pandas: a Foundational Python Library for Data Analysis and Statistics

pandas: a Foundational Python Library for Data Analysis and Statistics

Wes McKinney

Abstract—In this paper we will discuss **pandas**, a Python library of rich data structures and tools for working with structured data sets common to statistics, finance, social sciences, and many other fields. The library provides integrated, intuitive routines for performing common data manipulations and analysis on such data sets. It aims to be the foundational layer for the future of statistical computing in Python. It serves as a strong complement to the existing scientific Python stack while implementing and improving upon the kinds of data manipulation tools found in other statistical programming languages such as R. In addition to detailing its design and features of **pandas**, we will discuss future avenues of work and growth opportunities for statistics and data analysis applications in the Python language.

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.

One 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.

Another issue preventing many from using Python in the past for data analysis applications has been the lack of rich data structures with integrated handling of *metadata*. By metadata we mean labeling information about data points. For example, a table or spreadsheet of data will likely have labels for the columns and possibly also the rows. Alternately, some columns in a table might be used for grouping and aggregating data into a pivot or contingency table. In the case of a time series data set, the row labels could be time stamps. It is often necessary to have the labeling information available to allow many kinds of data manipulations, such as merging data sets or performing an aggregation or "group by" operation, to be expressed in an intuitive and concise way. Domain-specific database languages like SQL and statistical languages like R and SAS have a wealth of such tools. Until relatively recently, Python had few tools providing the same level of richness and expressiveness for working with labeled data sets.

The **pandas** library, under development since 2008, is intended to close the gap in the richness of available data analysis tools between Python, a general purpose systems and scientific computing language, and the numerous domain-specific statistical computing platforms and database languages. We not only aim to provide equivalent functionality but also implement many features, such as automatic data alignment and hierarchical indexing, which are not readily available in such a tightly integrated way in any other libraries or computing environments to our knowledge. While initially developed for financial data analysis applications, we hope that **pandas** will enable scientific Python to be a more attractive and practical statistical computing environment for academic and industry practitioners alike. The library's name derives from ***pan**el **da**ta*, a common term for multidimensional data sets encountered in statistics and econometrics.

While we offer a vignette of some of the main features of interest in **pandas**, this paper is by no means comprehensive. For more, we refer the interested reader to the online documentation at `http://pandas.sf.net` ([pandas]).

Structured data sets

Structured 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),
```

Corresponding author can be contacted at: wesmckinn@gmail.com.

```
        ('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) NumPy 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 stores mixed-type data as a collection of independent columns. 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  price   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 which we will discuss. Here we convert the structured array above into a pandas `DataFrame` object and similarly add the same column:

```
>>> from pandas import DataFrame
>>> data = DataFrame(data)
>>> data
   item  date        price  volume
0  GOOG  2009-12-28  622.9  1.698e+06
1  GOOG  2009-12-29  619.4  1.425e+06
2  GOOG  2009-12-30  622.7  1.466e+06
3  GOOG  2009-12-31  620    1.22e+06
4  AAPL  2009-12-28  211.6  2.3e+07
5  AAPL  2009-12-29  209.1  1.587e+07
6  AAPL  2009-12-30  211.6  1.47e+07
7  AAPL  2009-12-31  210.7  1.257e+07
>>> data['ind'] = data['item'] == 'GOOG'
>>> data
   item  date        price  volume     ind
0  GOOG  2009-12-28  622.9  1.698e+06  True
1  GOOG  2009-12-29  619.4  1.425e+06  True
2  GOOG  2009-12-30  622.7  1.466e+06  True
3  GOOG  2009-12-31  620    1.22e+06   True
4  AAPL  2009-12-28  211.6  2.3e+07    False
5  AAPL  2009-12-29  209.1  1.587e+07  False
6  AAPL  2009-12-30  211.6  1.47e+07   False
7  AAPL  2009-12-31  210.7  1.257e+07  False
```

This data can be reshaped or "pivoted" on the `date` and `item` columns into a different form for future examples by means of the `DataFrame` method `pivot`:

```
>>> del data['ind'] # delete ind column
>>> data.pivot('date', 'item')
            price         volume
  item      AAPL   GOOG   AAPL       GOOG
date
2009-12-28  211.6  622.9  2.3e+07    1.698e+06
2009-12-29  209.1  619.4  1.587e+07  1.425e+06
2009-12-30  211.6  622.7  1.47e+07   1.466e+06
2009-12-31  210.7  620    1.257e+07  1.22e+06
```

The result of the pivot operation has a *hierarchical index* for the columns. As we will show in a later section, this is a powerful and flexible way of representing and manipulating multidimensional data. Currently the `pivot` method of DataFrame only supports pivoting on two columns to reshape the data, but could be augmented to consider more than just two columns. By using hierarchical indexes, we can guarantee that the result will always be two-dimensional. Later in the paper we will demonstrate the `pivot_table` function which can produce spreadsheet-style pivot table data summaries as DataFrame objects with hierarchical rows and columns.

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
```

pandas data model

Each axis of a **pandas** data structure has an `Index` object which stores labeling information about each tick along that axis. The most general `Index` is simply a 1-dimensional vector of labels (stored in a NumPy `ndarray`). It's convenient to think about the `Index` as an implementation of an *ordered set*. In the stock data above, the row index contains simply sequential observation numbers, while the column index contains the column names. The labels are **not** required to be sorted, though a *subclass* of `Index` could be implemented to require sortedness and provide operations optimized for sorted data (e.g. time series data).

The `Index` object is used for many purposes:

- Performing *lookups* to select subsets of slices of an object
- Providing fast data alignment routines for aligning one object with another
- Enabling intuitive slicing / selection to form new `Index` objects
- Forming unions and intersections of `Index` objects

Here are some examples of how the index is used internally:

```
>>> index = Index(['a', 'b', 'c', 'd', 'e'])
>>> 'c' in index
True
>>> index.get_loc('d')
3
>>> index.slice_locs('b', 'd')
(1, 4)

# for aligning data
>>> index.get_indexer(['c', 'e', 'f'])
array([ 2,  4, -1], dtype=int32)
```

The basic `Index` uses a Python dict internally to map labels to their respective locations and implement these features, though subclasses could take a more specialized and potentially higher performance approach.

Multidimensional objects like `DataFrame` are not proper subclasses of NumPy's `ndarray` nor do they use arrays with structured dtype. In recent releases of **pandas** there is a new internal data structure known as `BlockManager` which manipulates a collection of *n*-dimensional `ndarray` objects we refer to as blocks. Since `DataFrame` needs to be able to store mixed-type data in the columns, each of these internal `Block` objects contains the data for a set of columns all having the same type. In the example from above, we can examine the `BlockManager`, though most users would never need to do this:

```
>>> data._data
BlockManager
Items: [item date price volume ind]
Axis 1: [0 1 2 3 4 5 6 7]
FloatBlock: [price volume], 2 x 8, dtype float64
ObjectBlock: [item date], 2 x 8, dtype object
BoolBlock: [ind], 1 x 8, dtype bool
```

The key importance of `BlockManager` is that many operations, e.g. anything row-oriented (as opposed to column-oriented), especially in homogeneous `DataFrame` objects, are significantly faster when the data are all stored in a single `ndarray`. However, as it is common to insert and delete columns, it would be wasteful to have a reallocate-copy step on each column insertion or deletion step. As a result, the `BlockManager` effectively provides a *lazy evaluation* scheme where-in newly inserted columns are stored in new `Block` objects. Later, either explicitly or when certain methods are called in `DataFrame`, blocks having the same type will be *consolidated*, i.e. combined together, to form a single homogeneously-typed `Block`:

```
>>> data['newcol'] = 1.

>>> data._data
BlockManager
Items: [item date price volume ind newcol]
Axis 1: [0 1 2 3 4 5 6 7]
FloatBlock: [price volume], 2 x 8
ObjectBlock: [item date], 2 x 8
BoolBlock: [ind], 1 x 8
FloatBlock: [newcol], 1 x 8

>>> data.consolidate()._data
BlockManager
Items: [item date price volume ind newcol]
Axis 1: [0 1 2 3 4 5 6 7]
BoolBlock: [ind], 1 x 8
FloatBlock: [price volume newcol], 3 x 8
ObjectBlock: [item date], 2 x 8
```

The separation between the internal `BlockManager` object and the external, user-facing `DataFrame` gives the **pandas** developers a significant amount of freedom to modify the internal structure to achieve better performance and memory usage.

Label-based data access

While standard `[]`-based indexing (using `__getitem__` and `__setitem__`) is reserved for column access in `DataFrame`, it is useful to be able to index both axes of a `DataFrame` in a matrix-like way using labels. We would like to be able to get or set data on any axis using one of the following:

- A list or array of labels or integers
- A slice, either with integers (e.g. `1:5`) or labels (e.g. `lab1:lab2`)
- A boolean vector
- A single label

To avoid excessively overloading the `[]`-related methods, leading to ambiguous indexing semantics in some cases, we have implemented a special label-indexing attribute `ix` on all of the pandas data structures. Thus, we can pass a tuple of any of the above indexing objects to get or set values.

```
>>> df
                 A       B       C       D
2000-01-03 -0.2047  1.007  -0.5397 -0.7135
2000-01-04  0.4789 -1.296   0.477  -0.8312
2000-01-05 -0.5194  0.275   3.249  -2.37
2000-01-06 -0.5557  0.2289 -1.021  -1.861
2000-01-07  1.966   1.353  -0.5771 -0.8608

>>> df.ix[:2, ['D', 'C', 'A']]
                 D       C       A
2000-01-03 -0.7135 -0.5397 -0.2047
2000-01-04 -0.8312  0.477   0.4789

>>> df.ix[-2:, 'B':]
                 B       C       D
2000-01-06  0.2289 -1.021  -1.861
2000-01-07  1.353  -0.5771 -0.8608
```

Setting values also works as expected.

```
>>> date1, date2 = df.index[[1, 3]]
>>> df.ix[date1:date2, ['A', 'C']] = 0
>>> df
                 A        B       C       D
2000-01-03 -0.6856   0.1362  0.3996  1.585
2000-01-04  0        0.8863  0       1.907
2000-01-05  0       -1.351   0       0.104
2000-01-06  0       -0.8863  0       0.1741
2000-01-07 -0.05927 -1.013   0.9923 -0.4395
```

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 creating Python C extensions and interfacing with C/C++ code, has been utilized to speed up these core algorithms.

Data alignment using `DataFrame` occurs automatically on both the column and row labels. This deeply integrated data alignment differs from any other tools outside of Python that we are aware of. Similar to the above, if the columns themselves are different, the resulting object will contain the union of the columns:

```
>>> df                        >>> df2
            AAPL   GOOG                   AAPL
2009-12-28  211.6  622.9      2009-12-28  2.3e+07
2009-12-29  209.1  619.4      2009-12-29  1.587e+07
2009-12-30  211.6  622.7      2009-12-30  1.47e+07
2009-12-31  210.7  620

>>> df / df2
            AAPL       GOOG
2009-12-28  9.199e-06  NaN
2009-12-29  1.318e-05  NaN
2009-12-30  1.44e-05   NaN
2009-12-31  NaN        NaN
```

This may seem like a simple feature, but in practice it grants immense freedom as there is no longer a need to sanitize data from an untrusted source. For example, if you loaded two data sets from a database and the columns and rows, they can be added together, say, without having to do any checking whether the labels are aligned. Of course, after doing an operation between two data sets, you can perform an ad hoc cleaning of the results using such functions as `fillna` and `dropna`:

```
>>> (df / df2).fillna(0)
            AAPL       GOOG
2009-12-28  9.199e-06  0
2009-12-29  1.318e-05  0
2009-12-30  1.44e-05   0
2009-12-31  0          0

>>> (df / df2).dropna(axis=1, how='all')
            AAPL
2009-12-28  9.199e-06
2009-12-29  1.318e-05
2009-12-30  1.44e-05
2009-12-31  NaN
```

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 the NumPy `MaskedArray` object 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 (though it is used in these arrays regardless).

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 `dropna` method to drop missing data, or we could use `fillna` to replace missing data with a specific value:

```
>>> (s1 + s2).dropna()
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
```

The `reindex` and `fillna` methods are equipped with a couple simple interpolation options to propagate values forward and backward, which is especially useful for time series data:

```
>>> ts                        >>> ts2
2000-01-03   0.03825          2000-01-03   0.03825
2000-01-04   -1.9884          2000-01-06   -0.0588
2000-01-05   0.73255          2000-01-11   0.04410
2000-01-06   -0.0588          2000-01-14   -0.1786
2000-01-07   -0.4767
2000-01-10   1.98008
2000-01-11   0.04410

>>> ts3 = ts + ts2
>>> ts3                      >>> ts3.fillna(method='ffill')
2000-01-03  0.07649          2000-01-03   0.07649
2000-01-04  NaN              2000-01-04   0.07649
2000-01-05  NaN              2000-01-05   0.07649
2000-01-06  -0.1177          2000-01-06   -0.1177
2000-01-07  NaN              2000-01-07   -0.1177
2000-01-10  NaN              2000-01-10   -0.1177
2000-01-11  0.08821          2000-01-11   0.08821
2000-01-14  NaN              2000-01-14   0.08821
```

`Series` and `DataFrame` also have explicit arithmetic methods with which a `fill_value` can be used to specify a treatment of missing data in the computation. An occasional choice is to treat missing values as 0 when adding two `Series` objects:

```
>>> ts.add(ts2, fill_value=0)
2000-01-03    0.0764931953608
2000-01-04    -1.98842046359
2000-01-05    0.732553684194
2000-01-06    -0.117727627078
2000-01-07    -0.476754320696
2000-01-10    1.9800873096
2000-01-11    0.0882102892097
2000-01-14    -0.178640361674
```

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`. NumPy core developers are currently working on an `NA` value implementation that will hopefully suit the needs of libraries like `pandas` in the future.

Hierarchical Indexing

A relatively recent addition to pandas is the ability for an axis to have a *hierarchical* index, known in the library as a `MultiIndex`. Semantically, this means that each a location on a single axis can have multiple labels associated with it.

```
>>> hdf
            A         B         C
foo  one   -0.9884    0.09406  1.263
     two    1.29      0.08242 -0.05576
     three  0.5366   -0.4897   0.3694
bar  one   -0.03457 -2.484    -0.2815
     two    0.03071  0.1091    1.126
baz  two   -0.9773   1.474    -0.06403
     three -1.283    0.7818   -1.071
qux  one    0.4412   2.354     0.5838
     two    0.2215  -0.7445    0.7585
     three  1.73    -0.965    -0.8457
```

Hierarchical indexing can be viewed as a way to represent higher-dimensional data in a lower-dimensional data structure (here, a 2D `DataFrame`). For example, we can select rows from the above `DataFrame` by specifying only a label from the left-most level of the index:

```
>>> hdf.ix['foo']
        A         B         C
one   -0.9884   0.09406  1.263
two    1.29     0.08242 -0.05576
three  0.5366  -0.4897   0.3694
```

Of course, if all of the levels are specified, we can select a row or column just as with a regular `Index`.

```
>>> hdf.ix['foo', 'three']
A    0.5366
B    -0.4897
C    0.3694

# same result
>>> hdf.ix['foo'].ix['three']
```

The hierarchical index can be used with any axis. From the `pivot` example earlier in the paper we obtained:

```
>>> pivoted = data.pivot('date', 'item')
>>> pivoted
            price          volume
            AAPL   GOOG    AAPL        GOOG
2009-12-28  211.6  622.9   2.3e+07     1.698e+06
2009-12-29  209.1  619.4   1.587e+07   1.425e+06
2009-12-30  211.6  622.7   1.47e+07    1.466e+06
2009-12-31  210.7  620     1.257e+07   1.22e+06

>>> pivoted['volume']
            AAPL        GOOG
2009-12-28  2.3e+07     1.698e+06
2009-12-29  1.587e+07   1.425e+06
2009-12-30  1.47e+07    1.466e+06
2009-12-31  1.257e+07   1.22e+06
```

There are several utility methods for manipulating a `MultiIndex` such as `swaplevel` and `sortlevel`:

```
>>> swapped = pivoted.swaplevel(0, 1, axis=1)
>>> swapped
            AAPL   GOOG   AAPL       GOOG
            price  price  volume     volume
2009-12-28  211.6  622.9  2.3e+07    1.698e+06
2009-12-29  209.1  619.4  1.587e+07  1.425e+06
2009-12-30  211.6  622.7  1.47e+07   1.466e+06
2009-12-31  210.7  620    1.257e+07  1.22e+06

>>> swapped['AAPL']
            price  volume
2009-12-28  211.6  2.3e+07
2009-12-29  209.1  1.587e+07
2009-12-30  211.6  1.47e+07
2009-12-31  210.7  1.257e+07
```

Here is an example for `sortlevel`:

```
>>> pivoted.sortlevel(1, axis=1)
            price  volume     price  volume
            AAPL   AAPL       GOOG   GOOG
2009-12-28  211.6  2.3e+07    622.9  1.698e+06
2009-12-29  209.1  1.587e+07  619.4  1.425e+06
2009-12-30  211.6  1.47e+07   622.7  1.466e+06
2009-12-31  210.7  1.257e+07  620    1.22e+06
```

Advanced pivoting and reshaping

Closely related to hierarchical indexing and the earlier pivoting example, we illustrate more advanced reshaping of data using the `stack` and `unstack` methods. `stack` reshapes by removing a level from the columns of a `DataFrame` object and moving that level to the row labels, producing either a 1D `Series` or another `DataFrame` (if the columns were a `MultiIndex`).

```
>>> 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

>>> df.stack()
2009-12-28  AAPL    211.61
            GOOG    622.87
2009-12-29  AAPL    209.1
            GOOG    619.4
2009-12-30  AAPL    211.64
            GOOG    622.73
2009-12-31  AAPL    210.73
            GOOG    619.98

>>> pivoted
            price         volume
            AAPL   GOOG   AAPL       GOOG
2009-12-28  211.6  622.9  2.3e+07    1.698e+06
2009-12-29  209.1  619.4  1.587e+07  1.425e+06
2009-12-30  211.6  622.7  1.47e+07   1.466e+06
2009-12-31  210.7  620    1.257e+07  1.22e+06

>>> pivoted.stack()
                  price  volume
2009-12-28  AAPL  211.6  2.3e+07
            GOOG  622.9  1.698e+06
2009-12-29  AAPL  209.1  1.587e+07
            GOOG  619.4  1.425e+06
2009-12-30  AAPL  211.6  1.47e+07
            GOOG  622.7  1.466e+06
2009-12-31  AAPL  210.7  1.257e+07
            GOOG  620    1.22e+06
```

By default, the *innermost* level is stacked. The level to stack can be specified explicitly:

```
>>> pivoted.stack(0)
                    AAPL       GOOG
2009-12-28  price   211.6      622.9
            volume  2.3e+07    1.698e+06
2009-12-29  price   209.1      619.4
            volume  1.587e+07  1.425e+06
2009-12-30  price   211.6      622.7
            volume  1.47e+07   1.466e+06
2009-12-31  price   210.7      620
            volume  1.257e+07  1.22e+06
```

The `unstack` method is the inverse of `stack`:

```
>>> df.stack()                   >>> df.stack().unstack()
2009-12-28  AAPL   211.61                    AAPL   GOOG
            GOOG   622.87        2009-12-28  211.6  622.9
2009-12-29  AAPL   209.1         2009-12-29  209.1  619.4
            GOOG   619.4         2009-12-30  211.6  622.7
2009-12-30  AAPL   211.64        2009-12-31  210.7  620
            GOOG   622.73
2009-12-31  AAPL   210.73
            GOOG   619.98
```

These reshaping methods can be combined with built-in `DataFrame` and `Series` method to select or aggregate data at a level. Here we take the maximum among `AAPL` and `GOOG` for each date / field pair:

```
>>> pivoted.stack(0)
                    AAPL       GOOG
2009-12-28  price   211.6      622.9
            volume  2.3e+07    1.698e+06
2009-12-29  price   209.1      619.4
            volume  1.587e+07  1.425e+06
2009-12-30  price   211.6      622.7
            volume  1.47e+07   1.466e+06
2009-12-31  price   210.7      620
            volume  1.257e+07  1.22e+06

>>> pivoted.stack(0).max(1).unstack()
            price  volume
2009-12-28  622.9  2.3e+07
2009-12-29  619.4  1.587e+07
2009-12-30  622.7  1.47e+07
2009-12-31  620    1.257e+07
```

These kinds of aggregations are closely related to "group by" operations which we discuss in the next section.

Group By: grouping and aggregating data

A very common operation in SQL-like languages and generally in statistical data analysis is to group data by some identifiers and perform either an aggregation or transformation of the data. For example, suppose we had a simple data set like this:

```
>>> df
   A    B      C        D
0  foo  one    -1.834   1.903
1  bar  one     1.772  -0.7472
2  foo  two    -0.67   -0.309
3  bar  three  0.04931  0.3939
4  foo  two    -0.5215  1.861
5  bar  two    -3.202   0.9365
6  foo  one     0.7927  1.256
7  foo  three  0.1461  -2.655
```

We could compute group means using the `A` column like so:

```
>>> df.groupby('A').mean()
     C        D
bar -0.4602  0.1944
```

```
foo -0.4173  0.4112
```

The object returned by groupby is a special intermediate object with a lot of nice features. For example, you can use it to iterate through the portions of the data set corresponding to each group:

```
>>> for key, group in df.groupby('A'):
...     print key
...     print group
bar
   A    B      C        D
1  bar  one    1.772   -0.7472
3  bar  three  0.04931  0.3939
5  bar  two   -3.202    0.9365

foo
   A    B      C        D
0  foo  one   -1.834    1.903
2  foo  two   -0.67    -0.309
4  foo  two   -0.5215   1.861
6  foo  one    0.7927   1.256
7  foo  three  0.1461  -2.65
```

Grouping by multiple columns is also possible:

```
df.groupby(['A', 'B']).mean()
            C        D
bar  one    1.772   -0.7472
     three  0.04931  0.3939
     two   -3.202    0.9365
foo  one   -0.5205   1.579
     three  0.1461  -2.655
     two   -0.5958   0.7762
```

The default result of a multi-key groupby aggregation is a hierarchical index. This can be disabled when calling groupby which may be useful in some settings:

```
df.groupby(['A', 'B'], as_index=False).mean()
   A    B      C        D
0  bar  one    1.772   -0.7472
1  bar  three  0.04931  0.3939
2  bar  two   -3.202    0.9365
3  foo  one   -0.5205   1.579
4  foo  three  0.1461  -2.655
5  foo  two   -0.5958   0.7762
```

In a completely general setting, groupby operations are about mapping axis labels to buckets. In the above examples, when we pass column names we are simply establishing a *correspondence* between the row labels and the group identifiers. There are other ways to do this; the most general is to pass a Python function (for single-key) or list of functions (for multi-key) which will be invoked on each each label, producing a group specification:

```
>>> dat
            A       B       C       D
2000-01-03  0.6371  0.672   0.9173  1.674
2000-01-04 -0.8178 -1.865  -0.23    0.5411
2000-01-05  0.314   0.2931 -0.6444 -0.9973
2000-01-06  1.913  -0.5867  0.273   0.4631
2000-01-07  1.308   0.426  -1.306   0.04358

>>> mapping
{'A': 'Group 1', 'B': 'Group 2',
 'C': 'Group 1', 'D': 'Group 2'}

>>> for name, group in dat.groupby(mapping.get,
...                                axis=1):
...     print name; print group
Group 1
            A       C
2000-01-03  0.6371  0.9173
2000-01-04 -0.8178 -0.23
2000-01-05  0.314  -0.6444
2000-01-06  1.913   0.273
2000-01-07  1.308  -1.306

Group 2
            B       D
2000-01-03  0.672   1.674
2000-01-04 -1.865   0.5411
2000-01-05  0.2931 -0.9973
2000-01-06 -0.5867  0.4631
2000-01-07  0.426   0.04358
```

Some creativity with grouping functions will enable the user to perform quite sophisticated operations. The object returned by groupby can either iterate, aggregate (with an arbitrary function), transform (compute a modified same-size version of each data group), or do a general apply-by-group. While we do not have space to go into great detail with examples of each of these, the apply function is interesting in that it attempts to combine the results of the aggregation into a pandas object. For example, we could group the df object above by column A, select just the C column, and apply the describe function to each subgroup like so:

```
>>> df.groupby('A')['C'].describe().T
       bar      foo
count  3        5
mean  -0.4602  -0.4173
std    2.526    0.9827
min   -3.202   -1.834
10%   -2.552   -1.368
50%    0.04931 -0.5215
90%    1.427    0.5341
max    1.772    0.7927
```

Note that, under the hood, calling describe generates and passes a dynamic function to apply which invokes describe on each group and glues the results together. We transposed the result with .T to make it more readable.

Easy spreadsheet-style pivot tables

An obvious application combining groupby and reshaping operations is creating *pivot tables*, a common way of summarizing data in spreadsheet applications such as Microsoft Excel. We'll take a brief look at a tipping data set collected from a restaurant ([Bryant]):

```
>>> tips.head()
   sex     smoker  time    day  size  tip_pct
1  Female  No      Dinner  Sun  2     0.05945
2  Male    No      Dinner  Sun  3     0.1605
3  Male    No      Dinner  Sun  3     0.1666
4  Male    No      Dinner  Sun  2     0.1398
5  Female  No      Dinner  Sun  4     0.1468
```

The pivot_table function in pandas takes a set of column names to group on the pivot table rows, another set to group on the columns, and optionally an aggregation function for each group (which defaults to mean):

```
>>> import numpy as np
>>> from pandas import pivot_table
>>> pivot_table(tips, 'tip_pct', rows=['time', 'sex'],
                cols='smoker')
  smoker       No      Yes
time   sex
Dinner Female  0.1568  0.1851
```

```
       Male    0.1594  0.1489
Lunch  Female  0.1571  0.1753
       Male    0.1657  0.1667
```

Conveniently, the returned object is a DataFrame, so it can be further reshaped and manipulated by the user:

```
>>> table = pivot_table(tips, 'tip_pct',
                        rows=['sex', 'day'],
                        cols='smoker', aggfunc=len)
>>> table
  smoker      No  Yes
sex    day
Female Fri    2   7
       Sat    13  15
       Sun    14  4
       Thur   25  7
Male   Fri    2   8
       Sat    32  27
       Sun    43  15
       Thur   20  10

>>> table.unstack('sex')
  smoker  No           Yes
  sex     Female  Male  Female  Male
day
Fri       2       2     7       8
Sat       13      32    15      27
Sun       14      43    4       15
Thur      25      20    7       10
```

For many users, this will be an attractive alternative to dumping a data set into a spreadsheet for the sole purpose of creating a pivot table.

```
>>> pivot_table(tips, 'size',
                rows=['time', 'sex', 'smoker'],
                cols='day', aggfunc=np.sum,
                fill_value=0)
  day                 Fri  Sat  Sun  Thur
time   sex    smoker
Dinner Female No      2    30   43   2
              Yes     8    33   10   0
Dinner Male   No      4    85   124  0
              Yes     12   71   39   0
Lunch  Female No      3    0    0    60
              Yes     6    0    0    17
Lunch  Male   No      0    0    0    50
              Yes     5    0    0    23
```

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 or a VLOOKUP operation in a spreadsheet such as Excel. It is possible to join on multiple keys, in which case the table being joined is currently required to have a hierarchical index corresponding to those keys. We will be working on more joining and merging methods in a future release of pandas.

Performance and use for Large Data Sets

Using DataFrame objects over homogeneous NumPy arrays for computation incurs overhead from a number of factors:

- Computational functions like `sum`, `mean`, and `std` have been overridden to omit missing data
- Most of the axis `Index` data structures are reliant on the Python `dict` for performing lookups and data alignment. This also results in a slightly larger memory footprint as the `dict` containing the label mapping is created once and then stored.
- The internal `BlockManager` data structure *consolidates* the data of each type (floating point, integer, boolean, object) into 2-dimensional arrays. However, this is an upfront cost that speeds up row-oriented computations and data alignment later.
- Performing repeated lookups of values by label passes through much more Python code than simple integer-based lookups on `ndarray` objects.

The savvy user will learn what operations are not very efficient in DataFrame and Series and fall back on working directly with the underlying `ndarray` objects (accessible via the `values` attribute) in such cases. What DataFrame sacrifices in performance it makes up for in flexibility and expressiveness.

With 64-bit integers representing timestamps, pandas in fact provides some of the fastest data alignment routines for differently-indexed time series to be found in open source software. As working with large, irregularly time series requires having a timestamp index, pandas is well-positioned to become the gold standard for high performance open source time series processing.

With regard to memory usage and large data sets, pandas is currently only designed for use with *in-memory* data sets. We would like to expand its capability to work with data sets that do not fit into memory, perhaps transparently using the `multiprocessing` module or a parallel computing backend to orchestrate large scale computations.

pandas for R users

Given the "DataFrame" name and feature overlap with the [R] project and its 3rd party packages, pandas will draw inevitable comparisons with R. pandas brings a robust, full-featured, and integrated data analysis toolset to Python while maintaining a simple and easy-to-use API. As nearly all data manipulations involving `data.frame` objects in R can be easily expressed using the pandas `DataFrame`, it is relatively straightforward in most cases to port R functions to Python. It would be useful to provide a migration guide for R users as we have not copied R's naming conventions or syntax in most places, rather naming based on common-sense and making the syntax and API as "Pythonic" as possible.

R does not provide indexing functionality in nearly such a deeply integrated way as pandas does. For example, operations between `data.frame` objects will proceed in R without regard to whether the labels match as long as they are the same length and width. Some R packages, such as `zoo` and `xts` provides indexed data structures with data alignment, but they are largely specialized to ordered time series data. Hierarchical indexing with constant-time subset selection is another significant feature missing from R's data structures.

Outside of the scope of this paper is a rigorous performance comparison of R and pandas. In almost all of the benchmarks we have run comparing R and pandas, pandas significantly outperforms R.

Other features of note

There are many other features in **pandas** worth exploring for the interested users:

- Time series functionality: date range generation, shifting and lagging, frequency conversion and forward/backward filling
- Integration with [matplotlib] to concisely generate plots with metadata
- Moving window statistics (e.g. moving standard deviation, exponentially weighted moving average) and moving window linear and panel regression
- 3-dimensional `Panel` data structure for manipulating collections of `DataFrame` objects
- Sparse versions of the data structures
- Robust IO tools for reading and writing pandas objects to flat files (delimited text, CSV, Excel) and HDF5 format

Related packages

A number of other Python packages have some degree of feature overlap with **pandas**. Among these, **la** ([Larry]) is the most similar, as it implements a labeled `ndarray` object intending to closely mimic NumPy arrays. Since `ndarray` is only applicable many problems in its homogeneous (non-structured dtype) form, in **pandas** we have distanced ourselves from `ndarray` to instead provide a more flexible, (potentially) heterogeneous, size-mutable data structure. The references include a some other packages of interest.

pandas will soon become a dependency of **statsmodels** ([StaM]), the main statistics and econometric library in Python, to make statistical modeling and data analysis tools in Python more cohesive and integrated. We plan to combine **pandas** with a formula framework to make specifying statistical models easy and intuitive when working with a `DataFrame` of data, for example.

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** provides a solid foundation upon which a very powerful data analysis ecosystem can be established.

References

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

[scipy2010] W. McKinney, *Data Structures for Statistical Computing in Python* Proceedings of the 9th Python in Science Conference, http://http://conference.scipy.org/. 2010

[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, *statsmodels: statistical modeling in Python*, http://statsmodels.sourceforge.net

[SciL] D. Cournapeau, et al., *scikit-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] Fernando Pérez, Brian E. Granger, *IPython: A System for Interactive Scientific Computing*, Computing in Science and Engineering, vol. 9, no. 3, pp. 21-29, May/June 2007, doi:10.1109/MCSE.2007.53. http://ipython.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

[Bryant] Bryant, P. G. and Smith, M (1995) *Practical Data Analysis: Case Studies in Business Statistics.* Homewood, IL: Richard D. Irwin Publishing: