# Python for scientific computing

Python has extensive packages to help with data analysis:

- numpy: matrices, linear algebra, Fourier transform, pseudorandom number generators
- scipy: advanced linear algebra and maths, signal processing, statistics
- pandas: DataFrames, data wrangling and analysis
- matplotlib: visualizations such as line charts, histograms, scatter plots.

```
<IPython.core.display.HTML object>
```

## NumPy

NumPy is the fundamental package required for high performance scientific computing in Python. It provides:

- `ndarray`: fast and space-efficient n-dimensional numeric array with vectorized arithmetic operations
- Functions for fast operations on arrays without having to write loops
- Linear algebra, random number generation, Fourier transform
- Integrating code written in C, C++, and Fortran (for faster operations)

`pandas` provides a richer, simpler interface to many operations. We'll focus on using `ndarrays` here because they are heavily used in scikit-learn.

### ndarrays

There are several ways to create numpy arrays.

```
[2]: # Convert normal Python array to 1-dimensional numpy array
     np.array((1, 2, 53))
```

```
array([ 1,  2, 53])
```

```
[3]: # Convert sequences of sequences of sequences ... to n-dim array
     np.array([(1.5, 2, 3), (4, 5, 6)])
```

```
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

```
[4]: # Define element type at creation time
     np.array([[1, 2], [3, 4]], dtype=complex)
```

```
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Useful properties of ndarrays:

```
[5]: my_array = np.array([[1, 0, 3], [0, 1, 2]])
     my_array.ndim      # number of dimensions (axes), also called the rank
     my_array.shape     # a matrix with n rows and m columns has shape (n,m)
     my_array.size      # the total number of elements of the array
     my_array.dtype     # type of the elements in the array
     my_array.itemsize # the size in bytes of each element of the array
```

```
2
```

```
(2, 3)
```

```
6
```

```
dtype('int64')
```

```
8
```

Quick array creation.
It is cheaper to create an array with placeholders than extending it later.

```
[6]: np.ones(3) # Default type is float64
     np.zeros([2, 2])
     np.empty([2, 2]) # Fills the array with whatever sits in memory
     np.random.random((2,3))
     np.random.randint(5, size=(2, 4))
```

```
array([ 1.,  1.,  1.])
```

```
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
array([[ 0.404,  0.114,  0.042],
       [ 0.866,  0.599,  0.222]])
```

```
array([[4, 3, 2, 0],
       [2, 0, 2, 3]])
```

Create sequences of numbers

```
[7]: np.linspace(0, 1, num=4)   # Linearly distributed numbers between 0 and 1
     np.arange(0, 1, step=0.3)  # Fixed step size
     np.arange(12).reshape(3,4) # Create and reshape
     np.eye(4)                  # Identity matrix
```

```
array([ 0.   ,  0.333,  0.667,  1.   ])
```

```
array([ 0. ,  0.3,  0.6,  0.9])
```

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

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

### Basic Operations

Arithmetic operators on arrays apply elementwise. A new array is created and filled with the result. Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.

```
[8]: a = np.array([20, 30, 40, 50])
     b = np.arange(4)
     a, b     # Just printing
     a-b
     b**2
     a > 32
     a += 1
     a
```

```
(array([20, 30, 40, 50]), array([0, 1, 2, 3]))
```

```
array([20, 29, 38, 47])
```

```
array([0, 1, 4, 9])
```

```
array([False, False,  True,  True], dtype=bool)
```

```
array([21, 31, 41, 51])
```

The product operator * operates elementwise.
The matrix product can be performed using dot()

```
[9]: A, B = np.array([[1,1], [0,1]]), np.array([[2,0], [3,4]]) # assign multiple
     A
     B
     A * B
     np.dot(A, B)
```

```
array([[1, 1],
       [0, 1]])
```

```
array([[2, 0],
       [3, 4]])
```

```
array([[2, 0],
       [0, 4]])
```

```
array([[5, 4],
       [3, 4]])
```

Upcasting: Operations with arrays of different types choose the more general/precise one.

```
[10]: a = np.ones(3, dtype=np.int) # initialize to integers
      b = np.linspace(0, np.pi, 3) # default type is float
      a.dtype, b.dtype, (a + b).dtype
```

```
(dtype('int64'), dtype('float64'), dtype('float64'))
```

ndarrays have most unary operations (max,min,sum,...) built in

```
[11]: a = np.random.random((2,3))
      a
      a.sum(), a.min(), a.max()
```

```
array([[ 0.922,  0.843,  0.676],
       [ 0.22 ,  0.97 ,  0.599]])
```

```
(4.2293775981866508, 0.22019608034032734, 0.96968406069707935)
```

By specifying the axis parameter you can apply an operation along a specified axis of an array

```
[12]: b = np.arange(12).reshape(3,4)
      b
      b.sum(axis=0)
      b.sum(axis=1)
```

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

```
array([12, 15, 18, 21])
```

```
array([ 6, 22, 38])
```

### Universal Functions

NumPy provides familiar mathematical functions such as sin, cos, exp, sqrt, floor,... In NumPy, these are called "universal functions" (ufunc), and operate elementwise on an array, producing an array as output.

```
[13]: np.sin(np.arange(0, 10))
```

```
array([ 0.   ,  0.841,  0.909,  0.141, -0.757, -0.959, -0.279,  0.657,
        0.989,  0.412])
```

### Shape Manipulation

Transpose, flatten, reshape,...

```
[14]: a = np.floor(10*np.random.random((3,4)))
      a
      a.transpose()
      b = a.ravel() # flatten array
      b
      b.reshape(3, -1) # reshape in 2 rows (and as many columns as needed)
```

```
array([[ 6.,  5.,  2.,  5.],
       [ 1.,  8.,  2.,  3.],
       [ 2.,  6.,  3.,  8.]])
```

```
array([[ 6.,  1.,  2.],
       [ 5.,  8.,  6.],
       [ 2.,  2.,  3.],
       [ 5.,  3.,  8.]])
```

```
array([ 6.,  5.,  2.,  5.,  1.,  8.,  2.,  3.,  2.,  6.,  3.,  8.])
```

```
array([[ 6.,  5.,  2.,  5.],
       [ 1.,  8.,  2.,  3.],
       [ 2.,  6.,  3.,  8.]])
```

Arrays can be split and stacked together

```
[15]: a = np.floor(10*np.random.random((2,6)))
      a
      b, c = np.hsplit(a, 2) # Idem: vsplit for vertical splits
      b
      c
      np.hstack((b, c)) # Idenm: vstack for vertical stacks
```

```
array([[ 3.,  4.,  4.,  6.,  8.,  3.],
       [ 2.,  9.,  6.,  5.,  3.,  1.]])
```

```
array([[ 3.,  4.,  4.],
       [ 2.,  9.,  6.]])
```

```
array([[ 6.,  8.,  3.],
       [ 5.,  3.,  1.]])
```

```
array([[ 3.,  4.,  4.,  6.,  8.,  3.],
       [ 2.,  9.,  6.,  5.,  3.,  1.]])
```

### Indexing and Slicing

Arrays can be indexed and sliced using [start:stop:stepsize]. Defaults are [0:ndim:1]

```
[16]: a = np.arange(10)**2
      a
```

```
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

```
[17]: a[2]
```

```
4
```

```
[18]: a[3:10:2]
```

```
array([ 9, 25, 49, 81])
```

```
[19]: a[::-1] # Defaults are used if indices not stated
```

```
array([81, 64, 49, 36, 25, 16,  9,  4,  1,  0])
```

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

```
array([ 0,  4, 16, 36, 64])
```

For multi-dimensional arrays, axes are comma-separated: [x,y,z].

```
[21]: b = np.arange(16).reshape(4,4)
      b
      b[2,3] # row 2, column 3
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
11
```

```
[22]: b[0:3,1] # Values 0 to 3 in column 1
      b[ : ,1] # The whole column 1
```

```
array([1, 5, 9])
```

```
array([ 1,  5,  9, 13])
```

```
[23]: b[1:3, : ] # Rows 1:3, all columns
```

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
[24]: # Return the last row
      b[-1]
```

```
array([12, 13, 14, 15])
```

Note: dots (. . . ) represent as many colons (:) as needed * x[1,2,. . . ] = x[1,2,:,:,:] * x[. . . ,3] = x[:,:,:,:,3] * x[4,. . . ,5,:] = x[4,:,:,5,:]

Arrays can also be indexed by arrays of integers and booleans.

```
[25]: a = np.arange(12)**2
      i = np.array([ 1,1,3,8,5 ])
      a
      a[i]
```

```
array([  0,   1,   4,   9,  16,  25,  36,  49,  64,  81, 100, 121])
```

```
array([ 1,  1,  9, 64, 25])
```

A matrix of indices returns a matrix with the corresponding values.

```
[26]: j = np.array([[ 3, 4], [9, 7]])
      a[j]
```

```
array([[ 9, 16],
       [81, 49]])
```

With boolean indices we explicitly choose which items in the array we want and which ones we don't.

```
[27]: a = np.arange(12).reshape(3,4)
      a
      a[np.array([False,True,True]), :]
      b = a > 4
      b
      a[b]
```

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

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
```

```
array([ 5,  6,  7,  8,  9, 10, 11])
```

#### Iterating

Iterating is done with respect to the first axis:

```
[28]: for row in b:
          print(row)
```

```
[False False False False]
[False  True  True  True]
[ True  True  True  True]
```

Operations on each element can be done by flattening the array (or nested loops)

```
[29]: for element in b.flat: # flat returns an iterator
          print(element)
```

```
False
False
False
False
False
True
True
True
True
True
True
True
```

#### Copies and Views (or: how to shoot yourself in a foot)

Assigning an array to another variable does NOT create a copy

```
[30]: a = np.arange(12)
      b = a
      a
```

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

```
[31]: b[0] = -100
      b
```

```
array([-100,    1,    2,    3,    4,    5,    6,    7,    8,    9,   10,
         11])
```

```
[32]: a
```

```
array([-100,    1,    2,    3,    4,    5,    6,    7,    8,    9,   10,
         11])
```

The view() method creates a NEW array object that looks at the same data.

```
[33]: a = np.arange(12)
      a
      c = a.view()
      c.resize((2, 6))
      c
```

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

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

```
[34]: a[0] = 123
      c # c is also changed now
```

```
array([[123,   1,   2,   3,   4,   5],
       [  6,   7,   8,   9,  10,  11]])
```

Slicing an array returns a view of it.

```
[35]: c
      s = c[ : , 1:3]
      s[:] = 10
      s
      c
```

```
array([[123,   1,   2,   3,   4,   5],
       [  6,   7,   8,   9,  10,  11]])
```

```
array([[10, 10],
       [10, 10]])
```

```
array([[123,   10,   10,    3,    4,    5],
       [  6,   10,   10,    9,   10,   11]])
```

The copy() method makes a deep copy of the array and its data.

```
[36]: d = a.copy()
      d[0] = -42
      d
```

```
array([-42,  10,  10,   3,   4,   5,   6,  10,  10,   9,  10,  11])
```

```
[37]: a
```

```
array([123,  10,  10,   3,   4,   5,   6,  10,  10,   9,  10,  11])
```

#### Numpy: further reading

- Numpy Tutorial: http://wiki.scipy.org/Tentative_NumPy_Tutorial
- "Python for Data Analysis" by Wes McKinney (O'Reilly)

### SciPy

SciPy is a collection of packages for scientific computing, among others:

- scipy.integrate: numerical integration and differential equation solvers
- scipy.linalg: linear algebra routines and matrix decompositions
- scipy.optimize: function optimizers (minimizers) and root finding algorithms
- scipy.signal: signal processing tools
- scipy.sparse: sparse matrices and sparse linear system solvers
- scipy.stats: probability distributions, statistical tests, descriptive statistics

#### Sparse matrices

Sparse matrices are used in scikit-learn for (large) arrays that contain mostly zeros. You can convert a dense (numpy) matrix to a sparse matrix.

```
[38]: from scipy import sparse
      eye = np.eye(4)
      eye
      sparse_matrix = sparse.csr_matrix(eye) # Compressed Sparse Row matrix
      sparse_matrix
      print("{}".format(sparse_matrix))
```

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

```
<4x4 sparse matrix of type '<class 'numpy.float64'>'
with 4 stored elements in Compressed Sparse Row format>
```

```
  (0, 0)        1.0
  (1, 1)        1.0
  (2, 2)        1.0
  (3, 3)        1.0
```

When the data is too large, you can create a sparse matrix by passing the values and coordinates (COO format).

```
[39]: data = np.ones(4)                              # [1,1,1,1]
      row_indices = col_indices = np.arange(4)   # [1,2,3,4]
      eye_coo = sparse.coo_matrix((data, (row_indices, col_indices)))
      print("{}".format(eye_coo))
```

```
  (0, 0)        1.0
  (1, 1)        1.0
  (2, 2)        1.0
  (3, 3)        1.0
```

### Further reading

Check the SciPy reference guide for tutorials and examples of all SciPy capabilities.

## pandas

pandas is a Python library for data wrangling and analysis. It provides:

- `DataFrame`: a table, similar to an R DataFrame that holds any structured data
  - Every column can have its own data type (strings, dates, floats,...)
- A great range of methods to apply to this table (sorting, querying, joining,...)
- Imports data from a wide range of data formats (CVS, Excel) and databases (e.g. SQL)

### Series

A one-dimensional array of data (of any numpy type), with indexed values. It can be created by passing a Python list or dict, a numpy array, a csv file,...

```
[40]: import pandas as pd
      pd.Series([1,3,np.nan]) # Default integers are integers
      pd.Series([1,3,5], index=['a','b','c'])
      pd.Series({'a' : 1, 'b': 2, 'c': 3 }) # when given a dict, the keys will b
      pd.Series({'a' : 1, 'b': 2, 'c': 3 }, index = ['b', 'c', 'd']) # this will
```

```
0    1.0
1    3.0
2    NaN
dtype: float64
```

```
a    1
b    3
c    5
dtype: int64


a    1
b    2
c    3
dtype: int64


b    2.0
c    3.0
d    NaN
dtype: float64
```

Functions like a numpy array, however with index labels as indices

```
[41]: a = pd.Series({'a' : 1, 'b': 2, 'c': 3 })
      a
      a['b']        # Retrieves a value
      a[['a','b']] # and can also be sliced
```

```
a    1
b    2
c    3
dtype: int64


2


a    1
b    2
dtype: int64
```

numpy array operations on Series preserve the index value

```
[42]: a
      a[a > 1]
      a * 2
      np.sqrt(a)
```

```
a    1
b    2
c    3
dtype: int64
```

```
b    2
c    3
dtype: int64


a    2
b    4
c    6
dtype: int64


a    1.00
b    1.41
c    1.73
dtype: float64
```

Operations over multiple Series will align the indices

```
[43]: a = pd.Series({'John' : 1000, 'Mary': 2000, 'Andre': 3000 })
      b = pd.Series({'John' : 100, 'Andre': 200, 'Cecilia': 300 })
      a + b
```

```
Andre      3200.0
Cecilia       NaN
John       1100.0
Mary          NaN
dtype: float64
```

### DataFrame

A DataFrame is a tabular data structure with both a row and a column index. It can be created by passing a dict of arrays, a csv file,...

```
[44]: data = {'state': ['Ohio', 'Ohio', 'Nevada', 'Nevada'], 'year': [2000, 2001
      'pop': [1.5, 1.7, 2.4, 2.9]}
      pd.DataFrame(data)
      pd.DataFrame(data, columns=['year', 'state', 'pop', 'color']) # Will match
```

```
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  2.4  Nevada  2001
3  2.9  Nevada  2002


   year   state  pop color
0  2000    Ohio  1.5   NaN
1  2001    Ohio  1.7   NaN
2  2001  Nevada  2.4   NaN
3  2002  Nevada  2.9   NaN
```

It can be composed with a numpy array and row and column indices, and decomposed

```
[45]: dates = pd.date_range('20130101',periods=4)
      df = pd.DataFrame(np.random.randn(4,4),index=dates,columns=list('ABCD'))
      df
```

```
                A     B     C     D
2013-01-01 -2.42  1.89  0.53 -0.97
2013-01-02 -0.79  1.12  0.42 -0.35
2013-01-03 -0.38 -2.38 -0.94  0.39
2013-01-04  0.28 -1.16  0.09  1.54
```

```
[46]: df.index
      df.columns
      df.values
```

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04'], dtype='d
```

```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

```
array([[-2.417,  1.887,  0.526, -0.967],
       [-0.794,  1.121,  0.422, -0.35 ],
       [-0.384, -2.378, -0.935,  0.393],
       [ 0.275, -1.162,  0.088,  1.543]])
```

DataFrames can easily read/write data from/to files

- `read_csv(source)`: load CSV data from file or url
- `read_table(source, sep=',')`: load delimited data with separator
- `df.to_csv(target)`: writes the DataFrame to a file

```
[47]: dfs = pd.read_csv('data.csv')
      dfs
      dfs.set_value(0, 'a', 10)
      dfs.to_csv('data.csv', index=False) # Don't export the row index
```

```
    a   b   c   d message
0  10   2   3   4   hello
1   5   6   7   8   world
2   9  10  11  12     foo
```

```
    a   b   c   d message
0  10   2   3   4   hello
1   5   6   7   8   world
2   9  10  11  12     foo
```

### Simple operations

```
[48]: df.head() # First 5 rows
      df.tail() # Last 5 rows
```

| | A | B | C | D |
|---|---|---|---|---|
| 2013-01-01 | -2.42 | 1.89 | 0.53 | -0.97 |
| 2013-01-02 | -0.79 | 1.12 | 0.42 | -0.35 |
| 2013-01-03 | -0.38 | -2.38 | -0.94 | 0.39 |
| 2013-01-04 | 0.28 | -1.16 | 0.09 | 1.54 |

| | A | B | C | D |
|---|---|---|---|---|
| 2013-01-01 | -2.42 | 1.89 | 0.53 | -0.97 |
| 2013-01-02 | -0.79 | 1.12 | 0.42 | -0.35 |
| 2013-01-03 | -0.38 | -2.38 | -0.94 | 0.39 |
| 2013-01-04 | 0.28 | -1.16 | 0.09 | 1.54 |

```
[49]: # Quick stats
      df.describe()
```

| | A | B | C | D |
|---|---|---|---|---|
| count | 4.00 | 4.00 | 4.00 | 4.00 |
| mean | -0.83 | -0.13 | 0.03 | 0.15 |
| std | 1.15 | 1.98 | 0.67 | 1.08 |
| min | -2.42 | -2.38 | -0.94 | -0.97 |
| 25% | -1.20 | -1.47 | -0.17 | -0.50 |
| 50% | -0.59 | -0.02 | 0.25 | 0.02 |
| 75% | -0.22 | 1.31 | 0.45 | 0.68 |
| max | 0.28 | 1.89 | 0.53 | 1.54 |

```
[50]: # Transpose
      df.T
```

| | 2013-01-01 | 2013-01-02 | 2013-01-03 | 2013-01-04 |
|---|---|---|---|---|
| A | -2.42 | -0.79 | -0.38 | 0.28 |
| B | 1.89 | 1.12 | -2.38 | -1.16 |
| C | 0.53 | 0.42 | -0.94 | 0.09 |
| D | -0.97 | -0.35 | 0.39 | 1.54 |

```
[51]: df.sort_index(axis=1, ascending=False) # Sort by index labels
      df.sort(columns='B') # Sort by values
```

| | D | C | B | A |
|---|---|---|---|---|
| 2013-01-01 | -0.97 | 0.53 | 1.89 | -2.42 |
| 2013-01-02 | -0.35 | 0.42 | 1.12 | -0.79 |
| 2013-01-03 | 0.39 | -0.94 | -2.38 | -0.38 |
| 2013-01-04 | 1.54 | 0.09 | -1.16 | 0.28 |

```
                A     B     C     D
2013-01-03 -0.38 -2.38 -0.94  0.39
2013-01-04  0.28 -1.16  0.09  1.54
2013-01-02 -0.79  1.12  0.42 -0.35
2013-01-01 -2.42  1.89  0.53 -0.97
```

### Selecting and slicing

```
[52]: df['A'] # Get single column by label
      df.A    # Shorthand
```

```
2013-01-01   -2.42
2013-01-02   -0.79
2013-01-03   -0.38
2013-01-04    0.28
Freq: D, Name: A, dtype: float64
```

```
2013-01-01   -2.42
2013-01-02   -0.79
2013-01-03   -0.38
2013-01-04    0.28
Freq: D, Name: A, dtype: float64
```

```
[53]: df[0:2]            # Get rows by index number
      df.iloc[0:2,0:2] # Get rows and columns by index number
      df['20130102':'20130103']                # or row label
      df.loc['20130102':'20130103', ['A','B']] # or row and column label
      df.ix[0:2, ['A','B']]    # allows mixing integers and labels
```

```
                A     B     C     D
2013-01-01 -2.42  1.89  0.53 -0.97
2013-01-02 -0.79  1.12  0.42 -0.35
```

```
                A     B
2013-01-01 -2.42  1.89
2013-01-02 -0.79  1.12
```

```
                A     B     C     D
2013-01-02 -0.79  1.12  0.42 -0.35
2013-01-03 -0.38 -2.38 -0.94  0.39
```

```
                A     B
2013-01-02 -0.79  1.12
2013-01-03 -0.38 -2.38
```

```
                 A     B
2013-01-01 -2.42  1.89
2013-01-02 -0.79  1.12
```

query() retrieves data matching a boolean expression

```
[54]: df
      df.query('A > 0.4') # Identical to df[df.A > 0.4]
      df.query('A > B')   # Identical to df[df.A > df.B]
```

```
                 A     B     C     D
2013-01-01 -2.42  1.89  0.53 -0.97
2013-01-02 -0.79  1.12  0.42 -0.35
2013-01-03 -0.38 -2.38 -0.94  0.39
2013-01-04  0.28 -1.16  0.09  1.54
```

```
Empty DataFrame
Columns: [A, B, C, D]
Index: []
```

```
                 A     B     C     D
2013-01-03 -0.38 -2.38 -0.94  0.39
2013-01-04  0.28 -1.16  0.09  1.54
```

Note: similar to NumPy, indexing and slicing returns a *view* on the data. Use copy() to make a deep copy.

### Operations

DataFrames offer a wide range of operations: max, mean, min, sum, std,…

```
[55]: df.mean()       # Mean of all values per column
      df.mean(axis=1) # Other axis: means per row
```

```
A   -0.83
B   -0.13
C    0.03
D    0.15
dtype: float64
```

```
2013-01-01   -0.24
2013-01-02    0.10
2013-01-03   -0.83
2013-01-04    0.19
Freq: D, dtype: float64
```

All of numpy's universal functions also work with dataframes

```
[56]: np.abs(df)
```

```
                 A     B     C     D
2013-01-01  2.42  1.89  0.53  0.97
2013-01-02  0.79  1.12  0.42  0.35
2013-01-03  0.38  2.38  0.94  0.39
2013-01-04  0.28  1.16  0.09  1.54
```

Other (custom) functions can be applied with apply(funct)

```
[57]: df
      df.apply(np.max)
      df.apply(lambda x: x.max() - x.min())
```

```
                 A     B     C     D
2013-01-01 -2.42  1.89  0.53 -0.97
2013-01-02 -0.79  1.12  0.42 -0.35
2013-01-03 -0.38 -2.38 -0.94  0.39
2013-01-04  0.28 -1.16  0.09  1.54


A    0.28
B    1.89
C    0.53
D    1.54
dtype: float64


A    2.69
B    4.26
C    1.46
D    2.51
dtype: float64
```

Data can be aggregated with groupby()

```
[58]: df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar'], 'B' : ['one', 'one'
                          'C' : np.random.randn(4), 'D' : np.random.randn(4)})
      df
      df.groupby('A').sum()
      df.groupby(['A','B']).sum()
```

```
     A      B     C     D
0  foo    one  0.34  1.30
1  bar    one  0.84 -0.05
2  foo    two -0.95 -1.67
3  bar  three -0.33  1.45


        C     D
A
bar  0.50  1.40
foo -0.61 -0.38
```

```
              C     D
A   B
bar one     0.84 -0.05
    three -0.33  1.45
foo one     0.34  1.30
    two   -0.95 -1.67
```

### Data wrangling (some examples)

Merge: combine two dataframes based on common keys

```
[59]: df1 = pd.DataFrame({'key': ['b', 'b', 'a'], 'data1': range(3)})
      df2 = pd.DataFrame({'key': ['a', 'b'], 'data2': range(2)})
      df1
      df2
      pd.merge(df1, df2)
```

```
   data1 key
0      0   b
1      1   b
2      2   a
```

```
   data2 key
0      0   a
1      1   b
```

```
   data1 key  data2
0      0   b      1
1      1   b      1
2      2   a      0
```

Append: append one dataframe to another

```
[60]: df = pd.DataFrame(np.random.randn(2, 4))
      df
      s = pd.DataFrame(np.random.randn(1,4))
      s
      df.append(s, ignore_index=True)
```

```
      0     1     2     3
0 -0.17 -0.81  1.95 -1.08
1 -0.32  1.73 -0.48  0.90
```

```
      0     1     2     3
0 -0.34  0.32  1.16 -1.03
```

```
      0     1     2     3
0 -0.17 -0.81  1.95 -1.08
1 -0.32  1.73 -0.48  0.90
2 -0.34  0.32  1.16 -1.03
```

Remove duplicates

```
[61]: df = pd.DataFrame({'k1': ['one'] * 3, 'k2': [1, 1, 2]})
      df
      df.drop_duplicates()
```

```
    k1  k2
0  one   1
1  one   1
2  one   2
```

```
    k1  k2
0  one   1
2  one   2
```

Replace values

```
[62]: df = pd.DataFrame({'k1': [1, -1], 'k2': [-1, 2]}) # Say that -1 is a senti
      df
      df.replace(-1, np.nan)
```

```
   k1  k2
0   1  -1
1  -1   2
```

```
    k1   k2
0  1.0  NaN
1  NaN  2.0
```

Discretization and binning

```
[63]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
      bins = [18, 25, 35, 60, 100]
      cats = pd.cut(ages, bins)
      cats.labels
      pd.value_counts(cats)
```

```
array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

```
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

### Further reading

- Pandas docs: http://pandas.pydata.org/pandas-docs/stable/
- https://bitbucket.org/hrojas/learn-pandas
- Python for Data Analysis (O'Reilly) by Wes McKinney (the author of pandas)

## matplotlib

matplotlib is the primary scientific plotting library in Python. It provides:

- Publication-quality visualizations such as line charts, histograms, and scatter plots.
- Integration in pandas to make plotting much easier.
- Interactive plotting in Jupyter notebooks for quick visualizations.
  - Requires some setup. See preamble and %matplotlib.
- Many GUI backends, export to PDF, SVG, JPG, PNG, BMP, GIF, etc.
- Ecosystem of libraries for more advanced plotting, e.g. Seaborn

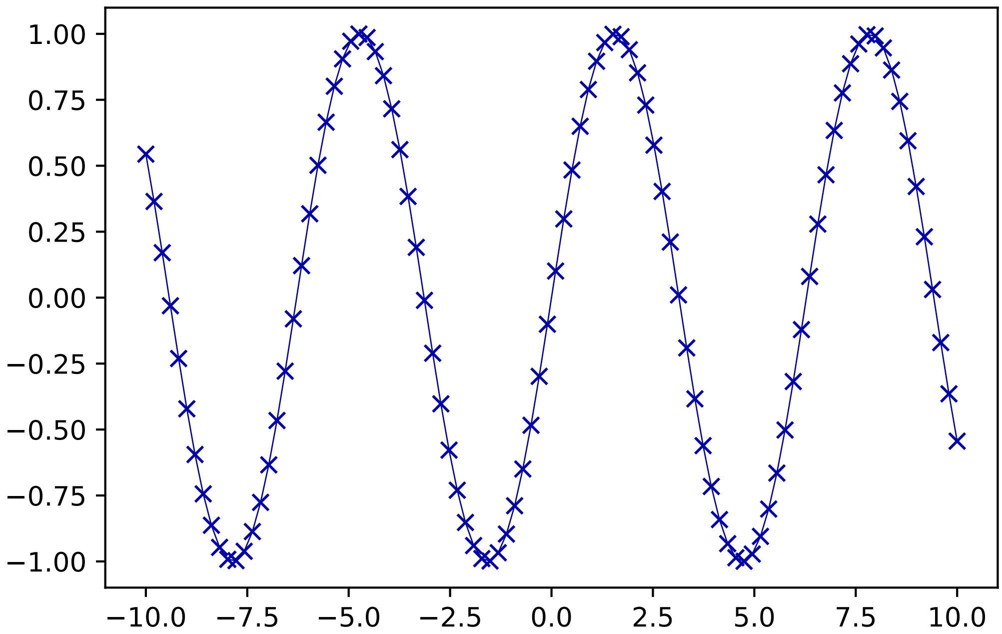
### Low-level usage

`plot()` is the main function to generate a plot (but many more exist):

```
plot(x, y)          Plot x vs y, default settings
plot(x, y, 'bo')    Plot x vs y, blue circle markers
plot(y, 'r+')       Plot y (x = array 0..N-1), red plusses
```

Every plotting function is completely customizable through a large set of options.

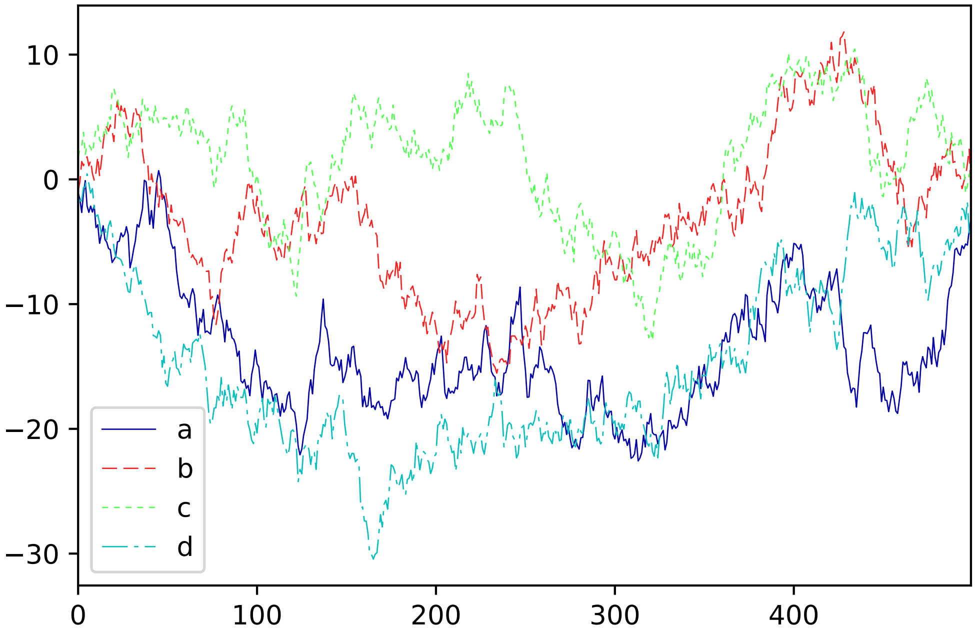
```
[71]: x = np.linspace(-10, 10, 100) # Sequence for X-axis
      y = np.sin(x) # sine values
      p = plt.plot(x, y, marker="x") # Line plot with marker x
```



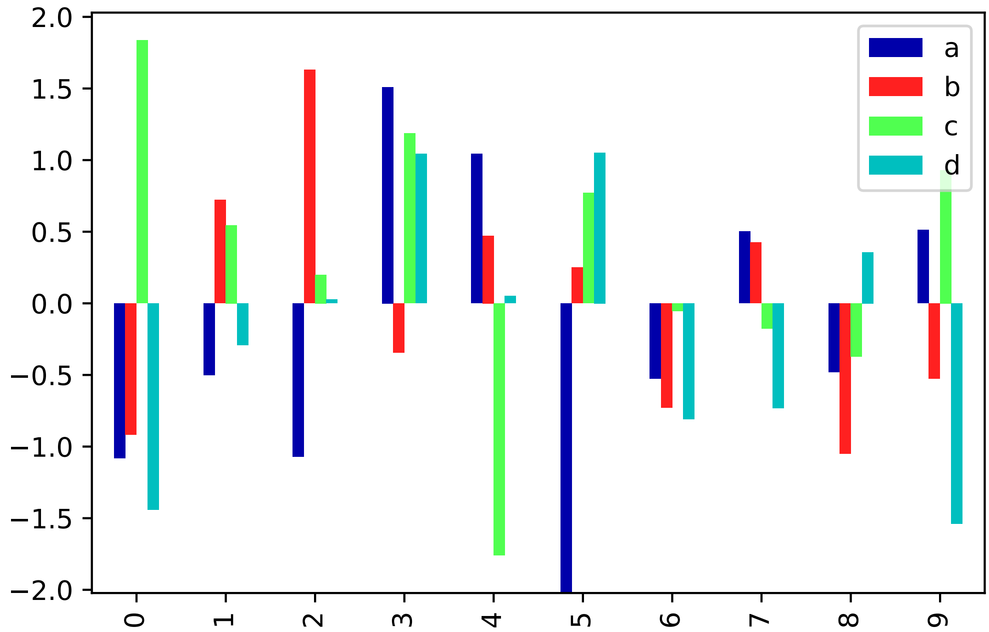
**pandas + matplotlib**

pandas DataFrames offer an easier, higher-level interface for matplotlib functions

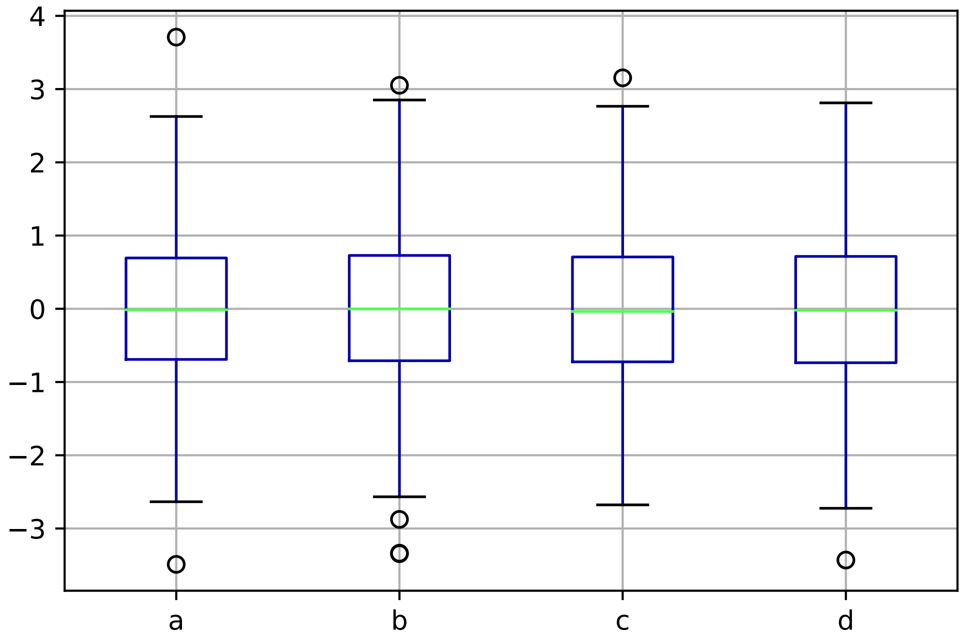
```
[65]: df = pd.DataFrame(np.random.randn(500, 4),
                        columns=['a', 'b', 'c', 'd']) # random 4D data
      p = df.cumsum().plot() # Plot cumulative sum of all series
```




```
[66]: p = df[:10].plot(kind='bar') # First 10 arrays as bar plots
```

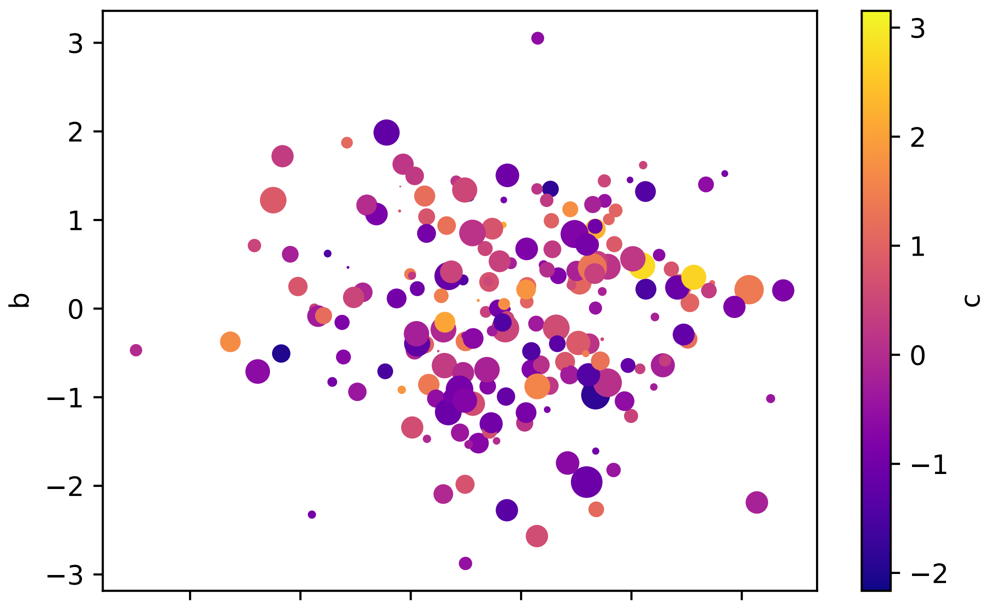


```
[67]: p = df.boxplot() # Boxplot for each of the 4 series
```



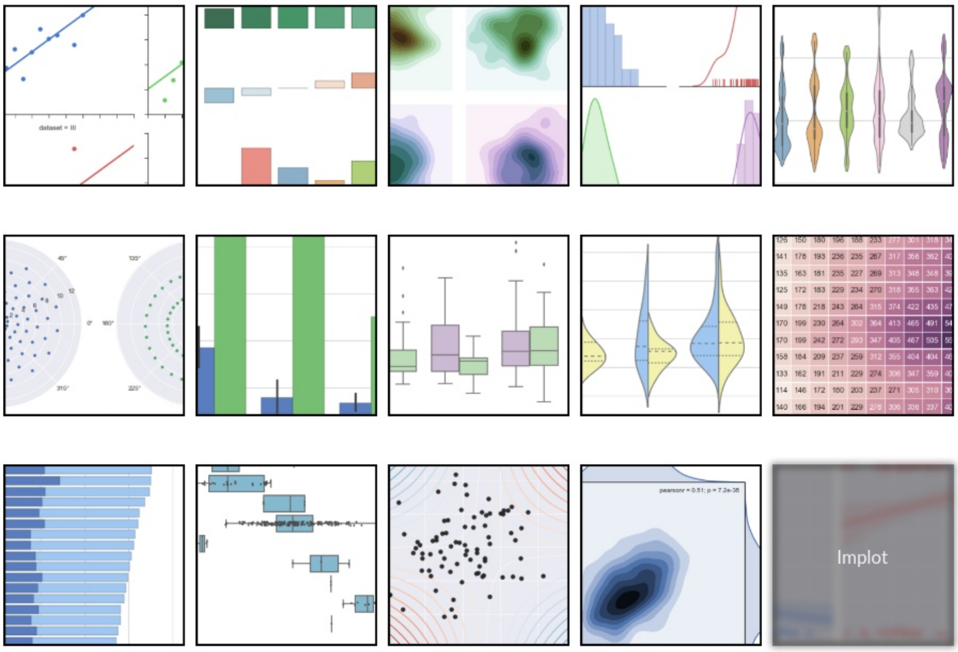

```
[68]: # Scatter plot using the 4 series for x, y, color, scale
      df[:300].plot(kind='scatter', x='a', y='b', c='c',
                    s=df['d']*50, linewidth='0', cmap='plasma')

<matplotlib.axes._subplots.AxesSubplot at 0x111f72400>
```



### Advanced plotting libraries

Several libraries, such as Seaborn offer more advanced plots and easier interfaces.




### Further reading links

- Matplotlib examples
- Plotting with pandas
- Seaborn examples