PANDAS

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:

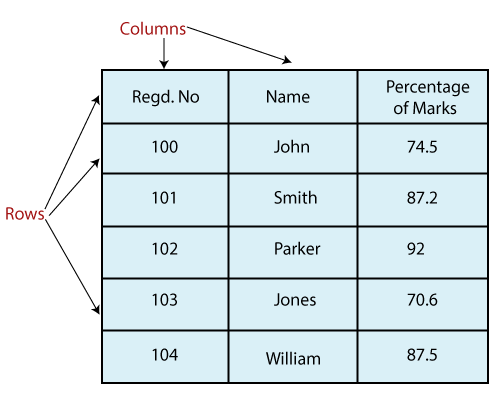
**data:** It consists of different forms like ndarray, series, map, constants, lists, array.

**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.



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:

1. # importing the pandas library
2. **import** pandas as pd
3. df = pd.DataFrame()
4. **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.

1. # importing the pandas library
2. **import** pandas as pd
3. # a list of strings
4. x = ['Python', 'Pandas']
6. # Calling DataFrame constructor on list
7. df = pd.DataFrame(x)
8. **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

1. # importing the pandas library
2. **import** pandas as pd
3. info = {'ID' :[101, 102, 103],'Department' :['B.Sc','B.Tech','M.Tech',]}
4. df = pd.DataFrame(info)
5. **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:

1. # importing the pandas library
2. **import** pandas as pd
4. info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
5. 'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}
7. d1 = pd.DataFrame(info)
8. **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.

1. # importing the pandas library
2. **import** pandas as pd
4. info = {'one' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f']),
5. 'two' : pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])}
7. d1 = pd.DataFrame(info)
8. **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:

1. # importing the pandas library
2. **import** pandas as pd
4. info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
5. 'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
7. df = pd.DataFrame(info)
9. # Add a new column to an existing DataFrame object
11. **print** ("Add new column by passing series")
12. df['three']=pd.Series([20,40,60],index=['a','b','c'])
13. **print** (df)
15. **print** ("Add new column using existing DataFrame columns")
16. df['four']=df['one']+df['three']
18. **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:

1. # importing the pandas library
2. **import** pandas as pd
4. info = {'one' : pd.Series([1, 2], index= ['a', 'b']),
5. 'two' : pd.Series([1, 2, 3], index=['a', 'b', 'c'])}
7. df = pd.DataFrame(info)
8. **print** ("The DataFrame:")
9. **print** (df)
11. # using del function
12. **print** ("Delete the first column:")
13. **del** df['one']
14. **print** (df)
15. # using pop function
16. **print** ("Delete the another column:")
17. df.pop('two')
18. **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.

1. # importing the pandas library
2. **import** pandas as pd
4. info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
5. 'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
7. df = pd.DataFrame(info)
8. **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.

1. # importing the pandas library
2. **import** pandas as pd
3. info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
4. 'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
5. df = pd.DataFrame(info)
6. **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.

1. # importing the pandas library
2. **import** pandas as pd
3. info = {'one' : pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e']),
4. 'two' : pd.Series([1, 2, 3, 4, 5, 6], index=['a', 'b', 'c', 'd', 'e', 'f'])}
5. df = pd.DataFrame(info)
6. **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.

1. # importing the pandas library
2. **import** pandas as pd
3. d = pd.DataFrame([[7, 8], [9, 10]], columns = ['x','y'])
4. d2 = pd.DataFrame([[11, 12], [13, 14]], columns = ['x','y'])
5. d = d.append(d2)
6. **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.

1. # importing the pandas library
2. **import** pandas as pd
4. a\_info = pd.DataFrame([[4, 5], [6, 7]], columns = ['x','y'])
5. b\_info = pd.DataFrame([[8, 9], [10, 11]], columns = ['x','y'])
7. a\_info = a\_info.append(b\_info)
9. # Drop rows with label 0
10. 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()](https://www.javatpoint.com/pandas-append) | Add the rows of other dataframe to the end of the given dataframe. |
| [Pandas DataFrame.apply()](https://www.javatpoint.com/pandas-apply) | Allows the user to pass a function and apply it to every single value of the Pandas series. |
| [Pandas DataFrame.assign()](https://www.javatpoint.com/pandas-dataframe-assign) | Add new column into a dataframe. |
| [Pandas DataFrame.astype()](https://www.javatpoint.com/pandas-dataframe-astype) | Cast the Pandas object to a specified dtype.astype() function. |
| [Pandas DataFrame.concat()](https://www.javatpoint.com/pandas-concatenation) | Perform concatenation operation along an axis in the DataFrame. |
| [Pandas DataFrame.count()](https://www.javatpoint.com/pandas-count) | Count the number of non-NA cells for each column or row. |
| [Pandas DataFrame.describe()](https://www.javatpoint.com/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()](https://www.javatpoint.com/pandas-dataframe-drop_duplicates) | Remove duplicate values from the DataFrame. |
| [Pandas DataFrame.groupby()](https://www.javatpoint.com/pandas-groupby) | Split the data into various groups. |
| [Pandas DataFrame.head()](https://www.javatpoint.com/pandas-dataframe-head) | Returns the first n rows for the object based on position. |
| [Pandas DataFrame.hist()](https://www.javatpoint.com/pandas-dataframe-hist) | Divide the values within a numerical variable into "bins". |
| [Pandas DataFrame.iterrows()](https://www.javatpoint.com/pandas-dataframe-iterrows) | Iterate over the rows as (index, series) pairs. |
| [Pandas DataFrame.mean()](https://www.javatpoint.com/pandas-dataframe-mean) | Return the mean of the values for the requested axis. |
| [Pandas DataFrame.melt()](https://www.javatpoint.com/pandas-melt) | Unpivots the DataFrame from a wide format to a long format. |
| [Pandas DataFrame.merge()](https://www.javatpoint.com/pandas-merge) | Merge the two datasets together into one. |
| [Pandas DataFrame.pivot\_table()](https://www.javatpoint.com/pandas-pivot-table) | Aggregate data with calculations such as Sum, Count, Average, Max, and Min. |
| [Pandas DataFrame.query()](https://www.javatpoint.com/pandas-dataframe-query) | Filter the dataframe. |
| [Pandas DataFrame.sample()](https://www.javatpoint.com/pandas-dataframe-sample) | Select the rows and columns from the dataframe randomly. |
| [Pandas DataFrame.shift()](https://www.javatpoint.com/pandas-shift) | Shift column or subtract the column value with the previous row value from the dataframe. |
| [Pandas DataFrame.sort()](https://www.javatpoint.com/python-pandas-sorting) | Sort the dataframe. |
| [Pandas DataFrame.sum()](https://www.javatpoint.com/pandas-sum) | Return the sum of the values for the requested axis by the user. |
| [Pandas DataFrame.to\_excel()](https://www.javatpoint.com/pandas-dataframe-to_excel) | Export the dataframe to the excel file. |
| [Pandas DataFrame.transpose()](https://www.javatpoint.com/pandas-dataframe-transpose) | Transpose the index and columns of the dataframe. |
| [Pandas DataFrame.where()](https://www.javatpoint.com/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:

1. 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:

1. **import** pandas as pd
2. # Create first Dataframe using dictionary
3. info1 = pd.DataFrame({"x":[25,15,12,19],
4. "y":[47, 24, 17, 29]})
5. # Create second Dataframe using dictionary
6. Info2 = pd.DataFrame({"x":[25, 15, 12],
7. "y":[47, 24, 17],
8. "z":[38, 12, 45]})
9. # append info2 at end in info1
10. info.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:

1. **import** pandas as pd
2. # Create first Dataframe using dictionary
3. info1 = info = pd.DataFrame({"x":[15, 25, 37, 42],
4. "y":[24, 38, 18, 45]})
5. # Create second Dataframe using dictionary
6. info2 = pd.DataFrame({"x":[15, 25, 37],
7. "y":[24, 38, 45]})
8. # print value of info1
9. print(info1, "\n")
10. # print values of info2
11. info2
12. # append info2 at the end of info1 dataframe
13. info1.append(df2)
14. # Continuous index value will maintained
15. # across rows in the **new** appended data frame.
16. 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:

1. 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.

Exception Handling in Java - Javatpoint

Example:

1. info = pd.DataFrame([[2, 7]] \* 4, columns=['P', 'Q'])
2. info.apply(np.sqrt)
3. info.apply(np.sum, axis=0)
4. info.apply(np.sum, axis=1)
5. info.apply(lambda x: [1, 2], axis=1)
6. info.apply(lambda x: [1, 2], axis=1, result\_type='expand')
7. info.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
8. info.apply(lambda x: [1, 2], axis=1, result\_type='broadcast')
9. 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.

Package in Java

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

Syntax:

1. 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:

1. **import** pandas as pd
2. **import** numpy as np
3. info=pd.DataFrame([[1,5,7],[10,12,15],[18,21,24],[np.nan,np.nan,np.nan]],columns=['X','Y','Z'])
4. info.agg(['sum','min'])

**Output:**

X Y Z

sum 29.0 38.0 46.0

min 1.0 5.0 7.0

Example2:

1. **import** pandas as pd
2. **import** numpy as np
3. info=pd.DataFrame([[1,5,7],[10,12,15],[18,21,24],[np.nan,np.nan,np.nan]],columns=['X','Y','Z'])
4. 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

1. 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:

1. **import** pandas as pd
2. # Create an empty dataframe
3. info = pd.DataFrame()
5. # Create a column
6. info['ID'] = [101, 102, 103]
8. # View the dataframe
9. info
10. # Assign a **new** column to dataframe called 'age'
11. info.assign(Name = ['Smith', 'Parker', 'John'])

**Output**

ID Name

0 101 Smith

1 102 Parker

2 103 John

Example 2:

1. **import** pandas as pd
2. # Create a dataframe
3. info = pd.DataFrame({'temp\_c': [17.0, 25.0]},
4. # Create an index that consist some values
5. index=['Canada', 'Australia'])
6. # View the dataframe
7. info
8. info.assign(temp\_f=lambda x: x.temp\_c \* 7 / 2 + 24)
9. info.assign(temp\_f=lambda x: x['temp\_c'] \* 6 / 2 + 21,
10. 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.astype()

The astype() method is generally used for casting the pandas object to a specified **dtype.astype()** function. It can also convert any suitable existing column to a categorical type.

It comes into use when we want to case a particular column data type to another data type. We can also use the input to Python dictionary to change more than one column type at once. In the dictionary, the key label corresponds to the column name, and the values label corresponds to the new data types that we want to be in the columns.

Syntax

1. DataFrame.astype(dtype, copy=True, errors='raise', \*\*kwargs)

Parameters

**dtype:** It uses numpy.dtype or the Python type for casting the entire pandas object to the same type. It can also use {col: dtype, ?} alternatively where col refers to the column label, and dtype is a numpy.dtype or Python type for casting one or more of the DataFrame's columns to column-specific types.

OOPs Concepts in Java

**copy:** If copy=True, it returns a copy. Be careful when setting copy= False because changes to values may propagate to other pandas objects.

**errors:** For provided dtype, it controls the raising of exceptions on the invalid data.

* **raise:** It allows the exception that is to be raised.
* **ignore:** It ignores the exception. It returns the original object on error.

**kwargs:** It is a keyword argument that is to be passed on to the constructor.

Returns

**casted:** It returns the same type as a caller.

Example

1. **import** pandas as pd
2. a = {'col1': [1, 2], 'col2': [3, 4]}
3. info = pd.DataFrame(data=a)
4. info.dtypes
5. # We convert it into 'int64' type.
6. info.astype('int64').dtypes
7. info.astype({'col1': 'int64'}).dtypes
8. x = pd.Series([1, 2], dtype='int64')
9. x.astype('category')
10. cat\_dtype = pd.api.types.CategoricalDtype(
11. categories=[2, 1], ordered=True)
12. x.astype(cat\_dtype)
13. x1 = pd.Series([1,2])
14. x2 = x1.astype('int64', copy=False)
15. x2[0] = 10
16. x1  # note that x1[0] has changed too

**Output**

0 12

1 2

dtype: int64

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:

1. 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()**.

Method Overloading vs Overriding in Java

1. **import** pandas as pd
2. **import** numpy as np
3. info = pd.DataFrame({"Person":["Parker", "Smith", "William", "John"],
4. "Age": [27., 29, np.nan, 32]
5. 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.

1. **import** pandas as pd
2. **import** numpy as np
3. info = pd.DataFrame({"Person":["Parker", "Smith", "William", "John"],
4. "Age": [27., 29, np.nan, 32]
5. info.count(axis='columns')

**Output**

0 2

1 2

2 1

3 2

dtype: int64

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:

1. 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.

Features of Java - Javatpoint

**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.
2. **bins:** It mainly refers to a **ndarray**

**Example1:** The below example segments the numbers into bins:

1. **import** pandas as pd
2. **import** numpy as np
3. info\_nums = pd.DataFrame({'num': np.random.randint(1, 50, 11)})
4. print(info\_nums)
5. info\_nums['num\_bins'] = pd.cut(x=df\_nums['num'], bins=[1, 25, 50])
6. print(info\_nums)
7. print(info\_nums['num\_bins'].unique())

**Output:**

num

0 48

1 36

2 7

3 2

4 25

5 2

6 13

7 5

8 7

9 25

10 10

num num\_bins

0 48 (1.0, 25.0]

1 36 (1.0, 25.0]

2 7 (1.0, 25.0]

3 2 (1.0, 25.0]

4 25 NaN

5 2 (1.0, 25.0]

6 13 (1.0, 25.0]

7 5 (1.0, 25.0]

8 7 (1.0, 25.0]

9 25 (1.0, 25.0]

10 10 NaN

[(1.0, 25.0], NaN]

Categories (1, interval[int64]): [(1, 25]]

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

1. **import** pandas as pd
2. **import** numpy as np
3. info\_nums = pd.DataFrame({'num': np.random.randint(1, 10, 7)})
4. print(info\_nums)
5. info\_nums['nums\_labels'] = pd.cut(x=info\_nums['num'], bins=[1, 7, 10], labels=['Lows', 'Highs'], right=False)
6. print(info\_nums)
7. print(info\_nums['nums\_labels'].unique())

**Output:**

num

0 9

1 9

2 4

3 9

4 4

5 7

6 2

num nums\_labels

0 9 Highs

1 9 Highs

2 4 Lows

3 9 Highs

4 4 Lows

5 7 Highs

6 2 Lows

[Highs, Lows]

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

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

1. 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

1. **import** pandas as pd
2. **import** numpy as np
3. a1 = pd.Series([1, 2, 3])
4. a1.describe()

**Output**

Polymorphism in Java | Dynamic Method Dispatch

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

1. **import** pandas as pd
2. **import** numpy as np
3. a1 = pd.Series(['p', 'q', 'q', 'r'])
4. a1.describe()

**Output**

count 4

unique 3

top q

freq 2

dtype: object

Example3

1. **import** pandas as pd
2. **import** numpy as np
3. a1 = pd.Series([1, 2, 3])
4. a1.describe()
5. a1 = pd.Series(['p', 'q', 'q', 'r'])
6. a1.describe()
7. info = pd.DataFrame({'categorical': pd.Categorical(['s','t','u']),
8. 'numeric': [1, 2, 3],
9. 'object': ['p', 'q', 'r']
10. })
11. info.describe(include=[np.number])
12. info.describe(include=[np.object])
13. info.describe(include=['category'])

**Output**

categorical

count 3

unique 3

top u

freq 1

Example4

1. **import** pandas as pd
2. **import** numpy as np
3. a1 = pd.Series([1, 2, 3])
4. a1.describe()
5. a1 = pd.Series(['p', 'q', 'q', 'r'])
6. a1.describe()
7. info = pd.DataFrame({'categorical': pd.Categorical(['s','t','u']),
8. 'numeric': [1, 2, 3],
9. 'object': ['p', 'q', 'r']
10. })
11. info.describe()
12. info.describe(include='all')
13. info.numeric.describe()
14. info.describe(include=[np.number])
15. info.describe(include=[np.object])
16. info.describe(include=['category'])
17. info.describe(exclude=[np.number])
18. 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

1. 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:
  + **first:** It drops the duplicate values except for the first occurrence.
  + **last:** It drops the duplicate values except for the last occurrence.
  + **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.

Hello Java Program for Beginners

Example

1. **import** pandas as pd
2. emp = {"Name": ["Parker", "Smith", "William", "Parker"],
3. "Age": [21, 32, 29, 21]}
4. info = pd.DataFrame(emp)
5. print(info)

**Output**

Name Age

0 Parker 21

1 Smith 32

2 William 29

3 Parker 21

1. **import** pandas as pd
2. emp = {"Name": ["Parker", "Smith", "William", "Parker"],
3. "Age": [21, 32, 29, 21]}
4. info = pd.DataFrame(emp)
5. info = info.drop\_duplicates()
6. print(info)

**Output**

Name Age

0 Parker 21

1 Smith 32

2 William 29

# Pandas DataFrame.groupby()

## Introduction

Pandas is a famous Python library for data manipulation and analysis. It gives a strong and adaptable method for dealing with data structures, for example, data frames and series. One of the critical highlights of Pandas is the capacity to group data by at least one variables utilizing the groupby() method. In this article, we'll investigate how to utilize groupby() to break down and control data.

## What is groupby() in Pandas?

The groupby() method in Pandas is a useful asset that permits you to group data in light of at least one variables. It is utilized to split an enormous data outline into more modest groups in light of a clear cut variable, for example, a column name, and afterward apply a capability to each group independently. This permits you to dissect subsets of your data independently and analyze them.

## Syntax:

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

### Parameters of Groupby:

**by:**

4.5M

467

History of Java

* 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. 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.

## Groupby Operations

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

Groupby splitting is a method utilized in data analysis to bunch data into subsets in light of the values of at least one variables. The fundamental thought is to parted a dataset into groups in light of a specific variable, and afterward perform some sort of analysis or calculation on each gathering separately. This procedure is in many cases utilized in statistical analysis and AI, as well as in data visualization and exploratory data analysis.

* The course of groupby splitting commonly includes the accompanying advances:
* Select at least one variables that you need to bunch your data by.
* Divide your data into separate groups in view of the values of these variables.
* Perform some sort of analysis or calculation on each gathering separately.Join the outcomes from each gathering to deliver a last rundown of the data.

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. Aggregation is a strong method in data analysis that includes joining various values in a dataset to get summary statistics, which help in acquiring experiences and going with informed choices.Aggregation can be performed on a single variable or across various variables in a dataset.

Pandas gives an extensive arrangement of functions for performing aggregation tasks, including mean(), sum(), count(), min(), max(), median(), var(), and std(). These functions can be applied to individual columns or gatherings of columns in a DataFrame.

**Example**

1. # **import** the pandas library
2. **import** pandas as pd
3. **import** numpy as np
4. data = {'Name': ['Parker', 'Smith', 'John', 'William'],
5. 'Percentage': [82, 98, 91, 87],
6. 'Course': ['B.Sc','B.Ed','M.Phill','BA']}
7. df = pd.DataFrame(data)
9. grouped = df.groupby('Course')
10. 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. In Pandas, transformation includes applying a function to a column or a group of columns in a DataFrame to modify the data, frequently with the end goal of standardization or scaling. Pandas gives the transform() strategy, which applies a function to each group of a groupby object and returns a changed rendition of the first DataFrame. The transform() strategy can be utilized to play out a variety of transformations, including normalizing data, scaling data, and filling missing values.

**Example**

1. # **import** the pandas library
2. **import** pandas as pd
3. **import** numpy as np
5. data = {'Name': ['Parker', 'Smith', 'John', 'William'],
6. 'Percentage': [82, 98, 91, 87],
7. 'Course': ['B.Sc','B.Ed','M.Phill','BA']}
8. df = pd.DataFrame(data)
10. grouped = df.groupby('Course')
11. Percentage = lambda x: (x - x.mean()) / x.std()\*10
12. 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. Filtration in Pandas refers to the most common way of choosing a subset of rows from a DataFrame in light of some condition or models. The interaction includes filtering out rows that don't meet the predetermined condition and holding those that do.

Pandas gives a few methods to filtration, including boolean indexing, query(), and filter(). These methods permit us to filter the rows of a DataFrame in view of a variety of models, for example, column values, index labels, or conditions including numerous columns.

**Example**

1. # **import** the pandas library
2. **import** pandas as pd
3. **import** numpy as np
5. data = {'Name': ['Parker', 'Smith', 'John', 'William'],
6. 'Percentage': [82, 98, 91, 87],
7. 'Course': ['B.Sc','B.Ed','M.Phill','BA']}
8. df = pd.DataFrame(data)
10. grouped = df.groupby('Course')
11. print (df.groupby('Course').filter(lambda x: len(x) >= 1))

**Output**

Name Percentage Course

0 Parker 82 B.Sc

1 Smith 98 B.Ed

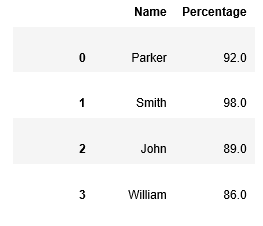
2 John 91 M.Phill

3 William 87 BA

**Example**

1. **import** pandas as pd
2. info = pd.DataFrame({'Name': ['Parker', 'Smith','John', 'William'],'Percentage': [92., 98., 89., 86.]})
3. info

**Output**



**Example**

1. # **import** the pandas library
2. **import** pandas as pd
4. data = {'Name': ['Parker', 'Smith', 'John', 'William'],
5. 'Percentage': [82, 98, 91, 87],}
6. info = pd.DataFrame(data)
8. print (info)

**Output**

Name Percentage

0 Parker 82

1 Smith 98

2 John 91

3 William 87

## Conclusion:

In this article, we've seen the groupby() technique in Pandas and how to utilize it to group data by at least one variables. We've perceived how to apply functions to groups, including various functions utilizing the agg() technique. We've likewise perceived how to utilize contingent articulations with groupby() to make more unambiguous groups. With the groupby() strategy, you can break down and control your data in a strong and adaptable manner.

# Pandas DataFrame.hist()

The hist() function is defined as a quick way to understand the distribution of certain numerical variables from the dataset. It divides the values within a numerical variable into "**bins**". It counts the number of examinations that fall into each of the bin. These bins are responsible for a rapid and intuitive sense of the distribution of the values within a variable by visualizing bins.

We can create a histogram by using the **DataFrame.hist()** method, which is a wrapper for the matplotlib pyplot API.

It is also a useful tool that quickly access the probability distribution.

3.1M

265

Java Collection MCQ Set 1

### Syntax

1. DataFrame.hist(data, column=None, by=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, ax=None, sharex=False, sharey=False, figsize=None, layout=None, bins=10, \*\*kwds)

### Parameters

* **data:** A DataFrame.  
  It is a pandas DataFrame object that holds the data.
* **column:** Refers to a string or sequence.  
  If it is passed, it will be used to limit the data to a subset of columns.
* **by:** It is an optional parameter. If it is passed, then it will be used to form the histogram for independent groups.
* **grid:** It is also an optional parameter. Used for showing the axis grid lines. Default value True.
* **xlabelsize:** Refers to the integer value. Default value None. Used for specifying the changes in the x-axis label size.
* **xrot:** Refers to float value. Used for rotating the x-axis labels. Default value None.
* **ylabelsize:** Refers to an integer value. Used for specifying the changes in the y-axis label size.
* **yrot:** Refers to the float value. Used for rotating the y-axis labels. Default value None.
* **ax:** Matplotlib axes object.  
  It defines the axis on which we need to plot the histogram. Default value None.
* **sharex:** Refers to the boolean value. Default value True, if ax is None else False. In the case of subplots, if value is True, it shares the x-axis and sets some of the x-axis labels to invisible. Its Default value is True.  
  If the ax is none, it returns False if an ax is passed in.

#### Note:**Passing true in both an ax and sharex, it will alter all x-axis labels for all the subplots.**

* **sharey:** Default value False. In the case of subplots is True, it shares the y-axis and sets some y-axis labels to invisible.
* **figsize:** Refers to the size in inches for the figure to create. By default, it uses the value in **matplotlib.rcParams**.
* **layout:** It is an optional parameter. It returns the tuple of (rows, columns) for the layout of the histograms.
* **bins:** Default value 10. It refers to the number of histogram bins that are to be used. If an integer value is given, then it returns the calculated value of bins +1 bin edges.
* **\*\*kwds:** Refers to all the other plotting keyword arguments that are to be passed to matplotlib.pyplot.hist().

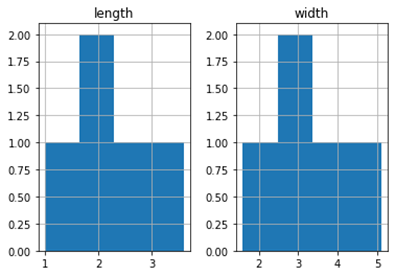
### Returns

It returns the matplotlib.AxesSubplot or numpy.ndarray.

### Example1

1. **import** pandas as pd
2. info = pd.DataFrame({
3. 'length': [2, 1.7, 3.6, 2.4, 1],
4. 'width': [4.2, 2.6, 1.6, 5.1, 2.9]
5. })
6. hist = info.hist(bins=4)

**Output**



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.

2.7M

254

Java Try Catch

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

1. **import** pandas as pd
2. **import** numpy as np
4. info = pd.DataFrame(np.random.randn(4,2),columns = ['col1','col2'])
5. **for** row\_index,row in info.iterrows():
6. print (row\_index,row)

Example2

1. # importing pandas module
2. **import** pandas as pd
4. # making data frame from csv file
5. data = pd.read\_csv("aa.csv")
7. **for** i, j in data.iterrows():
8. print(i, j)
9. 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

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

1. 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.

4.5M

497

Abstract class in Java | Abstraction in Java

Example

1. # importing pandas as pd
2. **import** pandas as pd
3. # Creating the dataframe
4. info = pd.DataFrame({"A":[8, 2, 7, 12, 6],
5. "B":[26, 19, 7, 5, 9],
6. "C":[10, 11, 15, 4, 3],
7. "D":[16, 24, 14, 22, 1]})
8. # Print the dataframe
9. info
10. # If axis = 0 is not specified, then
11. # by **default** method **return** the mean over
12. # the index axis
13. info.mean(axis = 0)

**Output**

A 7.0

B 13.2

C 8.6

D 15.4

dtype: float64

Example2

1. # importing pandas as pd
2. **import** pandas as pd
3. # Creating the dataframe
4. info = pd.DataFrame({"A":[5, 2, 6, 4, None],
5. "B":[12, 19, None, 8, 21],
6. "C":[15, 26, 11, None, 3],
7. "D":[14, 17, 29, 16, 23]})
8. # **while** finding mean, it skip **null** values
9. 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 melt()

The Pandas.melt() function is used to **unpivot** the DataFrame from a wide format to a long format.

Its main task is to massage a DataFrame into a format where some columns are identifier variables and remaining columns are considered as measured variables, are unpivoted to the row axis. It leaves just two non-identifier columns, variable and value.

Syntax

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

Parameters

* **frame:** It refers to the DataFrame.
* **id\_vars[tuple, list, or ndarray, optional]:** It refers to the columns to use as identifier variables.
* **value\_vars[tuple, list, or ndarray, optional]:** Refers to columns to unpivot. If it is not specified, use all columns that are not set as id\_vars.
* **var\_name[scalar]:** Refers to a name to use for the 'variable' column. If it is None, it uses frame.columns.name or 'variable'.
* **value\_name[scalar, default 'value']:** Refers to a name to use for the 'value' column.
* **col\_level[int or string, optional]: It will use this level to melt if the columns are MultiIndex.**

Returns

**It returns the unpivoted DataFrame as the output.**

Example

1. **# importing pandas as pd**
2. **import pandas as pd**
3. **# creating a dataframe**
4. **info = pd.DataFrame({'Name': {0: 'Parker', 1: 'Smith', 2: 'John'},**
5. **'Language': {0: 'Python', 1: 'Java', 2: 'C++'},**
6. **'Age': {0: 22, 1: 30, 2: 26}})**
8. **# Name is id\_vars and Course is value\_vars**
9. **pd.melt(info, id\_vars =['Name'], value\_vars =['Language'])**
10. **info**

**Output**

**Name Language Age**

**0 Parker Python 22**

**1 Smith Java 30**

**2 John C++ 26**

Example2

1. **import pandas as pd**
2. **info = pd.DataFrame({'A': {0: 'p', 1: 'q', 2: 'r'},**
3. **'B': {0: 40, 1: 55, 2: 25},**
4. **'C': {0: 56, 1: 62, 2: 42}})**
5. **pd.melt(info, id\_vars=['A'], value\_vars=['C'])**
6. **pd.melt(info, id\_vars=['A'], value\_vars=['B', 'C'])**
7. **pd.melt(info, id\_vars=['A'], value\_vars=['C'],**
8. **var\_name='myVarname', value\_name='myValname')**

**Output**

**A myVarname myValname**

**0 p C 56**

**1 q C 62**

**2 r C 42**

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:

1. pd.merge(left, right, how='inner', on=None, left\_on=None, right\_on=None,
2. 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

1. # **import** the pandas library
2. **import** pandas as pd
3. left = pd.DataFrame({
4. 'id':[1,2,3,4],
5. 'Name': ['John', 'Parker', 'Smith', 'Parker'],
6. 'subject\_id':['sub1','sub2','sub4','sub6']})
7. right = pd.DataFrame({
8. 'id':[1,2,3,4],
9. 'Name': ['William', 'Albert', 'Tony', 'Allen'],
10. 'subject\_id':['sub2','sub4','sub3','sub6']})
11. print (left)
12. 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:

1. **import** pandas as pd
2. left = pd.DataFrame({
3. 'id':[1,2,3,4,5],
4. 'Name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
5. 'subject\_id':['sub1','sub2','sub4','sub6','sub5']})
6. right = pd.DataFrame({
7. 'id':[1,2,3,4,5],
8. 'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
9. 'subject\_id':['sub2','sub4','sub3','sub6','sub5']})
10. 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**.

It also allows the user to sort and filter your data when the pivot table has been created.

Parameters:

* **data:** A DataFrame.
* **values:** It is an **optional** parameter and refers the column to aggregate.
* **index:** It refers to the column, Grouper, and array.

If we pass an array, it must be of the same length as data.

2.1M

232

C++ vs Java

* **columns:** Refers to column, Grouper, and array

If we pass an array, it must be of the same length as data.

* **aggfunc:** function, list of functions, dict, default numpy.mean  
  If we pass the list of functions, the resulting pivot table will have hierarchical columns whose top level are the function names.  
  If we pass a dict, the key is referred to as a column to aggregate, and value is function or list of functions.
* **fill\_value[scalar, default None]:** It replaces the missing values with a value.
* **margins[boolean, default False]:** It add all the row / columns (e.g. for subtotal / grand totals)
* **dropna[boolean, default True] :** It drops the columns whose entries are all NaN.
* **margins\_name[string, default 'All'] :** It refers to the name of the row/column that will contain the totals when margins are True.

Returns:

It returns a DataFrame as the output.

Example:

1. # importing pandas as pd
2. **import** pandas as pd
3. **import** numpy as np
5. # create dataframe
6. info = pd.DataFrame({'P': ['Smith', 'John', 'William', 'Parker'],
7. 'Q': ['Python', 'C', 'C++', 'Java'],
8. 'R': [19, 24, 22, 25]})
9. info
10. table = pd.pivot\_table(info, index =['P', 'Q'])
11. table

**Output**

P Q R

John C 24

Parker Java 25

Smith Python 19

William C 22

Example:

import pandas as pd

data = {

'Date': ['2023-08-01', '2023-08-01', '2023-08-02', '2023-08-02', '2023-08-01'],

'Product': ['A', 'B', 'A', 'B', 'A'],

'Sales': [100, 150, 200, 120, 180]

}

df = pd.DataFrame(data)

print(df)

pivot\_table = df.pivot\_table(index='Date', columns='Product', values='Sales', aggfunc='sum')

print(pivot\_table)

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:

1. 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.

4.3M

468

Exception Handling in Java - Javatpoint

**Example 1:** The below example renames a single column:

1. **import** pandas as pd
2. # Define a dictionary containing information of employees
3. info = {'name': ['Parker', 'Smith', 'William', 'Robert'],
4. 'age': [38, 47, 44, 34],
5. 'language': ['Java', 'Python', 'JavaScript', 'Python']}
6. # Convert dictionary into DataFrame
7. info\_pd = pd.DataFrame(info)
8. # Before renaming columns
9. print(info\_pd)
10. info\_pd.rename(columns = {'name':'Name'}, inplace = True)
11. # After renaming columns
12. 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:

1. **import** pandas as pd
2. # Define a dictionary containing information of employees
3. info = {'name': ['Parker', 'Smith', 'William', 'Robert'],
4. 'age': [38, 47, 44, 34],
5. 'language': ['Java', 'Python', 'JavaScript', 'Python']}
6. # Convert dictionary into DataFrame
7. info\_pd = pd.DataFrame(info)
8. # Before renaming columns
9. print(info\_pd)
10. info\_pd.rename(columns = {'name':'Name', 'age':'Age', 'language':'Language'}, inplace = True)
11. # After renaming columns
12. 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:

1. **import** pandas as pd
2. data = {'Name': ['Smith', 'Parker', 'William'], 'Emp\_ID': [101, 102, 103], 'Language': ['Python', 'Java', 'JavaScript']}
3. info1 = pd.DataFrame(data)
4. print('DataFrame:\n', info1)
5. info2 = info.rename(index={0: '#0', 1: '#1', 2: '#2'})
6. 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.sample()

The Pandas sample() is used to select the rows and columns from the DataFrame randomly. If we want to build a model from an extensive dataset, we have to randomly choose a smaller sample of the data that is done through a function **sample**.

Syntax

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

Parameters

* **n:** It is an optional parameter that consists of an integer value and defines the number of random rows generated.
* **frac:** It is also an optional parameter that consists of float values and returns **float value \* length of data frame values**. It cannot be used with a parameter n.
* **replace:** It consists of boolean value. If it is true, it returns a sample with replacement. The default value of the replace is false.
* **weights:** It is also an **optional** parameter that consists of str or ndarray-like. Default value "**None**" that results in equal probability weighting.  
  If a Series is being passed; it will align with the target object on the index. The index values in weights that are not found in the sampled object will be ignored, and index values in the sampled object not in weights will be assigned zero weights.  
  If a DataFrame is being passed when **axis =0;** it will accept the name of a column.  
  If the weights are Series; then, the weights must be of the same length as axis being sampled.  
  If the weights are not equal to 1; it will be normalized to the sum of 1.  
  The missing value in the weights column is considered as zero.  
  Infinite values are not allowed in the weights column.
* **random\_state:** It is also an **optional** parameter that consists of an integer or numpy.random.RandomState. If the value is int, it seeds for the random number generator or numpy RandomState object.
* **axis:** It is also an optional parameter that consists of integer or string value. 0 or '**row**' and 1 or 'column'.

Returns

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

Example1

1. **import** pandas as pd
2. info = pd.DataFrame({'data1': [2, 4, 8, 0],
3. 'data2': [2, 0, 0, 0],
4. 'data3': [10, 2, 1, 8]},
5. index=['John', 'Parker', 'Smith', 'William'])
6. info
7. info['data1'].sample(n=3, random\_state=1)
8. info.sample(frac=0.5, replace=True, random\_state=1)
9. info.sample(n=2, weights='data3', random\_state=1)

**Output**

data1 data2 data3

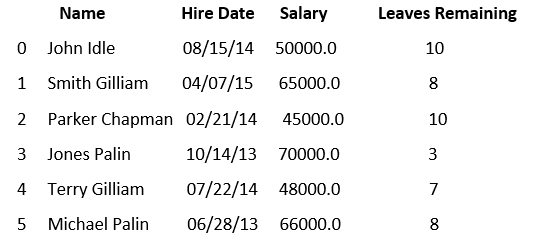
John 2 2 10

William 0 0 8

Example2

In this example, we take a csv file and extract random rows from the DataFrame by using a sample.

The csv file named as **aa** that contains the following dataset:



**Let's write a code that extract the random rows from the above dataset:**

1. # importing pandas **package**
2. **import** pandas as pd
3. # define data frame from csv file
4. data = pd.read\_csv("aa.csv")
5. # randomly select one row
6. row1 = data.sample(n = 1)
7. # display row
8. row1
9. # randomly select another row
10. row2 = data.sample(n = 2)
11. # display  row
12. row2

**Output**

Name Hire Date Salary Leaves Remaining

2 Parker Chapman 02/21/14 45000.0 10

5 Michael Palin 06/28/13 66000.0 8

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:

1. 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()**.

1. **import** pandas as pd
2. info= pd.DataFrame({'a\_data': [45, 28, 39, 32, 18],
3. 'b\_data': [26, 37, 41, 35, 45],
4. 'c\_data': [22, 19, 11, 25, 16]})
5. 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**.

1. **import** pandas as pd
2. info= pd.DataFrame({'a\_data': [45, 28, 39, 32, 18],
3. 'b\_data': [26, 38, 41, 35, 45],
4. 'c\_data': [22, 19, 11, 25, 16]})
5. info.shift(periods=2)
6. 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()

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:

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

**Output**

4.1M

470

Package in Java

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

1. **import** pandas as pd
2. **import** numpy as np
3. info=pd.DataFrame(np.random.randn(10,2),index=[1,2,5,4,8,7,9,3,0,6],columns = ['col4','col3'])
4. info2=info.sort\_index()
5. 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:

1. **import** pandas as pd
2. **import** numpy as np
3. info= pd.DataFrame(np.random.randn(10,2),index=[1,4,7,2,5,3,0,8,9,6],columns = ['col4','col5'])
5. info\_2 = info.sort\_index(ascending=False)
6. 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:

1. **import** pandas as pd
2. **import** numpy as np
4. info = pd.DataFrame(np.random.randn(10,2),index=[1,4,8,2,0,6,7,5,3,9],columns = ['col4','col7'])
5. info\_2=info.sort\_index(axis=1)
6. print(info\_2)

**Output**

col4 col7

1 -0.509367 -1.609514

4 -0.516731 0.397375

8 -0.201157 -0.009864

2 1.440567 1.058436

0 0.955486 -0.009777

6 -1.211133 0.415147

7 0.095644 0.531727

5 -0.881241 -0.871342

3 0.206327 -1.154724

9 1.418127 0.146788

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:

1. **import** pandas as pd
2. **import** numpy as np
3. info = pd.DataFrame({'col1':[7,1,8,3],'col2':[8,12,4,9]})
4. info\_2 = info.sort\_values(by='col2')
5. 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.

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.

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:

1. 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.

5.8M

760

OOPs Concepts in Java

**Next**

**Stay**

* **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:

1. **import** pandas as pd
2. # **default** min\_count = 0
3. pd.Series([]).sum()
4. # Passed min\_count = 1, then sum of an empty series will be NaN
5. pd.Series([]).sum(min\_count = 1)

**Output**

0.0

nan

Example2:

1. **import** pandas as pd
2. # making a dict of list
3. info = {'Name': ['Parker', 'Smith', 'William'],
4. 'age' : [32, 28, 39]}
5. data = pd.DataFrame(info)
6. # sum of all salary stored in 'total'
7. data['total'] = data['age'].sum()
8. print(data)

**Output**

Name age total

0 Parker 32 99

1 Smith 28 99

2 William 39 99

Pandas Concatenation

Pandas is capable of combining Series, DataFrame, and Panel objects through different kinds of set logic for the indexes and the relational algebra functionality.

The **concat()** function is responsible for performing concatenation operation along an axis in the DataFrame.

Syntax:

1. pd.concat(objs,axis=0,join='outer',join\_axes=None,
2. ignore\_index=False)

Parameters:

* **objs:** It is a sequence or mapping of series or DataFrame objects.  
  If we pass a dict in the DataFrame, then the sorted keys will be used as the **keys<.strong> argument, and the values will be selected in that case. If any non-objects are present, then it will be dropped unless they are all none, and in this case, a ValueError will be raised.**
* **axis: It is an axis to concatenate along.**
* **join: Responsible for handling indexes on another axis.**
* **join\_axes: A list of index objects. Instead of performing the inner or outer set logic, specific indexes use for the other (n-1) axis.**
* **ignore\_index: bool, default value False  
  It does not use the index values on the concatenation axis, if true. The resulting axis will be labeled as 0, ..., n - 1.**

Returns

**A series is returned when we concatenate all the Series along the axis (axis=0). In case if objs contains at least one DataFrame, it returns a DataFrame.**

Example1:

1. **import pandas as pd**
2. **a\_data = pd.Series(['p', 'q'])**
3. **b\_data = pd.Series(['r', 's'])**
4. **pd.concat([a\_data, b\_data])**

**Output**

**0 p**

**1 q**

**0 r**

**1 s**

**dtype: object**

**Example2: In the above example, we can reset the existing index by using the ignore\_index parameter. The below code demonstrates the working of ignore\_index.**

1. **import pandas as pd**
2. **a\_data = pd.Series(['p', 'q'])**
3. **b\_data = pd.Series(['r', 's'])**
4. **pd.concat([a\_data, b\_data], ignore\_index=True)**

**Output**

**0 p**

**1 q**

**2 r**

**3 s**

**dtype: object**

**Example 3: We can add a hierarchical index at the outermost level of the data by using the keys parameter.**

1. **import pandas as pd**
2. **a\_data = pd.Series(['p', 'q'])**
3. **b\_data = pd.Series(['r', 's'])**
4. **pd.concat([a\_data, b\_data], keys=['a\_data', 'b\_data'])**

**Output**

**a\_data 0 p**

**1 q**

**b\_data 0 r**

**1 s**

**dtype: object**

**Example 4: We can label the index keys by using the names parameter. The below code shows the working of names parameter.**

1. **import pandas as pd**
2. **a\_data = pd.Series(['p', 'q'])**
3. **b\_data = pd.Series(['r', 's'])**
4. **pd.concat([a\_data, b\_data], keys=['a\_data', 'b\_data'])**
5. **pd.concat([a\_data, b\_data], keys=['a\_data', 'b\_data'],**
6. **names=['Series name', 'Row ID'])**

**Output**

**Series name Row ID**

**a\_data 0 p**

**1 q**

**b\_data 0 r**

**1 s**

**dtype: object**

Concatenation using append

**The append method is defined as a useful shortcut to concatenate the Series and DataFrame.**

**Example:**

1. **import pandas as pd**
2. **one = pd.DataFrame({**
3. **'Name': ['Parker', 'Smith', 'Allen', 'John', 'Parker'],**
4. **'subject\_id':['sub1','sub2','sub4','sub6','sub5'],**
5. **'Marks\_scored':[98,90,87,69,78]},**
6. **index=[1,2,3,4,5])**
7. **two = pd.DataFrame({**
8. **'Name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],**
9. **'subject\_id':['sub2','sub4','sub3','sub6','sub5'],**
10. **'Marks\_scored':[89,80,79,97,88]},**
11. **index=[1,2,3,4,5])**
12. **print (one.append(two))**

**Output**

**Name subject\_id Marks\_scored**

**1 Parker sub1 98**

**2 Smith sub2 90**

**3 Allen sub4 87**

**4 John sub6 69**

**5 Parker sub5 78**

**1 Billy sub2 89**

**2 Brian sub4 80**

**3 Bran sub3 79**

**4 Bryce sub6 97**

**5 Betty sub5 88**

# Python Pandas Data operations

In Pandas, there are different useful data operations for DataFrame, which are as follows :

**Row and column selection**

We can select any row and column of the DataFrame by passing the name of the rows and column. When you select it from the DataFrame, it becomes one-dimensional and considered as Series.

5M

572

Features of Java - Javatpoint

**Next**

**Stay**

**Filter Data**

We can filter the data by providing some of the boolean expression in DataFrame.

#### **Note: If we want to pass the boolean results into a DataFrame, then it shows all the results.**

**Null values**

A Null value can occur when no data is being provided to the items. The various columns may contain no values which are usually represented as NaN. In Pandas, several useful functions are available for detecting, removing, and replacing the null values in Data Frame. These functions are as follows:

**isnull():** The main task of isnull() is to return the true value if any row has null values.

**notnull():** It is opposite of isnull() function and it returns true values for not null value.

**dropna():** This method analyzes and drops the rows/columns of null values.

**fillna():** It allows the user to replace the NaN values with some other values.

**replace():** It is a very rich function that replaces a string, regex, series, dictionary, etc.

**interpolate():** It is a very powerful function that fills null values in the DataFrame or series.

**String operation**

A set of a string function is available in Pandas to operate on string data and ignore the missing/NaN values. There are different string operation that can be performed using **.str.** option. These functions are as follows:

**lower():** It converts any strings of the series or index into lowercase letters.

**upper():** It converts any string of the series or index into uppercase letters.

**strip():** This function helps to strip the whitespaces including a new line from each string in the Series/index.

**split(' '):** It is a function that splits the string with the given pattern.

**cat(sep=' '):** It concatenates series/index elements with a given separator.

**contains(pattern):** It returns True if a substring is present in the element, else False.

**replace(a,b):** It replaces the value a with the value b.

**repeat(value):** It repeats each element with a specified number of times.

**count(pattern):** It returns the count of the appearance of a pattern in each element.

**startswith(pattern):** It returns True if all the elements in the series starts with a pattern.

**endswith(pattern):** It returns True if all the elements in the series ends with a pattern.

**find(pattern):** It is used to return the first occurrence of the pattern.

**findall(pattern):** It returns a list of all the occurrence of the pattern.

**swapcase:** It is used to swap the case lower/upper.

**islower():** It returns True if all the characters in the string of the Series/Index are in lowercase. Otherwise, it returns False.

**isupper():** It returns True if all the characters in the string of the Series/Index are in uppercase. Otherwise, it returns False.

**isnumeric():** It returns True if all the characters in the string of the Series/Index are numeric. Otherwise, it returns False.

**Count Values**

This operation is used to count the total number of occurrences using '**value\_counts()**' option.

**Plots**

Pandas plots the graph with the **matplotlib** library. The **.plot()** method allows you to plot the graph of your data.

**.plot()** function plots index against every column.

You can also pass the arguments into the **plot()** function to draw a specific column.

Pandas DataFrame.dropna()

If your dataset consists of null values, we can use the dropna() function to analyze and drop the rows/columns in the dataset.

Syntax:

1. DataFrameName.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)

Parameters:

* **axis :** {0 or 'index', 1 or 'columns'}, default value 0  
  It takes **int** or **string** values for rows/columns. The input can be 0 and 1 for the integers and **index** or **columns** for the string.
  + **0, or 'index':** Drop the rows which contain missing values.
  + **1, or 'columns':** Drop the columns which contain the missing value.
* **how :**  
  It determines if row or column is removed from DataFrame when we have at least one NA or all NA.  
  It takes a string value of only two kinds ('any' or 'all').
  + **any:** It drops the row/column if any value is null.
  + **all:** It drops only if all values are null.
* **thresh:**  
  It takes integer value that defines the minimum amount of NA values to drop.
* **subset:**  
  It is an array that limits the dropping process to passed rows/columns through the list.
* **inplace:**  
  It returns a boolean value that makes the changes in data frame itself if it is True.

Returns

It returns the DataFrame from which NA entries has been dropped.

Pandas DataFrame.fillna()

We can use the fillna() function to fill the null values in the dataset.

Syntax:

1. DataFrame.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, \*\*kwargs)

Parameters:

* **value:** It is a value that is used to fill the null values, alternately a Series/dict/DataFrame.
* **method:** A method that is used to fill the null values in the reindexed Series.
* **axis:** It takes int or string value for rows/columns. Axis along which we need to fill missing values.
* **inplace:** If it is True, it fills values at an empty place.
* **limit:** It is an integer value that specifies the maximum number of consecutive forward/backward NaN value fills.
* **downcast:** It takes a dict that specifies what to downcast like Float64 to int64.

Returns:

It returns an object in which the missing values are being filled.

# Pandas DataFrame.replace()

Pandas replace() is a very rich function that is used to replace a **string, regex, dictionary, list,** and **series** from the DataFrame. The values of the DataFrame can be replaced with other values dynamically. It is capable of working with the Python regex(regular expression).

It differs from updating with **.loc** or **.iloc**, which requires you to specify a location where you want to update with some value.

## Syntax:

1. DataFrame.replace(to\_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad', axis=None)

## Parameters:

* **to\_replace:** Defines a pattern that we are trying to replace in dataframe.
* **value:** It is a value that is used to fill holes in the DataFrame (e.g., 0), alternately a dict of values that specify which value to use for each column (columns not in the dict will not be filled).  
  It also allow such objects of regular expressions, strings, and lists or dicts, etc.
* **inplace:** If it is True, then it replaces in place.

#### **Note: It will also modify any other views on this object (e.g., a column from a DataFrame). Returns the caller if this is True.**

* **limit:** It defines the maximum size gap to forward or backward fill.
* **regex:** It checks whether to interpret to\_replace and/or value as regular expressions. If it is True, then to\_replace must be a string. Otherwise, **to\_replace** must be None because this parameter will be interpreted as a regular expression or a list, dict, or array of regular expressions.
* **method:** It is a method to use for replacement when to\_replace is a list.

**Returns:** It returns a DataFrame object after the replacement.

2.6M

340

Abstract Class vs Interface | Difference between Abstract class and Interface in Java

### Example1:

1. **import** pandas as pd
2. info = pd.DataFrame({'Language known': ['Python', 'Android', 'C', 'Android', 'Python', 'C++', 'C']},
3. index=['Parker', 'Smith', 'John', 'William', 'Dean', 'Christina', 'Cornelia'])
4. print(info)
5. dictionary = {"Python": 1, "Android": 2, "C": 3, "Android": 4, "C++": 5}
6. info1 = info.replace({"Language known": dictionary})
7. print("\n\n")
8. print(info1)

**Output**

Language known

Parker Python

Smith Android

John C

William Android

Dean Python

Christina C++

Cornelia C

Language known

Parker 1

Smith 4

John 3

William 4

Dean 1

Christina 5

Cornelia 3

### Example2:

The below example replaces a value with another in a DataFrame.

1. **import** pandas as pd
2. info = pd.DataFrame({
3. 'name':['Parker','Smith','John'],
4. 'age':[27,34,31],
5. 'city':['US','Belgium','London']
6. })
7. info.replace([29],38)

**Output**

name age City

0 Parker 27 US

1 Smith 34 Belgium

2 John 38 London

### Example3:

The below example replaces the values from a dict:

1. **import** pandas as pd
2. info = pd.DataFrame({
3. 'name':['Parker','Smith','John'],
4. 'age':[27,34,31],
5. 'city':['US','Belgium','London']
6. })
7. info.replace({
8. 34:29,
9. 'Smith':'William'
10. })

**Output**

name age City

0 Parker 27 US

1 William 29 Belgium

2 John 31 London

### Example4:

The below example replaces the values from regex:

1. **import** pandas as pd
2. info = pd.DataFrame({
3. 'name':['Parker','Smith','John'],
4. 'age':[27,34,31],
5. 'city':['US','Belgium','London']
6. })
7. info.replace('Sm.+','Ela',regex=True)

**Output**

name age City

0 Parker 27 US

1 Ela 34 Belgium

2 John 31 London

Pandas DataFrame.loc[]

The **DataFrame.loc[]** is used to retrieve the group of rows and columns by labels or a boolean array in the DataFrame. It takes only index labels, and if it exists in the caller DataFrame, it returns the rows, columns, or DataFrame.

The **DataFrame.loc[]** is a label based but may use with the boolean array.

The allowed inputs for **.loc[]** are:

4.1M

470

Package in Java

**Next**

**Stay**

* Single label, e.g.,**7** or **a**. Here, **7** is interpreted as the label of the index.
* List or array of labels, e.g. ['x', 'y', 'z'].
* Slice object with labels, e.g. 'x':'f'.
* A boolean array of the same length. e.g. [True, True, False].
* **callable** function with one argument.

Syntax

1. pandas.DataFrame.loc[]

Parameters

None

Returns

It returns Scalar, Series or DataFrame.

Example

**# importing pandas as pd**

1. **import** pandas as pd
2. # Creating the DataFrame
3. info = pd.DataFrame({'Age':[32, 41, 44, 38, 33],
4. 'Name':['Phill', 'William', 'Terry', 'Smith', 'Parker']})
5. # Create the index
6. index\_ = ['Row\_1', 'Row\_2', 'Row\_3', 'Row\_4', 'Row\_5']
8. # Set the index
9. info.index = index\_
11. # **return** the value
12. **final** = info.loc['Row\_2', 'Name']
14. # Print the result
15. print(**final**)

**Output:**

William

Example2:

1. # importing pandas as pd
2. **import** pandas as pd
3. # Creating the DataFrame
4. info = pd.DataFrame({"P":[28, 17, 14, 42, None],
5. "Q":[15, 23, None, 15, 12],
6. "R":[11, 23, 16, 32, 42],
7. "S":[41, None, 34, 25, 18]})
8. # Create the index
9. index\_ = ['A', 'B', 'C', 'D', 'E']
10. # Set the index
11. info.index = index\_
12. # Print the DataFrame
13. print(info)

**Output:**

P Q R S

A 28.0 15.0 11 41.0

B 17.0 23.0 23 NaN

C 14.0 NaN 16 34.0

D 42.0 15.0 32 25.0

E NaN 12.0 42 18.0

Now, we have to use **DataFrame.loc** attribute to return the values present in the DataFrame.

1. # **return** the values
2. result = info.loc[:, ['P', 'S']]
3. # Print the result
4. print(result)

**Output:**

P S

A 28.0 41.0

B 17.0 NaN

C14.0 34.0

D 42.0 25.0

ENaN 18.0

Pandas DataFrame.iloc[]

The **DataFrame.iloc[]** is used when the index label of the DataFrame is other than numeric series of 0,1,2,....,n or in the case when the user does not know the index label.

We can extract the rows by using an imaginary index position which is not visible in the DataFrame. It is an integer- based position(from 0 to length-1 of the axis), but may also be used with the boolean array.

The allowed inputs for **.loc[]** are:

Method Overloading vs Overriding in Java

* Integer value, e.g. 7.
* List or array of integers, e.g [2, 5, 6].
* Slice object with ints, e.g., 1:9.
* boolean array.
* A callable function with one argument that can be the calling Series or the DataFrame. It returns valid outputs for indexing.

It can raise the **IndexError** if we request the index is out-of-bounds, except slice indexers, which allow the out-of-bounds indexing.

Syntax:

1. pandas.DataFrame.iloc[]

Parameters:

None

Returns:

It returns the DataFrame or the Series.

Example:

1. **import** pandas as pd
2. a = [{'p': 2, 'q': 4, 'r': 6, 's': 8},
3. {'a': 200, 'b': 400, 'c': 600, 'd': 800},
4. {'p': 2000, 'q': 4000, 'r': 6000, 's': 8000 }]
5. info = pd.DataFrame(mydict)
6. type(info.iloc[0])
7. <**class** 'pandas.core.series.Series'>
8. info.iloc[0]

**Output:**

a1

b2

c 3

d4

Name: 0, dtype: int64

Pandas loc vs. iloc

The Pandas offers .loc[] and .iloc[] methods for **data slicing**. Data Slicing generally refers to inspect your data sets. These two methods belong to the index selection method that is used to set an identifier for each row of the data set. The indexing can take specific labels, and these labels can either be an integer or any other value specified by the user.

The .**loc[]** method is used to retrieve the group of rows and columns by labels or a boolean array present in the DataFrame. It takes only index labels, and if it exists in the caller DataFrame, it returns the rows, columns, or DataFrame. It is a label-based method but may be used with the boolean array.

Whereas, the **.iloc[]** method is used when the index label of the DataFrame is other than numeric series of 0,1,2,....,n, or in the case when the user does not know the index label.

Features of Java - Javatpoint

There are some differences between the above methods, which are given below:

1. The **.loc[]**method is a **label based** method that means it takes names or labels of the index when taking the slices, whereas **.iloc[]**method is based on the **index's position**. It behaves like a regular slicing where we just have to indicate the positional index number and simply get the appropriate slice.
2. The .**loc[]**method includes the last element of the table whereas **.iloc[]**method does not include the last element.
3. The .**loc[]**method is a **name-based indexing,**whereas the **.iloc[]**method is **positional based indexing**.
4. The arguments of .**iloc[]**can be:
   * list of rows and columns
   * range of rows and columns
   * single row and column

Whereas, the arguments of **.loc[]** can be:

* + row label
  + list of row label

1. The **.loc[]** method indexer can perform the boolean selection by passing the boolean series, but in the case of **.iloc[]**method, we cannot pass a boolean series.