CFDS® – Chartered Financial Data Scientist

Introduction to Python

Prof. Dr. Natalie Packham

29 November 2023

Table of Contents

- 2 Numerical and Computational Foundations
    - 2.1 Arrays with Python lists
    - 2.2 NumPy arrays
    - 2.3 Data Analysis with pandas: DataFrame

Numerical and Computational Foundations

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:

In [127]:
```
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:

In [128]:
```
m = [v, v, v]
m
```

Out[128]:
```
[[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

In [129]:
```
m[1]
```

Out[129]:
```
[0.5, 0.75, 1.0, 1.5, 2.0]
```

In [130]:
```
m[1][0]
```

Out[130]:
```
0.5
```

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!

In [131]:
```
v = [0.5, 0.75, 1.0, 1.5, 2.0]
m = [v, v, v]
m
```

Out[131]:
```
[[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]]
```

In [132]:
```
v[0] = 'Python'
m
```

Out[132]:
```
[['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]]
```

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

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:

In [133]:
```
import numpy as np # import numpy
a = np.array([0, 0.5, 1, 1.5, 2]) # array(...) is the constructor for r
```

In [134]:
```
type(a)
```

Out[134]:
```
numpy.ndarray
```

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

In [135]:
```
b = np.array([0, 'test'])
b
```

Out[135]:
```
array(['0', 'test'], dtype='<U21')
```

In [136]:
```
type(b[0])
```

Out[136]:
```
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?`

In [137]:
```
np.arange?
```

In [138]:
```
np.arange(0, 2.5, 0.5)
```

Out[138]:
```
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.

In [139]:
```
a.sum()
```
Out[139]:
```
5.0
```
In [140]:
```
a.std()
```
Out[140]:
```
0.7071067811865476
```
In [141]:
```
a.cumsum()
```
Out[141]:
```
array([0. , 0.5, 1.5, 3. , 5. ])
```

Slicing 1d-Arrays

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

In [142]:
```
a
```

Out[142]:
```
array([0. , 0.5, 1. , 1.5, 2. ])
```

In [143]:
```
a[1]
```

Out[143]:
```
0.5
```

In [144]:
```
a[:2]
```

Out[144]:
```
array([0. , 0.5])
```

In [145]:
```
a[2:]
```

Out[145]:
```
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.

In [146]:
```
l = [0, 0.5, 1, 1.5, 2]
l
```

Out[146]:
```
[0, 0.5, 1, 1.5, 2]
```

In [147]:
```
2 * l
```

Out[147]:
```
[0, 0.5, 1, 1.5, 2, 0, 0.5, 1, 1.5, 2]
```

- `ndarray`:

In [148]:
```
a = np.arange(0, 7, 1)
a
```

Out[148]:
```
array([0, 1, 2, 3, 4, 5, 6])
```

In [149]:
```
2 * a
```

Out[149]:
```
array([ 0,  2,  4,  6,  8, 10, 12])
```

Mathematical operations (cont'd)

In [150]:
```
a + a
```
Out[150]:
```
array([ 0,  2,  4,  6,  8, 10, 12])
```

In [151]:
```
a ** 2
```
Out[151]:
```
array([ 0,  1,  4,  9, 16, 25, 36])
```

In [152]:
```
2 ** a
```
Out[152]:
```
array([ 1,  2,  4,  8, 16, 32, 64])
```

In [153]:
```
a ** a
```
Out[153]:
```
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:

In [154]:
```
np.exp(a)
```

Out[154]:
```
array([  1.        ,   2.71828183,   7.3890561 ,  20.08553692,
        54.59815003, 148.4131591 , 403.42879349])
```

In [155]:
```
np.sqrt(a)
```

Out[155]:
```
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:

In [156]:
```
b = np.array([a, 2 * a])
b
```

Out[156]:
```
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 0,  2,  4,  6,  8, 10, 12]])
```

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

In [157]:
```
b[0]
```

Out[157]:
```
array([0, 1, 2, 3, 4, 5, 6])
```

In [158]:
```
b[1,1]
```

Out[158]:
```
2
```

In [159]:
```
b[:,1]
```

Multiple dimensions

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

In [160]:
```
b.sum()
```
Out[160]:
```
63
```
In [161]:
```
b.sum(axis = 0)
```
Out[161]:
```
array([ 0,  3,  6,  9, 12, 15, 18])
```
In [162]:
```
b.sum(axis = 1)
```
Out[162]:
```
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:

```
In [163]: np.zeros((2,3), dtype = 'i') # array with two rows and three columns
```

```
Out[163]: array([[0, 0, 0],
                 [0, 0, 0]], dtype=int32)
```

```
In [164]: np.ones((2,3,4), dtype = 'i') # array dimensions: 2 x 3 x 4
```

```
Out[164]: 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)
```

```
In [165]: np.empty((2,3))
```

```
Out[165]: array([[1.        , 1.41421356, 1.73205081],
                 [2.        , 2.23606798, 2.44948974]])
```

Further methods for creating arrays

In [166]:
```
np.eye(3)
```

Out[166]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [167]:
```
np.diag(np.array([1,2,3,4]))
```

Out[167]:
```
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

NumPy dtype objects

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)

Source: Python for Finance, 2nd ed.

Logical operations

- NumPy Arrays can be compared, just like lists.

In [168]:
```
first = np.array([0, 1, 2, 3, 3, 6,])
second = np.array([0, 1, 2, 3, 4, 5,])
```

In [169]:
```
first > second
```

Out[169]:
```
array([False, False, False, False, False,  True])
```

In [170]:
```
first.sum() == second.sum()
```

Out[170]:
```
True
```

In [171]:
```
np.any([a == 4])
```

Out[171]:
```
True
```

In [172]:
```
np.all([a == 4])
```

Out[172]:
```
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):

In [173]:
```
ar = np.arange(15)
ar
```

Out[173]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
14])
```

In [174]:
```
ar.reshape((3,5))
```

Out[174]:
```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

In [175]:
```
ar
```

Out[175]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
14])
```

Reshape and resize

In [176]:
```
ar.resize((5,3))
```

In [177]:
```
ar
```

Out[177]:
```
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.

In [178]:
```
ar
```

Out[178]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

In [179]:
```
np.resize(ar, (3,3))
```

Out[179]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Reshape and resize

In [180]:
```
np.resize(ar, (5,5))
```

Out[180]:
```
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]])
```

In [181]:
```
a.shape # returns the array's dimensions
```

Out[181]:
```
(7,)
```

Further operations

- Transpose:

In [182]:
```
g = np.arange(0,  6)
g.resize(2,3)
g
```

Out[182]:
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [183]:
```
g.T
```

Out[183]:
```
array([[0, 3],
       [1, 4],
       [2, 5]])
```

- Flattening:

In [184]:
```
g.flatten()
```

Out[184]:
```
array([0, 1, 2, 3, 4, 5])
```

Further operations

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

In [185]:
```
b = np.ones((2,3))
```

In [186]:
```
np.vstack((g, b))
```

Out[186]:
```
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.

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 **da**ta.
- We cover the following data structures:

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

Source: Python for Finance, 2nd ed.

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.

In [187]:
```
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
```

In [188]:
```
df
```

Out[188]:

	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:

In [189]:
```
df.index
```

Out[189]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

In [190]:
```
df.columns
```

Out[190]:
```
Index(['numbers'], dtype='object')
```

Simple operations

In [191]:
```
df.loc['c'] # selects value corresponding to index c
```

Out[191]:
```
numbers    30
Name: c, dtype: int64
```

In [192]:
```
df.loc[['a', 'd']] # selects values correponding t indices a and d
```

Out[192]:

	numbers
a	10
d	40

In [193]:
```
df.iloc[1:3] # select second and third rows
```

Out[193]:

	numbers
b	20
c	30

Simple operations

In [194]:
```
df.sum()
```

Out[194]:
```
numbers    100
dtype: int64
```

- Vectorised operations as with `ndarray`:

In [195]:
```
df ** 2
```

Out[195]:

	numbers
a	100
b	400
c	900
d	1600

Extending `DataFrame` objects

In [196]:
```
df['floats'] = (1.5, 2.5, 3.5, 4.5) # adds a new column
```

In [197]:
```
df
```

Out[197]:

	numbers	floats
a	10	1.5
b	20	2.5
c	30	3.5
d	40	4.5

In [198]:
```
df['floats']
```

Out[198]:
```
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:

In [199]:
```
df['names'] = pd.DataFrame(['Yves', 'Sandra', 'Lilli', 'Henry'],
                           index = ['d', 'a', 'b', 'c'])
```

In [200]:
```
df
```

Out[200]:

	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:

In [201]:
```
df = df.append(pd.DataFrame({'numbers': 100, 'floats': 5.75, 'names': '
                                index = ['y',]))
```

```
/var/folders/46/b127yp714m71zfmt9j7_lhwh0000gq/T/ipykernel_5161
5/4096332438.py:1: FutureWarning: The frame.append method is de
precated and will be removed from pandas in a future version. U
se pandas.concat instead.
  df = df.append(pd.DataFrame({'numbers': 100, 'floats': 5.75,
'names': 'Jill'},
```

In [202]:
```
df
```

Out[202]:

	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:

In [203]:
```
df.append({'numbers': 100, 'floats': 5.75, 'names': 'Jill'}, ignore_inc
```

```
/var/folders/46/b127yp714m71zfmt9j7_lhwh0000gq/T/ipykernel_5161
5/1910716993.py:1: FutureWarning: The frame.append method is de
precated and will be removed from pandas in a future version. U
se pandas.concat instead.
  df.append({'numbers': 100, 'floats': 5.75, 'names': 'Jill'},
ignore_index=True)
```

Out[203]:

	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:

In [204]:
```
df = df.append(pd.DataFrame({'names': 'Liz'},
                            index = ['z']),
                            sort = False)
```

```
/var/folders/46/b127yp714m71zfmt9j7_lhwh0000gq/T/ipykernel_5161
5/2026836976.py:1: FutureWarning: The frame.append method is de
precated and will be removed from pandas in a future version. U
se pandas.concat instead.
  df = df.append(pd.DataFrame({'names': 'Liz'},
```

In [205]:
```
df
```

Out[205]:

	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:

In [206]:
```
df[['numbers', 'floats']].sum()
```

Out[206]:
```
numbers    200.00
floats      17.75
dtype: float64
```

In [207]:
```
df['numbers'].var()
```

Out[207]:
```
1250.0
```

In [208]:
```
df['numbers'].max()
```

Out[208]:
```
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.

In [209]:
```
import numpy as np
import pandas as pd
np.random.seed(100)
a = np.random.standard_normal((9,4))
a
```

Out[209]:
```
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]])
```

In [210]:
```
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

In [211]:
```
df
```

Out[211]:

	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

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.

In [212]:
```
df.columns = ['No1', 'No2', 'No3', 'No4']
```

In [213]:
```
df
```

Out[213]:

	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

In [214]:
```
df['No3'].values.flatten()
```

Out[214]:
```
array([ 1.1530358 ,  0.22117967, -0.45802699,  0.67272081, -0.4
3813562,
```

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.

In [215]:
```
dates = pd.date_range('2019-1-1', periods = 9, freq = 'M')
dates
```

Out[215]:
```
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

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Frequency parameter of `date_range()` function:

Alias	Description
B	Business day frequency
C	Custom business day frequency (experimental)
D	Calendar day frequency
W	Weekly frequency
M	Month end frequency
BM	Business month end frequency

Alias	Description
MS	Month start frequency
BMS	Business month start frequency
Q	Quarter end frequency
BQ	Business quarter end frequency
QS	Quarter start frequency
BQS	Business quarter start frequency
A	Year end frequency
BA	Business year end frequency
AS	Year start frequency
BAS	Business year start frequency
H	Hourly frequency
T	Minutely frequency
S	Secondly frequency
L	Milliseconds
U	Microseconds

Practical example using `DataFrame` class

- Now set the row index to the dates:

In [216]:
```
df.index = dates

df
```

Out[216]:

	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:

In [217]:

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

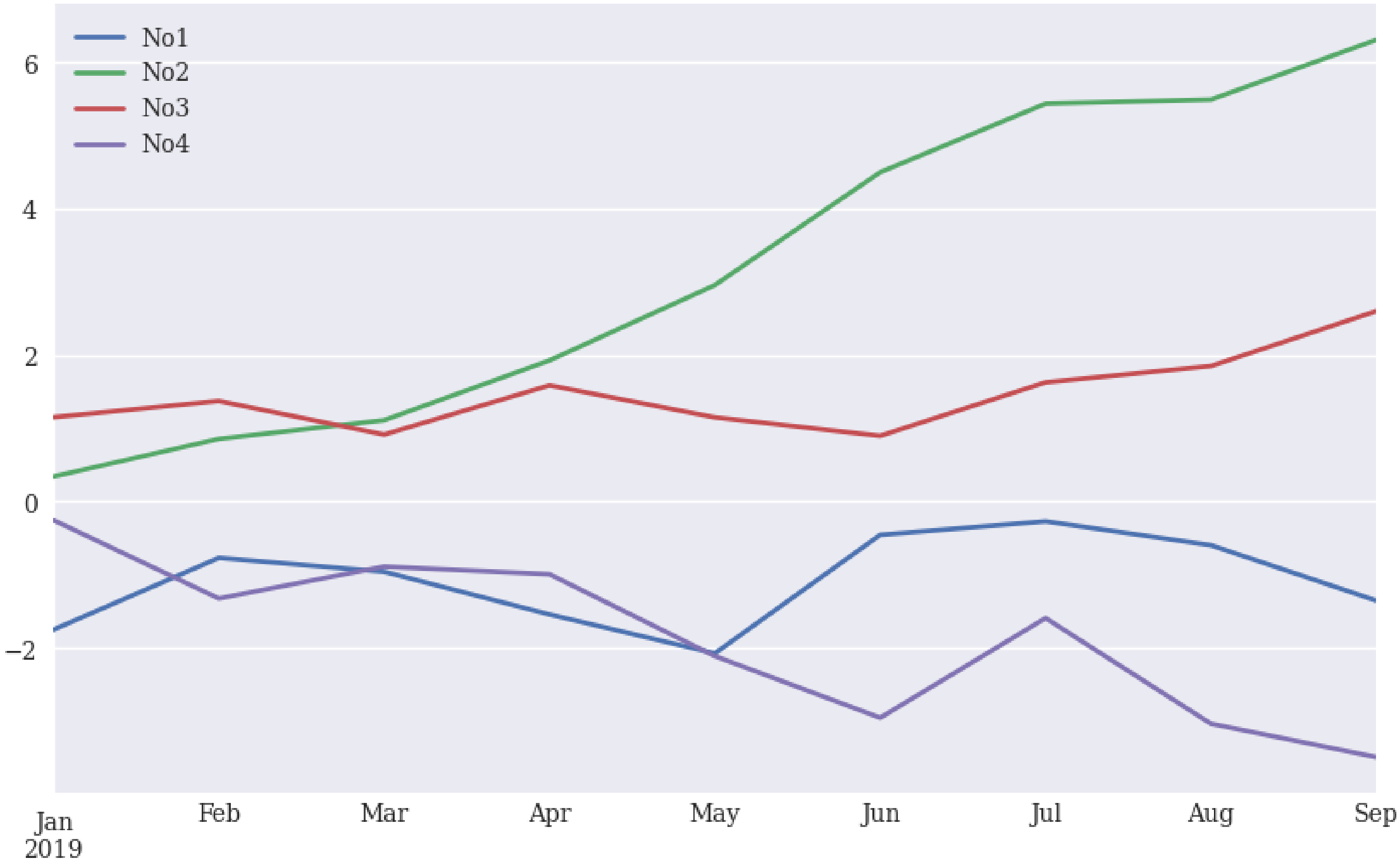
```
/var/folders/46/b127yp714m71zfmt9j7_lhwh0000gq/T/ipykernel_5161
5/276358035.py:2: MatplotlibDeprecationWarning: The seaborn sty
les shipped by Matplotlib are deprecated since 3.6, as they no
longer correspond to the styles shipped by seaborn. However, th
ey will remain available as 'seaborn-v0_8-<style>'. Alternative
ly, directly use the seaborn API instead.
  plt.style.use('seaborn') # This and the following lines custo
mise the plot style
```

- More about customising the plot style: here.

Practical example using `DataFrame` class

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

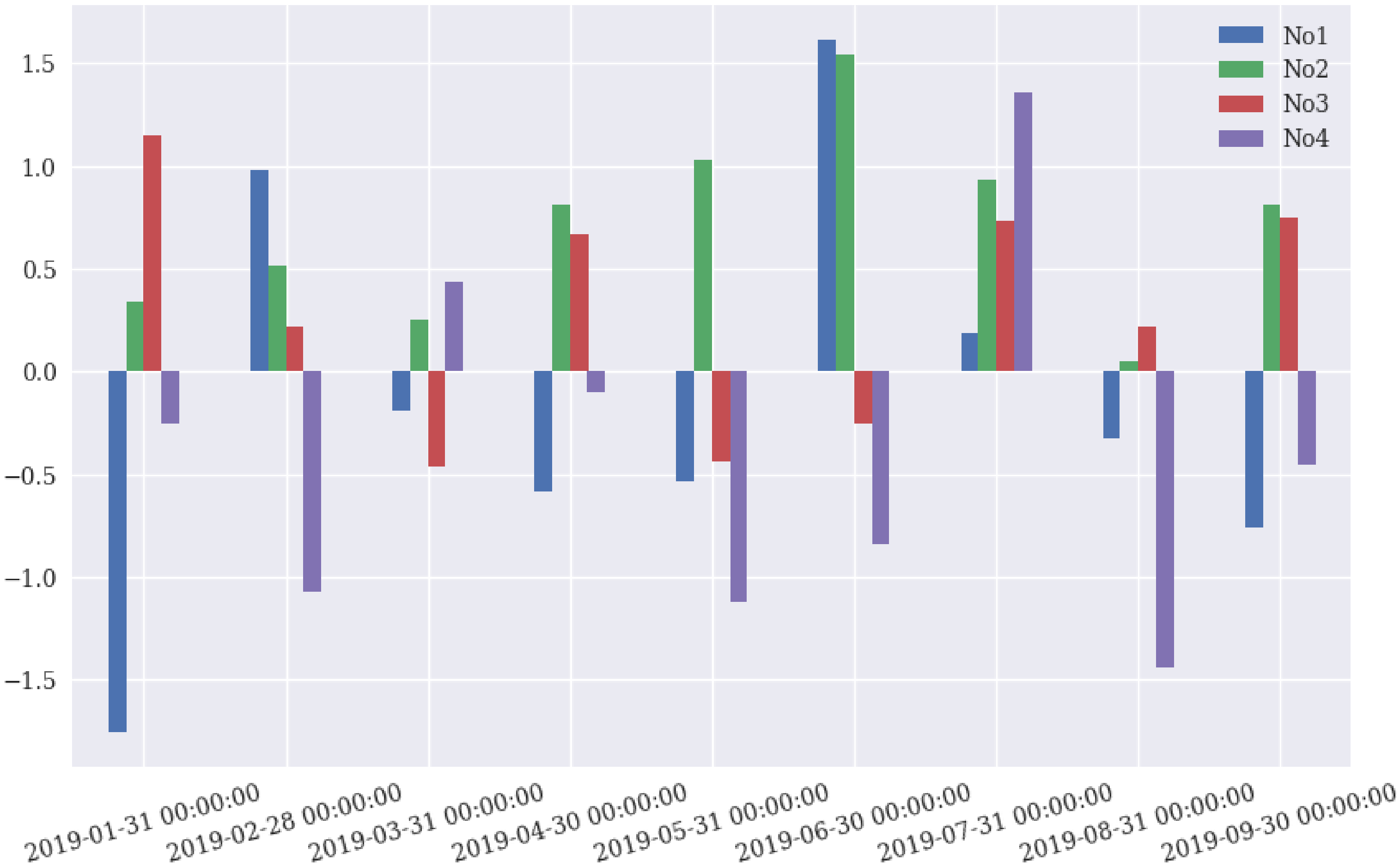
In [218]:
```
df.cumsum().plot(lw = 2.0, figsize = (10,6));
```


No1
No2
No3
No4
6
4
2
0
−2
Jan
2019
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep

Practical example using `DataFrame` class

- A bar chart:

In [219]:
```
df.plot.bar(figsize = (10,6), rot = 15);
```



No1
No2
No3
No4
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
2019-01-31 00:00:00
2019-02-28 00:00:00
2019-03-31 00:00:00
2019-04-30 00:00:00
2019-05-31 00:00:00
2019-06-30 00:00:00
2019-07-31 00:00:00
2019-08-31 00:00:00
2019-09-30 00:00:00

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

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Parameters of `plot()` method:

Parameter	Format	Description
`yticks`	sequence, default `Values`	Y-ticks for the plot
`xlim`	2-tuple, list	Boundaries for x-axis
`ylim`	2-tuple, list	Boundaries for y-axis
`rot`	integer, default `None`	Rotation of x-ticks
`secondary_y`	boolean/sequence, default `False`	Plot on secondary y-axis
`mark_right`	boolean, default `True`	Automatic labeling of secondary axis
`colormap`	string/colormap object, default `None`	Color map to use for plotting
`kwds`	keywords	Options to pass to `matplotlib`

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Useful functions:

In [220]:
```
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

In [221]:
```
df.sum()
```

Out[221]:
```
No1   -1.351906
No2    6.309298
No3    2.602739
No4   -3.490089
dtype: float64
```

In [222]:
```
df.mean(axis=0) # column-wise mean
```

Out[222]:
```
No1   -0.150212
No2    0.701033
No3    0.289193
No4   -0.387788
dtype: float64
```

In [223]:
```
df.mean(axis=1) # row-wise mean
```

Out[223]:
```
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
```

Advanced functions

- The `pandas` DataFrame is a very versatile object for storing data.
- More advanced functions (grouping, filtering, merging, joining) are explained below.
- This is for your reference as we will not have time to go through these in detail.
- By my own experience, it is sufficient to know about these operations and read about them when you need them.

Useful functions: `groupby()`

In [224]:
```
df['Quarter'] = ['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2', 'Q3', 'Q3', 'Q3',]
```

In [225]:
```
df
```

Out[225]:

	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()`

In [226]:
```
groups = df.groupby('Quarter')
```

In [227]:
```
groups.mean()
```

Out[227]:

	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

In [228]:
```
groups.max()
```

Out[228]:

	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()`

In [229]:
```
groups.aggregate([min, max]).round(3)
```

Out[229]:

	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.

In [230]:
```
data = np.random.standard_normal((10,2))
```

In [231]:
```
df = pd.DataFrame(data, columns = ['x', 'y'])
```

In [232]:
```
df.head(2) # the first two rows
```

Out[232]:

	x	y
0	1.189622	-1.690617
1	-1.356399	-1.232435

In [233]:
```
df.tail(2) # the last two rows
```

Out[233]:

	x	y
8	-0.940046	-0.827932
9	0.108863	0.507810

Selecting and filtering data

In [234]:
```
(df['x'] > 1) & (df['y'] < 1) # check if value in x-column is greater t
```

Out[234]:
```
0     True
1    False
2    False
3    False
4     True
5    False
6    False
7    False
8    False
9    False
dtype: bool
```

In [235]:
```
df[df['x'] > 1]
```

Out[235]:

	x	y
0	1.189622	-1.690617
4	1.299748	-1.733096

In [236]:
```
df.query('x > 1') # query()-method takes string as parameter
```

Out[236]:

	x	y
0	1.189622	-1.690617
4	1.299748	-1.733096

Selecting and filtering data

In [237]:
```
(df > 1).head(3) # Find values greater than 1
```

Out[237]:

	x	y
0	True	False
1	False	False
2	False	False

In [238]:
```
df[df > 1].head(3) # Select values greater than 1 and put NaN (not-a-nu
```

Out[238]:

	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()`:

In [239]:
```
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

In [240]:
```
df1.append(df2, sort = False)
```

```
/var/folders/46/b127yp714m71zfmt9j7_lhwh0000gq/T/ipykernel_5161
5/365867630.py:1: FutureWarning: The frame.append method is dep
recated and will be removed from pandas in a future version. Us
e pandas.concat instead.
  df1.append(df2, sort = False)
```

Out[240]:

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

Concatenation

In [241]:
```
pd.concat((df1, df2), sort = False)
```

Out[241]:

	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

In [242]:
```
df1.join(df2, how = 'left') # default join, based on indices of first c
```

Out[242]:

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

In [243]:
```
df1.join(df2, how = 'right') # based on indices of second dataset
```

Out[243]:

	A	B
f	NaN	200
b	200	150
d	400	50

Joining

In [244]:
```
df1.join(df2, how = 'inner') # preserves those index values that are fo
```

Out[244]:

	A	B
b	200	150
d	400	50

In [245]:
```
df1.join(df2, how = 'outer') # preserves indices found in both datasets
```

Out[245]:

	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`.

In [246]:
```
c = pd.Series([250, 150, 50], index = ['b', 'd', 'c'])
df1['C'] = c
df2['C'] = c
```

Merging

In [247]:
```
df1
```

Out[247]:

	A	C
a	100	NaN
b	200	250.0
c	300	50.0
d	400	150.0

In [248]:
```
df2
```

Out[248]:

	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:

In [249]:
```
pd.merge(df1, df2)
```

Out[249]:

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

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

In [250]:
```
pd.merge(df1, df2, how = 'outer')
```

Out[250]:

	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.

In [251]:
```
pd.merge(df1, df2, left_on = 'A', right_on = 'B')
```

Out[251]:

	A	C_x	B	C_y
0	200	250.0	200	NaN

In [252]:
```
pd.merge(df1, df2, left_on = 'A', right_on = 'B', how = 'outer')
```

Out[252]:

	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