP BY`.

Example with Pima Indian data set splitting on the 'type' column (elements are "yes" and "no") and taking the mean in each of the two groups:

```
>>> pima.groupby("type").mean()
         npreg         glu         bp       skin        bmi       ped        age
type
No    2.932735  108.188341  70.130045  27.340807  31.639910  0.464565  29.215247
Yes   4.614679  141.908257  74.770642  32.889908  36.512844  0.658963  35.614679
```

The returned object from `groupby` is a *DataFrameGroupBy* object while the `mean` method on that object/class returns a *DataFrame* object

. . . The Groupby

More elaborate with two aggregating methods:

```
>>> grouped_by_type = pima.groupby("type")
>>> grouped_by_type.agg([np.mean, np.std])
         npreg                glu                  bp                \
          mean       std      mean        std      mean        std
type
No    2.932735  2.781852  108.188341  22.645932  70.130045  12.381916
Yes   4.614679  3.901349  141.908257  32.035727  74.770642  13.128026

           skin                 bmi                ped                 age  \
           mean       std      mean       std      mean       std      mean
type
No    27.340807  9.567705  31.639910  6.648015  0.464565  0.315157  29.215247
Yes   32.889908  9.065951  36.512844  7.457548  0.658963  0.417949  35.614679


            std
type
No    10.131493
Yes   10.390441
```

...The Groupby

Without groupby checking mean (32.889908) and std (9.065951) for 'skin'='Yes':

```
>>> np.mean(pima[pima["type"]=="Yes"]["skin"])
32.88990825688737                                  # Correct

>>> np.std(pima[pima["type"]=="Yes"]["skin"])
9.024268451930089 1                                 # ???

>>> import scipy.stats
>>> scipy.stats.nanstd(pima[pima["type"]=="Yes"]["skin"])
9.065951207005341                                  # Ok

>>> np.std(pima[pima["type"]=="Yes"]["skin"], ddof=1)
9.065951207005341                                  # Degrees of freedom!
```

Numpy's std is the biased estimate while Pandas std is the unbiased estimate.

Cross-tabulation

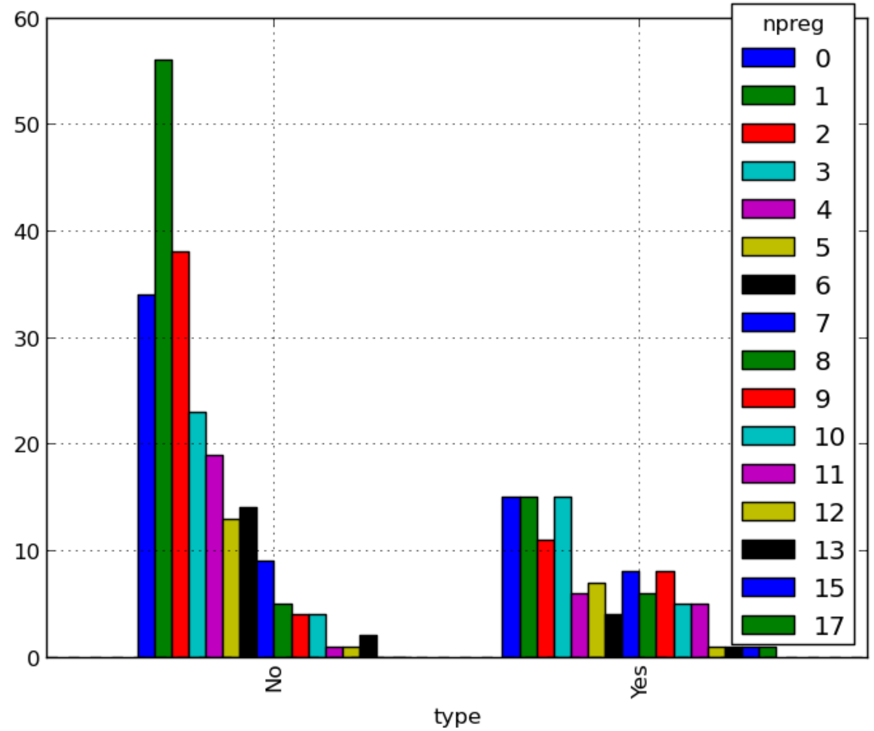
For categorical variables select two columns and generate a matrix with counts for occurences (McKinney, 2012, p. 277)

```
>>> pd.crosstab(pima.type, pima.npreg)
npreg  0   1   2   3   4   5   6   7   8   9   10  11  12  13  15  17
type
No     34  56  38  23  19  13  14   9   5   4   4   1   1   2   0   0
Yes    15  15  11  15   6   7   4   8   6   8   5   5   1   1   1   1
```

Remember:

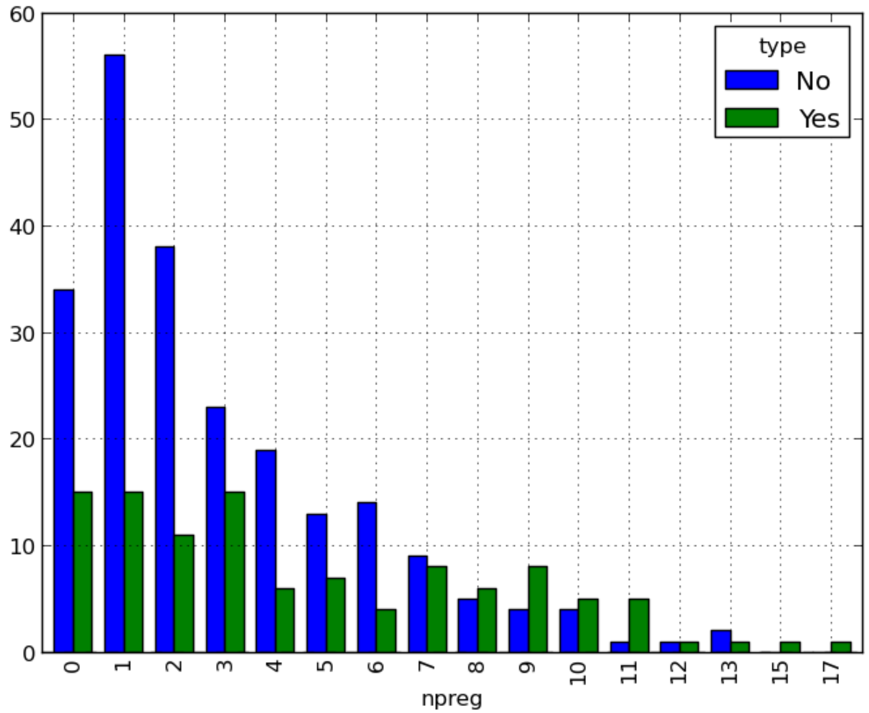
```
>>> pima[1:4]
   npreg  glu  bp  skin   bmi    ped  age type
2      1   85  66    29  26.6  0.351   31   No
3      1   89  66    23  28.1  0.167   21   No
4      3   78  50    32  31.0  0.248   26  Yes
```

Cross-tabulation plot



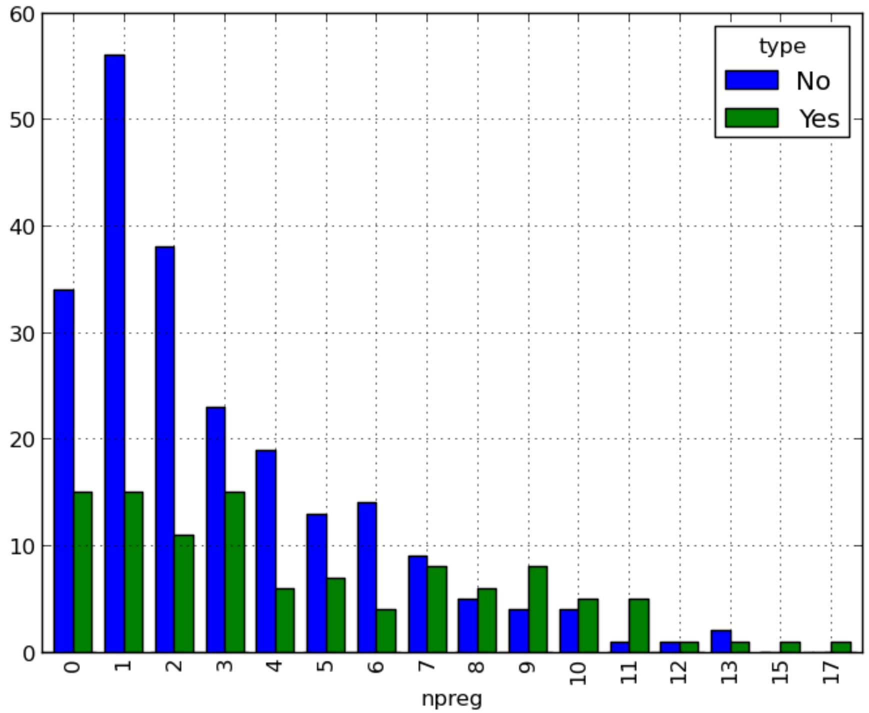

```
# Wrong ordering
pd.crosstab(pima.type, pima.npreg).plot(kind="bar")
```

Cross-tabulation plot




```
# Transpose
pd.crosstab(pima.type, pima.npreg).T.plot(kind="bar")
```

Cross-tabulation plot




```
# Or better:
pd.crosstab(pima.npreg, pima.type).plot(kind="bar")
```

Other Pandas capabilities

Hierarchical indexing (McKinney, 2012, page 147+)

Missing data support (McKinney, 2012, page 142+)

Pivoting (McKinney, 2012, chapter 9)

Time series (McKinney, 2012, chapter 10)

Statistical modeling with `statsmodels`

Example with Longley dataset.

Ordinary least squares fitting a dependent variable "TOTEMP" (Total Employment) from 6 independent variables:

```
import statsmodels.api as sm

# For 'load_pandas' you need a recent statsmodels
data = sm.datasets.longley.load_pandas()

# Endogeneous (response/dependent) & exogeneous variables (design matrix)
y, x = data.endog, data.exog

result = sm.OLS(y, x).fit() # OLS: ordinary least squares
result.summary()            # Print summary
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 TOTEMP   R-squared:                       0.988
Model:                            OLS   Adj. R-squared:                  0.982
Method:                 Least Squares   F-statistic:                     161.9
Date:                Mon, 17 Jun 2013   Prob (F-statistic):           3.13e-09
Time:                        13:56:35   Log-Likelihood:                -117.56
No. Observations:                  16   AIC:                             247.1
Df Residuals:                      10   BIC:                             251.8
Df Model:                           5
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
GNPDEFL      -52.9936    129.545     -0.409      0.691      -341.638   235.650
GNP            0.0711      0.030      2.356      0.040         0.004     0.138
UNEMP         -0.4235      0.418     -1.014      0.335        -1.354     0.507
ARMED         -0.5726      0.279     -2.052      0.067        -1.194     0.049
POP           -0.4142      0.321     -1.289      0.226        -1.130     0.302
YEAR          48.4179     17.689      2.737      0.021         9.003    87.832
==============================================================================
Omnibus:                        1.443   Durbin-Watson:                   1.277
Prob(Omnibus):                  0.486   Jarque-Bera (JB):                0.605
Skew:                           0.476   Prob(JB):                        0.739
Kurtosis:                       3.031   Cond. No.                     4.56e+05
==============================================================================
```

Statsmodels > 0.5

"Minimal example" from statsmodels documentation:

```
import numpy as np
import pandas as pd
import statsmodels.formula.api as smf

url = "http://vincentarelbundock.github.io/Rdatasets/csv/HistData/Guerry.csv"
dat = pd.read_csv(url)
results = smf.ols("Lottery ~ Literacy + np.log(Pop1831)", data=dat).fit()
results.summary()
```

Note: 1) Loading of data with URL, 2) `import statsmodels.formula.api` (possible in `statsmodels` > `0.5`), 3) R-like specification of linear model formula (from patsy).

More information

http://pandas.pydata.org/

The canonical book “Python for data analysis” (McKinney, 2012).

Will it Python?: Porting R projects to Python, exemplified though scripts from *Machine Learning for Hackers* (MLFH) by Drew Conway and John Miles White.

Summary

Pandas helps you represent your data (both numerical and categorical) and helps you keep track of what they refer to (by column and row name).

Pandas makes indexing easy.

Pandas has some basic statistics and plotting facilities.

Pandas may work more or less seamlessly with standard statistical models (e.g., general linear model with OLS-estimation)

Watch out: Pandas is still below version 1 numbering!

Standard packaging not up to date: Newest version of Pandas is 0.11.0, while, e.g., Ubuntu LTS 12.04 is 0.7.0: `sudo pip install --upgrade pandas`

Latest pip-version of statsmodels is 0.4.3, development version is 0.5 with `statsmodels.formula.api` that yields more R-like linear modeling.

References

McKinney, W. (2012). *Python for Data Analysis*. O'Reilly, Sebastopol, California, first edition. ISBN 9781449319793.