**Data Analysis using numpy and pandas**

**Introduction**:

When python was created, the main focus was on creating applications (web and otherwise) with the least anount of effort. Python, at that time, boasted of a python command line where one could run pieces of code to test them and once they worked fine, they could be incorporated into the main application code. So, a lot of programming time was saved, and people (including me) liked it. (I happen to come from a C and Perl background before I befriended Python, and I fell in love at the first sight). At that time, it was a great language to make business apps/processes and do automation of mundane tasks like testing applications, QA, etc. It did have some good math library back then, and you could use it to manipulate numerical data, plot basic charts, and so on. However, with the advent of big data, and Google's usage of it, Python was poised to take the center stage. Google employed Guido, the creator of the Python programming language and took the lead to utilize big data. With this in mind, Python took a major leap. Out came some of python's wonderful packages, frameworks and libraries to handle big data, numerical analysis, spatial analysis (things like face recognition), etc, and the two major packages in use today for such purposes are Numpy and Pandas. So here, we will take a rather shallow view of them since space is limited, but we will try to cover all aspects of the packages mentioned above (Each of these 2 packages can be written as books with atleast 300 pages each, so trying to go deep into them would not be possible here).

**NumPy:**

Numpy provides the programmer with a set of tools to handle lots of data in a relatively easy way. The most prominent datastructure provided by numpy is 'ndarray'. It is an array that is indexed numerically like a standard python array (lists and tuples), but it is not a heterogenous array like a python list or tuple. So, that begs the question why is it special and more useful than standard python array datastructures? Well, firstly, it can hold multi-dimensional data. Doing that with a python list or tuple is cumbersome. Apart from this, it has certain attributes that gives us an insight as to what the ndarray contains. We will explain some of the more important ones here, but we will try to list as many as possible here.

The attributes are: shape, dtype, offset, buffer, order and strides. These are the ones that are more notable, but there are others that serve a specific purpose in a specific scenario. We will be skipping them here.

In order to create numpy 'ndarray', we have 3 options, depending on what type of ndarray we are trying to create. These options are “array”, “zeros” and “empty”. There is also a low level constructor named “ndarray” with which we can create an ndarray. Take a look at the following example:

import numpy as np

np.ndarray(shape=(2,2), dtype=float,order='F')

*The result would be something like the following, as it is randomly generated.*

array([[0.00000000e+000, 6.10591585e-309], [3.24370112e+178, 6.52709565e+091]])

Now, let us take a look at some of the parameters of the “ndarray” constructor:

*size:* Number of elements in the array

*dtype:* Datatype of the elements in the ndarray

*data:* Pointer to the start of the array's data

*flags:* Defines the memory layout of the array

*ndim:* Number of dimensions of the array

*shape:* Tuples of array dimensions. We will look at it more closely later.

*strides:* Tuple of bytes to traverse in each direction when traversing the array.

*Itemsize:* size of one array element in bytes.

There are quite a few other attributes that we skipped here. If you want to get a detailed view of them, please refer to the following page: *https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.ndarray.html*

ndarray also has quite a few methods, but before we get into them, let us take a closer look at the 'shape' attriute.

An 'ndarray' is an n-dimensional array and the attribute 'shape' defines the length of these dimensions. The 'shape' attribute is a tuple of integers. For example, an array of one dimension (a list) will have a single element in the tuple that defines the length of the list. A 2D array will have 2 elements in the shape tuple and so on. Let's see some examples here:

*import numpy as np*

*arr = np.array([10, 11, 12])*

*arr.shape # Returns (3,)*

*# Please note above that the arguments to np.array()*

*# are passed as a list. Hence, the 'array' function treats*

*# them as a single argument.*

*arr2 = np.zeros((4, 5, 6)) # Note the elements are passed as tuples, not lists.*

*arr2.shape # Returns the tuple (4,5,6) as it identifies the tuple in the argument*

*# as an ndarray containing 3 dimensions of length, breadth and width as*

*# 4, 5, and 6 respectively.*

Another important associated function of ndarray is 'reshape()'. 'reshape()', as the name suggests, is a way of modifying the shape given to a datastructure (ndarray in this case). Please note that however, reshaping an array in-place will fail to create a copy. The parameters passed to the reshape() function are as follows:

Param #1: an array like object (could be an ndarray, or a normal python list). This is a required parameter.

Param #2: An integer, or a tuple of integers – this will be used to reshape the first argument. This is a required parameter.

Param #3: order, and this could be any of 'C', 'F' and 'A'. 'C' means to read and write the array in a C like indexing order (which means the outermost index changes the fastest and the innermost changes slowest). 'F' stands for Fortran like order, which is just the opposite of 'C' indexing. 'A' means to read/write the elements of the array in the first argument if it is contiguous in memory in a Fortran like manner, and C like manner otherwise. This is an optional parameter.

The return value is a ndarray which has been reshaped by the second parameter. Let's see some examples:

*>>>nd = np.ndarray(shape=(2,2), dtype=float,order='F')*

*>>>nd*

*array([[0.00000000e+000, 3.24370112e+178],*

*[6.10591585e-309, 6.52709565e+091]])*

*>>>rnd = nd.reshape((4,1))*

*>>>rnd*

*array([[0.00000000e+000], [3.24370112e+178], [6.10591585e-309], [6.52709565e+091]])*

So as you can see above that reshaping nd with a tuple containing (4,1) converts it to a ndarray with 4 rows and 1 column.

During analysis of data, you will mostly get the data as a string. So, how do we convert that string to a ndarray so that we can do numpy operations on it? The answer is 'genfromtxt()'.

“genfromtxt()” basically runs a couple of loops. The first loop converts each line in a file into a sequence of strings. The second loop converts each of those strings to an appropriate data type. This function is basically slower than other single loop counterparts (like “loadtxt()”, but it is flexible and can handle missing data that loadtxt() cannot).

Let's look at some examples.

*>>> import numpy as np*

*>>> from io import StringIO*

*>>>*

*>>> data = u”5,4,3\n2,1,0”*

*>>> np.genfromtxt(StringIO(data), delimiter=”,”)*

*array([[5., 4., 3.,], [2., 1., 0.]])*

The argument 'delimiter' determines how the lines are to be split. For example, a CSV file will specify the ',' character as the delimiter, while a TSV will specify '\t' as the delimiter. By default, that is if the delimiter is not specified, then the whitespace characters are considered to be the delimiter. You can refer to the following material to learn more about genfromtxt(): *[https://docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html#splitting-the-lines-into-columns](https://docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html" \l "splitting-the-lines-into-columns)*

While you would find a lot of situations where you would want to use numpy by itself, you would also use numpy indirectly when you use the 'pandas' module. So now that we have glanced at numpy, let us focus a bit on pandas and see how pandas can help us solve data crunching needs using numpy under its hood.

**Pandas** :

The “pandas” module is specifically useful for computing and finding patterns in data and hence it is one of the most useful tools for any data scientist/engineer. However, most of the time you would not be using pandas alone, but you would possibly be using modules like scipy and matplotlib along with pandas to achieve your ends. Here, we will be focussing on pandas only, but in our examples we will make use of scipy and matplotlib to display data distribution and such things.

Pandas provides the data scientist/analyst with 2 different (and very useful data structures) named “series” and “dataframe”. A “series” is nothing but a labelled column of data that can hold any datatype (int, float, string, objects, etc). A pandas “series” can be created using the following constructor call:

*pandas.Series(data, index, dtype, copy)*

The argument “data” is a list of data elements (mostly passed as a numpy ndarray), “index” is a unique hashable list with the same length as the “data” argument. “dtype” defines the data type (Series is a homogeneous collection of elements), and “copy” specifies if copy flag is set. By default, this is false.

The following code snippet creates a Series:

*import pandas as pd*

*s = pd.Series()*

*print s*

*import numpy as np*

*data = np.array(['w', 'x', 'y', 'z'])*

*r = pd.Series(data)*

*print r*

The output would be as follows:

*Series([], dtype: float64)*

*0 w*

*1 x*

*2 y*

*3 z*

A Dataframe is a multidimensional table made up of a collection of Series. First, let us see how a Series can be created using a collection of data objects:

*Let's create a Series from a Numpy ndarray:*

*import numpy as np*

*import pandas as pd*

*d = np.array(['q', 'w', 'e', 'r', 't'])*

*sr = pd.Series(d)*

*# Diplay sr*

*print(sr)*

*The output would be as follows:*

*0 q*

*1 w*

*2 e*

*3 r*

*4 t*

*Note that the first column designates the index of the ndarray. The pandas “Series” takes them and makes use of it inside the data structure.*

*However, you are free to use other numeric sequences as index. For example, the Series created below will be using indices from 1000.*

*d = np.array(['q', 'w', 'e', 'r', 't'])*

*sr = pd.Series(d, index=[1000, 1001, 1002, 1003, 1004])*

*print(sr)*

*Now the output will look like this:*

*1000 q*

*1001 w*

*1002 e*

*1003 r*

*1004 t*

*A Series can also be created from other data structures like a dictionary, scalars, etc, in a similar manner. Please refer to the documentation for an indepth study on those cases. We can't handle it here due to lack of space. However, they are quite simple and it should not take more than 15 mins to understand them and start making use of them in your programs.*

*Next, we need a way to access data in a Series using a location (index). After all, that is the primary reason behind using these data structures. So let's delve into it. Here, I am taking an example that was inspired by an example in an online tutorial, but we will extend on it later.*

*d= np.array(['q', 'w', 'e', 'r', 't'])*

*sr = pd.Series(d, index=['a', 'b', 'c', 'd', 'e']) # Please note here that our indices are not numerical, but are ASCII non-numeric characters. This is supported.*

*# The first element may be accessed using the following syntax:*

*print(sr[0])*

*The output will be 'q'.*

*Now, let us suppose we need to extract the first n elements of the Series. So how do we do that? It is simple and it is similar to the way standard python lists are accessed:*

*print(sr[:n])*

*The above expression will display something like this*

*a 'q'*

*b 'w'*

*c 'e'*

*d 'r'*

*e 't'*

*...*

*...*

*This will print out the values till the nth element.*

Now, let us focus on Pandas dataframes for a while. As for Series, I would strongly suggest you to go through the official pandas documentation to fill in the gaps that I am leaving. This article is meant to get you started on this topic, but it will definitely fall short in the effort to make you an expert on the topic. This article is simply not meant for that purpose.

Pandas works by putting the input data in a data structure called the “Dataframe”. So if you provide pandas with a CSV file, it will create a dataframe with the data passed to it and that will allow you to do whatever operations you want to do to the data. It will allow you to answer questions like :

* What is the average mean, median and mode for every column of data in the dataframe
* How does the distribution of data in column 'i' look like.
* Does the second column have any correlation with the seventh column?

These are some of the questions data scientists and analysts will ask themselves and pandas would allow them to answer such questions easily.

Let's look at an example first to see how such questions might be answered using the pandas module.

*Suppose we have the following dictionary containing the number of students in first, second, third and fourth years.*

*course\_population = {*

*'maths' : [20, 24,19, 22],*

*'physics' : [31, 33, 42, 29],*

*'chemistry' : [22, 24, 21, 20]*

*}*

*Now, let's pass this to the dataframe constructor:*

*import pandas as pd*

*df = pd.DataFrame(course\_population, index=['first' 'second', 'third', 'fourth'])*

*print(df)*

*The output would be as follows:*

|  |  |  |  |
| --- | --- | --- | --- |
|  | *Maths* | *Physics* | *Chemistry* |
| *first* | 20 | 31 | 22 |
| *second* | 24 | 33 | 24 |
| *third* | 19 | 42 | 21 |
| *fourth* | 22 | 29 | 20 |

*So, each key-value pair represents a column in the dataframe.*

*We will see how we can access specific elements. But before that, let's play around with the data, so that we can have an understanding of it, which will help us in making decisions regarding what methods to use on it to analyse it.*