Advanced Pandas:
with Brett Vanderblock

(LinkedIn Learning)

Below are the main new ideas from the course taken from the Jupyter Notebooks

01-02: Dataframes

→ df # Each column is a series
→ df['score'] # To return one column of the dataframe, can also call df.score
→ df['name_city'] = df['name'] + '_' + df['city'] # Create a new column with the name
→ df[df['score']>90] # Return only records where the condition score > 90 is true
→ iris.shape # Tells the dimensionality of our data, rows and columns
→ iris.**head**(3) # View first three rows
→ iris.**tail**(3) # View the last three rows

01-03: Top Functions

→ iris.dtypes # Get the data types for each column
→ iris.loc[3:5] # ***.loc()*** allows you to subset data, here row idx 3, 4, 5
→ iris.loc[3,'sepal_length'] #***.loc()*** given the row index and column name
→ iris.iloc[3,0] # With ***.iloc(),*** pass the row and column index
→ iris.**to_csv**('iris-output.csv',index=**False**) # export to csv

01-04: Customizing Options

→ pd.**set_option**('max_rows', 2) # Use max_rows to limit the number of rows returned
→ pd.**set_option**('max_columns', 2) # Limit the max number of columns
→ pd.options.display.float_format = '{:,.2f}'.format # suppress scientific notation

02-1: Data Types

→ planets.**mean**() # called on entire set, returns average for each column
→ planets['number'][0]/planets['mass'][0] # dividing int column by float column
→ planets['number'][0].**astype**(float) # change datatype with astype(type_desired)
→ planets['year'][0].**astype**(str) # change year to a string object
→ planets['year_dt'] = pd.**to_datetime**(planets['year'], format='%Y')
↳ add year in datetime format, int year has been changed to a datetime with first day of the year for each year.

02-02: Strings

→ names = names.str.**replace**(';',',') # replacing ; with ,
→ names.str.**len**() # get length of the strings in the series
→ names = names.str.**strip**() # strips white space from all strings in series

→ names = names.str.**lower**() # makes all strings in series all lowercase
→ names = names.str.**split**(', ') # puts a space after comma, creating tuple for each
→ names = pd.**Series**([i[::-1] **for** i **in** names]) # list comp switching name order first and last
→ names = [' '.**join**(i) **for** i **in** names] # This joins first and last names without comma

02-03: Working with Dates

→ daterange = pd.**period_range**('1/1/2020', freq='30d', periods=4)
↳ period_range will generate a series of dates, given the starting date, then frequency, and number of periods. This will give a series 4 dates, starting Jan 1, 2020, separated by 30 days.

→ Turn the above into a dataframe
date_df = pd.**DataFrame**(data=daterange,columns=['sample date'])

date_df['date difference'] = date_df['sample date'].**diff**(periods=1)
↳ ***.diff()*** will calculate the difference from a prior date
↳ ***.diff()*** is a method that calculates the difference between each row and the previous row
↳ it takes a parameter called periods, which is the number of periods to shift for calculating the difference
↳ it is useful for calculating the change in a value over time

→ Converting a date to the first day of the month. [M] gives this setting
date_df['first of month'] = date_df['sample date'].values.**astype**('datetime64[M]')

→ "sample date" column is period[30D] datatype. This will change it to datetime64.
↳ this helps with using further transformations
date_df['sample date'] = date_df['sample date'].dt.**to_timestamp**()

→ You can easily just subtract dates:
date_df['sample date'] - date_df['first of month']

→ Specify a time span to add or subtract from a date with the Timedelta func
date_df['sample date'] - pd.**Timedelta**('30 d')

→ dt.day_name will return the day of the week for a date
date_df['sample date'].dt.**day_name**()

02-04: Working with Missing Data

→ default Panda parameters will usually deal well with missing or null values
→ sums will treat null as zero, and means will ignore null values
→ ***.cumsum()*** skips nulls, unless set to false: temps['temperature_f'].**cumsum**(skipna=**False**)
→ be mindful when aggregating using ***grouby().*** The default is to exclude any records that have null values for any dimensions you are grouping by
↳ can specify to retain NA dimensions in grouping
temps.**groupby**(by=['measurement_type'],dropna=**False**).**max**()
→ to prevent ***groupby()*** from dropping values, pass with .dropna equal to false
Using `***isna()***` to identify null values in a dataframe
temps.**isna**() # Returns true for any cells with a missing value
→ the most simple way to get rid of null values is to drop records with null values using ***.dropna()***

↳ Consider carefully before utilizing
↳ It will drop ANY rows that contain null in ANY column
↳ drop rows with null using ***axis=0*** (default)

```
temps.dropna()
```

→ To drop any columns with null values, pass axis = 1 in ***.dropna()***
↳ drop columns with null using axis=1

```
temps.dropna(axis=1)
```

→ You can fill null values by using ***.fillna()***, often done with passing a zero to fill in for the null

```
temps.fillna(0)
```

→ The pad method will carry over values from a prior row, as not to completely mess up data such as averages
↳ Remember this is creating data out of thin air and could also be troublesome

```
temps.fillna(method='pad')
```

→ ***.interpolate()*** can be used instead, and by default, it will create a straight line estimate for the missing value

```
temps.interpolate()
```

02-05: .apply(), .map(), .applymap()

→ use these instead of looping through rows to make changes
→ ***.apply()*** allows you to use functions to alter values along an axis in a dataframe or series
→ lambda functions allow you to create the function right inside the ***.apply()*** statement without needing to create a function in advance

```
df['Profit'] = df.apply(lambda x: 'Profit' if x['Revenue']>x['Cost'] else 'Loss',axis=1)
```

↳ the lambda will put either “profit” or “loss” in the Profit column being created based on whether or not the revenue is greater than the cost column for each row

→ ***.map()*** can be used to substitute the value in a series (series only)using either a function, dictionary, or another series

```
team_map = {"One":"Red","Two":"Blue"}
df['Team Color'] = df['Team'].map(team_map)
```

↳ This will go through each row, and if the “Team” column has “One” in it, it will put “Red” in the column being created and for “Two” will put “Blue

→ ***.applymap()*** will apply a function to each element in the dataframe

```
df.applymap(lambda x: len(str(x)))
```

↳ this will return the length of each string in the dataframe as the dataframe but with the length values in each field instead of the actual values

→ Some applications might be too complicated for these functions, in which case a for-loop is called for

```
new_col = []
    for i in range(0,len(df)):
     rev = df['Revenue'][i]/df[df['Region']==df.loc[i,'Region']]['Revenue'].sum()
     new_col.append(rev)
    df['Revenue Share of Region'] = new_col
    df.sort_values(by='Region')
```

↳ In this case, we are calculating each squad’s revenue as a percent of the region’s overall revenue

03-01: GroupBy

→ In a groupby, you determine the dimensions you want to group by and then specify an aggregation method

species	sepal_length	sepal_width	petal_length	petal_width
setosa	5.8	4.4	1.9	0.6
versicolor	7.0	3.4	5.1	1.8
virginica	7.9	3.8	6.9	2.5

→ The following will groupby the irises by species and apply the maximum aggregation, which will return the max for each measurement, and there are three total species.

iris.**groupby**(['species']).**max**()

↳ can flatten hierarchical index with ***reset_index()***

↳ ***.reset_index()*** is a method that resets the index of a DataFrame so that the index is no longer a range of numbers

↳ ***.reset_index()*** takes a parameter called drop, which is a boolean that determines whether to drop the old index or to add it as a column

↳ It is useful for resetting the index of a DataFrame so that it can be used as a column

	sepal_length			sepal_width
	mean	min	max	count
species				
setosa	5.006	4.3	5.8	50
versicolor	5.936	4.9	7.0	50
virginica	6.588	4.9	7.9	50

→ You can also pass a number of different aggregations to be applied to different variables by passing a dictionary with the variables to be affected along with their associated aggregations

df = iris.**groupby**(['species']).**agg**({'sepal_length':['mean','min','max'],'sepal_width':'count'})

→ Notice the hierarchical indexes in the columns, which you will often want to flatten for simplicity, which can be done with ***.join()*** to concatenate the two levels

```
df.columns = ['_'.join(col).strip() for col in df.columns.values]
df.reset_index()
df
```

	sepal_length_mean	sepal_length_min	sepal_length_max	sepal_width_count
species				
setosa	5.006	4.3	5.8	50
versicolor	5.936	4.9	7.0	50
virginica	6.588	4.9	7.9	50

→ You can also create groupings with the ***.get_group()*** function, which allows you to filter the data according to the group you specify, ex: creating groups by species

groupings = iris.**groupby**(['species'])

groupings.**get_group**('setosa').**head**()

→ on defined groups, you can call aggregate and lambda functions

↳ groupings.**max**() and groupings.**apply**(**lambda** x: x.**max**()) ⇨ these both perform the same function as the max one above.

→ You can also use ***.filter()*** on your groups: groupings.**filter**(**lambda** x: x['petal_length'].**max**() <5)

↳ This will return only the species with a max petal length less than 5, which will only return setosa

df

	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

```
df.pivot(index='foo',
         columns='bar',
         values='baz')
```

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

03-02: Reshaping

→ with the ***.pivot()*** function, you can take, for example, a variable separating your rows and pivot that to your columns.

→ ***.pivot()*** takes two parameters, index and columns, which are the columns to use to index the result DataFrame and the columns to pivot, respectively. It is useful for reshaping a DataFrame so that it can be used for plotting

→ df.**pivot**(index='Region',columns='Team',values='Revenue')

↳ "Team" to columns, use "Region" as index, and "Revenue" as values

	Region	Team	Revenue	Cost
0	North	One	7500	5200
1	West	One	5500	5100
2	East	One	2750	4400
3	South	One	6400	5300
4	North	Two	2300	1250
5	West	Two	3750	1300
6	East	Two	1900	2100
7	South	Two	575	50

Team	One	Two
Region		
East	2750	1900
North	7500	2300
South	6400	575
West	5500	3750

→ to work in the opposite direction, use ***.stack()***, i.e. pivot column labels to rows, but first must set multi-index

df2

		A	B
first	second		
bar	one	1	2
	two	3	4
baz	one	5	6
	two	7	8

MultiIndex

```
stacked = df2.stack()
```

first	second		
bar	one	A	1
		B	2
	two	A	3
		B	4
baz	one	A	5
		B	6
	two	A	7
		B	8

MultiIndex

Region	Team		
North	One	Revenue	7500
		Cost	5200
West	One	Revenue	5500
		Cost	5100
East	One	Revenue	2750
		Cost	4400
South	One	Revenue	6400
		Cost	5300
North	Two	Revenue	2300
		Cost	1250
West	Two	Revenue	3750
		Cost	1300

→ df2 = df.**set_index**(['Region','Team'])
→ stacked = pd.**DataFrame**(df2.**stack**())
→ Now there are individual rows for revenue and cost, but the values for both are contained within the same column
→ This process can be reversed using the ***.unstack()*** function, to which you pass which column you want it to unstack to, i.e. which creates the columns labels, by default, it choose -1 ⇦

		Revenue	Cost
Region	Team		
East	One	2750	4400
	Two	1900	2100
North	One	7500	5200
	Two	2300	1250
South	One	6400	5300
	Two	575	50
West	One	5500	5100
	Two	3750	1300

→ if we choose “Region”: stacked.**unstack**('Region')

	Region	East	North	South	West
Team					
One	Revenue	2750	7500	6400	5500
	Cost	4400	5200	5300	5100
Two	Revenue	1900	2300	575	3750
	Cost	2100	1250	50	1300

→ ***.melt()*** ⇨ allows you to reformat your dataframe to identify columns as “ID variables” (which stay intact), while transforming all other columns, or “measure variables” to the row level.

df3

	first	last	height	weight
0	John	Doe	5.5	130
1	Mary	Bo	6.0	150

df3.melt(id_vars=['first', 'last'])

	first	last	variable	value
0	John	Doe	height	5.5
1	Mary	Bo	height	6.0
2	John	Doe	weight	130
3	Mary	Bo	weight	150

df.melt(id_vars=['Region','Team'], var_name='value type')

	Region	Team	Revenue	Cost
0	North	One	7500	5200
1	West	One	5500	5100
2	East	One	2750	4400
3	South	One	6400	5300
4	North	Two	2300	1250

⇨

	Region	Team	value type	value
0	North	One	Revenue	7500
1	West	One	Revenue	5500
2	East	One	Revenue	2750
3	South	One	Revenue	6400
4	North	Two	Revenue	2300

→ Thus far, all transformations have required that we have unique combinations for the row and column variables we are pivoting.

→ if you have multiple variables for a particular combination of, in this case, region-team, then you would want to use the ***.pivot_table()*** function, to which you give an index and the variables you want pivoted to columns, then designate the values.
→ by default, ***.pivot_table()*** uses mean to aggregate
→ for example, on the right, pivot table has given the average revenue for the teams ⇨

```
df.pivot_table(index='Team',values='Revenue')
```

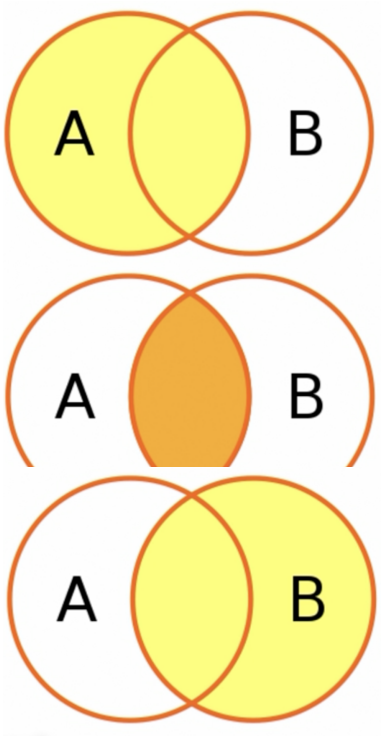
	Revenue
Team	
One	5537.50
Two	2131.25

→ if you add “Region” to columns, then you get the average for each region and team ⇨
→ ***.pivot_table()*** acts as a ***.groupby()*** and then unstacks by the variable you pass to columns
→ ***.pivot_table()*** is the most flexible of the transforming functions and a great place to start

```
df.pivot_table(index='Team',columns='Region',values='Revenue')
```

Region	East	North	South	West
Team				
One	2750	7500	6400	5500
Two	1900	2300	575	3750

03-03: Merging Data



→ joining from left, inner, right, etc ⇨ use ***.merge()***
→ for a left merge, start with the first data frame, the left one.
→ pass in the right dataframe, specifying the join type, in this case left
→ pass the join criteria, here the number column on both dataframes.
→ Only the rows in the passed dataframe that match up to the values in the left dataframe in the column selected to be joined will be brought in.
→ The first dataframe appears entirely and only applicable rows of the second

df1.merge(df2,how='left',on='number')

→ on an ***inner join***, only the data that matches from both sets will be brought in.
→ often the column joined on will have different names in your two data sources
→ use ***left_on =*** and ***right_on =*** to specify the join on criteria

df1.merge(df2,how='inner',left_on='number',right_on='number')

→ ***right join*** is just the reverse of left join.
→ here we will specify how we want the columns to look after joining using the ***suffixes =*** parameter, which will be applied to the right data

df1.merge(df2,how='right',on='number',suffixes=('','_right'))

→ ***pd.concat()*** can be used to union dataframes, essentially stacking dataframes on top of each other.
→ pass the list of the dataframes you want to combine, specify the index (***.reset_index()***), and set ***drop=True***
→ to make sure we do not have duplicate information, we specify ***.drop_ducplicates()***

df3 = pd.concat([df1,df2]).drop_duplicates().reset_index(drop=True)

→ ***pd.concat()*** can also be used to combine dataframes horizontally, by specifying ***axis=1***.

→ df4 = pd.concat([df1,df2],axis=1)

→ ***.append()*** allows you to add rows to your dataframe:

new_row = pd.Series(['Z',26],index=df3.columns)
df3.append(new_row,ignore_index=True)

→ You can also join dataframes using the index values using the ***.join()*** function, and when joining by the index, you do not need to specify the join criteria, but we do need a suffix for repeated column names

```
join_df = pd.DataFrame({'letter': ['F','G', 'H', 'I'],
                        'number': [6, 7, 8, 9]})
df2.join(join_df, rsuffix='_right')
```

03-04: Categorizing and Labeling Data

→ using categories, labels, and buckets
→ This dataframe has species of salmon, their population, and the count recorded for each

	Species	Population	Count
0	Chinook	Skokomish	1208
1	Chum	Lower Skokomish	2396
2	Coho	Skokomish	3220
3	Steelhead	Skokomish	6245
4	Bull Trout	SF Skokomish	8216

→ ***.cut()*** allows creation of bins for numerical data which can be added them to the dataframe
→ here, we are importing numpy to use ***numpy.inf*** for the top number of the highest bucket
→ buckets are given labels as well as numerical specifications
→ create a new column called “Count Category” that gets passed the “Count” column we already have, the bins, and the labels for the bins
⇨ This creates the buckets

	Species	Population	Count	Count Category
0	Chinook	Skokomish	1208	Low Return
1	Chum	Lower Skokomish	2396	Below Avg Return
2	Coho	Skokomish	3220	Below Avg Return
3	Steelhead	Skokomish	6245	Above Avg Return
4	Bull Trout	SF Skokomish	8216	High Return

```
bins = [0, 2000, 4000, 6000, 8000, np.inf]
labels = ['Low Return', 'Below Avg Return', 'Avg Return', 'Above Avg Return', 'High Return']
df['Count Category'] = pd.cut(df['Count'], bins, labels=labels)
```

→ You can also create a dictionary which matches data in your dataframe into groupings

Ex: mapping species to their endangered species statuses and apply the mappings to the species column using ***.map()***

```
fed_status ={"Chinook":"Threatened",
"Chum":"Not Warranted",
"Coho":"Not Warranted",
"Steelhead":"Threatened"}
df['Federal Status'] = df['Species'].map(fed_status)
```

→ Categorical Data Type ⇨ ***.categorical()*** to convert the 'Count Category' column, passing it the column to be converted, the labels from before, and ***ordered = True*** creates a categorical hierarchy, starting with "Low Returns" up to "High Return". This can then be used to sort the data

```
df['Count Category'] = pd.Categorical(df['Count Category'],
                                      ordered=True,
                                      categories=labels)
df.sort_values(by=['Count Category'],ascending=False)
```

	Species	Population	Count	Count Category	Federal Status
4	Bull Trout	SF Skokomish	8216	High Return	NaN
3	Steelhead	Skokomish	6245	Above Avg Return	Threatened
1	Chum	Lower Skokomish	2396	Below Avg Return	Not Warranted
2	Coho	Skokomish	3220	Below Avg Return	Not Warranted
0	Chinook	Skokomish	1208	Low Return	Threatened

→ ***.get_dummies()*** is useful for statistics and machine learning.

→ This makes it easy to perform ***one-hot encoding,*** creating a column for each possibilities and a 0 for False and 1 for True

```
pd.get_dummies(df['Count Category'])
```

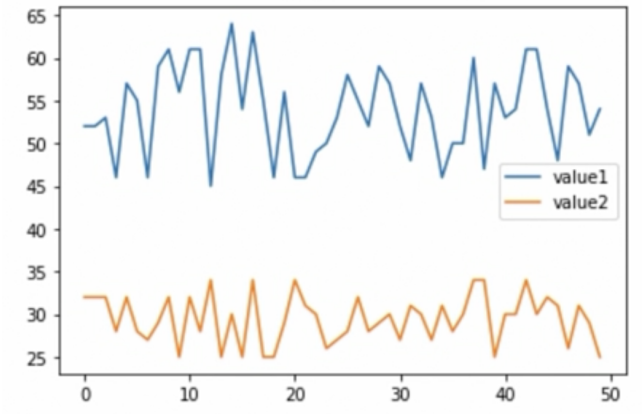
	Low Return	Below Avg Return	Avg Return	Above Avg Return	High Return
0	1	0	0	0	0
1	0	1	0	0	0
2	0	1	0	0	0
3	0	0	0	1	0
4	0	0	0	0	1

04-01: Plotting with Pandas

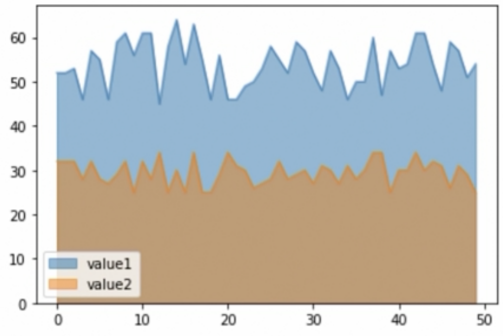
→ using Pandas with Matplotlib. Use more extensive data visualization for non-technical audiences, however
→ using a dataframe of randomly created values 50 rows long

	day	value1	value2
0	1950-01-01	52	32
1	1950-01-02	52	32
2	1950-01-03	53	32

```
ax = date_df.plot();
```

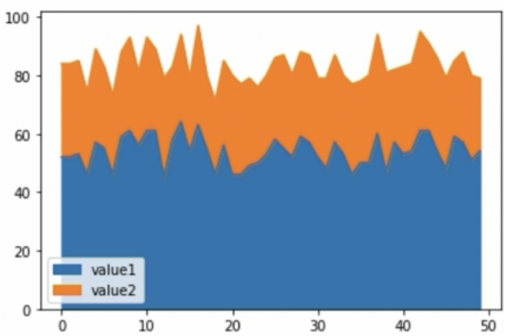



```
date_df.plot.area(stacked=False);
```

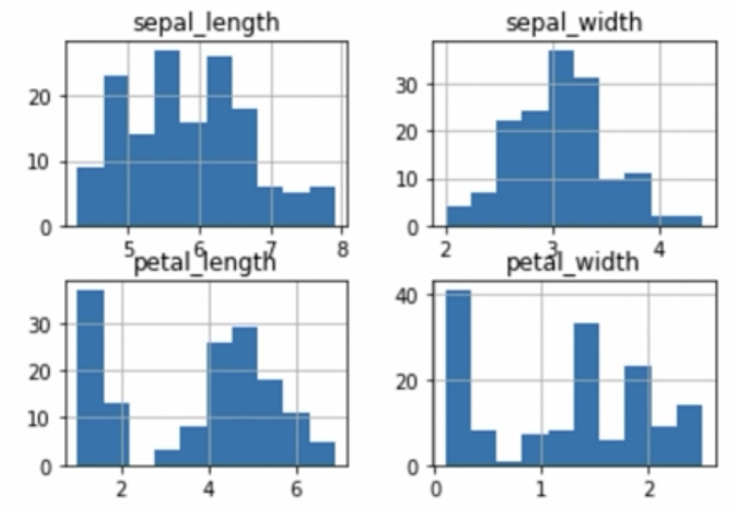


→ area plot with ***stacked=False*** vs ***stacked=True***

```
date_df.plot.area(stacked=True);
```



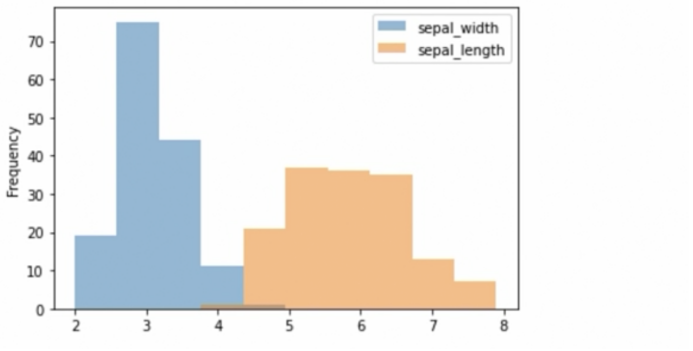
```
iris.hist();
```



⇦ using just ***.hist()***

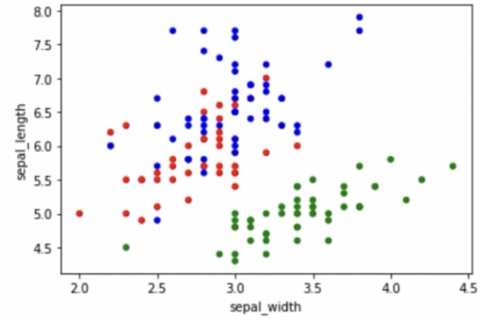
.plot.hist() however, can overlay. Just remember to make the opacity work accordingly by using the alpha parameter

```
iris[['sepal_width','sepal_length']].plot.hist(alpha=0.5);
```



→ in the following example, using .map() to assign a color to each species for the scatterplot

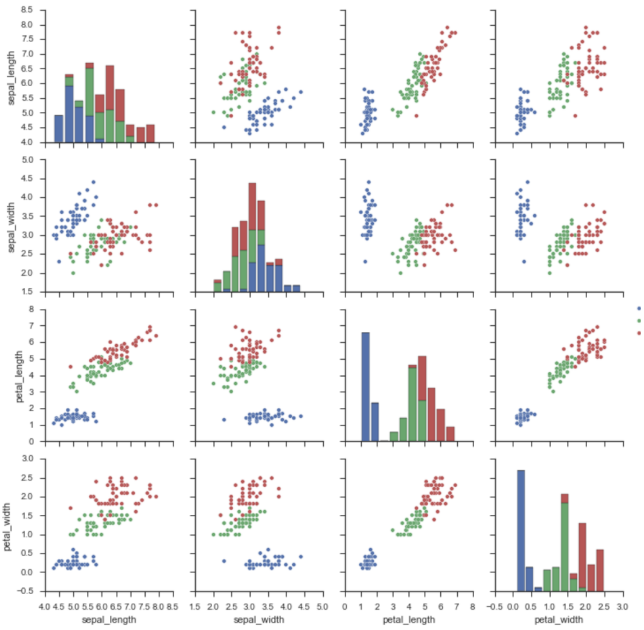
```
colors = {"versicolor":"red","setosa":"green","virginica":"blue"}
iris['colors'] = iris['species'].map(colors)
iris.plot.scatter(x='sepal_width', y='sepal_length', color=iris['colors']);
```



→ ***.scatter_matrix()*** ⇨ generates scatterplots showing the relationship between each of the variables.

→ Along the diagonal, there is a histogram showing how the observations are distributed.

→ very helpful visual to get a quick understanding of the data



04-02: Statistics with Pandas

→ ***.mean()*** called on an entire data set will return the average of all variables, and ***.median()*** and ***.mode()*** work similarly

→ To return the standard deviation for variables, use the ***.std()*** function. This will tell you the extent to which the variables deviate from their mean.

→ ***.describe()*** is the simplest way to get the statistical information about the data

→ ***.corr()*** is important for finding out how the variables you are working with relate and correlate with one another

↳ positive numbers show that as one variable increases, the other tends to do so as well. And negative shows the opposite. The closer to 1 or -1, the closer the relationship.

→ You can automatically draw a heatmap on your correlation matrix. The colors make it fairly clear how strongly and in which direction the variables are correlated.

iris.corr().style.background_gradient(cmap='RdYlGn', axis=None)

05-01: Accelerate EDA with Pandas-Profiling

→ Pandas profiling is important for accelerating exploratory analysis
→ will take dataframe and output an interactive and highly comprehensive profile for each variable in your data. (LITERALLY does all the work for you.)
→ it gives the descriptive statistics for each variable, provides a correlation matrix, and assesses the data quality
→ Much of the proceeding work from this course can be done more easily and extensively with Pandas profiling

```
!pip install pandas-profiling
import pandas as pd
from pandas_profiling import ProfileReport
profile = ProfileReport(iris,title="Iris Data Profile")

profile.to_notebook_iframe()
profile.to_widgets()          ⇦ Use this one for Jupyter
```

⇨ A good idea to always do this first!
⇨ can be output as HTML file
⇨ use ***.to_file("filename.html")***

05-01: GeoData with GeoPandas

⇨ Converting dataframe into a geo dataframe, pointing geometry to our longitude and latitude, and using the ***.points_from_xy()*** to help it read the longitude and latitude

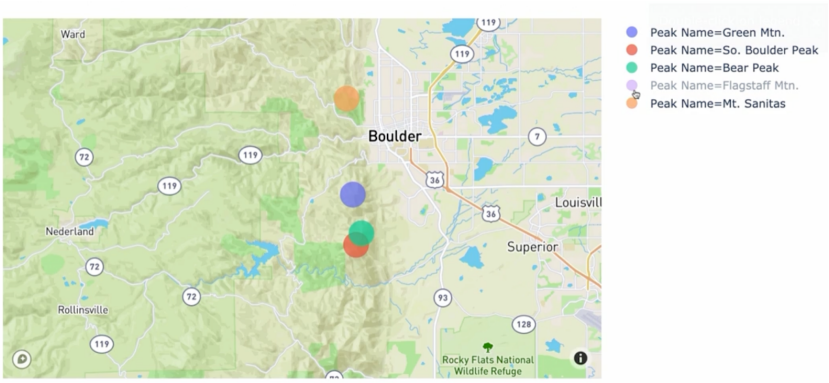
```
peaks = pd.DataFrame(
  {'Peak Name': ['Green Mtn.', 'So. Boulder Peak', 'Bear Peak', 'Flagstaff Mtn.', 'Mt. Sanitas'],
   'Latitude': [39.9821, 39.9539, 39.9603, 40.0017, 40.0360968],
   'Longitude': [-105.3016, -105.2992, -105.2952, -105.3075, -105.3061024]})

gdf = geopandas.GeoDataFrame(
  peaks, geometry=geopandas.points_from_xy(peaks.Longitude, peaks.Latitude))

import plotly.express as px
px.set_mapbox_access_token(token)
gdf['size'] = 65

fig = px.scatter_mapbox(gdf,                  # fig is like a dictionary that organizes the info for plotly
                        lat=gdf.geometry.y,   # lat and lon are pointed to the new geometry fields
                        lon=gdf.geometry.x,
                        color="Peak Name",
                        hover_name="Peak Name",
                        mapbox_style='outdoors',
                        size='size',
                        zoom=10)

fig.show()
```



05-01: Dask and Koalas (Spark) ⇨ Entire courses unto themselves

⇨ These are good when data is so large that pandas starts to feel constraining

⇨ **Dask** = a framework to speed up computing of large amounts of data by parallel computing

→ Very compatible with Pandas

→ Has a dataframe concept like Pandas, almost like a series of Pandas dataframes

⇨ **Spark** = easily called in Python using pyspark

→ Koalas is the way to work this into Pandas/Python

→ standard input:

```
import pandas as pd
import numpy as np
import databricks.koalas as ks
from pyspark.sql import SparkSession
```

→ transition from Pandas to Koalas / Spark:

```
pdf = pd.DataFrame(np.random.randn(6, 4),  columns=list('ABCD'))
kdf = ks.from_pandas(pdf)
```

Check out Python for Data Analysis by Wes McKinney