Data Structures for Statistical Computing in Python

Wes McKinney

Abstract—In this paper we are concerned with the practical issues of working with data sets common to finance, statistics, and other related fields. **pandas** is a new library which aims to facilitate working with these data sets and to provide a set of fundamental building blocks for implementing statistical models. We will discuss specific design issues encountered in the course of developing **pandas** with relevant examples and some comparisons with the R language. We conclude by discussing possible future directions for statistical computing and data analysis using Python.

Index Terms—data structure, statistics, R

Introduction

Python is being used increasingly in scientific applications traditionally dominated by [R], [MATLAB], [Stata], [SAS], other commercial or open-source research environments. The maturity and stability of the fundamental numerical libraries ([NumPy], [SciPy], and others), quality of documentation, and availability of "kitchen-sink" distributions ([EPD], [Pythonxy]) have gone a long way toward making Python accessible and convenient for a broad audience. Additionally [matplotlib] integrated with [IPython] provides an interactive research and development environment with data visualization suitable for most users. However, adoption of Python for applied statistical modeling has been relatively slow compared with other areas of computational science.

A major issue for would-be statistical Python programmers in the past has been the lack of libraries implementing standard models and a cohesive framework for specifying models. However, in recent years there have been significant new developments in econometrics ([StaM]), Bayesian statistics ([PyMC]), and machine learning ([SciL]), among others fields. However, it is still difficult for many statisticians to choose Python over R given the domain-specific nature of the R language and breadth of well-vetted open-source libraries available to R users ([CRAN]). In spite of this obstacle, we believe that the Python language and the libraries and tools currently available can be leveraged to make Python a superior environment for data analysis and statistical computing.

In this paper we are concerned with data structures and tools for working with data sets *in-memory*, as these are fundamental building blocks for constructing statistical models. **pandas** is a new Python library of data structures and statistical tools initially developed for quantitative finance applications. Most of our examples here stem from time series and cross-sectional data arising in financial modeling. The package's name derives from *panel data*, which is a term for 3-dimensional data sets encountered in statistics and econometrics. We hope that **pandas** will help make scientific Python a more attractive and practical statistical computing environment for academic and industry practitioners alike.

Wes McKinney is with AQR Capital Management, LLC. E-mail: wesmckinn@gmail.com.

Statistical data sets

Statistical data sets commonly arrive in tabular format, i.e. as a two-dimensional list of *observations* and names for the fields of each observation. Usually an observation can be uniquely identified by one or more values or *labels*. We show an example data set for a pair of stocks over the course of several days. The NumPy `ndarray` with structured dtype can be used to hold this data:

```
>>> data
array([('GOOG', '2009-12-28', 622.87, 1697900.0),
       ('GOOG', '2009-12-29', 619.40, 1424800.0),
       ('GOOG', '2009-12-30', 622.73, 1465600.0),
       ('GOOG', '2009-12-31', 619.98, 1219800.0),
       ('AAPL', '2009-12-28', 211.61, 23003100.0),
       ('AAPL', '2009-12-29', 209.10, 15868400.0),
       ('AAPL', '2009-12-30', 211.64, 14696800.0),
       ('AAPL', '2009-12-31', 210.73, 12571000.0)],
      dtype=[('item', '|S4'), ('date', '|S10'),
             ('price', '<f8'), ('volume', '<f8')])

>>> data['price']
array([622.87, 619.4, 622.73, 619.98, 211.61, 209.1,
       211.64, 210.73])
```

Structured (or record) arrays such as this can be effective in many applications, but in our experience they do not provide the same level of flexibility and ease of use as other statistical environments. One major issue is that they do not integrate well with the rest of NumPy, which is mainly intended for working with arrays of homogeneous dtype.

R provides the `data.frame` class which can similarly store mixed-type data. The core R language and its 3rd-party libraries were built with the `data.frame` object in mind, so most operations on such a data set are very natural. A `data.frame` is also flexible in size, an important feature when assembling a collection of data. The following code fragment loads the data stored in the CSV file `data` into the variable `df` and adds a new column of boolean values:

```
> df <- read.csv('data')
  item       date  price    volume
1 GOOG 2009-12-28 622.87   1697900
```

```
2 GOOG 2009-12-29 619.40  1424800
3 GOOG 2009-12-30 622.73  1465600
4 GOOG 2009-12-31 619.98  1219800
5 AAPL 2009-12-28 211.61 23003100
6 AAPL 2009-12-29 209.10 15868400
7 AAPL 2009-12-30 211.64 14696800
8 AAPL 2009-12-31 210.73 12571000

> df$ind <- df$item == "GOOG"
> df
  item       date  value   volume   ind
1 GOOG 2009-12-28 622.87  1697900  TRUE
2 GOOG 2009-12-29 619.40  1424800  TRUE
3 GOOG 2009-12-30 622.73  1465600  TRUE
4 GOOG 2009-12-31 619.98  1219800  TRUE
5 AAPL 2009-12-28 211.61 23003100 FALSE
6 AAPL 2009-12-29 209.10 15868400 FALSE
7 AAPL 2009-12-30 211.64 14696800 FALSE
8 AAPL 2009-12-31 210.73 12571000 FALSE
```

pandas provides a similarly-named `DataFrame` class which implements much of the functionality of its R counterpart, though with some important enhancements (namely, built-in data alignment) which we will discuss. Here we load the same CSV file as above into a `DataFrame` object using the `fromcsv` function and similarly add the above column:

```
>>> data = DataFrame.fromcsv('data', index_col=None)
     date          item     value      volume
0    2009-12-28    GOOG     622.9      1.698e+06
1    2009-12-29    GOOG     619.4      1.425e+06
2    2009-12-30    GOOG     622.7      1.466e+06
3    2009-12-31    GOOG     620        1.22e+06
4    2009-12-28    AAPL     211.6      2.3e+07
5    2009-12-29    AAPL     209.1      1.587e+07
6    2009-12-30    AAPL     211.6      1.47e+07
7    2009-12-31    AAPL     210.7      1.257e+07
>>> data['ind'] = data['item'] == 'GOOG'
```

This data can be reshaped into a different form for future examples by means of the `DataFrame` method `pivot`:

```
>>> df = data.pivot('date', 'item', 'value')
>>> df
              AAPL           GOOG
2009-12-28    211.6          622.9
2009-12-29    209.1          619.4
2009-12-30    211.6          622.7
2009-12-31    210.7          620
```

Beyond observational data, one will also frequently encounter *categorical* data, which can be used to partition identifiers into broader groupings. For example, stock tickers might be categorized by their industry or country of incorporation. Here we have created a `DataFrame` object `cats` storing country and industry classifications for a group of stocks:

```
>>> cats
       country   industry
AAPL   US        TECH
IBM    US        TECH
SAP    DE        TECH
GOOG   US        TECH
C      US        FIN
SCGLY  FR        FIN
BAR    UK        FIN
DB     DE        FIN
VW     DE        AUTO
RNO    FR        AUTO
F      US        AUTO
TM     JP        AUTO
```

We will use these objects above to illustrate features of interest.

pandas data model

The **pandas** data structures internally link the axes of a `ndarray` with arrays of unique labels. These labels are stored in instances of the `Index` class, which is a 1D `ndarray` subclass implementing an *ordered set*. In the stock data above, the row labels are simply sequential observation numbers, while the columns are the field names.

An `Index` stores the labels in two ways: as a `ndarray` and as a `dict` mapping the values (which must therefore be unique and hashable) to the integer indices:

```
>>> index = Index(['a', 'b', 'c', 'd', 'e'])
>>> index
Index([a, b, c, d, e], dtype=object)
>>> index.indexMap
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
```

Creating this `dict` allows the objects to perform lookups and determine membership in constant time.

```
>>> 'a' in index
True
```

These labels are used to provide alignment when performing data manipulations using differently-labeled objects. There are specialized data structures, representing 1-, 2-, and 3-dimensional data, which incorporate useful data handling semantics to facilitate both interactive research and system building. A general n-dimensional data structure would be useful in some cases, but data sets of dimension higher than 3 are very uncommon in most statistical and econometric applications, with 2-dimensional being the most prevalent. We took a pragmatic approach, driven by application needs, to designing the data structures in order to make them as easy-to-use as possible. Also, we wanted the objects to be idiomatically similar to those present in other statistical environments, such as R.

Data alignment

Operations between related, but differently-sized data sets can pose a problem as the user must first ensure that the data points are properly aligned. As an example, consider time series over different date ranges or economic data series over varying sets of entities:

```
>>> s1                  >>> s2
AAPL   0.044            AAPL   0.025
IBM    0.050            BAR    0.158
SAP    0.101            C      0.028
GOOG   0.113            DB     0.087
C      0.138            F      0.004
SCGLY  0.037            GOOG   0.154
BAR    0.200            IBM    0.034
DB     0.281
VW     0.040
```

One might choose to explicitly align (or *reindex*) one of these 1D `Series` objects with the other before adding them, using the `reindex` method:

```
>>> s1.reindex(s2.index)
AAPL     0.0440877763224
BAR      0.199741007422
C        0.137747485628
DB       0.281070058049
F        NaN
```

```
GOOG     0.112861123629
IBM      0.0496445829129
```

However, we often find it preferable to simply ignore the state of data alignment:

```
>>> s1 + s2
AAPL      0.0686791008184
BAR       0.358165479807
C         0.16586702944
DB        0.367679872693
F         NaN
GOOG      0.26666583847
IBM       0.0833057542385
SAP       NaN
SCGLY     NaN
VW        NaN
```

Here, the data have been automatically aligned based on their labels and added together. The result object contains the union of the labels between the two objects so that no information is lost. We will discuss the use of NaN (Not a Number) to represent missing data in the next section.

Clearly, the user pays linear overhead whenever automatic data alignment occurs and we seek to minimize that overhead to the extent possible. Reindexing can be avoided when Index objects are shared, which can be an effective strategy in performance-sensitive applications. [Cython], a widely-used tool for easily creating Python C extensions, has been utilized to speed up these core algorithms.

Handling missing data

It is common for a data set to have missing observations. For example, a group of related economic time series stored in a DataFrame may start on different dates. Carrying out calculations in the presence of missing data can lead both to complicated code and considerable performance loss. We chose to use NaN as opposed to using NumPy MaskedArrays for performance reasons (which are beyond the scope of this paper), as NaN propagates in floating-point operations in a natural way and can be easily detected in algorithms. While this leads to good performance, it comes with drawbacks: namely that NaN cannot be used in integer-type arrays, and it is not an intuitive "null" value in object or string arrays.

We regard the use of NaN as an implementation detail and attempt to provide the user with appropriate API functions for performing common operations on missing data points. From the above example, we can use the valid method to drop missing data, or we could use fillna to replace missing data with a specific value:

```
>>> (s1 + s2).valid()
AAPL     0.0686791008184
BAR      0.358165479807
C        0.16586702944
DB       0.367679872693
GOOG     0.26666583847
IBM      0.0833057542385

>>> (s1 + s2).fillna(0)
AAPL      0.0686791008184
BAR       0.358165479807
C         0.16586702944
DB        0.367679872693
F         0.0
GOOG      0.26666583847
IBM       0.0833057542385
SAP       0.0
SCGLY     0.0
VW        0.0
```

Common ndarray methods have been rewritten to automatically exclude missing data from calculations:

```
>>> (s1 + s2).sum()
1.3103630754662747

>>> (s1 + s2).count()
6
```

Similar to R's is.na function, which detects NA (Not Available) values, **pandas** has special API functions isnull and notnull for determining the validity of a data point. These contrast with numpy.isnan in that they can be used with dtypes other than float and also detect some other markers for "missing" occurring in the wild, such as the Python None value.

```
>>> isnull(s1 + s2)
AAPL     False
BAR      False
C        False
DB       False
F        True
GOOG     False
IBM      False
SAP      True
SCGLY    True
VW       True
```

Note that R's NA value is distinct from NaN. While the addition of a special NA value to NumPy would be useful, it is most likely too domain-specific to merit inclusion.

Combining or joining data sets

Combining, joining, or merging related data sets is a quite common operation. In doing so we are interested in associating observations from one data set with another via a *merge key* of some kind. For similarly-indexed 2D data, the row labels serve as a natural key for the join function:

```
>>> df1                           >>> df2
             AAPL     GOOG                     MSFT     YHOO
2009-12-24   209      618.5      2009-12-24    31       16.72
2009-12-28   211.6    622.9      2009-12-28    31.17    16.88
2009-12-29   209.1    619.4      2009-12-29    31.39    16.92
2009-12-30   211.6    622.7      2009-12-30    30.96    16.98
2009-12-31   210.7    620

>>> df1.join(df2)
            AAPL    GOOG    MSFT    YHOO
2009-12-24  209     618.5   31      16.72
2009-12-28  211.6   622.9   31.17   16.88
2009-12-29  209.1   619.4   31.39   16.92
2009-12-30  211.6   622.7   30.96   16.98
2009-12-31  210.7   620     NaN     NaN
```

One might be interested in joining on something other than the index as well, such as the categorical data we presented in an earlier section:

```
>>> data.join(cats, on='item')
     country  date        industry item   value
0    US       2009-12-28  TECH     GOOG   622.9
1    US       2009-12-29  TECH     GOOG   619.4
2    US       2009-12-30  TECH     GOOG   622.7
3    US       2009-12-31  TECH     GOOG   620
```

```
4    US        2009-12-28  TECH     AAPL   211.6
5    US        2009-12-29  TECH     AAPL   209.1
6    US        2009-12-30  TECH     AAPL   211.6
7    US        2009-12-31  TECH     AAPL   210.7
```

This is akin to a SQL join operation between two tables.

Categorical variables and "Group by" operations

One might want to perform an operation (for example, an aggregation) on a subset of a data set determined by a categorical variable. For example, suppose we wished to compute the mean value by industry for a set of stock data:

```
>>> s                   >>> ind
AAPL   0.044            AAPL   TECH
IBM    0.050            IBM    TECH
SAP    0.101            SAP    TECH
GOOG   0.113            GOOG   TECH
C      0.138            C      FIN
SCGLY  0.037            SCGLY  FIN
BAR    0.200            BAR    FIN
DB     0.281            DB     FIN
VW     0.040            VW     AUTO
                        RNO    AUTO
                        F      AUTO
                        TM     AUTO
```

This concept of "group by" is a built-in feature of many data-oriented languages, such as R and SQL. In R, any vector of non-numeric data can be used as an input to a grouping function such as `tapply`:

```
> labels
[1] GOOG GOOG GOOG GOOG AAPL AAPL AAPL AAPL
Levels: AAPL GOOG
> data
[1] 622.87 619.40 622.73 619.98 211.61 209.10
211.64 210.73

> tapply(data, labels, mean)
   AAPL    GOOG
210.770 621.245
```

pandas allows you to do this in a similar fashion:

```
>>> data.groupby(labels).aggregate(np.mean)
AAPL    210.77
GOOG    621.245
```

One can use `groupby` to concisely express operations on relational data, such as counting group sizes:

```
>>> s.groupby(ind).aggregate(len)
AUTO    1
FIN     4
TECH    4
```

In the most general case, `groupby` uses a function or mapping to produce groupings from one of the axes of a **pandas** object. By returning a `GroupBy` object we can support more operations than just aggregation. Here we can subtract industry means from a data set:

```
demean = lambda x: x - x.mean()

def group_demean(obj, keyfunc):
    grouped = obj.groupby(keyfunc)
    return grouped.transform(demean)

>>> group_demean(s1, ind)
AAPL     -0.0328370881632
BAR      0.0358663891836
C        -0.0261271326111
DB       0.11719543981
GOOG     0.035936259143
IBM      -0.0272802815728
SAP      0.024181110593
SCGLY    -0.126934696382
VW       0.0
```

Manipulating panel (3D) data

A data set about a set of individuals or entities over a time range is commonly referred to as *panel data*; i.e., for each entity over a date range we observe a set of variables. Such data can be found both in *balanced* form (same number of time observations for each individual) or *unbalanced* (different numbers of observations). Panel data manipulations are important for constructing inputs to statistical estimation routines, such as linear regression. Consider the Grunfeld data set [Grun] frequently used in econometrics (sorted by year):

```
>>> grunfeld
       capita    firm      inv       value     year
0      2.8       1         317.6     3078      1935
20     53.8      2         209.9     1362      1935
40     97.8      3         33.1      1171      1935
60     10.5      4         40.29     417.5     1935
80     183.2     5         39.68     157.7     1935
100    6.5       6         20.36     197       1935
120    100.2     7         24.43     138       1935
140    1.8       8         12.93     191.5     1935
160    162       9         26.63     290.6     1935
180    4.5       10        2.54      70.91     1935
1      52.6      1         391.8     4662      1936
21     50.5      2         355.3     1807      1936
41     104.4     3         45        2016      1936
61     10.2      4         72.76     837.8     1936
81     204       5         50.73     167.9     1936
101    15.8      6         25.98     210.3     1936
121    125       7         23.21     200.1     1936
141    0.8       8         25.9      516       1936
161    174       9         23.39     291.1     1936
181    4.71      10        2         87.94     1936
...
```

Really this data is 3-dimensional, with *firm*, *year*, and *item* (data field name) being the three unique keys identifying a data point. Panel data presented in tabular format is often referred to as the *stacked* or *long* format. We refer to the truly 3-dimensional form as the *wide* form. **pandas** provides classes for operating on both:

```
>>> lp = LongPanel.fromRecords(grunfeld, 'year',
                               'firm')
>>> wp = lp.toWide()
>>> wp
<class 'pandas.core.panel.WidePanel'>
Dimensions: 3 (items) x 20 (major) x 10 (minor)
Items: capital to value
Major axis: 1935 to 1954
Minor axis: 1 to 10
```

Now with the data in 3-dimensional form, we can examine the data items separately or compute descriptive statistics more easily (here the `head` function just displays the first 10 rows of the `DataFrame` for the `capital` item):

```
>>> wp['capital'].head()
    1935      1936      1937      1938      1939
1   2.8       265       53.8      213.8     97.8
2   52.6      402.2     50.5      132.6     104.4
3   156.9     761.5     118.1     264.8     118
4   209.2     922.4     260.2     306.9     156.2
5   203.4     1020      312.7     351.1     172.6
```

```
6   207.2    1099     254.2    357.8    186.6
7   255.2    1208     261.4    342.1    220.9
8   303.7    1430     298.7    444.2    287.8
9   264.1    1777     301.8    623.6    319.9
10  201.6    2226     279.1    669.7    321.3
```

In this form, computing summary statistics, such as the time series mean for each (item, firm) pair, can be easily carried out:

```
>>> wp.mean(axis='major')
      capital    inv        value
1     140.8      98.45      923.8
2     153.9      131.5      1142
3     205.4      134.8      1140
4     244.2      115.8      872.1
5     269.9      109.9      998.9
6     281.7      132.2      1056
7     301.7      169.7      1148
8     344.8      173.3      1068
9     389.2      196.7      1236
10    428.5      197.4      1233
```

As an example application of these panel data structures, consider constructing dummy variables (columns of 1's and 0's identifying dates or entities) for linear regressions. Especially for unbalanced panel data, this can be a difficult task. Since we have all of the necessary labeling data here, we can easily implement such an operation as an instance method.

Implementing statistical models

When applying a statistical model, data preparation and cleaning can be one of the most tedious or time consuming tasks. Ideally the majority of this work would be taken care of by the model class itself. In R, while `NA` data can be automatically excluded from a linear regression, one must either align the data and put it into a `data.frame` or otherwise prepare a collection of 1D arrays which are all the same length.

Using **pandas**, the user can avoid much of this data preparation work. As a exemplary model leveraging the **pandas** data model, we implemented ordinary least squares regression in both the standard case (making no assumptions about the content of the regressors) and the panel case, which has additional options to allow for entity and time dummy variables. Facing the user is a single function, `ols`, which infers the type of model to estimate based on the inputs:

```
>>> model = ols(y=Y, x=X)
>>> model.beta
AAPL         0.187984100742
GOOG         0.264882582521
MSFT         0.207564901899
intercept    -0.000896535166817
```

If the response variable `Y` is a `DataFrame` (2D) or `dict` of 1D `Series`, a panel regression will be run on stacked (pooled) data. The `x` would then need to be either a `WidePanel`, `LongPanel`, or a `dict` of `DataFrame` objects. Since these objects contain all of the necessary information to construct the design matrices for the regression, there is nothing for the user to worry about (except the formulation of the model).

The `ols` function is also capable of estimating a *moving window* linear regression for time series data. This can be useful for estimating statistical relationships that change through time:

```
>>> model = ols(y=Y, x=X, window_type='rolling',
                window=250)
>>> model.beta
<class 'pandas.core.matrix.DataFrame'<>
Index: 1103 entries , 2005-08-16 to 2009-12-31
Data columns:
AAPL         1103  non-null values
GOOG         1103  non-null values
MSFT         1103  non-null values
intercept    1103  non-null values
dtype: float64(4)
```

Here we have estimated a moving window regression with a window size of 250 time periods. The resulting regression coefficients stored in `model.beta` are now a `DataFrame` of time series.

Date/time handling

In applications involving time series data, manipulations on dates and times can be quite tedious and inefficient. Tools for working with dates in MATLAB, R, and many other languages are clumsy or underdeveloped. Since Python has a built-in `datetime` type easily accessible at both the Python and C / Cython level, we aim to craft easy-to-use and efficient date and time functionality. When the NumPy `datetime64` dtype has matured, we will, of course, reevaluate our date handling strategy where appropriate.

For a number of years **scikits.timeseries** [SciTS] has been available to scientific Python users. It is built on top of MaskedArray and is intended for fixed-frequency time series. While forcing data to be fixed frequency can enable better performance in some areas, in general we have found that criterion to be quite rigid in practice. The user of **scikits.timeseries** must also explicitly align data; operations involving unaligned data yield unintuitive results.

In designing **pandas** we hoped to make working with time series data intuitive without adding too much overhead to the underlying data model. The **pandas** data structures are *datetime-aware* but make no assumptions about the dates. Instead, when frequency or regularity matters, the user has the ability to generate date ranges or conform a set of time series to a particular frequency. To do this, we have the `DateRange` class (which is also a subclass of `Index`, so no conversion is necessary) and the `DateOffset` class, whose subclasses implement various general purpose and domain-specific time increments. Here we generate a date range between 1/1/2000 and 1/1/2010 at the "business month end" frequency `BMonthEnd`:

```
>>> DateRange('1/1/2000', '1/1/2010',
              offset=BMonthEnd())
<class 'pandas.core.daterange.DateRange'>
offset: <1 BusinessMonthEnd>
[2000-01-31 00:00:00, ..., 2009-12-31 00:00:00]
length: 120
```

A `DateOffset` instance can be used to convert an object containing time series data, such as a `DataFrame` as in our earlier example, to a different frequency using the `asfreq` function:

```
>>> monthly = df.asfreq(BMonthEnd())
             AAPL     GOOG     MSFT     YHOO
2009-08-31   168.2    461.7    24.54    14.61
```

```
2009-09-30   185.3     495.9     25.61     17.81
2009-10-30   188.5     536.1     27.61     15.9
2009-11-30   199.9     583       29.41     14.97
2009-12-31   210.7     620       30.48     16.78
```

Some things which are not easily accomplished in scikits.timeseries can be done using the `DateOffset` model, like deriving custom offsets on the fly or shifting monthly data forward by a number of business days using the `shift` function:

```
>>> offset = Minute(12)
>>> DateRange('6/18/2010 8:00:00',
              '6/18/2010 12:00:00',
              offset=offset)
<class 'pandas.core.daterange.DateRange'>
offset: <12 Minutes>
[2010-06-18 08:00:00, ..., 2010-06-18 12:00:00]
length: 21

>>> monthly.shift(5, offset=Bay())
              AAPL     GOOG     MSFT     YHOO
2009-09-07    168.2    461.7    24.54    14.61
2009-10-07    185.3    495.9    25.61    17.81
2009-11-06    188.5    536.1    27.61    15.9
2009-12-07    199.9    583      29.41    14.97
2010-01-07    210.7    620      30.48    16.78
```

Since **pandas** uses the built-in Python `datetime` object, one could foresee performance issues with very large or high frequency time series data sets. For most general applications financial or econometric applications we cannot justify complicating `datetime` handling in order to solve these issues; specialized tools will need to be created in such cases. This may be indeed be a fruitful avenue for future development work.

Related packages

A number of other Python packages have appeared recently which provide some similar functionality to **pandas**. Among these, **la** ([Larry]) is the most similar, as it implements a labeled `ndarray` object intending to closely mimic NumPy arrays. This stands in contrast to our approach, which is driven by the practical considerations of time series and cross-sectional data found in finance, econometrics, and statistics. The references include a couple other packages of interest ([Tab], [pydataframe]).

While **pandas** provides some useful linear regression models, it is not intended to be comprehensive. We plan to work closely with the developers of **scikits.statsmodels** ([StaM]) to generally improve the cohesiveness of statistical modeling tools in Python. It is likely that **pandas** will soon become a "lite" dependency of **scikits.statsmodels**; the eventual creation of a *superpackage* for statistical modeling including **pandas**, **scikits.statsmodels**, and some other libraries is also not out of the question.

Conclusions

We believe that in the coming years there will be great opportunity to attract users in need of statistical data analysis tools to Python who might have previously chosen R, MATLAB, or another research environment. By designing robust, easy-to-use data structures that cohere with the rest of the scientific Python stack, we can make Python a compelling choice for data analysis applications. In our opinion, **pandas** represents a solid step in the right direction.

References

[pandas] W. McKinney, AQR Capital Management, *pandas: a python data analysis library*, http://pandas.sourceforge.net

[Larry] K. Goodman. *la / larry: ndarray with labeled axes*, http://larry.sourceforge.net/

[SciTS] M. Knox, P. Gerard-Marchant, *scikits.timeseries: python time series analysis*, http://pytseries.sourceforge.net/

[StaM] S. Seabold, J. Perktold, J. Taylor, *scikits.statsmodels: statistical modeling in Python*, http://statsmodels.sourceforge.net

[SciL] D. Cournapeau, et al., *scikits.learn: machine learning in Python*, http://scikit-learn.sourceforge.net

[PyMC] C. Fonnesbeck, A. Patil, D. Huard, *PyMC: Markov Chain Monte Carlo for Python*, http://code.google.com/p/pymc/

[Tab] D. Yamins, E. Angelino, *tabular: tabarray data structure for 2D data*, http://parsemydata.com/tabular/

[NumPy] T. Oliphant, http://numpy.scipy.org

[SciPy] E. Jones, T. Oliphant, P. Peterson, http://scipy.org

[matplotlib] J. Hunter, et al., *matplotlib: Python plotting*, http://matplotlib.sourceforge.net/

[EPD] Enthought, Inc., *EPD: Enthought Python Distribution*, http://www.enthought.com/products/epd.php

[Pythonxy] P. Raybaut, *Python(x,y): Scientific-oriented Python distribution*, http://www.pythonxy.com/

[CRAN] *The R Project for Statistical Computing*, http://cran.r-project.org/

[Cython] G. Ewing, R. W. Bradshaw, S. Behnel, D. S. Seljebotn, et al., *The Cython compiler*, http://cython.org

[IPython] F. Perez, et al., *IPython: an interactive computing environment*, http://ipython.scipy.org

[Grun] Batalgi, *Grunfeld data set*, http://www.wiley.com/legacy/wileychi/baltagi/

[nipy] J. Taylor, F. Perez, et al., *nipy: Neuroimaging in Python*, http://nipy.sourceforge.net

[pydataframe] A. Straw, F. Finkernagel, *pydataframe*, http://code.google.com/p/pydataframe/

[R] R Development Core Team. 2010, *R: A Language and Environment for Statistical Computing*, http://www.R-project.org

[MATLAB] The MathWorks Inc. 2010, *MATLAB*, http://www.mathworks.com

[Stata] StatCorp. 2010, *Stata Statistical Software: Release 11* http://www.stata.com

[SAS] SAS Institute Inc., *SAS System*, http://www.sas.com