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 other 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.

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:

```python
>>> 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, 619.40, 622.73, 619.98, 211.61, 209.10, 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:

```r
> df <- read.csv(‘data’) item date price volume
1 GOOG 2009-12-28 622.87 1697900.0
2 GOOG 2009-12-29 619.40 1424800.0
3 GOOG 2009-12-30 622.73 1465600.0
4 GOOG 2009-12-31 619.98 1219800.0
5 AAPL 2009-12-28 211.61 23003100
```
The pandas data model

The pandas data structures internally link the axes of 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 an ndarray and as a dict: mapping the values (which must therefore be unique and hashable) to the integer indices:

```python
>>> index = Index(['a', 'b', 'c', 'd', 'e'])
>>> index
Index(['a', 'b', 'c', 'd', 'e'], dtype=object)
```

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

```python
>>> '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:

```python
>>> s1
AAPL 0.044
IBM 0.050
SAP 0.101
GOOG 0.113
C 0.138
SCGLY 0.037
BAR 0.200
DB 0.281
VW 0.040

>>> s2
AAPL 0.025
BAR 0.158
C 0.028
DB 0.087
F 0.004
GOOG 0.154
IBM 0.034
```

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

```python
>>> s1.reindex(s2.index)
AAPL 0.044
BAR 0.199
IBM 0.050
SAP 0.101
GOOG 0.113
C 0.138
SCGLY 0.037
BAR 0.200
DB 0.281
VW 0.040
```

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

```python
>>> s1 + s2
AAPL 0.0686791008184
BAR 0.358165479807
C 0.16586702944
```
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:

```python
>>> (s1 + s2).valid()
```

```
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:

```python
>>> df1.join(df2)
```

```
2009-12-24 209 618.5 2009-12-29 209.1 619.4 2009-12-30 211.6 622.7 2009-12-31 210.7 620
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
```

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:

```python
>>> data.join(cats, on='item')
```

```
2009-12-24 209 618.5 2009-12-29 209.1 619.4 2009-12-30 211.6 622.7 2009-12-31 210.7 620
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
```

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:

```python
>>> s >>> ind
```

```
AAPL 0.044
IBM 0.050
```

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.
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`:

```r
> 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
210.77 621.245
```

`pandas` allows you to do this in a similar fashion:

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

Finally, consider the Grunfeld data set [Grun] frequently used in econometrics (sorted by year):

```python
>>> grunfeld
capital 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 1171 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
```

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):

```python
>>> wp['capital'].head()
1935 1936 1937 1938 1939
1 140.8 265 53.8 213.8 97.8
2 52.6 402.2 50.5 132.6 118.1
3 209.2 922.4 260.2 306.9 156.2
4 203.4 1020 312.7 351.1 172.6
5 207.2 1099 254.2 357.8 186.6
```

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

```python
>>> wp.mean(axis='major')
capital inv value
1935 1936 1937 1938 1939
1 140.8 265 53.8 213.8 97.8
2 52.6 402.2 50.5 132.6 118.1
3 209.2 922.4 260.2 306.9 156.2
4 203.4 1020 312.7 351.1 172.6
5 207.2 1099 254.2 357.8 186.6
```

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

```python
>>> wp.drop(['firm'], axis=1).mean(axis='major')
capital inv value
1935 1936 1937 1938 1939
1 140.8 265 53.8 213.8 97.8
2 52.6 402.2 50.5 132.6 118.1
3 209.2 922.4 260.2 306.9 156.2
4 203.4 1020 312.7 351.1 172.6
5 207.2 1099 254.2 357.8 186.6
```

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 the majority of this work would be taken care of by the model, it can be one of the most tedious or time consuming tasks. Ideally necessary labeling data here, we can easily implement such a preparation of a collection of 1D arrays which are all the same length.

Using pandas, the user can avoid much of this data preparation work. As an 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:

```python
>>> 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:

```python
>>> model = ols(y=Y, x=X, window_type='rolling', window=250)
>>> model.beta
<class 'pandas.core.daterange.DateRange'>
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 datetimeware 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:

```python
>>> 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:

```python
>>> monthly = df.asfreq(BMonthEnd())
```

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:

```python
>>> 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
```

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