International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 12 | Dec 2020 www.irjet.net p-ISSN: 2395-0072

Scientific Computing and Data Analysis using NumPy and Pandas

Atharva Sapre[1], Shubham Vartak[2]

[1]Computer Engineering, Atharva College of Engineering, Mumbai, India
[2]Computer Engineering, Atharva College of Engineering, Mumbai, India

---

Abstract - *In this paper, we will understand the libraries including NumPy and Pandas. We will also study their functionalities for analyzing data sets common to finance, statistics, and other related fields. NumPy is the foundation library for scientific computing in Python since it provides data structures and high-performing functions that the basic packages of the Python cannot provide. Pandas is a new library which aims to facilitate working with these data sets and to provide a set of fundamental building blocks for implementing statistical models. We will conclude by discussing possible future directions for statistical computing and data analysis using Python.*

***Key Words*: NumPy, Pandas, Python, Package, Library, Foundation, Statistical computing, Data Analysis**

1.INTRODUCTION

Python is being utilized progressively in logical applications generally overwhelmed by R, MATLAB, Stata, SAS, other business or open-source research conditions. The development of what's more, security of the major mathematical libraries (SciPy, and others), nature of documentation, and accessibility of "kitchen-sink" conveyances (EPD, Pythonxy) have gone a long route toward making Python available and advantageous for a wide crowd. NumPy is an essential bundle for logical registering with Python and particularly for information examination. Indeed, this library is the premise of a lot of numerical and logical Python bundles. The information on the NumPy library is an essential to confront, in the most ideal way, all logical Python bundles, and especially, to utilize and see more about the pandas library and how to get the generally out of it. Pandas is a new Python library of information structures and factual apparatuses at first produced for quantitative money applications [1]. The majority of our models here come from time arrangement and cross-sectional information emerging in monetary display. The bundle's name comes from board information, which is a term for 3-dimensional informational collections experienced in insights and econometrics. We trust that pandas will help make logical Python a more alluring and functional factual figuring climate for scholastic and industry professionals the same.

2. The NumPy Library

NumPy stands for 'Numerical Python' and it constitutes the core of many other python libraries. It is most commonly used for scientific computing and especially for data analysis. This library, totally specialized for data analysis, is fully developed using the concepts introduced by NumPy. Built-in tools offered by standard python libraries prove to be inadequate for calculations in data analysis. That is why the knowledge of NumPy library is a prerequisite to analyse data and understand more about Pandas library [2].

2.1 Ndarray - The basis of NumPy library

In Python we have lists that serve the purpose of arrays, but they are slow to process. NumPy aims to provide an array object that is up to 50x faster than traditional Python lists. The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy. Arrays are very frequently used in data science, where speed and resources are very important [3].

The entire NumPy library depends on one principle object: ndarray (which represents N-dimensional exhibit). This object is a multidimensional homogenous array with a predetermined number of items: homogeneous on the grounds that all the items inside it are of a similar kind and of a similar size. The data type is indicated by another NumPy object called 'dtype' (data-type); each ndarray is related with just one sort of dtype. The number of the dimensions and array elements are characterized by its shape, a tuple of N-positive numbers that indicates the size for each dimension. The dimensions are characterized by axes and the number of axes as rank. Besides, another quirk of NumPy arrays is that their size is fixed, that is, when you define their size during creation, it stays unaltered. This property is unique in relation to Python lists, which can be altered. To define another ndarray, the easiest way is to use the **array()** function, passing a Python list containing the components as arguments.

```
a = np.array([1, 2, 3])
a
```

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

You can easily check what type of an object 'a' is by passing 'a' inside **type()** function

```
type(a)
```

```
numpy.ndarray
```

To know the associated dtype, use the dtype function

```
a.dtype
```

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

We can now use the **ndim** attribute for getting the axes, the **size** attribute to know the array length, and the **shape** attribute to get its shape.

```
print(a.ndim,end='\n')
print(a.size,end='\n')
print(a.shape,end='\n')
```

```
1
3
(3,)
```

Next crucial attribute is **itemsize** which defines the size in bytes of each item in the array, and **data** is the buffer containing the elements of the array.

```
print(b.itemsize,end='\n')
print(b.data,end='\n')
```

```
8
<memory at 0x7f67db3d4630>
```

We can also create multi-dimensional arrays using NumPy. For eg: if you define a two-dimensional array 2x2:

```
b = np.array([[1.3, 2.4],[0.3, 4.1]])
b
```

```
array([[1.3, 2.4],
       [0.3, 4.1]])
```

It will have the following properties:

```
print(b.ndim,end='\n')
print(b.size,end='\n')
print(b.shape,end='\n')
```

```
2
4
(2, 2)
```

2.2 Types of Data

NumPy arrays are designed to contain a wide variety of data types including Boolean, int8, int16, int64, float, complex, etc. For example, you can use the data type string:

```
g = np.array([['a', 'b'],['c', 'd']])
g
```

```
array([['a', 'b'],
       ['c', 'd']], dtype='<U1')
```

2.2.1 The dtype Option

The array function takes one more array called dtype where you can explicitly define the type of the array.

```
f = np.array([[1, 2, 3],[4, 5, 6]], dtype=complex)
f
```

```
array([[1.+0.j, 2.+0.j, 3.+0.j],
       [4.+0.j, 5.+0.j, 6.+0.j]])
```

2.3 Intrinsic Creation of an Array

The NumPy library provides a group of functions that generate the ndarrays with an initial content, created with some different value. In fact, they generate large amounts of knowledge using a single line of code. The zeros() function, for instance, creates a full array of zeros with dimensions using shape argument [2]. For instance, to create a two-dimensional array 3x3:

```
np.zeros((3, 3))
```

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

Similarly, we can create an array of ones:

```
np.ones((3, 3))

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

There is another feature **arange()** which generates Numpy arrays with sequences that correspond to the passed arguments. For instance, if you want to generate a sequence of values between 0 and 10, you will be passed only one argument to the function, that is the value with which you want to end the sequence.

```
np.arange(0, 10)

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

It is also possible to generate sequences with precise intervals by putting the third argument in the function.

```
np.arange(1, 10,2)

array([1, 3, 5, 7, 9])
```

Another function similar to **arange()** is **linspace()** . This function takes the first two arguments as the initial and end values of the sequence. In addition, the third argument defines the number of elements into which we want the interval to be split.

```
[14] np.linspace(0,10,5)

array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

random() function takes number of elements as the argument and fills the array with random numbers

```
[16] np.random.random(3)

array([0.00818431, 0.32479662, 0.30247164])
```

2.4 Arithmetic operators:

A number of arithmetic operations including can be performed on arrays. For instance:

```
a = np.arange(4)
a

array([0, 1, 2, 3])

[18] a+4

array([4, 5, 6, 7])

[19] a*4

array([ 0,  4,  8, 12])

[20] a/4

array([0.  , 0.25, 0.5 , 0.75])

[21] a-1

array([-1,  0,  1,  2])
```

2.5 The Matrix Product

In order to perform dot product of two matrices that is not element-wise, we use the **dot()** function.

For instance, the first code snippet is an element-wise product of matrix A and B.

The second code snippet is the matrix dot product of two matrices.

```
[22] A = np.arange(0, 9).reshape(3, 3)
     B = np.ones((3, 3))
     A*B

     array([[0., 1., 2.],
            [3., 4., 5.],
            [6., 7., 8.]])

[23] np.dot(A,B)

     array([[ 3.,  3.,  3.],
            [12., 12., 12.],
            [21., 21., 21.]])
```

2.6 Increment and Decrement Operators

We can use **+=** or **-=** operators to increment or decrement every element of an array by a certain value. Similarly, we can use ***=** or **/=** to multiple or divide every element of an array by a certain value.

```
[24] a = np.arange(4)
     a+=1
     a

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

[25] a-=1
     a

     array([0, 1, 2, 3])
```

2.7 Universal Functions

Functions including **sqrt(), log(), and sin()** come under this group. These functions work in element wise fashion.

```
[26] a = np.arange(1, 5)
     a

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

[27] np.sqrt(a)

     array([1.        , 1.41421356, 1.73205081, 2.        ])

[28] np.log(a)

     array([0.        , 0.69314718, 1.09861229, 1.38629436])

[29] np.sin(a)

     array([ 0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
```

2.8 Aggregate Functions

These sets of functions perform an operation on a set of values in an array to produce a single result.

```
[30] a = np.array([3.3, 4.5, 1.2, 5.7, 0.3])
     a.sum()

     15.0

[31] a.min()

     0.3

[32] a.max()

     5.7

[33] a.mean()

     3.0

     a.std()

     2.0079840636817816
```

2.9 Indexing

Indexing refers to **'[ ]'** to refer to the elements individually in order to extract a value, select a value or assign a new value.

```
[36] a=np.arange(10,16)
     a

     array([10, 11, 12, 13, 14, 15])

[37] a[4]

     14
```

You can also use negative indexes. For instance, **[-1]** can be used to refer to the last element of the array.

```
[38] a[-1]

     15
```

We can extract multiple items by passing an array of indexes within square brackets.

```
[39] a[[1,2,5]]

     array([11, 12, 15])
```

We can refer bidimensional array elements by using 2 values inside the square brackets. First value refers to the row number and the second one refers to the column number.

```
[40] A=np.arange(10,19).reshape((3,3))
     A

     array([[10, 11, 12],
            [13, 14, 15],
            [16, 17, 18]])

     A[1,2]

     15
```

2.10 Slicing

This function allows us to extract a part of an array to generate a new array.

```
[43] a=np.arange(10,16)
     a

     array([10, 11, 12, 13, 14, 15])

[46] a[1:4]

     array([11, 12, 13])
```

Now if we want to skip alternate elements from the previous array, we can do so by adding the third argument which defines the gap in the sequence of elements.

```
     a[1:4:2]

     array([11, 13])
```

Slicing also works on two dimensional arrays.

```
[49] A=np.arange(10,19).reshape((3,3))
     A

     array([[10, 11, 12],
            [13, 14, 15],
            [16, 17, 18]])

     A[0,:]

     array([10, 11, 12])
```

2.11 Iterating:

We can iterate the items in an array using a for loop.

```
[52] for i in a:
       print(i)

     10
     11
     12
     13
     14
     15
```

In the case of a bidimensional array, a single for loop with give us rows instead of a single element

```
[53] for row in A:
       print(row)

     [10 11 12]
     [13 14 15]
     [16 17 18]
```

If you need iteration element by element, we can use the **flat** function.

```
[55] for item in A.flat:
       print(item)

     10
     11
     12
     13
     14
     15
     16
     17
     18
```

2.12 Conditions and Boolean Arrays

For instance, there's an array containing 10 elements.

```
[56] A=np.random.random((4,4))
     A

     array([[0.34872296, 0.57636978, 0.28404594, 0.88679556],
            [0.06345152, 0.44537868, 0.7038293 , 0.58978474],
            [0.02338557, 0.38924813, 0.05563264, 0.21631681],
            [0.49616767, 0.09831178, 0.69327645, 0.31863093]])
```

Now, if we apply a condition, we will receive a boolean array containing true values in the positions in which the condition is satisfied.

```
[57] A<0.5

     array([[ True, False,  True, False],
            [ True,  True, False, False],
            [ True,  True,  True,  True],
            [ True,  True, False,  True]])
```

Now putting this condition inside the square brackets will extract only those elements who have satisfied the given condition.

```
A[A<0.5]

array([0.34872296, 0.28404594, 0.06345152, 0.44537868, 0.02338557,
       0.38924813, 0.05563264, 0.21631681, 0.49616767, 0.09831178,
       0.31863093])
```

2.13 Shape Manipulation

We can convert a one dimensional array into a matrix using **reshape()** function. For instance, let's create an array using **random()** function.

```
[3] a=np.random.random(12)
    a

    array([0.45514283, 0.13491042, 0.63457238, 0.98238381, 0.83834423,
           0.97550377, 0.44916751, 0.96212569, 0.25388814, 0.86213517,
           0.9774789 , 0.391008  ])
```

If we want to reshape this array into a matrix of 3 rows and 4 columns, we can do this by using the **reshape()** function:

```
[5] A=a.reshape(3,4)
    A

    array([[0.45514283, 0.13491042, 0.63457238, 0.98238381],
           [0.83834423, 0.97550377, 0.44916751, 0.96212569],
           [0.25388814, 0.86213517, 0.9774789 , 0.391008  ]])
```

We can also convert the above two dimensional array back to one dimensional array using **ravel()** function.

```
[9] B=A.ravel()
    B

    array([0.45514283, 0.13491042, 0.63457238, 0.98238381, 0.83834423,
           0.97550377, 0.44916751, 0.96212569, 0.25388814, 0.86213517,
           0.9774789 , 0.391008  ])
```

Another alternative to **ravel()** function is the **shape()** function.

```
[10] A.shape=(12)
     A

     array([0.45514283, 0.13491042, 0.63457238, 0.98238381, 0.83834423,
            0.97550377, 0.44916751, 0.96212569, 0.25388814, 0.86213517,
            0.9774789 , 0.391008  ])
```

We can perform transposition of a matrix using the **transpose()** function.

```
A.transpose()

array([[0.45514283, 0.83834423, 0.25388814],
       [0.13491042, 0.97550377, 0.86213517],
       [0.63457238, 0.44916751, 0.9774789 ],
       [0.98238381, 0.96212569, 0.391008  ]])
```

2.14 Array Manipulation

a) Joining Arrays.

We can join two different arrays using the NumPy library using the concept of stacking. For instance, we can use the function **vstack()** which adds a second array as a new row in the first array. Similarly, the **hstack()** function adds a second array as a new column in the first array.

```
[13] A=np.ones((3,3))
     B=np.zeros((3,3))
     np.vstack((A,B))

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

     np.hstack((A,B))

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

There are two more functions named **column_stack()** and **row_stack()** that stack multiple arrays as columns and rows to form a new two dimensional array respectively.

```
[15] a=np.array([0,1,2])
     b=np.array([3,4,5])
     c=np.array([6,7,8])
     np.column_stack((a,b,c))

     array([[0, 3, 6],
            [1, 4, 7],
            [2, 5, 8]])

     np.row_stack((a,b,c))

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

b) Splitting Arrays

We can also divide an array into several parts using **hsplit()** and **vsplit()**. The following code snippets show the working of these 2 functions:

```
A=np.arange(16).reshape((4,4))
A

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

[18] [B,C]=np.hsplit(A,2)
     B

     array([[ 0,  1],
            [ 4,  5],
            [ 8,  9],
            [12, 13]])

[19] C

     array([[ 2,  3],
            [ 6,  7],
            [10, 11],
            [14, 15]])

[20] [B,C]=np.vsplit(A,2)
     B

     array([[0, 1, 2, 3],
            [4, 5, 6, 7]])

[21] C

     array([[ 8,  9, 10, 11],
            [12, 13, 14, 15]])
```

There is another command for splitting arrays. It is the **split()** function, which allows you to split the array into nonsymmetrical parts. In addition, passing the array as an argument, you have also to specify the indexes of the parts to be divided. If you use the option axis = 1, then the indexes will be those of the columns; if instead the option is axis = 0, then they will be the row indexes [2].

a) Axis=1

```
[22] [A1,A2,A3] = np.split(A,[1,3],axis=1)
     A1

     array([[ 0],
            [ 4],
            [ 8],
            [12]])

[23] A2

     array([[ 1,  2],
            [ 5,  6],
            [ 9, 10],
            [13, 14]])

[24] A3

     array([[ 3],
            [ 7],
            [11],
            [15]])
```

b) Axis=0

```
[25] [A1,A2,A3] = np.split(A,[1,3],axis=0)
     A1

     array([[0, 1, 2, 3]])

[26] A2

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

[27] A3

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

2.15 Copies of Objects

Every assignment doesn't produce a copy of an array or any element contained within them.

```
[28] a=np.array([1,2,3,4])
     b=a
     b

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

     a[2]=0
     b

     array([1, 2, 0, 4])
```

If you assign array a to array b, you're not copying the contents of 'a' into 'b'. Instead, you're giving a and alternate name 'b'.

```
[30] c=a[0:2]
     c

     array([1, 2])

     a[0]=0
     c

     array([0, 2])
```

But if you want to generate a copy of an array, you can do so by using the **copy()** function.

```
a=np.array([1,2,3,4])
c=a.copy()
a[0]=0
c

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

2.16 Structured Arrays

Along with mono-dimensional and multidimensional arrays, NumPy allows us to create arrays that are complex not only in size but also in structure. These arrays are known as **structured arrays**.

```
struc=np.array([(1,'First',0.5,1+2j),(2,'Second',1.3,2-2j),
                (3,'Third',0.8,1+3j)],dtype=('int16,a6,float32,complex64'))
struc

array([(1, b'First', 0.5, 1.+2.j), (2, b'Second', 1.3, 2.-2.j),
       (3, b'Third', 0.8, 1.+3.j)],
      dtype=[('f0', '<i2'), ('f1', 'S6'), ('f2', '<f4'), ('f3', '<c8')])
```

2.17 Reading/Writing data on files

One of the important aspects of NumPy is its ability to read or write data into files. These methods are useful especially when you are handling large amounts of data. These operations are quite common in data analysis since the size of data to be analyzed is always huge. Numpy allows the reading and conversion of data within a file into an array.

2.17.1 Loading and Saving data in binary files

We can save or retrieve data stored in binary format using the functions **save()** and **load()**. Let's create a multidimensional array named data.

```
[38] data=np.random.random((4,4))
     data

     array([[0.89936649, 0.64797475, 0.13575682, 0.03384031],
            [0.88064697, 0.5539541 , 0.53487475, 0.81991136],
            [0.82515575, 0.1032317 , 0.87745807, 0.75177546],
            [0.8724328 , 0.53801681, 0.37093905, 0.31900059]])
```

For instance, you are working on your data and you want to save your results, then you can simply call the **save()** function to save data in a file which will have a **.npy** extension.

```
[39] np.save('saved_data',data)
```

Now, you can use the **load()** function to retrieve the data stored in that file.

```
load_data=np.load('saved_data.npy')
load_data

array([[0.89936649, 0.64797475, 0.13575682, 0.03384031],
       [0.88064697, 0.5539541 , 0.53487475, 0.81991136],
       [0.82515575, 0.1032317 , 0.87745807, 0.75177546],
       [0.8724328 , 0.53801681, 0.37093905, 0.31900059]])
```

2.17.2 Reading file with tabular data

Many times, the data you want to read are stored in **.csv/.txt** formats. These formats store data in a way that is completely different from the **.npy** format. In order to read data from such a file, NumPy provides a function **genfromtxt()** which takes 3 arguments including the name of the file, the character that separates a value from another, and the column header. If you want the function to guess the actual data type of each element then you can add the fourth argument **dtype=None.** This is because this function assumes every element as a float value by default. So any non-float values will be converted to NaNs in output.

Let's assume we have a data file named 'data.csv'.The data.csv file contains the following data:

	A	B	C	D
1	id	value1	value2	value3
2	1	123	Atharva Sapre	25
3	2	110	Shubham Vartak	56
4	3	164	XYZ	78
5				

Now let's use the **genfromtxt()** function to retrieve data from the file.

```
data=np.genfromtxt('data.csv',delimiter=',',names=True,dtype=None)
data

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: VisibleDeprecationWarni
  """Entry point for launching an IPython kernel.
array([(1, 123, b' Atharva Sapre', 25), (2, 110, b'Shubham Vartak', 56),
       (3, 164, b'XYZ', 78)],
      dtype=[('id', '<i8'), ('value1', '<i8'), ('value2', 'S14'), ('value3', '<i8')])
```

3. PANDAS - Python Data Analysis Library

Pandas is a Python bundle giving quick, adaptable, and expressive information structures intended to make working with "social" or "marked" information both simple and instinctive. It means to be the major elevated level structure block for doing down to earth, certifiable information examination in Python. Also, it has the more extensive objective of turning into the most remarkable and adaptable open source information investigation/control device accessible in any language. It is as of now well on its way toward this objective. pandas is appropriate for a wide range of sorts of information:

- Tabular information with heterogeneously-composed sections, as in a SQL table or Excel bookkeeping page
- Ordered and unordered (not really fixed-recurrence) time arrangement information.
- Arbitrary network information (homogeneously composed or heterogeneous) with line and section marks
- Any other type of observational/measurable informational indexes. The information really need not be named at all to be put into a pandas information structure.

3.1 Prologue to pandas Data Structures:

The core of pandas is only the two essential information structures on which all exchanges, which are for the most part made during the investigation of information, are concentrated:

- Series

- DataFrame

3.2 The Series :

The Series is the object of the pandas library intended to speak to one-dimensional information structures, likewise to a cluster however with some extra highlights. Its interior structure is basic and is made out of two exhibits related with one another. The primary cluster has the reason to hold the (information of any NumPy type) to which every component is related with a mark, contained inside the other exhibit, called the Index.

```
import numpy as np
import pandas as pd
data = np.array(['S','P','E','E','D'])

ser = pd.Series(data)
print(ser)
```

```
0    S
1    P
2    E
3    E
4    D
dtype: object
```

3.3 Operations between Series:

One of the extraordinary possibilities of this sort of information structures is the capacity of Series to adjust information tended to contrastingly between them by recognizing their relating name. In the accompanying model you aggregate two Series sharing just a few components practically speaking with name.

```
mydict = {'red': 2000, 'blue': 1000, 'yellow': 500, 'orange': 1000}
myseries = pd.Series(mydict)
colors = ['red','yellow','orange','blue','green']
myseries = pd.Series(mydict, index=colors)
mydict2 = {'red':400,'yellow':1000,'black':700}
myseries2 = pd.Series(mydict2)
myseries + myseries2
```

```
black        NaN
blue         NaN
green        NaN
orange       NaN
red       2400.0
yellow    1500.0
dtype: float64
```

3.4 The DataFrame

The DataFrame is an even information structure fundamentally the same as the Spreadsheet (the most recognizable are Excel accounting pages). This information structure is intended to expand the instance of the Series to numerous measurements. Truth be told, the DataFrame comprises an arranged assortment of sections, every one of which can contain an estimation of various sorts (numeric, string, Boolean, and so on).What's cool about Pandas is that it takes data (like a CSV or TSV file, or a SQL database) and creates a Python object with rows and columns called data frame that looks very similar to table in a statistical software (think Excel or SPSS for example. People who are familiar with R would see similarities to R too).

In contrast to Series, which had an Index exhibit containing names related to every component, for the situation of the information outline, there are two record exhibits. The first, related to the lines, has fundamentally the same as capacities to the record exhibit in Series. Indeed, each name is related with all the qualities in the column. The subsequent exhibit rather contains a progression of marks, each related with a specific segment. A DataFrame may likewise be perceived as a dict of Series, where the keys are the segment names and values are the Series that will shape the segments of the information outline. Moreover, all components of every Series are planned by a variety of marks, called Index.

```
data = {'color' : ['blue','green','yellow','red','black'],
        'object' : ['ball','pen','pencil','paper','mug'],
        'price' : [12,5.0,2,0.9,1.7]}
frame = pd.DataFrame(data)
frame
```

	color	object	price
0	blue	ball	12.0
1	green	pen	5.0
2	yellow	pencil	2.0
3	red	paper	0.9
4	black	mug	1.7

In the event that the item dict from which we need to make a DataFrame contains more information than we are intrigued, you can make a choice. In the constructor of the information outline, you can determine a succession of segments, utilizing the sections choice. The segments will be made in the request for the grouping paying little mind to how they are contained inside the article dict.

```
frame2 = pd.DataFrame(data, columns=['object','price'])
frame2
```

	object	price
0	ball	12.0
1	pen	5.0
2	pencil	2.0
3	paper	0.9
4	mug	1.7

3.5 Tasks between Data Structures

Since you have gotten comfortable with the information structures, for example, Series and DataFrame and you have perceived how different rudimentary activities can

be performed on them, it's an ideal opportunity to go to tasks including at least two of these structures. For instance, in the past area we perceived how the number-crunching administrators apply between two of these objects. Presently in this segment you will develop more the subject of activities that can be performed between two information structures.

3.5.1 Adaptable Arithmetic Methods

You've quite recently perceived how to utilize numerical administrators straightforwardly on the pandas information structures. The equivalent activities can likewise be performed utilizing suitable strategies, called Flexible number juggling techniques.

- add()
- sub()
- div()
- mul()

To call these capacities, you'll need to utilize a detail unique in relation to what you're utilized to managing numerical administrators. For instance, rather than composing a total between two DataFrame

'frame1 + frame2', you'll need to utilize the accompanying arrangement:

```
frame1 = pd.DataFrame(np.arange(16).reshape((4,4)),
index=['red','blue','yellow','white'],
columns=['ball','pen','pencil','paper'])
frame2 = pd.DataFrame(np.arange(12).reshape((4,3)),
index=['blue','green','white','yellow'],
columns=['mug','pen','ball'])
frame1.add(frame2)
```

	ball	mug	paper	pen	pencil
blue	6.0	NaN	NaN	6.0	NaN
green	NaN	NaN	NaN	NaN	NaN
red	NaN	NaN	NaN	NaN	NaN
white	20.0	NaN	NaN	20.0	NaN
yellow	19.0	NaN	NaN	19.0	NaN

As should be obvious the outcomes are equivalent to what you'd get utilizing the expansion administrator '+'. You can also note that if the indexes and column names differ greatly from one series to another. You'll end up with another information outline loaded with NaN esteems.

3.5.2 Activities among DataFrame and Series

Returning to the number-crunching administrators, pandas permits you to make exchanges even between various structures such as a DataFrame and a Series. For instance, we characterize these two structures in the following way.

```
frame = pd.DataFrame(np.arange(16).reshape((4,4)),
index=['red','blue','yellow','white'],
columns=['ball','pen','pencil','paper'])
ser = pd.Series(np.arange(4), index=['ball','pen','pencil','paper'])
frame - ser
```

	ball	pen	pencil	paper
red	0	0	0	0
blue	4	4	4	4
yellow	8	8	8	8
white	12	12	12	12

3.5.3 Function Application and Mapping

The pandas library is based on the establishments of NumPy, and afterward expands a large number of its highlights adjusting them to new information structures as Series and DataFrame. Among these are the widespread capacities, called **ufunc**. This class of capacities is specific since it works by component in the information structure.

```
frame = pd.DataFrame(np.arange(16).reshape((4,4)),
index=['red','blue','yellow','white'],
columns=['ball','pen','pencil','paper'])
frame
```

	ball	pen	pencil	paper
red	0	1	2	3
blue	4	5	6	7
yellow	8	9	10	11
white	12	13	14	15

For example you could calculate the square root of each value within the data frame, using the NumPy np.sqrt().

```
np.sqrt(frame)
```

	ball	pen	pencil	paper
red	0.000000	1.000000	1.414214	1.732051
blue	2.000000	2.236068	2.449490	2.645751
yellow	2.828427	3.000000	3.162278	3.316625
white	3.464102	3.605551	3.741657	3.872983

3.5.4 Capacities by Row or Column

The use of the capacities isn't restricted to the ufunc capacities, yet additionally incorporates those characterized by the client. Interestingly, they work on a one-dimensional cluster, giving a solitary number for result. For instance, we can characterize a lambda work that figures the reach canvassed by the components in a cluster.

```
f = lambda x: x.max() - x.min()
frame.apply(f)
```

```
ball      12
pen       12
pencil    12
paper     12
dtype: int64
```

The result, however, this time it is only one value for the column, but if you prefer to apply the function by row instead of by column, you have to specify the axis option set to 1.

```
frame.apply(f, axis=1)
```

```
red       3
blue      3
yellow    3
white     3
dtype: int64
```

3.6 pandas: Reading and Writing Data

In this segment you will see the entirety of the instruments given by pandas to perusing information put away in numerous sorts of media, (for example, documents and information bases). In equal, you will likewise perceive how to compose information structures straightforwardly on these organizations, without agonizing a lot over the innovations utilized. This segment is centered around a progression of I/O API works that pandas gives to encourage as much as conceivable the perusing and composing information measure straightforwardly as DataFrame objects on the entirety of the most usually utilized configurations. You begin to see the content documents, at that point move bit by bit to more mind boggling parallel organizations. Toward the finish of the part, you'll additionally figure out how to interface with every single normal information base, both SQL and NoSQL, with models telling the best way to store the information in a DataFrame straightforwardly in them. Simultaneously, you will perceive how to peruse the information contained in an information base and recover them as of now as a DataFrame.

3.6.1 I/O API Tools

pandas is a library particular for information investigation, so you expect that it is basically centered around estimation and information handling. In addition, even the way toward composing and perusing information from/to outer documents can be viewed as a piece of the information preparing. Indeed, you will perceive how, even at this stage, you can play out certain tasks to set up the approaching information to additional controls. Accordingly, this part is significant for information examination and along these lines a particular instrument for this reason should be available in the library pandas: a bunch of capacities called I/O API. These capacities are partitioned into two primary classes, totally even to one another: perusers and authors.

Readers	Writers
read_csv	to_csv
read_excel	to_excel
read_hdf	to_hdf
read_sql	to_sql
read_json	to_json
read_html	to_html
read_stata	to_stata
read_clipboard	to_clipboard

3.6.2 Reading Data in CSV or Text Files

From normal experience, the most widely recognized activity for an individual moving toward information investigation is to peruse the information contained in a CSV record, or possibly in a content document. In order to see how pandas handle this kind of data, start by creating a small CSV file in the working directory as shown in figure below and save it as CompleteDataset.csv.

```
w1=pd.read_csv('CompleteDataset.csv')
w1.head()
```

	Unnamed: 0	Name	Age	Photo	Nationality	Flag	Overall
0	0	Cristiano Ronaldo	32	https://cdn.sofifa.org/48/18/players/20801.png	Portugal	https://cdn.sofifa.org/flags/38.png	94
1	1	L. Messi	30	https://cdn.sofifa.org/48/18/players/158023.png	Argentina	https://cdn.sofifa.org/flags/52.png	93
2	2	Neymar	25	https://cdn.sofifa.org/48/18/players/190871.png	Brazil	https://cdn.sofifa.org/flags/54.png	92
3	3	L. Suárez	30	https://cdn.sofifa.org/48/18/players/176580.png	Uruguay	https://cdn.sofifa.org/flags/60.png	92
4	4	M. Neuer	31	https://cdn.sofifa.org/48/18/players/167495.png	Germany	https://cdn.sofifa.org/flags/21.png	92

5 rows × 75 columns

3.6.3 Utilizing RegExp for Parsing TXT Files

In different cases, it is conceivable that the records on which to parse the information don't show separators very much characterized as a comma or a semicolon. In these cases, the customary articulations go to our guide. Indeed, you can indicate a regexp inside the read_table() work utilizing the sep alternative. To more readily comprehend the utilization of a regexp and how you can apply it as a basis for partition of qualities, you can begin from a straightforward case. For instance, assume that your record, for example, a TXT document, has values isolated by spaces or tabs in a flighty request. For this situation, you need to utilize the regexp in light of the fact that just with it you will consider as a separator the two cases. You can do that utilizing the trump card/s*. /s represents space or then again tab character (on the off chance that you needed to demonstrate just the tab, you would have utilized/t), while the pound shows that these characters might be different . That is, the qualities might be isolated by more spaces or more tabs.

```
pd.read_table('text.txt')
```

	This file contains the actual data for your assignment - good luck!
0	9764 3155
1	Why should you learn to write programs?
2	Writing programs (or programming) is a very cr...
3	and rewarding activity. You can write program...
4	many reasons, ranging from making your living ...
...	...
481	fresh eyes. I assure you that once you learn ...
482	you will look back and see that it was all rea...
483	took you a bit of time to absorb it.
484	42
485	The end

486 rows × 1 columns

3.6.4 Pickling with pandas

As respects the activity of pickling (and unpickling) with the pandas library, everything stays a lot encouraged. No compelling reason to import the cPickle module in the Python meeting and furthermore the entire activity is performed certainly.

Additionally, the serialization design utilized by pandas isn't totally in ASCII.

```
frame = pd.DataFrame(np.arange(16).reshape(4,4), index = ['up','down','left','right'])
frame.to_pickle('frame.pkl')
pd.read_pickle('frame.pkl')
```

	0	1	2	3
up	0	1	2	3
down	4	5	6	7
left	8	9	10	11
right	12	13	14	15

Presently in your working catalog there is another record called frame.pkl containing all the data about the edge DataFrame. To open a PKL file and read the contents, simply use the command

pd.read_pickle('frame.pkl')

3.6.5 Perusing and Writing Data on Microsoft Excel Files

In the past area, you perceived how the information can be effectively perused from CSV records. It isn't exceptional, nonetheless, that there is information gathered in plain structure in the Excel bookkeeping page. pandas give explicit capacities likewise to this sort of arrangement. You have seen that the I/O API gives two capacities to this reason:

- to_excel()
- read_excel()

As respects perusing Excel records, the read_excel() work can peruse both Excel 2003 (.xls) documents and Dominate 2007 (.xlsx) records. This is conceivable gratitude to the coordination of the inside module xlrd. To start with, open an Excel record and enter the information as appeared in Figure below. At that point save it as data.xls.

```
pd.read_excel('data.xlsx')
```

	Name	College	GPA
0	Rohan	VJIT	8.99
1	Arav	SPIT	9.23
2	Rina	Atharva	8.20
3	Gaurav	TSEC	7.23

3. CONCLUSIONS

NumPy and Pandas are powerful open source libraries for data science, data manipulation and visualization. Both the libraries have grown tremendously in popularity to an estimated 5 to 10 million users and become "must-use" tools for data scientists worldwide. These libraries have been receiving regular updates too. There has been a lot of improvement of the NumPy library over the years and it has been optimized greatly. Recent upgradations of pandas include method chaining, deprecation of 'inplace' parameter, inclusion of Apache Arrow, supporting extension arrays, and other deprecations. In conclusion, pandas and NumPy provide a firm foundation upon which a very powerful data analysis ecosystem can be established.

REFERENCES

[1] McKinney, W. (2012). pandas: powerful Python data analysis. Wes McKinney. http://github.com/pydata/pandas

[2] Nelli, F. (2015). *Python Data Analytics*. Apress.

[3] Python Numpy. W3Schools. www.w3schools.com/python/numpy_into.asp

[4] Bronshtein, A. (2017, April 17th). *A Quick Introduction to the "Pandas" Python Library*. Towards Data Science. https://towardsdatascience.com/a-quick-introduction-to-the-pandas-python-library-f1b678f34673