Data Preparation

Mehrdad Pirnia

Overview of the module

- This 4-week module equips you with all the tools and techniques to prepare your data to be used in ML models. The module starts with a quick background on Python.

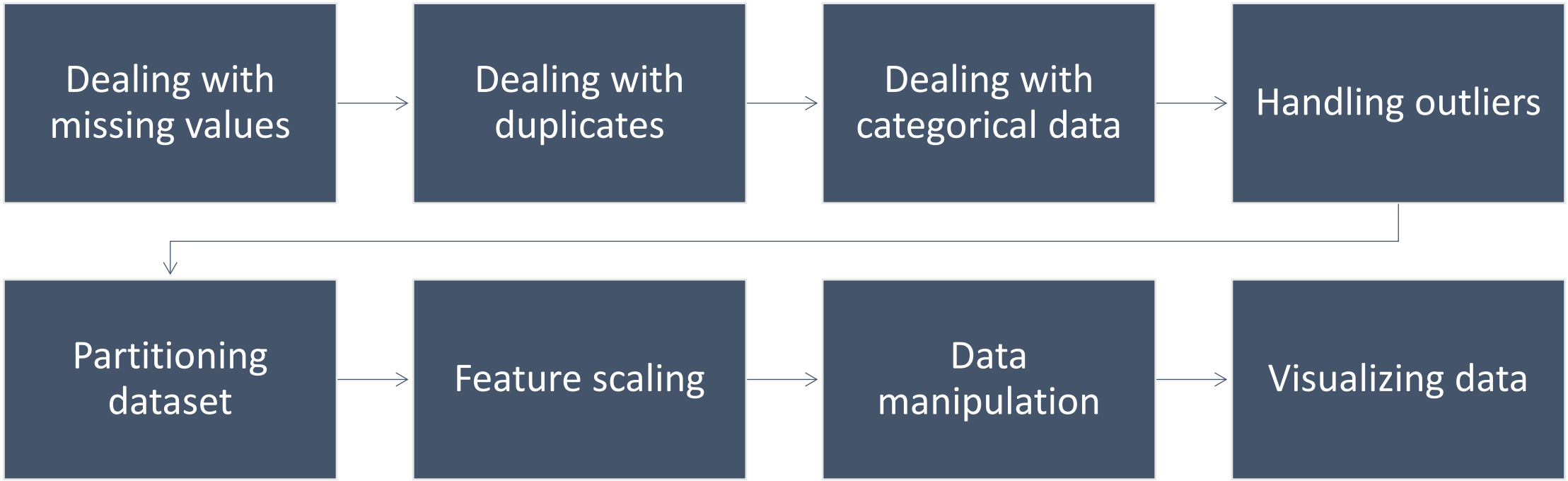
- By the end of the module you will learn how to:

1. Deal with missing and duplicated data (week1)
2. Handle categorical data (week2)
3. Partition dataset to train and test sets (week2)
4. Scale the feature parameters (week3)
5. Manipulate data (week3)
6. Visualize data using Python (Week4)

Steps for Data Preparation

- Data preparation is a very important portion of any ML project: loading, cleaning, transforming, and rearranging
- It should be considered as an initial step before building any models and training them
- A clean and prepared dataset helps with improvement of ML models. Otherwise, even the most advanced models wouldn’t be functional. This tasks sometimes take 80% of analysts’ time.

Steps for Data Preparation



2.2 Handling Missing Values

Missing Values

- In some cases, ML algorithms don't know how to process missing data and **return an error**
- Even if your algorithm can handle missing data, your model should get useful information to provide **better performance**
- Python uses a convention in R by presenting null values as NA (not available), which is referring to data that does not exist or exists but not observed, when collecting data.
- It is important to analyze missing data to find out whether it is related to data collection problems or potential biases in the data.

Handling Missing Values in Python

Python library: Pandas

- numeric data: **NaN** (Not a Number)
- non-numeric arrays: **None**

```
some_series = pd.Series([3.9, 7.1, np.nan, 81, np.nan, 0])
some_series
```

```
0     3.9
1     7.1
2     NaN
3    81.0
4     NaN
5     0.0
dtype: float64
```

```
some_string_data = pd.Series(['apples', None,'oranges', np.nan])
some_string_data
```

```
0     apples
1       None
2    oranges
3        NaN
dtype: object
```

Identifying Missing Values

Use '**isna()**' to identify the missing values represented by **NaN** or **None**:

```
some_series = pd.Series([3.9, 7.1, np.nan, 81, np.nan, 0])
```

```
some_string_data = pd.Series(['apples', None,'oranges', np.nan])
```

```
some_series.isna()
```

```
0    False
1    False
2     True
3    False
4     True
5    False
dtype: bool
```

```
some_string_data.isna()
```

```
0    False
1     True
2    False
3     True
dtype: bool
```

Filtering Out Missing Data

The '**dropna()**' function is helpful to drop all the null data.

```
from numpy import nan as NA

list_with_na = pd.Series([4, NA, NA, 7, NA,9])
list_drop_na = list_with_na.dropna()
list_drop_na
```

```
0    4.0
3    7.0
5    9.0
dtype: float64
```

Use '**notna()**' function to just show the list with no NaN value:

```
list_with_na[list_with_na.notna()]
```

```
0    4.0
3    7.0
5    9.0
dtype: float64
```

Removing Rows With NA

Use '**dropna()**' to drop all rows with a missing value in a DataFrame object containing NA:

```
data_with_na = pd.DataFrame([[2., 7.8, 3.9],[NA, NA, NA],
                             [NA, NA, 5.], [3., 8.6, 10.]])
data_cleaned = data_with_na.dropna()
data_with_na
```

	0	1	2
0	2.0	7.8	3.9
1	NaN	NaN	NaN
2	NaN	NaN	5.0
3	3.0	8.6	10.0

	0	1	2
0	2.0	7.8	3.9
3	3.0	8.6	10.0

Removing Rows With All NA

Pass **how='all'** to remove the rows that only contain NA and no other value:

```
data_cleaned_NArow = data_with_na.dropna(how='all')
data_cleaned_NArow
```

	0	1	2
0	2.0	7.8	3.9
1	NaN	NaN	NaN
2	NaN	NaN	5.0
3	3.0	8.6	10.0

	0	1	2
0	2.0	7.8	3.9
2	NaN	NaN	5.0
3	3.0	8.6	10.0

Removing Columns With NA

Pass **axis=1** to remove the columns containing NA:

```
# first add a column with NA

data_with_na [4] = NA
data_with_na
```

	0	1	2	4
0	2.0	7.8	3.9	NaN
1	NaN	NaN	NaN	NaN
2	NaN	NaN	5.0	NaN
3	3.0	8.6	10.0	NaN

```
data_with_na.dropna(axis=1, how = 'all')
```

	0	1	2
0	2.0	7.8	3.9
1	NaN	NaN	NaN
2	NaN	NaN	5.0
3	3.0	8.6	10.0

Filling in Missing Data: fillna()

Fill all the NAs with a constant value like zero:

```
data_with_zero = data_with_na.fillna(0)
data_with_zero
```

	0	1	2	4
0	2.0	7.8	3.9	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	5.0	0.0
3	3.0	8.6	10.0	0.0

Filling in Missing Data: fillna() + dict

Create a dictionary and fill NAs in each column with different values:

```
data_with_dict = data_with_na.fillna({0:0.7, 1:0.3, 2:0.5})
data_with_dict
```

	0	1	2
0	2.0	7.8	3.9
1	0.7	0.3	0.5
2	0.7	0.3	5.0
3	3.0	8.6	10.0

fillna() Function Arguments

Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation; by default `'ffill'` if function called with no other arguments
`axis`	Axis to fill on; default `axis=0`
`inplace`	Modify the calling object without producing a copy
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

Filling in Missing Data: fillna() + inplace

To reduce the memory usage, we can activate the **inplace=True** argument, so 'fillna()' would not return a new object and modify the object in place.

```
data_with_na
```

	0	1	2
0	2.0	7.8	3.9
1	NaN	NaN	NaN
2	NaN	NaN	5.0
3	3.0	8.6	10.0

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

```
data_with_na
```

	0	1	2
0	2.0	7.8	3.9
1	0.0	0.0	0.0
2	0.0	0.0	5.0
3	3.0	8.6	10.0

Filling in Missing Data: fillna() method

- Use **method=“ffill”** or **“bfill”** to propagate non-null values forward/backward
- Use **limit** to restrict the number of the Nan values, which could be changed

	0	1	2
0	2.0	7.8	3.9
1	NaN	NaN	NaN
2	NaN	NaN	5.0
3	3.0	8.6	10.0

```
data_with_na.fillna(method="bfill", limit = 2)
```

	0	1	2
0	2.0	7.8	3.9
1	3.0	8.6	5.0
2	3.0	8.6	5.0
3	3.0	8.6	10.0

```
data_with_na.fillna(method ="ffill", limit = 1)
```

	0	1	2
0	2.0	7.8	3.9
1	2.0	7.8	3.9
2	NaN	NaN	5.0
3	3.0	8.6	10.0

Filling in Missing Data: Imputation

We can also impute or replace missing values with **mean** or **median** statistics, using 'fillna()' function.

	0	1	2
0	2.0	7.8	3.9
1	NaN	NaN	NaN
2	NaN	NaN	5.0
3	3.0	8.6	10.0

```
data_with_na.fillna(data_with_na.mean())
```

	0	1	2
0	2.0	7.8	3.9
1	2.5	8.2	6.3
2	2.5	8.2	5.0
3	3.0	8.6	10.0

```
data_with_na.fillna(data_with_na.median())
```

	0	1	2
0	2.0	7.8	3.9
1	2.5	8.2	5.0
2	2.5	8.2	5.0
3	3.0	8.6	10.0

Practice

In the accompanied practice file, you will work with the diabetes dataset, and do the following tasks:

- Reading the dataset
- Review how to access rows and columns in a DataFrame
- Replace zero values with null
- Identify null values
- Remove the null values
- Replace null values with proper one
- Use dictionary to fill null values
- Use the `fillna` arguments, such as `limit`, `method`

2.3 Dealing with Duplicates

Identifying Duplicates

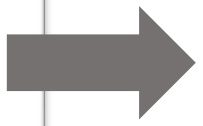
- There are many situations that we may observe duplicate rows in a dataset.
- In order to have higher quality dataset for training ML models, we should make sure the duplicates, would not cause your model to learn biases towards duplicated observations.
- Duplicate rows also add storage and processing overhead.
- Therefore, an important task of a data analyst is finding the duplicates and then removing them.

Identifying Duplicates: duplicated()

'**duplicated()**' returns a Boolean series to indicate whether a row is duplicated or not.

```
data_dup = pd.DataFrame({'Fruit':['apple','orange']*3 +['orange'],
                         'Quantity':[1,2,2,3,1,3,2]})
data_dup
```

	Fruit	Quantity
0	apple	1
1	orange	2
2	apple	2
3	orange	3
4	apple	1
5	orange	3
6	orange	2



```
data_dup.duplicated()
```

```
0    False
1    False
2    False
3    False
4     True
5     True
6     True
dtype: bool
```

Dropping Duplicates: drop_duplicates()

Use '**drop_duplicates()**' to drop the duplicated arrays, which returns a DataFrame with only false duplicated rows:

	Fruit	Quantity
0	apple	1
1	orange	2
2	apple	2
3	orange	3
4	apple	1
5	orange	3
6	orange	2

```
data_drop_dup = data_dup.drop_duplicates()
data_drop_dup
```

	Fruit	Quantity
0	apple	1
1	orange	2
2	apple	2
3	orange	3

Dropping Duplicates (One Column)

'drop_duplicates()' drops rows based on all the columns.

We can also drop duplicates based on a specific column:

	Fruit	Quantity
0	apple	1
1	orange	2
2	apple	2
3	orange	3
4	apple	1
5	orange	3
6	orange	2

```
data_drop_dupCol = data_dup.drop_duplicates(['Quantity'])
```

	Fruit	Quantity
0	apple	1
1	orange	2
3	orange	3

Dropping Duplicates: keep='last'

'duplicated()' and 'drop_duplicates()' keep the first observation and remove the rest. To keep the last observation, we pass **keep='last'**:

	Fruit	Quantity	Quality
0	apple	1	0
1	orange	2	1
2	apple	2	2
3	orange	3	3
4	apple	1	4
5	orange	3	5
6	orange	2	6

```
data_dup.drop_duplicates(['Fruit', 'Quantity'], keep ='last')
```

	Fruit	Quantity	Quality
2	apple	2	2
4	apple	1	4
5	orange	3	5
6	orange	2	6

Count Duplicate/Non-Duplicate Rows

Count 'True' in the obtained series from 'duplicated()', using '**sum()**' or '**value_counts()**':

	Fruit	Quantity
0	apple	1
1	orange	2
2	apple	2
3	orange	3
4	apple	1
5	orange	3
6	orange	2

```
data_dup.duplicated().sum()
```

```
3
```

```
data_dup.duplicated().value_counts()
```

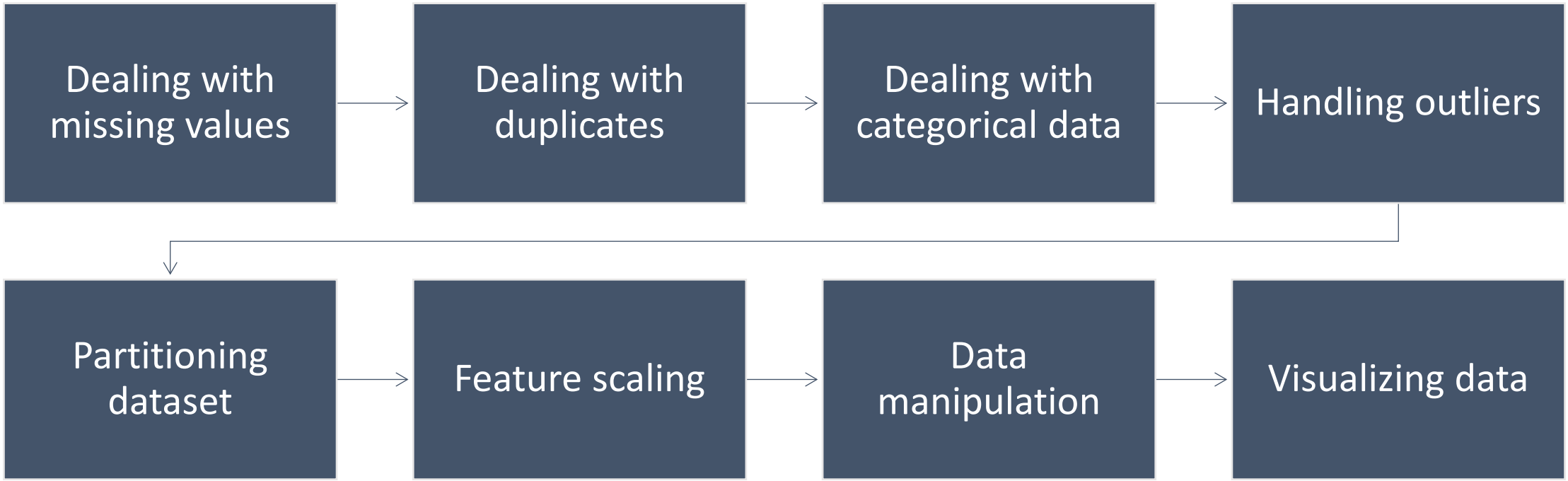
```
False    4
True     3
dtype: int64
```

Practice

In the accompanied notebook for week 2, you will use the diabetic dataset to:

- identify the duplicated values,
- remove duplicate rows
- use `keep` argument with `drop_duplicates` function
- count the duplicate values
- find the duplicate columns
- remove the duplicate columns

Steps for Data Preparation



2.4 Handling Categorical Data

Categorical Data

So far, we've been dealing with numerical values. However, in realistic datasets it is not uncommon to see categorical features.

Categorical data can be divided in into ordinal and nominal features.

Ordinal features as the name suggests can be sorted or put in order: severity of a disease, feeling spectrum

Nominal features do not have any orders: type of Covid vaccine type, colour of eyes

Handling Categorical Data

A column in a DataFrame may contain repeated instances of some distinct values, which can be classified as **categorical** data.

```
diseases = (['asthma', 'diabetes','covid','covid','diabetes','covid']*2)
diseases
```

```
['asthma',
 'diabetes',
 'covid',
 'covid',
 'diabetes',
 'covid',
 'asthma',
 'diabetes',
 'covid',
 'covid',
 'diabetes',
 'covid']
```

```
pd.unique(diseases)
```

```
array(['asthma', 'diabetes', 'covid'], dtype=object)
```

```
pd.value_counts(diseases)
```

```
covid       6
diabetes    4
asthma      2
dtype: int64
```

Dimension Table

We use a *dimension table* to enhance the storage requirement of categorical data, which contains the distinct or categorical values and stores them with integer key referencing.

```
values = pd.Series([0,1,2,2,1,2]*2)

dim = pd.Series(pd.unique(diseases))
dim
```

```
0      asthma
1    diabetes
2       covid
dtype: object
```

```
dim.take(values)
```

```
0      asthma
1    diabetes
2       covid
2       covid
1    diabetes
2       covid
0      asthma
1    diabetes
2       covid
2       covid
1    diabetes
2       covid
dtype: object
```

Converting Type to Categorical Extension: astype()

```
diseases = ['ADHD','Autism','COVID','COVID','Autism']*2
N = len(diseases)
rng = np.random.default_rng(seed = 12345)

df = pd.DataFrame({'disease':diseases,
                   'patient_id':np.arange(N),
                   'severity':rng.integers(1,10, size = N),
                   'weight':rng.uniform(50,150, size = N)},
                  columns = ['patient_id','disease','severity','weight'])
```

	patient_id	disease	severity	weight
0	0	ADHD	7	83.281393
1	1	Autism	3	109.830875
2	2	COVID	8	68.673419
3	3	COVID	3	117.275604
4	4	Autism	2	144.180287
5	5	ADHD	8	74.824571
6	6	Autism	6	144.888115
7	7	COVID	7	116.723745
8	8	COVID	9	59.589794
9	9	Autism	4	94.183967

```
disease_cat = df['disease'].astype('category')
disease_cat
```

```
0      ADHD
1    Autism
2     COVID
3     COVID
4    Autism
5      ADHD
6    Autism
7     COVID
8     COVID
9    Autism
Name: disease, dtype: category
Categories (3, object): ['ADHD', 'Autism', 'COVID']
```

Two Attributes of Categorical Object

'**categories**' and '**codes**'

```
c = disease_cat.array
type(c)
```

```
pandas.core.arrays.categorical.Categorical
```

```
c.categories
```

```
Index(['ADHD', 'Autism', 'COVID'],
```

```
c.codes
```

```
array([0, 1, 2, 2, 1, 0, 1, 2, 2, 1], dtype=int8)
```

```
dict(enumerate(c.categories))
```

```
{0: 'ADHD', 1: 'Autism', 2: 'COVID'}
```

Categorical Methods

The *accessor* attribute '**cat**' facilitates the usage of categorical methods, such as 'codes' and 'set_categories'.

```
s = pd.Series(['ADHD','Autism','COVID']*2)
s_cat = s.astype('category')
s_cat
```

```
0      ADHD
1    Autism
2     COVID
3      ADHD
4    Autism
5     COVID
dtype: category
Categories (3, object): ['ADHD', 'Autism', 'COVID']
```

```
s_cat.cat.codes
```

```
0    0
1    1
2    2
3    0
4    1
5    2
dtype: int8
```

```
s_cat.cat.categories
```

```
Index(['ADHD', 'Autism', 'COVID'], dtype='object')
```

Categorical Methods (cont.)

- value_counts()
- remove_unused_categories()

```
actual_categories = ['ADHD', 'Autism', 'COVID','flu','diabetes']

s_cat2 = s_cat.cat.set_categories(actual_categories)

s_cat2
```

```
0      ADHD
1    Autism
2     COVID
3      ADHD
4    Autism
5     COVID
dtype: category
Categories (5, object): ['ADHD', 'Autism', 'COVID', 'flu', 'diabetes']
```

```
s_cat.value_counts()
```

```
ADHD      2
Autism    2
COVID     2
dtype: int64
```

```
s_cat2.value_counts()
```

```
ADHD        2
Autism      2
COVID       2
flu         0
diabetes    0
dtype: int64
```

```
s_cat2.cat.remove_unused_categories()
```

```
0      ADHD
1    Autism
2     COVID
3      ADHD
4    Autism
5     COVID
dtype: category
Categories (3, object): ['ADHD', 'Autism', 'COVID']
```

Manual Mapping of Ordinal Features

Unfortunately, there is no automatic way to do such mapping, and it needs to be defined manually.

```
df = pd.DataFrame([
    ['Indian','sever','4500' ],
    ['Asian','medium','8200'],
    ['Hispanic','low','5100']
    ])
df.columns = ['race','impact','population']
```

	race	impact	population
0	Indian	sever	4500
1	Asian	medium	8200
2	Hispanic	low	5100

```
impact_map = {'sever':3, 'medium':2, 'low':1}

df['impact'] = df['impact'].map(impact_map)
```

	race	impact	population
0	Indian	3	4500
1	Asian	2	8200
2	Hispanic	1	5100

Inverse Mapping

To map the integer values to actual categorical values, we can use:

```
inv_impact_map = {v:k for k,v in impact_map.items()}
```

```
{3: 'sever', 2: 'medium', 1: 'low'}
```

```
df['impact']=df['impact'].map(inv_impact_map)
```

	race	impact	population
0	Indian	sever	4500
1	Asian	medium	8200
2	Hispanic	low	5100

Encoding Class Labels

We use the mapping method to encode class labels, and simply enumerate the labels starting from 0.

```
label_map = {label: code for code, label in
             enumerate(np.unique(df['classlabel']))}
```

```
{'class1': 0, 'class2': 1}
```

	race	impact	population	classlabel
0	Indian	sever	4500	1
1	Asian	medium	8200	0
2	Hispanic	low	5100	1

```
df['classlabel']=df['classlabel'].map(label_map)
```

	race	impact	population	classlabel
0	Indian	sever	4500	1
1	Asian	medium	8200	0
2	Hispanic	low	5100	1

Reversing Class Labels

We can reverse the mapping dictionary to create the original dataset with the actual class labels.

```
inv_class_map = {v:k for k, v in label_map.items()}
```

```
{0: 'class1', 1: 'class2'}
```

```
df['classlabel'] = df['classlabel'].map(inv_class_map)
```

	race	impact	population	classlabel
0	Indian	sever	4500	class2
1	Asian	medium	8200	class1
2	Hispanic	low	5100	class2

LabelEncoder()

This '**sklearn**' class provides an easy way to encode the categorical data.

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['classlabel'] = le.fit_transform(df['classlabel'].values)
```

	race	impact	population	classlabel
0	Indian	sever	4500	1
1	Asian	medium	8200	0
2	Hispanic	low	5100	1

```
le.inverse_transform(df['classlabel'])
```

```
array(['class2', 'class1', 'class2'], dtype=object)
```

One-Hot Encoding With Nominal Features

It creates a new dummy feature for each unique value in the nominal feature column.

```
X = df[['race','impact','population']].values
X
```

```
array([['Indian', 'sever', '4500'],
       ['Asian', 'medium', '8200'],
       ['Hispanic', 'low', '5100']], dtype=object)
```

```
race_le = LabelEncoder()
X[:,0] = race_le.fit_transform(X[:,0])
X
```

```
array([[2, 'sever', '4500'],
       [0, 'medium', '8200'],
       [1, 'low', '5100']], dtype=object)
```

```
from sklearn.preprocessing import OneHotEncoder

X = df[['race', 'impact', 'population']].values
race_ohe = OneHotEncoder()
race_ohe.fit_transform(X[:,0].reshape(-1,1)).toarray()
```

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

One-Hot Encoding for Multi-Columns

We use '**ColumnTransformer**' to select multi-columns, which is accepting a list of (**name**, **transformer**, **column(s)**) tuples.

```
from sklearn.compose import ColumnTransformer

X = df[['race', 'impact', 'population']].values

col_transf = ColumnTransformer([('onehot',OneHotEncoder(),[0]),
                                ('nothing','passthrough',[1,2])])

col_transf.fit_transform(X)
```

```
array([[0.0, 0.0, 1.0, 'sever', '4500'],
       [1.0, 0.0, 0.0, 'medium', '8200'],
       [0.0, 1.0, 0.0, 'low', '5100']], dtype=object)
```

Pandas Data Types

- The Python Pandas package was originally designed to work with numerical data, based on what was available in NumPy.
- That created some issues with handlining strings and other types of data.
- To overcome these deficiencies, Pandas introduced an *extension type* to consider data types not supported natively by NumPy.

Extension Type	Description
`BooleanDtype`	Nullable boolean data, use `"boolean"` when passing as string
`CategoricalDtype`	Categorical data type, use `"category"` when passing as string
`DatetimeTZDtype`	Datetime with time zone
`Float32Dtype`	32-bit nullable floating point, use `"Float32"` when passing as string
`Float64Dtype`	64-bit nullable floating point, use `"Float64"` when passing as string
`Int8Dtype`	8-bit nullable signed integer, use `"Int8"` when passing as string
`Int16Dtype`	16-bit nullable signed integer, use `"Int16"` when passing as string
`Int32Dtype`	32-bit nullable signed integer, use `"Int32"` when passing as string
`Int64Dtype`	64-bit nullable signed integer, use `"Int64"` when passing as string
`UInt8Dtype`	8-bit nullable unsigned integer, use `"UInt8"` when passing as string
`UInt16Dtype`	16-bit nullable unsigned integer, use `"UInt16"` when passing as string
`UInt32Dtype`	32-bit nullable unsigned integer, use `"UInt32"` when passing as string
`UInt64Dtype`	64-bit nullable unsigned integer, use `"UInt64"` when passing as string

Example of Data Types

```
Series = pd.Series([1,2, None,3])
```

```
0    1.0
1    2.0
2    NaN
3    3.0
dtype: float64
```

```
Series.dtype
```

```
dtype('float64')
```

```
Series = pd.Series([1,2,None,3], dtype=pd.Int64Dtype())
```

```
0       1
1       2
2    <NA>
3       3
dtype: Int64
```

```
Series.dtype
```

```
Int64Dtype()
```

```
Series = pd.Series(['one','two', None, 'three']
                   , dtype=pd.StringDtype())
```

```
0       one
1       two
2      <NA>
3     three
dtype: string
```

```
Series = pd.Series([1,2,None,3], dtype="Int64")
```

```
0       1
1       2
2    <NA>
3       3
dtype: Int64
```

Passing Extension Types

Use '**astype()**' method to pass extension types to the Series, allowing conversion from one data type to another:

```
df["BloodType"].isna
```

```
<bound method Series.isna of 0
1        B
2     None
3       AB
4        O
Name: BloodType, dtype: object>
```

```
df["BloodType"] = df["BloodType"].astype("string")
df["Patient"] = df["Patient"].astype("Int64")
df["Donor"] = df["Donor"].astype("boolean")
```

	BloodType	Patient	Donor
0	A	1.0	False
1	B	2.0	None
2	None	3.0	False
3	AB	NaN	True
4	O	4.0	True

	BloodType	Patient	Donor
0	A	1	False
1	B	2	<NA>
2	<NA>	3	False
3	AB	<NA>	True
4	O	4	True

Indicator Matrix: Dummy Variables

If a feature or column in a DataFrame has 'k' distinct values, we can create a matrix that contains 0s and 1s with 'k' columns, using '**pandas.get_dummies()**' function.

```
df = pd.DataFrame({"disease":["diabetes", "lupus","cancer", "Hepatitis",
                              "cancer", "lupus", "lupus","diabetes"],
                   "patient":range(8)})
```

	disease	patient
0	diabetes	0
1	lupus	1
2	cancer	2
3	Hepatitis	3
4	cancer	4
5	lupus	5
6	lupus	6
7	diabetes	7

```
pd.get_dummies(df["disease"])
```

	Hepatitis	cancer	diabetes	lupus
0	0	0	1	0
1	0	0	0	1
2	0	1	0	0
3	1	0	0	0
4	0	1	0	0
5	0	0	0	1
6	0	0	0	1
7	0	0	1	0

Prefix for get_dummies()

Add a prefix to the columns indicating the groups and join them with other columns to have a better representation of data:

```
indicator_matrix = pd.get_dummies(df["disease"],prefix = "disease")
```

	disease_Hepatitis	disease_cancer	disease_diabetes	disease_lupus
0	0	0	1	0
1	0	0	0	1
2	0	1	0	0
3	1	0	0	0
4	0	1	0	0
5	0	0	0	1
6	0	0	0	1
7	0	0	1	0

```
df_with_dummies = df[["patient"]].join(indicator_matrix)
```

	patient	disease_Hepatitis	disease_cancer	disease_diabetes	disease_lupus
0	0	0	0	1	0
1	1	0	0	0	1
2	2	0	1	0	0
3	3	1	0	0	0
4	4	0	1	0	0
5	5	0	0	0	1
6	6	0	0	0	1
7	7	0	0	1	0

Indicator Matrix With Cut

Use '**get_dummies()**' method with the '**cut()**' function to illustrate the assignments of data to different groups:

```
rnd = np.random.uniform(size = 10)
```

```
array([0.6274922 , 0.18854226, 0.5155142 , 0.86157069, 0.38889322,
       0.20482243, 0.95638882, 0.28438546, 0.87899835, 0.22164822])
```

```
buckets = [0,0.2,0.4,0.6,0.8,1]
pd.get_dummies(pd.cut(rnd,buckets))
```

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]
0	0	0	0	1	0
1	1	0	0	0	0
2	0	0	1	0	0
3	0	0	0	0	1
4	0	1	0	0	0
5	0	1	0	0	0
6	0	0	0	0	1
7	0	1	0	0	0
8	0	0	0	0	1
9	0	1	0	0	0

String Handling

Python is a very popular programming language to deal with text and strings, because of the string objects built-in methods.

Panda advances the built-in Python methods, by adding flexibility to the text-based arrays.

In this section we review some of the built-in methods for handling strings.

String Methods: split(), strip()

- Use '**split()**' to separate a string by some indicators (e.g., comma-separated string can be separated into pieces)

```
name = "patient1,   patient2,patient3,  patient4"
name.split(",")
```

```
['patient1', '   patient2', 'patient3', '  patient4']
```

- Use '**strip()**' to remove the whitespace. To use strip and split functions together, we use the following code

```
patients = [x.strip() for x in name.split(",")]
```

```
['patient1', 'patient2', 'patient3', 'patient4']
```

Useful Methods for String Manipulation

- Use '**in**' to find out whether a string is part of a string object

```
patients
```

```
['patient1', 'patient2', 'patient3', 'patient4']
```

```
"patient7" in patients
```

```
False
```

- Use '**count()**' to count the number of the occurrence of a sub-string is done

```
patients.count("patient4")
```

```
1
```

- Use '**replace()**' to replace some strings for others

```
name = "Mark, Hanna, Dave, Sara"
name.replace(",", "")
```

```
'Mark Hanna Dave Sara'
```

Practice

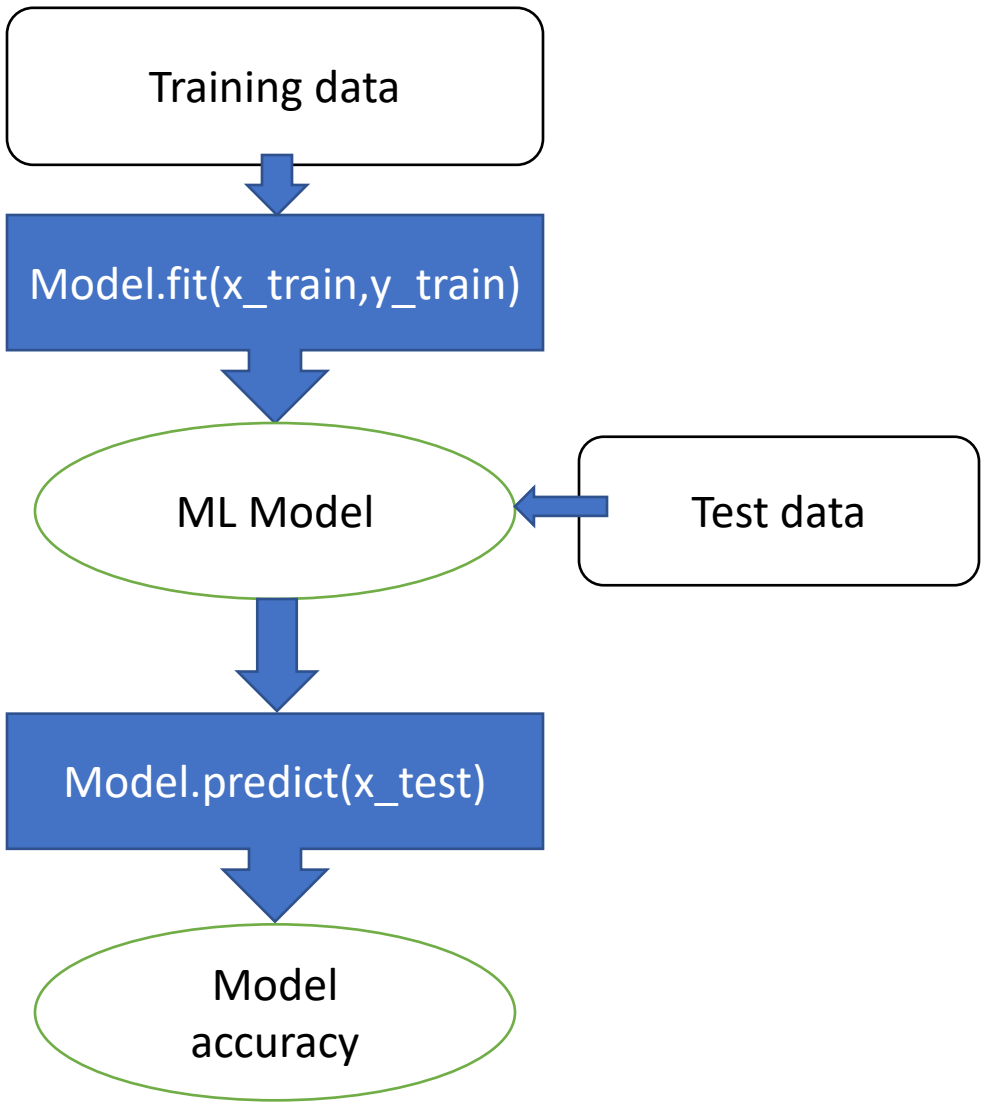
In the accompanied notebook for week 3, you will use the healthcare analytics dataset to:

- Use LabelEncoder to encode hospital_type_code from letters to numbers
- Use OneHotEncoder to encode hospital_type_code
- Apply get_dummies for encoding hospital_type_code
- Use dictionary to perform ordinal mapping on severity_of_illness column

2.5 Partitioning a Dataset

Partitioning a Dataset

- To train ML models we use the portion of the dataset, called train set.
- Another part of the dataset, test set, is put aside to test the performance of the model before releasing the models for real applications.



Appropriate Train/Test Ratio

- When we are dividing the dataset into train/test portions we might be removing some important information that could help with training our model. And therefore, we may not want to assign too much wight to test set.

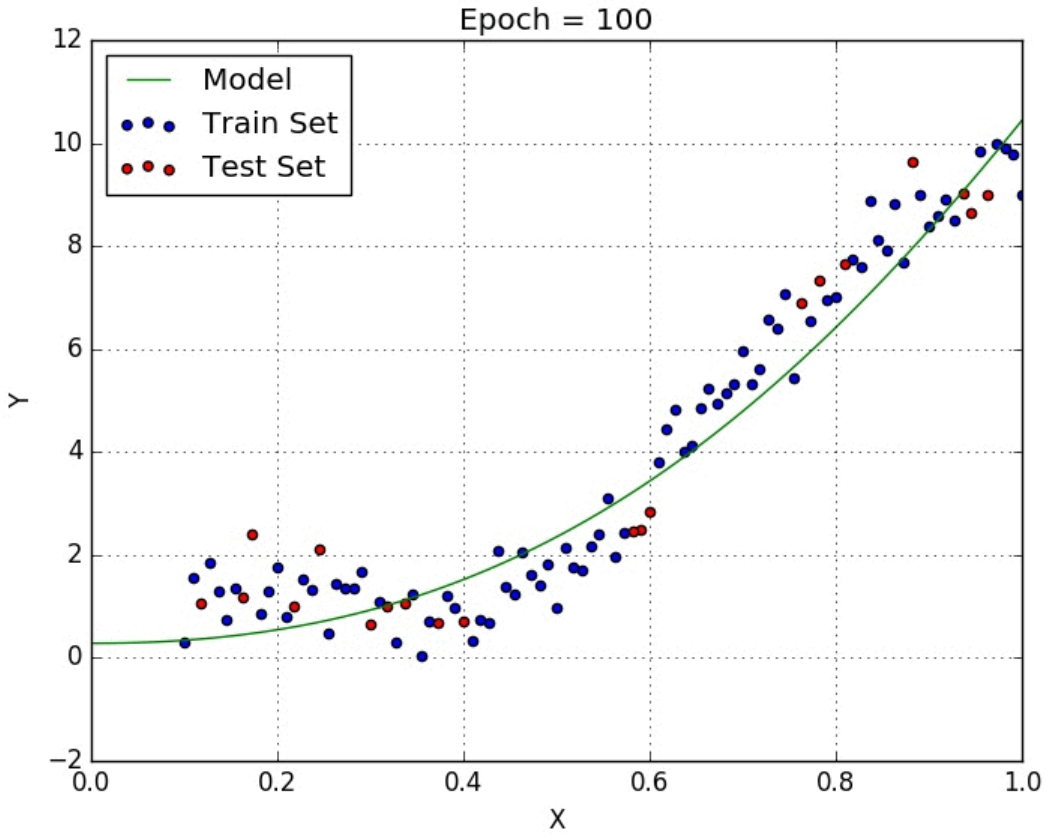
- However, the smaller test sets may yield inaccurate estimation of the accuracy of the model generalization capability.

- The proper ratio helps balancing this trade-off

Shortcomings of Poor Split Data

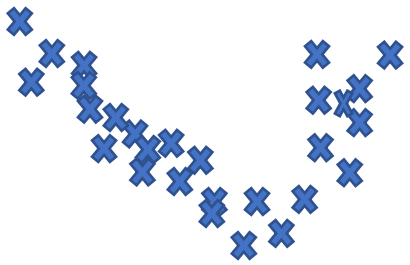
Two major issues are associated to poor split of datasets:

- **Underfitting** yields high *training* error
- **Overfitting** yields high *test* error

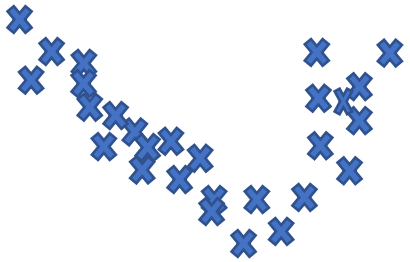


Source: Frossard, D. (2016, May 29). *Multiple linear regression: Extending linear regression to use multiple descriptive variables*. https://www.cs.toronto.edu/~frossard/post/multiple_linear_regression/

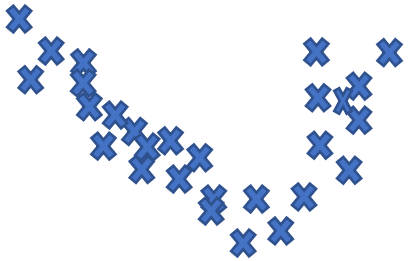
Examples of Overfitting and Underfitting



underfit



balance



overfit

Bias and Variance

- Bias is referred to the rigidity of the ML model, and its incapability for capturing the complex relationships among datapoints . For example applying a linear model for nonlinear data ~ high bias and low variance

- Variance is referred to the flexibility of the model and its efforts to capture precisely the location of each datapoint during training, resulting in too flexible and complicated model ~ high variance and low bias.

- During model fitting we use methods to trade off bias and variance in our training. First approach is using a proper split between test and training sets!

Common Practices for Train/Test Ratio

- In practice, it is common to have splits of train:test sets as. 80:20, 70-30, and even 60-40 depending on the size of the dataset.

- The smaller datasets, the more balanced the train/test splits are.

- For large datasets, more than 100,000 data points, it is common to see 90:10 or even 99:1 splits. As 10,000 datapoints could still provide good information regarding the accuracy of the model.

Data Split With Python

- Use '**train_test_split**' function from scikit-learn model selection submodule to randomly partition a dataset into test and train sets:

```
from sklearn.model_selection import train_test_split
```

- The parameters of the function are:

```
sklearn.model_selection.train_test_split(
    *arrays,
    test_size=None,
    train_size=None,
    random_state=None,
    shuffle=True,
    stratify=None
)
```

Data Splits in Diabetes Dataset

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148.0	72.0	35.00000	155.548223	33.6	0.627	50	1
1	1	85.0	66.0	29.00000	155.548223	26.6	0.351	31	0
2	8	183.0	64.0	29.15342	155.548223	23.3	0.672	32	1

```
train,test = train_test_split(diabData,test_size=0.25,random_state=0,stratify=diabData['Outcome'])
train_X = train[train.columns[:8]]
train_Y = train['Outcome']
test_X = test[test.columns[:8]]
test_Y = test['Outcome']
```

Practice

In the accompanied note book for week 3, you will use the wine dataset to:

- partition the dataset, using (80:20) and (70:30) splits and observe the differences
- turn the shuffling argument on/off and observe the changes
- find the error of the linear regression model when split is changed to (90:10)

2.6 Feature Scaling

What is feature scaling?

- Feature scaling is one of the most important steps of pre-processing, which is often ignored
- Before feeding data into our training models, we have to make sure the features value are within the same scale so those significant larger number would impact the model because of its magnitude.
- For instance, if data has two features one from 0 to 1000 and from 0 to 10, the ML model spends more time to resolve the issues related to the feature with larger magnitude
- Most of the ML algorithms provide higher accuracy if the features are on the same scale.
- The two common feature scaling methods are normalization and standardization.

Normalization – MinMax Scalar

- It usually rescale the features to a given range, for example [0,1]. The scaler can also shrink data within the range of [-1 , 1].

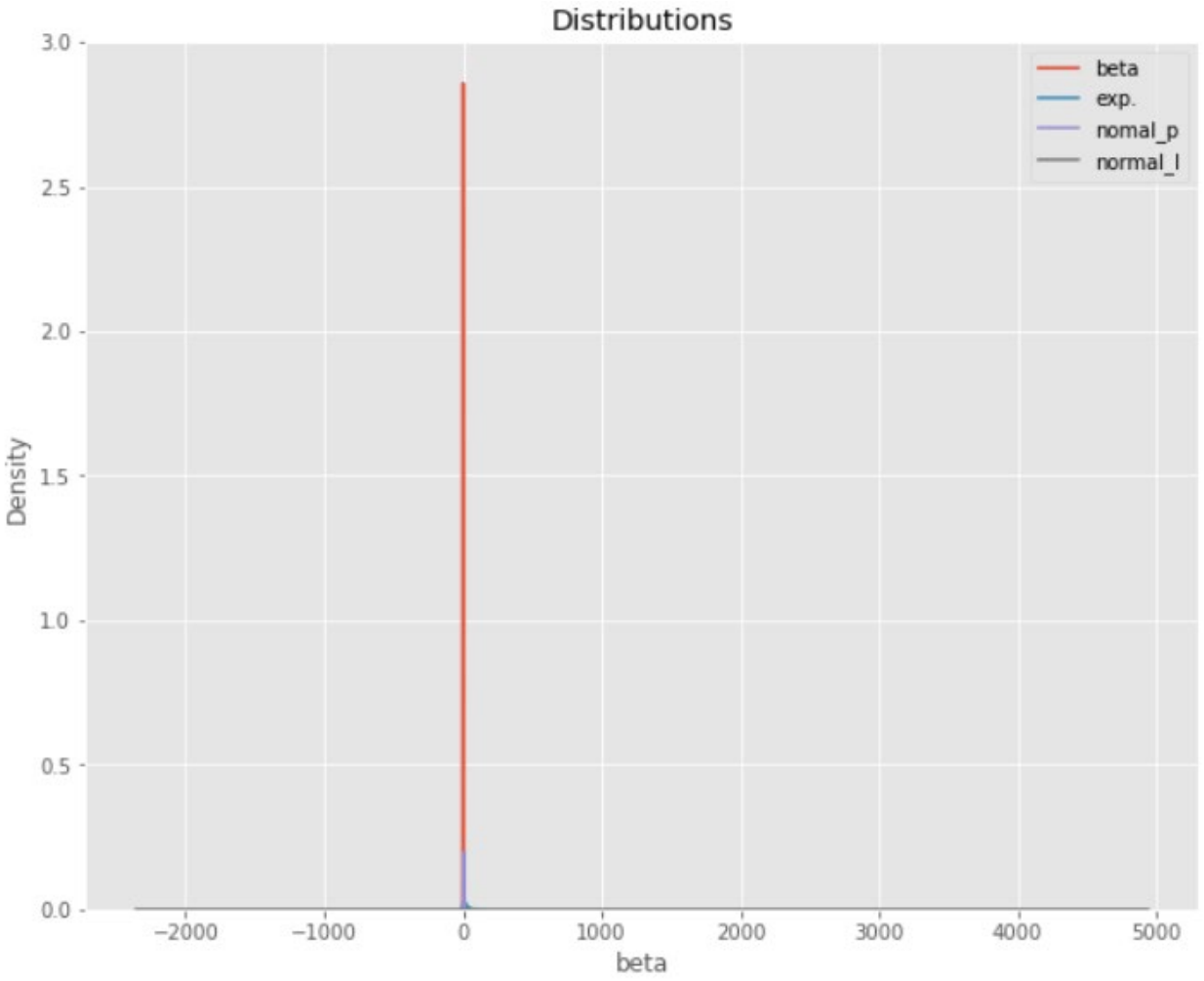
- The methods works well if the variance of data is small and the distribution is not Gaussian, and it's very sensitive to outliers.

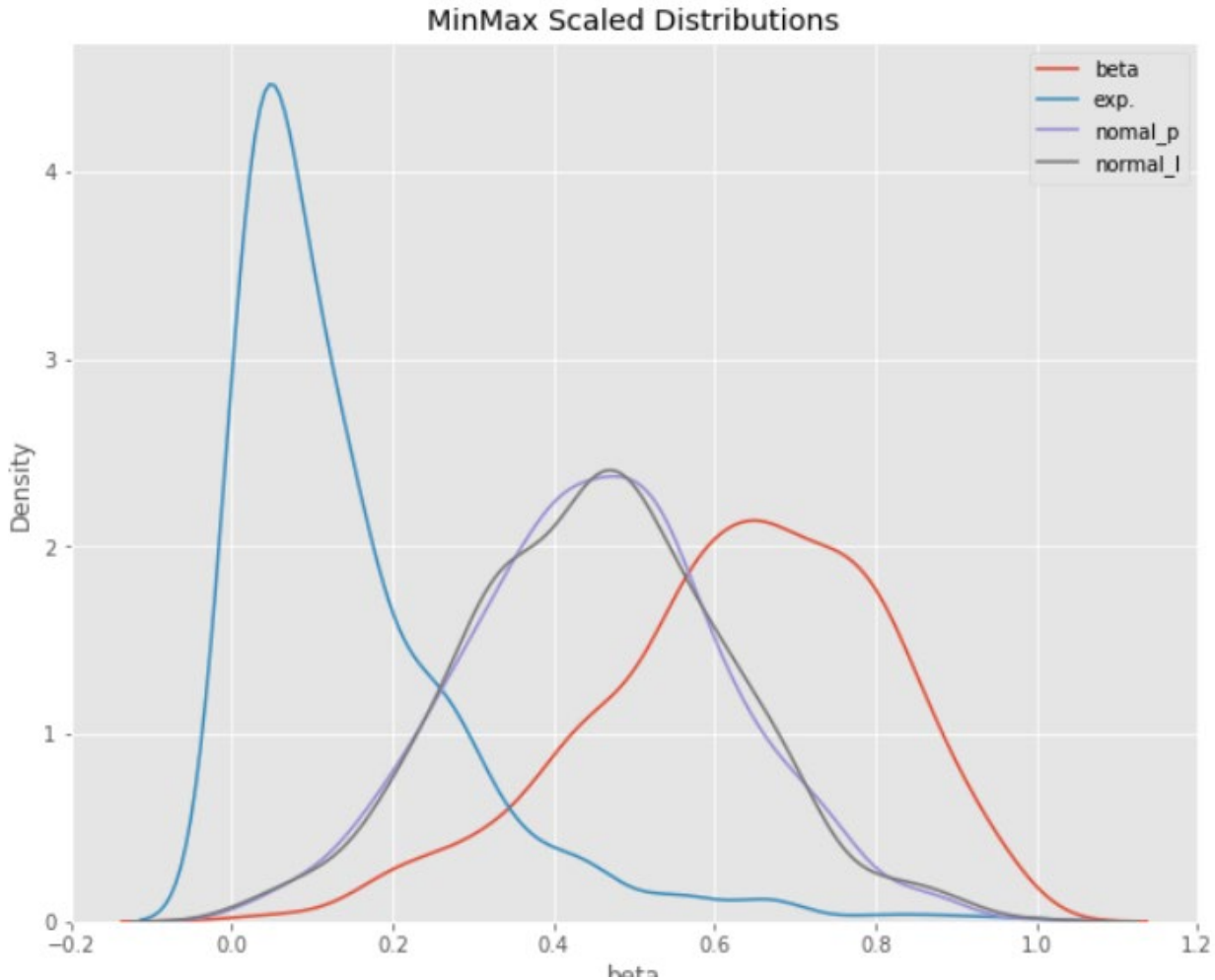
- In order to scale, x_i in a feature set with x_{min} (the min value of the feature) and x_{max} (the max value of the feature):

$$x_{norm} = (x_i - x_{min})/(x_{max} - x_{min})$$

Normalization in Python

```
from sklearn import preprocessing

min_max = preprocessing.MinMaxScaler()
df_mod = min_max.fit_transform(df)
```





Data Transformation With Min-Max Scaling

	normal_p	beta	exponential	normal_l
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	4.907546	0.699554	20.993917	10.246469
std	1.993466	0.136096	21.741122	9.727528
min	-0.552064	0.229112	0.054666	-16.254445
25%	3.562770	0.617608	5.983017	3.380680
50%	4.916578	0.710784	13.792226	10.359616
75%	6.141336	0.802235	28.918100	16.720447
max	11.519088	0.977630	144.841692	42.054036

Min-Max Scaling

	normal_p	beta	exponential	normal_l
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.452286	0.628498	0.144621	0.454495
std	0.165143	0.181820	0.150159	0.166829
min	0.000000	0.000000	0.000000	0.000000
25%	0.340882	0.519021	0.040945	0.336746
50%	0.453034	0.643501	0.094881	0.456436
75%	0.554496	0.765677	0.199351	0.565525
max	1.000000	1.000000	1.000000	1.000000

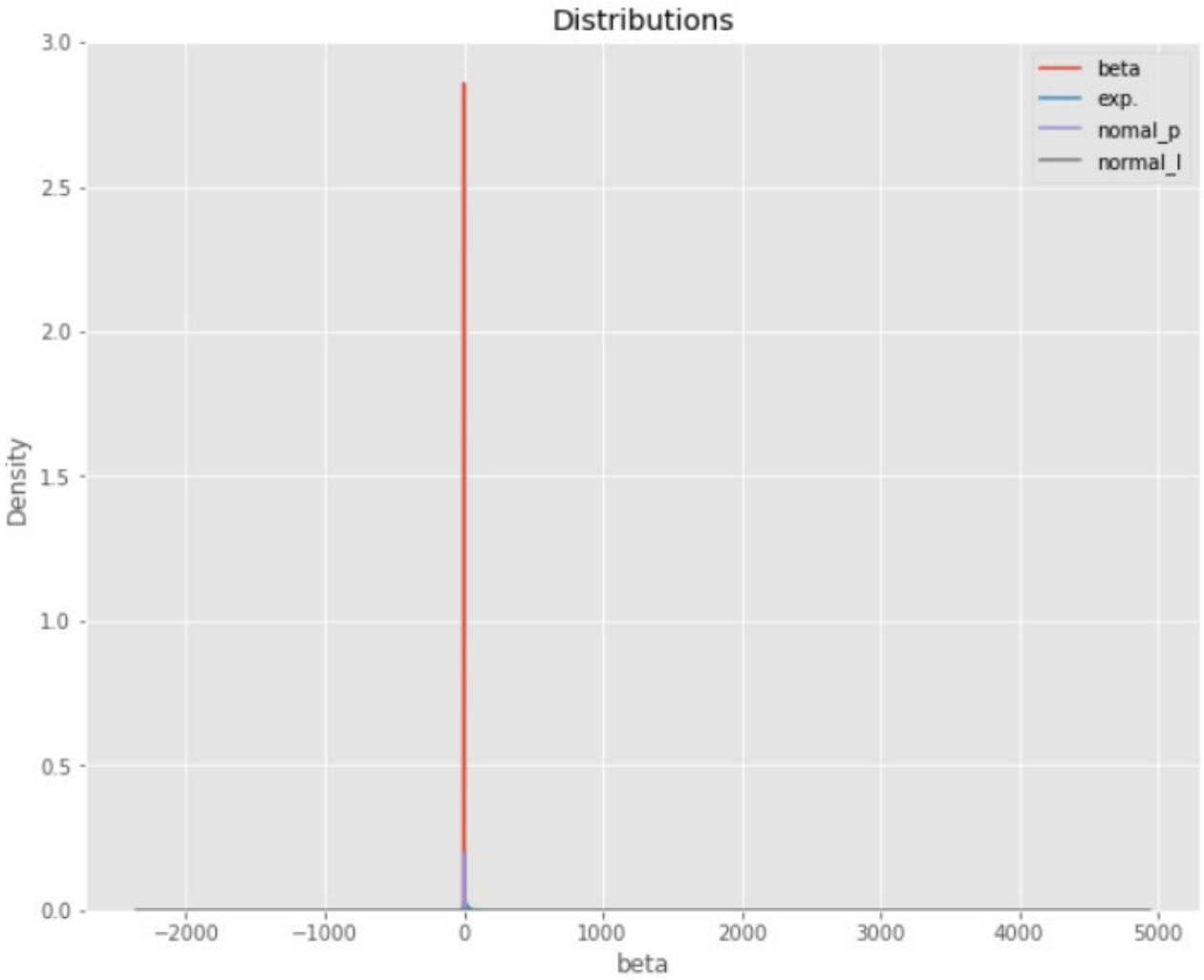
Standardization

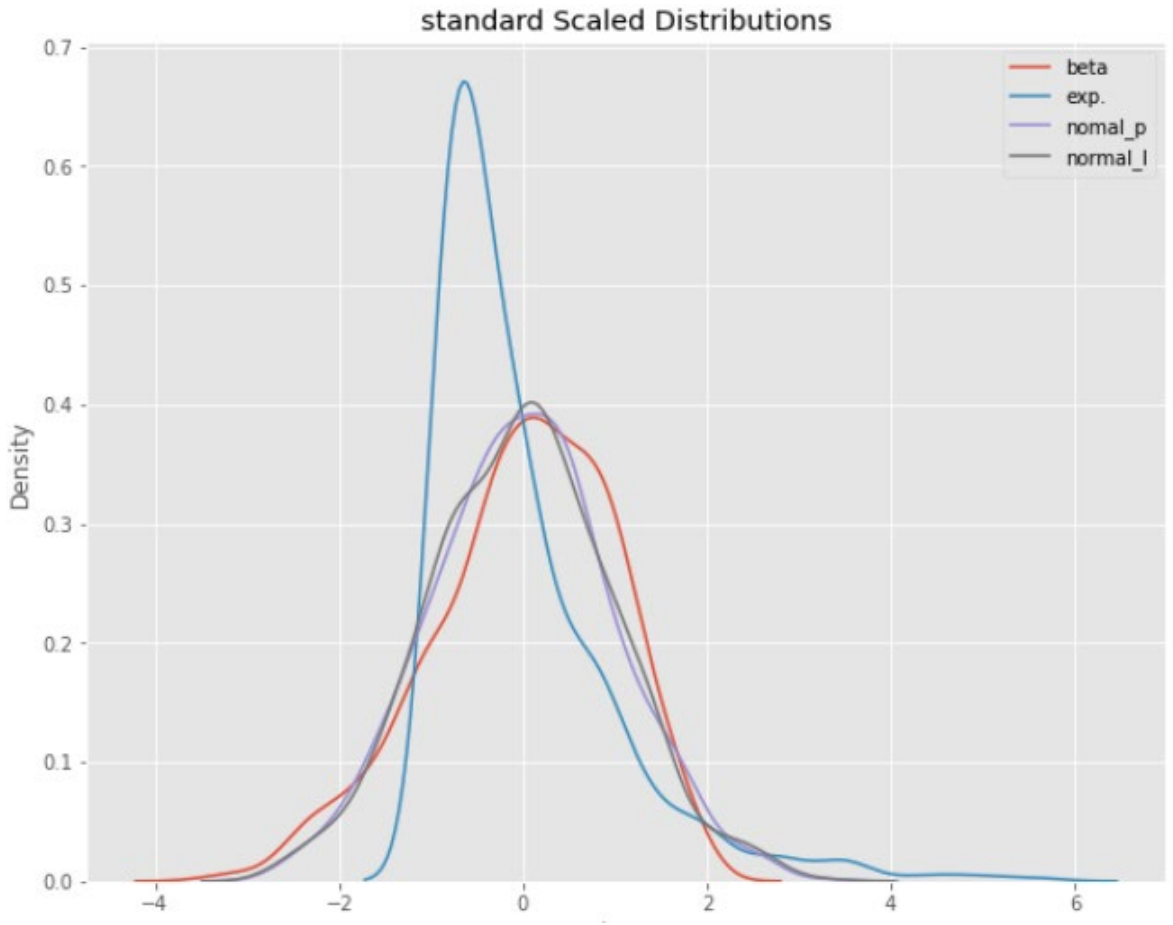
- Transform the feature values to have Zero mean and a variance of 1, so the column has the same parameters as a standard normal distribution.
- It is a practical method for many ML models, specially for optimization algorithms like gradient descent.
- It doesn't change the shape of distributions from non-normal to normal.
- It is less sensitive towards outliers and it maintains their information.
- The following is used for this scaling: μ_x is the sample mean, and σ_x is the standard devisation

$$x_i^{std} = (x_i - \mu_x)/\sigma_x$$

Normalization in Python

```
from sklearn import preprocessing

std_scaler = preprocessing.StandardScaler()
df_mod2 = std_scaler.fit_transform(df)
```





Data Transformation With Standardization

	normal_p	beta	exponential	normal_l
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	4.907546	0.699554	20.993917	10.246469
std	1.993466	0.136096	21.741122	9.727528
min	-0.552064	0.229112	0.054666	-16.254445
25%	3.562770	0.617608	5.983017	3.380680
50%	4.916578	0.710784	13.792226	10.359616
75%	6.141336	0.802235	28.918100	16.720447
max	11.519088	0.977630	144.841692	42.054036

Standard Scaling

	normal_p	beta	exponential	normal_l
count	1.000000e+03	1.000000e+03	1.000000e+03	1.000000e+03
mean	-4.013456e-16	9.414691e-17	1.649791e-16	1.623354e-16
std	1.000500e+00	1.000500e+00	1.000500e+00	1.000500e+00
min	-2.740122e+00	-3.458430e+00	-9.635992e-01	-2.725685e+00
25%	-6.749296e-01	-6.024172e-01	-6.907836e-01	-7.061634e-01
50%	4.533255e-03	8.255543e-02	-3.314132e-01	1.163750e-02
75%	6.192264e-01	7.548552e-01	3.646614e-01	6.658647e-01
max	3.318265e+00	2.044261e+00	5.699326e+00	3.271487e+00

Practice

In the accompanied notebook for week 4, you will use the wine dataset to:

- use MinMax normalization using default parameters
- use MinMax normalization by setting 'feature_range'
- Use 'standardscaler' for the standardization
- Investigate the impact of different scaling methods on accuracy of an ML classifier (KNN)

2.7 Data Manipulation and Dealing with Outliers

Transforming Data

For example: adding a certain heart condition for patients with particular problems. Let's create a hypothetical dataset for different types of meat:

	Meat	Fat(gr)
0	bacon	16
1	rib eye	23
2	ham	10
3	Rib eye	27
4	Chicken breast	9
5	Bacon	25
6	drumsticks	21
7	Ham	18

```
pd.DataFrame({'Food':['bacon','rib eye', 'ham',
                      'Rib eye', 'Chicken breast',
                      'Bacon','drumsticks', 'Ham'],
              'Ounces':[16, 23, 10, 27, 9, 25,21, 18]})
```

```
meat_lower = food['Meat'].str.lower()
```

```
0              bacon
1            rib eye
2                ham
3            rib eye
4    chicken breast
5              bacon
6         drumsticks
7                ham
```

Mapping: map()

The '**map()**' method accepts a function or a dict-like object containing the mapping information:

- Create a dict-like object
- Map the object to column

```
food_to_animal = {'bacon':'pig','rib eye':'cow',
                  'ham':'pig', 'chicken breast':'chicken',
                 'drumstic':'chicken'}
```

```
food['animal'] = meat_lower.map(food_to_animal)
```

	Meat	Fat(gr)	animal
0	bacon	16	pig
1	rib eye	23	cow
2	ham	10	pig
3	Rib eye	27	cow
4	Chicken breast	9	chicken
5	Bacon	25	pig
6	drumsticks	21	NaN
7	Ham	18	pig

Replacing Values: replace()

- Create a series: 9999 is a sentinel value for null
- Replace them with NaN

```
data = pd.Series([2530, 2071, 9999, 2342, 9999])

data.replace(9999, np.nan)
```

```
0    2530.0
1    2071.0
2       NaN
3    2342.0
4       NaN
dtype: float64
```

```
data = pd.Series([2530, 2071, 9999, 2342, -9999])

data.replace([9999,-9999],np.nan)
```

```
0    2530.0
1    2071.0
2       NaN
3    2342.0
4       NaN
dtype: float64
```

Replacing Different Values

We pass a list or a dict. as substitutes to replace by different values.

```
data = pd.Series([2530, 2071, 9999, 2342, -9999])

data.replace([9999,-9999],[np.nan,0])
```

```
0    2530.0
1    2071.0
2       NaN
3    2342.0
4       0.0
dtype: float64
```

```
data.replace({9999:np.nan, -9999:0})
```

```
0    2530.0
1    2071.0
2       NaN
3    2342.0
4       0.0
dtype: float64
```

Replacing Axis Indexes

- By a function or some form of mapping:

```
data = pd.DataFrame(np.arange(15).reshape((3,5)),
            index=['Ontario','British Col.', 'Alberta'],
                    columns = ['one','two','three','four','five'])
```

	one	two	three	four	five
Ontario	0	1	2	3	4
British Col.	5	6	7	8	9
Alberta	10	11	12	13	14

- Using map() method with function:

```
transform = lambda x:x[:4].upper()

data.index = data.index.map(transform)
```

	one	two	three	four	five
ONTA	0	1	2	3	4
BRIT	5	6	7	8	9
ALBE	10	11	12	13	14

Changing Axis Values: rename()

Use 'rename()' to keep the original version of the DataFrame when transforming axis values:

	one	two	three	four	five
ONTA	0	1	2	3	4
BRIT	5	6	7	8	9
ALBE	10	11	12	13	14

```
data.rename(index=str.title, columns=str.upper)
```

	ONE	TWO	THREE	FOUR	FIVE
Onta	0	1	2	3	4
Brit	5	6	7	8	9
Albe	10	11	12	13	14

rename() With a Dict-Like Object

To replace previous values or create new ones:

```
data.rename(index = {'ONTA':'Ontario','BRIT':'BC','ALBE':'Alberta'},
            columns = {'two':'Too'})
```

	one	Too	three	four	five
Ontario	0	1	2	3	4
BC	5	6	7	8	9
Alberta	10	11	12	13	14

Discretization and Binning: cut()

We normally discretize continuous values into different bins to analyze them better (e.g., binning the score of students to 3 bins: accomplished, developing, beginning).

```
grades = [53, 71, 89, 94, 81, 78, 62,
          98, 43, 84, 91, 32, 84, 23]

bins = [0,65,85,100]

groups = pd.cut(grades, bins)

groups
```

```
[(0, 65], (65, 85], (85, 100], (85, 100], (65, 85], ..., (65, 85], (85, 10
0], (0, 65], (65, 85], (0, 65]]
Length: 14
Categories (3, interval[int64, right]): [(0, 65] < (65, 85] < (85, 100]]
```

Labelling the Bins

Label the interval-based bins by passing a list to the labels:

```
bin_names = ['accomplished', 'developing', 'beginning']

groups = pd.cut(grades, bins, labels = bin_names)
```

```
['accomplished', 'developing', 'beginning', 'beginning', 'developing', ...,
'developing', 'beginning', 'accomplished', 'developing', 'accomplished']
Length: 14
Categories (3, object): ['accomplished' < 'developing' < 'beginning']
```

Properties of the Categories

```
pd.value_counts(groups)
```

```
(0, 65]      5
(65, 85]     5
(85, 100]    4
dtype: int64
```

```
groups.codes
```

```
array([0, 1, 2, 2, 1, 1, 0, 2, 0, 1, 2, 0, 1, 0], dtype=int8)
```

```
groups.categories
```

```
IntervalIndex([(0, 65], (65, 85], (85, 100]], dtype='interval[int64, righ
t]')
```

```
groups.categories[0]
```

```
Interval(0, 65, closed='right')
```

Equal Size Binning

We can automatically create equal size bins by dividing the space between min and max values of the series and passing the number of bins we want to see:

```
pd.cut(grades, 3, precision = 2)
```

```
[(48.0, 73.0], (48.0, 73.0], (73.0, 98.0], (73.0, 98.0], (73.0, 98.0], ...,
(73.0, 98.0], (73.0, 98.0], (22.92, 48.0], (73.0, 98.0], (22.92, 48.0]]
Length: 14
Categories (3, interval[float64, right]): [(22.92, 48.0] < (48.0, 73.0] <
(73.0, 98.0]]
```

Binning Based on Quantile

Use '**qcut()**' to bin data based on sample quartiles:

```
pd.qcut(grades, 4, precision = 3)
```

```
[(22.999, 55.25], (55.25, 79.5], (87.75, 98.0], (87.75, 98.0], (79.5, 87.7
5], ..., (79.5, 87.75], (87.75, 98.0], (22.999, 55.25], (79.5, 87.75], (22.
999, 55.25]]
Length: 14
Categories (4, interval[float64, right]): [(22.999, 55.25] < (55.25, 79.5]
< (79.5, 87.75] < (87.75, 98.0]]
```

Identifying Outliers

If we know the thresholds for acceptable data, we can identify the outliers which are beyond the threshold values and cap their values.

```
data = pd.DataFrame(np.random.standard_normal((1000,4)))
```

	0	1	2	3
0	-0.341027	2.297632	-0.633064	1.666115
1	0.806028	-1.808895	1.200169	-1.584038
2	0.337121	-0.820621	-0.426140	1.690999
3	-0.701258	0.305161	-0.799738	0.144737
4	0.266612	1.599341	0.386659	1.356866
...	...	...	...	...

```
sample = data[3]

sample[sample.abs()>3]
```

```
712   -3.404080
722    3.034730
757   -3.131987
Name: 3, dtype: float64
```

```
data[(data.abs()>3).any(axis = "columns")]
```

	0	1	2	3
107	-3.073083	-2.168603	0.380217	1.731546
201	1.987942	-2.487039	3.260914	1.505582
540	-3.166136	-1.474591	0.339012	0.597325
712	0.094401	0.486046	1.070580	-3.404080
722	-1.124540	-0.078595	1.462910	3.034730
757	-0.755023	0.129219	0.021425	-3.131987
886	-2.187518	-3.192074	2.556581	-0.405970
905	-3.539749	1.584037	1.459121	0.762561

Capping Outliers

After identifying the outliers, we want to replace them by either -3 or +3, depending on their **sign**.

```
data[data.abs()>3] = np.sign(data)*3

data.describe()
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.037939	0.039144	0.007414	0.003856
std	1.013093	0.999397	1.033865	0.992253
min	-3.000000	-3.000000	-2.751731	-3.000000
25%	-0.724467	-0.663403	-0.635937	-0.664165
50%	-0.043164	0.013819	0.057865	-0.007573
75%	0.669547	0.732505	0.707399	0.658153
max	2.951395	2.878413	3.000000	3.000000

Random Sampling With Permutation

Permutation is the random selection of values by reordering a series or the rows in a DataFrame.

```
dataF = pd.DataFrame(np.arange(40).reshape((5,8)))

dataF
```

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39

```
rndSample = np.random.permutation(3

rndSample
```

```
array([2, 1, 0])
```

```
dataF.take(rndSample)
```

	0	1	2	3	4	5	6	7
2	16	17	18	19	20	21	22	23
1	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7

```
dataF.iloc[rndSample]
```

	0	1	2	3	4	5	6	7
2	16	17	18	19	20	21	22	23
1	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7

Permutation on Columns

Use '**take()**' to create a permutation of columns instead of rows by invoking **axis="columns"**:

```
clm_sample = np.random.permutation(8)

clm_sample
```

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

```
dataF.take(clm_sample, axis = "columns")
```

	2	1	6	7	5	3	0	4
0	2	1	6	7	5	3	0	4
1	10	9	14	15	13	11	8	12
2	18	17	22	23	21	19	16	20
3	26	25	30	31	29	27	24	28
4	34	33	38	39	37	35	32	36

Sampling (With/Without Replacement)

Use '**sample()**' to choose a random sample from a series or DataFrame:

```
dataF.sample(n=3)
```

	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
0	0	1	2	3	4	5	6	7
3	24	25	26	27	28	29	30	31

```
series.sample(n=10, replace = True)
```

```
3     8
2    -2
1     6
4    11
1     6
5    31
2    -2
2    -2
2    -2
3     8
dtype: int64
```

Practice

In the accompanied notebook for week 4, you will use the HealthCare Analytics dataset to:

- convert the columns to lower/upper case
- replace different values
- encode data by replacement
- group data-set based on criteria
- bin the patient IDs into 3 equal bins
- bin the patient IDs using quartiles
- identify outliers
- cap the outliers by a coefficient of their standard deviation
- perform random sampling with permutation
- sample values with/without replacement

2.8 Visualization

Data Visualization

- Data visualization is a very important part of data exploration and can be used in all stages of any steps of an ML project.
- When given any data, we start by graphing the observations to find insights about patterns, correlations, outliers.
- When reporting the performance of ML models and the results, we use graphs.
- Usually in order to interpret the complex models, we project high dimensional datasets into 2D or 3D spaces for better visualization.

Visualization in Python

- There are many tools in Python to visualize data:
    - matplotlib.pyplot
    - Pandas
    - Seaborn
    - Plotly

- In this section we work with matplotlib, Pandas, and Seaborn for visualization. For the first two we generate our own data and for Seaborn we use the Seaborn datasets.

Using Matplotlib Library

- Import matplotlib to set up the necessary packages:

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

- A simple plot after generating numbers from 0 to 10:

```
import numpy as np

data = np.arange(10)
plt.plot(data);
```

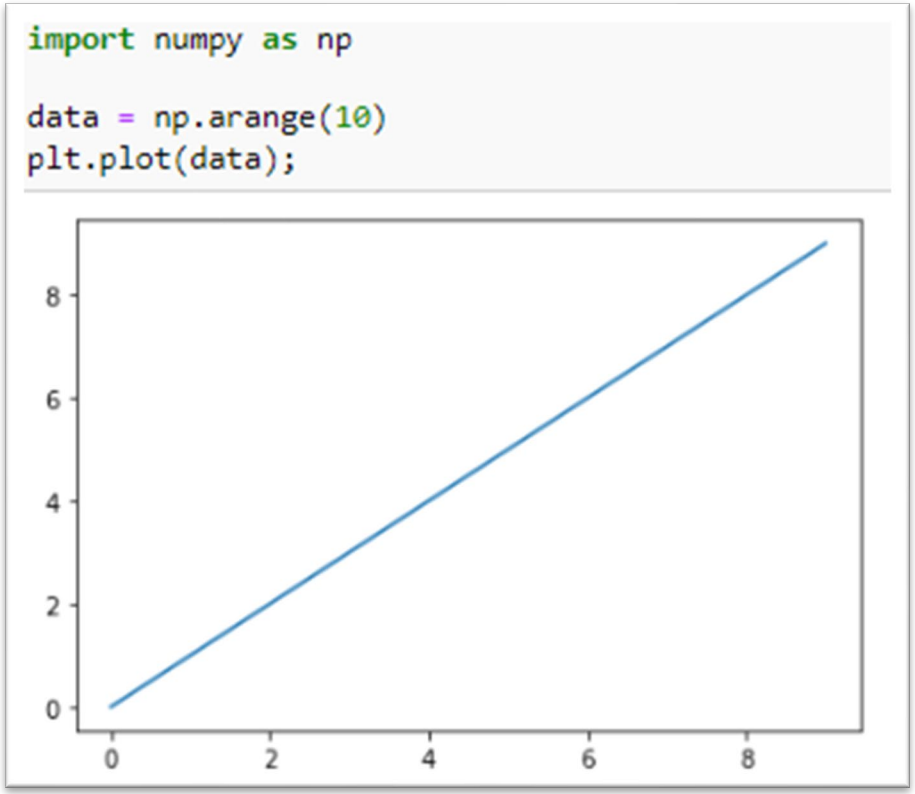


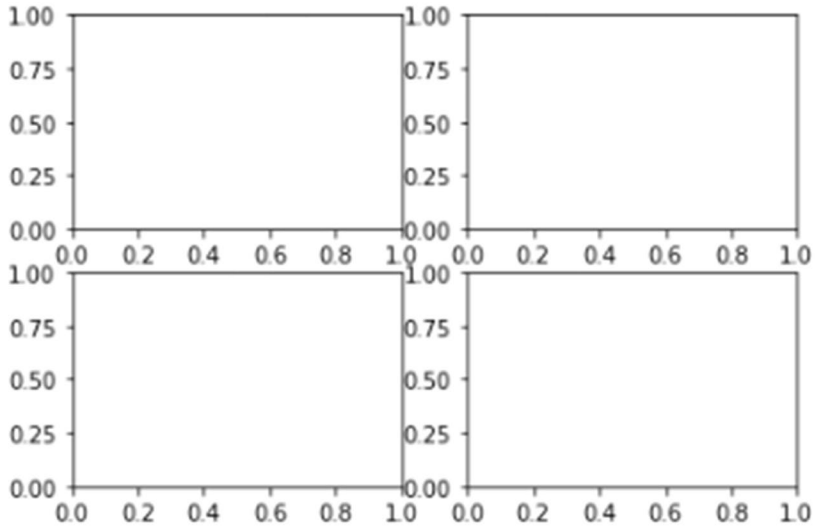
Figure Object

- Plots are within *figure object* in matplotlib, and new plots can be created with '**plt.figure()**'.

- We use '**add_subplot()**' to add a plot to the figure.

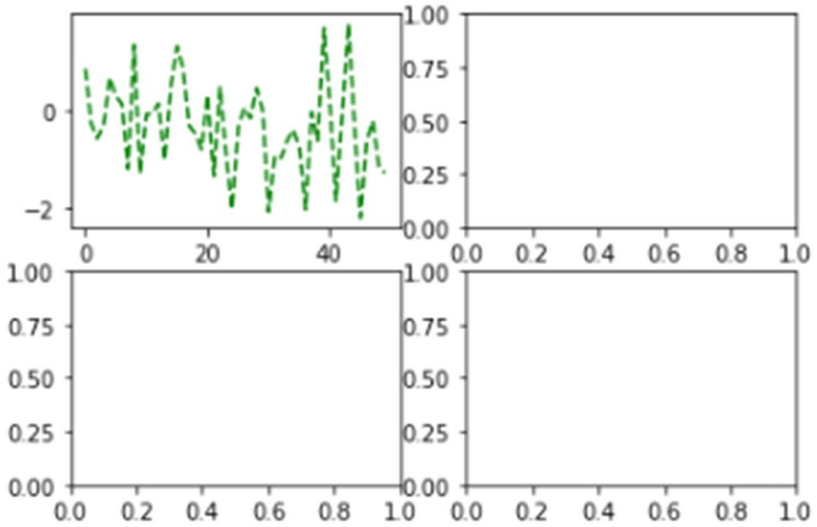
- '**add_subplot()**' has arguments to show how many plots will be inside a figure and how to locate each plot. For example, (2,2,1) indicates we have a 2x2 figure and we are focusing on the first plot.

```
fig = plt.figure()

plot1 = fig.add_subplot(2,2,1)
plot2 = fig.add_subplot(2,2,2)
plot3 = fig.add_subplot(2,2,3)
plot4 = fig.add_subplot(2,2,4);
```



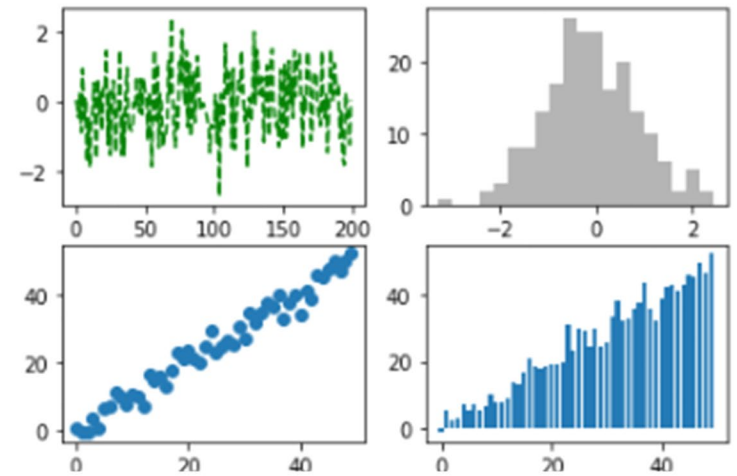
Adding Graphs to Subplots

```
fig = plt.figure()

plot1 = fig.add_subplot(2,2,1)
plot2 = fig.add_subplot(2,2,2)
plot3 = fig.add_subplot(2,2,3)
plot4 = fig.add_subplot(2,2,4)

plot1.plot(np.random.standard_normal(50),
           color = "green",linestyle="dashed");
```



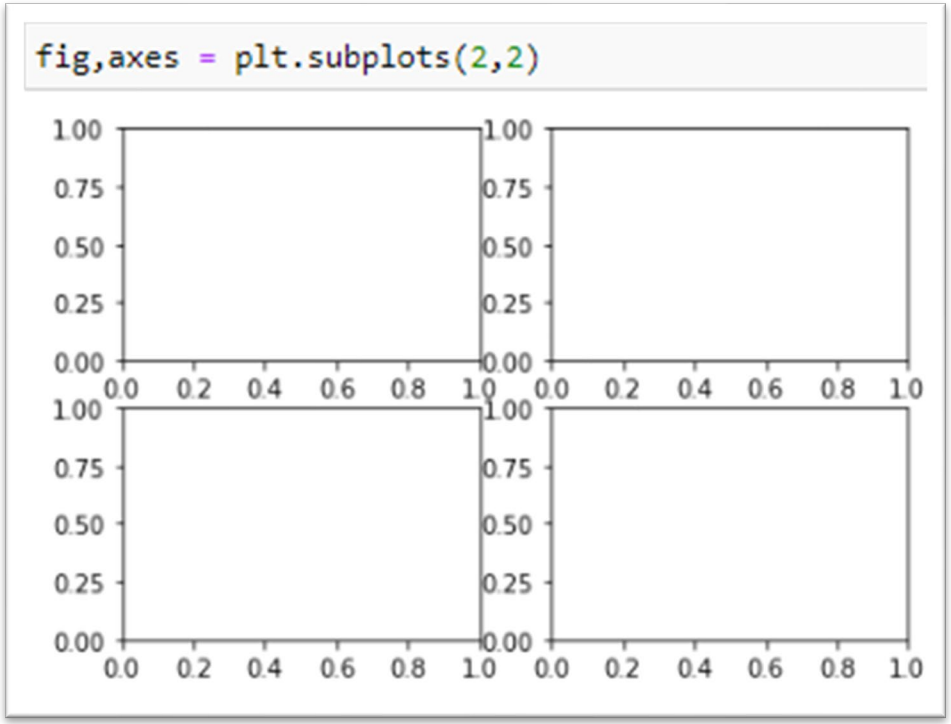
```
fig = plt.figure()

plot1 = fig.add_subplot(2,2,1)
plot2 = fig.add_subplot(2,2,2)
plot3 = fig.add_subplot(2,2,3)
plot4 = fig.add_subplot(2,2,4)

plot1.plot(np.random.standard_normal(200),
           color = "green",linestyle="dashed")

plot2.hist(np.random.standard_normal(200),
           bins = 20, color="black",alpha=0.3)

plot3.scatter(np.arange(50),np.arange(50)+
              3 * np.random.standard_normal(50))

plot4.bar(np.arange(50),np.arange(50)+
          3 * np.random.standard_normal(50))
;
```

''



Subplots Method

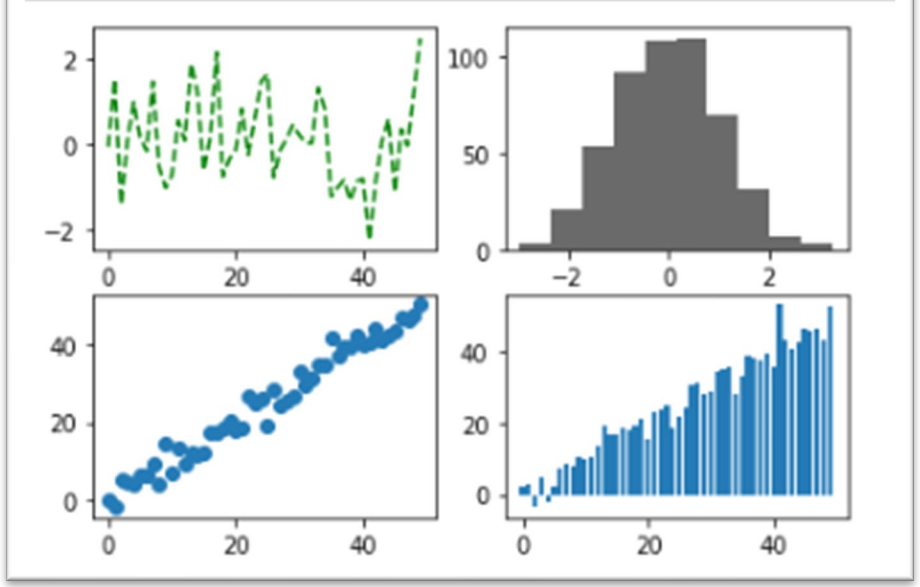
'**plt.subplots()**' helps with creating a grid-form figure with the desired subplot objects.



Subplots Method

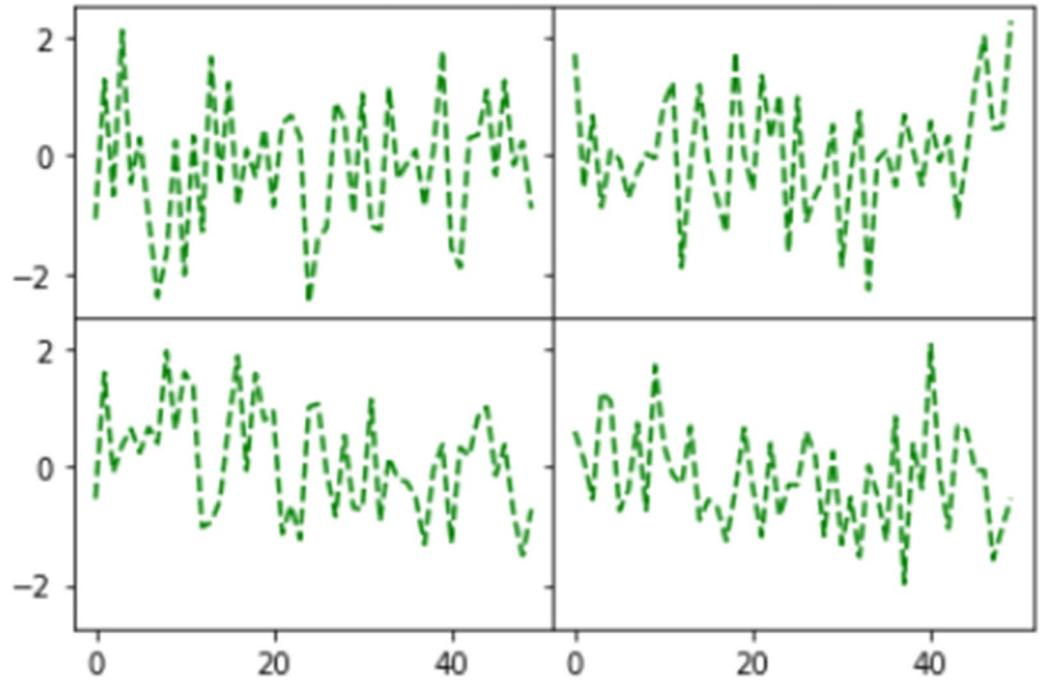
The axes or plots in the figure, which generated by 'plt.subplots()', can be indexed as a multi-dimensional array.

```
fig,axes = plt.subplots(2,2)

axes[0,0].plot(np.random.standard_normal(50),
          color = "green",linestyle="dashed")

axes[0,1].hist(np.random.standard_normal(500),
          bins = 10, color="black",alpha=0.6)

axes[1,0].scatter(np.arange(50),np.arange(50)+
              3 * np.random.standard_normal(50))

axes[1,1].bar(np.arange(50),np.arange(50)+
          3 * np.random.standard_normal(50));
```

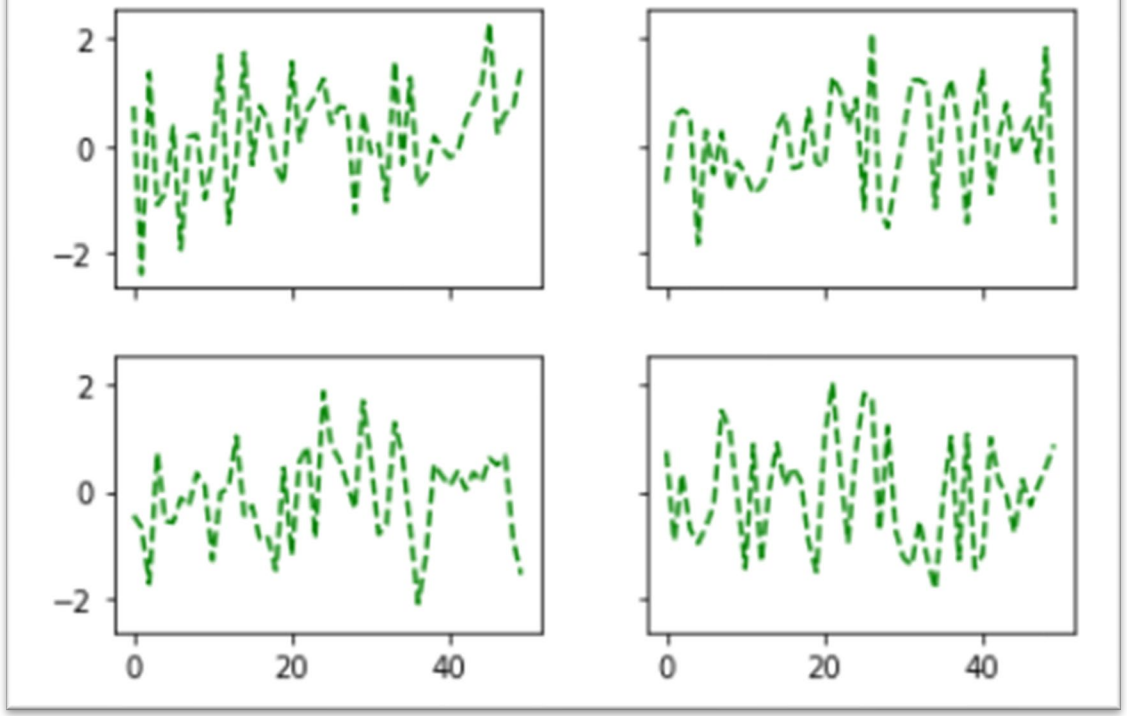


Sharing Axes Among Subplots

```
fig,axes = plt.subplots(2,2,sharex = True,sharey = True)

for i in range(2):
    for j in range(2):
        axes[i,j].plot(np.random.standard_normal(50),
           color = "green",linestyle="dashed")
fig.subplots_adjust(wspace=0.0,hspace=0.0)
```



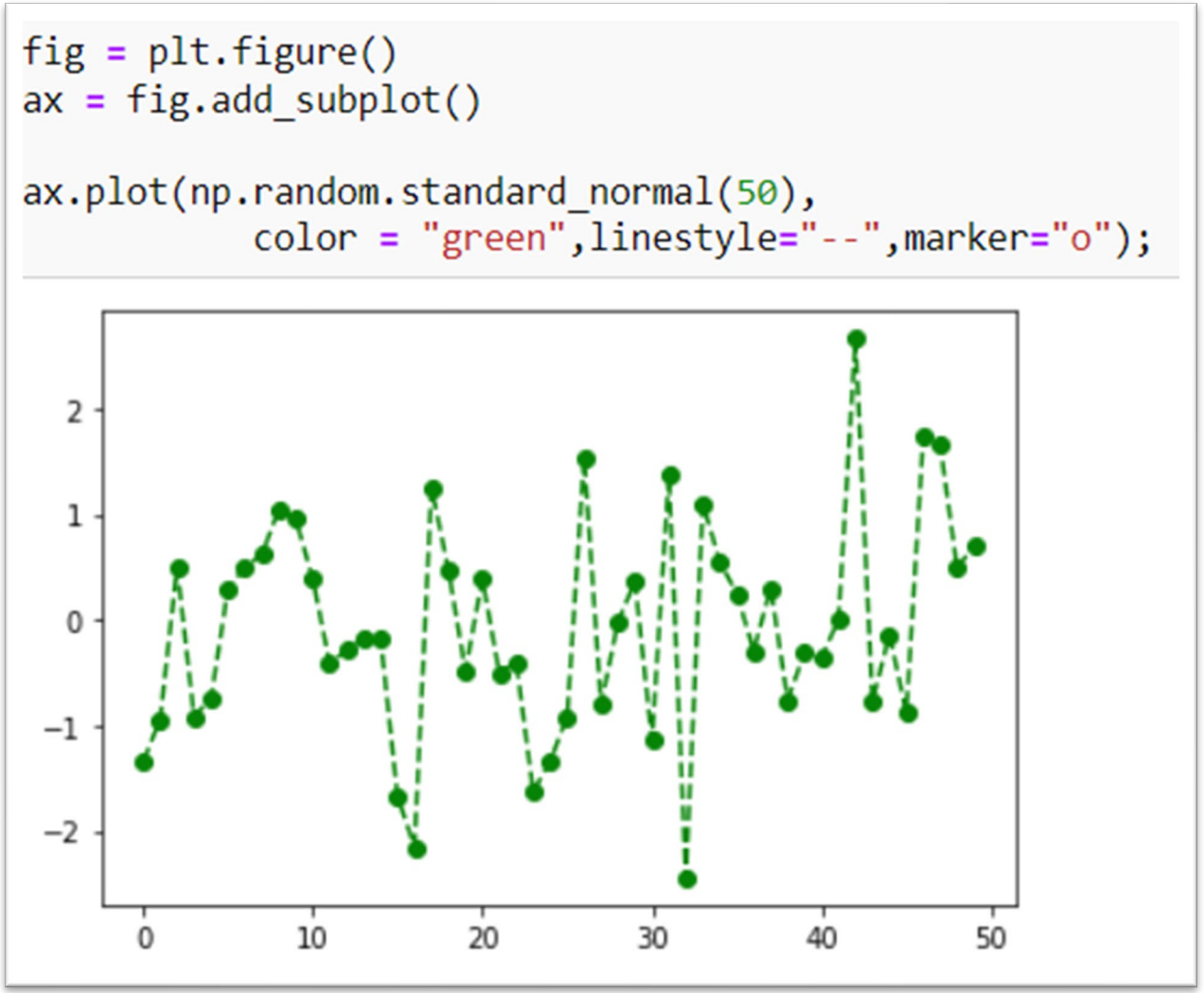

```
fig.subplots_adjust(wspace=.25,hspace=0.25)
```



Markers

Markers in the line plots are used to highlight some of the datapoints.

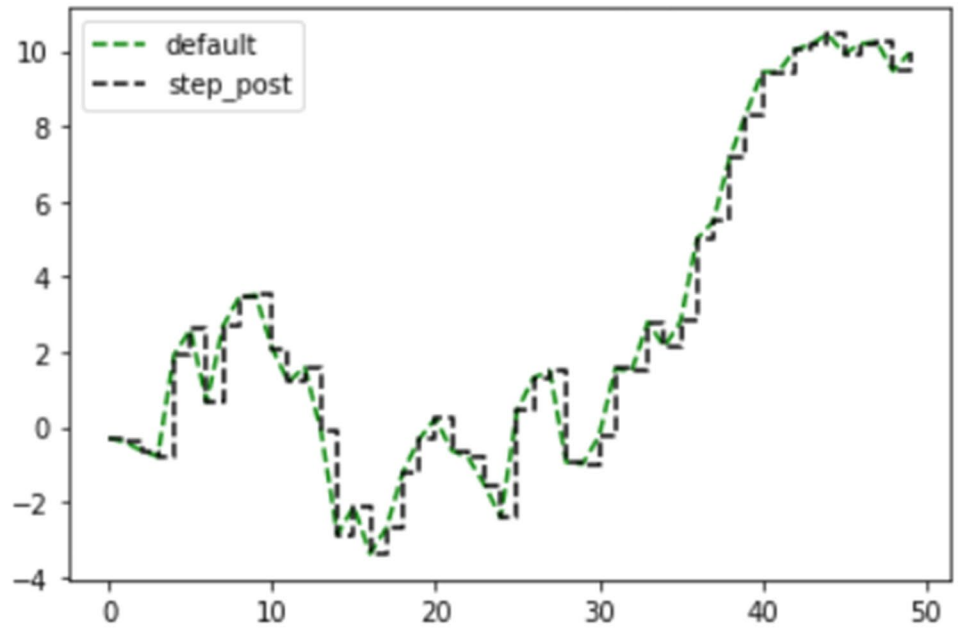
```
fig = plt.figure()
ax = fig.add_subplot()

ax.plot(np.random.standard_normal(50),
            color = "green",linestyle="--",marker="o");
```



drawstyle="steps-post"

```
fig = plt.figure()
ax = fig.add_subplot()

data = np.random.standard_normal(50).cumsum()

ax.plot(data, color = "green",linestyle="--",label = "default")

ax.plot(data, color = "black",linestyle="--",
        drawstyle="steps-post", label = "step_post")

ax.legend()
;
```

```
''
```



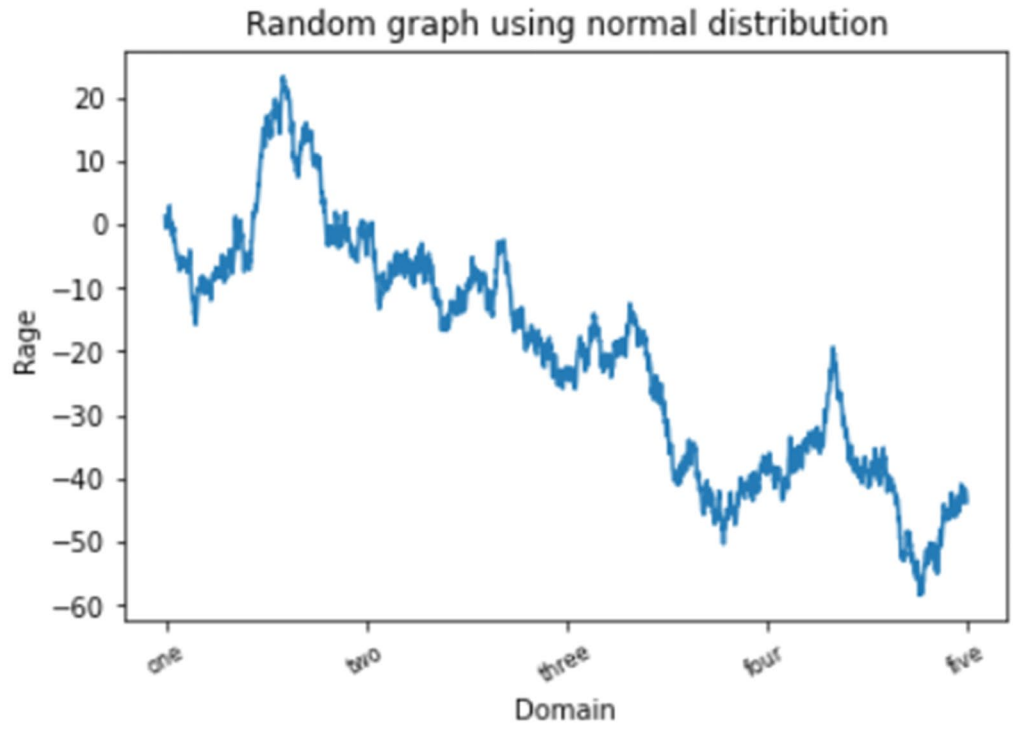
Legends and Ticks

We can control many features on axes objects by using:

- '**xlim()**' to control range
- '**xticks()**' to control tick locations
- '**xticklabels()**' to control the tick labels

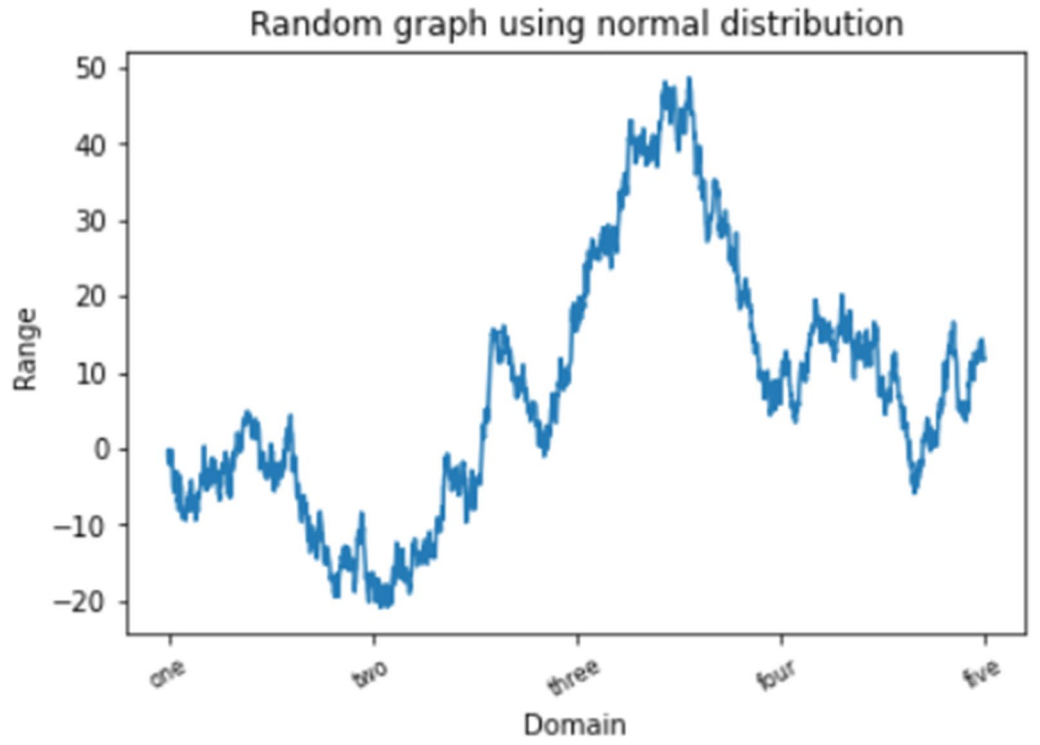
```
fig,ax = plt.subplots()

ax.plot(np.random.standard_normal(2000).cumsum())

ax.set_xticks([0, 500, 1000, 1500, 2000])
ax.set_xticklabels(["one","two","three","four","five"],
                                rotation = 30, fontsize = 8)

ax.set_xlabel("Domain")
ax.set_ylabel("Rage")
ax.set_title("Random graph using normal distribution")
```

```
Text(0.5, 1.0, 'Random graph using normal distribution')
```



Aggregated Way to Set Labels and Title

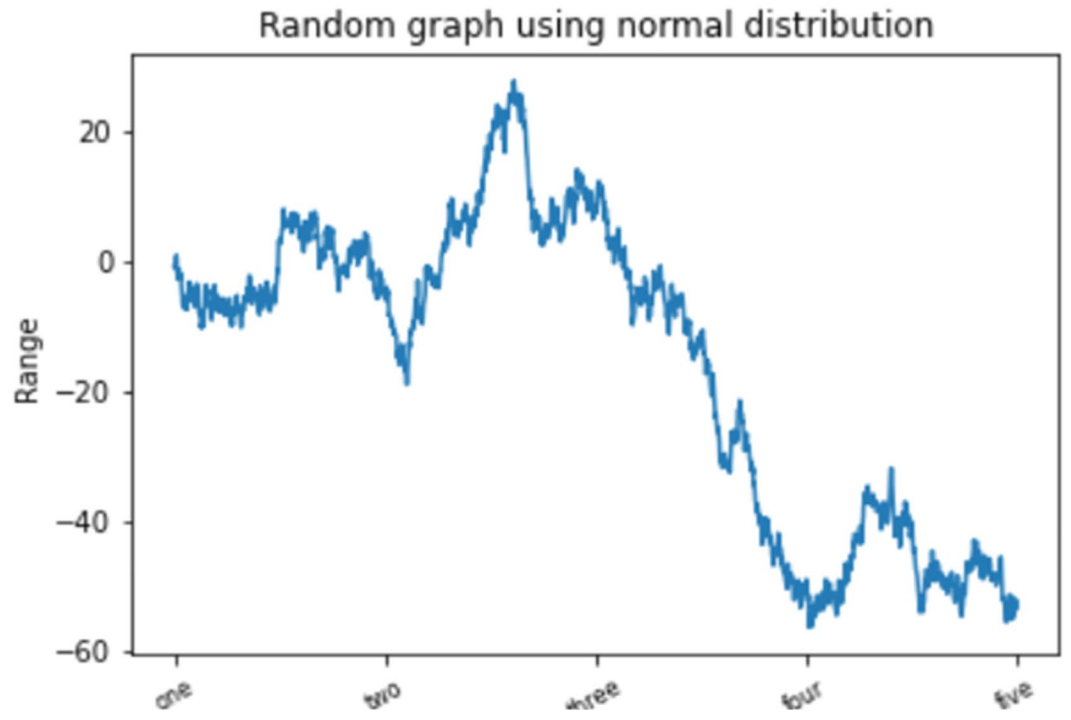
```
fig,ax = plt.subplots()

ax.plot(np.random.standard_normal(2000).cumsum())

ax.set_xticks([0, 500, 1000, 1500, 2000])
ax.set_xticklabels(["one","two","three","four","five"],
                                rotation = 30, fontsize = 8)

ax.set(title="Random graph using normal distribution",
       xlabel="Domain",ylabel="Range");
```



Saving Plots to File

The active figure can be saved using '**savefig()**' method.

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

ax.plot(np.random.standard_normal(2000).cumsum())

ax.set_xticks([0, 500, 1000, 1500, 2000])
ax.set_xticklabels(["one","two","three","four","five"],
                                rotation = 30, fontsize = 8)

ax.set(title="Random graph using normal distribution",
       xlabel="Domain",ylabel="Range")

fig.savefig("firstfigure.png", dpi=400);
```

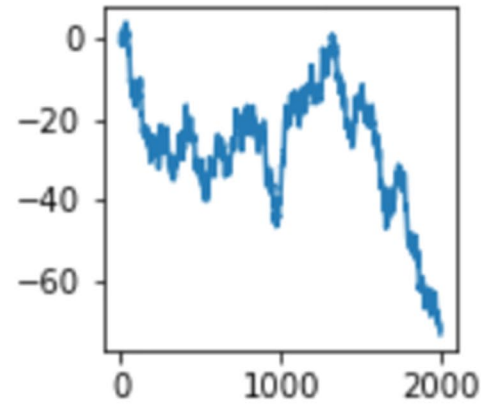


Global Configuration

- We can set the graph parameters globally, so they can apply to all the graphs and visualizations in our project, using '**rc()**' method.

- The first argument in this method is the component we want to configure (e.g., figure, axes, ytick, grid, legend, etc.).

```
plt.rc("figure",figsize=(2,2))

fig,ax = plt.subplots()
ax.plot(np.random.standard_normal(2000).cumsum());
```



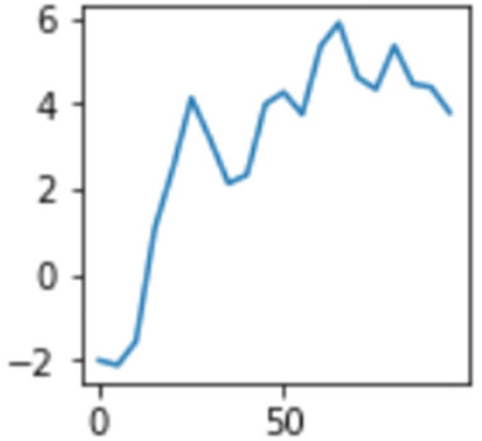
Plotting With Pandas

Plotting Series

```
import numpy as np
import pandas as pd

s1 = pd.Series (np.random.standard_normal(20).cumsum(),
                index = np.arange(0,100,5))
```

```
0    -2.015637
5    -2.133066
10   -1.569220
15    1.059192
20    2.494607
25    4.137073
30    3.178053
35    2.135551
```

```
s1.plot();
```

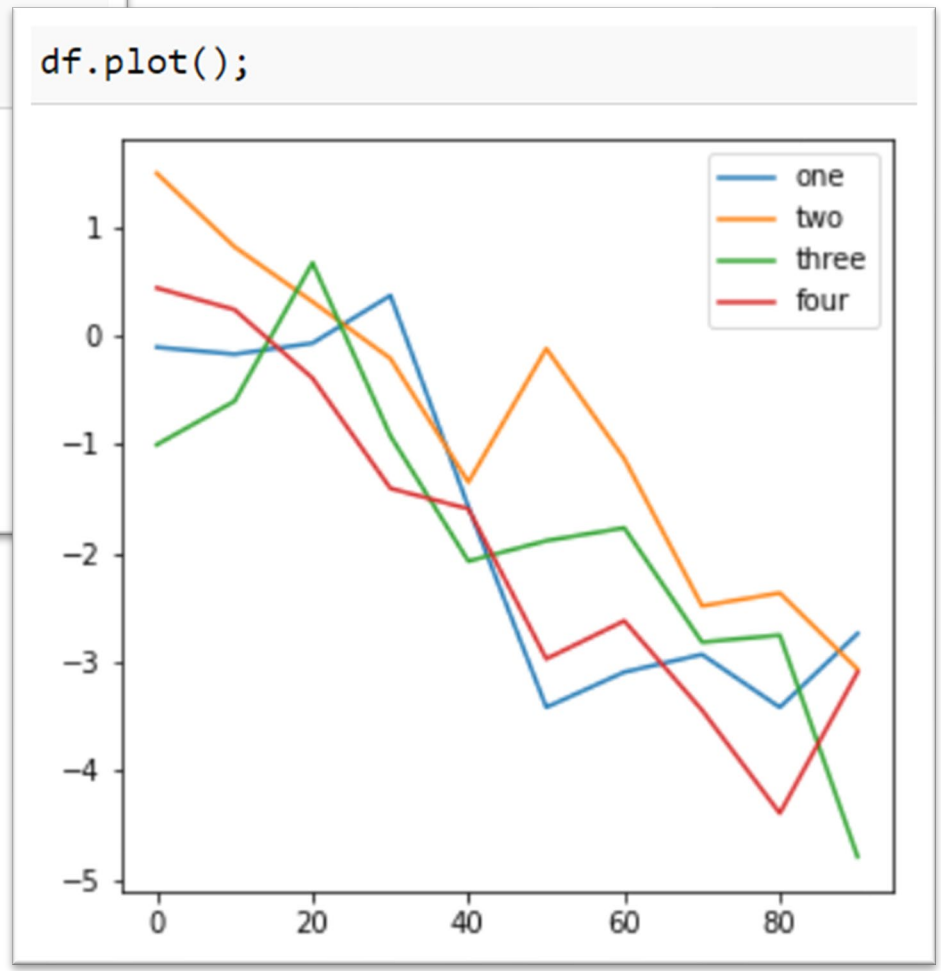


Plotting DataFrame

```
df = pd.DataFrame(np.random.standard_normal((10,4)).cumsum(0),
                  columns = ["one","two","three","four"],
                  index=np.arange(0,100,10))
```

	one	two	three	four
0	0.142014	1.582536	0.973458	-0.755628
10	1.624011	0.277420	1.701485	0.526614
20	1.585947	-0.961818	0.180747	0.312026
30	0.149437	-2.084239	-1.910615	0.696961

```
df.plot();
```



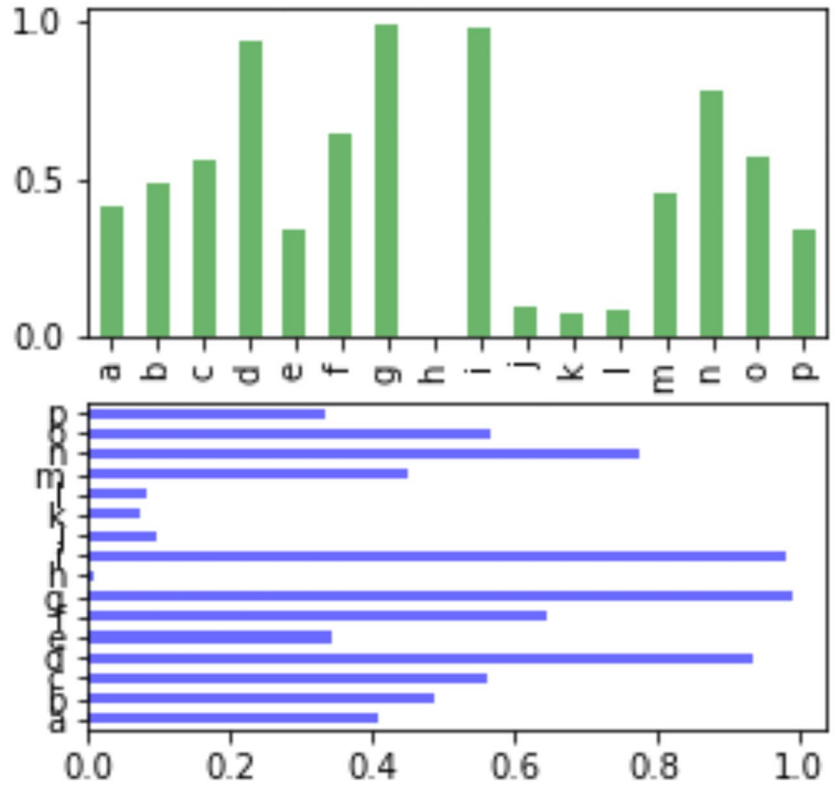
Families of Plots

The 'plot' attribute includes a family of graphs:

- '**df.plot.line()**' is the same as '**df.plot()**'
- '**plot.bar()**' is for bar graphs
- '**plot.barh()**' is for horizontal bar graphs
- Other plots:
  - '**hist**' – histogram
  - '**box**’ – boxplot
  - '**area**' – area plots
  - '**hexbin**' – hexagonal bin plot
  - '**pie**' – pie plot

Bar Plots

```
fig,axes =plt.subplots(2,1)

data.plot.bar(ax=axes[0],color="green",alpha = 0.6)
data.plot.barh(ax=axes[1],color="blue",alpha = 0.6);
```

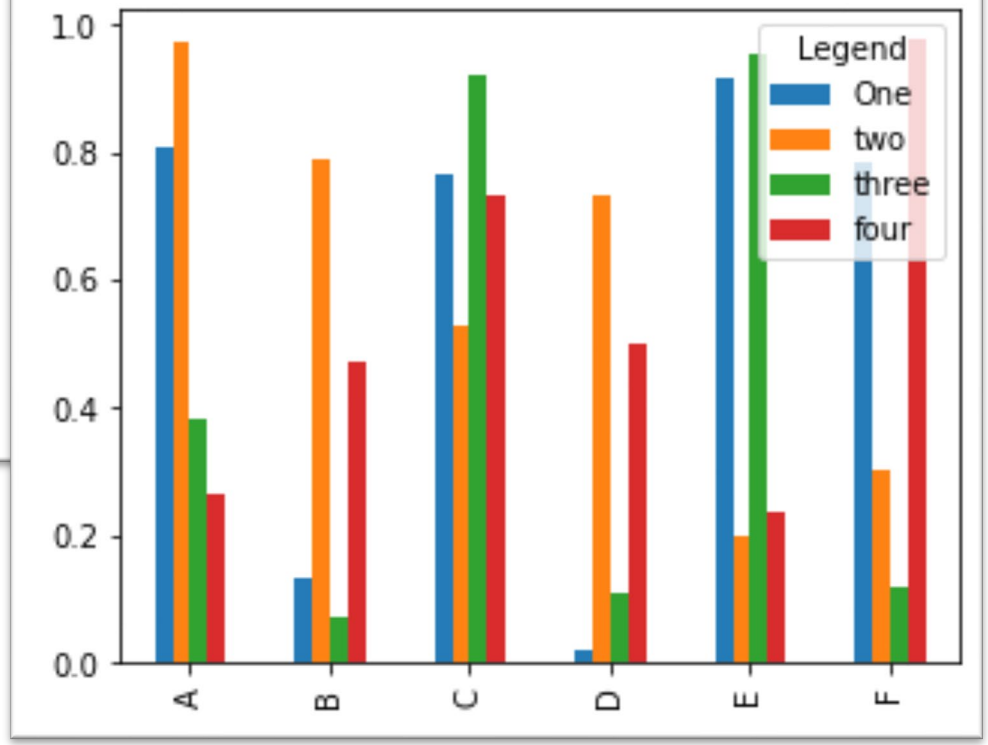


Bar Graphs for DataFrames

```
df = pd.DataFrame(np.random.uniform(size=(6,4)),
                  index = ["A","B","C","D","E","F"],
                  columns=pd.Index(["One","two", "three", "four"],
                                   name = "Legend"))
```

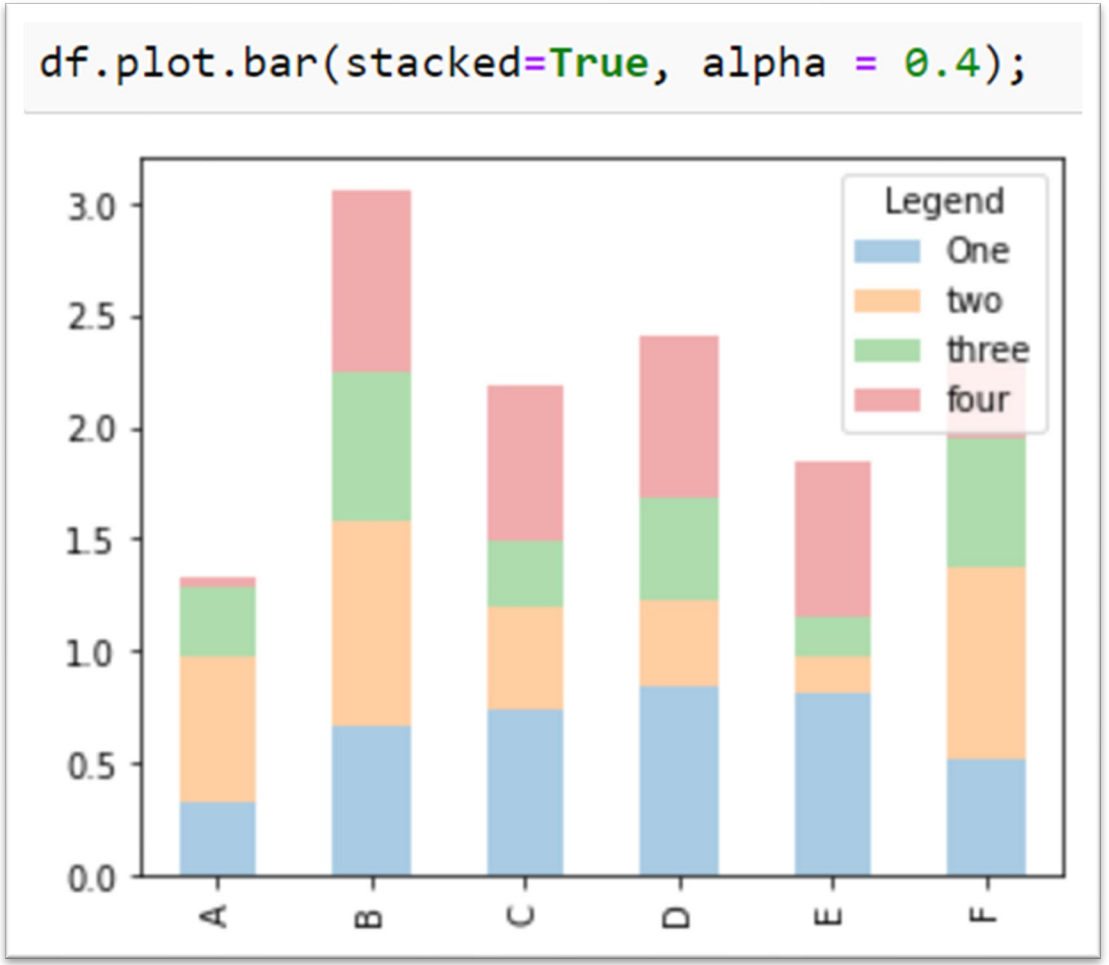
Legend	One	two	three	four
A	0.319612	0.660605	0.312404	0.033086
B	0.662677	0.912784	0.675865	0.807457
C	0.732356	0.470158	0.295481	0.689317
D	0.850035	0.378432	0.457186	0.728462
E	0.806301	0.170910	0.175955	0.689970

```
df.plot.bar();
```



Bar Plot With Stacked and Alpha

```
df.plot.bar(stacked=True, alpha = 0.4);
```



Seaborn

source https://seaborn.pydata.org/tutorial/distributions.html

Get Started With Seaborn

- Import the packages to start visualization with seaborn:

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

%matplotlib inline
```

- we will use seaborn datasets through the upcoming slides.

```
print(sns.get_dataset_names())
```

```
['anagrams', 'anscombe', 'attention', 'brain_networks', 'car_crashes',
'diamonds', 'dots', 'exercise', 'flights', 'fmri', 'gammas', 'geyser',
'iris', 'mpg', 'penguins', 'planets', 'taxis', 'tips', 'titanic']
```

Histograms With Penguin Dataset

We use penguin dataset to draw histograms:

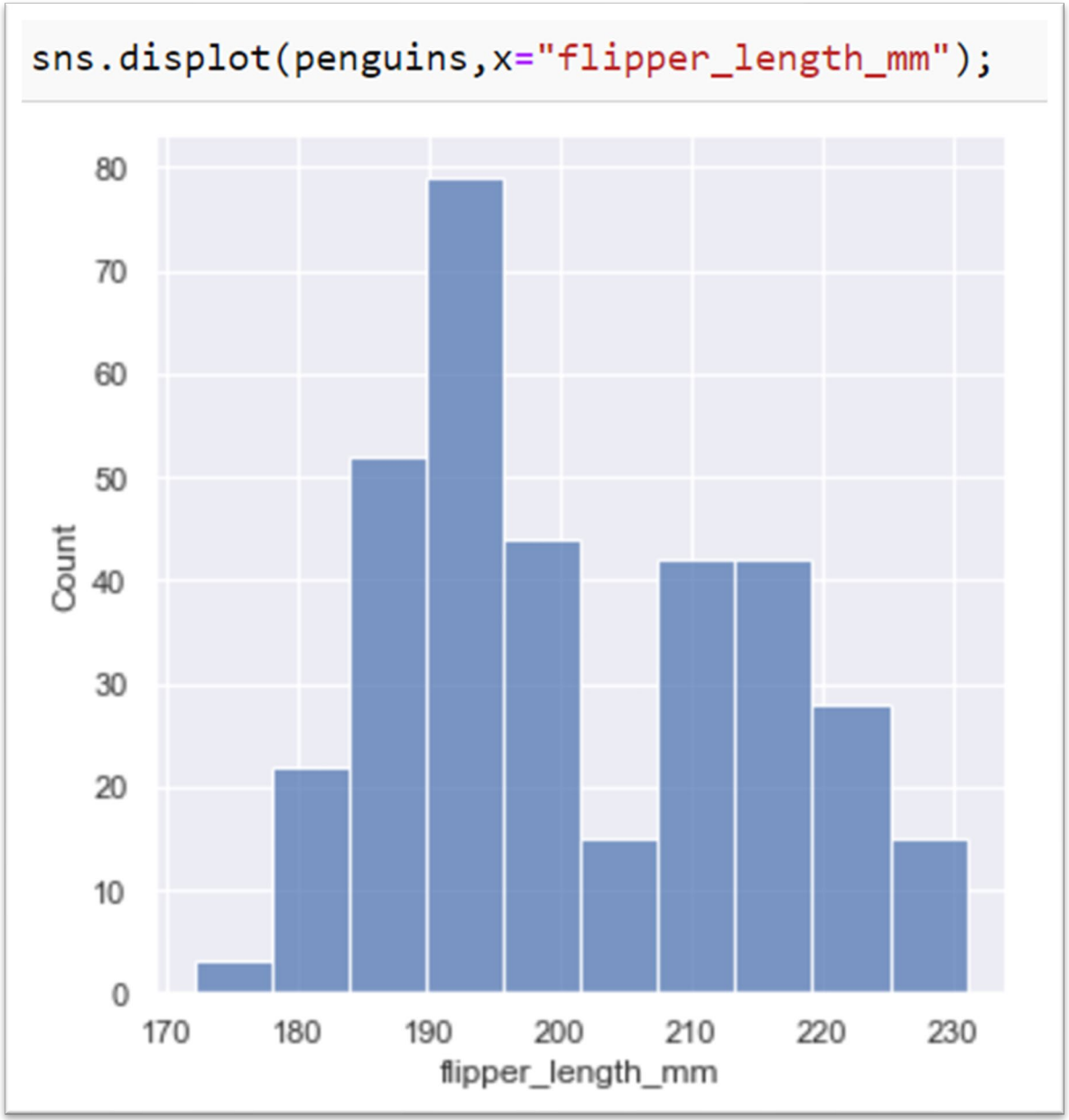
```
penguins = sns.load_dataset('penguins')
penguins.head()
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex
0	Adelie	Torgersen	39.1	18.7	181.0	3750.0	Male
1	Adelie	Torgersen	39.5	17.4	186.0	3800.0	Female
2	Adelie	Torgersen	40.3	18.0	195.0	3250.0	Female

Seaborn Histograms

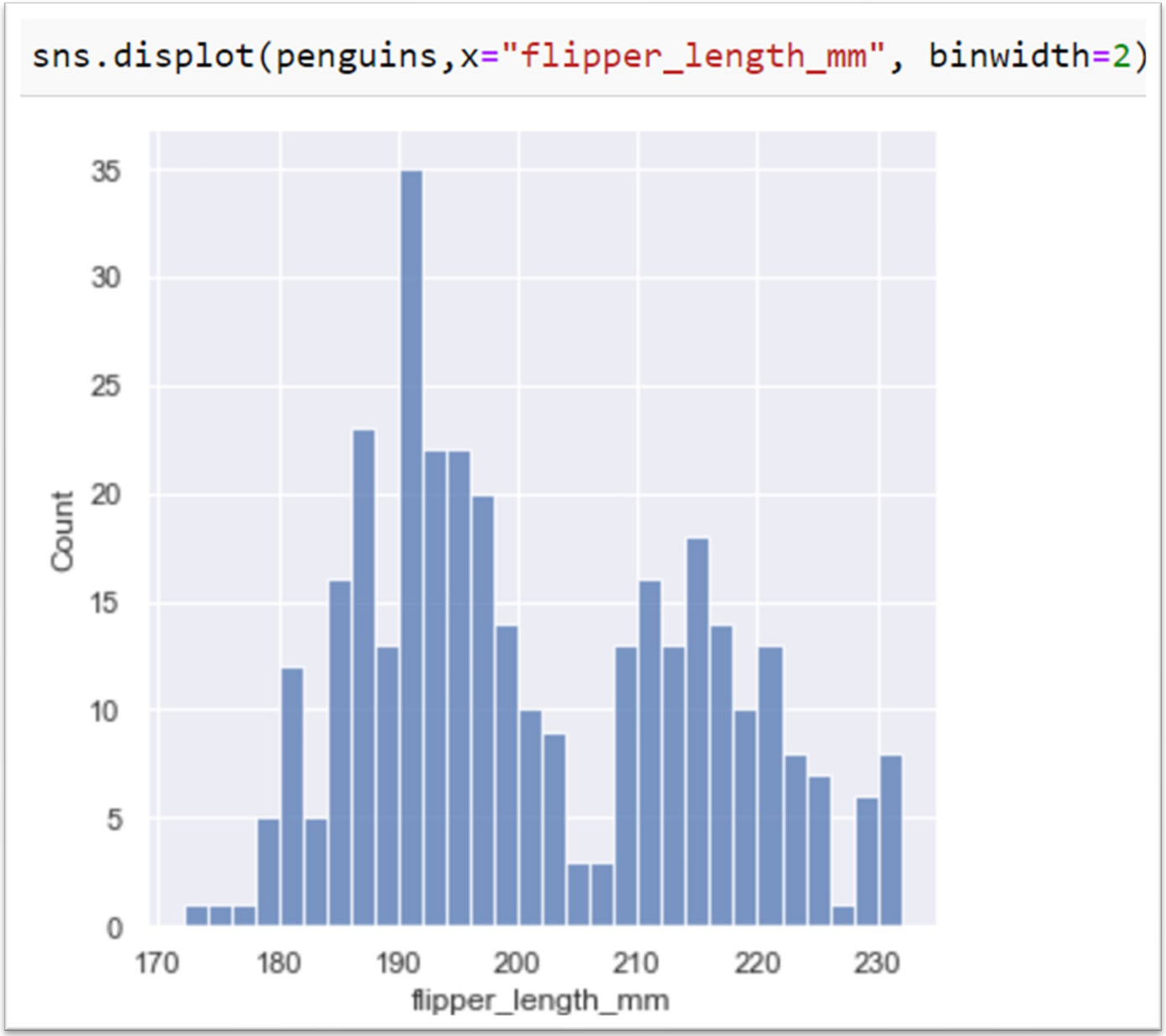
Use '**displot()**' function to visualize data through histograms:

```
sns.displot(penguins,x="flipper_length_mm");
```



Changing Histograms Bin Width

```
sns.displot(penguins,x="flipper_length_mm", binwidth=2)
```

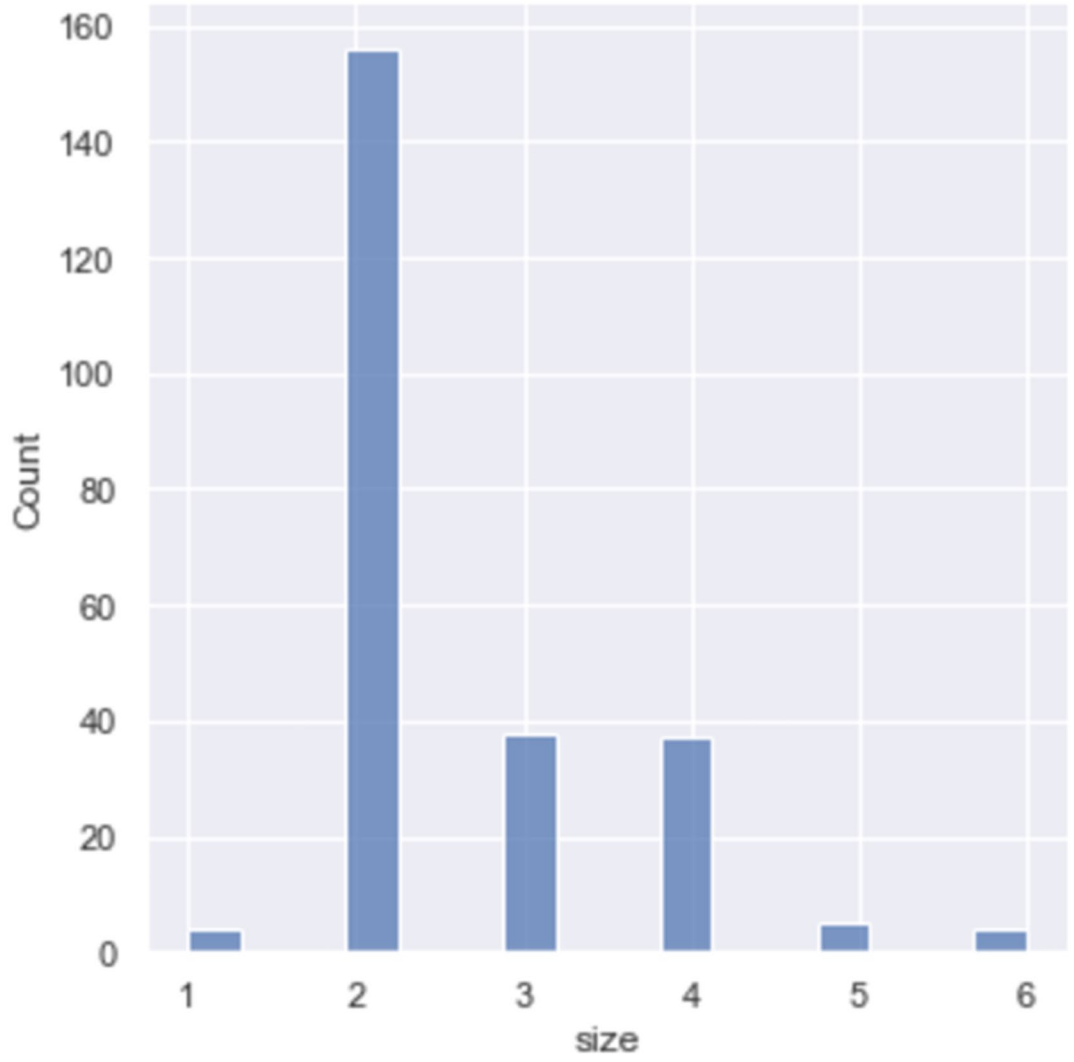


Histograms Bin Breaks

```
tips = sns.load_dataset("tips")
tips.head(3)
```

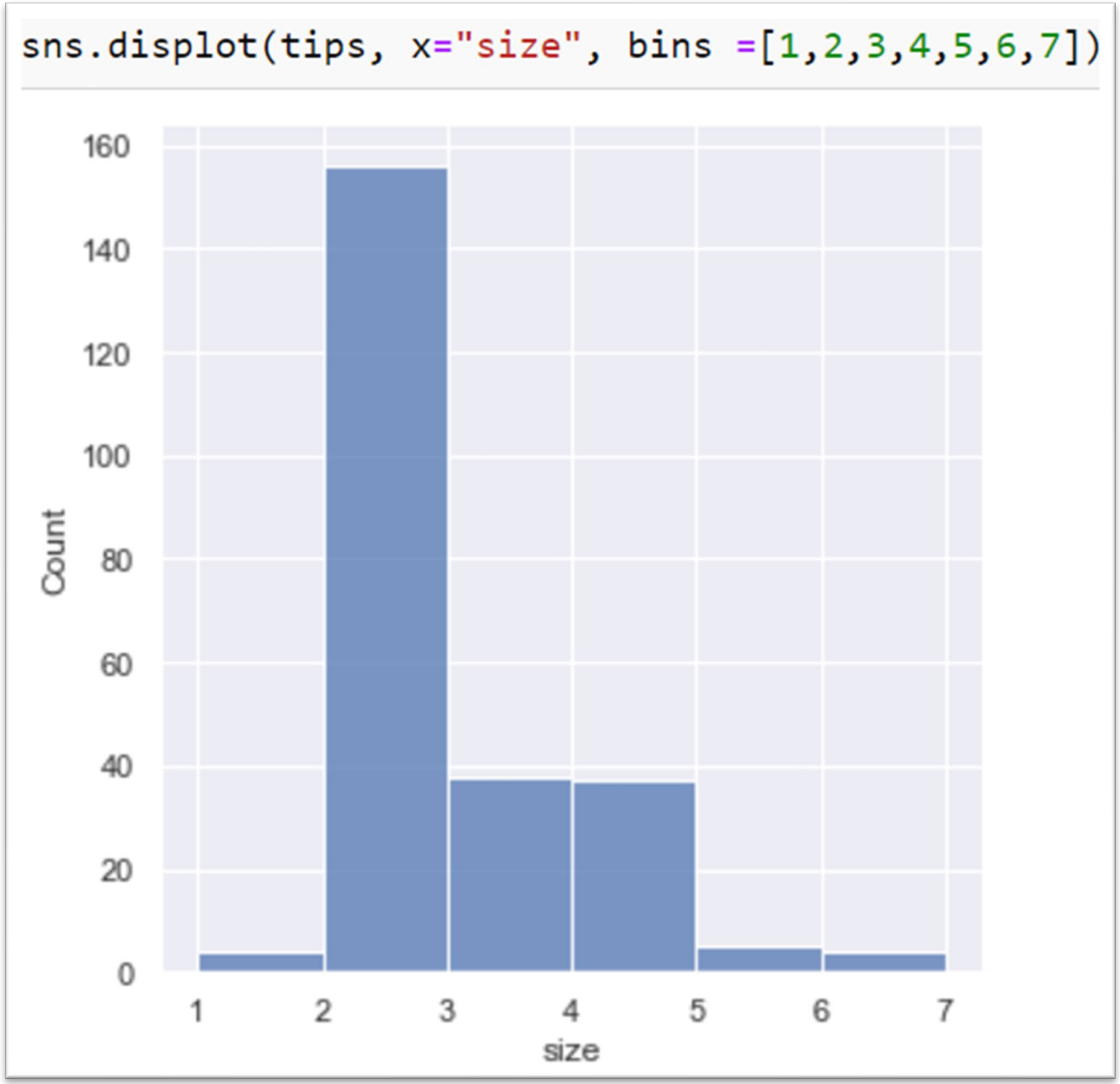
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3

```
sns.displot(tips, x="size")
```



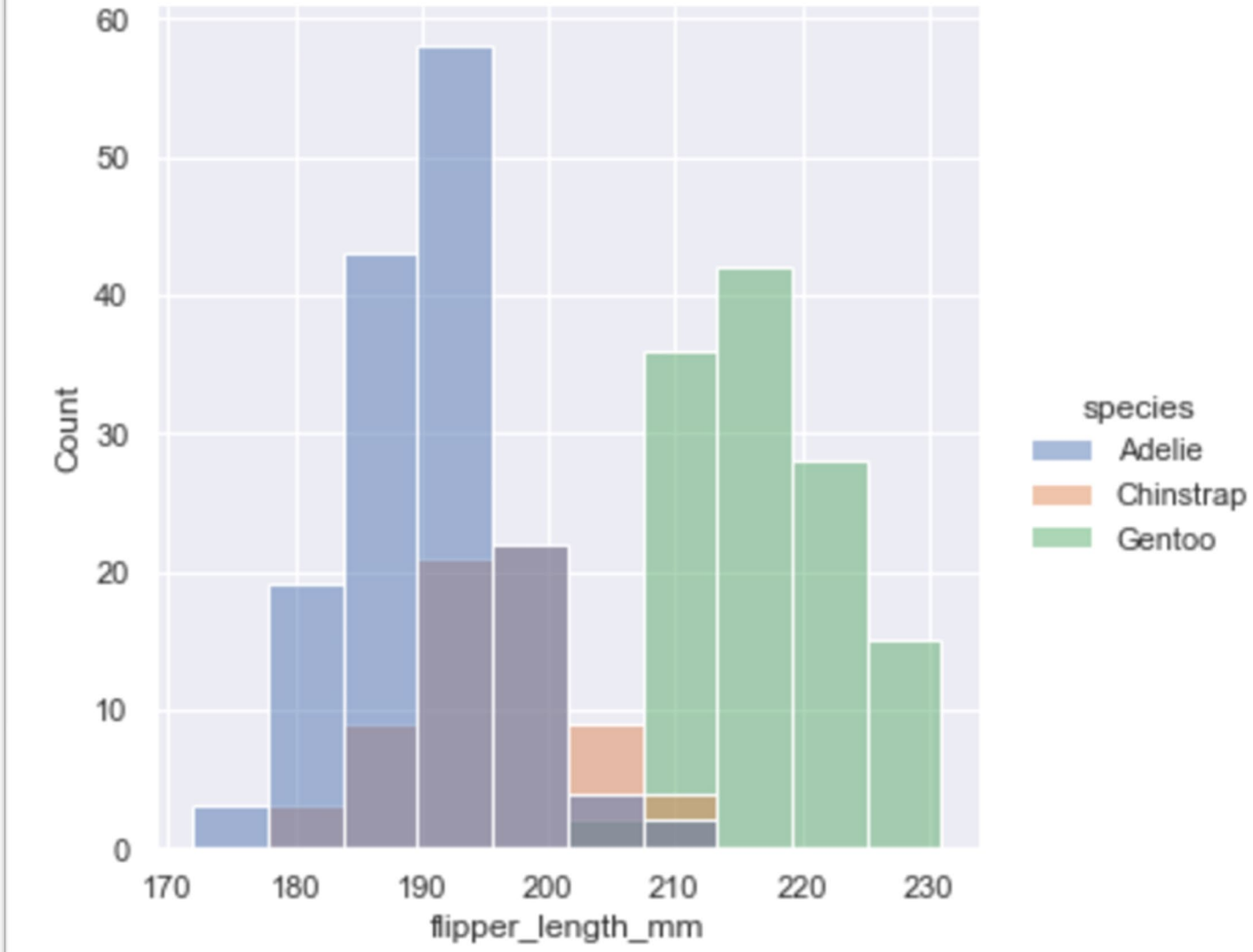
Passing an Array to Bins

```
sns.displot(tips, x="size", bins =[1,2,3,4,5,6,7])
```



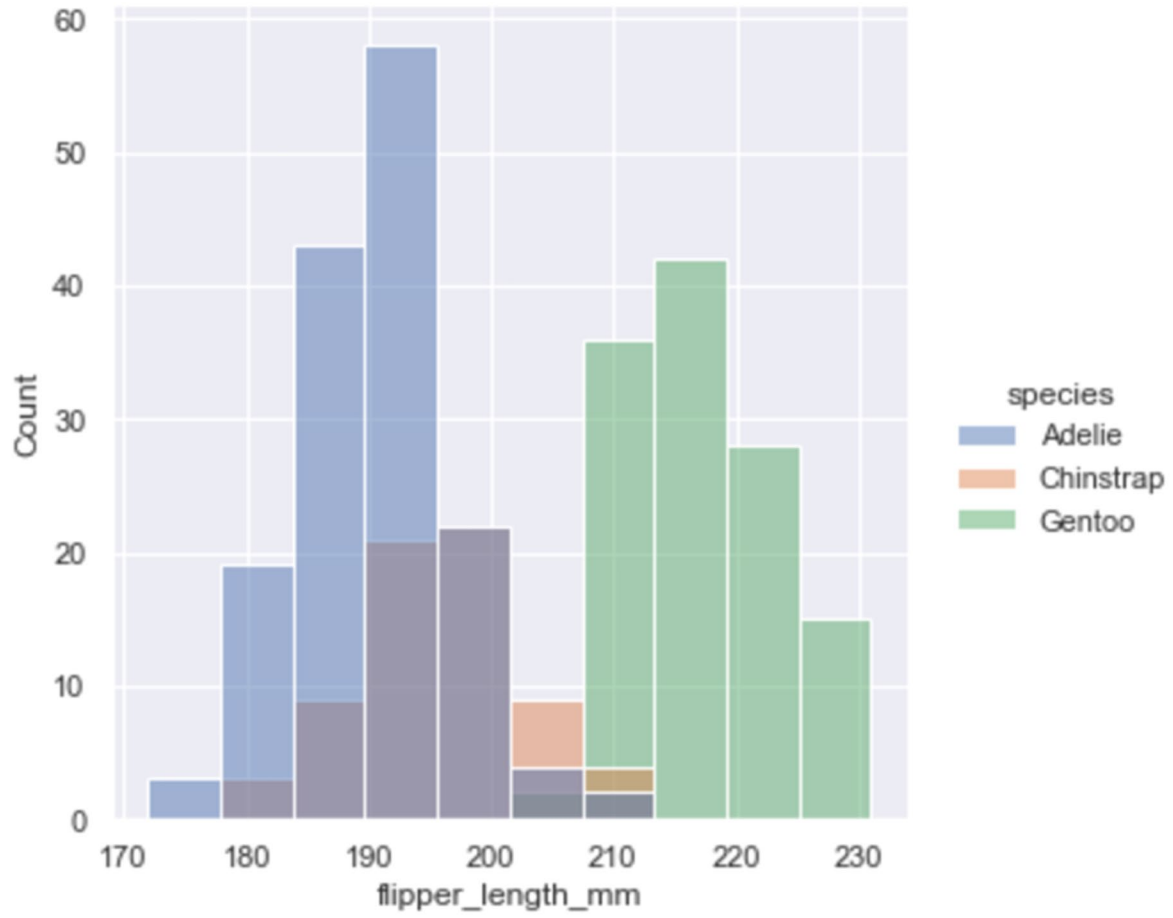
Histograms With 'hue'

```
sns.displot(penguins,x="flipper_length_mm",
            hue="species");
```

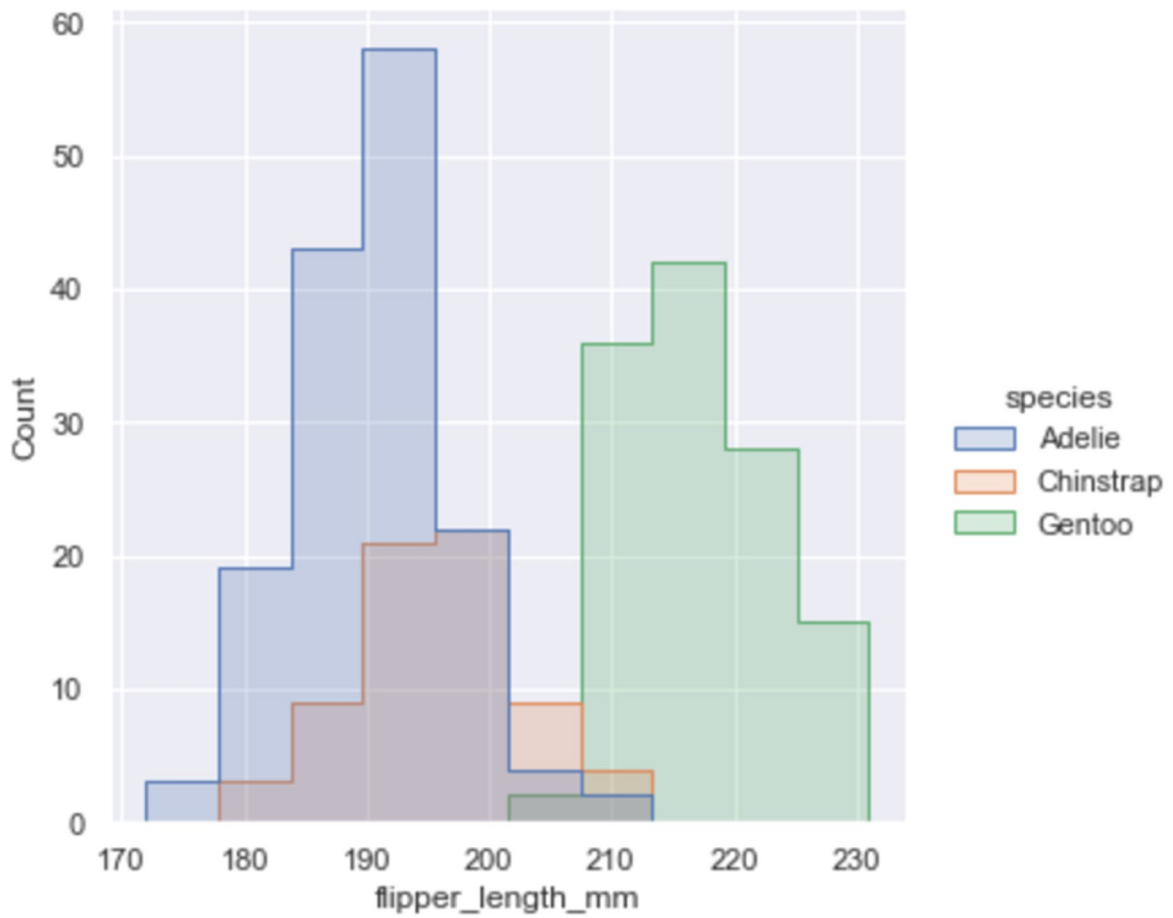


Histogram + 'element'

```
sns.displot(penguins,x="flipper_length_mm",
            hue="species");
```

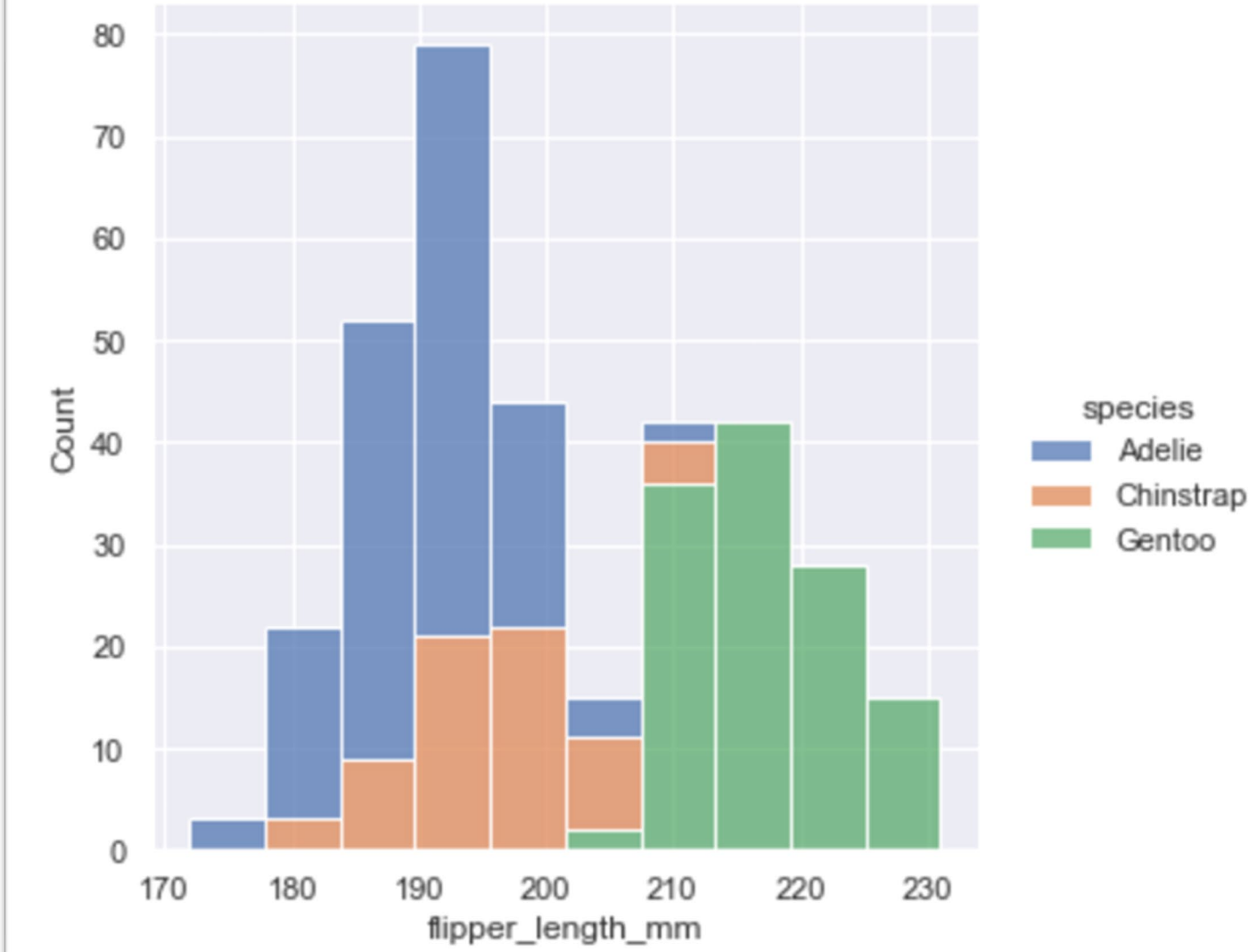



```
sns.displot(penguins,x="flipper_length_mm",
            hue="species", element ="step")
```



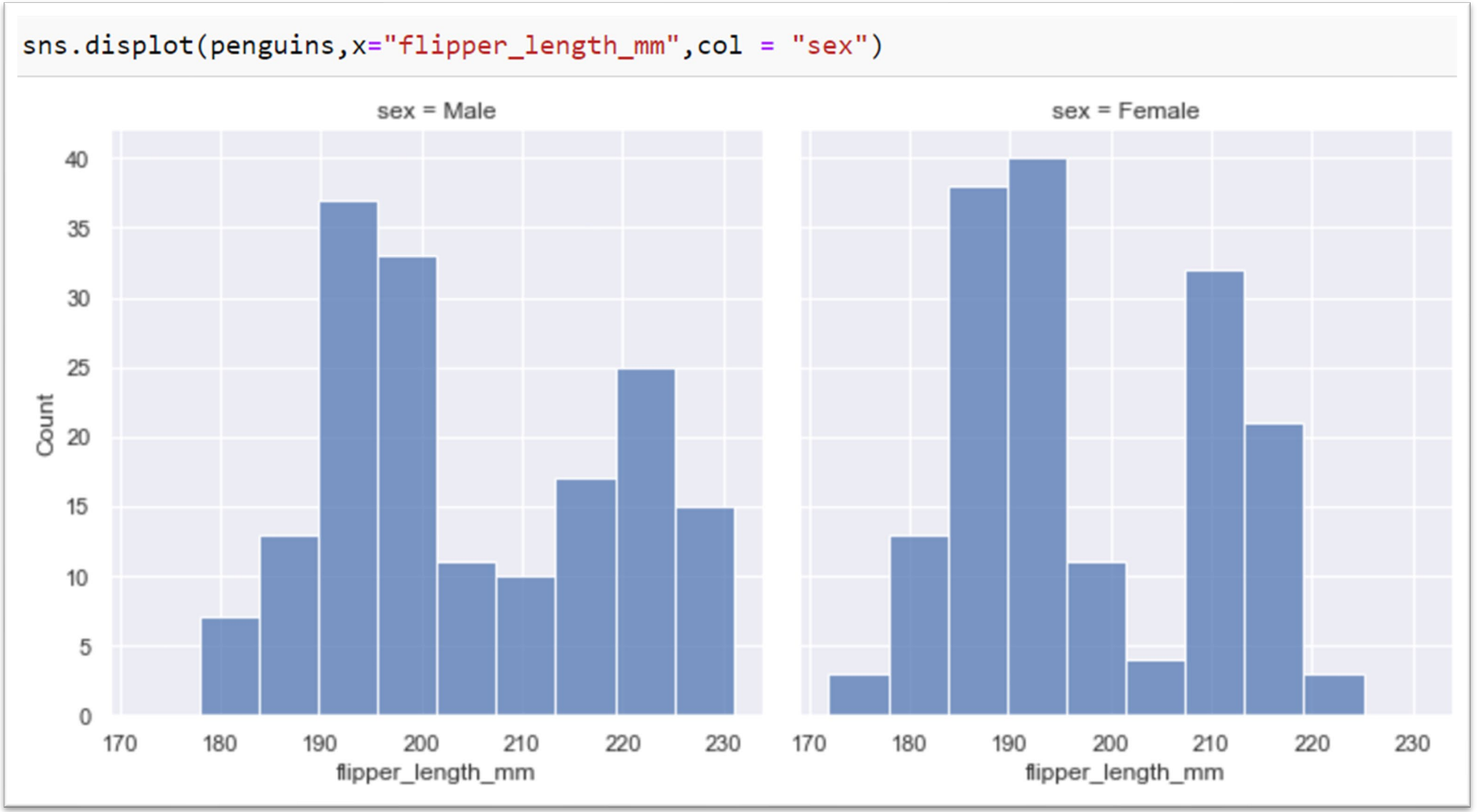
Stacking Histograms

```
sns.displot(penguins,x="flipper_length_mm",
            hue="species",  multiple="stack");
```



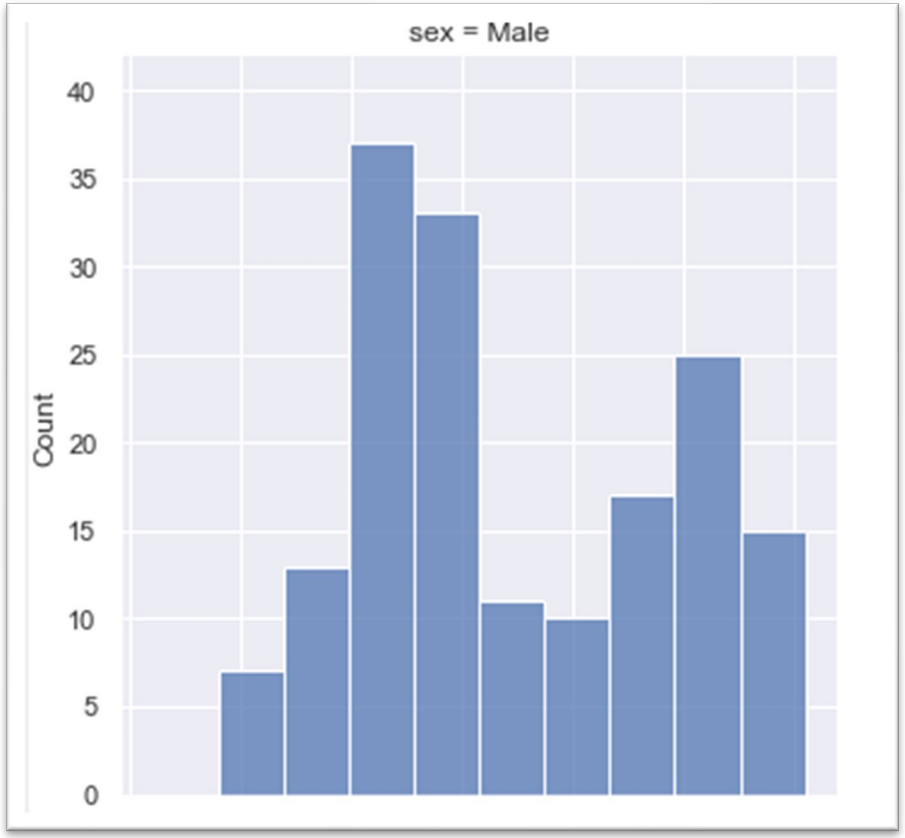
Multiple Plots: 'col'

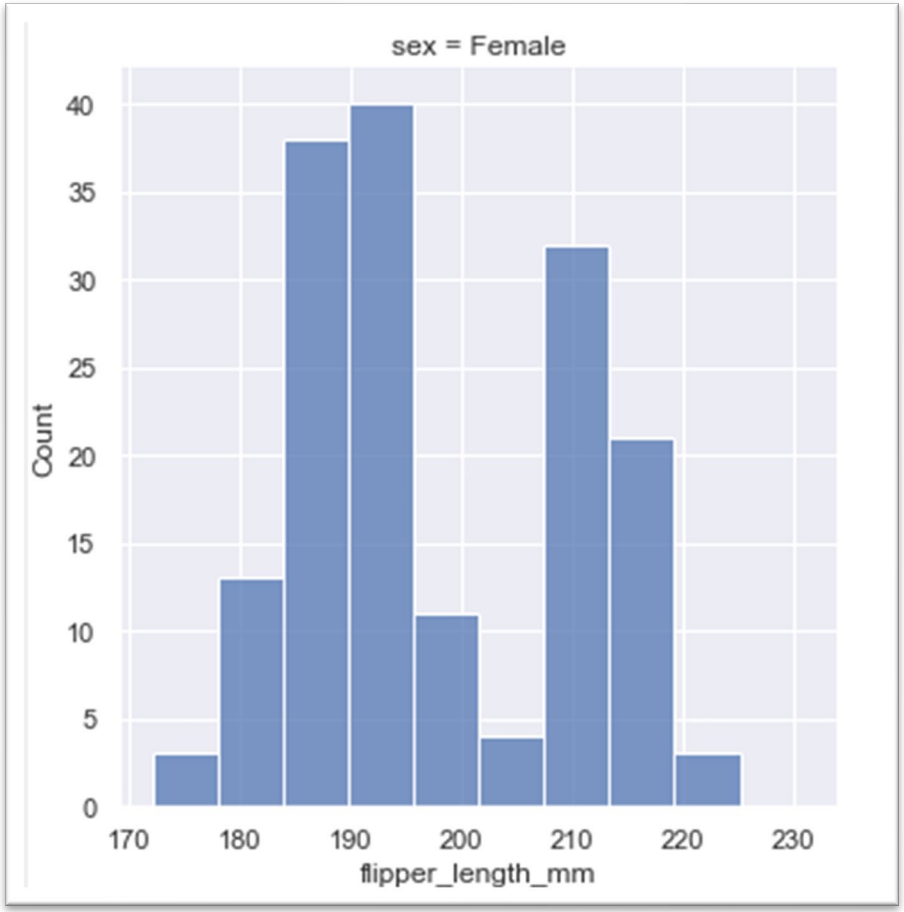
```
sns.displot(penguins,x="flipper_length_mm",col = "sex")
```



Multiple Plots: 'row'

```
sns.displot(penguins,x="flipper_length_mm",row="sex")
```



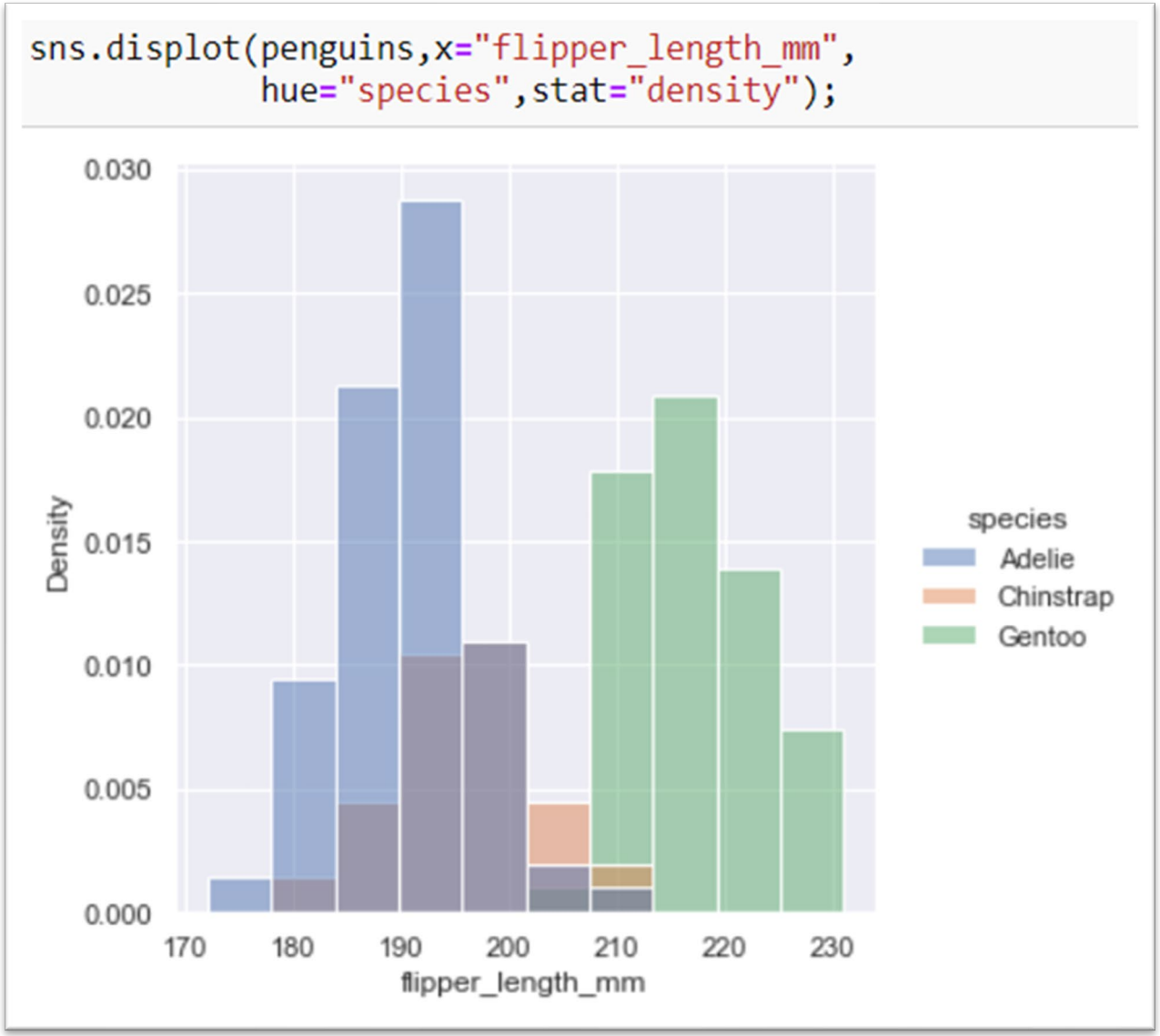


Normalizing Histograms

When the subsets of our datasets have unequal number of observations, comparing their distributions using counts may not be fair.

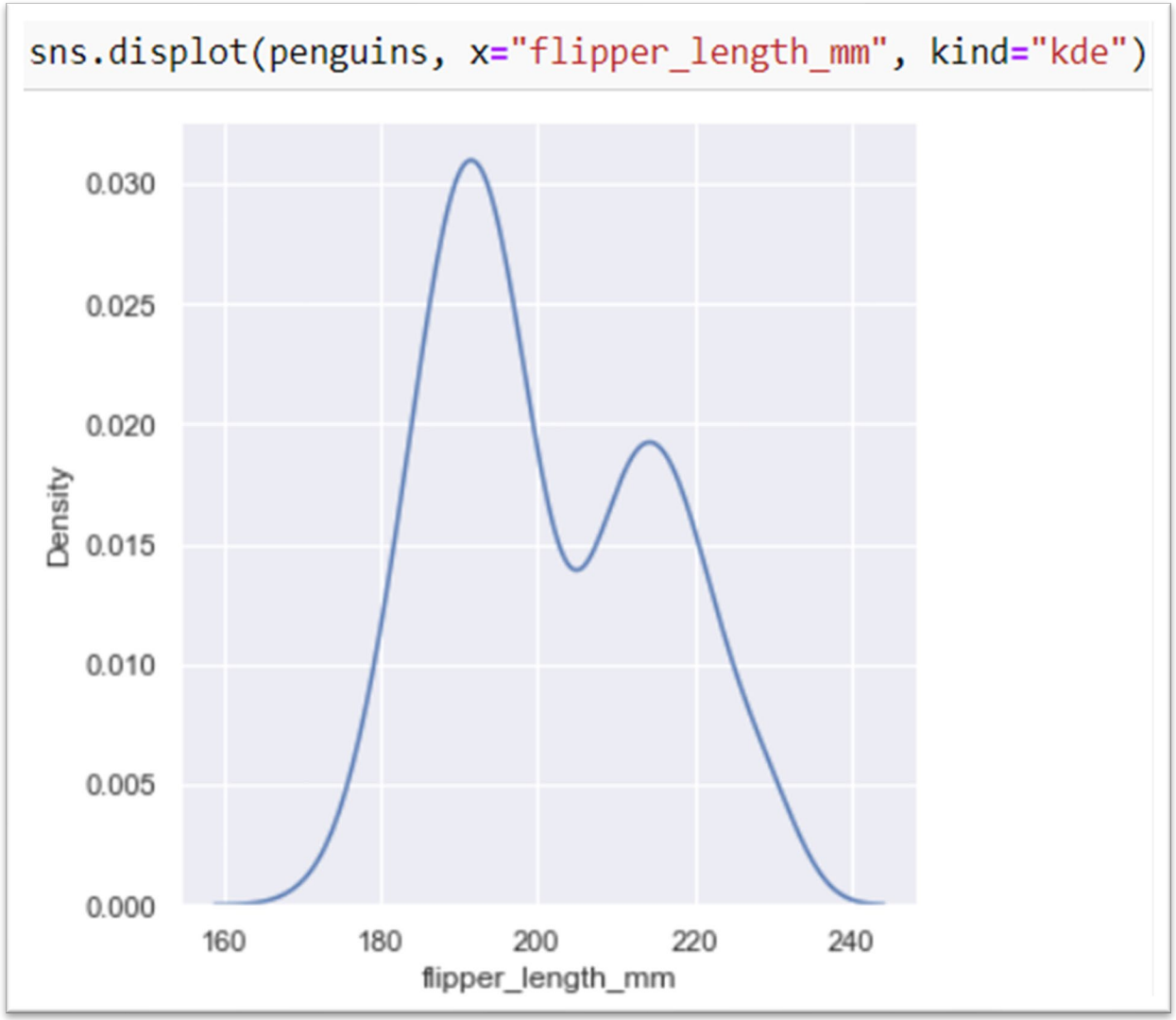
Therefore, we normalize data using 'stat' parameter.

```
sns.displot(penguins,x="flipper_length_mm",
            hue="species",stat="density");
```



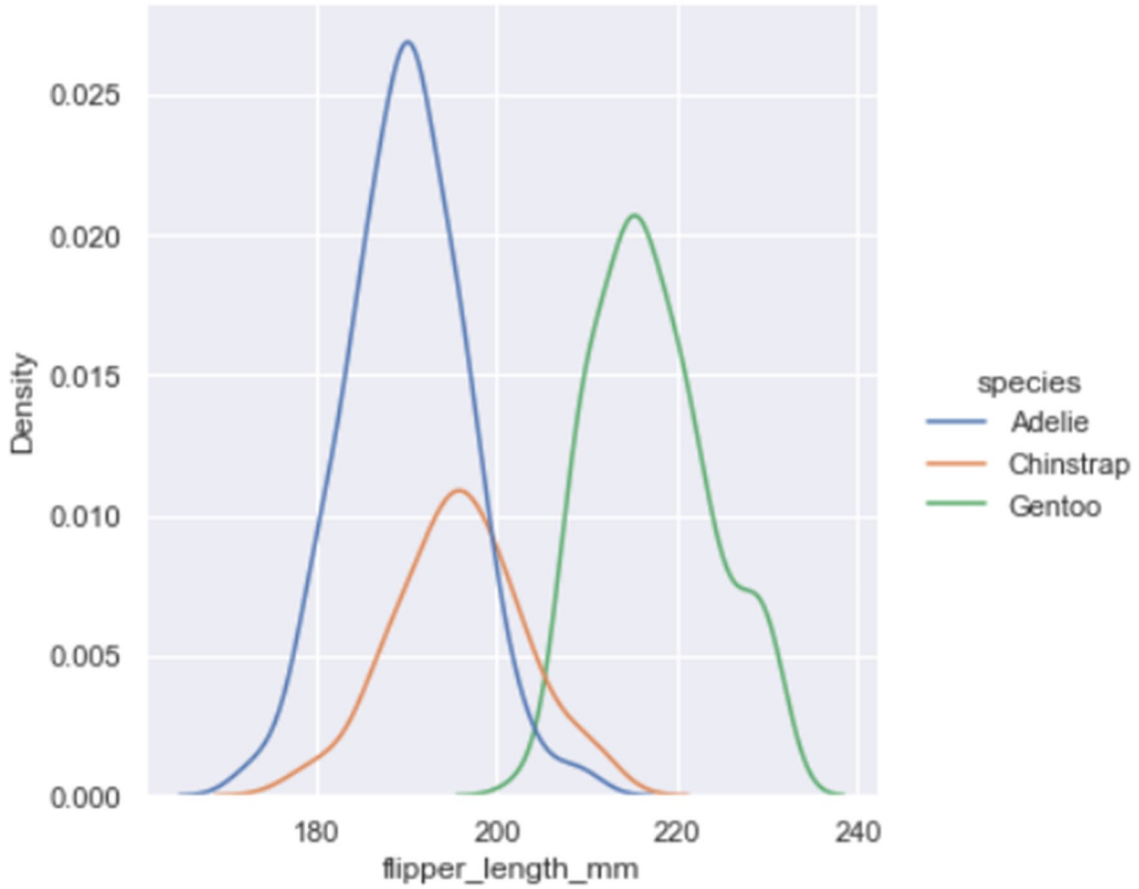
Kernel Density Estimation (KDE)

KDE graphs use curves to indicate the underlying distribution rather than using discrete bins.

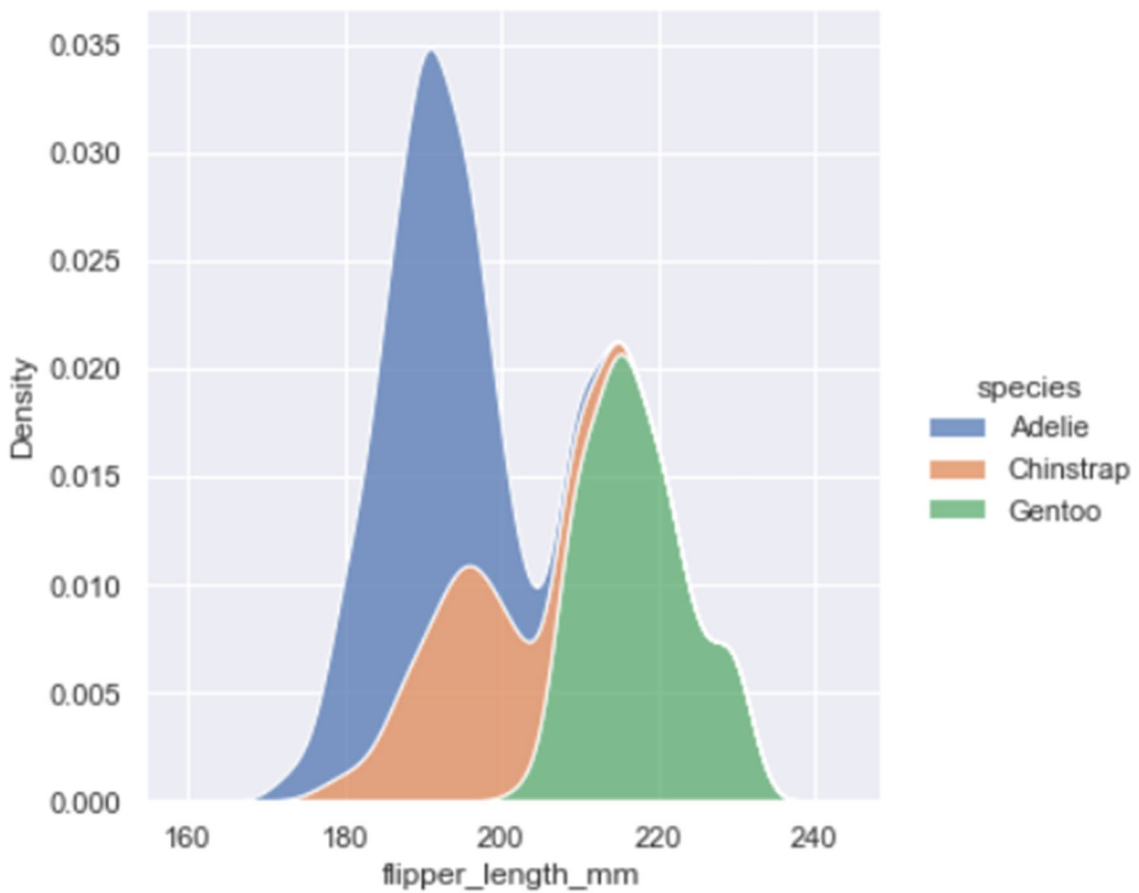


Conditioning Other Variables

```
sns.displot(penguins, x="flipper_length_mm",
            kind="kde", hue="species")
```

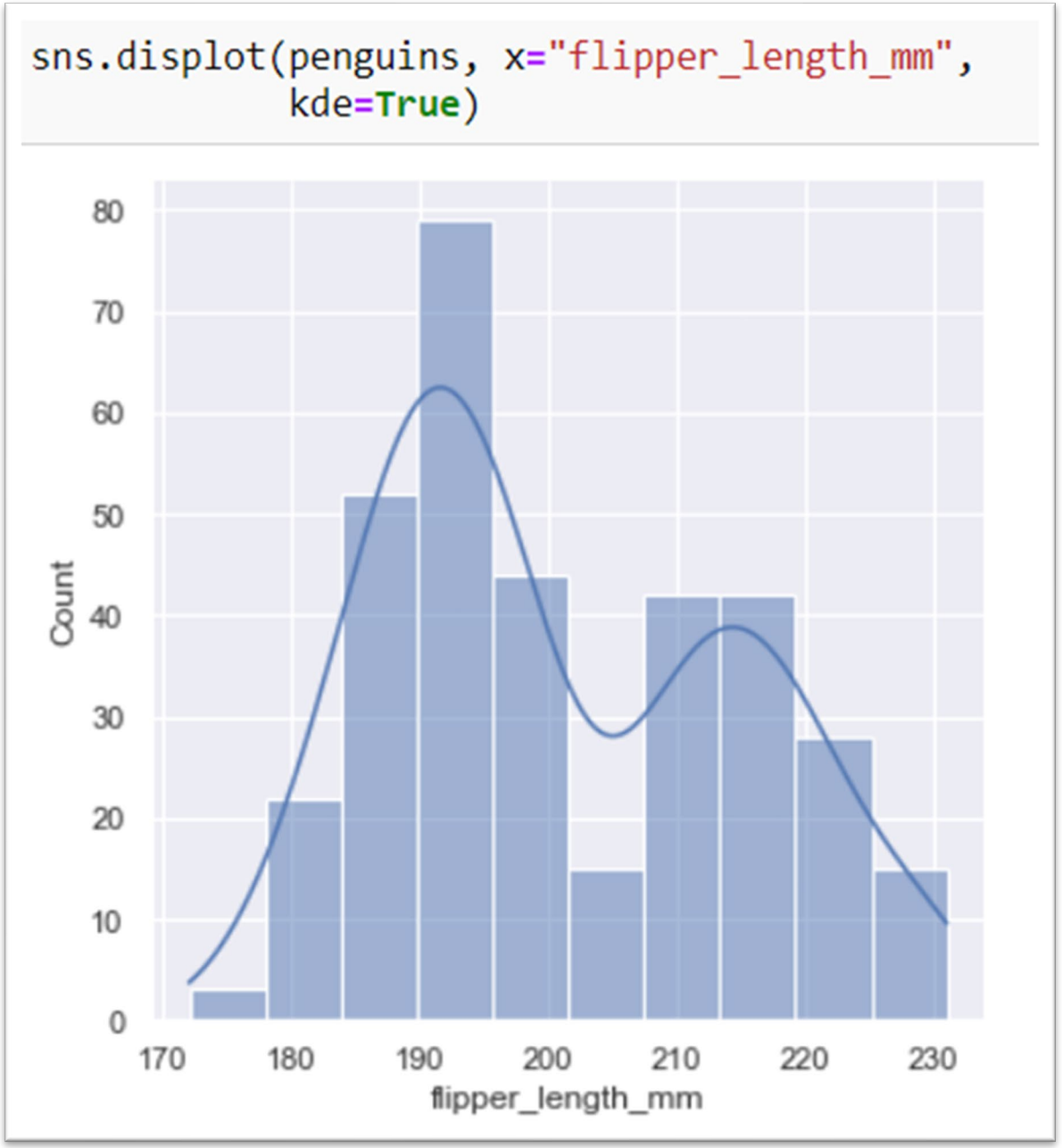



```
sns.displot(penguins, x="flipper_length_mm",
            kind="kde", hue="species", multiple = "stack")
```



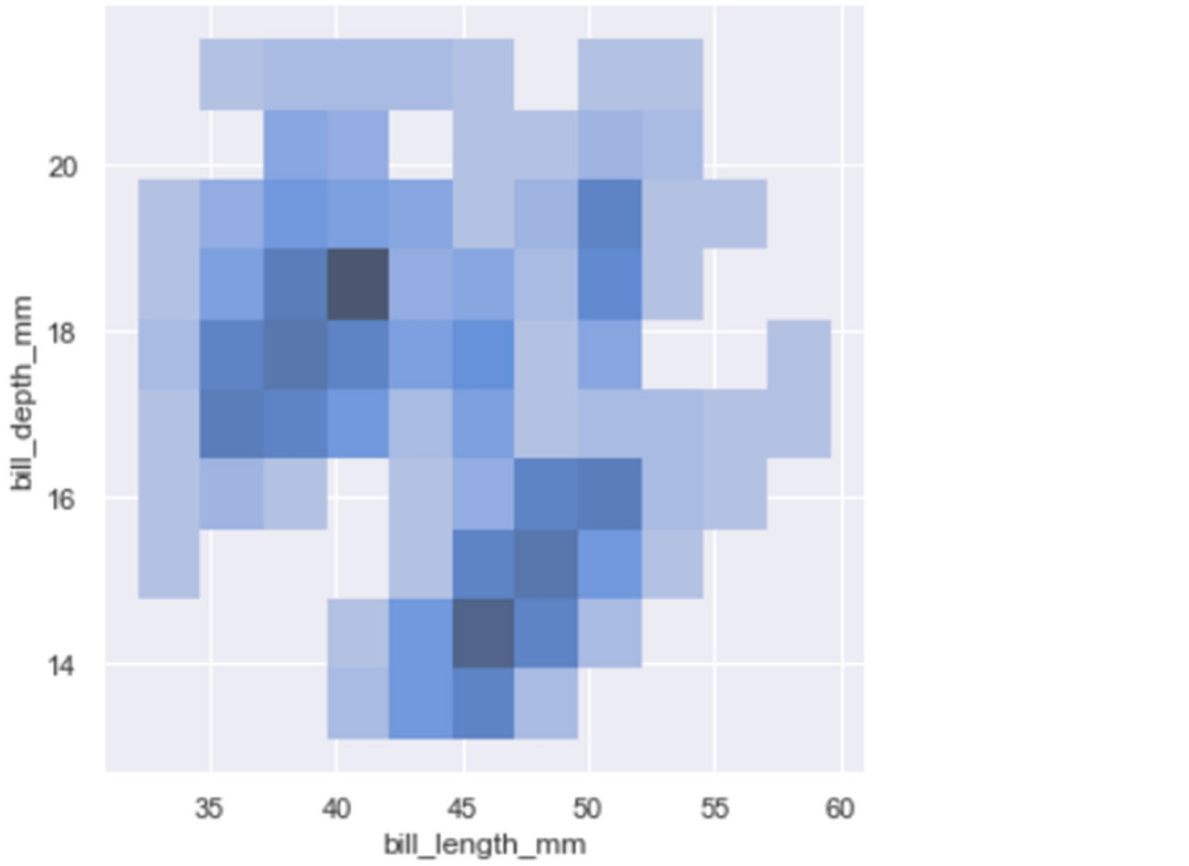
Overlaying 'kde' on Histograms

```
sns.displot(penguins, x="flipper_length_mm",
            kde=True)
```

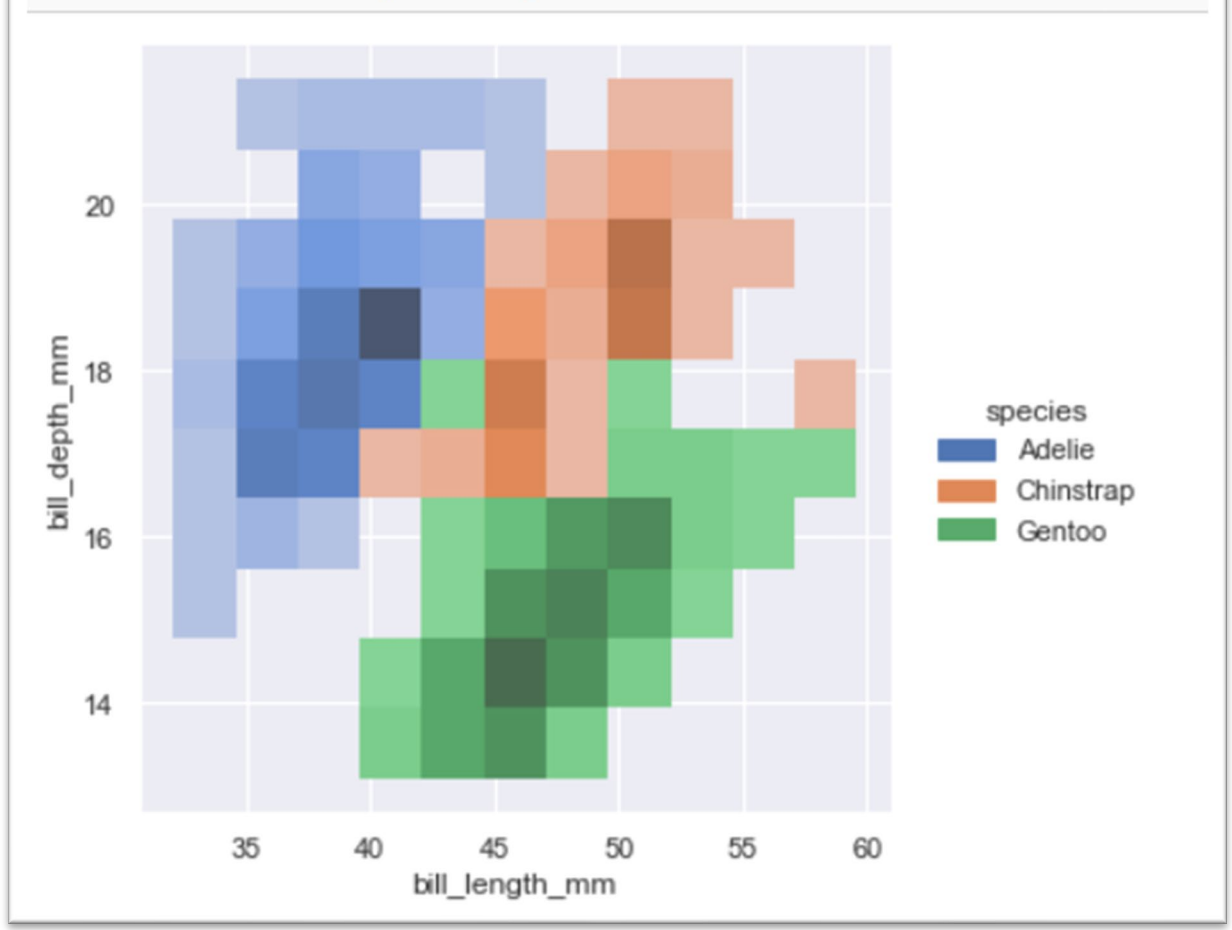


Visualizing Bivariate Distribution

```
sns.displot(penguins, x="bill_length_mm", y="bill_depth_mm")
```

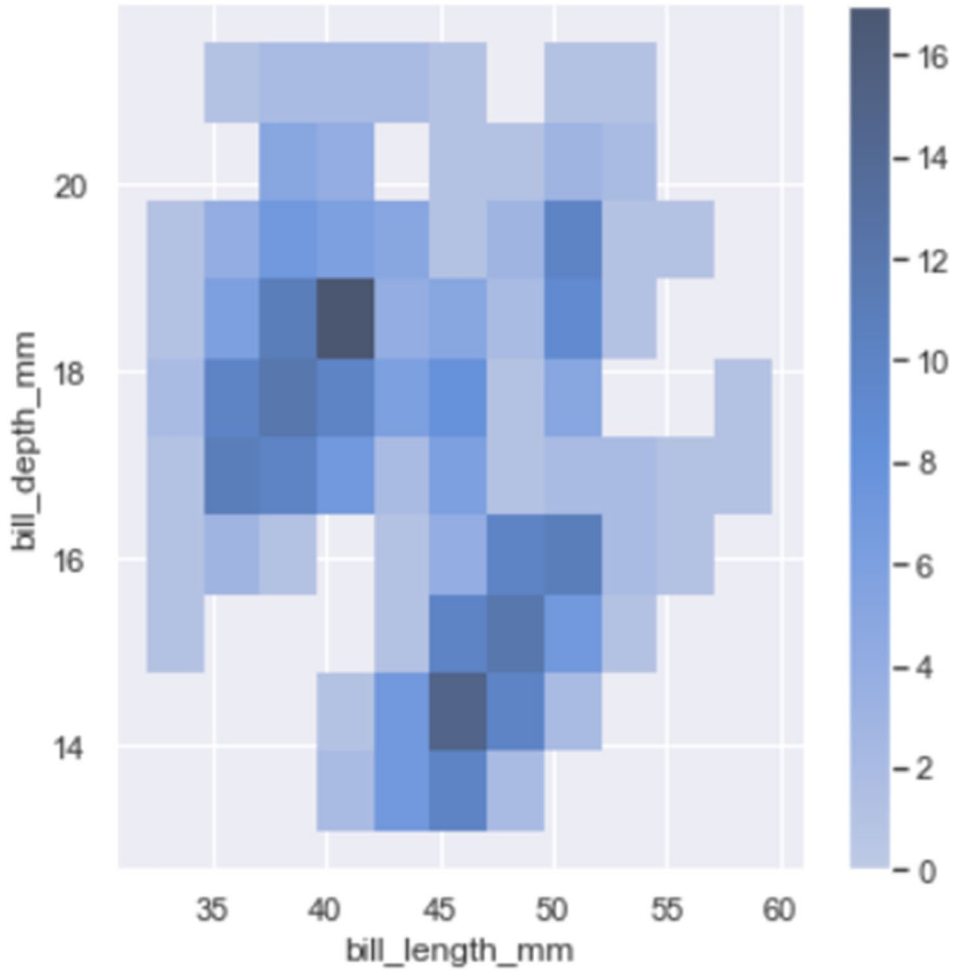



```
sns.displot(penguins, x="bill_length_mm", y="bill_depth_mm",
            hue="species")
```



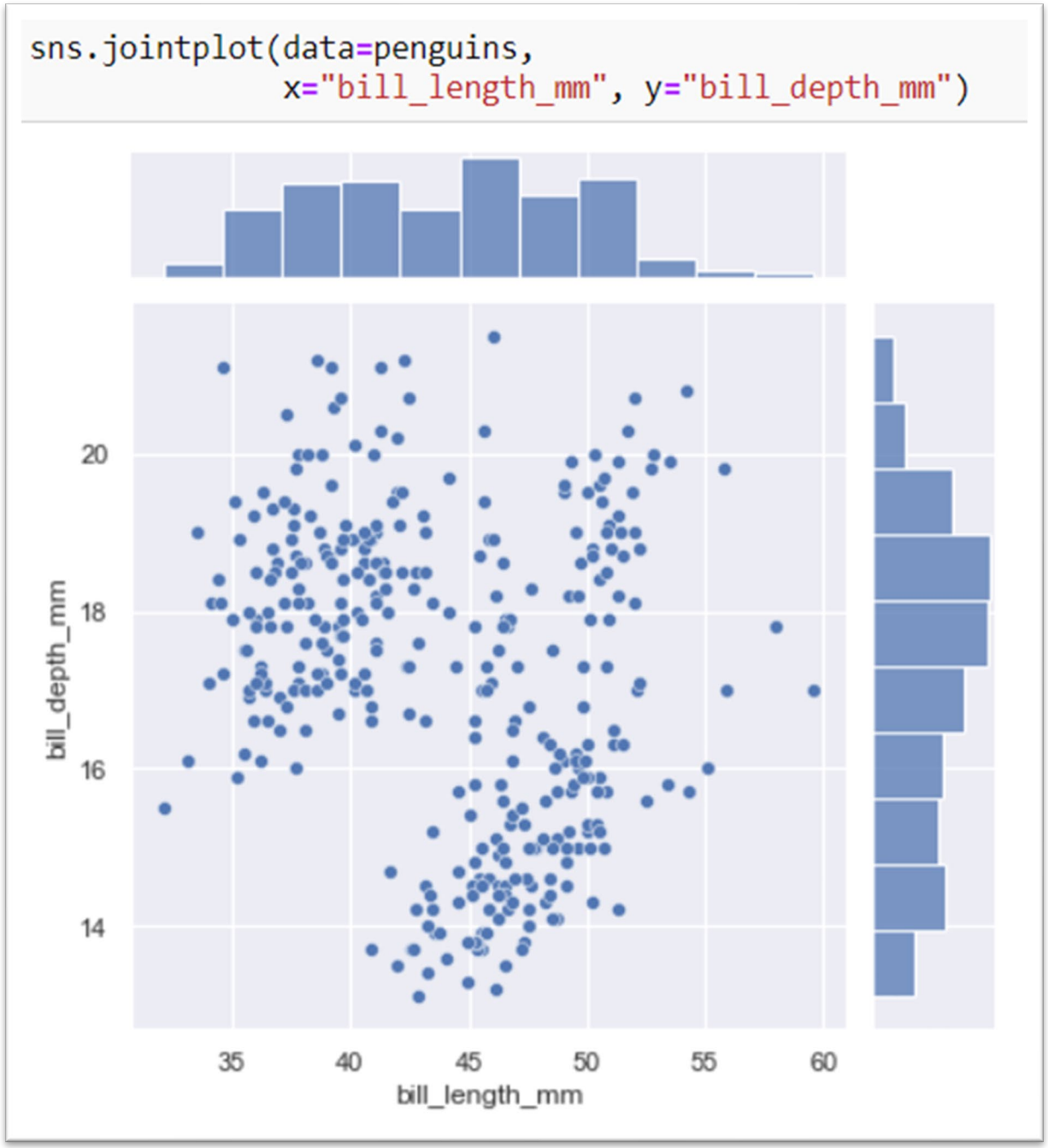
Bi-Variate Functions With 'cbar'

```
sns.displot(penguins, x="bill_length_mm", y="bill_depth_mm",
            cbar=True)
```



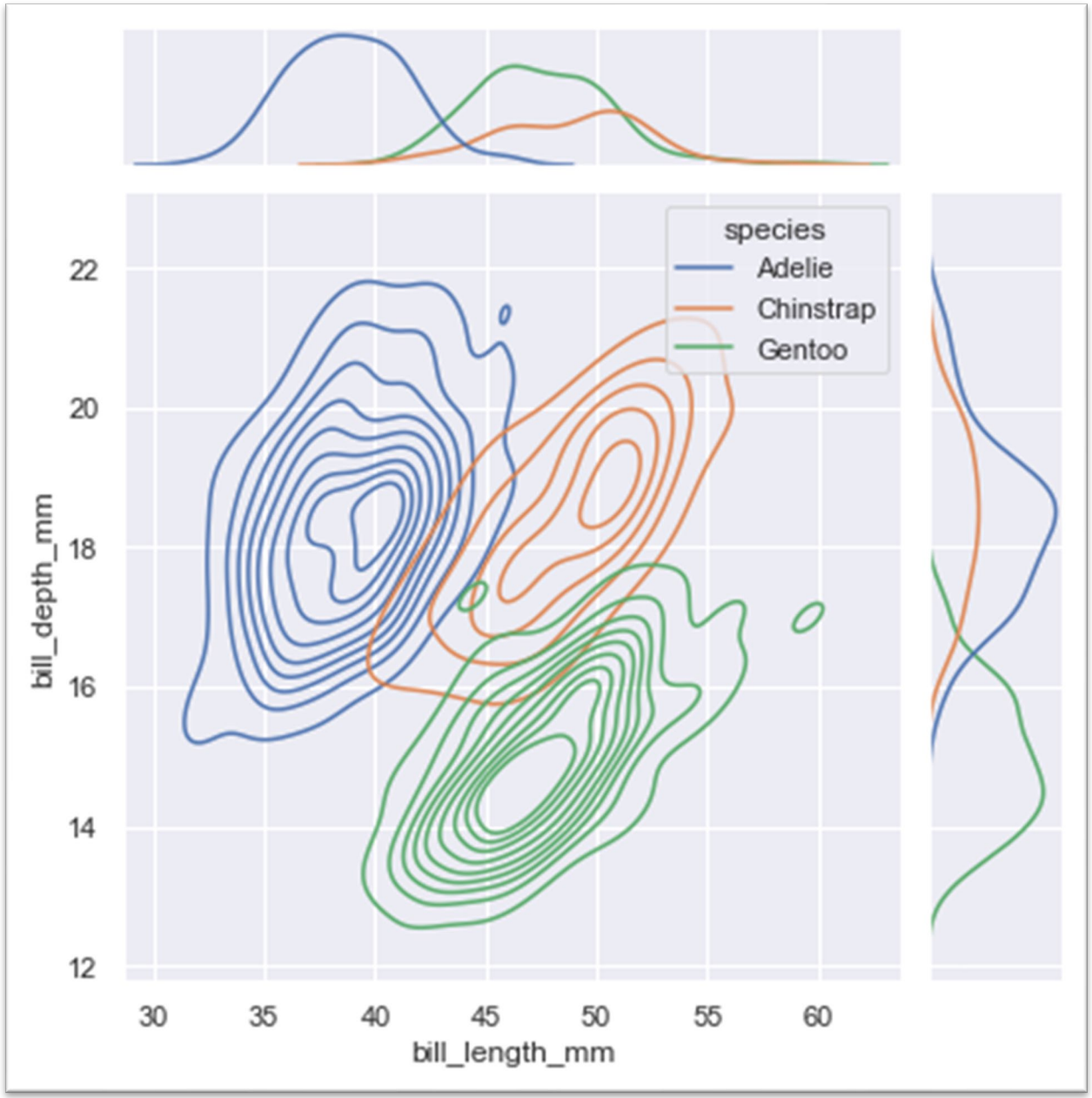
Plotting Joint and Marginal Distributions

```
sns.jointplot(data=penguins,
              x="bill_length_mm", y="bill_depth_mm")
```



Joint Plots With 'hue' and 'kde'

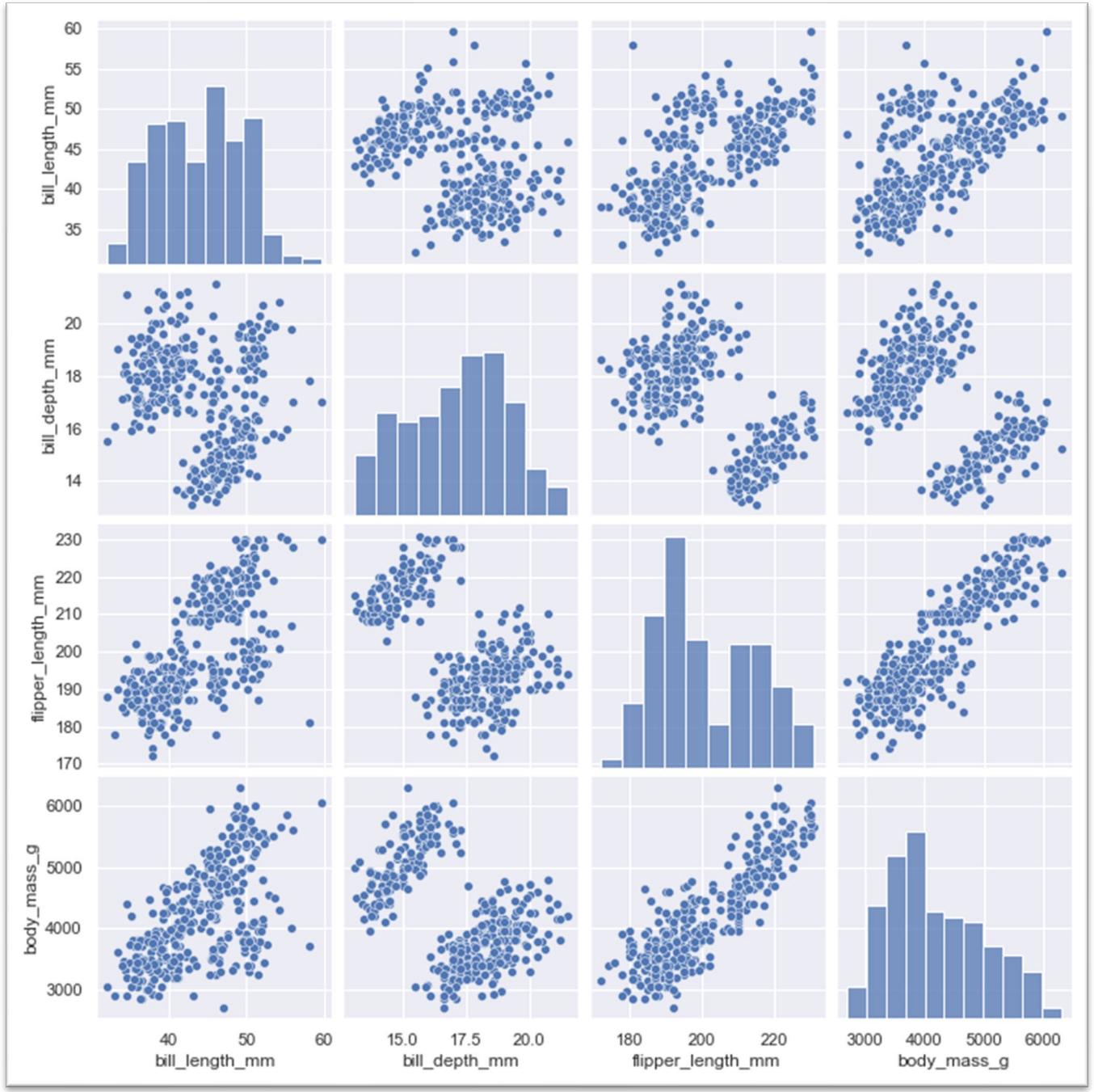
```
sns.jointplot(data=penguins,
              x="bill_length_mm", y="bill_depth_mm",
             hue = "species", kind = "kde")
```



Pairplots

'**pairplot()**' visualizes the univariate distribution of all variables in a dataset along with their pairwise relationship.

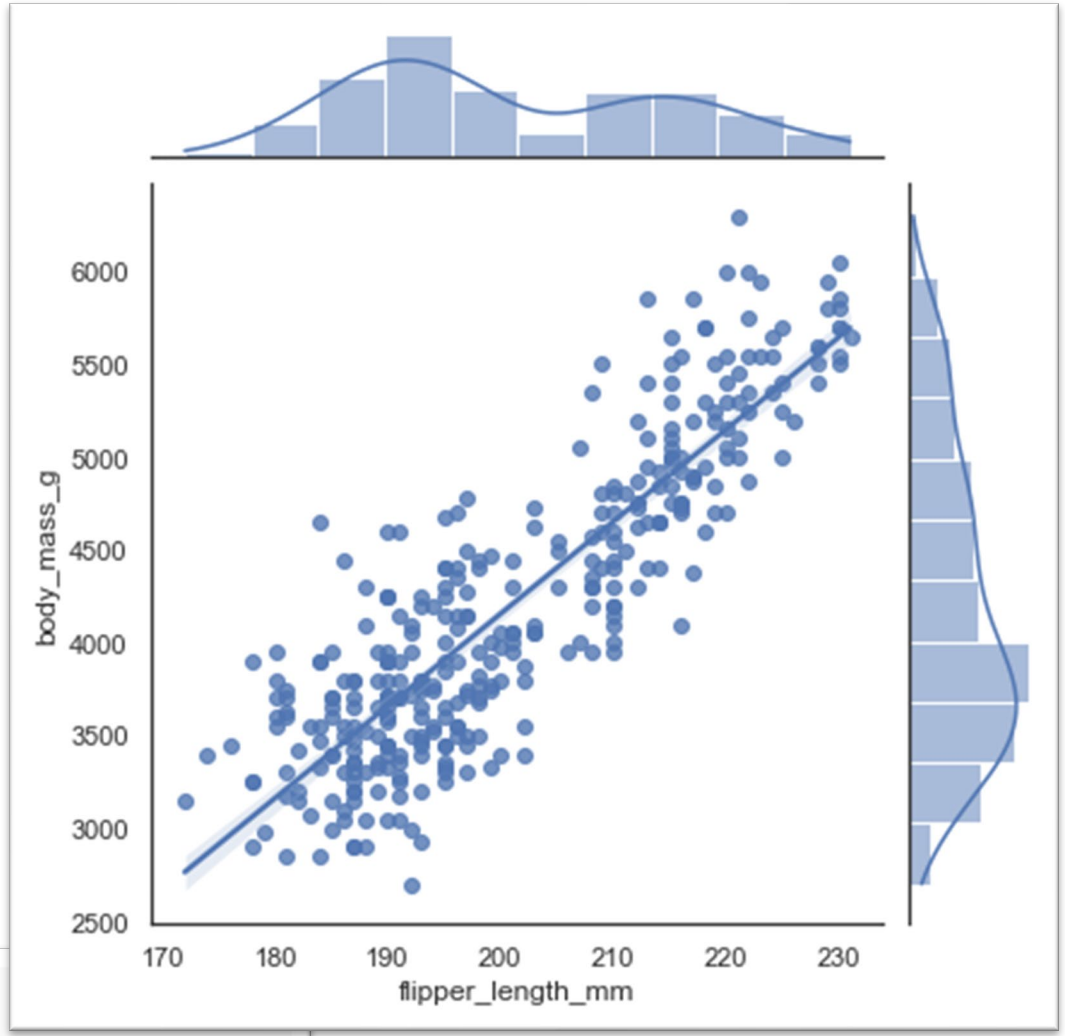
```
sns.pairplot(data=penguins)
```



Styling Plots

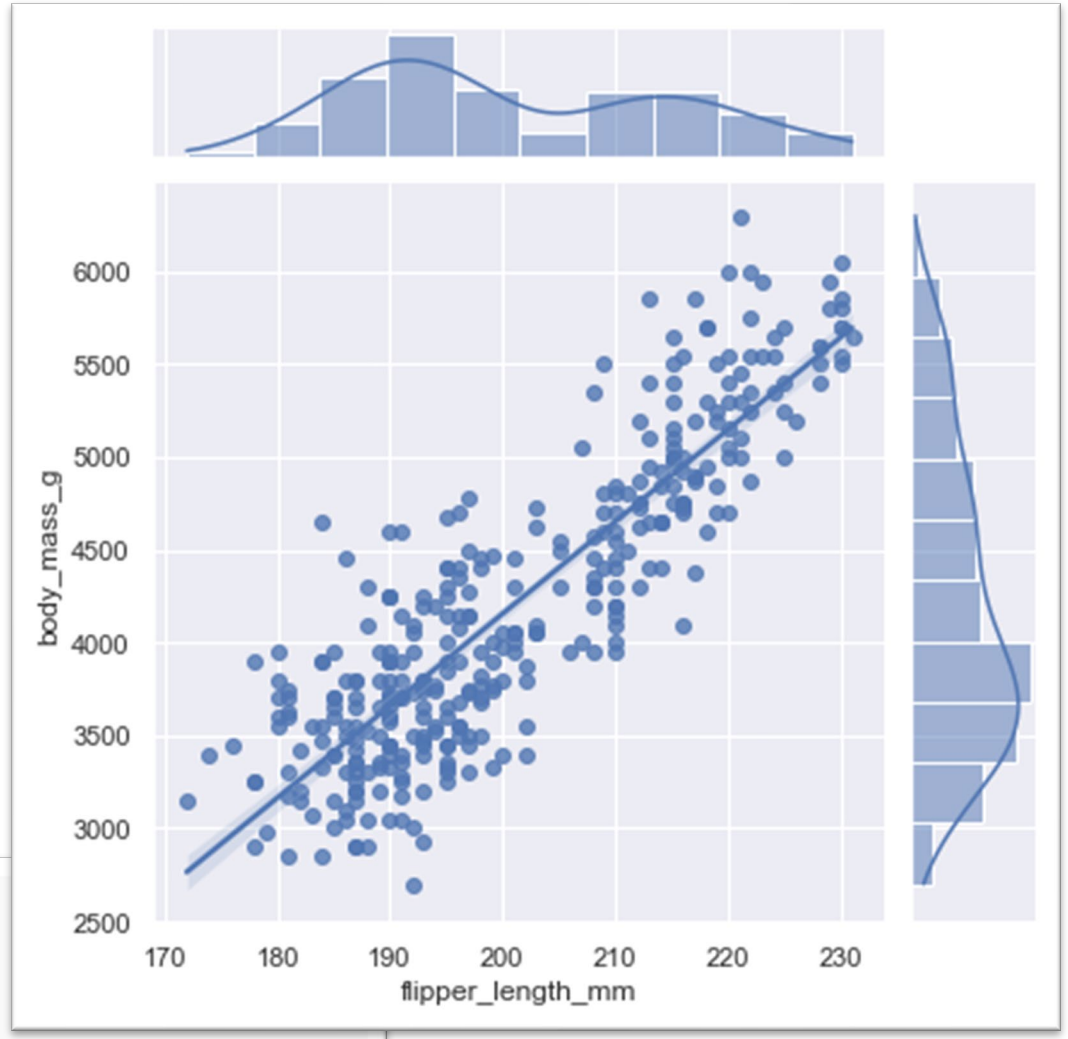
The arguments for '**set_style()**' are:

- darkgrid
- whitegrid
- dark
- white




```
sns.set_style('white')
sns.jointplot(data=penguins,
              x="flipper_length_mm", y="body_mass_g", kind='reg')
```

Style With 'darkgrid'



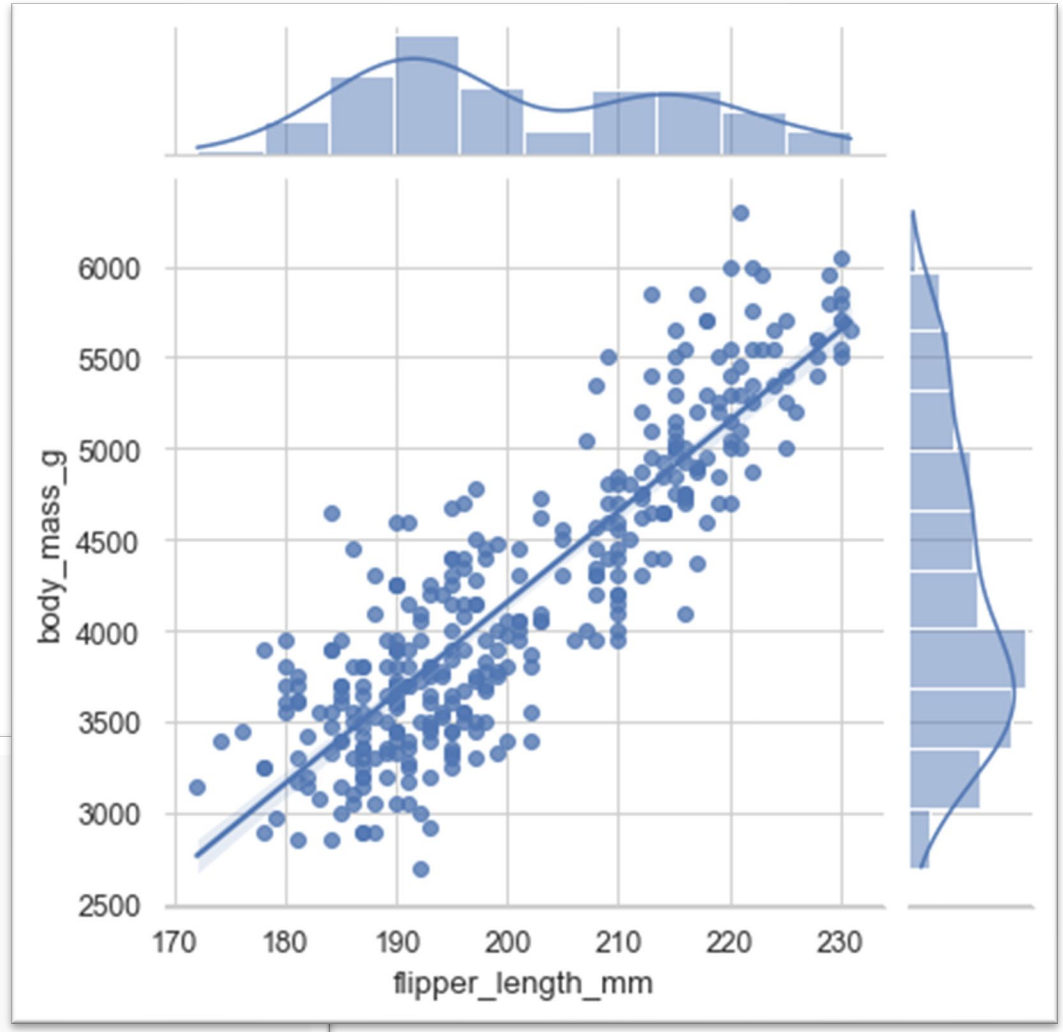

```
sns.set_style('darkgrid')

plt.figure(figsize=(2,1))
sns.jointplot(data=penguins,
              x="flipper_length_mm", y="body_mass_g", kind='reg')
```

Seaborn Context

Context can be:

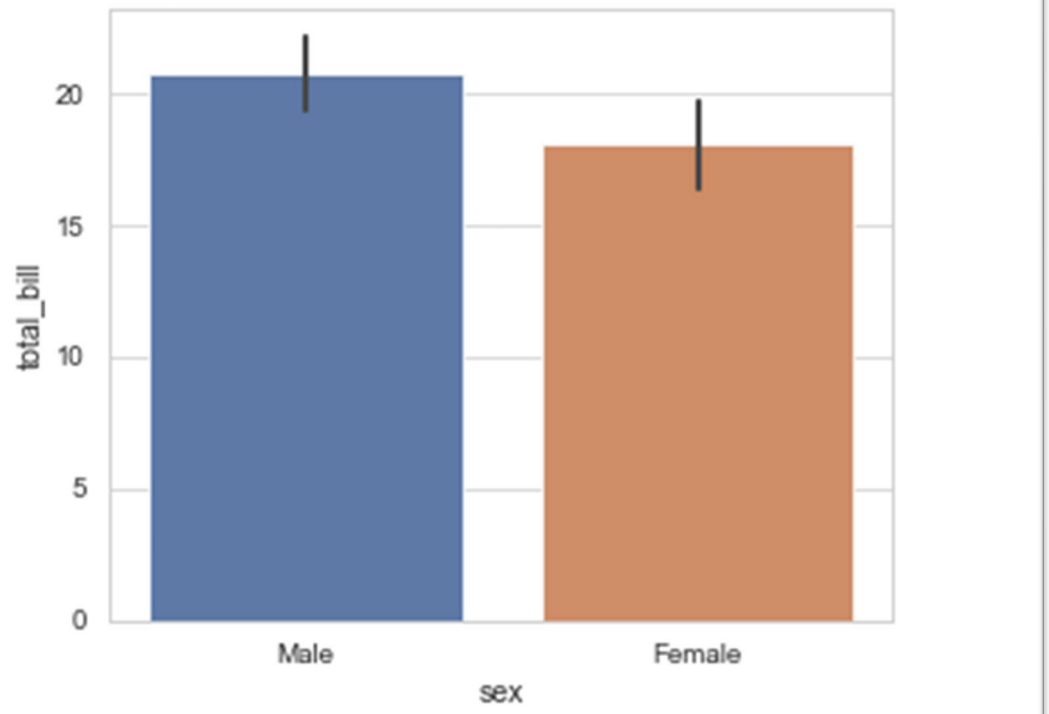
- paper
- notebook
- talk
- poster

```
sns.set_style('whitegrid')

plt.figure(figsize=(2,1))

sns.set_context('paper',font_scale=1.1)

sns.jointplot(data=penguins,
              x="flipper_length_mm", y="body_mass_g", kind='reg')

sns.despine (left=True)
```

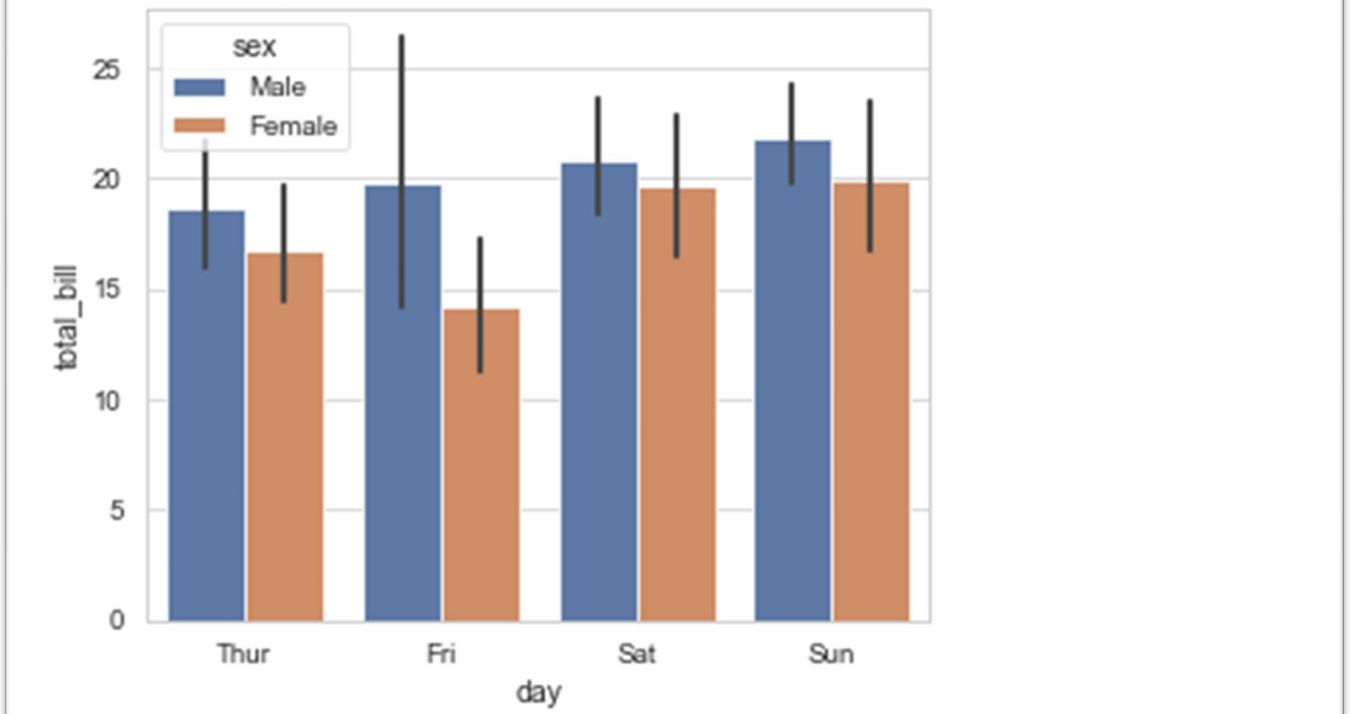


Seaborn for Categorical Variables: Bar Plots

```
sns.barplot(data = tips, x='sex',y='total_bill')
```




```
sns.barplot(data = tips, x='day', y='total_bill', hue='sex')
```



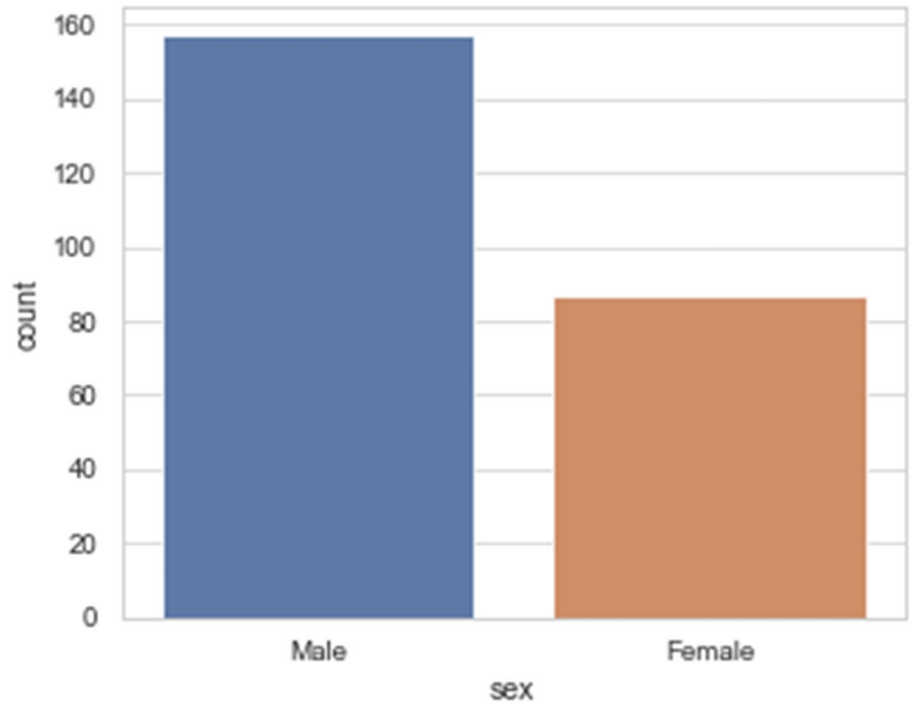
Bar Plots With 'estimator'

```
sns.barplot(data = tips, x='sex',y='total_bill', estimator=np.std)
```

Seaborn for Categorical Variables

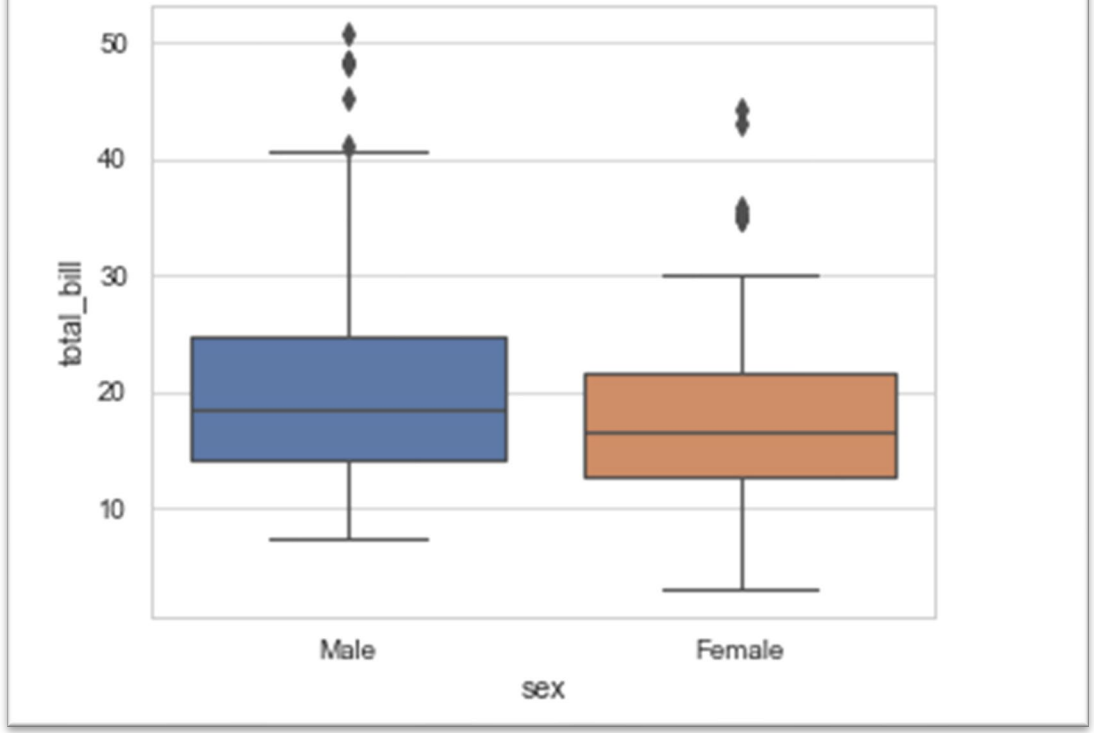
Count Plots

```
sns.countplot(data=tips, x='sex')
```



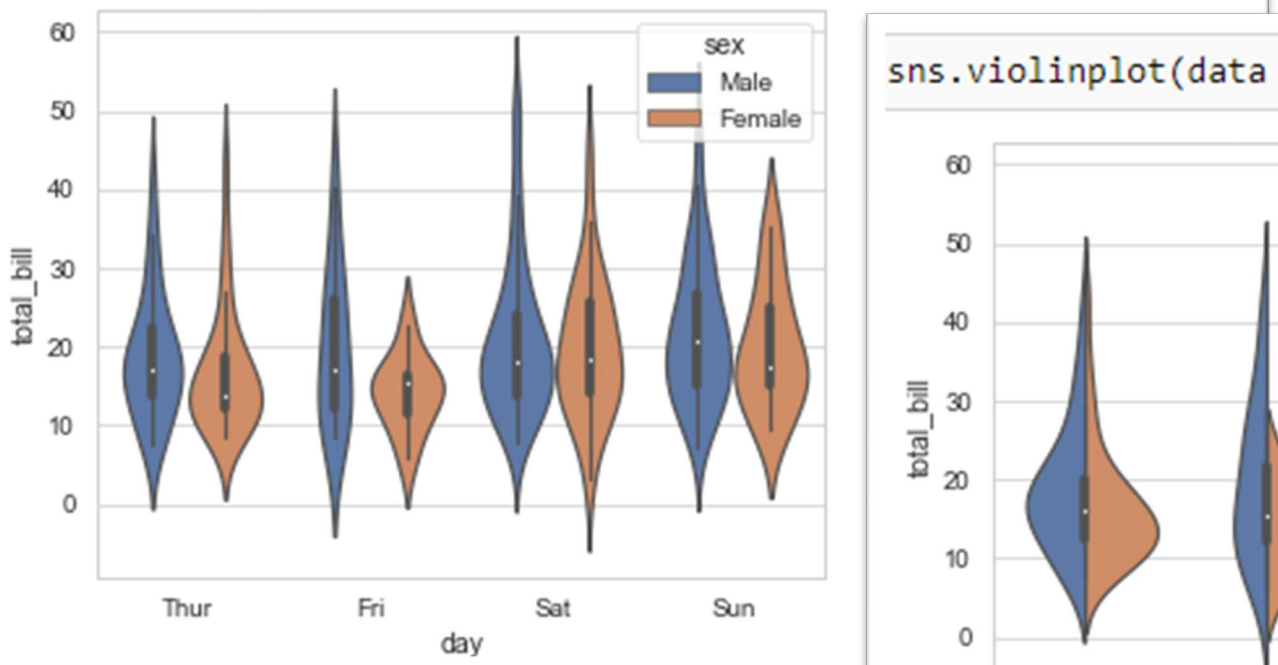
Box Plots

```
sns.boxplot(data = tips, x='sex',y='total_bill')
```

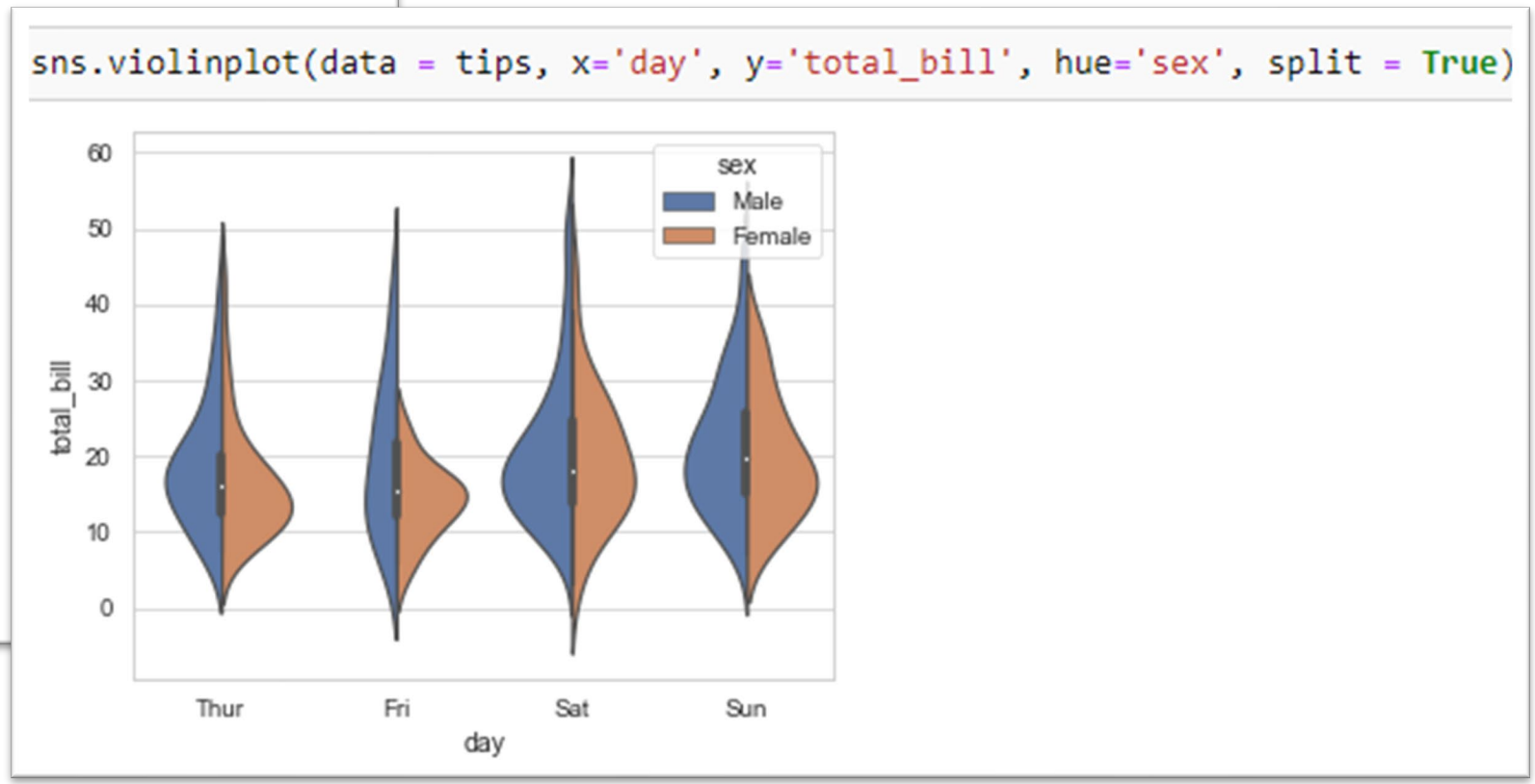


Violin Plots

```
sns.violinplot(data = tips, x='day', y='total_bill', hue='sex')
```

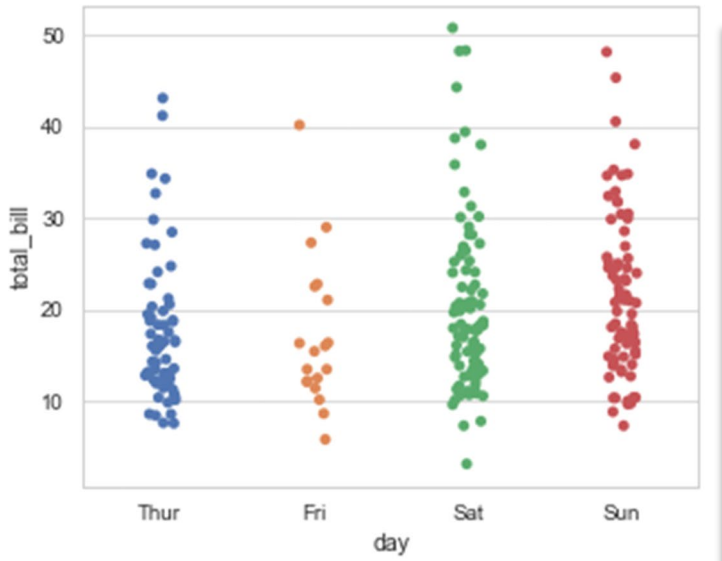



```
sns.violinplot(data = tips, x='day', y='total_bill', hue='sex', split = True)
```

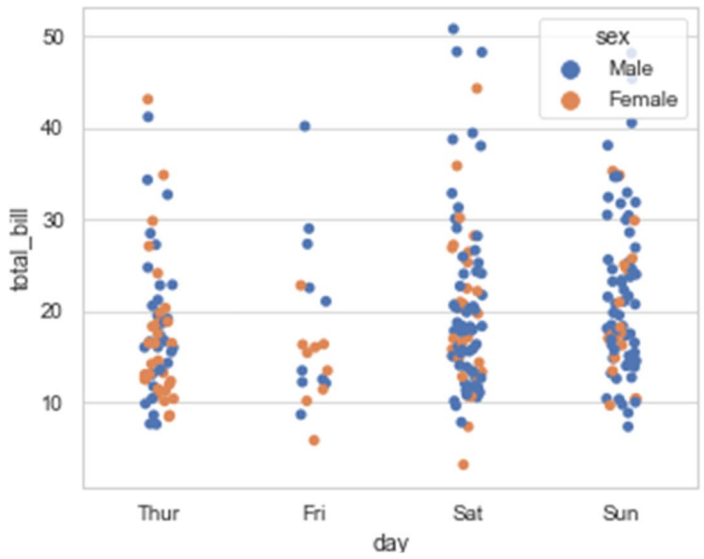


Strip Plots

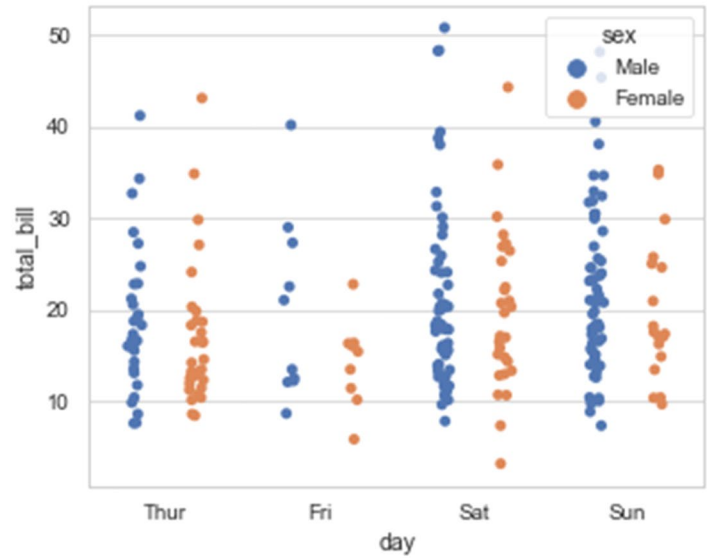
```
sns.stripplot(data = tips, x='day', y='total_bill')
```




```
sns.stripplot(data = tips, x='day', y='total_bill', hue='sex')
```

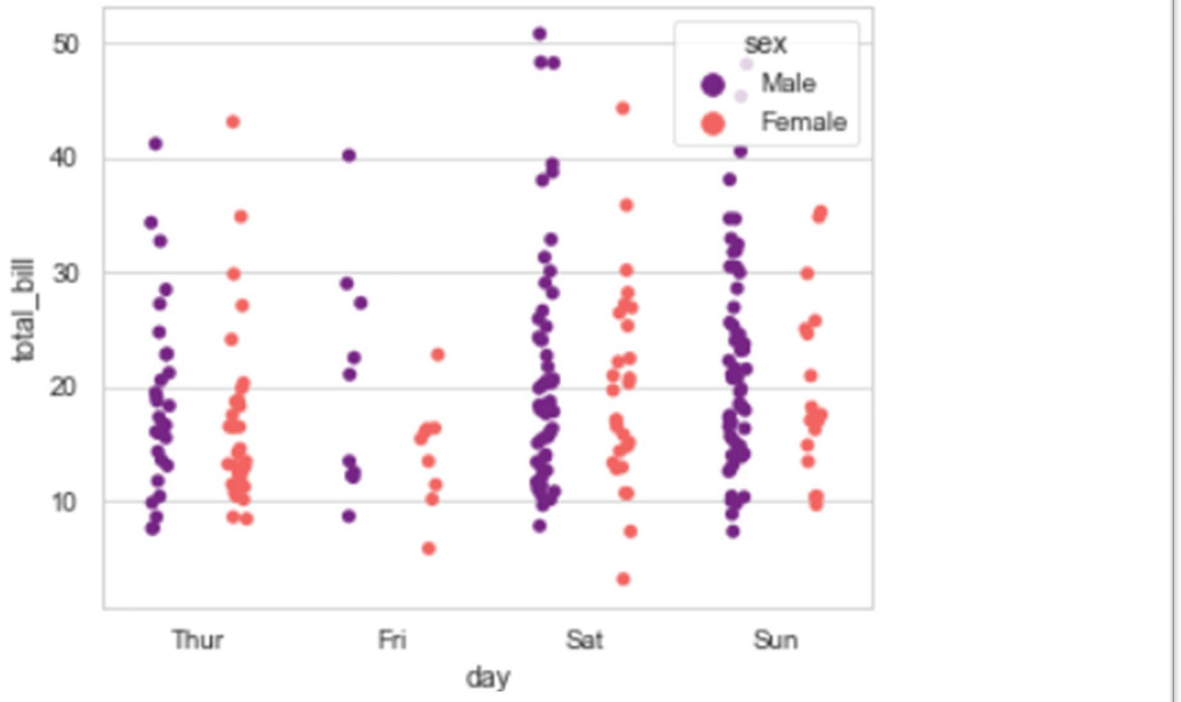



```
sns.stripplot(data = tips, x='day', y='total_bill',
              hue='sex', dodge=True);
```

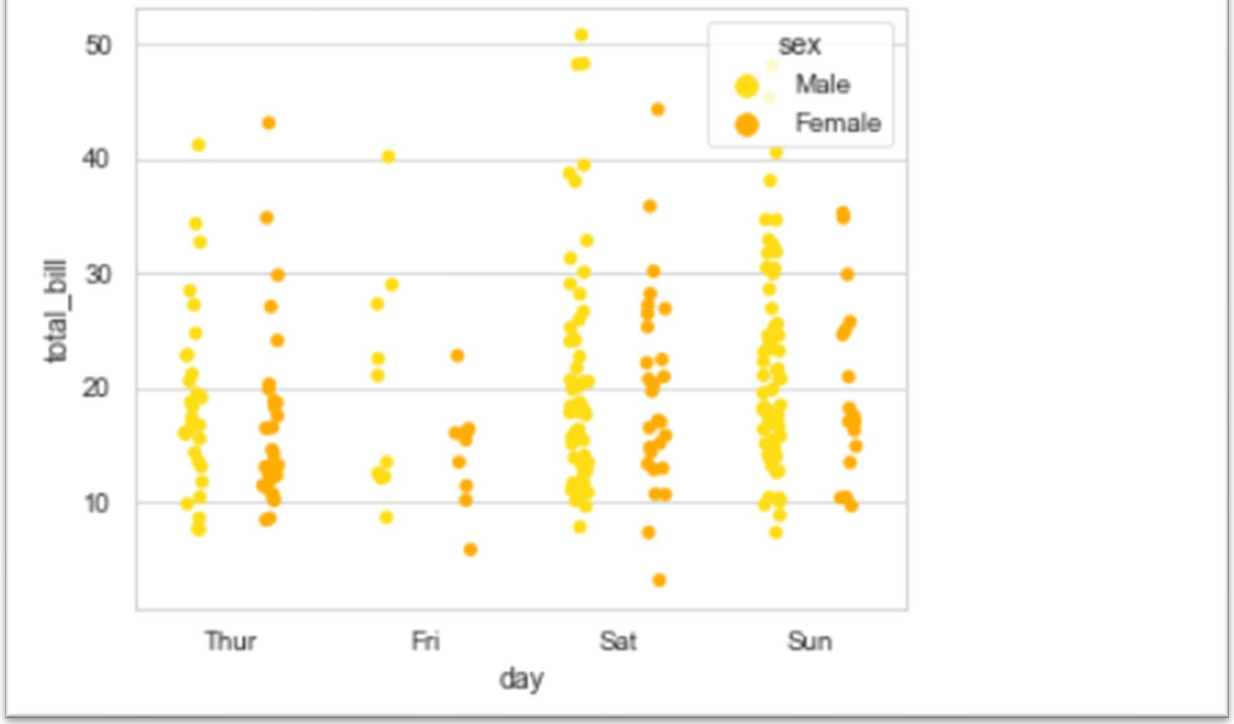


Pallets

```
sns.stripplot(data = tips, x='day', y='total_bill',
              hue='sex', dodge=True, palette = 'magma')
```




```
sns.stripplot(data = tips, x='day', y='total_bill',
              hue='sex', dodge=True, palette = 'Wistia')
```

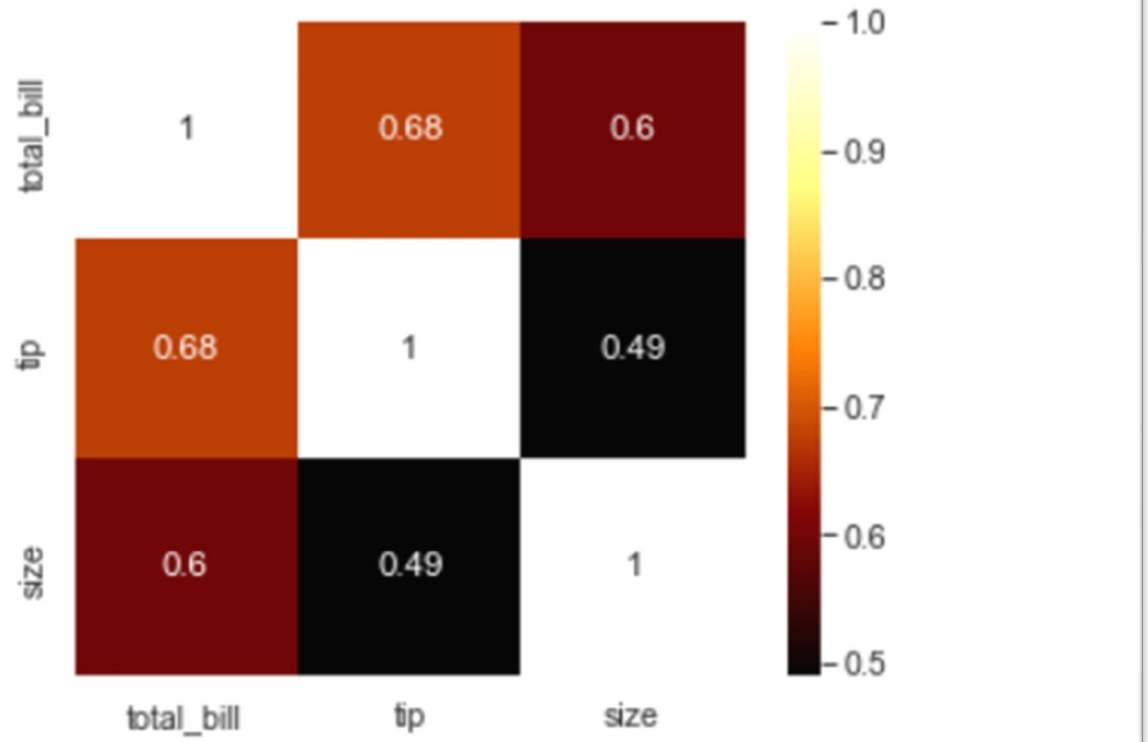


Matrix Plots: Heatmaps Correlation

```
tips_mx = tips.corr()
tips_mx
```

	total_bill	tip	size
total_bill	1.000000	0.675734	0.598315
tip	0.675734	1.000000	0.489299
size	0.598315	0.489299	1.000000

```
sns.heatmap(tips_mx, annot=True, cmap = 'afmhot')
```

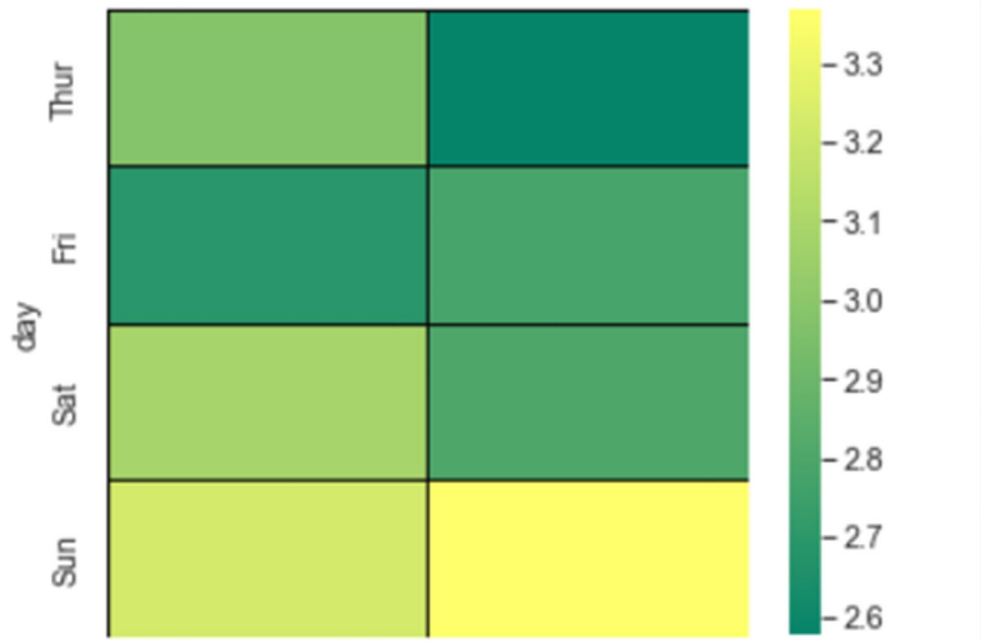


Heatmap With Pivot

```
tips_pt = tips.pivot_table (index='day',
                            columns = 'sex',values='tip')
```

sex	Male	Female
day		
Thur	2.980333	2.575625
Fri	2.693000	2.781111
Sat	3.083898	2.801786
Sun	3.220345	3.367222

```
sns.heatmap(tips_pt,cmap = 'summer',
            linecolor = 'black',linewidth=1)
```

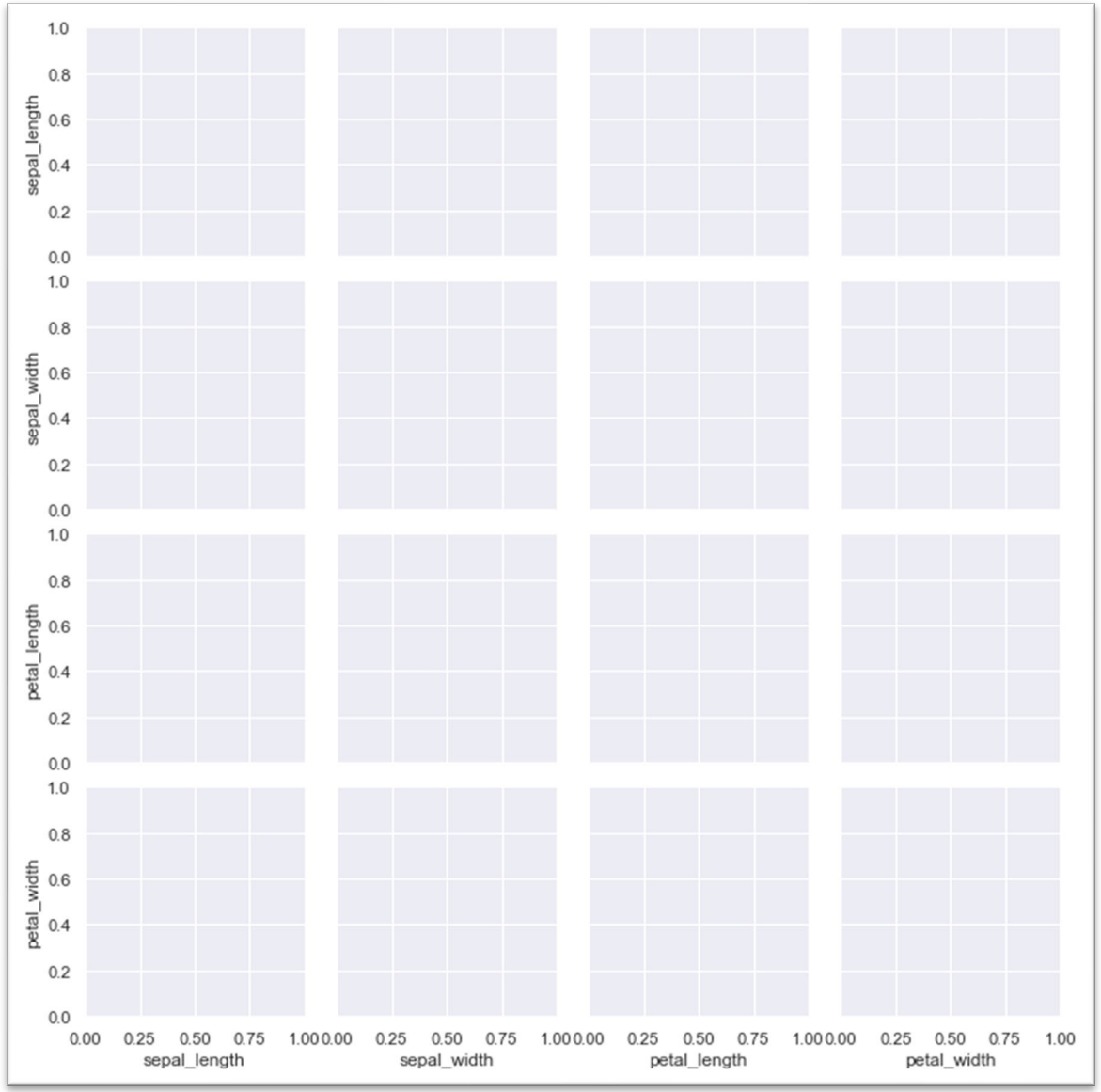


PairGrid

```
iris = sns.load_dataset('iris')
iris.head()
```

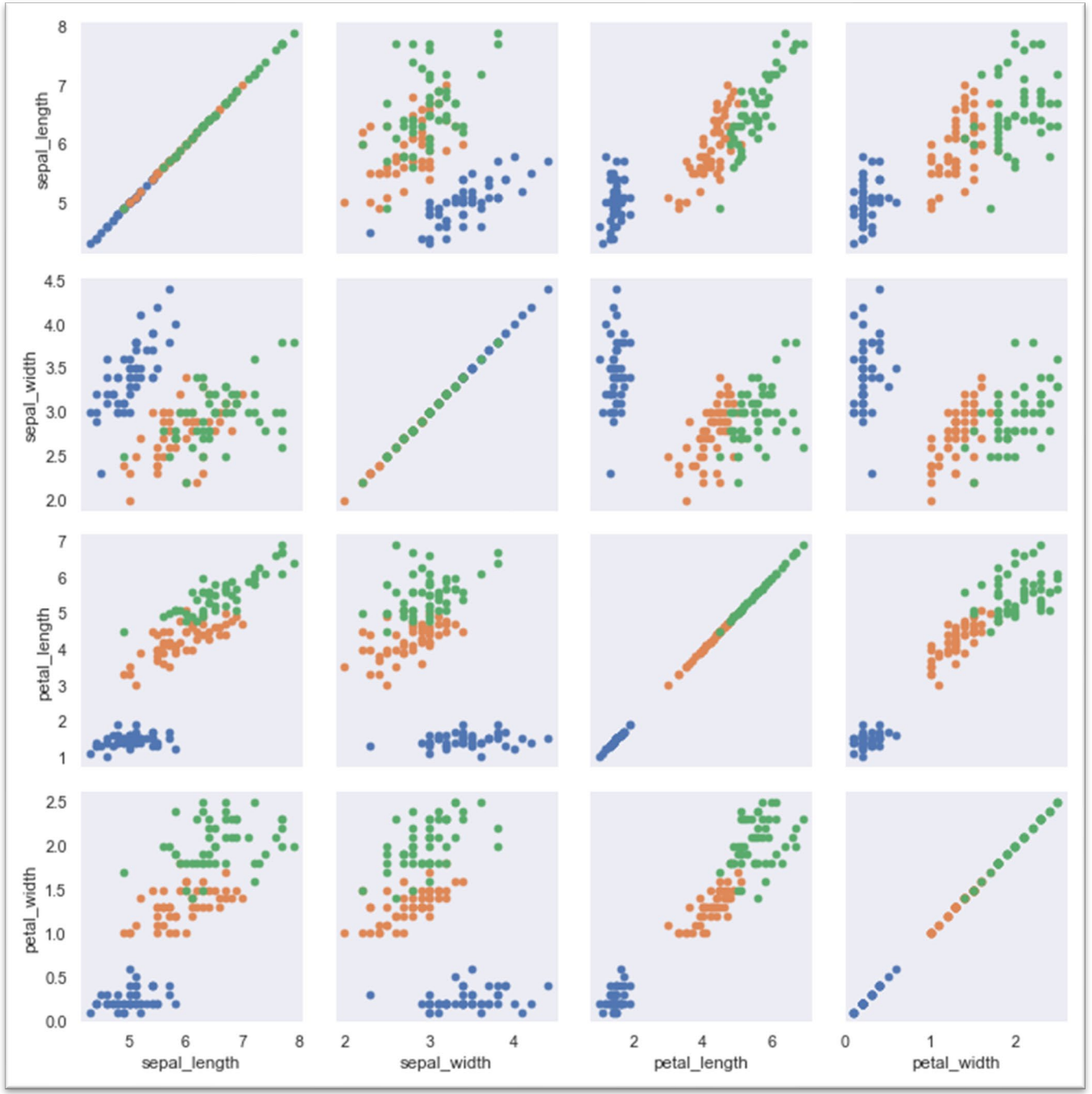
	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
iris_grid = sns.PairGrid(iris, hue = 'species')
```



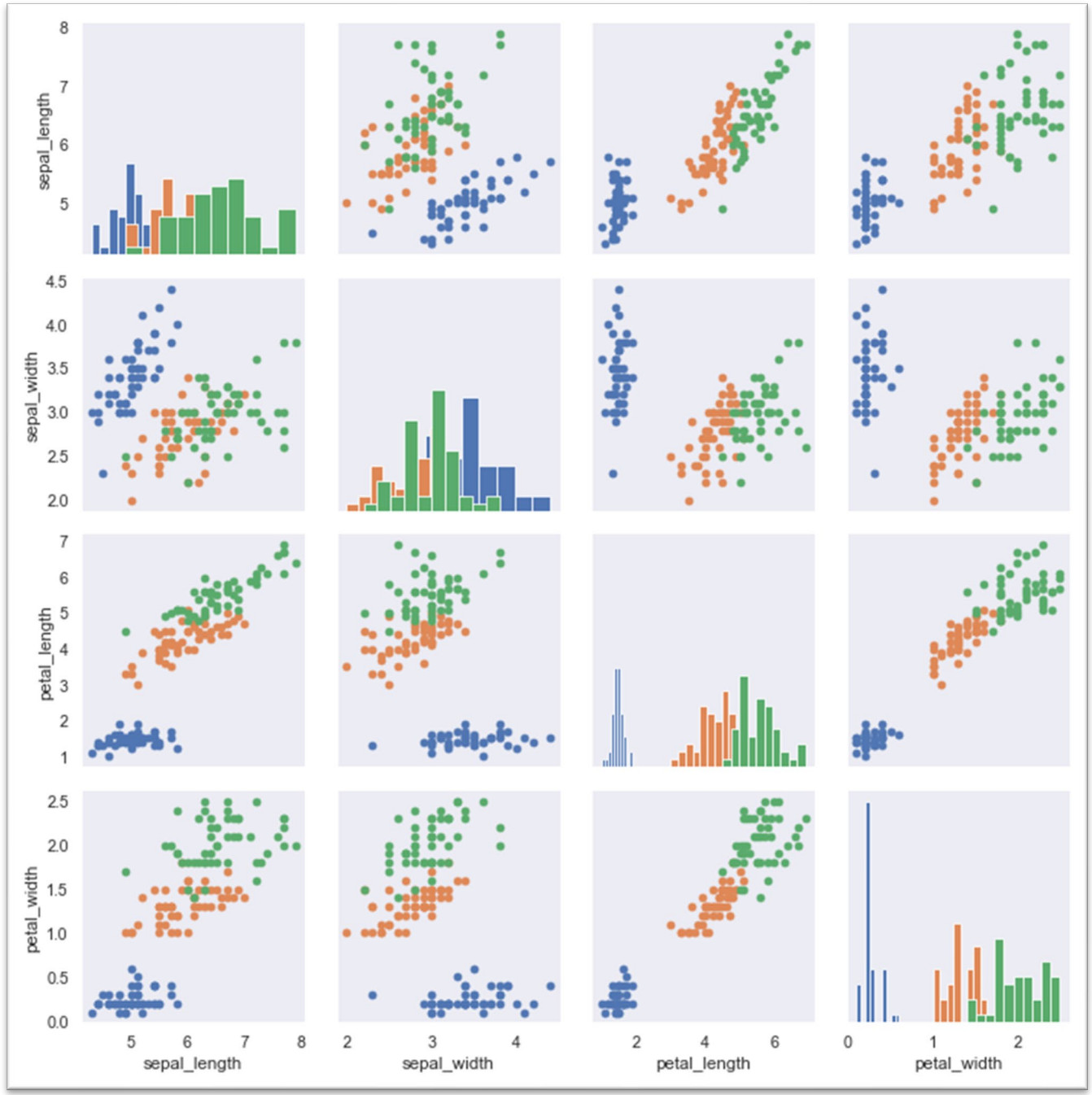
Filling PairGrid With ScatterPlots

```
iris_grid.map(plt.scatter)
```



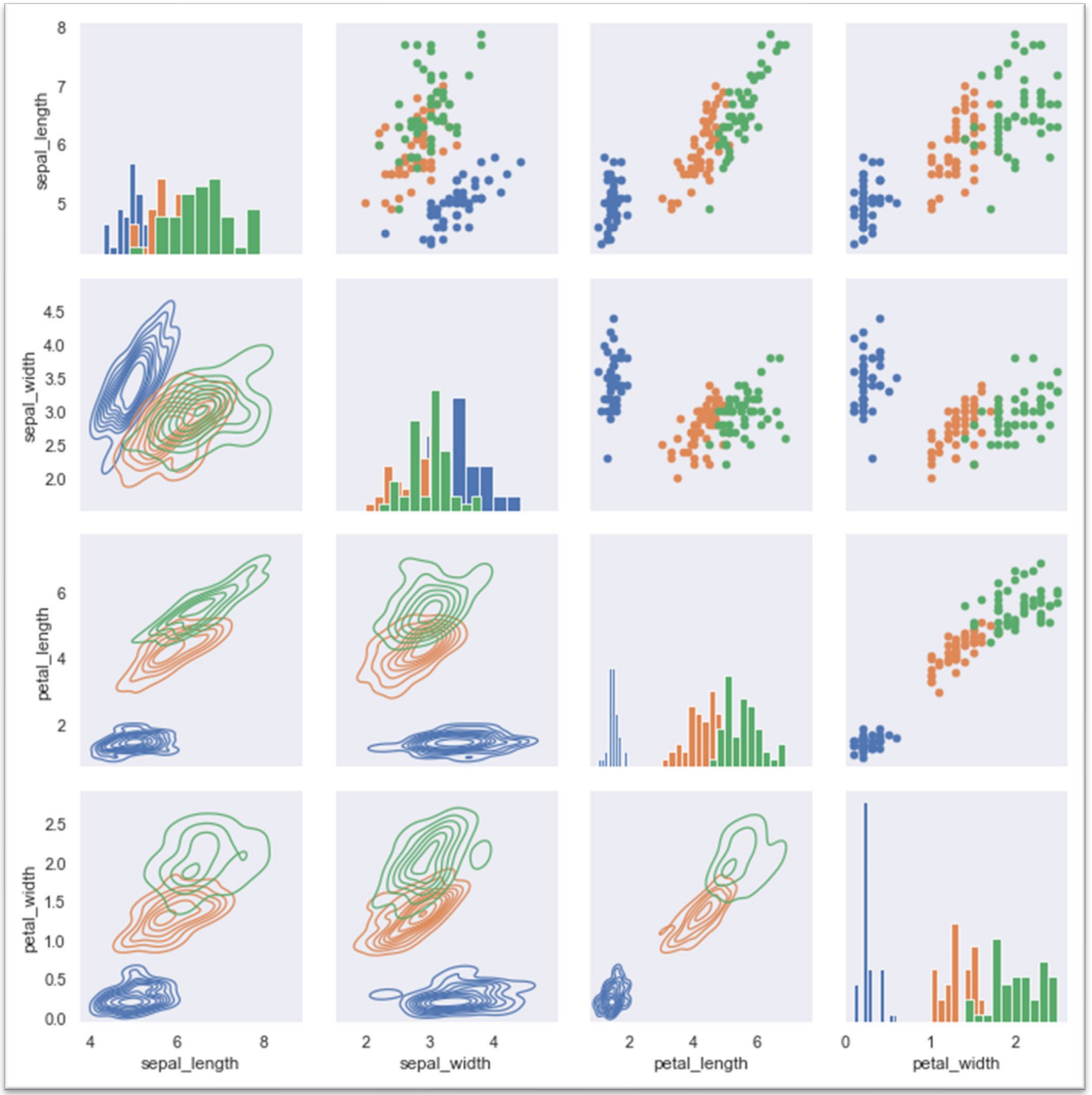
Different Types of Plots in PairGrids

```
iris_grid = sns.PairGrid(iris, hue = 'species')

iris_grid.map_diag(plt.hist)

iris_grid.map_offdiag(plt.scatter)
```



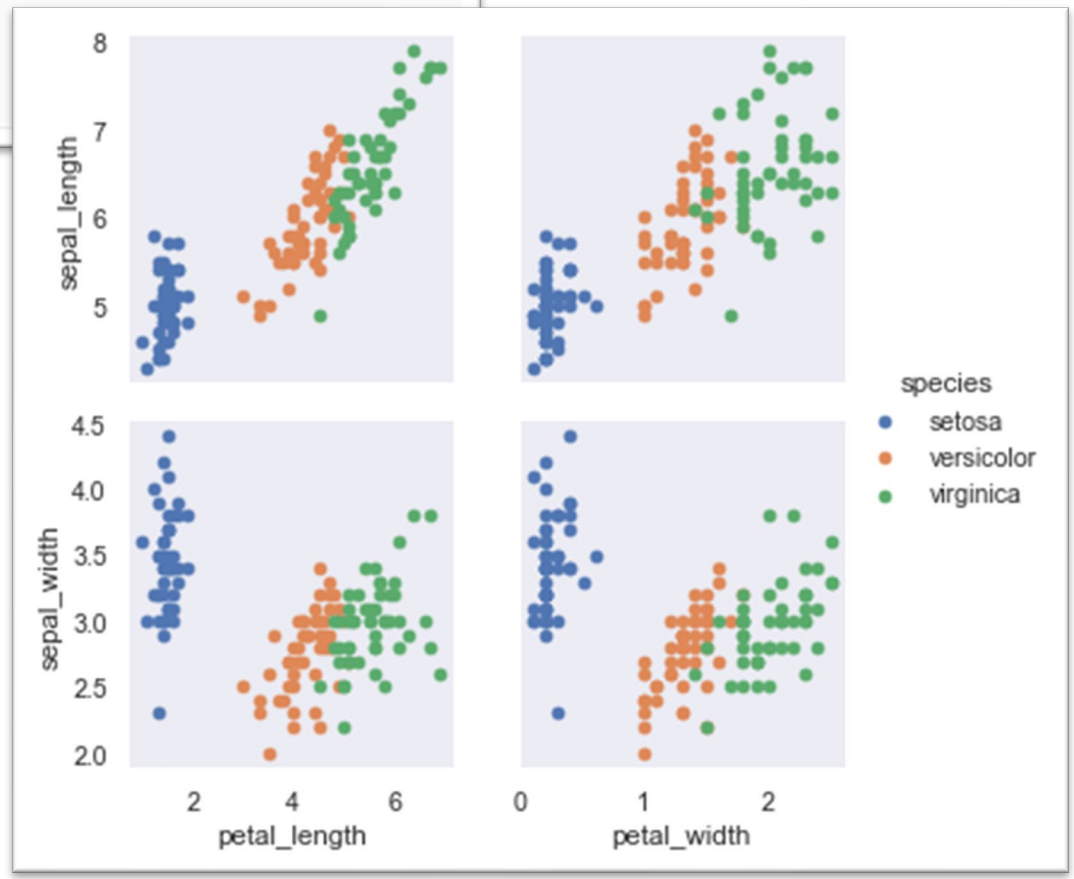
Different Types of Plots in PairGrids

```
iris_grid = sns.PairGrid(iris, hue = 'species')

iris_grid.map_diag(plt.hist)

iris_grid.map_lower(sns.kdeplot)

iris_grid.map_upper(plt.scatter)
```



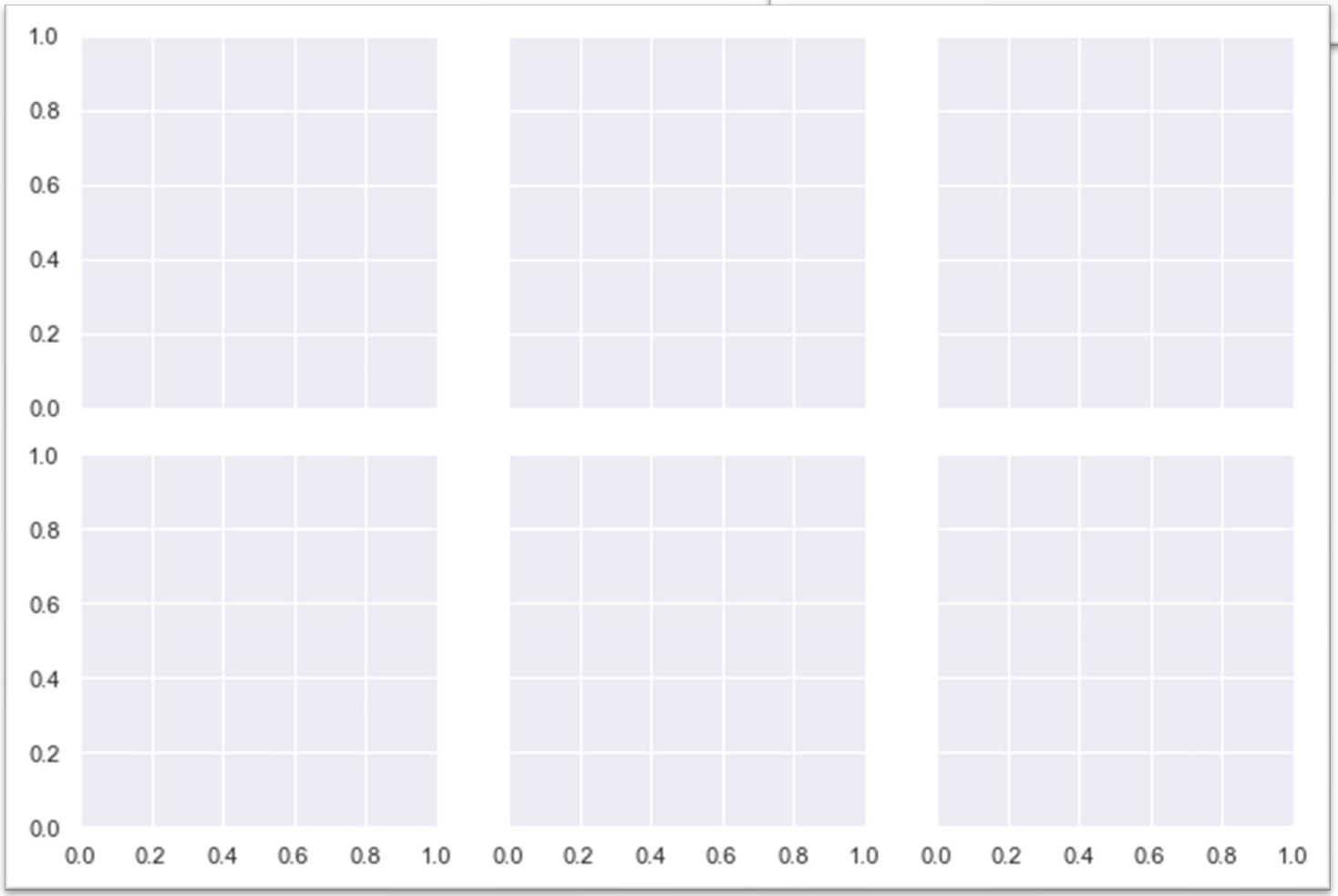
Customizing PairGrid for Axes

```
iris_grid = sns.PairGrid(iris, hue = 'species',
                         x_vars = ['petal_length','petal_width'],
                         y_vars = ['sepal_length','sepal_width'])

iris_grid.map(plt.scatter)
iris_grid.add_legend();
```



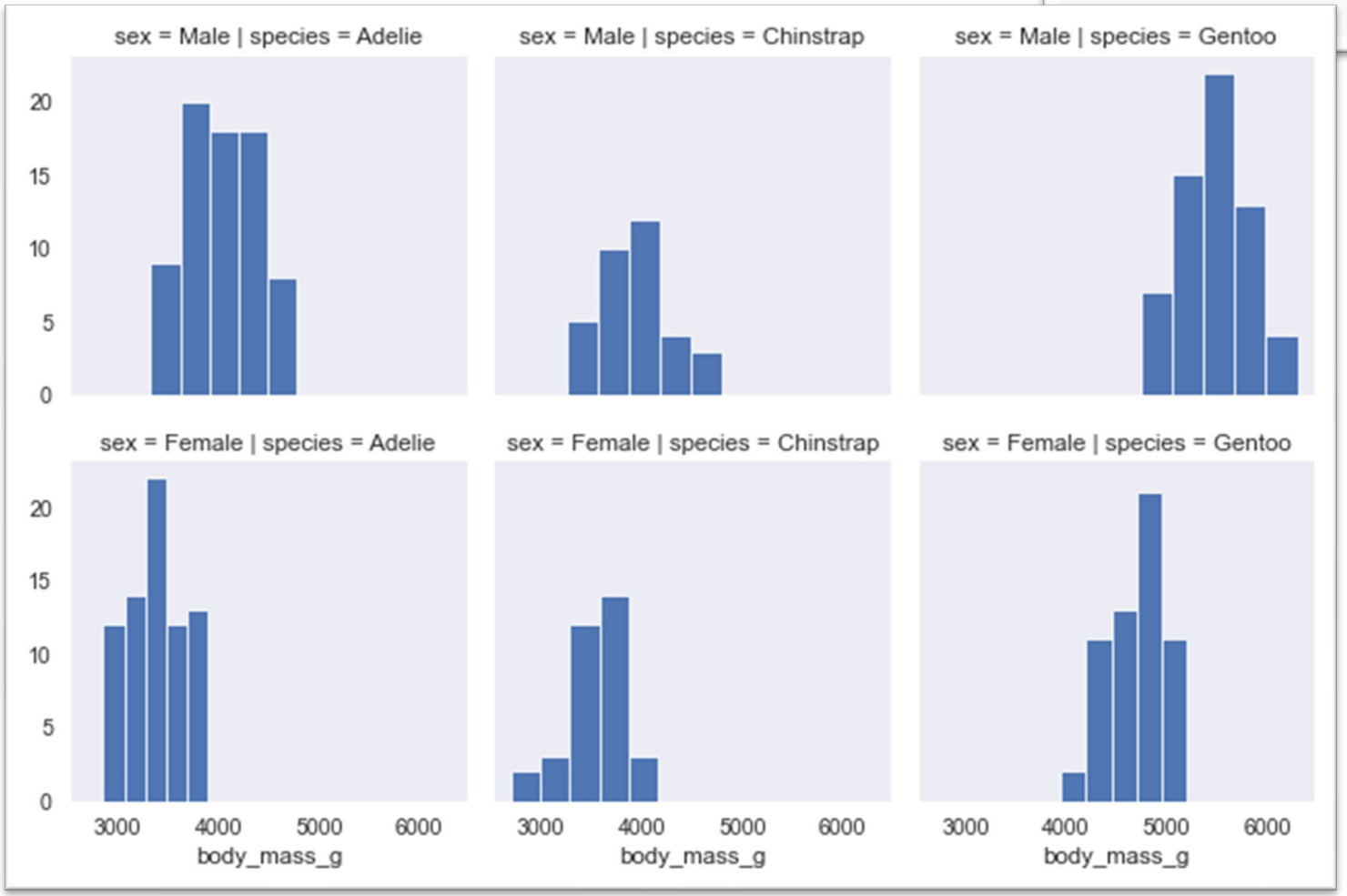
Facet Grids

```
penguins_fg = sns.FacetGrid(penguins,
                            col='species',
                            row='sex')
```

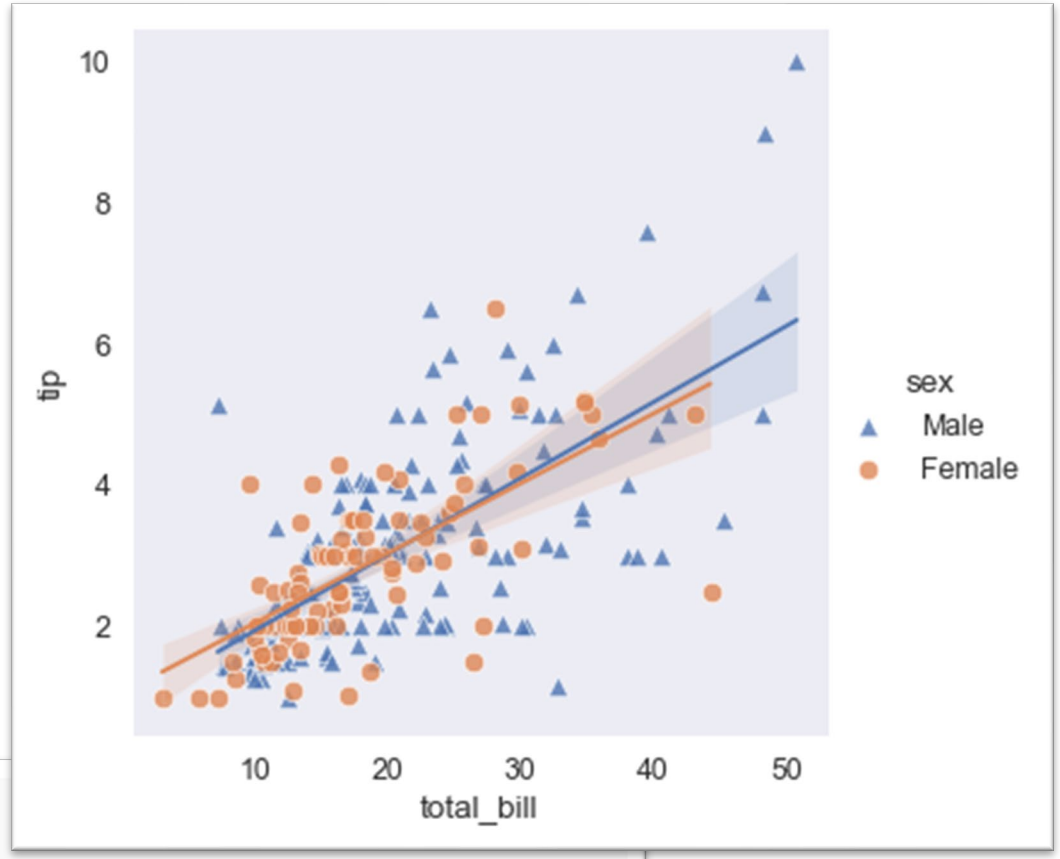


Facet Grids

```
penguins_fg.map(plt.hist,
                'body_mass_g',
                bins=5 );
```



Regression Plots




```
plt.figure(figsize=(8,6))
sns.set_context('paper', font_scale = 1.4)
sns.lmplot(data = tips, x='total_bill',y='tip',
           hue ='sex', markers = ['^','o'],
          scatter_kws = {'s':70,'edgecolor':'w', 'linewidth':0.7});
```

Regression Plots

```
sns.lmplot(data=tips,
           x='total_bill',
           y='tip',
           col = 'sex',
           row = 'time')
```