2030ICT/7030ICT

Introduction to Big Data Analytics

Lab 1.2 – Data preparation and pre-processing

Trimester 2 - 2020

Table of Contents

I. Pandas Introduction

Pandas is a Python library that makes handling tabular data easier. Since we're doing data science - this is something we'll use from time to time!

It's one of three libraries you'll encounter repeatedly in the field of data science:

Pandas

Introduces "Data Frames" and "Series" that allow you to slice and dice rows and columns of information.

NumPy

Usually you'll encounter "NumPy arrays", which are multi-dimensional array objects. It is easy to create a Pandas DataFrame from a NumPy array, and Pandas DataFrames can be cast as NumPy arrays. NumPy arrays are mainly important because of...

Scikit_Learn

The machine learning library we'll use throughout this course is scikit_learn, or sklearn, and it generally takes NumPy arrays as its input.

So, a typical thing to do is to load, clean, and manipulate your input data using Pandas. Then convert your Pandas DataFrame into a NumPy array as it's being passed into some Scikit_Learn function. That conversion can often happen automatically.

Let's start by loading some comma-separated value data using Pandas into a DataFrame:

In [1]:
```
%matplotlib inline
import numpy as np
import pandas as pd

df = pd.read_csv("PastHires.csv")
df.head()
```

Out[1]:

	Years Experience	Employed?	Previous employers	Level of Education	Top-tier school	Interned	Hired
0	10	Y	4	BS	N	N	Y
1	0	N	0	BS	Y	Y	Y
2	7	N	6	BS	N	N	N
3	2	Y	1	MS	Y	N	Y
4	20	N	2	PhD	Y	N	N

head() is a handy way to visualize what you've loaded. You can pass it an integer to see some specific number of rows at the beginning of your DataFrame:

```
In [2]: df.head(10)
```

Out[2]:

	Years Experience	Employed?	Previous employers	Level of Education	Top-tier school	Interned	Hired
0	10	Y	4	BS	N	N	Y
1	0	N	0	BS	Y	Y	Y
2	7	N	6	BS	N	N	N
3	2	Y	1	MS	Y	N	Y
4	20	N	2	PhD	Y	N	N
5	0	N	0	PhD	Y	Y	Y
6	5	Y	2	MS	N	Y	Y
7	3	N	1	BS	N	Y	Y
8	15	Y	5	BS	N	N	Y
9	0	N	0	BS	N	N	N

You can also view the end of your data with tail():

```
In [3]: df.tail(4)
```

Out[3]:

	Years Experience	Employed?	Previous employers	Level of Education	Top-tier school	Interned	Hired
9	0	N	0	BS	N	N	N
10	1	N	1	PhD	Y	N	N
11	4	Y	1	BS	N	Y	Y
12	0	N	0	PhD	Y	N	Y

We often talk about the "shape" of your DataFrame. This is just its dimensions. This particular CSV file has 13 rows with 7 columns per row:

```
In [4]: df.shape
Out[4]: (13, 7)
```

The total size of the data frame is the rows * columns:

```
In [5]: df.size
Out[5]: 91
```

The len() function gives you the number of rows in a DataFrame:

```
In [6]: len(df)
Out[6]: 13
```

If your DataFrame has named columns (in our case, extracted automatically from the first row of a .csv file,) you can get an array of them back:

```
In [7]: df.columns
Out[7]: Index(['Years Experience', 'Employed?', 'Previous employers',
           'Level of Education', 'Top-tier school', 'Interned', 'Hired'],
          dtype='object')
```

Extracting a single column from your DataFrame looks like this - this gives you back a "Series" in Pandas:

```
In [8]: df['Hired']
```

```
Out[8]: 0     Y
        1     Y
        2     N
        3     Y
        4     N
        5     Y
        6     Y
        7     Y
        8     Y
        9     N
        10    N
        11    Y
        12    Y
        Name: Hired, dtype: object
```

You can also extract a given range of rows from a named column, like so:

```
In [9]: df['Hired'][:5]
```

```
Out[9]: 0    Y
        1    Y
        2    N
        3    Y
        4    N
        Name: Hired, dtype: object
```

Or even extract a single value from a specified column / row combination:

```
In [10]: df['Hired'][5]
```

```
Out[10]: 'Y'
```

To extract more than one column, you pass in a list of column names instead of a single one:

```
In [11]: df[['Years Experience', 'Hired']]
```

Out[11]:

	Years Experience	Hired
0	10	Y
1	0	Y
2	7	N
3	2	Y
4	20	N
5	0	Y
6	5	Y
7	3	Y
8	15	Y
9	0	N
10	1	N
11	4	Y
12	0	Y

You can also extract specific ranges of rows from more than one column, in the way you'd expect:

```
In [12]: df[['Years Experience', 'Hired']][:5]
```

Out[12]:

	Years Experience	Hired
0	10	Y
1	0	Y
2	7	N
3	2	Y
4	20	N

Sorting your DataFrame by a specific column looks like this:

```
In [13]: df.sort_values(['Years Experience'])
```

Out[13]:

	Years Experience	Employed?	Previous employers	Level of Education	Top-tier school	Interned	Hired
1	0	N	0	BS	Y	Y	Y
5	0	N	0	PhD	Y	Y	Y
9	0	N	0	BS	N	N	N
12	0	N	0	PhD	Y	N	Y
10	1	N	1	PhD	Y	N	N
3	2	Y	1	MS	Y	N	Y
7	3	N	1	BS	N	Y	Y
11	4	Y	1	BS	N	Y	Y
6	5	Y	2	MS	N	Y	Y
2	7	N	6	BS	N	N	N
0	10	Y	4	BS	N	N	Y
8	15	Y	5	BS	N	N	Y
4	20	N	2	PhD	Y	N	N

You can break down the number of unique values in a given column into a Series using value_counts() - this is a good way to understand the distribution of your data:

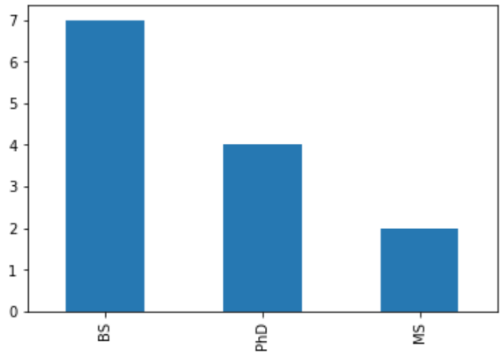
```
In [14]: degree_counts = df['Level of Education'].value_counts()
         degree_counts

Out[14]: BS     7
         PhD    4
         MS     2
         Name: Level of Education, dtype: int64
```

Pandas even makes it easy to plot a Series or DataFrame - just call plot():

```
In [15]: degree_counts.plot(kind='bar')

Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x28f6d2b0240>
```



II. Series

The first main data type we will learn about for pandas is the Series data type. Let's import Pandas and explore the Series object.

A Series is very similar to a NumPy array (in fact it is built on top of the NumPy array object). What differentiates the NumPy array from a Series, is that a Series can have axis labels, meaning it can be indexed by a label, instead of just a number location. It also doesn't need to hold numeric data, it can hold any arbitrary Python Object.

Let's explore this concept through some examples:

```
In [2]: import numpy as np
        import pandas as pd
```

1. Creating a Series

You can convert a list,numpy array, or dictionary to a Series:

```
In [3]: labels = ['a','b','c']
        my_list = [10,20,30]
        arr = np.array([10,20,30])
        d = {'a':10,'b':20,'c':30}
```

Using Lists

```
In [4]: pd.Series(data=my_list)
Out[4]: 0    10
        1    20
        2    30
        dtype: int64

In [5]: pd.Series(data=my_list,index=labels)
Out[5]: a    10
        b    20
        c    30
        dtype: int64

In [6]: pd.Series(my_list,labels)
Out[6]: a    10
        b    20
        c    30
        dtype: int64
```

NumPy Arrays

```
In [7]: pd.Series(arr)
Out[7]: 0    10
        1    20
        2    30
        dtype: int64

In [8]: pd.Series(arr,labels)
Out[8]: a    10
        b    20
        c    30
        dtype: int64
```

Dictionary

```
In [9]: pd.Series(d)
Out[9]: a    10
        b    20
        c    30
        dtype: int64
```

2. Data in Series

A pandas Series can hold a variety of object types:

```
In [10]: pd.Series(data=labels)
Out[10]: 0    a
         1    b
         2    c
         dtype: object
```

```
In [11]: # Even functions (although unlikely that you will use this)
         pd.Series([sum,print,len])
Out[11]: 0      <built-in function sum>
         1    <built-in function print>
         2      <built-in function len>
         dtype: object
```

3. Using an index

The key to using a Series is understanding its index. Pandas makes use of these index names or numbers by allowing for fast look ups of information (works like a hash table or dictionary).

Let's see some examples of how to grab information from a Series. Let us create two sereis, ser1 and ser2:

```
In [12]: ser1 = pd.Series([1,2,3,4],index = ['USA', 'Germany','USSR', 'Japan'])
```

```
In [13]: ser1
Out[13]: USA        1
         Germany    2
         USSR       3
         Japan      4
         dtype: int64
```

```
In [14]: ser2 = pd.Series([1,2,5,4],index = ['USA', 'Germany','Italy', 'Japan'])
```

```
In [15]: ser2
Out[15]: USA        1
         Germany    2
         Italy      5
         Japan      4
         dtype: int64
```

```
In [16]: ser1['USA']
Out[16]: 1
```

Operations are then also done based off of index:

```
In [17]: ser1 + ser2
Out[17]: Germany    4.0
         Italy      NaN
         Japan      8.0
         USA        2.0
         USSR       NaN
         dtype: float64
```

Let's stop here for now and move on to DataFrames, which will expand on the concept of Series!

III. DataFrames

DataFrames are the workhorse of pandas and are directly inspired by the R programming language. We can think of a DataFrame as a bunch of Series objects put together to share the same index. Let's use pandas to explore this topic!

```
In [183]: import pandas as pd
          import numpy as np

In [184]: from numpy.random import randn
          np.random.seed(101)

In [185]: df = pd.DataFrame(randn(5,4),index='A B C D E'.split(),columns='W X Y Z'.split())

In [186]: df
Out[186]:
```

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

1. Selection and Indexing

Let's learn the various methods to grab data from a DataFrame

```
In [187]: df['W']

Out[187]: A    2.706850
          B    0.651118
          C   -2.018168
          D    0.188695
          E    0.190794
          Name: W, dtype: float64

In [188]: # Pass a list of column names
          df[['W','Z']]

Out[188]:
```

	W	Z
A	2.706850	0.503826
B	0.651118	0.605965
C	-2.018168	-0.589001
D	0.188695	0.955057
E	0.190794	0.683509

```
In [189]: # SQL Syntax (NOT RECOMMENDED!)
          df.W

Out[189]: A    2.706850
          B    0.651118
          C   -2.018168
          D    0.188695
          E    0.190794
          Name: W, dtype: float64
```

DataFrame Columns are just Series

```
In [190]: type(df['W'])

Out[190]: pandas.core.series.Series
```

Creating a new column:

In [191]:
```
df['new'] = df['W'] + df['Y']
```

In [192]:
```
df
```

Out[192]:

	W	X	Y	Z	new
A	2.706850	0.628133	0.907969	0.503826	3.614819
B	0.651118	-0.319318	-0.848077	0.605965	-0.196959
C	-2.018168	0.740122	0.528813	-0.589001	-1.489355
D	0.188695	-0.758872	-0.933237	0.955057	-0.744542
E	0.190794	1.978757	2.605967	0.683509	2.796762

Removing Columns

In [193]:
```
# Return a new DataFrame with the 'new'
# column dropped
df.drop('new',axis=1)
```

Out[193]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

In [194]:
```
# Not inplace unless specified!
df
```

Out[194]:

	W	X	Y	Z	new
A	2.706850	0.628133	0.907969	0.503826	3.614819
B	0.651118	-0.319318	-0.848077	0.605965	-0.196959
C	-2.018168	0.740122	0.528813	-0.589001	-1.489355
D	0.188695	-0.758872	-0.933237	0.955057	-0.744542
E	0.190794	1.978757	2.605967	0.683509	2.796762

In [195]:
```
# Drop the 'new' column of DataFrame itself
df.drop('new',axis=1,inplace=True)
```

In [196]:
```
df
```

Out[196]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

Can also drop rows this way:

In [197]:
```
df.drop('E',axis=0)
```

Out[197]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057

Can also drop rows this way:

In [197]:
```
df.drop('E',axis=0)
```

Out[197]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057

Selecting Rows

In [198]:
```
df.loc['A']
```

Out[198]:
```
W    2.706850
X    0.628133
Y    0.907969
Z    0.503826
Name: A, dtype: float64
```

Or select based off of position instead of label

In [199]:
```
df.iloc[2]
```
Out[199]:
```
W   -2.018168
X    0.740122
Y    0.528813
Z   -0.589001
Name: C, dtype: float64
```

Selecting subset of rows and columns

In [200]:
```
df.loc['B','Y']
```
Out[200]:
```
-0.84807698340363147
```

In [201]:
```
df.loc[['A','B'],['W','Y']]
```
Out[201]:

	W	Y
A	2.706850	0.907969
B	0.651118	-0.848077

2. Conditional Selection

An important feature of pandas is conditional selection using bracket notation, very similar to numpy:

In [202]:
```
df
```
Out[202]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

In [203]:
```
df>0
```
Out[203]:

	W	X	Y	Z
A	True	True	True	True
B	True	False	False	True
C	False	True	True	False
D	True	False	False	True
E	True	True	True	True

In [204]:
```
df[df>0]
```
Out[204]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	NaN	NaN	0.605965
C	NaN	0.740122	0.528813	NaN
D	0.188695	NaN	NaN	0.955057
E	0.190794	1.978757	2.605967	0.683509

In [205]:
```
df[df['W']>0]
```
Out[205]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

```
In [206]: df[df['W']>0]['Y']
```

```
Out[206]: A    0.907969
          B   -0.848077
          D   -0.933237
          E    2.605967
          Name: Y, dtype: float64
```

```
In [207]: df[df['W']>0][['Y','X']]
```

Out[207]:

	Y	X
A	0.907969	0.628133
B	-0.848077	-0.319318
D	-0.933237	-0.758872
E	2.605967	1.978757

For two conditions you can use | and & with parenthesis:

```
In [208]: df[(df['W']>0) & (df['Y'] > 1)]
```

Out[208]:

	W	X	Y	Z
E	0.190794	1.978757	2.605967	0.683509

3. More Index Details

Let's discuss some more features of indexing, including resetting the index or setting it something else. We'll also talk about index hierarchy!

```
In [209]: df
```

Out[209]:

	W	X	Y	Z
A	2.706850	0.628133	0.907969	0.503826
B	0.651118	-0.319318	-0.848077	0.605965
C	-2.018168	0.740122	0.528813	-0.589001
D	0.188695	-0.758872	-0.933237	0.955057
E	0.190794	1.978757	2.605967	0.683509

```
In [210]: # Reset to default 0,1...n index
          df.reset_index()
```

Out[210]:

	index	W	X	Y	Z
0	A	2.706850	0.628133	0.907969	0.503826
1	B	0.651118	-0.319318	-0.848077	0.605965
2	C	-2.018168	0.740122	0.528813	-0.589001
3	D	0.188695	-0.758872	-0.933237	0.955057
4	E	0.190794	1.978757	2.605967	0.683509

```
In [211]: newind = 'CA NY WY OR CO'.split()
```

```
In [212]: df['States'] = newind
```

```
In [213]: df
```

Out[213]:

	W	X	Y	Z	States
A	2.706850	0.628133	0.907969	0.503826	CA
B	0.651118	-0.319318	-0.848077	0.605965	NY
C	-2.018168	0.740122	0.528813	-0.589001	WY
D	0.188695	-0.758872	-0.933237	0.955057	OR
E	0.190794	1.978757	2.605967	0.683509	CO

In [214]:
```
df.set_index('States')
```
Out[214]:

	W	X	Y	Z
States				
CA	2.706850	0.628133	0.907969	0.503826
NY	0.651118	-0.319318	-0.848077	0.605965
WY	-2.018168	0.740122	0.528813	-0.589001
OR	0.188695	-0.758872	-0.933237	0.955057
CO	0.190794	1.978757	2.605967	0.683509

In [215]:
```
df
```
Out[215]:

	W	X	Y	Z	States
A	2.706850	0.628133	0.907969	0.503826	CA
B	0.651118	-0.319318	-0.848077	0.605965	NY
C	-2.018168	0.740122	0.528813	-0.589001	WY
D	0.188695	-0.758872	-0.933237	0.955057	OR
E	0.190794	1.978757	2.605967	0.683509	CO

In [216]:
```
df.set_index('States',inplace=True)
```
In [218]:
```
df
```
Out[218]:

	W	X	Y	Z
States				
CA	2.706850	0.628133	0.907969	0.503826
NY	0.651118	-0.319318	-0.848077	0.605965
WY	-2.018168	0.740122	0.528813	-0.589001
OR	0.188695	-0.758872	-0.933237	0.955057
CO	0.190794	1.978757	2.605967	0.683509

4. Multi-Index and Index Hierarchy

Let us go over how to work with Multi-Index, first we'll create a quick example of what a Multi-Indexed DataFrame would look like:

In [253]:
```
# Index Levels
outside = ['G1','G1','G1','G2','G2','G2']
inside = [1,2,3,1,2,3]
hier_index = list(zip(outside,inside))
hier_index = pd.MultiIndex.from_tuples(hier_index)
```
In [254]:
```
hier_index
```
Out[254]:
```
MultiIndex(levels=[['G1', 'G2'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2]])
```

Out[254]:
```
MultiIndex(levels=[['G1', 'G2'], [1, 2, 3]],
           labels=[[0, 0, 0, 1, 1, 1], [0, 1, 2, 0, 1, 2]])
```


In [257]:
```
df = pd.DataFrame(np.random.randn(6,2),index=hier_index,columns=['A','B'])
df
```
Out[257]:

		A	B
	1	0.153661	0.167638
G1	**2**	-0.765930	0.962299
	3	0.902826	-0.537909
	1	-1.549671	0.435253
G2	**2**	1.259904	-0.447898
	3	0.266207	0.412580

Now let's show how to index this! For index hierarchy we use df.loc[], if this was on the columns axis, you would just use normal bracket notation df[]. Calling one level of the index returns the sub-dataframe:

In [260]:
```
df.loc['G1']
```

Out[260]:

	A	B
1	0.153661	0.167638
2	-0.765930	0.962299
3	0.902826	-0.537909

In [263]:
```
df.loc['G1'].loc[1]
```

Out[263]:
```
A    0.153661
B    0.167638
Name: 1, dtype: float64
```

In [265]:
```
df.index.names
```

Out[265]:
```
FrozenList([None, None])
```

In [266]:
```
df.index.names = ['Group','Num']
```

In [267]:
```
df
```

Out[267]:

		A	B
Group	Num		
G1	1	0.153661	0.167638
	2	-0.765930	0.962299
	3	0.902826	-0.537909
G2	1	-1.549671	0.435253
	2	1.259904	-0.447898
	3	0.266207	0.412580

In [270]:
```
df.xs('G1')
```

Out[270]:

	A	B
Num		
1	0.153661	0.167638
2	-0.765930	0.962299
3	0.902826	-0.537909

In [271]:
```
df.xs(['G1',1])
```

Out[271]:
```
A    0.153661
B    0.167638
Name: (G1, 1), dtype: float64
```

In [273]:
```
df.xs(1,level='Num')
```

Out[273]:

	A	B
Group		
G1	0.153661	0.167638
G2	-1.549671	0.435253

IV. Missing Data

Let's show a few convenient methods to deal with Missing Data in pandas:

In [1]:
```
import numpy as np
import pandas as pd
```

In [9]:
```
df = pd.DataFrame({'A':[1,2,np.nan],
                   'B':[5,np.nan,np.nan],
                   'C':[1,2,3]})
```

In [10]:
```
df
```

Out[10]:

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

In [12]:
```
df.dropna()
```

Out[12]:

	A	B	C
0	1.0	5.0	1

In [13]:
```
df.dropna(axis=1)
```

Out[13]:

	C
0	1
1	2
2	3

In [14]:
```
df.dropna(thresh=2)
```

Out[14]:

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2

In [15]:
```
df.fillna(value='FILL VALUE')
```

Out[15]:

	A	B	C
0	1	5	1
1	2	FILL VALUE	2
2	FILL VALUE	FILL VALUE	3

In [17]:
```
df['A'].fillna(value=df['A'].mean())
```

Out[17]:
```
0    1.0
1    2.0
2    1.5
Name: A, dtype: float64
```

1. GroupBy

The groupby method allows you to group rows of data together and call aggregate functions

In [31]:
```
import pandas as pd
# Create dataframe
data = {'Company':['GOOG','GOOG','MSFT','MSFT','FB','FB'],
       'Person':['Sam','Charlie','Amy','Vanessa','Carl','Sarah'],
       'Sales':[200,120,340,124,243,350]}
```

In [32]:
```
df = pd.DataFrame(data)
```

```
In [33]: df
```

Out[33]:

	Company	Person	Sales
0	GOOG	Sam	200
1	GOOG	Charlie	120
2	MSFT	Amy	340
3	MSFT	Vanessa	124
4	FB	Carl	243
5	FB	Sarah	350

Now you can use the .groupby() method to group rows together based off of a column name. For instance let's group based off of Company. This will create a DataFrameGroupBy object:

```
In [34]: df.groupby('Company')
Out[34]: <pandas.core.groupby.DataFrameGroupBy object at 0x113014128>
```

You can save this object as a new variable:

```
In [35]: by_comp = df.groupby("Company")
```

And then call aggregate methods off the object:

```
In [36]: by_comp.mean()
```

Out[36]:

Company	Sales
FB	296.5
GOOG	160.0
MSFT	232.0

```
In [37]: df.groupby('Company').mean()
```

Out[37]:

Company	Sales
FB	296.5
GOOG	160.0
MSFT	232.0

More examples of aggregate methods:

```
In [38]: by_comp.std()
```

Out[38]:

Company	Sales
FB	75.660426
GOOG	56.568542
MSFT	152.735065

```
In [39]: by_comp.min()
```

Out[39]:

Company	Person	Sales
FB	Carl	243
GOOG	Charlie	120
MSFT	Amy	124

In [40]:
```
by_comp.max()
```
Out[40]:

Company	Person	Sales
FB	Sarah	350
GOOG	Sam	200
MSFT	Vanessa	340

In [41]:
```
by_comp.count()
```
Out[41]:

Company	Person	Sales
FB	2	2
GOOG	2	2
MSFT	2	2

In [42]:
```
by_comp.describe()
```
Out[42]:

Company		Sales
FB	count	2.000000
	mean	296.500000
	std	75.660426
	min	243.000000
	25%	269.750000
	50%	296.500000
	75%	323.250000
	max	350.000000
GOOG	count	2.000000
	mean	160.000000
	std	56.568542
	min	120.000000
	25%	140.000000
	50%	160.000000
	75%	180.000000
	max	200.000000
	count	2.000000
	mean	232.000000
	std	152.735065

In [43]:
```
by_comp.describe().transpose()
```
Out[43]:

Company	FB								GOO				
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	ma
Sales	2.0	296.5	75.660426	243.0	269.75	296.5	323.25	350.0	2.0	160.0	...	180.0	200.

1 rows × 24 columns

In [44]:
```
by_comp.describe().transpose()['GOOG']
```
Out[44]:

	count	mean	std	min	25%	50%	75%	max
Sales	2.0	160.0	56.568542	120.0	140.0	160.0	180.0	200.0

V. Merging, Joining and Concatenating

There are 3 main ways of combining DataFrames together: Merging, Joining and Concatenating. In this lecture we will discuss these 3 methods with examples.

1. Concatenation

Example DataFrame

In [3]:
```
import pandas as pd
```

In [4]:
```
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])
```

In [5]:
```
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])
```

In [6]:
```
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])
```

In [7]:
```
df1
```

Out[7]:

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

In [8]:
```
df2
```

Out[8]:

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

In [12]:
```
df3
```

Out[12]:

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

Concatenation basically glues together DataFrames. Keep in mind that dimensions should match along the axis you are concatenating on. You can use **pd.concat** and pass in a list of DataFrames to concatenate together:

In [10]:
```
pd.concat([df1,df2,df3])
```

Out[10]:

	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

In [18]:
```
pd.concat([df1,df2,df3],axis=1)
```

Out[18]:

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

2. Merging

Example DataFrame

In [28]:
```
left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                     'A': ['A0', 'A1', 'A2', 'A3'],
                     'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                      'C': ['C0', 'C1', 'C2', 'C3'],
                      'D': ['D0', 'D1', 'D2', 'D3']})
```

In [29]:
```
left
```

Out[29]:

	A	B	key
0	A0	B0	K0
1	A1	B1	K1
2	A2	B2	K2
3	A3	B3	K3

In [30]:
```
right
```

Out[30]:

	C	D	key
0	C0	D0	K0
1	C1	D1	K1
2	C2	D2	K2
3	C3	D3	K3

The **merge** function allows you to merge DataFrames together using a similar logic as merging SQL Tables together. For example:

In [35]:
```
pd.merge(left,right,how='inner',on='key')
```

Out[35]:

	A	B	key	C	D
0	A0	B0	K0	C0	D0
1	A1	B1	K1	C1	D1
2	A2	B2	K2	C2	D2
3	A3	B3	K3	C3	D3

Or to show a more complicated example:

In [37]:
```
left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                     'key2': ['K0', 'K1', 'K0', 'K1'],
                        'A': ['A0', 'A1', 'A2', 'A3'],
                        'B': ['B0', 'B1', 'B2', 'B3']})

right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                              'key2': ['K0', 'K0', 'K0', 'K0'],
                                 'C': ['C0', 'C1', 'C2', 'C3'],
                                 'D': ['D0', 'D1', 'D2', 'D3']})
```

In [39]:
```
pd.merge(left, right, on=['key1', 'key2'])
```

Out[39]:

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A2	B2	K1	K0	C1	D1
2	A2	B2	K1	K0	C2	D2

In [40]:
```
pd.merge(left, right, how='outer', on=['key1', 'key2'])
```

Out[40]:

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A1	B1	K0	K1	NaN	NaN
2	A2	B2	K1	K0	C1	D1
3	A2	B2	K1	K0	C2	D2
4	A3	B3	K2	K1	NaN	NaN
5	NaN	NaN	K2	K0	C3	D3

In [41]:
```
pd.merge(left, right, how='right', on=['key1', 'key2'])
```

Out[41]:

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A2	B2	K1	K0	C1	D1
2	A2	B2	K1	K0	C2	D2
3	NaN	NaN	K2	K0	C3	D3

In [42]:
```
pd.merge(left, right, how='left', on=['key1', 'key2'])
```

Out[42]:

	A	B	key1	key2	C	D
0	A0	B0	K0	K0	C0	D0
1	A1	B1	K0	K1	NaN	NaN
2	A2	B2	K1	K0	C1	D1
3	A2	B2	K1	K0	C2	D2
4	A3	B3	K2	K1	NaN	NaN

3. Joining

Joining is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame.

In [46]:
```
left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
                     'B': ['B0', 'B1', 'B2']},
                      index=['K0', 'K1', 'K2'])

right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
                    'D': ['D0', 'D2', 'D3']},
                      index=['K0', 'K2', 'K3'])
```

In [47]:
```
left.join(right)
```

Out[47]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

In [48]:
```
left.join(right, how='outer')
```

Out[48]:

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2
K3	NaN	NaN	C3	D3

VI. Operations

There are lots of operations with pandas that will be really useful to you, but don't fall into any distinct category. Let's show them here in this lecture:

In [5]:
```
import pandas as pd
df = pd.DataFrame({'col1':[1,2,3,4],'col2':[444,555,666,444],'col3':['abc',
df.head()
```

Out[5]:

	col1	col2	col3
0	1	444	abc
1	2	555	def
2	3	666	ghi
3	4	444	xyz

1. Info on Unique Values

In [53]:
```
df['col2'].unique()
```

Out[53]:
```
array([444, 555, 666])
```

In [54]:
```
df['col2'].nunique()
```

Out[54]:
```
3
```

In [55]:
```
df['col2'].value_counts()
```

Out[55]:
```
444    2
555    1
666    1
Name: col2, dtype: int64
```

2. Selecting Data

In [56]:
```
#Select from DataFrame using criteria from multiple columns
newdf = df[(df['col1']>2) & (df['col2']==444)]
```

In [57]:
```
newdf
```

Out[57]:

	col1	col2	col3
3	4	444	xyz

3. Applying Functions

In [58]:
```
def times2(x):
    return x*2
```

In [59]:
```
df['col1'].apply(times2)
```

Out[59]:
```
0    2
1    4
2    6
3    8
Name: col1, dtype: int64
```

In [60]:
```
df['col3'].apply(len)
```

Out[60]:
```
0    3
1    3
2    3
3    3
Name: col3, dtype: int64
```

In [61]:
```
df['col1'].sum()
```

Out[61]:
```
10
```

Permanently Removing a Column

In [62]:
```
del df['col1']
```

In [63]:
```
df
```

Out[63]:

	col2	col3
0	444	abc
1	555	def
2	666	ghi
3	444	xyz

Get column and index names:

In [64]:
```
df.columns
```

Out[64]: `Index(['col2', 'col3'], dtype='object')`

In [65]:
```
df.index
```

Out[65]: `RangeIndex(start=0, stop=4, step=1)`

Sorting and Ordering a DataFrame:

In [66]:
```
df
```

Out[66]:

	col2	col3
0	444	abc
1	555	def
2	666	ghi
3	444	xyz

In [67]:
```
df.sort_values(by='col2') #inplace=False by default
```

Out[67]:

	col2	col3
0	444	abc
3	444	xyz
1	555	def
2	666	ghi

Find Null Values or Check for Null Values

In [68]:
```
df.isnull()
```

Out[68]:

	col2	col3
0	False	False
1	False	False
2	False	False
3	False	False

In [69]:
```
# Drop rows with NaN Values
df.dropna()
```

Out[69]:

	col2	col3
0	444	abc
1	555	def
2	666	ghi
3	444	xyz

Filling in NaN values with something else:

In [3]:
```
import numpy as np
```

In [6]:
```
df = pd.DataFrame({'col1':[1,2,3,np.nan],
                   'col2':[np.nan,555,666,444],
                   'col3':['abc','def','ghi','xyz']})
df.head()
```

Out[6]:

	col1	col2	col3
0	1.0	NaN	abc
1	2.0	555.0	def
2	3.0	666.0	ghi
3	NaN	444.0	xyz

In [7]:
```
df.isnull()
```

Out[7]:

	col1	col2	col3
0	False	True	False
1	False	False	False
2	False	False	False
3	True	False	False

In [8]:
```
df.dropna()
```

Out[8]:

	col1	col2	col3
1	2.0	555.0	def
2	3.0	666.0	ghi

In [9]:
```
df.fillna('FILL')
```

Out[9]:

	col1	col2	col3
0	1	FILL	abc
1	2	555	def
2	3	666	ghi
3	FILL	444	xyz

In [89]:
```
data = {'A':['foo','foo','foo','bar','bar','bar'],
     'B':['one','one','two','two','one','one'],
       'C':['x','y','x','y','x','y'],
       'D':[1,3,2,5,4,1]}

df = pd.DataFrame(data)
```

In [90]:
```
df
```

Out[90]:

	A	B	C	D
0	foo	one	x	1
1	foo	one	y	3
2	foo	two	x	2
3	bar	two	y	5
4	bar	one	x	4
5	bar	one	y	1

In [91]:
```
df.pivot_table(values='D',index=['A', 'B'],columns=['C'])
```

Out[91]:

	C	x	y
A	B		
bar	one	4.0	1.0
	two	NaN	5.0
foo	one	1.0	3.0
	two	2.0	NaN

VII. Data Input and Output

This notebook is the reference code for getting input and output, pandas can read a variety of file types using its pd.read_ methods. Let's take a look at the most common data types:

In [1]:
```
import numpy as np
import pandas as pd
```

1. CSV

CSV Input

In [25]:
```
df = pd.read_csv('example.csv')
df
```

Out[25]:

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

CSV Output

In [24]:
```
df.to_csv('example.csv',index=False)
```

2. Excel

Pandas can read and write excel files, keep in mind, this only imports data. Not formulas or images, having images or macros may cause this read_excel method to crash.

Excel Input

In [35]:
```
pd.read_excel('Excel_Sample.xlsx',sheetname='Sheet1')
```

Out[35]:

	a	b	c	d
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Excel Output

In [33]:
```
df.to_excel('Excel_Sample.xlsx',sheet_name='Sheet1')
```

In [36]:
```
from sqlalchemy import create_engine
```

In [37]:
```
engine = create_engine('sqlite:///:memory:')
```

In [40]:
```
df.to_sql('data', engine)
```

In [42]:
```
sql_df = pd.read_sql('data',con=engine)
```

In [43]:
```
sql_df
```

Out[43]:

	index	a	b	c	d
0	0	0	1	2	3
1	1	4	5	6	7
2	2	8	9	10	11
3	3	12	13	14	15

VIII. Exersises

1. Sales

Fill in the TODO cells in sales.ipynb notebook.

- ✓ Fix column datatypes.
- ✓ Drop if duplicated or null.
- ✓ Sanity check for value ranges and to check assumptions.
- ✓ Use regular expression and lambda function to parse data.

2. Job Market

Given the job market data in csv file. Create your own jupyter notebook and explore the data by:

- ✓ Load the data using Pandas.
- ✓ Visualize top 10 first rows
- ✓ Fix column datatypes.
- ✓ Check and clean the data.