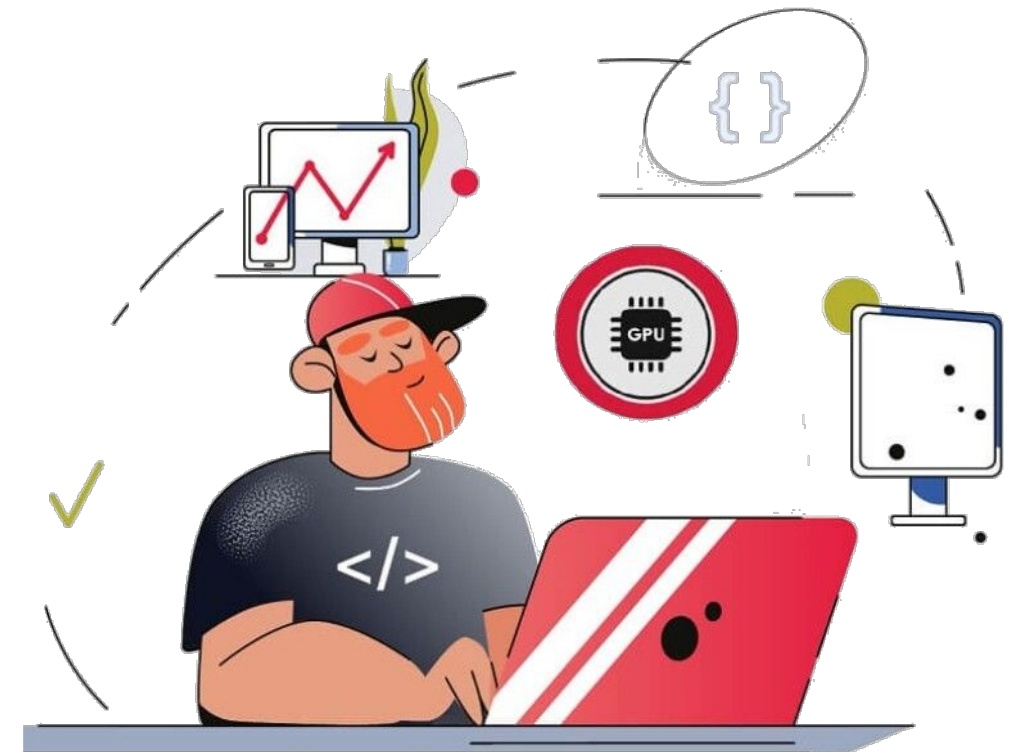


Programming Lab

Autumn Semester

Course code: PC503



Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar



Lecture 21

Python: Pandas

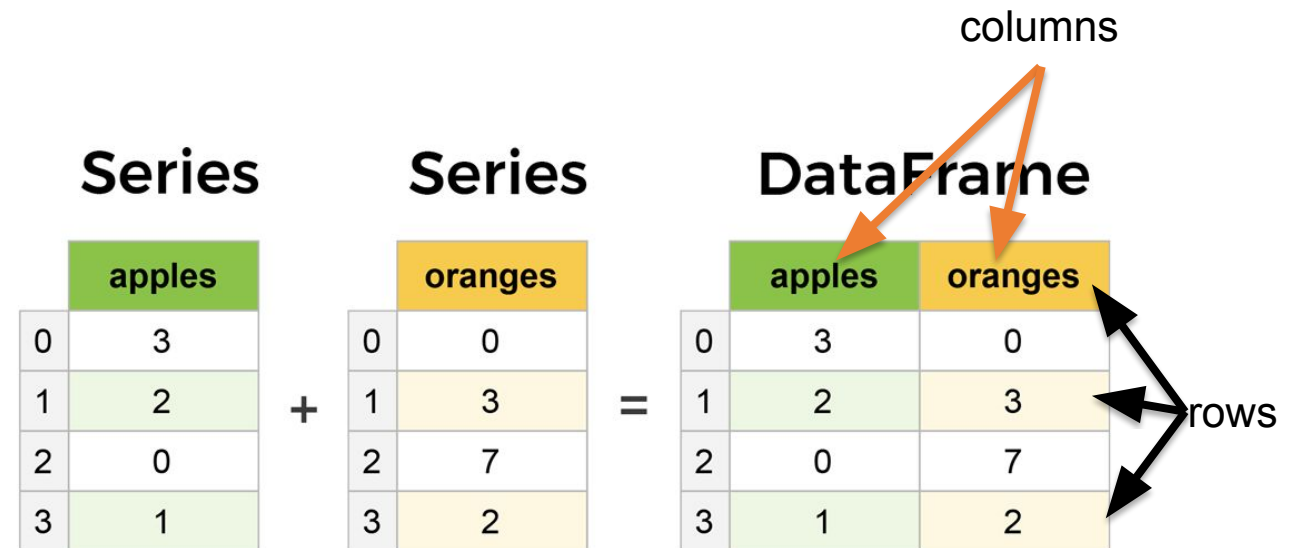
Introduction to DataFrame

Core components of pandas: Series & DataFrames

- The primary two components of pandas are the Series and DataFrame.
 - **Series** is essentially a **column**, and
 - **DataFrame** is a multi-dimensional table made up of a **collection of Series**.
- **DataFrames** and **Series** are quite similar in that many operations that you can do with one you can do with the other, such as filling in null values and calculating the mean.
 - A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

- **Features of DataFrame**

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (*rows* and *columns*)
- Can Perform Arithmetic operations on rows and columns



Types of Data Structure in Pandas

Data Structure	Dimensions	Description
Series	1	1D labeled <u>homogeneous</u> array with immutable size
Data Frames	2	General 2D labeled, size mutable tabular structure with potentially <u>heterogeneously</u> typed columns.
Panel	3	General 3D labeled, size mutable array.

- **Series & DataFrame**

- **Series** is a one-dimensional array (1D Array) like structure with homogeneous data.
- **DataFrame** is a two-dimensional array (2D Array) with heterogeneous data.

- **Panel**

- Panel is a three-dimensional data structure (3D Array) with heterogeneous data.
- It is hard to represent the panel in graphical representation.
- But a panel can be illustrated as a container of DataFrame

pandas.DataFrame

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

- **data:** data takes various forms like *ndarray*, *series*, *map*, *lists*, *dict*, constants and also another *DataFrame*.
- **index:** For the row labels, that are to be used for the resulting frame, Optional, Default is *np.arange(n)* if no index is passed.
- **columns:** For column labels, the optional default syntax is - *np.arange(n)*. This is only true if no index is passed.
- **dtype:** Data type of each column.
- **copy:** This command (or whatever it is) is used for copying of data, if the default is False.

• Create DataFrame

- A pandas DataFrame can be created using various inputs like –
 - Lists
 - dict
 - Series
 - Numpy ndarrays
 - Another DataFrame

Creating a DataFrame from scratch

Creating a DataFrame from scratch

- There are many ways to create a DataFrame from scratch, but a great option is to just use a simple dict. But first you must import pandas.

```
import pandas as pd
```

- Let's say we have a fruit stand that sells apples and oranges. We want to have a column for each fruit and a row for each customer purchase. To organize this as a dictionary for pandas we could do something like:

```
data = { 'apples':[3, 2, 0, 1] , 'oranges':[0, 3, 7, 2] }
```

- And then pass it to the pandas DataFrame constructor:

```
df = pd.DataFrame(data)
```



	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

How did that work?

- Each (key, value) item in data corresponds to a column in the resulting DataFrame.
- The Index of this DataFrame was given to us on creation as the numbers 0–3, but we could also create our own when we initialize the DataFrame.
- E.g. if you want to have customer names as the index:

```
df = pd.DataFrame(data, index=['Ahmad', 'Ali', 'Rashed', 'Hamza'])
```

	apples	oranges
Ahmad	3	0
Ali	2	3
Rashed	0	7
Hamza	1	2

- So now we could locate a customer's order by using their names:

```
df.loc['Ali']
```

```
apples    2
oranges    3
Name: Ali, dtype: int64
```


pandas.DataFrame.from_dict

```
pandas.DataFrame.from_dict(data, orient='columns', dtype=None, columns=None)
```

- **data** : dict
 - Of the form {**field:array-like**} or {**field:dict**}.
- **orient** : { **'columns'**, **'index'** }, default **'columns'**
 - The “orientation” of the data.
 - If the keys of the passed dict should be the columns of the resulting DataFrame, pass **'columns'** (default).
 - Otherwise if the keys should be rows, pass **'index'**.
- **dtype** : dtype, default **None**
 - Data type to force, otherwise infer.
- **columns** : list, default **None**
 - Column labels to use when **orient='index'**. Raises a **ValueError** if used with **orient='columns'**.

https://pandas.pydata.org/pandas-docs/version/0.23/generated/pandas.DataFrame.from_dict.html

pandas' **orient** keyword

```
data = {'col_1':[3, 2, 1, 0], 'col_2':['a','b','c','d']}  
pd.DataFrame.from_dict(data)
```



	col_1	col_2
0	3	a
1	2	b
2	1	c
3	0	d

```
data = {'row_1':[3, 2, 1, 0], 'row_2':['a','b','c','d']}  
pd.DataFrame.from_dict(data,  
                        orient='index')
```



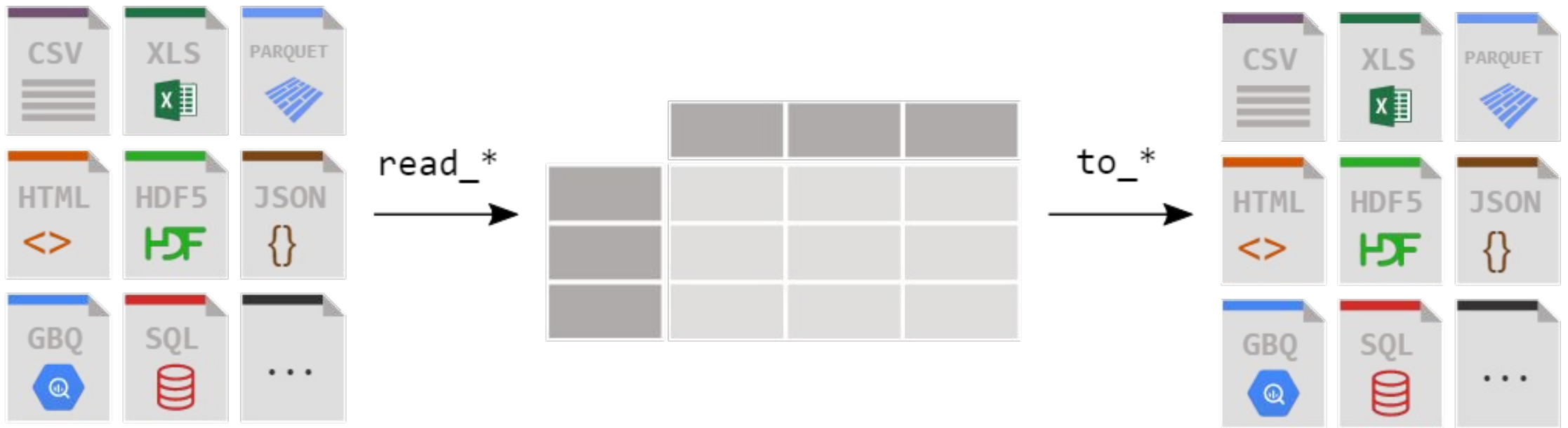
	0	1	2	3
row_1	3	2	1	0
row_2	a	b	c	d

```
data = {'row_1':[3, 2, 1, 0], 'row_2':['a','b','c','d']}  
pd.DataFrame.from_dict(data,  
                        orient = 'index',  
                        columns = ['A','B','C','D'])
```

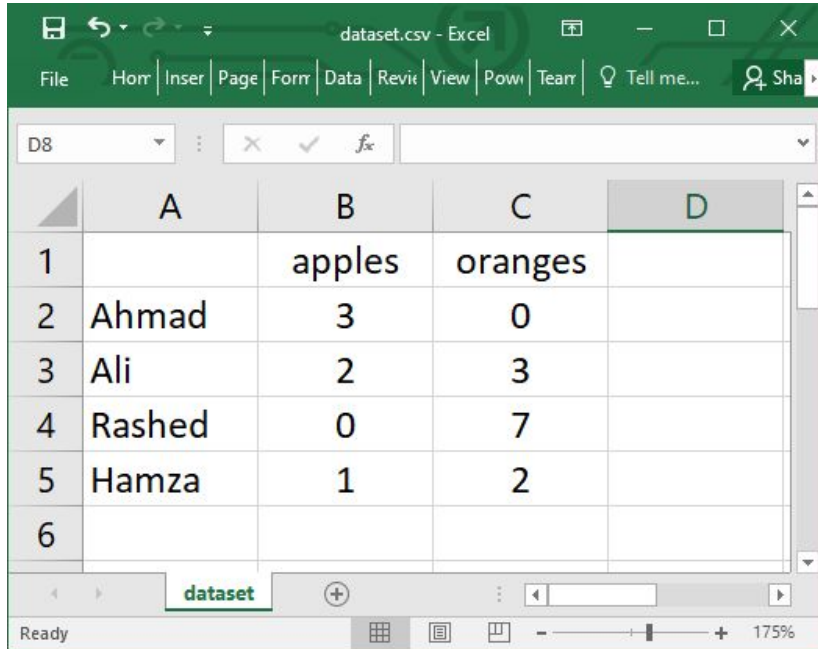


	A	B	C	D
row_1	3	2	1	0
row_2	a	b	c	d

Loading a DataFrame from files



Reading data from a CSV file



	A	B	C	D
1		apples	oranges	
2	Ahmad	3	0	
3	Ali	2	3	
4	Rashed	0	7	
5	Hamza	1	2	
6				



```
File Edit Format Run Options Window Help
1 import pandas as pd
2
3 df = pd.read_csv('dataset.csv')
4 print(df)
5
6 # OR
7
8 df = pd.read_csv('dataset.csv', index_col=0)
9 print(df)
10
```

Ln: 6 Col: 0

Reading data from CSVs

- With CSV files, all you need is a single line to load in the data:

```
df = pd.read_csv('dataset.csv')
```

	Unnamed: 0	apples	oranges
0	Ahmad	3	0
1	Ali	2	3
2	Rashed	0	7
3	Hamza	1	2

- CSVs don't have indexes like our DataFrames, so all we need to do is just designate the **index_col** when reading:

```
df = pd.read_csv('dataset.csv', index_col=0)
```

	apples	oranges
Ahmad	3	0
Ali	2	3
Rashed	0	7
Hamza	1	2

- Note: here we're setting the index to be column zero.*

Reading data from JSON

- If you have a JSON file — which is essentially a stored Python dict — pandas can read this just as easily:

```
df = pd.read_json('dataset.json')
```

- Notice this time our index came with us correctly since using JSON allowed indexes to work through nesting.
- Pandas will try to figure out how to create a DataFrame by analyzing structure of your JSON, and sometimes it doesn't get it right.
- Often you'll need to set the `orient` keyword argument depending on the structure

Example #1: Reading data from JSON

```
{  
  "apples" : { "Ahmad": 3, "Ali": 2, "Rashed": 0, "Hamza": 1 },  
  "oranges" : { "Ahmad": 0, "Ali": 3, "Rashed": 7, "Hamza": 2 }  
}
```



```
File Edit Format Run Options Window Help  
1 import pandas as pd  
2  
3 df = pd.read_json('dataset.json')  
4 print(df)  
Ln: 1 Col: 0
```



	apples	oranges
Ahmad	3	0
Ali	2	3
Rashed	0	7
Hamza	1	2

Example #2: Reading data from JSON

```
{  
  "Ahmad" : {"apples":3, "oranges":0},  
  "Ali" : {"apples":2, "oranges":3},  
  "Rashed" : {"apples":0, "oranges":7},  
  "Hamza" : {"apples":1, "oranges":2}  
}
```



```
File Edit Format Run Options Window Help  
1 import pandas as pd  
2  
3 df = pd.read_json('dataset.json')  
4 print(df)  
Ln: 1 Col: 0
```



	Ahmad	Ali	Rashed	Hamza
apples	3	2	0	1
oranges	0	3	7	2

Example #3: Reading data from JSON

```
{
  "Ahmad" : {"apples":3, "oranges":0},
  "Ali" : {"apples":2, "oranges":3},
  "Rashed" : {"apples":0, "oranges":7},
  "Hamza" : {"apples":1, "oranges":2}
}
```

```
File Edit Format Run Options Window Help
1 import pandas as pd
2
3 df = pd.read_json('dataset.json',
4                   orient='column')
5 print(df)
```

Ln: 6 Col: 0

	Ahmad	Ali	Rashed	Hamza
apples	3	2	0	1
oranges	0	3	7	2

```
File Edit Format Run Options Window Help
1 import pandas as pd
2
3 df = pd.read_json('dataset.json',
4                   orient='index')
5 print(df)
```

Ln: 6 Col: 0

	apples	oranges
Ahmad	3	0
Ali	2	3
Rashed	0	7
Hamza	1	2

Converting back to a CSV or JSON

- So after extensive work on cleaning your data, you're now ready to save it as a file of your choice. Similar to the ways we read in data, pandas provides intuitive commands to save it:

```
df.to_csv('new_dataset.csv')  
df.to_json('new_dataset.json')
```

- When we save JSON and CSV files, all we have to input into those functions is our desired filename with the appropriate file extension.

Most important DataFrame operations

- DataFrames possess hundreds of methods and other operations that are crucial to any analysis.
- As a beginner, you should know the operations that:
 - that perform simple transformations of your data and those
 - that provide fundamental statistical analysis on your data.

Loading dataset

- We're loading this dataset from a CSV and designating the movie titles to be our index.

```
movies_df = pd.read_csv("movies.csv", index_col="title")
```

Viewing your data

- The first thing to do when opening a new dataset is print out a few rows to keep as a visual reference. We accomplish this with `.head()`:

```
movies_df.head()
```

- `.head()` outputs the first five rows of your DataFrame by default, but we could also pass a number as well: `movies_df.head(10)` would output the top ten rows, for example.
- To see the last five rows use `.tail()` that also accepts a number, and in this case we printing the bottom two rows.:

```
movies_df.tail(2)
```

Getting info about your data

- `.info()` should be one of the very first commands you run after loading your data
- `.info()` provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.

```
movies_df.info()
```

```
movies_df.shape
```

OUT:

```
<class 'pandas.core.frame.DataFrame'>  
Index: 1000 entries, Guardians of the Galaxy to Nine Lives  
Data columns (total 11 columns):  
Rank                1000 non-null int64  
Genre               1000 non-null object  
Description          1000 non-null object  
Director            1000 non-null object  
Actors              1000 non-null object  
Year               1000 non-null int64  
Runtime (Minutes)   1000 non-null int64  
Rating              1000 non-null float64  
Votes               1000 non-null int64  
Revenue (Millions)  872 non-null float64  
Metascore           936 non-null float64  
dtypes: float64(3), int64(4), object(4)  
memory usage: 93.8+ KB
```

OUT:

```
(1000, 11)
```

Handling duplicates

- This dataset does not have duplicate rows, but it is always important to verify you aren't aggregating duplicate rows.
- To demonstrate, let's simply just double up our movies DataFrame by appending it to itself:
- Using `append()` will return a copy without affecting the original DataFrame. We are capturing this copy in **temp** so we aren't working with the real data.
- Notice call `.shape` quickly proves our DataFrame rows have doubled.

```
temp_df = movies_df.append(movies_df)
temp_df.shape
```

OUT:

(2000, 11)

Now we can try dropping duplicates:

```
temp_df = temp_df.drop_duplicates()
temp_df.shape
```

OUT:

(1000, 11)

Handling duplicates

- Just like `append()`, the `drop_duplicates()` method will also return a copy of your DataFrame, but this time with duplicates removed. Calling `.shape` confirms we're back to the 1000 rows of our original dataset.
- It's a little verbose to keep assigning DataFrames to the same variable like in this example. For this reason, pandas has the `inplace` keyword argument on many of its methods. Using `inplace=True` will modify the DataFrame object in place:

```
temp_df.drop_duplicates(inplace=True)
```

- Another important argument for `drop_duplicates()` is `keep`, which has three possible options:
 - **first**: (default) Drop duplicates except for the first occurrence.
 - **last**: Drop duplicates except for the last occurrence.
 - **False**: Drop all duplicates.

Understanding your variables

- Using `.describe()` on an entire DataFrame we can get a summary of the distribution of continuous variables:

```
movies_df.describe()
```

OUT:

	rank	year	runtime	rating	
count	1000.000000	1000.000000	1000.000000	1000.000000	1.00
mean	500.500000	2012.783000	113.172000	6.723200	1.65
std	288.819436	3.205962	18.810908	0.945429	1.88
min	1.000000	2006.000000	66.000000	1.900000	6.10
25%	250.750000	2010.000000	100.000000	6.200000	3.6
50%	500.500000	2014.000000	111.000000	6.800000	1.10
75%	750.250000	2016.000000	123.000000	7.400000	2.3
max	1000.000000	2016.000000	191.000000	9.000000	1.75

- `.describe()` can also be used on a categorical variable to get the count of rows, unique count of categories, top category, and freq of top category:

```
movies_df['genre'].describe()
```

OUT:

```
count          1000
unique          207
top    Action,Adventure,Sci-Fi
freq           50
Name: genre, dtype: object
```

- This tells us that the genre column has 207 unique values, the top value is Action/Adventure/Sci-Fi, which shows up 50 times (freq).

More Examples

```
import pandas as pd
data = [1,2,3,10,20,30]
df = pd.DataFrame(data)
print(df)
```



0	1
1	2
2	3
3	10
4	20
5	30

```
import pandas as pd
data = {'Name' : ['AA', 'BB'], 'Age': [30,45]}
df = pd.DataFrame(data)
print(df)
```



	Name	Age
0	AA	30
1	BB	45

More Examples

```
import pandas as pd
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
print(df)
```



	a	b	c
0	1	2	NaN
1	5	10	20.0

```
import pandas as pd
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print(df)
```



	a	b	c
first	1	2	NaN
second	5	10	20.0

More Examples

E.g. This shows how to create a DataFrame with a list of dictionaries, row indices, and column indices.

```
import pandas as pd
data = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])

#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])

print(df1)
print('.....')
print(df2)
```

	a	b
first	1	2
second	5	10
.....		
	a	b1
first	1	NaN
second	5	NaN

More Examples:

Create a DataFrame from Dict of Series

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3] , index=['a', 'b', 'c']),
     'two' : pd.Series([1,2, 3, 4], index=['a', 'b', 'c', 'd'])
}
df = pd.DataFrame(d)
print(df)
```

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	NaN	4

More Examples: Column Addition

```
import pandas as pd
d = {'one':pd.Series([1,2,3], index=['a','b','c']),
     'two':pd.Series([1,2,3,4], index=['a','b','c','d'])
}
df = pd.DataFrame(d)
# Adding a new column to an existing DataFrame object
# with column label by passing new series

print("Adding a new column by passing as Series:")
df['three'] = pd.Series([10,20,30],index=['a','b','c'])
print(df)

print("Adding a column using an existing columns in
DataFrame:")
df['four'] = df['one']+df['three']
print(df)
```

Adding a column using Series:

	one	two	three
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

Adding a column using columns:

	one	two	three	four
a	1.0	1	10.0	11.0
b	2.0	2	20.0	22.0
c	3.0	3	30.0	33.0
d	NaN	4	NaN	NaN

More Examples: Column Deletion

```
# Using the previous DataFrame, we will delete a column
# using del function
import pandas as pd
d = {'one'      : pd.Series([1, 2, 3],      index=['a', 'b', 'c']),
      'two'      : pd.Series([1, 2, 3, 4],   index=['a', 'b', 'c', 'd']),
      'three'    : pd.Series([10,20,30],     index=['a','b','c'])
    }
df = pd.DataFrame(d)
print ("Our dataframe is:")
print(df)

# using del function
print("Deleting the first column using DEL function:")
del df['one']
print(df)

# using pop function
print("Deleting another column using POP function:")
df.pop('two')
print(df)
```

Our dataframe is:

	one	two	three
a	1.0	1	10.0
b	2.0	2	20.0
c	3.0	3	30.0
d	NaN	4	NaN

Deleting the first column:

	two	three
a	1	10.0
b	2	20.0
c	3	30.0
d	4	NaN

Deleting another column:

a	10.0
b	20.0
c	30.0
d	NaN

More Examples: **Slicing** in DataFrames

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
}
df = pd.DataFrame(d)
print(df[2:4])
```

	one	two
c	3.0	3
d	NaN	4

More Examples: **Addition** of rows

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
}
df = pd.DataFrame(d)
print(df)

df2 = pd.DataFrame([[5,6], [7,8]], columns = ['a', 'b'])
df = df.append(df2 )
print(df)
```

	one	two		
a	1.0	1		
b	2.0	2		
c	3.0	3		
d	NaN	4		
	one	two	a	b
a	1.0	1.0	NaN	NaN
b	2.0	2.0	NaN	NaN
c	3.0	3.0	NaN	NaN
d	NaN	4.0	NaN	NaN
0	NaN	NaN	5.0	6.0
1	NaN	NaN	7.0	8.0

More Examples: Deletion of rows

```
import pandas as pd
d = {'one':pd.Series([1, 2, 3], index=['a', 'b', 'c']),
     'two':pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
}
df = pd.DataFrame(d)
print(df)

df2 = pd.DataFrame([[5,6], [7,8]], columns = ['a', 'b'])
df = df.append(df2 )
print(df)

df = df.drop(0)
print(df)
```

	one	two
a	1.0	1
b	2.0	2
c	3.0	3
d	NaN	4

	one	two	a	b
a	1.0	1.0	NaN	NaN
b	2.0	2.0	NaN	NaN
c	3.0	3.0	NaN	NaN
d	NaN	4.0	NaN	NaN
0	NaN	NaN	5.0	6.0
1	NaN	NaN	7.0	8.0

	one	two	a	b
a	1.0	1.0	NaN	NaN
b	2.0	2.0	NaN	NaN
c	3.0	3.0	NaN	NaN
d	NaN	4.0	NaN	NaN
1	NaN	NaN	7.0	8.0

More Examples: Reindexing

```
import pandas as pd
# Creating the first dataframe
df1 = pd.DataFrame({"A": [1, 5, 3, 4, 2],
                    "B": [3, 2, 4, 3, 4],
                    "C": [2, 2, 7, 3, 4],
                    "D": [4, 3, 6, 12, 7]},
                    index = ["A1", "A2", "A3", "A4", "A5"])

# Creating the second dataframe
df2 = pd.DataFrame({"A": [10, 11, 7, 8, 5],
                    "B": [21, 5, 32, 4, 6],
                    "C": [11, 21, 23, 7, 9],
                    "D": [1, 5, 3, 8, 6]},
                    index = ["A1", "A3", "A4", "A7", "A8"])

# Print the first dataframe
print(df1)
print(df2)
# find matching indexes
df1.reindex_like(df2)
```

- Pandas `dataframe.reindex_like()` function return an object with matching indices to myself.
- Any non-matching indexes are filled with NaN values.

Out[72]:

	A	B	C	D
A1	1.0	3.0	2.0	4.0
A3	3.0	4.0	7.0	6.0
A4	4.0	3.0	3.0	12.0
A7	NaN	NaN	NaN	NaN
A8	NaN	NaN	NaN	NaN

More Examples:

Concatenating Objects (Data Frames)

```
import pandas as pd
df1 = pd.DataFrame({'Name': ['A', 'B'], 'SSN': [10, 20], 'marks': [90, 95] })
df2 = pd.DataFrame({'Name': ['B', 'C'], 'SSN': [25, 30], 'marks': [80, 97] })
df3 = pd.concat([df1, df2])
df3
```

Handling categorical data

- There are many data that are repetitive for example gender , country , and codes are always repetitive .
- Categorical variables can take on only a limited
- The categorical data type is useful in the following cases –
- A string variable consisting of only a few different values. Converting such a string variable to a categorical variable will save some memory.
- The lexical order of a variable is not the same as the logical order (“one”, “two”, “three”).
 - By converting to a categorical and specifying an order on the categories, sorting and min/max will use the logical order instead of the lexical order.
- As a signal to other python libraries that this column should be treated as a categorical variable (e.g. to use suitable statistical methods or plot types).

Examples

```
import pandas as pd
cat = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
print(cat)
```

```
import pandas as pd
import numpy as np
cat = pd.Categorical(["a", "c", "c", np.nan], categories=["b", "a", "c"])
df = pd.DataFrame({"cat": cat, "s": ["a", "c", "c", np.nan]})
print(df.describe())
print(df["cat"].describe())
```

Reading data from a SQL database

- If you're working with data from a SQL database you need to first establish a connection using an appropriate Python library, then pass a query to pandas. Here we'll use SQLite to demonstrate.
- First, we need `pysqlite3` installed, so run this command in your terminal:
 - `pip install pysqlite3`
 - Or run this cell if you're in a notebook: `!pip install pysqlite3`
- `sqlite3` is used to create a connection to a database which we can then use to generate a DataFrame through a `SELECT` query.
 - So first we'll make a connection to a SQLite database file:
- In this SQLite database we have a table called `purchases`, and our index is in a column called `"index"`.
- By passing a `SELECT` query and our `con`, we can read from the `purchases` table:

```
import sqlite3
con = sqlite3.connect("database.db")
```

```
df = pd.read_sql_query("SELECT * FROM purchases", con)
```

Reading data from a SQL database

- In this SQLite database we have a table called purchases, and our index is in a column called "index".
- By passing a SELECT query and our con, we can read from the purchases table:

```
df = pd.read_sql_query("SELECT * FROM purchases", con)
```

OUT:

	index	apples	oranges
0	June	3	0
1	Robert	2	3
2	Lily	0	7
3	David	1	2

- Just like with CSVs, we could pass index_col='index', but we can also set an index after-the-fact:
 - In fact, we could use set_index() on any DataFrame using any column at any time. Indexing Series and DataFrames is a very common task, and the different ways of doing it is worth remembering.

```
df = df.set_index('index')
```

OUT:

	apples	oranges
index		
June	3	0
Robert	2	3
Lily	0	7
David	1	2

Part II

Reading data using pandas

```
In [ ]: #Read csv file
df = pd.read_csv("http://rcc.bu.edu/examples/python/data_analysis/Salaries.csv")
```

Note: The above command has many optional arguments to fine-tune the data import process.

There is a number of pandas commands to read other data formats:

```
pd.read_excel('myfile.xlsx', sheet_name='Sheet1', index_col=None, na_values=['NA'])
```

```
pd.read_stata('myfile.dta')
```

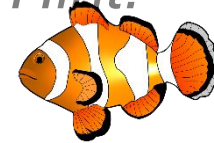
```
pd.read_sas('myfile.sas7bdat')
```

```
pd.read_hdf('myfile.h5', 'df')
```

Hands-on exercises

- ✓ Try to read the first 10, 20, 50 records;
- ✓ Can you guess how to view the last few records;

Hint:



Data Frame data types

Pandas Type	Native Python Type	Description
object	string	The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings).
int64	int	Numeric characters. 64 refers to the memory allocated to hold this character.
float64	float	Numeric characters with decimals. If a column contains numbers and NaNs(see below), pandas will default to float64, in case your missing value has a decimal.
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	Values meant to hold time data. Look into these for time series experiments.

Data Frame data types

```
In [4]: #Check a particular column type  
df['salary'].dtype
```

```
Out[4]: dtype('int64')
```

```
In [5]: #Check types for all the columns  
df.dtypes
```

```
Out[4]: rank          object  
discipline          object  
phd                 int64  
service             int64  
sex                 object  
salary              int64  
dtype: object
```

Data Frames attributes

Python objects have *attributes* and *methods*.

df.attribute	description
dtypes	list the types of the columns
columns	list the column names
axes	list the row labels and column names
ndim	number of dimensions
size	number of elements
shape	return a tuple representing the dimensionality
values	numpy representation of the data

Hands-on exercises

- ✓ Find how many records this data frame has;
- ✓ How many elements are there?
- ✓ What are the column names?
- ✓ What types of columns we have in this data frame?

Data Frames methods

Unlike attributes, python methods have *parenthesis*.

All attributes and methods can be listed with a *dir()* function: `dir(df)`

df.method()	description
head([n]), tail([n])	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample([n])	returns a random sample of the data frame
dropna()	drop all the records with missing values

Hands-on exercises

- ✓ Give the summary for the numeric columns in the dataset
- ✓ Calculate standard deviation for all numeric columns;
- ✓ What are the mean values of the first 50 records in the dataset? *Hint:*
use `head()` method to subset the first 50 records and then calculate the mean

Hands-on exercises

- ✓ Calculate the basic statistics for the *salary* column;
- ✓ Find how many values in the *salary* column (use *count* method);
- ✓ Calculate the average salary;

Data Frames *groupby* method

Using "group by" method we can:

- Split the data into groups based on some criteria
- Calculate statistics (or apply a function) to each group
- Similar to dplyr() function in R

```
In [ ]: #Group data using rank
df_rank = df.groupby( ['rank'] )
```

```
In [ ]: #Calculate mean value for each numeric column per each group
df_rank.mean()
```

	phd	service	salary
rank			
AssocProf	15.076923	11.307692	91786.230769
AsstProf	5.052632	2.210526	81362.789474
Prof	27.065217	21.413043	123624.804348

Data Frames *groupby* method

Once groupby object is create we can calculate various statistics for each

group:

```
In [ ]: #Calculate mean salary for each professor rank:  
df.groupby('rank')[['salary']].mean()
```

salary	
rank	
AssocProf	91786.230769
AsstProf	81362.789474
Prof	123624.804348

Note: If single brackets are used to specify the column (e.g. salary), then the output is Pandas Series object. When double brackets are used the output is a Data Frame

Data Frames *groupby* method

groupby performance notes:

- no grouping/splitting occurs until it's needed. Creating the *groupby* object only verifies that you have passed a valid mapping
- by default the group keys are sorted during the *groupby* operation. You may want to pass `sort=False` for potential speedup:

```
In [ ]: #Calculate mean salary for each professor rank:  
df.groupby(['rank'], sort=False)[['salary']].mean()
```

Data Frame: filtering

To subset the data we can apply Boolean indexing. This indexing is commonly known as a filter. For example if we want to subset the rows in which the salary value is greater than \$120K:

```
In [ ]: #Calculate mean salary for each professor rank:  
df_sub = df[ df['salary'] > 120000 ]
```

Any Boolean operator can be used to subset the data:

> greater; >= greater or equal;
< less; <= less or equal;
== equal; != not equal;

```
In [ ]: #Select only those rows that contain female professors:  
df_f = df[ df['sex'] == 'Female' ]
```

Data Frames: Slicing

There are a number of ways to subset the Data Frame:

- one or more columns
- one or more rows
- a subset of rows and columns

Rows and columns can be selected by their position or label

Data Frames: Slicing

When selecting one column, it is possible to use single set of brackets, but the resulting object will be a Series (not a DataFrame):

```
In [ ]: #Select column salary:  
df['salary']
```

When we need to select more than one column and/or make the output to be a DataFrame, we should use double brackets:

```
In [ ]: #Select column salary:  
df[['rank', 'salary']]
```


Data Frames: Selecting rows

If we need to select a range of rows, we can specify the range using ":"

```
In [ ]: #Select rows by their position:  
df[10:20]
```

Notice that the first row has a position 0, and the last value in the range is omitted:

So for 0:10 range the first 10 rows are returned with the positions starting with 0 and ending with 9

Data Frames: method loc

If we need to select a range of rows, using their labels we can use method loc:

```
In [ ]: #Select rows by their labels:  
df_sub.loc[10:20,['rank','sex','salary']]
```

```
Out[ ]:
```

	rank	sex	salary
10	Prof	Male	128250
11	Prof	Male	134778
13	Prof	Male	162200
14	Prof	Male	153750
15	Prof	Male	150480
19	Prof	Male	150500

Data Frames: method iloc

If we need to select a range of rows and/or columns, using their positions we can use method iloc:

```
In [ ]: #Select rows by their labels:  
df_sub.iloc[10:20,[0, 3, 4, 5]]
```

Out[]:

	rank	service	sex	salary
26	Prof	19	Male	148750
27	Prof	43	Male	155865
29	Prof	20	Male	123683
31	Prof	21	Male	155750
35	Prof	23	Male	126933
36	Prof	45	Male	146856
39	Prof	18	Female	129000
40	Prof	36	Female	137000
44	Prof	19	Female	151768
45	Prof	25	Female	140096

Data Frames: method iloc (summary)

```
df.iloc[0]    # First row of a data frame  
df.iloc[i]    #(i+1)th row  
df.iloc[-1]   # Last row
```

```
df.iloc[:, 0] # First column  
df.iloc[:, -1] # Last column
```

```
df.iloc[0:7]      #First 7 rows  
df.iloc[:, 0:2]    #First 2 columns  
df.iloc[1:3, 0:2]  #Second through third rows and first 2 columns  
df.iloc[[0,5], [1,3]] #1st and 6th rows and 2nd and 4th columns
```

Data Frames: Sorting

We can sort the data by a value in the column. By default the sorting will occur in ascending order and a new data frame is return.

```
In [ ]: # Create a new data frame from the original sorted by the column Salary  
df_sorted = df.sort_values( by ='service' )  
df_sorted.head()
```

```
Out[ ]:
```

	rank	discipline	phd	service	sex	salary
55	AsstProf	A	2	0	Female	72500
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000

Data Frames: Sorting

We can sort the data using 2 or more columns:

```
In [ ]: df_sorted = df.sort_values( by=['service', 'salary'], ascending = [True, False])
df_sorted.head(10)
```

Out[]:

	rank	discipline	phd	service	sex	salary
52	Prof	A	12	0	Female	105000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
55	AsstProf	A	2	0	Female	72500
57	AsstProf	A	3	1	Female	72500
28	AsstProf	B	7	2	Male	91300
42	AsstProf	B	4	2	Female	80225
68	AsstProf	A	4	2	Female	77500

Missing Values

Missing values are marked as NaN

```
In [ ]: # Read a dataset with missing values
        flights = pd.read_csv("http://rds.bu.edu/examples/python/data_analysis/flights.csv")
```

```
In [ ]: # Select the rows that have at least one missing value
        flights[flights.isnull().any(axis=1)].head()
```

```
Out[ ]:
```

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	hour	minute
330	2013	1	1	1807.0	29.0	2251.0	NaN	UA	N31412	1228	EWB	SAN	NaN	2425	18.0	7.0
403	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EHAA	791	LGA	DFW	NaN	1389	NaN	NaN
404	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EVAA	1925	LGA	MIA	NaN	1096	NaN	NaN
855	2013	1	2	2145.0	16.0	NaN	NaN	UA	N12221	1299	EWB	RSW	NaN	1068	21.0	45.0
858	2013	1	2	NaN	NaN	NaN	NaN	AA	NaN	133	JFK	LAX	NaN	2475	NaN	NaN

Missing Values

There are a number of methods to deal with missing values in the data frame:

df.method()	description
dropna()	Drop missing observations
dropna(how='all')	Drop observations where all cells is NA
dropna(axis=1, how='all')	Drop column if all the values are missing
dropna(thresh = 5)	Drop rows that contain less than 5 non-missing values
fillna(0)	Replace missing values with zeros
isnull()	returns True if the value is missing
notnull()	Returns True for non-missing values

Missing Values

- When summing the data, missing values will be treated as zero
- If all values are missing, the sum will be equal to NaN
- `cumsum()` and `cumprod()` methods ignore missing values but preserve them in the resulting arrays
- Missing values in `GroupBy` method are excluded (just like in R)
- Many descriptive statistics methods have *skipna* option to control if missing data should be excluded . This value is set to *True* by default (unlike R)

Aggregation Functions in Pandas

Aggregation - computing a summary statistic about each group, i.e.

- compute group sums or means
- compute group sizes/counts

Common aggregation functions:

min, max

count, sum, prod

mean, median, mode, mad

std, var

Aggregation Functions in Pandas

agg() method are useful when multiple statistics are computed per column:

```
In [ ]: flights[['dep_delay', 'arr_delay']].agg(['min', 'mean', 'max'])
```

Out[]:

	dep_delay	arr_delay
min	-16.000000	-62.000000
mean	9.384302	2.298675
max	351.000000	389.000000

Basic Descriptive Statistics

df.method()	description
describe	Basic statistics (count, mean, std, min, quantiles, max)
min, max	Minimum and maximum values
mean, median, mode	Arithmetic average, median and mode
var, std	Variance and standard deviation
sem	Standard error of mean
skew	Sample skewness
kurt	kurtosis

Graphics to explore the data

Seaborn package is built on matplotlib but provides high level interface for drawing attractive statistical graphics, similar to ggplot2 library in R. It specifically targets statistical data visualization

To show graphs within Python notebook include inline directive:

```
In [ ]: %matplotlib inline
```

Graphics

	description
distplot	histogram
barplot	estimate of central tendency for a numeric variable
violinplot	similar to boxplot, also shows the probability density of the data
jointplot	Scatterplot
regplot	Regression plot
pairplot	Pairplot
boxplot	boxplot
swarmplot	categorical scatterplot
factorplot	General categorical plot

Basic statistical Analysis

statsmodel and scikit-learn - both have a number of function for statistical analysis

The first one is mostly used for regular analysis using R style formulas, while scikit-learn is more tailored for Machine Learning.

statsmodels:

- linear regressions
- ANOVA tests
- hypothesis testings
- many more ...

scikit-learn:

- kmeans
- support vector machines
- random forests
- many more ...

See examples in the Tutorial Notebook

References

- pandas documentation
 - <https://pandas.pydata.org/pandas-docs/stable/index.html>
- pandas: Input/output
 - <https://pandas.pydata.org/pandas-docs/stable/reference/io.html>
- pandas: DataFrame
 - <https://pandas.pydata.org/pandas-docs/stable/reference/frame.html>
- pandas: Series
 - <https://pandas.pydata.org/pandas-docs/stable/reference/series.html>
- pandas: Plotting
 - <https://pandas.pydata.org/pandas-docs/stable/reference/plotting.html>