ĐẠI HỌC ĐÀ NẴNG
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG VIỆT - HÀN
Vietnam - Korea University of Information and Communication Technology

Chapter 6

Data Manipulation with Pandas

Thu Huong Nguyen, PhD
Si Thin Nguyen, PhD

KHMT
Computer Science

http://vku.udn.vn/

About Authors

Thu Huong Nguyen

PhD in Computer Science at Université Côte d’Azur, France

Email: nthuong@vku.udn.vn

Address: Faculty of Computer Science, VKU

Si Thin Nguyen

PhD in Computer Science at Soongsil University, Korea

Email: nsthin@vku.udn.vn

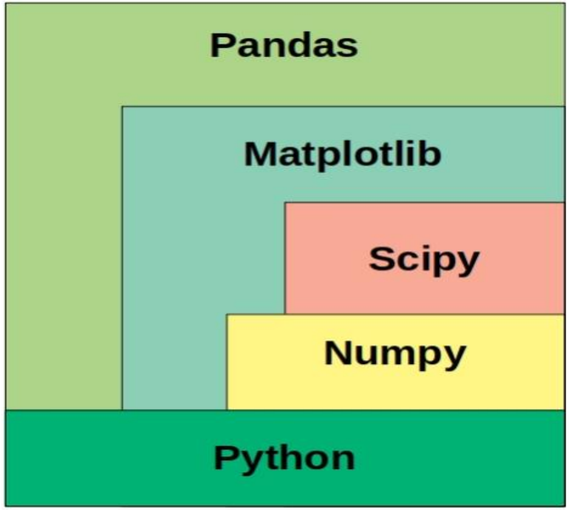
Address: Faculty of Computer Science, VKU

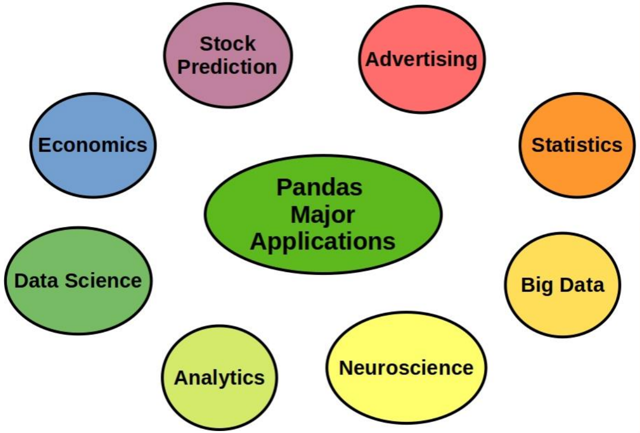
Chapter Content

- ➢ Introduction
- ➢ Pandas Getting Started With
- ➢ Pandas vs SQL
- ➢ Pandas Features
- ➢ Pandas Data Structure
- ➢ Operations on Pandas
- ➢ Working with text data
- ➢ Working with time series data

Introduction

→ **Pandas** is an essential tool to data analysis and manipulation.





Pandas Getting Started

➔ Installing Numpy: `pip install pandas`

➔ Import Numpy: `import pandas`

➔ Alias of Numpy: `import pandas as pd`

➔ Check Numpy version: `pd.__version__`

Why uses Pandas?

→ one of the most widely used data science libraries in the world.

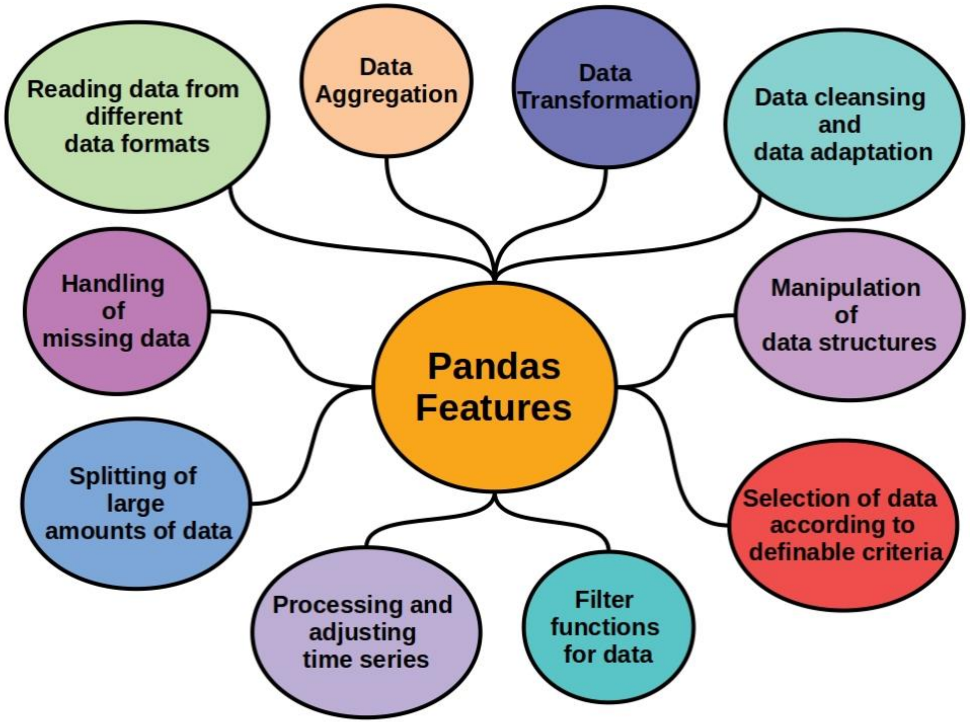
→ capable of handling huge sets of data.

→ Useful for data analysis and machine learning

→ Working with data in a new way

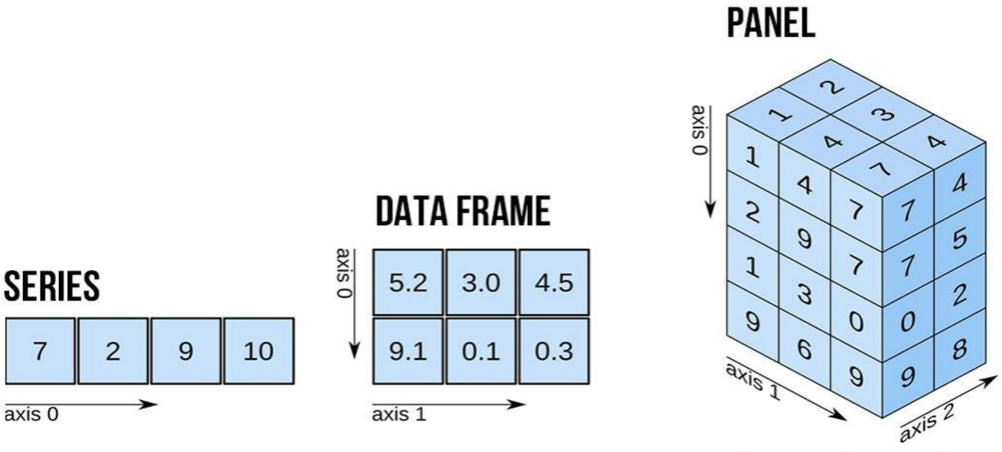
→ *"to master data science, you must be skillful in Pandas"*

Pandas Features



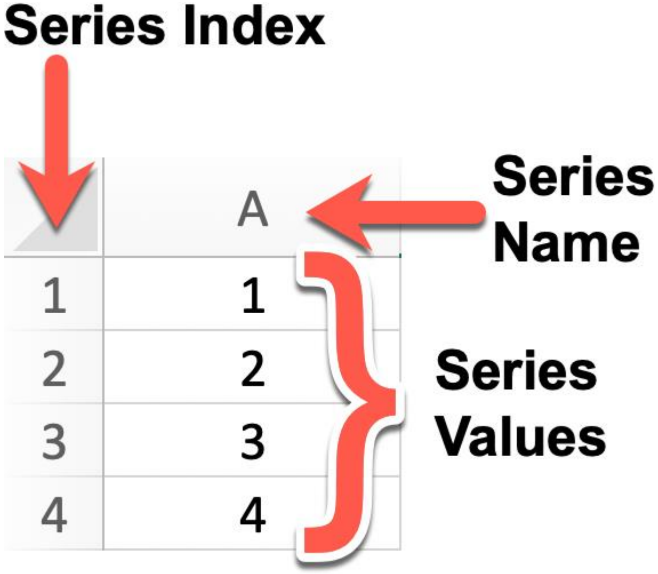
Pandas Data Structure

Data Structure	Dimension	Description
Series	1	• 1Dimentional • Size Immutable • Value of Data Mutable
Data-Frame	2	• 2Dimentional • Size Mutable • Heterogeneous typed columns
Panel	3	• 3Dimentional • Size Mutable



Pandas Series

➔ a one-dimensional labeled array capable of holding any data type

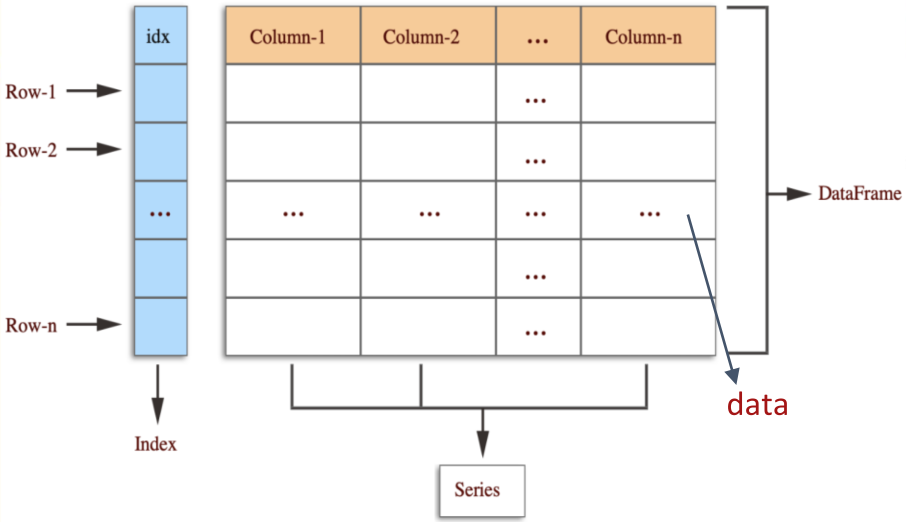



```
s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])
s
```

```
a    0.115188
b    0.893129
c    0.659912
d    2.502990
e   -0.956800
dtype: float64
```

Pandas DataFrame

➔ is a 2 dimensional data structure with mutable size and potentially heterogeneous tabular data.




```python
import pandas as pd
#create a simple dataframe of people
df = pd.DataFrame([["Anna","New York", 20], ["Peter","Paris", 15], ["John","London", 18]],
     columns=['Name','Location', 'Age'])
display(df)
```

	Name	Location	Age
0	Anna	New York	20
1	Peter	Paris	15
2	John	London	18

Pandas DataFrame Creation

➔ `pandas.DataFrame([data, index, columns, dtype, copy])`

- data – the data from which the dataframe will be made
- index – states the index from dataframe
- columns – states the column label
- dtype – the datatype for the dataframe
- copy – any copied data taken from inputs (True/False)

➔ Create an empty DataFrame

```python
import pandas as pd
df = pd.DataFrame()
```

➔ Create a DataFrame from the inputs like *dictionaries*, *ndarrays*, *Series*, *Lists*.

Pandas Panel

- → a 3-dimensional container of data
- → its origins in econometrics
- → partially responsible for the name of the library pandas: **pan**el dat**as**.

Pandas Panel Creation

➔ `pandas.Panel(data, items, major_axis, minor_axis, dtype, copy)`

- data: the data will be represented by the panel.
- items: can represent and compare to a DataFrame.
- major-axis: rows of a DataFrame.
- minor-axis: columns of a DataFrame.
- copy: Boolean value to denote whether data will be copied from inputs.
- dtype: specifies a data type.

Operations on Pandas

→ Retrieving Data from **csv** file

→ Handling of missing data

→ Data Extraction/Filter

→ Data Addition/Deletion

→ Concatenation DataFrame

→ Merging /Joining DataFrame

→ Data Grouping

Retrieving Data from CSV

- ➔ **CSV *(comma-separated value)*** files are a common file format for transferring and storing data.
- ➔ Using `read_csv()` function to retrieve data from CSV file , where the delimiter is a comma character.
- ➔ Demo

Handling of missing data

	ID	Name	Age	Address	Qualification
0	I0	John	27	Chicago	Btech
1	I1	Jim	24	NaN	NaN
2	I2	Jackson	22	Texas	B.A
3	I3	Amy	32	New York	Bcom

Missing data

➔ a very big problem in a real-life scenarios
➔ it exists and was not collected or it never existed
➔ represented for None and NaN (Not a Number) indicating missing or null values

➔ Checking for missing values using `isnull()` and `notnull()`
➔ Filling missing values using `fillna()`, `replace()` and `interpolate()`
➔ Dropping missing values using `dropna()`

Data Extraction

	Name	Location	Age
0	Anna	New York	20
1	Peter	Paris	15
2	John	London	18

	Location	Age
Name		
Anna	New York	20
Peter	Paris	15
John	London	18

➔ Extract a column data of DataFrame by calling it by the column name.

```
df[['Location']]
```

	Location
Anna	New York
Peter	Paris
John	London

➔ Extract a row data of DataFrame by using method `loc()` and `iloc()`.

```
df.loc[‘Peter’]
df.iloc[1]
```

```
Location    Paris
Age            15
Name: Peter, dtype: object
```

Data Filter

➔ Using `filter()` method

```
DataFrame.filter(items, like, regex, axis)
```

- **`item`** – Takes list of axis labels that need to filter.
- **`like`** – Takes axis string label that need to filter
- **`regex`** – regular expression
- **`axis`** – *{0 or 'index', 1 or 'columns', None}, default None*. When not specified it used columns.

Examples

	Location	Age
Anna	New York	20
Peter	Paris	15
John	London	18

```
df.filter(items=['Location'])
```

	Location
Anna	New York
Peter	Paris
John	London

```
df.filter(regex='e$', axis=1)
```

	Age
Anna	20
Peter	15
John	18

```
df.filter(like='er', axis=0)
```

	Location	Age
Peter	Paris	15

Data Addition

➔ Adding a new column data: declare a new list as a column data and add to a existing Dataframe

```
# Declare a list that is to be converted into a column
height = [1.6, 1.8, 1.5]
#Using 'Height' as the column name and equating it to the list
df['Height'] = height
```

	Location	Age	Height
Anna	New York	20	1.6
Peter	Paris	15	1.8
John	London	18	1.5

➔ Adding a new row data: concat the old dataframe with new one

```
#create a new row
new_row = pd.DataFrame({'Name':['Hoa'], 'Location':['Vietnam'], 'Age':[16], 'Height':[1.55]}, index=[0])
#concatenating the new row with the old dataframe
df=pd.concat([new_row,df]).reset_index(drop=True)
```

	Name	Location	Age	Height
0	Hoa	Vietnam	16	1.55
1	Anna	New York	20	1.60
2	Peter	Paris	15	1.80
3	John	London	18	1.50

Data Deletion

➔ Using the drop() method

➔ Delete a column:

```
#Droping columns with column names
df.drop(["Location"],axis=1,  inplace = True)
```

	Name	Age	Height
0	Hoa	16	1.55
1	Anna	20	1.60
2	Peter	15	1.80
3	John	18	1.50

➔ Deleting a new row: concat the old dataframe with new one

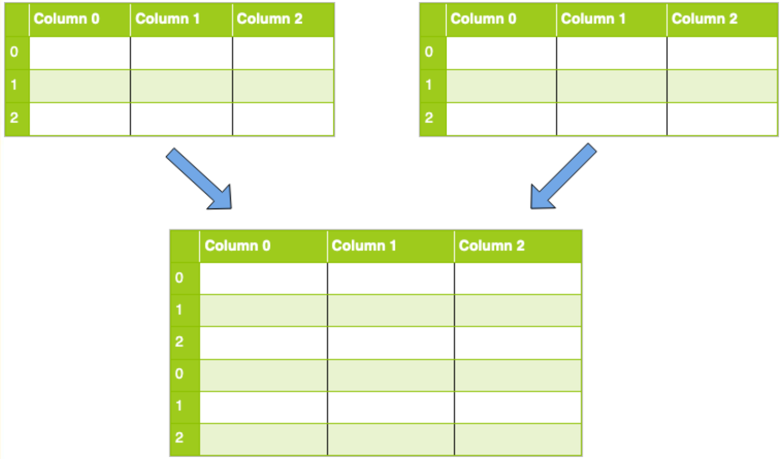
```
#Droping row with index labels
df.drop([3], inplace = True)
```

	Name	Age	Height
0	Hoa	16	1.55
1	Anna	20	1.60
2	Peter	15	1.80

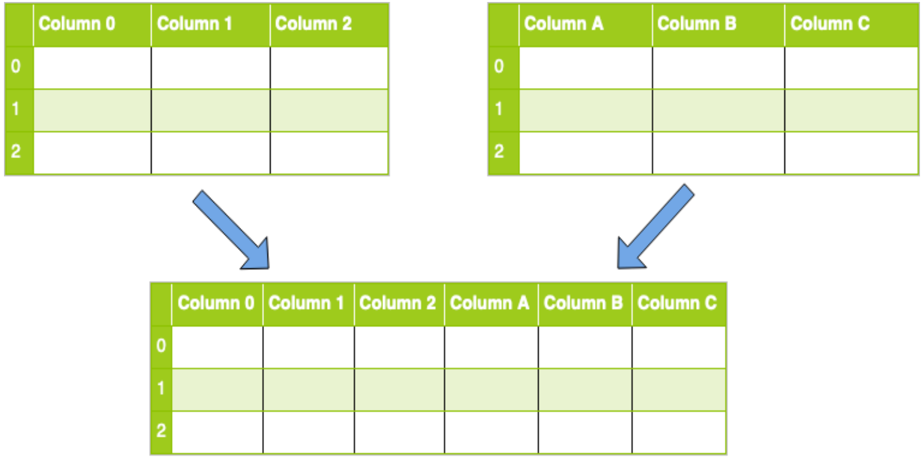
Concatenating DataFrame

→ Using `concat()` method to combine DataFrames across axes

- **axis =0** (rows)



- **axis=1** (columns)



Concatenating DataFrame

- With setting different logic on axes
  - Taking the union of them all with the argument **join**='outer' (default)
  - Taking the intersection with the argument **join**='inner'
- With ignoring indexes with the argument **ignore_index**=True
- With group keys with the argument **keys**

Examples

	Name	Location	Age
0	Anna	New York	20
1	Peter	Paris	15
2	John	London	18

	Name	Location	Age
3	Jim	Hensiki	21
2	John	London	29

```
# concating dataframe
res=pd.concat([df,df2])
```

	Name	Location	Age
0	Anna	New York	20
1	Peter	Paris	15
2	John	London	18
3	Jim	Hensiki	21
2	John	London	18

```
# concating dataframe with axes and join='outer'
res2 = pd.concat([df, df2], axis=1, sort=False)
```

	Name	Location	Age	Name	Location	Age
0	Anna	New York	20.0	NaN	NaN	NaN
1	Peter	Paris	15.0	NaN	NaN	NaN
2	John	London	18.0	John	London	29.0
3	NaN	NaN	NaN	Jim	Hensiki	21.0

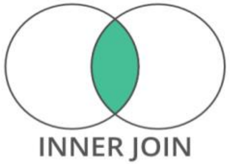
```
# concating dataframe with axes and join='inner'
res=pd.concat([df,df2], axis=1,join='inner')
```

	Name	Location	Age	Name	Location	Age
2	John	London	18	John	London	29

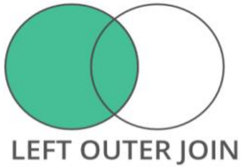
Data Merging/Joining

➔ Using `merge()`method to combine data on common columns or indices.

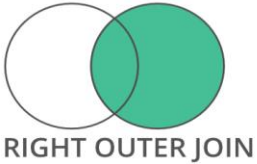
➔ Using `join()`method to combine the columns of two differently-indexed DataFrames into a single result DataFrame based on a <u>key column</u> or an <u>index</u>.



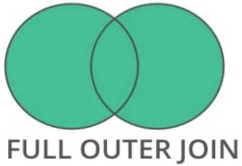
<u>**how**= ‘inner’</u>



<u>**how**= ‘left’</u>



<u>**how**= ‘right’</u>



<u>**how**= ‘outer’</u>

- with one unique key combination
  <u>**on** =[key]</u>
- using multiple join keys
  <u>**on**=[key1,key2,...]</u>

Joining Examples

	Name	Age
I0	John	27
I1	Jim	24
I2	Jackson	22
I3	Amy	32

	Address	Qualification
I0	Chicago	Btech
I2	Texas	B.A
I3	New York	Bcom
I4	Florida	B.hons

```python
# using join method to join dataframes (based on index)
res2 = df.join(df1)
```

	Name	Age	Address	Qualification
I0	John	27	Chicago	Btech
I1	Jim	24	NaN	NaN
I2	Jackson	22	Texas	B.A
I3	Amy	32	New York	Bcom

```python
# getting union
res1 = df.join(df1, how='outer')
```

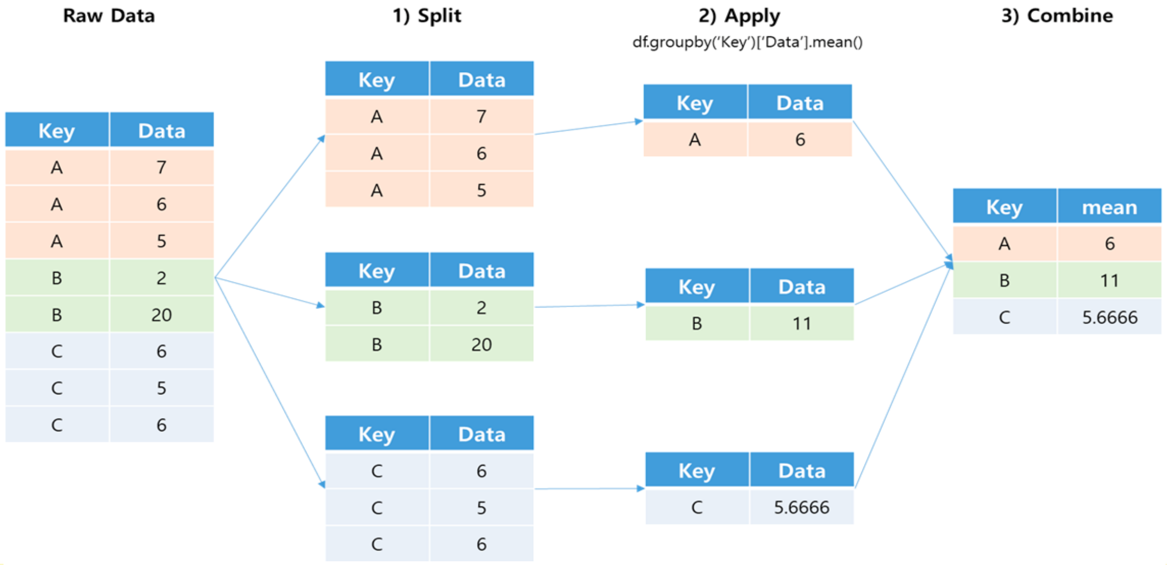
	Name	Age	Address	Qualification
I0	John	27.0	Chicago	Btech
I1	Jim	24.0	NaN	NaN
I2	Jackson	22.0	Texas	B.A
I3	Amy	32.0	New York	Bcom
I4	NaN	NaN	Florida	B.hons

Data Grouping

➔ grouping the data according to the categories and apply a function to the categories.

➔ often involves 3 operations:

- ***Splitting the Data Object***
- ***Applying a function***
- ***Combining the results***



Examples of Splitting Data Objects

Using groupby() for splitting the dataframe over some criteria into data subsets.

	Key	Data
0	A	7
1	A	6
2	A	5
3	B	2
4	B	20
5	C	6
6	C	5
7	C	6

- groupby('key')

```
df.groupby('Key').groups
```

```
{'A': [0, 1, 2], 'B': [3, 4], 'C': [5, 6, 7]}
```

- groupby(['key1','key2'])

```
df.groupby(['Key','Data']).groups
```

```
{('A', 5): [2], ('A', 6): [1], ('A', 7): [0], ('B', 2): [3], ('B', 20): [4], ('C', 5): [6], ('C', 6): [5, 7]}
```

Statistical functions in Pandas

➔ computing a summary statistic

sum()	Compute sum of column values
min()	Compute min of column values
max()	Compute max of column values
mean()	Compute mean of column
size()	Compute column sizes
describe()	Generates descriptive statistics

first()	Compute first of group values
last()	Compute last of group values
count()	Compute count of column values
std()	Standard deviation of column
var()	Compute variance of column
sem()	Standard error of the mean of column

Transformation Functions

→ Returns a self-produced dataframe with transformed values after applying the function specified in its parameter.

- `apply()`
- `applymap()`
- `melt()`
- `transform()`

`apply ()` function

➔ Used to apply a function along an axis of the DataFrame.

```
DataFrame.apply( func, axis, raw, reduce=None, result_type, args=(), **kwds)
```

- `func:` Function to apply to each column or row.
- `axis`: Axis along which the function is applied: 0 or ‘index’: apply function to each column; 1 or ‘columns’: apply function to each row.
- `raw`: False - passes each row or column as a Series to the function ; True - the passed function will receive ndarray objects instead.
- `result_type:` only act when axis=1 (columns): ‘expand’ : list-like results will be turned into columns; 'reduce' : returns a Series if possible rather than expanding list-like results; 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.
- `arg():` Positional arguments to pass to func in addition to the array/series.
- `**kwds:` Additional keyword arguments to pass as keywords arguments to func.

Examples

	P	Q
0	3.0	5.0
1	3.0	5.0
2	3.0	5.0

```
df.apply(np.sum, axis=1)
```

```
0    34
1    34
2    34
dtype: int64
```

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

```
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

```
df.apply(lambda x: [1, 2], axis=1, result_type='expand')
```

	0	1
0	1	2
1	1	2
2	1	2

```
df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
```

	P	Q
0	1	2
1	1	2
2	1	2

`applymap ()` function

➔ Used to apply a function to a Dataframe elementwise.

```
DataFrame.apply( func)
```

- `func`: Python function, returns a single value from a single value.

	0	1
0	2.000	3.120
1	4.356	5.567

```
df.applymap(lambda x: len(str(x)))
```

	0	1
0	3	4
1	5	5

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

	0	1
0	4.000000	9.734400
1	18.974736	30.991489

`melt()` function

➔ used to unpivot a given DataFrame from wide format to long format

`DataFrame.melt([id_vars], [value_vars],var_name, value_name, [col_level])`

- `[id_vars]`: column(s) to use as identifier variables.
- `[value_vars]`: column(s) to unpivot. If not specified, uses all columns that are not set as id_vars.
- `var_name`: name to use for the 'variable' column. If None it uses frame.columns.name or 'variable'.
- `value_name`: name to use for the 'value' column.
- `[col_level]`: if columns are a MultiIndex then use this level to melt.

Examples

	P	Q	R
0	p	1	2
1	q	3	4
2	r	5	6

```
df.melt(id_vars=['P'], value_vars=['Q'])
```

	P	variable	value
0	p	Q	1
1	q	Q	3
2	r	Q	5

```
df.melt(id_vars=['P'], value_vars=['Q', 'R'])
```

	P	variable	value
0	p	Q	1
1	q	Q	3
2	r	Q	5
3	p	R	2
4	q	R	4
5	r	R	6

```
df.melt(id_vars=['P'], value_vars=['Q'],
        var_name='myVarname', value_name='myValname')
```

	P	myVarname	myValname
0	p	Q	1
1	q	Q	3
2	r	Q	5

```
df.melt(col_level=0, id_vars=['P'], value_vars=['Q'])
```

	P	variable	value
0	p	Q	1
1	q	Q	3
2	r	Q	5

`transform()` function

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

```
DataFrame.transform(func, axis, *args, **kwargs)
```

- `func`: Function to use for transforming the data
- `axis`: 0 or ‘index’: apply function to each column; 1 or ‘columns’: apply function to each row.
- `*args`: Positional arguments to pass to func.
- `**kwargs`: Keyword arguments to pass to func.

Examples

	X	Y
0	0	2
1	1	3
2	2	4
3	3	5

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

	X	Y
0	1	3
1	2	4
2	3	5
3	4	6

```
df.transform([np.sqrt, np.exp])
```

	X		Y	
	sqrt	exp	sqrt	exp
0	0.000000	1.000000	1.414214	7.389056
1	1.000000	2.718282	1.732051	20.085537
2	1.414214	7.389056	2.000000	54.598150
3	1.732051	20.085537	2.236068	148.413159

Categorical Data

- ➔ a pandas data type corresponding to categorical variables in statistics, e.g. gender, social class, blood type,....
- ➔ Using the standard pandas `Categorical` constructor to create categorical object

`pandas.Categorical(values, categories, ordered)`

```
cat = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'], categories=["b","a","c"], ordered=True)
cat
```

```
['a', 'b', 'c', 'a', 'b', 'c']
Categories (3, object): ['b' < 'a' < 'c']
```

Working with time series data

- ➔ A ***time series*** is any data set where the values are measured at different points in time.
  - ◆ uniformly spaced. Eg. *hourly weather measurements*, *daily counts of web site visits*, or *monthly sales totals*.
  - ◆ irregularly spaced. Eg. *timestamped data in a computer system's event log*, *a history of 115 emergency calls*
- ➔ Pandas provides useful objects in working with time series data:
  - ◆ Timestamp Object
  - ◆ Period Object
  - ◆ Timedelta Object

Period Object

➔ Period object represents an interval in time used to check if a specific event occurs within a certain period such as when monitoring the number of flights taking off or the average stock price during a period.

```
# Create time period
p1 = pd.Period('2020-12-25')
# Create time stamp
t1 = pd.Timestamp('2020-12-25 18:12')
# Test Time interval
p1.start_time < t1 < p1.end_time
```

The function to_period()

➔ Used to convert a DatetimeIndex object to a PeriodIndex

```
period _daily= dates.to_period('D')
# output
PeriodIndex(['2020-12-25', '2020-07-04', '2018-10-06', '2017-07-07',
             '2020-05-08', '2020-04-22'],
            dtype='period[D]', freq='D')
```

```
period _daily= dates.to_period('M')
# output
PeriodIndex(['2020-12', '2020-07', '2018-10', '2017-07',
             '2020-05', '2020-04'],
            dtype='period[M]', freq='M')
```

Timedelta Object

➔ represents the temporal difference between two `datetime`objects used to calculate the difference between two dates.

```
# Subtract a specific date from dates
dates - pd.to_datetime('2020-05-15')
TimedeltaIndex(['224 days 00:00:00','50 days 00:00:00',
                 '-587 days +00:00:00', '-1043 days +00:00:00',
                   '-7 days +00:00:00',   '-23 days +20:34:48'],
               dtype='timedelta64[ns]', freq=None)
```

Date Range and Frequency

➔ Regular date sequences can be created using functions:

- The function `date_range()` for *timestamp*
- The function `period_range()` for *periods*
- The function `timedelta_range()` for *time deltas*

Working with textual data

lower()	Converts strings in the Series/Index to lower case.
upper()	Converts strings in the Series/Index to upper case.
len()	Computes String length().
strip()	Helps strip whitespace(including newline) from each string in the Series/index from both the sides.
split(' ')	Splits each string with the given pattern.
cat(sep=' ')	Concatenates the series/index elements with given separator.
get_dummies()	Returns the DataFrame with One-Hot Encoded values.
contains(pattern)	Returns a Boolean value True for each element if the substring contains in the element, else False.
replace(a,b)	Replaces the value a with the value b.