Introduction to Pandas

In this section of the course we will learn how to use pandas for data analysis. You can think of pandas as an extremely powerful version of Excel, with a lot more features. In this section of the course, you should go through the notebooks in this order:

- Introduction to Pandas
- Series
- DataFrames
- Missing Data
- GroupBy
- Merging,Joining,and Concatenating
- Operations
- Data Input and Output

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
```

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

```
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
```

Data in a 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
```

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
```

```
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!

Great Job!

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

In [2]:
```
from numpy.random import randn
np.random.seed(101)
```

In [3]:
```
df = pd.DataFrame(randn(5,4),index='A B C D E'.split(),columns='W X Y Z'.split())
```

In [4]:
```
df
```

Out[4]:

	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

Selection and Indexing

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

In [5]:
```
df['W']
```

Out[5]:
```
A    2.706850
B    0.651118
C   -2.018168
D    0.188695
E    0.190794
Name: W, dtype: float64
```

In [6]:
```
# Pass a list of column names
df[['W','Z']]
```

Out[6]:

	W	Z
A	2.706850	0.503826
B	0.651118	0.605965

	W	Z
C	-2.018168	-0.589001
D	0.188695	0.955057
E	0.190794	0.683509

In [7]:
```
# SQL Syntax (NOT RECOMMENDED!)
df.W
```

Out[7]:
```
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 [8]:
```
type(df['W'])
```

Out[8]:
```
pandas.core.series.Series
```

Creating a new column:

In [9]:
```
df['new'] = df['W'] + df['Y']
```

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

Out[10]:

	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 [11]:
```
df.drop('new',axis=1)
```

Out[11]:

	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 [12]:
```
# Not inplace unless specified!
df
```

Out[12]:

	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 [13]:
```
df.drop('new',axis=1,inplace=True)
```

In [14]:
```
df
```

Out[14]:

	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 [15]:
```
df.drop('E',axis=0)
```

Out[15]:

	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 [16]:
```
df.loc['A']
```

Out[16]:
```
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 [17]:
```
df.iloc[2]
```

Out[17]:
```
W   -2.018168
X    0.740122
Y    0.528813
Z   -0.589001
Name: C, dtype: float64
```

Selecting subset of rows and columns

In [18]:
```
df.loc['B','Y']
```

Out[18]:
```
-0.8480769834036315
```

In [19]:
```
df.loc[['A','B'],['W','Y']]
```

Out[19]:

	W	Y
A	2.706850	0.907969
B	0.651118	-0.848077

Conditional Selection

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

In [20]:
```
df
```

Out[20]:

	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 [21]:
```
df>0
```

Out[21]:

	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 [22]:
```
df[df>0]
```

Out[22]:

	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

Out[23]:

	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 [24]:
```
df[df['W']>0]['Y']
```

Out[24]:
```
A    0.907969
B   -0.848077
D   -0.933237
E    2.605967
Name: Y, dtype: float64
```

In [25]:
```
df[df['W']>0][['Y','X']]
```

Out[25]:

	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 [26]:
```
df[(df['W']>0) & (df['Y'] > 1)]
```

Out[26]:

	W	X	Y	Z
E	0.190794	1.978757	2.605967	0.683509

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 [27]:
```
df
```

Out[27]:

	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 [28]:
```
# Reset to default 0,1...n index
```

Out[28]:

	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 [29]:
```
newind = 'CA NY WY OR CO'.split()
```

In [30]:
```
df['States'] = newind
```

In [31]:
```
df
```

Out[31]:

	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 [32]:
```
df.set_index('States')
```

Out[32]:

	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 [33]:
```
df
```

Out[33]:

	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 [34]:
```
df.set_index('States',inplace=True)
```

In [35]:
```
df
```

Out[35]:

States	W	X	Y	Z
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

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 [36]:
```
# 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 [37]:
```
hier_index
```

Out[37]:
```
MultiIndex([('G1', 1),
            ('G1', 2),
            ('G1', 3),
            ('G2', 1),
            ('G2', 2),
            ('G2', 3)],
           )
```

In [38]:
```
df = pd.DataFrame(np.random.randn(6,2),index=hier_index,columns=['A','B'])
df
```

Out[38]:

		A	B
G1	1	0.302665	1.693723
	2	-1.706086	-1.159119
	3	-0.134841	0.390528
G2	1	0.166905	0.184502
	2	0.807706	0.072960
	3	0.638787	0.329646

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 [39]:
```
df.loc['G1']
```

Out[39]:

	A	B
1	0.302665	1.693723
2	-1.706086	-1.159119
3	-0.134841	0.390528

In [40]:
```
df.loc['G1'].loc[1]
```

Out[40]:
```
A    0.302665
B    1.693723
Name: 1, dtype: float64
```

In [41]:
```
df.index.names
```

Out[41]:
```
FrozenList([None, None])
```

In [42]:
```
df.index.names = ['Group','Num']
```

In [43]:
```
df
```

Out[43]:

		A	B
Group	**Num**		
G1	**1**	0.302665	1.693723
	2	-1.706086	-1.159119
	3	-0.134841	0.390528
G2	**1**	0.166905	0.184502
	2	0.807706	0.072960
	3	0.638787	0.329646

In [44]:
```
df.xs('G1')
```

Out[44]:

	A	B
Num		
1	0.302665	1.693723
2	-1.706086	-1.159119
3	-0.134841	0.390528

In [45]:
```
df.xs(['G1',1])
```

Out[45]:
```
A    0.302665
B    1.693723
Name: (G1, 1), dtype: float64
```

In [46]:
```
df.xs(1,level='Num')
```

Out[46]:

	A	B

Group	A	B

Group		
G1	0.302665	1.693723
G2	0.166905	0.184502

Great Job!

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

In [3]:
```
df
```

Out[3]:

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

In [4]:
```
df.dropna()
```

Out[4]:

	A	B	C
0	1.0	5.0	1

In [5]:
```
df.dropna(axis=1)
```

Out[5]:

	C
0	1
1	2
2	3

In [6]:
```
df.dropna(thresh=2)
```

Out[6]:

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2

In [7]:
```
df.fillna(value='FILL VALUE')
```

Out[7]:

	A	B	C

	A	B	C
0	1.0	5.0	1
1	2.0	FILL VALUE	2
2	FILL VALUE	FILL VALUE	3

In [8]:
```
df['A'].fillna(value=df['A'].mean())
```

Out[8]:
```
0    1.0
1    2.0
2    1.5
Name: A, dtype: float64
```

Great Job!

Groupby

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

In [1]:
```
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 [2]:
```
df = pd.DataFrame(data)
```

In [3]:
```
df
```

Out[3]:

	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 [4]:
```
df.groupby('Company')
```

Out[4]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000002527C2E97F0>
```

You can save this object as a new variable:

In [5]:
```
by_comp = df.groupby("Company")
```

And then call aggregate methods off the object:

In [6]:
```
by_comp.mean()
```

Out[6]:

	Sales
Company	
FB	296.5
GOOG	160.0

Company	Sales
MSFT	232.0

In [7]:
```
df.groupby('Company').mean()
```

Out[7]:

Company	Sales
FB	296.5
GOOG	160.0
MSFT	232.0

More examples of aggregate methods:

In [8]:
```
by_comp.std()
```

Out[8]:

Company	Sales
FB	75.660426
GOOG	56.568542
MSFT	152.735065

In [9]:
```
by_comp.min()
```

Out[9]:

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

In [10]:
```
by_comp.max()
```

Out[10]:

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

In [11]:
```
by_comp.count()
```

Out[11]:

Company	Person	Sales

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

In [12]:
```
by_comp.describe()
```

Out[12]:

	Sales							
	count	mean	std	min	25%	50%	75%	max
Company								
FB	2.0	296.5	75.660426	243.0	269.75	296.5	323.25	350.0
GOOG	2.0	160.0	56.568542	120.0	140.00	160.0	180.00	200.0
MSFT	2.0	232.0	152.735065	124.0	178.00	232.0	286.00	340.0

In [13]:
```
by_comp.describe().transpose()
```

Out[13]:

	Company	FB	GOOG	MSFT
Sales	count	2.000000	2.000000	2.000000
	mean	296.500000	160.000000	232.000000
	std	75.660426	56.568542	152.735065
	min	243.000000	120.000000	124.000000
	25%	269.750000	140.000000	178.000000
	50%	296.500000	160.000000	232.000000
	75%	323.250000	180.000000	286.000000
	max	350.000000	200.000000	340.000000

In [14]:
```
by_comp.describe().transpose()['GOOG']
```

Out[14]:
```
Sales  count      2.000000
       mean     160.000000
       std       56.568542
       min      120.000000
       25%      140.000000
       50%      160.000000
       75%      180.000000
       max      200.000000
Name: GOOG, dtype: float64
```

Great Job!

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.

Example DataFrames

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

In [2]:
```
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 [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])
```

In [4]:
```
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 [5]:
```
df1
```

Out[5]:

	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 [6]:
```
df2
```

Out[6]:

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6

	A	B	C	D
7	A7	B7	C7	D7

In [7]:
```
df3
```

Out[7]:

	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

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 [8]:
```
pd.concat([df1,df2,df3])
```

Out[8]:

	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 [9]:
```
pd.concat([df1,df2,df3],axis=1)
```

Out[9]:

	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
					NaN	A5	B5	C5	D5	NaN	NaN	NaN	NaN

	A	B	C	D	A	B	C	D	A	B	C	D
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

Example DataFrames

In [10]:
```
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 [11]:
```
left
```

Out[11]:

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

In [12]:
```
right
```

Out[12]:

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

Merging

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

In [13]:
```
pd.merge(left,right,how='inner',on='key')
```

Out[13]:

	key	A	B	C	D

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

Or to show a more complicated example:

In [14]:
```
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 [15]:
```
pd.merge(left, right, on=['key1', 'key2'])
```

Out[15]:

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

In [16]:
```
pd.merge(left, right, how='outer', on=['key1', 'key2'])
```

Out[16]:

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

In [17]:
```
pd.merge(left, right, how='right', on=['key1', 'key2'])
```

Out[17]:

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

In [18]:
```
pd.merge(left, right, how='left', on=['key1', 'key2'])
```

Out[18]:

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

Joining

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

In [19]:
```
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 [20]:
```
left.join(right)
```

Out[20]:

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

In [21]:
```
left.join(right, how='outer')
```

Out[21]:

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

Great Job!

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

Out[1]:

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

Info on Unique Values

In [2]:
```
df['col2'].unique()
```

Out[2]:
```
array([444, 555, 666], dtype=int64)
```

In [3]:
```
df['col2'].nunique()
```

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

In [4]:
```
df['col2'].value_counts()
```

Out[4]:
```
444    2
666    1
555    1
Name: col2, dtype: int64
```

Selecting Data

In [5]:
```
#Select from DataFrame using criteria from multiple columns
newdf = df[(df['col1']>2) & (df['col2']==444)]
```

In [6]:
```
newdf
```

Out[6]:

	col1	col2	col3
3	4	444	xyz

Applying Functions

In [7]:
```
def times2(x):
    return x*2
```

In [8]:
```
df['col1'].apply(times2)
```

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

In [9]:
```
df['col3'].apply(len)
```

Out[9]:
```
0    3
1    3
2    3
3    3
Name: col3, dtype: int64
```

In [10]:
```
df['col1'].sum()
```

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

Permanently Removing a Column

In [11]:
```
del df['col1']
```

In [12]:
```
df
```

Out[12]:

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

Get column and index names:

In [13]:
```
df.columns
```

Out[13]:
```
Index(['col2', 'col3'], dtype='object')
```

In [14]:
```
df.index
```

Out[14]:
```
RangeIndex(start=0, stop=4, step=1)
```

Sorting and Ordering a DataFrame:

In [15]:
```
df
```

Out[15]:

	col2	col3

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

In [16]:
```
df.sort_values(by='col2') #inplace=False by default
```

Out[16]:

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

Find Null Values or Check for Null Values

In [17]:
```
df.isnull()
```

Out[17]:

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

In [18]:
```
# Drop rows with NaN Values
df.dropna()
```

Out[18]:

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

Filling in NaN values with something else:

In [19]:
```
import numpy as np
```

In [20]:
```
df = pd.DataFrame({'col1':[1,2,3,np.nan],
                   'col2':[np.nan,555,666,444],
                   'col3':['abc','def','ghi','xyz']})
df.head()
```

Out[20]:

	col1	col2	col3
0	1.0	NaN	abc

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

In [21]:
```
df.fillna('FILL')
```

Out[21]:

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

In [22]:
```
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 [23]:
```
df
```

Out[23]:

	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 [24]:
```
df.pivot_table(values='D',index=['A', 'B'],columns=['C'])
```

Out[24]:

	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

Great Job!

SF Salaries Exercise - Solutions

Welcome to a quick exercise for you to practice your pandas skills! We will be using the SF Salaries Dataset from Kaggle! Just follow along and complete the tasks outlined in bold below. The tasks will get harder and harder as you go along.

Import pandas as pd.

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

Read Salaries.csv as a dataframe called sal.

In [2]:
```
sal = pd.read_csv('Salaries.csv')
```

Check the head of the DataFrame.

In [3]:
```
sal.head()
```

Out[3]:

	Id	EmployeeName	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay	Totall
0	1	NATHANIEL FORD	GENERAL MANAGER-METROPOLITAN TRANSIT AUTHORITY	167411.18	0.00	400184.25	NaN	567595.43	
1	2	GARY JIMENEZ	CAPTAIN III (POLICE DEPARTMENT)	155966.02	245131.88	137811.38	NaN	538909.28	
2	3	ALBERT PARDINI	CAPTAIN III (POLICE DEPARTMENT)	212739.13	106088.18	16452.60	NaN	335279.91	
3	4	CHRISTOPHER CHONG	WIRE ROPE CABLE MAINTENANCE MECHANIC	77916.00	56120.71	198306.90	NaN	332343.61	
4	5	PATRICK GARDNER	DEPUTY CHIEF OF DEPARTMENT, (FIRE DEPARTMENT)	134401.60	9737.00	182234.59	NaN	326373.19	

Use the .info() method to find out how many entries there are.

In [4]:
```
sal.info() # 148654 Entries
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 148654 entries, 0 to 148653
Data columns (total 13 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0   Id                148654 non-null  int64
 1   EmployeeName      148654 non-null  object
```

```
 2   JobTitle          148654 non-null  object
 3   BasePay           148045 non-null  float64
 4   OvertimePay       148650 non-null  float64
 5   OtherPay          148650 non-null  float64
 6   Benefits          112491 non-null  float64
 7   TotalPay          148654 non-null  float64
 8   TotalPayBenefits  148654 non-null  float64
 9   Year              148654 non-null  int64
 10  Notes             0 non-null       float64
 11  Agency            148654 non-null  object
 12  Status            0 non-null       float64
dtypes: float64(8), int64(2), object(3)
memory usage: 14.7+ MB
```

What is the average BasePay ?

In [5]:
```
sal['BasePay'].mean()
```

Out[5]:
```
66325.44884050643
```

What is the highest amount of OvertimePay in the dataset ?

In [6]:
```
sal['OvertimePay'].max()
```

Out[6]:
```
245131.88
```

What is the job title of JOSEPH DRISCOLL ? Note: Use all caps, otherwise you may get an answer that doesn't match up (there is also a lowercase Joseph Driscoll).

In [7]:
```
sal[sal['EmployeeName']=='JOSEPH DRISCOLL']['JobTitle']
```

Out[7]:
```
24    CAPTAIN, FIRE SUPPRESSION
Name: JobTitle, dtype: object
```

How much does JOSEPH DRISCOLL make (including benefits)?

In [8]:
```
sal[sal['EmployeeName']=='JOSEPH DRISCOLL']['TotalPayBenefits']
```

Out[8]:
```
24    270324.91
Name: TotalPayBenefits, dtype: float64
```

What is the name of highest paid person (including benefits)?

In [9]:
```
sal[sal['TotalPayBenefits']== sal['TotalPayBenefits'].max()] #['EmployeeName']
# or
# sal.loc[sal['TotalPayBenefits'].idxmax()]
```

Out[9]:

	Id	EmployeeName	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay	Totall
0	1	NATHANIEL FORD	GENERAL MANAGER-METROPOLITAN TRANSIT AUTHORITY	167411.18	0.0	400184.25	NaN	567595.43	

What is the name of lowest paid person (including benefits)? Do you notice something strange about how much he or she is paid?

In [10]:
```
sal[sal['TotalPayBenefits']== sal['TotalPayBenefits'].min()] #['EmployeeName']
# or
```

```
# sal.loc[sal['TotalPayBenefits'].idxmax()]['EmployeeName']

## ITS NEGATIVE!! VERY STRANGE
```

Out[10]:

	Id	EmployeeName	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay	To
148653	148654	Joe Lopez	Counselor, Log Cabin Ranch	0.0	0.0	-618.13	0.0	-618.13	

What was the average (mean) BasePay of all employees per year? (2011-2014) ?

In [11]:
```
sal.groupby('Year').mean()['BasePay']
```

Out[11]:
```
Year
2011    63595.956517
2012    65436.406857
2013    69630.030216
2014    66564.421924
Name: BasePay, dtype: float64
```

How many unique job titles are there?

In [12]:
```
sal['JobTitle'].nunique()
```

Out[12]:
```
2159
```

What are the top 5 most common jobs?

In [13]:
```
sal['JobTitle'].value_counts().head(5)
```

Out[13]:
```
Transit Operator                7036
Special Nurse                   4389
Registered Nurse                3736
Public Svc Aide-Public Works    2518
Police Officer 3                2421
Name: JobTitle, dtype: int64
```

How many Job Titles were represented by only one person in 2013? (e.g. Job Titles with only one occurence in 2013?)

In [14]:
```
sum(sal[sal['Year']==2013]['JobTitle'].value_counts() == 1) # pretty tricky way to do thi
```

Out[14]:
```
202
```

How many people have the word Chief in their job title? (This is pretty tricky)

In [15]:
```
def chief_string(title):
    if 'chief' in title.lower():
        return True
    else:
        return False
```

In [16]:
```
sum(sal['JobTitle'].apply(lambda x: chief_string(x)))
```

Out[16]:
```
627
```

Bonus: Is there a correlation between length of the Job Title string and Salary?

In [17]:
```
sal['title_len'] = sal['JobTitle'].apply(len)
```

In [18]:
```
sal['title_len']
```

Out[18]:
```
0         46
1         31
2         31
3         36
4         44
          ..
148649     9
148650    12
148651    12
148652    12
148653    26
Name: title_len, Length: 148654, dtype: int64
```

In [19]:
```
sal[['title_len','TotalPayBenefits']].corr() # No correlation.
```

Out[19]:

	title_len	TotalPayBenefits
title_len	1.000000	-0.036878
TotalPayBenefits	-0.036878	1.000000

Great Job!

Ecommerce Purchases Exercise - Solutions

In this Exercise you will be given some Fake Data about some purchases done through Amazon! Just go ahead and follow the directions and try your best to answer the questions and complete the tasks. Feel free to reference the solutions. Most of the tasks can be solved in different ways. For the most part, the questions get progressively harder.

Please excuse anything that doesn't make "Real-World" sense in the dataframe, all the data is fake and made-up.

Also note that all of these questions can be answered with one line of code.

Import pandas and read in the Ecommerce Purchases csv file and set it to a DataFrame called ecom.

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

In [2]:
```
ecom = pd.read_csv('Ecommerce Purchases')
```

Check the head of the DataFrame.

In [3]:
```
ecom.head()
```

Out[3]:

	Address	Lot	AM or PM	Browser Info	Company	Credit Card	CC Exp Date	CC Security Code	CC Provider
0	16629 Pace Camp Apt. 448\nAlexisborough, NE 77...	46 in	PM	Opera/9.56. (X11; Linux x86_64; sl-SI) Presto/2...	Martinez-Herman	6011929061123406	02/20	900	JCB 16 digit
1	9374 Jasmine Spurs Suite 508\nSouth John, TN 8...	28 rn	PM	Opera/8.93. (Windows 98; Win 9x 4.90; en-US) Pr...	Fletcher, Richards and Whitaker	3337758169645356	11/18	561	Mastercard
2	Unit 0065 Box 5052\nDPO AP 27450	94 vE	PM	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT ...	Simpson, Williams and Pham	675957666125	08/19	699	JCB 16 digit
3	7780 Julia Fords\nNew Stacy, WA 45798	36 vm	PM	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_0 ...	Williams, Marshall and Buchanan	6011578504430710	02/24	384	Discover

	Address	Lot	AM or PM	Browser Info	Company	Credit Card	CC Exp Date	CC Security Code	CC Provider
4	23012 Munoz Drive Suite 337\nNew Cynthia, TX 5...	20 IE	AM	Opera/9.58.(X11; Linux x86_64; it-IT) Presto/2...	Brown, Watson and Andrews	6011456623207998	10/25	678	Diners Club / Carte Blanche

How many rows and columns are there?

In [4]:
```
ecom.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Address           10000 non-null  object 
 1   Lot               10000 non-null  object 
 2   AM or PM          10000 non-null  object 
 3   Browser Info      10000 non-null  object 
 4   Company           10000 non-null  object 
 5   Credit Card       10000 non-null  int64  
 6   CC Exp Date       10000 non-null  object 
 7   CC Security Code  10000 non-null  int64  
 8   CC Provider       10000 non-null  object 
 9   Email             10000 non-null  object 
 10  Job               10000 non-null  object 
 11  IP Address        10000 non-null  object 
 12  Language          10000 non-null  object 
 13  Purchase Price    10000 non-null  float64
dtypes: float64(1), int64(2), object(11)
memory usage: 1.1+ MB
```

What is the average Purchase Price?

In [5]:
```
ecom['Purchase Price'].mean()
```

Out[5]:
```
50.34730200000025
```

What were the highest and lowest purchase prices?

In [6]:
```
ecom['Purchase Price'].max()
```

Out[6]:
```
99.99
```

In [7]:
```
ecom['Purchase Price'].min()
```

Out[7]:
```
0.0
```

How many people have English 'en' as their Language of choice on the website?

In [8]:
```
ecom[ecom['Language']=='en'].count()
```

Out[8]:
```
Address             1098
Lot                 1098
AM or PM            1098
Browser Info        1098
Company             1098
```

```
Credit Card         1098
CC Exp Date         1098
CC Security Code    1098
CC Provider         1098
Email               1098
Job                 1098
IP Address          1098
Language            1098
Purchase Price      1098
dtype: int64
```

How many people have the job title of "Lawyer" ?

In [9]:
```
ecom[ecom['Job'] == 'Lawyer'].info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 30 entries, 470 to 9979
Data columns (total 14 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Address           30 non-null     object 
 1   Lot               30 non-null     object 
 2   AM or PM          30 non-null     object 
 3   Browser Info      30 non-null     object 
 4   Company           30 non-null     object 
 5   Credit Card       30 non-null     int64  
 6   CC Exp Date       30 non-null     object 
 7   CC Security Code  30 non-null     int64  
 8   CC Provider       30 non-null     object 
 9   Email             30 non-null     object 
 10  Job               30 non-null     object 
 11  IP Address        30 non-null     object 
 12  Language          30 non-null     object 
 13  Purchase Price    30 non-null     float64
dtypes: float64(1), int64(2), object(11)
memory usage: 3.5+ KB
```

How many people made the purchase during the AM and how many people made the purchase during PM ?

(Hint: Check out value_counts())

In [10]:
```
ecom['AM or PM'].value_counts()
```

Out[10]:
```
PM    5068
AM    4932
Name: AM or PM, dtype: int64
```

What are the 5 most common Job Titles?

In [11]:
```
ecom['Job'].value_counts().head(5)
```

Out[11]:
```
Interior and spatial designer        31
Lawyer                               30
Social researcher                    28
Research officer, political party    27
Designer, jewellery                  27
Name: Job, dtype: int64
```

Someone made a purchase that came from Lot: "90 WT" , what was the Purchase Price for this transaction?

In [12]:
```
ecom[ecom['Lot']=='90 WT']['Purchase Price']
```

Out[12]:
```
513    75.1
Name: Purchase Price, dtype: float64
```

What is the email of the person with the following Credit Card Number: 4926535242672853

In [13]:
```
ecom[ecom["Credit Card"] == 4926535242672853]['Email']
```

Out[13]:
```
1234    bondellen@williams-garza.com
Name: Email, dtype: object
```

How many people have American Express as their Credit Card Provider *and* made a purchase above $95 ?

In [14]:
```
ecom[(ecom['CC Provider']=='American Express') & (ecom['Purchase Price']>95)].count()
```

Out[14]:
```
Address             39
Lot                 39
AM or PM            39
Browser Info        39
Company             39
Credit Card         39
CC Exp Date         39
CC Security Code    39
CC Provider         39
Email               39
Job                 39
IP Address          39
Language            39
Purchase Price      39
dtype: int64
```

Hard: How many people have a credit card that expires in 2025?

In [15]:
```
sum(ecom['CC Exp Date'].apply(lambda x: x[3:]) == '25')
```

Out[15]:
```
1033
```

Hard: What are the top 5 most popular email providers/hosts (e.g. gmail.com, yahoo.com, etc...)

In [16]:
```
ecom['Email'].apply(lambda x: x.split('@')[1]).value_counts().head(5)
```

Out[16]:
```
hotmail.com     1638
yahoo.com       1616
gmail.com       1605
smith.com         42
williams.com      37
Name: Email, dtype: int64
```

Great Job!