# Using NLTK with Other Python Libraries

In this chapter, we will explore some of the backbone libraries of Python for machine learning and natural language processing. Until now, we have used NLTK, Scikit, and genism, which had very abstract functions, and were very specific to the task in hand. Most of statistical NLP is heavily based on the vector space model, which in turn depends on basic linear algebra covered by NumPy. Also many NLP tasks, such as POS or NER tagging, are really classifiers in disguise. Some of the libraries we will discuss are heavily used in all these tasks.

The idea behind this chapter is to give you a quick overview of some the most fundamental Python libraries. This will help us understand more than just the data structure, design, and math behind some of the coolest libraries, such as NLTK and Scikit, which we have discussed in the previous chapters.

We will look at the following four libraries. I have tried to keep it short, but I highly encourage you to read in more detail about these libraries if you want Python to be a one-stop solution to most of your data science needs.

* NumPy (Numeric Python)
* SciPy (Scientific Python)
* Pandas (Data manipulation)
* Matplotlib (Visualization)

## NumPy

NumPy is a Python library for dealing with numerical operations, and it's really fast. NumPy provides some of the highly optimized data structures, such as ndarrays. NumPy has many functions specially designed and optimized to perform some of the most common numeric operations. This is one of the reasons NLTK, scikitlearn, pandas, and other libraries use NumPy as a base to implement some of the algorithms. This section will give you a brief summary with running examples of NumPy. This will not just help us understand the fundamental data structures beneath NLTK and other libraries, but also give us the ability to customize some of these functionalities to our needs.

Let's start with discussion on ndarrays, how they can be used as matrices, and how easy and efficient it is to deal with matrices in NumPy.

### ndarray

An ndarray is an array object that represents a multidimensional, homogeneous array of fixed-size items.

We will start with building an ndarray using an ordinary Python list:

**>>>x=[1,2,5,7,3,11,14,25]**

**>>>import numpy as np**

**>>>np\_arr=np.array(x)**

**>>>np\_arr**

As you can see, this is a linear 1D array. The real power of Numpy comes with 2D arrays. Let's move to 2D arrays. We will create one using a Python list of lists.

**>>>arr=[[1,2],[13,4],[33,78]]**

**>>>np\_2darr= np.array(arr) >>>type(np\_2darr) numpy.ndarray**

#### Indexing

The ndarray is indexed more like Python containers. NumPy provides a slicing method to get different views of the ndarray.

**>>>np\_2darr.tolist()**

**[[1, 2], [13, 4], [33, 78]] >>>np\_2darr[:] array([[1, 2], [13, 4], [33, 78]]) >>>np\_2darr[:2] array([[1, 2], [13, 4]]) >>>np\_2darr[:1] array([[1, 2]]) >>>np\_2darr[2] array([33, 78]) >>> np\_2darr[2][0]**

**>>>33 >>> np\_2darr[:-1] array([[1, 2], [13, 4]])**

### Basic operations

NumPy also has some other operations that can be used in various numeric processing. In this example, we want to get an array with values ranging from 0 to 10 with a step size of 0.1. This is typically required for any optimization routine. Some of the most common libraries, such as Scikit and NLTK, actually use these NumPy functions.

**>>>>import numpy as np >>>>np.arange(0.0, 1.0, 0.1) array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]**

We can do something like this, and generate a array with all ones and all zeros:

**>>>np.ones([2, 4]) array([[1., 1., 1., 1.], [1., 1., 1., 1.]]) >>>np.zeros([3,4])**

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

Wow!

If you have done higher school math, you know that we need all these matrixes to perform many algebraic operations. And guess what, most of the Python machine learning libraries also do that!

**>>>np.linspace(0, 2, 10) array([ 0., 0.22222222, 0.44444444, 0.66666667, 0.88888889, 1.11111111, 1.33333333, 1.55555556, 1.77777778, 2, ])**

The linespace function returns number samples which are evenly spaced, calculated over the interval from the start and end values. In the given example we were trying to get 10 sample in the range of 0 to 2.

Similarly, we can do this at the log scale. The function here is:

**>>>np.logspace(0,1) array([ 1., 1.04811313, 1.09854114, 1.1513954, 7.90604321, 8.28642773, 8.68511374, 9.10298178, 9.54095476, 10., ])**

You can still execute Python's help function to get more details about the parameter and the return values.

**>>>help(np.logspace) Help on function logspace in module NumPy.core.function\_base:**

**logspace(start, stop, num=50, endpoint=True, base=10.0) Return numbers spaced evenly on a log scale.**

**In linear space, the sequence starts at ``base \*\* start``**

**(`base` to the power of `start`) and ends with ``base \*\* stop``**

**(see `endpoint` below).**

**Parameters --------- start : float**

So we have to provide the start and end and the number of samples we want on the scale; in this case, we also have to provide a base.

### Extracting data from an array

We can do all sorts of manipulation and filtering on the ndarrays. Let's start with a new Ndarray, A:

**>>>A = array([[0, 0, 0], [0, 1, 2], [0, 2, 4], [0, 3, 6]])**

**>>>B = np.array([n for n in range n for n in range(4)]) >>>B array([0, 1, 2, 3])**

We can do this kind of conditional operation, and it's very elegant. We can observe this in the following example:

**>>>less\_than\_3 = B<3 # we are filtering the items that are less than 3. >>>less\_than\_3 array([ True, True, True, False], dtype=bool) >>>B[less\_than\_3] array([0, 1, 2])**

We can also assign a value to all these values, as follows:

**>>>B[less\_than\_3] = 0 >>>: B array([0, 0, 0, 3])**

There is a way to get the diagonal of the given matrix. Let's get the diagonal for our matrix A:

**>>>np.diag(A) array([0, 1, 4])**

### Complex matrix operations

One of the common matrix operations is element-wise multiplication, where we will multiply one element of a matrix by an element of another matrix. The shape of the resultant matrix will be same as the input matrix, for example:

**>>>A = np.array([[1,2],[3,4]]) >>>A \* A**

**array([[ 1, 4], [ 9, 16]])**

However, we can't perform the following operation, which will throw an error when executed:

**>>>A \* B**

**----------------------------------------------------------**

**-----------------**

**ValueError Traceback (most recent call last)**

**<ipython-input-53-e2f71f566704> in <module>()**

**----> 1 A\*B**

**ValueError**: Operands could not be broadcast together with shapes (2,2) (4,).

Simply, the numbers of columns of the first operand have to match the number of rows in the second operand for matrix multiplication to work.

Let's do the dot product, which is the backbone of many optimization and algebraic operations. I still feel doing this in a traditional environment was not very efficient.

Let's see how easy it is in NumPy, and how super-efficient it is in terms of memory.

**>>>np.dot(A, A) array([[ 7, 10], [15, 22]])**

We can do operations like add, subtract, and transpose, as shown in the following example:

**>>>A - A array([[0, 0], [0, 0]]) >>>A + A array([[2, 4], [6, 8]]) >>>np.transpose(A) array([[1, 3], [2, 4]]) >>>>A**

**array([[1, 2], [2, 3]])**

The same transpose operations can be performed using an alternative operation, such as this:

**>>>A.T**

**array([[1, 3], [2, 4]])**

We can also cast these ndarrays into a matrix and perform matrix operations, as shown in the following example:

**>>>M = np.matrix(A) >>>M matrix([[1, 2], [3, 4]]) >>> np.conjugate(M) matrix([[1, 2], [3, 4]]) >>> np.invert(M) matrix([[-2, -3], [-4, -5]])**

We can perform all sorts of complex matrix operations with NumPy, and they are pretty simple to use too! Please have a look at documentation for more information on NumPy.

Let's switch back to some of the common mathematics operations, such as min, max, mean, and standard deviation, for the given array elements. We have generated the normal distributed random numbers. Let's see how these things can be applied there:

**>>>N = np.random.randn(1,10) >>>N array([[ 0.59238571, -0.22224549, 0.6753678, 0.48092087,**

**-0.37402105, -0.54067842, 0.11445297, -0.02483442,**

**-0.83847935, 0.03480181, ]])**

**>>>N.mean()**

**-0.010232957191371551**

**>>>N.std()**

**0.47295594072935421**

This was an example demonstrating how NumPy can be used to perform simple mathematic and algebraic operations of finding out the mean and standard deviation of a set of numbers.

#### Reshaping and stacking

In case of some of the numeric, algebraic operations we do need to change the shape of resultant matrix based on the input matrices. NumPy has some of the easiest ways of reshaping and stacking the matrix in whichever way you want.

**>>>A**

**array([[1, 2], [3, 4]])**

If we want a flat matrix, we just need to reshape it using NumPy's reshape() function:

**>>>>(r, c) = A.shape # r is rows and c is columns**

**>>>>r,c**

**(2L, 2L) >>>>A.reshape((1, r \* c)) array([[1, 2, 3, 4]])**

This kind of reshaping is required in many algebraic operations. To flatten the ndarray, we can use the flatten() function:

**>>>A.flatten() array([1, 2, 3, 4])**

There is a function to repeat the same elements of the given array. We need to just specify the number of times we want the element to repeat. To repeat the ndarray, we can use the repeat() function:

**>>>np.repeat(A, 2) array([1, 1, 2, 2, 3, 3, 4, 4]) >>>>A**

**array([[1, 2],[3, 4]])**

In the preceding example, each element is repeated twice in sequence. A similar function known as tile() is used for for repeating the matrix, and is shown here:

**>>>np.tile(A, 4)**

**array([[1, 2, 1, 2, 1, 2, 1, 2], [3, 4, 3, 4, 3, 4, 3, 4]])**

There are also ways to add a row or a column to the matrix. If we want to add a row, we use the concatenate() function shown here:

**>>>B = np.array([[5, 6]]) >>>np.concatenate((A, B), axis=0) array([[1, 2], [3, 4], [5, 6]])**

This can also be achieved using the Vstack() function shown here:

**>>>np.vstack((A, B)) array([[1, 2], [3, 4], [5, 6]])**

Also, if you want to add a column, you can use the concatenate() function in the following manner:

**>>>np.concatenate((A, B.T), axis=1) array([[1, 2, 5], [3, 4, 6]])**

Alternatively, the hstack() function can be used to add columns. This is used very similarly to the vstack() function in the example shown above.

#### Random numbers

Random number generation is also used across many tasks involving NLP and machine learning tasks. Let's see how easy it is to get a random sample:

**>>>from numpy import random**

**>>>#uniform random number from [0,1] >>>random.rand(2, 5) array([[ 0.82787406, 0.21619509, 0.24551583, 0.91357419, 0.39644969], [ 0.91684427, 0.34859763, 0.87096617, 0.31916835, 0.09999382]])**

There is one more function called random.randn(), which generates normally distributed random numbers in the given range. So, in the following example, we want random numbers between 2 and 5.

**>>>>random.randn(2, 5) array([[-0.59998393, -0.98022613, -0.52050449, 0.73075943, -0.62518516], [ 1.00288355, -0.89613323, 0.59240039, -0.89803825, 0.11106479]])** This is achieved by using the function random.randn(2,5).

## SciPy

Scientific Python or SciPy is a framework built on top of NumPy and ndarray and was essentially developed for advanced scientific operations such as optimization, integration, algebraic operations, and Fourier transforms.

The concept was to efficiently use ndarrays to provide some of these common scientific algorithms in a memory-efficient manner. Because of NumPy and SciPy, we are in a state where we can focus on writing libraries such as scikit-learn and NLTK, which focus on domain-specific problems, while NumPy / SciPy do the heavy lifting for us. We will give you a brief overview of the data structures and common operations provided in SciPy. We get the details of some of the black-box libraries, such as scikit-learn and understand what goes on behind the scenes.

**>>>import scipy as sp**

This is how you import SciPy. I am using sp as an alias but you can import everything.

Let's start with something we are more familiar with. Let's see how integration can be achieved here, using the quad() function.

**>>>from scipy.integrate import quad, dblquad, tplquad >>>def f(x):**

**>>> return x**

**>>>x\_lower == 0 # the lower limit of x**

**>>>x\_upper == 1 # the upper limit of x**

**>>>val, abserr = = quad(f, x\_lower, x\_upper)**

**>>>print val,abserr**

**>>> 0.5 , 5.55111512313e-15**

If we integrate the x, it will be x2/2, which is 0.5. There are other scientific functions, such as these:

* Interpolation (scipy.interpolate)
* Fourier transforms (scipy.fftpack)
* Signal processing (scipy.signal)

But we will focus on only linear algebra and optimization because these are more relevant in the context of machine learning and NLP.

### Linear algebra

The linear algebra module contains a lot of matrix-related functions. Probably the best contribution of SciPy is sparse matrix (CSR matrix), which is used heavily in other packages for manipulation of matrices.

SciPy provides one of the best ways of storing sparse matrices and doing data manipulation on them. It also provides some of the common operations, such as linear equation solving. It has a great way of solving eigenvalues and eigenvectors, matrix functions (for example, matrix exponentiation), and more complex operations such as decompositions (SVD). Some of these are the behind-the-scenes optimization in our ML routines. For example, SVD is the simplest form of LDA (topic modeling) that we used in *Chapter 6*, *Text Classification*.

The following is an example showing how the linear algebra module can be used:

**>>>A = = sp.rand(2, 2)**

**>>>B = = sp.rand(2, 2)**

**>>>import Scipy**

**>>>X = = solve(A, B)**

**>>>from Scipy import linalg as LA**

**>>>X = = LA.solve(A, B)**

**>>>LA.dot(A, B)**

Detailed documentation is available at [IUUQ EPDTTDJQZPSHEPDTDJQZSFGFSFODFMJOBMHIUNM](http://docs.scipy.org/doc/scipy/reference/linalg.html).

### eigenvalues and eigenvectors

In some of the NLP and machine learning applications, we represent the documents as term document matrices. Eigenvalues and eigenvectors are typically calculated for many different mathematical formulations. Say *A* is our matrix, and there exists a vector *v* such that *Av=λv*.

In this case, *λ* will be our eigenvalue and *v* will be our eigenvector. One of the most commonly used operation, the **singular value decomposition** (**SVD**)will require some calculus functionality. It's quite simple to achieve this in SciPy.

**>>>evals = LA.eigvals(A) >>>evals**

**array([-0.32153198+0.j, 1.40510412+0.j])** And eigen vectors are as follows:

**>>>evals, evect = LA.eig(A)**

We can perform other matrix operations, such as inverse, transpose, and determinant:

**>>>LA.inv(A) array([[-1.24454719, 1.97474827], [ 1.84807676, -1.15387236]]) >>>LA.det(A)**

**-0.4517859060209965**

### The sparse matrix

In a real-world scenario, when we use a typical matrix, most of the elements of this matrix are zeroes. It is highly inefficient to go over all these non-zero elements for any matrix operation. As a solution to this kind of problem, a sparse matrix format has been introduced, with the simple idea of storing only non-zero items.

A matrix in which most of the elements are non-zeroes is called a dense matrix, and the matrix in which most of the elements are zeroes is called a sparse matrix.

A matrix is typically a 2D array with an index of row and column will provide the value of the element. Now there are different ways in which we can store sparse matrices:

* **DOK** (**Dictionary of keys**): Here, we store the dictionary with keys in the format (*row*, *col*) and the values are stored as dictionary values.
* **LOL** (**list of list**): Here, we provide one list per row, with only an index of the non-zero elements.
* **COL** (**Coordinate list**): Here, a list (*row*, *col*, *value*) is stored as a list.
* **CRS/CSR** (**Compressed row Storage**): A CSR matrix reads values first by column; a row index is stored for each value, and column pointers are stored (*val*, *row\_ind*, *col\_ptr*). Here, *val* is an array of the non-zero values of the matrix, *row\_ind* represents the row indices corresponding to the values, and *col\_ptr* is the list of *val* indexes where each column starts. The name is based on the fact that column index information is compressed relative to the COO format. This format is efficient for arithmetic operations, column slicing, and matrix-vector products.

See [IUUQEPDTTDJQZPSHEPDTDJQZ SFGFSFODFHFOFSBUFETDJQZTQBSTFDTS@NBUSJYIUNM](http://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.sparse.csr_matrix.html) for more information.

* **CSC** (**sparse column**): This is similar to CSR, except that the values are read first by column; a row index is stored for each value, and column pointers are stored. In otherwords, CSC is (*val*, *row\_ind*, *col\_ptr*).

Have a look at the documentation at:

[IUUQEPDTTDJQZPSHEPDTDJQZSFGFSFODF](http://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.sparse.csc_matrix.html)

[HFOFSBUFETDJQZTQBSTFDTD@NBUSJYIUNM](http://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.sparse.csc_matrix.html)

Let's have some hands-on experience with CSR matrix manipulation. We have a sparse matrix A:

**>>>from scipy import sparse as s**

**>>>A = array([[1,0,0],[0,2,0],[0,0,3]]) >>>A array([[1, 0, 0], [0, 2, 0], [0, 0, 3]]) >>>from scipy import sparse as sp**

**>>>C = = sp.csr\_matrix(A);**

**>>>C <3x3 sparse matrix of type '<type 'NumPy.int32'>' with 3 stored elements in Compressed Sparse Row format>**

If you read very carefully, the CSR matrix stored just three elements. Let's see what it stored:

**>>>C.toarray() array([[1, 0, 0], [0, 2, 0], [0, 0, 3]]) >>>C \* C.todense()**

**matrix([[1, 0, 0], [0, 4, 0], [0, 0, 9]])**

This is exactly what we are looking for. Without going over all the zeroes, we still got the same results with the CSR matrix.

**>>>dot(C, C).todense()**

### Optimization

I hope you understand that every time we have built a classifier or a tagger in the background, all these are some sort of optimization routine. Let's have some basic understanding about the function provided in SciPy. We will start with getting a minima of the given polynomial. Let's jump to one of the example snippets of the optimization routine provided by SciPy.

**>>>def f(x):**

**>>> returnx return x\*\*2-4**

**>>>optimize.fmin\_bfgs(f,0) Optimization terminated successfully.**

**Current function value: -4.000000**

**Iterations: 0**

**Function evaluations: 3 Gradient evaluations: 1 array([0])**

Here, the first argument is the function you want the minima of, and the second is the initial guess for the minima. In this example, we already knew that zero will be the minima. To get more details, use the function help(), as shown here:

**>>>help(optimize.fmin\_bfgs) Help on function fmin\_bfgs in module Scipy.optimize.optimize:**

**fmin\_bfgs(f, x0, fprime=None, args=(), gtol=1e-05, norm=inf, epsilon=1.4901161193847656e-08, maxiter=None, full\_output=0, disp=1, retall=0, callback=None) Minimize a function using the BFGS algorithm.**

**Parameters --------- f : callable f(x,\*args) Objective function to be minimized. x0 : ndarray Initial guess. >>>from scipy import optimize optimize.fsolve(f, 0.2) array([ 0.46943096])**

**>>>def f1 def f1(x,y):**

**>>> return x \*\* 2+ y \*\* 2 - 4 >>>optimize.fsolve(f1, 0, 0) array([ 0.])**

To summarize, we now have enough knowledge about SciPy's most basic data structures, and some of the most common optimization techniques. The intention was to motivate you to not just run black-box machine learning or natural language processing, but to go beyond that and get the mathematical context about the ML algorithms you are using and also have a look at the source code and try to understand it.

Implementing this will not just help your understanding about the algorithm, but also allow you to optimize/customize the implementation to your need.

## pandas

Let's talk about pandas, which is one of the most exciting Python libraries, especially for people who love **R** and want to play around with the data in a more vectorized manner. We will devote this part of the chapter only to pandas; we will discuss some basic data manipulation and handling in pandas frames.

### Reading data

Let's start with one of the most important tasks in any data analysis to parse the data from a CSV/other file.

I am using [IUUQTBSDIJWFJDTVDJFEVNMNBDIJOF MFBSOJOHEBUBCBTFTBEVMUBEVMUEBUB](https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data)

[https://archive.ics.uci.edu/ml/machine-learningdatabases/iris/iris.names](https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.names)

Feel free to use any other CSV file.

To begin, please download the data to your local storage from the preceding links, and load it into a pandas data-frame, as shown here:

**>>>import pandas as pd**

**>>># Please provide the absolute path of the input file**

**>>>data = pd.read\_csv("PATH\\iris.data.txt",header=0")**

**>>>data.head()**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **4.9** | **3.0** | **1.4** | **0.2** | **Iris-setosa** |
| **0** | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| **1** | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |
| **2** | 5.0 | 3.6 | 1.4 | 0.2 | Iris-setosa |

This will read a CSV file and store it in a DataFrame. Now, there are many options you have while reading a CSV file. One of the problems is that we read the first line of the data in this DataFrame as a header; to use the actual header, we need to set the option header to None, and pass a list of names as column names. If we already have the header in perfect form in the CSV, we don't need to worry about the header as pandas, by default, assumes the first line to be the header. The header 0 in the preceding code is actually the row number that will be treated as the header.

So let's use the same data, and add the header into the frame:

**>>>data = pd.read\_csv("PATH\\iris.data.txt", names=["sepal length",**

**"sepal width", "petal length", "petal width", "Cat"], header=None)**

**>>>data.head()**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **sepal length** | **sepal width** | **petal length** | **petal width** | **Cat** |
| **0** | 4.9 | 3.0 | 1.4 | 0.2 | Iris-setosa |
| **1** | 4.7 | 3.2 | 1.3 | 0.2 | Iris-setosa |
| **2** | 4.6 | 3.1 | 1.5 | 0.2 | Iris-setosa |

This has created temporary column names for the frame so that, in case you have headers in the file as a first row, you can drop the header option, and pandas will detect the first row of the file as the header. The other common options are Sep/ Delimiter, where you want to specify the delimiter used to separate the columns. There are at least 20 different options available, which can be used to optimize the way we read and cleanse our data, for example removing Na's, removing blank lines, and indexing based on the specific column. Please have a look at the different type of files:

* read\_csv: reading a CSV file.
* read\_excel: reading a XLS file.
* read\_hdf: reading a HDFS file.
* read\_sql: reading a SQL file.
* read\_json: reading a JSON file.

These can be the substitutes for all the different parsing methods we discussed in *Chapter 2*, *Text Wrangling and Cleansing*. The same numbers of options are available to write files too.

Now let's see the power of pandas frames. If you are an R programmer, you would love to see the summary and header option we have in R.

**>>>data.describe()**

The describe() function will give you a brief summary of each column and the unique values.

**>>>sepal\_len\_cnt=data['sepal length'].value\_counts()**

**>>>sepal\_len\_cnt**

**5.0 10**

**6.3 9**

**6.7 8**

**5.7 8**

**5.1 8 dtype: int64 >>>data['Iris-setosa'].value\_counts()**

**Iris-versicolor 50**

**Iris-virginica 50 Iris-setosa 48 dtype: int64**

Again for R lovers, we are now dealing with vectors, so that we can look for each value of the column by using something like this:

**>>>data['Iris-setosa'] == 'Iris-setosa'**

1. **True**
2. **True**
3. **False**
4. **False**

**Name: Iris-setosa, Length: 149, dtype: bool**

Now we can filter the DataFrame in place. Here the setosa will have only entries related to Iris-setosa.

**>>>sntsosa=data[data['Cat'] == 'Iris-setosa']**

**>>>sntsosa[:5]**

This is our typical SQL Group By function. We have all kinds of aggregate functions as well.

You can browse through the following link to look at Dow Jones data:

[https://archive.ics.uci.edu/ml/machine-learningdatabases/00312/](https://archive.ics.uci.edu/ml/machine-learning-databases/00312/)

### Series data

Pandas also have a neat way of indexing by date, and then using the frame for all sorts of time series kind of analysis. The best part is that once we have indexed the data by date some of the most painful operations on the dates will be a command away from us. Let's take a look at series data, such as stock price data for a few stocks, and how the values of the opening and closing stock change weekly.

**>>>import pandas as pd**

**>>>stockdata = pd.read\_csv("dow\_jones\_index.data",parse\_dates=['date'], index\_col=['date'], nrows=100) >>>>stockdata.head()**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **date** | **quarter** | **stock** | **open** | **high** | **low** | **close** | **volume** | **percent\_ change\_ price** |
| 01/07/2011 | 1 | AA | $15.82 | $16.72 | $15.78 | $16.42 | 239655616 | 3.79267 |
| 01/14/2011 | 1 | AA | $16.71 | $16.71 | $15.64 | $15.97 | 242963398 | -4.42849 |
| 01/21/2011 | 1 | AA | $16.19 | $16.38 | $15.60 | $15.79 | 138428495 | -2.47066 |

**>>>max(stockdata['volume'])**

**1453438639**

**>>>max(stockdata['percent\_change\_price'])**

**7.6217399999999991**

**>>>stockdata.index**

**<class 'pandas.tseries.index.DatetimeIndex'>**

**[2011-01-07, ..., 2011-01-28]**

**Length: 100, Freq: None, Timezone: None >>>stockdata.index.day**

**array([ 7, 14, 21, 28, 4, 11, 18, 25, 4, 11, 18, 25, 7, 14, 21, 28, 4,11, 18, 25, 4, 11, 18, 25, 7, 14, 21, 28, 4])**

The preceding command gives the day of the week for each date.

**>>>stockdata.index.month**

The preceding command lists different values by month.

**>>>stockdata.index.year**

The preceding command lists different values by year.

You can aggregate the data using a resample with whatever aggregation you want. It could be sum, mean, median, min, or max.

**>>>import numpy as np**

**>>>stockdata.resample('M', how=np.sum)**

### Column transformation

Say we want to filter out columns or to add a column. We can achieve this by just by providing a list of columns as an argument to axis 1. We can drop the columns from a data frame like this:

**>>>stockdata.drop(["percent\_change\_volume\_over\_last\_wk"],axis=1)** Let's filter out some of the unwanted columns, and work with a limited set of columns. We can create a new DataFrame like this:

**>>>stockdata\_new = pd.DataFrame(stockdata, columns=["stock","open","high"**

**,"low","close","volume"])**

**>>>stockdata\_new.head()**

We can also run R-like operations on the columns. Say I want to rename the columns. I can do something like this:

**>>>stockdata["previous\_weeks\_volume"] = 0**

This will change all the values in the column to 0. We can do it conditionally and create derived variables in place.

### Noisy data

A typical day in the life of a data scientist starts with data cleaning. Removing noise, cleaning unwanted files, making sure that date formats are correct, ignoring noisy records, and dealing with missing values. Typically, the biggest chunk of time is spent on data cleansing rather than on any other activity.

In a real-world scenario, the data is messy in most cases, and we have to deal with missing values, null values, Na's, and other formatting issues. So one of the major features of any data library is to deal with all these problems and address them in an efficient way. pandas provide some amazing features to deal with some of these problems.

**>>>stockdata.head()**

**>>>stockdata.dropna().head(2)**

Using the preceding command we get rid of all the Na's from our data.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **date** | **quarter** | **Stock** | **open** | **high** | **low** | **close** | **volume** | **percent\_ change\_ price** |
| 01/14/2011 | 1 | AA | $16.71 | $16.71 | $15.64 | $15.97 | 242963398 | -4.42849 |
| 01/21/2011 | 1 | AA | $16.19 | $16.38 | $15.60 | $15.79 | 138428495 | -2.47066 |
| 01/28/2011 | 1 | AA | $15.87 | $16.63 | $15.82 | $16.13 | 151379173 | 1.63831 |

You also noticed that we have a $ symbol in front of the value, which makes the numeric operation hard. Let's get rid of that, as it will give us noisy results otherwise (for example. $43.86 is not among the top values here).

**>>>import numpy >>>stockdata\_new.open.describe() count 100 unique 99 top $43.86 freq 2 Name: open, dtype: object**

We can perform some operations on two columns, and derive a new variable out of this:

**>>>stockdata\_new.open = stockdata\_new.open.str.replace('$', '').convert\_ objects(convert\_numeric=True)**

**>>>stockdata\_new.close = stockdata\_new.close.str.replace('$', ''). convert\_objects(convert\_numeric=True)**

**>>>(stockdata\_new.close - stockdata\_new.open).convert\_objects(convert\_**

**numeric=True)**

**>>>stockdata\_new.open.describe() count 100.000000 mean 51.286800 std 32.154889 min 13.710000 25% 17.705000**

**50% 46.040000 75% 72.527500 max 106.900000 Name: open, dtype: float64**

We can also perform some arithmetic operations, and create new variables out of it.

**>>>stockdata\_new['newopen'] = stockdata\_new.open.apply(lambda x: 0.8 \* x)**

**>>>stockdata\_new.newopen.head(5)**

We can filter the data on the value of a column in this way too. For example, let's filter out a dataset for one of the companies among all those that we have the stock values for.

**>>>stockAA = stockdata\_new.query('stock=="AA"')**

**>>>stockAA.head()**

To summarize, we have seen some useful functions related to data reading, cleaning, manipulation, and aggregation in this section of pandas. In the next section, will try to use some of these data frames to generate visualization out of this data.

## matplotlib

matplotlib is a very popular visualization library written in Python. We will cover some of the most commonly used visualizations. Let's start by importing the library:

**>>>import matplotlib**

**>>>import matplotlib.pyplot as plt**

**>>>import numpy**

We will use the same running data set from the Dow Jones index for some of the visualizations now. We already have stock data for company "AA". Let's make one more frame for a new company CSCO, and plot some of these:

**>>>stockCSCO = stockdata\_new.query('stock=="CSCO"')**

**>>>stockCSCO.head()**

**>>>from matplotlib import figure**

**>>>plt.figure()**

**>>>plt.scatter(stockdata\_new.index.date,stockdata\_new.volume)**

**>>>plt.xlabel('day') # added the name of the x axis**

**>>>plt.ylabel('stock close value') # add label to y-axis**

**>>>plt.title('title') # add the title to your graph**

**>>>plt.savefig("matplot1.jpg") # savefig in local**

You can also save the figure as a JPEG/PNG file. This can be done using the savefig() function shown here:

**>>>plt.savefig("matplot1.jpg")**

### Subplot

Subplot is the best way to layout your plots. This works as a canvas, where we can add not just one plot but multiple plots. In this example, we have tried to put four plots with the parameters numrow, numcol which will define the canvas and the next argument in the plot number.

**>>>plt.subplot(2, 2, 1)**

**>>>plt.plot(stockAA.index.weekofyear, stockAA.open, 'r--')**

**>>>plt.subplot(2, 2, 2)**

**>>>plt.plot(stockCSCO.index.weekofyear, stockCSCO.open, 'g-\*')**

**>>>plt.subplot(2, 2, 3)**

**>>>plt.plot(stockAA.index.weekofyear, stockAA.open, 'g--')**

**>>>plt.subplot(2, 2, 4)**

**>>>plt.plot(stockCSCO.index.weekofyear, stockCSCO.open, 'r-\*')**

**>>>plt.subplot(2, 2, 3)**

**>>>plt.plot(x, y, 'g--')**

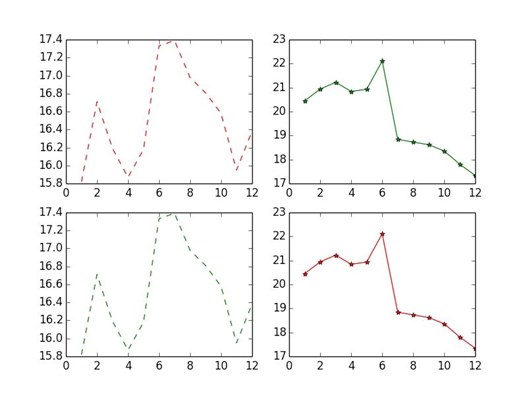
**>>>plt.subplot(2, 2, 4)**

**>>>plt.plot(x, y, 'r-\*')**

**>>>**

**fig.savefig("matplot2.png"**

**)**



We can do something more elegant for plotting many plots at one go!

**>>>fig, axes = plt.subplots(nrows=1, ncols=2) >>>for ax in axes:**

**>>> ax.plot(x, y, 'r')**

**>>> ax.set\_xlabel('x')**

**>>> ax.set\_ylabel('y')**

**>>> ax.set\_title('title');**

As you case see, there are ways to code a lot more like in typical Python to handle different aspects of the plots you want to achieve.

### Adding an axis

We can add an axis to the figure by using addaxis(). By adding an axis to the figure, we can define our own drawing area. addaxis() takes the following arguments:

**\*rect\* [\*left\*, \*bottom\*, \*width\*, \*height\*]**

**>>>fig = plt.figure()**

**>>>axes = fig.add\_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0 to 1) >>>axes.plot(x, y, 'r')**

Let' plot some of the most commonly used type of plots. The great thing is that most of the parameters, such as title and label, still work in the same way. Only the kind of plot will change.

If you want to add an x label, a y label, and a title with the axis; the commands are as follows:

**>>>fig = plt.figure()**

**>>>ax = fig.add\_axes([0.1, 0.1, 0.8, 0.8])**

**>>>ax.plot(stockAA.index.weekofyear,stockAA.open,label="AA")**

**>>>ax.plot(stockAA.index.weekofyear,stockCSCO.open,label="CSCO")**

**>>>ax.set\_xlabel('weekofyear')**

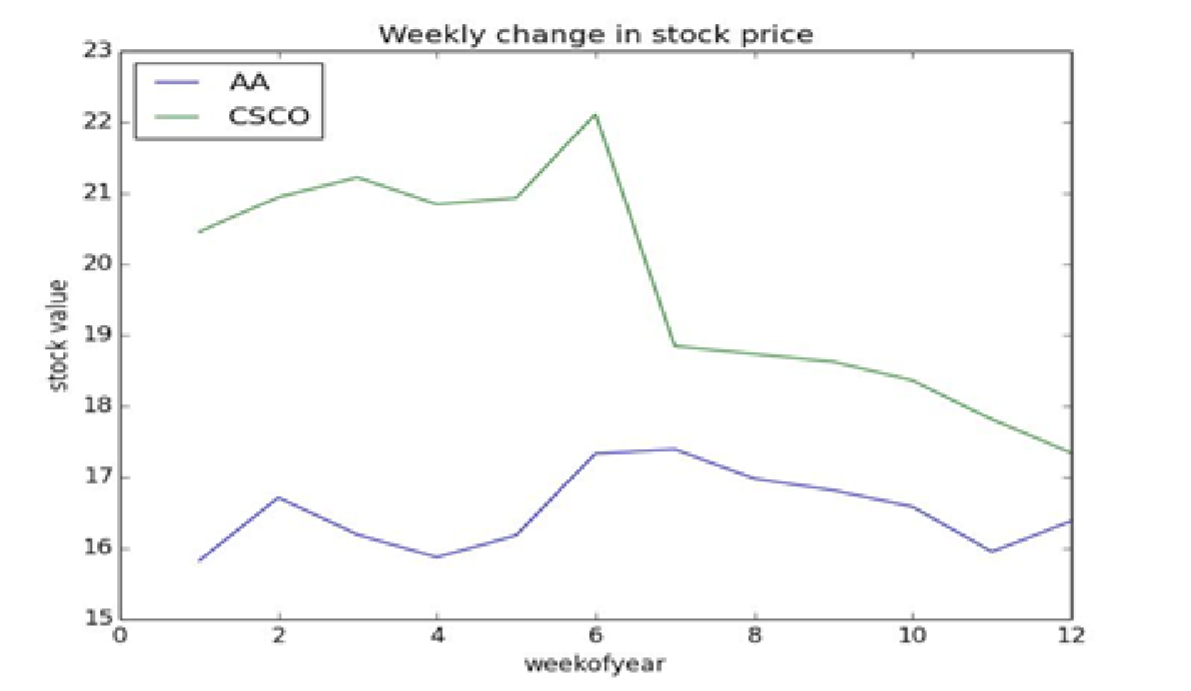
**>>>ax.set\_ylabel('stock value')**

**>>>ax.set\_title('Weekly change in stock price')**

**>>>ax.legend(loc=2); # upper left corner**

**>>>plt.savefig("matplot3.jpg")**

Try writing the preceding code and observe the output!



### A scatter plot

One of the simplest forms of plotting is to plot the y-axis point for different x-axis values. In the following example, we have tried to capture the variation of the stock price weekly in a scatter plot:

**>>>import matplotlib.pyplot as plt**

**>>>plt.scatter(stockAA.index.weekofyear,stockAA.open)**

**>>>plt.savefig("matplot4.jpg")**

**>>>plt.close()**

### A bar plot

Intuitively, the distribution of the y axis is shown against the x axis in the following bar chart. In the following example, we have used a bar plot to display data on a graph.

**>>>n = 12**

**>>>X = np.arange(n)**

**>>>Y1 = np.random.uniform(0.5, 1.0, n)**

**>>>Y2 = np.random.uniform(0.5, 1.0, n)**

**>>>plt.bar(X, +Y1, facecolor='#9999ff', edgecolor='white')**

**>>>plt.bar(X, -Y2, facecolor='#ff9999', edgecolor='white')**

### 3D plots

We can also build some spectacular 3D visualizations in matplotlib. The following example shows how one can create a 3D plot using matplotlib:

**>>>from mpl\_toolkits.mplot3d import Axes3D**

**>>>fig = plt.figure()**

**>>>ax = Axes3D(fig)**

**>>>X = np.arange(-4, 4, 0.25)**

**>>>Y = np.arange(-4, 4, 0.25)**

**>>>X, Y = np.meshgrid(X, Y)**

**>>>R = np.sqrt(X\*\*2+ + Y\*\*2)**

**>>>Z = np.sin(R)**

**>>>ax.plot\_surface(X, Y, Z, rstride=1, cstride=1, cmap='hot')**

## External references

I like to encourage readers to go over some of the following links for more details about the individual libraries, and for more resources:

* http://www.NumPy.org/
* <http://www.Scipy.org/>
* <http://pandas.pydata.org/>
* <http://matplotlib.org/>

## Summary

This chapter was a brief summary of some of the most fundamental libraries of

Python that do a lot of heavy lifting for us when we deal with text and other data. NumPy helps us in dealing with numeric operations and the kind of data structure required for some of these. SciPy has many scientific operations that are used in various Python libraries. We learned how to use these functions and data structures.

We have also touched upon pandas, which is a very efficient library for data manipulation, and has been getting a lot of mileage in recent times. Finally, we gave you a quick view of one of Python's most commonly used visualization libraries, matplotlib.

In the next chapter, we will focus on social media. We will see how to capture data from some of the common social networks and produce meaningful insights around social media.