Introduction to Pandas

In this section we learn how to use pandas for data analysis. Pandas is like a powerful version of Excel that you can use via python and provides many more features.

* Pandas is built on top of numpy
* It can load data from a variety of sources
* It excels in data cleaning and preparation
* It includes visualization tools

Be sure to check that `pandas` is installed in your environment (Anaconda Navigator)

We will see:

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

Pandas Series

- A `Series` object is similar to a numpy array that also provides an axis label that can be used to index data.
- Differently from numpy arrays, Series can hold any kind of python data and not just numbers.

In [1]:

```
import numpy as np
import pandas as pd
```

Creating Series

By converting other types (`list` , `numpy array` , `dictionary`)

In [2]:

```
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 [3]:

```
pd.Series(data=my_list)
```

Out[3]:

```
0    10
1    20
2    30
dtype: int64
```

In [4]:

```
pd.Series(data=my_list, index=labels)
```

Out[4]:

```
a    10
b    20
c    30
dtype: int64
```

In [5]:

```
pd.Series(my_list, labels)
```

Out[5]:

```
a    10
b    20
c    30
dtype: int64
```

NumPy Arrays

In [6]:

```
pd.Series(arr)
```

Out[6]:

```
0    10
1    20
2    30
dtype: int64
```

In [7]:

```
pd.Series(arr, labels)
```

Out[7]:

```
a    10
b    20
c    30
dtype: int64
```

Dictionary

In [8]:

```
pd.Series(d)
```

Out[8]:

```
a    10
b    20
c    30
dtype: int64
```

Data in a Series

Series can hold a variety of object types:

In [9]:

```
pd.Series(data=['alpha', 'beta', 'gamma'])
```

Out[9]:

```
0    alpha
1     beta
2    gamma
dtype: object
```

Using the index

Pandas makes use of the index as a way to identify information (lookup, operations).

In [10]:

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

In [11]:

```
ser1
```

Out[11]:

```
USA        1
Germany    2
USSR       3
Japan      4
dtype: int64
```

In [12]:

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

In [13]:

```
ser2
```

Out[13]:

```
USA        1
Germany    2
Italy      5
Japan      4
dtype: int64
```

In [14]:

```
ser1['USA']
```

Out[14]:

```
1
```

Operations are **based on the index**:

In [15]:

```
ser1 + ser2
```

Out[15]:

```
Germany    4.0
Italy      NaN
Japan      8.0
USA        2.0
USSR       NaN
dtype: float64
```

If you want to specify another default, you can user the `Series.add` method and use the `fill_value` parameter.

In [16]:

```
ser1.add(ser2, fill_value=10)
```

Out[16]:

```
Germany     4.0
Italy      15.0
Japan       8.0
USA         2.0
USSR       13.0
dtype: float64
```

Pandas DataFrames

- DataFrames are pandas most powerful datatypes
- They are inspired by R
- They look like multiple Series objects put together under a same index

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

In [2]:
```
from numpy.random import randn
np.random.seed(101)    # we can fix the generation of random numbers
```

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

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

How to grab data from a DataFrame (similarly to numpy)

In [5]:
```
# a single column
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]:
```
# SQL Syntax (not recommended as you might get confused with pandas methods!)
df.W
```

Out[6]:
```
A    2.706850
B    0.651118
C   -2.018168
D    0.188695
E    0.190794
Name: W, dtype: float64
```

DataFrame columns are `Series`

In [7]:
```
type(df['W'])
```

Out[7]:
```
pandas.core.series.Series
```

In [8]:
```python
# A list of column names
df[['W','Z']]
```

Out[8]:

	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

Create a new column:

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

In [10]:
```python
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

Remove a column

In [11]:
```python
# we need to specify axis=1 to delete columns (by default it deletes rows)
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]:
```python
# The object itself is not changed 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]:
```python
# an operation that changes the original data is called inplace
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

Delete row (by index):

```
In [15]: df.drop('E', axis=0)  # the parameter axis=0 is optional for rows.
         # if confused by the axis, you can use df.shape (like in numpy).
```

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

Select Rows

We can specify the index with `loc`

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

Out[16]: W    2.706850
         X    0.628133
         Y    0.907969
         Z    0.503826
         Name: A, dtype: float64
```

or the position with `iloc`

```
In [17]: df.iloc[2]

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

Select a subset of rows and columns:

```
In [18]: df.loc['B','Y']  # loc uses the same syntax as numpy

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

Similar to conditional selection in 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

In [23]:
```
df[df['W']>0]
```

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

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 index 0, 1, ..., n index
df.reset_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]:
```python
df.set_index('States')
```

Out[32]:

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

In [33]:
```python
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]:
```python
df.set_index('States', inplace=True)
```

In [35]:
```python
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

In [36]:
```python
# 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]:
```python
hier_index
```

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

In [38]:
```
data = np.random.randn(6, 2)
df = pd.DataFrame(data, 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

How to index this?

We use `df.loc[]` (if the hierarchical index is on the rows, and `df[]` if it was on the columns).

Indexing one level of the hierarchical dataframe 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]:
```
# We can add names to the index
df.index.names = ['Group', 'Num']
```

In [42]:
```
df
```

Out[42]:

		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 [43]:
```
# cross section (xs) allows to index on nested levels
df.xs(1, level='Num')
```

Out[43]:

	A	B
Group		
G1	0.302665	1.693723
G2	0.166905	0.184502

Missing Data

Methods to deal with missing data

In [44]:
```
df = pd.DataFrame({'A': [1, 2, np.nan],
                   'B': [5, np.nan, np.nan],
                   'C': [1, 2, 3]})
```

In [45]:
```
df
```

Out[45]:

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

In [46]:
```
# drop rows that contain NaN
df.dropna()
```

Out[46]:

	A	B	C
0	1.0	5.0	1

In [47]:
```
# drop columns that contain NaN
df.dropna(axis=1)
```

Out[47]:

	C
0	1
1	2
2	3

In [48]:
```
# we can specify a threshold of NaN to drop the row
df.dropna(thresh=2)
```

Out[48]:

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2

In [49]:
```
# or we can change the NaN with some default value
df.fillna(value='FILL VALUE')
```

Out[49]:

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

In [50]:
```
# we can fill the value with the mean of a column
df['A'].fillna(value=df['A'].mean())
```

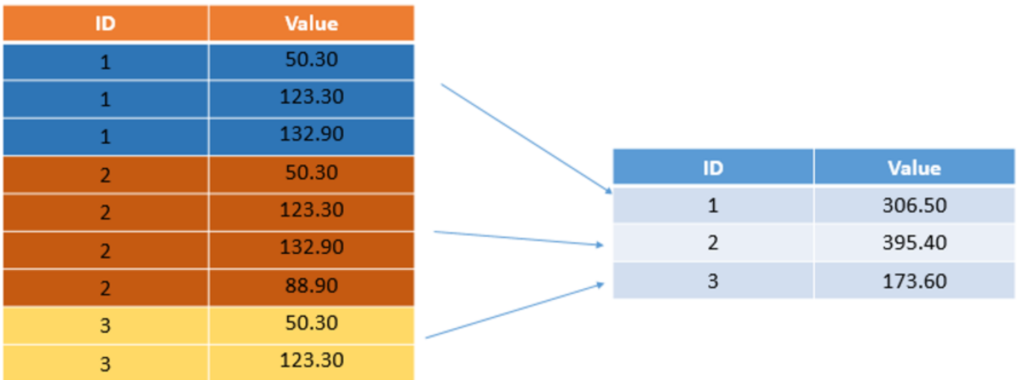
Out[50]:
```
0    1.0
1    2.0
2    1.5
Name: A, dtype: float64
```

Pandas DataFrame Operations

Groupby

GroupBy method can be used to group together rows based off of a column and perform an aggregate function on them.

In the example below, there are three partitions of IDS (1, 2, and 3) and several values for them. We can now group by the ID column and aggregate them using some sort of aggregate function. Here we are sum-ing the values and putting the values.



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

In [2]:
```
# Create sales dataframe
sales_data = {'Company': ['GOOG', 'GOOG', 'MSFT', 'MSFT', 'FB', 'FB'],
              'Person': ['Sam', 'Charlie', 'Amy', 'Vanessa', 'Carl', 'Sarah'],
              'Sales': [200, 120, 340, 124, 243, 350]}
```

In [3]:
```
sales = pd.DataFrame(sales_data)
sales
```

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

We can use `.groupby()` to group rows together based off of a column name. Let's group based off of `Company`.

This will create a `DataFrameGroupBy` object:

In [4]:
```
sales.groupby('Company')
```

Out[4]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x11bbb6d50>
```

We can save as a new variable:

In [5]:
```
sales_by_comp = sales.groupby("Company")
```

And then call aggregate methods:

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

Out[6]:

Company	Sales
FB	296.5
GOOG	160.0
MSFT	232.0

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

Out[7]:

Company	Sales
FB	296.5
GOOG	160.0
MSFT	232.0

Other aggregate methods:

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

Out[8]:

Company	Sales
FB	75.660426
GOOG	56.568542
MSFT	152.735065

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

Out[9]:

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

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

Out[10]:

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

In [11]:
```
sales_by_comp.max().loc['FB']
```

Out[11]:
```
Person     Sarah
Sales        350
Name: FB, dtype: object
```

In [12]:
```
sales_by_comp.count()
```

Out[12]:

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

In [13]:
```
# returns count, mean, std, min, max, and quartiles
sales_by_comp.describe()
```

Out[13]:

	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 [14]:
```
# we can also transpose it to have each company as a column.
sales_by_comp.describe().transpose()
```

Out[14]:

	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 [15]:
```
sales_by_comp.describe().transpose()['GOOG']
```

Out[15]:
```
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
```

Concatenating, Merging, and Joining

There are 3 ways of combining DataFrames together: `Concatenating` , `Merging` , and `Joining`

Example DataFrames

In [16]:
```
conc_df1 = pd.DataFrame({'A': range(10, 13),
                         'B': range(20, 23),
                         'C': range(30, 33)},
                         index=[0, 1, 2])
conc_df1
```

Out[16]:

	A	B	C
0	10	20	30
1	11	21	31
2	12	22	32

In [17]:
```
conc_df2 = pd.DataFrame({'A': range(13, 16),
                         'B': range(23, 26),
                         'C': range(33, 36)},
                         index=[4, 5, 6])
conc_df2
```

Out[17]:

	A	B	C
4	13	23	33
5	14	24	34
6	15	25	35

Concatenation

- Concatenation glues together DataFrames.
- Dimensions should match along the axis you are concatenating on.
- Use `pd.concat` and pass a **list** of DataFrames to concatenate together:

In [18]:
```
pd.concat([conc_df1, conc_df2])
```

Out[18]:

	A	B	C
0	10	20	30
1	11	21	31
2	12	22	32
4	13	23	33
5	14	24	34
6	15	25	35

In [19]:
```
# we can concatenate based on rows (but they do not match)
pd.concat([conc_df1, conc_df2], axis=1)
```

Out[19]:

	A	B	C	A	B	C
0	10.0	20.0	30.0	NaN	NaN	NaN
1	11.0	21.0	31.0	NaN	NaN	NaN
2	12.0	22.0	32.0	NaN	NaN	NaN
4	NaN	NaN	NaN	13.0	23.0	33.0
5	NaN	NaN	NaN	14.0	24.0	34.0
6	NaN	NaN	NaN	15.0	25.0	35.0

Example DataFrames

In [20]:
```
merge_left = pd.DataFrame({'key': ['a', 'b', 'c', 'd'],
                           'A': [10, 20, 30, 40],
                           'B': [100, 200, 300, 400]})
merge_left
```

Out[20]:

	key	A	B
0	a	10	100
1	b	20	200
2	c	30	300
3	d	40	400

In [21]:
```
merge_right = pd.DataFrame({'key': ['a', 'b', 'c', 'e'],
                            'C': [50, 60, 70, 80],
                            'D': [500, 600, 700, 800]})
merge_right
```

Out[21]:

	key	C	D
0	a	50	500
1	b	60	600
2	c	70	700
3	e	80	800

Merging

The **merge** function allows you to merge DataFrames together using a similar logic as joining SQL Tables together.

In [22]:
```
pd.merge(merge_left, merge_right, on='key')
```

Out[22]:

	key	A	B	C	D
0	a	10	100	50	500
1	b	20	200	60	600
2	c	30	300	70	700

In [23]:
```
pd.merge(merge_left, merge_right, on='key', how='left')
```

Out[23]:

	key	A	B	C	D
0	a	10	100	50.0	500.0
1	b	20	200	60.0	600.0
2	c	30	300	70.0	700.0
3	d	40	400	NaN	NaN

In [24]:
```
pd.merge(merge_left, merge_right, on='key', how='right')
```

Out[24]:

	key	A	B	C	D
0	a	10.0	100.0	50	500
1	b	20.0	200.0	60	600
2	c	30.0	300.0	70	700
3	e	NaN	NaN	80	800

In [25]:
```
pd.merge(merge_left, merge_right, on='key', how='outer')
```
Out[25]:

	key	A	B	C	D
0	a	10.0	100.0	50.0	500.0
1	b	20.0	200.0	60.0	600.0
2	c	30.0	300.0	70.0	700.0
3	d	40.0	400.0	NaN	NaN
4	e	NaN	NaN	80.0	800.0

Joining

Joining is similar to merge but uses the dataframe index.

In [26]:
```
join_left = merge_left.set_index('key')
join_right = merge_right.set_index('key')
```
In [27]:
```
join_left
```
Out[27]:

key	A	B
a	10	100
b	20	200
c	30	300
d	40	400

In [28]:
```
join_right
```
Out[28]:

key	C	D
a	50	500
b	60	600
c	70	700
e	80	800

In [29]:
```
join_left.join(join_right).dropna()
```
Out[29]:

key	A	B	C	D
a	10	100	50.0	500.0
b	20	200	60.0	600.0
c	30	300	70.0	700.0

Operations

In [30]:
```
data = pd.DataFrame({'col1': [1, 2, 3, 4],
                     'col2': [40, 50, 60, 40],
                     'col3': ['a', 'b', 'c', 'd']})
```

In [31]:
```
# returns the first n rows
data.head(2)
```

Out[31]:

	col1	col2	col3
0	1	40	a
1	2	50	b

Unique Values

In [32]:
```
# unique values
data['col2'].unique()
```

Out[32]:
```
array([40, 50, 60])
```

In [33]:
```
# number of unique values
data['col2'].nunique()
```

Out[33]:
```
3
```

In [34]:
```
# values and their counts
data['col2'].value_counts()
```

Out[34]:
```
40    2
60    1
50    1
Name: col2, dtype: int64
```

Applying Functions

In [35]:
```
def times2(x):
    return x * 2
```

In [36]:
```
data['col1'].apply(times2)
```

Out[36]:
```
0    2
1    4
2    6
3    8
Name: col1, dtype: int64
```

In [37]:
```
data['col3'].apply(len)
```

Out[37]:
```
0    1
1    1
2    1
3    1
Name: col3, dtype: int64
```

In [38]:
```
data['col1'].sum()
```

Out[38]:
```
10
```

Get column and index names:

In [39]:
```
data.columns.values
```

Out[39]:
```
array(['col1', 'col2', 'col3'], dtype=object)
```

In [40]:
```
data.index.values
```

Out[40]:
```
array([0, 1, 2, 3])
```

Sorting a DataFrame:

In [41]:
```
data
```
Out[41]:

	col1	col2	col3
0	1	40	a
1	2	50	b
2	3	60	c
3	4	40	d

In [42]:
```
data.sort_values('col2')
```
Out[42]:

	col1	col2	col3
0	1	40	a
3	4	40	d
1	2	50	b
2	3	60	c

In [43]:
```
piv_data = {'ind1': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'],
            'ind2': ['one', 'one', 'two', 'two', 'one', 'one'],
            'cols': ['x', 'y', 'x', 'y', 'x', 'y'],
            'vals': [1, 3, 2, 5, 4, 1]}

piv = pd.DataFrame(piv_data)
```
In [44]:
```
piv
```
Out[44]:

	ind1	ind2	cols	vals
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 [45]:
```
# We can create a pivot table by specifying the values, index, and columns
piv.pivot_table(values='vals', index=['ind1','ind2'], columns=['cols'])
```
Out[45]:

	cols	x	y
ind1	ind2		
bar	one	4.0	1.0
	two	NaN	5.0
foo	one	1.0	3.0
	two	2.0	NaN

Data Input and Output

Pandas can read and write a variety of file types using `pd.read_` and `pd.write_` methods.

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

CSV

CSV Input

In [2]:
```
# this can be a local file but also an url to a csv
banks_url = 'https://www.fdic.gov/bank/individual/failed/banklist.csv'
banks = pd.read_csv(banks_url)
banks
```

Out[2]:

	Bank Name	City	ST	CERT	Acquiring Institution	Closing Date
0	City National Bank of New Jersey	Newark	NJ	21111	Industrial Bank	1-Nov-19
1	Resolute Bank	Maumee	OH	58317	Buckeye State Bank	25-Oct-19
2	Louisa Community Bank	Louisa	KY	58112	Kentucky Farmers Bank Corporation	25-Oct-19
3	The Enloe State Bank	Cooper	TX	10716	Legend Bank, N. A.	31-May-19
4	Washington Federal Bank for Savings	Chicago	IL	30570	Royal Savings Bank	15-Dec-17
...	...	...	...	...	...	...
554	Superior Bank, FSB	Hinsdale	IL	32646	Superior Federal, FSB	27-Jul-01
555	Malta National Bank	Malta	OH	6629	North Valley Bank	3-May-01
556	First Alliance Bank & Trust Co.	Manchester	NH	34264	Southern New Hampshire Bank & Trust	2-Feb-01
557	National State Bank of Metropolis	Metropolis	IL	3815	Banterra Bank of Marion	14-Dec-00
558	Bank of Honolulu	Honolulu	HI	21029	Bank of the Orient	13-Oct-00

559 rows × 6 columns

CSV Output

In [3]:
```
banks.to_csv('banks.csv', index=False)
```

Excel

Pandas can read and write excel files (only data, no formulas).

Excel Output

In [4]:
```
banks.to_excel('banks.xlsx', sheet_name='Banks', index=False)
```

Excel Input

In [5]:
```
pd.read_excel('banks.xlsx', sheet_name='Banks')
```
Out[5]:

	Bank Name	City	ST	CERT	Acquiring Institution	Closing Date
0	City National Bank of New Jersey	Newark	NJ	21111	Industrial Bank	1-Nov-19
1	Resolute Bank	Maumee	OH	58317	Buckeye State Bank	25-Oct-19
2	Louisa Community Bank	Louisa	KY	58112	Kentucky Farmers Bank Corporation	25-Oct-19
3	The Enloe State Bank	Cooper	TX	10716	Legend Bank, N. A.	31-May-19
4	Washington Federal Bank for Savings	Chicago	IL	30570	Royal Savings Bank	15-Dec-17
...	...	...	...	...	...	...
554	Superior Bank, FSB	Hinsdale	IL	32646	Superior Federal, FSB	27-Jul-01
555	Malta National Bank	Malta	OH	6629	North Valley Bank	3-May-01
556	First Alliance Bank & Trust Co.	Manchester	NH	34264	Southern New Hampshire Bank & Trust	2-Feb-01
557	National State Bank of Metropolis	Metropolis	IL	3815	Banterra Bank of Marion	14-Dec-00
558	Bank of Honolulu	Honolulu	HI	21029	Bank of the Orient	13-Oct-00

559 rows × 6 columns

HTML

You may need to install `htmllib5`, `lxml`, and `BeautifulSoup4` from your `Anaconda Navigator` Environment tab. Then restart Jupyter Notebook.

Pandas can read table tables from an html page.

HTML Input

Pandas `read_html` reads all the tables from a webpage and returns a list of `DataFrame` ojects:

In [6]:
```
population_url = 'https://www.tuttitalia.it/comuni-piccoli/popolazione/'

tables = pd.read_html(population_url, decimal=',', thousands='.')
len(tables)
```
Out[6]: 1

In [7]:
```
population = tables[0]
population
```
Out[7]:

	Unnamed: 0	Comune	Prov	Reg	Popolazioneresidenti	Superficiekm²	Densitàabitanti/km²	Altitudinem s.l.m.
0	1	Castelletto d'Erro	AL	PIE	149	4.66	32.00	544
1	2	Oltressenda Alta	BG	LOM	148	17.33	8.54	737
2	3	Ornica	BG	LOM	148	15.10	9.80	922
3	4	Soglio	AT	PIE	146	3.28	45.00	223
4	5	Castelvecchio di RB	SV	LIG	146	16.14	9.04	430
...	...	...	...	...	...	...	...	...
126	127	Ingria	TO	PIE	44	14.75	2.98	816
127	128	Briga Alta	CN	PIE	40	52.18	0.77	1310
128	129	Pedesina	SO	LOM	38	6.30	6.03	1032
129	130	Moncenisio	TO	PIE	35	4.50	7.78	1461
130	131	Morterone	LC	LOM	33	13.71	2.41	1070

131 rows × 8 columns

In [8]:
```
population.drop('Unnamed: 0', axis=1, inplace=True)
```

In [9]:
```
population.columns
```

Out[9]:
```
Index(['Comune', 'Prov', 'Reg', 'Popolazioneresidenti', 'Superficiekm²',
       'Densitàabitanti/km²', 'Altitudinem s.l.m.'],
      dtype='object')
```

In [10]:
```
population_columns = ['Town', 'Province', 'Region', 'Population', 'Area', 'Density', 'Altit
ude']
population.columns = population_columns
```

In [11]:
```
population.head(2)
```

Out[11]:

	Town	Province	Region	Population	Area	Density	Altitude
0	Castelletto d'Erro	AL	PIE	149	4.66	32.00	544
1	Oltressenda Alta	BG	LOM	148	17.33	8.54	737

In [12]:
```
population[(population['Province'] == 'BG') & (population['Population'] < 100)]
```

Out[12]:

	Town	Province	Region	Population	Area	Density	Altitude
87	Piazzolo	BG	LOM	88	4.15	21.0	702
107	Blello	BG	LOM	75	2.20	34.0	815

SQL

Pandas can also connect to databases. It requires:

- `SQLAlchemy` (generic SQL interface)
- A library to connect to your specific database
  - `psycopg2` for PostgreSQL
  - `pymysql` for MySQL
  - SQLite library is included by default

If `SQLAlchemy` is not provided, only SQLite is supported.

The key functions are:

- `read_sql_table(table_name, con)`
  - Reads a SQL database table into a DataFrame.
- `read_sql_query(sql, con)`
  - Reads a SQL query into a DataFrame.
- `read_sql(sql, con)`
  - Reads a SQL query or database table into a DataFrame.
- `DataFrame.to_sql(name, con)`
  - Writes records stored in a DataFrame to a SQL database.

In [13]:
```
from sqlalchemy import create_engine
```

In [14]:
```
connection_string = 'sqlite:///:memory:'
engine = create_engine(connection_string)
```

In [15]:
```
population.to_sql('population', engine)
```

In [16]:
```
sql_population = pd.read_sql('population', engine)
```

In [17]:
```
sql_population.tail(2)
```

Out[17]:

	index	Town	Province	Region	Population	Area	Density	Altitude
129	129	Moncenisio	TO	PIE	35	4.50	7.78	1461
130	130	Morterone	LC	LOM	33	13.71	2.41	1070

Exercise

We want to analyze the price of gasoline over the years.

In [18]:
```
# google: "annual fuel price inurl:gov.it"

super_95_url = 'https://dgsaie.mise.gov.it/prezzi_carburanti_annuali.php?pid=1&lang=en_US'
oil_url = 'https://dgsaie.mise.gov.it/prezzi_carburanti_annuali.php?pid=2&lang=en_US'
```

In [19]:
```
super95 = pd.read_html(super_95_url, decimal=',', thousands='.')[0]
super95.head()
```

Out[19]:

	Year	Price	Excise	VAT	Net
0	2019	1574.25	728.4	283.88	561.97
1	2018	1599.37	728.4	288.41	582.56
2	2017	1528.80	728.4	275.69	524.71
3	2016	1444.03	728.4	260.40	455.24
4	2015	1534.84	728.4	276.77	529.66

In [20]:
```
oil = pd.read_html(oil_url, decimal=',', thousands='.')[0]
oil.head()
```

Out[20]:

	Year	Price	Excise	VAT	Net
0	2019	1479.52	617.4	266.80	595.32
1	2018	1488.29	617.4	268.38	602.50
2	2017	1384.40	617.4	249.65	517.35
3	2016	1282.11	617.4	231.20	433.51
4	2015	1405.32	617.4	253.42	534.50

In [21]:
```
super95.set_index('Year', inplace=True)
oil.set_index('Year', inplace=True)
```

In [22]:
```
super95.loc[1998]
```

Out[22]:
```
Price      909.21
Excise     527.96
VAT        151.53
Net        229.71
Name: 1998, dtype: float64
```

In [23]:
```
fuels = super95.merge(oil, left_on='Year', right_on='Year', suffixes=['_super95', '_oil'])
fuels.head(5)
```

Out[23]:

	Price_super95	Excise_super95	VAT_super95	Net_super95	Price_oil	Excise_oil	VAT_oil	Net_oil
Year								
2019	1574.25	728.4	283.88	561.97	1479.52	617.4	266.80	595.32
2018	1599.37	728.4	288.41	582.56	1488.29	617.4	268.38	602.50
2017	1528.80	728.4	275.69	524.71	1384.40	617.4	249.65	517.35
2016	1444.03	728.4	260.40	455.24	1282.11	617.4	231.20	433.51
2015	1534.84	728.4	276.77	529.66	1405.32	617.4	253.42	534.50

In [24]:
```
fuels[fuels['Price_super95'] > fuels['Price_oil'] * 1.2]
```

Out[24]:

	Price_super95	Excise_super95	VAT_super95	Net_super95	Price_oil	Excise_oil	VAT_oil	Net_oil
Year								
2003	1057.47	541.84	176.25	339.39	876.90	403.21	146.15	327.54
2002	1046.23	541.84	174.37	330.03	855.74	403.21	142.62	309.91
2001	1051.72	523.78	175.29	352.65	868.17	385.08	144.69	338.39
2000	1082.71	521.63	180.45	380.62	892.49	383.05	148.75	360.69
1999	957.52	539.04	159.59	258.90	759.60	400.30	126.60	232.69
1998	909.21	527.96	151.53	229.71	710.51	386.04	118.42	206.05
1997	942.21	527.96	152.08	262.17	743.97	386.04	120.06	237.87
1996	925.31	527.80	147.74	249.76	737.28	386.04	117.72	233.53

Now we want to find years in which the gasoline price dropped from January to December.

In [25]:
```
monthly_url = 'https://dgsaie.mise.gov.it/prezzi_carburanti_mensili.php?wm_page=1&lang=en_U
S'
```

We see that they have multiple pages, let's see if we can find a pattern.

What about that `wm_page=1` ?

In [26]:
```
url_pattern = 'https://dgsaie.mise.gov.it/prezzi_carburanti_mensili.php?wm_page={}&lang=en_
US'
```

In [27]:
```
monthly_test = pd.read_html(url_pattern.format(1), decimal=',', thousands='.')[0]
monthly_test.head(2)
```

Out[27]:

	Year	Month	Price	Excise	VAT	Net
0	2019	December	1584.91	728.4	285.81	570.70
1	2019	November	1575.67	728.4	284.14	563.13

In [28]:
```
monthly_test.set_index(['Year', 'Month']).head(2)
```

Out[28]:

		Price	Excise	VAT	Net
Year	**Month**				
2019	December	1584.91	728.4	285.81	570.70
	November	1575.67	728.4	284.14	563.13

In [29]:
```
def get_table(page):
    data = pd.read_html(url_pattern.format(page), decimal=',', thousands='.')[0]
    data.set_index(['Year', 'Month'], inplace=True)
    return data
```

In [30]:
```
get_table(1).head(2)
```

Out[30]:

Year	Month	Price	Excise	VAT	Net
2019	December	1584.91	728.4	285.81	570.70
	November	1575.67	728.4	284.14	563.13

In [31]:
```
tables = [get_table(page) for page in range(1, 9)]
len(tables)
```

Out[31]: 8

In [32]:
```
monthly_super95 = pd.concat(tables)
```

In [33]:
```
monthly_super95
```

Out[33]:

Year	Month	Price	Excise	VAT	Net
2019	December	1584.91	728.40	285.81	570.70
	November	1575.67	728.40	284.14	563.13
	October	1576.79	728.40	284.34	564.05
	September	1579.09	728.40	284.75	565.94
	August	1574.47	728.40	283.92	562.15
...	...	...	...	...	...
1996	May	929.94	527.96	148.48	253.50
	April	931.48	527.96	148.72	254.79
	March	917.83	527.96	146.54	243.32
	February	907.09	527.96	144.83	234.30
	January	904.18	525.77	144.37	234.05

288 rows × 4 columns

In [34]:
```
december = monthly_super95.xs('December', level='Month')
december.head(2)
```

Out[34]:

Year	Price	Excise	VAT	Net
2019	1584.91	728.4	285.81	570.70
2018	1509.60	728.4	272.22	508.98

In [35]:
```
january = monthly_super95.xs('January', level='Month')
january.head(2)
```

Out[35]:

Year	Price	Excise	VAT	Net
2019	1490.13	728.4	268.71	493.02
2018	1568.60	728.4	282.86	557.34

In [36]:
```
diff = january.join(december, lsuffix='_jan', rsuffix='_dec')
diff.head(2)
```

Out[36]:

Year	Price_jan	Excise_jan	VAT_jan	Net_jan	Price_dec	Excise_dec	VAT_dec	Net_dec
2019	1490.13	728.4	268.71	493.02	1584.91	728.4	285.81	570.70
2018	1568.60	728.4	282.86	557.34	1509.60	728.4	272.22	508.98

In [37]:
```
decreasing = diff[diff['Price_jan'] > diff['Price_dec']].copy()
decreasing
```

Out[37]:

Year	Price_jan	Excise_jan	VAT_jan	Net_jan	Price_dec	Excise_dec	VAT_dec	Net_dec
2018	1568.60	728.40	282.86	557.34	1509.60	728.40	272.22	508.98
2015	1472.04	728.40	265.45	478.19	1450.68	728.40	261.60	460.68
2014	1723.07	728.40	310.72	683.95	1585.65	730.80	285.94	568.91
2013	1749.94	728.40	303.71	717.83	1727.63	728.40	311.54	687.69
2008	1364.44	564.00	227.41	573.03	1120.88	564.00	186.81	370.07
2006	1248.31	564.00	208.05	476.26	1219.19	564.00	203.20	451.99
2003	1068.53	541.84	178.09	348.60	1036.82	541.84	172.80	322.18
2001	1046.74	520.32	174.46	351.96	993.15	541.84	165.52	285.78
1998	930.69	527.96	155.11	247.61	885.21	527.96	147.54	209.71
1997	938.82	527.96	149.90	260.96	935.79	527.96	155.96	251.86

In [38]:
```
decreasing['Liter_diff'] = (decreasing['Price_dec'] - decreasing['Price_jan']) / 1000
```

In [39]:
```
decreasing.sort_values("Liter_diff")
```

Out[39]:

Year	Price_jan	Excise_jan	VAT_jan	Net_jan	Price_dec	Excise_dec	VAT_dec	Net_dec	Liter_diff
2008	1364.44	564.00	227.41	573.03	1120.88	564.00	186.81	370.07	-0.24356
2014	1723.07	728.40	310.72	683.95	1585.65	730.80	285.94	568.91	-0.13742
2018	1568.60	728.40	282.86	557.34	1509.60	728.40	272.22	508.98	-0.05900
2001	1046.74	520.32	174.46	351.96	993.15	541.84	165.52	285.78	-0.05359
1998	930.69	527.96	155.11	247.61	885.21	527.96	147.54	209.71	-0.04548
2003	1068.53	541.84	178.09	348.60	1036.82	541.84	172.80	322.18	-0.03171
2006	1248.31	564.00	208.05	476.26	1219.19	564.00	203.20	451.99	-0.02912
2013	1749.94	728.40	303.71	717.83	1727.63	728.40	311.54	687.69	-0.02231
2015	1472.04	728.40	265.45	478.19	1450.68	728.40	261.60	460.68	-0.02136
1997	938.82	527.96	149.90	260.96	935.79	527.96	155.96	251.86	-0.00303

SF Salaries Exercise - Solutions

We will be using the SF Salaries Dataset (https://www.kaggle.com/kaggle/sf-salaries) from Kaggle

Import pandas as pd

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

Read `Salaries.csv` as a `DataFrame` and name the variable `sal` .

In [9]:
```
sal = pd.read_csv('Salaries.csv')
```

Use `.head()` to visualize the first entries.

In [10]:
```
sal.head()
```

Out[10]:

	Id	EmployeeName	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay	TotalPayBenefits	Year	No
0	1	NATHANIEL FORD	GENERAL MANAGER-METROPOLITAN TRANSIT AUTHORITY	167411.18	0.00	400184.25	NaN	567595.43	567595.43	2011	I
1	2	GARY JIMENEZ	CAPTAIN III (POLICE DEPARTMENT)	155966.02	245131.88	137811.38	NaN	538909.28	538909.28	2011	I
2	3	ALBERT PARDINI	CAPTAIN III (POLICE DEPARTMENT)	212739.13	106088.18	16452.60	NaN	335279.91	335279.91	2011	I
3	4	CHRISTOPHER CHONG	WIRE ROPE CABLE MAINTENANCE MECHANIC	77916.00	56120.71	198306.90	NaN	332343.61	332343.61	2011	I
4	5	PATRICK GARDNER	DEPUTY CHIEF OF DEPARTMENT,(FIRE DEPARTMENT)	134401.60	9737.00	182234.59	NaN	326373.19	326373.19	2011	I

Use the `.info()` method to see how many entries are in the dataset.

In [11]:
```
sal.info() # 148654 Entries
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 148654 entries, 0 to 148653
Data columns (total 13 columns):
Id                  148654 non-null int64
EmployeeName        148654 non-null object
JobTitle            148654 non-null object
BasePay             148045 non-null float64
OvertimePay         148650 non-null float64
OtherPay            148650 non-null float64
Benefits            112491 non-null float64
TotalPay            148654 non-null float64
TotalPayBenefits    148654 non-null float64
Year                148654 non-null int64
Notes               0 non-null float64
Agency              148654 non-null object
Status              0 non-null float64
dtypes: float64(8), int64(2), object(3)
memory usage: 14.7+ MB
```

What is the average of the `BasePay` column?

```
In [12]: sal['BasePay'].mean()
```

```
Out[12]: 66325.44884050643
```

What is the highest value in `OvertimePay` ?

```
In [13]: sal['OvertimePay'].max()
```

```
Out[13]: 245131.88
```

What is the job title of `DAVID BROWN` ?

```
In [14]: sal[sal['EmployeeName'] == 'DAVID BROWN']['JobTitle']
```

```
Out[14]: 608    LIEUTENANT, FIRE DEPARTMENT
         Name: JobTitle, dtype: object
```

How much does `DAVID BROWN` make (including benefits)?

```
In [15]: sal[sal['EmployeeName'] == 'DAVID BROWN']['TotalPayBenefits']
```

```
Out[15]: 608    182211.64
         Name: TotalPayBenefits, dtype: float64
```

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

```
In [16]: sal[sal['TotalPayBenefits'] == sal['TotalPayBenefits'].max()] #['EmployeeName']

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

Out[16]:

	Id	EmployeeName	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay	TotalPayBenefits	Year	No
0	1	NATHANIEL FORD	GENERAL MANAGER-METROPOLITAN TRANSIT AUTHORITY	167411.18	0.0	400184.25	NaN	567595.43	567595.43	2011	I

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

```
In [18]: sal[sal['TotalPayBenefits'] == sal['TotalPayBenefits'].min()] #['EmployeeName']

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

Out[18]:

	Id	EmployeeName	JobTitle	BasePay	OvertimePay	OtherPay	Benefits	TotalPay	TotalPayBenefits	Year
148653	148654	Joe Lopez	Counselor, Log Cabin Ranch	0.0	0.0	-618.13	0.0	-618.13	-618.13	2014

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

In [19]:
```
sal.groupby('Year').mean()['BasePay']
```

Out[19]:
```
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 [20]:
```
sal['JobTitle'].nunique()
```

Out[20]:
```
2159
```

What are the top 5 most common jobs?

In [21]:
```
sal['JobTitle'].value_counts().head(5)
```

Out[21]:
```
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 have the word `Chief` in their job title?

In [22]:
```
sum(sal['JobTitle'].apply(lambda job: 'chief' in job.lower()))
```

Out[22]:
```
627
```

Bonus: Is there a correlation between the length of `JobTitle` and `Salary` ?

In [24]:
```
sal['title_len'] = sal['JobTitle'].apply(len)
sal[['title_len', 'TotalPayBenefits']].corr()
```

Out[24]:

	title_len	TotalPayBenefits
title_len	1.000000	-0.036878
TotalPayBenefits	-0.036878	1.000000

Ecommerce Purchases Exercise - Solutions

Analyze some fake data about Amazon purchases.

Import pandas and read in the `Purchases.csv` into a DataFrame named `ecom`

In [1]:
```
import pandas as pd
ecom = pd.read_csv('Purchases.csv')
```

Check the head of the DataFrame.

In [2]:
```
ecom.head()
```

Out[2]:

	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	pdunlap@yal
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	anthony41@r
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	amymiller@ harri
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	brent16@olson-robir
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	christopherwright@gr

How many rows and columns are there?

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
Address             10000 non-null object
Lot                 10000 non-null object
AM or PM            10000 non-null object
Browser Info        10000 non-null object
Company             10000 non-null object
Credit Card         10000 non-null int64
CC Exp Date         10000 non-null object
CC Security Code    10000 non-null int64
CC Provider         10000 non-null object
Email               10000 non-null object
Job                 10000 non-null object
IP Address          10000 non-null object
Language            10000 non-null object
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 [4]: ecom['Purchase Price'].mean()
```

```
Out[4]: 50.34730200000025
```

What were the highest and lowest purchase prices?

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

```
Out[5]: 99.99
```

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

```
Out[6]: 0.0
```

How many people have English `'en'` as their `Language` of choice?

```
In [7]: ecom[ecom['Language']=='en'].count()
```

```
Out[7]: 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 [8]: ecom[ecom['Job'] == 'Lawyer'].info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 30 entries, 470 to 9979
Data columns (total 14 columns):
Address             30 non-null object
Lot                 30 non-null object
AM or PM            30 non-null object
Browser Info        30 non-null object
Company             30 non-null object
Credit Card         30 non-null int64
CC Exp Date         30 non-null object
CC Security Code    30 non-null int64
CC Provider         30 non-null object
Email               30 non-null object
Job                 30 non-null object
IP Address          30 non-null object
Language            30 non-null object
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?

```
In [9]: ecom['AM or PM'].value_counts()
```

```
Out[9]: PM    5068
        AM    4932
        Name: AM or PM, dtype: int64
```

What are the 5 most common `Job` titles?

```
In [10]: ecom['Job'].value_counts().head(5)
```

```
Out[10]: Interior and spatial designer    31
         Lawyer                           30
         Social researcher                28
         Purchasing manager               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 [11]: ecom[ecom['Lot']=='90 WT']['Purchase Price']
```

```
Out[11]: 513    75.1
         Name: Purchase Price, dtype: float64
```

What is the `Email` of the person with the following `Credit Card` number: `4926535242672853` ?

```
In [12]: ecom[ecom["Credit Card"] == 4926535242672853]['Email']
```

```
Out[12]: 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 [13]: ecom[(ecom['CC Provider']=='American Express') & (ecom['Purchase Price']>95)].count()
```

```
Out[13]: 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
```

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

```
In [14]: sum(ecom['CC Exp Date'].apply(lambda x: x[3:]) == '25')
```

```
Out[14]: 1033
```

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

```
In [15]: ecom['Email'].apply(lambda x: x.split('@')[1]).value_counts().head(5)
```

```
Out[15]: hotmail.com     1638
         yahoo.com       1616
         gmail.com       1605
         smith.com         42
         williams.com      37
         Name: Email, dtype: int64
```