# Introduction to Pandas

- Written by David Song

## Contents

# Introduction

Pandas is an open-source Python package that is commonly used for data science, data analysis, and machine learning tasks. It is built on top of another library named Numpy. It provides various data structures and operations for manipulating numerical data and time series and is very efficient in performing various functions like data visualization, data manipulation, data analysis, etc.

Pandas Data Structures: Series, DataFrame, Panel

- Series - a one-dimensional array-like structure with homogeneous data
- DataFrame - a two-dimensional array-like structure with heterogeneous data. It stores data in column wise, each column data is a series.
- Panel – a 3D data structure, not widely used.

Of the above there, DataFrame is the mostly used, it's the key component of Pandas package. Series is an auxiliary, since each column in a DataFrame is a Series. Selecting a column from a DataFrame gives a Series, while select a row gives a DataFrame. Think of manipulating data on a two dimensional space, instead of working on data at a single point in traditional way. A DataFrame can be viewed similar to MS Excel or a SQL table, a lot of operations in MS Excel and SQL Table can be found in Pandas data frame. If you are not sure which operations are available in DataFrame, think about what you can do in MS Excel and SQL, likely you will find similar operations in DataFrame.

Similar to MS Excel, a DataFrame has row and column names (labels), whereas row label is also referred as index. Thus, data within a DataFrame can be accessed or sliced through those row and column labels.

Pandas provides fruitful operations (functions, methods), some are under Pandas package, some are under DataFrame, some belong to Series.

This document will describe some Pandas DataFrame basic concepts and commonly used operations.
For more details, please refer to Pandas API official document site:
https://pandas.pydata.org/docs/reference

# Creating Series, DataFrame

Series and DataFrame have many similarities; we will show how to create Series and DataFrame in below.

```
import pandas as pd
```

## Series

A series is an array with index, it can be created from a Numpy array, a list, and a dictionary  with index or not. Default index is a series number starts from 0. A series can also be created from a scalar with index. A Numpy array is similar to a list, so we don't show examples of using Numpy array in this section.

- an empty Series

```
pd.Series(dtype=str)
Series([], dtype: object)
```

- from a list:

```
pd.Series(range(0,5))
0    0
1    1
2    2
3    3
4    4
dtype: int64


The left is the index of the Series
```

- create a series with index

```
pd.Series(range(0,5), index=['a','b','c','d','e'])
a    0
b    1
c    2
d    3
e    4
```

- from a dictionary

```
pd.Series({'A': 101,'B': 202,'C': 303})
A    101
B    202
C    303
```

- from a scalar

```
Create a series data values are all 10:
pd.Series(10, index=[0, 1, 2, 3, 4, 5])
```

## DataFrame

A DataFrame is a two-dimensional array-like structure with row and columns labels, similar to a MS excel sheet. It can be created from a list, a list of lists, dictionary of lists, or Series. It can also be created from a Numpy array.  As stated in Series creation section, a Numpy array is similar to a list, a list of list, so we don't show examples of using Numpy array in this section

- an empty DataFrame

```
pd.DataFrame()

without column and row labels:
Columns: []
Index: []
```

- from a list

```
pd.DataFrame(range(0,5))  #create a dataframe with 1 column
```

- from a list of list

  This will create a dataframe row wise, place one list on a row, and then next list on the next row.

```
pd.DataFrame([range(0,5), range(10,15)])
    0   1   2   3   4
0   0   1   2   3   4
1  10  11  12  13  14
```

  We can see default row and columns labels are an integer series starts from 0. We can give specific row and columns names when creating a DataFrame:

```
pd.DataFrame([range(0,5), range(10,15)], columns=['a', 'b','c','d','e'],
index=['r1','r2'])

     a   b   c   d   e
r1   0   1   2   3   4
r2  10  11  12  13  14
```

- From a dictionary with list

  Keys are the column names. This fills data in column wise, comparing to create from list of list, which fills data in row wise.

```
pd.DataFrame({'Col1':range(0,5), 'Col2':range(10,15)})
   Col1  Col2
0     0    10
1     1    11
2     2    12
3     3    13
4     4    14
```

- From Numpy Array reshape – a convenient way to create a DataFrame

```
1. create a one dimensional array, 2. reshape it to a two dimensional
   array. 3. Create a DataFrame from the two dimensional array.

pd.DataFrame(np.arange(6).reshape((3, 2)))
```

## Numpy Array, list, Series Conversion

NumPy stands for numerical Python. It's a Python library supports for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

Since Pandas is built on top of Numpy, sometimes we need to convert from Numpy array to DataFrame, Series and vice versa. We will discuss how to convert them here.

## Create a Numpy array

```
# create 1 dimension Numpy array

arr = np.array([1, 2, 3])
=>array([1, 2, 3])


# two dimensional:

arr=np.array([range(0,5), range(10,15),range(20,25)])
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24]])
```

## Series and DataFrame can be created from Numpy Array.

```
pd.Series(np.array([1, 2, 3]))
pd.DataFrame(np.array([1, 2, 3]))
```

## Numpy array tolist method returns a list

```
np.array([1, 2, 3]).tolist()
=>[1, 2, 3]

np.array([range(0,5), range(10,15),range(20,25)]).tolist()
=>[[0, 1, 2, 3, 4], [10, 11, 12, 13, 14], [20, 21, 22, 23, 24]]
```

## Series, DataFram to list, dictionary and Numpy array

### Series

```
ser = pd.Series(range(0,5), index=['a','b','c','d','e'])

ser.to_list()
[0, 1, 2, 3, 4]

ser.to_dict()
```

```
{'a': 0, 'b': 1, 'c': 2,'d': 3, 'e': 4}

ser.to_numpy()
array([0, 1, 2, 3, 4], dtype=int64)
```

### DataFrame

```
df = pd.DataFrame([range(0,5), range(10,15)], columns=['a','b','c','d','e'],
index=['r1','r2'])

# returns a dictionary of dictionary
df.to_dict()
{'a': {'r1': 0, 'r2': 10}, 'b': {'r1': 1, 'r2': 11}, 'c': {'r1': 2, 'r2': 12},
'd': {'r1': 3, 'r2': 13}, 'e': {'r1': 4, 'r2': 14}}

# provides an orient, will return a dictionary of list.
# Keys are the columns labels.
df.to_dict('list')
{'a': [0, 10], 'b': [1, 11], 'c': [2, 12], 'd': [3, 13], 'e': [4, 14]}

# display in row wise, transpose and then use to_dict():
df.T.to_dict('list')
{'r1': [0, 1, 2, 3, 4], 'r2': [10, 11, 12, 13, 14]}
```

The property df.values returns a Numpy array. This implies that DataFrame is built on top of Numpy array.

```
df.values.tolist() # there is no _ between to and list, as DataFrame methods.
[[0, 1, 2, 3, 4], [10, 11, 12, 13, 14]]

df.to_numpy()
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14]], dtype=int64)

df.to_numpy().tolist() # same effect as df.values.tolist()
[[0, 1, 2, 3, 4], [10, 11, 12, 13, 14]]
```

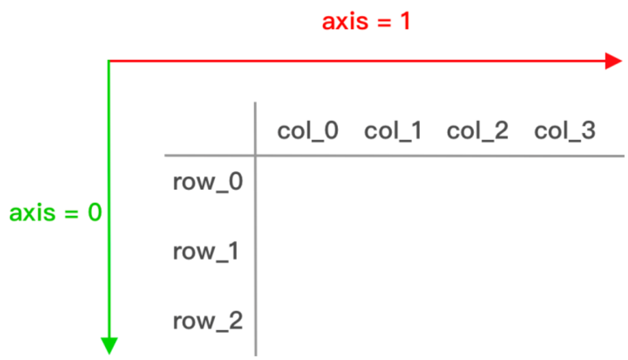
There are many conversion methods in DataFrame, such as:
`to_csv, to_excel, to_jason, to_parket etc.`
`Please refer to DataFrame official document:`
[https://pandas.pydata.org/docs/reference/frame.html](https://pandas.pydata.org/docs/reference/frame.html)

## Axis

The Axis concept comes from Numpy, and it’s widely used in many operations of DataFrame. Remember Pandas is built on top of Numpy, and it’s two dimensional arrays like structure.

In DataFrame, an axis is a direction; it tells the operation will be performed row wise or column wise, as shown in below figure.  Axis=0 means the operation will be performed downward, or column wise; while

Axis=1 means the operation will be performed horizontally, or row wise. The commonly used operations are performed column wise, so the default value of axis is 0.




Below example uses sum method to illustrate the axis concept.

```
data=[[1,1,1],[2,2,2], [3,3,3]]
df = pd.DataFrame(data)

   0  1  2
0  1  1  1
1  2  2  2
2  3  3  3

df.sum()
0    6
1    6
2    6
dtype: int64

df.sum(axis=1)
0    3
1    6
2    9
dtype: int64
```

The sum method returns a Series. Now let's add them into the data frame:

1. add a column for sum of each row

```
df['sum'] = df.sum(axis=1)
   0  1  2  sum
0  1  1  1    3
1  2  2  2    6
2  3  3  3    9
```

2. insert a row of sum for each column

```
df.loc['Total']= df.sum()
       0  1  2  sum
```

```
0        1  1  1    3
1        2  2  2    6
2        3  3  3    9
Total    6  6  6   18
```

To conclude in simple words, an axis is a direction. Like a Cartesian coordinate system has the x and y axis in a two dimensional space, or a unit vector in linear algebra, it's nothing but shows a direction.

# Access, Slicing, Subset data

## DataFrame row, column labels

A DataFrame has row, column labels, both are Index type. Normally, rows are referred as index, default labels are integer based ranges start from 0, like below:

```
df=pd.DataFrame(np.array(range(0,9)).reshape(3,3))
df
   0  1  2
0  0  1  2
1  3  4  5
2  6  7  8


df.index
RangeIndex(start=0, stop=3, step=1)

df.columns
RangeIndex(start=0, stop=3, step=1)
```

Row and Column labels can be given at a Dataframe creation, or set dynamically:

```
df.index=list('abc')
df.columns=list('ABC')

df
   A  B  C
a  0  1  2
b  3  4  5
c  6  7  8
```

Row and column indexes can have a name, which is useful when using multi-index.

```
df.index.name='row_level0'
df.columns.name='col_level0'

df
col_level0  A  B  C
row_level0
a           0  1  2
```

```
b              3   4   5
c              6   7   8
```

A row, column name is just a label to identify a row or a column, like row/column labels used to identify each row/column in a single row/column index.

## Basic Operations

Indexing operators `[]` and attribute operator `.` provide quick and easy access to pandas data structures

Basically to select a subset of a DataFrame:
`df.loc[row_indexer,column_indexer]`

Example:

```
df=pd.DataFrame(np.array(range(0,9)).reshape(3,3), index=list('abc'),
columns=list('ABC'))
df
   A   B   C
a  0   1   2
b  3   4   5
c  6   7   8
```

Select a subset giving a list of row and column labels:

```
df.loc[['a','b'], ['A','B']]
   A   B
a  0   1
b  3   4
```

## Rows selection

1. `.loc` - label based
2. `.iloc` - position based
3. `[]` – with passed in a Boolean vector

Both .loc and .iloc can accept a callable

Examples:
`df.loc[['a','b']]` is equivalent to `df.loc[['a','b'],:]`

```
   A   B   C
a  0   1   2
b  3   4   5
```

Note that, the symbol `':'` is a Python slicing operator, which is adopted in Pandas. In the above example, it means select all columns.

The same can achieved using `.iloc`, which uses a list of row positions:

```
df.iloc[[0,1]]
   A  B  C
a  0  1  2
b  3  4  5
```

## Columns selection

`[]` – the primary function of indexing for columns.

Examples:

```
df[['A', 'B']]
   A  B
a  0  1
b  3  4
c  6  7
```

## Attribute access

Columns can be access by attribute operator `'.'` with column label:

`df.A` is equivalent to `df['A']`:

```
a     0
b     3
c     6
```

**Important**: the operator `[]` can result in row or column selections. (1). When a single or a list of column labels passed in, it's a columns section. (2). When a boolean vector passed in, it's a rows selection. To avoid confusion, use `df.loc[]` for rows selections.

* A boolean vector here means, it could be (a) a boolean vector with True and False values. (2) any condition criterion, slicing, functions etc. which results in a boolean vector.

## Slicing

Symbol `':'` is a Python slicing operator, which is used to select a subset of a list. For example:

```
s=list('abc') -> ['a', 'b', 'c']
s[1:2] -> ['b']
s[::2] -> ['a', 'c']
```

Pandas (and Numpy) use this operator to select rows and columns.
Examples:

```
Ex. 1:
df[:2]
   A  B  C
```

```
a  0  1  2
b  3  4  5
```

Ex. 2:

```
df['a':'b']
   A  B  C
a  0  1  2
b  3  4  5
```

```
Ex. 3:
df[-1:]
   A  B  C
c  6  7  8
```

Ex. 4:

```
df.iloc[:2]
   A  B  C
a  0  1  2
b  3  4  5
```

```
Ex. 5:
df.loc['a':'b']
   A  B  C
a  0  1  2
b  3  4  5
```

```
Ex. 6:
df.loc['a':'b', 'A':'B']
   A  B
a  0  1
b  3  4
```

**Notes (pitfalls):**

Dataframe slicing creates a lot of confusing, need to use it carefully.

1. DataFrame `[]` slices the **rows**, this is easy to confuse with column selection. Remember that, when you see ':' inside [], it selects rows; when you see it inside .loc[], it selects row and columns (`Ex6`).

2. when you see integers given inside `[]` together with symbol ':', this means select rows by **position** (slicing by position), similar to `.iloc[] (Ex1 and Ex4);` when you see letters inside `[]`, this is slicing by label. Slicing by position exclude the last index (upper bound); while slicing by label **includes** the last index.

3. When slicing by labels, the dataframe index must be sorted first to avoid errors.

4. When a DataFrame doesn't provided with row and columns labels, then default integer range index is used. In that case, `df[0]` selects the first column, where `df[0:1]` select the first row. For good practice, it better to provide columns with meaningful labels.

5. To avoid confusions, when using slicing, better to use `df.loc[]` for label slicing, `df.iloc` for position.

## Conditional Selection

Specify a boolean criteria.

```
df[df>3]
     A    B    C
a  NaN  NaN  NaN
b  NaN  4.0  5.0
c  6.0  7.0  8.0

df[df.B>3]
   A  B  C
b  3  4  5
c  6  7  8
```

Same effect using:

```
df.loc[df.B>3]
   A  B  C
b  3  4  5
c  6  7  8
```

Use a callable:

```
df[lambda df: df['B'] > 3]
   A  B  C
b  3  4  5
c  6  7  8
```

Select only rows that meet criteria:

```
df2=df[df>3]
df2.dropna()
     A    B    C
c  6.0  7.0  8.0
```

**Important**: for any parameters passed inside the `[]`, such as conditional criterion, functions etc. if those parameters result in a Boolean vector, then it is a rows selection.

### There `where()` method

Selecting values from a DataFrame with a Boolean criterion now also preserves input data shape, with false positions filled as NaN:

```
df.where(df.B>3)
     A    B    C
a  NaN  NaN  NaN
b  3.0  4.0  5.0
c  6.0  7.0  8.0
```

The where() method can take an optional argument to replace those values where the condition is False.

Below example uses inverse value to fill rows where column B value is not large than 3:

```
df.where(df.B>3, -df)
   A  B  C
a  0 -1 -2
b  3  4  5
c  6  7  8
```

### The `query()` method

The query() method that allows selection using an expression.

```
df.query('B>3')
   A  B  C
b  3  4  5
c  6  7  8
```

```
df.query('A < B and B < C')
   A  B  C
a  0  1  2
b  3  4  5
c  6  7  8
```

### The `mask()` method

The mask() method is the inverse boolean operation of `where`, means mask values with criterion to NaN.

```
df.mask(df.B>3)
     A    B    C
a  0.0  1.0  2.0
b  NaN  NaN  NaN
c  NaN  NaN  NaN
```

## Reindex, set_index, reset_index, index rename

**reindex** – change (realign) the index of rows and columns by the new given index (or a list). If the given new index within the existing index, realign the index positions, otherwise fill the missing indexes with NaN.

Considering our existing df

```
   A  B  C
a  0  1  2
```

```
b  3  4  5
c  6  7  8

df.index -> Index(['a', 'b', 'c'], dtype='object')
```

Ex 1: new indexes are the same as existing index with different orders, result in reorder the index:

```
df.reindex(list('cba'))
   A  B  C
c  6  7  8
b  3  4  5
a  0  1  2
```

Ex 2: a new index value ‘d’, result with reorder the index, values in row with ‘d’ are filled with NaN:

```
df.reindex(list('dcba'))
     A    B    C
d  NaN  NaN  NaN
c  6.0  7.0  8.0
b  3.0  4.0  5.0
a  0.0  1.0  2.0
```

Ex 3: The same applies to reindex columns, provides `axis`:

```
df.reindex(list('CBAD'), axis=1)
   C  B  A   D
a  2  1  0 NaN
b  5  4  3 NaN
c  8  7  6 NaN
```

Ex 4: fill missing values:

```
df.reindex(list('CBAD'), axis=1, fill_value=-1)
   C  B  A  D
a  2  1  0 -1
b  5  4  3 -1
c  8  7  6 -1
```

Ex 5: choose a subset of index:

```
df.reindex(list('bc'))
   A  B  C
b  3  4  5
c  6  7  8
```

## set_index— changes a column as the new row index

Below example sets column ‘A’ as the new row idex:

```
df.set_index('A')
   B  C
A
```

```
0   1   2
3   4   5
6   7   8
```

We can see that, column 'A' becomes the row index, with name 'A'

**reset_index**— the inverse operation of set_index, transfers the row index into a column, and set row index as integer range index starts from 0.

```
df.reset_index()
  index  A  B  C
0     a  0  1  2
1     b  3  4  5
2     c  6  7  8
```

we can see the row index becomes a new column labeled with the index default name 'index'

In case we need to discard the row index and replace it with new integer ranged index:

```
df.reset_index(drop=True)
   A  B  C
0  0  1  2
1  3  4  5
2  6  7  8
```

**index rename** – give row index a new name

The simple way is directly change the index name like below:

```
df.index.name='idx'
df
     A  B  C
idx
a    0  1  2
b    3  4  5
c    6  7  8
```

We can see the row index has a name now.

PS: indexes are names or labels for row and columns, while index name and column name are names for names (labels for labels).  This is very useful for multi-indexes.

## Map, Apply, Transform

### map – applies a function that accepts and returns a scalar to every element of a DataFrame.

Note: Since version 2.1.0: df.applymap was deprecated and renamed to DataFrame.map. I am using version 1.3.5, so have to use applymap.

```
df.applymap(lambda x: x*2)
    A   B   C
a   0   2   4
b   6   8  10
c  12  14  16
```

From above example, we can see the map() method updates each element in the dataframe by multiply by 2.

### apply— apply a function row or column wise

```
Ex 1:
df.apply(lambda x: np.sum(x))
A     9
B    12
C    15

Ex 2:
df.apply(lambda x: np.sum(x), axis=1)
a     3
b    12
c    21
```

Ex 1 applies sum to each row, the result labels are column labels. Ex 2 apply sum on each column, the result labels are row labels.

### Transform— similar to map(), which reserves the shape of original dataframe, but can take multiple functions.

## Sort – sort_vlaues, sort_index

### sort_values – sort by values

```
df2=df.reindex(list('cba'))
df2
   A  B  C
c  6  7  8
```

```
b  3  4  5
a  0  1  2
```

Sort by values in column B:

```
df2.sort_values(by='B')
   A  B  C
a  0  1  2
b  3  4  5
c  6  7  8
```

Sort_index – sort row index

```
df2.sort_index()
   A  B  C
a  0  1  2
b  3  4  5
c  6  7  8
```

## Iteration

Don't use iteration unless you have to.

1. Iterating through pandas objects is generally slow.
2. **Never modify** something you are iterating over, since the iterator could return a copy and not a view, depends on the type.

Below are the three methods used for iteration:

`items()` – iterates over columns
`iterrows()` – iterates over rows
`itertuples()` – iterate over the rows of a DataFrame as named tuples of the values. It merely returns the values inside a named tuple. Therefore, itertuples() preserves the data type of the values and is generally faster as iterrows().

```
Ex 1: items() iterates each column:

for label, ser in df.items():
    print(label)
    print(ser)
A
a    0
b    3
c    6
Name: A, dtype: int32
```

```
B
a    1
b    4
c    7
Name: B, dtype: int32
C
a    2
b    5
c    8
Name: C, dtype: int32
```

Ex 2: `iterrows()`

```
for row_index, row in df.iterrows():
    print(row_index, row, sep="\n")
a
A    0
B    1
C    2
Name: a, dtype: int32
b
A    3
B    4
C    5
Name: b, dtype: int32
c
A    6
B    7
C    8
Name: c, dtype: int32
```

Ex 3: itertuples() - simple print

```
for row in df.itertuples():
    print(row)

Pandas(Index='a', A=0, B=1, C=2)
Pandas(Index='b', A=3, B=4, C=5)
Pandas(Index='c', A=6, B=7, C=8)
```

Ex 4: itertuples() - access row label and column value in each row:

```
for row in df.itertuples():
    print('row label:{}, Value in Column A:{}'.format(row.Index, row.A))

row label:a, Value in Column A:0
row label:b, Value in Column A:3
row label:c, Value in Column A:6
```

# Data type conversion

## Data Types (dtypes)

For the most part, Pandas uses NumPy arrays and `dtypes` for Series or individual columns of a DataFrame. NumPy provides support for float, int, bool, timedelta64, and datetime64. Pandas and third-party libraries *extend* NumPy's type system in a few places. We will list the mostly common used data types, for extended types please refer to [3].

```
object
int
float
bool
datetime
timedelta
category
```

Check data types (dtypes)

```
df.dtypes
A    int32
B    int32
C    int32
dtype: object
```

We can see those 3 columns are int32 type (different Pandas version may use 64 bits). We can apply arithmetic operations to those columns:

```
df+2
   A  B   C
a  2  3   4
b  5  6   7
c  8  9  10
```

Let's create a DataFrame with strings:

```
df2=pd.DataFrame(np.array(list('abcdefghi')).reshape(3,3))
df2
   0  1  2
0  a  b  c
1  d  e  f
2  g  h  i

df2.dtypes
0    object
1    object
2    object
dtype: object
```

An object is a string in panda, so it performs a string operation:

```
df2 + 'Y'
    0   1   2
0  aY  bY  cY
1  dY  eY  fY
2  gY  hY  iY
```

## Data Types conversion

- Use `astype()` to force convert columns to an appropriate dtype
- Create a custom function to convert the data
- Use pandas helper functions such as `to_numeric()` or `to_datetime()`

Using the `df.astype(dtype, copy=None, errors='raise')` function:

Convert it to string:

```
df_str=df.astype(str)

df_str.dtypes
A    object
B    object
C    object
dtype: object
```

Convert it float:

```
df_str=df.astype(float)
df_str
     A    B    C
a  0.0  1.0  2.0
b  3.0  4.0  5.0
c  6.0  7.0  8.0
```

Using `df.apply()` with a customized or lambda function to convert data types.

When date type convertion encounters errors, the `astype()` function  has only two options: raise the error or return the original, both means the conversion failed. Pandas helper functions are very flexible dealing with errors with a third choice ‘coerce’:

errors{‘ignore’, ‘raise’, ‘coerce’}, default ‘raise’

- If ‘raise’, then invalid parsing will raise an exception.
- If **‘coerce’**, then invalid parsing will be set as **NaN**.
- If ‘ignore’, then invalid parsing will return the input.

To illustrate, let’s change the upper rightmost to a letter and then convert the column ‘C’ to numeric:

```
df2=df.copy(deep=True)
df2.loc['a','C']='a'
df2
   A  B  C
a  0  1  a
b  3  4  5
c  6  7  8

pd.to_numeric(df2['C'], errors='coerce')
a    NaN
b    5.0
c    8.0
Name: C, dtype: float64
```

Note that, these helper functions are under Pandas package, while astype() is a DataFrame and a Series method, this means you can use astype() on a Series or on an entire DataFrame.

## Vectorized Accessor

Pandas provides dtype-specific methods under various accessors. These are separate namespaces within Series that only apply to specific data types [4].

| Data Type | Accessor |
|---|---|
| Datetime, Timedelta, Period | dt |
| String | str |
| Categorical | cat |

```
Ex. 1 str:

df2=pd.DataFrame(np.array(list('abcdefghi')).reshape(3,3), index=list('abc'),
columns=list('ABC'))
df2
   A  B  C
a  a  b  c
b  d  e  f
c  g  h  i

df2.A.str.upper()
a    A
b    D
c    G


Ex. 2 dt

df2['D']=list(pd.Series(pd.date_range("20130101 09:10:12", periods=3)))
```

```
df2
   A  B  C                   D
a  a  b  c 2013-01-01 09:10:12
b  d  e  f 2013-01-02 09:10:12
c  g  h  i 2013-01-03 09:10:12

df2.dtypes
A             object
B             object
C             object
D     datetime64[ns]
dtype: object
```

Convert it to string with only year and month:

```
df2.D = df2.D.dt.strftime('%Y%m')
df2
   A  B  C       D
a  a  b  c  201301
b  d  e  f  201301
c  g  h  i  201301

df2.dtypes
A    object
B    object
C    object
D    object
dtype: object
```

Convert it back to datatime:

```
pd.to_datetime(df2.D, format='%Y%m')
a   2013-01-01
b   2013-01-01
c   2013-01-01
Name: D, dtype: datetime64[ns]
```

# merge, join, concat – combining multiple datasets

## Concat

The `concat()` function concatenates multiple datasets into one, vertically or horizontally. It is a function of Pandas. The `append()` function in DataFrame for vertical concatenation is deprecated.

Ex 1 concatenate vertically:

```
df1
   A  B  C
```

```
a  0  1  2
b  3  4  5
c  6  7  8

df2
   A  B  C
a  a  b  c
b  d  e  f
c  g  h  i
```

Concatenates above two by providing a list of DataFrames:

```
pd.concat([df,df2])
   A  B  C
a  0  1  2
b  3  4  5
c  6  7  8
a  a  b  c
b  d  e  f
c  g  h  i
```

the index seems useless, we can drop it:

```
pd.concat([df,df2], ignore_index=True)
   A  B  C
0  0  1  2
1  3  4  5
2  6  7  8
3  a  b  c
4  d  e  f
5  g  h  i
```

To distinguish data, give a list of keys:
Note: cannot use ignore_index with keys together, since keys will create a new index, while ignore_index will drop all indexes.

```
df_concat = pd.concat([df,df2],keys=['df1', 'df2'])
df_concat
       A  B  C
df1 a  0  1  2
    b  3  4  5
    c  6  7  8
df2 a  a  b  c
    b  d  e  f
    c  g  h  i
```

remove original index in the above concatenated result:

```
df_concat.reset_index(level=1, drop=True)
     A  B  C
df1  0  1  2
```

```
df1   3   4   5
df1   6   7   8
df2   a   b   c
df2   d   e   f
df2   g   h   i
```

Ex 2 concatenate horizontally:

```
pd.concat([df,df2], axis=1)
   A   B   C   A   B   C
a  0   1   2   a   b   c
b  3   4   5   d   e   f
c  6   7   8   g   h   i
```

```
pd.concat([df,df2], ignore_index=True, axis=1)
   0   1   2   3   4   5
a  0   1   2   a   b   c
b  3   4   5   d   e   f
c  6   7   8   g   h   i
```

Note that, above examples use datasets with the same size and indexes (labels). If indexes are different, there is a `join` parameter, which indicates the concatenation is union or intersection (on indexes), default is union.

Adding a row or column to a DataFrame via concatenation

```
Example Add a row (index must match column names):

ser=pd.Series(list('xyz'), index=list('ABC'), name='d')
pd.concat([df1, ser.to_frame().T], axis=0)
   A   B   C
a  0   1   2
b  3   4   5
c  6   7   8
d  x   y   z
```

```
Add a column with a series (index must match row labels):

ser=pd.Series(list('xyz'), index=list('abc'), name='D')
pd.concat([df1, ser.to_frame()], axis=1)
   A   B   C   D
a  0   1   2   x
b  3   4   5   y
c  6   7   8   z
```

## merge

The merge() function is similar to relation database SQL join. The usage is straightforward for those who are familiar with SQL.

Note that, merge is a function in the pandas namespace, and it is also available as a DataFrame instance method merge(), with the calling DataFrame being implicitly considered the left object in the join. I noticed onetime the pd.merge() gives wrong result, meanwhile df.merge() gives correct result. Be aware of this!

A few key parameters that different from SQL:

- `how`: One of 'left', 'right', 'outer', 'inner', 'cross'. Defaults to inner.
- `suffixes`: A tuple of string suffixes to apply to overlapping columns. Defaults to ('_x', '_y').
- `indicator`: Add a column to the output DataFrame called _merge, with values `both, left_only, right_only.`
- `validate`: string, default None. If specified, checks if merge is of specified type, 'one_to_one', 'one_to_many' etc.

Example, change df1 column A values then merge with df2 on column A:

```
df1.A=list('adf')
df1
   A  B  C
a  a  1  2
b  d  4  5
c  f  7  8

pd.merge(df1, df2, on='A')
   A  B_x  C_x B_y C_y
0  a    1    2   b   c
1  d    4    5   e   f
```

We can see the overlapping columns B and C are appended with suffixes.

Change suffix, keep the left overlapping columns names, add right with '_y':

```
pd.merge(df1, df2, on='A', suffixes=('', '_y'))
   A  B  C B_y C_y
0  a  1  2   b   c
1  d  4  5   e   f
```

Example: outer join with indicator:

```
pd.merge(df1, df2, on='A', how='outer', indicator=True)
   A  B_x  C_x  B_y  C_y      _merge
0  a  1.0  2.0    b    c        both
1  d  4.0  5.0    e    f        both
2  f  7.0  8.0  NaN  NaN   left_only
3  g  NaN  NaN    h    i  right_only
```

Use DataFrame merge instead of Pandas merge:

```
df1.merge(df2, on='A')
   A  B_x  C_x B_y C_y
0  a    1    2   b   c
1  d    4    5   e   f
```

if row/columns labels are distinct, the `merge` result can be achieved the same result as `concat()` method.

## join

The `join()` is a convenient DataFrame method, it is a merge on index. The difference is if there are overlapping columns, you will have to specify suffixes differently, like below:

```
df1.join(df2, lsuffix="_x", rsuffix="_y")
  A_x  B_x  C_x A_y B_y C_y
a   a    1    2   a   b   c
b   d    4    5   d   e   f
c   f    7    8   g   h   i
```

For multi-index, specify index names in a list in `on=[]` parameter .

## compare

The `compare()` method is used to compare two DataFrame or Series, and summarize their differences. It **can only compare** identically-labeled DataFrame objects.

Example:

```
df2=df.copy(deep=True)
df2.loc['a', 'C']=0
df2.loc['b', 'B']=0
df2
   A  B  C
a  0  1  0
b  3  0  5
c  6  7  8

df.compare(df2)
     B          C
  self other self other
a  NaN   NaN  2.0   0.0
b  4.0   0.0  NaN   NaN
```

It shows only the rows with differences, equal values are marked as NaN. To view all rows and columns or original values, set `keep_shape=True` and/or `keep_equal=True`.

## Group by

Similar to SQL group by statement, the dataframe `groupby` method is used to split columns values into groups based on some key columns, then you can apply functions to each group, such as `mean, sum, min, max, first, last` etc.

Example: group a dataframe by column 'A', and select the sum of column 'B':

```
# create a dataframe
A=[0]*3 + [1]*3 + [2]*3
B=[1,2,3]*3
C=range(0,9)
df = pd.DataFrame({'A':A, 'B':B, 'C':C})
df.index=list('abcdefghi')
df
   A  B  C
a  0  1  0
b  0  2  1
c  0  3  2
d  1  1  3
e  1  2  4
f  1  3  5
g  2  1  6
h  2  2  7
i  2  3  8


df.groupby(by=['A'])[['B']].sum()
   B
A
0  6
1  6
2  6
```

The `groupby` method is a process involving one or more of the following steps:

- Splitting the data into groups based on some criteria.
- Applying a function to each group independently.
- Combining the results into a data structure.

### Splitting

Use `groupby` to split an object into groups by specifying one or a list of columns or indexes, it returns a `DataFrameGroupBy` object, which is a dictionary, whose keys are the distinct values of the tuple of columns or indexes in the `by` parameter, whose values are the original object's index (or tuples of indexes).

Basic Attributes:

- `.groups`

- `.get_group`
- `Iteration: for name, group in grouped`
- `.count().nunique() -- number of unique values of each group`

Example:

```
grouped=df.groupby(['A'])

grouped.groups
{0: ['a', 'b', 'c'], 1: ['d', 'e', 'f'], 2: ['g', 'h', 'i']}
```

We can see the grouped object's keys are those distinct values in column A, and original dataframe's row indexes are allocated into each group.

```
grouped.groups[0]
Index(['a', 'b', 'c'], dtype='object')

grouped.get_group(0)
   A  B  C
a  0  1  0
b  0  2  1
c  0  3  2
```

## Aggregation

An aggregation is a `GroupBy` operation that reduces the dimension of the grouping object. There are many built-in methods, such as `mean, sum, min, max, first, last` etc.

The `aggregate()` (`agg` for short) method can accept many different types of inputs

Example:

```
grouped.agg("sum")
   B   C
A
0  6   3
1  6  12
2  6  21
```

with user defined function:

```
grouped.agg(lambda x: set(x))
            B          C
A
0  {1, 2, 3}  {0, 1, 2}
1  {1, 2, 3}  {3, 4, 5}
2  {1, 2, 3}  {8, 6, 7}
```

## Transformation

An operation whose result is indexed the same as the one being grouped or to the size of group chunk. Built-in methods are `bfill, diff, rank` etc.

Example:

```
grouped.diff()
     B    C
a  NaN  NaN
b  1.0  1.0
c  1.0  1.0
d  NaN  NaN
e  1.0  1.0
f  1.0  1.0
g  NaN  NaN
h  1.0  1.0
i  1.0  1.0
```

The above example shows the value of each row subtract previous row on each group, the first row in each group off couse is NaN.

The `transform()` method can accept string aliases to the built-in transformation methods or user defined functions.

Example: fill N/A values with mean of each group:

```
df2=df.copy(deep=True)
df2.loc['a','C']=np.nan
df2.fillna(df2.groupby('A').transform('mean'))
   A  B    C
a  0  1  1.5
b  0  2  1.0
c  0  3  2.0
d  1  1  3.0
e  1  2  4.0
f  1  3  5.0
g  2  1  6.0
h  2  2  7.0
i  2  3  8.0
```

We can see that `df2.loc['a','C']` is filled with the mean of `(b,C)` and `(c,C)`.

Example: find out the sum on each group is larger than 3

```
grouped.transform(lambda x: x.sum() > 3)
      B      C
a  True  False
b  True  False
c  True  False
d  True   True
e  True   True
f  True   True
g  True   True
h  True   True
i  True   True
```

### Filtration

This operation may either filter out entire groups, part of groups, or both. Built-in functions are `head`, `tail, nth`. These built-in functions are also included in the grouped object.
The `filter()` method takes a User-Defined Function (UDF) that, when applied to an entire group, returns either True or False.

```
grouped.first()
   B  C
A
0  1  0
1  1  3
2  1  6
grouped.head(1)
   A  B  C
a  0  1  0
d  1  1  3
g  2  1  6
grouped.nth(0)
   B  C
A
0  1  0
1  1  3
2  1  6
```

## Reshape - pivot, melt, stack, and unstack

### pivot (pivot_table), melt

Data is often stored in so-called "stacked" or "wide" format; in relational database, it also commonly called "vertical" or "horizontal" (flat) format. Pivot is to change a stacked data to wide (flat) format; while melt is the reverse operation.

The `pivot_table` method can handle duplicates, and pivoting with aggregation of numeric data. Personally thinking, use `pivot_table` is used more frequently than `pivot` method. Just be aware of duplicates of '`index`' and '`columns`' combinations. In that case, aggregation will be applied.

#### Pivot

Pivot table is to use a column(s) as index, another column(s)'s distinct values (or a set of value) as columns to identify another column(s) values. As in the result, the (index, column) values will be a set of those **distinct** values.

Parameters:

- `index` – column(s) will be used as row index(s), with column name as the row index name

- `columns` – column(s), its distinct values will be used as column labels, with column name as the column label's name.
- `values` – column(s) , values under these column(s) will be shown in the new dataframe, under each columns ( '`columns`' ), the point is to use (index, columns) to identify a value.

If intended 'index' and 'column' values are unique, then the result will be a diagonal matrix form, like blow:

```
df = pd.DataFrame(np.array(range(0,9)).reshape(3,3), index=list('abc'),
columns=list('ABC'))
pivotedDf = df.pivot(index='A', columns='B', values='C')

pivotedDf
B    1    4    7
A
0  2.0  NaN  NaN
3  NaN  5.0  NaN
6  NaN  NaN  8.0

pivotedDf.columns
Int64Index([1, 4, 7], dtype='int64', name='B')
pivotedDf.index
Int64Index([0, 3, 6], dtype='int64', name='A')
```

We can see that column A label is used as index name, and column B label is used as columns name.

The trick is using 'index' and 'columns' values should be able to identify the 'values' columns value, this mean **the combination of 'index' and 'columns' values should be unique**; otherwise exception will be raised.

```
A=[0]*3 + [1]*3 + [2]*3
B=[1,2,3]*3
C=range(0,9)
df = pd.DataFrame({'A':A, 'B':B, 'C':C})

df
   A  B  C
0  0  1  0
1  0  2  1
2  0  3  2
3  0  1  3
4  1  2  4
5  1  3  5
6  2  1  6
7  2  2  7
8  2  3  8


df.pivot(index='A', columns='B', values='C')
B  1  2  3
A
0  0  1  2
```

```
1   3   4   5
2   6   7   8
```

If we change A[3]=0, to make the combination of (A,B) has duplicated (0,1), pivot will raise a duplicated index error.

```
df.iloc[3,0]=0
df
   A  B  C
0  0  1  0
1  0  2  1
2  0  3  2
3  0  1  3
4  1  2  4
5  1  3  5
6  2  1  6
7  2  2  7
8  2  3  8

df.pivot(index='A', columns='B', values='C')
ValueError: Index contains duplicate entries, cannot reshape
```

Use `pivot_table` will work:

```
df.pivot_table(index='A', columns='B', values='C')
B    1    2    3
A
0  1.5  1.0  2.0
1  NaN  4.0  5.0
2  6.0  7.0  8.0

we can see (A,B)=(0,1) shows value 1.5 which is the mean of df.iloc[0,3] and
df.iloc[3,3]
```

**Note**: Pivot parameters index and columns can be a list after version 1.1.0

If the values given to pivot is a list, then the result will be a multi-index columns data frame, with each column label in ‘values’ as top level, the ‘column’ label is below. To access each column in values’, use resultDf[‘column label’]. Below example is taken from Pandas Pivot API document:

```
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']})
df
    foo   bar  baz  zoo
0   one   A    1    x
1   one   B    2    y
2   one   C    3    z
3   two   A    4    q
```

```
4   two   B    5    w
5   two   C    6    t

pivotedDf = df.pivot(index='foo', columns='bar')
    baz        zoo
bar   A  B  C   A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t

pivotedDf['zoo']
bar  A  B  C
foo
one  x  y  z
two  q  w  t
```

### melt

The `melt` method is the reverse operation if pivot, convert flat data to stacked (horizontal to vertical)

- id_vars – `a list of columns that will be kept the same`
- value_vars – `the columns whose values need to be stacked.`

The operation will create a new dataframe, columns in the id_vars will be kept intact, add two new columns: "variable" and "value". Columns names given in value_vars will be listed in 'Variable' column, while values will be list in 'value' column. The names of those columns can be customized by supplying the var_name and value_name parameters.

```
Example:
Use the original df
df.melt(id_vars='A', value_vars='C')
   A variable  value
0  0        C      0
1  0        C      1
2  0        C      2
3  1        C      3
4  1        C      4
5  1        C      5
6  2        C      6
7  2        C      7
8  2        C      8
```

We can see column B is dropped.
Multiple columns to be melted, the variable column is a list of values in value_vars:

```
df.melt(id_vars='A', value_vars=['B','C'])
    A variable  value
0   0        B      1
1   0        B      2
2   0        B      3
3   1        B      1
4   1        B      2
5   1        B      3
6   2        B      1
7   2        B      2
8   2        B      3
```

```
9   0          C       0
10  0          C       1
11  0          C       2
12  1          C       3
13  1          C       4
14  1          C       5
15  2          C       6
16  2          C       7
17  2          C       8
```

Keep columns needed in `id_vars`:

```
df.melt(id_vars=['A','B'], value_vars='C')
   A  B variable  value
0  0  1        C      0
1  0  2        C      1
2  0  3        C      2
3  1  1        C      3
4  1  2        C      4
5  1  3        C      5
6  2  1        C      6
7  2  2        C      7
8  2  3        C      8

Revert it back:

meltedDf = meltedDf.pivot_table(index=['A','B'], columns='variable',
values='value')

meltedDf
variable  C
A B
0 1       0
  2       1
  3       2
1 1       3
  2       4
  3       5
2 1       6
  2       7
  3       8

meltedDf=meltedDf.reset_index()
meltedDf.columns.name=None
meltedDf
   A  B  C
0  0  1  0
1  0  2  1
2  0  3  2
3  1  1  3
4  1  2  4
5  1  3  5
6  2  1  6
7  2  2  7
8  2  3  8
```

## A real life example

1. add new columns [S1,S2,S3] and a cob date to the existing df
2. use melt method to stack data
3. use pivot table to revert it back

Create a new dataframe:

```
scen={
    'S1':range(11,20),
    'S2':range(21,30),
    'S3':range(31,40),
}

scen_df = pd.DataFrame(scen)
report_df = pd.concat([df, scen_df], axis=1)
report_df['COB Date']='20230929'
report_df
   A  B  C  S1  S2  S3  COB Date
0  0  1  0  11  21  31  20230929
1  0  2  1  12  22  32  20230929
2  0  3  2  13  23  33  20230929
3  1  1  3  14  24  34  20230929
4  1  2  4  15  25  35  20230929
5  1  3  5  16  26  36  20230929
6  2  1  6  17  27  37  20230929
7  2  2  7  18  28  38  20230929
8  2  3  8  19  29  39  20230929
```

Stack columns [S1,S2,S3]:

```
value_vars=['S1','S2','S3']
id_vars=(report_df.columns.difference(value_vars))

report_melted_df = report_df.melt(id_vars=id_vars, value_vars=value_vars,
var_name='Scen', value_name='Price')


report_melted_df
    A  B  C  COB Date Scen  Price
0   0  1  0  20230929   S1     11
1   0  2  1  20230929   S1     12
2   0  3  2  20230929   S1     13
3   1  1  3  20230929   S1     14
4   1  2  4  20230929   S1     15
......
21  1  1  3  20230929   S3     34
22  1  2  4  20230929   S3     35
23  1  3  5  20230929   S3     36
24  2  1  6  20230929   S3     37
25  2  2  7  20230929   S3     38
26  2  3  8  20230929   S3     39
```

Revert it back:

```
report_pivoted_df = report_melted_df.pivot_table(index=list(id_vars),
columns='Scen', values='Price')

report_pivoted_df
Scen              S1  S2  S3
A B C COB Date
0 1 0 20230929  11  21  31
  2 1 20230929  12  22  32
  3 2 20230929  13  23  33
1 1 3 20230929  14  24  34
  2 4 20230929  15  25  35
  3 5 20230929  16  26  36
2 1 6 20230929  17  27  37
  2 7 20230929  18  28  38
  3 8 20230929  19  29  39


# change indexes to columns
report_pivoted_df=report_pivoted_df.reset_index()
report_pivoted_df
Scen   A  B  C  COB Date  S1  S2  S3
0      0  1  0  20230929  11  21  31
1      0  2  1  20230929  12  22  32
2      0  3  2  20230929  13  23  33
3      1  1  3  20230929  14  24  34
4      1  2  4  20230929  15  25  35
5      1  3  5  20230929  16  26  36
6      2  1  6  20230929  17  27  37
7      2  2  7  20230929  18  28  38
8      2  3  8  20230929  19  29  39

# remove column name and rearrange columns positions
report_pivoted_df.columns.name=None

final_df=report_pivoted_df[report_df.columns]
final_df
   A  B  C  S1  S2  S3  COB Date
0  0  1  0  11  21  31  20230929
1  0  2  1  12  22  32  20230929
2  0  3  2  13  23  33  20230929
3  1  1  3  14  24  34  20230929
4  1  2  4  15  25  35  20230929
5  1  3  5  16  26  36  20230929
6  2  1  6  17  27  37  20230929
7  2  2  7  18  28  38  20230929
8  2  3  8  19  29  39  20230929
```

## stack() and unstack()

These methods are similar to pivot and melt, but designed to work with multi-indexes.
stack – similar to melt, level -> columns indexes
unstack – similar to pivot, level -> row indexes

Note

1. These two methods take a `level` parameter, for `stack`, `level` uses the column indexes; for `unstack`, `level` means the row indexes.

2. Returns either a series if the dataframe is single indexed or a dataframe if the dataframe is multi-indexed.

3. In some circumstances, need to use `set_index/reset_index` together with these two methods.

```
df2=df.set_index(['A','B'])
df2
     C
A B
0 1  0
  2  1
  3  2
1 1  3
  2  4
  3  5
2 1  6
  2  7
  3  8
```

```
stacked = df2.stack()
stacked
A  B
0  1  C    0
   2  C    1
   3  C    2
1  1  C    3
   2  C    4
   3  C    5
2  1  C    6
   2  C    7
   3  C    8
dtype: int64
```

We can see the operation is similar to melt(), column ‘C’ becomes the ‘variable’ column, and its value become the values of the series.

```
type(stacked)
<class 'pandas.core.series.Series'>
```

```
stacked.index
MultiIndex([(0, 1, 'C'),
            (0, 2, 'C'),
            (0, 3, 'C'),
            (1, 1, 'C'),
            (1, 2, 'C'),
            (1, 3, 'C'),
            (2, 1, 'C'),
            (2, 2, 'C'),
```

```
              (2, 3, 'C')],
            names=['A', 'B', None])
```

To revert it back:

```
stacked.unstack()
     C
A B
0 1  0
  2  1
  3  2
1 1  3
  2  4
  3  5
2 1  6
  2  7
  3  8
```

```
stacked.unstack().reset_index()
   A  B  C
0  0  1  0
1  0  2  1
2  0  3  2
3  1  1  3
4  1  2  4
5  1  3  5
6  2  1  6
7  2  2  7
8  2  3  8
```

## References:


1. Pandas API official document site: https://pandas.pydata.org/docs/reference
2. Pandas user guide: https://pandas.pydata.org/docs/user_guide/
3. https://pandas.pydata.org/docs/reference/arrays.html
4. https://pandas.pydata.org/docs/reference/series.html
5.