Fast Data Mining with pandas and PyTables

Dr. Yves J. Hilpisch

05 July 2012

EuroPython Conference 2012 in Florence

Visixion GmbH
Finance, Derivatives Analytics & Python Programming
1 Data Mining

2 Some pandas Fundamentals
   - Series class
   - DataFrame class
   - Plotting

3 Financial Data Mining in Action

4 High-Frequency Financial Data

5 Some PyTables Fundamentals
   - Why PyTables and Basic Concepts
   - Introductory PyTables Example

6 Out-Of-Memory Monte Carlo Simulation
To begin with: What is Data Mining?

“The overall goal of the data mining process is to extract knowledge from an existing data set and transform it into a human-understandable structure for further use. Besides the raw analysis step, it involves database and data management aspects, data preprocessing, model and inference considerations, interestingness metrics, complexity considerations, post-processing of found structures, visualization, and online updating.”

Why Data Mining at all?

- Available data from public, commercial and in-house sources increases exponentially over time.
- To make profound strategic, operational and financial decisions, corporations must increasingly rely on diligent data mining.
- Therefore, efficient data management and analysis, i.e. data mining, becomes paramount in many industries, like financial services, utilities.
- From a more general point of view, efficient data management and analysis is essential in almost any area of software development and deployment.
- In addition, the majority of research fields nowadays requires the management and analysis of large data sets, like in physics or finance.
Data management is a huge industry, driven by ever increasing data volumes

Corporations invest huge amounts of money to manage data:²

- 100.000.000.000 bn USD spent in 2011 on data center infrastructure/hardware
- 24.000.000.000 bn USD spent in 2011 on database technology/software
- “The world’s No. 1 provider of data center real estate, Digital Realty Trust, is buying three properties near London for $1.1 billion.”³

²Source: Gartner Group; as reported in Bloomberg Businessweek, 2 July 2012, “Data Centers – Revenge of the Nerdiest Nerds”
³Source: Bloomberg Businessweek, 2 July 2012, “Bid & Ask”
Fast Data Mining =
Rapid Implementation
+ Quick Execution
Recent question in client project: “How beneficial are costly guarantees in unit-linked insurance policies from a policy holder perspective?”

Reframed question: “How often would a policy holder would have lost money with 10-/15-/20-years straight and mixed savings plans in popular stock indices?”

Solution: Concise Python script—using mainly pandas—to efficiently analyze the question for different parametrizations and with real, i.e. historic, financial market data.

Effort (for first prototype): Approximately one hour coding and testing (= playing); one hour for preparing a brief presentation with selected results (text + graphics).
Major problems in data management and analysis

- **sources**: data typically comes from different sources, like from the Web, from in-house databases or it is generated in-memory
- **formats**: data typically comes in different formats, like SQL databases/tables, Excel files, CSV files, NumPy arrays
- **structure**: data typically comes differently structured, like unstructured, simply indexed, hierarchically indexed, in table form, in matrix form, in multidimensional arrays
- **completeness**: real-world data typically comes in an incomplete form, i.e. there is missing data (e.g. along an index)
- **convention**: for some types of data there are many conventions with regard to formatting, like for dates and time
- **interpretation**: some data sets typically contain information that can be intelligently interpreted, like a time index
- **performance**: reading, streamlining, aligning, analyzing (large) data sets might be slow
What this talk is about

We will talk mainly about two libraries

- **pandas**: a library that conveniently enhances Python’s data management and analysis capabilities; its major focus are in-memory operations

- **PyTables**: a popular database which optimizes writing, reading and analyzing large data sets out-of-memory, i.e. on disk

We will illustrate their use mainly be the means of examples

- **Introductory pandas Example**—illustration of some fundamental pandas classes and their methods

- **Financial Data Mining in Action**—simple, but real world, example

- **High-Frequency Financial Data**—reading and analyzing high-frequency financial data with pandas

- **Introductory PyTables Example**—illustration of some fundamental pandas classes and their methods

- **Out-Of-Memory Monte Carlo Simulation**—implementing a Monte Carlo simulation with PyTables out-of-memory
Throughout the talk: Results matter more than Style

Bruce Lee—The Tao of Jeet Kune Do:

“There is no mystery about my style. My movements are simple, direct and non-classical. The extraordinary part of it lies in its simplicity. Every movement in Jeet Kune Do is being so of itself. There is nothing artificial about it. I always believe that the easy way is the right way.”

The Tao of My Python:

“There is no mystery about my style. My lines of code are simple, direct and non-classical. The extraordinary part of it lies in its simplicity. Every line of code in my Python is being so of itself. There is nothing artificial about it. I always believe that the easy way is the right way.”
Some pandas Fundamentals  
Series class

A fundamental class in pandas is the Series class (I)

- The Series class is explicitly designed to handle indexed (time) series\(^4\)
- If \(s\) is a Series object, \(s.index\) gives its index
- A simple example is \(s=\text{Series([1,2,3,4,5],index=[’a’,’b’,’c’,’d’,’e’])}\)

\begin{verbatim}
In [16]: s=Series([1,2,3,4,5],index=[’a’,’b’,’c’,’d’,’e’])

In [17]: s
Out[17]:
a   1
b   2
c   3
d   4
e   5

In [18]: s.index
Out[18]: Index([a, b, c, d, e], dtype=object)

In [19]: s.mean()
Out[19]: 3.0
In [20]:
\end{verbatim}

There are lots of useful methods in the Series class ...

\(^4\) The major pandas source is http://pandas.sourceforge.net
A fundamental class in pandas is the Series class (II)

- A major strength of pandas is the handling of time series data, i.e. data indexed by dates and times
- An simple example using the DateRange function shall illustrate the time series management

```python
In [3]: x=standard_normal(250)
In [4]: index=DateRange('01/01/2012',periods=len(x))
In [5]: s=Series(x,index=index)
In [6]: s
Out[6]:
2012-01-02  1.06959238875
2012-01-03  0.794515407245
2012-01-04  1.01590534404
2012-01-05  0.751618588824
...```
The offset parameter of the `DateRange` function allows flexible, automatic indexing

```python
In [33]: datetools.
datetools.bday                datetools.Minute
datetools.BDay               datetools.monthEnd
datetools.bmonthEnd          datetools.MonthEnd
datetools.BMonthEnd          datetools.normalize_date
datetools.bquarterEnd        datetools.ole2datetime
datetools.BQuarterEnd        datetools.OLE_TIME_ZERO
datetools.businessDay        datetools.parser
datetools.businessMonthEnd   datetools.relativeDelta
datetools.byearEnd           datetools.Second
datetools.BYearEnd           datetools.thisBMonthEnd
datetools.CacheableOffset    datetools.thisBQuarterEnd
datetools.calendar           datetools.thisMonthEnd
datetools.DateOffset         datetools.thisYearBegin
datetools.datetime           datetools.thisYearEnd
datetools.day                 datetools.Tick
datetools.format             datetools.timedelta
datetools.getOffset          datetools.to_datetime
datetools.getOffsetName      datetools.v
datetools.hasOffsetName       datetools.week
datetools.Hour               datetools.Week
datetools.i                   datetools.weekday
datetools.inferTimeRule      datetools.WeekOfMonth
datetools.isBMonthEnd         datetools.yearBegin
datetools.isBusinessDay      datetools.YearBegin
datetools.isMonthEnd         datetools.yearEnd
datetools.k                   datetools.YearEnd

In [33]: index=DateRange('01/01/2012',periods=len(x),offset=datetools.DateOffset(2))
```
Another fundamental class in pandas is DataFrame

- This class’s intellectual father is the `data.frame` class from the statistical language/package R
- The DataFrame class is explicitly designed to handle multiple, maybe hierarchically indexed (time) series
- The following example illustrates some convenient features of the DataFrame class, i.e. data alignment and handling of missing data

```python
In [35]: s=Series(standard_normal(4),index=['1','2','3','5'])
In [36]: t=Series(standard_normal(4),index=['1','2','3','4'])
In [37]: df=DataFrame({'s':s,'t':t})
In [38]: df['SUM']=df['s']+df['t']
In [39]: print df.to_string()
    s   t      SUM
  1 -0.125697 0.016357 -0.109340
  2  0.135457 -0.907421 -0.771964
  3  1.549149 -0.599659  0.949491
  4  NaN   0.734753    NaN
  5 -1.236310    NaN    NaN
In [40]: df['SUM'].mean()
Out[40]: 0.022728863312009556
```
The two main pandas classes have methods for easy plotting

- The Series and DataFrame classes have methods to easily generate plots
- The two major methods are plot and hist
- Again, an example shall illustrate the usage of the methods

In [54]: index=DateRange(start='1/1/2013',periods=250)
In [55]: x=standard_normal(250)
In [56]: y=standard_normal(250)
In [57]: df=DataFrame({'x':x,'y':y},index=index)
In [58]: df.cumsum().plot()
Out[58]: <matplotlib.axes.AxesSubplot at 0x3082c10>
In [59]: df['x'].hist()
Out[59]: <matplotlib.axes.AxesSubplot at 0x3468190>
In [60]:
The results of which can then be saved for further use.

Figure: Some example plots with pandas
The first ‘real’ example should give an impression of the efficiency of working with pandas

1. **Data gathering**: read historical quotes of the Apple stock (ticker AAPL) beginning with 01 January 2006 from finance.yahoo.com and store it in a pandas DataFrame object.

2. **Data analysis**: calculate the daily log returns (use the `shift` method of the pandas Series object) and generate a new column with the log returns in the DataFrame object.

3. **Plotting**: plot the log returns together with the daily Apple quotes into a single figure.

4. **Simulation**: simulate the Apple stock price development using the last Close quote as starting value and the historical yearly volatility of the Apple stock (short rate 2.5%)—the difference equation is given, for \( s = t - \Delta t \) and \( z_t \) standard normal, by

\[
S_t = S_s \cdot \exp((r - \sigma^2/2)\Delta t + \sigma\sqrt{\Delta t}z_t)
\]

5. **Option valuation**: calculate the value of a European call option with strike of 110% of the last Close quote and time-to-maturity of 1 year.

6. **Data storage**: save the pandas DataFrame to a PyTables/HDF5 database (use the HDFStore function).
1. Data Gathering

# Rapid Financial Engineering
# with pandas and PyTables
# RFE.py
#
# (c) Visixion GmbH
# Script for Illustration Purposes Only.
#
from pylab import *

# 1. Data Gathering

from pandas.io.data import *

AAPL=DataReader('AAPL', 'yahoo', start='01/01/2006')
2. Data Analysis (I)

```python
# 2. Data Analysis

from pandas import *

AAPL['Ret'] = log(AAPL['Close']/AAPL['Close'].shift(1))
```
2. Data Analysis

Python 2.7.3 (default, Apr 20 2012, 22:39:59)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.

```
>>> ================================ RESTART ================================
>>> Call Value 88.336
>>> print AAPL[-10:].to_string()

<table>
<thead>
<tr>
<th>Date</th>
<th>Open</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Volume</th>
<th>Adj Close</th>
<th>Ret</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-06-11</td>
<td>587.72</td>
<td>588.50</td>
<td>570.63</td>
<td>571.17</td>
<td>21094900</td>
<td>571.17</td>
<td>-0.015893</td>
</tr>
<tr>
<td>2012-06-12</td>
<td>574.46</td>
<td>576.62</td>
<td>566.70</td>
<td>576.16</td>
<td>15549300</td>
<td>576.16</td>
<td>0.008699</td>
</tr>
<tr>
<td>2012-06-13</td>
<td>574.52</td>
<td>578.48</td>
<td>570.38</td>
<td>572.16</td>
<td>10485000</td>
<td>572.16</td>
<td>-0.006967</td>
</tr>
<tr>
<td>2012-06-14</td>
<td>571.24</td>
<td>573.50</td>
<td>567.26</td>
<td>571.53</td>
<td>12341900</td>
<td>571.53</td>
<td>-0.001102</td>
</tr>
<tr>
<td>2012-06-15</td>
<td>571.00</td>
<td>574.62</td>
<td>569.55</td>
<td>574.13</td>
<td>11954200</td>
<td>574.13</td>
<td>0.004539</td>
</tr>
<tr>
<td>2012-06-18</td>
<td>587.96</td>
<td>589.89</td>
<td>570.37</td>
<td>585.78</td>
<td>15708100</td>
<td>585.78</td>
<td>0.020088</td>
</tr>
<tr>
<td>2012-06-19</td>
<td>583.40</td>
<td>590.00</td>
<td>583.10</td>
<td>587.41</td>
<td>12896200</td>
<td>587.41</td>
<td>0.002779</td>
</tr>
<tr>
<td>2012-06-20</td>
<td>588.21</td>
<td>589.25</td>
<td>580.80</td>
<td>585.74</td>
<td>12819400</td>
<td>585.74</td>
<td>-0.002847</td>
</tr>
<tr>
<td>2012-06-21</td>
<td>585.44</td>
<td>588.22</td>
<td>577.44</td>
<td>577.67</td>
<td>11655400</td>
<td>577.67</td>
<td>-0.013873</td>
</tr>
<tr>
<td>2012-06-22</td>
<td>579.04</td>
<td>582.19</td>
<td>575.42</td>
<td>582.10</td>
<td>10159700</td>
<td>582.10</td>
<td>0.007639</td>
</tr>
</tbody>
</table>
```

# 3. Plotting

```python
# 3. Plotting

subplot(211)
AAPL['Close'].plot()
ylabel('Index Level')

subplot(212)
AAPL['Ret'].plot()
ylabel('Log Returns')
```
3. Plotting (II)\textsuperscript{6}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{plot.png}
\caption{Graph showing stock price and log returns from 2007 to 2012.}
\end{figure}

# 4. Monte Carlo Simulation

```python
# 4. Monte Carlo Simulation

## Market Parameters
S0=AAPL['Close'][-1]  # End Value = Starting Value
vol=std(AAPL['Ret'])*sqrt(252)  # Historical Volatility
r=0.025  # Constant Short Rate

## Option Parameters
K=S0*1.1  # 10% OTM Call Option
T=1.0  # Maturity 1 Year

## Simulation Parameters
M=50; dt=T/M  # Time Steps
I=10000  # Simulation Paths

# Simulation
S=zeros((M+1,I)); S[0,:]=S0
for t in range(1,M+1):
    ran=standard_normal(I)
    S[t,:]=S[t-1,:]*exp((r-vol**2/2)*dt+vol*sqrt(dt)*ran)
```
5. Option Valuation

```python
# 5. Option Valuation
V0 = exp(-r*T)*sum(maximum(S[-1]-K,0))/I
print "Call Value %.3f" %V0
```
# 5. Data Storage

```python
h5file = HDFStore('AAPL.h5')
h5file['AAPL'] = AAPL
h5file.close()
```
The whole Python script

```python
... from pylab import *
# 1. Data Gathering
from pandas.io.data import *
AAPL=DataReader('AAPL', 'yahoo', start='01/01/2006')

# 2. Data Analysis
from pandas import *
AAPL['Ret']=log(AAPL['Close']/AAPL['Close'].shift(1))

# 3. Plotting
subplot(211)
AAPL['Close'].plot(); ylabel('Index Level')
subplot(212)
AAPL['Ret'].plot(); ylabel('Log Returns')

# 4. Monte Carlo Simulation
S0=AAPL['Close'][-1]
vol=std(AAPL['Ret'])*sqrt(252)
r=0.025; K=S0*1.1; T=1.0; M=50; dt=T/M; I=10000
S=zeros((M+1,I)); S[0,:]=S0
for t in range(1,M+1):
    ran=standard_normal(I)
    S[t,:]=S[t-1,:]*exp((r-vol**2/2)*dt+vol*sqrt(dt)*ran)

# 5. Option Valuation
V0=exp(-r*T)*sum(maximum(S[-1]-K,0))/I
print "Call Value %8.3f" %V0

# 6. Data Storage
h5file=HDFStore('AAPL.h5'); h5file['AAPL']=AAPL; h5file.close()
```
This example is about high-frequency stock data

- In this example, we are going to analyze intraday stock price data for Apple (ticker AAPL) and Google (ticker GOOG).
- Intraday data for US stocks is available from Netfonds (http://www.netfonds.no), a Norwegian online stock broker.
- We retrieve intraday data for both stocks for 22 June 2012 as a CSV file.
- The Apple stock price data file contains 16,465 rows; the Google stock price data file only 7,937 rows.
In the following, we will implement 8 typical data mining tasks

1. **data gathering**: retrieve data for Apple and Google from Web source and save as CSV file
2. **data reading**: read data from CSV files into two pandas DataFrame objects
3. **data pre-processing**: delete such rows with double time entries and use time data to generate time index for DataFrame objects
4. **data merging**: merge the bid quotes of both Apple and Google into a single DataFrame object
5. **data cleaning**: delete all quotes before 10 am on 22 June 2012
6. **data output**: print selected data for the new DataFrame object and plot the stock quotes
7. **data aggregation**: aggregate the tick data to average hourly quotes for both Apple and Google; print and plot the results
8. **data analysis**: get some statistics for tick data and hourly data (e.g. mean, min, max, correlation)
1. Data Gathering (I)

# Analyzing High-Frequency Stock Data
# with pandas
#
# (c) Visixion GmbH
# Script for illustration purposes only.
#
from pylab import *
from pandas import *
from urllib import urllib

# 1. Data Gathering
url='http://hopey.netfonds.no/posdump.php?date=20120622&
paper=%s.0&csv_format=csv'

urllibretrieve(url %'AAPL','AAPL.csv')
urllibretrieve(url %'GOOG','GOOG.csv')
1. Data Gathering (II)

Raw CSV data for Apple stock quotes:

<table>
<thead>
<tr>
<th>time</th>
<th>bid</th>
<th>bid_depth</th>
<th>bid_depth_total</th>
<th>offer</th>
<th>offer_depth</th>
<th>offer_depth_total</th>
</tr>
</thead>
<tbody>
<tr>
<td>20120622T100201</td>
<td>577.33</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>20120622T100231</td>
<td>577.33</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>20120622T100233</td>
<td>577.33</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>20120622T100236</td>
<td>577.33</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>20120622T100257</td>
<td>577.33</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>20120622T100258</td>
<td>577.33</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>20120622T100301</td>
<td>577.71</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>20120622T100316</td>
<td>577.71</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>20120622T100318</td>
<td>577.71</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>20120622T100334</td>
<td>578.11</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>20120622T100439</td>
<td>578.11</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>300</td>
<td>300</td>
</tr>
<tr>
<td>20120622T100445</td>
<td>578.11</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>20120622T100513</td>
<td>578.26</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>20120622T100533</td>
<td>578.26</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>20120622T100540</td>
<td>578.26</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>20120622T100557</td>
<td>578.26</td>
<td>400</td>
<td>400</td>
<td>579.71</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# 2. Data Reading

```python
AAPL = read_csv('AAPL.csv')
GOOG = read_csv('GOOG.csv')
```
# 3. Data Pre-Processing

```
AAPL = AAPL.drop_duplicates(cols='time')
GOOG = GOOG.drop_duplicates(cols='time')
for i in AAPL.index:
    AAPL['time'][i] = datetime.strptime(AAPL['time'][i], '%Y%m%d%H%M%S')
AAPL.index = AAPL['time']; del AAPL['time']
for i in GOOG.index:
    GOOG['time'][i] = datetime.strptime(GOOG['time'][i], '%Y%m%d%H%M%S')
GOOG.index = GOOG['time']; del GOOG['time']
```
3. Data Pre-Processing (II)

```python
print AAPL[['bid','offer']].ix[1000:1015].to_string()
```

<table>
<thead>
<tr>
<th>time</th>
<th>bid</th>
<th>offer</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-06-22 13:57:09</td>
<td>578.71</td>
<td>579.50</td>
</tr>
<tr>
<td>2012-06-22 13:57:16</td>
<td>578.71</td>
<td>579.48</td>
</tr>
<tr>
<td>2012-06-22 13:57:22</td>
<td>578.72</td>
<td>579.48</td>
</tr>
<tr>
<td>2012-06-22 13:57:47</td>
<td>578.73</td>
<td>579.48</td>
</tr>
<tr>
<td>2012-06-22 13:57:51</td>
<td>578.74</td>
<td>579.48</td>
</tr>
<tr>
<td>2012-06-22 13:57:52</td>
<td>578.75</td>
<td>579.48</td>
</tr>
<tr>
<td>2012-06-22 13:57:56</td>
<td>578.51</td>
<td>579.48</td>
</tr>
<tr>
<td>2012-06-22 13:57:57</td>
<td>578.53</td>
<td>579.48</td>
</tr>
<tr>
<td>2012-06-22 13:57:59</td>
<td>578.51</td>
<td>579.48</td>
</tr>
<tr>
<td>2012-06-22 13:58:20</td>
<td>578.51</td>
<td>579.46</td>
</tr>
<tr>
<td>2012-06-22 13:58:33</td>
<td>578.75</td>
<td>579.46</td>
</tr>
<tr>
<td>2012-06-22 13:58:36</td>
<td>578.76</td>
<td>579.46</td>
</tr>
<tr>
<td>2012-06-22 13:58:37</td>
<td>578.75</td>
<td>579.46</td>
</tr>
<tr>
<td>2012-06-22 13:58:51</td>
<td>578.76</td>
<td>579.46</td>
</tr>
<tr>
<td>2012-06-22 13:59:29</td>
<td>578.76</td>
<td>579.46</td>
</tr>
</tbody>
</table>
# 4. Data Merging

```python
DATA = DataFrame({'AAPL': AAPL['bid'], 'GOOG': GOOG['bid']})
```
# 5. Data Cleaning

```python
DATA = DATA[data.index > datetime(2012, 6, 22, 9, 59, 0)]
```
6. Data Output (I)

# 6. Data Output
print DATA.ix[:20].to_string()
DATA.plot(subplots=True)
6. Data Output (II)

```python
print AAPL[['bid','offer']].ix[1000:1015].to_string()
```

<table>
<thead>
<tr>
<th></th>
<th>AAPL</th>
<th>GOOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012-06-22 10:02:01</td>
<td>577.33</td>
<td>566.3</td>
</tr>
<tr>
<td>2012-06-22 10:02:31</td>
<td>577.33</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:02:33</td>
<td>577.33</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:02:36</td>
<td>577.33</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:02:57</td>
<td>577.33</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:02:58</td>
<td>577.33</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:03:01</td>
<td>577.71</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:03:16</td>
<td>577.71</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:03:18</td>
<td>577.71</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:03:34</td>
<td>578.11</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:04:39</td>
<td>578.11</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:04:45</td>
<td>578.11</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:05:13</td>
<td>578.26</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:05:33</td>
<td>578.26</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:05:36</td>
<td>578.26</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:05:40</td>
<td>578.26</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:05:57</td>
<td>578.26</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:06:00</td>
<td>578.26</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:06:07</td>
<td>578.26</td>
<td>NaN</td>
</tr>
<tr>
<td>2012-06-22 10:06:12</td>
<td>578.26</td>
<td>NaN</td>
</tr>
</tbody>
</table>
6. Data Output (III)\textsuperscript{7}

\textsuperscript{7}Quelle: http://finance.yahoo.com, 24. June 2012
# 7. Data Aggregation
by = lambda x: lambda y: getattr(y, x)
D = DATA.groupby([by('day'), by('hour')]).mean()
print D; D.plot()
## 7. Data Aggregation (II)

<table>
<thead>
<tr>
<th>key_0</th>
<th>key_1</th>
<th>AAPL</th>
<th>GOOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>10</td>
<td>578.688760</td>
<td>566.300000</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>578.758111</td>
<td>566.300000</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>579.211250</td>
<td>566.300000</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>578.739874</td>
<td>566.400000</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>578.973806</td>
<td>566.521786</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>578.547614</td>
<td>568.020159</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>577.727252</td>
<td>568.609922</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>577.405185</td>
<td>568.513652</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>577.299690</td>
<td>568.655632</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>577.302453</td>
<td>568.308739</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>579.156171</td>
<td>568.956426</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>580.020014</td>
<td>569.639033</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>582.090000</td>
<td>571.470000</td>
</tr>
</tbody>
</table>
7. Data Aggregation (III)
# 8. Data Analysis

print "\n\nSummary Statistics for Tick Data\n", DATA.describe()
print "\n\nCorrelation for Tick Data\n", DATA.corr()

print "\n\nSummary Statistics for Hourly Data\n", D.describe()
print "\n\nCorrelation for Hourly Data\n", D.corr()
8. Data Analysis (II)

<table>
<thead>
<tr>
<th>Summary Statistics for Tick Data</th>
<th>AAPL</th>
<th>GOOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>14104.000000</td>
<td>7595.000000</td>
</tr>
<tr>
<td>mean</td>
<td>578.320379</td>
<td>568.682132</td>
</tr>
<tr>
<td>std</td>
<td>1.191263</td>
<td>0.907999</td>
</tr>
<tr>
<td>min</td>
<td>575.410000</td>
<td>565.800000</td>
</tr>
<tr>
<td>25%</td>
<td>577.380000</td>
<td>568.180000</td>
</tr>
<tr>
<td>50%</td>
<td>578.400000</td>
<td>568.640000</td>
</tr>
<tr>
<td>75%</td>
<td>579.190000</td>
<td>569.220000</td>
</tr>
<tr>
<td>max</td>
<td>582.130000</td>
<td>571.470000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Correlation for Tick Data</th>
<th>AAPL</th>
<th>GOOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAPL</td>
<td>1.000000</td>
<td>0.735884</td>
</tr>
<tr>
<td>GOOG</td>
<td>0.735884</td>
<td>1.000000</td>
</tr>
</tbody>
</table>
8. Data Analysis (III)

### Summary Statistics for Hourly Data

<table>
<thead>
<tr>
<th></th>
<th>AAPL</th>
<th>GOOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>13.000000</td>
<td>13.000000</td>
</tr>
<tr>
<td>mean</td>
<td>578.763091</td>
<td>567.999642</td>
</tr>
<tr>
<td>std</td>
<td>1.300395</td>
<td>1.586889</td>
</tr>
<tr>
<td>min</td>
<td>577.299690</td>
<td>566.300000</td>
</tr>
<tr>
<td>25%</td>
<td>577.727252</td>
<td>566.400000</td>
</tr>
<tr>
<td>50%</td>
<td>578.739874</td>
<td>568.308739</td>
</tr>
<tr>
<td>75%</td>
<td>579.156171</td>
<td>568.655632</td>
</tr>
<tr>
<td>max</td>
<td>582.090000</td>
<td>571.470000</td>
</tr>
</tbody>
</table>

### Correlation for Hourly Data

<table>
<thead>
<tr>
<th></th>
<th>AAPL</th>
<th>GOOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAPL</td>
<td>1.000000</td>
<td>0.417359</td>
</tr>
<tr>
<td>GOOG</td>
<td>0.417359</td>
<td>1.000000</td>
</tr>
</tbody>
</table>
Major benefits and characteristics of PyTables

- **hierarchy**: structure your data in a hierarchical fashion (as with directories) and add user-specific data to each group/node
- **main objects**: PyTables knows tables as well as NumPy arrays; however, tables may also contain arrays
- **speed**: PyTables is optimized for I/O speed
- **operations**: it is ideally suited to do mathematical operations on your data
- **file**: it is file based and can be used on any notebook/desktop
- **concurrency**: only for reading operations, not really for writing
- **integration**: it integrates seamlessly with all kinds of Python applications
- **syntax**: the syntax is really Pythonic and quite close to standard NumPy syntax, e.g. with respect to indexing/slicing
- **relational database**: PyTables is NOT a replacement for a relational database (e.g. MySQL); it is a complementary work horse for computationally demanding tasks
Some of the most important PyTables functions/methods

- `openFile`: create new file or open existing file, like in
  ```python
  h5 = openFile('data.h5', 'w'); 'r' = read only, 'a' = read/write
  ```
- `.close()`: close database, like in `h5.close()`
- `h5.createGroup`: create a new group, as in
  ```python
  group = h5.createGroup(root, 'Name')
  ```
- `IsDescription`: class for column descriptions of tables, used as in:
  ```python
class Row(IsDescription):
    name = StringCol(20, pos=1)
    data = FloatCol(pos=2)
  ```
- `h5.createTable`: create new table, as in
  ```python
  tab = h5.createTable(group, 'Name', Row)
  ```
- `tab.iterrows()`: iterate over table rows
- `tab.where('condition')`: SQL-like queries with flexible conditions
- `tab.row`: return current/last row of table, used as in `r = tab.row`
- `row.append()`: append row to table, as in `r.append()`
- `tab.flush()`: flush table buffer to disk/file
- `h5.createArray`: create an array, as in
  ```python
  arr = h5.createArray(group, 'Name', zeros((10, 5))
  ```
Let's start with a simple example (I)

```
In [59]: from tables import *

In [60]: h5=openFile('Test_Data.h5','w')

In [61]: class Row(IsDescription):
    ....:    number = FloatCol(pos=1)
    ....:    sqrt   = FloatCol(pos=2)
    ....:

In [62]: tab=h5.createTable(h5.root,'Numbers',Row)

In [63]: tab
Out[63]:
/Numbers (Table(0,)) ''
description := {
  "number": Float64Col(shape=(), dflt=0.0, pos=0),
  "sqrt": Float64Col(shape=(), dflt=0.0, pos=1)}
byteorder := 'little'
chunkshape := (512,)

In [64]: r=tab.row

In [65]: for x in range(1000):
    ....:    r['number']=x
    ....:    r['sqrt']=sqrt(x)
    ....:    r.append()
    ....:
```
Let's start with a simple example (II)

```
In [66]: tab
Out[66]:
/Numbers (Table(0,)) ''
    description := {
        "number": Float64Col(shape=(), dflt=0.0, pos=0),
        "sqrt": Float64Col(shape=(), dflt=0.0, pos=1)}
    byteorder := 'little'
    chunkshape := (512,)

In [67]: tab.flush()

In [68]: tab
Out[68]:
/Numbers (Table(1000,)) ''
    description := {
        "number": Float64Col(shape=(), dflt=0.0, pos=0),
        "sqrt": Float64Col(shape=(), dflt=0.0, pos=1)}
    byteorder := 'little'
    chunkshape := (512,)
In [69]: tab[:5]
Out[69]:
array([(0.0, 0.0), (1.0, 1.0), (2.0, 1.4142135623730951),
       (3.0, 1.7320508075688772), (4.0, 2.0)],
      dtype=[('number', '<f8'), ('sqrt', '<f8')])

In [70]:
```
Let’s start with a simple example (III)

```python
In [7]: h5=openFile('Test_Data.h5','a')

In [8]: h5
Out[8]:
File(filename=Test_Data.h5, title='', mode='a', rootUEP='/', filters=Filters(complevel=0, shuffle=False, fletcher32=False))
/ (RootGroup) 
/Numbers (Table(1000,))
    description := {
        "number": Float64Col(shape=(), dflt=0.0, pos=0),
        "sqrt": Float64Col(shape=(), dflt=0.0, pos=1)}
    byteorder := 'little'
    chunkshape := (512,)

In [9]: tab=h5.root.Numbers

In [10]: tab[:5]['sqrt']
Out[10]: array([ 0. , 1. , 1.41421356, 1.73205081, 2. ])

In [11]: from pylab import *

In [12]: plot(tab[:]['sqrt'])
Out[12]: [<matplotlib.lines.Line2D at 0x7fe65cf12d10>]

In [13]: show()
```
You can also inspect the database graphically with **ViTables**

**Figure:** ViTables—a graphical interface to PyTables files

---

8 You find it under [http://vitables.berlios.de](http://vitables.berlios.de)
To illustrate PyTables’s math capabilities consider the following Python script (I)

```python
# Monte Carlo with Normal Arrays
# American Option with Least-Squares MCS
# LSM_Memory.py
#
# from pylan import *
from time import *
t0=time()
# Option Parameters
S0=36.;K=40.;r=0.06;T=1.0;vol=0.2
# MCS Parameters
M=200;I=400000;dt=T/M
# Arrays
ran=standard_normal((M+1,I))
S=zeros_like(ran)
V=zeros_like(ran)
```

Y. Hilpisch (Visixion GmbH)
To illustrate PyTables’s math capabilities consider the following Python script (II)

```python
# Simulation
S[0]=S0
for t in range(1,M+1):
    S[t]=S[t-1]*exp((r-0.5*(vol**2))*dt+vol*sqrt(dt)*ran[t])
# Valuation
df=exp(-r*dt)
h=max(smax(K-S,0)
V[-1,:]=h[-1,:]
for t in range(M-1,0,-1):
    rg = polyfit(S[t,:],V[t+1,:]*df,3)
    C = polyval(rg,S[t,:])
    V[t,:]=where(h[t,:]>C,h[t,:],V[t+1,:]*df)
V0=df*sum(V[1,:])/I
# Output
print "Option Value is %7.3f" %V0
print "Time in Seconds %7.3f" %((t1-t0))
```
Out-Of-Memory Monte Carlo Simulation

With **PyTables** you can use database objects like **NumPy arrays** (I)

```python
# Monte Carlo with PyTables Arrays -- Writing and Reading
# American Option with Least-Squares MCS
# LSM_PyTab.py
#
from pylab import *
from tables import *
from time import *
t0=time()
# Open HDF5 file for Array Storage
data=openFile('LSM_Data.h5','w')
# Option Parameters
S0=36.;K=40.;r=0.06;T=1.0;vol=0.2
# MCS Parameters
M=200;I=400000;dt=T/M
# Arrays
ran=data.createArray('/', 'ran', zeros((M+1,I),'f'),
    'Random Numbers')
for t in range(M+1):
    ran[t]=standard_normal(I)
S=data.createArray('/', 'S', zeros((M+1,I),'d'), 'Index Levels')
h=data.createArray('/', 'h', zeros((M+1,I),'d'), 'Inner Values')
V=data.createArray('/', 'V', zeros((M+1,I),'d'), 'Option Values')
C=data.createArray('/', 'C', zeros((I),'d'), 'Continuation Values')
```
With PyTables you can use database objects like NumPy arrays (II)

```python
# Simulation
S[0]=S0
for t in range(1,M+1):
    S[t]=S[t-1]*exp((-0.5*(vol**2))*dt+vol*sqrt(dt)*ran[t])
# Valuation
df=exp(-r*dt)
h=maximunm(K-S[:,:],0)
V[-1,:]=h[-1,:]
for t in range(M-1,0,-1):
    rg = polyfit(S[t,:],V[t+1,:]*df,3)
    C = polyval(rg,S[t,:])
    V[t,:] = where(h[t,:]>C,h[t,:],V[t+1,:]*df)
V0=df*sum(V[1,:])/I
# Output
data.close();t1=time()
print "Option Value is %7.3f" %V0
print "Time in Seconds %7.3f" %(t1-t0)
```
If you only read from a PyTables database, computations are quite fast

```python
# Monte Carlo with PyTables Array -- Reading from File
# American Option with Least-Squares MCS
# LSM_PyTab_RO.py
#
from pylab import *
from tables import *
from time import *
from LSM_PyTab import K,r,T,M,I,dt,df
t0=time()
# Open HDF5 file for Array Reading
data=openFile('LSM_Data.h5','a')
S=data.root.S
h=data.root.h
V=data.root.V
C=data.root.C
# Valuation
for t in range(M-1,0,-1):
    rg = polyfit(S[t,:],V[t+1,:]*df,3)
    C = polyval(rg,S[t,:])
    V[t,:] = where(h[t,:]>C,h[t,:],V[t+1,:]*df)
V0=df*sum(V[1,:])/I
# Output
data.close();t1=time()
print "Option Value is %.3f" %V0
print "Time in Seconds %.3f" %(t1-t0)
```
In addition, recent versions of PyTables support improved math capabilities

- **NumPy**: fast in-memory array manipulations and operations
- **numexpr**: (memory) improved array operations for faster execution
- **tables.Expr**: combining the strengths of numexpr with PyTables’ I/O capabilities
A simple script illustrates how to apply the three alternatives

```python
# Evaluating Complex Expressions
# Expr_Comparison.py

from pylab import *
from numexpr import *
from tables import *

# Assumption and Input Data

expr = '0.3*x**3+2.0*x**2+log(abs(x))-3'
new = True
size = 10E5

x = standard_normal(size)

if new == True:
    h5 = openFile('expr.h5', 'w')
    h5.createArray(h5.root, 'x', x)
    h5.close()

# Three Evaluation Routines

def num_py():
    y = eval(expr)
    return y

def num_ex():
    y = evaluate(expr)
    return y

def tab_ex():
    h5 = openFile('expr.h5', 'r')
    x = h5.root.x
    ex = Expr(expr)
    y = ex.eval()
    h5.close()
    return y
```

Y. Hilpisch (Visixion GmbH)
Interestingly, reading from HDF5 file and using Expr is faster than pure NumPy.

In [43]: %run Expr_Comparison.py

In [44]: %timeit num_py()
10 loops, best of 3: 177 ms per loop

In [45]: %timeit num_ex()
100 loops, best of 3: 12.6 ms per loop

In [46]: %timeit tab_ex()
10 loops, best of 3: 33.3 ms per loop

In [47]: size
Out[47]: 1000000.0

In [48]:
Visixion’s experience with Python

- **DEXISION**: full-fledged Derivatives Analytics suite implemented in Python and delivered On Demand (since 2006, www.dexision.com)
- **research**: Python used to implement a number of numerical research projects (see www.visixion.com)
- **trainings**: Python trainings with focus on Finance for clients from the financial services industry
- **client projects**: Python used to implement client specific financial applications
- **teaching**: Python used to implement and illustrate financial models in derivatives course at Saarland University (see Course Web Site)
- **talks**: we have given a number of talks at Python conferences about the use of Python for Finance
- **book**: Python used to illustrate financial models in our recent book “Derivatives Analytics with Python—Market-Based Valuation of European and American Stock Index Options”
Contact

Dr. Yves J. Hilpisch
Visixion GmbH
Rathausstrasse 75-79
66333 Voelklingen
Germany
www.visixion.com
www.dxevo.com
www.dexision.com
E contact@visixion.com
T +49 6898 932350
F +49 6898 932352