Introduction to Python

Objectives of this Lesson

1. Installing Third Party Libraries - pip
2. NumPy
3. MatPlotLib
4. Pandas

NumPy

NumPy is an acronym for "Numeric Python" or "Numerical Python".

It is an open source extension module for Python, which provides fast precompiled functions for mathematical and numerical routines.

Furthermore, NumPy enriches the programming language Python with powerful data structures for efficient computation of multi-dimensional arrays and matrices.

The most powerful feature of NumPy is n-dimensional array. This library also contains basic linear algebra functions, Fourier transforms, advanced random number capabilities and tools for integration with other low level languages like Fortran, C and C++.

NumPy

NumPy’s main object is the **homogeneous multidimensional array**.

It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.

In NumPy dimensions are called *axes*. The number of axes is *rank*.

For example, the coordinates of a point in 3D space `[1, 2, 1]` is an array of rank 1, because it has one axis. That axis has a length of 3.

In the example below, the array has rank 2 (it is 2-dimensional). The first dimension (axis) has a length of 2, the second dimension has a length of 3.

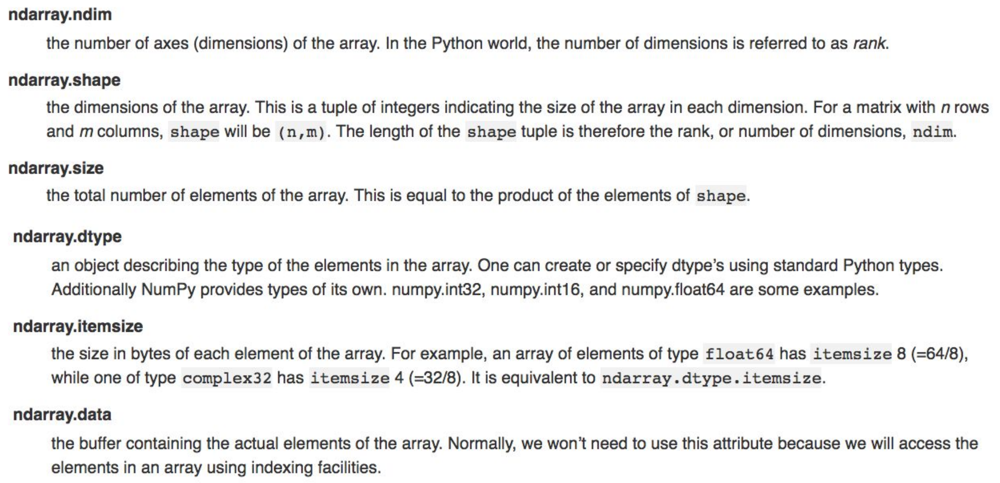
```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

NumPy

NumPy’s array class is called **`ndarray`**. It is also known by the alias **`array`**.

Note that **`numpy.array`** is not the same as the Standard Python Library class **`array.array`**, which only handles one-dimensional arrays and offers less functionality.

The more important attributes of an `ndarray` object are:



ndarray.ndim

the number of axes (dimensions) of the array. In the Python world, the number of dimensions is referred to as *rank*.

ndarray.shape

the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with *n* rows and *m* columns, `shape` will be `(n,m)`. The length of the `shape` tuple is therefore the rank, or number of dimensions, `ndim`.

ndarray.size

the total number of elements of the array. This is equal to the product of the elements of `shape`.

ndarray.dtype

an object describing the type of the elements in the array. One can create or specify dtype’s using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

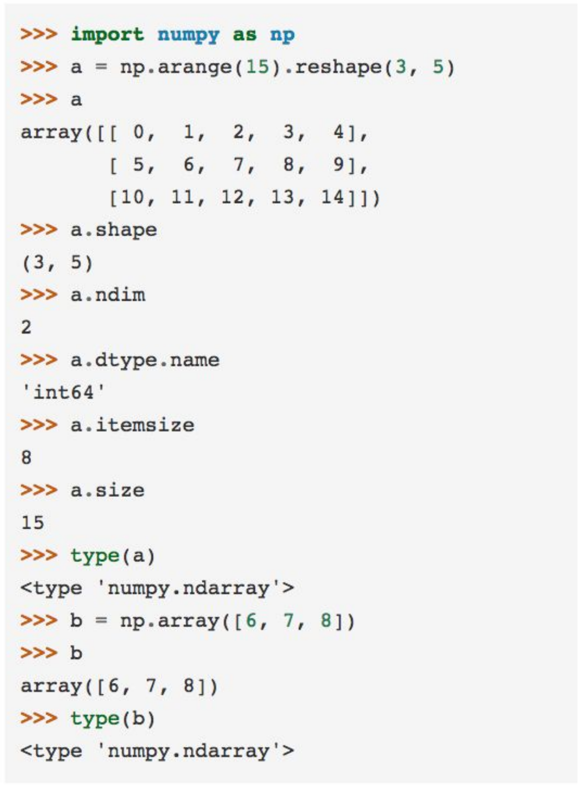
ndarray.itemsize

the size in bytes of each element of the array. For example, an array of elements of type `float64` has `itemsize` 8 (=64/8), while one of type `complex32` has `itemsize` 4 (=32/8). It is equivalent to `ndarray.dtype.itemsize`.

ndarray.data

the buffer containing the actual elements of the array. Normally, we won’t need to use this attribute because we will access the elements in an array using indexing facilities.

NumPy



```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<type 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<type 'numpy.ndarray'>
```

NumPy

You can create an array from a regular Python list or tuple using the `array` function.
The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
```

```
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

A frequent error consists in calling `array` with multiple numeric arguments, rather than providing a single list of numbers as an argument.

```
>>> a = np.array(1,2,3,4)    # WRONG
>>> a = np.array([1,2,3,4])  # RIGHT
```

NumPy

array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

NumPy

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function zeros creates an array full of zeros, the function ones creates an array full of onesBy default, the dtype of the created array is float64.

```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

```
>>> np.ones( (2,3,4), dtype=np.int16 )                # dtype can also be specified
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=int16)
```

NumPy

To create sequences of numbers, NumPy provides a function analogous to `range` that returns arrays instead of lists.

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
```

```
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([ 0. ,  0.3,  0.6,  0.9,  1.2,  1.5,  1.8])
```

When `arange` is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function `linspace` that receives as an argument the number of elements that we want, instead of the step:

```
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ,  1.25,  1.5 ,  1.75,  2.  ])
```

NumPy

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])


>>> c = a-b
>>> c
array([20, 29, 38, 47])


>>> b**2
array([0, 1, 4, 9])


>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])


>>> a<35
array([ True, True, False, False], dtype=bool)
```

NumPy

Unlike in many matrix languages, the product operator `*` operates elementwise in NumPy arrays. The matrix product can be performed using the `dot` function or method.

```
>>> A = np.array( [[1,1], [0,1]] )
>>> B = np.array( [[2,0], [3,4]] )
```

```
>>> A*B                          # elementwise product
array([[2, 0],[0, 4]])
```

```
>>> A.dot(B)                      # matrix product
array([[5, 4], [3, 4]])
```

```
>>> np.dot(A, B)                  # another matrix product
array([[5, 4],[3, 4]])
```

NumPy

Some operations, such as `+=` and `*=`, act in place to modify an existing array rather than create a new one.

```
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
```

```
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
```

```
>>> b += a
>>> b
array([[ 3.417022  ,  3.72032449,  3.00011437],
       [ 3.30233257,  3.14675589,  3.09233859]])
```

NumPy

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the `ndarray` class.

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.18626021,  0.34556073,  0.39676747],
       [ 0.53881673,  0.41919451,  0.6852195 ]])
```

```
>>> a.sum()
2.5718191614547998
```

```
>>> a.min()
0.1862602113776709
```

```
>>> a.max()
0.6852195003967595
```

NumPy

By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the `axis` parameter you can apply an operation along the specified axis of an array:

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

>>> b.sum(axis=0)                              # sum of each column
array([12, 15, 18, 21])

>>> b.min(axis=1)                              # min of each row
array([0, 4, 8])

>>> b.cumsum(axis=1)                           # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

NumPy

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
>>> a = np.arange(10)**3
>>> a
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])


>>> a[2]
8


>>> a[2:5]
array([ 8, 27, 64])


>>> a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000; from start to position 6, exclusive, set every 2nd element to -1000
>>> a
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,   729])
>>> a[ : :-1]                                 # reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
>>> for i in a:
...     print(i**(1/3.))


nan
1.0
nan
3.0
...
```

NumPy

Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas:

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
```

```
>>> b[2,3]
23
>>> b[0:5, 1]                       # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]                        # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]                      # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

NumPy

Iterating over multidimensional arrays is done with respect to the first axis:

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

However, if one wants to perform an operation on each element in the array, one can use the `flat` attribute which is an iterator over all the elements of the array:

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
10
11
12
13
```

MatPlotLib

Matplotlib is a plotting library.

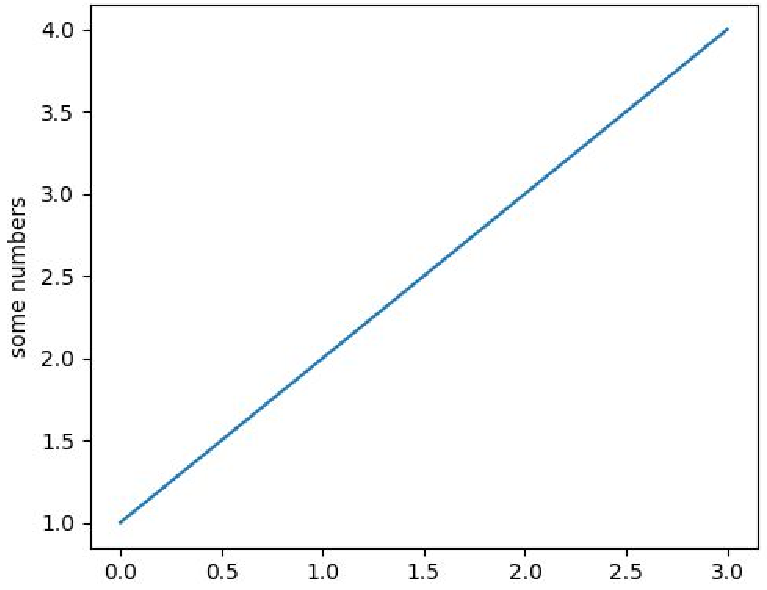
matplotlib.pyplot is a collection of command style functions that make matplotlib work like MATLAB.
Matplotlib is widely considered to be a perfect alternative to MATLAB, if it is used in combination with Numpy and Scipy.
And whereas MATLAB is expensive and closed source, Matplotlib is free and open source code.

Matplotlib is object-oriented and can be used in an object oriented way.
Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

MatPlotLib

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```

If you provide a single list or array to the `plot()`command,
matplotlib assumes it is a sequence of y values,
and automatically generates the x values for you.
Since python ranges start with 0,
the default x vector has the same length as y but starts with 0.
Hence the x data are `[0,1,2,3]`.



MatPlotLib

`plot()` is a versatile command, and will take an arbitrary number of arguments.
For example, to plot x versus y, you can issue the command:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and linetype of the plot.
The default format string is ‘b-‘, which is a solid blue line.
For example, to plot the above with red circles, you would issue

```
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
```

The `axis()` command in the example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the viewport of the axes.

MatPlotLib

Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally.
The example below illustrates a plotting several lines with different format styles in one command using arrays.

```
import numpy as np
import matplotlib.pyplot as plt

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```

MatPlotLib

Pyplot has the concept of the current figure and the current axes.
All plotting commands apply to the current axes.
The function gca() returns the current axes, and gcf() returns the current figure.

```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
```

The subplot() command specifies `numrows, numcols, fignum` where `fignum` ranges from 1 to `numrows*numcols`. The commas in the `subplot` command are optional if `numrows*numcols<10`. So `subplot(211)` is identical to `subplot(2, 1, 1)`

MatPlotLib

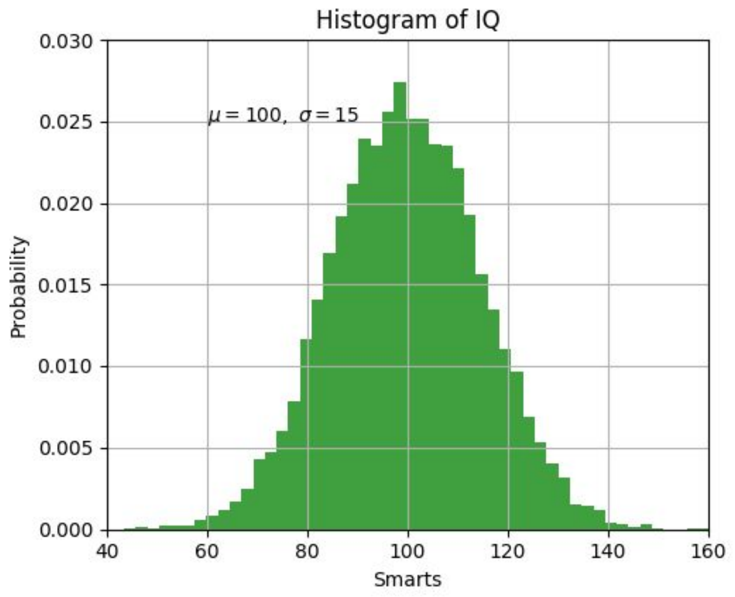
You can create multiple figures by using multiple `figure()` calls with an increasing figure number. Of course, each figure can contain as many axes and subplots as your heart desires:

```
import matplotlib.pyplot as plt
plt.figure(1)                # the first figure
plt.subplot(211)             # the first subplot in the first figure
plt.plot([1, 2, 3])
plt.subplot(212)             # the second subplot in the first figure
plt.plot([4, 5, 6])


plt.figure(2)                # a second figure
plt.plot([4, 5, 6])          # creates a subplot(111) by default

plt.figure(1)                # figure 1 current; subplot(212) still current
plt.subplot(211)             # make subplot(211) in figure1 current
plt.title('Easy as 1, 2, 3') # subplot 211 title
```

MatPlotLib

The `text()` command can be used to add text in an arbitrary location, and the `xlabel()`, `ylabel()` and `title()` are used to add text in the indicated locations

```
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
```



MatPlotLib

`matplotlib.pyplot` supports not only linear axis scales, but also logarithmic and logit scales.
This is commonly used if data spans many orders of magnitude.
Changing the scale of an axis is easy:

```
plt.xscale(‘log’)
```

Pandas

Pandas is a Python module, which is rounding up the capabilities of Numpy, Scipy and Matplotlab.

The word pandas is an acronym which is derived from "Python and data analysis" and "panel data".

There is often some confusion about whether Pandas is an alternative to Numpy, SciPy and Matplotlib.
The truth is that it is built on top of Numpy.
This means that Numpy is required by pandas.
Scipy and Matplotlib on the other hand are not required by pandas but they are extremely useful.
That's why the Pandas project lists them as "optional dependency".

There are two important data structures of Pandas:

- Series
- DataFrame

Pandas

Series

A Series is a one-dimensional labelled array-like object.
It is capable of holding any data type, e.g. integers, floats, strings, Python objects, and so on.
It can be seen as a data structure with two arrays: one functioning as the index, i.e. the labels, and the other one contains the actual data.

We define a simple Series object in the following example by instantiating a Pandas Series object with a list.

```
import pandas as pd
S = pd.Series([11, 28, 72, 3, 5, 8])
S
```

The above code returned the following output:

```
0    11
1    28
2    72
3     3
4     5
5     8
dtype: int64
```

Pandas created a default index starting with 0 going to 5, which is the length of the data minus 1.

Pandas

Series

We can directly access the index and the values of our Series S:

```
print(S.index)
print(S.values)
```

```
RangeIndex(start=0, stop=6, step=1)
[11 28 72  3  5  8]
```

Pandas

Series

We can start defining Series objects with individual indices:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
quantities = [20, 33, 52, 10]
S = pd.Series(quantities, index=fruits)
S
```

The previous Python code returned the following result:

```
apples      20
oranges     33
cherries    52
pears       10
dtype: int64
```

Pandas

Series
We can use arbitrary indices.
If we add two series with the same indices, we get a new series with the same index and the corresponding values will be added:

```
fruits = ['apples', 'oranges', 'cherries', 'pears']
S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits)
print(S + S2)
print("sum of S: ", sum(S))
```

```
apples      37
oranges     46
cherries    83
pears       42
dtype: int64
sum of S:  115
```

The indices do not have to be the same for the Series addition. The index will be the "union" of both indices. If an index doesn't occur in both Series, the value for this Series will be NaN:

```
fruits = ['peaches', 'oranges', 'cherries', 'pears']
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']
S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

```
cherries        83.0
```

Pandas

Series

The indices do not have to be the same for the Series addition. The index will be the "union" of both indices. If an index doesn't occur in both Series, the value for this Series will be NaN:

```
fruits = ['peaches', 'oranges', 'cherries', 'pears']
fruits2 = ['raspberries', 'oranges', 'cherries', 'pears']
S = pd.Series([20, 33, 52, 10], index=fruits)
S2 = pd.Series([17, 13, 31, 32], index=fruits2)
print(S + S2)
```

```
cherries       83.0
oranges        46.0
peaches         NaN
pears          42.0
raspberries     NaN
dtype: float64
```

Pandas

Series

It's possible to access single values of a Series or more than one value by a list of indices:

```
print(S['apples'])

20

print(S[['apples', 'oranges', 'cherries']])

apples      20
oranges     33
cherries    52
dtype: int64
```

Pandas

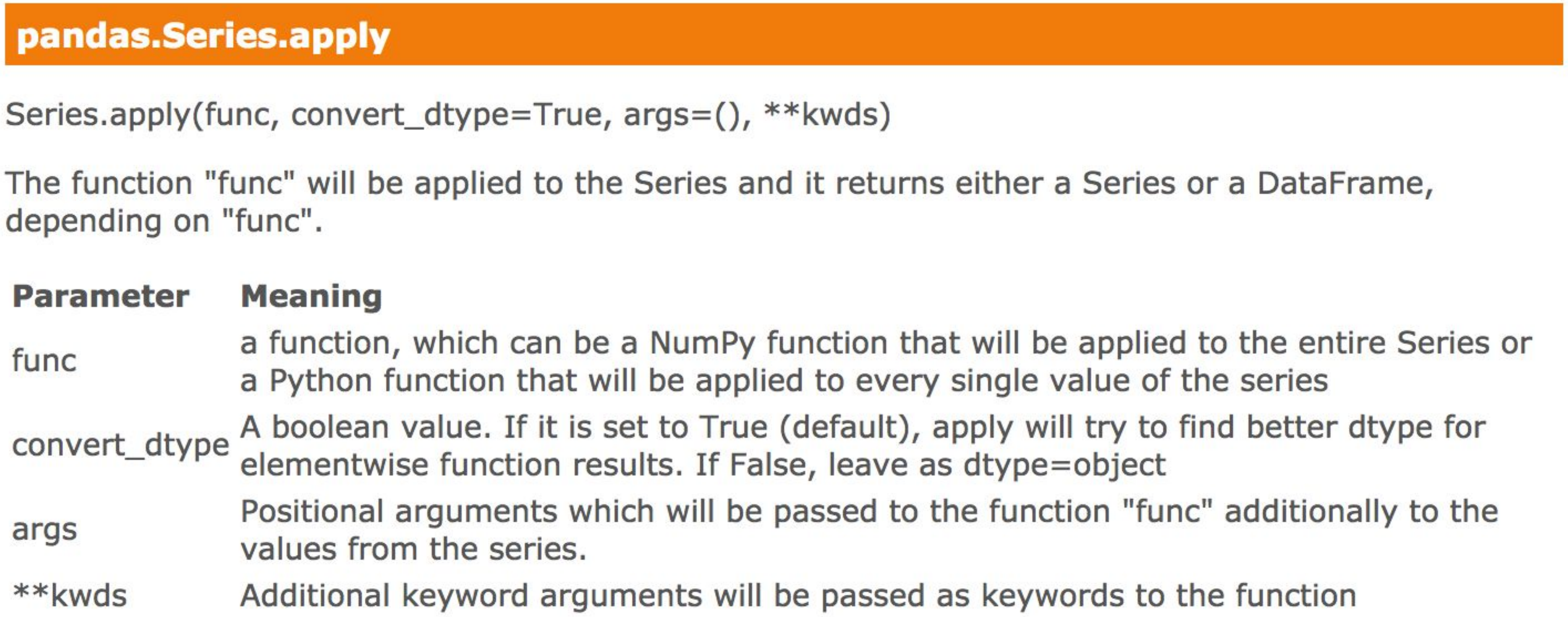
Series

Similar to Numpy we can use scalar operations or mathematical functions on a series:

```
import numpy as np
print((S + 3) * 4)
print("======================")
print(np.sin(S))
```

```
apples       92
oranges     144
cherries    220
pears        52
dtype: int64
======================
apples      0.912945
oranges     0.999912
cherries    0.986628
pears      -0.544021
dtype: float64
```

Pandas



pandas.Series.apply

Series.apply(func, convert_dtype=True, args=(), **kwds)

The function "func" will be applied to the Series and it returns either a Series or a DataFrame, depending on "func".

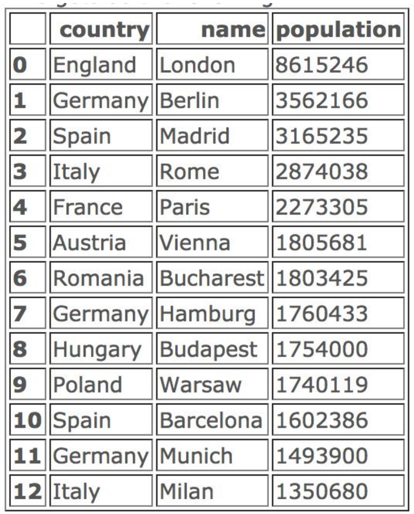
Parameter	Meaning
func	a function, which can be a NumPy function that will be applied to the entire Series or a Python function that will be applied to every single value of the series
convert_dtype	A boolean value. If it is set to True (default), apply will try to find better dtype for elementwise function results. If False, leave as dtype=object
args	Positional arguments which will be passed to the function "func" additionally to the values from the series.
**kwds	Additional keyword arguments will be passed as keywords to the function

Pandas

DataFrame

The underlying idea of a DataFrame is based on spreadsheets.
It contains an ordered collection of columns.
Each column consists of a unique data typye, but different columns can have different types, e.g. the first column may consist of integers, while the second one consists of boolean values and so on.
A DataFrame has a row and column index; it's like a dict of Series with a common index.

```
cities = {"name": ["London", "Berlin", "Madrid", "Rome",
                   "Paris", "Vienna", "Bucharest", "Hamburg",
                   "Budapest", "Warsaw", "Barcelona",
                   "Munich", "Milan"],
          "population": [8615246, 3562166, 3165235, 2874038,
                         2273305, 1805681, 1803425, 1760433,
                         1754000, 1740119, 1602386, 1493900,
                         1350680],
          "country": ["England", "Germany", "Spain", "Italy",
                      "France", "Austria", "Romania",
                      "Germany", "Hungary", "Poland", "Spain",
                      "Germany", "Italy"]}
city_frame = pd.DataFrame(cities)
city_frame
```



	country	name	population
0	England	London	8615246
1	Germany	Berlin	3562166
2	Spain	Madrid	3165235
3	Italy	Rome	2874038
4	France	Paris	2273305
5	Austria	Vienna	1805681
6	Romania	Bucharest	1803425
7	Germany	Hamburg	1760433
8	Hungary	Budapest	1754000
9	Poland	Warsaw	1740119
10	Spain	Barcelona	1602386
11	Germany	Munich	1493900
12	Italy	Milan	1350680

Pandas

DataFrame

Rearranging the Order of Columns

We can also define or rearrange the order of the columns.

```
city_frame = pd.DataFrame(cities, columns=["name", "country", "population"])
```

Pandas

DataFrame

We can calculate the sum of all the columns of a DataFrame or the sum of certain columns:

```
print(city_frame.sum())
```

```
name          LondonBerlinMadridRomeParisViennaBucharestHamb...
population                                             33800614
dtype: object
```

```
city_frame["population"].sum()
```

The previous code returned the following output:

```
33800614
```

Pandas

DataFrame

We can use "cumsum" to calculate the cumulative sum:

```
x = city_frame["population"].cumsum()
print(x)
```

```
country
England      8615246
Germany     12177412
Spain       15342647
Italy       18216685
France      20489990
Austria     22295671
Romania     24099096
Germany     25859529
Hungary     27613529
Poland      29353648
Spain       30956034
Germany     32449934
Italy       33800614
Name: population, dtype: int64
```

Pandas

Assigning New Values to Columns

x is a Pandas Series. We can reassign the previously calculated cumulative sums to the population column:

```
city_frame["population"] = x
```

Instead of replacing the values of the population column with the cumulative sum, we want to add the cumulative population sum as a new column with the name "cum_population".

```
city_frame = pd.DataFrame(cities, columns=["country", "population", "cum_population"])
```

The column "cum_population" is set to Nan, as we haven't provided any data for it.

We will assign now the cumulative sums to this column:

```
city_frame["cum_population"] = city_frame["population"].cumsum()
```

Pandas

Accessing the Columns of a DataFrame

There are two ways to access a column of a DataFrame. The result is in both cases a Series:

```
# in a dictionary-like way:
print(city_frame["population"])

# as an attribute
print(city_frame.population)
```

Accessing the Rows of a DataFrame

We can also access the rows directly. We access the info of the fourth city in the following way:

```
city_frame.ix["Hamburg"]
```

Pandas

Reading a csv File

We want to read in a csv file with the population data of all countries (July 2014). The delimiter of the file a a space and commas are used to separate groups of thousands in the numbers:

```
pop = pd.read_csv("countries_population.csv",
                  header=None,
                  names=["Country", "Population"],
                  index_col=0,
                  quotechar="'",
                  sep=" ",
                  thousands=",")
```