highradius

Pandas

Table of Contents

Introduction To Pandas

You can refer to the Jupyter Notebook attached here. Make sure you create a copy of this in your drive before executing the same.

What is Pandas?

Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the python programming language.

Pandas is quite a game changer when it comes to **analyzing data** with Python and it is one of the most preferred and widely used tools in **data munging/wrangling**. Pandas is an open source, free to use and it was originally written by Wes McKinney .

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 a table in a statistical software (like Excel).

Importing Pandas :
After the pandas have been installed into the system, you need to import the library. This module is generally imported as:

```
1 import pandas as pd
```

Here, pd is referred to as an alias to the Pandas. However, it is not necessary to import the library using the alias, it just helps in writing less code every time a method or property is called.

What are Pandas Data Frames?

Pandas DataFrame is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).
In general, we can say that the Pandas DataFrame consists of three main components: the data, the index, and the columns. DataFrames are extremely important going forward, as we can read & store excel sheets into DataFrames and use many manipulation techniques on them, as we'll learn ahead.

Advantages of Pandas

1. Fast and efficient for manipulating and analyzing data.
2. Data from different file objects can be loaded.

3. Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data
4. Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
5. Data set merging and joining.
6. Flexible reshaping and pivoting of data sets
7. Provides time-series functionality.
8. Powerful group by functionality for performing split-apply-combine operations on data sets.

Data Structures in Pandas

Introduction

Series :
Pandas Series is a one-dimensional labeled array capable of holding any data type. So, in terms of Pandas DataStructure, A Series represents a single column in memory, which is either independent or belongs to a Pandas DataFrame.

Why do we use series in pandas?
Pandas Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The object supports both integer and label-based indexing and provides a host of methods for performing operations involving the index.

Dataframe :
Pandas DataFrame is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes known as rows and columns. a dataframe is a collection of series that can be used to analyse the data. DataFrame can be created from the lists, dictionary, and from a list of dictionaries etc.

Basic Operations

Series -

Creating a Series -
Using Default Indexing.

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print s
```

Its **output** is as follows –

```
0   a
1   b
2   c
3   d
dtype: object
```

Using Custom Indexing,

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data,index=[100,101,102,103])
print s
```

Its **output** is as follows –

```
100  a
101  b
102  c
103  d
dtype: object
```

Creating a Series from Dictionary -

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print s
```

Its **output** is as follows –

```
a 0.0
b 1.0
c 2.0
dtype: float64
```

Accessing Data from Series -

Using Position.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first element
print s[0]
```

Its **output** is as follows –

```
1
```

Using Index labels.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve a single element
print s['a']
```

Its **output** is as follows –

```
1
```

Accessing First 3 Elements:-

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first three element
print s[:3]
```

Its **output** is as follows –

```
a  1
b  2
c  3
dtype: int64
```

Accessing Multiple Elements using Index Label.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve multiple elements
print s[['a','c','d']]
```

Its **output** is as follows –

```
a  1
c  3
d  4
dtype: int64
```

Dataframe :

Creating Dataframe using list -

```
# import pandas as pd
import pandas as pd

# list of strings
lis = ['USA', 'covid', 'cases', 'are', '2.56m', 'and', 'counting']

# Calling DataFrame constructor on list
df = pd.DataFrame(lis)
df
```

	0
0	USA
1	covid
2	cases

Creating dataframe from dict of ndarray/lists -

```
import pandas as pd

# intialise data of lists.
data = {'Name':['Tom', 'Nick', 'Anne', 'Jack'],
        'Age':[20, 21, 19, 18],
        'Gender' : ['M', 'M', 'F','M']}

# Create DataFrame
df = pd.DataFrame(data)

df
```

	Name	Age	Gender
0	Tom	20	M
1	Nick	21	M
2	Anne	19	F
3	Jack	18	M

Create a DataFrame from List of Dicts -

```
import pandas as pd
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
df
```

	a	b	c
0	1	2	NaN
1	5	10	20.0

```
#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])
df1
```

	a	b
first	1	2
second	5	10

Column selection -

```
#selecting single column
df.Name
```

```
0     Tom
1    Nick
2    Anne
3    Jack
Name: Name, dtype: object
```

```
#selecting list of columns
df[['Name', 'Age']]
```

	Name	Age
0	Tom	20
1	Nick	21
2	Anne	19
3	Jack	18

Other Dataframe Operations can be found here.

Introduction to Reading and Saving Dataframes :

Introduction

CSV

A comma-separated values (CSV) file is a plaintext file with a .csv extension that holds tabular data. This is one of the most popular file formats for storing large amounts of data. Each row of the CSV file represents a single table row. The values in the same row are by default separated with commas, but you could change the separator to a semicolon, tab, space, or some other character.

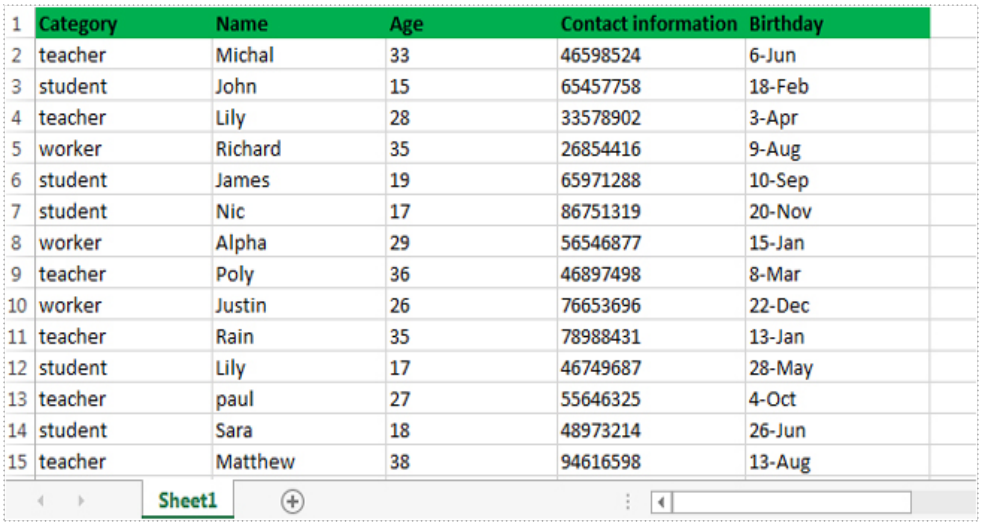
Eg:

```
time,signal1,signal2,time,signal3
0,1,1,0,4
1,2,4,2,8
2,3,9,3,15
3,3,9,5,16
3,4,16
4,5,25
```

Excel

Excel is a popular spreadsheet program used with data like numbers and formulas, text, and drawing shapes. Excel is part of the Microsoft Office Suite of software. XLS files use a Binary Interchange File Format to store spreadsheet data and are proprietary to Microsoft.

Eg:



	Category	Name	Age	Contact information	Birthday
2	teacher	Michal	33	46598524	6-Jun
3	student	John	15	65457758	18-Feb
4	teacher	Lily	28	33578902	3-Apr
5	worker	Richard	35	26854416	9-Aug
6	student	James	19	65971288	10-Sep
7	student	Nic	17	86751319	20-Nov
8	worker	Alpha	29	56546877	15-Jan
9	teacher	Poly	36	46897498	8-Mar
10	worker	Justin	26	76653696	22-Dec
11	teacher	Rain	35	78988431	13-Jan
12	student	Lily	17	46749687	28-May
13	teacher	paul	27	55646325	4-Oct
14	student	Sara	18	48973214	26-Jun
15	teacher	Matthew	38	94616598	13-Aug

Sheet1

Reading & Saving

CSV

Read a CSV File

Once your data is saved in a CSV file, you'll likely want to load and use it from time to time. You can do that with the Pandas read_csv() function, by passing the destination of the csv file in the function.

Eg:

```
df = pd.read_csv('data.csv', index_col=0)
```

In this case, the Pandas read_csv() function returns a new DataFrame with the data and labels from the file data.csv, which you specified with the first argument. This string can be any valid path, including URLs.

Write a CSV File

You can save your Pandas DataFrame as a CSV file with .to_csv():

Eg:

```
df.to_csv('data.csv')
```

Excel

Read an Excel File

You can load data from Excel files with read_excel():

Eg:

```
df = pd.read_excel('data.xlsx', index_col=0)
```

read_excel() returns a new DataFrame that contains the values from data.xlsx.

Write an Excel File

Once you have those packages installed, you can save your DataFrame in an Excel file with .to_excel()

Eg:

```
df.to_excel('data.xlsx')
```

The argument 'data.xlsx' represents the target file and, optionally, its path. The above statement should create the file data.xlsx in your current working directory.

Dataframe Operation:

Adding a row/column

Adding a row

You may want to add another row to a dataframe in many scenarios. Here's how you can do it

Eg:

	Region	Company	Product	Month	Sales
0	West	Costco	Dinner Set	September	2500
1	North	Walmart	Grocery	July	3096
2	South	Home Depot	Gardening tools	February	8795

A data dictionary with the values of one Region - East that we want to enter in the above dataframe (df).
The data is basically a list with a dictionary having columns as keys and their corresponding values.

```
data=[{'Region':'East','Company':'Shop  Rite','Product':'Fruits','Month':'December','Sales': 1265}]
```

First Way - Using append function

It basically creates a new dataframe object with the new data row at the end of the dataframe. The old dataframe will be unchanged.

Eg:

```
df.append(data,ignore_index=True,sort=False)
```

	Region	Company	Product	Month	Sales
0	West	Costco	Dinner Set	September	2500
1	North	Walmart	Grocery	July	3096
2	South	Home Depot	Gardening tools	February	8795
3	East	Shop Rite	Fruits	December	1265

Second Way - Using loc:

loc is used to access a group of rows and columns by labels or a boolean array. However with an assignment(=) operator you can also set the value of a cell or insert a new row all together at the bottom of the dataframe.

Eg:

```
>>> df.loc[3]=list(data[0].values())
Or
>>> df.loc[len(df.index)]=list(data[0].values())
>>> df
```

	Region	Company	Product	Month	Sales
0	West	Costco	Dinner Set	September	2500
1	North	Walmart	Grocery	July	3096
2	South	Home Depot	Gardening tools	February	8795
3	East	Shop Rite	Fruits	December	1265

Third Way - Using iloc

We can replace a row with the new data as well using iloc, which is integer-location based indexing for selection by position. In our original dataframe we want to replace the data for North Region with the new data of the East Region. Update the row at index position 1 using iloc and list values of the data dictionary i.e

list(data[0].values()) = ['East','Shop Rite','Fruits','December',1265]

Eg: Update row at Index position 1

```
>>>df.iloc[1]=list(data[0].values())
>>>df
```

	Region	Company	Product	Month	Sales
0	West	Costco	Dinner Set	September	2500
1	East	Shop Rite	Fruits	December	1265
2	South	Home Depot	Gardening tools	February	8795

Adding columns :

Pandas allows users to add a new column by initializing on the fly. For example: the list below is the purchase value of three different regions i.e. West, North and South. We want to add this new column to our existing dataframe above

Eg:

```
>>> purchase = [3000, 4000, 3500]
>>> df.assign(Purchase=purchase)
```

Or

```
>>> df["Purchase"] = purchase
```

	Region	Company	Product	Month	Sales	Purchase
0	West	Costco	Dinner Set	September	2500	3000
1	North	Walmart	Grocery	July	3096	4000
2	South	Home Depot	Gardening tools	February	8795	3500

Add Multiple Columns to a Dataframe

Lets add these three list (Date, City, Purchase) as column to the existing dataframe using assign with a dict of column names and values

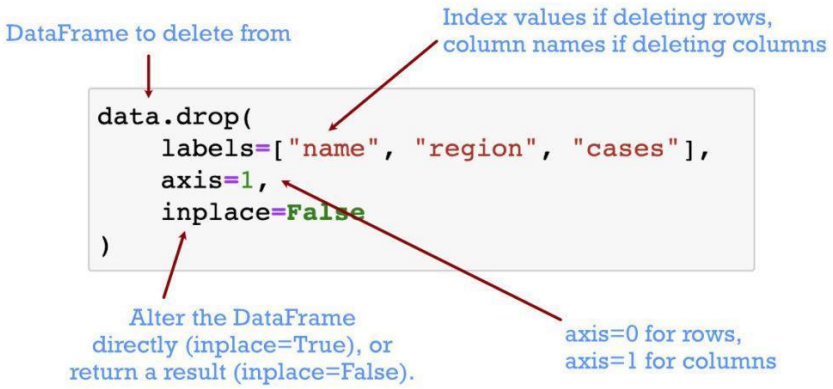
Eg:

```
>>> Date = ['1/9/2017','2/6/2018','7/12/2018']
>>> City = ['SFO', 'Chicago', 'Charlotte']
>>> Purchase = [3000, 4000, 3500]
>>> df.assign(**{'City' : City, 'Date' : Date,'Purchase':Purchase})
```

	Region	Company	Product	Month	Sales	City	Date	Purchase
0	West	Costco	Dinner Set	September	2500	SFO	1/9/2017	3000
1	North	Walmart	Grocery	July	3096	Chicago	2/6/2018	4000
2	South	Home Depot	Gardening tools	February	8795	Charlotte	7/12/2018	3500

Deleting a row/column :

The Pandas "drop" function is used to delete columns or rows from a Pandas DataFrame.



Eg:

Creating dataframe df, to perform operations.

```
>>> df = pd.DataFrame(np.arange(12).reshape(3, 4), columns=['A', 'B', 'C', 'D'])
>>> df
```

	A	B	C	D
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

Dropping columns:

```
>>>df.drop(['B', 'C'], axis=1)
```

output:

	A	D
0	0	3
1	4	7
2	8	11

>>> df.drop(columns=['B', 'C'])
output:

	A	D
0	0	3
1	4	7
2	8	11

Drop a row by index

>>> df.drop([0, 1])
output:

	A	B	C	D
2	8	9	10	11

Sorting a DataFrame

To sort the DataFrame based on the values in a single column, use .sort_values(). By default, this will return a new DataFrame sorted in ascending order. It does not modify the original DataFrame.

Syntax:
DataFrame.sort_values(*by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last', ignore_index=False, key=None)*

Axis: {0 or 'index', 1 or 'columns'}, default 0.
Ascending : bool or list of bool, default True.
Inplace : bool, default False . If True, perform operation in-place.

Sorting by a Column in Ascending Order
To use .sort_values(), pass a single argument to the method containing the name of the column you want to sort by.

In this example, you sort a DataFrame by the city08 column,

>>> df.sort_values("city08")

Output:

```
      city08  cylinders fuelType  ...  mpgData            trany  year
99         9          8   Premium  ...        N  Automatic 4-spd  1993
1          9         12   Regular  ...        N     Manual 5-spd  1985
80         9          8   Regular  ...        N  Automatic 3-spd  1985
47         9          8   Regular  ...        N  Automatic 3-spd  1985
3         10          8   Regular  ...        N  Automatic 3-spd  1985
..        ...        ...      ...  ...      ...              ...   ...
9         23          4   Regular  ...        Y  Automatic 4-spd  1993
8         23          4   Regular  ...        Y     Manual 5-spd  1993
7         23          4   Regular  ...        Y  Automatic 3-spd  1993
76        23          4   Regular  ...        Y     Manual 5-spd  1993
2         23          4   Regular  ...        Y     Manual 5-spd  1985
[100 rows x 10 columns]
```

This sorts your DataFrame using the column values from city08,
By default, .sort_values() sorts your data in ascending order.

Sorting by a Column in Descending Order

To sort the DataFrame in descending order, pass "ascending=False" as one of the parameter as shown in the example below,

Eg:

```
>>> df.sort_values( by="city08", ascending=False)
```

Output:

```
   city08  cylinders fuelType  ...  mpgData            trany  year
9      23          4  Regular  ...        Y  Automatic 4-spd  1993
2      23          4  Regular  ...        Y     Manual 5-spd  1985
7      23          4  Regular  ...        Y  Automatic 3-spd  1993
8      23          4  Regular  ...        Y     Manual 5-spd  1993
76     23          4  Regular  ...        Y     Manual 5-spd  1993
..    ...        ...      ...  ...      ...              ...   ...
58     10          8  Regular  ...        N  Automatic 3-spd  1985
80      9          8  Regular  ...        N  Automatic 3-spd  1985
1       9         12  Regular  ...        N     Manual 5-spd  1985
```

```
47        9           8   Regular  ...         N  Automatic 3-spd  1985
99        9           8   Premium  ...         N  Automatic 4-spd  1993
[100 rows x 10 columns]
```

Null Handling

There are numerous ways to detect the presence of missing values in a dataset.

isna()

The isna() function is used to check missing (null) values. It is a boolean function that looks for the missing values and returns TRUE where it detects a missing value.
Look at the below syntax:

dataframe.isna()

In this example below, we have made use of .isna() function to check for the presence of missing values. The cell of the dataframe containing missing values only returns TRUE, otherwise it returns FALSE.

Dataframe:

	Name	Age
0	Adam	10.0
1	Steve	15.0
2	John	NaN

Checking missing values using isna():

```
1  df.isna()
```

	Name	Age
0	False	False
1	False	False
2	False	True

We can also use isna() in pandas series.

Example 2: Using isna() function to detect missing values in a pandas series object.

```
# Creating the series
ser = pd.Series([12, 5, None, 5, None, 11])

# Print the series
ser

```

```
0    12.0
1     5.0
2     NaN
3     5.0
4     NaN
5    11.0
dtype: float64
```

```
# to detect the missing values
ser.isna()
```

```
0    False
1    False
2     True
3    False
4     True
5    False
dtype: bool
```

It returned True for all the missing values in the series.

notna()

Similarly We have notna(). With notna() function, we can easily pick out data that does not occupy missing values or NA values. The notna() function returns TRUE, if the data is free from missing values else it returns FALSE (if null values are encountered).

Dataframe:

	Name	Age
0	Adam	10.0
1	Steve	15.0
2	John	NaN

Checking missing values using notna():

```
1 df.notna()
```

	Name	Age
0	True	True
1	True	True
2	True	False

It returned False only for the Age of John (which was the only missing value in the dataframe).

isnull()

isnull() function also works the same as **isna()**, detects missing values in the given series or dataframe. It returns a boolean same-sized object indicating if the values are NA (null). Missing values get mapped to True and non-missing values get mapped to False.

Look at the below syntax:

Series.isnull() Or,
Dataframe.isnull()

We will take the same example for easy understanding,

Dataframe:

	Name	Age
0	Adam	10.0
1	Steve	15.0
2	John	NaN

Checking missing values using isnull():

```
1 df.isnull()
```

	Name	Age
0	False	False
1	False	False
2	False	True

Similar to notna(), there is also **notnull()** function in pandas.

```
df.notnull()
```

	Name	Age
0	True	True
1	True	True
2	True	False

fillna()

fillna() manages and lets the user replace NaN values with some value of their own.

Syntax:

dataframe["Column name"].fillna("Value", inplace = True)

Eg:

Dataframe:

	Name	Age
0	Adam	10.0
1	Steve	15.0
2	John	NaN

We will replace missing values in Age (Age of john) with 'Age missing'

```
# replacing missing values in Age with 'Age missing'
dataf["Age"].fillna("Age missing", inplace = True)
dataf
```

	Name	Age
0	Adam	10.0
1	Steve	15.0
2	John	Age missing

replace()

Pandas replace() is a very rich function that is used to replace a string, regex, dictionary, list, and series from the DataFrame. The values of the DataFrame can be replaced with other values dynamically. It is capable of working with the Python regex (regular expression). It differs from updating with .loc or .iloc, which requires you to specify a location where you want to update with some value.

Syntax:
DataFrame.replace(to_replace= " ", value=" ")

Let us take an example:

Dataframe:

	Name	Age
0	Adam	10.0
1	Steve	15.0
2	John	NaN

We are going to replace team "John" with "Michael" in the 'df' data frame.

```
df.replace(to_replace ="John",
                 value ="Michael")
```

	Name	Age
0	Adam	10.0
1	Steve	15.0
2	Michael	NaN

Example 2: Replacing more than one value at a time.
Let us take a new dataframe.

	Name	Subject
0	Tom	Maths
1	nick	Bio
2	krish	Phy
3	jack	Bio
4	steve	Maths
5	David	Bio
6	Adam	Phy

Using python list as an argument We are going to replace “Phy” and “Bio” with “Science” in the dataframe.

```
dff.replace(to_replace =["Bio", "Phy"],
                            value ="Science")
```

	Name	Subject
0	Tom	Maths
1	nick	Science
2	krish	Science
3	jack	Science
4	steve	Maths
5	David	Science
6	Adam	Science

Can we replace missing values using replace() function?
Yes, we can !!!

See how :

Example 3: Replace the Nan value in the data frame with a specific value

Let us try to replace missing values (age of john) with 12.0
Dataframe:

	Name	Age
0	Adam	10.0
1	Steve	15.0
2	John	NaN

We will use numpy.nan for NaN values.

```
import numpy as np
#replace missing values (age of john) with 12.0
df.replace(to_replace =np.nan,
                            value =12.0)
```

	Name	Age
0	Adam	10.0
1	Steve	15.0
2	John	12.0

Aggregation of Groups (Group By) :

Introduction

Pandas group by is used for grouping the data according to the categories and apply a function to the categories. It also helps to aggregate data efficiently. Pandas dataframe.groupby() function is used to split the data into groups based on some criteria.

In this play, you will learn how to do analysis on groups(individual categories) in a pandas dataframe of a categorical feature. The concept of split-apply-combine, manipulates groups separately based on different rules.

When you are working on a dataset, be it data preparation phase, EDA phase, you will often come upon a situation where you would like to get some statistical inference from the data within each individual category.

Syntax -
DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False)

Too much to process what I just said? Let's think with an example.
For example,
Let's say you have a dataset where you have records of your friends and the duration of talk you had with that friend each time you met him/her.

S.no	Friend	Duration of Meet
1.	A	10 mins
2.	A	13 mins
3.	B	6 mins
4.	B	2 mins
5.	B	3 mins
6.	C	40 mins
7.	D	1 min

In real life let's assume you had about 1000s to 10,000s of such records, now someday you want to just see whom you are more in touch with, whom you talked the most etc. etc.

The result we want to generate would be something like this,

Friend	Average talk time	Frequency of meet	Longest Talk	Shortest talk
A	11.5 mins	2	13 mins	10 mins
B	3.67 mins	3	6 mins	2 mins
C	40 mins	1	40 mins	40 mins
D	1 min	1	1 mins	1 mins

So as you can see we just checked some statistical analysis on group level (your each friend).

Aggregate Functions

An aggregated function returns a single aggregated value for each group. Once the group by object is created, several aggregation operations can be performed on the grouped data.

Sum ()

groupby() function takes up the column name as argument followed by sum() function.

	Name	Subject	Rating	Num
8	Name_8	Subject_0	4.663962	7
5	Name_5	Subject_3	1.506786	4
9	Name_9	Subject_4	7.946482	8
0	Name_0	Subject_5	2.413453	6
3	Name_3	Subject_5	8.347007	4
2	Name_2	Subject_0	1.532989	7
7	Name_7	Subject_1	2.389926	8
4	Name_4	Subject_2	5.854676	9
1	Name_1	Subject_2	4.413439	8
6	Name_6	Subject_3	5.256156	6

Suppose HouseOne is the name of the dataframe. We are grouping the data by Subject and also calculating the sum of the num column for each subject.

```
In [31]:  1 HouseOne.groupby("Subject")['Num'].sum()

Out[31]: Subject
         Subject_0    14
         Subject_1     8
         Subject_2    17
         Subject_3    10
         Subject_4     8
         Subject_5    10
         Name: Num, dtype: int32
```

Count ()

groupby() function takes up the column name as argument followed by count() function.

For the same dataframe, we are grouping the data by Num and also calculating the count of the same column. It will five us the frequency of each number in the num column.

```
In [38]:  1 HouseOne.groupby("Num")['Num'].count()

Out[38]: Num
         4    2
         6    2
         7    2
         8    3
         9    1
         Name: Num, dtype: int64
```

Mean, Median, Mode

groupby() function takes up the column name as an argument followed by the name of the function or by using agg() function.

For the same dataframe,we are grouping the data by Subject and also calculating the mean of the Rating column for each category in the subject.

```
In [39]:  1 HouseOne.groupby("Subject")['Rating'].mean()

Out[39]: Subject
         Subject_0    3.098475
         Subject_1    2.389926
         Subject_2    5.134058
         Subject_3    3.381471
         Subject_4    7.946482
         Subject_5    5.380230
         Name: Rating, dtype: float64

In [41]:  1 HouseOne.groupby("Subject")['Rating'].agg(np.mean)

Out[41]: Subject
         Subject_0    3.098475
         Subject_1    2.389926
         Subject_2    5.134058
         Subject_3    3.381471
         Subject_4    7.946482
         Subject_5    5.380230
         Name: Rating, dtype: float64
```

Lambda Functions

What are Lambda functions?

In Python, an anonymous function is a function that is defined without a name.
While defining normal functions, we are using the def keyword in Python, but while defining anonymous functions we are using the lambda keyword.
Hence, anonymous functions also are called Lambda functions.

How to use lambda Functions in Python?

A lambda function in python has the following syntax.

lambda argument(s): expression

A lambda function evaluates an expression for a given argument. We give the function a value (argument) and then provide the operation (expression). The keyword lambda must come first. A full colon (:) separates the argument and the expression.

Eg :

Input :

```
1  # Program to show the use of lambda functions
2  double = lambda x: x * 2
3
4  print(double(5))
```

Output :

```
10
```

In the above example , **lambda x: x * 2** is the lambda function. Here **x** is the argument and **x * 2** is the expression that gets evaluated and returned.

Implementation of Lambda Function in Python

We use lambda functions when we require a nameless function for a short period of time.

While working with DataFrames, lambda functions can help us in applying any operations on individual values of a series, instead of applying the operation on the series as a whole.

For example -

Let's say we have a dataframe where a column called Number contains some alpha numeric values, and we want to fetch the first three characters of these values. We can do this easily using lambda functions, which would otherwise be difficult to do.

We already know we can fetch the first 3 characters of a string value like below -

```
1  string_value = "Hello"
2  string_value[:3]
```

```
'Hel'
```

We can apply this same operation on each value of the series/column Number without using loops as shown below,

```
1  df
```

	Number
0	ABC123
1	AZZ0011
2	XYZ555

```
1  df["Number_first_3_characters"] = df["Number"].apply(lambda x : x[:3])
2  df
```

	Number	Number_first_3_characters
0	ABC123	ABC
1	AZZ0011	AZZ
2	XYZ555	XYZ

Similarly, you can use lambda functions to apply any operations on individual values of a Series.

Additional References :

Reference-1

Reference-2

Joining DataFrames

Introduction

There are many times when you'd need to merge/join two dataframes either horizontally or vertically.

Vertical Join (Concat Function)

Data Frame One

Var1	Var2	Var3
1		
2		
3		
4		

Data Frame Two

Var1	Var2	Var3
5		
6		
7		
8		

Var1	Var2	Var3
1		
2		
3		
4		
5		
6		
7		
8		

Horizontal Join (Merge Function)

Table A

ID	Description
1	Dog
2	Cat
3	Cow

Table B

ID	Description
1	Dog
2	Cat
4	Horse

Join Result

ID	Description	Usage
1	Dog	Table A+B
2	Cat	Table A+B
3	Cow	Table A
4	Horse	Table B

Join

Join is not only one of the most important concepts to master for data manipulation but also one of the easiest.

Let us start by understanding different types of joins,

1) Left Join
2) Right Join
3) Inner Join
4) Outer Join

Left Join, returns all records from the left table and the matched results from the right table.

In Simpler Words with an example: You have a table consisting of student names and Respective marks. In another table, you have other details of the students which can be helpful for you. The easiest way to lookup for these details is by performing a simple Left Join.

	name	marks
0	Walter	70
1	White	75
2	Saul	80
3	Goodman	90

Table A(Left) : Names and their Corresponding marks.

	name	age	Hobby
0	Walter	21	Cooking
1	Saul	22	Reading
2	Goodman	20	Playing

Table B(Right) : Names and the other details associated with the Students.

```
Left_Join = Marks.merge(Age, on = 'name', how = 'left')
Left_Join
```

	name	marks	age	Hobby
0	Walter	70	21.0	Cooking
1	White	75	NaN	NaN
2	Saul	80	22.0	Reading
3	Goodman	90	20.0	Playing

Performed a simple Left Join to bring all the details in the same table. This is a very useful operation when you need to look up the corresponding details associated with an id that exists in a different table. Like in this case, id is 'name' and 'age' and 'hobby' associated with it exists in a different table.

Right Join, Very Similar to the Left Join, but it returns all the records from the right table and the matched records from the left table.

```
Right_Join = Marks.merge(Age, on = 'name', how = 'right')
Right_Join
```

	name	marks	age	Hobby
0	Walter	70	21	Cooking
1	Saul	80	22	Reading
2	Goodman	90	20	Playing

As we can see, by using right join, we can populate the right table (Age) with marks from left table (Marks). Both Left Join and Right Join are very similar in nature.

On the Contrary, Inner Join is slightly different from both of the other joins. It returns a table which have ids (name in our case), which exists in both the tables, in technical terms, it returns the intersection of the both the table based on the id (name).

	name	marks
0	Walter	70
1	White	75
2	Saul	80
3	Goodman	90

Table A (Marks) : Has an extra id of 'white'.

	name	age	Hobby
0	Walter	21	Cooking
1	Saul	22	Reading
2	Goodman	20	Playing
3	Hank	24	Collecting Minerals

Table B (Age) : Has an extra id of 'Hank'.

Note : By extra it means, not common in both the tables.

```
Inner_Join = Marks.merge(Age, on = 'name', how = 'inner')
Inner_Join
```

	name	marks	age	Hobby
0	Walter	70	21	Cooking
1	Saul	80	22	Reading
2	Goodman	90	20	Playing

In Inner Join, we can clearly see, it returns a table that consists only of the ids (name) that is common in both the tables. (i.e Walter, Saul and Goodman.)

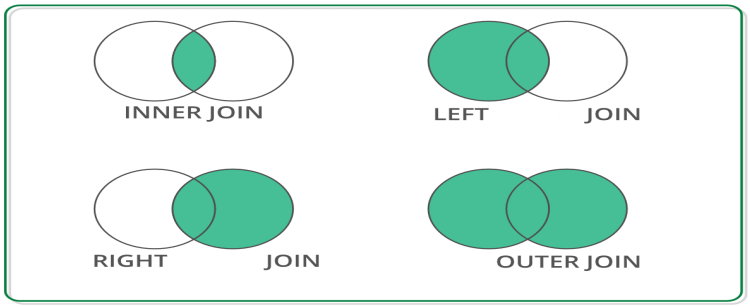
The Final one being Outer Join, it is just the opposite of Inner Join. In technical terms, it returns all the records from both tables (right and left) based on the id that we set. It is the Union of both the tables.

```
Outer_Join = Marks.merge(Age, on = 'name', how = 'outer')
Outer_Join
```

	name	marks	age	Hobby
0	Walter	70.0	21.0	Cooking
1	White	75.0	NaN	NaN
2	Saul	80.0	22.0	Reading
3	Goodman	90.0	20.0	Playing
4	Hank	NaN	24.0	Collecting Minerals

As Evident, it is returning all the records present in both the tables. 'White' is only present in the Marks tables, whereas 'Hank' is only present in the Age table, still it is returning those records as well.

Note : NaN is Null, since the marks for 'Hank' are not present. Similarly, the age and hobby of 'White' is not known.



In this image, Venn Diagrams of all the joins are present for effective visualization of different types of joins.

Concat

The **concat** function does all of the heavy lifting of performing concatenation operations along an axis.

You can vertically/horizontally concat any number of tables by using this function from pandas library.

```
Marks1 = pd.DataFrame({'name' : ['Walter','White','Saul','Goodman'],
    'marks' : [70,75,80,90]})
```

```
Marks2 = pd.DataFrame({'name' : ['Hank','Pinkman','Mike','Fring'],
    'marks' : [88,72,75,90]})
```

```
pd.concat([Marks1,Marks2], axis=0)
```

	name	marks
0	Walter	70
1	White	75
2	Saul	80
3	Goodman	90
0	Hank	88
1	Pinkman	72
2	Mike	75
3	Fring	90

The Above image is an example of vertical (axis = 0) concatenation of two dataframes. Similarly you can do horizontal concatenation of two tables.

Introduction to Some Basic Functions :

UNIQUE function

unique() function is used to return unique values from a 1D array. This method works only on a single column and not on whole Data Frames. This method includes NULL value as a unique value.

Syntax: Series.unique()
Dataframe.column.unique()

Return Type: Numpy array of unique values in that column

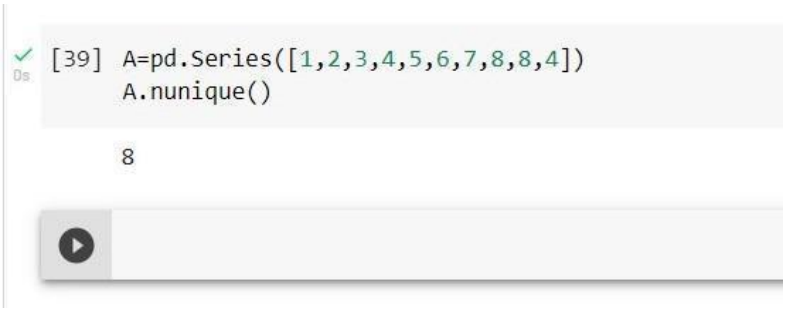
```
[38] A=pd.Series([1,2,3,4,5,6,7,8,8,4])
     A.unique()

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

NUNIQUE function

nunique() function is used to find the number of unique values in a series. This method works only on a single column and not on whole Data Frames. It counts null value as a unique value.

Syntax: Series.nunique(axis=0, dropna=True)
DataFrame.column.nunique(dropna=True)

Parameters :
axis : {0 or 'index', 1 or 'columns'}, default 0.
dropna : Don't include NaN in the counts.

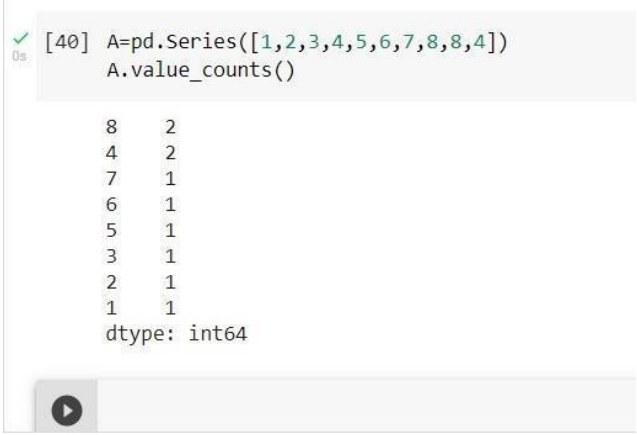


VALUE_COUNTS function

value_counts() function returns a Series containing counts of unique values. The result will be in descending order ,i.e. the first element is the most frequently-occurring element.

Syntax: Series.value_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)

Parameter:
normalize : If True then the object returned will contain the relative frequencies of the unique values.
sort : Sort by values.
ascending : Sort in ascending order.
dropna : Don't include counts of NaN.



DESCRIBE function

The describe() method is used for calculating some statistical data like percentile, mean and std of the numerical values of the Series or DataFrame. It analyzes both numeric and object series and also the DataFrame column sets of mixed data types.

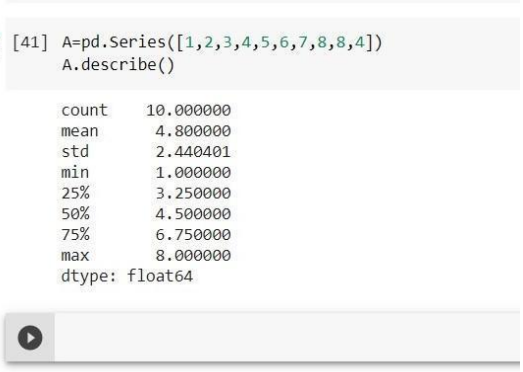
Syntax: Series.describe(percentiles=None, include=None, exclude=None)

Parameters:

percentile: list like data type of numbers between 0-1 to return the respective percentile

include: List of data types to be included while describing dataframe. Default is None

exclude: List of data types to be Excluded while describing dataframe. Default is None



```
[41] A=pd.Series([1,2,3,4,5,6,7,8,8,4])
     A.describe()
```

```
count    10.000000
mean      4.800000
std       2.440401
min       1.000000
25%       3.250000
50%       4.500000
75%       6.750000
max       8.000000
dtype: float64
```

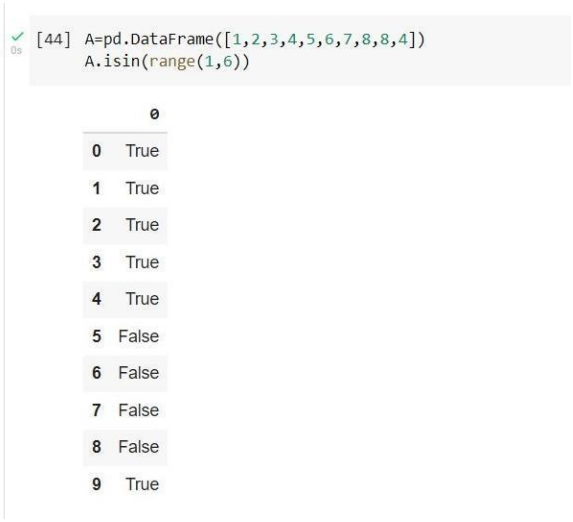
ISIN function

The isin() function is used to check whether each element in the DataFrame or Series is contained in values or not.

Syntax: DataFrame.isin(values)

Parameters:

values: iterable, Series, List, Tuple, DataFrame or dictionary to check in the caller Series/Data Frame.



```
[44] A=pd.DataFrame([1,2,3,4,5,6,7,8,8,4])
     A.isin(range(1,6))
```

	0
0	True
1	True
2	True
3	True
4	True
5	False
6	False
7	False
8	False
9	True

Introduction to Enumerate and Zip :

Enumerate

Often, when dealing with iterators, we also get a need to keep a count of iterations.Python eases the programmers' task by providing a built-in function enumerate() for this task. Enumerate() method adds a counter to an iterable and returns it in a form of enumerate object.

Enumerate() method adds a counter to an iterable and returns it in the form of an enumerating object. This enumerated object can then be used directly for loops or converted into a list of tuples using the list() method.

Syntax : enumerate(iterable, start=0)

Parameters :

- Iterable: any object that supports iteration.
- Start: the index value from which the counter is to be started, by default it is 0.

Example : -

In [ ]:

```
# Using enumerate object in lists

l1 = ["eat","sleep","repeat"]
s1 = "geek"

# creating enumerate objects
obj1 = enumerate(l1)
obj2 = enumerate(s1)

print ("Return type:",type(obj1))
print (list(enumerate(l1)))

# changing start index to 2 from 0
print(list(enumerate(s1,2)))
```

```
Return type: <class 'enumerate'>
[(0, 'eat'), (1, 'sleep'), (2, 'repeat')]
[(2, 'g'), (3, 'e'), (4, 'e'), (5, 'k')]
```

Zip

If you are a regular computer user, you should have used the .zip file extension. Do you know what it is? Basically, .zip is a container itself. It holds the real file inside.

Similarly,the purpose of the Python zip() method is to map the similar index of multiple containers so that they can be used just as a single entity. Python zip() takes iterable elements as input, and returns an iterator. If it gets no iterable elements, it returns an empty iterator.

Syntax : zip(*iterators)

Parameters :

Python iterables or containers (list, string etc)

Return Value :

Returns a single iterator object, having mapped values from all the containers.

Example :-

In [1]:

```
list1 = ['Alpha', 'Beta', 'Gamma', 'Sigma']
list2 = ['one', 'two', 'three', 'six']

test = zip(list1, list2)  # zip the values

print('\nPrinting the values of zip')
for values in test:
    print(values)  # print each tuples
```

```
Printing the values of zip
('Alpha', 'one')
('Beta', 'two')
('Gamma', 'three')
('Sigma', 'six')
```