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 (https://docs.scipy.org/doc/numpy/user/index.html)).
- An array is similar to a `list`, except that
  - it can have more than one dimension;
  - its elements are homogenous (they all have the same type).
- NumPy provides a large number of functions (*ufuncs*) that operate elementwise on arrays. 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 (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:

In [7]:
```
np.eye(3, dtype='float64') #identity matrix. float64 is the default dtype and can b
e 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 result in an uninitializ
ed 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] #indexing [row, column]. Equivalent to b[0][0]
```

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

In [12]:
```
b=a[:, 0]; b #First column. Note that this yields a 1-dimensional array, not a matr
ix
```

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], dt
ype=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`, `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
```

Out[23]:
```
(array([1, 2, 3, 4]),
 array([5, 6, 7, 8]),
 array([ 6,  8, 10, 12]),
 array([-4, -4, -4, -4]),
 array([0, 0, 0, 0]))
```

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 element-wise. It starts with the trailing dimensions, and 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 (2d array) with a single row or column. 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 'miss
ing' 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([ 0.91547487,  1.05269445,  0.97301281])
```

- By default, reductions work on a flattened version of the array. 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('.'))
```

```
['week1.ipynb', 'README.md', 'week2.ipynb', '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 as CSV file (comma seperated value
s, can be read by MS Excel)
```

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

Pandas Dataframes

Introduction to Pandas

- `pandas` (from *pa*nel *da*ta) is another fundamental package in the SciPy stack ([user guide (http://pandas.pydata.org/pandas-docs/stable/overview.html)](http://pandas.pydata.org/pandas-docs/stable/overview.html)).
- 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 o
ptional
```

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 ufuncs operate on series and preserve the index:

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 `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 series
```

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='Cur
rency')
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 duplicate
s
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 datatypes 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, 10, 26, 19, 40, 49, 68706)
```

- `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`.
- 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]:
```
2017-10-20    20.983895
2017-10-23    21.566867
2017-10-24    20.484217
2017-10-25    22.493990
2017-10-26    22.226591
Freq: B, Name: AAPL, dtype: float64
```

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

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

Out[61]:
```
24.746785943720582
```

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

Out[62]:
```
2017-10-05    24.746786
2017-10-06    24.240927
2017-10-09    24.844183
2017-10-10    24.112079
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 `shift(k)` method which lags by k periods (or leads if $k < 0$).

In [63]:

```
ret=np.log(aapl)-np.log(aapl).shift(1)
ret.tail()
```

Out[63]:

```
2017-10-20   -0.003872
2017-10-23    0.027403
2017-10-24   -0.051503
2017-10-25    0.093593
2017-10-26   -0.011959
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)
returns=pd.concat([aapl, msft], axis=1)
returns.tail()
```

Out[64]:

	AAPL	MSFT
2017-10-20	20.983895	35.293386
2017-10-23	21.566867	34.630556
2017-10-24	20.484217	37.298781
2017-10-25	22.493990	37.233592
2017-10-26	22.226591	35.273647

- 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 (http://pandas-datareader.readthedocs.io/en/latest/remote_data.html)).
- It is no longer included in pandas, so we need to install it.

In [66]:
```
#uncomment the next line to install. (Note: ! executes shell commands)
#!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
Date					
2017-10-20	2567.560059	2575.439941	2567.560059	2575.209961	257
2017-10-23	2578.080078	2578.290039	2564.330078	2564.979980	256
2017-10-24	2568.659912	2572.179932	2565.580078	2569.129883	256
2017-10-25	2566.520020	2567.399902	2544.000000	2557.149902	255
2017-10-26	2560.080078	2567.070068	2559.800049	2564.149902	256

Regression Analysis

- Like in the book, we analyze the *leverage effect*: negative stock returns decrease the value of equity and hence increase debt-to-equity, so 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()
df['SP500']=np.log(p['Adj Close'])-np.log(p['Adj Close']).shift(1) #make sure there
's no ^ in the column name
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		
2017-10-20	0.005104	-0.007992
2017-10-23	-0.003980	0.104658
2017-10-24	0.001617	0.008097
2017-10-25	-0.004674	0.006253
2017-10-26	0.002734	-0.021603

- 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 (http://www.statsmodels.org/stable/index.html), so we have to use a different incantation than in the book.
- Also, we will use a different interface (API) which allows us to specify regressions using R-style formulas (user guide (http://www.statsmodels.org/stable/example_formulas.html)).
- 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:     0.654
Dependent Variable: VIX              AIC:                -6715.8073
Date:               2017-10-26 19:40 BIC:                -6704.6367
No. Observations:   1969             Log-Likelihood:     3359.9
Df Model:           1                F-statistic:        842.5
Df Residuals:       1967             Prob (F-statistic): 1.78e-154
R-squared:          0.654            Scale:              0.0019311
--------------------------------------------------------------------
                Coef.   Std.Err.     z      P>|z|    [0.025   0.975]
--------------------------------------------------------------------
Intercept      0.0024    0.0009    2.6424  0.0082   0.0006   0.0041
SP500         -6.4410    0.2219  -29.0253  0.0000  -6.8759  -6.0060
-------------------------------------------------------------------
Omnibus:              195.841      Durbin-Watson:         2.118
Prob(Omnibus):        0.000        Jarque-Bera (JB):      1175.748
Skew:                 0.245        Prob(JB):              0.000
Kurtosis:             6.754        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([[ 434.61877137]]), p=4.83325272639e-157, df_denom=1967, df_num
=2>
```

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

Out[72]:
```
Intercept    0.002379
SP500       -6.440975
dtype: float64
```