CFDS® – Chartered Financial Data Scientist

Introduction to Python

Prof. Dr. Natalie Packham

4 December 2025

Table of Contents

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

`list` objects

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

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

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

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

NumPy arrays

NumPy arrays

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

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:

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

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

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

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

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

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

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

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

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

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

Slicing 1d-Arrays

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

```
In [ ]: a
```

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

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

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

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

- `ndarray`:

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

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

Mathematical operations (cont'd)

```
In [ ]: a + a
```

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

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

```
In [ ]: a ** a
```

Universal functions in NumPy

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

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

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

Multiple dimensions

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

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

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

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

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

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

Multiple dimensions

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

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

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

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

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

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

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

Further methods for creating arrays

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

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

NumPy dtype objects

dtype object

Source: Python for Finance, 2nd ed.

Logical operations

- NumPy Arrays can be compared, just like lists.

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

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

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

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

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

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

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

```
In [ ]: ar
```

Reshape and resize

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

```
In [ ]: ar
```

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 [ ]: ar
```

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

Reshape and resize

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

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

Further operations

- Transpose:

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

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

- Flattening:

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

Further operations

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

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

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

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:

Pandas datatypes

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 [ ]:
```python
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 ent
df
```

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 [ ]:
```python
df.index
```

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

Simple operations

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

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

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

Simple operations

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

- Vectorised operations as with `ndarray`:

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

Extending `DataFrame` objects

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

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

Extending `DataFrame` objects

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

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

Extending `DataFrame` objects

- Appending data:

```
In [ ]: df = pd.concat([df, pd.DataFrame({'numbers': [100], 'floats': [5.75
        df
```

Extending `DataFrame` objects

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

```
In [ ]: df = pd.concat([df, pd.DataFrame([{'numbers': 100, 'floats': 5.75,
        df
```

Extending `DataFrame` objects

- Appending with missing data:

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

Mathematical operations on Data Frames

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

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

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

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

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

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

Practical example using `DataFrame` class

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

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.

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

In [ ]:
```
df
```

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

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

Practical example using `DataFrame` class

- Parameters of the `date_range()` function:

Date range parameters

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Frequency parameter of `date_range()` function:

Date range frequencies Date range frequencies

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Now set the row index to the dates:

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

df
```

Practical example using `DataFrame` class

- Next, we visualise the data:

In [ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```

- More about customising the plot style: here.

Practical example using `DataFrame` class

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

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

Practical example using `DataFrame` class

- A bar chart:

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

Practical example using `DataFrame` class

- Parameters of `plot()` method:

Parameters of plot method

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Parameters of `plot()` method:

Plot_parameters

Source: Python for Finance, 2nd ed.

Practical example using `DataFrame` class

- Useful functions:

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

Practical example using `DataFrame` class

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

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

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

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

In [ ]:
```
df
```

Useful functions: `groupby()`

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

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

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

Useful functions: `aggregate()`

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

Selecting and filtering data

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

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

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

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

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

Selecting and filtering data

In [ ]:
```
(df['x'] > 1) & (df['y'] < 1) # check if value in x-column is great
```

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

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

Selecting and filtering data

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

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

Concatenation

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

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

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

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

Joining

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

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

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

Merging

```
In [ ]: df1
```

```
In [ ]: df2
```

Merging

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

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

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

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

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

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