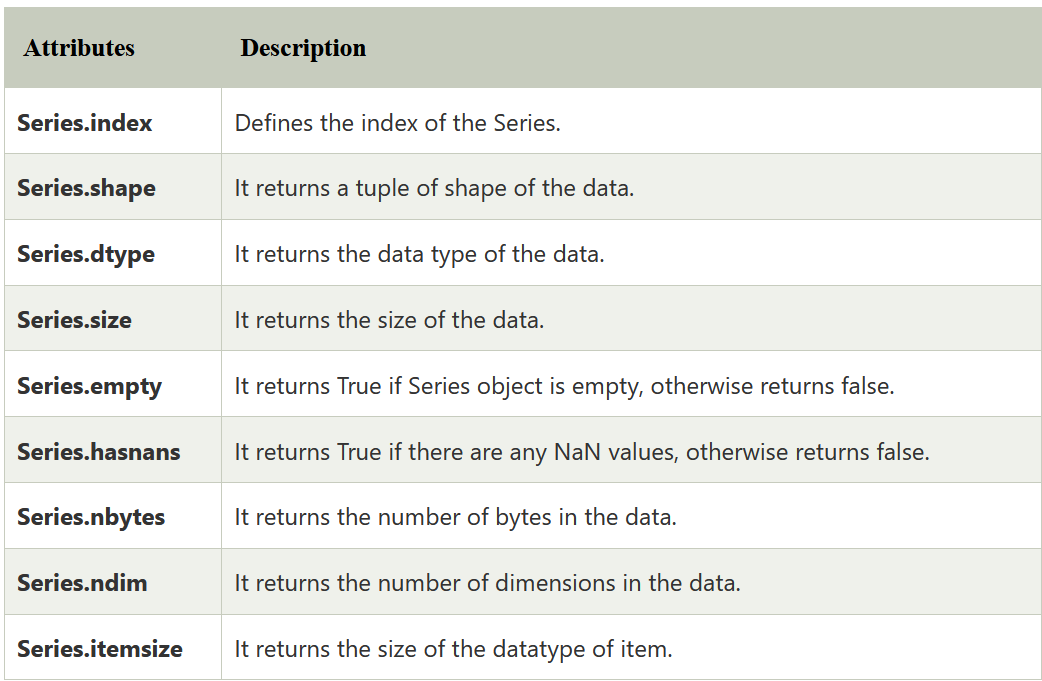
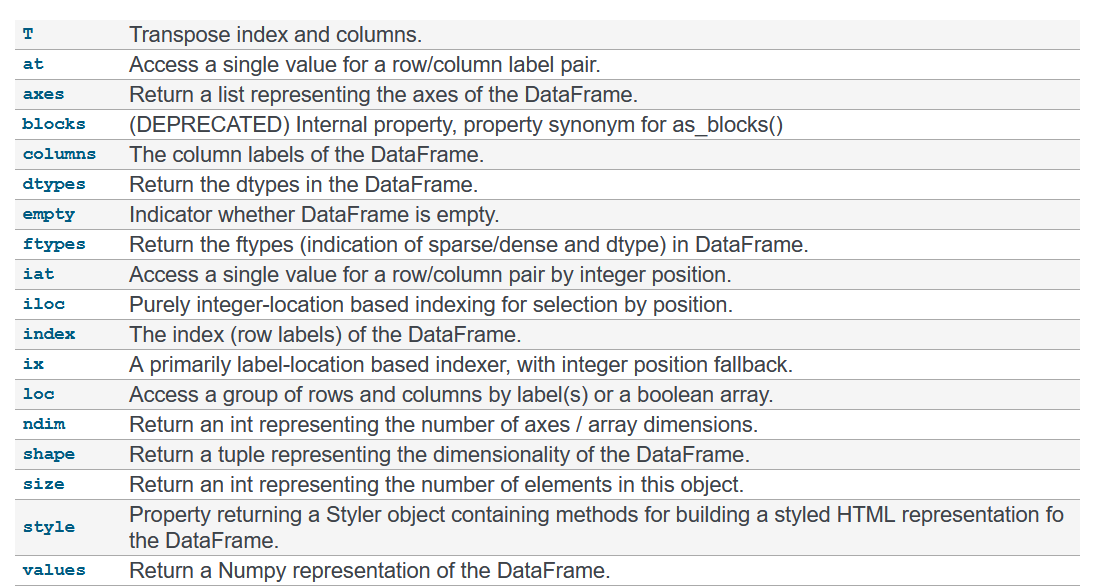
Pandas Series attributes:

Pandas series have below attributes-



**Pandas Dataframe attributes**

Pandas dataframe have below attributes-



Deciding axis in pandas:

+------------+---------+--------+

| | A | B |

+------------+---------+---------

| 0 | 0.626386| 1.52325|----axis=1----->

+------------+---------+--------+

| |

| axis=0 |

↓ ↓

Axis=0 -------> means for each column

Axis=1 -------> means for each row

**Pandas Index**

Pandas Index is an immutable ndarray implementing an ordered, sliceable set. It is the basic object which stores the axis labels for all pandas objects.

pandas.Index(data=None, dtype=None, copy=False, name=None, tupleize\_cols=True, \*\*kwargs)

**data**: array-like (1-dimensional)

**dtype**: NumPy dtype (default: object), If dtype is None, we find the dtype that best fits the data. If an actual dtype is provided, we coerce to that dtype if it’s safe. Otherwise, an error will be raised.

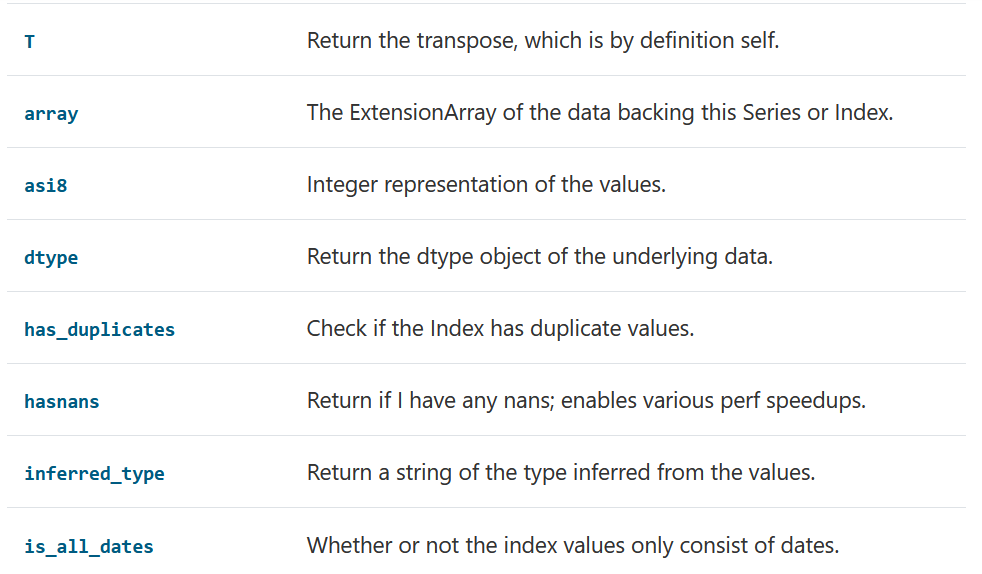
**copy**: bool, Make a copy of input ndarray.

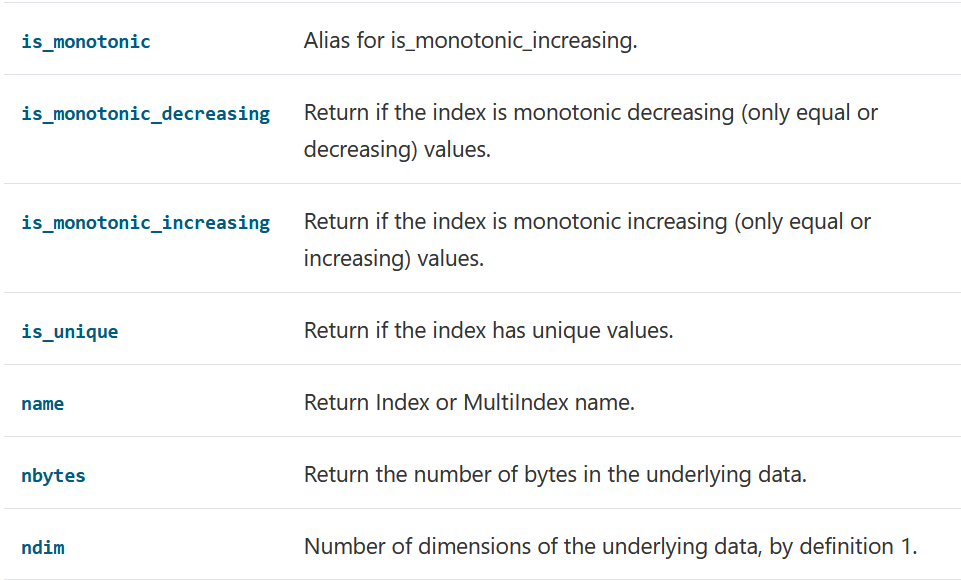
idx = pd.Index(['Jan', 'Feb', 'Mar', 'Apr', 'May'])

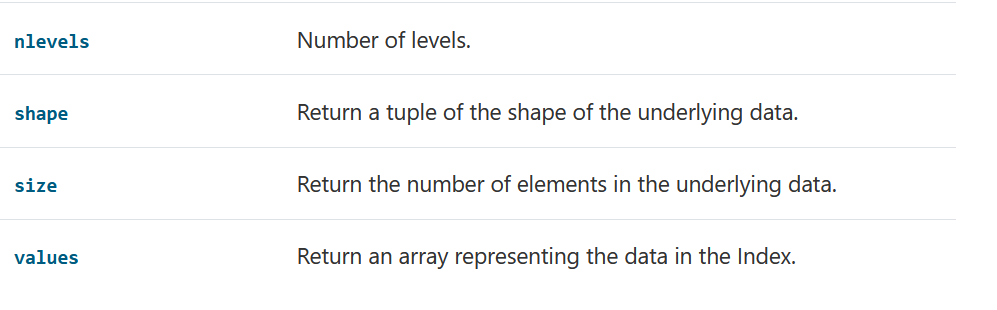
# Print the index

print(idx)

**Attributes of pandas Index**







**Selection methods for pandas Indexes**

Index.get\_loc(key, method=None, tolerance=None)

Returns loc: int if unique index, slice if monotonic index, else mask

**Key**: labels or datas

**default**: exact matches only.

**pad / ffill**: find the PREVIOUS index value if no exact match.

**backfill / bfill**: use NEXT index value if no exact match

**nearest**: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

**tolerance**: int or float, optional

**Example**

pd.Index(list('abc'))

unique\_index.get\_loc('b') #1

**Example**

monotonic\_index = pd.Index(list('abbc'))

monotonic\_index.get\_loc('b') #slice(1, 3, None)

**Example**

non\_monotonic\_index = pd.Index(list('abcb'))

non\_monotonic\_index.get\_loc('b') # array([False, True, False, True]) --- this is mask

Index.get\_value(series, key) ------ Returns series or scalar

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you’re doing.

Index.get\_slice\_bound(label, side, kind=None)

Calculate slice bound that corresponds to given label.

Returns leftmost (one-past-the-rightmost if side=='right') position of given label.

Returns: int, index od label

Index.isin(values, level=None)

Return a boolean array where the index values are in values.

Index.slice\_locs(start=None, end=None, step=None, kind=None)

Compute slice locations for input labels. Returns start,end as integer

start: label, default None

end: label, default None

step: int, defaults None

In output first values will be index value of start argument and second value will be position value of end argument.

**Note:**

This method only works if the index is monotonic or unique.

Example:

idx = pd.Index(list('abcd'))

idx.slice\_locs(start='b', end='c') #(1, 3) 1--> index of b, 3 --> position of c

In output first values will be index value of start argument and second value will be position value of end argument.

**Question ------ Good**

Write a Pandas program to get the positions of items of a given series(series2) in another given series(series1).

<https://www.w3resource.com/python-exercises/pandas/python-pandas-data-series-exercise-23.php>

series1 = pd.Series([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

series2 = pd.Series([1, 3, 5, 7, 10])

print("Original Series:")

print(series1)

print(series2)

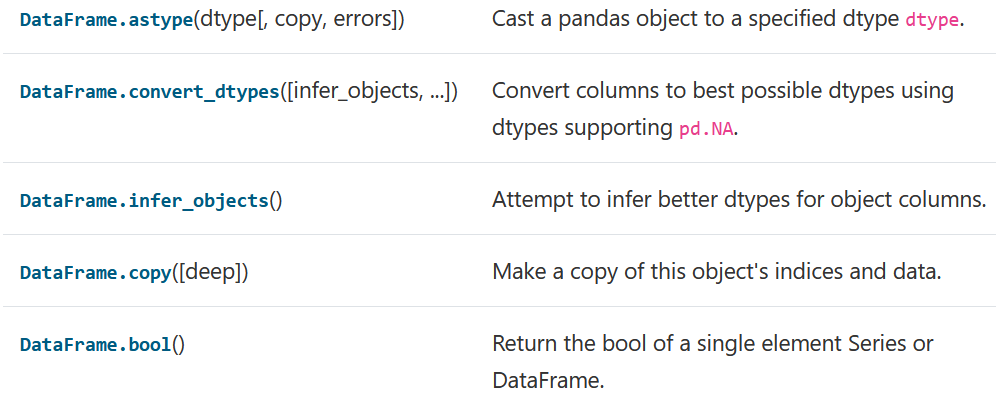
**Solution**

result = [pd.Index(series1).get\_loc(i) for i in series2]

print("Positions of items of series2 in series1:")

print(result)

**Dataframe conversion function**



DataFrame.astype(dtype, copy=True, errors='raise')

Cast a pandas object to a specified dtype dtype.

**dtype**: data type which to be casted, it's numpy.dtype or Python type.

**copy**: bool, default True, Returns a copy when copy=True

**errors**: {‘raise’, ‘ignore’}, default. Control raising of exceptions on invalid data for provided dtype.

raise : allow exceptions to be raised

ignore : suppress exceptions. On error return original object.

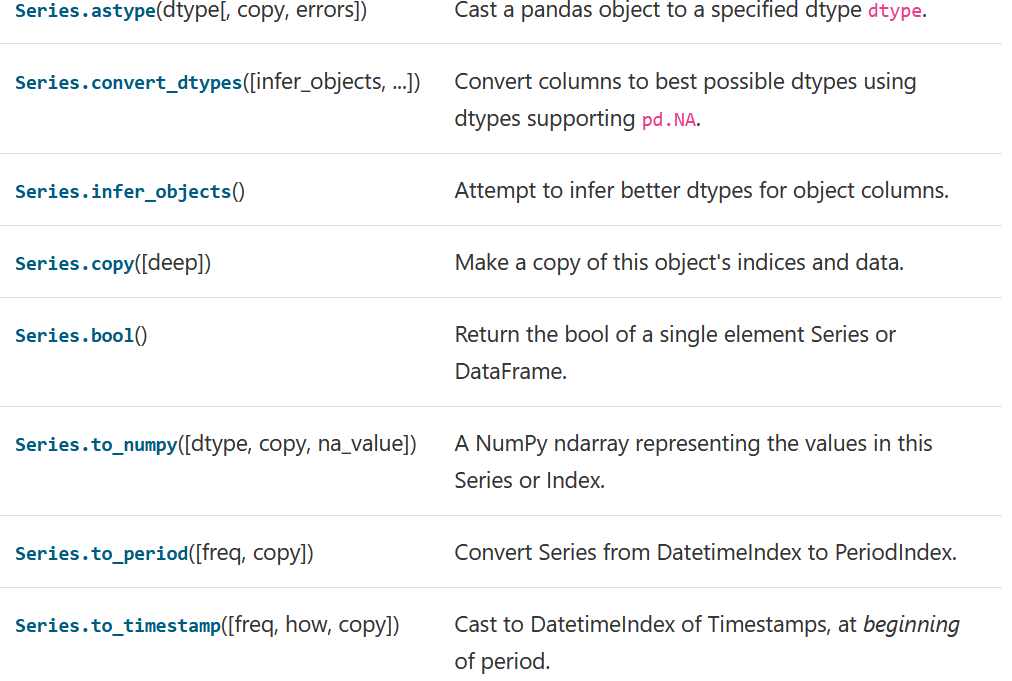
DataFrame.copy(deep=True)

Make copy (deep or shallow copy) of dataframe and return it.

**deep**: bool, default True

**Series Conversion functions**

We have below methods for conversion will see few of them.





Series.to\_list() ------- Values in list form.

Series.to\_numpy ------ Values as ndarrary

Series.values ----- Values as ndarray

Series.astype() --- same as df

Series.copy() ---- same as df

Series.to\_numpy(dtype=None, copy=False, na\_value=NoDefault.no\_default, \*\*kwargs)

Gives a NumPy ndarray representing the values in this Series or Index.

*dtype*: str or numpy.dtype, optional

*copy*: bool, default False

*na\_value* : Any, optional

Series.to\_timestamp(freq=None, how='start', copy=True)

Cast to DatetimeIndex of Timestamps, at beginning of period

Series.to\_list()

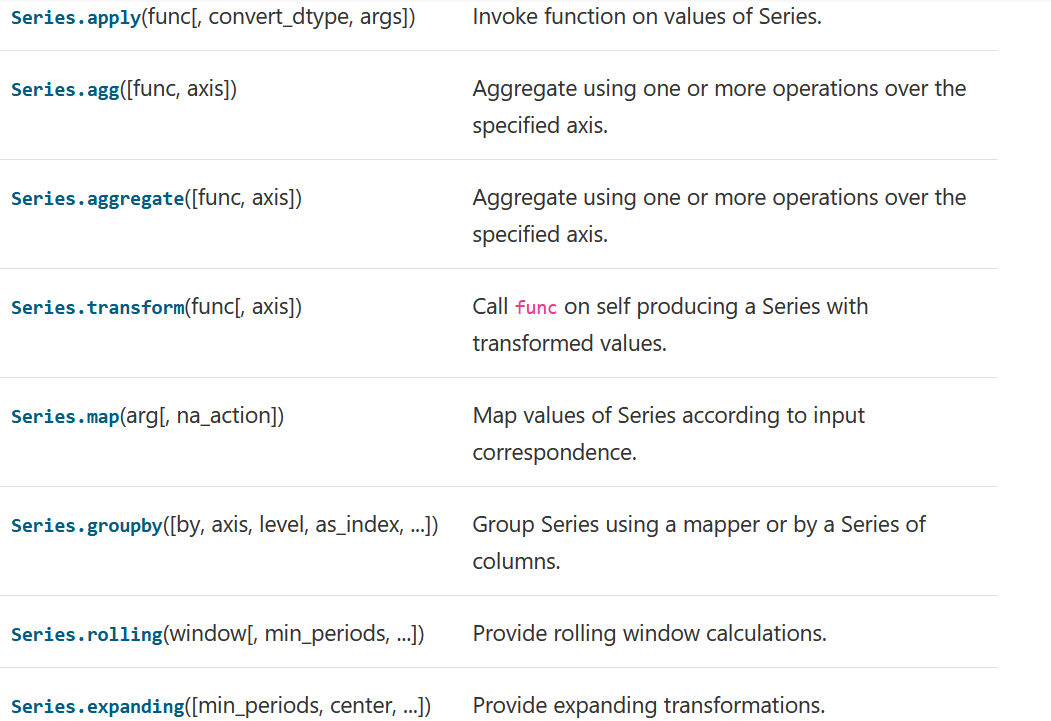
Convert numpy series to a list types and returns a list of values.

##########################################

# Series Function application, GroupBy & window #

##########################################

We have below function but we will see few commonly used.



Series.apply(func, convert\_dtype=True, args=(), \*\*kwargs)

It invokes ‘func’ on on called series and **returns dataframe or series**. Function can be ufunc(universal function) or Python function that only works on single values.

*func*: function --Python function or NumPy ufunc to apply

*args*: tuple --Positional arguments passed to func after the series value.

If func returns a Series object the result will be a DataFrame.

**Question**:

Below Series contains temperature for given city, square the temperature value for each city.

s = pd.Series([20, 21, 12],index=['London', 'New York', 'Helsinki'])

Answer:

print(s.apply(lambda x:x\*x))

**Question**:

For above temperature series, add 5 in temperature.

Answer:

We create a custom method for adding and will pass 5 as argument. ---- **Try using transform()**

def my\_add(x,arg):

    return x+5

print(s.apply(my\_add,args=(5,)))

s.apply(lambda x:x+5) -------- Using lambda function

Series.agg(func=None, axis=0, \*args, \*\*kwargs)

Aggregate using one or more operations over the specified axis.

It returns scalar, Series or DataFrame

*scalar* : when Series.agg is called with single function

*Series* : when DataFrame.agg is called with a single function or series with many function

*DataFrame* : when DataFrame.agg is called with several functions

*func*: function, str, list or dict----Function to use for aggregating the data.

*axis*: {0 or ‘index’} ---Parameter needed for compatibility with DataFrame.

*\*args*: ---Positional arguments to pass to func.

Example:

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

s.agg('min') # 1------------ it will be scalar

s.agg(['min', 'max']) #this will be series

min 1

max 4

dtype: int64

Series.aggregate(func=None, axis=0, \*args, \*\*kwargs)

Aggregate using one or more function. Exactly same as agg, agg is alias of aggregate.

Series.transform(func, axis=0, \*args, \*\*kwargs)

Call func on self producing a Series with transformed values. Produced Series will have same axis length as self. It A Series that must have the same length as self.

*func*: function, str, list-like or dict-like.

*axis*: {0 or ‘index’}

*\*args*: Positional arguments to pass to func.

**Example**:

For given dataframe add 1 for each value.

df = pd.DataFrame({'A': range(3), 'B': range(1, 4)})

Asnwer:

df.transform(lambda x: x + 1) #

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

Map values of Series according to input correspondence and retuns series

Used for substituting each value in a Series with another value, that may be derived from a function.

*map accepts a dict or a Series.*

*arg*: function, collections.abc.Mapping subclass or Series

Mapping correspondence.

*na\_action*: {None, ‘ignore’}, default None

If ‘ignore’, propagate NaN values, without passing them to the mapping correspondence.

Example:

For given series map cat to kitten and dog to puppy.

s = pd.Series(['cat', 'dog', np.nan, 'rabbit'])

Answer:

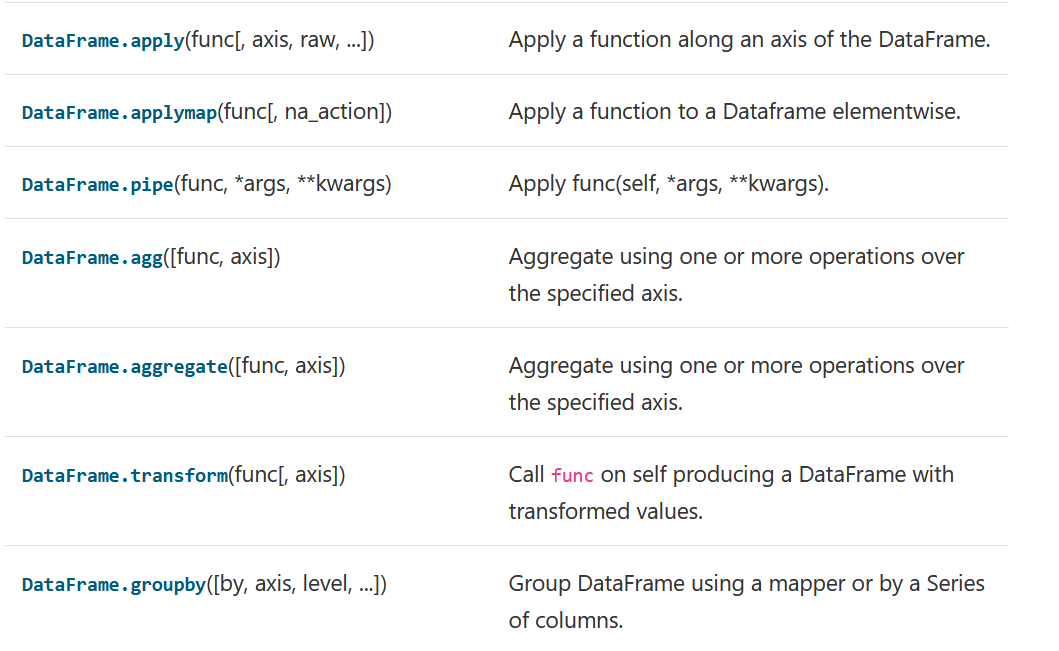
|  |  |
| --- | --- |
| print(s.map({'cat': 'kitten', 'dog': 'puppy'})) |  |

##############################################

# Dataframe Function application, GroupBy & window #

##############################################

We have below methods for dataframe grouping and application. They are almost same as that of series with minor defference. We will check few



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

Apply a function (func) along an axis of the DataFrame.

*func*: function --- function to be applied

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

*0 or ‘index’: apply function to each column*

1 or ‘columns’: apply function to each row

Example:

For below dataframe calculate sum for each row and each column.

df = pd.DataFrame([[4, 9]] \* 3, columns=['A', 'B'])

For each column

|  |  |
| --- | --- |
| print(df.apply(np.sum)) #defaults for col |  |

For each row

|  |  |
| --- | --- |
| print(df.apply(np.sum,axis=1))  axis=1 means for each row |  |

DataFrame.applymap(func, na\_action=None, \*\*kwargs)

Apply a function to a Dataframe **elementwise or on each element.**

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

----------This is same as of pandas series map function.

Example:

Squre the value of each element of given dataframe.

pd.DataFrame([[1, 2.12], [3.356, 4.567]])

Answer:

df.applymap(lambda x: x\*\*2)

DataFrame.agg(func=None, axis=0, \*args, \*\*kwargs)

Aggregate using one or more operations over the specified axis.

It’s same as series agg/aggregate function, additionally we have to specify the axis.

It returns scalar, Series or DataFrame.

*scalar* : when Series.agg is called with single function

*Series* : when DataFrame.agg is called with a single function

*DataFrame* : when DataFrame.agg is called with several functions

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

*0 or ‘index’: apply function to each column.*

*1 or ‘columns’: apply function to each row.*

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

Call func on self producing a DataFrame with transformed values.

Produced DataFrame will have same axis length as self.

It accepts same parameter as that of series with additional axis argument.

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

*0 or ‘index’: apply function to each column.*

*1 or ‘columns’: apply function to each row.*

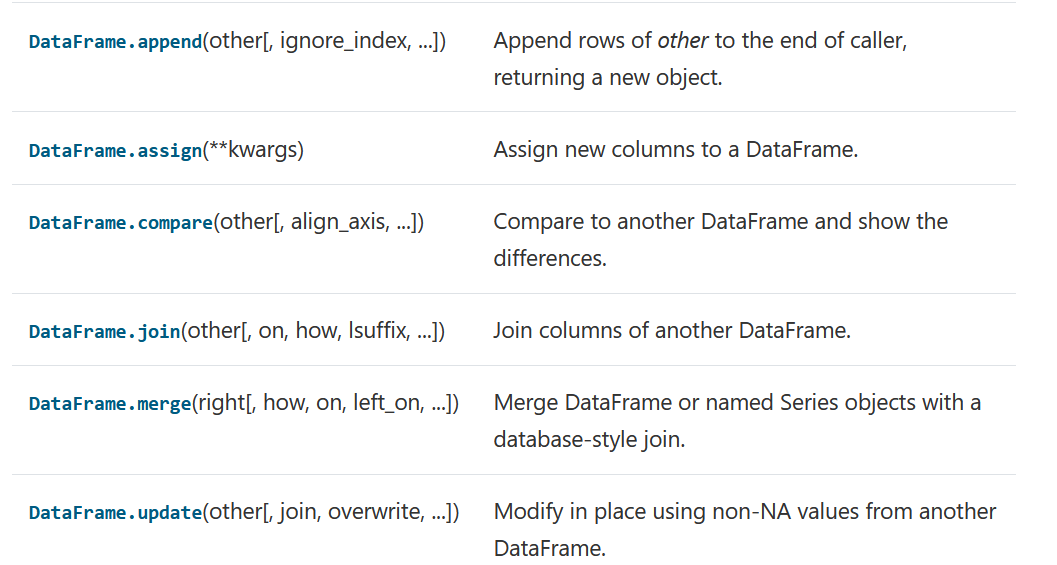
It returns dataframe that have same length as that of self.

################################################

# Dataframe Combining / comparing / joining / merging #

################################################

We have below methods for combining/comparing/joining purpose.



DataFrame.append(other, ignore\_index=False, verify\_integrity=False, sort=False)

Append rows of 'other' to the end of caller, returning a new object/DataFrame .

*Columns in 'other' that are not in the caller are added as new columns.*

*other*: DataFrame or Series/dict-like object, or list of these --- Data to append

*ignore\_index*: bool,---default False, If True, the resulting axis will be labeled 0, 1, …, n - 1

*verify\_integrity*: bool, default False, ---If True, raise ValueError on creating index with duplicates.

**CASE:** **If column name same**

df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'), index=['x', 'y'])

df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'), index=['p', 'q'])

df1.append(df2)

|  |  |
| --- | --- |
| df.append(df2) | A B  x 1 2  y 3 4  p 5 6  q 7 8 |

**Case 2**: **If column names are not same**

If any column which is not common in both is added as new column name in result.

df1 = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'), index=['x', 'y'])

df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AC'), index=['x', 'y'])

df1.append(df2,ignore\_index=False)

|  |  |
| --- | --- |
| df1.append(df2) |  |

DataFrame.assign(\*\*kwargs)

Assign new columns to a DataFrame and returns a new object with all original columns in addition to new ones.

Existing columns that are re-assigned will be overwritten.

\*\*kwargs: dict of {str: callable or Series}

The column names are keywords. If the values are callable, they are computed on the DataFrame and assigned to the new columns.

**Note:**

We can assign/create new column and assign value in below way –

df[‘col\_name’]=[new\_value]

df.insert (location , col\_name, value)

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

Join columns of another DataFrame. Join columns with other DataFrame either on index or on a key column. **Requires a common column name in both dataframe.**

*other*: DataFrame, Series, or list of DataFrame

*on*: str,list of str, or array-like, **optional,**

Column or index level name(s) in the caller to join on the index in other, otherwise joins index-on-index.

*how*: {‘left’, ‘right’, ‘outer’, ‘inner’}, default ‘left’

left ---- all from left and common in both

right --- all from right and common in both

outer --- all from both dataframe

inner --- common in both dataframe

*lsuffix*: str, default ‘’---Suffix to use from left frame’s overlapping columns.

*rsuffix*: str, default ‘’---Suffix to use from right frame’s overlapping columns.

**Note:**

* *Join needs common column name in both dataframe* on which join will be performed.
* *append doesn’t need any common column name in both dataframe*.

Example1:

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],

        'Age':[27, 24, 22, 32],

        'Key':['K0', 'K1', 'K2', 'K3']}

data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],

        'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}

df = pd.DataFrame(data1)

print(df)

df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])

print(df1)

res2 = df.join(df1, on='Key')

print(res2)

|  |  |
| --- | --- |
| Left dataframe : df    Right dataframe: df1 | Result |

**Note:**

*If for common column name, data are not same in both dataframes then use lsuffix and rsuffix.*

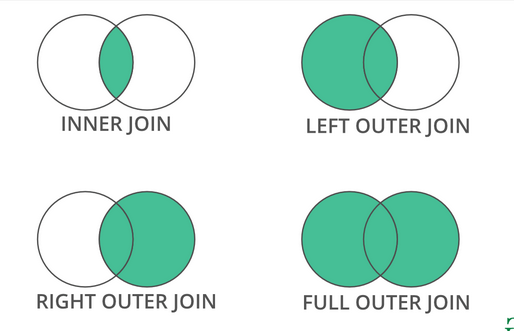
df = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})

other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],'B': ['B0', 'B1', 'B2']})

|  |  |
| --- | --- |
| print(df.join(df2,lsuffix='\_caller', rsuffix='\_other')) | key\_caller A key\_other B  0 K0 A0 K0 B0  1 K1 A1 K1 B1  2 K2 A2 K2 B2  3 K3 A3 NaN NaN  4 K4 A4 NaN NaN  5 K5 A5 NaN NaN |

DataFrame.merge(right, how='inner', on=None, left\_on=None, right\_on=None, left\_index=False, right\_index=False, sort=False, suffixes=('\_x', '\_y'), copy=True, indicator=False, validate=None)

Merge DataFrame or named Series objects with a database-style join. There are 4 way to handle it



It *doesn’t require common column names in both dataframe.*

*Returns DataFrame of the two merged objects*

right: DataFrame or named Series --- Object to merger with

how: {‘left’, ‘right’, ‘outer’, ‘inner’, ‘cross’}, **default ‘inner’**

left ---- all from left and common in both

right --- all from right and common in both

outer --- all from both dataframe

inner --- common in both dataframe , unique condition

*on*: label or list

*suffixes*: list-like, default is (“\_x”, “\_y”) --- **default it adds \_x and \_y for left and right dataframes**

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

column names from left and right dataframe on which merge will be performed.

Example: Do the merge operation on below two df

left = pd.DataFrame({

   'id':[1,2,3,4,5],

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

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

right = pd.DataFrame({

    'id':[1,2,3,4,5],

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

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

print(left)

print(right)

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

|  |  |
| --- | --- |
| Left    Right | Result |

**Question**:

Merger below dfs on Merge df1 and df2 on the lkey and rkey columns.

df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],'value': [1, 2, 3, 5]})

df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],'value': [5, 6, 7, 8]})

|  |  |
| --- | --- |
| **df1**  lkey value  0 foo 1  1 bar 2  2 baz 3  3 foo 5  **df2**  rkey value  0 foo 5  1 bar 6  2 baz 7  3 foo 8 | print(df1.merge(df2, left\_on='lkey', right\_on='rkey'))  lkey value\_x rkey value\_y  0 foo 1 foo 5  1 foo 1 foo 8  2 foo 5 foo 5  3 foo 5 foo 8  4 bar 2 bar 6  5 baz 3 baz 7 |

**##############**

**# Merge vs Join #**

**##############**

|  |  |
| --- | --- |
| **Join**  The **join** method takes two dataframes and joins them on their indexes (technically, you can pick the column to join on for the left dataframe). If there are overlapping columns, the join will want you to add a suffix to the overlapping column name from the left dataframe. | **merge**  At a basic level, **merge** more or less does the same thing as join. Both methods are used to combine two dataframes together, but merge is more versatile, it requires specifying the columns as a merge key. We can specify the overlapping columns with parameter ***on,*** or can separately specify it with ***left\_on*** and ***right\_on*** parameters.  If col names are overlapping then adds ‘\_x’ and ‘\_y’ to differentiate left and right df col names |

DataFrame.update(other, join='left', overwrite=True, filter\_func=None, errors='ignore')

Modify in place using non-NA values from another DataFrame.

#############################################

# Series Combining / comparing / joining / merging #

#############################################

Series.append(to\_append, ignore\_index=False, verify\_integrity=False)

Concatenate two or more Series.

It’s all parameter are same as that of dataframe parameter.

Series.update(other)

Modify Series in place using values from passed Series

##############################

# Dataframe missing data handling #

##############################

There are many methods for this but going to see really few ones.

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

Remove missing values.

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

0, or ‘index’ : Drop rows which contain missing values

1, or ‘columns’ : Drop columns which contain missing value

*how*: {‘any’, ‘all’}, default ‘any’

any’ : If any NA values are present, drop that row or column

‘all’ : If all values are NA, drop that row or column.

*inplace*: bool, default False

If True, do operation inplace and return None.

DataFrame.isna()

Detect missing values. Returns mask of bool values for each element in DataFrame that indicates whether an element is an NA value

DataFrame.isnull()

Detect missing values.Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

DataFrame.notna() /DataFrame.notnull()

Detect existing (non-missing) values. Returns mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

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

Replace values given in 'to\_replace' with 'replace\_by'

*to\_replace*: str, regex, list, dict, Series, int, float, or None

it denotes which values will be replaced

*replaced\_by*: scalar, dict, list, str, regex, default None

it denotes new value for replacement

*inplace*: bool, default False

If True, performs operation inplace and returns None.

*regex*: bool or same types as to\_replace, default False

Whether to interpret to\_replace and/or value as regular expressions.

**Questions**: --- Use of regex

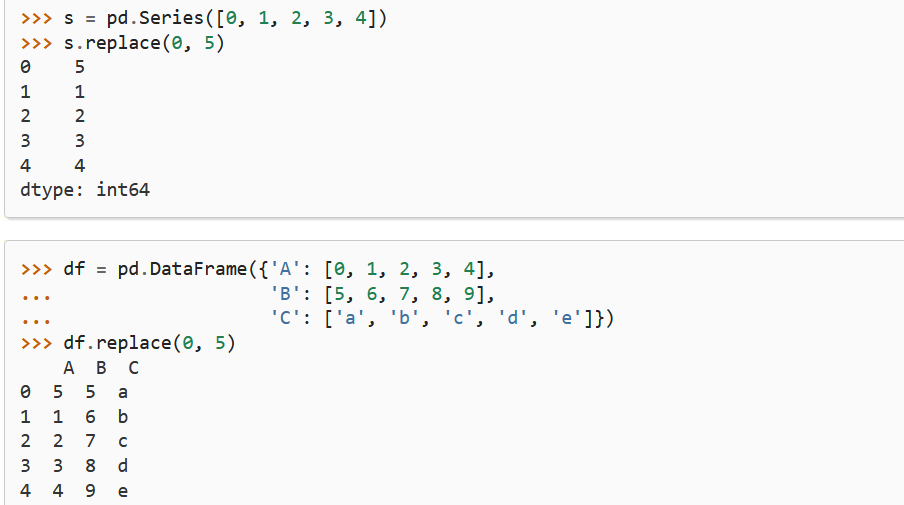
In below dataframe replace all words which starts with ‘b’ followed by ‘a’ ending with anything.

df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],'B': ['abc', 'bar', 'xyz']})

**Answer**

|  |  |
| --- | --- |
| df  A B  0 bat abc  1 foo bar  2 bait xyz | df.replace(to\_replace=r'^ba.$', value='NEW', regex=True,inplace=True)  print(df)  A B  0 NEW abc  1 foo NEW  2 bait xyz |

**Example** 2:



**Questions: ------ Good**

Replace yes and no for qualify column by True and False. For below dataframe by using

1. map function
2. replace function

exam\_data  = {'name': ['Anastasia', 'Dima', 'Katherine', 'James', 'Emily', 'Michael', 'Matthew', 'Laura', 'Kevin', 'Jonas'],

        'score': [12.5, 9, 16.5, np.nan, 9, 20, 14.5, np.nan, 8, 19],

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

        'qualify': ['yes', 'no', 'yes', 'no', 'no', 'yes', 'yes', 'no', 'no', 'yes']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']

<https://www.w3resource.com/python-exercises/pandas/python-pandas-data-frame-exercise-17.php>

**Using map function**

df.qualify=df.qualify.map({'yes':True,'no':False})

**Using relace function**

df.replace(to\_replace='yes',value=True,inplace=True)

df.replace(to\_replace='no',value=False,inplace=True)

##########################

# Series missing data handling #

##########################

It same methods as above one, check above for this.

########################################

# Reshaping, sorting, transposing of Dataframe #

########################################

DataFrame.pivot(index=None, columns=None, values=None)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a “pivot” table) based on column values. *Uses unique values from specified index / columns* to form axes of the resulting DataFrame.

*index*: str or object or a list of str, optional

Column to use to make new frame’s index. If None, uses existing index.

*columns*: str or object or a list of str

Column to use to make new frame’s columns.

*values*: str, object or a list of the previous, optional

Column(s) to use for populating new frame’s values. If not specified, all remaining columns will be used

**Example:**

Pivot below dataframe, index will be – column foo, column will be bar, value will be column baz.

df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two','two'],'bar': ['A', 'B', 'C', 'A', 'B', 'C'],

'baz': [1, 2, 3, 4, 5, 6],'zoo': ['x', 'y', 'z', 'q', 'w', 't']})

Answer:

df.pivot(index='foo', columns='bar', values='baz')

|  |  |
| --- | --- |
| df.pivot(index='foo', columns='bar', values='baz') | **bar** A B C  **foo**  one 1 2 3  two 4 5 6 |

**Note:**

We can see it have taken only unique values.

DataFrame.pivot\_table(values=None, index=None, columns=None, aggfunc='mean', fill\_value=None, margins=False, dropna=True, margins\_name='All', observed=False, sort=True)

Create a spreadsheet-style pivot table as a DataFrame. It accepts aggregate function

Values: column to aggregate, optional

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

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

Example:

We have below dataframe, pivot this table where index will be [‘A’,’B’] for value column D for column C.

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

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

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

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

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

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

Answer:

|  |  |
| --- | --- |
| A B C D E  0 foo one small 1 2  1 foo one large 2 4  2 foo one large 2 5  3 foo two small 3 5  4 foo two small 3 6  5 bar one large 4 6  6 bar one small 5 8  7 bar two small 6 9  8 bar two large 7 9 | df1=pd.pivot\_table(df, values='D', index=['A', 'B'],columns=['C'], aggfunc=np.sum)  print(df1)  A B  bar one 4.0 5.0  two 7.0 6.0  foo one 4.0 1.0  two NaN 6.0 |

DataFrame.sort\_values(by, axis=0, ascending=True, inplace=False, kind='quicksort', na\_position='last', ignore\_index=False, key=None)

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

*level*: int or level name or list of ints or list of level names

*ascending*: bool or list-like of bools, default True

*inplace*: bool, default False

*ignore*\_index: bool, default False

pandas.crosstab(index, columns, values=None, rownames=None, colnames=None, aggfunc=None, margins=False, margins\_name='All', dropna=True, normalize=False)

Compute a simple cross tabulation of two (or more) factors. By default computes a frequency table of the factors unless an array of values and an aggregation function are passed.

##############################################

#Dataframe Reindexing / selection / label manipulation#

##############################################

DataFrame.reindex(labels=None, index=None, columns=None, axis=None, method=None, copy=True, level=None, fill\_value=nan, limit=None, tolerance=None)

Conform Series/DataFrame to **new index** with optional filling logic.

Basically renames the index of dataframe with optional filling logic.

**Note**:

By default, values in the new index that do not have corresponding records in the dataframe are assigned NaN.

keywords for axes: array-like, optional

New labels / index to conform to, should be specified using keywords

method: {None, ‘backfill’/’bfill’, ‘pad’/’ffill’, ‘nearest’}

Method to use for filling holes in reindexed DataFrame.

copy: bool, default True

fill\_value: scalar, default np.NaN

Value to use for missing values. Defaults to NaN

**Question**:

For given below dataframe change the index value to - ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10','Chrome']

index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']

df = pd.DataFrame({'http\_status': [200, 200, 404, 404, 301],'response\_time': [0.04, 0.02, 0.07, 0.08, 1.0]},index=index)

Answer

new\_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10','Chrome']

print(df.reindex(new\_index))

http\_status response\_time

Safari 404.0 0.07

Iceweasel NaN NaN

Comodo Dragon NaN NaN

IE10 404.0 0.08

Chrome 200.0 0.02

Here we can see for new index which are not present are df for them values are assigned as NaN.

**Note**:

By default values in the new index that do not have corresponding records in the dataframe are assigned NaN.

**Question**:

For above question, fill by value 0 instead of NaN.

**Answer**:

df.reindex(new\_index, fill\_value=0)

http\_status response\_time

Safari 404.0 0.07

Iceweasel 0 0

Comodo Dragon 0 0

IE10 404.0 0.08

Chrome 200.0 0.02

df1.reindex\_like(df2, method=None, copy=True, limit=None, tolerance=None)

Return an object with matching indices as df2 object.

df2: Object of the same data type

Its row and column indices are used to define the new indices of this object.

method: {None, ‘backfill’/’bfill’, ‘pad’/’ffill’, ‘nearest’}

**Question**

Reindex dataframe df1 to dataframe df2.

df1 = pd.DataFrame([[24.3, 75.7, 'high'],[31, 87.8, 'high'],[22, 71.6, 'medium'],[35, 95, 'medium']],

columns=['temp\_celsius', 'temp\_fahrenheit','windspeed'],index=pd.date\_range(start='2014-02-12',

end='2014-02-15', freq='D'))

df2 = pd.DataFrame([[28, 'low'],[30, 'low'],[35.1, 'medium']],columns=['temp\_celsius', 'windspeed'],

index=pd.DatetimeIndex(['2014-02-12', '2014-02-13','2014-02-15']))

|  |  |
| --- | --- |
| df1  temp\_celsius temp\_fahrenheit windspeed  2014-02-12 24.3 75.7 high  2014-02-13 31.0 87.8 high  2014-02-14 22.0 71.6 medium  2014-02-15 35.0 95.0 medium  df2  temp\_celsius windspeed  2014-02-12 28.0 low  2014-02-13 30.0 low  2014-02-15 35.1 medium |  |

We can see in df1 there is one column and one row in addition to all rows and columns of df2.

df2.reindex\_like(df1)

temp\_celsius temp\_fahrenheit windspeed

2014-02-12 28.0 NaN low

2014-02-13 30.0 NaN low

2014-02-14 NaN NaN NaN

2014-02-15 35.1 NaN medium

We can for additional row and column of df1 , it filled with NaN.

**Note:**

For uncommon row and column name, data will be filled with NaN.

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

Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don’t throw an error.

*mapper*: dict-like or function,

Dict-like or function transformations to apply to that axis’ values.

*index*: dict-like or function

Alternative to specifying axis (mapper, axis=0 is equivalent to index=mapper).

*columns*: dict-like or function

Alternative to specifying axis (mapper, axis=1 is equivalent to columns=mapper).

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

Axis to target with mapper.

**Question**:

For below dataframe, rename column into lower case (convert columns into lower case).

df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})

**A C**

0 1 4

1 2 5

2 3 6

**Answer**:

df.rename(*columns={"A": "a", "B": "c"})*

**a c**

0 1 4

1 2 5

2 3 6

**Question**:

For same dataframe, rename indexes to x,y,z

**Answer**:

df.rename(*index={0: "x", 1: "y", 2: "z"*})

A B

**x** 1 4

**y** 2 5

**z** 3 6

**Questions**:

Rename/change the column name of above df using function or by parameter style.

**Answer**:

df.rename(str.lower, axis='columns')

**a b**

0 1 4

1 2 5

2 3 6

df.rename\_axis(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False)

Set the name of the axis for the index or columns.

*mapper*: scalar, list-like, optional

Value to set the axis name attribute.

*index, columns*: scalar, list-like, dict-like or function, optional

A scalar, list-like, dict-like or functions transformations to apply to that axis’ values.

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

axis to rename

**Example**:

We have below dataframe, rename it’s index axis.

df = pd.DataFrame({"num\_legs": [4, 4, 2],

"num\_arms": [0, 0, 2]},

["dog", "cat", "monkey"])

num\_legs num\_arms

dog 4 0

cat 4 0

monkey 2 2

df = df.rename\_axis("animal")

num\_legs num\_arms # Index axis is renamed, default axis=0

**animal**

dog 4 0

cat 4 0

monkey 2 2

df = df.rename\_axis("limbs", axis="columns")

limbs num\_legs num\_arms

animal

dog 4 0

cat 4 0

monkey 2 2

DataFrame.reset\_index(level=None, drop=False, inplace=False, col\_level=0, col\_fill='')

Reset the index, or a level of it. Reset the index of the DataFrame, and use the default(numbers)

one instead

Example:

df = pd.DataFrame([('bird', 389.0),

('bird', 24.0),

('mammal', 80.5),

('mammal', np.nan)],

index=['falcon', 'parrot', 'lion', 'monkey'],

columns=('class', 'max\_speed'))

class max\_speed

falcon bird 389.0

parrot bird 24.0

lion mammal 80.5

monkey mammal NaN

**df.reset\_index()** #index is resetted, new index is now numbers

index class max\_speed

0 falcon bird 389.0

1 parrot bird 24.0

2 lion mammal 80.5

3 monkey mammal NaN

**df.reset\_index(drop=True)**

class max\_speed #Index is resetted and old index column is droped.

0 bird 389.0

1 bird 24.0

2 mammal 80.5

3 mammal NaN

DataFrame.set\_axis(labels, axis=0, inplace=False)

---- Alternate of rename()

This is for changing the name of any axis.

*labels*: list-like, Index

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

**Questions**:

For below DF change the row and column names.

df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})

Changing row/index labels

df.set\_axis(['a', 'b', 'c'], axis='index')

A B

a 1 4

b 2 5

c 3 6

Changing column labels.

df.set\_axis(['I', 'II'], axis='columns')

I II

0 1 4

1 2 5

2 3 6

Df.drop\_duplicates(subset=None, keep='first', inplace=False, ignore\_index=False)

################################################################################

# Indexing/indexers in pandas #

################################################################################

In pandas we have below methods for Indexing –

1. Dataframe[] ---- Indexing operator
2. df.loc[] ---- used for labels
3. df.iloc[] -------- for position or integer based
4. df.ix[] ------ label and integer based
5. df.at(row,col)
6. df.index[[x1,x2….xn]] ------ getting indexes for row number x1,x2….xn

**Dataframe[]**

This is used for selecting required column(s) from a dataframe.

df[ column\_name(s) ] ---- If multiple column name then give in list.

df[ **[col1,col2…coln]** ] ----------- if multiple columns.

Questions:

From dataframe 'df', give all records for column- age , salary

Answer:

df[[‘age’,’salary’]]

**Dataframe.loc[]**

This is used to select data by labels of row and column.

Let’s discuss using below possible cases-

df.loc[ start\_row : end\_row , start\_col : end\_col ] --- **selecting a range of rows and columns**

df.loc[ [row1,row2..rown], [col1, col2….coln] ] ------- **selecting given rows and column**

df.loc[ :, [col1,col2…coln] ] ----------- **Selecting all rows and given column names**

df[ [col1,col2…coln] ] ----------------- **Selecting all rows are columns – col1,col2….coln**

**Case 1: Selecting Single row and all/few column**

df.loc[‘index\_name\_of\_row’]

**Case 2: Selecting multiple rows**

df.loc[ [row\_names(s)] ]

**Dataframe.iloc[]**

This is used to select specific column names bases on index/position of rows and columns.

df.iloc[n1 : n2 , x1 : x2] ---- Select rows from n1 to n2 and columns x1 to x2

df.iloc[ [n1, n1 …n], [x1, x2 … xn] ] ----- Select nows n1,n2…n and columns x1,x2…xn

df.iloc[ : n1, :n2 ] ------ Select rows from position 0 to (n1-1) and columns 0 to (n2-1)

###############################################################################

# Applying condition on df #

###############################################################################

For applying condition on dataframe we will use indexers in pandas. Condition could be

~ -------- Not operator

== --- equality operator

!= ---- not equal to

< --- less than

> --- greater than

<= less than equal to

and so on

**Selecting all rows, applying condition on columns**

df[ df[ ‘col\_name’ ] **condition**] ------ select all rows by applying condition on a column

df[ df.col\_name **condition** ] ------ Applying condition on selected column names

df.**loc**[ df[ ‘col\_name’ ] condition ] ---- select all rows by applying condition on a column using loc

df.**loc**[ df.col\_mame condition ] ------ Applying condition on selected column names using loc

sr.[location(s)] ------- Getting value from series from given location

**Question**: Select those record for which number of attempt>2

df[df. 'attempts ' >2]

**Questions**: Delete row(s) where value of ‘col2’ is not equal to 5. ----- Good

d = {'col1': [1, 4, 3, 4, 5], 'col2': [4, 5, 6, 7, 8], 'col3': [7, 8, 9, 0, 1]}

df = pd.DataFrame(data=d)

print("Original DataFrame")

print(df)

df = df[df.col2 != 5]

print("New DataFrame")

print(df)

**Selecting few rows and few col names by applying condition on columns**

This is achieved by applying condition and then selecting required columns.

df[ df[ ‘col\_names’ ] condition] ].loc loc[ start\_row : end\_row , start\_col : end\_col ]

df[ df[ ‘col\_names’ ] condition] ]. loc[ [row1,row2..rown], [col1, col2….coln] ]

df[ df[ ‘col\_names’ ] condition] ]. loc[ :, [col1,col2…coln] ]

**Applying multiple condition**

For applying multiple condition we can take use of &,| operator.

& ------ For AND

| ------ For OR

df.loc [ (df[ col\_name ]condition1) & (df[ col\_name ]condition2)]

df.loc[ (df.loc condition) & (df.loc condition) ]

**################################################################################# Indexing, iteration #**

**################################################################################**

In pandas we have below methods for integration in pandas.

df.head(n) ------- Returns n number of rows

df.at(rows,columns) ----- Selecting data at specified rows and columns

df.loc() ------ Selecting one/more rows/columns

df.iloc() ----- Selecting one/more rows/columns using position

df.iteritems()

Iterate over (column name, Series) pairs. Gives column\_name and data of that columns.

Iterates over the DataFrame columns, returning a tuple with the column name and the content as a Series.

exam\_data = [{'name':'Anastasia', 'score':12.5}, {'name':'Dima','score':9}, {'name':'Katherine','score':16.5}]

df = pd.DataFrame(exam\_data)

print(df)

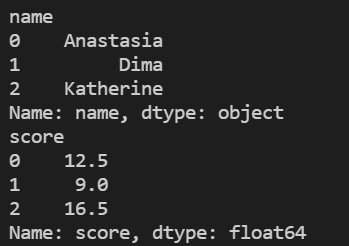
print()

for x,y in df.iteritems():

    print(x) # Gives column name

    print(y) # Gives data of that column

Result



df.iterms()

It’s exactly same as above.

df.iterrows()

Iterate over DataFrame rows as (index, Series) pairs.

It returns the rows/indexes and data for each column.

exam\_data = [{'name':'Anastasia', 'score':12.5}, {'name':'Dima','score':9}, {'name':'Katherine','score':16.5}]

df = pd.DataFrame(exam\_data)

print()

for x,y in df.iterrows():

    print(x) # rows/index

    print(y) #column for that index

df.insert(loc, column, value, allow\_duplicates=False)

Insert column into DataFrame at specified location.

Raises a ValueError if column is already contained in the DataFrame, unless allow\_duplicates is set to True.

**loc**: int

Insertion index. Must verify 0 <= loc <= len(columns).

**column**:str, number, or hashable object

Label of the inserted column.

**value**:int, Series, or array-like ---- Values to be inserted

**allow\_duplicates**: bool, optional

**Note:**

We can insert a column using below way –

df[‘col\_name’]=[val1,val2….valn] -------- Another way to insert column