Exploratory Data Analysis

What is Exploratory Data Analysis

EDA is an approach for data analysis using variety of techniques to gain insights about the data.

Basic steps in any exploratory data analysis:

- Cleaning and preprocessing
- Statistical Analysis
- Visualization for trend analysis, anomaly detection, outlier detection (and removal).

Importance of EDA

- Improve understanding of variables by extracting averages, mean, minimum, and maximum values, etc.
- Discover errors, outliers, and missing values in the data.
- Identify patterns by visualizing data in graphs such as bar graphs, scatter plots, heatmaps and histograms.

EDA using Pandas

- Import data into workplace(Jupyter notebook, Google colab, Python IDE)
- Descriptive statistics
- Removal of nulls
- Visualization

1. Packages and data import

- Step 1 : Import pandas to the workplace.
  - "Import pandas"
- Step 2 : Read data/dataset into Pandas dataframe. Different input formats include:
  - Excel : read_excel
  - CSV: read_csv
  - JSON: read_json
  - HTML and many more

2. Descriptive Stats (Pandas)

- Used to make preliminary assessments about the population distribution of the variable.
- Commonly used statistics:

  1. Central tendency :
     - Mean – The average value of all the data points. : dataframe.mean()
     - Median – The middle value when all the data points are put in an ordered list: dataframe.median()
     - Mode – The data point which occurs the most in the dataset :dataframe.mode()
  2. Spread : It is the measure of how far the datapoints are away from the mean or median
     - Variance - The variance is the mean of the squares of the individual deviations: dataframe.var()
     - Standard deviation - The standard deviation is the square root of the variance:dataframe.std()
  3. Skewness: It is a measure of asymmetry: dataframe.skew()

Descriptive Stats (contd.)

Other methods to get a quick look on the data:

- Describe() : Summarizes the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.
  - Syntax: pandas.dataframe.describe()
- Info() :Prints a concise summary of the dataframe. This method prints information about a dataframe including the index dtype and columns, non-null values and memory usage.
  - Syntax: pandas.dataframe.info()

3. Null values

Detecting

Detecting Null-values:

- Isnull(): It is used as an alias for dataframe.isna(). This function returns the dataframe with boolean values indicating missing values.
- Syntax : dataframe.isnull()

Handling

Handling null values:

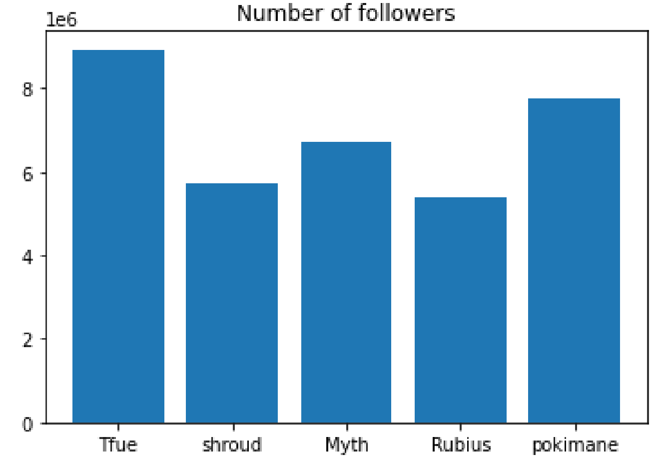
- Dropping the rows with null values: dropna() function is used to delete rows or columns with null values.
- Replacing missing values: fillna() function can fill the missing values with a special value value like mean or median.

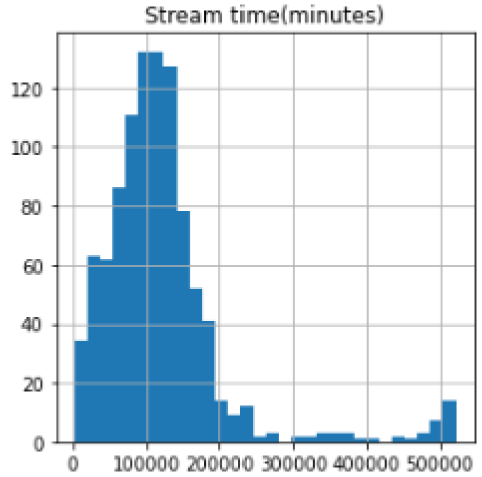
4. Visualization

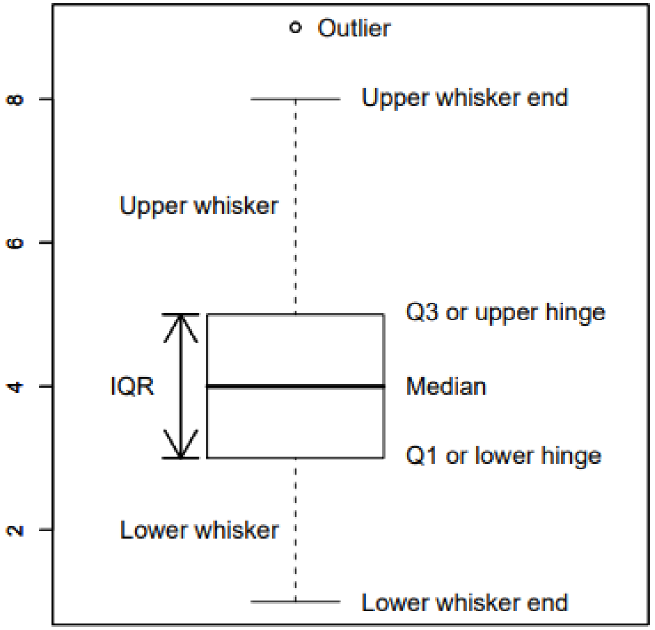
- Univariate: Looking at one variable/column at a time
  - Bar-graph
  - Histograms
  - Boxplot
- Multivariate : Looking at relationship between two or more variables
  - Scatter plots
  - Pie plots
  - Heatmaps(seaborn)

Bar-Graph,Histogram and Boxplot

- Bar graph: A bar plot is a plot that presents data with rectangular bars with lengths proportional to the values that they represent.
- Boxplot : Depicts numerical data graphically through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2).
- Histogram: A histogram is a representation of the distribution of data.



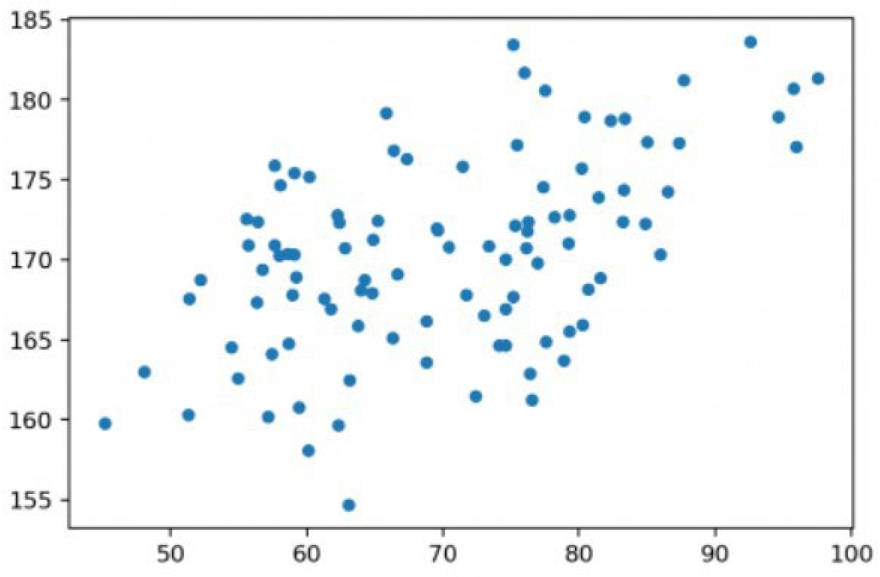


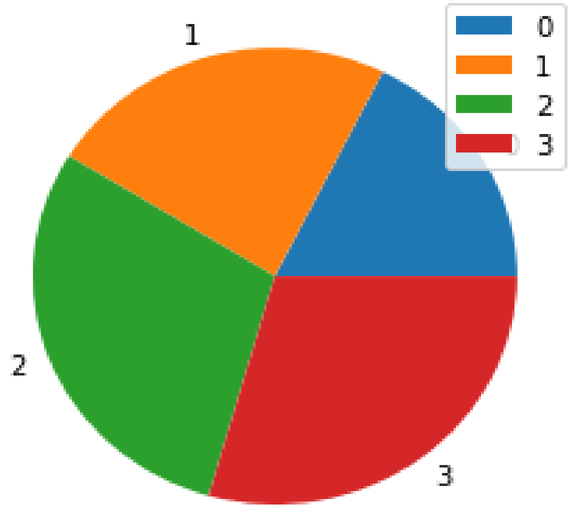


Scatterplot, Pieplot

- Scatterplot : Shows the data as a collection of points.
  - Syntax: dataframe.plot.scatter(x = 'x_column_name', y = 'y_columnn_name’)

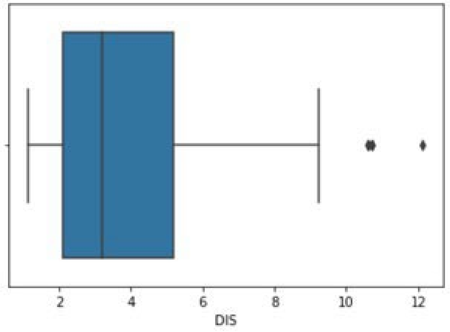
- Pie plot : Proportional representation of the numerical data in a column.
  - Syntax: dataframe.plot.pie(y=‘column_name’)

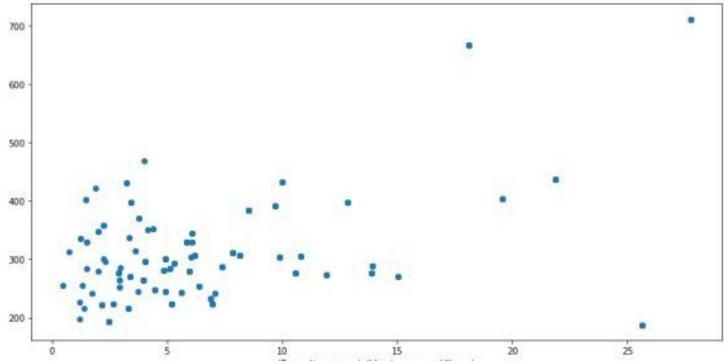




Outlier detection

- An outlier is a point or set of data points that lie away from the rest of the data values of the dataset..
- Outliers are easily identified by visualizing the data.
- For e.g.
  - In a boxplot, the data points which lie outside the upper and lower bound can be considered as outliers
  - In a scatterplot, the data points which lie outside the groups of datapoints can be considered as outliers





Outlier removal

- Calculate the IQR as follows:
  - Calculate the first and third quartile (Q1 and Q3)
  - Calculate the interquartile range, **IQR = Q3-Q1**
  - Find the lower bound which is **Q1*1.5**
  - Find the upper bound which is **Q3*1.5**
  - Replace the data points which lie outside this range.
  - They can be replaced by mean or median.

Exploratory Data Analysis (Part-2)

Numerical Python (NumPy)

NumPy is the most foundational package for numerical computing in Python.
If you are going to work on data analysis or machine learning projects, then having a solid understanding of NumPy is nearly mandatory.
Indeed, many other libraries, such as pandas and scikit-learn, use NumPy's array objects as the *lingua franca* for data exchange.
One of the reasons as to why NumPy is so important for numerical computations is because it is designed for efficiency with large arrays of data. The reasons for this include:

- It stores data internally in a continuous block of memory, independent of other in-built Python objects.
- It performs complex computations on entire arrays without the need for for loops.

What you'll find in NumPy

ndarray: an efficient multidimensional array providing fast array-orientated arithmetic operations and flexible *broadcasting* capabilities.
Mathematical functions for fast operations on entire arrays of data without having to write loops.
Tools for reading/writing array data to disk and working with memory-mapped files.
Linear algebra, random number generation, and Fourier transform capabilities.
A C API for connecting NumPy with libraries written in C, C++, and FORTRAN. This is why Python is the language of choice for wrapping legacy codebases.

The NumPy ndarray: A multi-dimensional array object

The NumPy ndarray object is a fast and flexible container for large data sets in Python.
NumPy arrays are a bit like Python lists, but are still a very different beast at the same time.
Arrays enable you to store multiple items of the same data type. It is the facilities around the array object that makes NumPy so convenient for performing math and data manipulations.

Ndarray vs. lists

By now, you are familiar with Python lists and how incredibly useful they are.
So, you may be asking yourself:

"I can store numbers and other objects in a Python list and do all sorts of computations and manipulations through list comprehensions, for-loops etc. What do I need a NumPy array for?"

There are very significant advantages of using NumPy arrays overs lists.

Creating a NumPy array

To understand these advantages, lets create an array.
One of the most common, of the many, ways to create a NumPy array is to create one from a list by passing it to the np.array() function.

In:
```
import numpy as np
list1 = [0, 1, 2, 3, 4]
arr = np.array(list1)

print(type(arr))
print(arr)
```

Out:
```
In [1]: runfile('C:/Users,
wdir='C:/Users/Lew_laptop,
<type 'numpy.ndarray'>
[0 1 2 3 4]
```

Differences between lists and ndarrays

The key difference between an array and a list is that arrays are designed to handle vectorised operations while a python lists are not.
That means, if you apply a function, it is performed on every item in the array, rather than on the whole array object.

Let’s suppose you want to add the number 2 to every item in the list. The intuitive way to do this is something like this:

In:
```
import numpy as np
list1 = [0, 1, 2, 3, 4]
list1 = list1+2
```

Out
```
File "C:/Users/Lew_laptop/.spyder-py3/temp.py", line 9, i
  list1 = list1+2

TypeError: can only concatenate list (not "int") to list
```

- That was not possible with a list, but you can do that on an array:

In:
```
import numpy as np
list1 = [0, 1, 2, 3, 4]
arr = np.array(list1)
print(arr)
arr = arr+2
print(arr)
```

Out:
```
In [7]: runfile('C:/Users
Lew_laptop/.spyder-py3')
[0 1 2 3 4]
[2 3 4 5 6]
```

It should be noted here that, once a Numpy array is created, you cannot increase its size.
To do so, you will have to create a new array.

Create a 2d array from a list of list

You can pass a list of lists to create a matrix-like a 2d array.

In:

```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr2=np.array(list2)
print(arr2)
```

Out:

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

The dtype argument

You can specify the data-type by setting the dtype() argument.
Some of the most commonly used NumPy dtypes are: float, int, bool, str, and object.

In:

```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print(arr3)
```

Out:

```
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
```

The astype argument

You can also convert it to a different data-type using the astype method.

In:

```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print(arr3)
arr3_s = arr3.astype('int').astype('str')
print(arr3_s)
```

Out:

```
[[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
[['0' '1' '2']
 ['3' '4' '5']
 ['6' '7' '8']]
```

- Remember that, unlike lists, all items in an array have to be of the same type.

dtype='object'

However, if you are uncertain about what data type your array will hold, or if you want to hold characters and numbers in the same array, you can set the dtype as 'object'.

In:
```
arr_obj = np.array([1, 'a'], dtype='object')
print(arr_obj)
```

Out:
```
[1 'a']
```

The tolist() function

You can always convert an array into a list using the tolist() command.

In:
```
arr_list = arr_obj.tolist()
print(arr_list)
```

Out:
```
[1, 'a']
```

Inspecting a NumPy array

There are a range of functions built into NumPy that allow you to inspect different aspects of an array:

In:

```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print('Shape:', arr3.shape)
print('Data type:', arr3.dtype)
print('Size:', arr3.size)
print('Num dimensions:', arr3.ndim)
```

Out:

```
Shape: (3, 3)
Data type: float64
Size: 9
Num dimensions: 2
```

Extracting specific items from an array

You can extract portions of the array using indices, much like when you're working with lists.
Unlike lists, however, arrays can optionally accept as many parameters in the square brackets as there are number of dimensions

In:

```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
print("whole:", arr3)
print("Part:", arr3[:2, :2])
```

Out:

```
whole: [[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
Part: [[0. 1.]
 [3. 4.]]
```

Boolean indexing

A boolean index array is of the same shape as the array-to-be-filtered, but it only contains TRUE and FALSE values.

In:
```
import numpy as np
list2 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
arr3=np.array(list2, dtype='float')
boo = arr3>2
print(boo)
```

Out:
```
[[False False False]
 [ True  True  True]
 [ True  True  True]]
```

Pandas

Pandas, like NumPy, is one of the most popular Python libraries for data analysis.
It is a high-level abstraction over low-level NumPy, which is written in pure C.
Pandas provides high-performance, easy-to-use data structures and data analysis tools.
There are two main structures used by pandas; *data frames* and *series*.

Indices in a pandas series

A pandas series is similar to a list, but differs in the fact that a series associates a label with each element. This makes it look like a dictionary.
If an index is not explicitly provided by the user, pandas creates a `RangeIndex` ranging from 0 to *N*-1.
Each series object also has a data type.

In:
```python
import pandas as pd
new_series = pd.Series([5, 6, 7, 8, 9, 10])
print(new_series)
```

Out:
```
0     5
1     6
2     7
3     8
4     9
5    10
dtype: int64
```

As you may suspect by this point, a series has ways to extract all of the values in the series, as well as individual elements by index.

In:
```
import pandas as pd
new_series = pd.Series([5, 6, 7, 8, 9, 10])
print(new_series.values)
print('_______________')
print(new_series[4])
```

Out:
```
[ 5  6  7  8  9 10]
_______________
9
```

- You can also provide an index manually.

In:
```
import pandas as pd
new_series = pd.Series([5, 6, 7, 8, 9, 10], index=['a', 'b', 'c', 'd', 'e', 'f'])
print(new_series.values)
print('_______________')
print(new_series['f'])
```

Out:
```
[ 5  6  7  8  9 10]
_______________
10
```

It is easy to retrieve several elements of a series by their indices or make group assignments.

In:

```
import pandas as pd
new_series = pd.Series([5, 6, 7, 8, 9, 10], index=['a', 'b', 'c', 'd', 'e', 'f'])
print(new_series)
print('_______________')
new_series[['a', 'b', 'f']] = 0
print(new_series)
```

Out:

```
a     5
b     6
c     7
d     8
e     9
f    10
dtype: int64
_______________
a    0
b    0
c    7
d    8
e    9
f    0
dtype: int64
```

Filtering and maths operations

Filtering and maths operations are easy with Pandas as well.

In:

```
import pandas as pd
new_series = pd.Series([5, 6, 7, 8, 9, 10], index=['a', 'b', 'c', 'd', 'e', 'f'])
new_series2 = new_series[new_series>0]
print(new_series2)
print('_____________')
new_series2[new_series2>0]*2
print(new_series2)
```

Out:

```
a     5
b     6
c     7
d     8
e     9
f    10
dtype: int64
_______________
a     5
b     6
c     7
d     8
e     9
f    10
dtype: int64
```

Pandas data frame

Simplistically, a data frame is a table, with rows and columns.
Each column in a data frame is a series object.
Rows consist of elements inside series.

Case ID	Variable one	Variable two	Variable 3
1	123	ABC	10
2	456	DEF	20
3	789	XYZ	30

Creating a Pandas data frame

Pandas data frames can be constructed using Python dictionaries.

In:

```
import pandas as pd
df = pd.DataFrame({
     'country': ['Kazakhstan', 'Russia', 'Belarus', 'Ukraine'],
     'population': [17.04, 143.5, 9.5, 45.5],
     'square': [2724902, 17125191, 207600, 603628]})
print(df)
```

Out:

```
      country  population    square
0  Kazakhstan       17.04   2724902
1      Russia      143.50  17125191
2     Belarus        9.50    207600
3     Ukraine       45.50    603628
```

You can also create a data frame from a list.

In:

```
import pandas as pd
list2 = [[0,1,2],[3,4,5],[6,7,8]]
df = pd.DataFrame(list2)
print(df)
df.columns = ['V1', 'V2', 'V3']
print(df)
```

Out:

```
In [12]: runfile('
   0  1  2
0  0  1  2
1  3  4  5
2  6  7  8
   V1  V2  V3
0   0   1   2
1   3   4   5
2   6   7   8
```

You can ascertain the type of a column with the type() function.

In:
```
print(type(df['country']))
```

Out:
```
<class 'pandas.core.series.Series'>
```

A Pandas data frame object as two indices; a column index and row index. Again, if you do not provide one, Pandas will create a RangeIndex from 0 to *N*-1.

In:

```
import pandas as pd
df = pd.DataFrame({
     'country': ['Kazakhstan', 'Russia', 'Belarus', 'Ukraine'],
     'population': [17.04, 143.5, 9.5, 45.5],
     'square': [2724902, 17125191, 207600, 603628]})
print(df.columns)
print('__________')
print(df.index)
```

Out:

```
Index(['country', 'population', 'square'], dtype='object')
__________
RangeIndex(start=0, stop=4, step=1)
```

There are numerous ways to provide row indices explicitly.
For example, you could provide an index when creating a data frame:

In:

```python
import pandas as pd
df = pd.DataFrame({
    'country': ['Kazakhstan', 'Russia', 'Belarus', 'Ukraine'],
    'population': [17.04, 143.5, 9.5, 45.5],
     'square': [2724902, 17125191, 207600, 603628]
}, index=['KZ', 'RU', 'BY', 'UA'])
print(df)
```

Out:

```
       country  population    square
KZ  Kazakhstan       17.04   2724902
RU      Russia      143.50  17125191
BY     Belarus        9.50    207600
UA     Ukraine       45.50    603628
```

- or do it during runtime.
- Here, I also named the index ‘country code’.

In:

```python
import pandas as pd
df = pd.DataFrame({
    'country': ['Kazakhstan', 'Russia', 'Belarus', 'Ukraine'],
    'population': [17.04, 143.5, 9.5, 45.5],
     'square': [2724902, 17125191, 207600, 603628]
})
print(df)
print('____________')
df.index = ['KZ', 'RU', 'BY', 'UA']
df.index.name = 'Country Code'
print(df)
```

Out:

```
      country  population    square
0  Kazakhstan       17.04   2724902
1      Russia      143.50  17125191
2     Belarus        9.50    207600
3     Ukraine       45.50    603628
____________
                 country  population    square
Country Code
KZ            Kazakhstan       17.04   2724902
RU                Russia      143.50  17125191
BY               Belarus        9.50    207600
UA               Ukraine       45.50    603628
```

Row access using index can be performed in several ways.
First, you could use .loc() and provide an index label.

In:

```
print(df.loc['KZ'])
```

Out:

```
country       Kazakhstan
population         17.04
square           2724902
Name: KZ, dtype: object
```

- Second, you could use .iloc() and provide an index number

In:

```
print(df.iloc[0])
```

Out:

```
country       Kazakhstan
population         17.04
square           2724902
Name: KZ, dtype: object
```

A selection of particular rows and columns can be selected this way.

In:
```
print(df.loc[['KZ', 'RU'], 'population'])
```

Out:
```
Country Code
KZ      17.04
RU     143.50
Name: population, dtype: float64
```

- You can feed .loc() two arguments, index list and column list, slicing operation is supported as well:

In:
```
print(df.loc['KZ':'BY', :])
```

Out:
```
                 country  population    square
Country Code
KZ             Kazakhstan       17.04   2724902
RU                 Russia      143.50  17125191
BY                Belarus        9.50    207600
```

Filtering

Filtering is performed using so-called Boolean arrays.

```
print(df[df.population > 10][['country', 'square']])
```

```
                 country    square
Country Code                      
KZ            Kazakhstan   2724902
RU                Russia  17125191
UA               Ukraine    603628
```

Deleting columns

You can delete a column using the drop() function.

In:

```
print(df)
```

Out:

```
                 country  population    square
Country Code
KZ            Kazakhstan       17.04   2724902
RU                Russia      143.50  17125191
BY               Belarus        9.50    207600
UA               Ukraine       45.50    603628
```

In:

```
df = df.drop(['population'], axis='columns')
print(df)
```

Out:

```
                 country    square
Country Code
KZ            Kazakhstan   2724902
RU                Russia  17125191
BY               Belarus    207600
UA               Ukraine    603628
```

Reading from and writing to a file

Pandas supports many popular file formats including CSV, XML, HTML, Excel, SQL, JSON, etc.

Out of all of these, CSV is the file format that you will work with the most.

You can read in the data from a CSV file using the read_csv() function.

```
df = pd.read_csv('filename.csv', sep=',')
```

Similarly, y

```
df.to_csv('filename.csv')
```

Pandas has the capacity to do much more than what we have covered here, such as grouping data and even data visualisation.
However, as with NumPy, we don't have enough time to cover every aspect of pandas here.

Pandas vs NumPy	
Pandas	**NumPy**
When we have to work on **Tabular data**, we prefer the pandas module.	When we have to work on **Numerical data**, we prefer the NumPy module.
The powerful tools of pandas are **DataFrame and Series.**	Whereas the powerful tool of NumPy is **Arrays.**
Pandas consume **more memory**.	Numpy is **memory efficient.**
Pandas have a better performance when the number of rows is **500K or more.**	Numpy has a better performance when number of rows is **50K or less.**
Indexing of the Pandas series is **very slow** as compared to Numpy arrays.	Indexing of Numpy arrays is **very fast.**
Pandas have a 2D table object called **DataFrame.**	Numpy is capable of providing **multi-dimensional arrays.**

Exploratory data analysis (EDA)

Exploring your data is a crucial step in data analysis. It involves:

Organising the data set

Plotting aspects of the data set

Maybe producing some numerical summaries; central tendency and spread, etc.

"Exploratory data analysis can never be the whole story, but nothing else can serve as the foundation stone."

- John Tukey.

Download the data

Download the Pokemon dataset from:

https://www.kaggle.com/datasets/rounakbanik/pokemon/data

Unzip the folder, and save the data file in a location you'll remember.

Reading in the data

First we import the Python packages we are going to use.
Then we use Pandas to load in the dataset as a data frame.

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
df1 = pd.read_csv('C:/Users/lb690/Google Drive/Teaching/Q-Step_workshops_2019_2020/pokemon_dataset.csv', index_col=0, encoding = "ISO-8859-1")
```

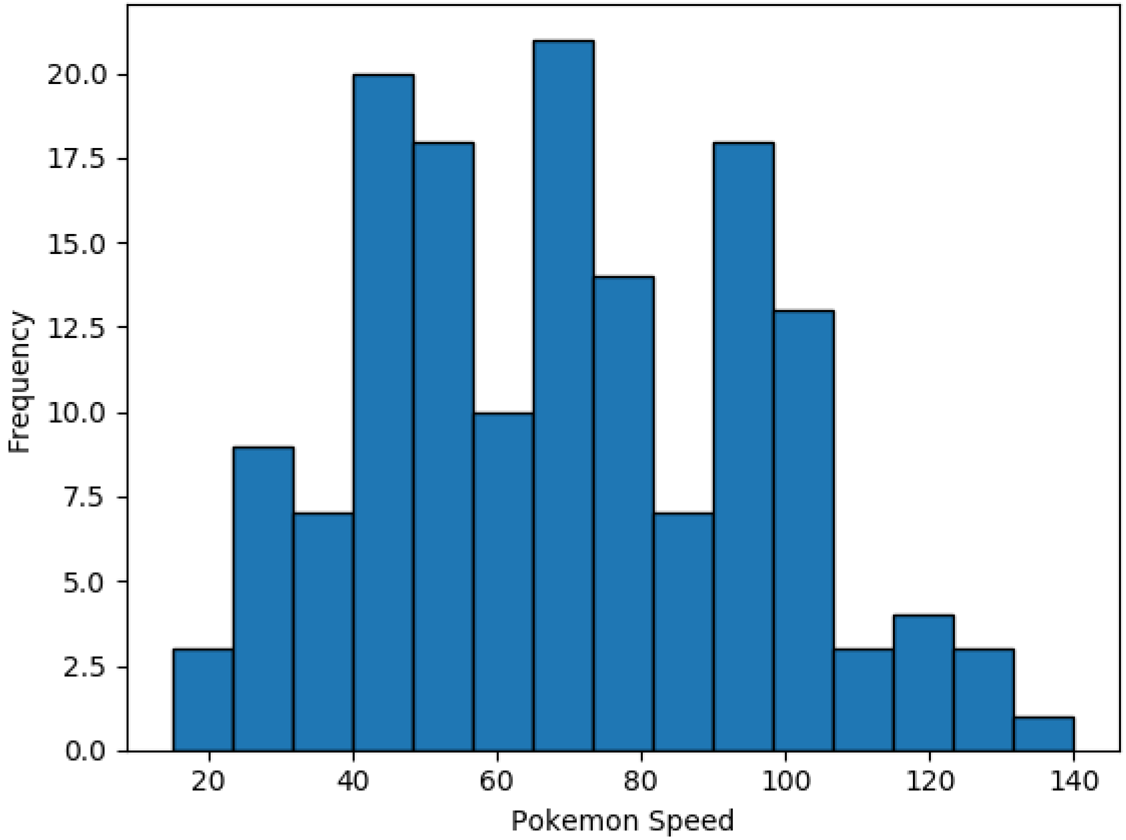
NOTE: The argument `index_col` argument states that we'll treat the first column of the dataset as the ID column.
NOTE: The encoding argument allows us to by pass an input error created by special characters in the data set.

Examine the data set

```
print(df1.head())
         Name Type 1  Type 2  Total  ...  Sp. Def  Speed  Stage  Legendary
#                                    ...
1   Bulbasaur  Grass  Poison    318  ...       65     45      1      False
2     Ivysaur  Grass  Poison    405  ...       80     60      2      False
3    Venusaur  Grass  Poison    525  ...      100     80      3      False
4  Charmander   Fire     NaN    309  ...       50     65      1      False
5  Charmeleon   Fire     NaN    405  ...       65     80      2      False
```

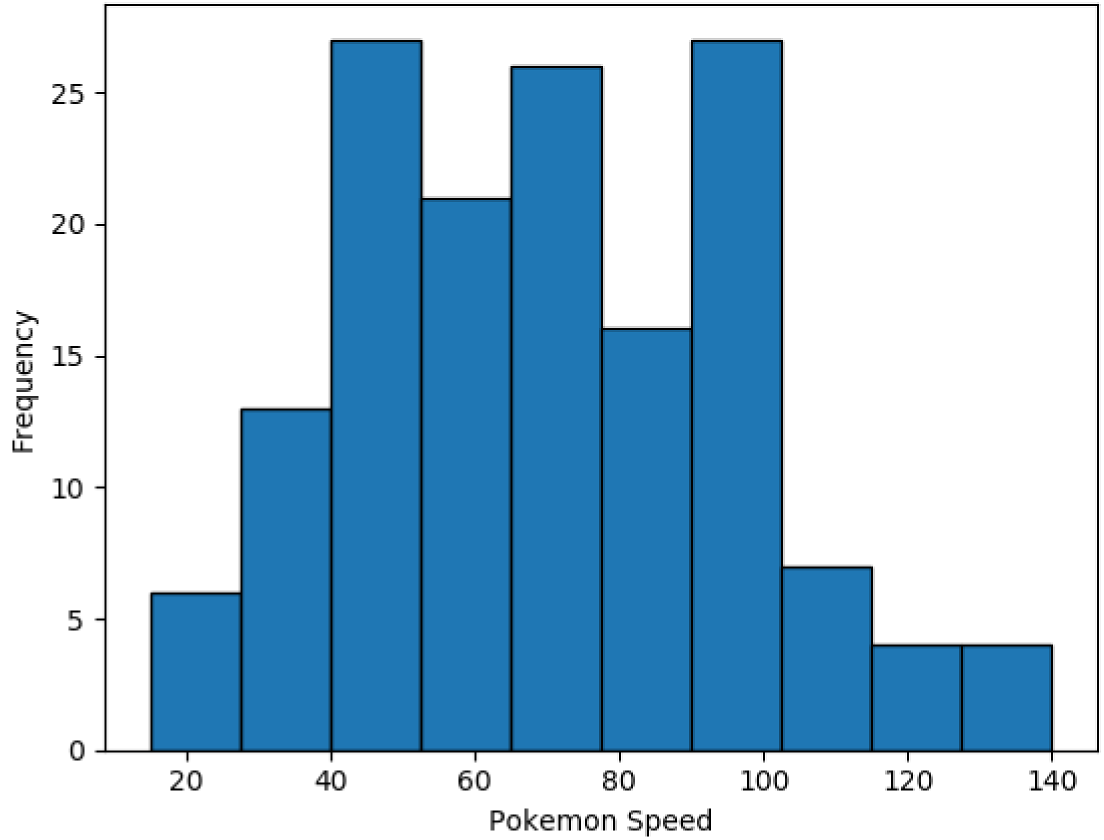
```
print(df1.describe())

           Total          HP      Attack  ...     Sp. Def       Speed       Stage
count  151.00000  151.000000  151.000000  ...  151.000000  151.000000  151.000000
mean   407.07947   64.211921   72.549669  ...   66.019868   68.933775    1.582781
std     99.74384   28.590117   26.596162  ...   24.197926   26.746880    0.676832
min    195.00000   10.000000    5.000000  ...   20.000000   15.000000    1.000000
25%    320.00000   45.000000   51.000000  ...   49.000000   46.500000    1.000000
50%    405.00000   60.000000   70.000000  ...   65.000000   70.000000    1.000000
75%    490.00000   80.000000   90.000000  ...   80.000000   90.000000    2.000000
max    680.00000  250.000000  134.000000  ...  125.000000  140.000000    3.000000
```

We could spend time staring at these numbers, but that is unlikely to offer us any form of insight.
We could begin by conducting all of our statistical tests.
However, a good field commander never goes into battle without first doing a recognisance of the terrain…
This is exactly what EDA is for…



Plotting a histogram in Python:Matplotlib

```
g = plt.hist(df1['Speed'], histtype='bar', ec='black',)
g = plt.xlabel('Pokemon Speed')
g = plt.ylabel('Frequency')
plt.show()
```

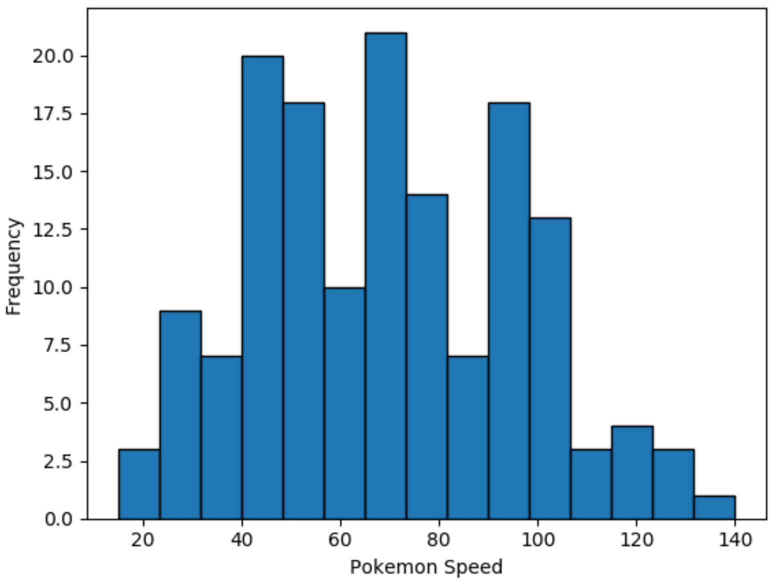


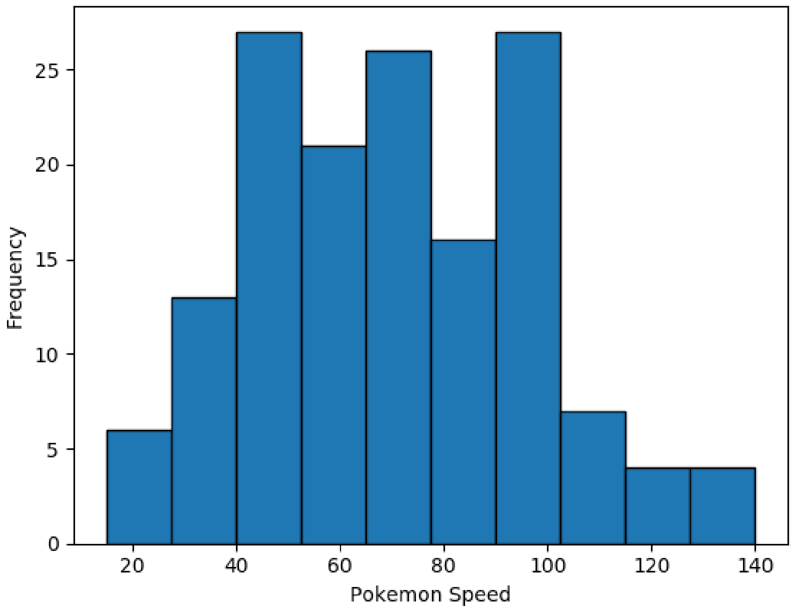
Bins

You may have noticed the two histograms we've seen so far look different, despite using the **exact** same data.

This is because they have different bin values.

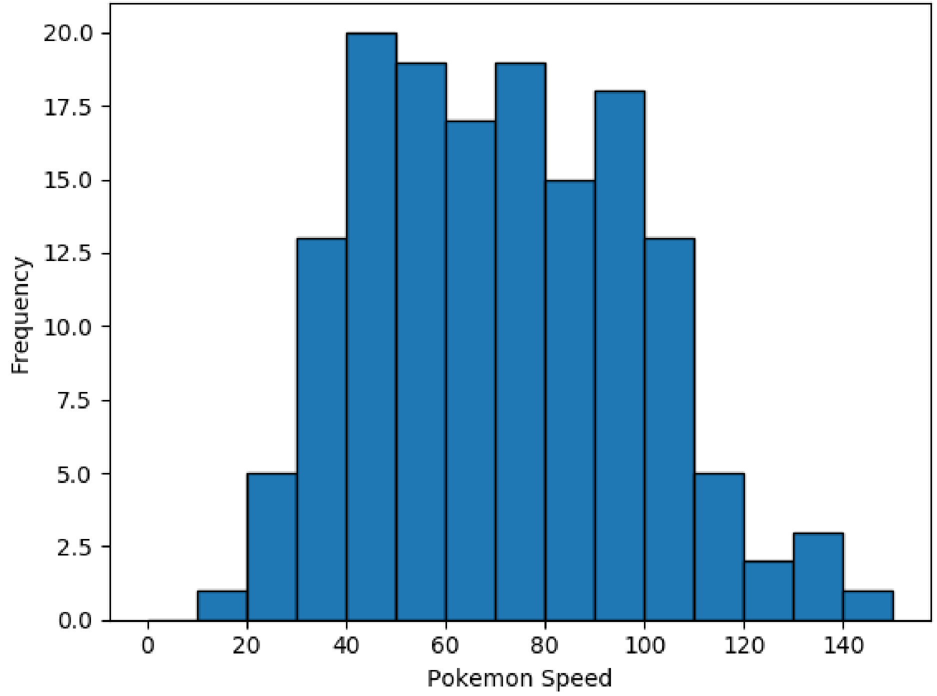
The left graph used the default bins generated by plt.hist(), while the one on the right used bins that I specified.





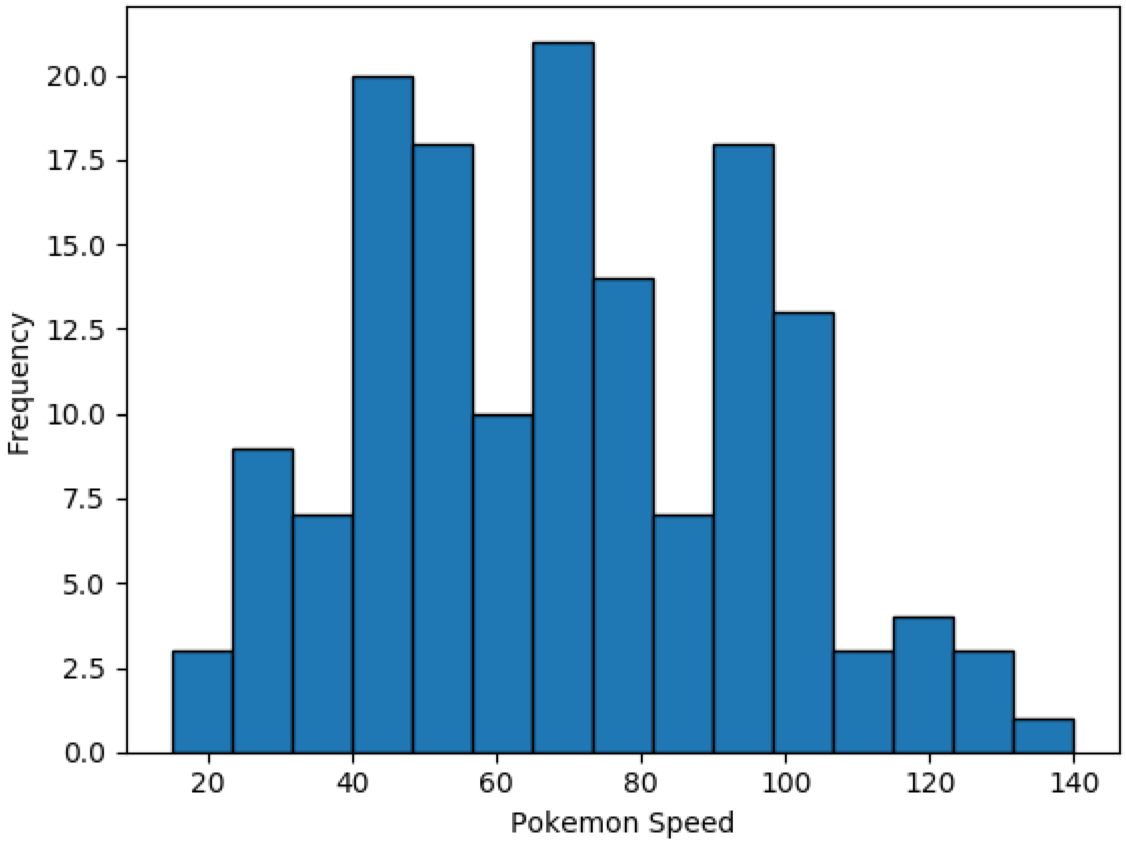
There are a couple of ways to manipulate bins in matplotlib.
Here, I specified where the edges of the bars of the histogram are; the bin edges.

```
bin_edges = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150]
g = plt.hist(df1['Speed'], histtype='bar', ec='black', bins=bin_edges)
g = plt.xlabel('Pokemon Speed')
g = plt.ylabel('Frequency')
plt.show()
```



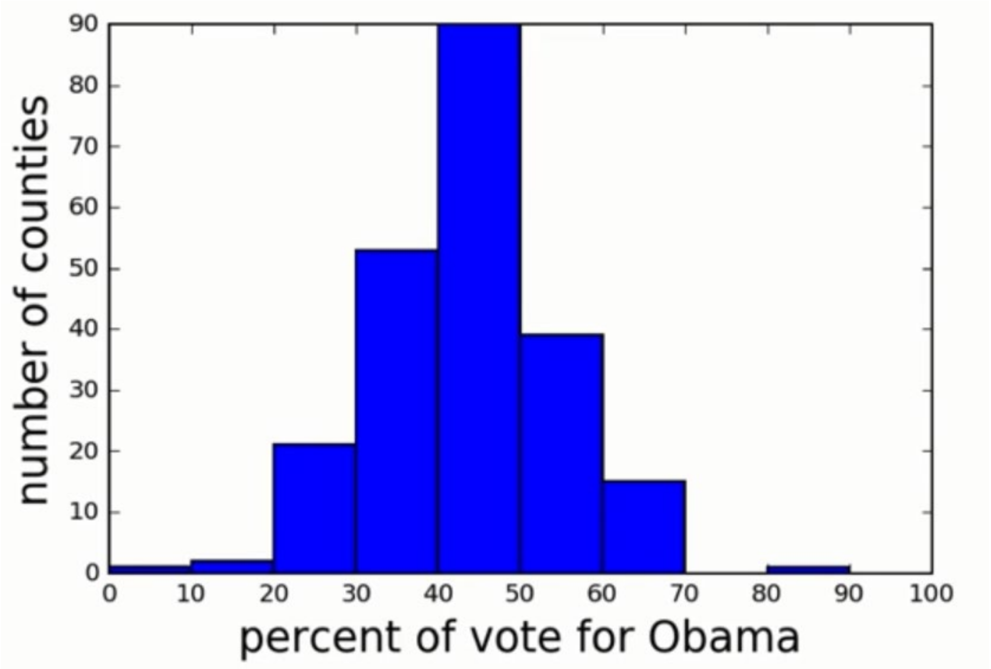
You could also specify the number of bins, and Matplotlib will automatically generate a number of evenly spaced bins.

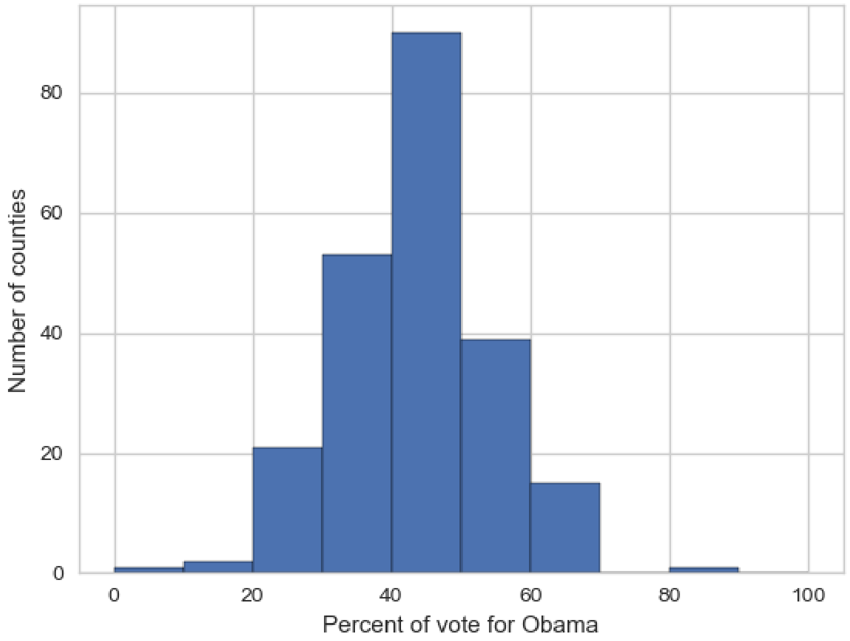
```
g = plt.hist(df1['Speed'], histtype='bar', ec='black', bins=15)
g = plt.xlabel('Pokemon Speed')
g = plt.ylabel('Frequency')
plt.show()
```



Seaborn

Matplotlib is a powerful, but sometimes unwieldy, Python library.
Seaborn provides a high-level interface to Matplotlib and makes it easier to produce graphs like the one on the right.
Some IDEs incorporate elements of this “under the hood” nowadays.





Benefits of Seaborn

Seaborn offers:

- Using default themes that are aesthetically pleasing.
- Setting custom colour palettes.
- Making attractive statistical plots.
- Easily and flexibly displaying distributions.
- Visualising information from matrices and DataFrames.

The last three points have led to Seaborn becoming the exploratory data analysis tool of choice for many Python users.

Features	Matplotlib	Seaborn
Functionality	It is utilized for making basic graphs. Datasets are visualized with the help of bar graphs, histograms, pie charts, scatter plots, lines, and so on.	Seaborn contains several patterns and plots for data visualization. It uses fascinating themes. It helps in compiling whole data into a single plot. It also provides the distribution of data.
Syntax	It uses comparatively complex and lengthy syntax. Example: Syntax for bar graph- matplotlib.pyplot.bar(x_axis, y_axis).	It uses comparatively simple syntax which is easier to learn and understand. Example: Syntax for bargraph- seaborn.barplot(x_axis, y_axis).
Dealing Multiple Figures	We can open and use multiple figures simultaneously. However, they are closed distinctly. Syntax to close one figure at a time: matplotlib.pyplot.close(). Syntax to close all the figures: matplotlib.pyplot.close("all")	Seaborn sets the time for the creation of each figure. However, it may lead to (OOM) out of memory issues
Visualization	Matplotlib is well connected with Numpy and Pandas and acts as a graphics package for data visualization in Python. Pyplot provides similar features and syntax as in MATLAB. Therefore, MATLAB users can easily study it.	Seaborn is more comfortable in handling Pandas data frames. It uses basic sets of methods to provide beautiful graphics in Python.
Pliability	Matplotlib is a highly customized and robust	Seaborn avoids overlapping plots with the help of its default themes
Data Frames and Arrays	Matplotlib works efficiently with data frames and arrays.It treats figures and axes as objects. It contains various stateful APIs for plotting. Therefore plot() like methods can work without parameters.	Seaborn is much more functional and organized than Matplotlib and treats the whole dataset as a single unit. Seaborn is not so stateful and therefore, parameters are required while calling methods like plot()

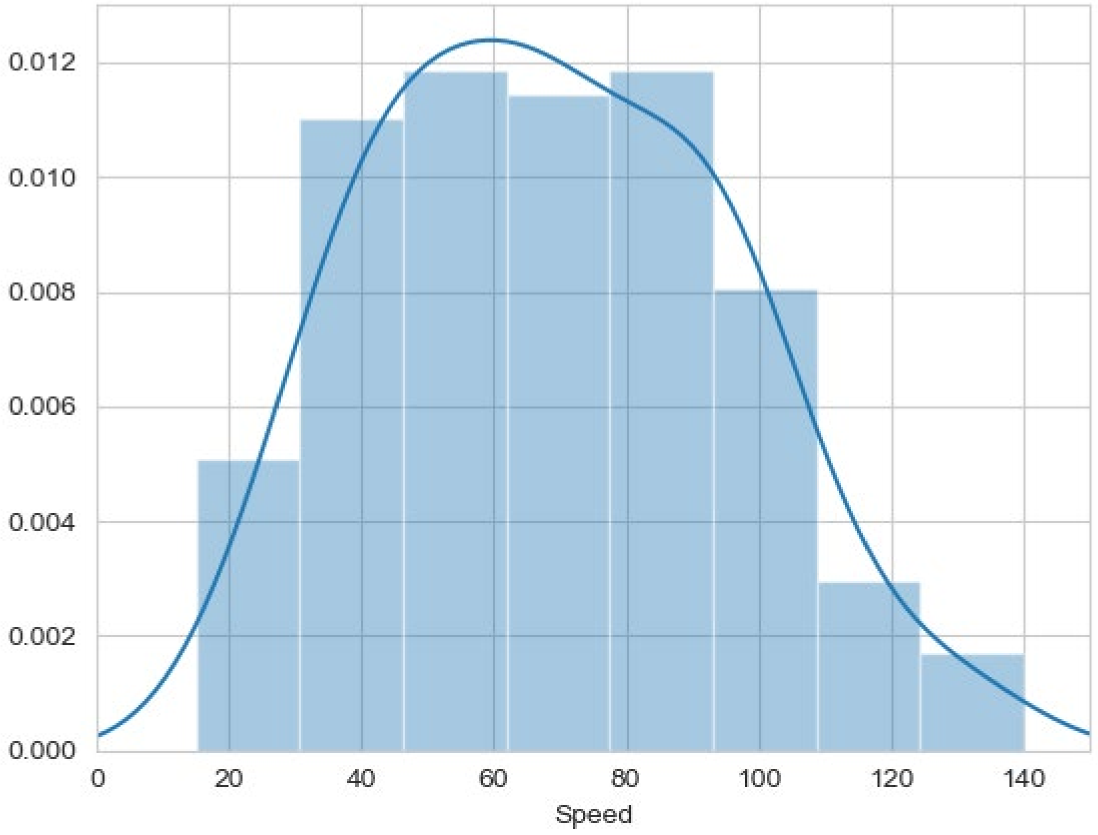
Plotting with Seaborn

One of Seaborn's greatest strengths is its diversity of plotting functions.
Most plots can be created with one line of code.
For example….

Histograms

Allow you to plot the distributions of numeric variables.

```
sns.set_style()
sns.distplot(df1.Speed)
```



Other types of graphs: Creating a scatter plot

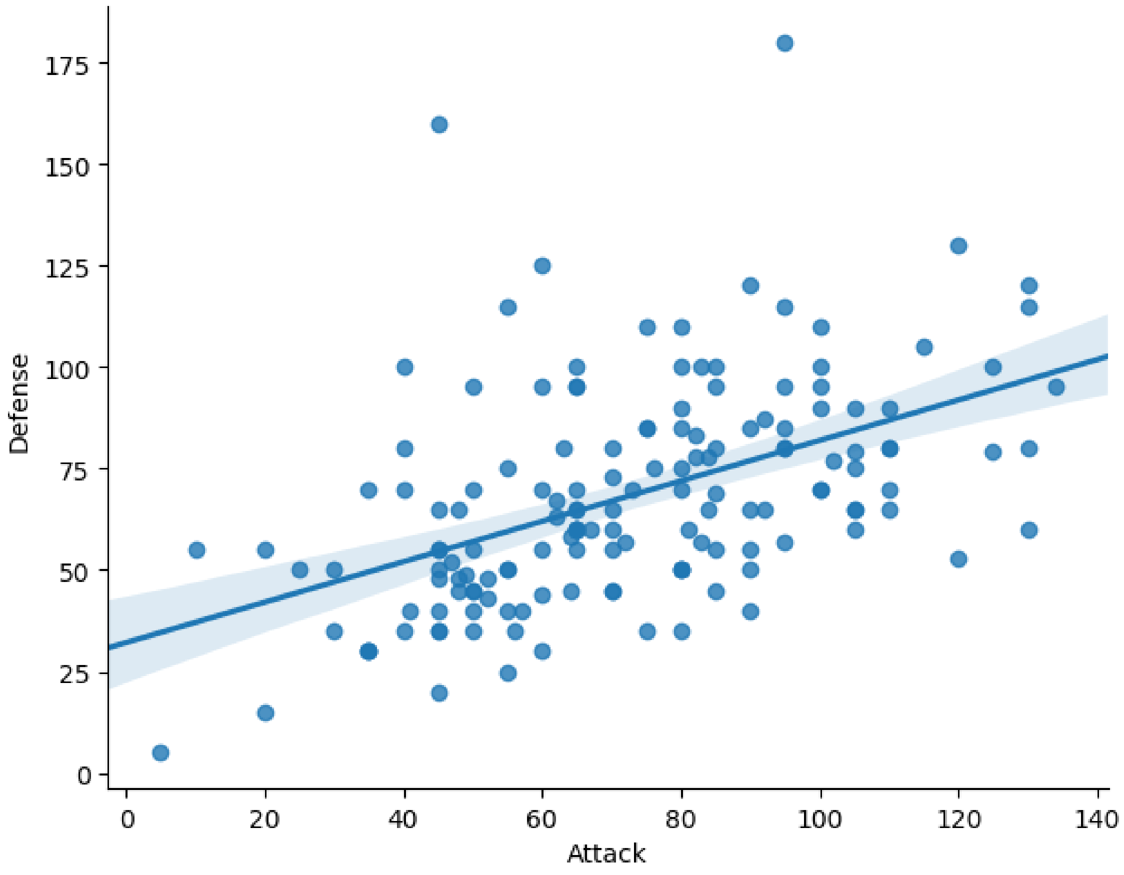
Name of variable we want on the x-axis

Name of our dataframe fed to the “data=“ command

```
sns.lmplot(x='Attack', y='Defense', data=df1)
```

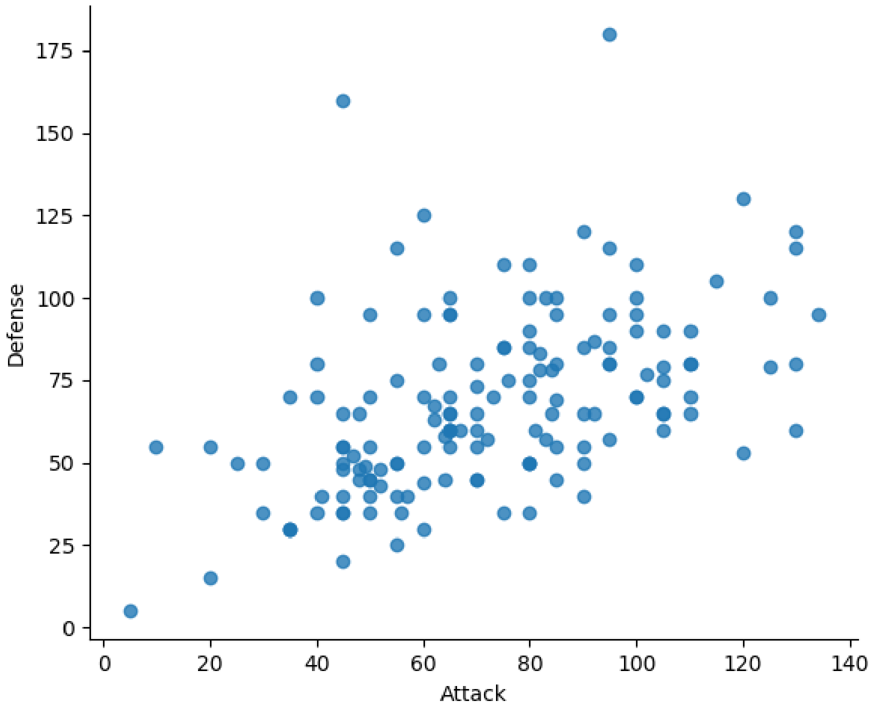
Seaborn “linear model plot” function for creating a scatter graph

Name of variable we want on the y-axis



Seaborn doesn't have a dedicated scatter plot function.
We used Seaborn's function for fitting and plotting a regression line; hence lmplot()
However, Seaborn makes it easy to alter plots.
To remove the regression line, we use the fit_reg=False command

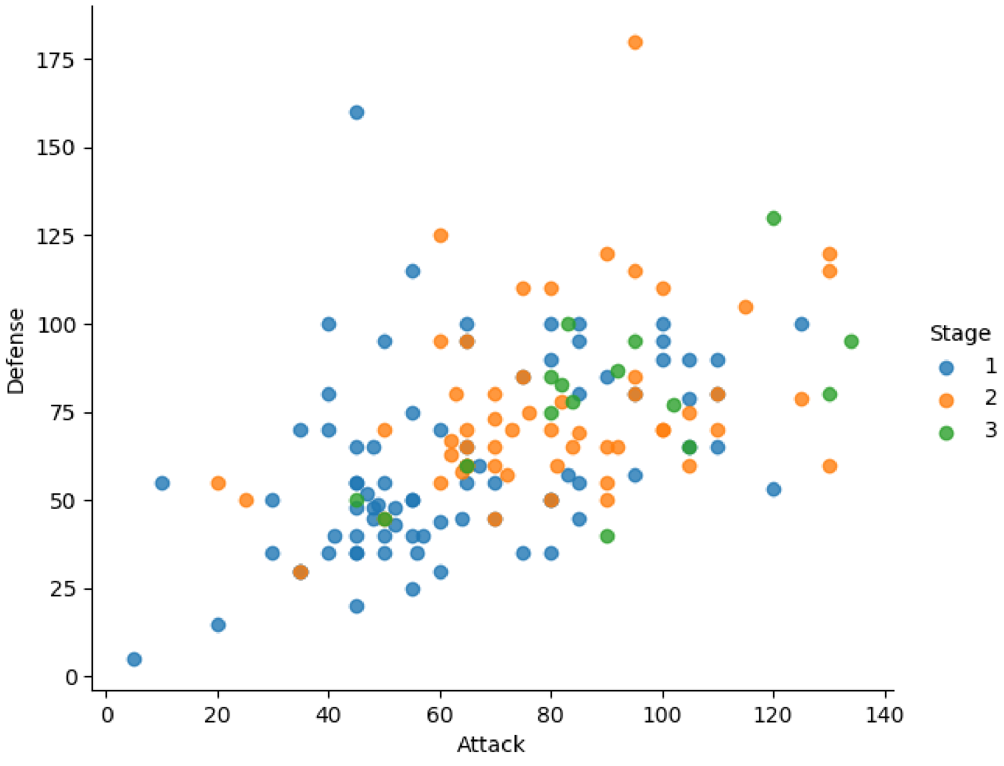
```
sns.lmplot(x='Attack', y='Defense', data=df1, fit_reg=False)
```



The hue function

Another useful function in Seaborn is the hue function, which enables us to use a variable to colour code our data points.

```
sns.lmplot(x='Attack', y='Defense', data=df1,
           fit_reg=False,
           hue='Stage')
```



Factor plots

Make it easy to separate plots by categorical classes.

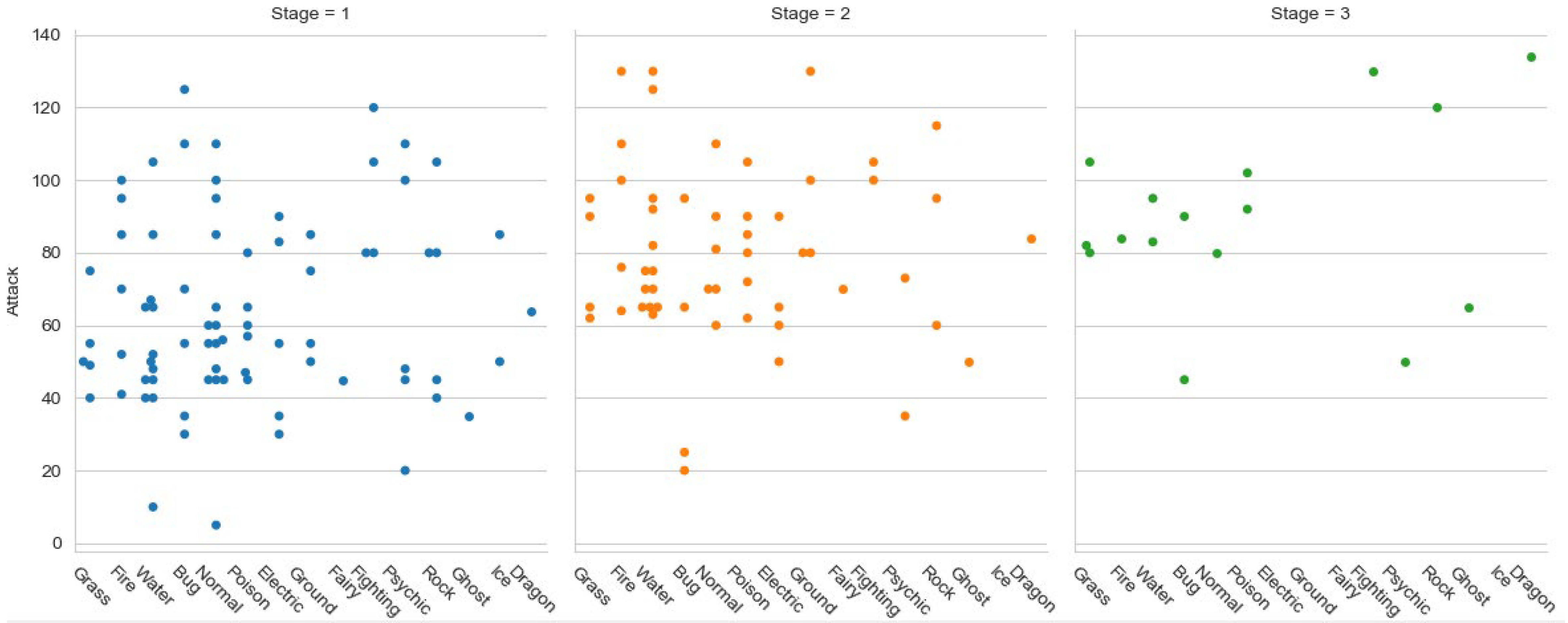
```
g = sns.factorplot(x='Type 1',
                   y='Attack',
                   data=df1,
                   hue='Stage',
                   col='Stage',
                   kind='swarm')
g.set_xticklabels(rotation=-45)
```

Colour by stage.

Separate by stage.

Generate using a swarmplot.

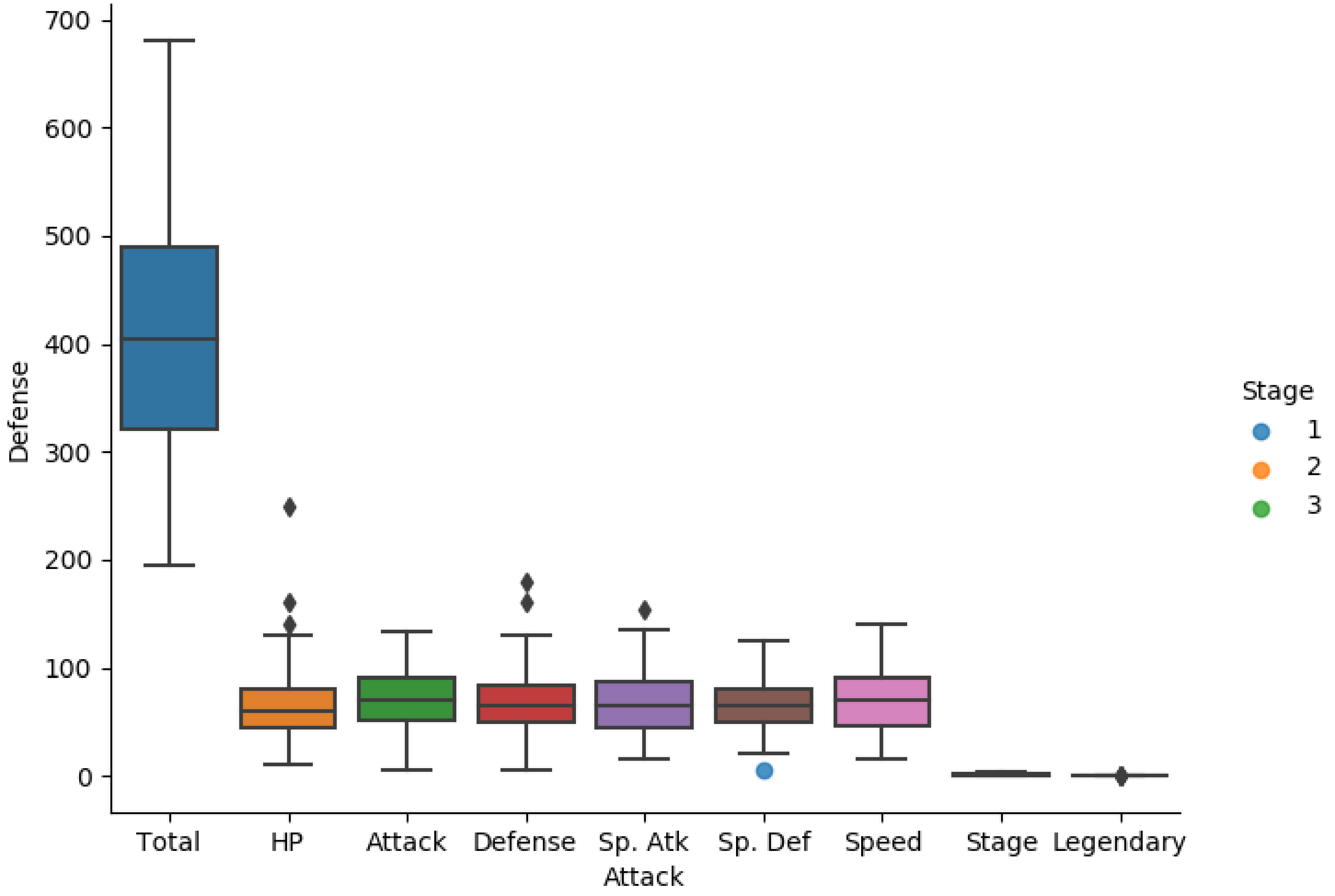
Rotate axis on x-ticks by 45 degrees.



Stage = 1
Stage = 2
Stage = 3
Attack
0
20
40
60
80
100
120
140
Grass
Fire
Water
Bug
Normal
Poison
Electric
Ground
Fairy
Fighting
Psychic
Rock
Ghost
Ice
Dragon

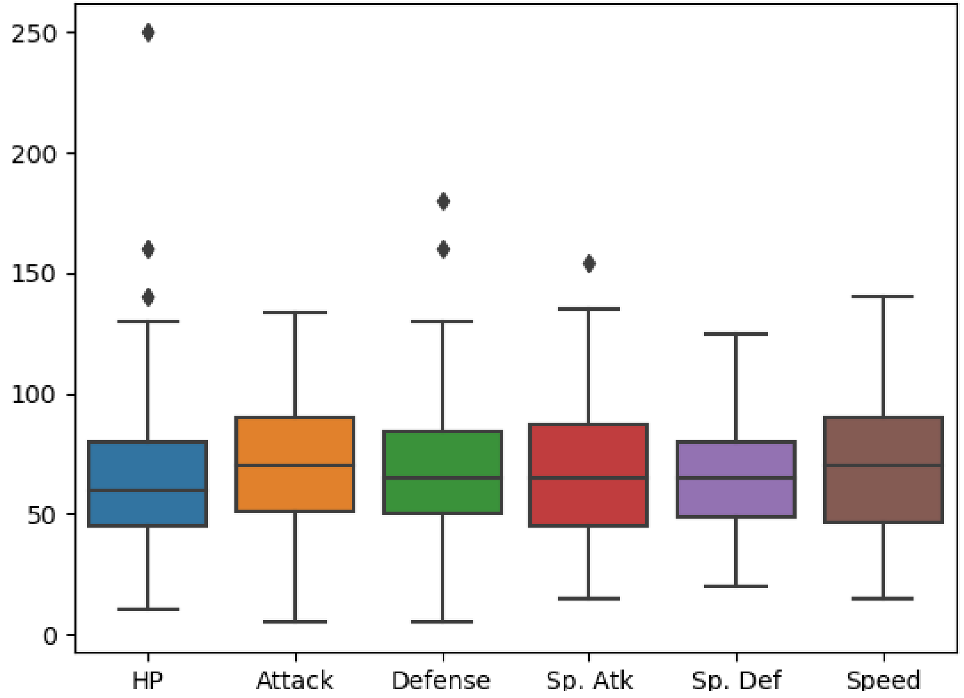
A box plot

```
sns.boxplot(data=df1)
```



The total, stage, and legendary entries are not combat stats so we should remove them. Pandas makes this easy to do, we just create a new dataframe
We just use Pandas' .drop() function to create a dataframe that doesn't include the variables we don't want.

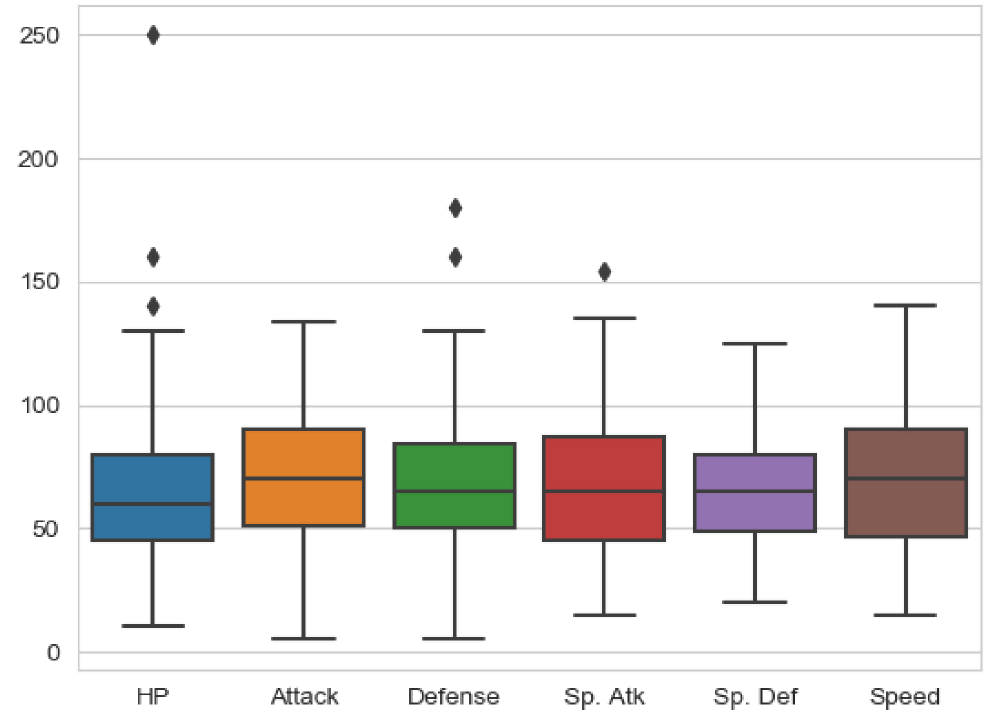
```
stats_df = df1.drop(['Total', 'Stage', 'Legendary'], axis=1)
sns.boxplot(data=stats_df)
```



Seaborn’s theme

Seaborn has a number of themes you can use to alter the appearance of plots. For example, we can use “whitegrid” to add grid lines to our boxplot.

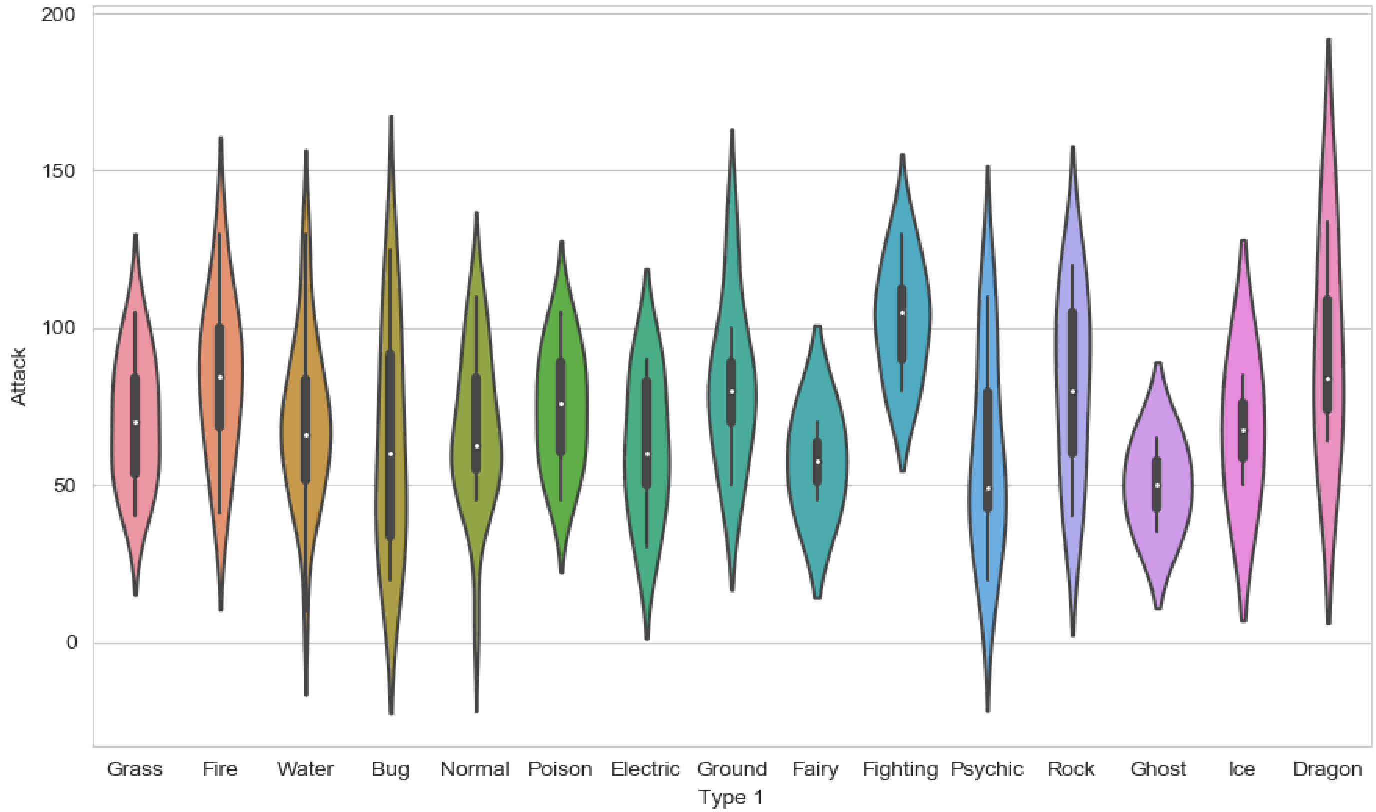
```
stats_df = df1.drop(['Total', 'Stage', 'Legendary'], axis=1)
sns.set_style('whitegrid')
sns.boxplot(data=stats_df)
```



Violin plots

Violin plots are useful alternatives to box plots.
They show the distribution of a variable through the thickness of the violin.
Here, we visualise the distribution of `attack` by Pokémon's primary type:

```
sns.violinplot(x='Type 1', y='Attack', data=df1)
```



- Dragon types tend to have higher Attack stats than Ghost types, but they also have greater variance. But there is something not right here….
- The colours!

Seaborn’s colour palettes

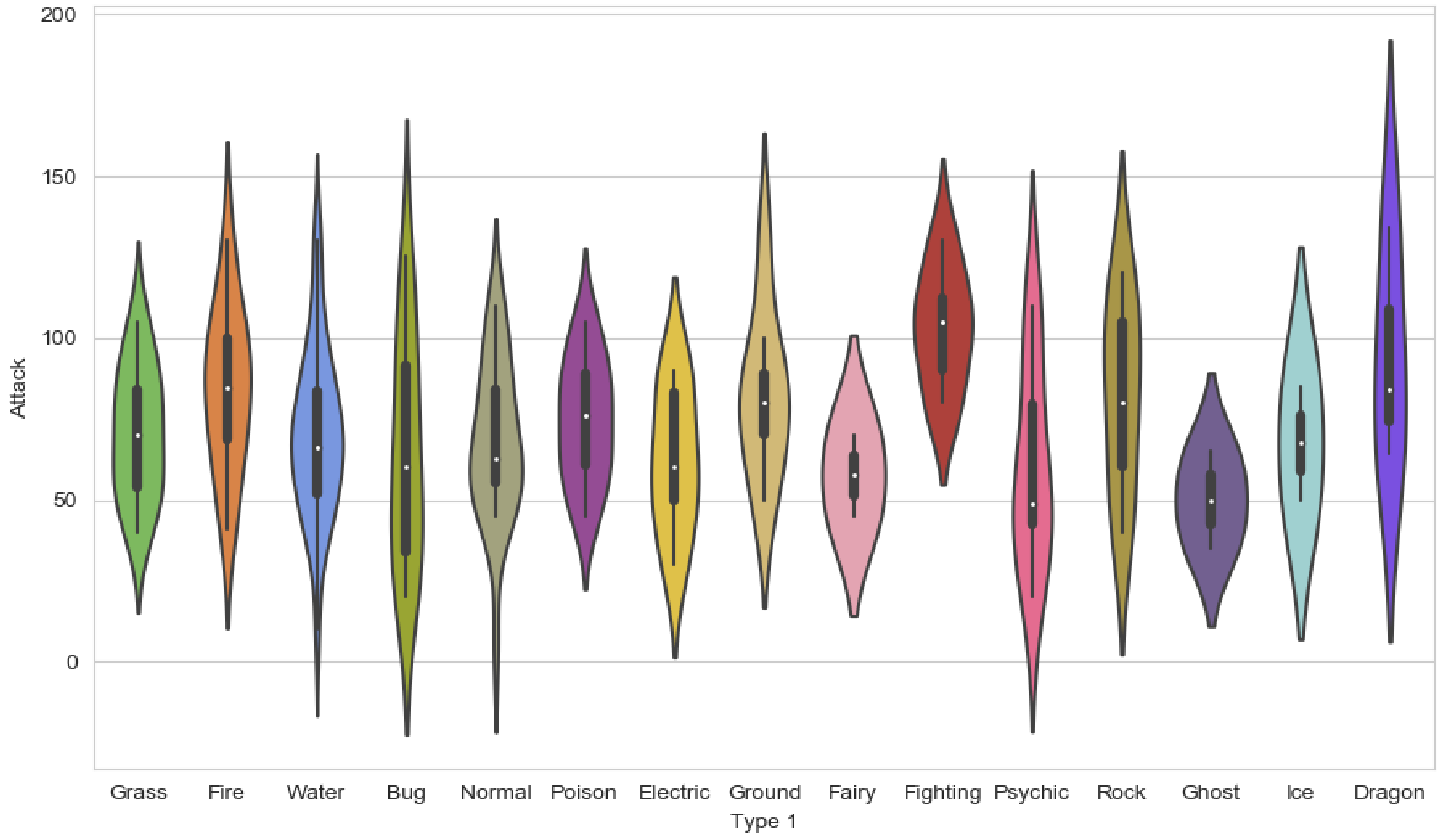
Seaborn allows us to easily set custom colour palettes by providing it with an ordered list of colour hex values.

We first create our colours list.

```
type_colors = ['#78C850',  # Grass
                    '#F08030',  # Fire
                    '#6890F0',  # Water
                    '#A8B820',  # Bug
                    '#A8A878',  # Normal
                    '#A040A0',  # Poison
                    '#F8D030',  # Electric
                    '#E0C068',  # Ground
                    '#EE99AC',  # Fairy
                    '#C03028',  # Fighting
                    '#F85888',  # Psychic
                    '#B8A038',  # Rock
                    '#705898',  # Ghost
                    '#98D8D8',  # Ice
                    '#7038F8',  # Dragon
                   ]
```

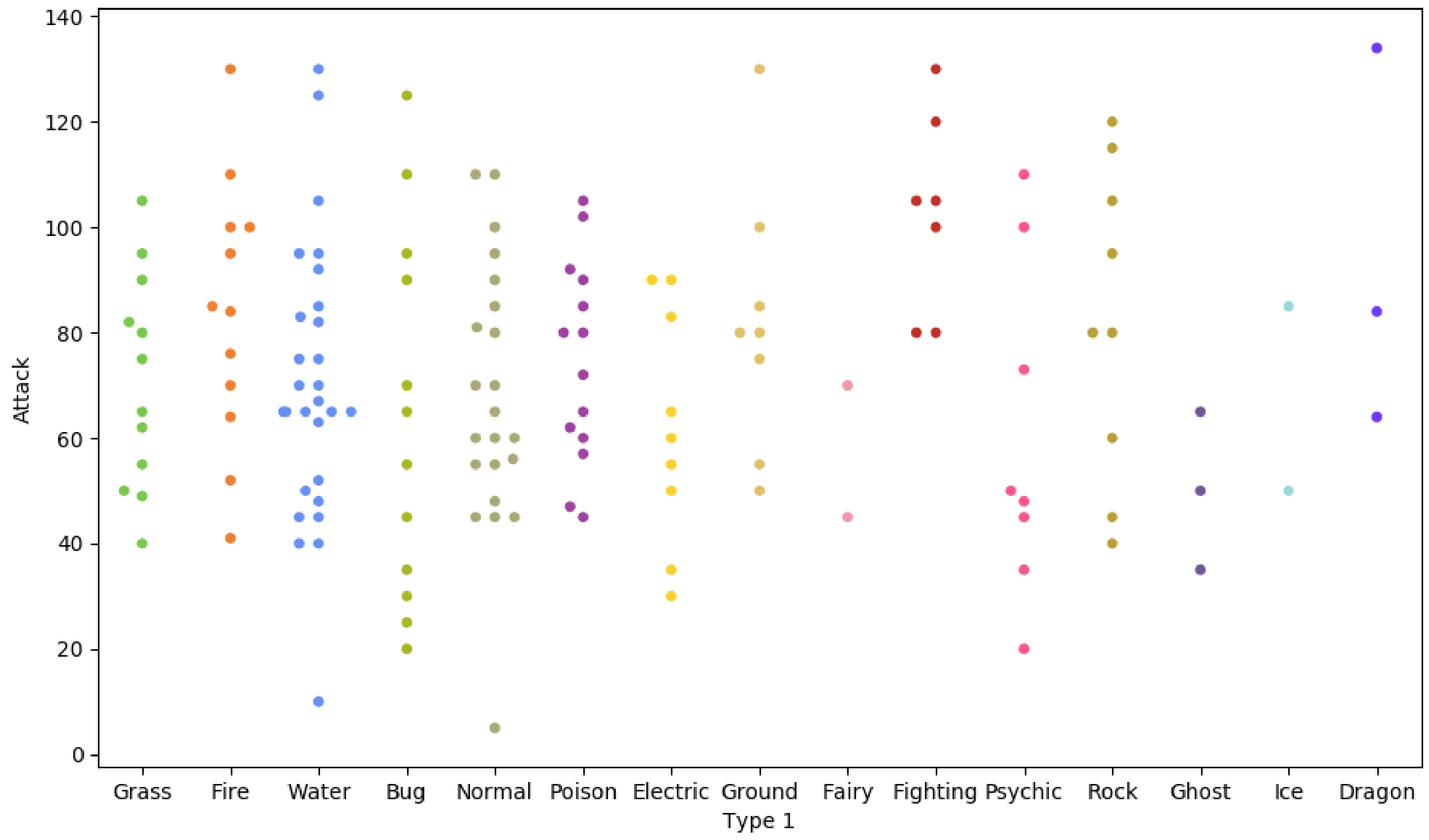
Then we just use the palette= function and feed in our colours list.

```
sns.violinplot(x='Type 1', y='Attack', data=df1,
               palette=type_colors)
```



Because of the limited number of observations, we could also use a swarm plot. Here, each data point is an observation, but data points are grouped together by the variable listed on the x-axis.

```
sns.swarmplot(x='Type 1', y='Attack',
              data=df1,
              palette=type_colors)
```



Overlapping plots

Both of these show similar information, so it might be useful to overlap them.

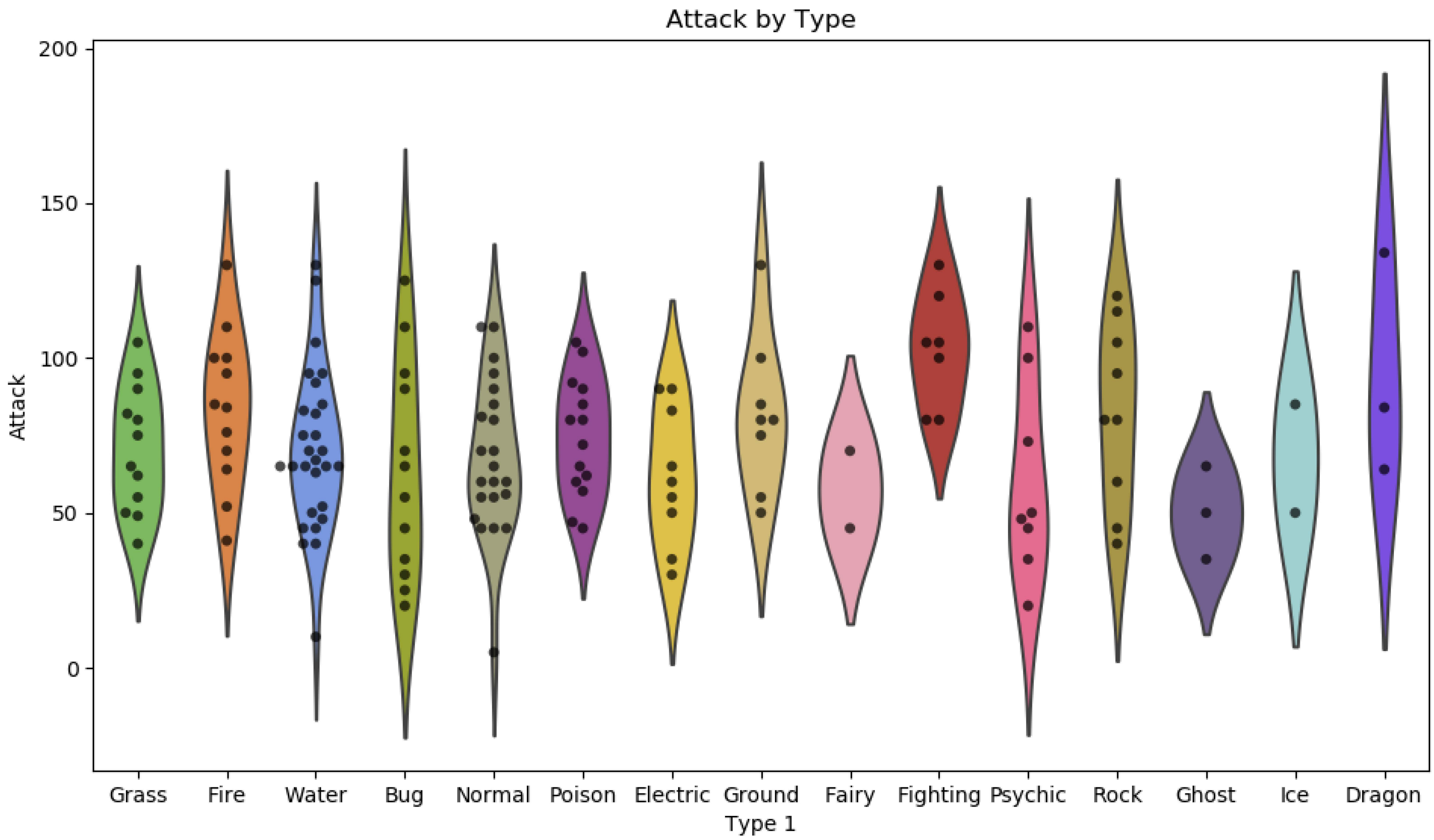
```
plt.figure(figsize=(10,6))
sns.violinplot(x='Type 1',
               y='Attack',
               data=df1,
               inner=None,
               palette=type_colors)
sns.swarmplot(x='Type 1',
              y='Attack',
              data=df1,
              color='k',
              alpha=0.7)
plt.title('Attack by Type')
```

Set size of print canvas.

Remove bars from inside the violins

Make bars black and slightly transparent

Give the graph a title


Attack by Type
Attack
Type 1
200
150
100
50
0
Grass
Fire
Water
Bug
Normal
Poison
Electric
Ground
Fairy
Fighting
Psychic
Rock
Ghost
Ice
Dragon

Data wrangling with Pandas

What if we wanted to create such a plot that included all of the other stats as well?
In our current dataframe, all of the variables are in different columns:

```
print(df1.head())
```

```
         Name Type 1  Type 2  Total  ...  Sp. Def  Speed  Stage  Legendary
#                                    ...
1   Bulbasaur  Grass  Poison    318  ...       65     45      1      False
2     Ivysaur  Grass  Poison    405  ...       80     60      2      False
3    Venusaur  Grass  Poison    525  ...      100     80      3      False
4  Charmander   Fire     NaN    309  ...       50     65      1      False
5  Charmeleon   Fire     NaN    405  ...       65     80      2      False
```

If we want to visualise all stats, then we'll have to "melt" the dataframe.

```
stats_df = df1.drop(['Total', 'Stage', 'Legendary'], axis=1)
melted_df = pd.melt(stats_df,
                    id_vars=["Name", "Type 1", "Type 2"],
                    var_name="Stat")
print(melted_df.head())
```

We use the .drop() function again to re-create the dataframe without these three variables.

The dataframe we want to melt.

The variables to keep, all others will be melted.

A name for the new, melted, variable.

```
In [6]: runfile('C:/Users/lb690/Google Dri
pokemon_tutorial.py', wdir='C:/Users/lb69
         Name Type 1  Type 2 Stat  value
0   Bulbasaur  Grass  Poison   HP     45
1     Ivysaur  Grass  Poison   HP     60
2    Venusaur  Grass  Poison   HP     80
3  Charmander   Fire     NaN   HP     39
4  Charmeleon   Fire     NaN   HP     58
```

- All 6 of the stat columns have been "melted" into one, and the new Stat column indicates the original stat (HP, Attack, Defense, Sp. Attack, Sp. Defense, or Speed).
- It's hard to see here, but each pokemon now has 6 rows of data; hende the melted_df has 6 times more rows of data.

```
print( stats_df.shape )
print( melted_df.shape )
```

```
(151, 9)
(906, 5)
```

```python
sns.swarmplot(x='Stat', y='value', data=melted_df,
              hue='Type 1')
```

This graph could be made to look nicer with a few tweaks.

```
plt.figure(figsize=(10,6))
sns.swarmplot(x='Stat',
              y='value',
              data=melted_df,
              hue='Type 1',
              split=True,
              palette=type_colors)
plt.ylim(0, 260)
plt.legend(bbox_to_anchor=(1, 1), loc=2)
```

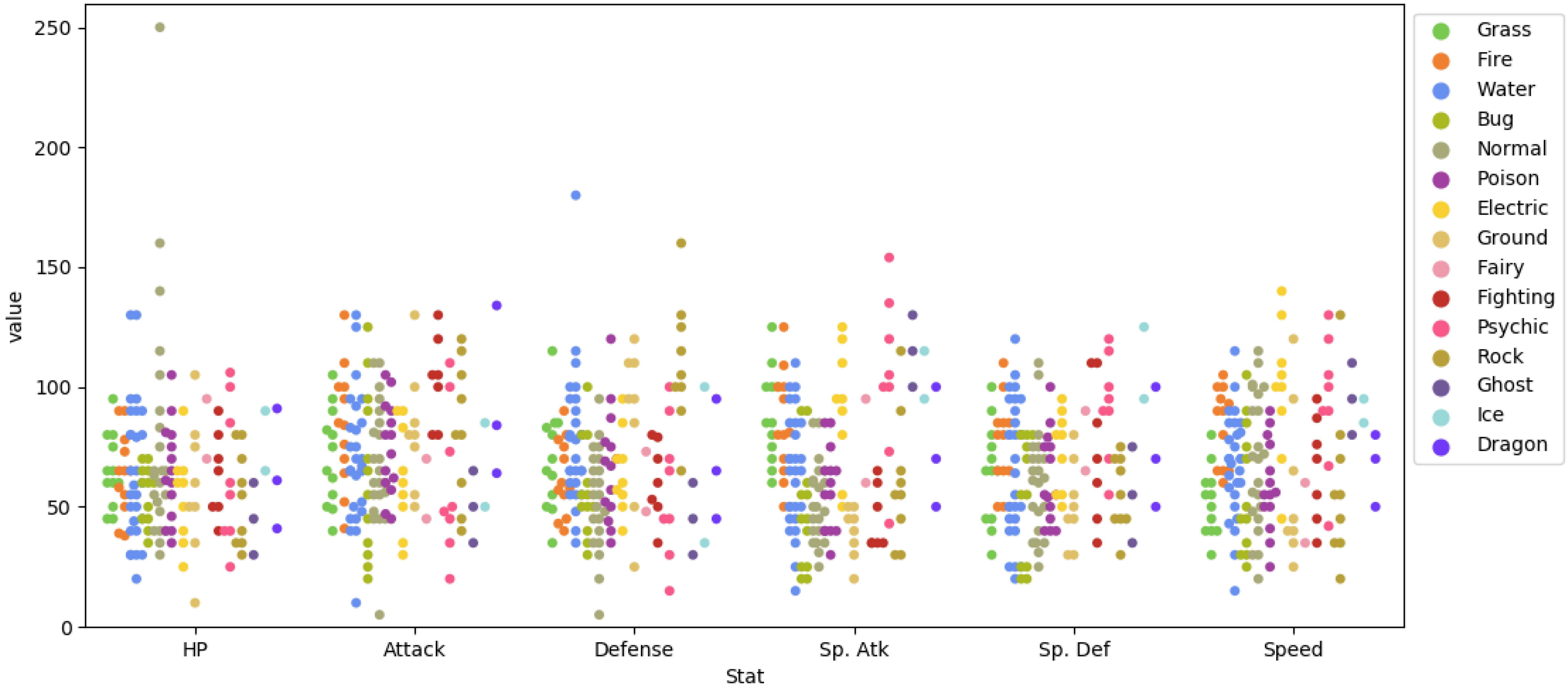
Enlarge the plot.

Separate points by hue.

Use our special Pokemon colour palette

Adjust the y-axis.

Move the legend box outside of the graph and place to the right of it..

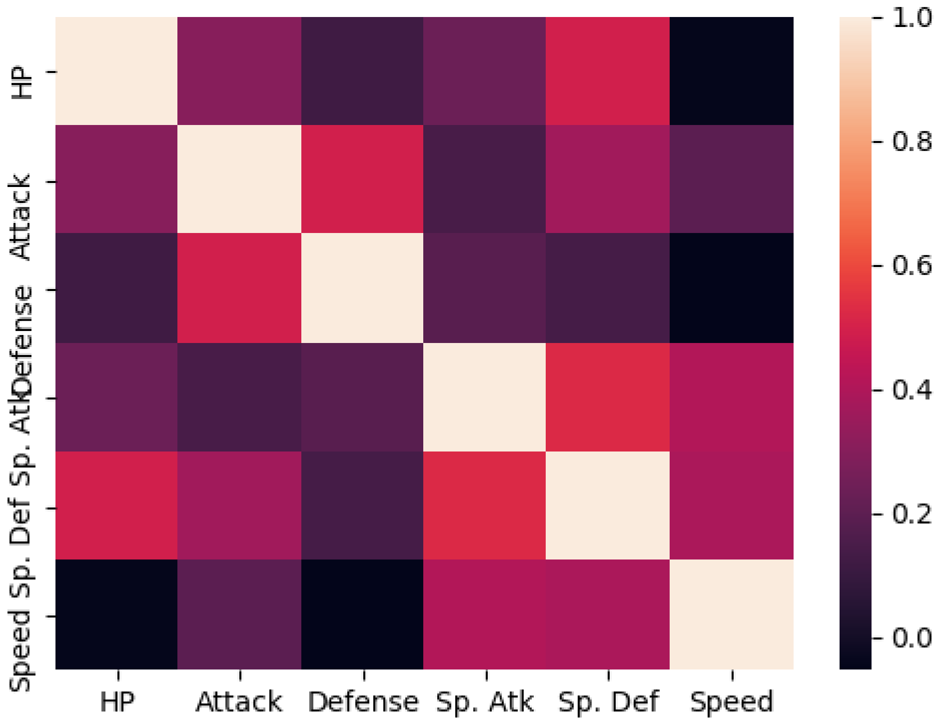


value
250
200
150
100
50
0
HP
Attack
Defense
Sp. Atk
Sp. Def
Speed
Stat
Grass
Fire
Water
Bug
Normal
Poison
Electric
Ground
Fairy
Fighting
Psychic
Rock
Ghost
Ice
Dragon

Heatmaps

Useful for visualising matrix-like data.
Here, we'll plot the correlation of the stats_df variables

```
corr = stats_df.corr()
sns.heatmap(corr)
```

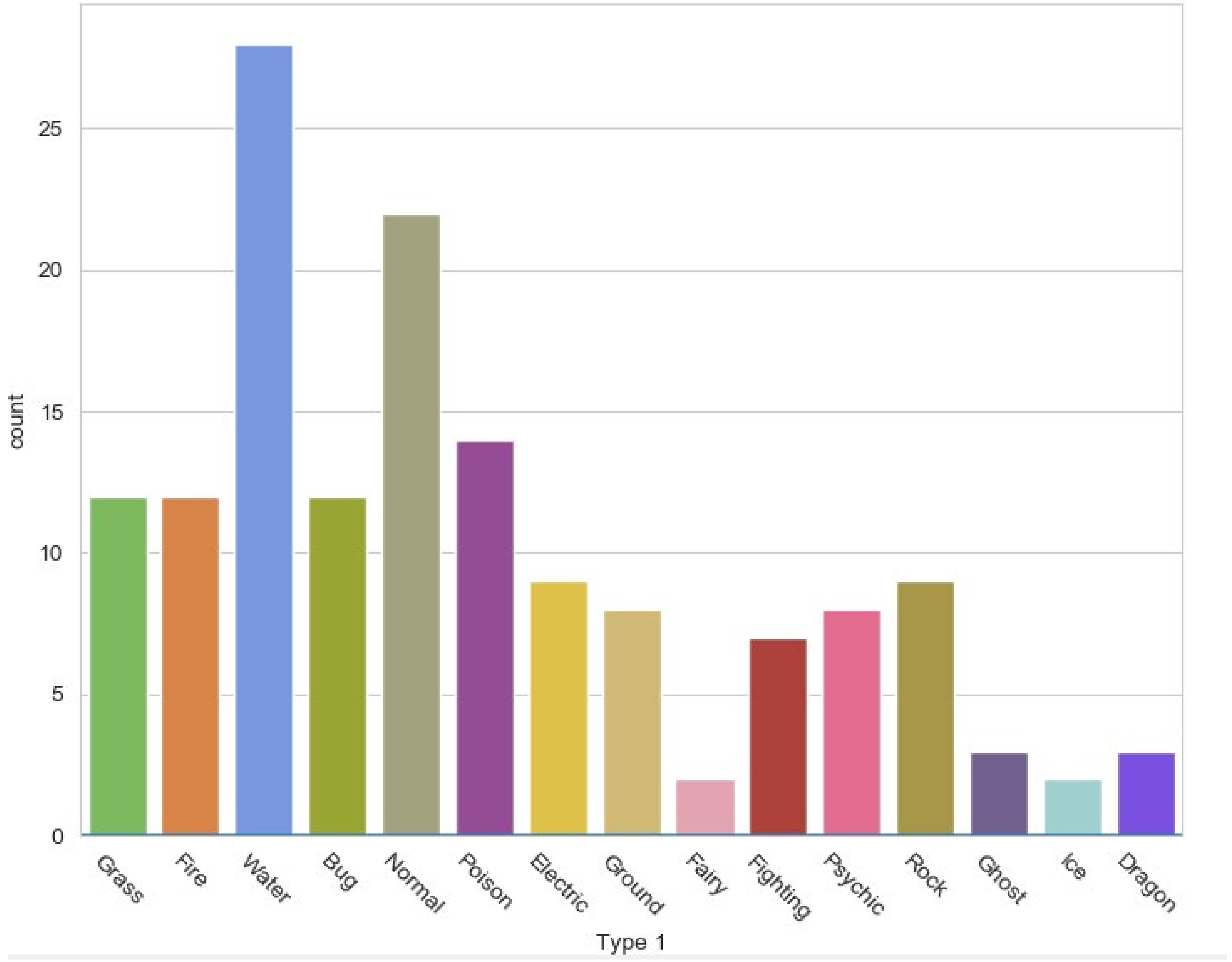


Bar plot

Visualises the distributions of categorical variables.

```
sns.countplot(x='Type 1', data=df1,
              palette=type_colors)
plt.xticks(rotation=-45)
```

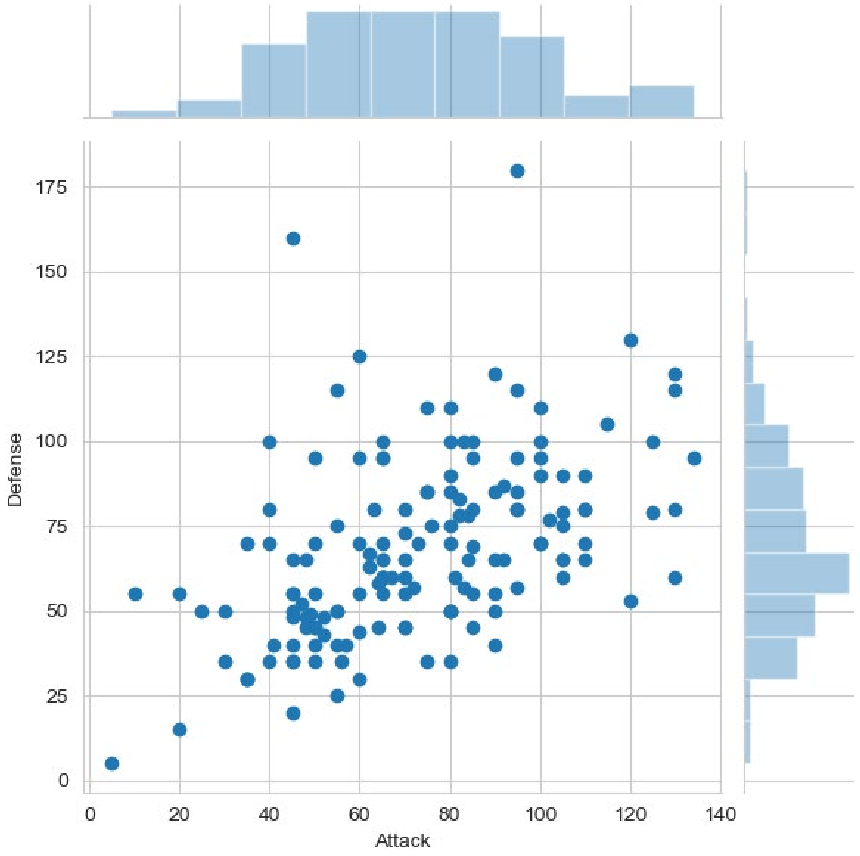
Rotates the x-ticks 45 degrees



Joint Distribution Plot

Joint distribution plots combine information from scatter plots and histograms to give you detailed information for bi-variate distributions.

```
sns.jointplot(x='Attack',
              y='Defense',
              data=df1)
```



Thanks