Computational Finance

AMSTERDAM BUSINESS SCHOOL

Business

Dealing with Data

More Datatypes

NumPy Arrays

- The most fundamental data type in scientific Python is `ndarray`, provided by the NumPy package (user guide).
- An array is similar to a `list`, except that
    - it can have more than one dimension;
    - its elements are homogeneous (they all have the same type).
- NumPy provides a large number of functions (*ufuncs*) that operate elementwise on arrays. This allows *vectorized* code, avoiding loops (which are slow in Python).

Constructing Arrays

- Arrays can be constructed using the `array` function which takes sequences (e.g, lists) and converts them into arrays. The data type is inferred automatically or can be specified.

In [2]:
```
import numpy as np
a = np.array([1, 2, 3, 4])
a.dtype
```
Out[2]:
```
dtype('int64')
```

In [3]:
```
a = np.array([1, 2, 3, 4], dtype='float64')  #or np.array([1., 2., 3., 4.])
a.dtype
```
Out[3]:
```
dtype('float64')
```

- NumPy uses C++ data types which differ from Python's (though `float64` is equivalent to Python's `float`).

- Nested lists result in multidimensional arrays. We won't need anything beyond two-dimensional (i.e., a matrix or table).

In [4]:
```
a = np.array([[1., 2.], [3., 4.]]); a
```

Out[4]:
```
array([[ 1.,  2.],
       [ 3.,  4.]])
```

In [5]:
```
a.ndim  #Number of dimensions.
```

Out[5]:
```
2
```

In [6]:
```
a.shape #Number of rows and columns.
```

Out[6]:
```
(2, 2)
```

- Other functions for creating arrays include:

In [7]:
```
np.eye(3, dtype='float64')  #Identity matrix. float64 is the default dtype and can be omitted
```

Out[7]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

In [8]:
```
np.ones([2, 3])  #There's also np.zeros, and np.empty (which results in an uninitialized array).
```

Out[8]:
```
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

In [9]:
```
np.arange(0, 10, 2)  #Like range, but creates an array instead of a list.
```

Out[9]:
```
array([0, 2, 4, 6, 8])
```

In [10]:
```
np.linspace(0, 10, 5) #5 equally spaced points between 0 and 10
```

Out[10]:
```
array([  0. ,   2.5,   5. ,   7.5,  10. ])
```

Indexing

- Indexing and slicing operations are similar to lists:

In [11]:
```
a = np.array([[1., 2.], [3., 4.]])
a[0, 0] #Element [row, column]. Equivalent to a[0][0].
```

Out[11]:
```
1.0
```

In [12]:
```
b = a[:, 0]; b #Entire first column. Note that this yields a 1-dimensional array (vector), not a matrix with o
```

Out[12]:
```
array([ 1.,  3.])
```

- Slicing returns *views* into the original array (unlike slicing lists):

In [13]:
```
b[0] = 42
```

In [14]:
```
a
```

Out[14]:
```
array([[ 42.,   2.],
       [  3.,   4.]])
```

- Apart from indexing by row and column, arrays also support *Boolean* indexing:

```
In [15]: a = np.arange(10); a
```

```
Out[15]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [16]: ind = a < 5; ind
```

```
Out[16]: array([ True,  True,  True,  True,  True, False, False, False, False, False], dtype=bool)
```

```
In [17]: a[ind]
```

```
Out[17]: array([0, 1, 2, 3, 4])
```

Concatenation and Reshaping

- To combine two arrays in NumPy, use `concatenate` or `stack`:

In [18]:
```
a = np.array([1, 2, 3]); b = np.array([4, 5, 6])
```

In [19]:
```
c = np.concatenate([a, b]); c  #Concatenate along an existing axis.
```

Out[19]:
```
array([1, 2, 3, 4, 5, 6])
```

In [20]:
```
d = np.stack([a, b]); d  #Concatenate along a new axis (e.g., vectors to matrix).
```

Out[20]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

- `reshape(n, m)` changes the shape of an array into `(n,m)`, taking the elements row-wise. A dimension given as `-1` will be computed automatically.

In [21]:
```
d.reshape(3,-1)  #3 rows, number of columns determined automatically.
```

Out[21]:
```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Arithmetic and ufuncs

- NumPy ufuncs are functions that operate elementwise:

```
In [22]: a = np.arange(1, 5); np.sqrt(a)
```

```
Out[22]: array([ 1.        ,  1.41421356,  1.73205081,  2.        ])
```

- Other useful ufuncs are `exp`, `log`, `abs`, and `sqrt`.
- Basic arithmetic on arrays works elementwise:

```
In [23]: a = np.arange(1, 5); b = np.arange(5, 9); a, b, a+b, a-b, a/b.astype(float)
```

```
Out[23]: (array([1, 2, 3, 4]),
          array([5, 6, 7, 8]),
          array([ 6,  8, 10, 12]),
          array([-4, -4, -4, -4]),
          array([ 0.2       ,  0.33333333,  0.42857143,  0.5       ]))
```

Broadcasting

- Operations between scalars and arrays are also supported:

In [24]:
```
np.array([1, 2, 3, 4]) + 2
```

Out[24]:
```
array([3, 4, 5, 6])
```

- This is a special case of a more general concept known as *broadcasting*, which allows operations between arrays of different shapes.
- NumPy compares the shapes of two arrays dimension-wise. It starts with the trailing dimensions, and then works its way forward. Two dimensions are compatible if
    - they are equal, or
    - one of them is 1 (or not present).
- In the latter case, the singleton dimension is "stretched" to match the larger array.

- Example:

In [25]:
```
x = np.arange(6).reshape((2, 3)); x  #x has shape (2,3).
```

Out[25]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [26]:
```
m = np.mean(x, axis=0); m  #m has shape (3,).
```

Out[26]:
```
array([ 1.5,  2.5,  3.5])
```

In [27]:
```
x-m  #the trailing dimension matches, and m is stretched to match the 2 rows of x.
```

Out[27]:
```
array([[-1.5, -1.5, -1.5],
       [ 1.5,  1.5,  1.5]])
```

- NumPy's `newaxis` feature is sometimes useful to enable broadcasting. It introduces a new dimension of length 1; e.g, it can turn a vector (1d array) into a matrix with a single row or column (2d array). Example:

In [28]:
```
u = np.array([1, 2, 3])  #u has shape (3,).
v = np.array([4, 5, 6, 7])  #v has shape (4,).
w = u[:, np.newaxis]  #w has shape (3, 1); a matrix with 3 rows and one column.
w*v  #(3, 1) x (4,); starting from the back, 4 and 1 are compatible, and 3 and 'missing' are too -> (3, 4).
```

Out[28]:
```
array([[ 4,  5,  6,  7],
       [ 8, 10, 12, 14],
       [12, 15, 18, 21]])
```

- In this particular case, the same result could have been obtained by taking the outer product of u and v (in mathematical notation, uv'):

In [29]:
```
np.outer(u, v)
```

Out[29]:
```
array([[ 4,  5,  6,  7],
       [ 8, 10, 12, 14],
       [12, 15, 18, 21]])
```

Array Reductions

- *Array reductions* are operations on arrays that return scalars or lower-dimensional arrays, such as the `mean` function used above.
- They can be used to summarize information about an array, e.g., compute the standard deviation:

```
In [30]: a = np.random.randn(300, 3)  #Create a 300x3 matrix of standard normal variates.
         a.std(axis=0)  #or np.std(a, axis=0)

Out[30]: array([ 1.01486449,  1.04777986,  1.01134656])
```

- By default, reductions operate on the *flattened* array (i.e., on all the elements). For row- or columnwise operation, the `axis` argument has to be given.
- Other useful reductions are `sum`, `median`, `min`, `max`, `argmin`, `argmax`, `any`, and `all` (see help).

Saving Arrays to Disk

- There are several ways to save an array to disk:

In [31]:
```
np.save('myfile.npy', a)  #Save `a` as a binary .npy file.
```

In [32]:
```
import os
print(os.listdir('.'))
```

```
['week5.ipynb', 'week1.ipynb', 'README.md', 'week2.ipynb', 'pdf', 'week4.ipynb', 'myfile.npy', 'week3.ipynb',
'.ipynb_checkpoints', 'img']
```

In [33]:
```
b = np.load('myfile.npy')  #Load the data into variable b.
os.remove('myfile.npy')  #Clean up.
```

In [34]:
```
np.savetxt('myfile.csv', a, delimiter=',')  #Save `a` as a CSV file (comma seperated values, can be read by MS
```

In [35]:
```
b = np.loadtxt('myfile.csv', delimiter=',')  #Load data into `b`.
os.remove('myfile.csv')
```

Pandas Dataframes

Introduction to Pandas

- `pandas` (from *pan*el *da*ta) is another fundamental package in the SciPy stack (user guide).
- It provides a number of datastructures (*series*, *dataframes*, and *panels*) designed for storing observational data, and powerful methods for manipulating (*munging*, or *wrangling*) these data.
- It is usually imported as pd:

In [36]:
```
import pandas as pd
```

Series

- A pandas `Series` is essentially a NumPy array with an associated index:

In [37]:
```
pop = pd.Series([5.7, 82.7, 17.0], name='Population'); pop  #The descriptive name is optional.
```

Out[37]:
```
0     5.7
1    82.7
2    17.0
Name: Population, dtype: float64
```

- The difference is that the index can be anything, not just a list of integers:

In [38]:
```
pop.index=['DK', 'DE', 'NL']
```

- The index can be used for indexing (duh...):

In [39]:
```
pop['NL']
```

Out[39]:
```
17.0
```

- NumPy's `ufuncs` preserve the index when operating on a `Series`:

In [40]:
```
gdp = pd.Series([3494.898, 769.930], name='Nominal GDP in Billion USD', index=['DE', 'NL']); gdp
```

Out[40]:
```
DE    3494.898
NL     769.930
Name: Nominal GDP in Billion USD, dtype: float64
```

In [41]:
```
gdp/pop
```

Out[41]:
```
DE    42.259952
DK          NaN
NL    45.290000
dtype: float64
```

- One advantage of a `Series` compared to NumPy arrays is that they can handle missing data, represented as `NaN` (not a number).

Dataframes

- A `DataFrame` is a collection of `Series` with a common index (which labels the rows).

In [42]:
```
data = pd.concat([gdp, pop], axis=1); data  #Concatenate two Series to a DataFrame.
```

Out[42]:

	Nominal GDP in Billion USD	Population
DE	3494.898	82.7
DK	NaN	5.7
NL	769.930	17.0

- Columns are indexed by column name:

In [43]:
```
data.columns
```

Out[43]:
```
Index([u'Nominal GDP in Billion USD', u'Population'], dtype='object')
```

In [44]:
```
data['Population']  #data.Population works too
```

Out[44]:
```
DE    82.7
DK     5.7
NL    17.0
Name: Population, dtype: float64
```

- Rows are indexed with the `loc` method (note: the `ix` method listed in the book (p. 139) is deprecated):

In [45]:
```
data.loc['NL']
```

Out[45]:
```
Nominal GDP in Billion USD    769.93
Population                     17.00
Name: NL, dtype: float64
```

- Unlike arrays, dataframes can have columns with different datatypes.
- There are different ways to add columns. One is to just assign to a new column:

In [46]:
```
data['Language'] = ['German', 'Danish', 'Dutch']  #Add a new column from a list.
```

- Another is to use the `join` method:

In [47]:
```
s = pd.Series(['EUR', 'DKK', 'EUR', 'GBP'], index=['NL', 'DK', 'DE', 'UK'], name='Currency')
data.join(s)  #Add a new column from a series or dataframe.
```

Out[47]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
DK	NaN	5.7	Danish	DKK
NL	769.930	17.0	Dutch	EUR

- Notes:
  - The entry for 'UK' has disappeared. Pandas takes the *intersection* of indexes ('inner join') by default.
  - The returned series is a temporary object. If we want to modify `data`, we need to assign to it.

- To take the union of indexes ('outer join'), pass the keyword argument `how='outer'`:

In [48]:
```
data = data.join(s, how='outer'); data  #Assignment to store the modified frame.
```

Out[48]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
DK	NaN	5.7	Danish	DKK
NL	769.930	17.0	Dutch	EUR
UK	NaN	NaN	NaN	GBP

- The `join` method is in fact a convenience method that calls `pd.merge` under the hood, which is capable of more powerful SQL style operations.

- To add rows, use `loc` or `append`:

In [49]:
```
data.loc['AT'] = [386.4, 8.7, 'German', 'EUR']  #Add a row with index 'AT'.
s = pd.DataFrame([[511.0, 9.9, 'Swedish', 'SEK']], index=['SE'], columns=data.columns)
data = data.append(s)  #Add a row by appending another dataframe. May create duplicates.
data
```

Out[49]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
DK	NaN	5.7	Danish	DKK
NL	769.930	17.0	Dutch	EUR
UK	NaN	NaN	NaN	GBP
AT	386.400	8.7	German	EUR
SE	511.000	9.9	Swedish	SEK

- The `dropna` method can be used to delete rows with missing values:

In [50]:
```
data = data.dropna(); data
```

Out[50]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
NL	769.930	17.0	Dutch	EUR
AT	386.400	8.7	German	EUR
SE	511.000	9.9	Swedish	SEK

- Useful methods for obtaining summary information about a dataframe are `mean`, `std`, `info`, `describe`, `head`, and `tail`.

In [51]:
```
data.describe()
```

Out[51]:

	Nominal GDP in Billion USD	Population
count	4.000000	4.000000
mean	1290.557000	29.575000
std	1478.217475	35.605559
min	386.400000	8.700000
25%	479.850000	9.600000
50%	640.465000	13.450000
75%	1451.172000	33.425000
max	3494.898000	82.700000

In [52]:
```
data.head()  #Show the first few rows. data.tail shows the last few.
```

Out[52]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
NL	769.930	17.0	Dutch	EUR
AT	386.400	8.7	German	EUR
SE	511.000	9.9	Swedish	SEK

- To save a dataframe to disk as a csv file, use

In [53]:
```
data.to_csv('myfile.csv')  #To_excel exists as well.
```

In [54]:
```
with open('myfile.csv', 'r') as f:
    print(f.read())
```

```
,Nominal GDP in Billion USD,Population,Language,Currency
DE,3494.898,82.7,German,EUR
NL,769.93,17.0,Dutch,EUR
AT,386.4,8.7,German,EUR
SE,511.0,9.9,Swedish,SEK
```

- To load data into a dataframe, use `pd.read_csv` (see Table 6.6 in the book):

In [55]:
```
pd.read_csv('myfile.csv', index_col=0)
```

Out[55]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
NL	769.930	17.0	Dutch	EUR
AT	386.400	8.7	German	EUR
SE	511.000	9.9	Swedish	SEK

In [56]:
```
os.remove('myfile.csv')  #Clean up.
```

- Other, possibly more efficient, methods exist; see Chapter 7 of Hilpisch (2014).

Working with Time Series

Data Types

- Different data types for representing times and dates exist in Python.
- The most basic one is `datetime` from the eponymous package, and also accesible from Pandas:

In [57]:
```
pd.datetime.today()
```
Out[57]:
```
datetime.datetime(2017, 11, 19, 14, 17, 10, 832751)
```

- `datetime` objects can be created from strings using `strptime` and a format specifier:

In [58]:
```
pd.datetime.strptime('2017-03-31', '%Y-%m-%d')
```
Out[58]:
```
datetime.datetime(2017, 3, 31, 0, 0)
```

- Pandas uses `Timestamps` instead of `datetime` objects. Unlike timestamps, they store frequency and time zone information. The two can mostly be used interchangeably. See Appendix C for details.

In [59]:
```
pd.Timestamp('2017-03-31')
```
Out[59]:
```
Timestamp('2017-03-31 00:00:00')
```

- A time series is a `Series` with a special index, called a `DatetimeIndex`; essentially an array of `Timestamps`.
- It can be created using the `date_range` function; see Tables 6.2 and 6.3.

In [60]:
```
myindex = pd.date_range(end=pd.Timestamp.today(), normalize=True, periods=100, freq='B')
P = 20+np.random.randn(100).cumsum()  #Make up some share prices.
aapl = pd.Series(P, name="AAPL", index=myindex)
aapl.tail()
```
Out[60]:
```
2017-11-13    44.530210
2017-11-14    43.766867
2017-11-15    44.185038
2017-11-16    44.835533
2017-11-17    46.538283
Freq: B, Name: AAPL, dtype: float64
```

- As a convenience, Pandas allows indexing timeseries with date strings:

In [61]:
```
aapl['10/5/2017']
```

Out[61]:
```
33.100060364166268
```

In [62]:
```
aapl['10/5/2017':'10/10/2017']
```

Out[62]:
```
2017-10-05    33.100060
2017-10-06    33.773791
2017-10-09    34.235886
2017-10-10    35.551414
Freq: B, Name: AAPL, dtype: float64
```

Financial Returns

- We mostly work with returns rather than prices, because their statistical properties are more desirable (stationarity).
- There exist two types of returns: *simple returns* $R_t \equiv (P_t - P_{t-1})/P_{t-1}$, and *log returns* $r_t \equiv \log(P_t/P_{t-1}) = \log P_t - \log P_{t-1}$.
- Log returns are usually preferred, though the difference is typically small.
- To convert from prices to returns, use the `shift(k)` method, which lags by k periods (or leads if $k < 0$).

In [63]:
```
aapl=np.log(aapl)-np.log(aapl).shift(1)
aapl.tail()
```

Out[63]:
```
2017-11-13   -0.006957
2017-11-14   -0.017291
2017-11-15    0.009509
2017-11-16    0.014615
2017-11-17    0.037274
Freq: B, Name: AAPL, dtype: float64
```

- Note: for some applications (e.g., CAPM regressions), *excess returns* $r_t - r_{f,t}$ are required, where $r_{f,t}$ is the return on a "risk-free" investment.
- These are conveniently constructed as follows: suppose you have a data frame containing raw returns for a bunch of assets:

In [64]:
```
P = 20+np.random.randn(100).cumsum()  #Some more share prices.
rf = 1+np.random.randn(100)/100  #And a yield.
msft = pd.Series(P,  name="MSFT", index=myindex)
msft = np.log(msft)-np.log(msft).shift(1)
returns=pd.concat([aapl, msft], axis=1)
returns.tail()
```

Out[64]:

	AAPL	MSFT
2017-11-13	-0.006957	0.052854
2017-11-14	-0.017291	-0.083161
2017-11-15	0.009509	0.026796
2017-11-16	0.014615	0.084122
2017-11-17	0.037274	0.228075

- Then the desired operation can be expressed as

In [65]:
```
excess_returns=returns.sub(rf, axis='index')  #Subtract series rf from all columns.
```

Fetching Data

- `pandas_datareader` makes it easy to fetch data from the web (user guide).
- It is no longer included in pandas, so we need to install it.

In [66]:
```
#uncomment the next line to install.
#!conda install -y pandas-datareader
import pandas_datareader.data as web  #Not 'import pandas.io.data as web' as in the book.
```

In [67]:
```
start = pd.datetime(2010, 1, 1)
end = pd.datetime.today()
p = web.DataReader("^GSPC", 'yahoo', start, end)  #S&P500
p.tail()
```

Out[67]:

	Open	High	Low	Close	Adj Close	Volume
Date						
2017-11-13	2576.530029	2587.659912	2574.479980	2584.840088	2584.840088	3402930000
2017-11-14	2577.750000	2579.659912	2566.560059	2578.870117	2578.870117	3641760000
2017-11-15	2569.449951	2572.840088	2557.449951	2564.620117	2564.620117	3558890000
2017-11-16	2572.949951	2590.090088	2572.949951	2585.639893	2585.639893	3312710000
2017-11-17	2582.939941	2583.959961	2577.620117	2578.850098	2578.850098	3300160000

Regression Analysis

- Like in the book, we analyze the *leverage effect*: negative stock returns decrease the value of the equity and hence increase debt-to-equity, so the cashflow to shareholders as residual claimants becomes more risky; i.e., volatility increases.
- Hilpisch uses the VSTOXX index. Here, we use the VIX, which measures the volatility of the S&P500 based on implied volatilities from the option market.
- We already have data on the S&P500. We'll convert them to returns and do the same for the VIX. We'll store everything in a dataframe `df`.

In [68]:
```
df = pd.DataFrame()  #If in doubt, start with an empty DataFrame.
df['SP500'] = np.log(p['Adj Close']) - np.log(p['Adj Close']).shift(1)  #Make sure there's no ^ in the column
p = web.DataReader("^VIX", 'yahoo', start, end)
df['VIX'] = np.log(p['Adj Close'])-np.log(p['Adj Close']).shift(1)
df.tail()
```

Out[68]:

	SP500	VIX
Date		
2017-11-13	0.000983	0.018430
2017-11-14	-0.002312	0.007796
2017-11-15	-0.005541	0.124757
2017-11-16	0.008163	-0.110196
2017-11-17	-0.002629	-0.028462

- Next, we run an OLS regression of the VIX returns on those of the S&P.
- Note that this functionality has been moved from Pandas to the `statsmodels` package, so we have to use a different incantation from the one in the book.
- Also, we will use a different interface (API) which allows us to specify regressions using R-style formulas (user guide).
- We will use heteroskedasiticy and autocorrelation consistent (HAC) standard errors.

In [69]:
```python
import statsmodels.formula.api as smf
model = smf.ols('VIX ~ SP500', data=df)
result = model.fit(cov_type="HAC", cov_kwds={'maxlags':5})
print(result.summary2())
```

```
                   Results: Ordinary least squares
==================================================================
Model:              OLS              Adj. R-squared:     0.653
Dependent Variable: VIX              AIC:                -6765.5638
Date:               2017-11-19 14:17 BIC:                -6754.3771
No. Observations:   1985             Log-Likelihood:     3384.8
Df Model:           1                F-statistic:        843.5
Df Residuals:       1983             Prob (F-statistic): 8.03e-155
R-squared:          0.653            Scale:              0.0019358
-------------------------------------------------------------------
                Coef.   Std.Err.     z      P>|z|    [0.025   0.975]
-------------------------------------------------------------------
Intercept       0.0024    0.0009    2.6751  0.0075   0.0006   0.0042
SP500          -6.4530    0.2222  -29.0422  0.0000  -6.8885  -6.0175
------------------------------------------------------------------
Omnibus:             193.539      Durbin-Watson:         2.119
Prob(Omnibus):       0.000        Jarque-Bera (JB):      1143.001
Skew:                0.238        Prob(JB):              0.000
Kurtosis:            6.687        Condition No.:         107
==================================================================
```

- Conclusion: We indeed find a significant negative effect of the index returns, confirming the existence of the leverage effect.
- Note: for a regression without an intercept, we would use `model = smf.ols('VIX ~ -1+SP500', data=df)`.
- The `result` object has useful methods and variables:

In [70]:
```
print(result.f_test('SP500=0, Intercept=0'))
```

```
<F test: F=array([[ 434.84895607]]), p=2.56661684112e-157, df_denom=1983, df_num=2>
```

In [72]:
```
result.params
```

Out[72]:
```
Intercept    0.002403
SP500       -6.453038
dtype: float64
```