**Python Pandas Introduction**

Pandas is defined as an open-source library that provides high-performance data manipulation in Python. The name of Pandas is derived from the word Panel Data, which means an Econometrics from Multidimensional data. It is used for data analysis in Python and developed by Wes McKinney in 2008.

Data analysis requires lots of processing, such as restructuring, cleaning or merging, etc. There are different tools are available for fast data processing, such as Numpy, Scipy, Cython, and Panda. But we prefer Pandas because working with Pandas is fast, simple and more expressive than other tools.

**Pandas is built on top of the Numpy package, means Numpy is required for operating the Pandas.**

Before Pandas, Python was capable for data preparation, but it only provided limited support for data analysis. So, Pandas came into the picture and enhanced the capabilities of data analysis. It can perform five significant steps required for processing and analysis of data irrespective of the origin of the data, i.e., load, manipulate, prepare, model, and analyze.

**Key Features of Pandas**

* It has a fast and efficient DataFrame object with the default and customized indexing.
* Used for reshaping and pivoting of the data sets.
* Group by data for aggregations and transformations.
* It is used for data alignment and integration of the missing data.
* Provide the functionality of Time Series.
* Process a variety of data sets in different formats like matrix data, tabular heterogeneous, time series.
* Handle multiple operations of the data sets such as subsetting, slicing, filtering, groupBy, re-ordering, and re-shaping.
* It integrates with the other libraries such as SciPy, and scikit-learn.
* Provides fast performance, and If you want to speed it, even more, you can use the Cython.

**Benefits of Pandas**

The benefits of pandas over using other language are as follows:

* Data Representation: It represents the data in a form that is suited for data analysis through its DataFrame and Series.
* Clear code: The clear API of the Pandas allows you to focus on the core part of the code. So, it provides clear and concise code for the user.

**Read data from excel to Series**:

import pandas as pd

import numpy as np

df=pd.read\_excel(r"D:\excelfiles\test.xlsx")

s=df[‘colname’]

print(s)

**Python Pandas Data Structure**

The Pandas provides two data structures for processing the data, i.e., Series and DataFrame, which are discussed below:

1) **Series**

It is defined as a one-dimensional array that is capable of storing various data types. The row labels of series are called the index. We can easily convert the list, tuple, and dictionary into series using "series' method. A Series cannot contain multiple columns. It has one parameter:

**Data**: It can be any list, dictionary, or scalar value.

**Creating Series from Array**:

Before creating a Series, Firstly, we have to import the numpy module and then use array() function in the program.

import pandas as pd

import numpy as np

info = np.array(['P','a','n','d','a','s'])

a = pd.Series(info)

print(a)

Output

0 P

1 a

2 n

3 d

4 a

5 s

dtype: object

**Explanation**: In this code, firstly, we have imported the pandas and numpy library with the pd and np alias. Then, we have taken a variable named "info" that consist of an array of some values. We have called the info variable through a Series method and defined it in an "a" variable. The Series has printed by calling the print(a) method.

**Python Pandas DataFrame**

It is a widely used data structure of pandas and works with a two-dimensional array with labeled axes (rows and columns). DataFrame is defined as a standard way to store data and has two different indexes, i.e., row index and column index. It consists of the following properties:

* The columns can be heterogeneous types like int, bool, and so on.
* It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.

**Create a DataFrame using List:**

We can easily create a DataFrame in Pandas using list.

import pandas as pd

# a list of strings

x = ['Python', 'Pandas']

# Calling DataFrame constructor on list

df = pd.DataFrame(x)

print(df)

Output

0

0 Python

1 Pandas

**Explanation**: In this code, we have defined a variable named "x" that consist of string values. The DataFrame constructor is being called on a list to print the values.

**Python Pandas Series**

The Pandas Series can be defined as a one-dimensional array that is capable of storing various data types. We can easily convert the list, tuple, and dictionary into series using "series' method. The row labels of series are called the index. A Series cannot contain multiple columns. It has the following parameter:

* data: It can be any list, dictionary, or scalar value.
* index: The value of the index should be unique and hashable. It must be of the same length as data. If we do not pass any index, default np.arrange(n) will be used.
* dtype: It refers to the data type of series.
* copy: It is used for copying the data.

**Creating a Series:**

We can create a Series in two ways:

* Create an empty Series
* Create a Series using inputs.

**Create an Empty Series**:

We can easily create an empty series in Pandas which means it will not have any value.

The syntax that is used for creating an Empty Series:

<series object> = pandas.Series()

The below example creates an Empty Series type object that has no values and having default datatype, i.e., float64.

Example

import pandas as pd

x = pd.Series()

print (x)

Output

Series([], dtype: float64)

**Creating a Series using inputs**:

We can create Series by using various inputs:

* Array
* Dict
* Scalar value

**Creating Series from Array**:

Before creating a Series, firstly, we have to import the numpy module and then use array() function in the program. If the data is ndarray, then the passed index must be of the same length.

If we do not pass an index, then by default index of range(n) is being passed where n defines the length of an array, i.e., [0,1,2,....range(len(array))-1].

Example

import pandas as pd

import numpy as np

info = np.array(['P','a','n','d','a','s'])

a = pd.Series(info)

print(a)

Output

0 P

1 a

2 n

3 d

4 a

5 s

dtype: object

**Create a Series from dict**

We can also create a Series from dict. If the dictionary object is being passed as an input and the index is not specified, then the dictionary keys are taken in a sorted order to construct the index.

If index is passed, then values correspond to a particular label in the index will be extracted from the dictionary.

#import the pandas library

import pandas as pd

import numpy as np

info = {'x' : 0., 'y' : 1., 'z' : 2.}

a = pd.Series(info)

print (a)

Output

x 0.0

y 1.0

z 2.0

dtype: float64

**Create a Series using Scalar**:

If we take the scalar values, then the index must be provided. The scalar value will be repeated for matching the length of the index.

#import pandas library

import pandas as pd

import numpy as np

x = pd.Series(4, index=[0, 1, 2, 3])

print (x)

Output

0 4

1 4

2 4

3 4

dtype: int64

**Accessing data from series with Position**:

Once you create the Series type object, you can access its indexes, data, and even individual elements.

The data in the Series can be accessed similar to that in the ndarray.

import pandas as pd

x = pd.Series([1,2,3],index = ['a','b','c'])

#retrieve the first element

print (x[0])

Output

1

**Series object attributes**

The Series attribute is defined as any information related to the Series object such as size, datatype. etc. Below are some of the attributes that you can use to get the information about the Series object:

|  |  |
| --- | --- |
| Attributes | Description |
| Series.index | Defines the index of the Series. |
| Series.shape | It returns a tuple of shape of the data. |
| Series.dtype | It returns the data type of the data. |
| Series.size | It returns the size of the data. |
| Series.empty | It returns True if Series object is empty, otherwise returns false. |
| Series.hasnans | It returns True if there are any NaN values, otherwise returns false. |
| Series.nbytes | It returns the number of bytes in the data. |
| Series.ndim | It returns the number of dimensions in the data. |
| Series.itemsize | It returns the size of the datatype of item. |

**Retrieving Index array and data array of a series object**

We can retrieve the index array and data array of an existing Series object by using the attributes index and values.

import numpy as np

import pandas as pd

x=pd.Series(data=[2,4,6,8])

y=pd.Series(data=[11.2,18.6,22.5], index=['a','b','c'])

print(x.index)

print(x.values)

print(y.index)

print(y.values)

Output

RangeIndex(start=0, stop=4, step=1)

[2 4 6 8]

Index(['a', 'b', 'c'], dtype='object')

[11.2 18.6 22.5]

**Retrieving Types (dtype) and Size of Type (itemsize)**

You can use attribute dtype with Series object as <objectname> dtype for retrieving the data type of an individual element of a series object, you can use the itemsize attribute to show the number of bytes allocated to each data item.

import numpy as np

import pandas as pd

a=pd.Series(data=[1,2,3,4])

b=pd.Series(data=[4.9,8.2,5.6],

index=['x','y','z'])

print(a.dtype)

print(a.itemsize)

print(b.dtype)

print(b.itemsize)

Output

int64

8

float64

8

**Retrieving Shape**

The shape of the Series object defines total number of elements including missing or empty values(NaN).

import numpy as np

import pandas as pd

a=pd.Series(data=[1,2,3,4])

b=pd.Series(data=[4.9,8.2,5.6],index=['x','y','z'])

print(a.shape)

print(b.shape)

Output

(4,)

(3,)

**Retrieving Dimension, Size and Number of bytes:**

import numpy as np

import pandas as pd

a=pd.Series(data=[1,2,3,4])

b=pd.Series(data=[4.9,8.2,5.6],

index=['x','y','z'])

print(a.ndim, b.ndim)

print(a.size, b.size)

print(a.nbytes, b.nbytes)

Output

1 1

4 3

32 24

**Checking Emptiness and Presence of NaNs**

To check the Series object is empty, you can use the empty attribute. Similarly, to check if a series object contains some NaN values or not, you can use the hasans attribute.

Example

import numpy as np

import pandas as pd

a=pd.Series(data=[1,2,3,np.NaN])

b=pd.Series(data=[4.9,8.2,5.6],index=['x','y','z'])

c=pd.Series()

print(a.empty,b.empty,c.empty)

print(a.hasnans,b.hasnans,c.hasnans)

print(len(a),len(b))

print(a.count( ),b.count( ))

Output

False False True

True False False

4 3

3 3

**Series Functions**

There are some functions used in Series which are as follows:

|  |  |
| --- | --- |
| Functions | Description |
| Pandas Series.map() | Map the values from two series that have a common column. |
| Pandas Series.std() | Calculate the standard deviation of the given set of numbers, DataFrame, column, and rows. |
| Pandas Series.to\_frame() | Convert the series object to the dataframe. |
| Pandas Series.value\_counts() | Returns a Series that contain counts of unique values. |

**Pandas Series.map()**

The main task of map() is used to map the values from two series that have a common column. To map the two Series, the last column of the first Series should be the same as the index column of the second series, and the values should be unique.

Syntax

Series.map(arg, na\_action=None)

Parameters

* arg: function, dict, or Series.

It refers to the mapping correspondence.

* na\_action: {None, 'ignore'}, Default value None. If ignore, it returns null values, without passing it to the mapping correspondence.

Returns

It returns the Pandas Series with the same index as a caller.

Example

import pandas as pd

import numpy as np

a = pd.Series(['Java', 'C', 'C++', np.nan])

a.map({'Java': 'Core'})

Output

0 Core

1 NaN

2 NaN

3 NaN

dtype: object

Example2

import pandas as pd

import numpy as np

a = pd.Series(['Java', 'C', 'C++', np.nan])

a.map({'Java': 'Core'})

a.map('I like {}'.format, na\_action='ignore')

Output

0 I like Java

1 I like C

2 I like C++

3 I like nan

dtype: object

Example3

import pandas as pd

import numpy as np

a = pd.Series(['Java', 'C', 'C++', np.nan])

a.map({'Java': 'Core'})

a.map('I like {}'.format)

a.map('I like {}'.format, na\_action='ignore')

Output

0 I like Java

1 I like C

2 I like C++

3 NaN

dtype: object

**Pandas Series.std()**

The Pandas std() is defined as a function for calculating the standard deviation of the given set of numbers, DataFrame, column, and rows. In respect to calculate the standard deviation, we need to import the package named "statistics" for the calculation of median.

The standard deviation is normalized by N-1 by default and can be changed using the ddof argument.

Syntax:

Series.std(axis=None, skipna=None, level=None, ddof=1, numeric\_only=None, \*\*kwargs)

Parameters:

* axis: {index (0), columns (1)}
* skipna: It excludes all the NA/null values. If NA is present in an entire row/column, the result will be NA.
* level: It counts along with a particular level, and collapsing into a scalar if the axis is a MultiIndex (hierarchical).
* ddof: Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.
* numeric\_only: boolean, default value None

It includes only float, int, boolean columns. If it is None, it will attempt to use everything, so use only numeric data.

It is not implemented for a Series.

Returns:

It returns Series or DataFrame if the level is specified.

Example1:

import pandas as pd

# calculate standard deviation

import numpy as np

print(np.std([4,7,2,1,6,3]))

print(np.std([6,9,15,2,-17,15,4]))

Output

2.1147629234082532

10.077252622027656

Example2:

import pandas as pd

import numpy as np

#Create a DataFrame

info = {

'Name':['Parker','Smith','John','William'],

'sub1\_Marks':[52,38,42,37],

'sub2\_Marks':[41,35,29,36]}

data = pd.DataFrame(info)

data

# standard deviation of the dataframe

data.std()

Output

sub1\_Marks 6.849574

sub2\_Marks 4.924429

dtype: float64

**Pandas Series.to\_frame()**

Series is defined as a type of list that can hold an integer, string, double values, etc. It returns an object in the form of a list that has an index starting from 0 to n where n represents the length of values in Series.

The main difference between Series and Data Frame is that Series can only contain a single list with a particular index, whereas the DataFrame is a combination of more than one series that can analyze the data.

The Pandas Series.to\_frame() function is used to convert the series object to the DataFrame.

Syntax

Series.to\_frame(name=None)

Parameters

name: Refers to the object. Its Default value is None. If it has one value, the passed name will be substituted for the series name.

Returns

It returns DataFrame representation of Series.

Example1

s = pd.Series(["a", "b", "c"],

name="vals")

s.to\_frame()

Output

vals

0 a

1 b

2 c

Example2

import pandas as pd

import matplotlib.pyplot as plt

emp = ['Parker', 'John', 'Smith', 'William']

id = [102, 107, 109, 114]

emp\_series = pd.Series(emp)

id\_series = pd.Series(id)

frame = { 'Emp': emp\_series, 'ID': id\_series }

result = pd.DataFrame(frame)

print(result)

Output

Emp ID

0 Parker 102

1 John 107

2 Smith 109

3 William 114

**Pandas Series.unique()**

While working with the DataFrame in Pandas, you need to find the unique elements present in the column. For doing this, we have to use the unique() method to extract the unique values from the columns. The Pandas library in Python can easily help us to find unique data.

The unique values present in the columns are returned in order of its occurrence. This does not sort the order of its appearance. In addition, this method is based on the hash-table.

It is significantly faster than numpy.unique() method and also includes null values.

Syntax:

pandas.unique(values)

Parameters:

values: It refers to a 1d array-like object that consists of an array value.

Returns:

This method returns a numpy.ndarray or ExtensionArray object and can be:

* index: Returns when user passes index as an input.
* Categorical: Returns when user passes a Categorical dtype as an input.
* ndarray: Returns when user passes a ndarray/Series as an input.

Example 1:

import pandas as pd

pd.unique(pd.Series([2, 1, 3, 3]))

pd.unique(pd.Series([pd.Timestamp('20160101'),

pd.Timestamp('20160101')]))

Output:

array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')

Example 2: The below example extracts the unique timestamp from the Index:

import pandas as pd

import numpy as np

pd.unique(pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),

pd.Timestamp('20160101', tz='US/Eastern')]))

Output:

DatetimeIndex(['2016-01-01 00:00:00-05:00'], dtype='datetime64[ns, US/Eastern]', freq=None)

**Pandas Series.value\_counts()**

The value\_counts() function returns a Series that contain counts of unique values. It returns an object that will be in descending order so that its first element will be the most frequently-occurred element.

By default, it excludes NA values.

Syntax

Series.value\_counts(normalize=False, sort=True, ascending=False, bins=None, dropna=True)

Parameters

* normalize: If it is true, then the returned object will contain the relative frequencies of the unique values.
* sort: It sort by the values.
* ascending: It sort in the ascending order.
* bins: Rather than counting the values, it groups them into the half-open bins that provide convenience for the pd.cut, which only works with numeric data.
* dropna: It does not include counts of NaN.

Returns

It returns the counted series.

Example1

import pandas as pd

import numpy as np

index = pd.Index([2, 1, 1, np.nan, 3])

index.value\_counts()

Output

1.0 2

3.0 1

2.0 1

dtype: int64

Example2

import pandas as pd

import numpy as np

index = pd.Index([2, 1, 1, np.nan, 3])

a = pd.Series([2, 1, 1, np.nan, 3])

a.value\_counts(normalize=True)

Output

1.0 0.50

3.0 0.25

2.0 0.25

dtype: float64

Example3

import pandas as pd

index = pd.Index([1, 3, 2, 2, 1, np.nan])

index.value\_counts()

a = pd.Series([1, 3, 2, 2, 1, np.nan])

a.value\_counts(bins=2)

Output

(0.997, 2.0] 4

(2.0, 3.0] 1

dtype: int64

Example4

import pandas as pd

index = pd.Index([1, 3, 2, 2, 1, np.nan])

index.value\_counts()

a = pd.Series([1, 3, 2, 2, 1, np.nan])

a.value\_counts(dropna=False)

Output

2.0 2

1.0 2

NaN 1

3.0 1

dtype: int64

**Python Pandas DataFrame**

Pandas DataFrame is a widely used data structure which works with a two-dimensional array with labeled axes (rows and columns). DataFrame is defined as a standard way to store data that has two different indexes, i.e., row index and column index. It consists of the following properties:

* The columns can be heterogeneous types like int, bool, and so on.
* It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.

Parameter & Description:

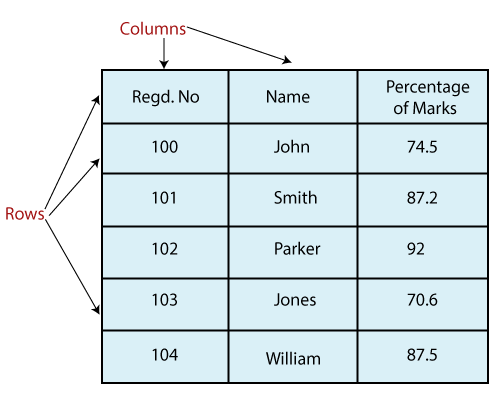
data: It consists of different forms like ndarray, series, map, constants, lists, array.

index: The Default np.arrange(n) index is used for the row labels if no index is passed.

columns: The default syntax is np.arrange(n) for the column labels. It shows only true if no index is passed.

dtype: It refers to the data type of each column.

copy(): It is used for copying the data.



**Read data from excel to Dataframe**:

import pandas as pd

import numpy as np

df=pd.read\_excel(r"D:\excelfiles\test.xlsx")

print(df)

**Create a DataFrame**

We can create a DataFrame using following ways:

* dict
* Lists
* Numpy ndarrrays
* Series

**Create an empty DataFrame**

The below code shows how to create an empty DataFrame in Pandas:

# importing the pandas library

import pandas as pd

df = pd.DataFrame()

print (df)

Output

Empty DataFrame

Columns: []

Index: []

Explanation: In the above code, first of all, we have imported the pandas library with the alias pd and then defined a variable named as df that consists an empty DataFrame. Finally, we have printed it by passing the df into the print.

**Create a DataFrame using List**:

We can easily create a DataFrame in Pandas using list.

# importing the pandas library

import pandas as pd

# a list of strings

x = ['Python', 'Pandas']

# Calling DataFrame constructor on list

df = pd.DataFrame(x)

print(df)

Output

0

0 Python

1 Pandas

Explanation: In the above code, we have defined a variable named "x" that consist of string values. The DataFrame constructor is being called for a list to print the values.

**Create a DataFrame from Dict of ndarrays/ Lists**

# importing the pandas library

import pandas as pd

info = {'ID' :[101, 102, 103],'Department' :['B.Sc','B.Tech','M.Tech',]}

df = pd.DataFrame(info)

print (df)

Output

ID Department

0 101 B.Sc

1 102 B.Tech

2 103 M.Tech

Explanation: In the above code, we have defined a dictionary named "info" that consist list of ID and Department. For printing the values, we have to call the info dictionary through a variable called df and pass it as an argument in print().

**Create a DataFrame from Dict of Series:**

# importing the pandas library

import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),

'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)

print (d1)

Output

one two

a 1.0 1

b 2.0 2

c 3.0 3

d 4.0 4

e 5.0 5

f 6.0 6

g NaN 7

h NaN 8

Explanation: In the above code, a dictionary named "info" consists of two Series with its respective index. For printing the values, we have to call the info dictionary through a variable called d1 and pass it as an argument in print().

**Column Selection**

We can select any column from the DataFrame. Here is the code that demonstrates how to select a column from the DataFrame.

# importing the pandas library

import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),

'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}

d1 = pd.DataFrame(info)

print (d1 ['one'])

Output

a 1.0

b 2.0

c 3.0

d 4.0

e 5.0

f 6.0

g NaN

h NaN

Name: one, dtype: float64

Explanation: In the above code, a dictionary named "info" consists of two Series with its respective index. Later, we have called the info dictionary through a variable d1 and selected the "one" Series from the DataFrame by passing it into the print().

**Column Addition**

We can also add any new column to an existing DataFrame. The below code demonstrates how to add any new column to an existing DataFrame:

# importing the pandas library

import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),

'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}

df = pd.DataFrame(info)

# Add a new column to an existing DataFrame object

print ("Add new column by passing series")

df['three']=pd.Series([20,40,60],index=['a','b','c'])

print (df)

print ("Add new column using existing DataFrame columns")

df['four']=df['one']+df['three']

print (df)

Output

Add new column by passing series

one two three

a 1.0 1 20.0

b 2.0 2 40.0

c 3.0 3 60.0

d 4.0 4 NaN

e 5.0 5 NaN

f NaN 6 NaN

Add new column using existing DataFrame columns

one two three four

a 1.0 1 20.0 21.0

b 2.0 2 40.0 42.0

c 3.0 3 60.0 63.0

d 4.0 4 NaN NaN

e 5.0 5 NaN NaN

f NaN 6 NaN NaN

Explanation: In the above code, a dictionary named as f consists two Series with its respective index. Later, we have called the info dictionary through a variable df.

To add a new column to an existing DataFrame object, we have passed a new series that contain some values concerning its index and printed its result using print().

We can add the new columns using the existing DataFrame. The "four" column has been added that stores the result of the addition of the two columns, i.e., one and three.

**Column Deletion:**

We can also delete any column from the existing DataFrame. This code helps to demonstrate how the column can be deleted from an existing DataFrame:

# importing the pandas library

import pandas as pd

info = {'one' : pd.Series([1, 2], index= ['a', 'b']),

'two' : pd.Series([1, 2, 3], index=['a', 'b', 'c'])}

df = pd.DataFrame(info)

print ("The DataFrame:")

print (df)

# using del function

print ("Delete the first column:")

del df['one']

print (df)

# using pop function

print ("Delete the another column:")

df.pop('two')

print (df)

Output

The DataFrame:

one two

a 1.0 1

b 2.0 2

c NaN 3

Delete the first column:

two

a 1

b 2

c 3

Delete the another column:

Empty DataFrame

Columns: []

Index: [a, b, c]

Explanation:

In the above code, the df variable is responsible for calling the info dictionary and print the entire values of the dictionary. We can use the delete or pop function to delete the columns from the DataFrame.

In the first case, we have used the delete function to delete the "one" column from the DataFrame whereas in the second case, we have used the pop function to remove the "two" column from the DataFrame.

**Row Selection, Addition, and Deletion**

**Row Selection**:

We can easily select, add, or delete any row at anytime. First of all, we will understand the row selection. Let's see how we can select a row using different ways that are as follows:

Selection by Label:

We can select any row by passing the row label to a loc function.

# importing the pandas library

import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),

'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}

df = pd.DataFrame(info)

print (df.loc['b'])

Output

one 2.0

two 2.0

Name: b, dtype: float64

Explanation: In the above code, a dictionary named as info that consists two Series with its respective index.

For selecting a row, we have passed the row label to a loc function.

**Selection by integer location:**

The rows can also be selected by passing the integer location to an iloc function.

# importing the pandas library

import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),

'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}

df = pd.DataFrame(info)

print (df.iloc[3])

Output

one 4.0

two 4.0

Name: d, dtype: float64

Explanation: Explanation: In the above code, we have defined a dictionary named as info that consists two Series with its respective index.

For selecting a row, we have passed the integer location to an iloc function.

**Slice Rows**

It is another method to select multiple rows using ':' operator.

# importing the pandas library

import pandas as pd

info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),

'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}

df = pd.DataFrame(info)

print (df[2:5])

Output

one two

c 3.0 3

d 4.0 4

e 5.0 5

Explanation: In the above code, we have defined a range from 2:5 for the selection of row and then printed its values on the console.

**Addition of rows:**

We can easily add new rows to the DataFrame using append function. It adds the new rows at the end.

# importing the pandas library

import pandas as pd

d = pd.DataFrame([[7, 8], [9, 10]], columns = ['x','y'])

d2 = pd.DataFrame([[11, 12], [13, 14]], columns = ['x','y'])

d = d.append(d2)

print (d)

Output

x y

0 7 8

1 9 10

0 11 12

1 13 14

Explanation: In the above code, we have defined two separate lists that contains some rows and columns. These columns have been added using the append function and then result is displayed on the console.

**Deletion of rows:**

We can delete or drop any rows from a DataFrame using the index label. If in case, the label is duplicate then multiple rows will be deleted.

# importing the pandas library

import pandas as pd

a\_info = pd.DataFrame([[4, 5], [6, 7]], columns = ['x','y'])

b\_info = pd.DataFrame([[8, 9], [10, 11]], columns = ['x','y'])

a\_info = a\_info.append(b\_info)

# Drop rows with label 0

a\_info = a\_info.drop(0)

Output

x y

1 6 7

1 10 11

Explanation: In the above code, we have defined two separate lists that contains some rows and columns.

Here, we have defined the index label of a row that needs to be deleted from the list.

**DataFrame Functions**

There are lots of functions used in DataFrame which are as follows:

|  |  |
| --- | --- |
| Functions | Description |
| Pandas DataFrame.append() | Add the rows of other dataframe to the end of the given dataframe. |
| Pandas DataFrame.apply() | Allows the user to pass a function and apply it to every single value of the Pandas series. |
| Pandas DataFrame.assign() | Add new column into a dataframe. |
| Pandas DataFrame.astype() | Cast the Pandas object to a specified dtype.astype() function. |
| Pandas DataFrame.concat() | Perform concatenation operation along an axis in the DataFrame. |
| Pandas DataFrame.count() | Count the number of non-NA cells for each column or row. |
| Pandas DataFrame.describe() | Calculate some statistical data like percentile, mean and std of the numerical values of the Series or DataFrame. |
| Pandas DataFrame.drop\_duplicates() | Remove duplicate values from the DataFrame. |
| Pandas DataFrame.groupby() | Split the data into various groups. |
| Pandas DataFrame.head() | Returns the first n rows for the object based on position. |
| Pandas DataFrame.hist() | Divide the values within a numerical variable into "bins". |
| Pandas DataFrame.iterrows() | Iterate over the rows as (index, series) pairs. |
| Pandas DataFrame.mean() | Return the mean of the values for the requested axis. |
| Pandas DataFrame.melt() | Unpivots the DataFrame from a wide format to a long format. |
| Pandas DataFrame.merge() | Merge the two datasets together into one. |
| Pandas DataFrame.pivot\_table() | Aggregate data with calculations such as Sum, Count, Average, Max, and Min. |
| Pandas DataFrame.query() | Filter the dataframe. |
| Pandas DataFrame.sample() | Select the rows and columns from the dataframe randomly. |
| Pandas DataFrame.shift() | Shift column or subtract the column value with the previous row value from the dataframe. |
| Pandas DataFrame.sort() | Sort the dataframe. |
| Pandas DataFrame.sum() | Return the sum of the values for the requested axis by the user. |
| Pandas DataFrame.to\_excel() | Export the dataframe to the excel file. |
| Pandas DataFrame.transpose() | Transpose the index and columns of the dataframe. |
| Pandas DataFrame.where() | Check the dataframe for one or more conditions. |

**Pandas DataFrame.append()**

The Pandas append() function is used to add the rows of other dataframe to the end of the given dataframe, returning a new dataframe object. The new columns and the new cells are inserted into the original DataFrame that are populated with NaN value.

Syntax:

DataFrame.append(other, ignore\_index=False, verify\_integrity=False, sort=None)

Parameters:

* other: DataFrame or Series/dict-like object, or a list of these

It refers to the data to be appended.

* ignore\_index: If it is true, it does not use the index labels.
* verify\_integrity: If it is true, it raises ValueError on creating an index with duplicates.
* sort: It sorts the columns if the columns of self and other are not aligned. The default sorting is deprecated, and it will change to not-sorting in a future version of pandas. We pass sort=True Explicitly for silence the warning and the sort, whereas we pass sort=False Explicitly for silence the warning and not the sort.

Returns:

It returns the appended DataFrame as an output.

Example1:

import pandas as pd

# Create first Dataframe using dictionary

info1 = pd.DataFrame({"x":[25,15,12,19],

"y":[47, 24, 17, 29]})

# Create second Dataframe using dictionary

Info2 = pd.DataFrame({"x":[25, 15, 12],

"y":[47, 24, 17],

"z":[38, 12, 45]})

# append info2 at end in info1

Info1.append(info2, ignore\_index = True)

Output

x y z

0 25 47 NaN

1 15 24 NaN

2 12 17 NaN

3 19 29 NaN

4 25 47 38.0

5 15 24 12.0

6 12 17 45.0

Example2:

import pandas as pd

# Create first Dataframe using dictionary

info1 = info = pd.DataFrame({"x":[15, 25, 37, 42],

"y":[24, 38, 18, 45]})

# Create second Dataframe using dictionary

info2 = pd.DataFrame({"x":[15, 25, 37],

"y":[24, 38, 45]})

# print value of info1

print(info1, "\n")

# print values of info2

info2

# append info2 at the end of info1 dataframe

info1.append(df2)

# Continuous index value will maintained

# across rows in the new appended data frame.

info.append(info2, ignore\_index = True)

Output

x y

0 15 24

1 25 38

2 37 18

3 42 45

4 15 24

5 25 38

6 37 45

**Pandas DataFrame.apply()**

The Pandas apply() function allows the user to pass a function and apply it to every single value of the Pandas series. This function improves the capabilities of the panda's library because it helps to segregate data according to the conditions required. So that it can be efficiently used for data science and machine learning.

The objects that are to be passed to function are Series objects whose index is either the DataFrame's index, i.e., axis=0 or the DataFrame's columns, i.e., axis=1. By default, the result\_type=None and the final return type is inferred from the return type of the applied function. Otherwise, it depends on the result\_type argument.

Syntax:

DataFrame.apply(func, axis=0, broadcast=None, raw=False, reduce=None, result\_type=None, args=(), \*\*kwds)

Parameters:

* func: It is a function that is to be applied to each column or row.
* axis: {0 or 'index', 1 or 'columns'}, default value 0

It is an axis along which the function is applied. It can have two values:

* + - 0 or 'index': It applies the function to each of the columns.
    - 1 or 'columns': It applies the function to each of the rows.
* broadcast: It is an optional parameter that returns the boolean values.

Only relevant for aggregation functions:

False or None: It returns a Series whose length will be the length of the index or the number of columns based on the axis parameter.

True: The results will be broadcasted to the original shape of the frame; the original index and columns will be retained.

* raw: bool, default value False

False: It passes each row or column as a Series to the function.

True: The passed function will receive a ndarray objects. If you are applying a NumPy reduction function, it will achieve better performance.

* reduce: bool or None, default value None

It tries to apply the reduction procedures. If the DataFrame is empty, the apply will use the reduce to determine whether the result should be a Series or a DataFrame.

By default, reduce=None, the apply's return value will be guessed by calling func on an empty Series (note: All the exceptions that are to be raised by func will be ignored while guessing). If reduce=True, Series will always be returned, whereas reduce=False, Always the DataFrame will be returned.

* result\_type: {'expand', 'reduce', 'broadcast', None}, default value None

These only act when axis=1 (columns):

'expand': It defines the list-like results that will be turned into columns.

'reduce': It is the opposite of 'expand'. If possible, it returns a Series rather than expanding list-like results.

'broadcast': It broadcast the results to the original shape of the DataFrame, the original index, and the columns will be retained.

The default value None depends on the return value of the applied function , i.e., list-like results returned as a Series of those.

If apply returns a Series, it expands to the columns.

* args: It is a positional argument that is to be passed to func in addition to the array/series.
* \*\*kwds: It is an optional keyword argument, which is used to pass as keywords arguments to func.

Returns:

It returns the result of applying func along the given axis of the DataFrame.

Example:

info = pd.DataFrame([[2, 7]] \* 4, columns=['P', 'Q'])

info.apply(np.sqrt)

info.apply(np.sum, axis=0)

info.apply(np.sum, axis=1)

info.apply(lambda x: [1, 2], axis=1)

info.apply(lambda x: [1, 2], axis=1, result\_type='expand')

info.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)

info.apply(lambda x: [1, 2], axis=1, result\_type='broadcast')

info

Output

A B

0 2 7

1 2 7

2 2 7

3 2 7

**Pandas DataFrame.aggregate()**

The main task of DataFrame.aggregate() function is to apply some aggregation to one or more column. Most frequently used aggregations are:

sum: It is used to return the sum of the values for the requested axis.

min: It is used to return the minimum of the values for the requested axis.

max: It is used to return the maximum values for the requested axis.

Syntax:

DataFrame.aggregate(func, axis=0, \*args, \*\*kwargs)

Parameters:

func: It refers callable, string, dictionary, or list of string/callables.

It is used for aggregating the data. For a function, it must either work when passed to a DataFrame or DataFrame.apply(). For a DataFrame, it can pass a dict, if the keys are the column names.

axis: (default 0): It refers to 0 or 'index', 1 or 'columns'

0 or 'index': It is an apply function for each column.

1 or 'columns': It is an apply function for each row.

\*args: It is a positional argument that is to be passed to func.

\*\*kwargs: It is a keyword argument that is to be passed to the func.

Returns:

It returns the scalar, Series or DataFrame.

scalar: It is being used when Series.agg is called with the single function.

Series: It is being used when DataFrame.agg is called for the single function.

DataFrame: It is being used when DataFrame.agg is called for the several functions.

Example:

import pandas as pd

import numpy as np

info=pd.DataFrame([[1,5,7],[10,12,15],[18,21,24],[np.nan,np.nan,np.nan]],columns=['X','Y','Z'])

info.agg(['sum','min'])

Output:

X Y Z

sum 29.0 38.0 46.0

min 1.0 5.0 7.0

Example2:

import pandas as pd

import numpy as np

info=pd.DataFrame([[1,5,7],[10,12,15],[18,21,24],[np.nan,np.nan,np.nan]],columns=['X','Y','Z'])

df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})

Output:

X Y

max NaN 21.0

min 1.0 12.0

sum 29.0 NaN

**Pandas DataFrame.assign()**

The assign() method is also responsible for adding a new column into a DataFrame.

If we re-assign an existing column, then its value will be overwritten.

Signature

DataFrame.assign(\*\*kwargs)

Parameters

* kwargs: keywords are the column names. These keywords are assigned to the new column if the values are callable. If the values are not callable, they are simply assigned.

Returns

It returns a new DataFrame with the addition of the new columns.

Example 1:

import pandas as pd

# Create an empty dataframe

info = pd.DataFrame()

# Create a column

info['ID'] = [101, 102, 103]

# View the dataframe

info

# Assign a new column to dataframe called 'age'

info.assign(Name = ['Smith', 'Parker', 'John'])

Output

ID Name

0 101 Smith

1 102 Parker

2 103 John

Example 2:

import pandas as pd

# Create a dataframe

info = pd.DataFrame({'temp\_c': [17.0, 25.0]},

# Create an index that consist some values

index=['Canada', 'Australia'])

# View the dataframe

info

info.assign(temp\_f=lambda x: x.temp\_c \* 7 / 2 + 24)

info.assign(temp\_f=lambda x: x['temp\_c'] \* 6 / 2 + 21,

temp\_k=lambda x: (x['temp\_f'] + 342.27) \* 6 / 4)

Output

temp\_c temp\_f temp\_k

Canada 17.0 72.0 621.405

Australia 25.0 96.0 657.405

**Pandas DataFrame.count()**

The Pandas count() is defined as a method that is used to count the number of non-NA cells for each column or row. It is also suitable to work with the non-floating data.

Syntax:

DataFrame.count(axis=0, level=None, numeric\_only=False)

Parameters:

axis: {0 or 'index', 1 or 'columns'}, default value 0

0 or 'index' is used for row-wise, whereas 1 or 'columns' is used for column-wise.

level: int or str

It is an optional parameter. If an axis is hierarchical, it counts along with the particular level and collapsing into the DataFrame.

numeric\_only: bool, default value False

It only includes int, float, or Boolean data.

Returns:

It returns the count of Series or DataFrame if the level is specified.

Example 1: The below example demonstrates the working of the count().

import pandas as pd

import numpy as np

info = pd.DataFrame({"Person":["Parker", "Smith", "William", "John"],

"Age": [27., 29, np.nan, 32]

info.count()

Output

Person 5

Age 3

dtype: int64

Example 2: If we want to count for each of the row, we can use the axis parameter. The below code demonstrates the working of the axis parameter.

import pandas as pd

import numpy as np

info = pd.DataFrame({"Person":["Parker", "Smith", "William", "John"],

"Age": [27., 29, np.nan, 32]

info.count(axis='columns')

Output

0 2

1 2

2 1

3 2

dtype: int64

**Pandas DataFrame.describe()**

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

DataFrame.describe(percentiles=None, include=None, exclude=None)

Parameters

* percentile: It is an optional parameter which is a list like data type of numbers that should fall between 0 and 1. Its default value is [.25, .5, .75], which returns the 25th, 50th, and 75th percentiles.
* include: It is also an optional parameter that includes the list of the data types while describing the DataFrame. Its default value is None.
* exclude: It is also an optional parameter that exclude the list of data types while describing DataFrame. Its default value is None.

Returns

It returns the statistical summary of the Series and DataFrame.

Example1

import pandas as pd

import numpy as np

a1 = pd.Series([1, 2, 3])

a1.describe()

Output

count 3.0

mean 2.0

std 1.0

min 1.0

25% 1.5

50% 2.0

75% 2.5

max 3.0

dtype: float64

Example2

import pandas as pd

import numpy as np

a1 = pd.Series(['p', 'q', 'q', 'r'])

a1.describe()

Output

count 4

unique 3

top q

freq 2

dtype: object

Example3

import pandas as pd

import numpy as np

a1 = pd.Series([1, 2, 3])

a1.describe()

a1 = pd.Series(['p', 'q', 'q', 'r'])

a1.describe()

info = pd.DataFrame({'categorical': pd.Categorical(['s','t','u']),

'numeric': [1, 2, 3],

'object': ['p', 'q', 'r']

})

info.describe(include=[np.number])

info.describe(include=[np.object])

info.describe(include=['category'])

Output

categorical

count 3

unique 3

top u

freq 1

Example4

import pandas as pd

import numpy as np

a1 = pd.Series([1, 2, 3])

a1.describe()

a1 = pd.Series(['p', 'q', 'q', 'r'])

a1.describe()

info = pd.DataFrame({'categorical': pd.Categorical(['s','t','u']),

'numeric': [1, 2, 3],

'object': ['p', 'q', 'r']

})

info.describe()

info.describe(include='all')

info.numeric.describe()

info.describe(include=[np.number])

info.describe(include=[np.object])

info.describe(include=['category'])

info.describe(exclude=[np.number])

info.describe(exclude=[np.object])

Output

categorical numeric

count 3 3.0

unique 3 NaN

top u NaN

freq 1 NaN

mean NaN 2.0

std NaN 1.0

min NaN 1.0

25% NaN 1.5

50% NaN 2.0

75% NaN 2.5

max NaN 3.0

**Pandas DataFrame.drop\_duplicates()**

The drop\_duplicates() function performs common data cleaning task that deals with duplicate values in the DataFrame. This method helps in removing duplicate values from the DataFrame.

Syntax

DataFrame.drop\_duplicates(subset=None, keep='first', inplace=False)

Parameters

* subset: It takes a column or the list of column labels. It considers only certain columns for identifying duplicates. Default value None.
* keep: It is used to control how to consider duplicate values. It has three distinct values that are as follows:
  1. first: It drops the duplicate values except for the first occurrence.
  2. last: It drops the duplicate values except for the last occurrence.
  3. False: It drops all the duplicates.
* inplace: Returns the boolean value. Default value is False.

If it is true, it removes the rows with duplicate values.

Return

Depending on the arguments passed, it returns the DataFrame with the removal of duplicate rows.

Example

import pandas as pd

emp = {"Name": ["Parker", "Smith", "William", "Parker"],

"Age": [21, 32, 29, 21]}

info = pd.DataFrame(emp)

print(info)

Output

Name Age

0 Parker 21

1 Smith 32

2 William 29

3 Parker 21

import pandas as pd

emp = {"Name": ["Parker", "Smith", "William", "Parker"],

"Age": [21, 32, 29, 21]}

info = pd.DataFrame(emp)

info = info.drop\_duplicates()

print(info)

Output

Name Age

0 Parker 21

1 Smith 32

2 William 29

**Pandas DataFrame.groupby()**

In Pandas, groupby() function allows us to rearrange the data by utilizing them on real-world data sets. Its primary task is to split the data into various groups. These groups are categorized based on some criteria. The objects can be divided from any of their axes.

**Syntax**:

DataFrame.groupby(by=None, axis=0, level=None, as\_index=True, sort=True, group\_keys=True, squeeze=False, \*\*kwargs)

This operation consists of the following steps for aggregating/grouping the data:

* Splitting datasets
* Analyzing data
* Aggregating or combining data

Note: The result of Groupby operation is not a DataFrame, but dict of DataFrame objects.

**Split data into groups**

There are multiple ways to split any object into the group which are as follows:

* obj.groupby('key')
* obj.groupby(['key1','key2'])
* obj.groupby(key,axis=1)

We can also add some functionality to each subset. The following operations can be performed on the applied functionality:

* Aggregation: Computes summary statistic.
* Transformation: It performs some group-specific operation.
* Filtration: It filters the data by discarding it with some condition.

**Aggregations**

It is defined as a function that returns a single aggregated value for each of the groups. We can perform several aggregation operations on the grouped data when the groupby object is created.

Example

# import the pandas library

import pandas as pd

import numpy as np

data = {'Name': ['Parker', 'Smith', 'John', 'William'],

'Percentage': [82, 98, 91, 87],

'Course': ['B.Sc','B.Ed','M.Phill','BA']}

df = pd.DataFrame(data)

grouped = df.groupby('Course')

print(grouped['Percentage'].agg(np.mean))

Output

Course

B.Ed 98

B.Sc 82

BA 87

M.Phill 91

Name: Percentage, dtype: int64

**Transformations**

It is an operation on a group or column that performs some group-specific computation and returns an object that is indexed with the same size as of the group size.

Example

# import the pandas library

import pandas as pd

import numpy as np

data = {'Name': ['Parker', 'Smith', 'John', 'William'],

'Percentage': [82, 98, 91, 87],

'Course': ['B.Sc','B.Ed','M.Phill','BA']}

df = pd.DataFrame(data)

grouped = df.groupby('Course')

Percentage = lambda x: (x - x.mean()) / x.std()\*10

print(grouped.transform(Percentage))

Output

Percentage

0 NaN

1 NaN

2 NaN

3 NaN

**Filtration**

The filter() function filters the data by defining some criteria and returns the subset of data.

Example

# import the pandas library

import pandas as pd

import numpy as np

data = {'Name': ['Parker', 'Smith', 'John', 'William'],

'Percentage': [82, 98, 91, 87],

'Course': ['B.Sc','B.Ed','M.Phill','BA']}

df = pd.DataFrame(data)

grouped = df.groupby('Course')

print (df.groupby('Course').filter(lambda x: len(x) >= 1))

Output

Name Percentage Course

0 Parker 82 B.Sc

1 Smith 98 B.Ed

2 John 91 M.Phill

3 William 87 BA

**Parameters of Groupby:**

* by: mapping, function, str, or iterable

Its main task is to determine the groups in the groupby. If we use by as a function, it is called on each value of the object's index. If in case a dict or Series is passed, then the Series or dict VALUES will be used to determine the groups.

If a ndarray is passed, then the values are used as-is determine the groups.

We can also pass the label or list of labels to group by the columns in the self.

* axis: {0 or 'index', 1 or 'columns'}, default value 0
* level: int, level name, or sequence of such, default value None.

It is used when the axis is a MultiIndex (hierarchical), so, it will group by a particular level or levels.

* as\_index: bool, default True

It returns the object with group labels as the index for the aggregated output.

* sort: bool, default True

It is used to sort the group keys. Get better performance by turning this off.

Note: It does not influence the order of observations within each group. The Groupby preserves the order of rows within each group.

* group\_keys: bool, default value True

When we call it, it adds the group keys to the index for identifying the pieces.

* observed: bool, default value False

It will be used only if any of the groupers are the Categoricals. If the value is True, then it will show only the observed values for categorical groupers. Otherwise, it will show all of its values.

* \*\*kwargs

It is an optional parameter that only accepts the keyword argument 'mutated' that is passed to groupby.

Returns

It returns the DataFrameGroupBy or SeriesGroupBy. The return value depends on the calling object that consists of information about the groups.

Example:

df2=df1.groupby(["colname1"," colname2"])[" colname"].sum().reset\_index(name ='Total\_val')

**Pandas DataFrame.head()**

The head() returns the first n rows for the object based on position. If your object has the right type of data in it, it is useful for quick testing. This method is used for returning top n (**by default value 5)** rows of a data frame or series.

Syntax

DataFrame.head(n=5)

Parameters

n: It refers to an integer value that returns the number of rows.

Return

It returns the DataFrame with top n rows.

Example1

info = pd.DataFrame({'language':['C', 'C++', 'Python', 'Java','PHP']})

info.head()

info.head(3)

Output

language

0 C

1 C++

2 Python

Example 2

We have a csv file "aa.csv" that have the following dataset.

Name Hire Date Salary Leaves Remaining

0 John Idle 03/15/14 50000.0 10

1 Smith Gilliam 06/01/15 65000.0 8

2 Parker Chapman 05/12/14 45000.0 10

3 Jones Palin 11/01/13 70000.0 3

4 Terry Gilliam 08/12/14 48000.0 7

By using the head() in the below example, we showed only top 2 rows from the dataset.

# importing pandas module

import pandas as pd

# making data frame

data = pd.read\_csv("aa.csv")

# calling head() method

# storing in new variable

data\_top = data.head(2)

# display

data\_top

Name Hire Date Salary Leaves Remaining

0 John Idle 03/15/14 50000.0 10

1 Smith Gilliam 06/01/15 65000.0 8

**Pandas DataFrame.mean()**

The mean() function is used to return the mean of the values for the requested axis. If we apply this method on a Series object, then it returns a scalar value, which is the mean value of all the observations in the dataframe.

If we apply this method on a DataFrame object, then it returns a Series object which contains mean of values over the specified axis.

Syntax

DataFrame.mean(axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs)

Parameters

axis: {index (0), columns (1)}.

This refers to the axis for a function that is to be applied.

skipna: It excludes all the null values when computing result.

level: It counts along with a particular level and collapsing into a Series if the axis is a MultiIndex (hierarchical),

numeric\_only: It includes only int, float, boolean columns. If None, it will attempt to use everything, then use only numeric data. Not implemented for Series.

Returns

It returns the mean of the Series or DataFrame if the level is specified.

Example

# importing pandas as pd

import pandas as pd

# Creating the dataframe

info = pd.DataFrame({"A":[8, 2, 7, 12, 6],

"B":[26, 19, 7, 5, 9],

"C":[10, 11, 15, 4, 3],

"D":[16, 24, 14, 22, 1]})

# Print the dataframe

print(info)

# If axis = 0 is not specified, then

# by default method return the mean over

# the index axis

info.mean(axis = 0)

Output

A 7.0

B 13.2

C 8.6

D 15.4

dtype: float64

Example2

# importing pandas as pd

import pandas as pd

# Creating the dataframe

info = pd.DataFrame({"A":[5, 2, 6, 4, None],

"B":[12, 19, None, 8, 21],

"C":[15, 26, 11, None, 3],

"D":[14, 17, 29, 16, 23]})

# while finding mean, it skip null values

info.mean(axis = 1, skipna = True)

Output

0 11.500000

1 16.000000

2 15.333333

3 9.333333

4 15.666667

dtype: float64

**Pandas DataFrame.sum()**

Pandas DataFrame.sum() function is used to return the sum of the values for the requested axis by the user. If the input value is an index axis, then it will add all the values in a column and works same for all the columns. It returns a series that contains the sum of all the values in each column.

It is also capable of skipping the missing values in the DataFrame while calculating the sum in the DataFrame.

Syntax:

DataFrame.sum(axis=None, skipna=None, level=None, numeric\_only=None, min\_count=0, \*\*kwargs)

Parameters

axis: {index (0), columns (1)}. 0 or 'index' is used for row-wise, whereas 1 or 'columns' is used for column-wise.

skipna: bool, default True. It is used to exclude all the null values.

level: int or level name, default None. It counts along a particular level and collapsing into a series, if the axis is a multiindex.

numeric\_only: bool, default value None. It includes only int, float, and boolean columns. If it is None, it will attempt to use everything, so numeric data should be used.

min\_count: int, default value 0. It refers to the required number of valid values to perform any operation. If it is fewer than the min\_count non-NA values are present, then the result will be NaN.

\*\*kwargs: It is an optional parameter that is to be passed to a function.

Returns:

It returns the sum of Series or DataFrame if a level is specified.

Example1:

import pandas as pd

# default min\_count = 0

pd.Series([]).sum()

# Passed min\_count = 1, then sum of an empty series will be NaN

pd.Series([]).sum(min\_count = 1)

Output

0.0

nan

Example2:

import pandas as pd

# making a dict of list

info = {'Name': ['Parker', 'Smith', 'William'], 'age' : [32, 28, 39]}

data = pd.DataFrame(info)

# sum of all salary stored in 'total'

data['total'] = data['age'].sum()

print(data)

Output

Name age total

0 Parker 32 99

1 Smith 28 99

2 William 39 99

**Pandas DataFrame.iterrows()**

If you want to loop over the DataFrame for performing some operations on each of the rows then you can use iterrows() function in Pandas.

Pandas use three functions for iterating over the rows of the DataFrame, i.e., iterrows(), iteritems() and itertuples().

**Iterate rows with Pandas iterrows:**

The iterrows () is responsible for loop through each row of the DataFrame. It returns an iterator that contains index and data of each row as a Series.

We have the next function to see the content of the iterator.

This function returns each index value along with a series that contain the data in each row.

* iterrows() - used for iterating over the rows as (index, series) pairs.
* iteritems() - used for iterating over the (key, value) pairs.
* itertuples() - used for iterating over the rows as namedtuples.

**Yields:**

* index: Returns the index of the row and a tuple for the MultiIndex.
* data: Returns the data of the row as a Series.
* it: Returns a generator that iterates over the rows of the frame.

Example1

import pandas as pd

import numpy as np

info = pd.DataFrame(np.random.randn(4,2),columns = ['col1','col2'])

for row\_index,row in info.iterrows():

print (row\_index,row)

Output

0 name John

degree B.Tech

score 90

Name: 0, dtype: object

1 name Smith

degree B.Com

score 40

Name: 1, dtype: object

2 name Alexander

degree M.Com

score 80

Name: 2, dtype: object

3 name William

degree M.Tech

score 98

Name: 3, dtype: object

Example2

# importing pandas module

import pandas as pd

# making data frame from csv file

data = pd.read\_csv("aa.csv")

for i, j in data.iterrows():

print(i, j)

print()

Output

0 Name Hire Date Salary Leaves Remaining 0 John Idle 03/15/14 50...

Name: 0, dtype: object

1 Name Hire Date Salary Leaves Remaining 1 Smith Gilliam 06/01/15 65000...

Name: 1, dtype: object

2 Name Hire Date Salary Leaves Remaining 2 Parker Chapman 05/12/14 45000.0 ...

Name: 2, dtype: object

3 Name Hire Date Salary Leaves Remaining 3 Jones Palin 11/01/13 700...

Name: 3, dtype: object

4 Name Hire Date Salary Leaves Remaining 4 Terry Gilliam 08/12/14 4800...

Name: 4, dtype: object

5 Name Hire Date Salary Leaves Remaining 5 Michael Palin 05/23/13 66000...

Name: 5, dtype: object

**iteritems()**

Example

import pandas as pd

import numpy as np

info = pd.DataFrame(np.random.randn(4,2),columns = ['col1','col2'])

for label,content in info.iteritems():

print (“Label”,label)

print (“Content”,content)

**itertuples()**

Example:

import pandas as pd

import numpy as np

info = pd.DataFrame(np.random.randn(4,2),columns = ['col1','col2'])

for row in info.itertuples():

print (row)

**Pandas DataFrame.query()**

For analyzing the data, we need a lot of filtering operations. Pandas provide a query() method to filter the DataFrame.

It offers a simple way of making the selection and also capable of simplifying the task of index-based selection.

Syntax

DataFrame.query(expr, inplace=False, \*\*kwargs)

Parameters

expr: Refers to an expression in string form to filter data.

inplace: If the value is True, it makes the changes in the original DataFrame.

kwargs: Refers to the other keyword arguments.

Return

It returns a DataFrame that results from the query expression.

Note: This method only works if the column name doesn't have any empty spaces. You can replace the spaces in column names with '\_'

Example1

info = pd.DataFrame({'X': range(1, 6),

'Y': range(10, 0, -2),

'Z Z': range(10, 5, -1)})

info

info.query('X > Y')

info[info.X > info.Y]

info[info.Y == info['Z Z']]

Output

X Y Z Z

0 1 10 10

**Pandas DataFrame.where()**

Pandas where() method is used to check a data frame for one or more condition and return the result accordingly. By default, The rows not satisfying the condition are filled with NaN value.

Syntax:

DataFrame.where(cond, other=nan, inplace=False, axis=None, level=None, errors=’raise’, try\_cast=False, raise\_on\_error=None)

Parameters:

cond: One or more condition to check data frame for.

other: Replace rows which don’t satisfy the condition with user defined object, Default is NaN

inplace: Boolean value, Makes changes in data frame itself if True

axis: axis to check( row or columns)

Example #1: Single Condition operation

In this example, rows having particular Team name will be shown and rest will be replaced by NaN using .where() method.

# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read\_csv("nba.csv")

# sorting dataframe

data.sort\_values("Team", inplace = True)

# making boolean series for a team name

filter = data["Team"]=="Atlanta Hawks"

# filtering data

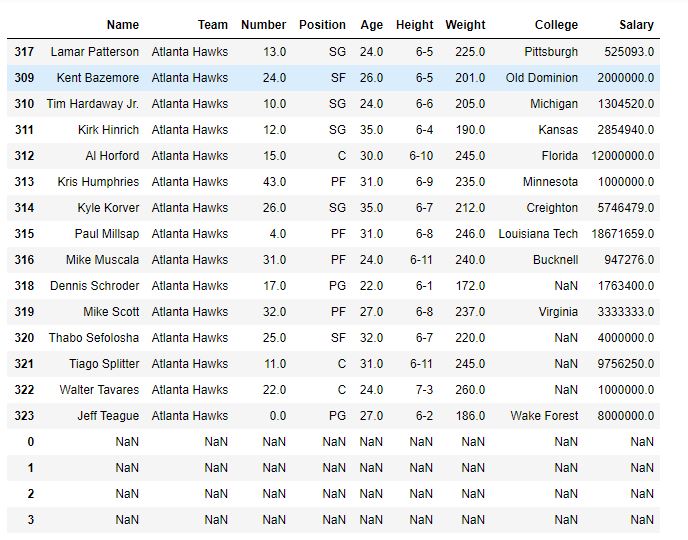
data.where(filter, inplace = True)

# display

data

Output:

As shown in the output image, every row which doesn’t have Team = Atlanta Hawks is replaced with NaN.



Example #2: Multi-condition Operations

Data is filtered on the basis of both Team and Age. Only the rows having Team name “Atlanta Hawks” and players having age above 24 will be displayed.

# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read\_csv("nba.csv")

# sorting dataframe

data.sort\_values("Team", inplace = True)

# making boolean series for a team name

filter1 = data["Team"]=="Atlanta Hawks"

# making boolean series for age

filter2 = data["Age"]>24

# filtering data on basis of both filters

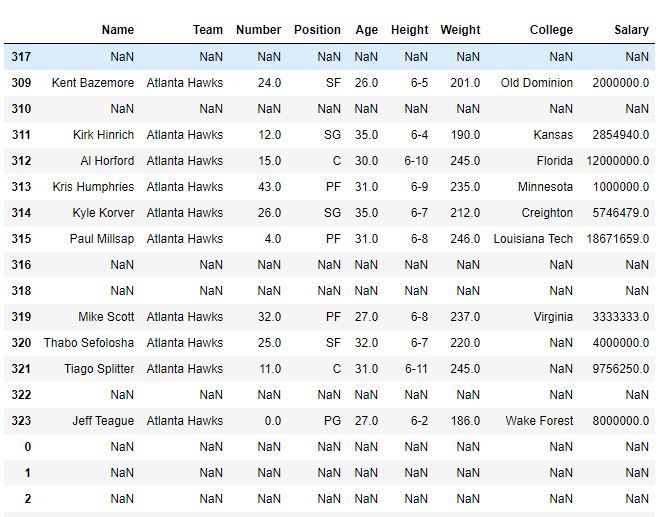
data.where(filter1 & filter2, inplace = True)

# display

data

Output:

As shown in the output image, Only the rows having Team name “Atlanta Hawks” and players having age above 24 are displayed.



**Pandas DataFrame.transform**

Pandas DataFrame.transform() function call func on self producing a DataFrame with transformed values and that has the same axis length as self.

Syntax: DataFrame.transform(func, axis=0, \*args, \*\*kwargs)

Parameter :

func : Function to use for transforming the data

axis : {0 or ‘index’, 1 or ‘columns’}, default 0

\*args : Positional arguments to pass to func.

\*\*kwargs : Keyword arguments to pass to func.

Returns : DataFrame

Example #1 : Use DataFrame.transform() function to add 10 to each element in the dataframe.

# importing pandas as pd

import pandas as pd

# Creating the DataFrame

df = pd.DataFrame({"A":[12, 4, 5, None, 1],

"B":[7, 2, 54, 3, None],

"C":[20, 16, 11, 3, 8],

"D":[14, 3, None, 2, 6]})

# Create the index

index\_ = ['Row\_1', 'Row\_2', 'Row\_3', 'Row\_4', 'Row\_5']

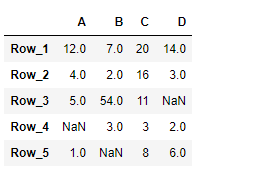
# Set the index

df.index = index\_

# Print the DataFrame

print(df)

Output :



Now we will use DataFrame.transform() function to add 10 to each element of the dataframe.

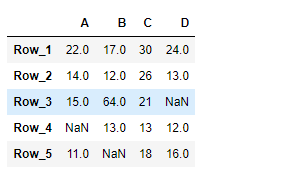
# add 10 to each element of the dataframe

result = df.transform(func = lambda x : x + 10)

# Print the result

print(result)

Output :



As we can see in the output, the DataFrame.transform() function has successfully added 10 to each element of the given Dataframe.

Example #2 : Use DataFrame.transform() function to find the square root and the result of euler’s number raised to each element of the dataframe.

# importing pandas as pd

import pandas as pd

# Creating the DataFrame

df = pd.DataFrame({"A":[12, 4, 5, None, 1],

"B":[7, 2, 54, 3, None],

"C":[20, 16, 11, 3, 8],

"D":[14, 3, None, 2, 6]})

# Create the index

index\_ = ['Row\_1', 'Row\_2', 'Row\_3', 'Row\_4', 'Row\_5']

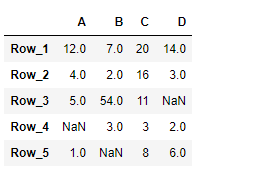
# Set the index

df.index = index\_

# Print the DataFrame

print(df)

Output :



Now we will use DataFrame.transform() function to find the square root and the result of euler’s number raised to each element of the dataframe.

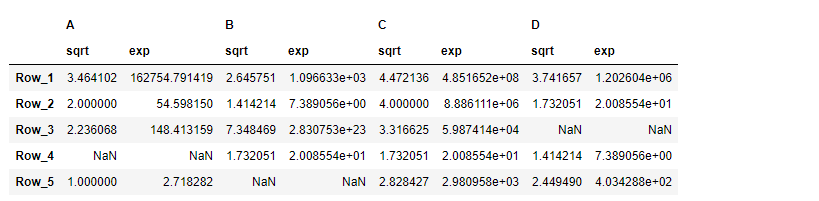
# pass a list of functions

result = df.transform(func = ['sqrt', 'exp'])

# Print the result

print(result)

Output :



As we can see in the output, the DataFrame.transform() function has successfully performed the desired operation on the given dataframe.

**Pandas DataFrame.transpose()**

The transpose() function helps to transpose the index and columns of the dataframe. It reflects DataFrame over its main diagonal by writing the rows as columns and vice-versa.

Syntax

DataFrame.transpose(\*args, \*\*kwargs)

Parameters

copy: If its value is True, then the underlying data is being copied. Otherwise, by default, no copy is made, if possible.

\*args, \*\*kwargs: Both are additional keywords that do not affect, but has an acceptance that provide compatibility with a numpy.

Returns

It returns the transposed DataFrame.

Example1

# importing pandas as pd

import pandas as pd

# Creating the DataFrame

info = pd.DataFrame({'Weight':[27, 44, 38, 10, 67],

'Name':['William', 'John', 'Smith', 'Parker', 'Jones'],

'Age':[22, 17, 19, 24, 27]})

# Create the index

index\_ = pd.date\_range('2010-10-04 06:15', periods = 5, freq ='H')

# Set the index

info.index = index\_

# Print the DataFrame

print(info)

# return the transpose

result = info.transpose()

# Print the result

print(result)

Output

Weight Name Age

2010-10-04 06:15:00 27 William 22

2010-10-04 07:15:00 44 John 7

2010-10-04 08:15:00 38 Smith 19

2010-10-04 09:15:00 10 Parker 24

2010-10-04 10:15:00 67 Jones 27

2010-10-04 06:15:00 2010-10-04 07:15:00 2010-10-04 08:15:00 2010-10-04 09:15:00 2010-10-04 10:15:00

Weight 27 44 38 10 67

Name William John Smith Parker Jones

Age 22 7 19 24 27

Example2

# importing pandas as pd

import pandas as pd

# Creating the DataFrame

info = pd.DataFrame({"A":[8, 2, 7, None, 6],

"B":[4, 3, None, 9, 2],

"C":[17, 42, 35, 18, 24],

"D":[15, 18, None, 11, 12]})

# Create the index

index\_ = ['Row1', 'Row2', 'Row3', 'Row4', 'Row5']

# Set the index

info.index = index\_

# Print the DataFrame

print(info)

# return the transpose

result = info.transpose()

# Print the result

print(result)

Output

A B C D

Row\_1 8.0 4.0 17 15.0

Row\_2 2.0 3.0 42 18.0

Row\_3 7.0 NaN 35 NaN

Row\_4 NaN 9.0 18 11.0

Row\_5 6.0 2.0 24 12.0

Row1 Row2 Row3 Row4 Row5

A 8.0 2.0 7.0 NaN 6.0

B 4.0 3.0 NaN 9.0 2.0

C 17.0 42.0 35.0 18.0 24.0

D 15.0 18.0 NaN 11.0 12.0

**Add new columns in a DataFrame using insert()**

We can add a new column at any position in an existing DataFrame using a method name insert.

For the demonstration, first, we have to write a code to read the existing file that consists of some columns in a DataFrame.

import pandas as pd

aa = pd.read\_csv("aa.csv")

aa.head()s

The above code read the existing csv file and shown the data values column in the output.

Output

Name Hire Date Salary Leaves Remaining

0 John Idle 03/15/14 50000.0 10

1 Smith Gilliam 06/01/15 65000.0 8

2 Parker Chapman 05/12/14 45000.0 10

3 Jones Palin 11/01/13 70000.0 3

4 Terry Gilliam 08/12/14 48000.0 7

5 Michael Palin 05/23/13 66000.0 8

Let's add a new column name "Department" into an existing "aa" csv file using insert method.

import pandas as pd

aa = pd.read\_csv("aa.csv")

aa.insert(2, column = "Department", value = "B.Sc")

aa.head()

Output

Name Hire Date Department Salary Leaves Remaining

0 John Idle 03/15/14 B.Sc 50000.0 10

1 Smith Gilliam 06/01/15 B.Sc 65000.0 8

2 Parker Chapman 05/12/14 B.Sc 45000.0 10

3 Jones Palin 11/01/13 B.Sc 70000.0 3

4 Terry Gilliam 08/12/14 B.Sc 48000.0 7

5 Michael Palin 05/23/13 B.Sc 66000.0 8

**Pandas DataFrame.join():**

When we want to concatenate our DataFrames, we can add them with each other by stacking them either vertically or side by side. Another method to combine these DataFrames is to use columns in each dataset that contain common values. The method of combining the DataFrame using common fields is called "joining". The method that we use for combining the DataFrame is a join() method. The columns that contain common values are called "join key".

The join() method is often useful when one DataFrame is a lookup table that contains additional data added into the other DataFrame. It is a convenient method that can combine the columns of two differently-indexed DataFrames into a single DataFrame.

**Identifying join keys**

To determine the appropriate join keys, first, we have to define required fields that are shared between the DataFrames. Both the DataFrames consist of the columns that have the same name and also contain the same data.

**Inner joins**

Inner join can be defined as the most commonly used join. Basically, its main task is to combine the two DataFrames based on a join key and returns a new DataFrame. The returned DataFrame consists of only selected rows that have matching values in both of the original DataFrame.

**Left joins**

If we want to add some information into the DataFrame without losing any of the data, we can simply do it through a different type of join called a "left outer join" or "left join".

Like an inner join, left join also uses the join keys to combine two DataFrames, but unlike inner join, it returns all of the rows from the left DataFrame, even those rows whose join keys do not include the values in the right DataFrame.

Syntax:

DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)

Parameters:

other: It refers to the DataFrame or Series.

In this case, the index should be similar to one of the columns. If we pass a Series, the named attribute has to be set for using it as the column name in the resulting joined DataFrame.

on: It is an optional parameter that refers to array-like or str values.

**It refers to a column or index level name in the caller to join on the index. Otherwise, it joins index-on-index**. If multiple values are present, then the other DataFrame must have MultiIndex. It is like an Excel VLOOKUP operation that can pass an array as the join key if it is not already contained within the calling DataFrame.

how: It refers to 'left', 'right', 'outer', 'inner' values that mainly work on how to handle the operation of the two objects. The default value of how is left.

left: It uses a calling frame's index or column if the parameter on is specified.

right: It uses the other index.

outer: It is used to form a union of calling frame's index or column if parameter on is specified with other's index, and also sort it lexicographically.

inner: It is used to form an intersection of calling frame's index or column if parameter on is specified with other's index. So, due to this, it preserves the order of the calling object.

lsuffix: It refers to a string object that has the default value ''. It uses the Suffix from the left frame's overlapping columns.

rsuffix: It refers to a string value, that has the default value ''. It uses the Suffix from the right frame's overlapping columns.

sort: It consists of a boolean value that sorts the resulting DataFrame lexicographically by the join key. If we pass False value, then the order of the join key mainly depends on the join type, i.e., how.

Example: The below example shows the working of join() function.

import pandas as pd

info = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],

'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})

x = pd.DataFrame({'key': ['K0', 'K1', 'K2'],

'B': ['B0', 'B1', 'B2']})

info.join(x, lsuffix='\_caller', rsuffix='\_x')

info.set\_index('key').join(x.set\_index('key'))

info.join(x.set\_index('key'), on='key')

Output:

key A B

0 K0 A0 B0

1 K1 A1 B1

2 K2 A2 B2

3 K3 A3 NaN

4 K4 A4 NaN

5 K5 A5 NaN

Example2: The below example joins the two MultiIndexes:

import pandas as pd

leftindex = pd.MultiIndex.from\_product([list('xyz'), list('pq'), [1, 2]], names=['xyz', 'pq', 'num'])

left = pd.DataFrame({'value': range(12)}, index=leftindex)

print(left)

Output:

value

xyz pq num

x p 1 0

2 1

q 1 2

2 3

y p 1 4

2 5

q 1 6

2 7

z p 1 8

2 9

q 1 10

2 11

**Pandas DataFrame.merge()**

Pandas merge() is defined as the process of bringing the two datasets together into one and aligning the rows based on the common attributes or columns. It is an entry point for all standard database join operations between DataFrame objects:

Syntax:

pd.merge(left, right, how='inner', on=None, left\_on=None, right\_on=None,

left\_index=False, right\_index=False, sort=True)

Parameters:

right: DataFrame or named Series

It is an object which merges with the DataFrame.

how: {'left', 'right', 'outer', 'inner'}, default 'inner'

Type of merge to be performed.

left: It use only keys from the left frame, similar to a SQL left outer join; preserve key order.

right: It use only keys from the right frame, similar to a SQL right outer join; preserve key order.

outer: It used the union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.

inner: It use the intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.

on: label or list

It is a column or index level names to join on. It must be found in both the left and right DataFrames. If on is None and not merging on indexes, then this defaults to the intersection of the columns in both DataFrames.

left\_on: label or list, or array-like

It is a column or index level names from the left DataFrame to use as a key. It can be an array with length equal to the length of the DataFrame.

right\_on: label or list, or array-like

It is a column or index level names from the right DataFrame to use as keys. It can be an array with length equal to the length of the DataFrame.

left\_index : bool, default False

It uses the index from the left DataFrame as the join key(s), If true. In the case of MultiIndex (hierarchical), many keys in the other DataFrame (either the index or some columns) should match the number of levels.

right\_index : bool, default False

It uses the index from the right DataFrame as the join key. It has the same usage as the left\_index.

sort: bool, default False

If True, it sorts the join keys in lexicographical order in the result DataFrame. Otherwise, the order of the join keys depends on the join type (how keyword).

suffixes: tuple of the (str, str), default ('\_x', '\_y')

It suffixes to apply to overlap the column names in the left and right DataFrame, respectively. The columns use (False, False) values to raise an exception on overlapping.

copy: bool, default True

If True, it returns a copy of the DataFrame.

Otherwise, It can avoid the copy.

indicator: bool or str, default False

If True, It adds a column to output DataFrame "\_merge" with information on the source of each row. If it is a string, a column with information on the source of each row will be added to output DataFrame, and the column will be named value of a string. The information column is defined as a categorical-type and it takes value of:

"left\_only" for the observations whose merge key appears only in 'left' of the DataFrame, whereas,

"right\_only" is defined for observations in which merge key appears only in 'right' of the DataFrame,

"both" if the observation's merge key is found in both of them.

validate: str, optional

If it is specified, it checks the merge type that is given below:

"one\_to\_one" or "1:1": It checks if merge keys are unique in both the left and right datasets.

"one\_to\_many" or "1:m": It checks if merge keys are unique in only the left dataset.

"many\_to\_one" or "m:1": It checks if merge keys are unique in only the right dataset.

"many\_to\_many" or "m:m": It is allowed, but does not result in checks.

Example1: Merge two DataFrames on a key

# import the pandas library

import pandas as pd

left = pd.DataFrame({

'id':[1,2,3,4],

'Name': ['John', 'Parker', 'Smith', 'Parker'],

'subject\_id':['sub1','sub2','sub4','sub6']})

right = pd.DataFrame({

'id':[1,2,3,4],

'Name': ['William', 'Albert', 'Tony', 'Allen'],

'subject\_id':['sub2','sub4','sub3','sub6']})

print (left)

print (right)

Output

id Name subject\_id

0 1 John sub1

1 2 Parker sub2

2 3 Smith sub4

3 4 Parker sub6

id Name subject\_id

0 1 William sub2

1 2 Albert sub4

2 3 Tony sub3

3 4 Allen sub6

Example2: Merge two DataFrames on multiple keys:

import pandas as pd

left = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],

'subject\_id':['sub1','sub2','sub4','sub6','sub5']})

right = pd.DataFrame({

'id':[1,2,3,4,5],

'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],

'subject\_id':['sub2','sub4','sub3','sub6','sub5']})

print pd.merge(left,right,on='id')

Output

id Name\_x subject\_id\_x Name\_y subject\_id\_y

0 1 John sub1 William sub2

1 2 Parker sub2 Albert sub4

2 3 Smith sub4 Tony sub3

3 4 Parker sub6 Allen sub6

**Pandas DataFrame.pivot\_table()**

The Pandas pivot\_table() is used to calculate, aggregate, and summarize your data. It is defined as a powerful tool that aggregates data with calculations such as Sum, Count, Average, Max, and Min.

Syntax:

pandas.pivot\_table(data, values=None, index=None, columns=None, aggfunc='mean', fill\_value=None, margins=False, dropna=True, margins\_name='All', observed=False)[source]

Parameters:

Data: DataFrame

Values: column to aggregate, optional

Index: column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

Columns: column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

Aggfunc: function, list of functions, dict, default numpy.mean

If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions.

fill\_values: calar, default None

Value to replace missing values with (in the resulting pivot table, after aggregation).

Margins: bool, default False

Add all row / columns (e.g. for subtotal / grand totals).

Dropna: bool, default True

Do not include columns whose entries are all NaN.

margins\_name: str, default ‘All’

Name of the row / column that will contain the totals when margins is True.

Observed: bool, default False

This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

Returns: DataFrame

An Excel style pivot table.

Examples:

df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",

"bar", "bar", "bar", "bar"],

"B": ["one", "one", "one", "two", "two",

"one", "one", "two", "two"],

"C": ["small", "large", "large", "small",

"small", "large", "small", "small",

"large"],

"D": [1, 2, 2, 3, 3, 4, 5, 6, 7],

"E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})

print(df)

Output:

A B C D E

0 foo one small 1 2

1 foo one large 2 4

2 foo one large 2 5

3 foo two small 3 5

4 foo two small 3 6

5 bar one large 4 6

6 bar one small 5 8

7 bar two small 6 9

8 bar two large 7 9

Example:

table = pd.pivot\_table(df, values='D', index=['A', 'B'],

columns=['C'], aggfunc=np.sum)

print(table)

Output:

C large small

A B

bar one 4.0 5.0

two 7.0 6.0

foo one 4.0 1.0

two NaN 6.0

Example:

table = pd.pivot\_table(df, values='D', index=['A', 'B'],

columns=['C'], aggfunc=np.sum, fill\_value=0)

print(table)

Output:

C large small

A B

bar one 4 5

two 7 6

foo one 4 1

two 0 6

Example:

table = pd.pivot\_table(df, values=['D', 'E'], index=['A', 'C'],

aggfunc={'D': np.mean,

'E': np.mean})

Print(table)

Output:

D E

A C

bar large 5.500000 7.500000

small 5.500000 8.500000

foo large 2.000000 4.500000

small 2.333333 4.333333

Example:

table = pd.pivot\_table(df, values=['D', 'E'], index=['A', 'C'],

aggfunc={'D': np.mean,

'E': [min, max, np.mean]})

Print(table)

Output:

D E

mean max mean min

A C

bar large 5.500000 9.0 7.500000 6.0

small 5.500000 9.0 8.500000 8.0

foo large 2.000000 5.0 4.500000 4.0

small 2.333333 6.0 4.333333 2.0

Example:

import pandas as pd

import numpy as np

df4=pd.read\_excel(r"D:\Brajesh\Anurag\New folder\archive\Raw\_Data.xlsx")

table = pd.pivot\_table(df4, index =['Client'],values=['Amount Requested'], aggfunc=np.sum)

print(table)

**Pandas melt()**

To make analysis of data in table easier, we can reshape the data into a more computer-friendly form using Pandas in Python. Pandas.melt() is one of the function to do so..

Pandas.melt() unpivots a DataFrame from wide format to long format.

melt() function is useful to massage a DataFrame into a format where one or more columns are identifier variables, while all other columns, considered measured variables, are unpivoted to the row axis, leaving just two non-identifier columns, variable and value.

Syntax :

pandas.melt(frame, id\_vars=None, value\_vars=None, var\_name=None, value\_name='value', col\_level=None)

Parameters:

frame : DataFrame

id\_vars[tuple, list, or ndarray, optional] : Column(s) to use as identifier variables.

value\_vars[tuple, list, or ndarray, optional]: Column(s) to unpivot. If not specified, uses all columns that are not set as id\_vars.

var\_name[scalar]: Name to use for the ‘variable’ column. If None it uses frame.columns.name or ‘variable’.

value\_name[scalar, default ‘value’]: Name to use for the ‘value’ column.

col\_level[int or string, optional]: If columns are a MultiIndex then use this level to melt.

Example:

# Create a simple dataframe

# importing pandas as pd

import pandas as pd

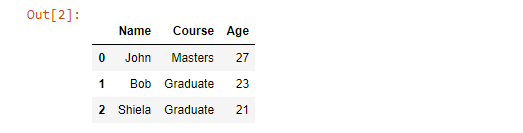
# creating a dataframe

df = pd.DataFrame({'Name': {0: 'John', 1: 'Bob', 2: 'Shiela'},

'Course': {0: 'Masters', 1: 'Graduate', 2: 'Graduate'},

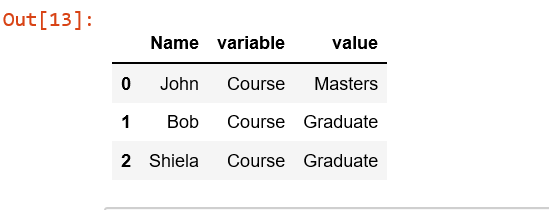
'Age': {0: 27, 1: 23, 2: 21}})

Print(df)



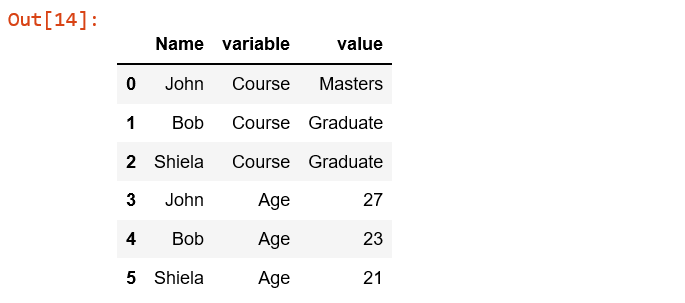
# Name is id\_vars and Course is value\_vars

pd.melt(df, id\_vars =['Name'], value\_vars =['Course'])



# multiple unpivot columns

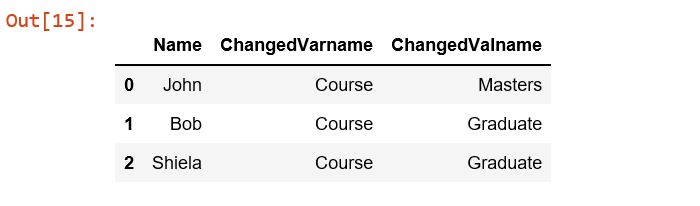
pd.melt(df, id\_vars =['Name'], value\_vars =['Course', 'Age'])



# Names of ‘variable’ and ‘value’ columns can be customized

pd.melt(df, id\_vars =['Name'], value\_vars =['Course'],

var\_name ='ChangedVarname', value\_name ='ChangedValname')



**Pandas DataFrame.rename()**

The main task of the Pandas rename() function is to rename any index, column, or row. This method is useful for renaming some selected columns because we have to specify the information only for those columns that we want to rename.

It mainly alters the axes labels based on some of the mapping (dict or Series) or the arbitrary function. The function must be unique and should range from 1 to -1. The labels will be left, if it is not contained in a dict or Series. If you list some extra labels, it will throw an error.

Syntax:

DataFrame.rename(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False, level=None, errors='ignore')

Parameters:

mapper: It is a dict-like or function transformation that is to be applied to a particular axis label. We can use either mapper or axis to specify the axis targeted with mapper, index, and

index: It is an alternative of specifying the axis (mapper, axis =0 is equivalent to the index=mapper).

columns: It is an alternative to specify an axis (mapper, axis =1 is equivalent to the columns=mapper).

axis: It refers to an int or str value that defines the axis targeted with the mapper. It can be either the axis name ('index', 'columns') or the number.

copy: It refers to a boolean value that copies the underlying data. The default value of the copy is True.

inplace: It refers to a boolean value and checks whether to return the new DataFrame or not. If it is true, it makes the changes in the original DataFrame. The default value of the inplace is True.

level: It refers to an int or level name values that specify the level, if DataFrame has a multiple level index. The default value of the level is None.

errors: It refers to ignore, raise If we specify raise value, it raises a KeyError if any of the labels are not found in the selected axis.

Returns:

It returns the DataFrame with renamed axis labels.

Example 1: The below example renames a single column:

import pandas as pd

# Define a dictionary containing information of employees

info = {'name': ['Parker', 'Smith', 'William', 'Robert'], 'age': [38, 47, 44, 34],

'language': ['Java', 'Python', 'JavaScript', 'Python']}

# Convert dictionary into DataFrame

info\_pd = pd.DataFrame(info)

# Before renaming columns

print(info\_pd)

info\_pd.rename(columns = {'name':'Name'}, inplace = True)

# After renaming columns

print("\nAfter modifying first column:\n", info\_pd.columns

Output:

name age language

0 Parker 38 Java

1 Smith 47 Python

2 William 44 JavaScript

3 Robert 34 Python

After modifying first column:

Index(['Name', 'age', 'language'], dtype='object')

Example2: The below example renames the multiple columns:

import pandas as pd

# Define a dictionary containing information of employees

info = {'name': ['Parker', 'Smith', 'William', 'Robert'],

'age': [38, 47, 44, 34],

'language': ['Java', 'Python', 'JavaScript', 'Python']}

# Convert dictionary into DataFrame

info\_pd = pd.DataFrame(info)

# Before renaming columns

print(info\_pd)

info\_pd.rename(columns = {'name':'Name', 'age':'Age', 'language':'Language'}, inplace = True)

# After renaming columns

print(info\_pd.columns)

Output:

name age language

0 Parker 38 Java

1 Smith 47 Python

2 William 44 JavaScript

3 Robert 34 Python

Index(['Name', 'Age', 'Language'], dtype='object')

Example3: The below example renames indexes of a particular column:

import pandas as pd

data = {'Name': ['Smith', 'Parker', 'William'], 'Emp\_ID': [101, 102, 103], 'Language': ['Python', 'Java', 'JavaScript']}

info1 = pd.DataFrame(data)

print('DataFrame:\n', info1)

info2 = info.rename(index={0: '#0', 1: '#1', 2: '#2'})

print('Renamed Indexes:\n', info2)

Output:

DataFrame:

Name Emp\_ID Language

0 Smith 101 Python

1 Parker 102 Java

2 William 103 JavaScript

Renamed Indexes:

Name Emp\_ID Language

#0 Smith 101 Python

#1 Parker 102 Java

#2 William 103 JavaScript

**Pandas DataFrame.shift()**

If you want to shift your column or subtract the column value with the previous row value from the DataFrame, you can do it by using the shift() function. It consists of a scalar parameter called period, which is responsible for showing the number of shifts to be made over the desired axis. It is also capable of dealing with time-series data.

Syntax:

DataFrame.shift(periods=1, freq=None, axis=0)

Parameters:

periods: It consists of an integer value that can be positive or negative. It defines the number of periods to move.

freq: It can be used with DateOffset, tseries module, str or time rule (e.g., 'EOM').

axis: 0 is used for shifting the index, whereas 1 is used for shifting the column.

fill\_value: Used for filling newly missing values.

Returns

It returns a shifted copy of DataFrame.

Example1: The below example demonstrates the working of the shift().

import pandas as pd

info= pd.DataFrame({'a\_data': [45, 28, 39, 32, 18],

'b\_data': [26, 37, 41, 35, 45],

'c\_data': [22, 19, 11, 25, 16]})

info.shift(periods=2)

Output

a\_data b\_data c\_data

0 NaN NaN NaN

1 NaN NaN NaN

2 45.0 26.0 22.0

3 28.0 37.0 19.0

4 39.0 41.0 11.0

Example2: The example shows how to fill the missing values in the DataFrame using the fill\_value.

import pandas as pd

info= pd.DataFrame({'a\_data': [45, 28, 39, 32, 18],

'b\_data': [26, 38, 41, 35, 45],

'c\_data': [22, 19, 11, 25, 16]})

info.shift(periods=2)

info.shift(periods=2,axis=1,fill\_value= 70)

Output

a\_data b\_data c\_data

0 70 70 45

1 70 70 28

2 70 70 39

3 70 70 32

4 70 70 18

**Pandas DataFrame.sort()**

Parameters

columns: Before Sorting, you have to pass an object or the column names.

ascending: A Boolean value is passed that is responsible for sorting in the ascending order. Its default value is True.

axis: 0 or index; 1 or 'columns'. The default value is 0. It decides whether you sort by index or columns.

inplace: A Boolean value is passed. The default value is false. It will modify any other views on this object and does not create a new instance while sorting the DataFrame.

kind: 'heapsort', 'mergesort', 'quicksort'. It is an optional parameter that is to be applied only when you sort a single column or labels.

na\_position: 'first', 'last'. The 'first' puts NaNs at the beginning, while the 'last' puts NaNs at the end. Default option last.

We can efficiently perform sorting in the DataFrame through different kinds:

By label

By Actual value

Before explaining these two kinds of sorting, first we have to take the dataset for demonstration:

import pandas as pd

import numpy as np

info=pd.DataFrame(np.random.randn(10,2),index=[1,3,7,2,4,5,9,8,0,6],columns=['col2','col1'])

print(info)

Output

col2 col1

1 -0.456763 -0.931156

3 0.242766 -0.793590

7 1.133803 0.454363

2 -0.843520 -0.938268

4 -0.018571 -0.315972

5 -1.951544 -1.300100

9 -0.711499 0.031491

8 1.648080 0.695637

0 2.576250 -0.625171

6 -0.301717 0.879970

In the above DataFrame, the labels and the values are unsorted. So, let's see how it can be sorted:

**By label**

The DataFrame can be sorted by using the sort\_index() method. It can be done by passing the axis arguments and the order of sorting. The sorting is done on row labels in ascending order by default.

Example

import pandas as pd

import numpy as np

info=pd.DataFrame(np.random.randn(10,2),index=[1,2,5,4,8,7,9,3,0,6],columns = ['col4','col3'])

info2=info.sort\_index()

print(info2)

Output

col4 col3

0 0.698346 1.897573

1 1.247655 -1.208908

2 -0.469820 -0.546918

3 -0.793445 0.362020

4 -1.184855 -1.596489

5 1.500156 -0.397635

6 -1.239635 -0.255545

7 1.110986 -0.681728

8 -1.797474 0.108840

9 0.063048 1.512421

Order of Sorting

The order of sorting can be controlled by passing the Boolean value to the ascending parameter.

Example:

import pandas as pd

import numpy as np

info= pd.DataFrame(np.random.randn(10,2),index=[1,4,7,2,5,3,0,8,9,6],columns = ['col4','col5'])

info\_2 = info.sort\_index(ascending=False)

print(info)

Output

col4 col5

1 0.664336 -1.846533

4 -0.456203 -1.255311

7 0.537063 -0.774384

2 -1.937455 0.257315

5 0.331764 -0.741020

3 -0.082334 0.304390

0 -0.983810 -0.711582

8 0.208479 -1.234640

9 0.656063 0.122720

6 0.347990 -0.410401

**Sort the Columns**:

We can sort the columns labels by passing the axis argument respected to its values 0 or 1. By default, the axis=0, it sort by row.

Example:

import pandas as pd

import numpy as np

info = pd.DataFrame(np.random.randn(10,2),index=[1,4,8,2,0,6,7,5,3,9],columns = ['col7','col4'])

info\_2=info.sort\_index(axis=1)

print(info\_2)

Output

col4 col7

1 0.197021 0.876053

4 -0.803485 1.116486

8 2.783560 0.306012

2 0.417733 -0.809810

0 1.400857 -1.780468

6 -0.523755 0.735315

7 0.659082 -0.436512

5 -0.616561 0.757169

3 0.121360 -0.042020

9 -0.238969 0.234543

**By Actual Value**

It is another kind through which sorting can be performed in the DataFrame. Like index sorting, sort\_values() is a method for sorting by the values.

It also provides a feature in which we can specify the column name of the DataFrame with which values are to be sorted. It is done by passing the 'by' argument.

Example:

import pandas as pd

import numpy as np

info = pd.DataFrame({'col1':[7,1,8,3],'col2':[8,12,4,9]})

info\_2 = info.sort\_values(by='col2')

print(info\_2)

Output

col1 col2

2 8 4

0 7 8

3 3 9

1 1 12

In the above output, observe that the values are sorted in col2 only, and the respective col1 value and row index will alter along with col2. Thus, they look unsorted.

**Pandas DataFrame.astype()**

DataFrame.astype() method is used to cast a pandas object to a specified dtype. astype() function also provides the capability to convert any suitable existing column to categorical type.

DataFrame.astype() function comes very handy when we want to case a particular column data type to another data type. Not only that but we can also use a Python dictionary input to change more than one column type at once. The key label in dictionary is corresponding to the column name and the values label in the dictionary is corresponding to the new data types we want the columns to be of.

Syntax: DataFrame.astype(dtype, copy=True, errors=’raise’, \*\*kwargs)

Parameters:

dtype : Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, …}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame’s columns to column-specific types.

copy : Return a copy when copy=True (be very careful setting copy=False as changes to values then may propagate to other pandas objects).

errors : Control raising of exceptions on invalid data for provided dtype.

raise : allow exceptions to be raised

ignore : suppress exceptions. On error return original object

kwargs :keyword arguments to pass on to the constructor

Returns: casted : type of caller

Example #1: Convert the Weight column data type.

# importing pandas as pd

import pandas as pd

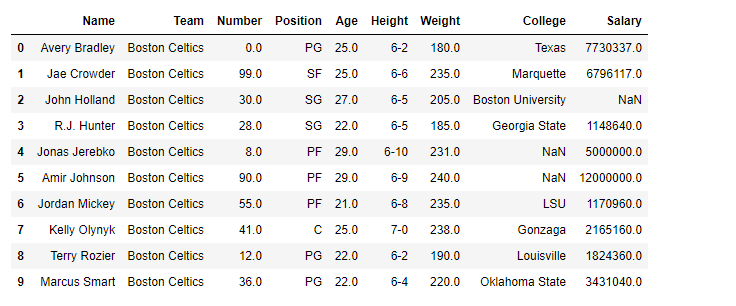
# Making data frame from the csv file

df = pd.read\_csv("nba.csv")

# Printing the first 10 rows of

# the data frame for visualization

Print(df[:10])

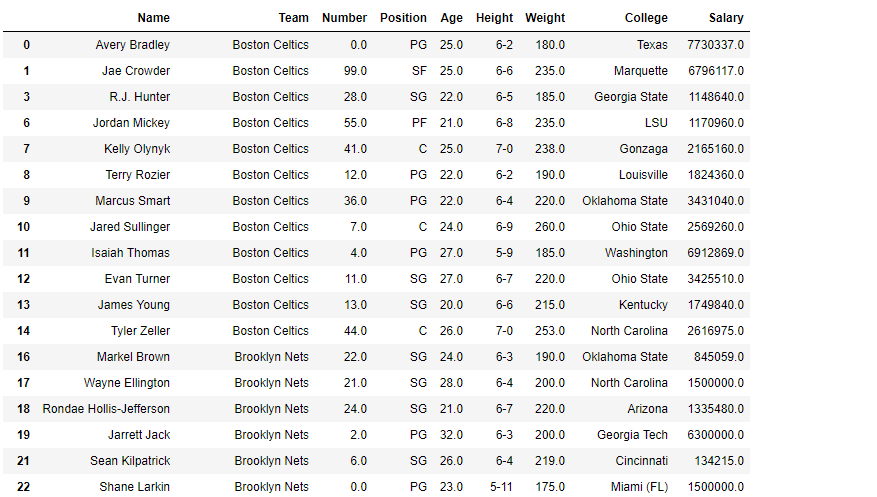


As the data have some “nan” values so, to avoid any error we will drop all the rows containing any nan values.

# drop all those rows which

# have any 'nan' value in it.

df.dropna(inplace = True)



# let's find out the data type of Weight column

before = type(df.Weight[0])

# Now we will convert it into 'int64' type.

df.Weight = df.Weight.astype('int64')

# let's find out the data type after casting

after = type(df.Weight[0])

# print the value of before

before

# print the value of after

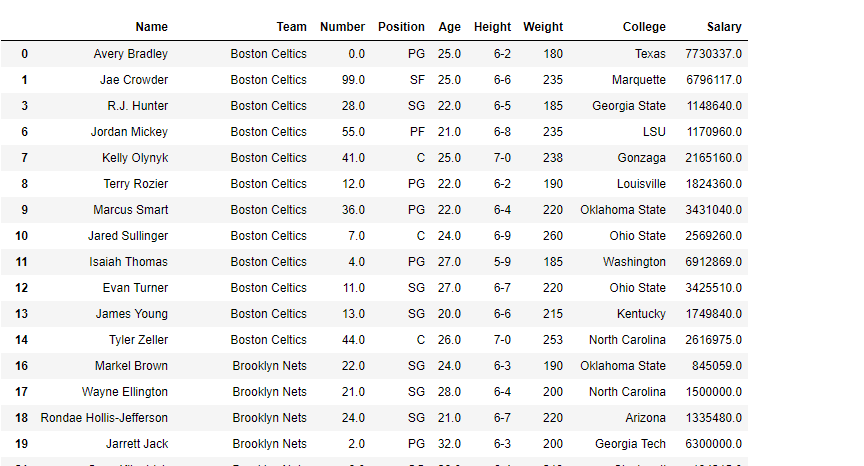
After

# print the data frame and see

# what it looks like after the change

Df

Output:



Example #2: Change the data type of more than one column at once

Change the Name column to categorical type and Age column to int64 type.

# importing pandas as pd

import pandas as pd

# Making data frame from the csv file

df = pd.read\_csv("nba.csv")

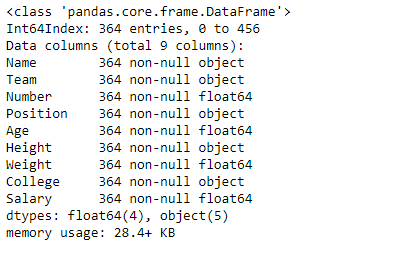
# Drop the rows with 'nan' values

df = df.dropna()

# print the existing data type of each column

df.info()

Output:



Now let’s change both the columns data type at once.

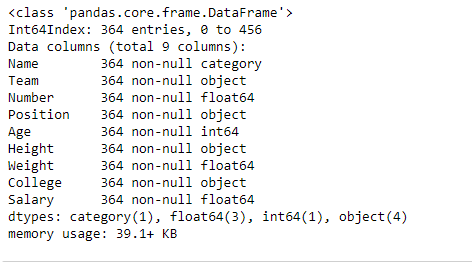
# Passed a dictionary to astype() function

df = df.astype({"Name":'category', "Age":'int64'})

# Now print the data type

# of all columns after change

df.info()



# print the data frame

# too after the change

Df



**Pandas DataFrame.cut()**

The cut() method is invoked when you need to segment and sort the data values into bins. It is used to convert a continuous variable to a categorical variable. It can also segregate an array of elements into separate bins. The method only works for the one-dimensional array-like objects.

If we have a large set of scalar data and perform some statistical analysis on it, we can use the cut() method.

Syntax:

pandas.cut(x, bins, right=True, labels=None, retbins=False, precision=3, include\_lowest=False, duplicates='raise')

Parameters:

x: It generally refers to an array as an input that is to be bin. The array should be a one-dimensional array.

bins: It refers to an int, sequence of scalars, or IntervalIndex values that define the bin edges for the segmentation. Most of the time, we have numerical data on a very large scale. So, we can group the values into bins to easily perform descriptive statistics as a generalization of patterns in data. The criteria for binning the data into groups are as follows:

* int: It defines the number of equal-width bins that are in the range of x. We can also extend the range of x by .1% on both sides to include the minimum and maximum values of x.
* sequence of scalars: It mainly defines the bin edges that are allowed for non-uniform width.
* IntervalIndex: It refers to an exact bin that is to be used in the function. It should be noted that the IntervalIndex for bins must be non-overlapping.
* right: It consists of a boolean value that checks whether the bins include the rightmost edge or not. Its default value is True, and it is ignored when bins is an
* labels: It is an optional parameter that mainly refers to an array or a boolean value. Its main task is to specify the labels for the returned The length of the labels must be the same as the resulting bins. If we set its value to False, it returns only integer indicator of the bins. This argument is ignored if bins is an IntervalIndex.
* retbins: It refers to a boolean value that checks whether to return the bins or not. It is often useful when bins are provided as a scalar value. The default value of retbins is False.
* precision: It is used to store and display the bins labels. It consists of an integer value that has the default value 3.
* include\_lowest: It consists of a boolean value that is used to check whether the first interval should be left-inclusive or not.
* duplicates: It is an optional parameter that decides whether to raise a ValueError or drop duplicate values if the bin edges are not unique.

Returns:

This method returns two objects as output which are as follows:

1. out: It mainly refers to a Categorical, Series, or ndarray that is an array-like object which represents the respective bin for each value of These objects depend on the value of labels. The possible values than can be returned are as follows:

* True: It is a default value that returns a Series or a Categorical variable. The values stored in these objects are Interval data type.
* sequence of scalars: It also returns a Series or a Categorical variable. The values that are stored in these objects are the type of the sequence.
* False: The false value returns an ndarray of integers.

1. bins: It mainly refers to a ndarray

Example1: The below example segments the numbers into bins:

import pandas as pd

import numpy as np

info\_nums = pd.DataFrame({'num': np.random.randint(1, 50, 11)})

print(info\_nums)

info\_nums['num\_bins'] = pd.cut(x=info\_nums['num'], bins=[1, 25, 50])

print(info\_nums)

print(info\_nums['num\_bins'].unique())

Output:

num

0 27

1 41

2 25

3 9

4 45

5 6

6 5

7 7

8 38

9 13

10 35

num num\_bins

0 27 (25, 50]

1 41 (25, 50]

2 25 (1, 25]

3 9 (1, 25]

4 45 (25, 50]

5 6 (1, 25]

6 5 (1, 25]

7 7 (1, 25]

8 38 (25, 50]

9 13 (1, 25]

10 35 (25, 50]

[(25, 50], (1, 25]]

Categories (2, interval[int64]): [(1, 25] < (25, 50]]

Example2: The below example shows how to add labels to bins:

import pandas as pd

import numpy as np

info\_nums = pd.DataFrame({'num': np.random.randint(1, 10, 7)})

print(info\_nums)

info\_nums['nums\_labels'] = pd.cut(x=info\_nums['num'], bins=[1, 7, 10], labels=['Lows', 'Highs'], right=False)

print(info\_nums)

print(info\_nums['nums\_labels'].unique())

Output:

num

0 6

1 9

2 2

3 6

4 3

5 6

6 4

num nums\_labels

0 6 Lows

1 9 Highs

2 2 Lows

3 6 Lows

4 3 Lows

5 6 Lows

6 4 Lows

['Lows', 'Highs']

Categories (2, object): ['Lows' < 'Highs']

**Pandas Dataframe.sample()**

Pandas sample() is used to generate a sample random row or column from the function caller data frame.

Syntax

DataFrame.sample(n=None, frac=None, replace=False, weights=None, random\_state=None, axis=None)

Parameters

n: int value, Number of random rows to generate.

frac: Float value, Returns (float value \* length of data frame values ). frac cannot be used with n.

replace: Boolean value, return sample with replacement if True.

random\_state: int value or numpy.random.RandomState, optional. if set to a particular integer, will return same rows as sample in every iteration.

axis: 0 or ‘row’ for Rows and 1 or ‘column’ for Columns.

Returns

It returns a new object of the same type as a caller that contains n items randomly sampled from the caller object.

Example #1: Random row from Data frame

In this example, two random rows are generated by the .sample() method and compared later.

# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read\_csv("employees.csv")

# generating one row

row1 = data.sample(n = 1)

# display

row1

# generating another row

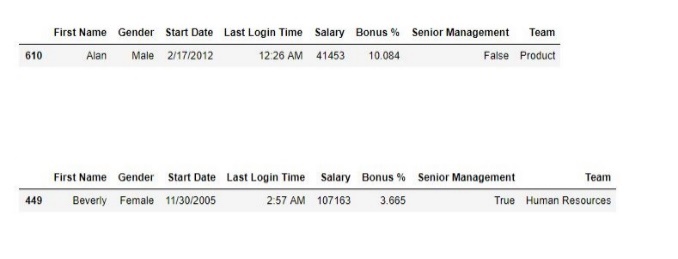
row2 = data.sample(n = 1)

# display

row2

Output:

As shown in the output image, the two random sample rows generated are different from each other.



Example #2: Generating 25% sample of data frame

In this example, 25% random sample data is generated out of the Data frame.

# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read\_csv("employees.csv")

# generating one row

rows = data.sample(frac =.25)

# checking if sample is 0.25 times data or not

if (0.25\*(len(data))== len(rows)):

print( "Cool")

print(len(data), len(rows))

# display

rows

Output:

As shown in the output image, the length of sample generated is 25% of data frame. Also the sample is generated randomly.



**Pandas DataFrame.to\_excel()**

We can export the DataFrame to the excel file by using the to\_excel() function.

To write a single object to the excel file, we have to specify the target file name. If we want to write to multiple sheets, we need to create an ExcelWriter object with target filename and also need to specify the sheet in the file in which we have to write.

The multiple sheets can also be written by specifying the unique sheet\_name. It is necessary to save the changes for all the data written to the file.

Note: If we create an ExcelWriter object with a file name that already exists, it will erase the content of the existing file.

Syntax

DataFrame.to\_excel(excel\_writer, sheet\_name='Sheet1', na\_rep='', float\_format=None, columns=None, header=True, index=True, index\_label=None, startrow=0, startcol=0, engine=None, merge\_cells=True, encoding=None, inf\_rep='inf', verbose=True, freeze\_panes=None)

Parameters

excel\_writer: A file path or existing ExcelWriter.

sheet\_name: It refers to the name of the sheet that contains the DataFrame.

na\_repr: Missing Data representation.

float\_format: It is an optional parameter that formats the string for floating-point numbers.

columns: Refers the column to write.

header: It writes out the column names. If a list of the string is given, it is assumed to be the aliases for the column names.

index: It writes the index.

index\_label: Refers to the column label for the index column. If it is not specified, and the header and index are True, then the index names are used. If DataFrame uses MultiIndex, a sequence should be given.

startrow: Default value 0. It refers to the upper left cell row to dump the DataFrame.

startcol: Default value 0. It refers to the upper left cell column to dump the DataFrame.

engine: It is an optional parameter that writes the engine to use, openpyxl, or xlsxwriter.

merge\_cells: It returns the boolean value and its default value is True. It writes MultiIndex and Hierarchical rows as the merged cells.

encoding: It is an optional parameter that encodes the resulting excel file. It is only necessary for the xlwt.

inf\_rep: It is also an optional parameter and its default value is inf. It usually represents infinity.

verbose: It returns a boolean value. It's default value is True.

It is used to display more information in the error logs.

freeze\_panes: It is also an optional parameter that specifies the one based bottommost row and rightmost column that is to be frozen.

Example

import pandas as pd

# create dataframe

info\_marks = pd.DataFrame({'name': ['Parker', 'Smith', 'William', 'Terry'],

'Maths': [78, 84, 67, 72],

'Science': [89, 92, 61, 77],

'English': [72, 75, 64, 82]})

# render dataframe as html

writer = pd.ExcelWriter('output.xlsx')

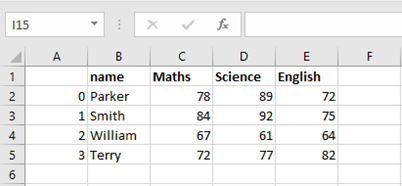
info\_marks.to\_excel(writer)

writer.save()

print('DataFrame is written successfully to the Excel File.')

Output

DataFrame is written successfully to the Excel file



**Convert Pandas DataFrame to Numpy array**

For performing some high-level mathematical functions, we can convert Pandas DataFrame to numpy arrays. It uses the DataFrame.to\_numpy() function.

The DataFrame.to\_numpy() function is applied on the DataFrame that returns the numpy ndarray.

Syntax

DataFrame.to\_numpy(dtype=None, copy=False)

Parameters

dtype: It is an optional parameter that pass the dtype to numpy.asarray().

copy: It returns the boolean value that has the default value False.

It ensures that the returned value is not a view on another array.

Returns

It returns the numpy.ndarray as an output.

Example1

import pandas as pd

pd.DataFrame({"P": [2, 3], "Q": [4, 5]}).to\_numpy()

info = pd.DataFrame({"P": [2, 3], "Q": [4.0, 5.8]})

info.to\_numpy()

info['R'] = pd.date\_range('2000', periods=2)

info.to\_numpy()

Output

array([[2, 4.0, Timestamp('2000-01-01 00:00:00')],

[3, 5.8, Timestamp('2000-01-02 00:00:00')]], dtype=object)

Example2

import pandas as pd

#initializing the dataframe

info = pd.DataFrame([[17, 62, 35],[25, 36, 54],[42, 20, 15],[48, 62, 76]],

columns=['x', 'y', 'z'])

print('DataFrame\n----------\n', info)

#convert the dataframe to a numpy array

arr = info.to\_numpy()

print('\nNumpy Array\n----------\n', arr)

Output

DataFrame

----------

x y z

0 17 62 35

1 25 36 54

2 42 20 15

3 48 62 76

Numpy Array

----------

[[17 62 35]

[25 36 54]

[42 20 15]

[48 62 76]]

**Convert Pandas DataFrame to CSV**

The Pandas to\_csv() function is used to convert the DataFrame into CSV data. To write the CSV data into a file, we can simply pass a file object to the function. Otherwise, the CSV data is returned in a string format.

Syntax:

DataFrame.to\_csv(path\_or\_buf=None, sep=', ', na\_rep='', float\_format=None, columns=None, header=True, index=True, index\_label=None, mode='w', encoding=None, compression='infer', quoting=None, quotechar='"', line\_terminator=None, chunksize=None, date\_format=None, doublequote=True, escapechar=None, decimal='.')

Parameters:

path\_or\_buf: It refers to str or file handle. Basically, it defines the path of file or object. The default value is None and if None value is passed, then it returns a string value.

If we pass a file object, it should be opened with newline =" and disable the universal newlines.

sep: It refers to a string value and consists a string of length 1. It's default value is comma(,).

na\_rep: It refers to a string value that represents null or missing values. The empty string is the default value.

float\_format: It also consists a string value that is responsible for formatting a string for the floating-point numbers.

columns: It is an optional parameter that refers to a sequence to specify the columns that need to be included in the CSV output.

header: It generally consists a boolean value or a list of string. If its value is set to False, then the column names are not written in the output. It's default value is True.

If we pass a list of string as an input, it generally writes the column names in the output. The length of the list of the file should be same as number of columns being written in the CSV file.

index: If the value is set to True, index is included in the CSV data. Otherwise, the index value is not written in CSV output.

index\_label: It consists a str value or a sequence that is used to specify the column name for index. It's default value is None.

mode: It refers a string value that is used for writing mode. It's default value is w.

encoding: It is an optional parameter consisting a string value that represents an encoding used in the output file. The default value of encoding is UTF-8.

compression: It refers a str value that compress the mode among the following values{'infer', 'gzip', 'bz2', 'zip', 'xz', None}. It detects the compression from the extensions: '.gz', '.bz2', '.zip' or '.xz' if infer and path\_or\_buf is a path-like, otherwise no compression occurs.

quoting: It is an optional parameter that is defined as a constant from the csv module. Its default value is csv.QUOTE\_MINIMAL. If you set a float\_format then the floating value is converted to strings and csv.QUOTE\_NONNUMERIC is treated as a non-numeric value.

quotechar: It refers to a str value of length 1. It is a character that is used to quote the fields.

line\_terminator: It is an optional parameter that refers to a string value. Its main task is to terminate the line. It is a newline character that is to be used in the output file. Its default value is set to os.linesep that mainly depends on the OS. An individual method is called to define the operating system ('n' for linux, 'rn' for 'Windows').

chunksize: It consists None or integer value and define the rows to write at the current time.

date\_format: It consists str value and used to format a string for the datetime objects. The default value of date\_format is None.

doublequote: It consists a boolean value that have the default value True. It is mainly used for controlling the quote of quotechar inside the field.

escapechar: It consists a string value of length 1. Basically, it is a character that is used to escape sep and quotechar. The default value of escapechar is None.

decimal: It consists a string value that identify a character as decimal separator. Ex: use ',' for European data.

Returns:

It returns str or None value. If a parameter value named as path\_or\_buf is None, it returns the resulting csv format as a string. Otherwise, it returns None.

Example1: The below example convert a DataFrame to CSV String:

import pandas as pd

data = {'Name': ['Smith', 'Parker'], 'ID': [101, 102], 'Language': ['Python', 'JavaScript']}

info = pd.DataFrame(data)

print('DataFrame Values:\n', info)

# default CSV

csv\_data = info.to\_csv()

print('\nCSV String Values:\n', csv\_data)

Output:

DataFrame Values:

Name ID Language

0 Smith 101 Python

1 Parker 102 JavaScript

CSV String Values:

,Name,ID,Language

0 ,Smith,101,Python

1 ,Parker,102,JavaScript

Example2: The below example shows Null or missing Data Representation in the CSV Output file:

import pandas as pd

data = {'Name': ['Smith', 'Parker'], 'ID': [101, pd.NaT], 'Language': [pd.NaT, 'JavaScript']}

info = pd.DataFrame(data)

print('DataFrame Values:\n', info)

csv\_data = info.to\_csv()

print('\nCSV String Values:\n', csv\_data)

csv\_data = info.to\_csv(na\_rep="None")

print('CSV String with Null Data Values:\n', csv\_data)

Output:

DataFrame Values:

Name ID Language

0 Smith 101 NaT

1 Parker NaT JavaScript

CSV String Values:

,Name,ID,Language

0,Smith,101,

1,Parker,,JavaScript

CSV String with Null Data Values:

,Name,ID,Language

0,Smith,101,None

1,Parker,None,JavaScript

Example3: The below example specify the delimiter for the CSV output.

import pandas as pd

data = {'Name': ['Smith', 'Parker'], 'ID': [101, pd.NaT], 'Language': [Python, 'JavaScript']}

info = pd.DataFrame(data)

print('DataFrame:\n', info)

csv\_data = info.to\_csv(sep='|')

print(csv\_data)

Output:

DataFrame:

Name ID Language

0 Smith 101 Python

1 Parker NaT JavaScript

|Name|ID|Language

0|Smith|101|Python

1|Parker||JavaScript

**Python Pandas Reading Files**

**Reading from CSV File**

A csv stands for Comma Separated Values, which is defined as a simple file format that uses specific structuring to arrange tabular data. It stores tabular data such as spreadsheet or database in plain text and has a common format for data interchange. The csv file is opened into the excel file, and the rows and columns data define the standard format.

Reading the csv file into a pandas DataFrame is quick and straight forward. We don't require to write several lines of code to open, analyze, and read the csv file in pandas. Instead, we can perform these operations in a single line, and it stores the data in DataFrame.

For reading the Pandas files, firstly we have to load data from file formats into a DataFrame. You need only a single line to load your data in code.

Name,Hire Date,Salary,Leaves Remaining

John Idle,08/15/14,50000.00,10

Smith Gilliam,04/07/15,65000.00,6

Parker Chapman,02/21/14,45000.00,7

Jones Palin,10/14/13,70000.00,3

Terry Gilliam,07/22/14,48000.00,9

Michael Palin,06/28/13,66000.00,8

df = pd.read\_csv('a.csv')

Code

import pandas

df = pandas.read\_csv('hrdata.csv')

print(df)

In the above, the three lines of code are enough to read the file, and only one of them is doing the actual work, i.e., pandas.read\_csv().

Output:

Name Hire Date Salary Leaves Remaining

0 John Idle 08/15/14 50000.0 10

1 Smith Gilliam 04/07/15 65000.0 8

2 Parker Chapman 02/21/14 45000.0 10

3 Jones Palin 10/14/13 70000.0 3

4 Terry Gilliam 07/22/14 48000.0 7

5 Michael Palin 06/28/13 66000.0 8

However, the pandas are also using the zero-based integer indices in the DataFrame; we didn't tell it what our index should be.

**Reading from JSON**

If you have any JSON file, Pandas can easily read it through a single line of code.

df =pd.read\_json('hrdata.json')

It allowed indexes to work through nesting.

Pandas convert a list of lists into a DataFrame and also define the column names separately. A JSON parser is responsible for converting a JSON text into another representation that must accept all the texts according to the JSON grammar. It can also accept non JSON forms or extensions.

We have to import the JSON file before reading.

import pandas as pd

data = pd.read\_json('hrdata.json')

print(data)

Output:

Name Hire Date Salary Leaves Remaining

0 John Idle 08/15/14 50000.0 10

1 Smith Gilliam 06/01/15 65000.0 6

2 Parker Chapman 05/12/14 45000.0 7

3 Jones Palin 11/01/13 70000.0 3

4 Terry Gilliam 08/12/14 48000.0 9

5 Michael Palin 05/23/13 66000.0 8

**Reading from the SQL database**

For reading a file from the SQL, first, you need to establish a connection using the Python library and then pass the query to pandas. Here, we use SQLite for demonstration.

Firstly, we have to install pysqlite3 and run this command into the terminal:

pip install pysqlite3

sqlite3 is used to establish a connection to the database, and then we can use it to generate a DataFrame through SELECT query.

For establishing a connection to the SQLite database file:

import sqlite3

con = sqlite3.connect("database.db")

A table called information is present in the SQLite database, and the index of the column called "index". We can read data from the information table by passing the SELECT query and the con.

df = pd.read\_sql\_query("SELECT \* FROM information", con)

Output:

Index E\_id Designation

0 46 M.Com

1 47 B.Com

2 48 B.Com

**Pandas Built-in Data Visualization**

Data Visualization is the presentation of data in graphical format. It helps people understand the significance of data by summarizing and presenting a huge amount of data in a simple and easy-to-understand format and helps communicate information clearly and effectively.

**Style Sheets** –

Matplotlib has style sheets which can be used to make the plots look a little nicer. These style sheets include plot\_bmh, plot\_fivethirtyeight, plot\_ggplot and more. They basically create a set of style rules that your plots follow. We recommend using them, they make all your plots have the same look and feel more professional. We can even create our own if want company’s plots to all have the same look (it is a bit tedious to create on though).

Example:

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

data = {'Name': ['Parker', 'Smith', 'John', 'William'],

'Percentage': [82, 98, 91, 87],

'Course': ['B.Sc','B.Ed','M.Phill','BA']}

df = pd.DataFrame(data)

df['Percentage'].hist()

plt.style.use('bmh')

df['Percentage'].hist()

plt.style.use('dark\_background')

df['Percentage'].hist()

plt.style.use('fivethirtyeight')

df['Percentage'].hist()

plt.show()

**To Save plotted graph on computer:**

plt.savefig('python\_pretty\_plot.png')

**Plot Types –**

There are several plot types built-in to pandas, most of them statistical plots by nature:

df.plot.area

df.plot.barh

df.plot.density

df.plot.hist

df.plot.line

df.plot.scatter

df.plot.bar

df.plot.box

df.plot.hexbin

df.plot.kde

df.plot.pie

You can also just call df.plot(kind='hist') or replace that kind argument with any of the key terms shown in the list above (e.g. ‘box’, ‘barh’, etc.).

1.) Area

An area chart or area graph displays graphically quantitative data. It is based on the line chart. The area between axis and line are commonly emphasized with colors, textures and hatchings. Commonly one compares two or more quantities with an area chart.

df2.plot.area(alpha = 0.4)

2.) Barplots

A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally. A vertical bar chart is sometimes called a line graph.

df2.plot.bar()

df2.plot.bar(stacked = True)

3.) Histograms

A histogram is a plot that lets you discover, and show, the underlying frequency distribution (shape) of a set of continuous data. This allows the inspection of the data for its underlying distribution (e.g., normal distribution), outliers, skewness, etc.

df1['A'].plot.hist(bins = 50)

4.) Line Plots

A line plot is a graph that shows frequency of data along a number line. It is best to use a line plot when the data is time series. It is a quick, simple way to organize data.

df1.plot.line(x = df1.index, y ='B', figsize =(12, 3), lw = 1)

5.) Scatter Plots

Scatter plots are used when you want to show the relationship between two variables. Scatter plots are sometimes called correlation plots because they show how two variables are correlated.

df1.plot.scatter(x ='A', y ='B')

You can use c to color based off another column value Use cmap to indicate colormap to use. For all the colormaps, check out: <http://matplotlib.org/users/colormaps.html>.

df1.plot.scatter(x ='A', y ='B', c ='C', cmap ='coolwarm')

**Or use s to indicate size based off another column. s parameter needs to be an array, not just the name of a column:**

df1.plot.scatter(x ='A', y ='B', s = df1['C']\*200)

6.) Box Plots

It is a plot in which a rectangle is drawn to represent the second and third quartiles, usually with a vertical line inside to indicate the median value. The lower and upper quartiles are shown as horizontal lines either side of the rectangle.

A boxplot is a standardized way of displaying the distribution of data based on a five number summary (“minimum”, first quartile (Q1), median, third quartile (Q3), and “maximum”). It can tell you about your outliers and what their values are. It can also tell you if your data is symmetrical, how tightly your data is grouped, and if and how your data is skewed.

df2.plot.box()

7.) Hexagonal Bin Plots

Hexagonal Binning is another way to manage the problem of having to many points that start to overlap. Hexagonal binning plots density, rather than points. Points are binned into gridded hexagons and distribution (the number of points per hexagon) is displayed using either the color or the area of the hexagons.

Useful for Bivariate Data, alternative to scatterplot:

df = pd.DataFrame(np.random.randn(1000, 2), columns =['a', 'b'])

df.plot.hexbin(x ='a', y ='b', gridsize = 25, cmap ='Oranges')

8.) Kernel Density Estimation plot (KDE)

KDE is a technique that let’s you create a smooth curve given a set of data.

This can be useful if you want to visualize just the “shape” of some data, as a kind of continuous replacement for the discrete histogram. It can also be used to generate points that look like they came from a certain dataset – this behavior can power simple simulations, where simulated objects are modeled off of real data.

df2['a'].plot.kde()

df2.plot.density()

**Pandas Plot**

It is used to make plots of DataFrame using matplotlib / pylab. Every plot kind has a corresponding method on the DataFrame.plot accessor: df.plot(kind='line') that are generally equivalent to the df.plot.line().

Syntax:

DataFrame.plot(x=None, y=None, kind='line', ax=None, subplots=False, sharex=None, sharey=False, layout=None, figsize=None, use\_index=True, title=None, grid=None, legend=True, style=None, logx=False, logy=False, loglog=False, xticks=None, yticks=None, xlim=None, ylim=None, rot=None, fontsize=None, colormap=None, table=False, yerr=None, xerr=None, secondary\_y=False, sort\_columns=False, \*\*kwds)

Parameters:

data: DataFrame

x: Refers to label or position, default value None

y: Refers to label, position or list of label, positions, default value None

It allows the plotting of one column versus another.

kind: str

* line': line plot (default)
* 'bar': vertical bar plot
* 'barh': horizontal bar plot
* 'hist': histogram
* 'box': boxplot
* 'kde': Kernel Density Estimation plot
* 'density': same as 'kde'
* 'area': area plot
* 'pie': pie plot
* 'scatter': scatter plot
* 'hexbin': hexbin plot

ax: matplotlib axes object, default None

subplots: boolean, default False

Make separate subplots for each column

sharex: It returns the boolean value and default value True if the ax is None else returns False.

If the subplots =True, it shares the x-axis and set some x-axis labels to the invisible;

Its default value is True if ax is None; otherwise, if an ax is passed, it returns false. If you pass True on both an ax and shareax, it will alter all the x-axis labels.

sharey: It also returns a boolean value that default value False.

If the subplots= True, it shares the y-axis and set some y-axis to the labels to invisible.

layout: It is an optional parameter that refers to the tuple for the layout of subplots.

figsize: Refers to a tuple (width, height) in inches.

use\_index: It returns the boolean value; default value True.

It uses the index as ticks for the x-axis.

title: Refers to a string or list that defines a title for the plot. If we pass a string, it will print string at the top of the figure. If we pass a list and subplots as True, it will print each item in the list in the corresponding subplot.

grid: Returns the boolean value, the default value is None. It defines the axis grid lines.

legend: Returns the False/True/'reverse' and place the legend on axis subplots.

style: Returns the list or dict. It defines the matplotlib line style per column.

logx: Returns the boolean value; the default value is False.

It generally uses a log scale on the x-axis.

logy: Returns the boolean value; the default value is False.

It generally uses log scaling on the y-axis.

loglog: Returns the boolean value; the default value is False.

It uses log scaling on both x and y axes

xticks: Refers to a sequence that consists of values to use for the xticks.

yticks: Refers to a sequence that consists of values to use for the yticks.

xlim: It consists 2-tuple/list.

ylim: It consists 2-tuple/list

rot: Refers to an integer value; the default value None

It generally Rotates for ticks (xticks for vertical, yticks for horizontal plots)

fontsize: Refers to an integer value; the default value is None.

Its main task is to specify the font size for xticks and yticks.

colormap: Refers to str or matplotlib colormap object, default value is None.

It provides colormap to select colors. If a value is a string, it loads colormap with that name from matplotlib.

colorbar: It is an optional parameter that returns a boolean value.

If the value is True, it plots the colorbar (only relevant for 'scatter' and 'hexbin' plots)

position: Refers to float value.

Its main task is to specify the relative alignments for the bar plot layout. Its value ranges from 0 (left/bottom-end) to 1 (right/top-end). The default value is 0.5 (center).

table: Returns the boolean value, Series or DataFrame, default value False

If the value is True, it draws a table using the data in the DataFrame.

If we pass a Series or DataFrame, it will pass data to draw a table.

yerr: Refers to the DataFrame, Series, array-like, dict, and str.

xerr: It is the same type as yerr.

stacked: Returns the boolean value; the default value is False in line and

bar plots, and True in area plot. If the value is True, it creates a stacked plot.

sort\_columns: Returns the boolean value; the default value is False

It sorts column names to determine plot ordering

secondary\_y: Returns the boolean value or sequence; the default value is False.

It checks whether to plot on the secondary y-axis. If a list/tuple, it plots the columns of list /tuple on the secondary y-axis

mark\_right: Returns the boolean value; the default value is True.

It is used when using a secondary\_y axis, automatically mark the column labels with "(right)" in the legend

'\*\*kwds': It is an optional parameter that refers to the options to pass to the matplotlib plotting method.

Example:

# import libraries

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

p = pd.Series(np.random.randn(2000), index = pd.date\_range(

'2/2/2000', periods = 2000))

p = ts.cumsum()

p.plot()

plt.show()

**Pandas Cheat Sheet**

Pandas can be used as the most important Python package for Data Science. It helps to provide a lot of functions that deal with the data in easier way. It's fast, flexible, and expressive data structures are designed to make real-world data analysis.

Pandas Cheat Sheet is a quick guide through the basics of Pandas that you will need to get started on wrangling your data with Python. If you want to begin your data science journey with Pandas, you can use it as a handy reference to deal with the data easily.

This cheat sheet will guide through the basics of the Pandas library from the data structure to I/O, selection, sorting and ranking, etc.

**Key and Imports**

We use following shorthand in the cheat sheet:

df: Refers to any Pandas Dataframe object.

s: Refers to any Pandas Series object. You can use the following imports to get started:

**Importing Data**

* pd.read\_csv(filename) : It read the data from CSV file.
* pd.read\_table(filename) : It is used to read the data from delimited text file.
* pd.read\_excel(filename) : It read the data from an Excel file.
* pd.read\_sql(query,connection \_object) : It read the data from a SQL table/database.
* pd.read\_json(json \_string) : It read the data from a JSON formatted string, URL or file.
* pd.read\_html(url) : It parses an html URL, string or the file and extract the tables to a list of dataframes.
* pd.read\_clipboard() : It takes the contents of clipboard and passes it to the read\_table() function.
* pd.DataFrame(dict) : From the dict, keys for the columns names, values for the data as lists.
* Exporting data
* df.to\_csv(filename): It writes to a CSV file.
* df.to\_excel(filename): It writes to an Excel file.
* df.to\_sql(table\_name, connection\_object): It writes to a SQL table.
* df.to\_json(filename) : It write to a file in JSON format.

**Create Test objects**

It is useful for testing the code segments.

* pd.DataFrame(np.random.rand(7,18)): Refers to 18 columns and 7 rows of random floats.
* pd.Series(my\_list): It creates a Series from an iterable my\_list.
* df.index= pd.date\_range('1940/1/20', periods=df.shape[0]): It adds the date index.
* Viewing/Inspecting Data
* df.head(n): It returns first n rows of the DataFrame.
* df.tail(n): It returns last n rows of the DataFrame.
* df.shape: It returns number of rows and columns.
* df.info(): It returns index, Datatype, and memory information.
* s.value\_counts(dropna=False): It views unique values and counts.
* df.apply(pd.Series.value\_counts): It refers to the unique values and counts for all the columns.

**Selection**

* df[col1]: It returns column with the label col as Series.
* df[[col1, col2]]: It returns columns as a new DataFrame.
* s.iloc[0]: It select by the position.
* s.loc['index\_one']: It select by the index.
* df.iloc[0,:]: It returns first row.
* df.iloc[0,0]: It returns the first element of first column.

**Data cleaning**

* df.columns = ['a','b','c']: It rename the columns.
* pd.isnull(): It checks for the null values and returns the Boolean array.
* pd.notnull(): It is opposite of pd.isnull().
* df.dropna(): It drops all the rows that contain the null values.
* df.dropna(axis= 1): It drops all the columns that contain null values.
* df.dropna(axis=1,thresh=n): It drops all the rows that have less than n non null values.
* df.fillna(x): It replaces all null values with x.
* s.fillna(s.mean()): It replaces all the null values with the mean(the mean can be replaced with almost any function from the statistics module).
* s.astype(float): It converts the datatype of series to float.
* s.replace(1, 'one'): It replaces all the values equal to 1 with 'one'.
* s.replace([1,3],[ 'one', 'three']):It replaces all 1 with 'one' and 3 with 'three'.
* df.rename(columns=lambda x: x+1):It rename mass of the columns.
* df.rename(columns={'old\_name': 'new\_ name'}): It consist selective renaming.
* df.set\_index('column\_one'): Used for changing the index.
* df.rename(index=lambda x: x+1): It rename mass of the index.

**Filter, Sort, and Groupby**

* df[df[col] > 0.5]: Returns the rows where column col is greater than 0.5
* df[(df[col] > 0.5) & (df[col] < 0.7)] : Returns the rows where 0.7 > col > 0.5
* df.sort\_values(col1) :It sorts the values by col1 in ascending order.
* df.sort\_values(col2,ascending=False) :It sorts the values by col2 in descending order.
* df.sort\_values([col1,col2],ascending=[True,False]) :It sort the values by col1 in ascending order and col2 in descending order.
* df.groupby(col1): Returns a groupby object for the values from one column.
* df.groupby([col1,col2]) :Returns a groupby object for values from multiple columns.
* df.groupby(col1)[col2]) :Returns mean of the values in col2, grouped by the values in col1.
* df.pivot\_table(index=col1,values=[col2,col3],aggfunc=mean) :It creates the pivot table that groups by col1 and calculate mean of col2 and col3.
* df.groupby(col1).agg(np.mean) :It calculates the average across all the columns for every unique col1 group.
* df.apply(np.mean) :Its task is to apply the function np.mean() across each column.
* nf.apply(np.max,axis=1) :Its task is to apply the function np.max() across each row.

**Join/Combine**

* df1.append(df2): Its task is to add the rows in df1 to the end of df2(columns should be identical).
* pd.concat([df1, df2], axis=1): Its task is to add the columns in df1 to the end of df2(rows should be identical).
* df1.join(df2,on=col1,how='inner'): SQL-style join the columns in df1 with the columns on df2 where the rows for col have identical values, 'how' can be of 'left', 'right', 'outer', 'inner'.

**Statistics**

The statistics functions can be applied to a Series, which are as follows:

* df.describe(): It returns the summary statistics for the numerical columns.
* df.mean() : It returns the mean of all the columns.
* df.corr(): It returns the correlation between the columns in the dataframe.
* df.count(): It returns the count of all the non-null values in each dataframe column.
* df.max(): It returns the highest value from each of the columns.
* df.min(): It returns the lowest value from each of the columns.
* df.median(): It returns the median from each of the columns.
* df.std(): It returns the standard deviation from each of the columns.