Friday, 30 December 2022

NumPy and Pandas

**Numpy**

* Numpy forms the basis of pandas. It introduced new data structures which makes data processing more convenient for data analysis.
* Both numpy and pandas has some built in functions which helps for data analysis.

**Pandas**

* Python’s widely used library for data analysis.
* Most commonly used data structure of pandas is data frames which is similar to SQL tables or Excel sheets. It has much for features than both of them.
* Data frame has column headers on top of each column and row numbers for each row on the left.

**Numpy -**

* Numpy stands for numerical python. It is the foundation for other libraries like pandas. Pandas data frames are built on top of numpy arrays so it can leverage numpy functions.
* Since pandas uses numpy arrays for data frames, Pandas just add wrappers like indices and titles for working with the arrays in a more convenient way.
* **Numpy Arrays.**
* Fixed size containers of items that are more efficient than python list and tuples for data processing.

1) They can store only a single type of data. Any mixed type will be stored as object datatype.

2) They can one dimensional or multi dimensional

3) Array elements can be modified but the size cannot be changed.

These makes arrays more efficient than list and tuples. Ie arrays has to store a single type of data and the size is fixed so python can calculate the size of the object and allocate memory for the objects in contiguous memory locations. This makes the operations faster.

* The first step for creating numpy array is to import numpy library
* Import numpy as np

np is standard alias for numpy library

* The next step is to create a list or tuple to store the data and pass it to the array() function of the numpy object to create a numpy array.

eg:- sales = [0,5,12,22,45,56]

sales\_array = np.array(sales)

* The array() function will convert the list or tuple to numpy array.
* NumPy arrays have the following properties
  + - * ndim - number of dimensions / axis es (common are 1d and 2d)
      * Shape - size of the array for each dimension
      * Size - total number of elements in the array
      * dtype-datatype of elements in the array
* To check the type of the array we can use the type() function

eg : - type(sales\_array)

will give numpy.ndarray

* To create a 2d array first create a nested list then pass it into the array function

eg:- sales = [[1,2,3,4],[5,6,7,8]]

sales\_array = array(sales)

it will give

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

[5,6,7,8]])

* One important thing to note here is that the nested lists must be of equal lengths
* To transpose a 2d array just use array\_name.T
* **Array Creation**
* Instead of creating arrays from the list we can create arrays using different methods.
* One way to do this is through np.zeroes((rows,cols),dtype) and np.ones((rows,cols),dtype) which will create arrays of zeroes and ones respectively.

eg:- np.ones(4,)

will give array([1.,1.,1.,1.,])

similarly np.zeroes((2,5),dtype=int)

will give a 2d array of zeroes with integer datatype

array([[0,0,0,0,0],

[0,0,0,0,0]])

* Np.arange(start, stop,step) - is equivalent to range() function in python which let’s you create array of elements with a range.

eg:- np.arange(10)

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

by default start is 0 and step is 1. Here stop is not included

* Np.linspace(start,stop,step) will create an array of n floating point numbers between the start and stop where n is the step which we provide as argument.

eg:- np.linspace(0,100,5)

array([0.,25.,50.,75.,100.])

Here stop is included.

**Use to get evenly spaced intervals.**

* Np.reshape(row,cols) lets you reshape an array with the provided row and column if applicable.

eg:- np.arange(1,9,2).reshape(2,2)

will give

array([[1,3],

[5,7]])

by default np.arange(1,9,2) will generate a 1d array of odd numbers from 1 to 7(because here 9 is not inclusive). By this we can split it into 2 rows.

We can also chain the reshape() method with other methods

* np.identity(n) will create an identity matrix of nxn dimension.

eg:- np.identity(3)

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

[0.,1.,0.],

[0.,0.,1.]])

We can reshape this array using reshape() method if we want to.

* numpy has a lot of mathematical functions we can utilise but we might not need to utilise all of that.

**Random number arrays**

* we can generate random numbers in various ways.

1) default\_rng(seed) - creates a random number generator. Seed is for reproducibility. Ie if you use a particular seed to generate a sequence of numbers another person can also use this same seed to generate the same sequence.

To use this we need to import it from numpy

ie, from numpy.random import default\_rng or you can directly reference it by np.random.default\_rng(seed).

example:-

rng = default\_rng(12345)

Np.default\_rng(seed)

The seed is only optional. But make sure that you create a seed so that you or other people can recreate your work.

2) random() returns n number of random values uniformly distributed between 0 and 1

eg: random\_array = rng.random(10)

will give 10 values in which 5 are less than 0.5 and the remaining 5 are greater than 0.5

3) normal() returns n number of random numbers from a normal distribution

rng.normal(mean,standard\_deviation,n) where n is the number of values we need.

rng is the standard variable name for the default number generator.

eg:- rng = default\_rng(12345)

mean ,stddev = 5,1

random\_normal = rng.normal(mean,stddev,size = 10)

* If you need random integers you can use rng.integers(start, stop, size)

eg:- rng.integers(0,10,100)

this will give 100 random values between 0 and 10.

**Indexing and Slicing Arrays**

* we can access array elements with index positions like we do for lists. We can also slice array similarly like a list using start : stop : step.
* We can use -ve indexes to get the elements from the last position of the array
  + Example: array1 = np.array([‘apple’,’banana’,’orange’]). We can use array1[-1] to get ‘orange’
  + Note that the first element start from zero and the last element starts from -1

- To get n elements from the end of the array we can use array[-n:].

* To get n number of elements from the array we can use the array[:n]. Here we are not specifying the start position, so by default it is assumed as 0, similarly we are not specifying step so it is assumed to be 1. It will return the n number of elements that is the [index + 1]th element.

eg:- array2[:2] will give array([‘apple’,’banana’])

If you want to get the last 2 elements you can use array2[1::] it will output array2([‘banana’,’orange’]). Note that in this it will use the index number we provide for start and we will get elements from that position(inclusive)

* In case of 2D arrays we need to specify the row and column to access an element. So similarly in the case of slicing we need to provide the slicing parameters for row and columns
* Example:-product= array([‘fruits’,’vegetables’,’cereal’,’dairy’,’eggs’],

[‘snacks’,’beverages’,’coffee’,’tea’,’spices’])

To get ‘coffee’ we need to provide product[1,2]

- If you want to get elements from all rows and from a particular column n

we can use array[:,n:]

eg:- product[ : , 2:] will return all the rows and elements starting from the 2nd

Column

ie, array([‘cereal’,’dairy’,’eggs’],

[‘coffee’,tea’,’spices']);

If we want to output a particular column we can also use array[:,c]

Where c is the column index.

**Array Operations**

* arithmetic operators can be used to perform array operations.
* If you need to add a number to each value of the array we can just specify the array\_name + value

eg:- sales\_array = array([0,5,7,9])

sales\_array + 2

will be array([2,7,9,11])

* if two arrays have same length we can directly multiple both the arrays using \*.

eg:- sales\_array = array([0,5,155,0,518]

,[0,1827,616,317,325])

if you want to choose 1st row as quantity and 2nd row as price of the quantity we can use slicing method and convert them to independent arrays and multiply them like :

Quantity = sales\_array[0: , :]

Price = sales\_array[1: , :]

Quantity \* Price

We will get 1d array which contains the product of quantity and price.

* Numpy array eliminates the need for writing loops to loops to perform operations.
* Array operations are applied by vectorisation and broadcasting that is why we don’t need to use for loops. Behind the scenes the looping is happening but those looping statements are written in C which is much more memory efficient than python. Also that we have defined the array elements in contiguous memory locations the iteration become faster.
* We can use any arithematic operator in python on dumpy arrays.
* **Note that we can always convert numpy arrays to sequence types of python such as lists and tuples.**
* We can use array\_name.sum() method to find the sum of an array, array\_name.round(n) to round the value of floating point numbers of an array to n digits.

**Filtering Arrays**

* Filtering data is a core and important functionality. For filtering we do a logical test on the array and only the elements that satisfies the condition will be returned.
* If we perform a logical operation on an array it will return an array with elements with true for the elements where the condition is met and false for the conditions not met.
* Eg: - sales\_array = array([[0,5,155,0,518],

[0,1827,616,317,325]])

sales\_array != 0

returns array([[False,True,True,False,True],

[False,True,True,True,True]])

* Another example for using this is sales\_array[sales\_array != 0]

This will return all the array elements which are not equal to 0.

array([5,155,518,1827,616,317,325])

* We can also use and and or operators for logical tests.

**Note that we use & for and and | for or**

**Also note that if there is more than one condition we should enclose each condition in paranthesis(), otherwise we will get error.**

* Example sales\_array[(sales\_array == 616 | (sales\_array < 100)]

sales\_array[(sales\_array > 100) & (sales\_array < 500)]

* We can also use variables to define logic that and place the variable instead of the entire logical operation for filters.

eg:- mask = (sales\_array > 100 ) & (sales\_array < 500)

sales\_array[mask]

we typically use the name mask

It is a good practice to use masks if you have more than one condition

* We can use values of one array to filter out values of another array

eg:- product\_array = array([‘fruits’,’vegetables’,’cereal’,’dairy’,’eggs’])

We can filter values of product array based on the values of sales array. If we need to filter out the the items that has no sales we can use

product\_array[sales\_array > 0]

We will get array([‘vegetables’,’cereal’,’eggs’])

Note that the size of both arrays must be equal.

* We can also modify array values. But this is not recommended to use unless there is a particular reason.

To change the value of an array just assign a value to the specific index we want to assign.

Eg:- sales\_array[1] = 25

We can also use logical conditions to replace elements in an array.

Eg: - sales\_array[ sales\_array == 0] = 5

**Where() function**

-The np.where() function performs a logical test on an array and returns a value if it is true, if it is false it will return another value.

np.where(logical test expression, value if true, value if false)

Eg:- inventory\_array = np.array([12,102,18,0,0])

product\_array = np.array([‘fruits’,’vegetables’,’cereal’,’dairy’,’eggs’])

the np.where will return a new array if we perform a logical test on inventory\_array like

np.where(inventory\_array <= 0,”Out of stock”,”In stock”)

We will get the new array as

array([‘In stock’,’In stock’,’In stock’,’Out of stock’,’Out of stock’])

Where in stock is for inventory\_array element’s whose value is greater than 0 and Out of stock is for inventory items whose values is less than or equal to 0

* We can also return items from array instead of values

eg:- np.where(inventory\_array <=0 , “Out of stock”, product\_array)

This will return an array with values of product\_array if the value of the item corresponding to the inventory\_array is not zero else it will provide “Out of stock” string

So we get the array as

array([‘fruits’,’vegetables’,’cereal’,’Out of stock’,’Out of stock’])

* It is also possible to chain where conditions inside the where condition. For example

np.where(my\_array % 2 == 0, ‘even’, np.where(my\_array == 9, my\_array,’odd’))

Here first it will check whether the number is even or not then if it is even it will return even else if it is odd again another condition is checked to see if the number is 9; if it is 9 it will return the element from my\_array else it will return odd.

**Array Aggregation Methods**

* array.sum() calculates the sum of all elements in the array

eg:- sales\_array.sum()

* array.mean() calculates the mean value(average) of the elements in the array

eg:- sales\_array.mean()

* array.max() returns the maximum value of the array

eg:- sales\_array.max()

* array.min() returns the minimum value of the array

eg:- sales\_array.min()

* array.argmax() returns the index of the highest value in an array
* Array.argmin() returns the index of the smallest value in an array.
* There are many more aggregate methods in numpy, you can refer it in the documentation.
* We can use aggregate functions across rows and columns.

sales\_array = np.array([0,5,155,0,518],

[0,1827,616,317,325])

we can use the sum aggregate method to calculate the sum of all columns(vertically)

By specifying axis = 0 as a parameter for array.sum() method.

eg:- sales\_array.sum(axis=0)

gives

array([0,1832,771,317,84])

Similarly if we need to find the sum row wise(horizontally) we can specify axis = 1 as parameter

Eg:- sales\_array.sum(axis=1)

Will give array([678,3085])

* we can use the aggregate methods while performing other arithmetic or logical operations on arrays.

eg:- (sales\_array \* prices).sum()

will give the sum of the product of elements of sales\_array and prices array

**Array Functions in Numpy**

* Array function will help you in performing other aggregations like calculating median and percentiles.
* np.median(array) will give the median value of the array passed as the argument.

The median is a good statistical tool when calculating a centre point for skew quantities like prices, where the mean value cannot pin point a central value if there are large values in the array.

* np.percentile(array, n ) is used to calculate the nth percentile of the given array.
* np.unique(array) returns the unique values of the array.
* np.sqrt(array) will give an array with square root of all the elements of the passed array.

**Sorting Arrays in Numpy**

* Pandas provide the most number of convenient sort functions. But we can also use the sort method of numpy also if required
* The sort method is going to return an array that is sorted not in place which means the values underlying the object will not be replaced and we will get a new array.
* Array.sort() sorts the arrays row by row that is it will have the default value of axis=1 if we don’t pass any parameter

sales\_array = np.array([0,5,155,0,518],

[0,1827,616,317,325])

sales\_array.sort() will give

array([ 0 , 0,5,155,518],

[0,317,325,616,1827])

* if we specify array.sort(axis = 0) it will sort column by column

eg:- sales\_array.sort(axis= 0)

will give

array([0,5,155,0,325],

[0,1827,616,317,518])

**Vectorisation**

- Vectorisation is the process of pushing the array operations to optimised c code, which is more efficient than writing loops.

* Python is written on C. C is much more memory efficient compared to python.
* The best practice is to use pythons(numpy’s) built-in functions as much as possible for faster operations, and avoid loops.

**Broadcasting**

* It is process that let’s us do vectorisation on arrays of different sizes. Here numpy array will expand to fit the larger one.
* Single values or scalars can be broadcasted to any dimensions.
* Eg:-

test\_array = np.array([[1,2,3],[1,2,3],[1,2,3]])

if we do test\_array + 1

we get array([[2,3,4],

[2,3,4],

[2,3,4]])

Behind the scenes numpy will create a 3x3 array will all elements 1 and then the addition operation on the test\_array and we get the result.

* Another possibility is to have array with more than one dimension with length more same
* Eg:-

if we do test\_array + np.array([3,2,1])

this will expand the 2nd array to 2 more rows to match the same dimension of the 1st array.

so [3,2,1]

[3,2,1]

[3,2,1]

Will be created and added so we get

[4,4,4]

[4,4,4]

[4,4,4]

* The same can be said true if we reshape the 2nd array to only single column and do the operation. The only difference will be the change in final result due to the operation being performed row wise. Here also numpy will expand the 2nd array to match the size of the first array.
* The broadcasting cannot be done on arrays of incompatible shapes. It will result in value error.
* In-order to avoid this ValueError of incompatible shapes at least one dimension of both the array should match.

**Pandas**

**Series**

* It is equivalent to a column in sql or excel, with additional features.
* They are numpy array with additional features built into them
* We can give series a custom index and an optional name compared to a numpy array.
* We can create a series from numpy arrays or from lists in python. But most commonly used method is importing from csv files or sql databases.
* Two or more series formed together to form a pandas data frame[data frame is similar to a table]
* The standard alias for pandas library is pd

to use pandas

import pandas as pd

we might also need to import numpy also

* Pandas ‘Series()’ function converts python lists and numpy array to a pandas series. [Note that that the S is capital]
* If we do not provide any name for the index, it will provide default index from 0 to the length of data
* Example

sales = [0, 5, 155, 0, 518, 0, 1827, 616, 317, 325]

sales\_series = pd.Series(sales, name = “sales”)

sales\_series

will print

0 0

1 5

2 155

3 0

4 518

5 0

6 0

7 1827

8 616

9 325

Name : sales , dtype: int64

The dtype and name is returned by default

**Note that the first parameter for the Series function is the list or array of the values, then the remaining parameter such as name etc should be supplied as keyword arguments**

**Also it is important to note that values fed to the Series() function must be 1 dimensional. Otherwise we will get an error.**

**Properties of series**

* Pandas series have the following properties
  + - Values : It is the data array in the series (since it holds the actual data values)

sales\_array.values will return a numpy array. If we want we can perform filtering and aggregation on these using numpy functions.

array([0, 5, 155, 0, 518, 0, 1827, 616, 317, 325])

* + - Index : It holds the index array of the series

sales\_series.index

will return RangeIndex(start = 0, stop = 10, step = 1).

If we want to modify the index we can use sales\_series.index = [10,20,30,40 etc] which will change the index to these values

* + - Name: Holds the name of series. Particularly useful when using data frames to fetch values from a series

sales\_series.name

will give ‘Sales’ which is the name we defined

we can change the name of the series by assigning a different name to the series.name

eg:- sales\_series.name = “New name”

we can assign name also like this if we haven’t defined a name while creating the series.

* + - Dtype : holds the datatype of the elements in the value array

sales\_array.dtype will return dtype(‘int64’)

because it has integer values only

**Data types in Pandas**

* Data types in pandas are mostly equivalent to that of base python and numpy
* There are 6 numeric data types in pandas
* But pandas 1.0 introduced new data types which made the performance improvements and optimisations in the use of null values or missing values.
  + - * Boolean- nullable boolean true / false - size 8 bits
      * Int64 - nullableWhole numbers - size (8,16,32,64)bits
      * Float64 - nullable Decimal numbers - Size(8,16,32,64)bits

Are the newly introduced types.

- (bool) - boolean true/ false - size 8 bits

- (int64) - whole numbers - size(8,16,32,64)bits

- (Float64) - Decimal numbers - size(8,16,32,64)bits

* Even though for boolean is True and False, the value behind the scenes is stored as 0 and 1
* If we calculate the mean of a boolean column it will tell us the average number of times the value of column is true.
* If we calculate the sum of a boolean column it will tell us the number of times the value is true.
* There are 3 object/ text data types in series
  + - * Object - any python object can be stored in this, which makes it flexible as well as hight in-efficient in space.
      * String - only contains string or text
      * Category - Maps categorical data to numeric array for efficient.
* There are 3 Time series data types in pandas which lets handling date and time very efficiently
  + Datetime64 - a single moment in time or a time stamp eg:-(Januray 4,2015,2:00:00PM)
  + Timedelta - duration between two dates or times (eg:- 10days, 3 seconds etc)
  + Period - a span of time ( a day, a week etc)
* We can convert series of one data type to a series of another datatype using .astype() method if they are compatible.
* Example :- sales\_series.astype(“float”)

will convert the sales\_series which have int64 datatype to float64

* Similarly we can convert the numeric types to boolean here the values which are greater than 0 will be true and the values with 0 will be false.
* Numbers can easily be converted to most other data types.

**Indexing**

* Indexes lets easy access to the rows of the pandas series
* Indexes let’s you access any element in a series, similar to numpy arrays.
* We can also slice the series using indices.

eg:- sales\_series[2:4]

2 155

3 0

it follows series\_name[start(inclusive) : stop(excluding)]

* Series also support custom indices.

there are two ways we can do this.

1) when creating the series:

here when we are creating series, we pass the keyword parameter index to the pd.Series() method. The value of the index will be a numpy array or a list whose length is same as the length of series

eg : items = [“coffee”, “banana”, “tea”, “coconut”, “sugar”]

sales\_series = pd.Series(sales\_array, index= items, name = “sales\_series”)

2) after creating the series:

we can change the index after series is created. We can define series\_name.index = index to change the default numeric index. The value assigning to this mush be a python list or numpy array with same length of the number of elements in the series.

* We can also access the elements with the custom index similar to that of numeric index.

ie, series\_name[index] will return the value of the particular index.

* There is major difference when slicing the series using custom indices. For numerical indices the upper limit is non inclusive and lower limit is inclusive, but for custom indices both the lower index and upper index is inclusive.

eg:- sales\_series[“banana”,”coconut”]

will return 3 records from the series.

* We can also define step sizes for slicing series.

**ILOC[ ] method**

* Is the commonly used method for accessing the values by their positional index.
* It works with series with custom index or numerical index
* It is more efficient than slicing and is recommended by pandas creators.
* The iloc[ ] method applies to both series and data frames.
* We can use dataframe\_name.iloc[row position, column position] to access the value.
* We can pass a single value to get the value of a single row. df.iloc[0] will give the first row
* We can also use multiple values separated by commas to get values from multiple rows using a list as index.

eg:- df.iloc[[5,9]] will return row 5 and 9

* We can slice the through the rows similar to slicing we used for numpy arrays.

ie :- df.iloc[0:11] gives a range of rows from 0th row to 11th row(here 11 is not included)

* Note that we are calling the .iloc[ ] on the series on which we want to access or filter the data
* Also note that iloc[ ] uses numerical index, even when the series have a custom index.

**.loc[ ] method**

* It is the preferred way to access values by their custom labels
* We can use df.loc[ row label, column label ] to access the value.

where df is the data frame. We can use it for series also

* Similar to .iloc[ ] method we can access values of
  + - Single row: by specifying label of the row

eg:- sales\_series.loc[“tea”] will return 155

* + - Multiple rows : by passing a list of labels to the method
    - Range of rows : by passing starting and ending labels like slicing.[limits are inclusive]

eg:- sales\_series.loc[“banana”,”coconut”]

will return

banana 5

tea 155

coconut 0

* Also note that .loc[ ] method works even for numeric indices. But if the numeric indices are not default indices ie 0 to n-1 the value will be returned based on passed numeric index label and not the actual position of the value.
* ??Compared to .iloc[ ] method which works only on numeric indices in such a way that if an index is missing when slicing it will result in an error. But in case of .loc[ ] method it will continue slicing from start to stop. —check again.
* **To drop an index we can use series\_name.index(drop = True) which will reset the series index to default numeric index.**