Computational Finance

Week 2: Dealing with Data

Last week: Introduction to Python

- **Python** is a popular free open-source programming language, that is beginner-friendly.
- Each Python objects is a **data type**. Each type has specific atributes and methods. We discussed the following
    - Numeric types: integer, float
    - Boolean
    - Sequence types: string, list (`[]`), tuple (`()`)
- `if-else` statements, `while` loops and `for` loops allow code to be non-linear by controlling under what conditions and how often lines of code are executed.
- Python functionality is organized in **modules** from which functions can be imported.
- User-specified **functions** can be created too.
- Be careful with the structure when defining a function or loop: use indentation, and don't forget the colon (`:`)!

This week: Dealing with Data

- Numpy
    - Array
    - Vectorization
- Pandas
    - Series
    - DataFrames
- Working with Time Series
- Fetching Data
- Regression Analysis

- **Assignment 1 available**

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, tuples, etc.) and converts them into arrays. The data type is inferred automatically or can be specified.

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

Out[3]:
```
dtype('int32')
```

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

Out[4]:
```
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). If not otherwise stated, each nested list translates into another row.

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

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

In [6]:
```
a.ndim  #Number of dimensions.
```

Out[6]:
```
2
```

In [7]:
```
a.shape #Number of rows and columns.
```

Out[7]:
```
(2, 2)
```

- Other functions for creating arrays include:

```
In [8]: np.eye(3)  #Identity matrix. float64 is the default dtype and can be omitted

Out[8]: array([[1., 0., 0.],
               [0., 1., 0.],
               [0., 0., 1.]])
```

```
In [9]: np.ones([2, 3])  #There's also np.zeros, and np.empty (which results in an uninitialized array).

Out[9]: array([[1., 1., 1.],
               [1., 1., 1.]])
```

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

Out[10]: array([0, 2, 4, 6, 8])
```

```
In [11]: np.linspace(0, 10, 5) #5 equally spaced points between 0 and 10

Out[11]: array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

Indexing

- Indexing and slicing operations are similar to lists:

In [12]:
```
a = np.array([[1., 2.], [3., 4.]])
a[0,0] #Element [row, column]. Equivalent to a[0][0].
```

Out[12]:
```
1.0
```

In [13]:
```
b = a[:, 0]; b  #All rows of first column. Yields a 1-dimensional array (vector), not a matrix wi
```

Out[13]:
```
array([1., 3.])
```

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

In [14]:
```
b[0] = 42; b
```

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

In [15]:
```
c=a.copy(); print(c)
```

```
[[42.  2.]
 [ 3.  4.]]
```

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

In [16]:

```
a = np.arange(10);
b = np.arange(5,15); a, b
```

Out[16]:

```
(array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
 array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14]))
```

In [17]:

```
foo = a > 5; foo
```

Out[17]:

```
array([False, False, False, False, False, False,  True,  True,  True,
        True])
```

In [18]:

```
b[foo]  #feeding a boolean index for slicing
```

Out[18]:

```
array([11, 12, 13, 14])
```

Concatenation and Reshaping

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

In [19]:
```
a = np.array([1, 2, 3]); b = np.array([4, 5, 6]); a,b
```

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

In [20]:
```
c = np.concatenate([a, b]); c  #Concatenate along an existing axis.
```

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

In [21]:
```
d = np.stack([a, b]); d  #Concatenate along a new axis (e.g., vectors to matrix).
```

Out[21]:
```
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` (i.e., unspecified) will be inferred automatically.

In [22]:
```
d.shape
```

Out[22]:
```
(2, 3)
```

In [23]:
```
d = d.reshape(3,-1); d  #3 rows, number of columns determined automatically -1 here inferred to b
```

Out[23]:
```
array([[1, 2],
```

```
       [3, 4],
       [5, 6]])
```

Arithmetic and ufuncs

- NumPy ufuncs are functions that operate elementwise:

In [24]:
```
a = np.arange(1, 5); np.sqrt(a)
```

Out[24]:
```
array([1.        , 1.41421356, 1.73205081, 2.        ])
```

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

In [25]:
```
a = np.arange(1, 5); b = np.arange(5, 9); a, b, a + b, a - b, a / b  #Note: a/b.astype(float) for
```

Out[25]:
```
(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 [26]:
```
np.array([1, 2, 3, 4]) + 2  # same result: np.array([1, 2, 3, 4]) + np.array([2, 2, 2, 2])
```

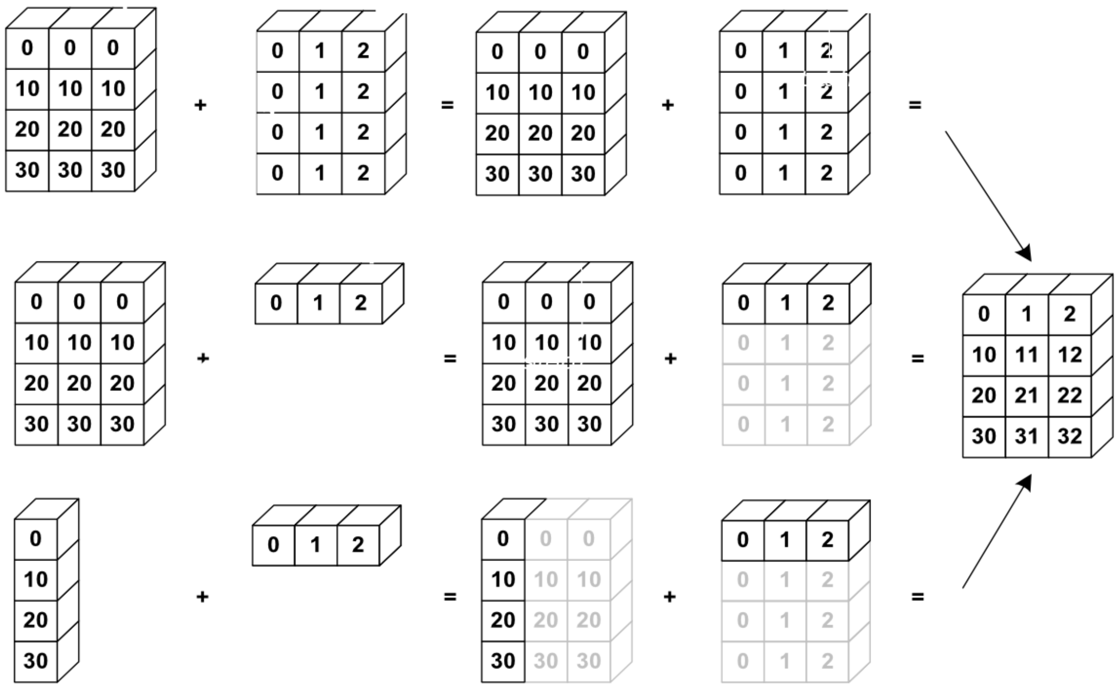
Out[26]:
```
array([3, 4, 5, 6])
```

- 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 [27]: x = np.arange(6).reshape((2, 3)); x  #x has shape (2,3).

Out[27]: array([[0, 1, 2],
                [3, 4, 5]])

In [28]: m = np.mean(x, axis=0); m  #m has shape (3,) and the mean is computed across (not within) 0=rows

Out[28]: array([1.5, 2.5, 3.5])

In [29]: x-m  #the trailing dimension matches, and m is stretched to match the 2 rows of x.

Out[29]: array([[-1.5, -1.5, -1.5],
                [ 1.5,  1.5,  1.5]])
```

- The idea of broadcasting in a picture (these are all 2-dimensional arrays):



- 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 [30]:
```
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
```

Out[30]:
```
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' or $uv^\top$):

In [31]:
```
np.outer(u, v)  # u*v.transpose() and/or u*v.T will not work
```

Out[31]:
```
array([[ 4,  5,  6,  7],
       [ 8, 10, 12, 14],
       [12, 15, 18, 21]])
```

In [32]:
```
u[:, np.newaxis]*v.transpose()  #adding the explicit 1th dimension works | or u[:, np.newaxis]*v[
```

Out[32]:
```
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 [33]:
```
a = np.random.rand(300, 3)  #create a 300x3 matrix of standard normal variates.
a.std(axis=0)  #or np.std(a, axis=0) - again standard deviation are calculated across rows
```

Out[33]:
```
array([0.27401563, 0.28868048, 0.27392285])
```

- 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 [34]:
```
np.save('myfile.npy', a)  #save `a` as a binary .npy file.
```

In [35]:
```
import os
print(os.listdir('.'))
```

```
['.ipynb_checkpoints', 'archive', 'clips', 'data', 'data.zip', 'img', 'img.zip',
'myfile.npy', 'notes', 'pdf', 'README.md', 'week1.ipynb', 'week1.slides.html', 'w
eek2.ipynb', 'week2.slides.html', 'week3.ipynb', 'week3.slides.html', 'week4-Copy
1.ipynb', 'week4.ipynb', 'week4.slides.html', 'week5.ipynb', 'week5.slides.html',
'week6.ipynb', 'week6.slides.html', 'week6bonus.ipynb']
```

In [36]:
```
b = np.load('myfile.npy')  #load the data into variable b.
os.remove('myfile.npy')  #clean up.
```

In [37]:
```
np.savetxt('myfile.csv', a, delimiter=',')  #save `a` as a CSV file (comma seperated values, can
```

In [38]:
```
b = np.loadtxt('myfile.csv', delimiter=',')  #Load data into `b`.
os.remove('myfile.csv')
```

Pandas Dataframes

Introduction to Pandas

- `pandas` (from panel data) is another fundamental package in the SciPy stack (user guide & 'cookbook' (short examples of common operations)).
- It provides a number of datastructures (*series* and *dataframes*) designed for storing observational data, and powerful methods for manipulating (*munging*, or *wrangling*) these data.
- It is usually imported as `pd`:

In [39]:
```
import pandas as pd
```

Series

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

In [40]:
```
pop = pd.Series([5.7, 82.7, 17.0], name='Population'); pop  #the descriptive name is optional.
```

Out[40]:
```
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 [41]:
```
pop.index=['DK', 'DE', 'NL'];
pop
```

Out[41]:
```
DK     5.7
DE    82.7
NL    17.0
Name: Population, dtype: float64
```

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

In [42]:
```
pop['NL']
```

Out[42]:
```
17.0
```

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

In [43]:
```
gdp = pd.Series([3494.898, 769.930], name='Nominal GDP in Billion USD', index=['DE', 'NL']); gdp
```

Out[43]:
```
DE    3494.898
NL     769.930
Name: Nominal GDP in Billion USD, dtype: float64
```

In [44]:
```
gdp / pop
```

Out[44]:
```
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 [45]:
```
data = pd.concat([gdp, pop], axis=1); data  #concatenate two Series to a DataFrame.
```

Out[45]:

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

- Columns are indexed by column name:

In [46]:
```
data.columns
```

Out[46]:
```
Index(['Nominal GDP in Billion USD', 'Population'], dtype='object')
```

In [47]:
```
data['Population']  #data.Population works too
```

Out[47]:
```
DE    82.7
NL    17.0
DK     5.7
Name: Population, dtype: float64
```

- Rows are indexed with the `loc` method:

In [48]:
```
data.loc['NL']
```

Out[48]:
```
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 [49]:
```
data['Language'] = ['German', 'Danish', 'Dutch']; data #Add a new column from a list.
```

Out[49]:

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

- Another is to use the `join` method:

In [50]:
```
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[50]:

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

- 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 [51]:
```
data = data.join(s, how='outer'); data  #Assignment to store the modified frame.
```

Out[51]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
DK	NaN	5.7	Dutch	DKK
NL	769.930	17.0	Danish	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 [52]:
```
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[52]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
DK	NaN	5.7	Dutch	DKK
NL	769.930	17.0	Danish	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 [53]:
```
data = data.dropna(); data
```

Out[53]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
NL	769.930	17.0	Danish	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 [54]:
```
data.describe()
```

Out[54]:

	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 [55]:

```
data.head()  #Show the first few rows. data.tail shows the last few.
```

Out[55]:

	Nominal GDP in Billion USD	Population	Language	Currency
DE	3494.898	82.7	German	EUR
NL	769.930	17.0	Danish	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 [56]:
```
data.to_csv('myfile.csv')  #to_excel exists as well
```

In [57]:
```
with open('myfile.csv', 'r') as file:  #any files opened in the `with`statement will be closed au
    print(file.read())
    info = os.stat('myfile.csv')
    print(info.st_size)  #reported in bytes
```

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

165
```

- To load data into a dataframe, use `pd.read_csv`:

In [58]:
```
pd.read_csv('myfile.csv', index_col=0)
```

Out[58]:

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

In [59]:
```
os.remove('myfile.csv')  #clean up
```

- Other, possibly more efficient, methods exist; see Chapter 5 of Hilpisch (2019).

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 [60]:
```
pd.datetime.today()
```

Out[60]:
```
datetime.datetime(2021, 10, 28, 14, 55, 46, 594384)
```

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

In [61]:
```
pd.datetime.strptime('2021-10-31', '%Y-%m-%d')
```

Out[61]:
```
datetime.datetime(2021, 10, 31, 0, 0)
```

- Pandas uses `Timestamps` instead of `datetime` objects. Unlike datetime, they store frequency and time zone information. The two can mostly be used interchangeably. See Hilpisch (2019) Appendix A for details.

In [62]:
```
pd.Timestamp('2021-10-31')
```

Out[62]:
```
Timestamp('2021-10-31 00:00:00')
```

- A time series is a `Series` with a special index, called a `DatetimeIndex`; essentially an array of `Timestamp`s.
- It can be created using the `date_range` function; see Tables 5.2 and 5.3 in Hilpisch (2019).
- Note that `freq='B'` is a particular choice of a so-called *Offset Alias* and stands for business day frequency.

In [63]:
```
myindex = pd.date_range(end=pd.Timestamp.today(), normalize=True, periods=100, freq='B')
P = 20 + np.random.randn(100).cumsum()  #simulate share prices
aapl = pd.Series(P, name="AAPL", index=myindex)
aapl.tail()
```

Out[63]:
```
2021-10-22    18.418944
2021-10-25    18.407497
2021-10-26    18.879239
2021-10-27    21.630680
2021-10-28    21.659399
Freq: B, Name: AAPL, dtype: float64
```

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

In [64]:
```
aapl['2021-08-19']  #similarly, aapl.loc['2021-08-19'] is possible
```

Out[64]:
```
26.437155896142514
```

In [65]:
```
aapl['2021-08-21':'2021-08-30']
```

Out[65]:
```
2021-08-23    22.390427
2021-08-24    21.845474
2021-08-25    21.069111
2021-08-26    22.498693
2021-08-27    20.559248
2021-08-30    20.168249
Freq: B, Name: AAPL, dtype: float64
```

Financial Returns

- We mostly work with returns rather than prices, because their statistical properties are more desirable (e.g., stationarity and ergodicity).
- 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.
- Log returns are *time-additive* (but should not be aggregated across assets).
- To convert from prices to returns, use the `shift(k)` method, which lags by k periods (or leads if $k < 0$).

In [66]:

```
aapl = np.log(aapl) - np.log(aapl).shift(1)
aapl.head()
```

Out[66]:

```
2021-06-11         NaN
2021-06-14    0.099008
2021-06-15    0.039313
2021-06-16   -0.021097
2021-06-17    0.048697
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 [67]:

```
P = 20 + np.random.randn(100).cumsum()  #simulate share prices
rf = 1 + np.random.randn(100) / 100  #simulate 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)  #concatenate pandas objects along the column axis
returns.tail()
```

Out[67]:

	AAPL	MSFT
2021-10-22	0.015653	-0.009168
2021-10-25	-0.000622	-0.025175
2021-10-26	0.025305	-0.011567
2021-10-27	0.136050	0.062138
2021-10-28	0.001327	-0.046720

- Then the desired operation can be expressed as

In [68]:

```
excess_returns = returns.sub(rf, axis='index')  #subtract series rf from all columns
```

Fetching Data

- `pandas_datareader` allows one to fetch data from the web (user guide).
- It is a separate package (not part of pandas), so we need to install it.

In [69]:

```
#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 [70]:

```
start = pd.datetime(2011, 1, 1)
end = pd.datetime.today()
p = web.DataReader('SP500', 'fred', start, end)  #S&P500 from St. Louis Fed (pulls Adj. Close)
p.tail()
```

Out[70]:

	SP500
DATE	
2021-10-21	4549.78
2021-10-22	4544.90
2021-10-25	4566.48
2021-10-26	4574.79
2021-10-27	4551.68

Stock market data sources

- For US stock market data, several options are available:
    - FRED: Stock market indexes, no individual stocks.
    - Yahoo: Yahoo Finance is an important data source, since it has wide and long coverage. Recently supported again — still development phase though
    - WIKI Prices from Nasdaq Data Link (was Quandl): data up to 04/2018.
    - Tiingo: Personal use only.
    - Alpha Vantage: 20 years of data.
- Check the user guide for currently available sources.
- May need API (*application programming interface*) key to access data.
    - Can be requested for free.

- Example using an API key to read Pfizer (PFE) stock market data from WIKI Prices.
- Note that `'quandl'` call still works even though QUANDL has become Nasdaq Data Link.

In [71]:

```
QUANDL_API_KEY = "P3v6YJ-K1DhibzGF1EgX"  # This is my personal API key; request your own at data
p2 = web.DataReader('WIKI/PFE', 'quandl', start, end, access_key=QUANDL_API_KEY)
p2.head()
```

Out[71]:

Date	Open	High	Low	Close	Volume	ExDividend	SplitRatio	AdjOpen	AdjHigh	AdjLow	AdjClose	AdjVolume
2018-03-27	35.18	35.5600	34.78	35.01	25418639.0	0.0	1.0	35.18	35.5600	34.78	35.01	25418639.0
2018-03-26	34.93	35.1500	34.32	35.04	23464967.0	0.0	1.0	34.93	35.1500	34.32	35.04	23464967.0
2018-03-23	35.49	35.5000	34.44	34.49	27489780.0	0.0	1.0	35.49	35.5000	34.44	34.49	27489780.0
2018-03-22	36.03	36.1433	35.47	35.60	20288017.0	0.0	1.0	36.03	36.1433	35.47	35.60	20288017.0
2018-03-21	36.49	36.7300	36.20	36.27	16405382.0	0.0	1.0	36.49	36.7300	36.20	36.27	16405382.0

Regression Analysis

- Like in Hilpisch (2019), 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. Consequently, asset volatility should increase.
- 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 [72]:

```
df = pd.DataFrame()  # when in doubt, start with an empty DataFrame
df['SP500'] = np.log(p['SP500']) - np.log(p['SP500']).shift(1)
p = web.DataReader('VIXCLS', 'fred', start, end)  #VIX volatility index
df['VIX'] = np.log(p['VIXCLS']) - np.log(p['VIXCLS']).shift(1)
df = df.dropna(axis=0, how='any')
df.head()
```

Out[72]:

DATE	SP500	VIX
2011-10-31	-0.025049	0.199966
2011-11-01	-0.028340	0.148892
2011-11-02	0.015976	-0.060157
2011-11-03	0.018608	-0.070871
2011-11-04	-0.006300	-0.011210

- Next, we run an OLS regression of the VIX returns on those of the S&P.
- The regression functionality is stored in the `statsmodels` package (user guide).
- We will use a different interface (API) which allows us to specify regressions using R-style formulas (user guide).
- Loading this package may yield a warning. A warning is *not* an error, the code will continue. Also, this warning is probably a bug, see e.g. here.
- We will use heteroskedasticity and autocorrelation consistent (HAC) standard errors.

In [73]:

```
import statsmodels.formula.api as smf  # Use import statsmodels.api as sm if you don't want R-sty
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:     0.549
Dependent Variable: VIX              AIC:                -7330.6816
Date:               2021-10-28 14:55 BIC:                -7319.0969
No. Observations:   2422             Log-Likelihood:     3667.3
Df Model:           1                F-statistic:        86.37
Df Residuals:       2420             Prob (F-statistic): 3.23e-20
R-squared:          0.549            Scale:              0.0028359
--------------------------------------------------------------------
             Coef.    Std.Err.     z      P>|z|    [0.025    0.975]
--------------------------------------------------------------------
Intercept    0.0016     0.0010   1.5873   0.1124   -0.0004    0.0036
SP500       -5.5881     0.6013  -9.2933   0.0000   -6.7666   -4.4095
------------------------------------------------------------------
Omnibus:              757.018      Durbin-Watson:          2.233
Prob(Omnibus):        0.000        Jarque-Bera (JB):       16354.977
Skew:                 0.950        Prob(JB):               0.000
Kurtosis:             15.588       Condition No.:          95
==================================================================
```

- We can run the above within one line: `result = smf.ols('VIX ~ SP500', data=df).fit(cov_type="HAC", cov_kwds={'maxlags':5})`

In [74]:

```
result.tvalues
```

Out[74]:

```
Intercept     1.587347
SP500        -9.293342
dtype: float64
```

- Conclusion: We indeed find a significant negative effect of the index returns ($t = -9.29$), 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 other useful methods and variables (for the entire list check here):

In [75]:

```
print(result.f_test('SP500=0, Intercept=0'))  #joint test that regressors are equal to zero
```

```
<F test: F=array([[48.05627339]]), p=3.413627617271817e-21, df_denom=2.42e+03, df
_num=2>
```

In [76]:

```
result.params
```

Out[76]:

```
Intercept     0.001624
SP500        -5.588073
dtype: float64
```

Summary

- `Numpy`'s `ndarray`s are a sequence type with mutable homogenous elements.
    - ufuncs and broadcasting allow us to vectorize code, such that the code is more efficient.
- The `pandas` package introduces `Series` and `DataFrames`.
    - Useful for analyzing panel data, and in particular time series.
    - Effective methods to manipulate the data.
- Directly communicate with online databases to load (financial) data using `pandas_datareader`.
- Functionality for regression analysis is stored in the `statsmodels` package.

Copyright Statement

- Course slides were created by Simon Broda for Python 2.7 — Andreas Rapp adapted the 3.6. Maintained and updated by Bart Keijsers.
- Week 4 slides were created by Bart Keijsers. The hierarchical indexing example is from th Data Science Methods by Cees Diks and Bram Wouters.
- All figures have been produced for this course using Python. Empirical results are based data available from FRED, Quandl/WIKI, Kenneth French's website and Yahoo Finance.
- This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 Internationa
- More information on Simon Broda's Github.