Berlin School of Economics and Law
Prof. Dr. Natalie Packham

Summer term 2020
LV-Nr. 425093

Computational Finance and FinTech
Numerical and Computational Foundations

Contents

2 Numerical and Computational Foundations

- Further reading: **Py4Fi, Chapters 4 and 5**

2.1 Arrays with Python lists

Introduction to Python arrays

- Before introducing more sophisticated objects for data storage, let's take a look at the built-in Python `list` object.
- A `list` object is a one-dimensional array:

[1]:
```
v = [0.5, 0.75, 1.0, 1.5, 2.0]
```

- `list` objects can contain arbitrary objects.
- In particular, a `list` can contain other `list` objects, creating two- or higher-dimensional arrays:

[2]:
```
m = [v, v, v]
m
```

[2]:
```
[[0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0]]
```

`list` objects

[3]:
```
m[1]
```

[3]:
```
[0.5, 0.75, 1.0, 1.5, 2.0]
```

[4]:
```
m[1][0]
```

[4]:
```
0.5
```

- Feel free to push this to higher dimensions…

[5]:
```
v1 = [0.5, 1.5]
v2 = [1, 2]
```

```
m = [v1, v2]
c = [m, m]
c
```

[5]:
```
[[[0.5, 1.5], [1, 2]], [[0.5, 1.5], [1, 2]]]
```

[6]:
```
c[1][1][0]
```

[6]:
```
1
```

Reference pointers

- Important: `list`'s work with **reference pointers**.
- Internally, when creating new objects out of existing objects, only pointers to the objects are copied, not the data!

[7]:
```
v = [0.5, 0.75, 1.0, 1.5, 2.0]
m = [v, v, v]
m
```

[7]:
```
[[0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0],
 [0.5, 0.75, 1.0, 1.5, 2.0]]
```

[8]:
```
v[0] = 'Python'
m
```

[8]:
```
[['Python', 0.75, 1.0, 1.5, 2.0],
 ['Python', 0.75, 1.0, 1.5, 2.0],
 ['Python', 0.75, 1.0, 1.5, 2.0]]
```

Python array class

- Python also has an `array` module
- See Documentation

2.2 NumPy arrays

NumPy arrays

- `NumPy` is a library for richer array data structures.
- The basic object is `ndarray`, which comes in two flavours:

Object type	Meaning	Used for
`ndarray` (regular)	*n*-dimensional array object	Large arrays of numerical data
`ndarray` (record)	2-dimensional array object	Tabular data organized in columns

ndarray

Source: Python for Finance, 2nd ed.

- The `ndarray` object is more specialised than the `list` object, but comes with more functionality.
- An array object represents a multidimensional, homogeneous array of fixed-size items.
- Here is a useful tutorial

Regular NumPy arrays

- Creating an array:

[9]:
```
import numpy as np # import numpy
a = np.array([0, 0.5, 1, 1.5, 2]) # array(...) is the constructor for ndarray's
```

[10]:
```
type(a)
```

[10]:
```
numpy.ndarray
```

- `ndarray` assumes objects of the same type and will modify types accordingly:

[11]:
```
b = np.array([0, 'test'])
b
```

[11]:
```
array(['0', 'test'], dtype='<U21')
```

[12]:
```
type(b[0])
```

[12]:
```
numpy.str_
```

Constructing arrays by specifying a range

- `np.arange()` creates an array spanning a range of numbers (= a sequence).
- Basic syntax: `np.arange(start, stop, steps)`
- It is possible to specify the data type (e.g. `float`)
- To invoke an explanation of `np.arange` (or any other object or method), type `np.arange?`

[13]:
```
np.arange?
```

[14]:
```
np.arange(0, 2.5, 0.5)
```

[14]:
```
array([0. , 0.5, 1. , 1.5, 2. ])
```

`NOTE: The interval specification refers to a half-open interval: [start, stop).`

`ndarray` methods

- The `ndarray` object has a multitude of useful built-in methods, e.g.
  - `sum()` (the sum),
  - `std()` (the standard deviation),
  - `cumsum()` (the cumulative sum).
- Type `a.` and hit `TAB` to obtain a list of the available functions.
- More documentation is found here.

[15]:
```
a.sum()
```

[15]:
```
5.0
```

[16]:
```
a.std()
```

[16]:
```
0.7071067811865476
```

[17]:
```
a.cumsum()
```

[17]:
```
array([0. , 0.5, 1.5, 3. , 5. ])
```

Slicing 1d-Arrays

- With one-dimensional `ndarray` objects, indexing works as usual.

[18]:
```
a[1]
```

[18]:
```
0.5
```

[19]:
```
a[:2]
```

[19]:
```
array([0. , 0.5])
```

[20]:
```
a[2:]
```

[20]:
```
array([1. , 1.5, 2. ])
```

Mathematical operations

- Mathematical operations are applied in a **vectorised** way on an `ndarray` object.
- Note that these operations work differently on `list` objects.

[21]:
```
l = [0, 0.5, 1, 1.5, 2]
l
```

[21]:
```
[0, 0.5, 1, 1.5, 2]
```

[22]:
```
2 * l
```

[22]:
```
[0, 0.5, 1, 1.5, 2, 0, 0.5, 1, 1.5, 2]
```

- `ndarray`:

[23]:
```
a = np.arange(0, 7, 1)
a
```

[23]:
```
array([0, 1, 2, 3, 4, 5, 6])
```

[24]:
```
2 * a
```

[24]:
```
array([ 0,  2,  4,  6,  8, 10, 12])
```

Mathematical operations (cont'd)

[25]:
```
a + a
```

[25]:
```
array([ 0,  2,  4,  6,  8, 10, 12])
```

[26]:
```
a ** 2
```

[26]:
```
array([ 0,  1,  4,  9, 16, 25, 36])
```

[27]:
```
2 ** a
```

[27]:
```
array([ 1,  2,  4,  8, 16, 32, 64])
```

[28]:
```
a ** a
```

[28]:
```
array([    1,     1,     4,    27,   256,  3125, 46656])
```

Universal functions in NumPy

- A number of universal functions in `NumPy` are applied element-wise to arrays:

[29]:
```
np.exp(a)
```

[29]:
```
array([  1.        ,   2.71828183,   7.3890561 ,  20.08553692,
        54.59815003, 148.4131591 , 403.42879349])
```

[30]:
```
np.sqrt(a)
```

[30]:
```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974])
```

Multiple dimensions

- All features introduced so far carry over to multiple dimensions.
- An array with two rows:

[31]:
```
b = np.array([a, 2 * a])
b
```

[31]:
```
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 0,  2,  4,  6,  8, 10, 12]])
```

- Selecting the first row, a particular element, a column:

[32]:
```
b[0]
```

[32]:
```
array([0, 1, 2, 3, 4, 5, 6])
```

[33]:
```
b[1,1]
```

[33]:
```
2
```

[34]:
```
b[:,1]
```

[34]:
```
array([1, 2])
```

Multiple dimensions

- Calculating the sum of all elements, column-wise and row-wise:

[35]:
```
b.sum()
```

[35]:
```
63
```

[36]:
```
b.sum(axis = 0)
```

[36]:
```
array([ 0,  3,  6,  9, 12, 15, 18])
```

[37]:
```
b.sum(axis = 1)
```

[37]:
```
array([21, 42])
```

Note: `axis = 0` refers to column-wise and `axis = 1` to row-wise.

Further methods for creating arrays

- Often, we want to create an array and populate it later.
- Here are some methods for this:

[38]:
```
np.zeros((2,3), dtype = 'i') # array with two rows and three columns
```

[38]:
```
array([[0, 0, 0],
       [0, 0, 0]], dtype=int32)
```

[39]:
```
np.ones((2,3,4), dtype = 'i') # array dimensions: 2 x 3 x 4
```

[39]:
```
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int32)
```

[40]:
```
np.empty((2,3))
```

[40]:
```
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974]])
```

Further methods for creating arrays

[41]:
```
np.eye(3)
```

[41]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

[42]:
```
np.diag(np.array([1,2,3,4]))
```

[42]:
```
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

NumPy dtype objects

Source: Python for Finance, 2nd ed.

Logical operations

- NumPy Arrays can be compared, just like lists.

dtype	Description	Example
`?`	Boolean	`?` (`True` or `False`)
`i`	Signed integer	`i8` (64-bit)
`u`	Unsigned integer	`u8` (64-bit)
`f`	Floating point	`f8` (64-bit)
`c`	Complex floating point	`c32` (256-bit)
`m`	`timedelta`	`m` (64-bit)
`M`	`datetime`	`M` (64-bit)
`O`	Object	`O` (pointer to object)
`U`	Unicode	`U24` (24 Unicode characters)
`V`	Raw data (void)	`V12` (12-byte data block)

dtype object

[43]:
```
first = np.array([0, 1, 2, 3, 3, 6,])
second = np.array([0, 1, 2, 3, 4, 5,])
```

[44]:
```
first > second
```

[44]:
```
array([False, False, False, False, False,  True])
```

[45]:
```
first.sum() == second.sum()
```

[45]:
```
True
```

[46]:
```
np.any([a == 4])
```

[46]:
```
True
```

[47]:
```
np.all([a == 4])
```

[47]:
```
False
```

Reshape and resize

- `ndarray` objects are immutable, but they can be reshaped (changes the view on the object) and resized (creates a new object):

[48]:
```
ar = np.arange(15)
ar
```

[48]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

[49]:
```
ar.reshape((3,5))
```

[49]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

[50]:
```
ar
```

[50]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Reshape and resize

[51]:
```
ar.resize((5,3))
```

[52]:
```
ar
```

[52]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Note: `reshape()` did not change the original array. `()resize` did change the array's shape permanently.

Reshape and resize

- `reshape()` does not alter the total number of elements in the array.
- `resize()` can decrease (down-size) or increase (up-size) the total number of elements.

[53]:
```
ar
```

[53]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

[54]:
```
np.resize(ar, (3,3))
```

[54]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Reshape and resize

[55]:
```
np.resize(ar, (5,5))
```

[55]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9]])
```

[56]:
```
a.shape # returns the array's dimensions
```

[56]:
```
(7,)
```

Further operations

- Transpose:

[57]:
```
g = np.arange(0, 6)
g.resize(2,3)
g
```

[57]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

[58]:
```
g.T
```

[58]:
```
array([[0, 3],
       [1, 4],
       [2, 5]])
```

- Flattening:

[59]:
```
g.flatten()
```

[59]:
```
array([0, 1, 2, 3, 4, 5])
```

Further operations

- Stacking: `hstack` or `vstack` can used to connect two arrays horizontally or vertically.

[60]:
```
b = np.ones((2,3))
```

[61]:
```
np.vstack((g, b))
```

[61]:
```
array([[0., 1., 2.],
       [3., 4., 5.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
NOTE: The size of the to-be connected dimensions must be equal.
```

2.3 Structured NumPy arrays

Structured NumPy arrays

- The specialisation of `ndarray` may be to narrow.
- However, one can instantiate `ndarray` with a dedicated `dtype`.
- This allows to build database-like data sets where each row corresponds to an “entry”.

Structured NumPy arrays

- Creating a data type:

[62]:
```
dt = np.dtype([('Name', 'S10'), ('Age', 'i4'),
               ('Height', 'f'), ('Children/Pets', 'i4', 2)])
dt
```

[62]:
```
dtype([('Name', 'S10'), ('Age', '<i4'), ('Height', '<f4'), ('Children/Pets',
'<i4', (2,))])
```

- Equivalently:

[63]:
```
dt = np.dtype({'names': ['Name', 'Age', 'Height', 'Children/Pets'],
              'formats':'O int float int,int'.split()})

dt
```

[63]:
```
dtype([('Name', 'O'), ('Age', '<i8'), ('Height', '<f8'), ('Children/Pets',
[('f0', '<i8'), ('f1', '<i8')])])
```

Structured NumPy arrays

- Now create the `ndarray` with the new data type:

[64]:
```
s = np.array([('Smith', 45, 1.83, (0, 1)),
              ('Jones', 53, 1.72, (2, 2))], dtype=dt)

s
```

[64]:
```
array([('Smith', 45, 1.83, (0, 1)), ('Jones', 53, 1.72, (2, 2))],
      dtype=[('Name', 'O'), ('Age', '<i8'), ('Height', '<f8'), ('Children/Pets',
[('f0', '<i8'), ('f1', '<i8')])])
```

[65]:
```
type(s)
```

[65]:
```
numpy.ndarray
```

Structured NumPy arrays

- The columns can be accessed through their names:

[66]:
```
s['Name']
```

[66]:
```
array(['Smith', 'Jones'], dtype=object)
```

[67]:
```
s['Height'].mean()
```

[67]:
```
1.775
```

[68]:
```
s[0]
```

[68]:
```
('Smith', 45, 1.83, (0, 1))
```

[69]:
```
s[1]['Age']
```

[69]:
```
53
```

2.4 Data Analysis with pandas: DataFrame

Data analysis with pandas

- `pandas` is a powerful Python library for data manipulation and analysis. Its name is derived from **pan**el **data**.
- We cover the following data structures:

Source: Python for Finance, 2nd ed.

Object type	Meaning	Used for
`DataFrame`	2-dimensional data object with index	Tabular data organized in columns
`Series`	1-dimensional data object with index	Single (time) series of data

Pandas datatypes

DataFrame Class

- DataFrame is a class that handles tabular data, organised in columns.
- Each row corresponds to an entry or a data record.
- It is thus similar to a table in a relational database or an Excel spreadsheet.

[70]:
```
import pandas as pd

df = pd.DataFrame([10,20,30,40], # data as a list
                  columns=['numbers'], # column label
                  index=['a', 'b', 'c', 'd']) # index values for entries
```

[71]:
```
df
```

[71]:
```
   numbers
a       10
b       20
c       30
d       40
```

DataFrame Class

- The `columns` can be named (but don't need to be).
- The `index` can take different forms such as numbers or strings.
- The input data for the `DataFrame` Class can come in different types, such as `list`, `tuple`, `ndarray` and `dict` objects.

Simple operations

- Some simple operations applied to a `DataFrame` object:

[72]:
```
df.index
```

[72]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

[73]:
```
df.columns
```

[73]:
```
Index(['numbers'], dtype='object')
```

Simple operations

[74]:
```
df.loc['c'] # selects value corresponding to index c
```

[74]:
```
numbers    30
Name: c, dtype: int64
```

[75]:
```
df.loc[['a', 'd']] # selects values correponding t indices a and d
```

[75]:
```
   numbers
a       10
d       40
```

[76]:
```
df.iloc[1:3] # select second and third rows
```

[76]:
```
   numbers
b       20
c       30
```

Simple operations

[77]:
```
df.sum()
```

[77]:
```
numbers    100
dtype: int64
```

- Vectorised operations as with `ndarray`:

[78]:
```
df ** 2
```

[78]:
```
   numbers
a      100
b      400
c      900
d     1600
```

Extending `DataFrame` objects

[79]:
```
df['floats'] = (1.5, 2.5, 3.5, 4.5) # adds a new column
```

[80]:
```
df
```

[80]:
```
   numbers  floats
a       10     1.5
b       20     2.5
c       30     3.5
d       40     4.5
```

[81]:
```
df['floats']
```

[81]:
```
a    1.5
b    2.5
c    3.5
d    4.5
Name: floats, dtype: float64
```

Extending `DataFrame` objects

- A `DataFrame` object can be taken to define a new column:

[82]:
```
df['names'] = pd.DataFrame(['Yves', 'Sandra', 'Lilli', 'Henry'],
                           index = ['d', 'a', 'b', 'c'])
```

[83]:
```
df
```

[83]:

	numbers	floats	names
a	10	1.5	Sandra
b	20	2.5	Lilli
c	30	3.5	Henry
d	40	4.5	Yves

Extending `DataFrame` objects

- Appending data:

[84]:
```
df = df.append(pd.DataFrame({'numbers': 100, 'floats': 5.75, 'names': 'Jill'},
                            index = ['y',]))
```

[85]:
```
df
```

[85]:

	numbers	floats	names
a	10	1.50	Sandra
b	20	2.50	Lilli
c	30	3.50	Henry
d	40	4.50	Yves
y	100	5.75	Jill

Extending `DataFrame` objects

- Be careful when appending without providing an index – the index gets replaced by a simple range index:

[86]:
```
df.append({'numbers': 100, 'floats': 5.75, 'names': 'Jill'}, ignore_index=True)
```

[86]:

	numbers	floats	names
0	10	1.50	Sandra
1	20	2.50	Lilli
2	30	3.50	Henry
3	40	4.50	Yves
4	100	5.75	Jill
5	100	5.75	Jill

Extending `DataFrame` objects

- Appending with missing data:

[87]:
```
df = df.append(pd.DataFrame({'names': 'Liz'},
                            index = ['z']),
                            sort = False)
```

[88]:
```
df
```

[88]:

	numbers	floats	names
a	10.0	1.50	Sandra
b	20.0	2.50	Lilli

```
c     30.0    3.50   Henry
d     40.0    4.50    Yves
y    100.0    5.75    Jill
z      NaN     NaN     Liz
```

Mathematical operations on Data Frames

- A lot of mathematical methods are implemented for `DataFrame` objects:

[89]:
```
df[['numbers', 'floats']].sum()
```

[89]:
```
numbers    200.00
floats      17.75
dtype: float64
```

[90]:
```
df['numbers'].var()
```

[90]:
```
1250.0
```

[91]:
```
df['numbers'].max()
```

[91]:
```
100.0
```

Time series with Data Frame

- In this section we show how a DataFrame can be used to manage time series data.
- First, we create a `DataFrame` object using random numbers in an `ndarray` object.

[92]:
```
import numpy as np
import pandas as pd
np.random.seed(100)
a = np.random.standard_normal((9,4))
a
```

[92]:
```
array([[-1.74976547,  0.3426804 ,  1.1530358 , -0.25243604],
       [ 0.98132079,  0.51421884,  0.22117967, -1.07004333],
       [-0.18949583,  0.25500144, -0.45802699,  0.43516349],
       [-0.58359505,  0.81684707,  0.67272081, -0.10441114],
       [-0.53128038,  1.02973269, -0.43813562, -1.11831825],
       [ 1.61898166,  1.54160517, -0.25187914, -0.84243574],
       [ 0.18451869,  0.9370822 ,  0.73100034,  1.36155613],
       [-0.32623806,  0.05567601,  0.22239961, -1.443217  ],
       [-0.75635231,  0.81645401,  0.75044476, -0.45594693]])
```

[93]:
```
df = pd.DataFrame(a)
```

Note: To learn more about Python's built-in pseudo-random number generator (PRNG), see here.

Practical example using `DataFrame` class

[94]:
```
df
```

[94]:
```
          0         1         2         3
0 -1.749765  0.342680  1.153036 -0.252436
1  0.981321  0.514219  0.221180 -1.070043
2 -0.189496  0.255001 -0.458027  0.435163
```

```
3 -0.583595  0.816847  0.672721 -0.104411
4 -0.531280  1.029733 -0.438136 -1.118318
5  1.618982  1.541605 -0.251879 -0.842436
6  0.184519  0.937082  0.731000  1.361556
7 -0.326238  0.055676  0.222400 -1.443217
8 -0.756352  0.816454  0.750445 -0.455947
```

Practical example using `DataFrame` class

- Arguments to the `DataFrame()` function for instantiating a `DataFrame` object:

Parameter	Format	Description
`data`	`ndarray/dict/DataFrame`	Data for `DataFrame`; `dict` can contain `Series`, `ndarray`, `list`
`index`	`Index`/array-like	Index to use; defaults to `range(n)`
`columns`	`Index`/array-like	Column headers to use; defaults to `range(n)`
`dtype`	`dtype`, default `None`	Data type to use/force; otherwise, it is inferred
`copy`	`bool`, default `None`	Copy data from inputs

DataFrame object

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- In the next steps, we set column names and add a time dimension for the rows.

[95]:
```python
df.columns = ['No1', 'No2', 'No3', 'No4']
```

[96]:
```python
df
```

[96]:
```
        No1       No2       No3       No4
0 -1.749765  0.342680  1.153036 -0.252436
1  0.981321  0.514219  0.221180 -1.070043
2 -0.189496  0.255001 -0.458027  0.435163
3 -0.583595  0.816847  0.672721 -0.104411
4 -0.531280  1.029733 -0.438136 -1.118318
5  1.618982  1.541605 -0.251879 -0.842436
6  0.184519  0.937082  0.731000  1.361556
7 -0.326238  0.055676  0.222400 -1.443217
8 -0.756352  0.816454  0.750445 -0.455947
```

[97]:
```python
df['No3'].values.flatten()
```

[97]:
```
array([ 1.1530358 ,  0.22117967, -0.45802699,  0.67272081, -0.43813562,
       -0.25187914,  0.73100034,  0.22239961,  0.75044476])
```

Practical example using `DataFrame` class

- `pandas` is especially strong at handling times series data efficiently.
- Assume that the data rows in the `DataFrame` consist of monthtly observations starting in January 2019.
- The method `date_range()` generates a `DateTimeIndex` object that can be used as the row index.

[98]:
```
dates = pd.date_range('2019-1-1', periods = 9, freq = 'M')
dates
```

[98]:
```
DatetimeIndex(['2019-01-31', '2019-02-28', '2019-03-31', '2019-04-30',
               '2019-05-31', '2019-06-30', '2019-07-31', '2019-08-31',
               '2019-09-30'],
              dtype='datetime64[ns]', freq='M')
```

Practical example using `DataFrame` class

- Parameters of the `date_range()` function:

Parameter	Format	Description
`start`	`string/datetime`	Left bound for generating dates
`end`	`string/datetime`	Right bound for generating dates
`periods`	`integer/None`	Number of periods (if `start` or `end` is `None`)
`freq`	`string/DateOffset`	Frequency string, e.g., `5D` for 5 days
`tz`	`string/None`	Time zone name for localized index
`normalize`	`bool`, default `None`	Normalizes `start` and `end` to midnight
`name`	`string`, default `None`	Name of resulting index

Date range parameters

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Frequency parameter of `date_range()` function:

<table>
    <td></td>
    <td></td>
</table>

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Now set the row index to the dates:

[99]:
```
df.index = dates

df
```

[99]:
```
                 No1       No2       No3       No4
2019-01-31 -1.749765  0.342680  1.153036 -0.252436
2019-02-28  0.981321  0.514219  0.221180 -1.070043
2019-03-31 -0.189496  0.255001 -0.458027  0.435163
2019-04-30 -0.583595  0.816847  0.672721 -0.104411
2019-05-31 -0.531280  1.029733 -0.438136 -1.118318
2019-06-30  1.618982  1.541605 -0.251879 -0.842436
2019-07-31  0.184519  0.937082  0.731000  1.361556
```

```
2019-08-31 -0.326238  0.055676  0.222400 -1.443217
2019-09-30 -0.756352  0.816454  0.750445 -0.455947
```

Practical example using `DataFrame` class

- Next, we visualise the data:

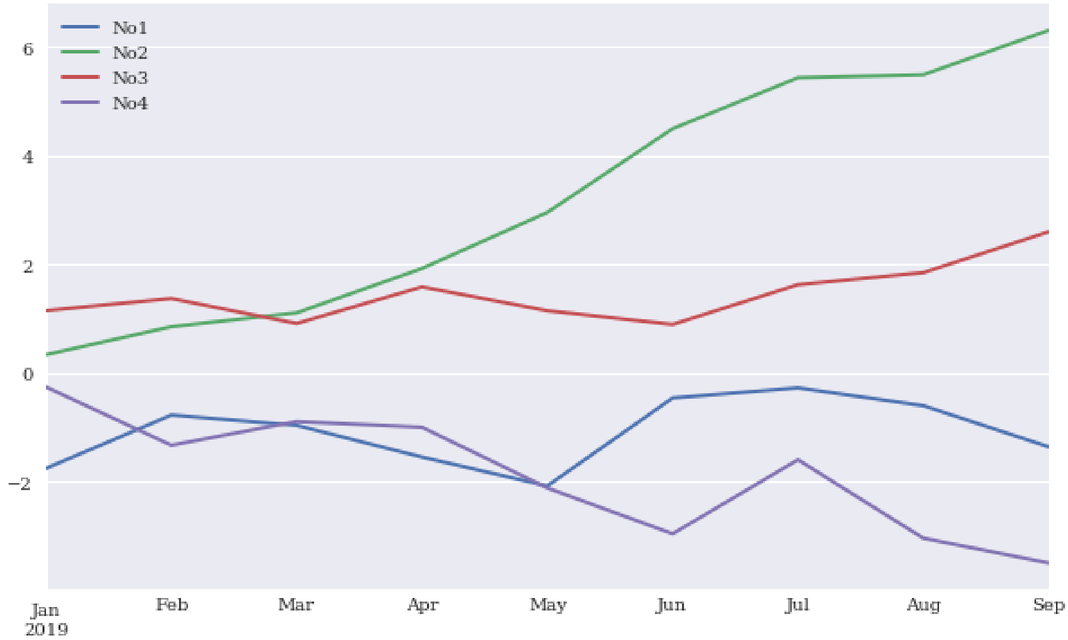
[100]:
```
from pylab import plt, mpl # imports for visualisation
plt.style.use('seaborn') # This and the following lines customise the plot style
mpl.rcParams['font.family'] = 'serif'
%matplotlib inline
```

- More about customising the plot style: here.

Practical example using `DataFrame` class

- Plot the cumulative sum for each column of `df`:

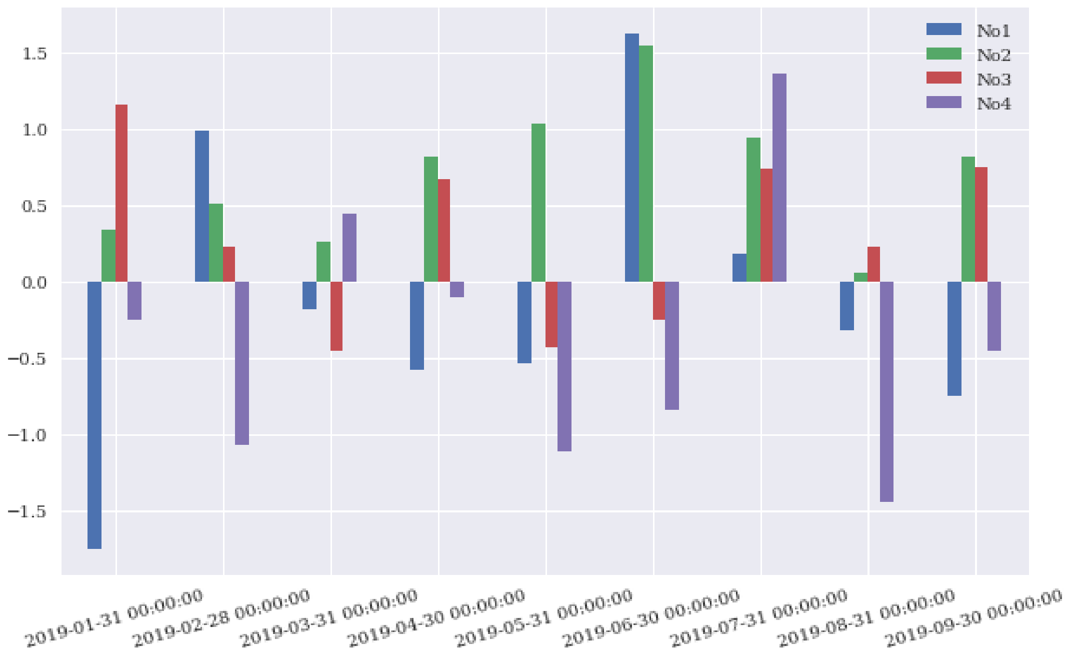
[101]:
```
df.cumsum().plot(lw = 2.0, figsize = (10,6));
```



Practical example using `DataFrame` class

- A bar chart:

[102]:
```
df.plot.bar(figsize = (10,6), rot = 15);
```



Practical example using `DataFrame` class

- Parameters of `plot()` method:

Parameter	Format	Description
`x`	label/position, default None	Only used when column values are x-ticks
`y`	label/position, default None	Only used when column values are y-ticks
`subplots`	boolean, default `False`	Plot columns in subplots
`sharex`	boolean, default `True`	Share the x-axis
`sharey`	boolean, default `False`	Share the y-axis
`use_index`	boolean, default `True`	Use `DataFrame.index` as x-ticks
`stacked`	boolean, default `False`	Stack (only for bar plots)
`sort_columns`	boolean, default `False`	Sort columns alphabetically before plotting
`title`	string, default None	Title for the plot
`grid`	boolean, default `False`	Show horizontal and vertical grid lines
`legend`	boolean, default `True`	Show legend of labels
`ax`	`matplotlib` axis object	`matplotlib` axis object to use for plotting
`style`	string or list/dictionary	Line plotting style (for each column)
`kind`	string (e.g., `"line"`, `"bar"`, `"barh"`, `"kde"`, `"density"`)	Type of plot
`logx`	boolean, default `False`	Use logarithmic scaling of x-axis
`logy`	boolean, default `False`	Use logarithmic scaling of y-axis
`xticks`	sequence, default `Index`	X-ticks for the plot

Parameters of plot method

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Parameters of `plot()` method:

```
<table>
    <td align=top><img src="pics/plot_2.png" alt="plot" width="600"/></td>
</table>
```

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Useful functions:

[103]:
```python
df.info() # provide basic information
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 9 entries, 2019-01-31 to 2019-09-30
Freq: M
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   No1     9 non-null      float64
 1   No2     9 non-null      float64
 2   No3     9 non-null      float64
 3   No4     9 non-null      float64
dtypes: float64(4)
memory usage: 360.0 bytes
```

Practical example using `DataFrame` class

[104]:
```python
df.sum()
```

[104]:
```
No1   -1.351906
No2    6.309298
No3    2.602739
No4   -3.490089
dtype: float64
```

[105]:
```python
df.mean(axis=0) # column-wise mean
```

[105]:
```
No1   -0.150212
No2    0.701033
No3    0.289193
No4   -0.387788
dtype: float64
```

[106]:
```python
df.mean(axis=1) # row-wise mean
```

[106]:
```
2019-01-31   -0.126621
2019-02-28    0.161669
2019-03-31    0.010661
2019-04-30    0.200390
2019-05-31   -0.264500
2019-06-30    0.516568
2019-07-31    0.803539
2019-08-31   -0.372845
2019-09-30    0.088650
Freq: M, dtype: float64
```

Useful functions: `groupby()`

[107]:
```
df['Quarter'] = ['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2', 'Q3', 'Q3', 'Q3',]
```

[108]:
```
df
```

[108]:
```
                 No1       No2       No3       No4 Quarter
2019-01-31 -1.749765  0.342680  1.153036 -0.252436      Q1
2019-02-28  0.981321  0.514219  0.221180 -1.070043      Q1
2019-03-31 -0.189496  0.255001 -0.458027  0.435163      Q1
2019-04-30 -0.583595  0.816847  0.672721 -0.104411      Q2
2019-05-31 -0.531280  1.029733 -0.438136 -1.118318      Q2
2019-06-30  1.618982  1.541605 -0.251879 -0.842436      Q2
2019-07-31  0.184519  0.937082  0.731000  1.361556      Q3
2019-08-31 -0.326238  0.055676  0.222400 -1.443217      Q3
2019-09-30 -0.756352  0.816454  0.750445 -0.455947      Q3
```

Useful functions: `groupby()`

[109]:
```
groups = df.groupby('Quarter')
```

[110]:
```
groups.mean()
```

[110]:
```
              No1       No2       No3       No4
Quarter
Q1      -0.319314  0.370634  0.305396 -0.295772
Q2       0.168035  1.129395 -0.005765 -0.688388
Q3      -0.299357  0.603071  0.567948 -0.179203
```

[111]:
```
groups.max()
```

[111]:
```
              No1       No2       No3       No4
Quarter
Q1       0.981321  0.514219  1.153036  0.435163
Q2       1.618982  1.541605  0.672721 -0.104411
Q3       0.184519  0.937082  0.750445  1.361556
```

Useful functions: `groupby()`

[112]:
```
groups.aggregate([min, max]).round(3)
```

[112]:
```
           No1           No2           No3           No4
           min    max    min    max    min    max    min    max
Quarter
Q1      -1.750  0.981  0.255  0.514 -0.458  1.153 -1.070  0.435
Q2      -0.584  1.619  0.817  1.542 -0.438  0.673 -1.118 -0.104
Q3      -0.756  0.185  0.056  0.937  0.222  0.750 -1.443  1.362
```

Selecting and filtering data

- Logical operators can be used to filter data.
- First, construct a `DataFrame` filled with random numbers to work with.

[113]:
```
data = np.random.standard_normal((10,2))
```

[114]:
```
df = pd.DataFrame(data, columns = ['x', 'y'])
```

[115]:
```
df.head(2) # the first two rows
```

[115]:
```
          x         y
0  1.189622 -1.690617
1 -1.356399 -1.232435
```

[116]:
```
df.tail(2) # the last two rows
```

[116]:
```
          x         y
8 -0.940046 -0.827932
9  0.108863  0.507810
```

Selecting and filtering data

[117]:
```
(df['x'] > 1) & (df['y'] < 1) # check if value in x-column is greater than 1 and␣
↪value in y-column is smaller than 1
```

[117]:
```
0     True
1    False
2    False
3    False
4     True
5    False
6    False
7    False
8    False
9    False
dtype: bool
```

[118]:
```
df[df['x'] > 1]
```

[118]:
```
          x         y
0  1.189622 -1.690617
4  1.299748 -1.733096
```

[119]:
```
df.query('x > 1') # query()-method takes string as parameter
```

[119]:
```
          x         y
0  1.189622 -1.690617
4  1.299748 -1.733096
```

Selecting and filtering data

[120]:
```
(df > 1).head(3) # Find values greater than 1
```

[120]:
```
       x      y
0   True  False
1  False  False
2  False  False
```

[121]:
```
df[df > 1].head(3) # Select values greater than 1 and put NaN (not-a-number) in the␣
↪other entries
```

[121]:

```
          x    y
0  1.189622  NaN
1       NaN  NaN
2       NaN  NaN
```

Concatenation

- Adding rows from one data frame to another data frame can be done with `append()` or `concat()`:

[122]:

```
df1 = pd.DataFrame(['100', '200', '300', '400'],
                   index = ['a', 'b', 'c', 'd'],
                   columns = ['A',])

df2 = pd.DataFrame(['200', '150', '50'],
                   index = ['f', 'b','d'],
                   columns = ['B',])
```

Concatenation

[123]:

```
df1.append(df2, sort = False)
```

[123]:

```
     A    B
a  100  NaN
b  200  NaN
c  300  NaN
d  400  NaN
f  NaN  200
b  NaN  150
d  NaN   50
```

Concatenation

[124]:

```
pd.concat((df1, df2), sort = False)
```

[124]:

```
     A    B
a  100  NaN
b  200  NaN
c  300  NaN
d  400  NaN
f  NaN  200
b  NaN  150
d  NaN   50
```

Joining

- In Python, `join()` refers to joining `DataFrame` objects according to their index values.
- There are four different types of joining:

  1. `left` join
  2. `right` join
  3. `inner` join
  4. `outer` join

Joining

[125]:
```
df1.join(df2, how = 'left') # default join, based on indices of first dataset
```

[125]:
```
     A    B
a  100  NaN
b  200  150
c  300  NaN
d  400   50
```

[126]:
```
df1.join(df2, how = 'right') # based on indices of second dataset
```

[126]:
```
     A    B
f  NaN  200
b  200  150
d  400   50
```

Joining

[127]:
```
df1.join(df2, how = 'inner') # preserves those index values that are found in both␣
↪datasets
```

[127]:
```
     A    B
b  200  150
d  400   50
```

[128]:
```
df1.join(df2, how = 'outer') # preserves indices found in both datasets
```

[128]:
```
     A    B
a  100  NaN
b  200  150
c  300  NaN
d  400   50
f  NaN  200
```

Merging

- Join operations on `DataFrame` objects are based on the datasets indices.
- **Merging** operates on a shared column of two `DataFrame` objects.
- To demonstrate the usage we add a new column `C` to `df1` and `df2`.

[129]:
```
c = pd.Series([250, 150, 50], index = ['b', 'd', 'c'])
df1['C'] = c
df2['C'] = c
```

Merging

[130]:
```
df1
```

[130]:
```
     A      C
a  100    NaN
b  200  250.0
c  300   50.0
d  400  150.0
```

[131]:
```
df2
```

[131]:

	B	C
f	200	NaN
b	150	250.0
d	50	150.0

Merging

- By default, a merge takes place on a shared column, preserving only the shared data rows:

[132]:
```
pd.merge(df1, df2)
```

[132]:

	A	C	B
0	100	NaN	200
1	200	250.0	150
2	400	150.0	50

- An **outer merge** preserves all data rows:

[133]:
```
pd.merge(df1, df2, how = 'outer')
```

[133]:

	A	C	B
0	100	NaN	200
1	200	250.0	150
2	300	50.0	NaN
3	400	150.0	50

Merging

- There are numerous other ways to merge `DataFrame` objects.
- To learn more about merging in Python, see the pandas document on DataFrame merging.

[134]:
```
pd.merge(df1, df2, left_on = 'A', right_on = 'B')
```

[134]:

	A	C_x	B	C_y
0	200	250.0	200	NaN

[135]:
```
pd.merge(df1, df2, left_on = 'A', right_on = 'B', how = 'outer')
```

[135]:

	A	C_x	B	C_y
0	100	NaN	NaN	NaN
1	200	250.0	200	NaN
2	300	50.0	NaN	NaN
3	400	150.0	NaN	NaN
4	NaN	NaN	150	250.0
5	NaN	NaN	50	150.0