WEEK 14

Introduction to Python Data Analysis (Pandas)

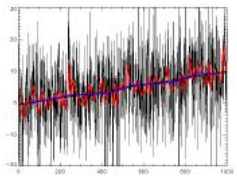
Topics

- Introduction to Pandas
- Data Ingestion
- Descriptive Statistics
- Data Cleaning
- Data Visualisation
- Frequent Data Operations
- Merging Data Frames
- Frequent String Operations
- Parsing Timestamps
- Case Study: Movie Rating Notebook

Introduction to Pandas

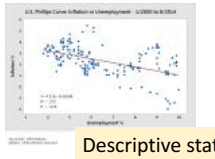
pandas Benefits

pandas



Support for time-series data

Visualizations

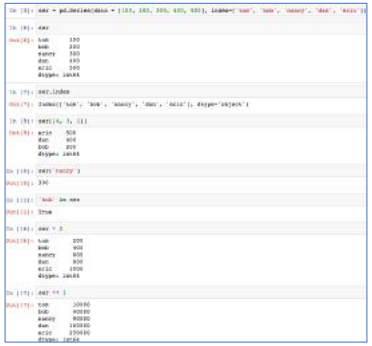


Descriptive statistics

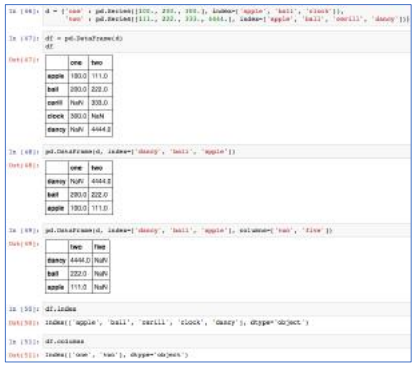
Image Source: http://www.kdnuggets.com/wp-content/uploads/data-variety.png

- Data variety support
- Data integration
- Data transformation

pandas
Data Structures

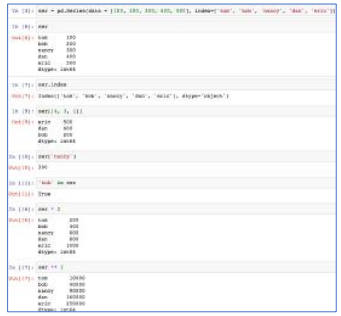


pandas Series



pandas DataFrame

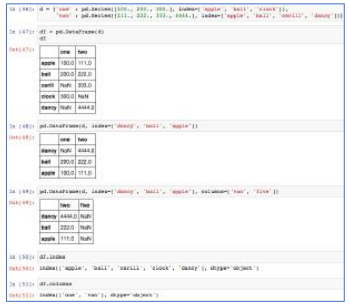
pandas Series



- A 1-dimensional labeled array
- Supports many data types
- Axis labels → index
  - get and set values by index label
- Valid argument to most NumPy methods

pandas DataFrame

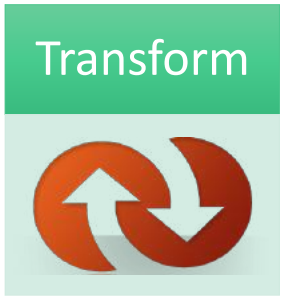
- A 2-dimensional labeled data structure
- A dictionary of Series objects
  - Columns can be of potentially different types
  - Optionally parameters for fine-tuning:
    - index (row labels)
    - columns (column labels)



Pandas provides many constructors to create DataFrames!

Summary

Pandas supports all steps of DS pipeline

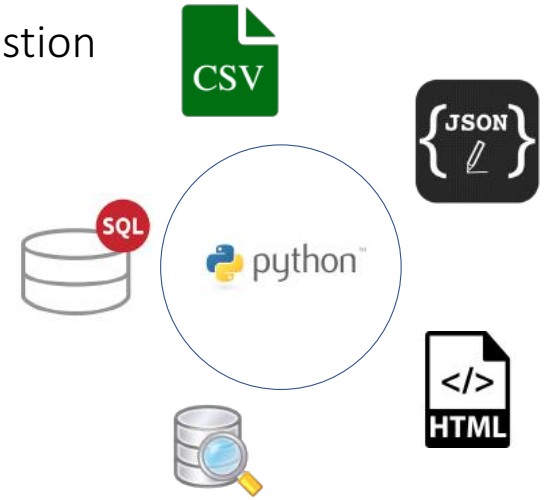


Data Ingestion

By the end of this video, you should be able to:

- Describe the efficient and easy to use methods that *pandas* provides for importing data into memory
- Identify functions such as '`read_csv`' for reading a CSV file into a DataFrame
- Discuss about other data sources that *pandas* can directly import from

Data Ingestion



read_csv

- Input : Path to a Comma Separated File
- Output: Pandas DataFrame object containing contents of the file

movies.csv

```
1,Toy Story (1995),Adventure|Animation|Children|Comedy|Fantasy
2,Jumanji (1995),Adventure|Children|Fantasy
3,Grumpier Old Men (1995),Comedy|Romance
4,Waiting to Exhale (1995),Comedy|Drama|Romance
5,Father of the Bride Part II (1995),Comedy
6,Heat (1995),Action|Crime|Thriller
7,Sabrina (1995),Comedy|Romance
8,Tom and Huck (1995),Adventure|Children
9,Sudden Death (1995),Action
10,GoldenEye (1995),Action|Adventure|Thriller
11,"American President, The (1995)",Comedy|Drama|Romance
12,Dracula: Dead and Loving It (1995),Comedy|Horror
13,Balto (1995),Adventure|Animation|Children
14,Nixon (1995),Drama
15,Cutthroat Island (1995),Action|Adventure|Romance
16,Casino (1995),Crime|Drama
17,Sense and Sensibility (1995),Drama|Romance
18,Four Rooms (1995),Comedy
19,Ace Ventura: When Nature Calls (1995),Comedy
20,Money Train (1995),Action|Comedy|Crime|Drama|Thriller
21,Get Shorty (1995),Comedy|Crime|Thriller
22,Copycat (1995),Crime|Drama|Horror|Mystery|Thriller
23,Assassins (1995),Action|Crime|Thriller
24,Powder (1995),Drama|Sci-Fi
25,Leaving Las Vegas (1995),Drama|Romance
26,Othello (1995),Drama
27,Now and Then (1995),Children|Drama
28,Persuasion (1995),Drama|Romance
29,"City of Lost Children, The (Cité des enfants perdus, La) (1995)",Adventure|Drama|
Fantasy|Mystery|Sci-Fi
```

read_json

- Input : Path to a JSON file or a valid JSON String
- Output: Pandas DataFrame or a Series object containing the contents

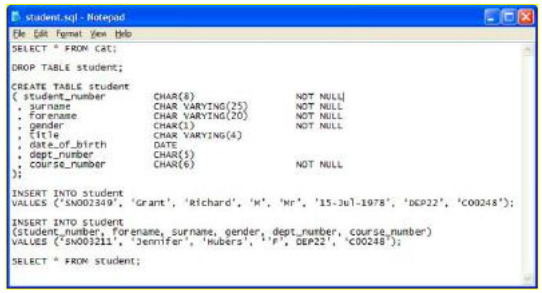
read_html

- Input : A URL or a file or a raw HTML String
- Output: A list of Pandas DataFrames


```
<!DOCTYPE html>
<html>
    <head>
        <title>Example<title>
        <link rel="stylesheet" href="sty
    </head>
    <body>
        <h1>
            <a href="/">Header</a>
        </h1>
        <nav>
            <a href="one/">One</a>
            <a href="two/">Two</a>
            <a href="three/">Three</a>
        </nav>
```

read_sql_query

- Input1 : SQL Query
- Input2 : Database connection
- Output: Pandas DataFrame object containing contents of the file



```
student.sql - Notepad
File  Edit  Format  View  Help
SELECT * FROM Cat;

DROP TABLE student;

CREATE TABLE student
( student_number        CHAR(8)                 NOT NULL
, surname               CHAR VARYING(25)        NOT NULL
, forename              CHAR VARYING(20)        NOT NULL
, gender                CHAR(1)                 NOT NULL
, title                 CHAR VARYING(4)
, date_of_birth         DATE
, dept_number           CHAR(5)
, course_number         CHAR(6)                 NOT NULL
);

INSERT INTO student
VALUES ('SN002349', 'Grant', 'Richard', 'M', 'Mr', '15-Jul-1978', 'DEP22', 'C00248');

INSERT INTO student
(student_number, forename, surname, gender, dept_number, course_number)
VALUES ('SN003211', 'Jennifer', 'Hubers', 'F', DEP22', 'C00248');

SELECT * FROM student;
```

Image Source: http://www.sqa.org.uk/e-learning/SQLIntro01CD/images/pic024.jpg

read_sql_table

- Input1 : Name of SQL table in database
- Input2 : Database connection
- Output : Pandas DataFrame object containing contents of the table

select * from bookstore

ISBN_NO	SHORT_DESC	AUTHOR	PUBLISHER	PRICE
0201703092	The Practical SQL, Fourth Edition	Judith S. Bowman	Addison Wesley	39
0471777781	Professional Ajax	Jeremy McPeak, Joe Fawcett	Wrox	32
0672325764	Sams Teach Yourself XML in 21 Days, Third Edition	Steven Holzner	Sams Publishing	49
0764557599	Professional C#	Simon Robinson and Jay Glynn	Wrox	42
0764579088	Professional JavaScript for Web Developers	Nicholas C. Zakas	Wrox	35
1861002025	Professional Visual Basic 6 Databases	Charles Williams	Wrox	38
1861006314	GDI+ Programming: Creating Custom Controls Using C#	Eric White	Wrox	29

Image Source: http://www.w3processing.com/SQL/images/SQL002.png

Summary

- There are many other methods available in Pandas to ingest data:
  - Google Big Query
  - SAS files
  - Excel tables
  - Clipboard contents
  - Pickle files
  - http://pandas.pydata.org/pandas-docs/stable/api.html#input-output

Descriptive Statistics

describe()

- Syntax : data_frame.describe()
- Output: Shows summary statistics of the dataframe

```
ratings['rating'].describe()
```

```
count    2.000026e+07
mean     3.525529e+00
std      1.051989e+00
min      5.000000e-01
25%      3.000000e+00
50%      3.500000e+00
75%      4.000000e+00
max      5.000000e+00
Name: rating, dtype: float64
```

corr()

- Syntax: data_frame.corr()
- Computes pairwise Pearson coefficient (ρ) of columns
- Other coefficients available: Kendall, Spearman

Covariance

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y}$$

Standard deviation

func = min(), max(), mode(), median()

- The general syntax for calling these functions is
  - data_frame.func()
  - Frequently used optional parameter:
    - axis = 0 (rows) or 1 (columns)

mean()

- Syntax: data_frame.mean(axis={0 or 1})
  - Axis = 0 : Index
  - Axis = 1 : Columns
- Output: Series or DataFrame with the mean values

std()

- Syntax: data_frame.std(axis={0 or 1})
  - Axis = 0 : Index
  - Axis = 1 : Columns
- Output: Series or DataFrame with the Standard Deviation values
  - Normalized by N-1

any()

- Output: Returns whether ANY element is True
- Benefits:
  - Can detect if a cell matches a condition very quickly

all()

- Output: Returns whether ALL element is True
- Benefits:
  - Can detect if a column or row matches a condition very quickly

Summary

- Some other functions that are worth exploring:
  - Count()
  - Clip()
  - Rank()
  - Round()
  - http://pandas.pydata.org/pandas-docs/stable/api.html#api-dataframe-stats

Data Cleaning

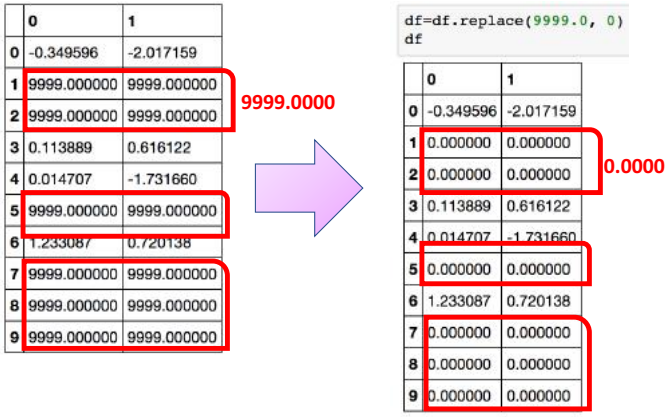
Real-world data is messy!

- Missing values
- Outliers in the data
- Invalid data (e.g. negative values for age)
- NaN value (np.nan)
- None value

Handling Data Quality Issues

- Replace the value
- Fill gaps forward / backward
- Drop fields
- Interpolation

df.replace()



	0	1
0	-0.349596	-2.017159
1	9999.000000	9999.000000
2	9999.000000	9999.000000
3	0.113889	0.616122
4	0.014707	-1.731660
5	9999.000000	9999.000000
6	1.233087	0.720138
7	9999.000000	9999.000000
8	9999.000000	9999.000000
9	9999.000000	9999.000000

	0	1
0	-0.349596	-2.017159
1	0.000000	0.000000
2	0.000000	0.000000
3	0.113889	0.616122
4	0.014707	-1.731660
5	0.000000	0.000000
6	1.233087	0.720138
7	0.000000	0.000000
8	0.000000	0.000000
9	0.000000	0.000000

df.replace()

	0	1
0	-0.349596	-2.017159
1	9999.000000	9999.000000
2	9999.000000	9999.000000
3	0.113889	0.616122
4	0.014707	-1.731660
5	9999.000000	9999.000000
6	1.233087	0.720138
7	9999.000000	9999.000000
8	9999.000000	9999.000000
9	9999.000000	9999.000000

9999.0000

```
df=df.replace(9999.0, 0)
df
```

	0	1
0	-0.349596	-2.017159
1	0.000000	0.000000
2	0.000000	0.000000
3	0.113889	0.616122
4	0.014707	-1.731660
5	0.000000	0.000000
6	1.233087	0.720138
7	0.000000	0.000000
8	0.000000	0.000000
9	0.000000	0.000000

0.0000

Fill missing data gaps forward and backward

```
df
```

	0	1
0	0.061038	1.339673
1	NaN	NaN
2	1.578293	0.637435
3	NaN	NaN
4	NaN	NaN
5	NaN	NaN
6	NaN	NaN
7	NaN	NaN
8	NaN	NaN
9	-1.145787	0.052887

```
df.fillna(method='ffill')
```

	0	1
0	0.061038	1.339673
1	0.061038	1.339673
2	1.578293	0.637435
3	1.578293	0.637435
4	1.578293	0.637435
5	1.578293	0.637435
6	1.578293	0.637435
7	1.578293	0.637435
8	1.578293	0.637435
9	-1.145787	0.052887

```
df.fillna(method='backfill')
```

	0	1
0	0.061038	1.339673
1	1.578293	0.637435
2	1.578293	0.637435
3	-1.145787	0.052887
4	-1.145787	0.052887
5	-1.145787	0.052887
6	-1.145787	0.052887
7	-1.145787	0.052887
8	-1.145787	0.052887
9	-1.145787	0.052887

http://pandas.pydata.org/pandas-docs/stable/missing_data.html

Drop fields using dropna()

```
df
```

	0	1	2
0	NaN	NaN	-0.335410
1	NaN	NaN	0.685743
2	0.077005	0.085073	0.565144
3	0.394961	-1.829587	0.494039
4	1.486227	-0.480726	-0.127278
5	NaN	NaN	-0.047668
6	NaN	NaN	-1.504804
7	-0.530518	-0.881817	-2.687352
8	0.825376	1.042468	-0.311527
9	0.097617	1.373572	-0.682435

```
df.dropna(axis=0)
```

	0	1	2
2	0.077005	0.085073	0.565144
3	0.394961	-1.829587	0.494039
4	1.486227	-0.480726	-0.127278
7	-0.530518	-0.881817	-2.687352
8	0.825376	1.042468	-0.311527
9	0.097617	1.373572	-0.682435

```
df.dropna(axis=1)
```

	2
0	-0.335410
1	0.685743
2	0.565144
3	0.494039
4	-0.127278
5	-0.047668
6	-1.504804
7	-2.687352
8	-0.311527
9	-0.682435

Drop fields using dropna() – axis=0

```
df
```

	0	1	2
0	NaN	NaN	-0.335410
1	NaN	NaN	0.685743
2	0.077005	0.085073	0.565144
3	0.394961	-1.829587	0.494039
4	1.486227	-0.480726	-0.127278
5	NaN	NaN	-0.047668
6	NaN	NaN	-1.504804
7	-0.530518	-0.881817	-2.687352
8	0.825376	1.042468	-0.311527
9	0.097617	1.373572	-0.682435

```
df.dropna(axis=0)
```

	0	1	2
2	0.077005	0.085073	0.565144
3	0.394961	-1.829587	0.494039
4	1.486227	-0.480726	-0.127278
7	-0.530518	-0.881817	-2.687352
8	0.825376	1.042468	-0.311527
9	0.097617	1.373572	-0.682435

Drop fields using dropna() -- axis=1

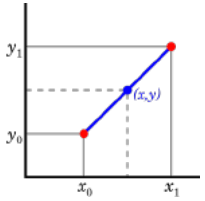
```
df
```

	0	1	2
0	NaN	NaN	-0.335410
1	NaN	NaN	0.685743
2	0.077005	0.085073	0.565144
3	0.394961	-1.829587	0.494039
4	1.486227	-0.480726	-0.127278
5	NaN	NaN	-0.047668
6	NaN	NaN	-1.504804
7	-0.530518	-0.881817	-2.687352
8	0.825376	1.042468	-0.311527
9	0.097617	1.373572	-0.682435

```
df.dropna(axis=1)
```

	2
0	-0.335410
1	0.685743
2	0.565144
3	0.494039
4	-0.127278
5	-0.047668
6	-1.504804
7	-2.687352
8	-0.311527
9	-0.682435

Perform linear interpolation




```
df
```

	0	1	2
0	-0.260156	-1.666998	-0.492616
1	NaN	NaN	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	NaN	NaN	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

```
df.interpolate()
```

	0	1	2
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

Summary

- There are many other ways to transform missing data:
  - Using 'polynomial' interpolation
  - Using Regular Expressions for replacement
- More : http://pandas.pydata.org/pandas-docs/version/0.15.2/missing_data.html#numeric-replacement

Data Visualisation

DataFrame

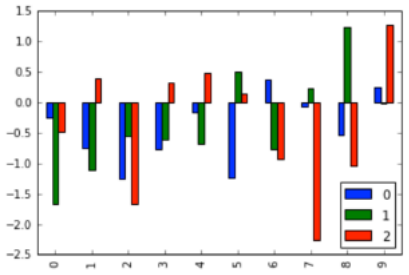
df

	0	1	2
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

df.plot.bar()

df

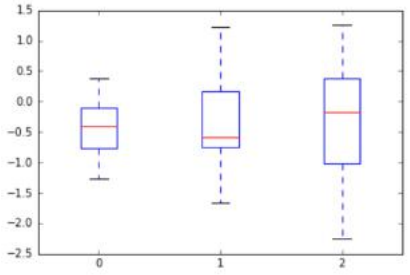
	0	1	2
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333



df.plot.box()

df

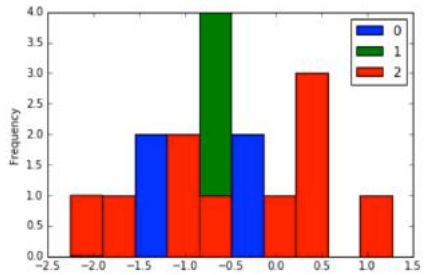
	0	1	2
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333



df.plot.hist()

df

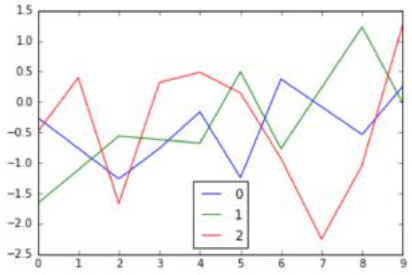
	0	1	2
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333



df.plot()

df

	0	1	2
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333



Summary

Explore here: http://pandas.pydata.org/pandas-docs/stable/api.html#api-dataframe-plotting

`DataFrame.plot`([x, y, kind, ax, ...])	DataFrame plotting accessor and method
`DataFrame.plot.area`([x, y])	Area plot
`DataFrame.plot.bar`([x, y])	Vertical bar plot
`DataFrame.plot.barh`([x, y])	Horizontal bar plot
`DataFrame.plot.box`([by])	Boxplot
`DataFrame.plot.density`(**kwds)	Kernel Density Estimate plot
`DataFrame.plot.hexbin`(x, y[, C, ...])	Hexbin plot
`DataFrame.plot.hist`([by, bins])	Histogram
`DataFrame.plot.kde`(**kwds)	Kernel Density Estimate plot
`DataFrame.plot.line`([x, y])	Line plot
`DataFrame.plot.pie`([y])	Pie chart
`DataFrame.plot.scatter`(x, y[, s, c])	Scatter plot
`DataFrame.boxplot`([column, by, ax, ...])	Make a box plot from DataFrame col columns or
`DataFrame.hist`(data[, column, by, grid, ...])	Draw histogram of the DataFrame's

Frequent Data Operations

Slice Out Columns

df

	sensor1	sensor2	sensor3
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

```
df['sensor1']
```

```
0   -0.260156
1   -0.762055
2   -1.263953
3   -0.765183
4   -0.165719
5   -1.243191
6    0.373786
7   -0.081455
8   -0.536697
9    0.254220
Name: sensor1, dtype: float64
```

Filter Out Rows

```
df
```

	sensor1	sensor2	sensor3
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

```
#Select rows where sensor2 is positive
df[df['sensor2'] > 0]
```

	sensor1	sensor2	sensor3
5	-1.243191	0.494006	0.145171
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728

Insert New Column

```
df
```

	sensor1	sensor2	sensor3
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

```
df['sensor4']=df['sensor3']**2
```

```
df
```

	sensor1	sensor2	sensor3	sensor4
0	-0.260156	-1.666998	-0.492616	0.242671
1	-0.762055	-1.114774	0.396817	0.157464
2	-1.263953	-0.562550	-1.670459	2.790434
3	-0.765183	-0.619429	0.316981	0.100477
4	-0.165719	-0.678431	0.485722	0.235925
5	-1.243191	0.494006	0.145171	0.021075
6	0.373786	-0.769120	-0.929956	0.864819
7	-0.081455	0.229613	-2.251816	5.070676
8	-0.536697	1.228345	-1.040728	1.083115
9	0.254220	-0.021794	1.268333	1.608669

Add a New Row

```
df
```

	sensor1	sensor2	sensor3
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

```
df.loc[10] = [11,22,33,44]
```

```
df
```

	sensor1	sensor2	sensor3	sensor4
0	-0.260156	-1.666998	-0.492616	0.242671
1	-0.762055	-1.114774	0.396817	0.157464
2	-1.263953	-0.562550	-1.670459	2.790434
3	-0.765183	-0.619429	0.316981	0.100477
4	-0.165719	-0.678431	0.485722	0.235925
5	-1.243191	0.494006	0.145171	0.021075
6	0.373786	-0.769120	-0.929956	0.864819
7	-0.081455	0.229613	-2.251816	5.070676
8	-0.536697	1.228345	-1.040728	1.083115
9	0.254220	-0.021794	1.268333	1.608669
10	11.000000	22.000000	33.000000	44.000000

Delete a Row

```
df
```

	sensor1	sensor2	sensor3
0	-0.260156	-1.666998	-0.492616
1	-0.762055	-1.114774	0.396817
2	-1.263953	-0.562550	-1.670459
3	-0.765183	-0.619429	0.316981
4	-0.165719	-0.678431	0.485722
5	-1.243191	0.494006	0.145171
6	0.373786	-0.769120	-0.929956
7	-0.081455	0.229613	-2.251816
8	-0.536697	1.228345	-1.040728
9	0.254220	-0.021794	1.268333

```
df.drop(df.index[[5]])
```

	sensor1	sensor2	sensor3	sensor4
0	-0.260156	-1.666998	-0.492616	0.242671
1	-0.762055	-1.114774	0.396817	0.157464
2	-1.263953	-0.562550	-1.670459	2.790434
3	-0.765183	-0.619429	0.316981	0.100477
4	-0.165719	-0.678431	0.485722	0.235925
6	0.373786	-0.769120	-0.929956	0.864819
7	-0.081455	0.229613	-2.251816	5.070676
8	-0.536697	1.228345	-1.040728	1.083115
9	0.254220	-0.021794	1.268333	1.608669

Delete a Column

```
df
```

	sensor1	sensor2	sensor3	sensor4
0	-0.260156	-1.666998	-0.492616	0.242671
1	-0.762055	-1.114774	0.396817	0.157464
2	-1.263953	-0.562550	-1.670459	2.790434
3	-0.765183	-0.619429	0.316981	0.100477
4	-0.165719	-0.678431	0.485722	0.235925
5	-1.243191	0.494006	0.145171	0.021075
6	0.373786	-0.769120	-0.929956	0.864819
7	-0.081455	0.229613	-2.251816	5.070676
8	-0.536697	1.228345	-1.040728	1.083115
9	0.254220	-0.021794	1.268333	1.608669

```
del df['sensor1']
```

```
df
```

	sensor2	sensor3	sensor4
0	-1.666998	-0.492616	0.242671
1	-1.114774	0.396817	0.157464
2	-0.562550	-1.670459	2.790434
3	-0.619429	0.316981	0.100477
4	-0.678431	0.485722	0.235925
5	0.494006	0.145171	0.021075
6	-0.769120	-0.929956	0.864819
7	0.229613	-2.251816	5.070676
8	1.228345	-1.040728	1.083115
9	-0.021794	1.268333	1.608669

Group By and Aggregate

```
df
```

	student_id	physics	chemistry	biology
0	2	198	92	108
1	2	111	134	122
2	2	37	174	25
3	4	121	128	63
4	4	191	97	178
5	4	102	157	182
6	12	70	76	181
7	12	101	62	128
8	12	26	51	56
9	100	148	78	159

```
df.groupby('student_id').mean()
```

	physics	chemistry	biology
student_id			
2	115.333333	133.333333	85.000000
4	138.000000	127.333333	141.000000
12	65.666667	63.000000	121.666667
100	148.000000	78.000000	159.000000

Summary

We saw a subset of transformation, more to explore here :

http://pandas.pydata.org/pandas-docs/stable/api.html

Merging Data Frames

Example Dataframes

left

	_key1	_key2	city	user_name
0	K0	z0	city_0	user_0
1	K1	z1	city_1	user_1
2	K2	z2	city_2	user_2
3	K3	z3	city_3	user_3

left

right

	_key1	_key2	hire_date	profession
0	K0	z0	h_0	p_0
1	K1	z1	h_1	p_1
2	K2	z2	h_2	p_2
3	K3	z3	h_3	p_3

right

pandas.concat() : Stack Dataframes

```
pd.concat([left, left])
```

	_key1	_key2	city	user_name
0	K0	z0	city_0	user_0
1	K1	z1	city_1	user_1
2	K2	z2	city_2	user_2
3	K3	z3	city_3	user_3
0	K0	z0	city_0	user_0
1	K1	z1	city_1	user_1
2	K2	z2	city_2	user_2
3	K3	z3	city_3	user_3

pandas.concat() : Stack Dataframes

```
pd.concat([left, right])
```

	_key1	_key2	city	hire_date	profession	user_name
0	K0	z0	city_0	NaN	NaN	user_0
1	K1	z1	city_1	NaN	NaN	user_1
2	K2	z2	city_2	NaN	NaN	user_2
3	K3	z3	city_3	NaN	NaN	user_3
0	K0	z0	NaN	h_0	p_0	NaN
1	K1	z1	NaN	h_1	p_1	NaN
2	K2	z2	NaN	h_2	p_2	NaN
3	K3	z3	NaN	h_3	p_3	NaN

pandas.concat() : Stack Dataframes

```
pd.concat([left, right])
```

	_key1	_key2	city	hire_date	profession	user_name
0	K0	z0	city_0	NaN	NaN	user_0
1	K1	z1	city_1	NaN	NaN	user_1
2	K2	z2	city_2	NaN	NaN	user_2
3	K3	z3	city_3	NaN	NaN	user_3
0	K0	z0	NaN	h_0	p_0	NaN
1	K1	z1	NaN	h_1	p_1	NaN
2	K2	z2	NaN	h_2	p_2	NaN
3	K3	z3	NaN	h_3	p_3	NaN

Inner Join using pandas.concat()

```
pd.concat([left, right], axis=1, join='inner')
```

	_key1	_key2	city	user_name	_key1	_key2	hire_date	profession
0	K0	z0	city_0	user_0	K0	z0	h_0	p_0
1	K1	z1	city_1	user_1	K1	z1	h_1	p_1
2	K2	z2	city_2	user_2	K2	z2	h_2	p_2
3	K3	z3	city_3	user_3	K3	z3	h_3	p_3

Inner Join using pandas.concat()

```
pd.concat([left, right], axis=1, join='inner')
```

	_key1	_key2	city	user_name	_key1	_key2	hire_date	profession
0	K0	z0	city_0	user_0	K0	z0	h_0	p_0
1	K1	z1	city_1	user_1	K1	z1	h_1	p_1
2	K2	z2	city_2	user_2	K2	z2	h_2	p_2
3	K3	z3	city_3	user_3	K3	z3	h_3	p_3

Stack DataFrames using append()

```
left.append(right)
```

	_key1	_key2	city	hire_date	profession	user_name
0	K0	z0	city_0	NaN	NaN	user_0
1	K1	z1	city_1	NaN	NaN	user_1
2	K2	z2	city_2	NaN	NaN	user_2
3	K3	z3	city_3	NaN	NaN	user_3
0	K0	z0	NaN	h_0	p_0	NaN
1	K1	z1	NaN	h_1	p_1	NaN
2	K2	z2	NaN	h_2	p_2	NaN
3	K3	z3	NaN	h_3	p_3	NaN

Inner Join using merge()

```
pd.merge(left, right, how='inner')
```

	_key1	_key2	city	user_name	hire_date	profession
0	K0	z0	city_0	user_0	h_0	p_0
1	K1	z1	city_1	user_1	h_1	p_1
2	K2	z2	city_2	user_2	h_2	p_2
3	K3	z3	city_3	user_3	h_3	p_3

Summary

More adventure:

http://pandas.pydata.org/pandas-docs/stable/merging.html#database-style-dataframe-joining-merging

Frequent String Operations

df

	_key1	_key2	city	user_name	hire_date	profession
0	K0	z0	city_0	user_0	h_0	p_0
1	K1	z1	city_1	user_1	h_1	p_1
2	K2	z2	city_2	user_2	h_2	p_2
3	K3	z3	city_3	user_3	h_3	p_3

str.split()

```
df['city'].str.split('_')
0    [city, 0]
1    [city, 1]
2    [city, 2]
3    [city, 3]
dtype: object
```

df

	_key1	_key2	city	user_name	hire_date	profession
0	K0	z0	city_0	user_0	h_0	p_0
1	K1	z1	city_1	user_1	h_1	p_1
2	K2	z2	city_2	user_2	h_2	p_2
3	K3	z3	city_3	user_3	h_3	p_3

str.contains()

```
df['city'].str.contains('2')
0    False
1    False
2     True
3    False
Name: city, dtype: bool
```

```
df
```

	_key1	_key2	city	user_name	hire_date	profession
0	K0	z0	city_0	user_0	h_0	p_0
1	K1	z1	city_1	user_1	h_1	p_1
2	K2	z2	city_2	user_2	h_2	p_2
3	K3	z3	city_3	user_3	h_3	p_3

str.replace()

```
df['city'].str.replace('_', '##')
```

```
0    city##0
1    city##1
2    city##2
3    city##3
Name: city, dtype: object
```

str.extract() – Returns **first** match found

```
df
```

	_key1	_key2	city	user_name
0	K0	z0	city 0	user_0
1	K1	z1	city 1	user_1
2	K2	z2	city 2	user_2
3	K3	z3	city 3	user_3

```
# Extract words in the strings
df['city'].str.extract('([a-z]\w{0,})')
```

```
0    city
1    city
2    city
3    city
Name: city, dtype: object
```

```
# Extract single digit in the strings
df['city'].str.extract('(\d)')
```

```
0    0
1    1
2    2
3    3
Name: city, dtype: object
```

Summary

Explore more :

http://pandas.pydata.org/pandas-docs/stable/text.html#text-string-methods

Parsing Timestamps

Unix time / POSIX time / epoch time

- Number of seconds elapsed since
  - 00:00:00
  - Coordinated Universal Time (UTC),
  - Thursday, 1 January 1970
- Prominent in UNIX like systems
- Parsing Timestamp: We have to read POSIX time and understand what the exact time stamp was

Data Types for Timestamps

- **Generic data type:** `datetime64[ns]`
- **Convert** `int64` timestamp to `<M8[ns]` or `>M8[ns]` on your machine

```
tags.dtypes
```

```
userId          int64
movieId         int64
tag            object
timestamp       int64
dtype: object
```

```
dtype('<M8[ns]')
```

Convert Timestamp to Python Format

`to_datetime()`

```
tags['parsed_time'] = pd.to_datetime(tags['timestamp'], unit='s')
```

```
tags.head(2)
```

	userId	movieId	tag	timestamp	parsed_time
0	18	4141	Mark Waters	1240597180	2009-04-24 18:19:40
1	65	208	dark hero	1368150078	2013-05-10 01:41:18

Select Rows Based on Timestamps

```
greater_than_t = tags['parsed_time'] > '2015-02-01'
```

```
selected_rows = tags[greater_than_t]
```

Sort Tables in Chronological Order

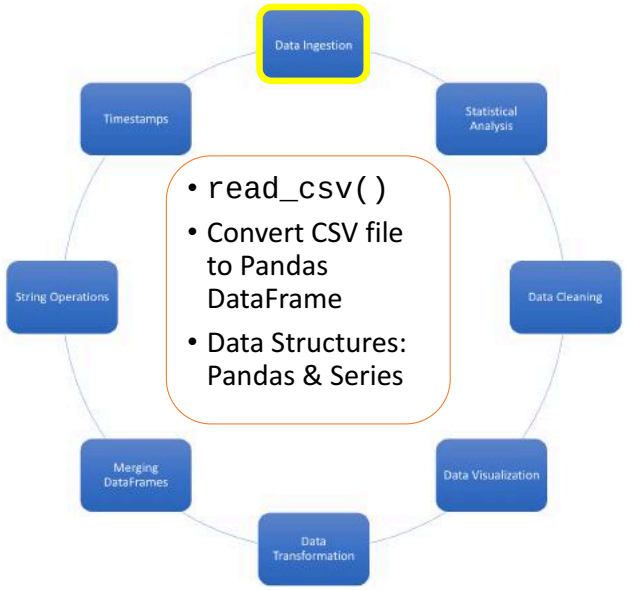
```
tags.sort_values(by='parsed_time', ascending=True)[:10]
```

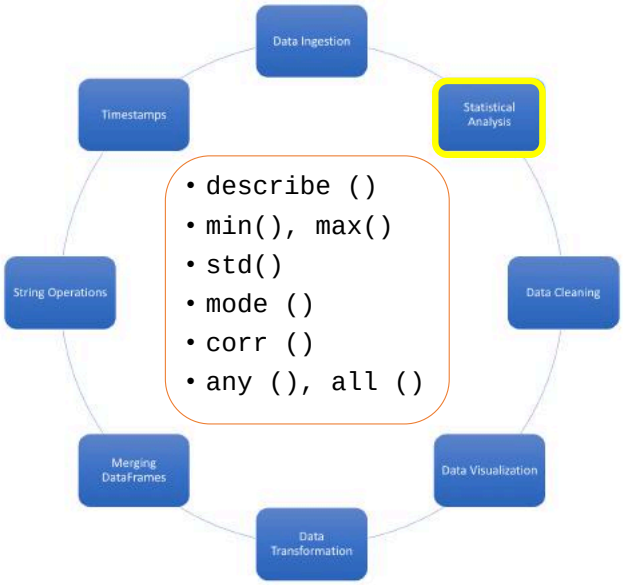
	userId	movieId	tag	timestamp	parsed_time
333932	100371	2788	monty python	1135429210	2005-12-24 13:00:10
333927	100371	1732	coen brothers	1135429236	2005-12-24 13:00:36
333924	100371	1206	stanley kubrick	1135429248	2005-12-24 13:00:48
333923	100371	1193	jack nicholson	1135429371	2005-12-24 13:02:51
333939	100371	5004	peter sellers	1135429399	2005-12-24 13:03:19
333922	100371	47	morgan freema	1135429412	2005-12-24 13:03:32
333921	100371	47	brad pitt	1135429412	2005-12-24 13:03:32
333936	100371	4011	brad pitt	1135429431	2005-12-24 13:03:51
333937	100371	4011	guy ritchie	1135429431	2005-12-24 13:03:51
333920	100371	32	bruce willis	1135429442	2005-12-24 13:04:02

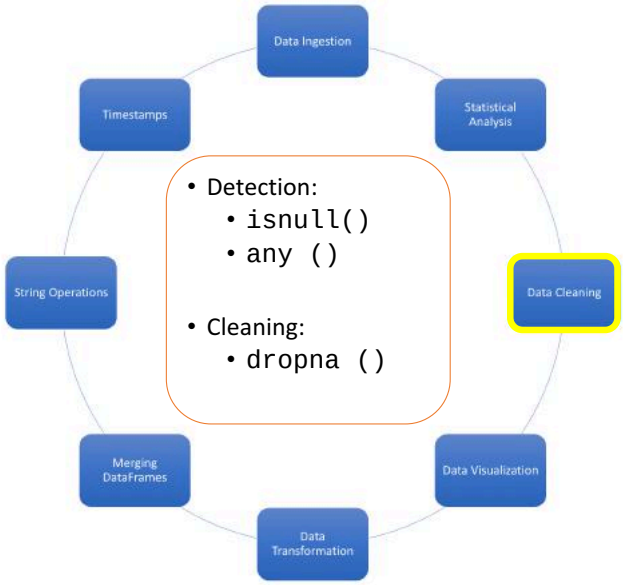
Summary

- POSIX / Unix time can be hard to read for users
- Converting to Python datetime format gives practical ways to:
  - Select data based on human readable time stamps
  - Create conditions using understandable time stamps

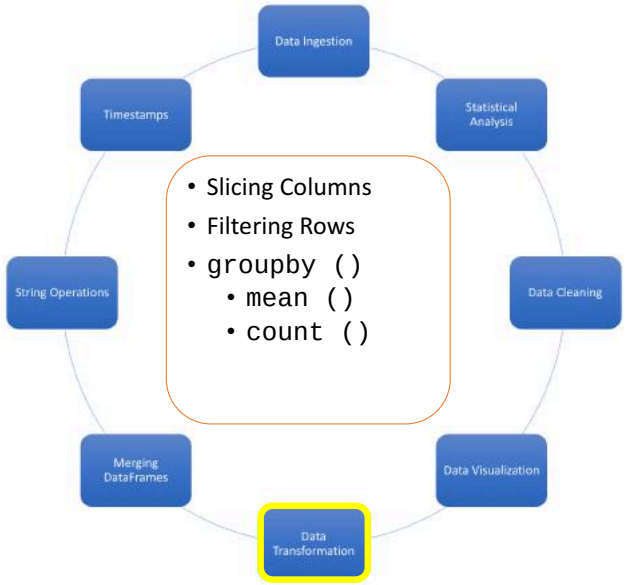
Case Study: Movie Rating Notebook

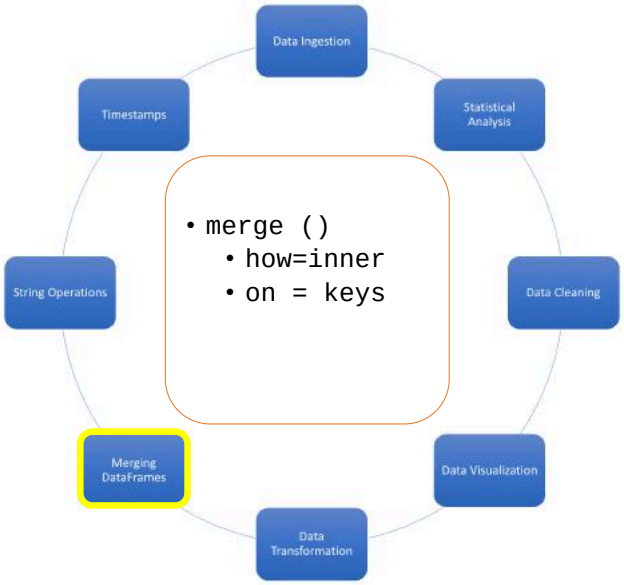


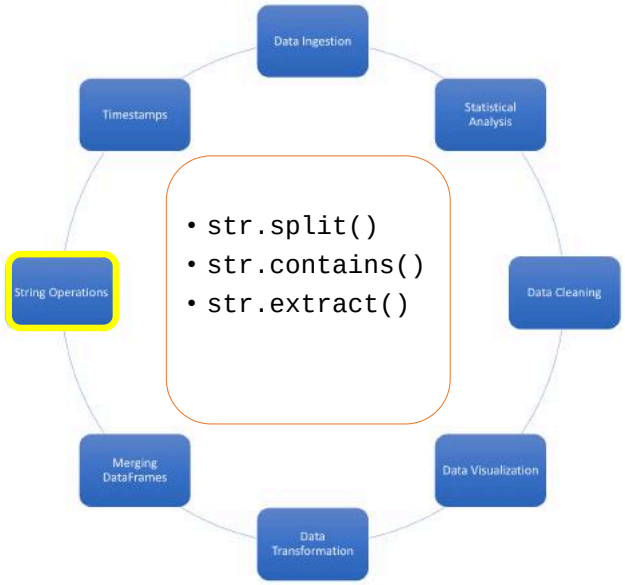

Data Ingestion
Statistical Analysis
Data Cleaning
Data Visualization
Data Transformation
Merging DataFrames
String Operations
Timestamps
• describe ()
• min(), max()
• std()
• mode ()
• corr ()
• any (), all ()

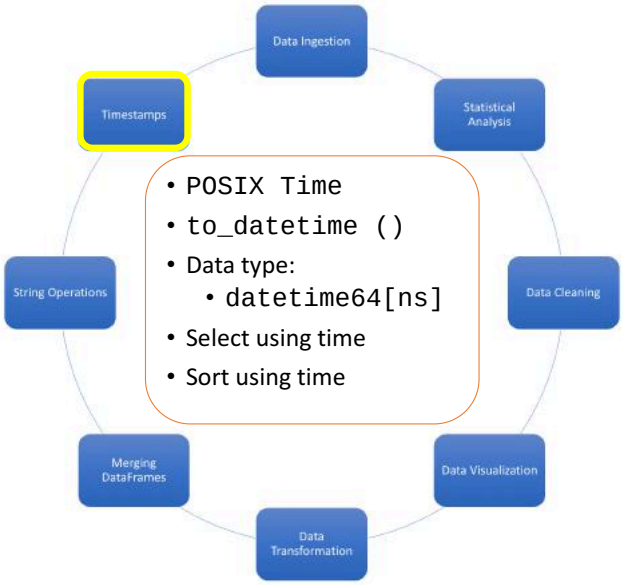

Data Ingestion
Statistical Analysis
Data Cleaning
Data Visualization
Data Transformation
Merging DataFrames
String Operations
Timestamps
• Detection:
• isnull()
• any ()
• Cleaning:
• dropna ()


Data Ingestion
Statistical Analysis
Data Cleaning
Data Visualization
Data Transformation
Merging DataFrames
String Operations
Timestamps
• Inline plotting
• Histograms
• Boxplots
• Changing limits on Y-axis


Data Ingestion
Statistical Analysis
Data Cleaning
Data Visualization
Data Transformation
Merging DataFrames
String Operations
Timestamps
• Slicing Columns
• Filtering Rows
• groupby ()
• mean ()
• count ()







Summary

- A typical data ingestion and transformation cycle
- Movies notebook as a representative example

References

Aforementioned contents are adapted from the course:

- **"Python for Data Science" taught on edX by Dr İlkay Altıntaş and Dr Leo Porter from the University of California at San Diego.**