Lecture 5

Introduction to PANDAS

Introduction

Pandas is an open source Python library for data analysis.

It gives Python the ability of fast data loading, manipulating, aligning, and merging, among other functions

Data Frames and Series

The DataFrame represents your entire spreadsheet or rectangular data

the Series is a single column of the DataFrame

A Pandas DataFrame can also be thought of as a dictionary or collection of Series objects.

1 Objectives

1. Loading a simple delimited data file

2. Counting how many rows and columns were loaded

3. Determining which type of data was loaded

4. Looking at different parts of the data by subsetting rows and columns

Installing Pandas

Install anaconda from https://www.continuum.io/downloads

Run:

```
conda install pandas jupyter
```

Run:

```
jupyter-notebook
```

Loading Your first dataset

- ✘ With the library loaded, we can use the read_csv function to load a CSV data file.

```
import pandas
```

Loading Your first dataset

- ✘ by default the read_csv function will read a comma-separated file
- ✘ The below Gapminder data that we are using is separated by tabs
- ✘ We can use the sep parameter and indicate a tab with \t

```
df = pandas.read_csv('../data/gapminder.tsv', sep='\t')
```

Note: The repository can be found at: www.github.com/jennybc/gapminder.

Loading Your first dataset

- ✗ we use the head method so Python shows us only the first 5 rows

```
print(df.head())
```

Note: The repository can be found at: www.github.com/jennybc/gapminder.

Loading Your first dataset

- ✘ we use the head method so Python shows us only the first 5 rows

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
1	Afghanistan	Asia	1957	30.332	9240934	820.853030
2	Afghanistan	Asia	1962	31.997	10267083	853.100710
3	Afghanistan	Asia	1967	34.020	11537966	836.197138
4	Afghanistan	Asia	1972	36.088	13079460	739.981106

Loading Your first dataset

- ✘ It is common to use pd as alias for pandas

```python
import pandas as pd

df = pd.read_csv('../data/gapminder.tsv', sep='\t')
```

```python
print(type(df))
```

```
<class 'pandas.core.frame.DataFrame'>
```

Loading Your first dataset

- ✘ The shape attribute returns a tuple in which the first value is the number of rows and the second number is the number of columns.

```
# get the number of rows and columns

print(df.shape)

 (1704, 6)
```

Gapminder data set has 1704 rows and 6 columns.

Loading Your first dataset

```python
# get column names

print(df.columns)
```

```
Index(['country', 'continent', 'year', 'lifeExp', 'pop',

       'gdpPercap'],

      dtype='object')
```

Loading Your first dataset

Question

What is the `type` of the column names?

Loading Your first dataset

```python
# get the dtype of each column

print(df.dtypes)
```

```
country        object

continent      object

year            int64

lifeExp       float64

pop             int64

gdpPercap     float64

dtype: object
```

Loading your first dataset

```python
# get more information about our data

print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>

RangeIndex: 1704 entries, 0 to 1703

Data columns (total 6 columns):

country      1704 non-null object

continent    1704 non-null object

year         1704 non-null int64

lifeExp      1704 non-null float64

pop          1704 non-null int64

gdpPercap    1704 non-null float64

dtypes: float64(2), int64(2), object(2)

memory usage: 80.0+ KB

None
```

Python Type vs Pandas Type

Pandas Type	Python Type	Description
`object`	`string`	Most common data type
`int64`	`int`	Whole numbers
`float64`	`float`	Numbers with decimals
`datetime64`	`datetime`	`datetime` is found in the Python standard library (i.e., it is not loaded by default and needs to be imported)

Looking at Columns, Rows, and Cells

```python
# just get the country column and save it to its own variable
country_df = df['country']


# show the first 5 observations
print(country_df.head())
```

```
0     Afghanistan
1     Afghanistan
2     Afghanistan
3     Afghanistan
4     Afghanistan
Name: country, dtype: object
```

Looking at Columns, Rows, and Cells

```
# show the last 5 observations

print(country_df.tail())
```

```
1699    Zimbabwe

1700    Zimbabwe

1701    Zimbabwe

1702    Zimbabwe

1703    Zimbabwe

Name: country, dtype: object
```

Looking at Columns, Rows, and Cells

```python
# Looking at country, continent, and year

subset = df[['country', 'continent', 'year']]

print(subset.head())
```

```
       country continent  year

0  Afghanistan      Asia  1952

1  Afghanistan      Asia  1957

2  Afghanistan      Asia  1962

3  Afghanistan      Asia  1967

4  Afghanistan      Asia  1972
```

Looking at Columns, Rows, and Cells

```python
# Looking at country, continent, and year
subset = df[['country', 'continent', 'year']]
print(subset.tail())
```

```
      country continent  year
1699  Zimbabwe    Africa  1987
1700  Zimbabwe    Africa  1992
1701  Zimbabwe    Africa  1997
1702  Zimbabwe    Africa  2002
1703  Zimbabwe    Africa  2007
```

Subsetting Columns/Rows by Index Position

→ use the loc attribute on the dataframe to subset rows based on the index label.

```
# get the first row

# Python counts from 0

print(df.loc[0])
```

```
country      Afghanistan

continent           Asia

year                1952

lifeExp           28.801

pop              8425333

gdpPercap        779.445

Name: 0, dtype: object
```

Subsetting Columns/rows by Index Position

→ use the loc attribute on the dataframe to subset rows based on the index label.

```python
# get the 100th row
# Python counts from 0
print(df.loc[99])
```

```
country      Bangladesh
continent          Asia
year               1967
lifeExp          43.453
pop            62821884
gdpPercap       721.186
Name: 99, dtype: object
```

Subsetting Columns/rows by Index Position

```
# get the last row (correctly)

# use the first value given from shape to get the number of rows

number_of_rows = df.shape[0]



# subtract 1 from the value since we want the last index value

last_row_index = number_of_rows - 1



# now do the subset using the index of the last row

print(df.loc[last_row_index])
```

```
country        Zimbabwe

continent        Africa

year               2007

lifeExp          43.487

pop            12311143

gdpPercap       469.709

Name: 1703, dtype: object
```

SELECTING MULTIPLE ROWS

```
# select the first, 100th, and 1000th rows

# note the double square brackets similar to the syntax used to

# subset multiple columns

print(df.loc[[0, 99, 999]])
```

	country	continent	year	lifeExp	pop	gdpPercap
0	Afghanistan	Asia	1952	28.801	8425333	779.445314
99	Bangladesh	Asia	1967	43.453	62821884	721.186086
999	Mongolia	Asia	1967	51.253	1149500	1226.041130

SELECTING MULTIPLE ROWS

```
# subset columns with loc

# note the position of the colon

# it is used to select all rows

subset = df.loc[:, ['year', 'pop']]

print(subset.head())
```

	year	pop
0	1952	8425333
1	1957	9240934
2	1962	10267083
3	1967	11537966
4	1972	13079460

SELECTING MULTIPLE ROWS

```
# using loc

print(df.loc[42, 'country'])
```

```
# using iloc

print(df.iloc[42, 0])
```

```
Angola
```

“

Grouped Means

- We first split our data into various parts, then apply a function of our choosing to each of the split parts.
- Finally we combine all the individual split calculations into a single dataframe.

Grouped/Aggregate Computations using Grouped Means

```
# For each year in our data, what was the average life expectancy?
```

```
# To answer this question,

# we need to split our data into parts by year;

# then we get the 'lifeExp' column and calculate the mean
```

Grouped/Aggregate Computations using Groupby

```
print(df.groupby('year')['lifeExp'].mean())
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

Grouped/Aggregate Computations using Grouped Means

```
grouped_year_df = df.groupby('year')
```

```
grouped_year_df_lifeExp = grouped_year_df['lifeExp']
```

```
mean_lifeExp_by_year = grouped_year_df_lifeExp.mean()

print(mean_lifeExp_by_year)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

Grouped Means on Multiple Columns

```
multi_group_var = df.\
    groupby(['year', 'continent'])\
    [['lifeExp', 'gdpPercap']].\
    mean()
print(multi_group_var)
```

		lifeExp	gdpPercap
year	continent		
1952	Africa	39.135500	1252.572466
	Americas	53.279840	4079.062552
	Asia	46.314394	5195.484004
	Europe	64.408500	5661.057435
	Oceania	69.255000	10298.085650
1957	Africa	41.266346	1385.236062
	Americas	55.960280	4616.043733
	Asia	49.318544	5787.732940
	Europe	66.703067	6963.012816
	Oceania	70.295000	11598.522455
1962	Africa	43.319442	1598.078825
	Americas	58.398760	4901.541870
	Asia	51.563223	5729.369625
	Europe	68.539233	8365.486814
	Oceania	71.085000	12696.452430

OBSERVATIONS

The output data is grouped by year and continent.

For each year-continent pair, we calculated the average life expectancy and average GDP.

There is some hierarchical structure between the year and continent row indices.

Flattening the Dataframe

```
flat = multi_group_var.reset_index()

print(flat.head(15))
```

	year	continent	lifeExp	gdpPercap
0	1952	Africa	39.135500	1252.572466
1	1952	Americas	53.279840	4079.062552
2	1952	Asia	46.314394	5195.484004
3	1952	Europe	64.408500	5661.057435
4	1952	Oceania	69.255000	10298.085650
5	1957	Africa	41.266346	1385.236062
6	1957	Americas	55.960280	4616.043733
7	1957	Asia	49.318544	5787.732940
8	1957	Europe	66.703067	6963.012816
9	1957	Oceania	70.295000	11598.522455
10	1962	Africa	43.319442	1598.078825
11	1962	Americas	58.398760	4901.541870
12	1962	Asia	51.563223	5729.369625
13	1962	Europe	68.539233	8365.486814
14	1962	Oceania	71.085000	12696.452430

CALCULATING FREQUENCIES

We can use the nunique and value_counts methods, respectively, to get counts of unique values and frequency counts on Pandas Series.

```python
# use the nunique (number unique)

# to calculate the number of unique values in a series

print(df.groupby('continent')['country'].nunique())
```

```
continent

Africa      52

Americas    25

Asia        33

Europe      30

Oceania      2

Name: country, dtype: int64
```

Basic Plots

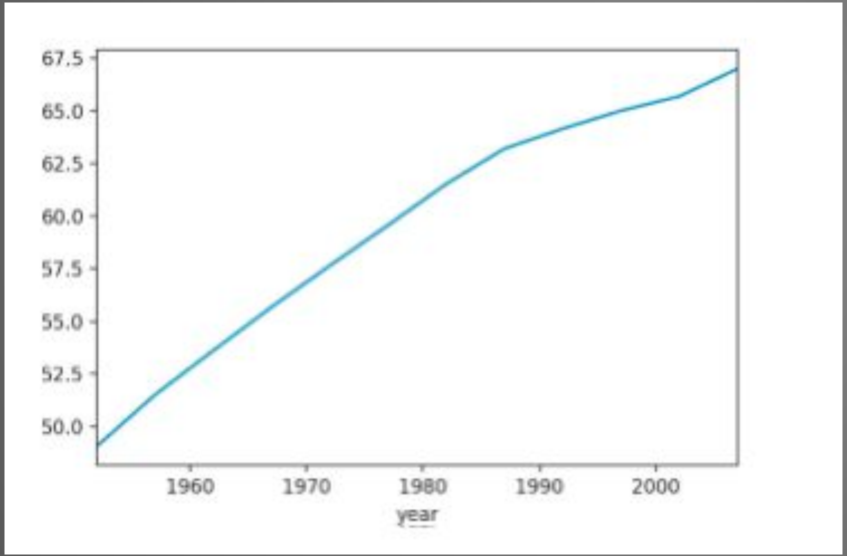
```
global_yearly_life_expectancy = df.groupby('year')['lifeExp'].mean()

print(global_yearly_life_expectancy)
```

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

Basic Plots

```
global_yearly_life_expectancy.plot()
```



Basic plot in Pandas showing average life expectancy over time

```
year
1952    49.057620
1957    51.507401
1962    53.609249
1967    55.678290
1972    57.647386
1977    59.570157
1982    61.533197
1987    63.212613
1992    64.160338
1997    65.014676
2002    65.694923
2007    67.007423
Name: lifeExp, dtype: float64
```

DATA STRUCTURES : OBJECTIVES

1. Loading in manual data
2. The Series object
3. Basic operations on Series objects
4. The DataFrame object
5. Conditional subsetting and fancy slicing and indexing
6. Saving out data

DATA STRUCTURES : SERIES

The Pandas Series is a one-dimensional container, similar to the built-in Python list.

It is the data type that represents each column of the DataFrame.

Each column in a dataframe must be of the same dtype.

DATA STRUCTURES : DATAFRAMES

A Dataframe can be thought of a dictionary of Series objects, where each key is the column name and the value is the Series

Creating a series

```python
import pandas as pd



s = pd.Series(['banana', 42])

print(s)
```

```
0     banana

1         42

dtype: object
```

Creating a series

```python
# manually assign index values to a series
# by passing a Python list
s = pd.Series(['Wes McKinney', 'Creator of Pandas'],
              index=['Person', 'Who'])
print(s)
```

```
Person         Wes McKinney
Who       Creator of Pandas
dtype: object
```

Creating a dataframe

DataFrame can be thought of as a dictionary of Series objects

The key represents the column name, and the values are the contents of the column.

Creating a dataframe

```python
scientists = pd.DataFrame({

    'Name': ['Rosaline Franklin', 'William Gosset'],

    'Occupation': ['Chemist', 'Statistician'],

    'Born': ['1920-07-25', '1876-06-13'],

    'Died': ['1958-04-16', '1937-10-16'],

    'Age': [37, 61]})

print(scientists)
```

	Age	Born	Died	Name	Occupation
0	37	1920-07-25	1958-04-16	Rosaline Franklin	Chemist
1	61	1876-06-13	1937-10-16	William Gosset	Statistician

Notice that the order is not guaranteed

Creating a dataframe

```
scientists = pd.DataFrame(

    data={'Occupation': ['Chemist', 'Statistician'],

          'Born': ['1920-07-25', '1876-06-13'],

          'Died': ['1958-04-16', '1937-10-16'],

    'Age': [37, 61]},

    index=['Rosaline Franklin', 'William Gosset'],

    columns=['Occupation', 'Born', 'Died', 'Age'])

print(scientists)
```

We can set the columns parameter or specify the column order.

If we wanted to use the name column for the row index, we can use the index parameter.

	Occupation	Born	Died	Age
Rosaline Franklin	Chemist	1920-07-25	1958-04-16	37
William Gosset	Statistician	1876-06-13	1937-10-16	61

Creating a dataframe

```
from collections import OrderedDict

# note the round brackets after OrderedDict
# then we pass a list of 2-tuples
scientists = pd.DataFrame(OrderedDict([
    ('Name', ['Rosaline Franklin', 'William Gosset']),
    ('Occupation', ['Chemist', 'Statistician']),
    ('Born', ['1920-07-25', '1876-06-13']),
    ('Died', ['1958-04-16', '1937-10-16']),
    ('Age', [37, 61])
    ])
)
print(scientists)
```

```
                Name    Occupation        Born        Died  Age
0  Rosaline Franklin       Chemist  1920-07-25  1958-04-16   37
1     William Gosset  Statistician  1876-06-13  1937-10-16   61
```

Subsetting the first row of scientists dataframe

```python
# create our example dataframe

# with a row index label

scientists = pd.DataFrame(

    data={'Occupation': ['Chemist', 'Statistician'],

          'Born': ['1920-07-25', '1876-06-13'],

          'Died': ['1958-04-16', '1937-10-16'],

    'Age': [37, 61]},

    index=['Rosaline Franklin', 'William Gosset'],

    columns=['Occupation', 'Born', 'Died', 'Age'])

print(scientists)
```

```python
# select by row index label

first_row = scientists.loc['William Gosset']

print(type(first_row))
```

```
<class 'pandas.core.series.Series'>
```

Subsetting the first row of scientists dataframe

```python
# create our example dataframe

# with a row index label

scientists = pd.DataFrame(

    data={'Occupation': ['Chemist', 'Statistician'],

          'Born': ['1920-07-25', '1876-06-13'],

          'Died': ['1958-04-16', '1937-10-16'],

    'Age': [37, 61]},

    index=['Rosaline Franklin', 'William Gosset'],

    columns=['Occupation', 'Born', 'Died', 'Age'])

print(scientists)
```

```python
print(first_row)
```

When a series is printed, the **index** is printed as the first "column," and the **values** are printed as the second "column."

```
Occupation    Statistician

Born            1876-06-13

Died            1937-10-16

Age                     61

Name: William Gosset, dtype: object
```

Attributes and Methods of series

```
print(first_row.index)
```

```
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

```
print(first_row.values)
```

```
['Statistician' '1876-06-13' '1937-10-16' 61]
```

```
print(first_row.keys())
```

```
Index(['Occupation', 'Born', 'Died', 'Age'], dtype='object')
```

Attributes and Methods of series

```
# get the first index using an attribute

print(first_row.index[0])
```

```
Occupation
```

```
# get the first index using a method

print(first_row.keys()[0])
```

```
Occupation
```

Attributes and Methods of series

Series	Attributes Description
`loc`	Subset using index value
`iloc`	Subset using index position
`ix`	Subset using index value and/or position
`dtype` or `dtypes`	The type of the `Series` contents
`T`	Transpose of the series
`shape`	Dimensions of the data
`size`	Number of elements in the `Series`
`values`	`ndarray` or `ndarray`-like of the `Series`

Attributes and Methods of series

Series Methods	Description
append	Concatenates two or more Series
corr	Calculate a correlation with another Series *
cov	Calculate a covariance with another Series *
describe	Calculate summary statistics *
drop_duplicates	Returns a Series without duplicates
equals	Determines whether a Series has the same elements
get_values	Get values of the Series; same as the values attribute
hist	Draw a histogram

to_frame	Converts a Series to a DataFrame
transpose	Returns the transpose
unique	Returns a numpy.ndarray of unique values

Attributes and Methods of series

`isin`	Checks whether values are contained in a `Series`
`min`	Returns the minimum value
`max`	Returns the maximum value
`mean`	Returns the arithmetic mean
`median`	Returns the median
`mode`	Returns the mode(s)
`quantile`	Returns the value at a given quantile
`replace`	Replaces values in the `Series` with a specified value
`sample`	Returns a random sample of values from the `Series`
`sort_values`	Sorts values

Attributes and Methods of series

```
# get the 'Age' column
ages = scientists['Age']
print(ages)
```

```
Rosaline Franklin    37
William Gosset       61
Name: Age, dtype: int64
```

Attributes and Methods of series

```
print(ages.mean())
```

```
49.0
```

```
print(ages.min())
```

```
37
```

```
print(ages.max())
```

```
61
```

```
print(ages.std())
```

```
16.9705627485
```

MISSING VALUES

Missing Values

1. Missing values are marked as NaN

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

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

Missing Values

1. Missing values are marked as NaN

	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	EWR	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	EWR	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

1. Missing values are marked as NaN

	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	EWR	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	EWR	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 values 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

1. When summing the data, missing values will be treated as zero.
2. If all values are missing, the sum will be equal to NaN.
3. cumsum() and cumprod() methods ignore missing while calculating values but preserve them in resulting array.
4. Missing values in GroupBy Method are excluded.
5. Many descriptive statistics methods have *skipna* option to control if missing data should be excluded.
6. By default, This value is set to true *(unlike R).*

Aggregation Functions

1. Aggregation
   i. Computing a Summary Statistic about each group
      1. E.g., Compute group sums or means
      2. E.g., Compute group sizes/counts.

Aggregation Functions

1. Common Aggregation Functions
   - A. min, max
   - B. count, sum, prod
   - C. mean, median, mode, mad
   - D. std. var

Aggregation Functions

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

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

Aggregation Functions

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

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

Aggregation Functions

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

Pandas reader and writer functions

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

EXCEL FILES

```
# Returns a DataFrame
read_excel('path_to_file.xls', sheet_name='Sheet1')
```

```
xlsx = pd.ExcelFile('path_to_file.xls')
df = pd.read_excel(xlsx, 'Sheet1')
```

EXCEL FILES

```python
with pd.ExcelFile('path_to_file.xls') as xls:
    df1 = pd.read_excel(xls, 'Sheet1')
    df2 = pd.read_excel(xls, 'Sheet2')
```

```python
# Returns a DataFrame
read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

```python
# Returns a DataFrame
read_excel('path_to_file.xls', 0, index_col=None, na_values=['NA'])
```

EXCEL FILES

```
df.to_excel('path_to_file.xlsx', sheet_name='Sheet1')
```

```
with ExcelWriter('path_to_file.xlsx') as writer:
    df1.to_excel(writer, sheet_name='Sheet1')
    df2.to_excel(writer, sheet_name='Sheet2')
```

SAS FILES

```
df = pd.read_sas('sas_data.sas7bdat')
```

```
rdr = pd.read_sas('sas_xport.xpt', chunk=100000)
for chunk in rdr:
    do_something(chunk)
```

SQL QUERY

`read_sql_table`(table_name, con[, schema, ...])	Read SQL database table into a DataFrame.
`read_sql_query`(sql, con[, index_col, ...])	Read SQL query into a DataFrame.
`read_sql`(sql, con[, index_col, ...])	Read SQL query or database table into a DataFrame.
`DataFrame.to_sql`(name, con[, schema, ...])	Write records stored in a DataFrame to a SQL database.

SQL QUERY

```
with engine.connect() as conn, conn.begin():
    data = pd.read_sql_table('data', conn)
```

```
In [529]: data.to_sql('data', engine)
```

SQL QUERY

```
In [533]: pd.read_sql_table('data', engine)
Out[533]:
   index  id       Date Col_1  Col_2  Col_3
0      0  26 2010-10-18     X  27.50   True
1      1  42 2010-10-19     Y -12.50  False
2      2  63 2010-10-20     Z   5.73   True
```

SQL QUERY

```
In [534]: pd.read_sql_table('data', engine, index_col='id')
Out[534]:
    index       Date Col_1  Col_2  Col_3
id
26      0 2010-10-18     X  27.50   True
42      1 2010-10-19     Y -12.50  False
63      2 2010-10-20     Z   5.73   True

In [535]: pd.read_sql_table('data', engine, columns=['Col_1', 'Col_2'])
Out[535]:
  Col_1  Col_2
0     X  27.50
1     Y -12.50
2     Z   5.73
```

SQL QUERY

```
In [537]: pd.read_sql_query('SELECT * FROM data', engine)
Out[537]:
   index  id                        Date Col_1  Col_2  Col_3
0      0  26  2010-10-18 00:00:00.000000     X  27.50      1
1      1  42  2010-10-19 00:00:00.000000     Y -12.50      0
2      2  63  2010-10-20 00:00:00.000000     Z   5.73      1
```

SQL QUERY

```python
from pandas.io import sql
sql.execute('SELECT * FROM table_name', engine)
sql.execute('INSERT INTO table_name VALUES(?, ?, ?)', engine,
            params=[('id', 1, 12.2, True)])
```

SQL QUERY

```python
from sqlalchemy import create_engine

engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')

engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')

engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

engine = create_engine('mssql+pyodbc://mydsn')

# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine('sqlite:///foo.db')

# or absolute, starting with a slash:
engine = create_engine('sqlite:////absolute/path/to/foo.db')
```

SQL QUERY

```python
from sqlalchemy import create_engine
```

```python
# Parameters
ServerName = "DAVID-THINK"
Database = "BizIntel"
Driver = "driver=SQL Server Native Client 11.0"

# Create the connection
engine = create_engine('mssql+pyodbc://' + ServerName + '/' + Database + "?" + Driver)

df = pd.read_sql_query("SELECT top 5 * FROM data", engine)
df
```

	Date	Symbol	Volume
0	2013-01-01	A	0.0
1	2013-01-02	A	200.0
2	2013-01-03	A	1200.0
3	2013-01-04	A	1001.0
4	2013-01-05	A	1300.0

ConCATENATE

```python
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                   index=[0, 1, 2, 3])


df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                    index=[4, 5, 6, 7])


df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                   index=[8, 9, 10, 11])
```

ConCATENATE

```
frames = [df1, df2, df3]

result = pd.concat(frames)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

CONCATENATE

```python
result = pd.concat(frames, keys=['x', 'y', 'z'])
```

ConCATENATE

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

		A	B	C	D
x	0	A0	B0	C0	D0
x	1	A1	B1	C1	D1
x	2	A2	B2	C2	D2
x	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
y	5	A5	B5	C5	D5
y	6	A6	B6	C6	D6
y	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
z	9	A9	B9	C9	D9
z	10	A10	B10	C10	D10
z	11	A11	B11	C11	D11

ConCATENATE

```
result = pd.concat([df1, df4], axis=1, join='inner')
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

ConCATENATE

```
result = df1.append(df2)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

CONCATENATE

```
result = df1.append(df4)
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df4

	B	D	F
2	B2	D2	F2
3	B3	D3	F3
6	B6	D6	F6
7	B7	D7	F7

Result

	A	B	C	D	F
0	A0	B0	C0	D0	NaN
1	A1	B1	C1	D1	NaN
2	A2	B2	C2	D2	NaN
3	A3	B3	C3	D3	NaN
2	NaN	B2	NaN	D2	F2
3	NaN	B3	NaN	D3	F3
6	NaN	B6	NaN	D6	F6
7	NaN	B7	NaN	D7	F7

ConCATENATE

```
result = df1.append([df2, df3])
```

df1

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

df2

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

df3

	A	B	C	D
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Result

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Advantages of Pandas

1. Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data
2. Intelligent label-based slicing, fancy indexing, and subsetting of large data sets
3. Robust IO tools for loading data from flat files (CSV and delimited), Excel files, databases
4. Intuitive merging and joining data sets

THANKS!

Any questions?

You can find me at
ankita.sinha@gmail