Fall 21. Nov 01, 2021. 12:18 pm EST. This notebook integrates two - the first part introduces basics of Pandas; the lower part suggests applying pandas/python as first steps getting to know our data.

See also https://pandas.pydata.org

This week, we get ready for Project 2, review import first steps for analysis and stats with Pandas, optionally look at large data architectures, and underscore how to think about integrating pandas & python in approaching unknown data sets.

This week's topics:

This week we look at Pandas. It is important, too, to begin to contextualize these skills in analsys, practice, associate programming with research practices, and more, as rationales for many of Panda's tools.

Agenda

1. Data Exploration & Analysis - in General

2. Overview: Integrating Python and Pandas into this work
3. Exploration: transforming data
4. Optional: architectures and work practices; data wrangling review pages
5. Pandas: series, data frames, panel
6. Pandas Code Samples
7. Optional Pandas-on-the-job with visuals
8. Optional Review
9. Breakout Room
10. Project 2

---

1. Data Exploration & Analysis

Data **Exploration**:

- For data integrity
- To develop questions based on the variables
- To break your model - better now than in production

Data **Analysis**:

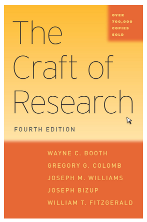
- Answer a research question or hypothesis
- Usually involves complex math, modeling statistics
- Likely to combine datasets
- Explore data by collapsing in groups in various ways
- Some functions are useful in exploration & analysis

Why use Pandas?

Pandas and other tools help in machine learning, scaling data, multiple regression, and more! Usually work together with numpy, scipy, matplotlib, scikit-learn, pysqlite3, psycopg2, and others. These two sites introduce a *lot* of features for python to read lots of data sources and more efficient problem-solving techniques: https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html (https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html) and https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-hdf5 (https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html#io-hdf5).

2. Overview Exploration & Analysis: Discuss

Take time to get to know your data - the domain & range, parametric or non-parametric; explore the measures of central tendancy ... Here are some handy commands:



- `value_counts()`
- `describe()`
- `min(), max(), isnull()`
- Plot your data during exploration, too, not just during analysis.
- Consider the source(s) of your data and research the topic:
  - basic research methods require looking at threats to validity,
  - cross-validating the data,
  - issues of research "bias",
  - lack of precision in definitions (e.g., mismatched metadata when combining data)
  - look for any professional/industrial gold standards for measurement
  - what might be confounding events in your data?

You might want to learn more about the expectations of "research" by reading Booth, et al., *Craft of research*.

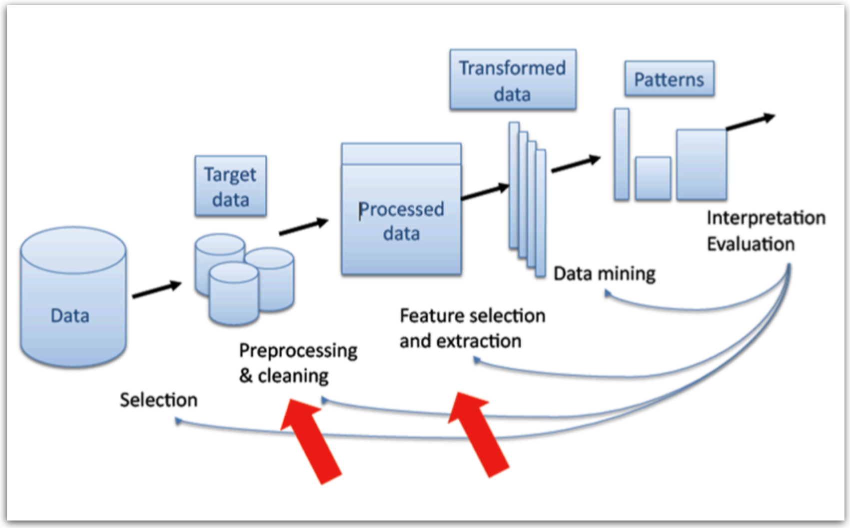
3. Exploration & Analysis: Transformations for analysis

- filter based on some condition (e.g., too high values? duplicated data?)
- create new columns, when necessary
- aggregate or collapse by groups
- join two+ datasets together

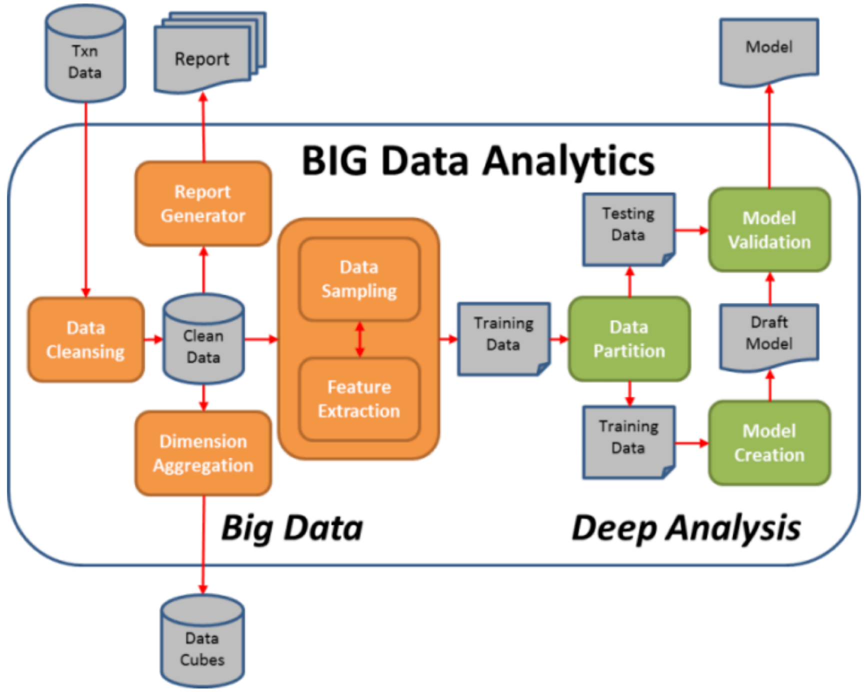
4. Optional Discussion

Applying exploration & analysis in larger settings: data mining, big data analytics, Hadoop/activities and helpers.

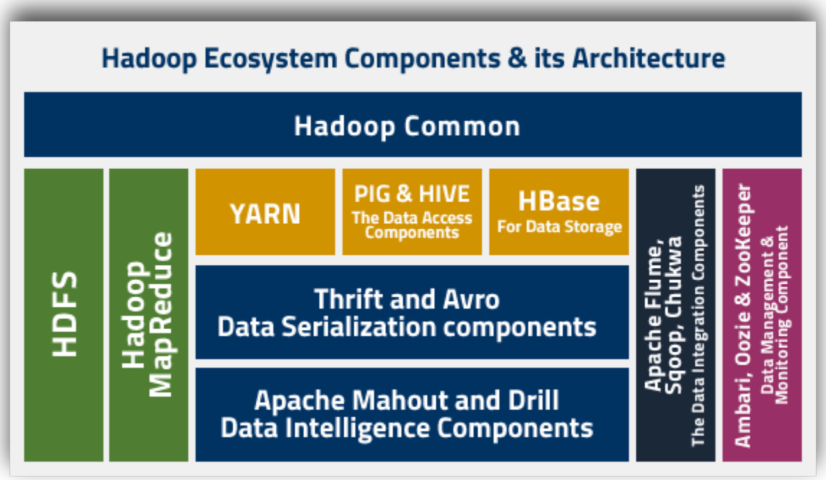
Notice that as datasets get larger and exploration/analysis more complex, we identify specific work behaviors, warehouses, and architecdures/other programming languages as part of the job. See the Big Data book in the optional resources.



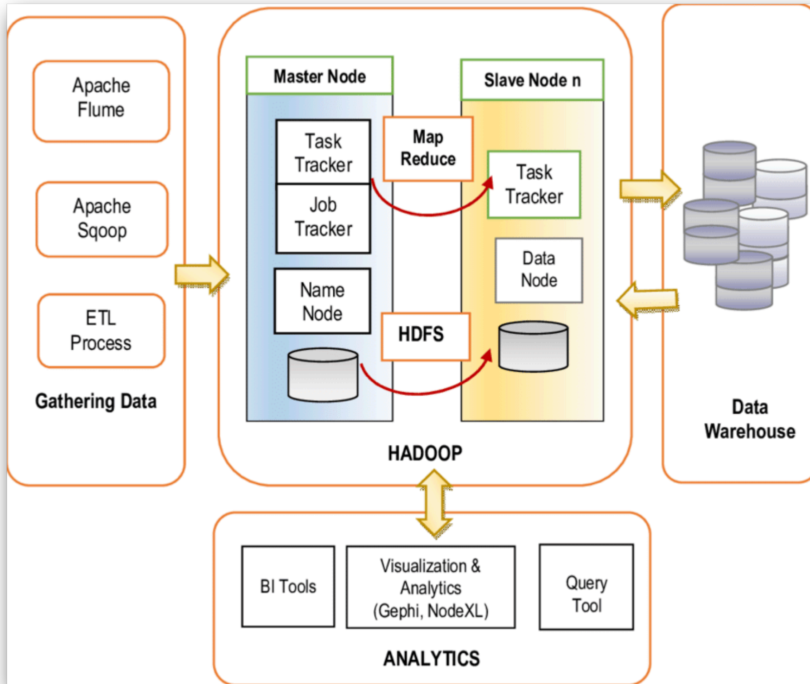
Data Mining Activities



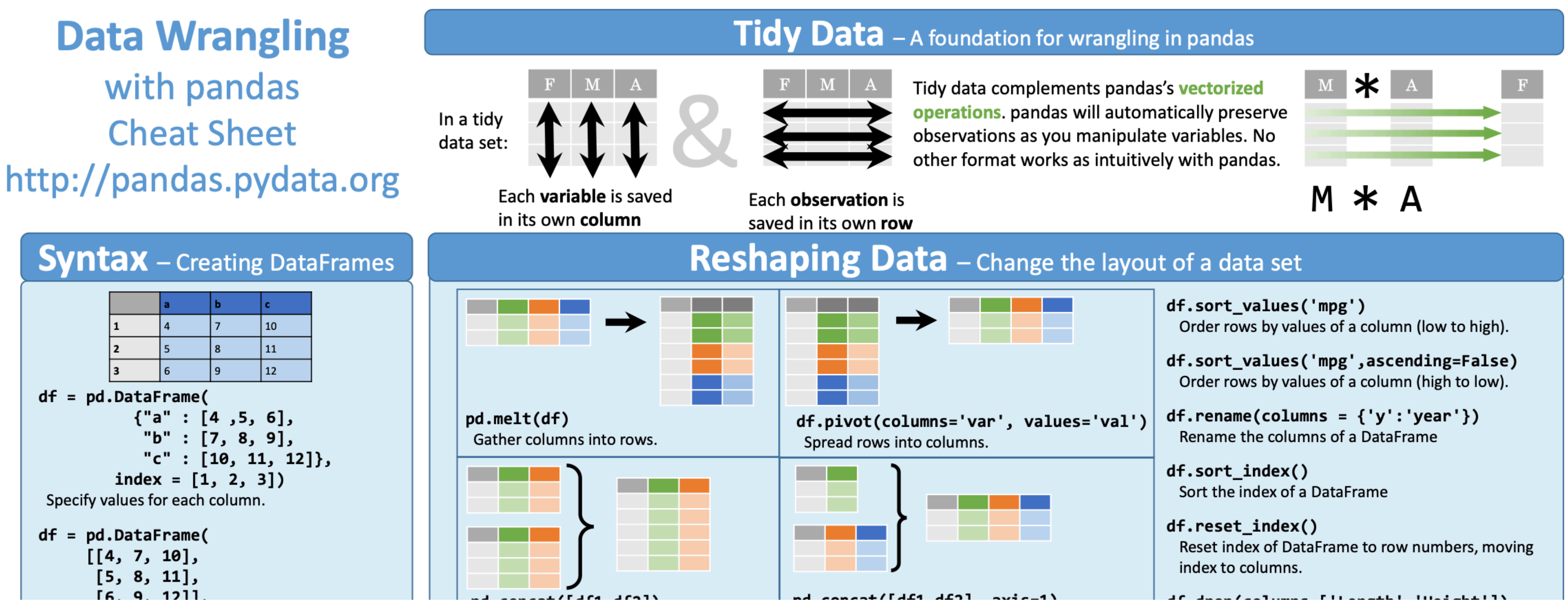
Big Data



Hadoop



Tools



Data Wrangling
with pandas
Cheat Sheet
http://pandas.pydata.org

Tidy Data – A foundation for wrangling in pandas

In a tidy data set:

Each **variable** is saved in its own **column**

&

Each **observation** is saved in its own **row**

Tidy data complements pandas's vectorized operations. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.

M * A

Syntax – Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
          {"a" : [4 ,5, 6],
           "b" : [7, 8, 9],
           "c" : [10, 11, 12]},
        index = [1, 2, 3])
```

Specify values for each column.

```
df = pd.DataFrame(
     [[4, 7, 10],
      [5, 8, 11],
```

Reshaping Data – Change the layout of a data set

`pd.melt(df)`
Gather columns into rows.

`df.pivot(columns='var', values='val')`
Spread rows into columns.

`df.sort_values('mpg')`
Order rows by values of a column (low to high).

`df.sort_values('mpg',ascending=False)`
Order rows by values of a column (high to low).

`df.rename(columns = {'y':'year'})`
Rename the columns of a DataFrame

`df.sort_index()`
Sort the index of a DataFrame

`df.reset_index()`
Reset index of DataFrame to row numbers, moving index to columns.

```
     index=[1, 2, 3],
     columns=['a', 'b', 'c'])
```
Specify values for each row.

		a	b	c
n	v			
d	1	4	7	10
	2	5	8	11
e	2	6	9	12

```
df = pd.DataFrame(
          {"a" : [4 ,5, 6],
           "b" : [7, 8, 9],
           "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
          [('d',1),('d',2),('e',2)],
             names=['n','v'])))
```
Create DataFrame with a MultiIndex

Method Chaining

Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

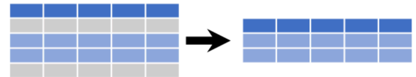
```
df = (pd.melt(df)
        .rename(columns={
                'variable' : 'var',
                'value' : 'val'})
        .query('val >= 200')
     )
```

`pd.concat([df1,df2])`
Append rows of DataFrames

`pd.concat([df1,df2], axis=1)`
Append columns of DataFrames

`df.drop(columns=['Length','Height'])`
Drop columns from DataFrame

Subset Observations (Rows)



`df[df.Length > 7]`
Extract rows that meet logical criteria.

`df.drop_duplicates()`
Remove duplicate rows (only considers columns).

`df.head(n)`
Select first n rows.

`df.tail(n)`
Select last n rows.

`df.sample(frac=0.5)`
Randomly select fraction of rows.

`df.sample(n=10)`
Randomly select n rows.

`df.iloc[10:20]`
Select rows by position.

`df.nlargest(n, 'value')`
Select and order top n entries.

`df.nsmallest(n, 'value')`
Select and order bottom n entries.

Logic in Python (and pandas)

<	Less than	!=	Not equal to
>	Greater than	df.column.isin(*values*)	Group membership
==	Equals	pd.isnull(*obj*)	Is NaN
<=	Less than or equals	pd.notnull(*obj*)	Is not NaN
>=	Greater than or equals	&,|,~,^,df.any(),df.all()	Logical and, or, not, xor, any, all

Subset Variables (Columns)

`df[['width','length','species']]`
Select multiple columns with specific names.

`df['width']` *or* **`df.width`**
Select single column with specific name.

`df.filter(regex='`*`regex`*`')`
Select columns whose name matches regular expression *regex*.

regex (Regular Expressions) Examples

'\.'	Matches strings containing a period '.'
'Length$'	Matches strings ending with word 'Length'
'^Sepal'	Matches strings beginning with the word 'Sepal'
'^x[1-5]$'	Matches strings beginning with 'x' and ending with 1,2,3,4,5
''^(?!Species$).*'	Matches strings except the string 'Species'

`df.loc[:,'x2':'x4']`
Select all columns between x2 and x4 (inclusive).

`df.iloc[:,[1,2,5]]`
Select columns in positions 1, 2 and 5 (first column is 0).

`df.loc[df['a'] > 10, ['a','c']]`
Select rows meeting logical condition, and only the specific columns .

Summarize Data

`df['w'].value_counts()`
Count number of rows with each unique value of variable

`len(df)`
# of rows in DataFrame.

`df['w'].nunique()`
# of distinct values in a column.

`df.describe()`
Basic descriptive statistics for each column (or GroupBy)

pandas provides a large set of **summary functions** that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

`sum()`
Sum values of each object.

`count()`
Count non-NA/null values of each object.

`median()`
Median value of each object.

`quantile([0.25,0.75])`
Quantiles of each object.

`apply(`*`function`*`)`
Apply function to each object.

`min()`
Minimum value in each object.

`max()`
Maximum value in each object.

`mean()`
Mean value of each object.

`var()`
Variance of each object.

`std()`
Standard deviation of each object.

Group Data

`df.groupby(by="col")`
Return a GroupBy object, grouped by values in column named "col".

`df.groupby(level="ind")`
Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group. Additional GroupBy functions:

`size()`
Size of each group.

`agg(`*`function`*`)`
Aggregate group using function.

Windows

`df.expanding()`
Return an Expanding object allowing summary functions to be applied cumulatively.

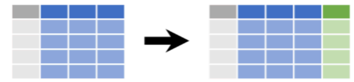
`df.rolling(n)`
Return a Rolling object allowing summary functions to be applied to windows of length n.

Handling Missing Data

`df.dropna()`
Drop rows with any column having NA/null data.

`df.fillna(value)`
Replace all NA/null data with value.

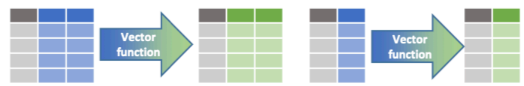
Make New Columns



`df.assign(Area=lambda df: df.Length*df.Height)`
Compute and append one or more new columns.

`df['Volume'] = df.Length*df.Height*df.Depth`
Add single column.

`pd.qcut(df.col, n, labels=False)`
Bin column into n buckets.



pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

`max(axis=1)`
Element-wise max.

`clip(lower=-10,upper=10)`
Trim values at input thresholds

`min(axis=1)`
Element-wise min.

`abs()`
Absolute value.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

`shift(1)`
Copy with values shifted by 1.

`rank(method='dense')`
Ranks with no gaps.

`rank(method='min')`
Ranks. Ties get min rank.

`rank(pct=True)`
Ranks rescaled to interval [0, 1].

`rank(method='first')`
Ranks. Ties go to first value.

`shift(-1)`
Copy with values lagged by 1.

`cumsum()`
Cumulative sum.

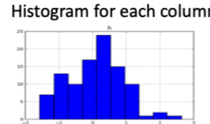
`cummax()`
Cumulative max.

`cummin()`
Cumulative min.

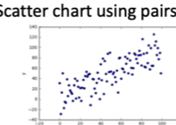
`cumprod()`
Cumulative product.

Plotting

`df.plot.hist()`
Histogram for each column



`df.plot.scatter(x='w',y='h')`
Scatter chart using pairs of points



Combine Data Sets

adf

x1	x2
A	1
B	2
C	3

\+

bdf

x1	x3
A	T
B	F
D	T

=

Standard Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NaN

```
pd.merge(adf, bdf,
         how='left', on='x1')
```
Join matching rows from bdf to adf.

x1	x2	x3
A	1.0	T
B	2.0	F
D	NaN	T

```
pd.merge(adf, bdf,
         how='right', on='x1')
```
Join matching rows from adf to bdf.

x1	x2	x3
A	1	T
B	2	F

```
pd.merge(adf, bdf,
         how='inner', on='x1')
```
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NaN
D	NaN	T

```
pd.merge(adf, bdf,
         how='outer', on='x1')
```
Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

`adf[adf.x1.isin(bdf.x1)]`
All rows in adf that have a match in bdf.

x1	x2
C	3

`adf[~adf.x1.isin(bdf.x1)]`
All rows in adf that do not have a match in bdf.

ydf

x1	x2
A	1
B	2
C	3

\+

zdf

x1	x2
B	2
C	3
D	4

=

Set-like Operations

x1	x2
B	2
C	3

`pd.merge(ydf, zdf)`
Rows that appear in both ydf and zdf (Intersection).

x1	x2
A	1
B	2
C	3
D	4

`pd.merge(ydf, zdf, how='outer')`
Rows that appear in either or both ydf and zdf (Union).

x1	x2
A	1

```
pd.merge(ydf, zdf, how='outer',
         indicator=True)
.query('_merge == "left_only"')
.drop(columns=['_merge'])
```
Rows that appear in ydf but not zdf (Setdiff).

End of the optional section.

5. Exploration & Analysis: Transformations for analysis

Type	Description	Example

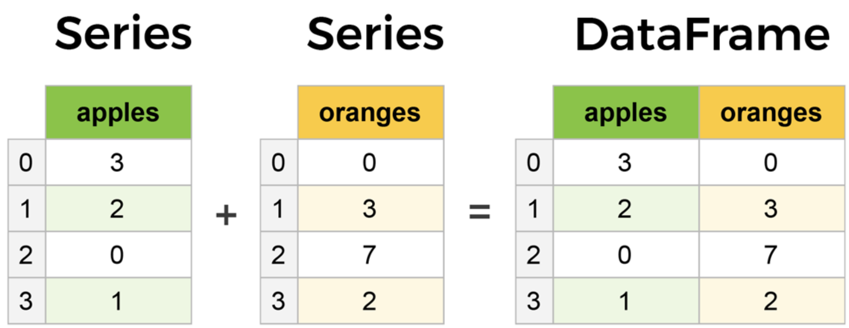
Series	1D labeled homogeneous array, size is immutable.	`import pandas as pd`
		`import numpy as np`
		`data = np.array(['a','b','c'])`
		`s = pd.Series(data)`
		`print(s)`

Data Frames	General 2d labeled, size-mutable tabular structure with potentially heterogeneous-typed columns	`import pandas as pd`
		`data = [1, 2, 3, 4, 5]`
		`df = pd.DataFrame(data)`

Panel	General 3d labeled, size-mutable array	`import pandas as pd`
		`import numpy as np`
		`data = np.random.rand(2, 4, 5)`
		`p = pd.Panel(data)`

6. Examples Demonstrating Pandas

Pandas components are `Series` (a column) and `DataFrame`, a multi-dimensional table made up of Series.



Use pandas to create a frame (that looks like a combo of json and a dictionary):

```
data = {
    'cats': [1, 4, 2, 8],
    'dogs': [1, 3, 3, 2]
}
pets = pd.DataFrame(data)
```

In [22]:

```python
import pandas as pd

data = {
    'cats': [1, 4, 2, 8],
    'dogs': [1, 3, 3, 2]
}
pets = pd.DataFrame(data)
pets
```

Out[22]:

	cats	dogs
0	1	1
1	4	3
2	2	3
3	8	2

In [23]:

```python
import pandas as pd

data = {
    'cats': [123, 14, 42, 8],
    'dogs': [81, 33, 113, 132]
}
new_pets = pd.DataFrame(data, index=['Paris','Boston','Rome','Monterey'])
new_pets
```

Out[23]:

	cats	dogs
Paris	123	81
Boston	14	33
Rome	42	113
Monterey	8	132

In [24]:
```
""" if the data set were really large
    it'd be nice to locate by key
"""
new_pets.loc['Paris']
```

Out[24]:
```
cats    123
dogs     81
Name: Paris, dtype: int64
```

Building up class examples

In these examples, we want to build up from basics to increasingly complex uses of pandas for coding practices.

In [25]:

```python
import pandas as pd

data = pd.Series([1,2,3,4,6,0,85,45,7,53,321,4,32,2355,6])

# select values from data; those < 10
print("select data < 10: ", data[data < 10])
print("range of data < 10 > 5: ", data[(data < 10) & (data < 5)])
print("_"*50,"\n")
# same result but “chained”
print("same results but 'chained': ", data[(data < 10) & (data > 5)])
print("same results without using &: ", data[data < 10][data > 5])
```

```
select data < 10:  0     1
1     2
2     3
3     4
4     6
5     0
8     7
11    4
14    6
dtype: int64
range of data < 10 > 5:  0     1
1     2
2     3
3     4
5     0
11    4
dtype: int64
__________________________________________________ 

same results but 'chained':  4     6
8     7
14    6
dtype: int64
same results without using &:  4     6
8     7
14    6
dtype: int64
```

Slicing and Manipulating

Here we'll make a copy first (for safekeeping) and explore using head and slicing and replacing, etc.

In [26]:

```python
d5 = data.head().copy()
print("d5: ", d5)
d6 = data.head(70).copy() # specify the size of the head
print("d6: ", d6)

print(data[0:10:2]) # slide the data

data[0] = 10000 # value replacement
```

```
d5:  0    1
1    2
2    3
3    4
4    6
dtype: int64
d6:  0        1
1        2
2        3
3        4
4        6
5        0
6       85
7       45
8        7
9       53
10     321
11       4
12      32
13    2355
14       6
dtype: int64
0     1
2     3
4     6
6    85
8     7
dtype: int64
```

Using `any` and `all` condition testing

In [27]:

```
print("any: ",data[data < 10].any()) # should return true
print("all: ",data[data < 10].all()) # false
print("alternative: ",(data > 10000).any()) # alternative
print("all, diff syntax", (data > 0).all) # false
```

```
any:  True
all:  False
alternative:  False
all, diff syntax <bound method Series.all of 0      True
1      True
2      True
3      True
4      True
5     False
6      True
7      True
8      True
9      True
10     True
11     True
12     True
13     True
14     True
dtype: bool>
```

Get to know your data for basic stats

In [28]:

```
print("Length of the data set:", len(data))

# individual measurements - can use the describe() for the whole set
```

```python
# individual measurements - can use the describe() for the whole set
print("mean: ", data.mean())
print("mode: ", data.mode())
print("median: ", data.median())
print("count: ", data.count())
print("std: ", data.std())
print("unique: ", data.unique())

print("\ndescribe: ", data.describe())
print("\n\nvalue_counts and shape of the data.\nNote that shape is an attribute, not a metho
print("\tValue counts: ",data.value_counts())
print("\tShape: ", data.shape)
```

```
Length of the data set: 15
mean:  861.5333333333333
mode:  0    4
1    6
dtype: int64
median:  7.0
count:  15
std:  2598.470975309097
unique:  [10000     2     3     4     6     0    85    45     7    53   321    32
  2355]

describe:  count       15.000000
mean       861.533333
std       2598.470975
min          0.000000
25%          4.000000
50%          7.000000
75%         69.000000
max      10000.000000
dtype: float64


value_counts and shape of the data.
Note that shape is an attribute, not a method
	Value counts:  6        2
4        2
85       1
53       1
```

```
2355     1
10000    1
45       1
7        1
32       1
3        1
2        1
321      1
0        1
dtype: int64
        Shape:  (15,)
```

Adding, editing data

In [29]:

```python
print("by single index names: ", data[10])  # by single index names


print("look up by index location, using dict stye[]")
print(data.iloc[ [0,3] ])
```

```
by single index names:  321
look up by index location, using dict stye[]
0    10000
3        4
dtype: int64
```

Filling data cells

It's often useful to fill cells with some placeholder data and interpolate missing values. This is common in k-nearest neighbor and any situation to minimize need to check for data-oriented problems when running your scripts.

In [30]:

```python
combo = pd.Series([0,0,0,0,0])
new_combo = pd.Series([0,0,0,0,0])
new_combination = pd.Series([0,0,0,0,0])

# set the fill value.
print("combo after reindex")
combo.reindex([0, 2, 15, 21], fill_value = 0)
print(combo)

# fill the NaNs
new_combo.fillna(0)

# forward and back fill to guess at missing values.
new_combo.ffill()

new_combo.bfill()

new_combination.interpolate() # one of many techniques.
```

```
combo after reindex
0    0
1    0
2    0
3    0
4    0
dtype: int64
```

Out[30]:

```
0    0
1    0
2    0
3    0
4    0
dtype: int64
```

Integrating data - input/output to csv, sql, json

Typical to read in data - sometimes with the metadata (such as an SQL table header) or not - change to .json and import to a dictionary.

Examples:
`myframe = pd.read_csv('mydata.csv', index_col = 0)`
`myframe = pd.read_json('mydata.json')` . May need the orient keyword.

If you're reading from sql, you need to `import sqlite3`. Then you can establish a connection to the databae and then select data from your table(s), e.g., `my_connection = sqlite3.connect("mydatabase.db")`

Output/convert to different data types:

```
dataframe.to_csv('newfile.csv')
dataframe.to_json('newfile.json')
dataframe.to_sql('newfile', my_connection)
```

Get to know your data first

1. What's the dimensions of our data? `my_dataframe.shape`
2. Get some info about your data: `my_dataframe.info()`
3. See the first 5 rows (the head) of your data: `my_dataframe.head()`
4. last rows: `my_dataframe.tail()`
5. remove duplicates `drop_duplicates()`
6. copy the frame to a new one, or overwrite it `my_dataframe.drop_duplicates(inplace = True)` (We can keep the first of the duplicates (the default) or drop all duplicates: `temp_df = my_dataframe.append(my_dataframe)` tomake a copy; `temp_df.drop_duplicates(inplace=True, keep=False)` would overwrite the df and remove all duplicates (keep=False).

Movie data from [here (https://gist.github.com/tiangechen/b68782efa49a16edaf07dc2cdaa855ea#file-movies-csv)](https://gist.github.com/tiangechen/b68782efa49a16edaf07dc2cdaa855ea#file-movies-csv)
`movies_df = pd.read_csv("movies.csv", index_col="Film")`

In [31]:
```
movies_df = pd.read_csv("files/movies.csv", index_col="Film")
movies_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 77 entries, Zack and Miri Make a Porno to (500) Days of Summer
Data columns (total 7 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Genre              77 non-null     object 
 1   Lead Studio        77 non-null     object 
 2   Audience score %   77 non-null     int64  
 3   Profitability      77 non-null     float64
 4   Rotten Tomatoes %  77 non-null     int64  
 5   Worldwide Gross    77 non-null     object 
 6   Year               77 non-null     int64  
dtypes: float64(1), int64(3), object(3)
memory usage: 4.8+ KB
```

In [32]:
```
movies_df = pd.read_csv("files/movies.csv", index_col="Film")
movies_df.shape
```

Out[32]:
```
(77, 7)
```

```
In [33]: movies_df.head()
```

Out[33]:

Film	Genre	Lead Studio	Audience score %	Profitability	Rotten Tomatoes %	Worldwide Gross	Year
Zack and Miri Make a Porno	Romance	The Weinstein Company	70	1.747542	64	$41.94	2008
Youth in Revolt	Comedy	The Weinstein Company	52	1.090000	68	$19.62	2010
You Will Meet a Tall Dark Stranger	Comedy	Independent	35	1.211818	43	$26.66	2010
When in Rome	Comedy	Disney	44	0.000000	15	$43.04	2010
What Happens in Vegas	Comedy	Fox	72	6.267647	28	$219.37	2008

```
In [34]: movies_df.tail()
```

Out[34]:

Film	Genre	Lead Studio	Audience score %	Profitability	Rotten Tomatoes %	Worldwide Gross	Year
Across the Universe	romance	Independent	84	0.652603	54	$29.37	2007
A Serious Man	Drama	Universal	64	4.382857	89	$30.68	2009
A Dangerous Method	Drama	Independent	89	0.448645	79	$8.97	2011
27 Dresses	Comedy	Fox	71	5.343622	40	$160.31	2008
(500) Days of Summer	comedy	Fox	81	8.096000	87	$60.72	2009

Discussion

How might you *read in* your data and how *why* might you use Pandas and these commands?

Optional

Getting to you know your data - visually

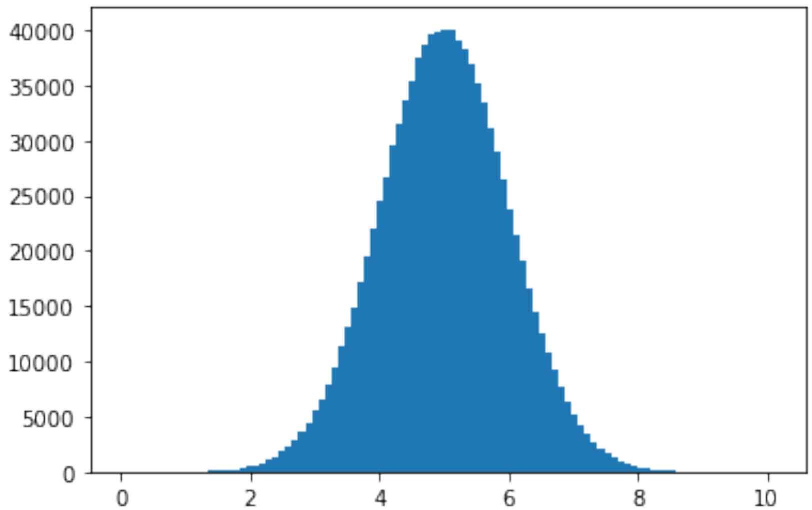
Next class we look at data visualization. This section is an optional teaser integrating the exploration and analysis possible with pandas with another language of science, graphics.

After you've ingested and gotten to know your data, you'll want to explore it - checking the data distribution, domain and range, and test your data against some specific test or idea.

Normal Distribution

In [35]:

```
import numpy
import matplotlib.pyplot as plt

x = numpy.random.normal(5.0, 1.0, 1000000)

plt.hist(x, 100)
plt.show()
```

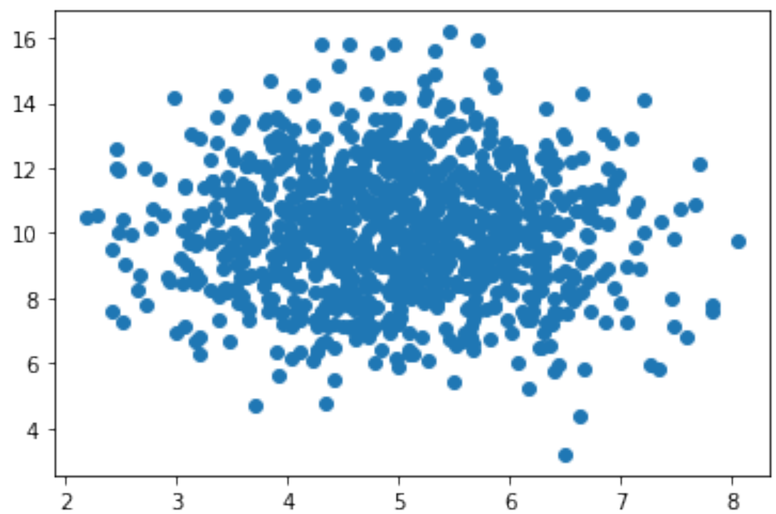


Scatter Plot

Two arrays with 1000 random numbers; first array has the mean set to 5.0 with std dev of 1.0; 2nd array has mean set to 10 with std dev 2.0

In [36]:

```
# see also https://matplotlib.org/stable/users/dflt_style_changes.html

import numpy
import matplotlib.pyplot as plt

x = numpy.random.normal(5.0, 1.0, 1000)
y = numpy.random.normal(10.0, 2.0, 1000)

plt.scatter(x, y)
plt.show()
```



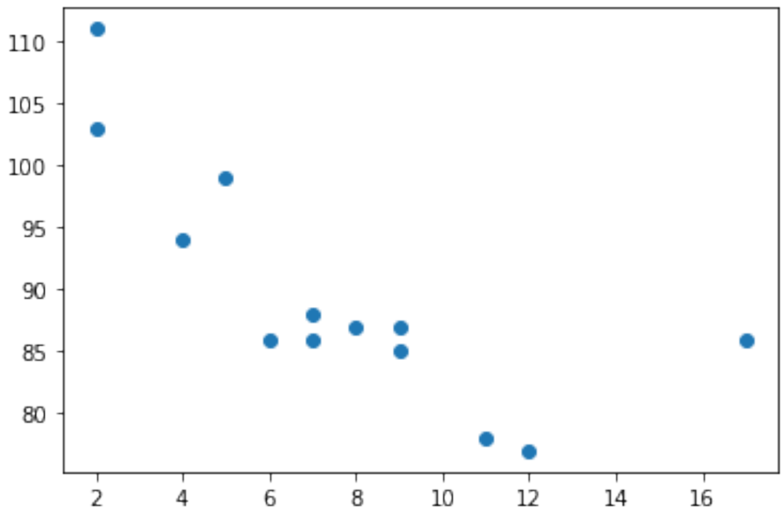
Linear Regression

Here the x-axis represents some value, say age, and y is the height in millimeters.

2nd plot includes the correlation coefficient (*r*).
[See the SciPy.org Tutorial (https://docs.scipy.org/doc/scipy/reference/tutorial/index.html)](https://docs.scipy.org/doc/scipy/reference/tutorial/index.html)

In [37]:

```
import matplotlib.pyplot as plt

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

plt.scatter(x, y)
plt.show()

# can you determine r and r^2?
```



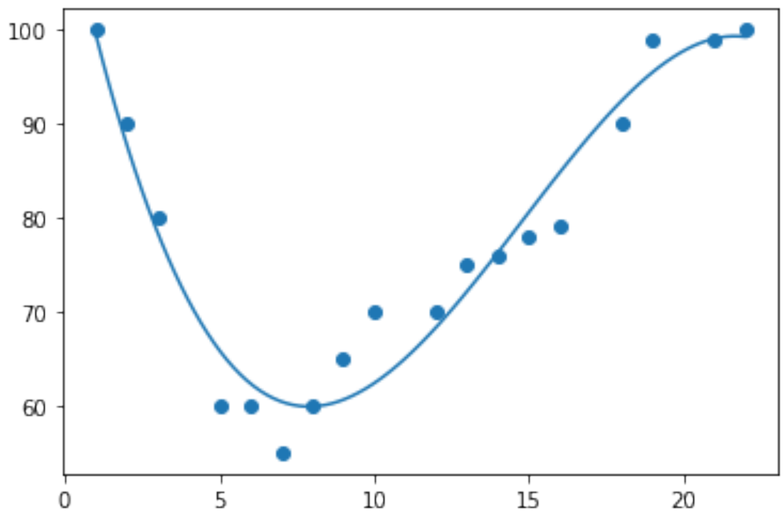
In [38]:

```
import matplotlib.pyplot as plt
from scipy import stats

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

slope, intercept, r, p, std_err = stats.linregress(x, y)

print(r)
```

```
-0.758591524376155
```

In [39]:

```
import numpy
import matplotlib.pyplot as plt

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

myline = numpy.linspace(1, 22, 100)

plt.scatter(x, y)
plt.plot(myline, mymodel(myline))
plt.show()
```



Example: In this example, read data from a .csv file into a DataFrame to measure carbon dioxide. The x var is independent; y is dependent.

The human_subjects.csv file represents a paired-sample groups of people who are overweight, exercised a set number of hours, and to predict their blood oxygen saturation.

In [40]:

```
import pandas
from sklearn import linear_model

df = pandas.read_csv("files/human_subjects.csv")

X = df[['hours', 'weight']]
y = df['O2']

regr = linear_model.LinearRegression()
regr.fit(X, y)

predictedO2 = regr.predict([[25, 190]])

print(predictedO2)
```

```
[95.54921072]
```

Review

This section demonstrates a kind of first on-the-job looking at our data.

Pandas Exploration ... and Review
Say you're working with some data and you want to explore using a Series or a DataFrame...

Series

In [41]:
```
import pandas as pd
s = pd.Series([100,500,400,300,222])
print(s)
# lets see the data as a list by index
print( list(s.index) )
```

```
0    100
1    500
2    400
3    300
4    222
dtype: int64
[0, 1, 2, 3, 4]
```

Labeling of data

makes it a lot easier to keep track of things.

In [42]:

```
s.index = ['Tom','Jane','Ming','Felicia','Toby']
s.name = "Cousins"
print(s)
```

```
Tom         100
Jane        500
Ming        400
Felicia     300
Toby        222
Name: Cousins, dtype: int64
```

In [43]:

```
""" or all at once """
s2 = pd.Series(['02334','19284','11111','96823','55555'],
               name='ZipCodes',
               index=['Providence','New Jersey','Delaware','Kansas City','Selma'])
print(s2)
print("")
print("Where's Kansas City?")
print(s2['Kansas City'])
```

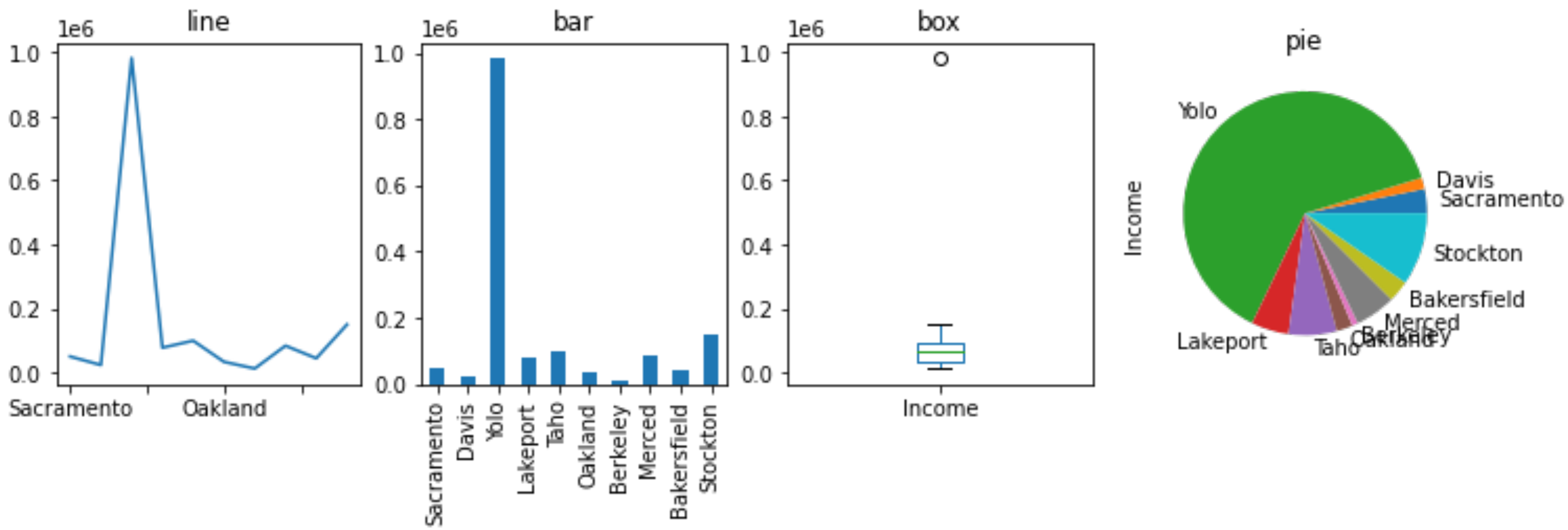
```
Providence     02334
New Jersey     19284
Delaware       11111
Kansas City    96823
Selma          55555
Name: ZipCodes, dtype: object

Where's Kansas City?
96823
```

```
In [44]:
s3 = pd.Series([50000,23423,982874,77558,99382,33423,12345,
                83728,44412,150000],
               name='Income',
               index=['Sacramento','Davis','Yolo',
                      'Lakeport','Taho','Oakland',
                      'Berkeley','Merced','Bakersfield','Stockton'])
print("_"*50,"\n",s3)
print("\nMedian: ", s3.median())
print("Mean: ",s3.mean())
print("STD: ",s3.std())
print("\nWho has the most and least?")
print("The least: $", s3.min(), " and the most $", s3.max())
```

```
__________________________________________________ 
 Sacramento      50000
Davis           23423
Yolo           982874
Lakeport        77558
Taho            99382
Oakland         33423
Berkeley        12345
Merced          83728
Bakersfield     44412
Stockton       150000
Name: Income, dtype: int64

Median:  63779.0
Mean:  155714.5
STD:  293496.96320073836

Who has the most and least?
The least: $ 12345  and the most $ 982874
```

In [45]:
```python
import matplotlib.pyplot as plt

fig, axes = plt.subplots(1, 4, figsize=(12,3))
s3.plot(ax=axes[0], kind='line', title='line')
s3.plot(ax=axes[1], kind='bar', title='bar')
s3.plot(ax=axes[2], kind='box', title='box')
s3.plot(ax=axes[3], kind='pie', title='pie')
```

Out[45]: `<AxesSubplot:title={'center':'pie'}, ylabel='Income'>`



Data Frames

In [46]:
```python
import pandas as pd

# make some data
df = pd.DataFrame([[100000, "London"],
    [500000,"Paris"],
    [250000, "Rome"],
    [200000, "New_York"]])
```

In [47]:

```python
# more useful to add column names that make sense:
df.index = ["UK","France","Italy","USA"]
df.columns = ['Good_Bakeries', 'Country']
""" Ouput some tests """
print(df)

print("\n","-"*40)
print("Just the bakeries and cities")
print(df.Good_Bakeries)

print("\n","-"*40)
print("What about USA? (using .loc)")
print( df.loc["USA"] )

print("\n","-"*40)
print("Can we compare 2 countries? (using .loc)")
print( df.loc[["UK", "USA"]])

print("\n","-"*40,"\nLearn about the data frame ... ")
print(df.info())
print("\nMay be silly but how about some stats? mean,std, median, min, max, etc.?")
print(df.mean())
```

```
        Good_Bakeries   Country
UK             100000    London
France         500000     Paris
Italy          250000      Rome
USA            200000  New_York

 ----------------------------------------
Just the bakeries and cities
UK        100000
France    500000
Italy     250000
USA       200000
Name: Good_Bakeries, dtype: int64

 ----------------------------------------
What about USA? (using .loc)
Good_Bakeries      200000
```

```
Country          New_York
Name: USA, dtype: object

 ----------------------------------------
Can we compare 2 countries? (using .loc)
     Good_Bakeries   Country
UK          100000    London
USA         200000  New_York

 ----------------------------------------
Learn about the data frame ...
<class 'pandas.core.frame.DataFrame'>
Index: 4 entries, UK to USA
Data columns (total 2 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Good_Bakeries  4 non-null      int64
 1   Country        4 non-null      object
dtypes: int64(1), object(1)
memory usage: 256.0+ bytes
None

May be silly but how about some stats? mean,std, median, min, max, etc.?
Good_Bakeries    262500.0
dtype: float64
```

Reading some data for demo

Let's get some fake real data from a .csv file (cities.csv) ... to imagine we've gottan real files from work.
NB: in our source file, strings should be in double-quotes, not single ones.

The fake data look like this:

```
Rank,City,Country,Population,CensusDate
1,London,United Kingdom,"9,000,000",June 2014
2,Berlin,Deutschland,"4,000,000",June 2014
3,Paris,République française,"1,500,000",Aug 2014
4,Marseilles,République française,"2,500,000",Aug 2014
5,München,Deutschland,"2,200,100",April 2014
6,香港,中国,"7,577,231",Oct 2021
```

In [48]:
```python
df_pop = pd.read_csv("files/cities.csv", delimiter=",", encoding="utf-8", header=0)
print("First some info about our objects from the file and then the head()")
df_pop.info()
print("")
print(df_pop.head())
```

```
First some info about our objects from the file and then the head()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Rank        6 non-null      int64
 1   City        6 non-null      object
 2   Country     6 non-null      object
 3   Population  6 non-null      object
 4   CensusDate  6 non-null      object
dtypes: int64(1), object(4)
memory usage: 368.0+ bytes

   Rank        City               Country Population  CensusDate
0     1      London        United Kingdom  9,000,000   June 2014
1     2      Berlin           Deutschland  4,000,000   June 2014
2     3       Paris  République française  1,500,000    Aug 2014
3     4  Marseilles  République française  2,500,000    Aug 2014
4     5     München           Deutschland  2,200,100  April 2014
```

In [49]:
```python
""" note that df.head(n) is the same as df[:n] """
```

Out[49]:
```
' note that df.head(n) is the same as df[:n] '
```

In [50]:

```python
print("\n","-"*40,"\nNot happy with this messy data.  Let's clean it up a bit by removing the
df_pop["NumericPopulation"] = df_pop.Population.apply(lambda x: int(x.replace(",", "")))

df_pop["Country"] = df_pop["Country"].apply(lambda x: x.strip())
# and confirm
print(df_pop.head())
```

```
 ----------------------------------------
Not happy with this messy data.  Let's clean it up a bit by removing the comma; use apply to co
nvert strings to ints; make a new column
   Rank        City               Country Population  CensusDate  \
0     1      London        United Kingdom  9,000,000   June 2014
1     2      Berlin           Deutschland  4,000,000   June 2014
2     3       Paris  République française  1,500,000    Aug 2014
3     4  Marseilles  République française  2,500,000    Aug 2014
4     5     München           Deutschland  2,200,100  April 2014

   NumericPopulation
0            9000000
1            4000000
2            1500000
3            2500000
4            2200100
```

In [51]:

```python
print("\n","-"*50,"Lets index the data by the city name.  Change index by \"set_index()\" and
print("Sort by country and the city ... so set sort_index(level=0) (meaning first index); rep
df_pop2 = df_pop.set_index("City")
df_pop2 = df_pop2.sort_index()
print("just city: ", df_pop2.head())
```

```
 -------------------------------------------------- Lets index the data by the city name.  Chan
ge index by "set_index()" and sort.
Sort by country and the city ... so set sort_index(level=0) (meaning first index); replace 0 wi
th whatever int you want
just city:              Rank                Country Population  CensusDate  \
City
Berlin         2            Deutschland  4,000,000   June 2014
London         1         United Kingdom  9,000,000   June 2014
Marseilles     4  République française  2,500,000    Aug 2014
München        5            Deutschland  2,200,100  April 2014
Paris          3  République française  1,500,000    Aug 2014

            NumericPopulation
City
Berlin                4000000
London                9000000
Marseilles            2500000
München               2200100
Paris                 1500000
```

In [52]:

```
print("now country and city")
df_pop3 = df_pop.set_index(["Country", "City"]).sort_index(level=0)
df_pop3.head()
```

```
now country and city
```

Out[52]:

		Rank	Population	CensusDate	NumericPopulation
Country	**City**				
Deutschland	**Berlin**	2	4,000,000	June 2014	4000000
	München	5	2,200,100	April 2014	2200100
République française	**Marseilles**	4	2,500,000	Aug 2014	2500000
	Paris	3	1,500,000	Aug 2014	1500000
United Kingdom	**London**	1	9,000,000	June 2014	9000000

In [53]:

```
print("\nGet some counts by Country: how many cities appear for each country?")
city_count = df_pop.Country.value_counts()
print(city_count.head())

print("\nDropping, grouping, reducing ... Drop a column, groupby another column ... and can a
df_pop5 = (df_pop.drop("Rank", axis=1)
    .groupby("Country").sum()
    .sort_values("NumericPopulation",
    ascending=False))

print(df_pop5.head())
```

```
Get some counts by Country: how many cities appear for each country?
République française    2
Deutschland             2
United Kingdom          1
中国                     1
Name: Country, dtype: int64

Dropping, grouping, reducing ... Drop a column, groupby another column ... and can apply a redu
ction (like sum, mean...)
                      NumericPopulation
Country
United Kingdom                  9000000
中国                             7577231
Deutschland                     6200100
République française            4000000
```

```
<hr />
<p>The section above is offered as a way of thinking about your data - exploring them and
asking questions of them, useful prepatory activities before further analysis or ingesting
the wrong data!</p>
```

End of the notebook.