FROM DATA TO VISUALIZATION

NUMPY, PANDAS, MATPLOTLIB, SEABORN

INSTALL NUMPY

```
pip install numpy
```

Or:

```
conda install numpy
```

NUMPY

POWERFUL N-DIMENSIONAL ARRAYS

Fast and versatile, the NumPy vectorization, indexing, and broadcasting concepts are the de-facto standards of array computing today.

NUMERICAL COMPUTING TOOLS

NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.

INTEROPERABLE

NumPy supports a wide range of hardware and computing platforms, and plays well with distributed, GPU, and sparse array libraries.

PERFORMANT

The core of NumPy is well-optimized C code. Enjoy the flexibility of Python with the speed of compiled code.

EASY TO USE

NumPy's high level syntax makes it accessible and productive for programmers from any background or experience level.

OPEN SOURCE

Distributed under a liberal BSD license, NumPy is developed and maintained publicly on GitHub by a vibrant, responsive, and diverse community.

https://numpy.org/

ARRAY VS LIST

- An **array** is also a data structure that stores a collection of items. Like lists, arrays are **ordered**, **mutable**, enclosed in **square brackets**, and able to store **non-unique** items.
  - Array is via numpy
  - List is python basic

NUMPY CREATE ARRAYS

- A range of an array via np.arange(start, stop, step)

```
np.arange(0,10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.arange(0,11,2)
```

```
array([ 0,  2,  4,  6,  8, 10])
```

- A zero array via np.zeros(number of zeros)
  - A zero array via np.zeros((rows, columns))

```
np.zeros(3)
```

```
array([ 0.,  0.,  0.])
```

```
np.zeros((2,3))
```

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

- An array of ones via np.ones(number of ones)
  - An array of ones via np.ones((rows, columns))

```
np.ones(4)
```

```
array([ 1.,  1.,  1.,  1.])
```

```
np.ones((3,4))
```

```
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

- A unit matrix or identity matrix via np.eye(matrix size)

```
np.eye(4)
```

```
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

LINEARLY SPACED ARRAY

- Return evenly spaced numbers over a specified interval,

  via np.linspace(start, stop, number of points)

```
np.linspace(0,5,10)
```

```
array([ 0.        ,  0.55555556,  1.11111111,  1.66666667,  2.22222222,
        2.77777778,  3.33333333,  3.88888889,  4.44444444,  5.        ])
```

- Question: How to return numbers spaced evenly on a log scale?
  - np.logspace(start, stop, end)

QUESTION

- What is returned of np.logspace(1,2,2)?

A. array([10, 100])

B. array([1, 2])

Note:

```
np.logspace(1,2,2)
```

```
Signature:
np.logspace(
    start,
    stop,
    num=50,
    endpoint=True,
    base=10.0,
    dtype=None,
    axis=0,
)
```

RANDOM NUMBERS np.random

- np.random. + tab
  - np.random.rand → This is linear distribution [0,1)

```
np.random.rand(5)
```

```
array([ 0.92251232,  0.07198415,  0.12872838,  0.99571974,  0.94977206])
```

  - np.random.randn→ This is normal (Gaussian) distribution

```
np.random.randn(2)
```

```
array([ 0.1748905 , -0.91796204])
```

  - np.random.randint → random integer between two numbers

```
np.random.randint(1,100,10)
```

```
array([80, 15, 64, 51, 32, 84, 34, 19, 13, 88])
```

RESHAPE A 1D ARRAY INTO A MULTI DIMENSIONAL

Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension.

- In 2D: arr.reshape(rows,columns)

```
arr=np.arange(1,31)
arr
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30])
```

```
arr.reshape(5,6)
```

```
array([[ 1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18],
       [19, 20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29, 30]])
```

You may check the shape of a matrix/array via

- Arr.shape → Note: no paranthesis at the end!
- Also: np.shape(arr) → less commonly used!

```
arr=arr.reshape(2,4)
```

```
arr.shape
```

```
(2, 4)
```

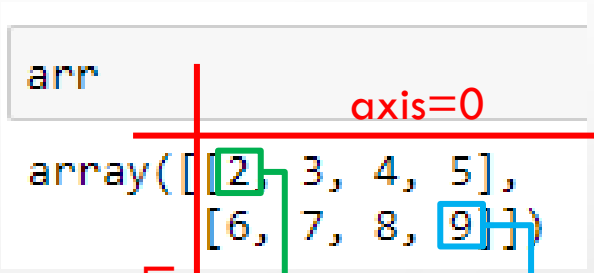
A FEW USEFUL BUILT-IN FUNCTIONS

- np.max(arr) or arr.max()
- np.min(arr)
- np.argmax(arr)
- np.argmin(arr)

```
arr.max()
9

np.max(arr)
9

arr.min()
2

np.min(arr)
2
```

```
arr.max(axis=0)
array([6, 7, 8, 9])

np.max(arr, axis=1)
array([5, 9])

arr.min(axis=1)
array([2, 6])

np.min(arr,axis=0)
array([2, 3, 4, 5])
```

```
np.argmax(arr)
7

np.argmin(arr)
0
```

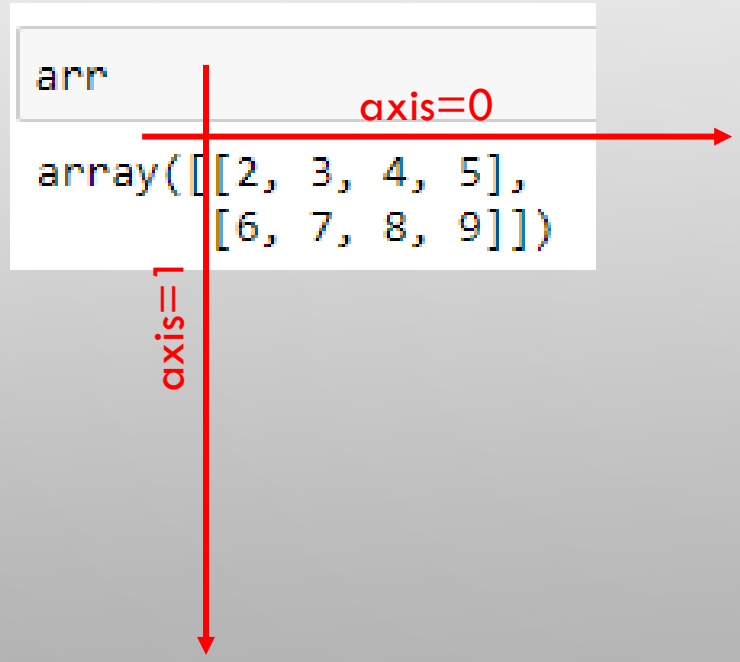


Index of the max value = 7

Index of the min value = 0

A FEW USEFUL BUILT-IN FUNCTIONS

- Arr.sum(axis=?)




```
arr.sum()
```

```
44
```

```
arr.sum(axis=0)
```

```
array([ 8, 10, 12, 14])
```

```
arr.sum(axis=1)
```

```
array([14, 30])
```

INDEXING AND SLICING

```
arr
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30])
```

```
arr[1]
```

Array[where index is equal to 1]

```
2
```

```
arr[1:11]
```

Array[where index is equal to 1 to 11]

```
array([ 2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

```
arr[1:11:2]
```

Array[where index is equal to 1 to 11 with the steps of 2]

```
array([ 2,  4,  6,  8, 10])
```

INDEXING AND SLICING

- Arr[where some condition is True]

```
arr=np.arange(1,10)
arr
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
arr[arr>3]
```

```
array([4, 5, 6, 7, 8, 9])
```

QUESTION

- Let's say arr is defined as:

```
arr
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30])
```

- Now, I am going to create a slice of it, and set all the parameters of the slice to 999
- Note: I am only changing the values in the slice and not the original array

```
slice_of_array = arr[2:7]
slice_of_array
```

```
array([3, 4, 5, 6, 7])
```

```
slice_of_array[:]=999
```

What is the outcome if now, I call the original arr? Do the values in it also change?

A.

```
arr
```

```
array([  1,   2, 999, 999, 999, 999, 999,   8,   9,  10,  11,  12,  13,
        14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,  26,
        27,  28,  29,  30])
```

B.

```
arr
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30])
```

arr.copy()

- So, make a copy of it if you don't want to change the parameters in the original array!

```
arr=np.arange(1,31)
arr_copy = arr.copy()
arr
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30])
```

```
slice_of_array = arr_copy[2:7]
print(slice_of_array)
slice_of_array[:]=999
print(slice_of_array)
```

```
[3 4 5 6 7]
[999 999 999 999 999]
```

```
arr
```

```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30])
```

INDEXING AND SLICING

- arr[row, column]

Use this method!

```
arr
```

columns

	0	1	2	3	4	5
0	1	2	3	4	5	6
1	7	8	9	10	11	12
2	13	14	15	16	17	18
3	19	20	21	22	23	24
4	25	26	27	28	29	30

rows

```
arr[1,2]
```

```
9
```

```
arr[1][2]
```

Less commonly used

Or has a different meaning?!

```
9
```

```
arr[:3,2:]
```

```
array([[ 3,  4,  5,  6],
       [ 9, 10, 11, 12],
       [15, 16, 17, 18]])
```

ARRAY OPERATIONS

More information on numpy universal functions: https://numpy.org/doc/stable/reference/ufuncs.html

```
arr1=np.array([1,2,3,4,5])
arr2=np.array([0,1,2,3,4])
```

```
arr1+arr2
```

```
array([1, 3, 5, 7, 9])
```

```
arr1+2
```

```
array([3, 4, 5, 6, 7])
```

```
arr1/arr2
```

```
<ipython-input-88-e677913e0f55>:1: RuntimeWarning: divide by zero encountered in true_divide
  arr1/arr2
```

```
array([       inf, 2.        , 1.5       , 1.33333333, 1.25      ])
```

```
arr**2
```

```
array([[   0,    1,    4,    9,   16,   25,   36,   49,   64,   81],
       [ 100,  121,  144,  169,  196,  225,  256,  289,  324,  361],
       [ 400,  441,  484,  529,  576,  625,  676,  729,  784,  841],
       [ 900,  961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521],
       [1600, 1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401]],
      dtype=int32)
```

```
np.log(arr2)
```

```
<ipython-input-90-0fb2bd2eb7ae>:1: RuntimeWarning: divide by zero encountered in log
  np.log(arr2)
```

```
array([      -inf, 0.        , 0.69314718, 1.09861229, 1.38629436])
```

```
np.sqrt(arr2)
```

```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ])
```

type(something) vs .dtype

- type of the object, if only one object parameter is passed
- Dtype: type of data
  - A data type object (an instance of numpy.dtype class) describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted.

```
arr.dtype
```

```
dtype('int32')
```

```
type(arr)
```

```
numpy.ndarray
```

STRETCH YOUR CODING MUSCLES!

Part_II_Numpy

INSTALL PANDAS

```
pip install pandas
```

Or:

```
conda install pandas
```

IMPORT PANDAS

```
import pandas as pd
```

pd.Series()

- Similar to numpy array but more powerful

```
import pandas as pd
```

```
lst = [1, 2, 3]
label = ['a', 'b', 'c']
```

```
pd.Series(lst)
```

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

```
pd.Series(data=lst, index=label)
```

```
a    1
b    2
c    3
dtype: int64
```

```
ser1 = pd.Series(data=lst, index=label)
```

```
ser1['a']
```

```
1
```

QUESTION

- Let's say we use Arduino to read some inputs.
- Which is the outcome of ser_a + ser_b?

```
ser_a=pd.Series(data=[2.2, 1.3, 2, 5],
                index=['pin3','pin4','pin5','pin6'])
ser_a
```

```
pin3    2.2
pin4    1.3
pin5    2.0
pin6    5.0
dtype: float64
```

```
ser_b=pd.Series(data=[1.2, 1.1, 0.5, 2],
                index=['pin3','pin4','pin6','pin7'])
ser_b
```

```
pin3    1.2
pin4    1.1
pin6    0.5
pin7    2.0
dtype: float64
```

A.

```
ser_a+ser_b
```

```
pin3    3.4
pin4    2.4
pin5    NaN
pin6    5.5
pin7    NaN
dtype: float64
```

B.

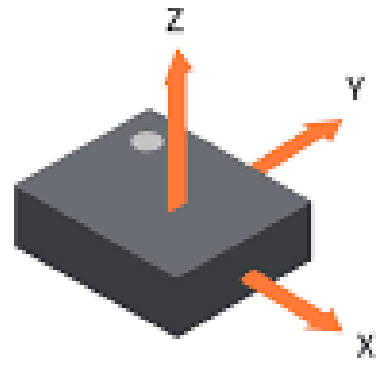
```
pin3    3.4
pin4    2.4
pin5    2.0
pin6    5.5
pin7    2.0
dtype: float64
```

pd.DataFrame

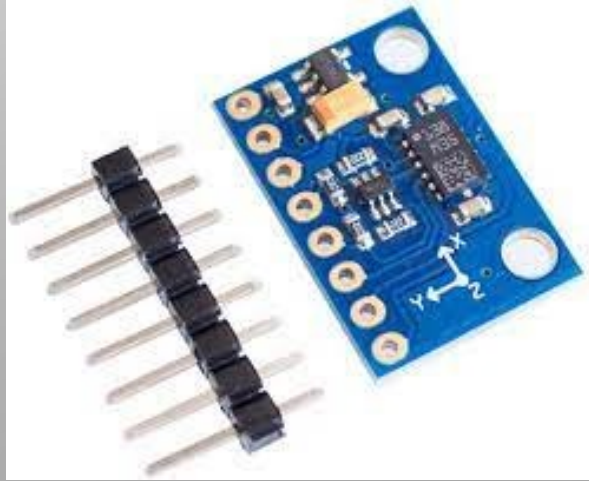
- Think about an accelerometer
  - This is a collection of dataseries in x,y,z coordinate

```
pd.DataFrame(data=position,
             index=['pnt1','pnt2','pnt3','pnt4','pnt5'],
             columns=['X','Y','Z']
            )
```

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721



Accelerometer sensing axis orientation



pd.DataFrame

```
from numpy.random import randn

np.random.seed(100)
position = randn(5,3)
position
```

```
array([[-1.74976547,  0.3426804 ,  1.1530358 ],
       [-0.25243604,  0.98132079,  0.51421884],
       [ 0.22117967, -1.07004333, -0.18949583],
       [ 0.25500144, -0.45802699,  0.43516349],
       [-0.58359505,  0.81684707,  0.67272081]])
```

Note that I used random to create the position matrix

```
df=pd.DataFrame(data=position,
                index=['pnt1','pnt2','pnt3','pnt4','pnt5'],
                columns=['X','Y','Z']
               )
df
```

The entire table is your data frame

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

Each column is data series

```
type(df)
```

```
pandas.core.frame.DataFrame
```

```
df['X']
```

```
pnt1   -1.749765
pnt2   -0.252436
pnt3    0.221180
pnt4    0.255001
pnt5   -0.583595
Name: X, dtype: float64
```

```
type(df['X'])
```

```
pandas.core.series.Series
```

pd.DataFrame, add/remove a column

- df['name'] = ?? → adding a new column
- df.drop(what, axis, inplace=True) → deleting a column/row
  → inplace = True, ensures not to accidentally lose data

```
df['position'] = np.sqrt(df['X']**2 + df['Y']**2 + df['Z']**2)
```

```
df
```

axis=0

axis=1

	X	Y	Z	position
pnt1	-1.749765	0.342680	1.153036	2.123347
pnt2	-0.252436	0.981321	0.514219	1.136281
pnt3	0.221180	-1.070043	-0.189496	1.108973
pnt4	0.255001	-0.458027	0.435163	0.681309
pnt5	-0.583595	0.816847	0.672721	1.208460

```
df.drop('position', axis=1)
```

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

```
df.drop('position', axis=1, inplace=True)
```

```
df
```

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

pd.DataFrame – Calling Columns

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

- df [‘name of one column’]
- df [[‘name of columns you want to call’]] → note two brakets

```
df['X']
```

```
pnt1   -1.749765
pnt2   -0.252436
pnt3    0.221180
pnt4    0.255001
pnt5   -0.583595
Name: X, dtype: float64
```

```
df[['X','Y']]
```

	X	Y
pnt1	-1.749765	0.342680
pnt2	-0.252436	0.981321
pnt3	0.221180	-1.070043
pnt4	0.255001	-0.458027
pnt5	-0.583595	0.816847

pd.DataFrame – Calling Rows

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

- df.loc['name of the row']
- df.iloc[index of the row]

```
df.loc['pnt2']
```

```
X   -0.252436
Y    0.981321
Z    0.514219
Name: pnt2, dtype: float64
```

```
df.iloc[1]
```

```
X   -0.252436
Y    0.981321
Z    0.514219
Name: pnt2, dtype: float64
```

pd.DataFrame – Calling value in a given column and row

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

- df.loc['name of row', 'name of column']
- df.iloc[index of row, index of column]

```
df.loc['pnt1', 'X']
```

```
-1.7497654730546974
```

```
df.iloc[0, 0]
```

```
-1.7497654730546974
```

- df.loc[['names of rows'], ['names of columns']]
- df.iloc[[indices of rows], [indices of columns]]

```
df.loc[['pnt1', 'pnt4'],['Y','Z']]
```

	Y	Z
pnt1	0.342680	1.153036
pnt4	-0.458027	0.435163

```
df.iloc[[0,3],[1,2]]
```

	Y	Z
pnt1	0.342680	1.153036
pnt4	-0.458027	0.435163

pd.DataFrame Conditioning

- df[where condition is True]

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

```
df[['X','Y']]
```

	X	Y
pnt1	-1.749765	0.342680
pnt2	-0.252436	0.981321
pnt3	0.221180	-1.070043
pnt4	0.255001	-0.458027
pnt5	-0.583595	0.816847

```
df[df>0]
```

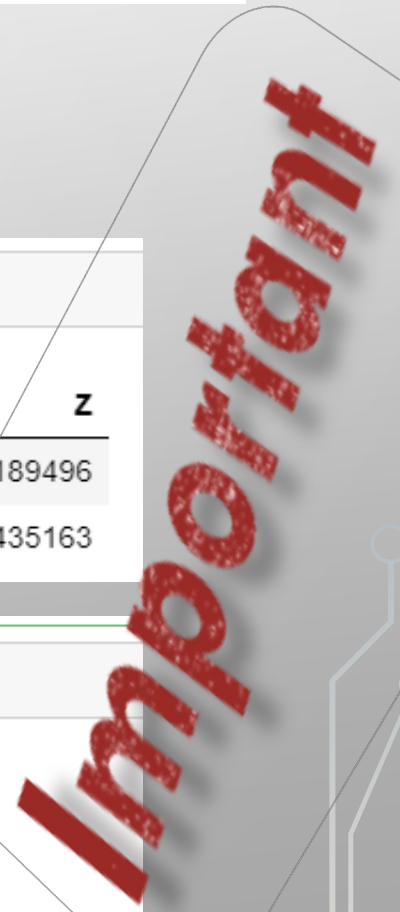
	X	Y	Z
pnt1	NaN	0.342680	1.153036
pnt2	NaN	0.981321	0.514219
pnt3	0.221180	NaN	NaN
pnt4	0.255001	NaN	0.435163
pnt5	NaN	0.816847	0.672721

```
df[df['X']>0]
```

	X	Y	Z
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163

```
df['X'][df['Y']>0]
```

```
pnt1   -1.749765
pnt2   -0.252436
pnt5   -0.583595
Name: X, dtype: float64
```



QUESTION

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

- What is the outcome of

```
df[df['X']>0][['X','Y']]
```

??

```
df[df['X']>0]
```

	X	Y	Z
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163

```
df[df['X']>0][['X','Y']]
```

	X	Y
pnt3	0.221180	-1.070043
pnt4	0.255001	-0.458027

pd.DataFrame – Multiple Conditions

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

- df[(condition1) & (condition2)]
- Note: Panda gets confused if we use 'and' as opposed to '&'
  - Python used 'and' to compare a single Boolean to another
  - Now, that we want a comparison of two series of Booleans we use '&'

```
df[(df['X']>0) & (df['Z']<0)]
```

	X	Y	Z
pnt3	0.22118	-1.070043	-0.189496

pd.DataFrame – Multiple Conditions

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt3	0.221180	-1.070043	-0.189496
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

- df[(condition1) | (condition2)]
- Note: Panda gets confused if we use 'or' as opposed to '|'
  - Python used 'or' to compare a single Boolean to another
  - Now, that we want a comparison of two series of Booleans we use '|'

```
df[(df['Y']>0) | (df['Z']>0)]
```

	X	Y	Z
pnt1	-1.749765	0.342680	1.153036
pnt2	-0.252436	0.981321	0.514219
pnt4	0.255001	-0.458027	0.435163
pnt5	-0.583595	0.816847	0.672721

pd.DataFrame – Re-index-ing

	X	Y	Z	position
pnt1	-1.749765	0.342680	1.153036	2.123347
pnt2	-0.252436	0.981321	0.514219	1.136281
pnt3	0.221180	-1.070043	-0.189496	1.108973
pnt4	0.255001	-0.458027	0.435163	0.681309
pnt5	-0.583595	0.816847	0.672721	1.208460

- Explore the followings:
  - df.reset_index(inplace=True)
  - df.set_index('position', inplace=True)

```
df.reset_index()
```

	index	X	Y	Z	position
0	pnt1	-1.749765	0.342680	1.153036	2.123347
1	pnt2	-0.252436	0.981321	0.514219	1.136281
2	pnt3	0.221180	-1.070043	-0.189496	1.108973
3	pnt4	0.255001	-0.458027	0.435163	0.681309
4	pnt5	-0.583595	0.816847	0.672721	1.208460

```
df.set_index('position')
```

	X	Y	Z
position			
2.123347	-1.749765	0.342680	1.153036
1.136281	-0.252436	0.981321	0.514219
1.108973	0.221180	-1.070043	-0.189496
0.681309	0.255001	-0.458027	0.435163
1.208460	-0.583595	0.816847	0.672721

DATA FRAMES OPERATIONS

- .dropna()
- .fillna()
- .astype()
- .unique()
- .nunique()
- .value_counts()
- .apply(function)
- .head()
- .tail()
- .describe()
- .info()
- .columns
- .index
- .sort_values(by)
- .corr()

.dropna()

- DataFrame.dropna(*, axis=0, how=_NoDefault.no_default, thresh=_NoDefault.no_default, subset=None, inplace=False)[source]
- Remove missing values.
  - thresh: Require that many non-NA values. Cannot be combined with how.

```
df
```

	150 um	250 um	450 um	850 um
0	2.2	NaN	NaN	3.3
1	3.5	4.2	3.2	4.0
2	4.1	NaN	4.5	5.2

```
df.dropna()
```

	150 um	250 um	450 um	850 um
1	3.5	4.2	3.2	4.0

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

	150 um	850 um
0	2.2	3.3
1	3.5	4.0
2	4.1	5.2

```
df.dropna(thresh=3)
```

	150 um	250 um	450 um	850 um
1	3.5	4.2	3.2	4.0
2	4.1	NaN	4.5	5.2

.fillna()

- DataFrame.fillna(value=None, *, method=None, axis=None, inplace=False, limit=None, downcast=None)[source]
- Fill NA/NaN values using the specified method.

```
df.fillna(value=df.mean())
```

	150 um	250 um	450 um	850 um
0	2.2	4.2	3.85	3.3
1	3.5	4.2	3.20	4.0
2	4.1	4.2	4.50	5.2

```
df.dropna(thresh=3)
```

	150 um	250 um	450 um	850 um
1	3.5	4.2	3.2	4.0
2	4.1	NaN	4.5	5.2

```
df['250+450']=df['150 um']+df['250 um']
```

```
df
```

	150 um	250 um	450 um	850 um	250+450
0	2.2	NaN	NaN	3.3	NaN
1	3.5	4.2	3.2	4.0	7.7
2	4.1	NaN	4.5	5.2	NaN

```
df.fillna(0, inplace=True)
```

```
df['250+450']=df['150 um']+df['250 um']
df
```

	150 um	250 um	450 um	850 um	250+450
0	2.2	0.0	0.0	3.3	2.2
1	3.5	4.2	3.2	4.0	7.7
2	4.1	0.0	4.5	5.2	4.1

- Let's say we have the following data:

df

Observed fluxes

Galactic Longitude

Evolutionary Flag

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

.head() vs .tail()

df

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

df.head()

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0

df.tail()

	150um	250um	350um	500um	long	Evolution
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

.info() and .describe()

```
df.info()
```

df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   150um      10 non-null     float64
 1   250um      10 non-null     float64
 2   350um      10 non-null     float64
 3   500um      10 non-null     float64
 4   long       10 non-null     int64
 5   Evolution  10 non-null     int64
dtypes: float64(4), int64(2)
memory usage: 608.0 bytes
```

```
df
```

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

```
df.describe()
```

df.idescribe()

	150um	250um	350um	500um	long	Evolution
count	10.000000	10.000000	10.000000	10.000000	10.000000	10.000000
mean	0.506386	0.622530	0.555749	0.471654	20.000000	0.500000
std	0.276498	0.242824	0.346749	0.304313	10.540926	0.527046
min	0.083561	0.189939	0.028474	0.137869	10.000000	0.000000
25%	0.240975	0.523891	0.360391	0.243325	10.000000	0.000000
50%	0.600838	0.587108	0.546195	0.347882	20.000000	0.500000
75%	0.712484	0.819109	0.867113	0.750637	30.000000	1.000000
max	0.865100	0.994318	0.965483	0.922469	30.000000	1.000000

.astype()

- https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.astype.html
- Cast a pandas object to a specified dtype

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 6 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   150um      10 non-null     float64
 1   250um      10 non-null     float64
 2   350um      10 non-null     float64
 3   500um      10 non-null     float64
 4   long       10 non-null     int64
 5   Evolution  10 non-null     int64
dtypes: float64(4), int64(2)
memory usage: 608.0 bytes
```

```
df['long'].astype('float')

0     10.0
1     10.0
2     30.0
3     10.0
4     30.0
5     30.0
6     10.0
7     10.0
8     30.0
9     30.0
Name: long, dtype: float64
```

df

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
				224	0.952877	0.343632	30	1
				278	0.538161	0.922469	30	1

.columns and .index

df

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

```
df.columns
```

```
Index(['150um', '250um', '350um', '500um', 'long', 'Evolution'], dtype='object')
```

```
df.index
```

```
RangeIndex(start=0, stop=10, step=1)
```

.isnull()

```
df.isnull()
```

	150um	250um	350um	500um	long	Evolution
0	False	False	False	False	False	False
1	False	False	False	False	False	False
2	False	False	False	False	False	False
3	False	False	False	False	False	False
4	False	False	False	False	False	False
5	False	False	False	False	False	False
6	False	False	False	False	False	False
7	False	False	False	False	False	False
8	False	False	False	False	False	False
9	False	False	False	False	False	False

```
df
```

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

.unique() vs .nunique()

- In order to figure out the unique values:

```
df['Evolution'].unique()
```

```
array([0, 1], dtype=int64)
```

- We can find the number of unique parameters as well:

```
df['Evolution'].nunique()
```

```
2
```

df

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

value_counts()

- Alternatively, you may want to count the number of the unique values

```
df['Evolution'].value_counts()
```

```
0    5
1    5
Name: Evolution, dtype: int64
```

df

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

.sort_values(by= …)

```
df.sort_values(by='150um')
```

	150um	250um	350um	500um	long	Evolution
4	0.083561	0.603548	0.728993	0.276239	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
6	0.186967	0.994318	0.520665	0.578790	10	0
8	0.402998	0.357224	0.952877	0.343632	30	1
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
5	0.685306	0.517867	0.048485	0.137869	30	0
2	0.721544	0.189939	0.554228	0.352132	30	0
7	0.734819	0.541962	0.913154	0.807920	10	1
9	0.865100	0.830278	0.538161	0.922469	30	1

```
df
```

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

apply()

- In order to apply a function:

```
df['150um'].apply(round)
```

```
0    1
1    1
2    1
3    0
4    0
5    1
6    0
7    1
8    0
9    1
Name: 150um, dtype: int64
```

```
df['150um'].apply(lambda x: x*0.85)
```

```
0    0.438939
1    0.582485
2    0.613312
```

```
def correction(x):
    return x*0.85

df['150um'].apply(correction)
```

```
0    0.438939
1    0.582485
2    0.613312
3    0.154609
4    0.071027
5    0.582510
```

```
df
```

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

IS THERE A CORRELATION BETWEEN COLUMNS?

```
df[['150um','500um']].corr()
```

	150um	500um
150um	1.00000	0.48181
500um	0.48181	1.00000

df

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

Bonus: What does correlation mathematically mean?
What do +1, 0, and -1 mean?

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html

DataFrame.groupby()

df.groupby()

- Let's say you want to group your data by Galactic Longitude
  - You might be interested to figure out the mean of fluxes!

df

Observed fluxes

Galactic Longitude

Evolutionary Flag

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

df.groupby()

```
grouped = df.groupby('long')
```

```
grouped.mean()
```

long	150um	250um	350um	500um	Evolution
10	0.461071	0.745289	0.546949	0.536840	0.6
30	0.551702	0.499771	0.564549	0.406468	0.4

```
df.groupby('long')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy
```

```
df.groupby('long').mean()
```

long	150um	250um	350um	500um	Evolution
10	0.461071	0.745289	0.546949	0.536840	0.6
30	0.551702	0.499771	0.564549	0.406468	0.4

```
df.groupby('long').sum()
```

```
df.groupby('long').std()
```

```
df.groupby('long').count()
```

long	150um	250um	350um	500um	Evolution
10	5	5	5	5	5
30	5	5	5	5	5

```
df
```

	150um	250um	350um	500um	long	Evolution
0	0.516399	0.570668	0.028474	0.171522	10	0
1	0.685277	0.833897	0.306966	0.893613	10	1
2	0.721544	0.189939	0.554228	0.352132	30	0
3	0.181892	0.785602	0.965483	0.232354	10	1
4	0.083561	0.603548	0.728993	0.276239	30	0
5	0.685306	0.517867	0.048485	0.137869	30	0
6	0.186967	0.994318	0.520665	0.578790	10	0
7	0.734819	0.541962	0.913154	0.807920	10	1
8	0.402998	0.357224	0.952877	0.343632	30	1
9	0.865100	0.830278	0.538161	0.922469	30	1

df.groupby()

```
df.groupby('long').describe()
```

	150um								250um		...	500um		Evolution				
	count	mean	std	min	25%	50%	75%	max	count	mean	...	75%	max	count	mean	std	min	25%
long																		
10	5.0	0.316285	0.179862	0.122073	0.184867	0.278681	0.438612	0.55719	5.0	0.364438	...	0.76252	0.829918	5.0	0.6	0.547723	0.0	0.
30	5.0	0.432625	0.193985	0.248595	0.256753	0.417359	0.533277	0.70714	5.0	0.705619	...	0.33838	0.430418	5.0	0.4	0.547723	0.0	0.

2 rows × 40 columns

```
df.groupby('long').describe().T
```

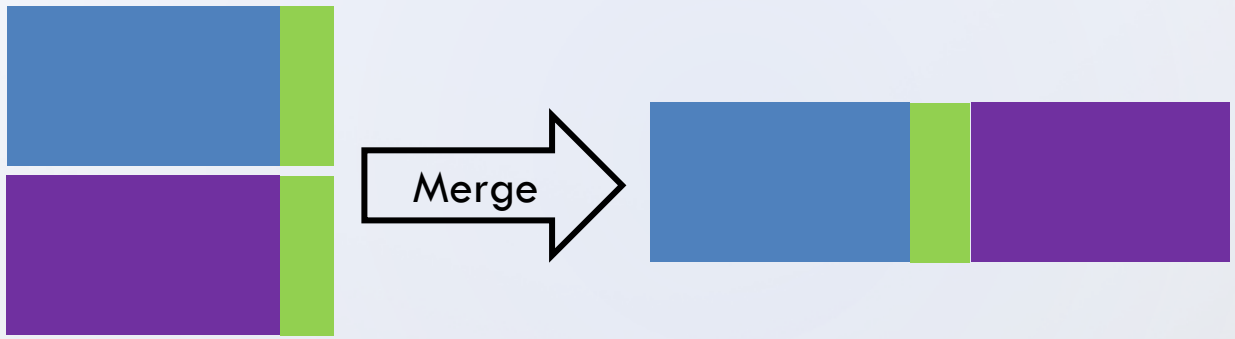
Transpose Matrix

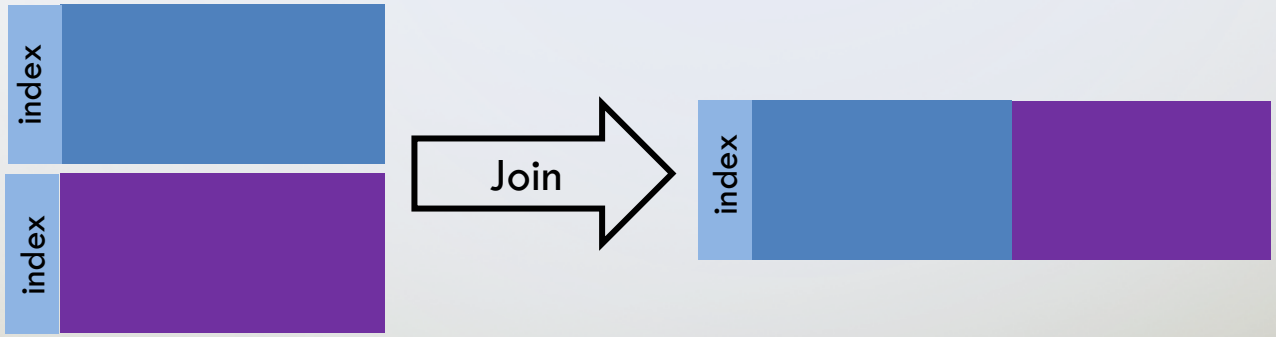
	long	10	30
150um	count	5.000000	5.000000
	mean	0.316285	0.432625
	std	0.179862	0.193985
	min	0.122073	0.248595
	25%	0.184867	0.256753
	50%	0.278681	0.417359
	75%	0.438612	0.533277
	max	0.557190	0.707140
250um	count	5.000000	5.000000
	mean	0.364438	0.705619
	std	0.297123	0.222368
	min	0.133277	0.352804
	25%	0.154798	0.677290
	50%	0.197953	0.723229
	75%	0.520390	0.832064
	max	0.815775	0.942709
350um	count	5.000000	5.000000
	mean	0.631148	0.353121
	std	0.261274	0.303377
	min	0.287423	0.036434
	25%	0.422001	0.132985
	50%	0.750619	0.285546
	75%	0.809279	0.528465
	max	0.886419	0.782176
500um	count	5.000000	5.000000
	mean	0.485464	0.293920
	std	0.287392	0.135417
	min	0.230020	0.066668
	25%	0.265592	0.313339
	50%	0.339272	0.320793
	75%	0.762520	0.338380
	max	0.829918	0.430418
Evolution	count	5.000000	5.000000
	mean	0.600000	0.400000
	std	0.547723	0.547723
	min	0.000000	0.000000
	25%	0.000000	0.000000
	50%	1.000000	0.000000
	75%	1.000000	1.000000
	max	1.000000	1.000000

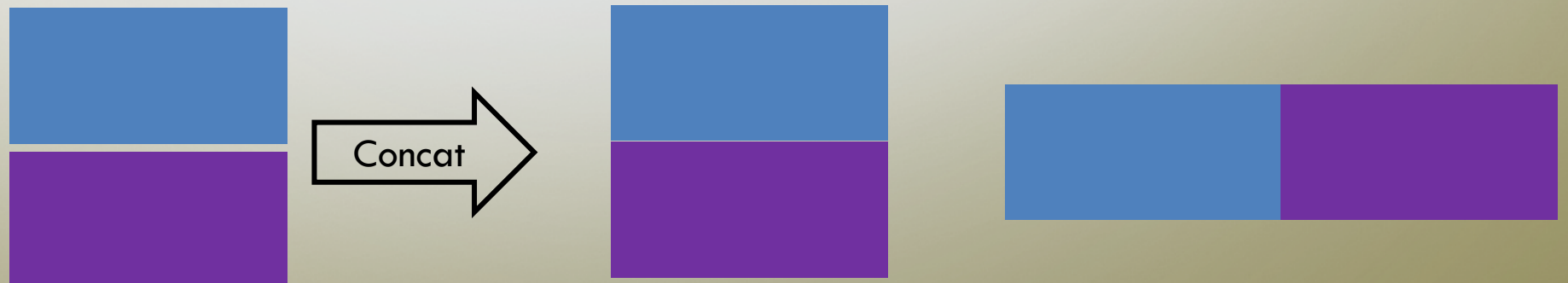
EXERCISE

- Import rand
- Create an array of 40 random values and reshape it to 10 rows and 4 columns
- Create a DataFrame of this array and call the columns ['150um', '250um','350um','500um']
- Add the two following columns to the DataFrame
  - df['long'] = [10,10,30,10,30,30,10,10,30,30]
  - df['Evolution'] = [0,1,0,1,0,0,0,1,1,1]
- Now, groupby 'long' and:
  - Explore mean, sum, std, describe, etc …

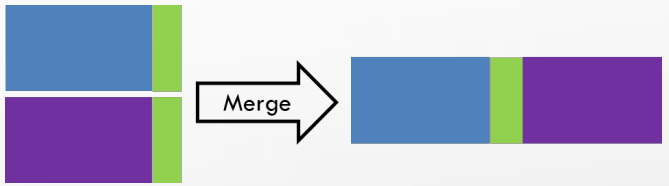
MERGE VS JOIN VS CONCAT







MERGING DATAFRAMES



df1

	150um	250um	ID
0	0.830265	0.856099	111
1	0.815043	0.417518	112
2	0.606254	0.745214	113
3	0.928163	0.437889	114
4	0.796525	0.192480	115
5	0.392574	0.199985	116
6	0.488727	0.790390	117
7	0.301641	0.517221	118
8	0.329071	0.184599	119
9	0.433268	0.371879	120

df2

	350um	500um	ID
0	0.108690	0.437375	120
1	0.922930	0.771665	119
2	0.177019	0.574087	118
3	0.166418	0.613235	117
4	0.208856	0.278053	116
5	0.772448	0.636974	115
6	0.609622	0.455972	114
7	0.831996	0.180423	113
8	0.799834	0.780027	112
9	0.328818	0.751890	111

```
pd.merge(df1,df2,on='ID')
```

	150um	250um	ID	350um	500um
0	0.830265	0.856099	111	0.328818	0.751890
1	0.815043	0.417518	112	0.799834	0.780027
2	0.606254	0.745214	113	0.831996	0.180423
3	0.928163	0.437889	114	0.609622	0.455972
4	0.796525	0.192480	115	0.772448	0.636974
5	0.392574	0.199985	116	0.208856	0.278053
6	0.488727	0.790390	117	0.166418	0.613235
7	0.301641	0.517221	118	0.177019	0.574087
8	0.329071	0.184599	119	0.922930	0.771665
9	0.433268	0.371879	120	0.108690	0.437375

MERGING DATAFRAMES

df1

	150um	250um	ID
0	0.830265	0.856099	111
1	0.815043	0.417518	112
2	0.606254	0.745214	113
3	0.928163	0.437889	114
4	0.796525	0.192480	115
5	0.392574	0.199985	116
6	0.488727	0.790390	117
7	0.301641	0.517221	118
8	0.329071	0.184599	119
9	0.433268	0.371879	120

df2

	350um	500um	ID
0	0.108690	0.437375	120
1	0.922930	0.771665	119
2	0.177019	0.574087	118
3	0.166418	0.613235	117
4	0.208856	0.278053	116
5	0.772448	0.636974	115
6	0.609622	0.455972	114
7	0.831996	0.180423	113
8	0.799834	0.780027	112
9	0.328818	0.751890	111
10	0.283425	0.898644	110

Default

```
pd.merge(df1,df2, how='inner' ,on='ID')
```

	150um	250um	ID	350um	500um
0	0.830265	0.856099	111	0.328818	0.751890
1	0.815043	0.417518	112	0.799834	0.780027
2	0.606254	0.745214	113	0.831996	0.180423
3	0.928163	0.437889	114	0.609622	0.455972
4	0.796525	0.192480	115	0.772448	0.636974
5	0.392574	0.199985	116	0.208856	0.278053
6	0.488727	0.790390	117	0.166418	0.613235
7	0.301641	0.517221	118	0.177019	0.574087
8	0.329071	0.184599	119	0.922930	0.771665
9	0.433268	0.371879	120	0.108690	0.437375

MERGING DATAFRAMES

Changed to outer!

df1

	150um	250um	ID
0	0.830265	0.856099	111
1	0.815043	0.417518	112
2	0.606254	0.745214	113
3	0.928163	0.437889	114
4	0.796525	0.192480	115
5	0.392574	0.199985	116
6	0.488727	0.790390	117
7	0.301641	0.517221	118
8	0.329071	0.184599	119
9	0.433268	0.371879	120

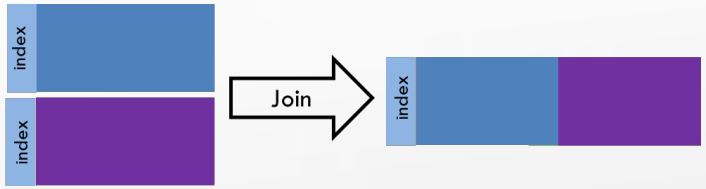
df2

	350um	500um	ID
0	0.108690	0.437375	120
1	0.922930	0.771665	119
2	0.177019	0.574087	118
3	0.166418	0.613235	117
4	0.208856	0.278053	116
5	0.772448	0.636974	115
6	0.609622	0.455972	114
7	0.831996	0.180423	113
8	0.799834	0.780027	112
9	0.328818	0.751890	111
10	0.283425	0.898644	110

```
pd.merge(df1,df2,how='outer',on='ID')
```

	150um	250um	ID	350um	500um
0	0.830265	0.856099	111	0.328818	0.751890
1	0.815043	0.417518	112	0.799834	0.780027
2	0.606254	0.745214	113	0.831996	0.180423
3	0.928163	0.437889	114	0.609622	0.455972
4	0.796525	0.192480	115	0.772448	0.636974
5	0.392574	0.199985	116	0.208856	0.278053
6	0.488727	0.790390	117	0.166418	0.613235
7	0.301641	0.517221	118	0.177019	0.574087
8	0.329071	0.184599	119	0.922930	0.771665
9	0.433268	0.371879	120	0.108690	0.437375
10	NaN	NaN	110	0.283425	0.898644

JOIN DATAFRAMES



df3

	350um	500um
a	0.689986	0.840446
b	0.510564	0.332990
c	0.002479	0.991588
d	0.025441	0.226073
e	0.818724	0.201089
f	0.870840	0.156989
g	0.488987	0.576040
h	0.622634	0.863371
i	0.833045	0.267212
j	0.727543	0.848315
k	0.418458	0.113081

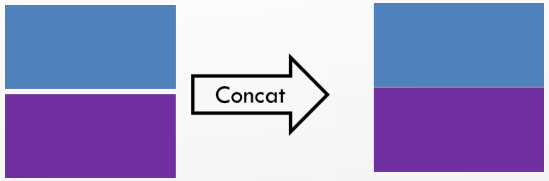
df4

	150um	250um
a	0.838322	0.529164
b	0.498260	0.692531
c	0.614688	0.865564
d	0.267630	0.199455
e	0.962236	0.342652
f	0.004878	0.121251
g	0.278178	0.809031
h	0.962500	0.681270
i	0.416705	0.994289
j	0.881555	0.912759

```
df4.join(df3)
```

	150um	250um	350um	500um
a	0.838322	0.529164	0.689986	0.840446
b	0.498260	0.692531	0.510564	0.332990
c	0.614688	0.865564	0.002479	0.991588
d	0.267630	0.199455	0.025441	0.226073
e	0.962236	0.342652	0.818724	0.201089
f	0.004878	0.121251	0.870840	0.156989
g	0.278178	0.809031	0.488987	0.576040
h	0.962500	0.681270	0.622634	0.863371
i	0.416705	0.994289	0.833045	0.267212
j	0.881555	0.912759	0.727543	0.848315

CONCAT DATAFRAMES



df4

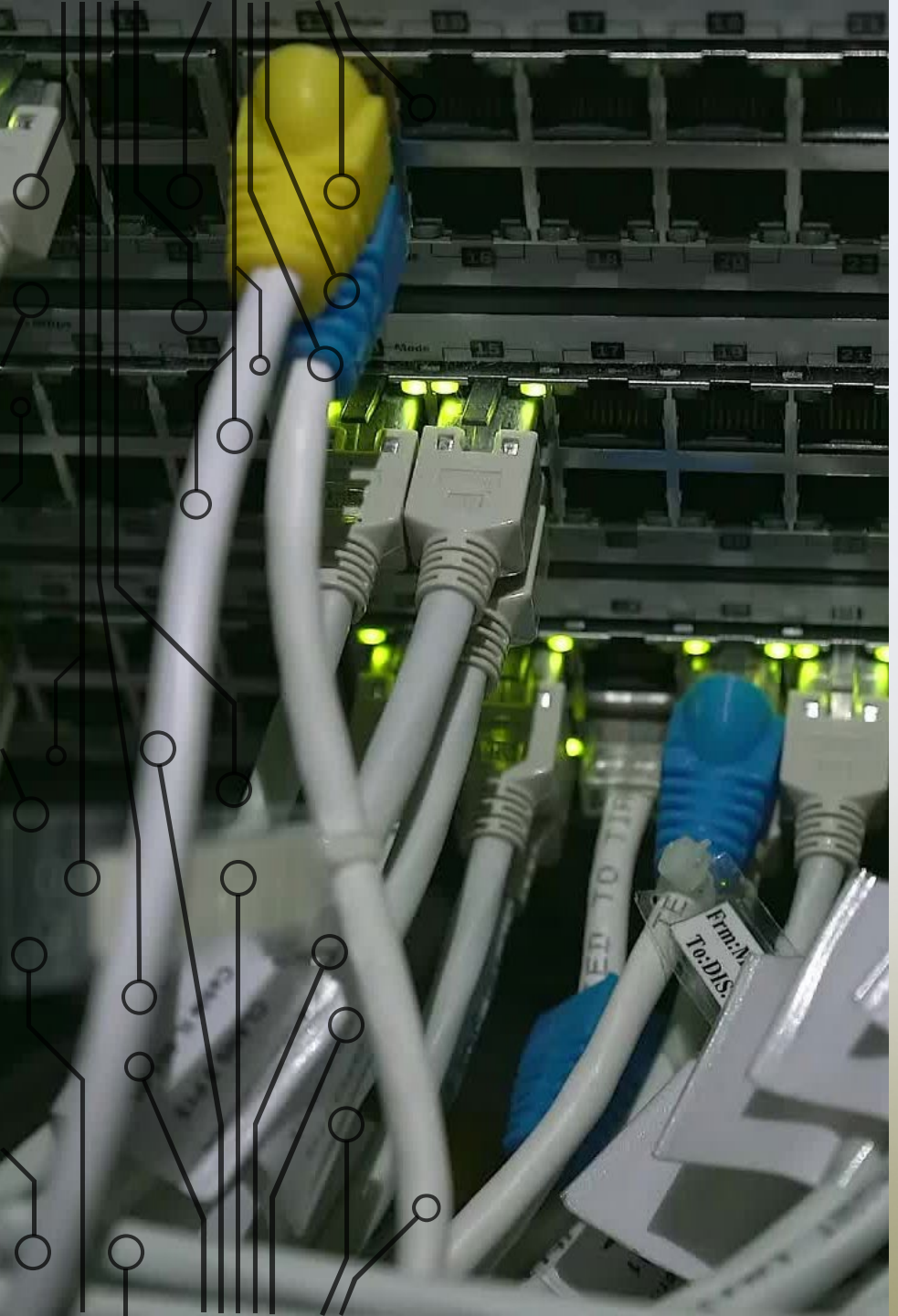
	150um	250um
a	0.838322	0.529164
b	0.498260	0.692531
c	0.614688	0.865564
d	0.267630	0.199455
e	0.962236	0.342652
f	0.004878	0.121251
g	0.278178	0.809031
h	0.962500	0.681270
i	0.416705	0.994289
j	0.881555	0.912759

df5

	150um	250um
k	0.627725	0.746178
l	0.968120	0.636644
m	0.767799	0.602385
n	0.729234	0.570438
o	0.059060	0.892397
p	0.615531	0.602143
q	0.259967	0.146037
r	0.552684	0.259272
s	0.452009	0.076105
t	0.849954	0.031987

```
pd.concat([df4,df5], axis=0)
```

	150um	250um
a	0.838322	0.529164
b	0.498260	0.692531
c	0.614688	0.865564
d	0.267630	0.199455
e	0.962236	0.342652
f	0.004878	0.121251
g	0.278178	0.809031
h	0.962500	0.681270
i	0.416705	0.994289
j	0.881555	0.912759
k	0.627725	0.746178
l	0.968120	0.636644
m	0.767799	0.602385
n	0.729234	0.570438
o	0.059060	0.892397
p	0.615531	0.602143
q	0.259967	0.146037
r	0.552684	0.259272
s	0.452009	0.076105
t	0.849954	0.031987



DATA INPUT & OUTPUT

- pd.read_csv(‘input’)
  - df.to_csv(‘output’, index =False)
- pd.read_excel(‘input',sheetname='Sheet1')
  - df.to_excel(‘output',sheet_name='Sheet1’)
- pd.read_table(‘input’)
- data = pd.read_html(‘url’)
  - data[0]

PD.READ_CSV

```
cat = pd.read_csv('Data/table_of_data.tab',
                  delimiter= ':',
                  delim_whitespace=True,
                  nrows = 3,
                  skiprows=[0,1,2,3],
                  names=['ID','DESIGNATION','GLAT','RA','DEC']
                 )
```

Location of your data!

delimiter or sep: How to separate columns. Note: you can use regex, e.g. \t , \s+

Use it, if columns are separated with space. Note: Not to be used with delimiter

Number of rows to read from your data

Skip these rows

Rename the header to this

	ID	DESIGNATION	GLON	GLAT	RA	DEC
0	1	HIGALBM0.0002+0.2799	0.000153	0.279931	266.13256	-28.789917
1	2	HIGALBM0.0035-0.0235	0.003490	-0.023500	266.43010	-28.945437
2	3	HIGALBM0.0043+0.1541	0.004339	0.154110	266.25750	-28.852094

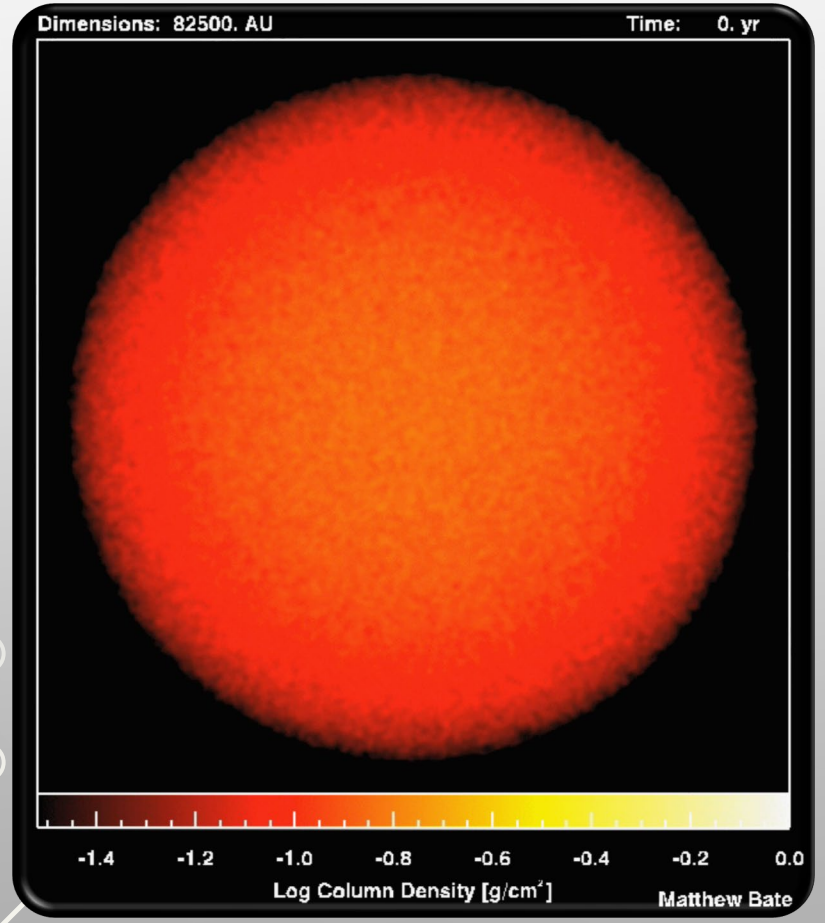
STRETCH YOUR CODING MUSCLES!

I need to introduce your data first!

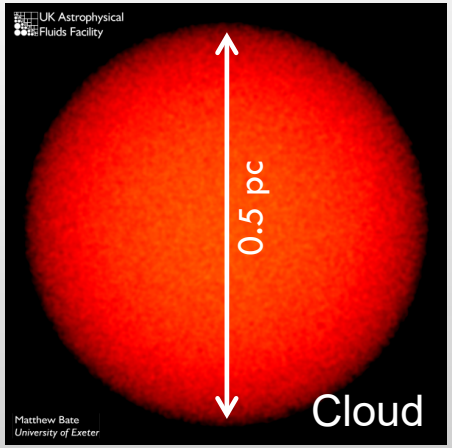
Part_III_Pandas_a

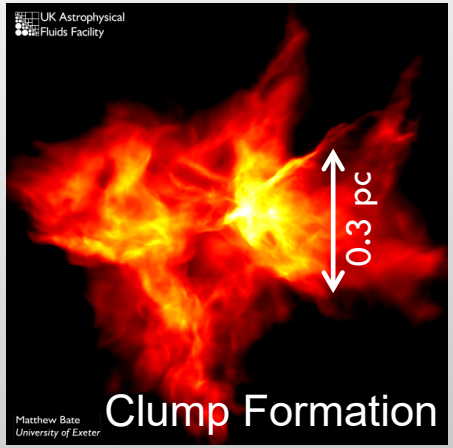
Part_III_Pandas_b

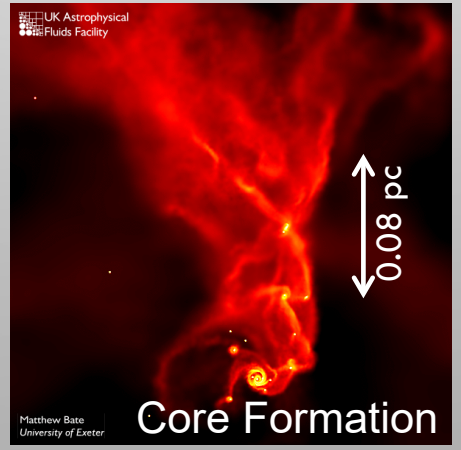
BASICS OF STAR FORMATION



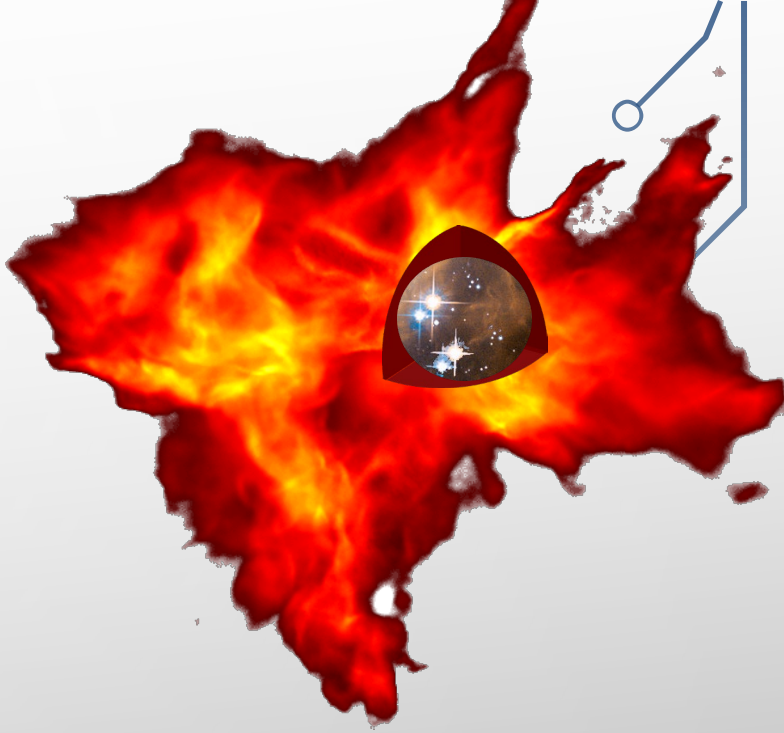
Bate et al. 2002



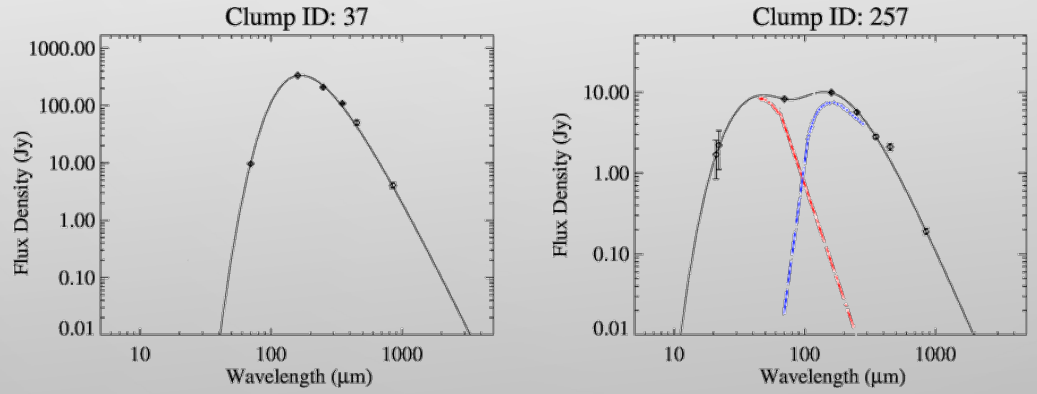




BASICS OF STAR FORMATION



$$F_\nu = [(1 - e^{-\tau_c})B_\nu(T_c)\Omega_c]_c + [(1 - e^{-\tau_w})B_\nu(T_w)\Omega_w]_w$$




```
cat.head()
```

GLON	GLAT	ra	dec	DESIGNATION_70	F70	DF70	F70_TOT	...	EVOL_FLAG	MASS	DMASS	TEMP	DTEMP	LAM_0_TK	L_BOL	LF
9.254488	1.084173	270.54295	-20.408795	-	0.0	0.0	0.0	...	1.0	-7.63	-1.86	10.84	0.48	0.0	-1.19	
9.256889	-0.282502	271.81796	-21.076705	-	0.0	0.0	0.0	...	1.0	-30.79	-6.40	10.31	0.42	70.3	-3.38	
9.258172	-0.472514	271.99662	-21.167985	-	0.0	0.0	0.0	...	1.0	-10.36	-2.69	12.32	0.66	0.0	-3.49	

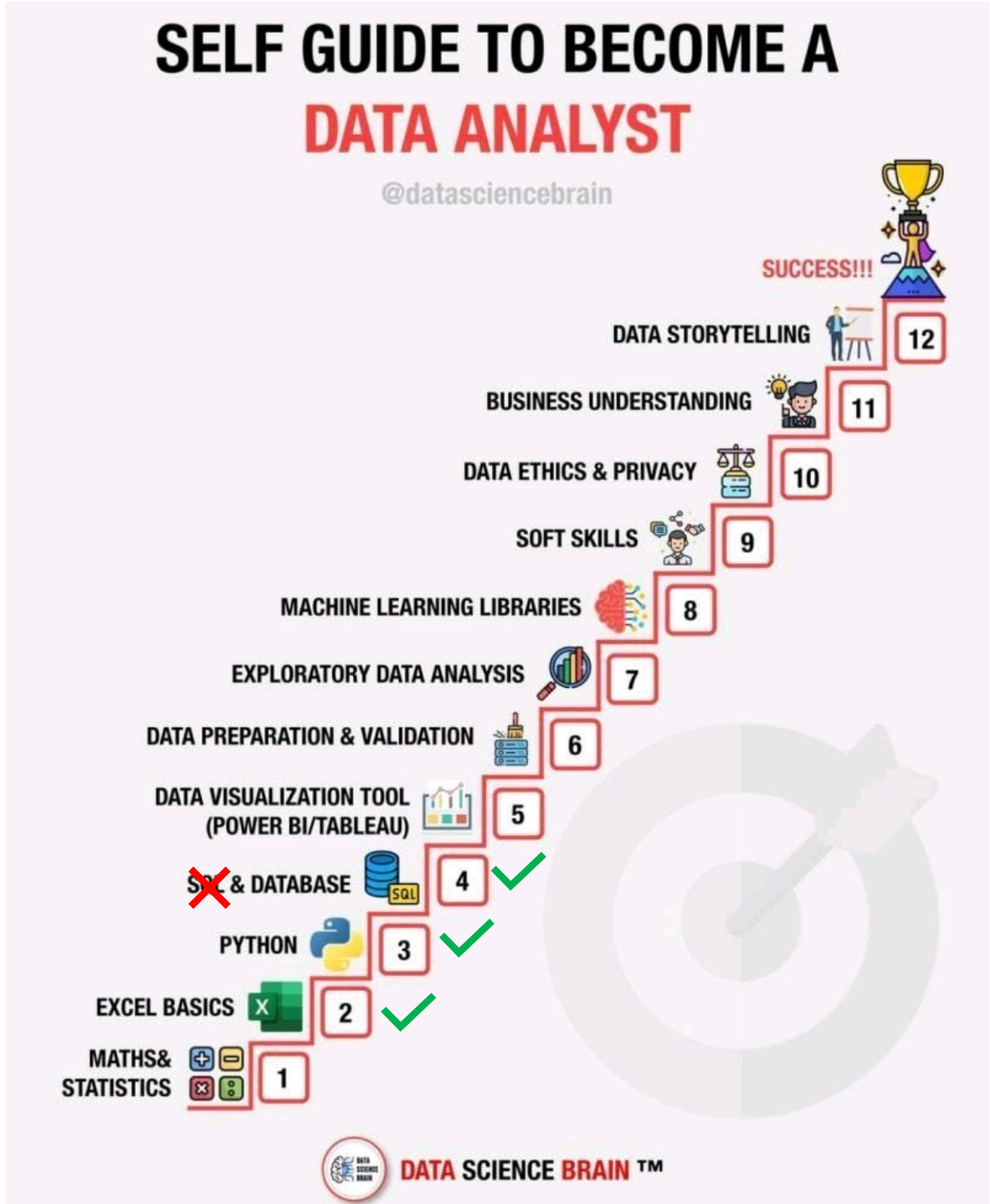
STRETCH YOUR CODING MUSCLES!

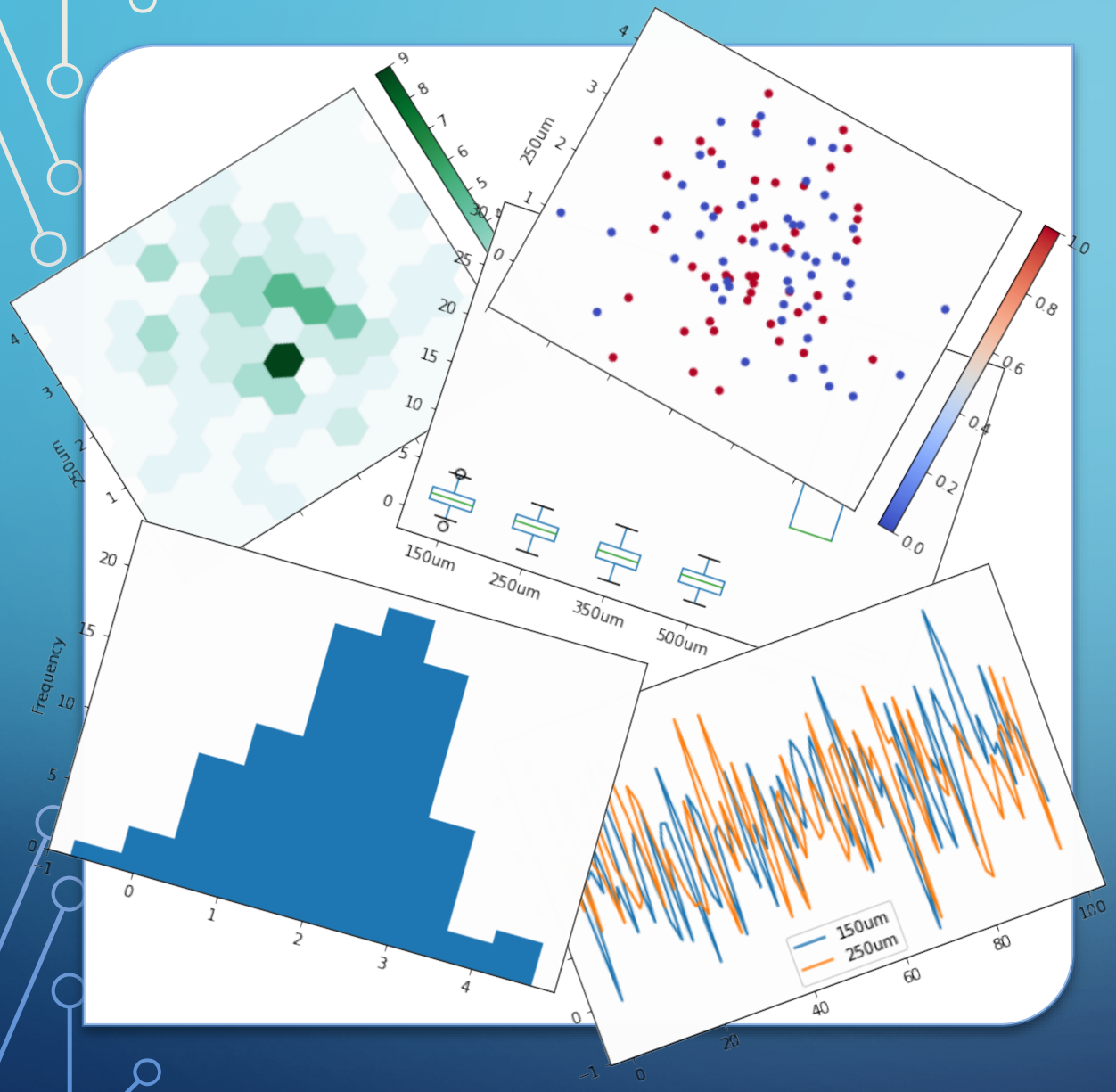
Now let's stretch!

Part_III_Pandas_a

Part_III_Pandas_b

OUR SKILLSETS SO FAR!





DATA VISUALIZATION

- First, we learn how python can be used for some quick data visualization
- However, for more appropriate data visualization we should use:
  - Matplot.pyplot
  - Seaborn
  - SciPy

CODE WITH ME! CREATE YOUR DATA

- Import pandas, numpy,
- Import rand, randn
- Use random seed 101
- Create an array of 400 randn numbers and add 2 to their values
- Reshape them to (100, 4)
- Use pd.dataframe to create a df with the columns=['150um', '250um','350um','500um']
- Randomly add 'long' and 'Evolution'
  - df['long'] = it should be 10 if round of random number is 0 and should be 30 if round is 1
  - df['Evolution'] = 0 if round of random number is 0, else it should be 1

```
import numpy as np
import pandas as pd
from numpy.random import rand, randn
```

```
np.random.seed(101)
df=pd.DataFrame(np.array(randn(400)+2).reshape(100,4), columns=['150um', '250um','350um','500um'])
df['long'] = [10 if x==0 else 30 for x in rand(100).round()]
df['Evolution'] = rand(100).round()
df
```

	150um	250um	350um	500um	long	Evolution
0	4.706850	2.628133	2.907969	2.503826	10	0.0
1	2.651118	1.680682	1.151923	2.605965	30	1.0
2	-0.018168	2.740122	2.528813	1.410999	30	1.0
3	2.188695	1.241128	1.066763	2.955057	30	1.0
4	2.190794	3.978757	4.605967	2.683509	10	0.0
...	...	...	...	...	...	...
95	3.351807	2.257183	2.126086	2.466044	10	0.0
96	1.864187	1.289333	2.863646	1.273259	30	0.0
97	2.417412	3.273269	0.805301	1.854769	10	0.0
98	2.013519	-0.156488	1.505083	2.071770	30	1.0
99	0.708441	3.001002	3.669011	3.480148	10	1.0

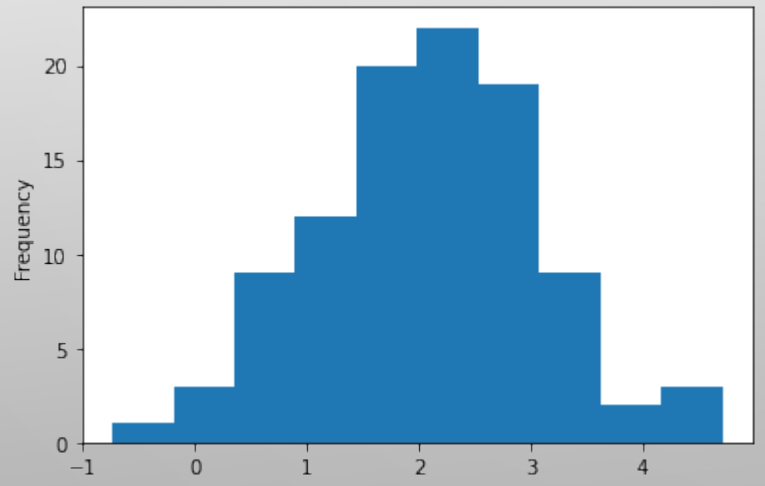
100 rows × 6 columns

df.plot.hist(bins=)

Histogram

```
df['150um'].plot.hist(bins=10)
```

There are different alternatives. However, to keep it consistent I recommend this formatting.



	150um	250um	350um	500um	long	Evolution
0	4.706850	2.628133	2.907969	2.503826	10	0.0
1	2.651118	1.680682	1.151923	2.605965	30	1.0
2	-0.018168	2.740122	2.528813	1.410999	30	1.0
3	2.188695	1.241128	1.066763	2.955057	30	1.0
4	2.190794	3.978757	4.605967	2.683509	10	0.0
...	...	...	...	...	...	...
95	3.351807	2.257183	2.126086	2.466044	10	0.0
96	1.864187	1.289333	2.863646	1.273259	30	0.0
97	2.417412	3.273269	0.805301	1.854769	10	0.0
98	2.013519	-0.156488	1.505083	2.071770	30	1.0
99	0.708441	3.001002	3.669011	3.480148	10	1.0

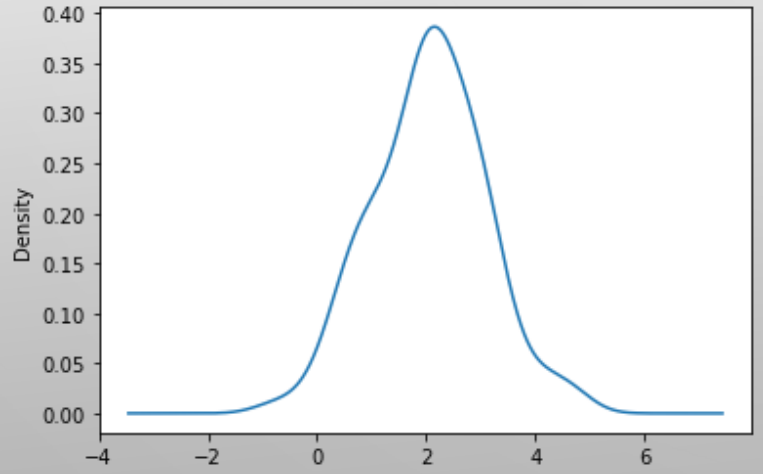
100 rows × 6 columns

df.plot.kde()

```
df['150um'].plot.kde()
```

Kernel Density

- https://en.wikipedia.org/wiki/Kernel_density_estimation



	150um	250um	350um	500um	long	Evolution
0	4.706850	2.628133	2.907969	2.503826	10	0.0
1	2.651118	1.680682	1.151923	2.605965	30	1.0
2	-0.018168	2.740122	2.528813	1.410999	30	1.0
3	2.188695	1.241128	1.066763	2.955057	30	1.0
4	2.190794	3.978757	4.605967	2.683509	10	0.0
...	...	...	...	...	...	...
95	3.351807	2.257183	2.126086	2.466044	10	0.0
96	1.864187	1.289333	2.863646	1.273259	30	0.0
97	2.417412	3.273269	0.805301	1.854769	10	0.0
98	2.013519	-0.156488	1.505083	2.071770	30	1.0
99	0.708441	3.001002	3.669011	3.480148	10	1.0

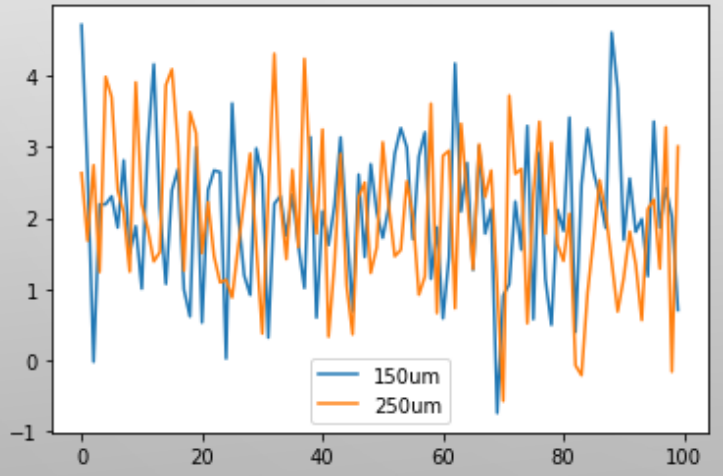
100 rows × 6 columns

df.plot.line()

Line Plot

```
df[['150um','250um']].plot.line()
```

- Use shift+tab for more options
- Note you can define x=? & y=?
  - If x is not defined index will be used!

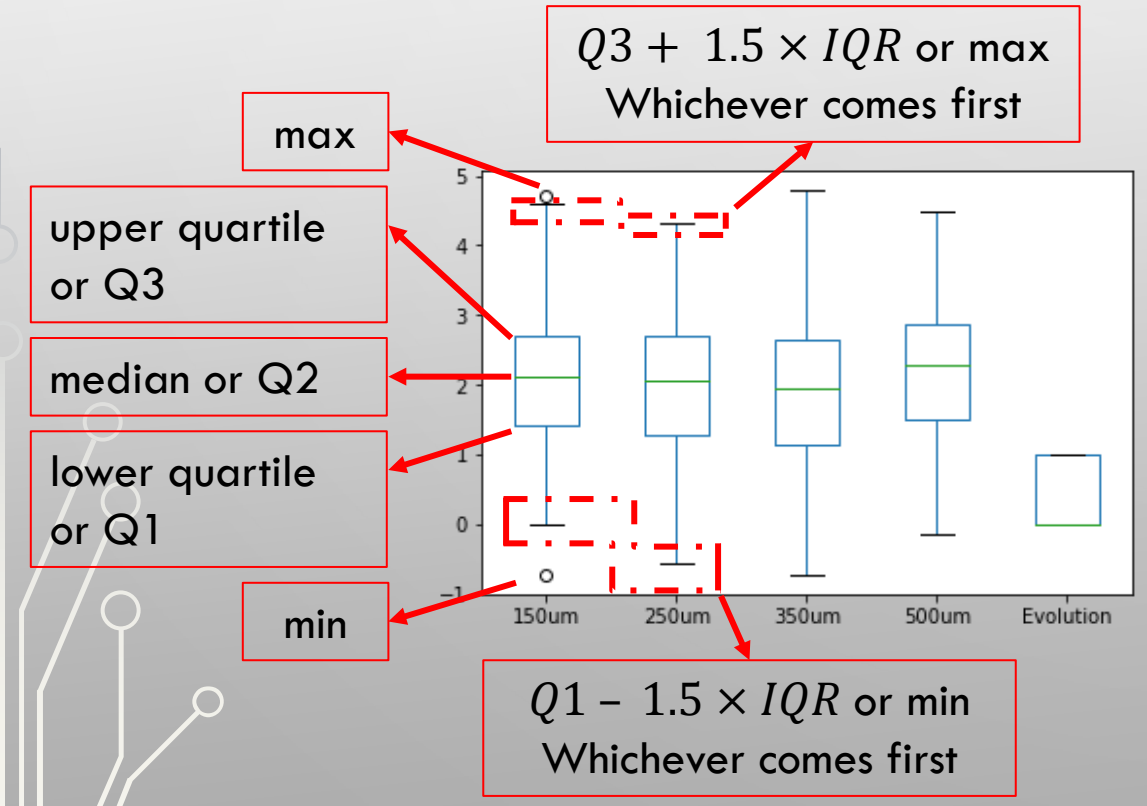


	150um	250um	350um	500um	long	Evolution
0	4.706850	2.628133	2.907969	2.503826	10	0.0
1	2.651118	1.680682	1.151923	2.605965	30	1.0
2	-0.018168	2.740122	2.528813	1.410999	30	1.0
3	2.188695	1.241128	1.066763	2.955057	30	1.0
4	2.190794	3.978757	4.605967	2.683509	10	0.0
...	...	...	...	...	...	...
95	3.351807	2.257183	2.126086	2.466044	10	0.0
96	1.864187	1.289333	2.863646	1.273259	30	0.0
97	2.417412	3.273269	0.805301	1.854769	10	0.0
98	2.013519	-0.156488	1.505083	2.071770	30	1.0
99	0.708441	3.001002	3.669011	3.480148	10	1.0

100 rows × 6 columns

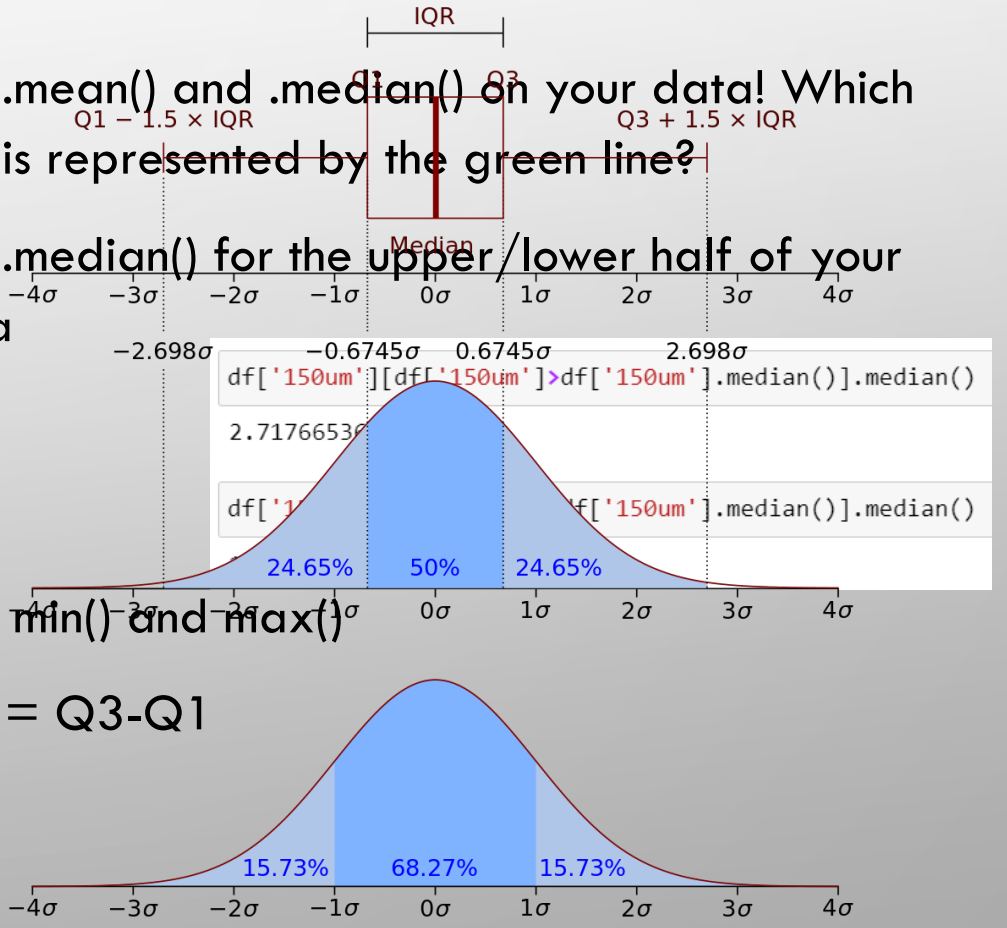
df.plot.box()

```
df.drop('long', axis=1).plot.box()
```



Box Plot

- Use .mean() and .median() on your data! Which one is represented by the green line?
- Use .median() for the upper/lower half of your data
- Find min() and max()
- IQR = Q3-Q1



More info https://en.wikipedia.org/wiki/Interquartile_range

.plot.scatter()

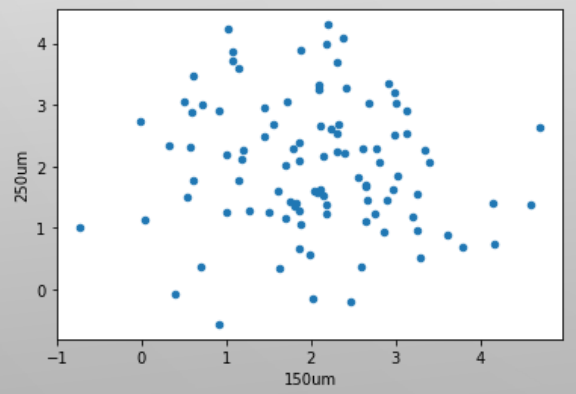
Scatter Plot

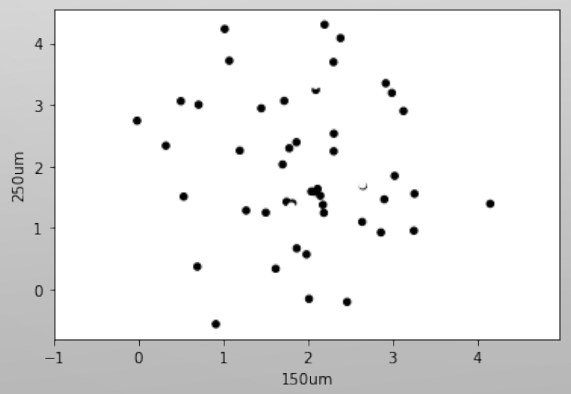
```
df.plot.scatter(x='150um', y='250um' )
                #, c=df['Evolution']
                , s = df['Evolution']*20+10
                , cmap='coolwarm'
               )
```

c is the color. However, I am setting the color to one of the columns

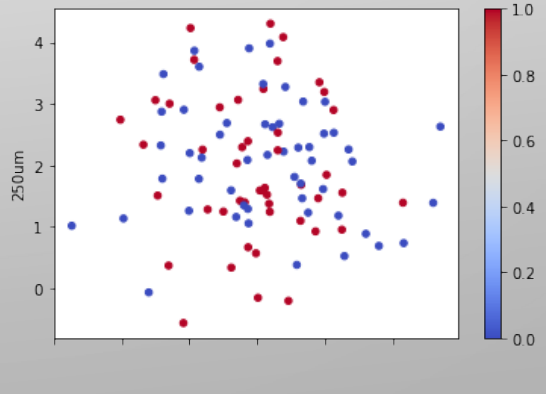
s is the size. However, I am setting the size to one of the columns

cmap → color map → explore here: https://matplotlib.org/stable/tutorials/colors/colormaps.html

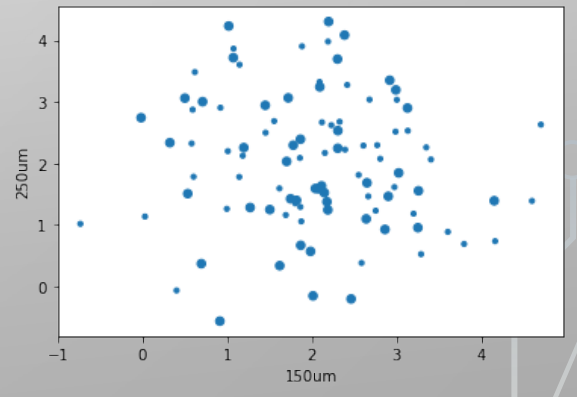




c=df['Evolution']



cmap ='coolwarm'



s =df['Evolution']*20+10

STRETCH YOUR CODING MUSCLES!

Part_IV_Data_Visualization_a_Pandas

INSTALL MATPLOTLIB

```
pip install matplotlib
```

Or:

```
conda install matplotlib
```

matplotlib

Plot types Examples Tutorials Reference User guide Develop Release notes stable

Section Navigation

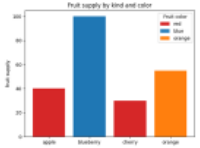
- Lines, bars and markers
- Images, contours and fields
- Subplots, axes and figures
- Statistics
- Pie and polar charts
- Text, labels and annotations
- pyplot
- Color
- Shapes and collections
- Style sheets
- axes_grid1
- axisartist
- Showcase
- Animation
- Event handling
- Miscellaneous
- 3D plotting
- Scales
- Specialty Plots
- Spines
- Ticks
- Units
- Embedding Matplotlib in graphical user interfaces
- Userdemo

Examples

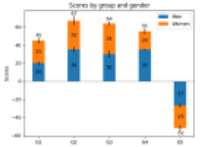
This page contains example plots. Click on any image to see the full image and source code.

For longer tutorials, see our tutorials page. You can also find external resources and a FAQ in our user guide.

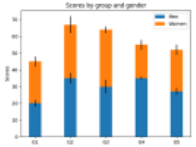
Lines, bars and markers



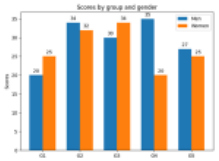
Bar color demo



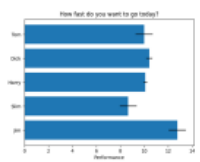
Bar Label Demo



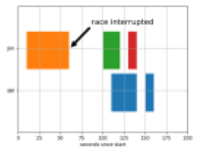
Stacked bar chart



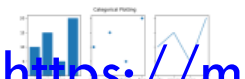
Grouped bar chart with labels

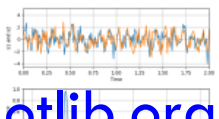


Horizontal bar chart



Broken Barh





On this page

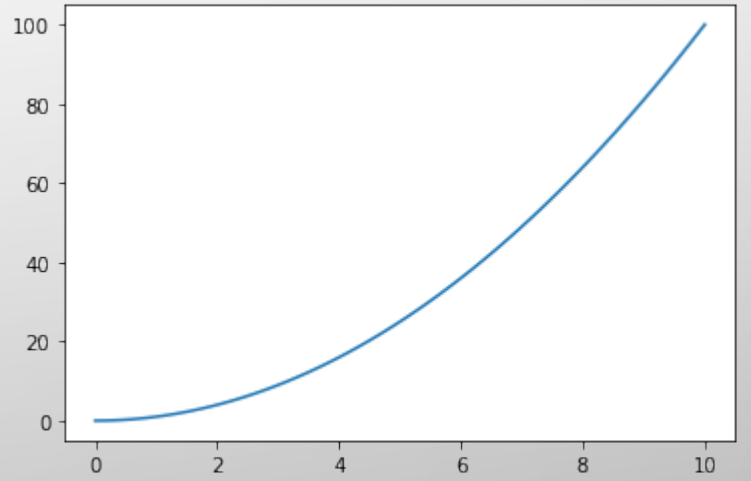
- https://matplotlib.org/stable/gallery/index.html

import matplotlib.pyplot as plt

```
import matplotlib.pyplot as plt
```

- Two methods:
  1. Functional → plt.plot(x,y)

```
x= np.linspace(0,10,100)
y= x**2

#Functional
plt.plot(x,y)
```



- Note: that you may need plt.show() if you are not using Jupyter
- SORRY! I KNOW IT LOOKS SIMPLER, BUT I HIGHLY SUGGEST NOT TO USE THIS ONE FOR NOW!

  2. Object Oriented
- ALWAYS USE THIS METHOD THROUGHOUT THIS COURSE! PLEASE DON'T HATE ME, YET!

```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

ax.plot(x,y)
```

import matplotlib.pyplot as plt, Object Oriented

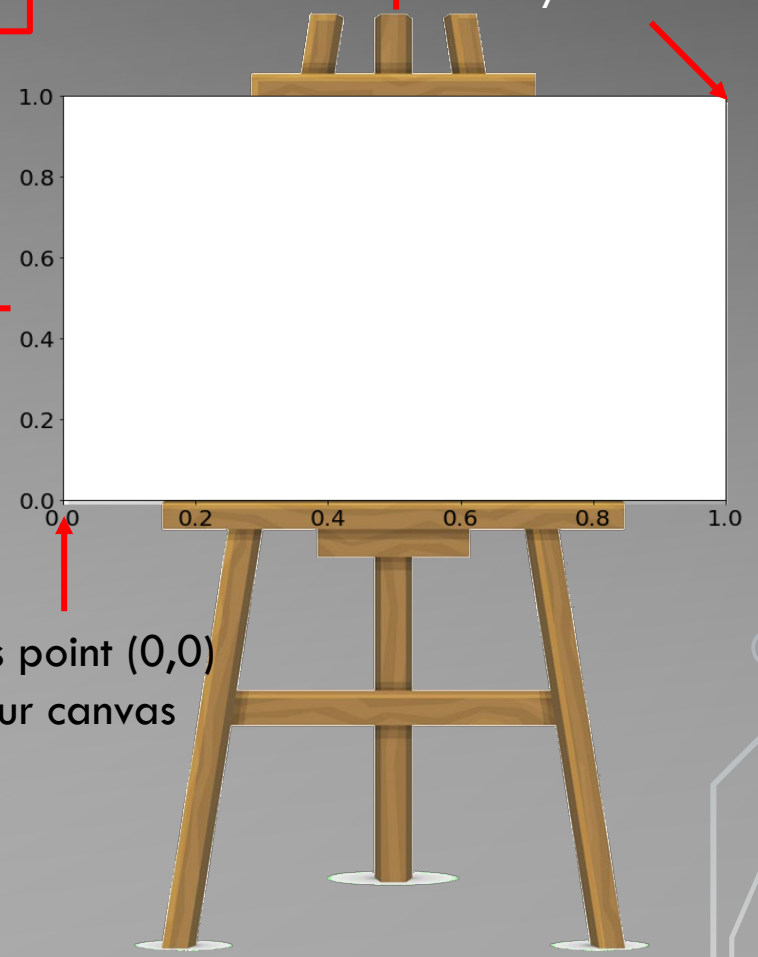
```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

ax.plot(x,y)
```

This line creates your canvas (figure)

Adds axes at [0,0] & stretch it for [1,1] from this point

This is point (1,1) of your canvas

This is point (0,0) of your canvas



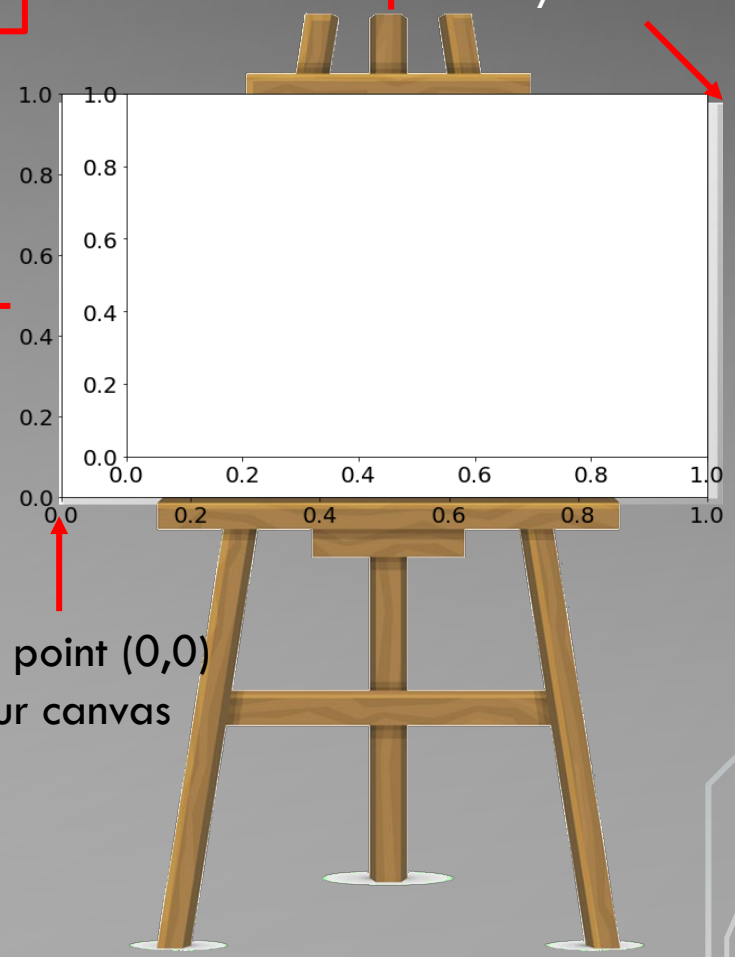
import matplotlib.pyplot as plt, Object Oriented

```
fig = plt.figure()
ax = fig.add_axes([0.1,0.1,0.9,0.9])

ax.plot(x,y)
```

This line creates your canvas (figure)

Adds axes at [0.1,0.1] & stretch it for [0.9,0.9] from [0.1,0.1]

This is point (1,1) of your canvas



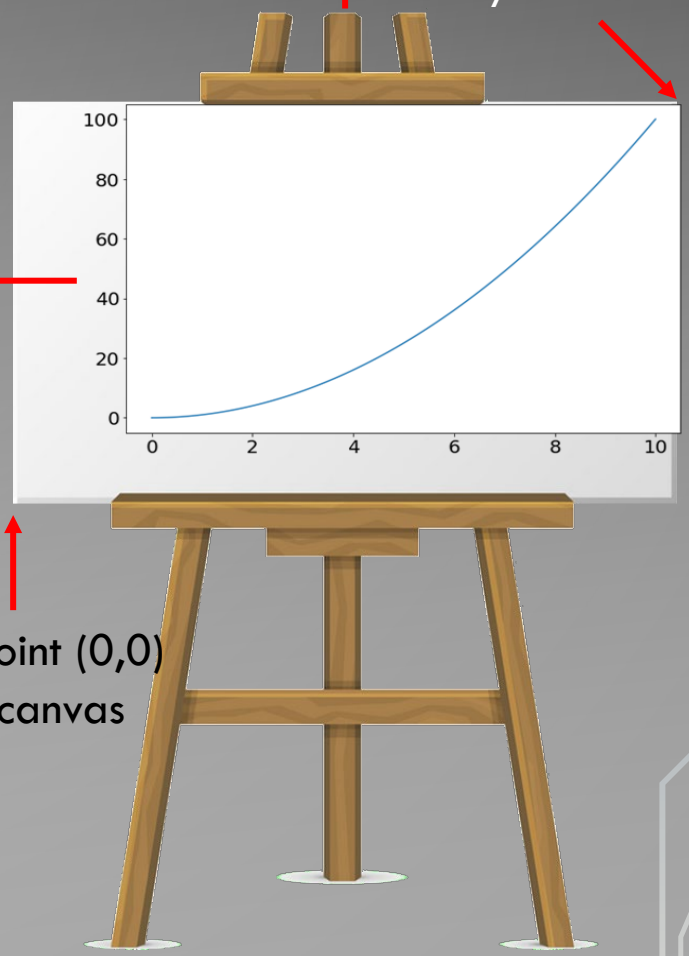
This is point (0,0) of your canvas

import matplotlib.pyplot as plt, Object Oriented

```
fig = plt.figure()
ax = fig.add_axes([0.1,0.1,0.9,0.9])

ax.plot(x,y)
```

This line creates your canvas (figure)

Note that the plot may have different range but still [0,0] & [1,1] are referring to 0% of your canvas and 100% stretch



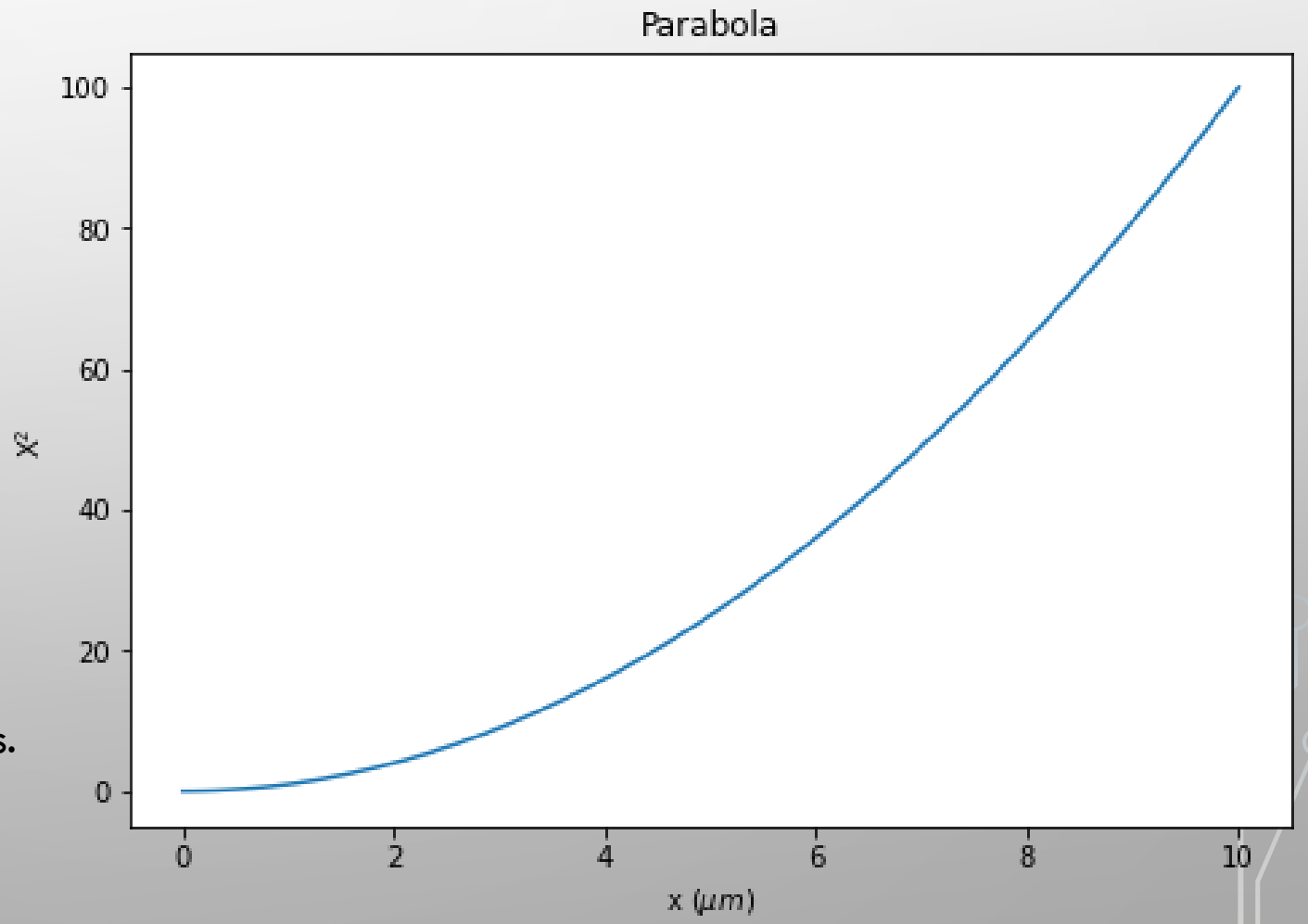
EXERCISE

- Create a figure with two axes:
  1. ax1 = Stretches 100% of your figure
  2. ax2 = Starts from point [0.1, 0.3] and stretches to [0.3,0.8]
  - Plot x,x**2 on ax1 and x,x**0.5 on ax2, where x in (0,10)

SET TITLES AND XY-LABELS

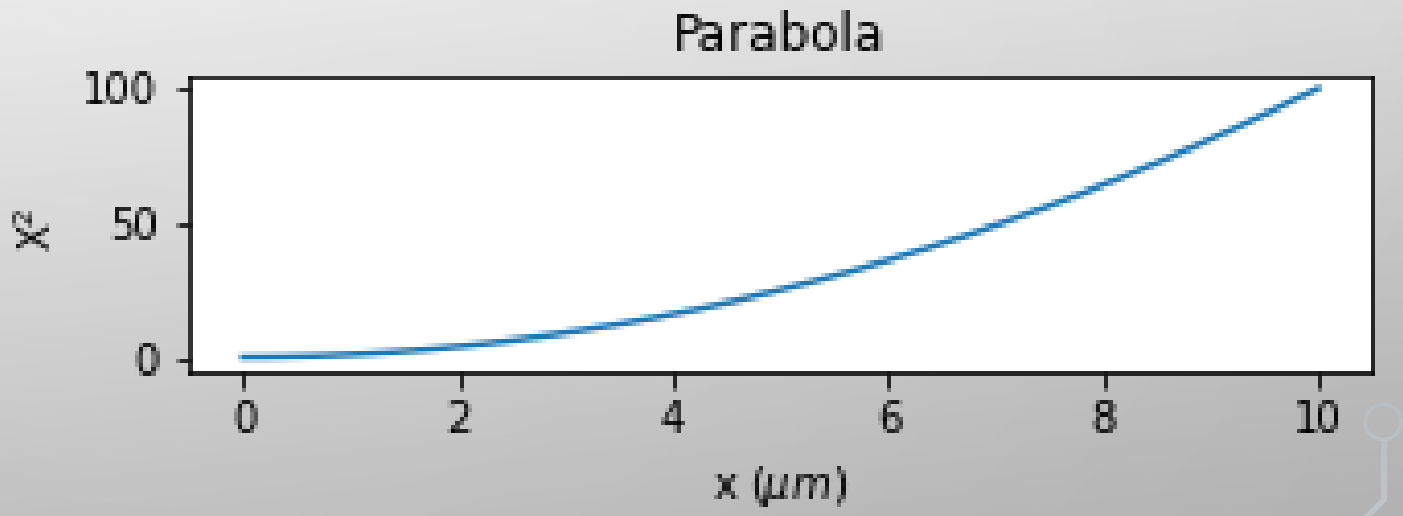
```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

ax.set_title('Parabola')
ax.set_xlabel('x ($\mu m$)')
ax.set_ylabel('X$^2$')

ax.plot(x,x**2)
```

Note: You may use LaTeX notations.



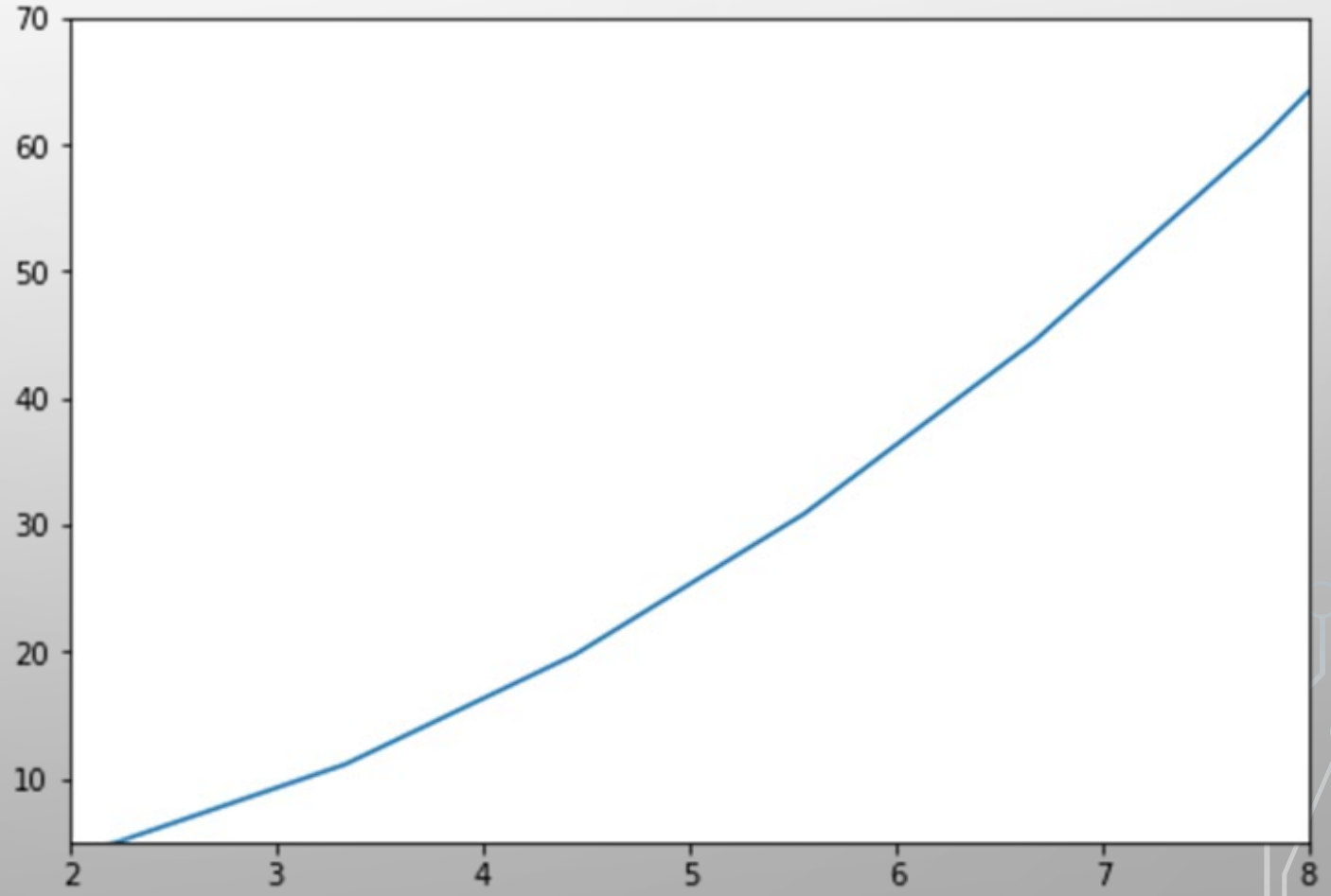
FIGSIZES(WIDTH & HEIGHT IN INCHES)

```
fig = plt.figure(figsize=(4, 1))
ax = fig.add_axes([0,0,1,1])

ax.set_title('Parabola')
ax.set_xlabel('x ($\mu m$)')
ax.set_ylabel('X$^2$')

ax.plot(x,x**2)
```



set_xlim([]) & set_ylim([])

```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

ax.set_xlim([2,8])
ax.set_ylim([5,70])

ax.plot(x,x**2)
```



LEGENDS

Labels of your plot

```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

ax.plot(x,x**2, label='Parabola')
ax.plot(x,x**1, label='Linear')

ax.legend(loc=0)
```

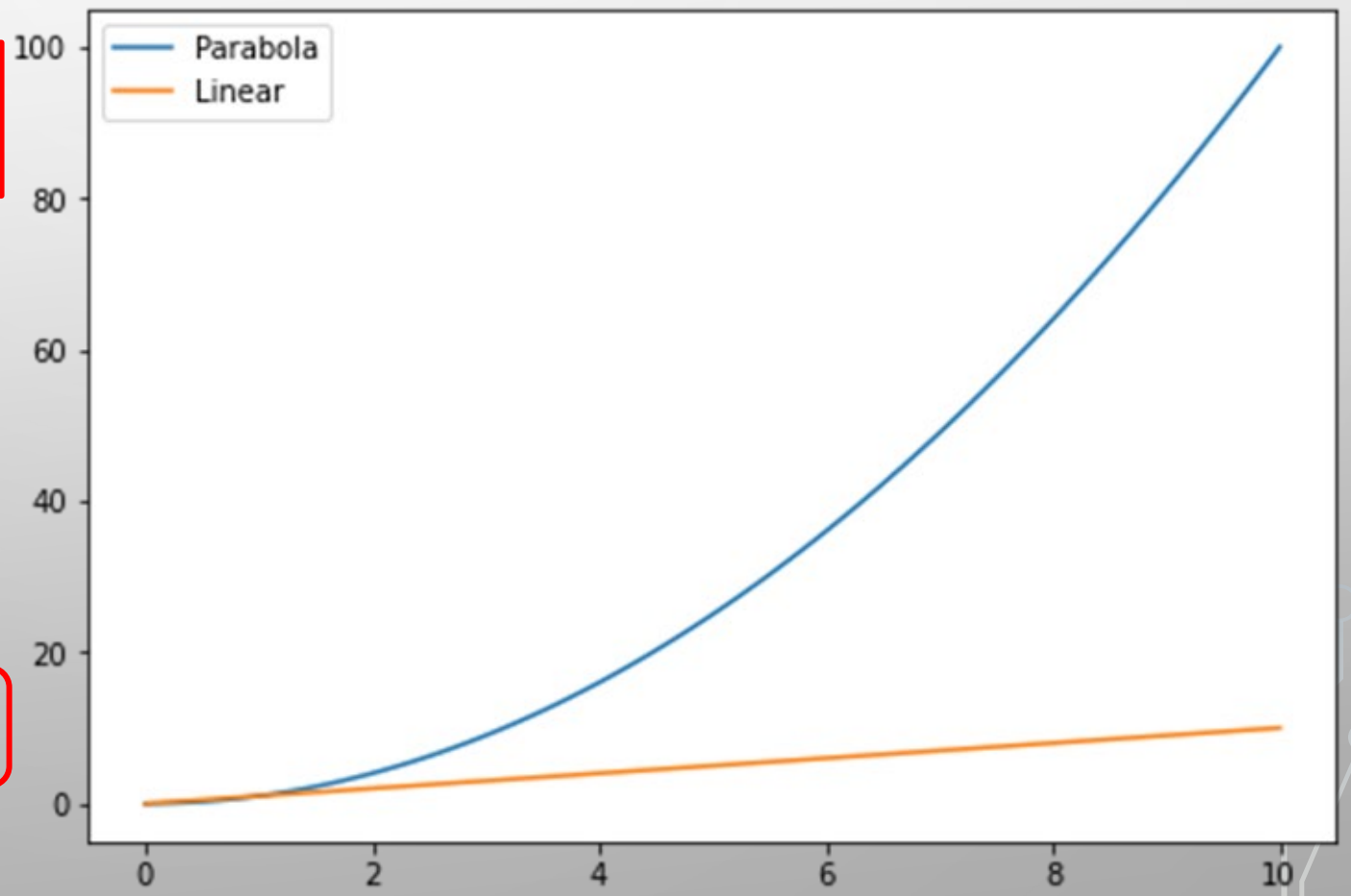
Prints the legends

3 main choices

Leave Empty

'best'	0
'upper right'	1
'upper left'	2
'lower left'	3
'lower right'	4

A tuple!
e.g. loc=(0.1,0.8)



FANCY UP

```
x=np.linspace(0,10,10)
```
→ Note: only 10 points created

```
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])

ax.plot(x,x**2,
        c='red',
        lw=2,
        ls='--',
        alpha =0.8,
        marker='o',
        markersize=10,
        markerfacecolor='yellow',
        markeredgewidth=2,
        markeredgecolor='blue'
       )
```

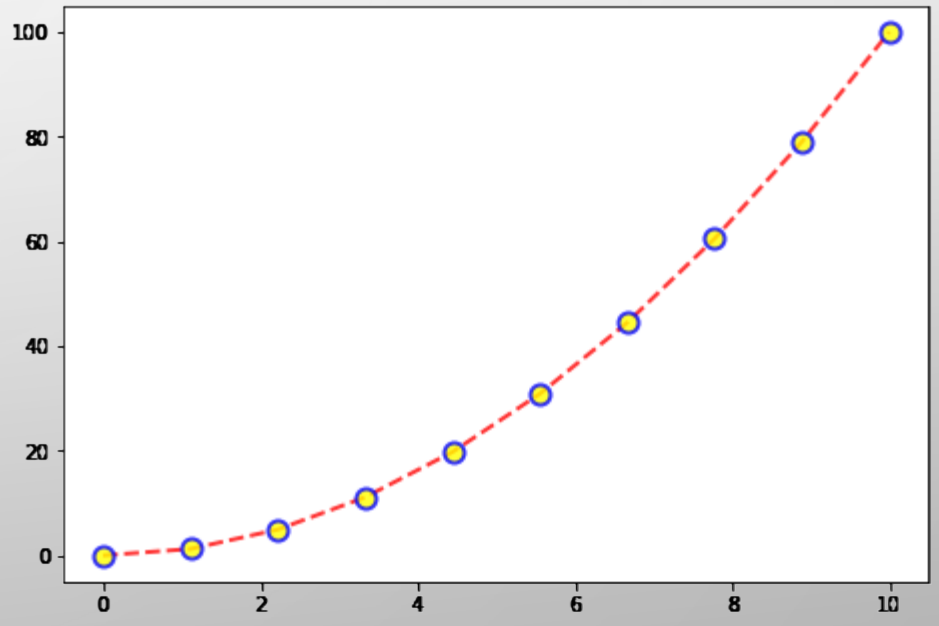
→ c or color = ‘b,g,r,c,m,y,k’→ this link

→ lw or linewidth = value

→ ls or linestyle = ‘-’, ‘--’, ‘-.’, ‘:’, ‘ ’

→ transparency

→ Point marker.
Try: 'o', 'v', '^', '<', '>', 's', 'p', '*', 'h', 'H', 'D', 'd', 'P', 'X'

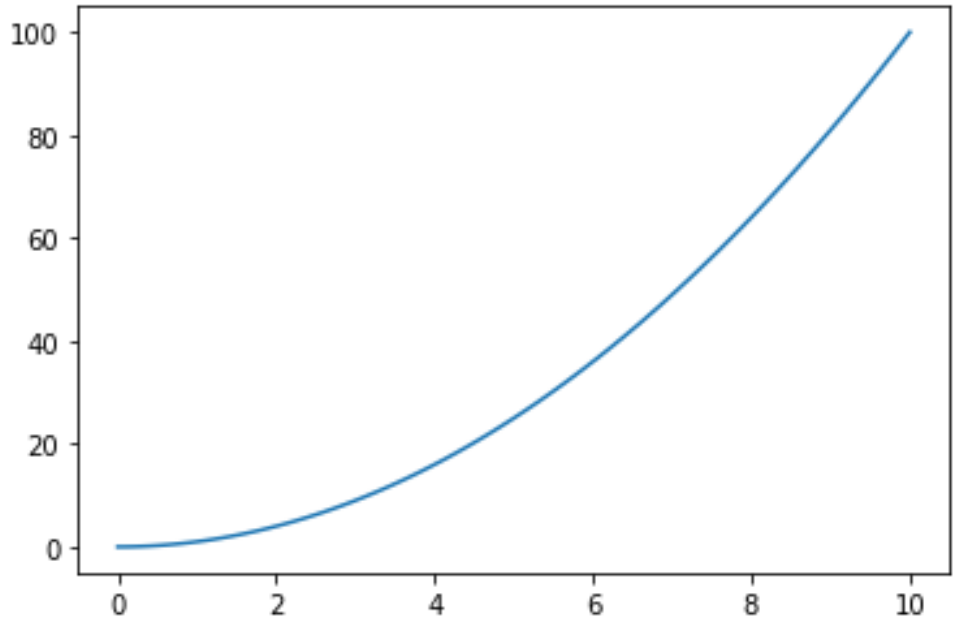


SUBPLOTS

fig, ax = plt.subplots(nrows=?, ncols=?)

```
fig, ax = plt.subplots()

ax.plot(x, x**2)
```

```
[<matplotlib.lines.Line2D at 0x2623fd23a60>]
```




```
fig, ax = plt.subplots(nrows=1, ncols=2)

ax[0].plot(x, x**2)
ax[1].plot(x,x**0.5)
```

```
[<matplotlib.lines.Line2D at 0x2623ee154f0>]
```

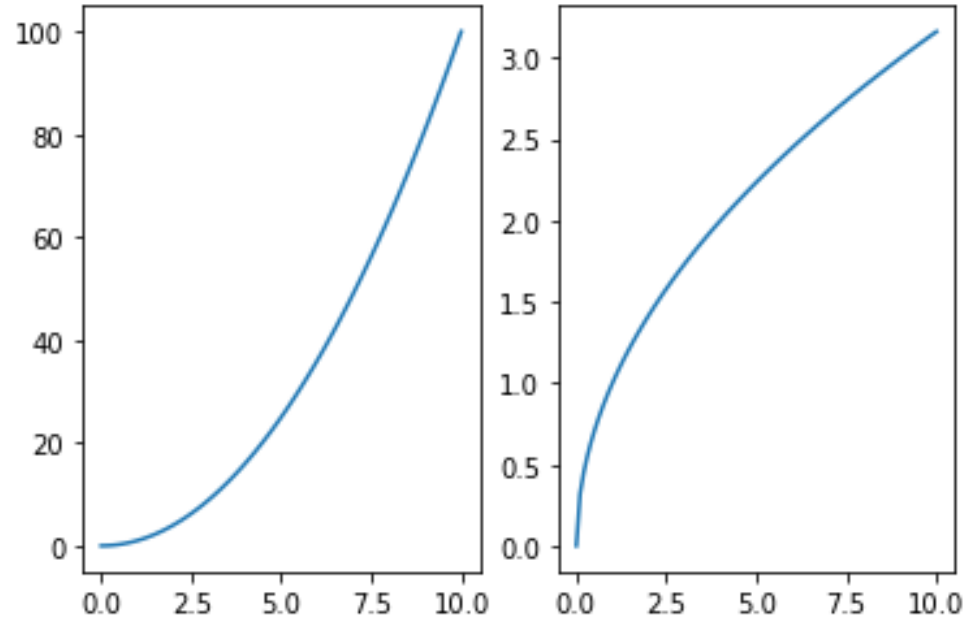
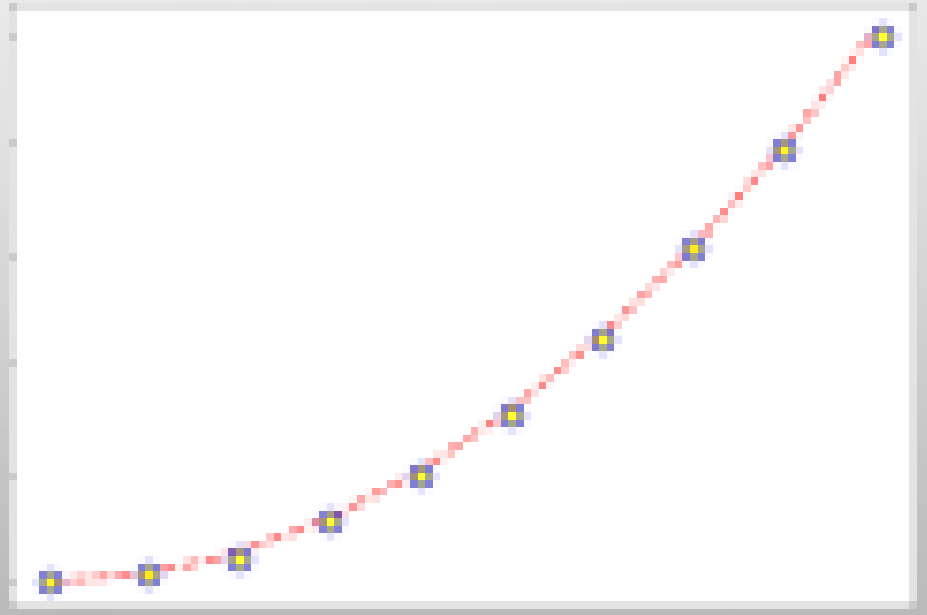
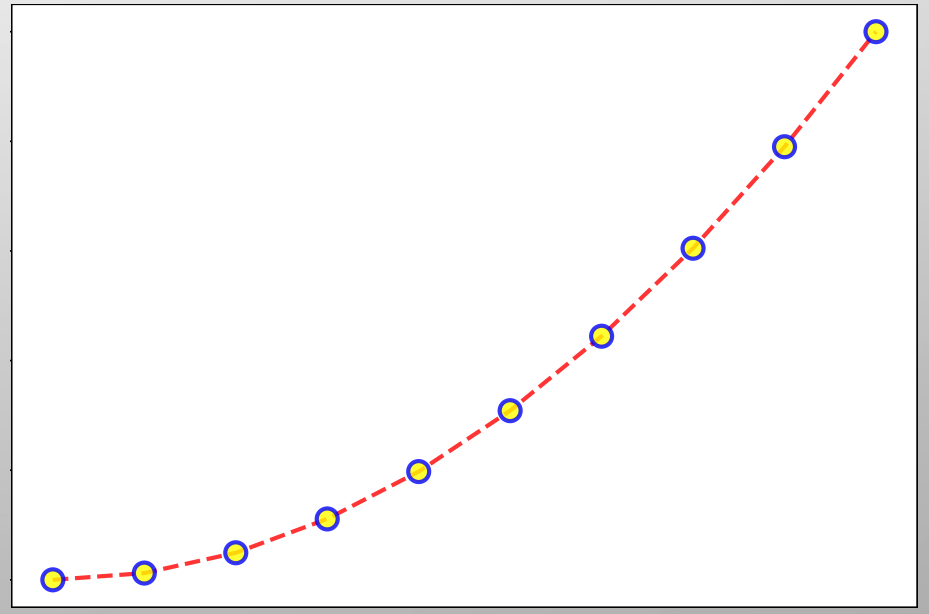


FIG.SAVEFIG

```
fig.savefig('test.png')
```



```
fig.savefig('test.svg')
```



STRETCH YOUR CODING MUSCLES!

Part_IV_Data_Visualization_b_matplotlib

INSTALL SEABORN

```
pip install seaborn
```

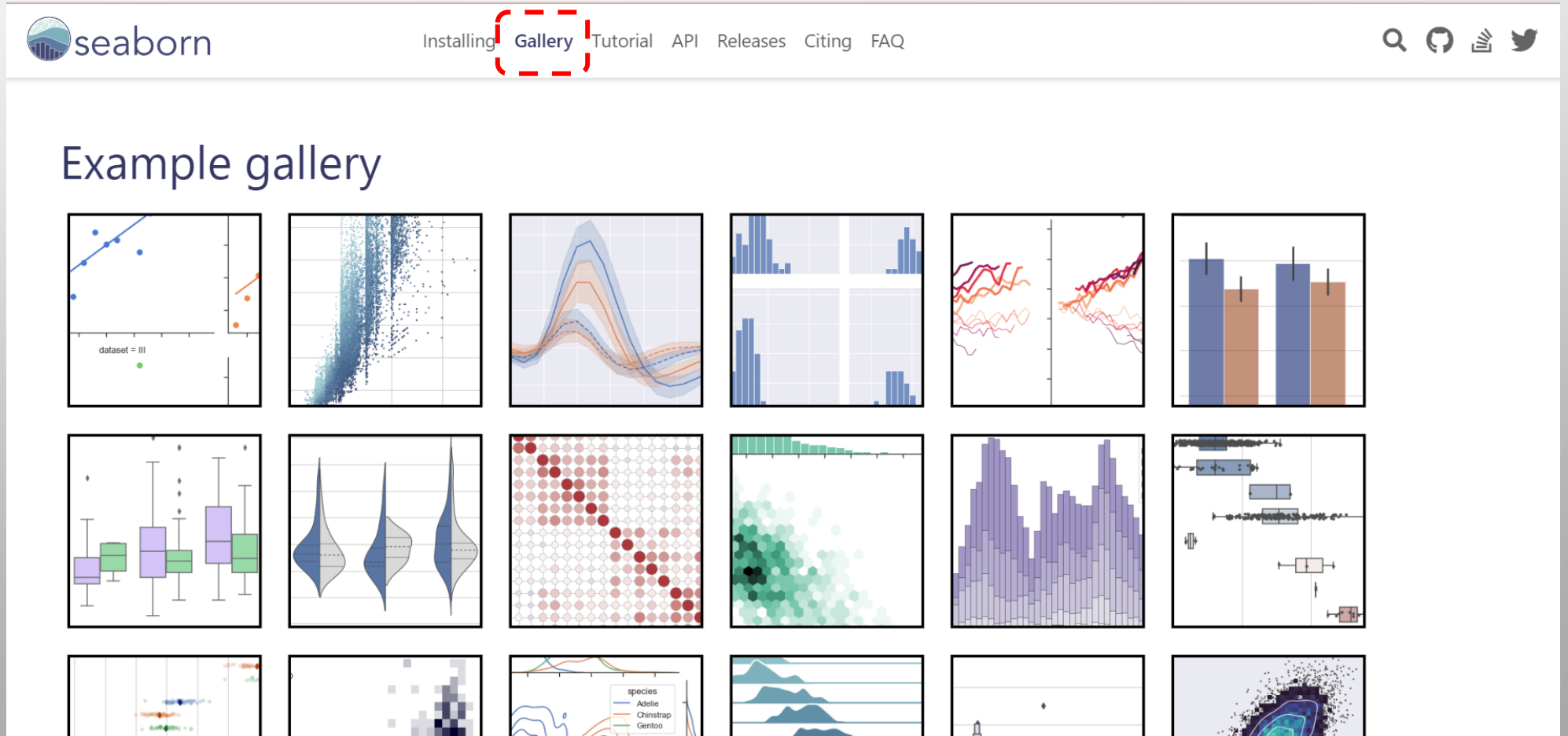
Or:

```
conda install seaborn
```

GitHub: https://github.com/mwaskom/seaborn
Documentation: https://seaborn.pydata.org/

GitHub: https://github.com/mwaskom/seaborn

Documentation: https://seaborn.pydata.org/



```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

import seaborn as sns

FROM NOW ON WE IMPORT ALL FOUR THROUGHOUT THIS COURSE

LET'S GET SOME REAL DATA!

- **CERN Electron Collision Data**
- https://www.kaggle.com/datasets/fedesoriano/cern-electron-collision-data

1) Run: The run number of the event.
2) Event: The event number.
3, 11) E1, E2: The total energy of the electron (GeV) for electrons 1 and 2.
4, 5, 6, 12, 13, 14) px1,py1,pz1,px2,py2,pz2: The components of the momemtum of the electron 1 and 2 (GeV).
7, 15) pt1, pt2: The transverse momentum of the electron 1 and 2 (GeV).
8, 16) eta1, eta2: The pseudorapidity of the electron 1 and 2.
9, 17) phi1, phi2: The phi angle of the electron 1 and 2 (rad).
10, 18) Q1, Q2: The charge of the electron 1 and 2.
19) M: The invariant mass of two electrons (GeV).

```
df=df[(df['E1']<200) &
      (df['E2']<200) &
      (abs(df['px1 '])<50) &
      (abs(df['px2'])<50) &
      (abs(df['py1'])<50) &
      (abs(df['pz1'])<50) &
      (abs(df['py2'])<50) &
      (abs(df['pz2'])<50)
     ]

df=df[:200]
```

```
df=pd.read_csv('Data/Cern/archive/dielectron.csv')
```

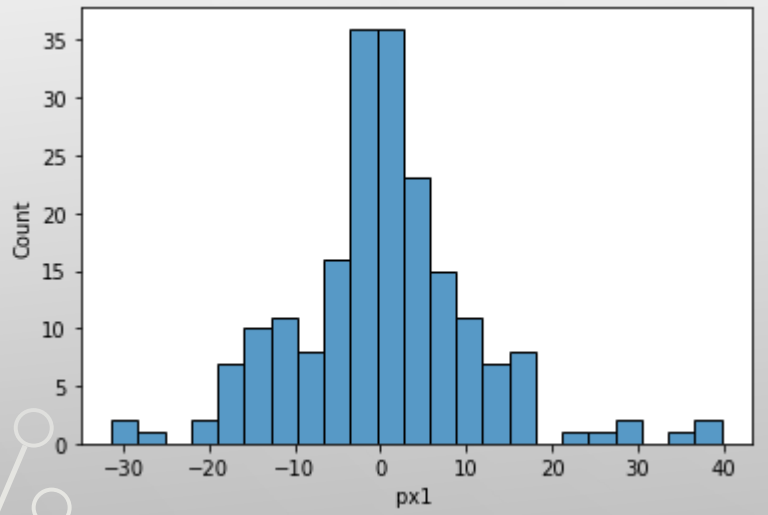
```
df.head()
```

	Run	Event	E1	px1	py1	pz1	pt1	eta1	phi1	Q1	E2	px2	py2	pz2	pt2	
0	147115	366639895	58.71410	-7.31132	10.531000	-57.29740	12.82020	-2.20267	2.17766	1	11.2836	-1.032340	-1.88066	-11.0778	2.14537	-2.34
1	147115	366704169	6.61188	-4.15213	-0.579855	-5.11278	4.19242	-1.02842	-3.00284	-1	17.1492	-11.713500	5.04474	11.4647	12.75360	0.80
2	147115	367112316	25.54190	-11.48090	2.041680	22.72460	11.66100	1.42048	2.96560	1	15.8203	-1.472800	2.25895	-15.5888	2.69667	-2.45
3	147115	366952149	65.39590	7.51214	11.887100	63.86620	14.06190	2.21838	1.00721	1	25.1273	4.087860	2.59641	24.6563	4.84272	2.33
4	147115	366523212	61.45040	2.95284	-14.622700	-59.61210	14.91790	-2.09375	-1.37154	-1	13.8871	-0.277757	-2.42560	-13.6708	2.44145	-2.42

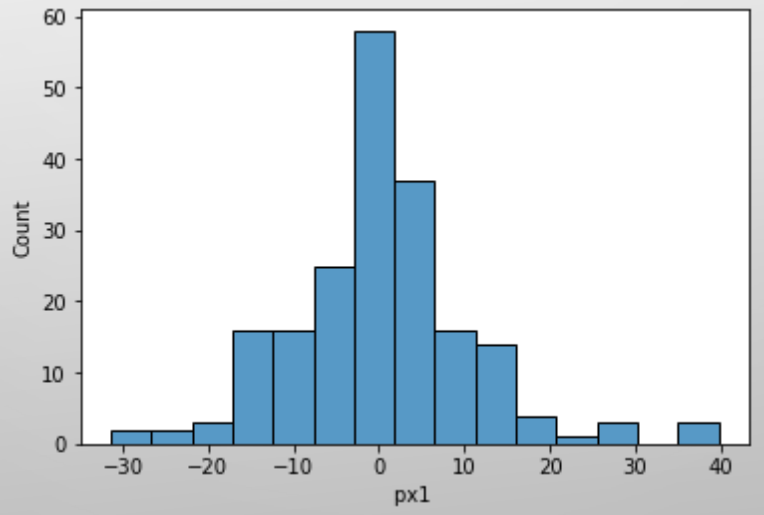
Note: for the simplicity I removed all the high energetic ones and only chose the first 200 data entries!

sns.histplot()*

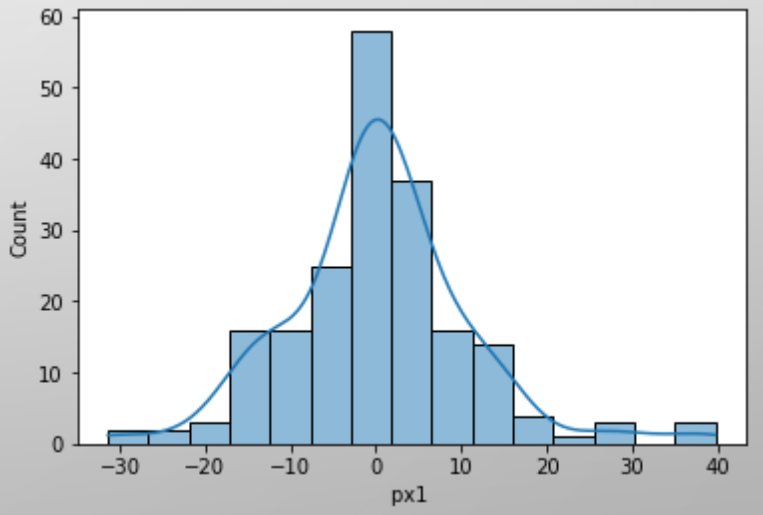
```
sns.histplot(df['px1 '])
```




```
sns.histplot(df['px1 '], bins=15)
```



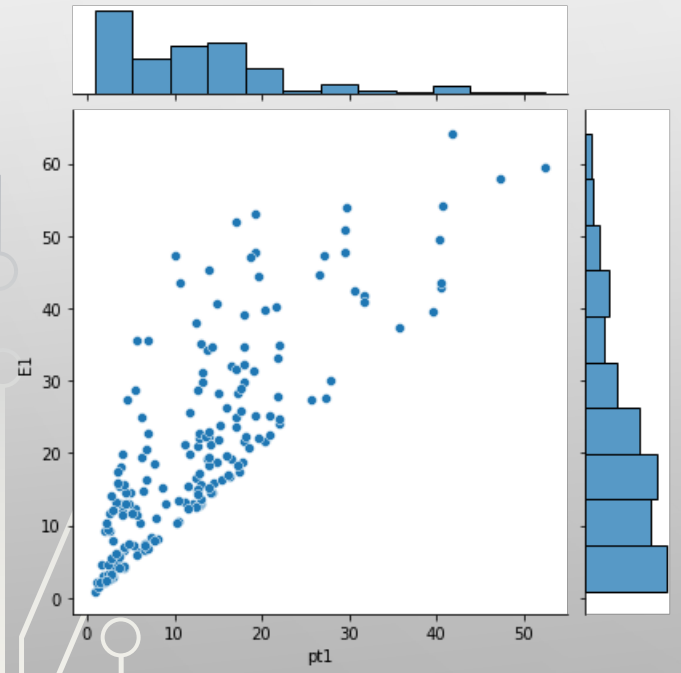

```
sns.histplot(df['px1 '], bins=50, kde=True)
```



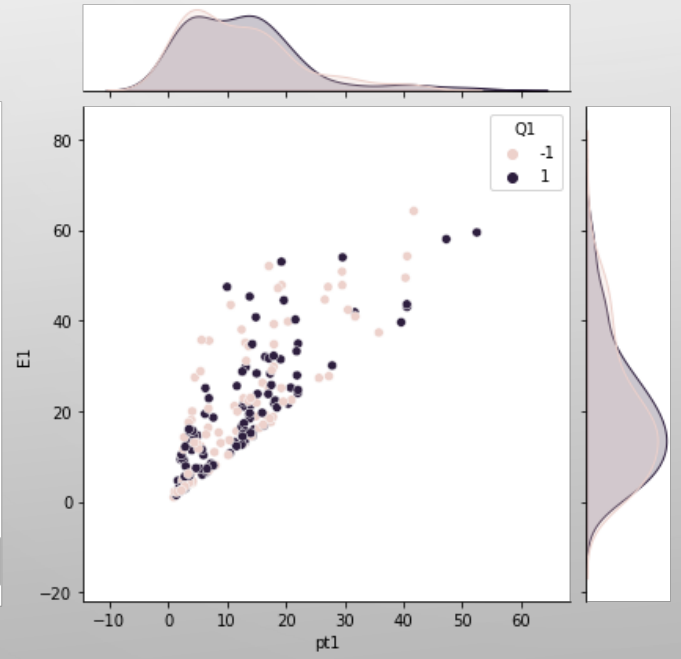
* You may see distplot() or displot() (no t in the name)

sns.jointplot(x= , y= , data= , hue= , palette=)

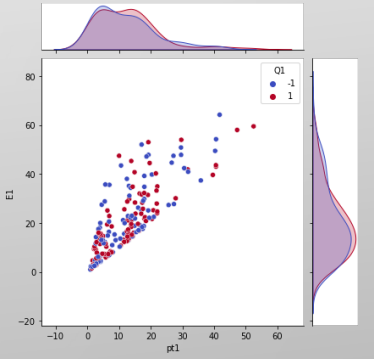
```
sns.jointplot(x='pt1', y='E1', data=df)
```



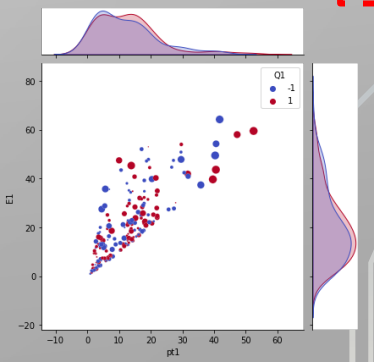

```
sns.jointplot(x='pt1', y='E1', data=df, hue='Q1')
```




```
sns.jointplot(x='pt1', y='E1', data=df, hue='Q1', palette='coolwarm')
```

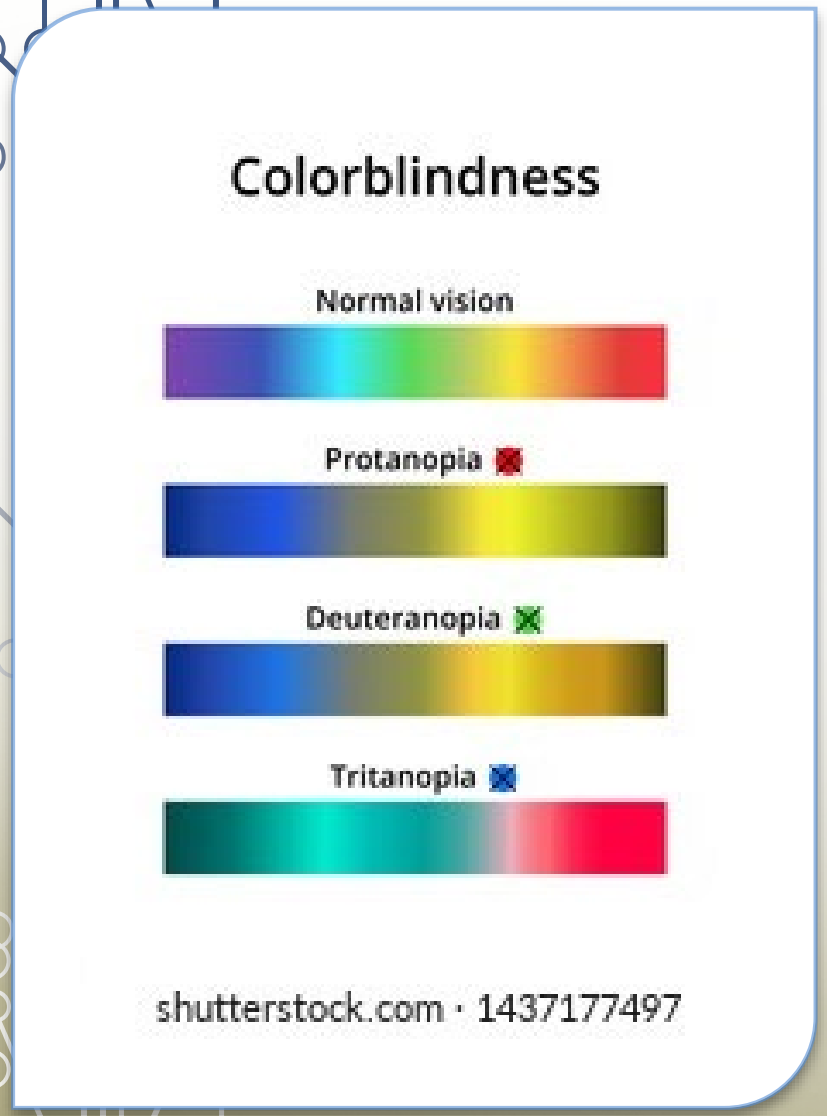



```
sns.jointplot(x='pt1', y='E1', data=df[:200], hue='Q1', palette='coolwarm', s=df['M'])
```



Try this for fun:

```
sns.jointplot(x='pt1', y='E1', data=df[:200], hue='pt1', palette='rainbow')
```



BE MINDFUL!

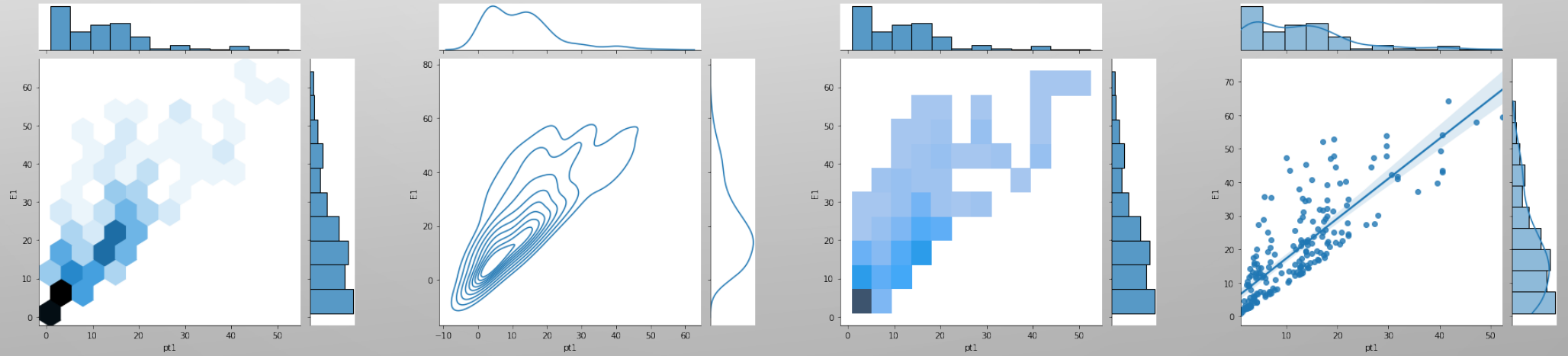
PEOPLE MOST LIKELY MIGHT BE ABLE TO DISTINGUISH RED AND BLUE

sns.jointplot(x= , y= , data= , hue= , palette=)

```
sns.jointplot(x='pt1', y='E1', data=df, kind='hex')
```

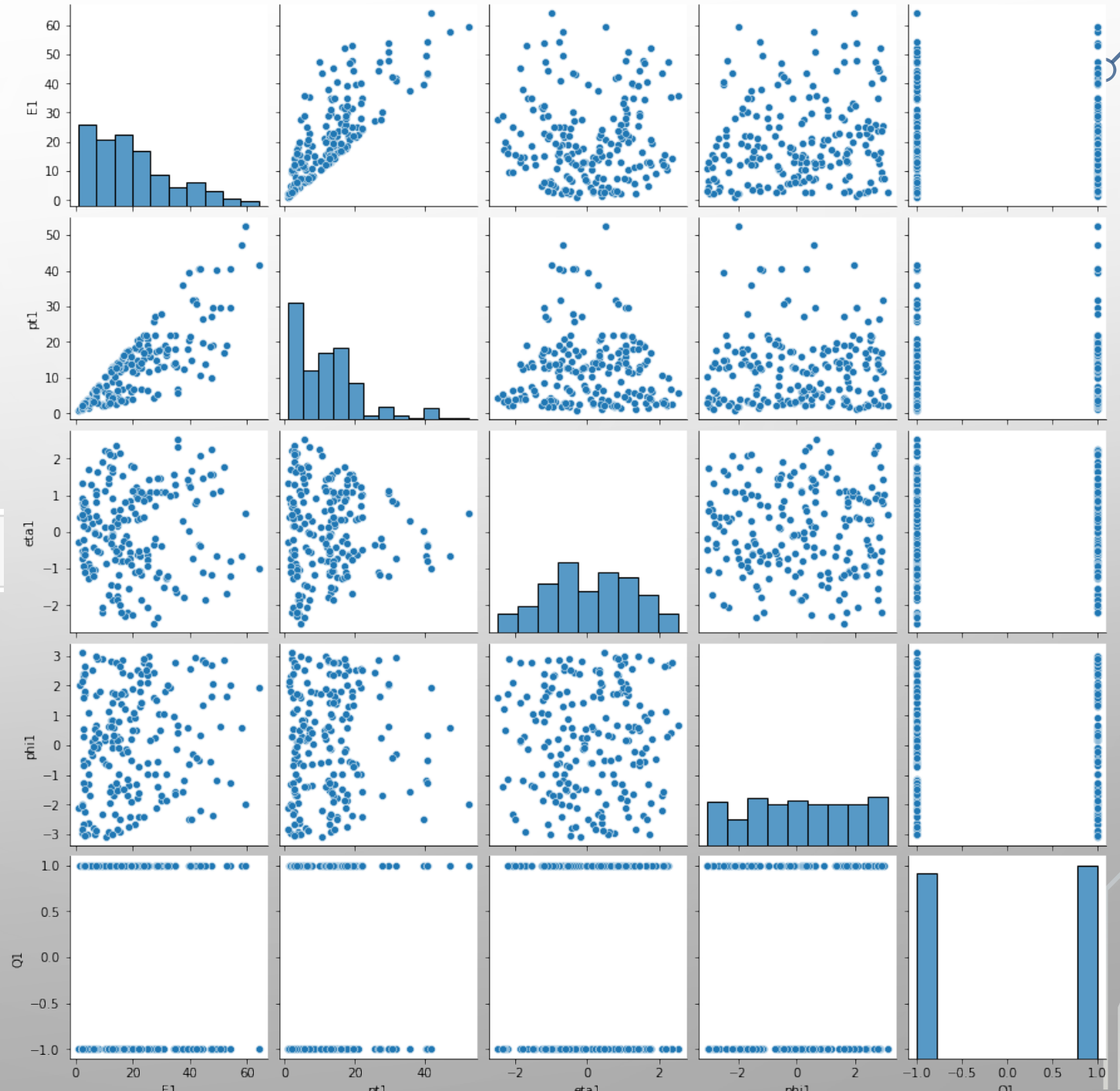
Try: kind= ‘hex’, ‘kde’, ‘hist’, ‘reg’

```
Signature:
sns.jointplot(
    *,
    x=None,
    y=None,
    data=None,
    kind='scatter',
    color=None,
    height=6,
    ratio=5,
```



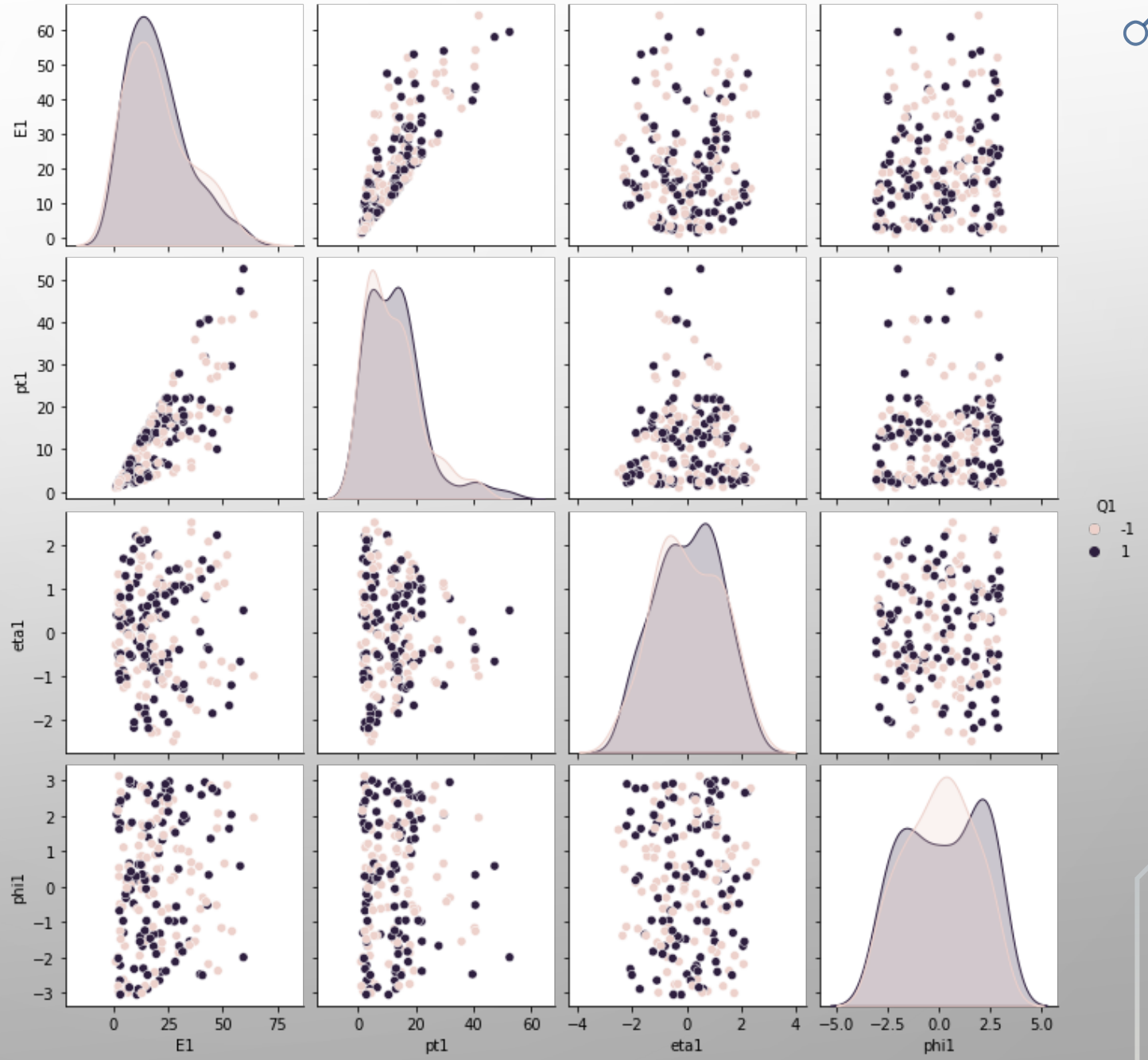
sns.pairplot() Everything vs Everything!

```
df1 = df[['E1', 'pt1', 'eta1', 'phi1', 'Q1']]
sns.pairplot(df1)
```



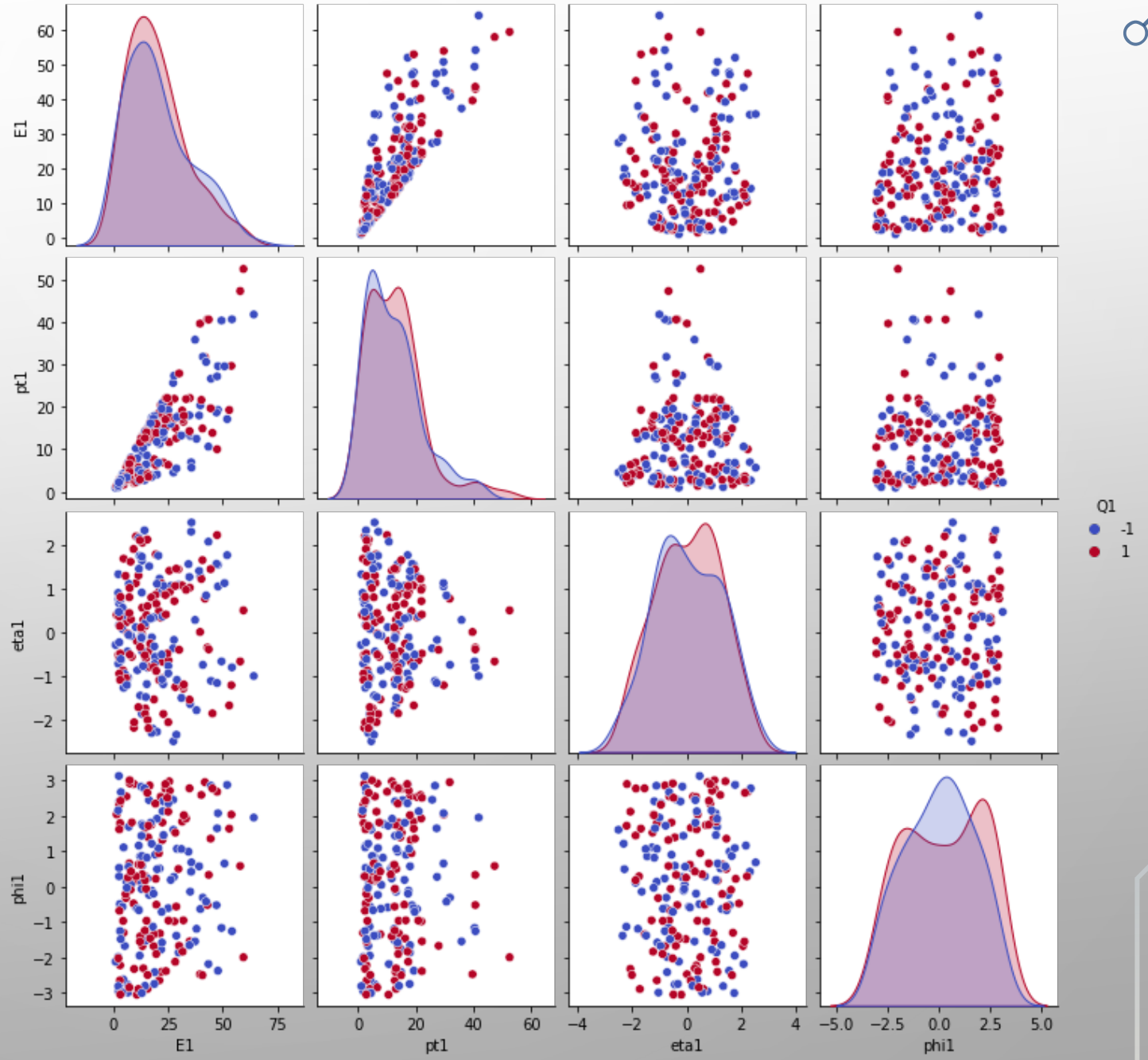
sns.pairplot() Everything vs Everything!

```
df1 = df[['E1', 'pt1', 'eta1', 'phi1', 'Q1']]
sns.pairplot(df1, hue='Q1')
```



sns.pairplot() Everything vs Everything!

```
df1 = df[['E1', 'pt1', 'eta1', 'phi1', 'Q1']]
sns.pairplot(df1, hue='Q1', palette='coolwarm')
```

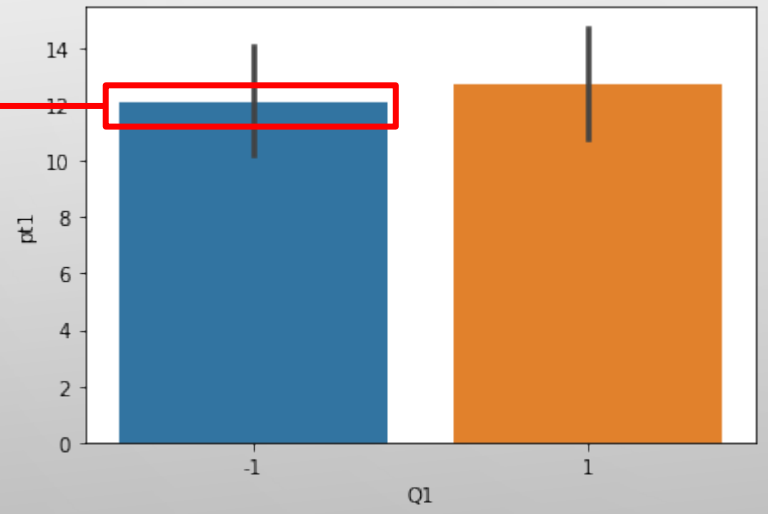


EXPLORE YOURSELF

- sns.PairGrid()
- sns.FacetGrid()

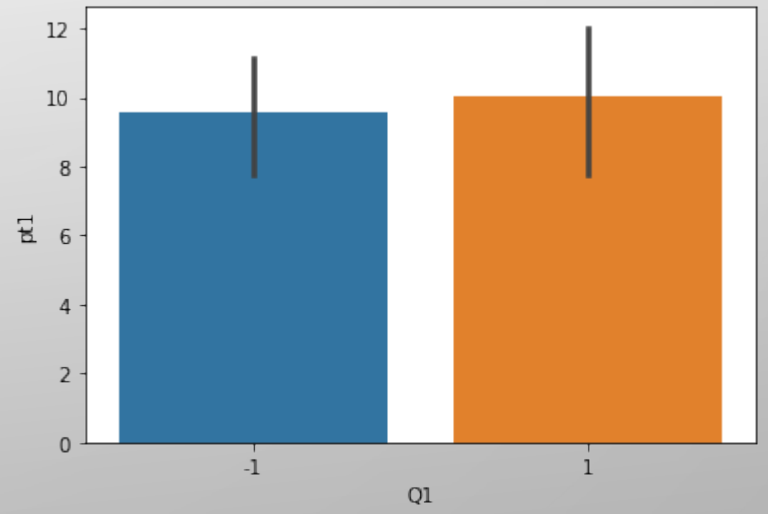
sns.barplot(x=, y=, data=)

```
sns.barplot(x='Q1', y='pt1', data=df)
```



By default, this is the mean value!

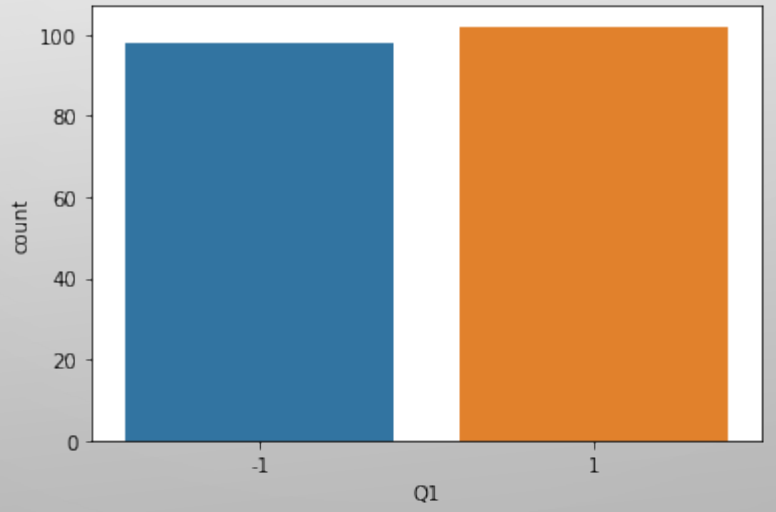
```
sns.barplot(x='Q1', y='pt1', data=df, estimator=np.std)
```



You may change it to a different estimator though!

sns.countplot(x=, data=)

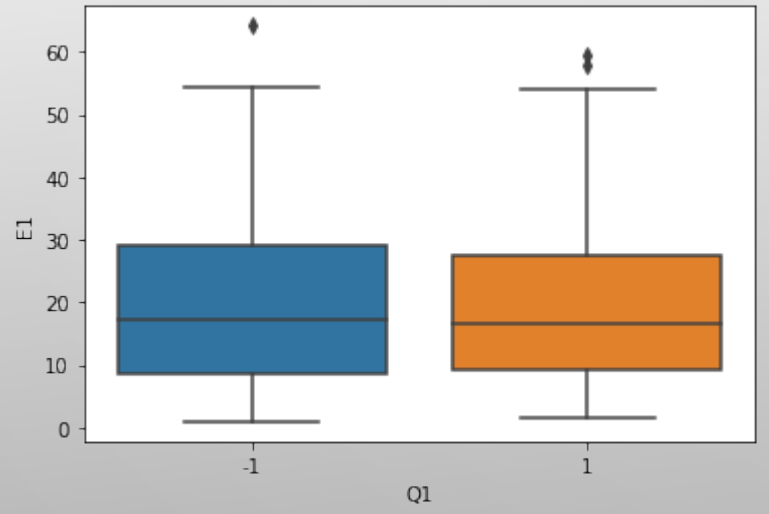
```
sns.countplot(x='Q1', data=df)
```



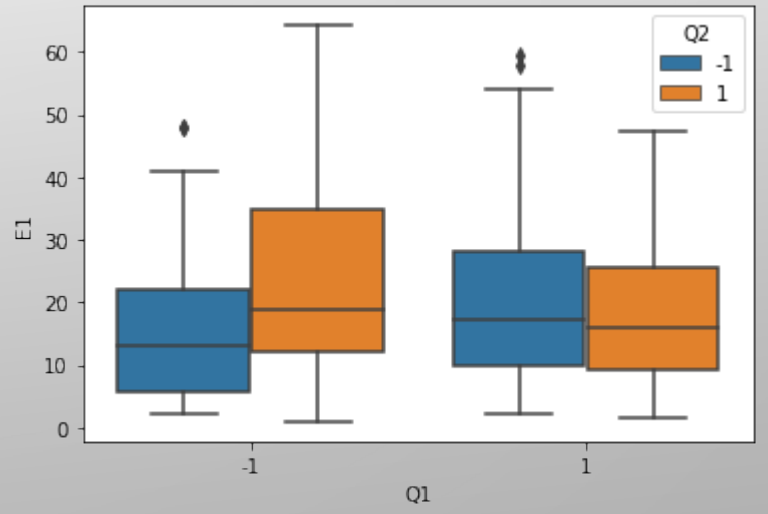
sns.boxplot(x=, y=, data=)

An example of true power of seaborn!

```
sns.boxplot(x='Q1', y='E1', data=df)
```

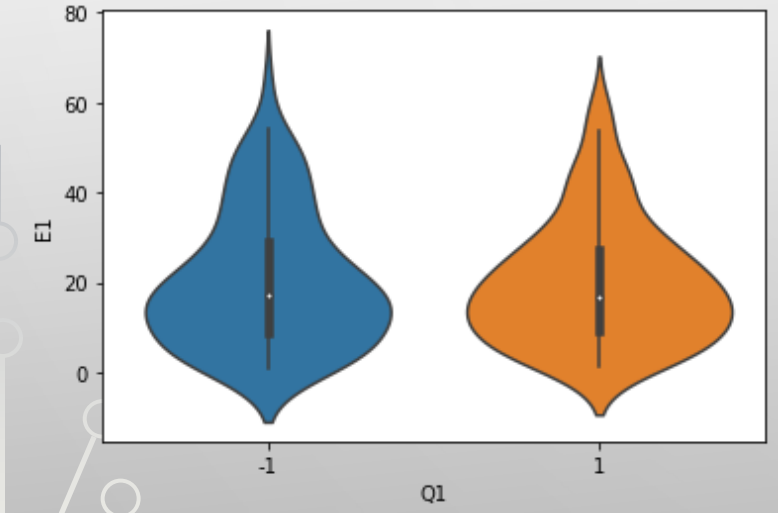


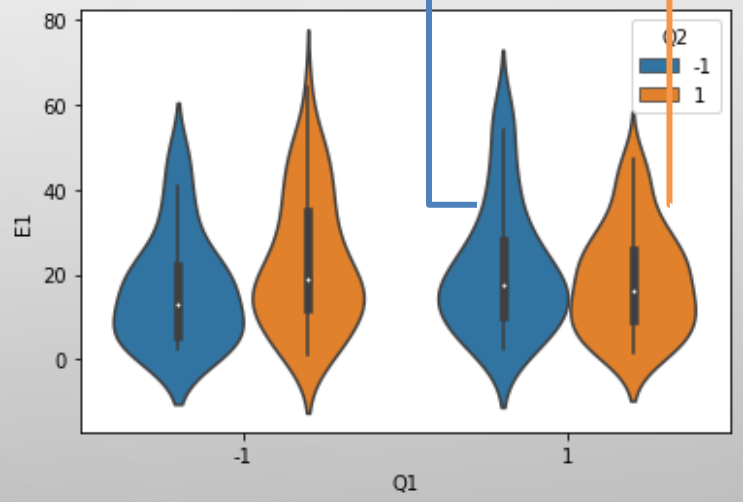

```
sns.boxplot(x='Q1', y='E1', data=df, hue='Q2')
```

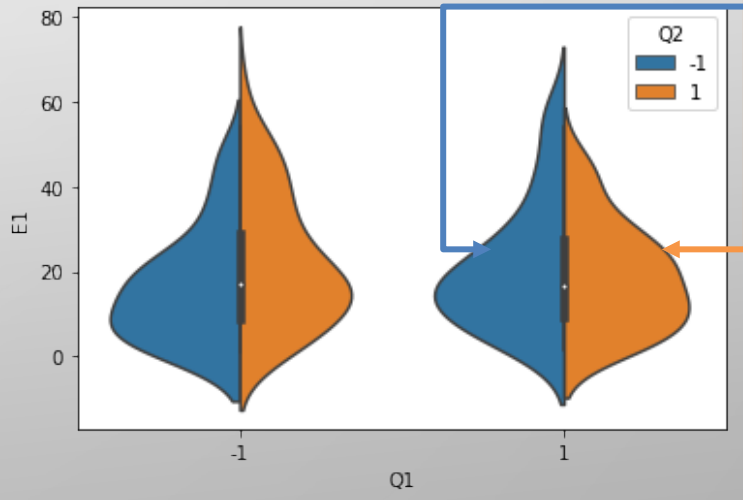


sns.violinplot(x=, y=, data=)

```
sns.violinplot(x='Q1', y='E1', data=df, hue='Q2', split=True)
```





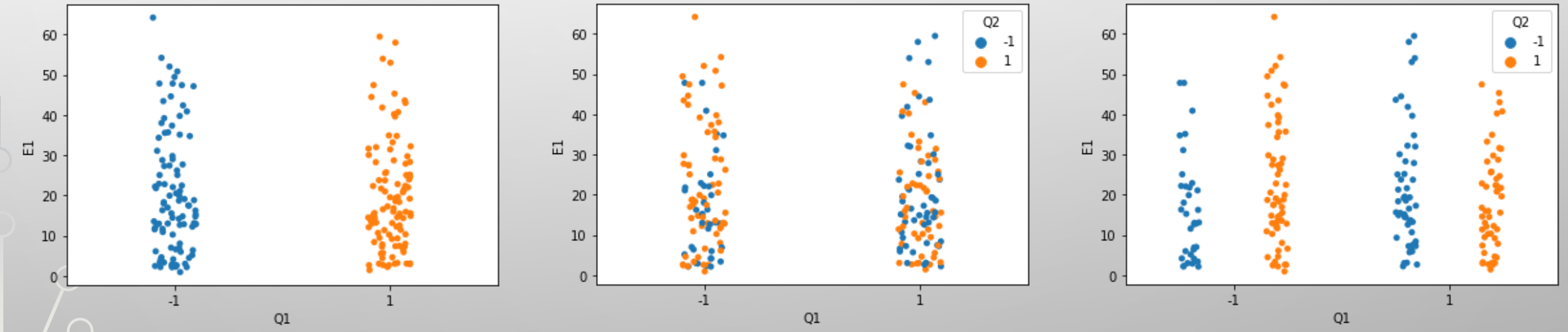



```
sns.violinplot(x='Q1', y='E1', data=df)
```

```
sns.violinplot(x='Q1', y='E1', data=df, hue='Q2')
```

sns.stripplot(x=, y=, data=)

```
sns.stripplot(x='Q1', y='E1', data=df, hue='Q2', dodge=True)
```

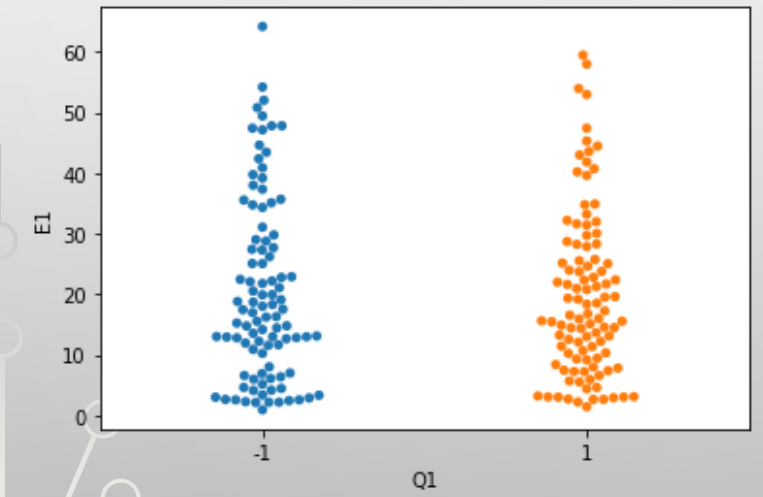


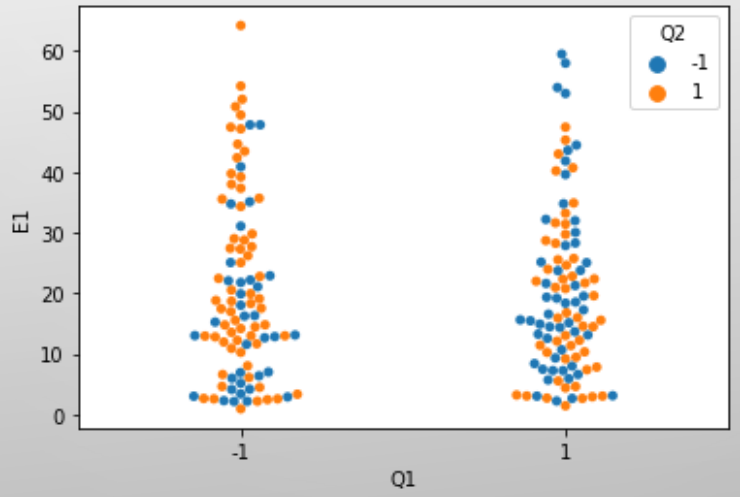

```
.stripplot(x='Q1', y='E1', data=df)
```

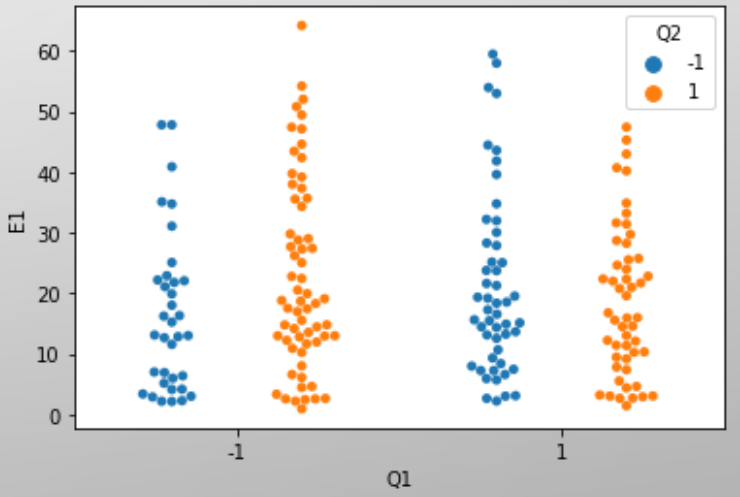
```
sns.stripplot(x='Q1', y='E1', data=df, hue='Q2')
```

sns.swarmplot(x=, y=, data=) violin+stripplot

```
sns.swarmplot(x='Q1', y='E1', data=df, hue='Q2', dodge=True)
```








```
sns.swarmplot(x='Q1', y='E1', data=df)
```

```
sns.swarmplot(x='Q1', y='E1', data=df, hue='Q2')
```

QUICK NOTE!

Personally:

- I used barplot, & boxplot
- I have never used violinplot, stripplot, or swarmplot
- Good to know though!

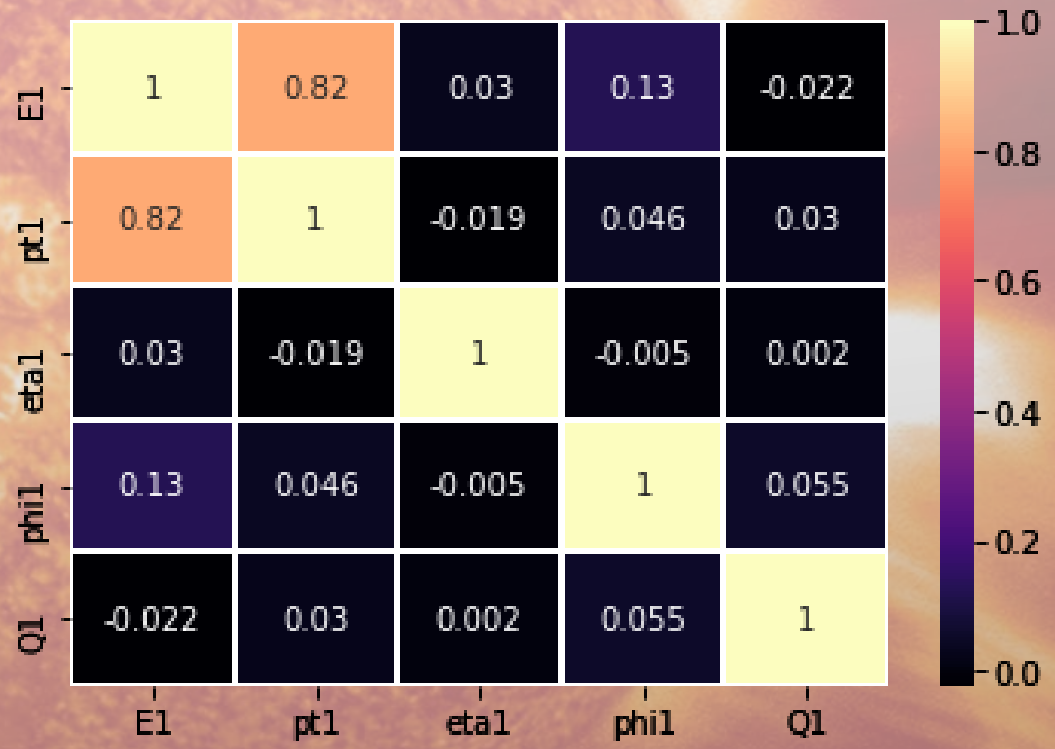
- Lots of power! Use the power wisely!

sns.heatmap(dataframe, annot=, cmap=)

```
df1 = df[['E1', 'pt1', 'eta1', 'phi1', 'Q1']]
```

```
cor_1=df1.corr()
```

	E1	pt1	eta1	phi1	Q1
E1	1.000000	0.817679	0.030464	0.127600	-0.022010
pt1	0.817679	1.000000	-0.019016	0.045827	0.029695
eta1	0.030464	-0.019016	1.000000	-0.005012	0.001969
phi1	0.127600	0.045827	-0.005012	1.000000	0.054918
Q1	-0.022010	0.029695	0.001969	0.054918	1.000000




```
sns.heatmap(cor_1, cmap='magma', annot=True, linewidth=1, linecolor='white')
```

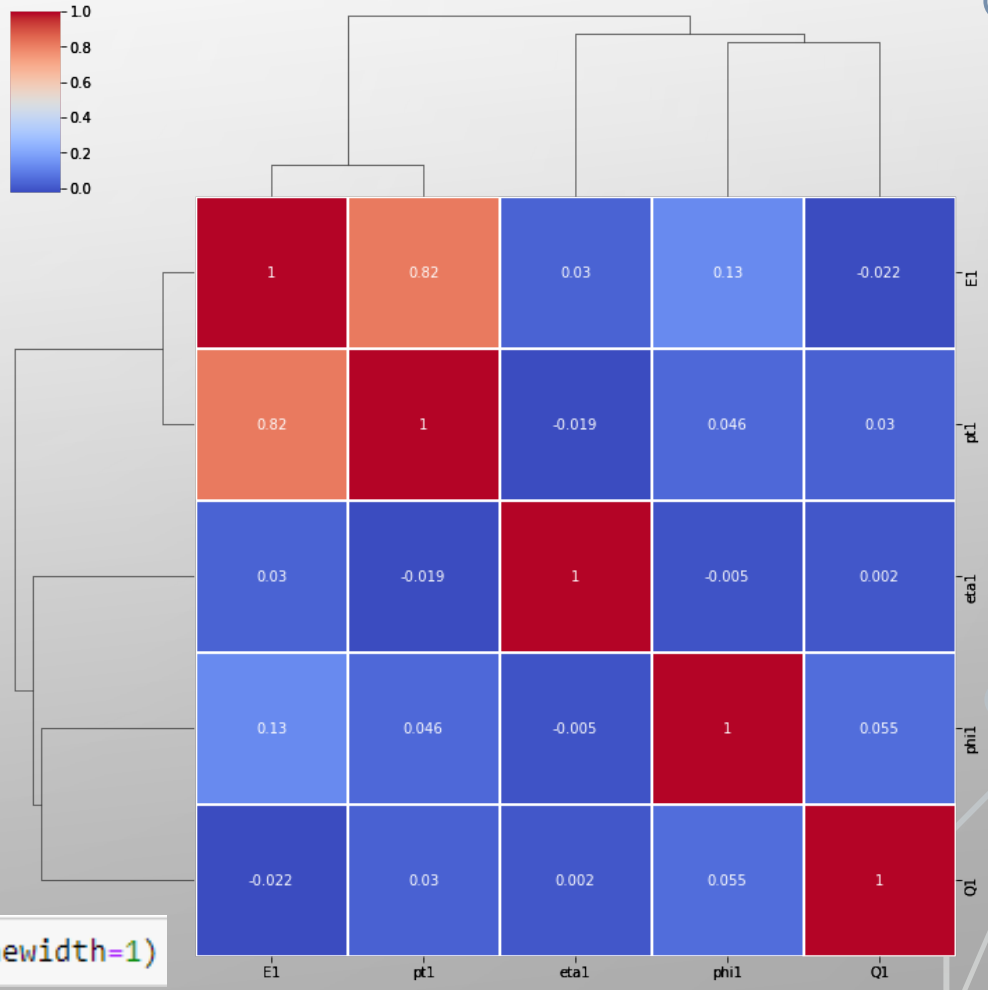
sns.clustermap(dataframe, annot=, cmap=)

```
df1 = df[['E1', 'pt1', 'eta1', 'phi1', 'Q1']]
```

```
cor_1=df1.corr()
```

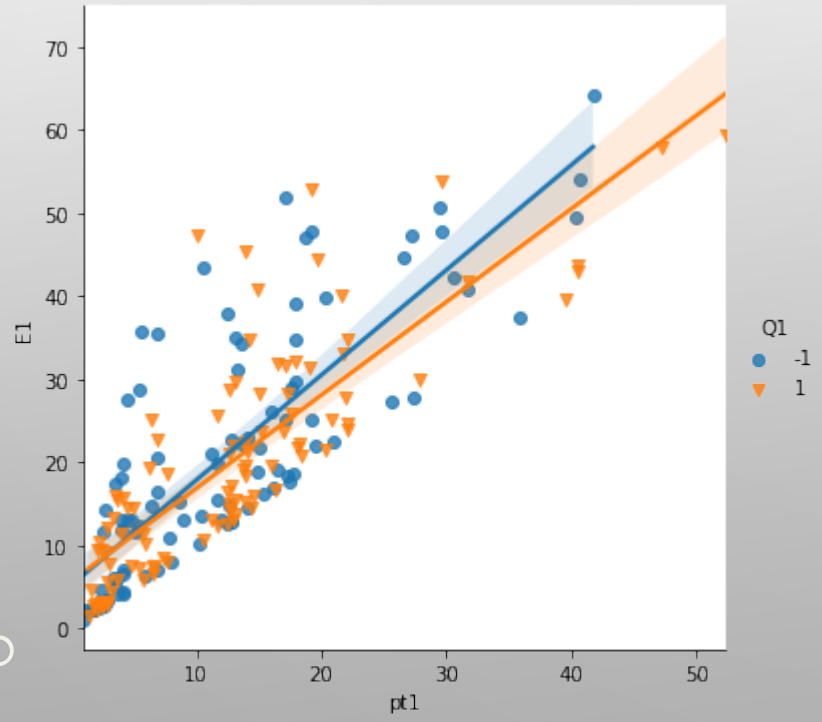
	E1	pt1	eta1	phi1	Q1
E1	1.000000	0.817679	0.030464	0.127600	-0.022010
pt1	0.817679	1.000000	-0.019016	0.045827	0.029695
eta1	0.030464	-0.019016	1.000000	-0.005012	0.001969
phi1	0.127600	0.045827	-0.005012	1.000000	0.054918
Q1	-0.022010	0.029695	0.001969	0.054918	1.000000

```
sns.clustermap(cor_1, cmap='coolwarm', annot=True, linecolor='w', linewidth=1)
```



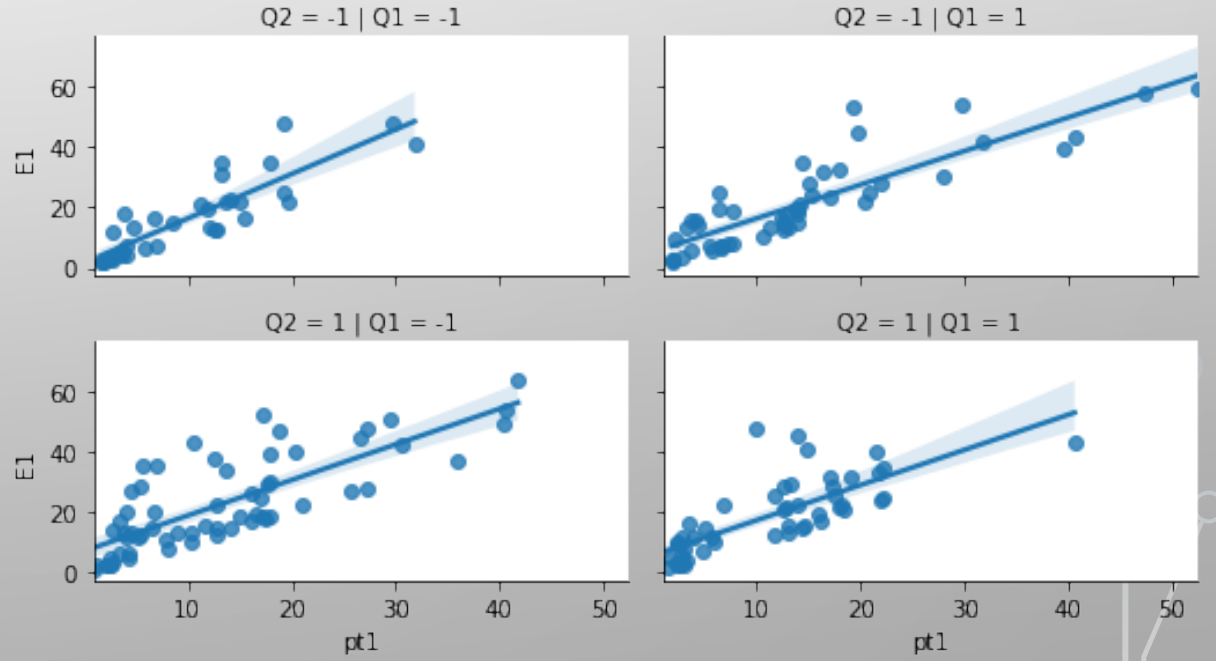
Sns.Lmplot(x=, y=, data=)

```
sns.lmplot(x='pt1', y='E1', data=df,
           hue='Q1',
           markers=['o','v'])
```




```
sns.lmplot(x='pt1', y='E1', data=df,
           col='Q1',
           row='Q2',
           height=2,
           aspect=1.8
          )
```

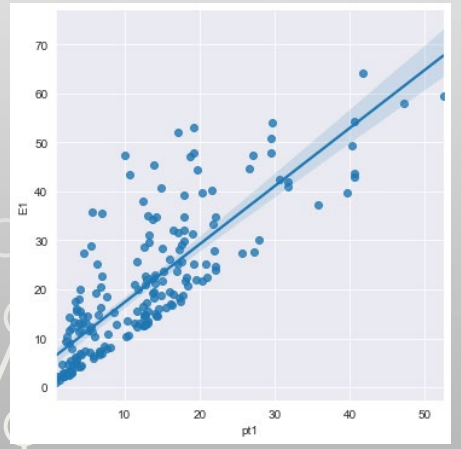
```
height : scalar Height (in inches)
of each facet.
aspect : scalar Aspect ratio of
each facet, so that ``aspect * height``
gives the width of each facet in inches.
```



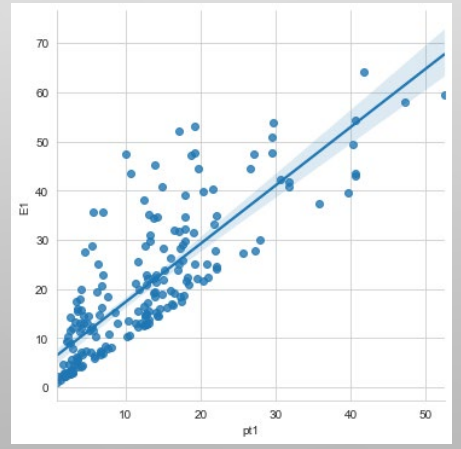
sns.set_style(darkgrid, whitegrid, dark, white, ticks)

- sns.set_style(darkgrid, whitegrid, dark, white, ticks)

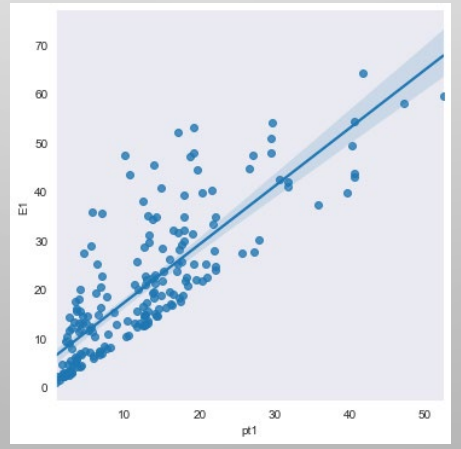
darkgrid,



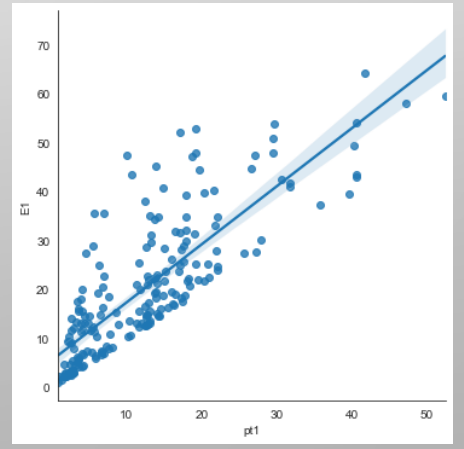
whitegrid,



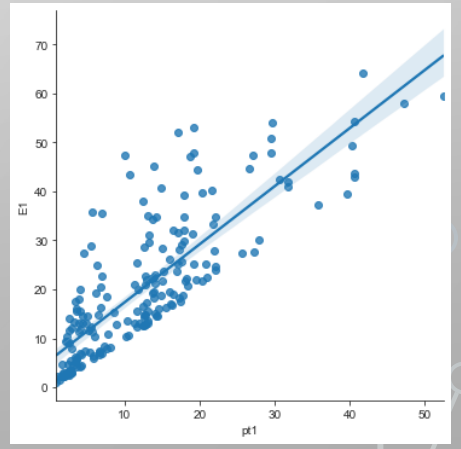
dark,



white,

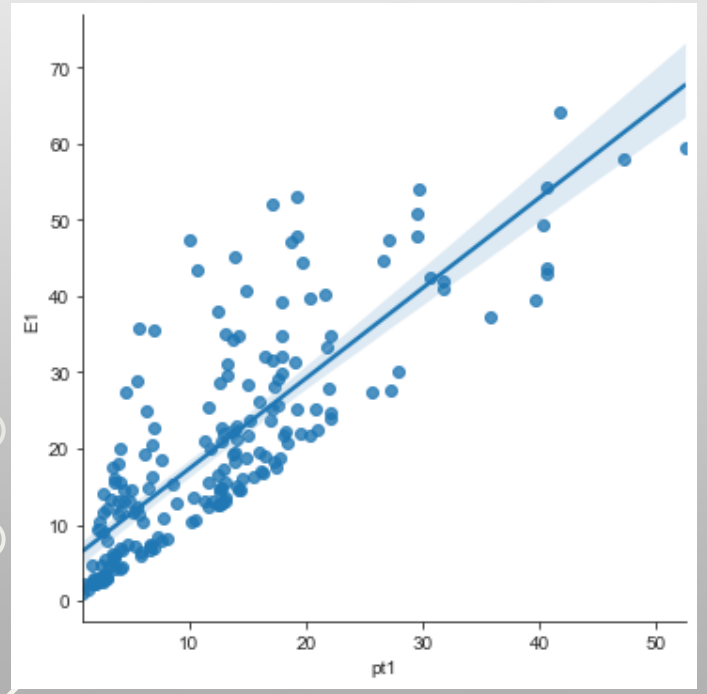


ticks

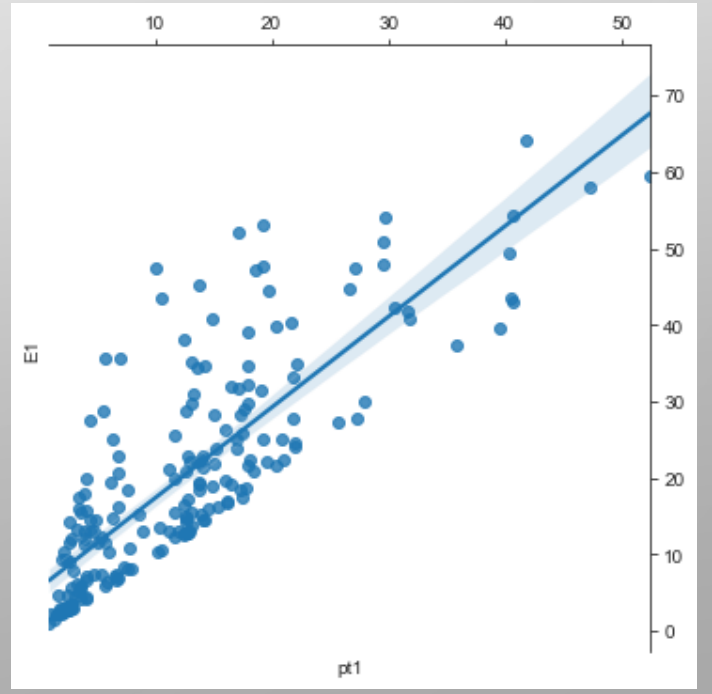


sns.despine( )

```
Signature:
sns.despine(
    fig=None,
    ax=None,
    top=True,
    right=True,
    left=False,
```



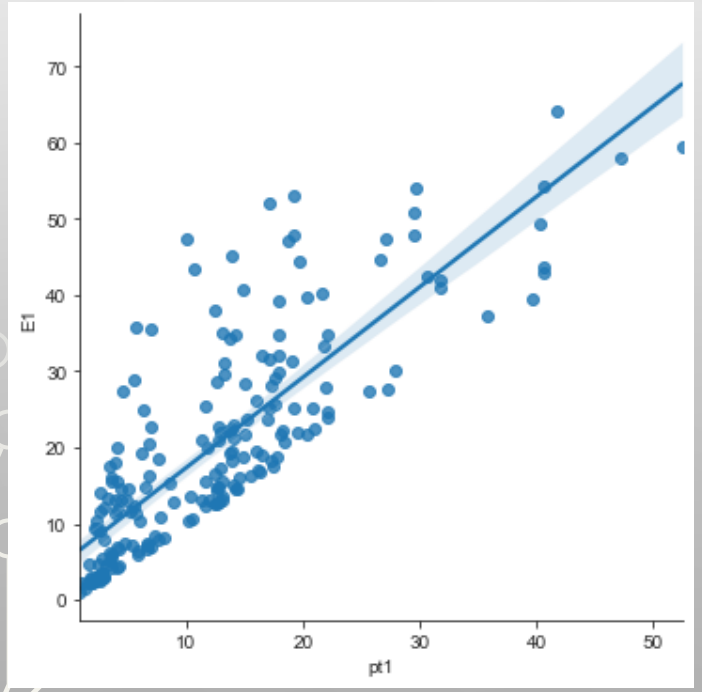

```
sns.despine(left=True, bottom=True,
            top=False, right=False)
```



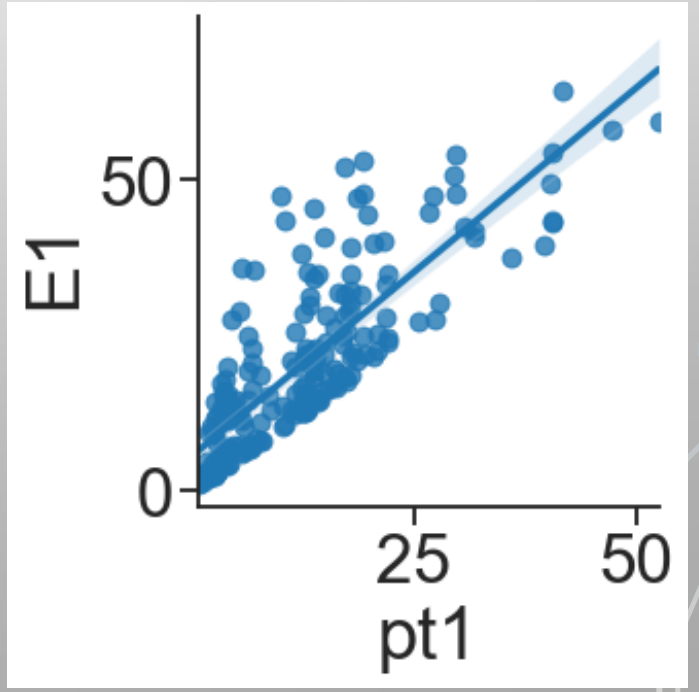
sns.set_contex(contex= , fontscale=)

{paper, notebook, talk, poster}

- sns.set_style(darkgrid, whitegrid, dark, white, ticks)



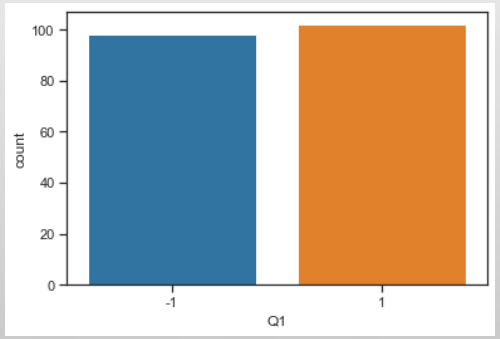

```
sns.set_context(context='talk', font_scale=2)
```



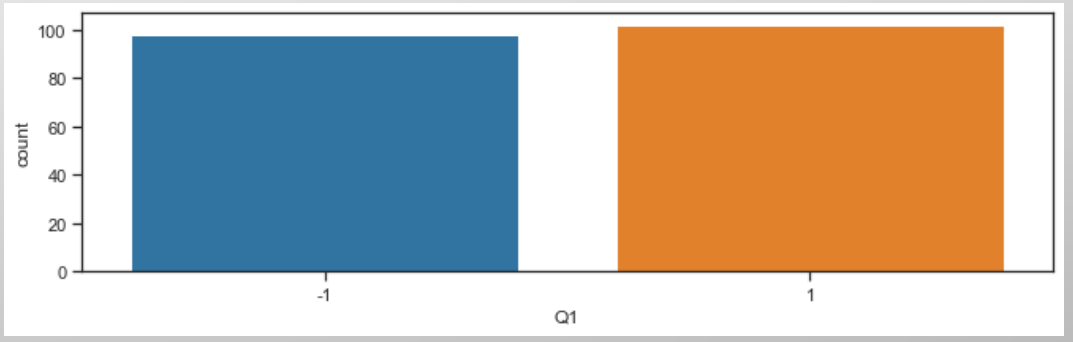
TWO MORE THINGS

- Use your Matplotlib knowledge and combine it with seaborn:

```
#plt.figure(figsize=(11,3))
sns.countplot(x='Q1', data=df)
```

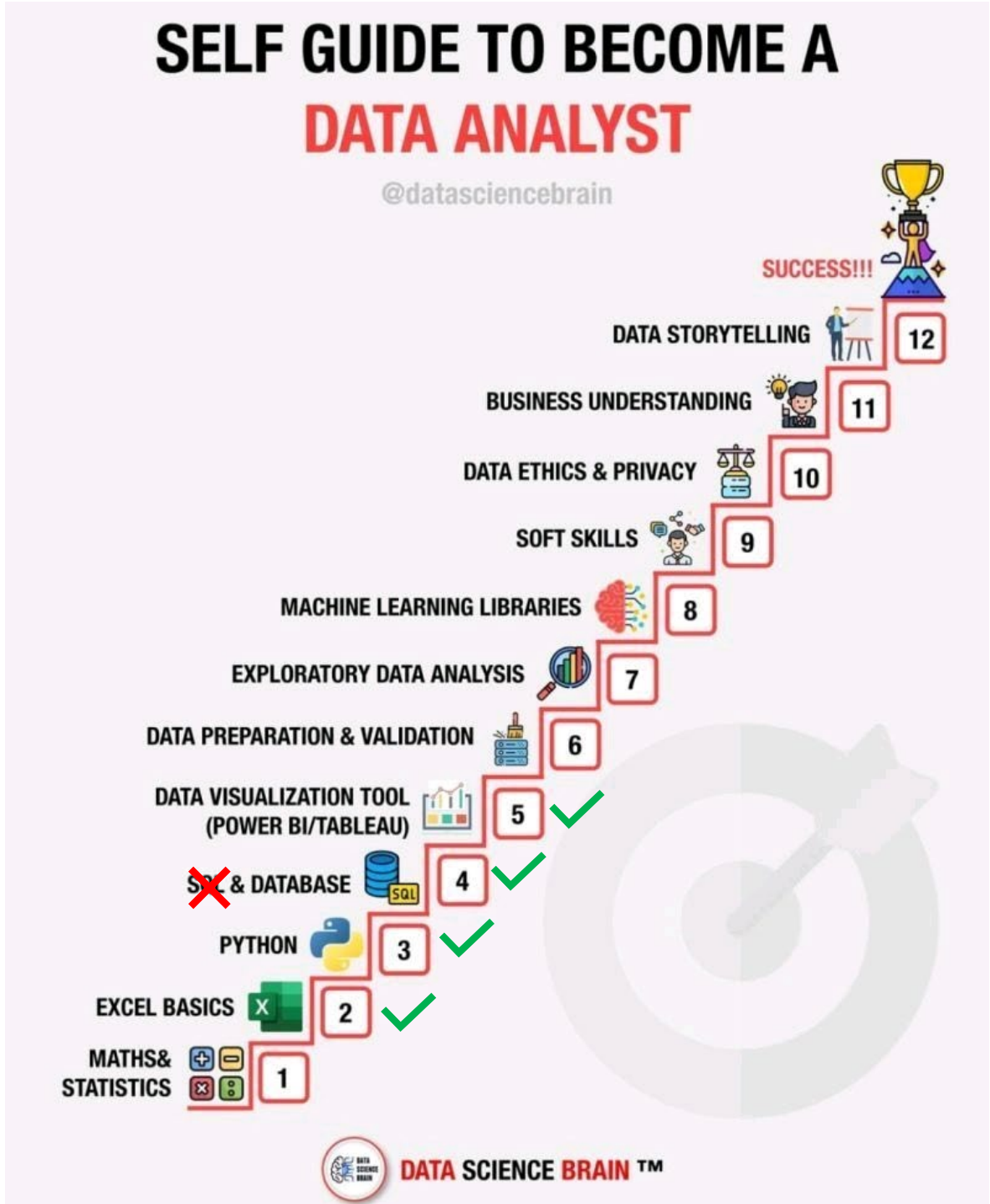



```
plt.figure(figsize=(11,3))
sns.countplot(x='Q1', data=df)
```



- Check out the color map
  - https://matplotlib.org/stable/tutorials/colors/colormaps.html

OUR SKILLSETS SO FAR!



DATA VISUALIZATION IS YOUR SUPER-POWER

- Use your power wisely!

1. You are a scientist. You need to do lots of literature review, data collection, and data cleaning.
2. Only then, you would get to perform data analysis, data visualization, and apply data science methods
3. Then, you need to spend quite some time to understand your plots!

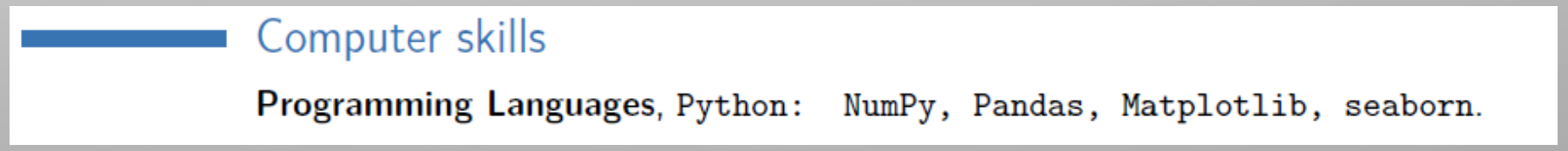
STRETCH YOUR CODING MUSCLES!

Part_IV_Data_Visualization_c_seaborn

THINGS TO DO

Upload your codes on your Github

Add your skill sets on your resume



BONUS

- Plot relative data in your other courses and get some bonus!