Section B2 Data Science Libraries

NumPy

NumPy Arrays

import numpy as np

The core of NumPy is the **array** object class. Unlike a Python List, **all the elements of a NumPy array must be of the same type;** commonly, a numeric type like an integer (int) or decimal (float). In NumPy, vectors (one dimension) and matrices (two or more dimensions) are both called *arrays*.

N-dimensional arrays (ndarrays)

NumPy supports higher dimensional arrays denoted as *ndarrays*. All NumPy arrays have the property of shape. It denotes the dimensionality of the array. The shape property returns a tuple with the number of elements in the tuple corresponding to the number of dimensions of the array. For example, a 3D array of shape (2, 3, 3) has 2 rows, 3 columns with 3 elements in each column.

- Remember not to use the len() function with multi-dimensional arrays as this will only give you the number of rows, not the total number of elements!

```
import numpy as np

#The shape vs size of an array
a2d = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

print('The total number of elements in the array is:',a2d.size)

rows, columns = a2d.shape
print('The number of rows is:',rows,'the number of columns is:',columns)
```

Vectorisation

Vectorised arithmetic operators (ufunc)

NumPy's universal functions (ufuncs) provide vectorised implementations of arithmetic functions. Use these whenever you need to do operations over large data sets in arrays, instead of using a for loop.

Operator	unfunc	Description
`+`	`np.add`	Addition
`-`	`np.subtract`	Subtraction
`-`	`np.negative`	Unary negation (e.g -5)
`*`	`np.multiply`	Multiplication
`/`	`np.divide`	Division
`**`	`np.power`	Exponentiation
`//`	`np.floor_divide`	Integer division
`%`	`np.mod`	Modulo (division remainder)

Other ufunc Functions

NumPy has many other functions. Some of these are the NaN-safe version. This means that NumPy will ignore missing values when applying the functions.

The following table lists other functions in ufuncs.

Function	Nan-safe Version	Description
`np.all()`	Not available	Evaluate whether all elements are true
`np.any()`	Not available	Evaluate whether any elements are true
`np.argmax()`	`np.nanargmax()`	Find index of maximum value
`np.argmin()`	`np.nanargmin()`	Find index of minimum value
`np.max()`	`np.nanmax()`	Find maximum value
`np.mean()`	`np.nanmean()`	Compute mean of elements
`np.median()`	`np.nanmedian()`	Compute median of elements
`np.min()`	`np.nanmin()`	Find minimum value
`np.percentile()`	`np.nanpercentile()`	Compute rank-based statistics of elements
`np.prod()`	`np.nanprod()`	Compute product of elements
`np.std()`	`np.nanstd`	Compute standard deviation
`np.sort()`	Not available	return a sorted copy of an array
`np.sum()`	`np.nansum()`	Compute sum of elements
`np.transpose()`	Not available	Permute the dimensions of an array
`np.var()`	`np.nanvar()`	Compute variance

Creating and Manipulating Arrays

Creating a 1D array

There are several ways you can create arrays with NumPy:

- Using the arange() function.
- Providing a list.
- Using the zeros() function.

Creating a 1D array with a range

arange([start,] stop[, step,], dtype=None)

- **start** of the interval is a numeric value included in the interval and it is optional. The default start value is 0.
- **stop** is the end of the interval. It is a numeric value not included in the interval.
- **step** is the space between the values. it is a numeric and optional value. The default step size is 1.
- **dtype** indicates the type of the array. If dtype is not given, infer the data type from the other input arguments.

Creating a 1D array from lists or tuples

If you have your data in list form , you can convert it into an array using the function **array()**. The main arguments of the function are the *list* to be converted into the array and the *type*.

Creating a 1D array with zeros ()

The **zeros()** function will create an array filled with zeros for you. This is useful when some values may be missing in your data — you can represent the missing information with zeros.

zeros(shape, dtype , order)

- **shape** int or tuple of ints.
- **dtype** indicates the type of the array and it is optional or you can use

any of the types define in NumPy. For example np.int64 or np.float64.
- **order** defines how to store the multi-dimensional data row-major (programming language C-style) or column-major (programming language Fortran-style). The argument order is optional and the default is 'C'.

Creating 2D Arrays

The shape property can be used to create zero-filled arrays with the zeros() function by specifying the number of rows and columns in the array as a tuple.

```
# Creating a multi-dimensional array of zeros
a4 = np.zeros((4,3))
a4
```

You can also create 2D arrays from a list of lists.

```
# Create a 2D array from nested lists
a4 = np.array([[1,2,3], [4,5,6], [7,9,9]])
a4
```

Slicing Sections of an Array

You take a slice of an array with the slice notation, marked by the colon (:) character.
array1[start: stop: step]
start, stop and step have the same behaviour as in the function **arange()**. Remember if any of these parameters are unspecified, they default to the values start=0, stop=last index, step=1.

Having a negative step can be confusing, but the effect is that the values for start and stop are swapped. This can convenient when you want to reverse an array.

```
# Reversing all the elements
a1[::-1]
```

Out[18]:

```
array([19, 18, 17, 16, 15, 14, 13, 12, 11, 10,  9,  8,  7,  6,  5,  4,  3,
        2,  1,  0])
```

In [19]:

```
# Reversing a part of the array, starting with the element at index 10
a1[10::-2]
```

Out[19]:

```
array([10,  8,  6,  4,  2,  0])
```

```
# Create a multidimensional array
a2 = np.array([[3, 12, 5, 65],[13, 90, 2, 49], [35, 79, 1, 8]])
a2
```

In [22]:

```
# Slice all the rows and the first column
a2[:,:1]
```

Out[22]:

```
array([[ 3],
       [13],
       [35]])
```

In [23]:

```
# Slice all the rows and every other column
a2[:3, ::2]
```

Out[23]:

```
array([[ 3,  5],
       [13,  2],
       [35,  1]])
```

In [24]:

```
# Use slices to invert the rows and columns
# Slice the entire array, but step -1 for both rows and columns
print('original a2 \n', a2)
print('inverted a2 \n', a2[::-1, ::-1])
```

Out [24]:

```
original a2
 [[ 3 12  5 65]
 [13 90  2 49]
 [35 79  1  8]]
inverted a2
 [[ 8  1 79 35]
 [49  2 90 13]
 [65  5 12  3]]
```

Reshaping arrays

The **reshape()** function takes the number of rows and columns to use for the 2D array.

Concatenate arrays

We can combine multiple arrays joining them into one array using:

- **np.concatenate()** Join a sequence of arrays along an existing axis (rows or columns).
  Parameters:

- A sequence of arrays to concatenate. The arrays must have the same shape, except in the dimension corresponding to axis (rows by default). This means that there can be different numbers of rows, but all of the arrays must have the same number of columns.
    - axis : int, optional. The axis along which the arrays will be joined. Default is 0 (rows). 1 selects columns.
- **np.vstack()** Stack arrays in sequence vertically (row wise). Parameters:
    - Arrays to concatenate. The arrays must have the same shape along all but the first axis (rows). 1D arrays must have the same length.
- **np.hstack()** Stack arrays in sequence horizontally (column wise). Parameters
    - Arrays to concatenate. The arrays must have the same shape along all but the second axis (columns), except 1D arrays which can be any length.

Split arrays

You can also split an array into several arrays using:

- np.split(): Split an array into multiple sub-arrays. Parameters:
    - Array to be divided into sub-arrays
    - Indices or sections: int or one-dimensional array.
        - If indices_or_sections is an integer, N, the array will be divided into N equal arrays along axis.
        - If indices_or_sections is an array, each number in the array indicates where the next section ends.
        - If such a split is not possible, an error is raised.
- np.vsplit()
    - Array to be divided into sub-arrays
    - Indexes of sections
- np.hsplit()
    - Array to be divided into sub-arrays
    - Indexes of sections

Copy arrays

Arrays present the same problem as lists when you assign an array to another array. If you modify either of the arrays the other array is modified as well. This behaviour occurs as NumPy arrays are mutable objects. This can be avoided by using the copy() method or np.array(), which will give you a new array that has the same values.

```
# Copy an array in another array with copy()

# If you want an independent copy use copy() or np.array
a2 =np.array([[3, 12, 5, 65],[13, 90, 2, 49], [35, 79, 1, 8]])
print('a2 = \n',a2)

a2Slice= a2[:2,:2].copy()
print('\n a2Slice with copy() = \n',a2Slice)

# What happens if we modified a position of the array?
a2Slice[0, 0] = -10
print('\n a2Slice after modified= \n',a2Slice)
print('\n a2 is NOT modified =\n', a2)

a2Slice = np.array(a2[:2,:2])
print('\n a2Slice with np.array() = \n',a2Slice)

# What happens if we modified a position of the array?
a2Slice[0, 0] = -10
print('\n a2Slice after modified= \n',a2Slice)
print('\n a2 is NOT modified =\n', a2)
```

pandas

The core of Pandas are the following data structures:

- **Series** — an object used to represent a 1-dimensional array of indexed data.
- **Dataframe** — consists of an ordered collection of columns (similar to a spreadsheet), each column can contain a value of a different type like string, date, numeric, etc.

The main difference between a pandas series and NumPy array is the index. While the NumPy array only has an **implicitly** defined integer *index used to access the values*, the Pandas series can have an **explicitly** defined *index used to access the values*. With series, the indexes can be numeric (default) or any immutable type.

Constructing Series Objects

pd.Series(data, index= index)

- data can be an array-like, iterable, dict, or scalar (ie: a single value, like int, string, etc.) value.
- index is an optional parameter, by default it is an integer sequence starting from zero.

In [5]:

```
# Construct a series from specific indexes from a dictionary

pd.Series({2:'orange', 3:'apple', 1:'nectarine', 4:'strawberry'}, index=[2,4])
```

Out[5]:

```
2        orange
4    strawberry
dtype: object
```

Remember, if you are using the implicit (default) numeric index, the index starts at zero.
Also, you can access a slice of the data as you did with arrays in NumPy.
<series_name> [start, stop, step]
Series are mutable objects. Recall that if you assign mutable objects, they refer to the same object. Be sure to use the Series copy() method if you want a copy.

DataFrame
The pandas Series is a 1-dimensional labelled structure. DataFrame is a 2-dimensional labelled structure.
You can create a DataFrame from lists, dictionaries, arrays, Series or combinations of these. The most common way to create a new Data Frame is using a DataFrame() constructor and passing a dictionary as a paramete

In [18]:

```
# Create a DataFrame using 3 Series and a Dictionary

s1 = pd.Series(np.arange(1,6))
s2 = pd.Series(np.arange(6,11))
s3 = pd.Series(np.arange(11,16))
df2 = pd.DataFrame({'C1':s1, 'C2': s2, 'C3':s3})
df2

# Note how the Dictionary keys become the column labels
```

Out[18]:

	C1	C2	C3
0	1	6	11
1	2	7	12
2	3	8	13
3	4	9	14
4	5	10	15

In [19]:

```
# Create a DataFrame from a Dictionary with List values

data_dict = {'product' : ['ball','pen','pencil','paper','mug'],
             'color' : ['blue','green','yellow','red','white'],
             'price' : [1.2,1.0,0.6,0.9,1.7],
             'price_discount': [1.1,0.8, 0.5,0.8,1.5]}
df3 = pd.DataFrame(data_dict)
df3

#Again note how the keys become the column labels and the values become the data
```

If the labels for the index (row) are not explicitly specified in your DataFrame, pandas, by default, assigns a numeric sequence starting from 0. As you saw earlier, if you want to assign labels to the indexes (rows) you can include the index parameter when you create your DataFrame object to assign the labels you want.

In [20]:

```
# Assign labels to the index (rows)

df3 = pd.DataFrame(data_dict, index=['one','two','three','four','five'])
df3
```

Out[20]:

	product	color	price	price_discount
one	ball	blue	1.2	1.1
two	pen	green	1.0	0.8
three	pencil	yellow	0.6	0.5
four	paper	red	0.9	0.8
five	mug	white	1.7	1.5

What happens if your Series or arrays don't have the same size? No worries! Pandas will automatically fill these spaces with not a number (NaN) values.

DataFrame Operations

Missing Values

Pandas allows you to explicitly define Not a Number (**NaN**) values and add them to a Series or a DataFrame.

You can find out how many NaNs you have in your Series by checking if the values are null with the **isnull( )** and **notnull( )** functions.

In [37]:

```
# isnull() function returns true when is a null value or NaN
series1.isnull()

# mandarin is null because it has NaN for value
```

Out[37]:

```
apple        False
blueberry    False
banana       False
orange       False
mandarin      True
dtype: bool
```

Unique Values, Counts and NaNs

- To get the unique values in a Series, you use the **unique()** function.

- You may also want to know how many times a value is repeated or its occurrences. If so, you use **value_counts()** function.
- Finally, with the **isin()** function you can determine if values are contained in the Series.

```
# Getting the unique values

colors = pd.Series(['yellow', 'white', 'black', 'black', 'white', 'red', 'yellow', 'red', 'purple','black', 'black', 'red'])
print('colours =')
print(colors)

print('unique values in colours \n', colors.unique())
```

```
colours =
0     yellow
1      white
2      black
3      black
4      white
5        red
6     yellow
7        red
8     purple
9      black
10     black
11       red
dtype: object

unique values in colours
 ['yellow' 'white' 'black' 'red' 'purple']
```

In [31]:

```
# Counting the occurrences in a series

colors.value_counts()
```

Out[31]:

```
black     4
red       3
yellow    2
white     2
purple    1
dtype: int64
```

In [32]:

```
# Evaluating the membership of the values

colors.isin(['white','green'])

# Note this function returns a a series with True for the elements
# that match or False for those that don't match
```

Out[32]:

```
0     False
1      True
2     False
3     False
4      True
5     False
6     False
7     False
8     False
9     False
10    False
11    False
dtype: bool
```

In [33]:

```
# Filtering the data using the boolean values that the isin() function returns

colors[colors.isin(['white','red'])]
```

Out[33]:

```
1     white
4     white
5       red
7       red
11      red
dtype: object
```

- You can count how many data values are valid or missing by combining isnull() or notnull() with sum(). **<seriesName>.isnull().sum()**
- If you want to filter the data to only see valid data or only missing data, you can use isnull() or notnull() as conditions on the index. **<seriesName>[<seriesName>.notnull()]** (you'll get the notnull (i.e. valid) data).

In [39]:

```
# Counting notnull and null data

print('Number of null data values in series2 is',series2.isnull().sum())
print('Number of valid data values in series2 is',series2.notnull().sum())
```

```
Number of null data values in series2 is 3
Number of valid data values in series2 is 4
```

In [40]:

```
# Filtering NOT null data

print('The valid data in series2 are')
print(series2[series2.notnull()])
print('Missing data values in series2 are')
print(series2[series2.isnull()])
```

```
The valid data is series2 are
apple         1.50
blueberry     3.50
banana        2.99
orange        4.30
dtype: float64
Missing data values in series2 are
mandarin      NaN
rockmelon     NaN
strawberry    NaN
dtype: float64
```

Setting Column Labels

You can specify the labels of the columns when you construct the DataFrame object by adding the arguments: **columns = [<array_of_names>]**.

You can get the names of the columns in a DataFrame using the columns attribute.

You can also change the names of the columns by assigning values to the columns attribute.

Remember that the column names are a pandas index, so you can access the label for a specific column by using its position.

```
# Setting names to the columns

df1 = pd.DataFrame(np.array([[6.5, 90.3], [3.6, 3.2]]), columns = ['col1','col2'])
df1
```

Out[45]:

	col1	col2
0	6.5	90.3
1	3.6	3.2

In [46]:

```
# Getting the names of the columns using the DataFrame columns attribute

print('name of the columns ',df1.columns)
```

```
name of the columns  Index(['col1', 'col2'], dtype='object')
```

In [47]:

```
# Getting the names of a specific columns using the index

print('column 1 label:', df1.columns[0])
print('column 2 label:', df1.columns[1])
```

```
column 1 label: col1
column 2 label: col2
```

In [48]:

```
# Setting new names for the columns using the DataFrame columns attribute

df1.columns = ['c1','c2']
print('column 1 label:', df1.columns[0])
print('column 2 label:', df1.columns[1])
df1
```

```
column 1 label: c1
column 2 label: c2
```

Out[48]:

	c1	c2
0	6.5	90.3
1	3.6	3.2

Setting Row (Index) Labels
Index (row) labels work similarly to columns. Just remember that pandas DataFrames call the row, *index*.
You can assign index abels when you are constructing your DataFrame object by adding the arguments: **index = [< array_of_names >]**.
You can access the names of the indexes (rows) with the Data Frame index attribute.

Setting a Column as an Index

You may have found while working with the data that it would have been easier to have the city names as the indexes (row labels) rather than numbers.you can do this using the **set_index()** function, but the index values must all be unique

```
# Change the index labels
# set_index takes the column(s) that you want to use for the indexes

df_new_pop = df_pop.set_index('cities')
print(df_pop)
print(df_new_pop)
```

Selecting Data

you can select the data you want from your DataFrame by using attributes such as values (for all of the data) or by using indexes (labels) to select specific data.

Note how np.nan has been used to fill missing values.

```
# Create a DataFrame containing population data to use in selecting data examples

# Create DataFrame using Series of population density, state, unemployment rate and city names.
s_city = pd.Series(['Sydney','Melbourne','Brisbane','Perth','Adelaide','Gold Coast','Canberra','Newcastle','Wollongong','Logan City'])
s_density = pd.Series([4627345, 4246375, 2189878, 1896548,1225235, 591473, 367752, 308308, 292190, 282673])
s_state = pd.Series(['New South Wales', 'Victoria','Queensland', 'Western Australia','South Australia',
                     'Queensland','Australian Capital Territory','New South Wales','New South Wales', 'Queensland'])
s_unemployed_rate = pd.Series([4.3, 4.9, np.nan, np.nan, 7.3, 6.4, 3.5, 4.3, 4.3, 6.4])

df_pop = pd.DataFrame({'cities':s_city, 'density':s_density, 'state':s_state, 'unemployed_rate':s_unemployed_rate})
df_pop
```

Out[58]:

	cities	density	state	unemployed_rate
0	Sydney	4627345	New South Wales	4.3
1	Melbourne	4246375	Victoria	4.9
2	Brisbane	2189878	Queensland	NaN
3	Perth	1896548	Western Australia	NaN
4	Adelaide	1225235	South Australia	7.3
5	Gold Coast	591473	Queensland	6.4
6	Canberra	367752	Australian Capital Territory	3.5
7	Newcastle	308308	New South Wales	4.3
8	Wollongong	292190	New South Wales	4.3
9	Logan City	282673	Queensland	6.4

Now, if you want to select *all* the data as a 2-dimensional array, use the values attribute.

If you want to the values of one column, use:

<DataFrame_name>.<name_column> or

<DataFrame_name>['<name_column>']

```
# values property gives all data as an array

df_pop.values
```

```
# Selecting only cities

df_pop.cities
```

```python
# Selecting only unemployment rate

df_pop['unemployed_rate']
```

Slicing a DataFrame
<DataFrame_name>[start: stop: step]

- **start** is a numeric value included where the selection will start and it is optional. The default start value is 0.
- **stop** is the end of the section. It is a numeric value not included in the interval.
- **step** is the step size between the values. it is a numeric and optional value. The default step size is 1.

Slicing with Conditions

Another way you can slice is by using a condition inside of loc:
<DataFrame>.loc[<condition>, [<col1,..., coln>]]
You can also use more than one condition by joining conditions with: "**&**" is always used for "and" and "**|**" for "or".

In [82]:

```python
# Example of Slicing with a condition
# Getting the cities and population density columns for population densities bigger than 2 millions

df_pop.loc[df_pop.density > 2000000,['cities','density']]
```

Out[82]:

	cities	density
0	Sydney	4627345
1	Melbourne	4246375
2	Brisbane	2189878

In [83]:

```python
# Example of Slicing with combined conditions
# Get all of the information for rows where the unemployment rate is between 4.0 and 4.5
df_pop.loc[(df_pop.unemployed_rate > 4.0) & (df_pop.unemployed_rate > 4.5)]
```

Out[83]:

	cities	area	density	state	unemployed_rate	min_Temp	max_Temp
0	Sydney	12368.0	4627345	New South Wales	4.3	7.50	38.64
7	Newcastle	261.8	308308	New South Wales	4.3	4.68	41.36
8	Wollongong	714.0	292190	New South Wales	4.3	2.05	25.57

Visualisation Using Pandas

NOTE: The Pandas library does *not* draw the plots; instead, pandas tells Matplotlib how to draw the plots using Pandas data, selecting the Series for plots, labelling axis, creating legends, and determining appropriate visual elements, such as choosing colours and markers.

```
#libraries
import numpy as np #manipulation of arrays
import pandas as pd #manipulation of data as DataFrame or Series
import matplotlib.pyplot as plt # setting style and plot attributes such as the range for x and y axes

# this is a 'magic function' (Python's official word for it!)
# it makes your plots appear in your notebook
%matplotlib inline

#We'll use the white grid style for our plots
plt.style.use('seaborn-whitegrid')
```

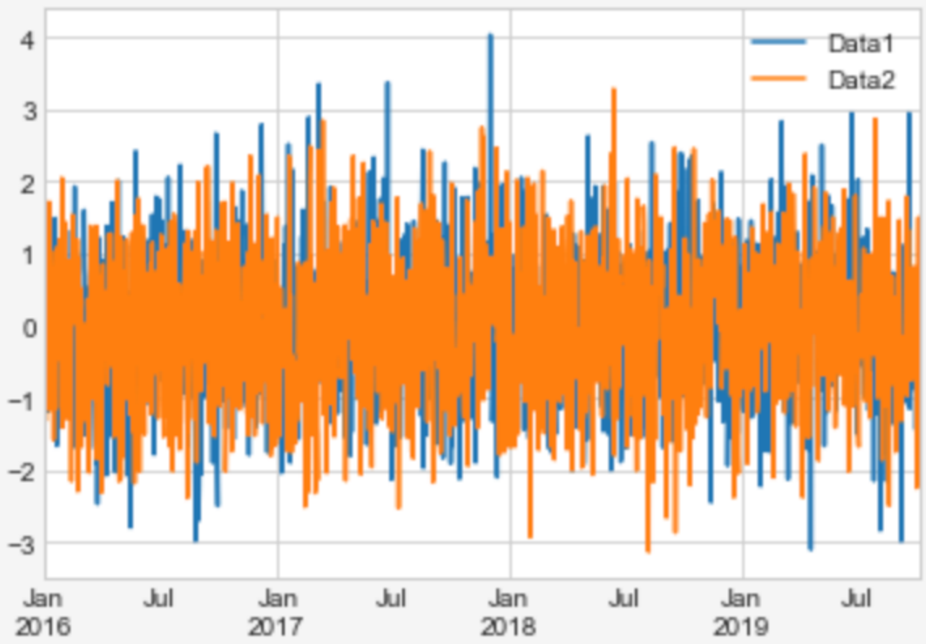
You can use the **plot()** function to plot the data in a Data Frame.

In [6]:

```
# plot it!
df.plot()
```

Out[6]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x106e6f908>
```



You can add a title, x-label, y-label and legend to your Pandas plots and change the colour and marker of the line, just like you did in Matplotlib. By setting the values for the attributes when you call plot:

- color
- marker
- linestyle
- title

You can set x and y labels and the legends using:

- set_xlabel
- set_ylabel
- legend

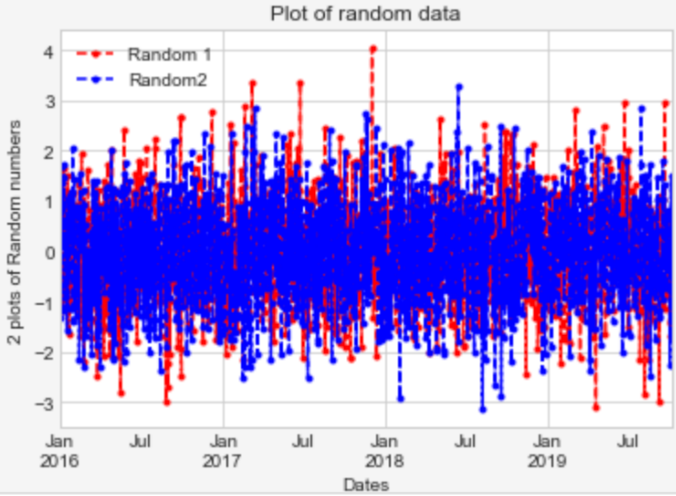
```
# The x and y labels are part of the axes of the plot
# the axes are returned by the plot() function
axes= df.plot(color=['r','b'], marker='.', linestyle='--', title='Plot of random data')

# set the label attributes of the axes
axes.set_xlabel('Dates')
axes.set_ylabel('2 plots of Random numbers')

# change the legend labels (default set to the column names: Data1 and Data2)
# move the legend to the top left
axes.legend(['Random 1', 'Random2'], loc='upper left')
```

Out[8]:

```
<matplotlib.legend.Legend at 0x118d52e48>
```



As an example Series, we're going to use a Time Series. Of course, you could use other Series' that have an index other than time; the plotting will happen the same way. But Time Series is a common type of Series data.
For our Time Series data, we will use random numbers between 0 and 1.

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

%matplotlib inline

# create random data for our Time Series
# the seed to generate the same random numbers as in this example
seedValue =20
np.random.seed(seedValue)

# Create a range of dates that will become the index
dateRange = pd.date_range('2016-01-01', '2019-10-07')

# Generate the random data values (1 for each of the dates in the date range)
numberGen = len(dateRange)#1376 numbers
data = np.random.randn(numberGen)

#Create the Time Series (a Series with a DateTimeIndex)
timeSeries = pd.Series(data, index = dateRange)
timeSeries.plot()
```

Out[3]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x118d655f8>
```