Data Visualization using Python, Matplotlib and Seaborn

Part 8 of "Data Analysis with Python: Zero to Pandas"

This tutorial is the eighth in a series on introduction to programming and data analysis using the Python language. These tutorials take a practical coding-based approach, and the best way to learn the material is to execute the code and experiment with the examples. Check out the full series here:

1. First Steps with Python and Jupyter
2. A Quick Tour of Variables and Data Types
3. Branching using Conditional Statements and Loops
4. Writing Reusable Code Using Functions
5. Reading from and Writing to Files
6. Numerical Computing with Python and Numpy
7. Analyzing Tabular Data using Pandas
8. Data Visulation using Matplotlib & Seaborn

How to run the code

This tutorial hosted on Jovian.ml, a platform for sharing data science projects online. You can "run" this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your own computer*.

> This tutorial is a Jupyter notebook - a document made of "cells", which can contain explanations in text or code written in Python. Code cells can be executed and their outputs e.g. numbers, messages, graphs, tables, files etc. can be viewed within the notebook, which makes it a really powerful platform for experimentation and analysis. Don't afraid to experiment with the code & break things - you'll learn a lot by encoutering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top of the notebook.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing this notebook is to click the "Run" button at the top of this page, and select "Run on Binder". This will run the notebook on mybinder.org, a free online service for running Jupyter notebooks. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on Google Colab or Kaggle to use these platforms.

Option 2: Running on your computer locally

You'll need to install Python and download this notebook on your computer to run in locally. We recommend using the Conda distribution of Python. Here's what you need to do to get started:

1. Install Conda by following these instructions. Make sure to add Conda binaries to your system `PATH` to be able to run the `conda` command line tool from your Mac/Linux terminal or Windows command prompt.
2. Create and activate a Conda virtual environment called `zerotopandas` which you can use for this tutorial series:

   ```
   conda create -n zerotopandas -y python=3.8
   conda activate zerotopandas
   ```

   You'll need to create the environment only once, but you'll have to activate it every time want to run the notebook. When the environment is activated, you should be able to see a prefix `(python-matplotlib-data-visualization)` within your terminal or command prompt.
3. Install the required Python libraries within the environmebt by the running the following command on your terminal or command prompt:

   ```
   pip install jovian jupyter numpy pandas matplotlib seaborn --upgrade
   ```

4. Download the notebook for this tutorial using the `jovian clone` command:

   ```
   jovian clone aakashns/python-matplotlib-data-visualization
   ```

   The notebook is downloaded to the directory `python-matplotlib-data-visualization`.
5. Enter the project directory and start the Jupyter notebook:

   ```
   cd python-matplotlib-data-visualization
   jupyter notebook
   ```

6. You can now access Jupyter's web interface by clicking the link that shows up on the terminal or by visiting http://localhost:8888 on your browser. Click on the notebook `python-matplotlib-data-visualization.ipynb` to open it and run the code. If you want to type out the code yourself, you can also create a new notebook using the "New" button.

Introduction

Data visualization is the graphic representation of data. It involves producing images that communicate relationships among the represented data to viewers. Visualizing data is an essential part of data analysis and machine learning. In this tutorial, we'll use Python libraries Matplotlib and Seaborn to learn and apply some popular data visualization techniques.

To begin let's import the libraries. We'll use the `matplotlib.pyplot` for basic plots like line & bar charts. It is often imported with the alias `plt`. The `seaborn` module will be used for more advanced plots, and it is imported with the alias `sns`.

```
# Uncomment the next line to install the required libraries
# !pip install matplotlib seaborn --upgrade --quiet
```

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Notice this we also include the special command `%matplotlib inline` to ensure that plots are shown and embedded within the Jupyter notebook itself. Without this command, sometimes plots may show up in pop-up windows.

Line Chart

Line charts are one of the simplest and most widely used data visualization techniques. A line chart displays information as a series of data points or markers, connected by straight lines. You can customize the shape, size, color and other aesthetic elements of the markers and lines for better visual clarity.

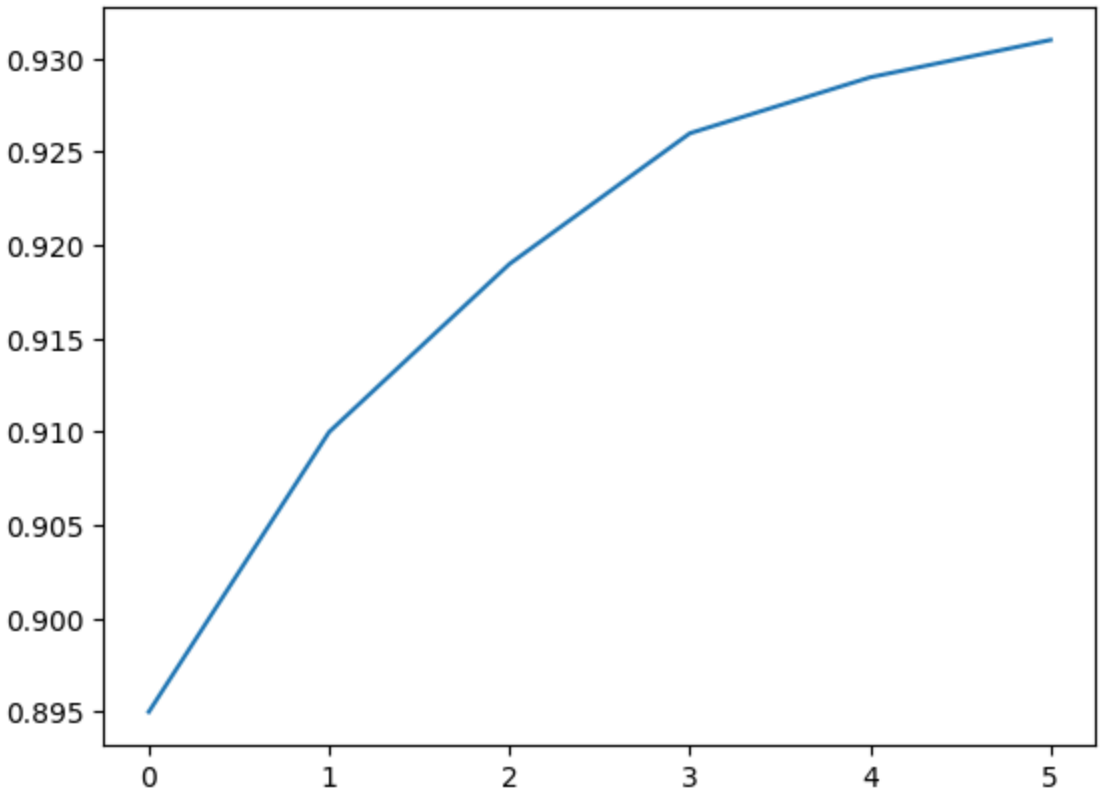
Here's a Python list showing the yield of apples (tons per hectare) over 6 years in an imaginary country called Kanto.

```
yield_apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931]
```

We can visualize how the yield of apples changes over time using a line chart. To draw a line chart, we can use the `plt.plot` function.

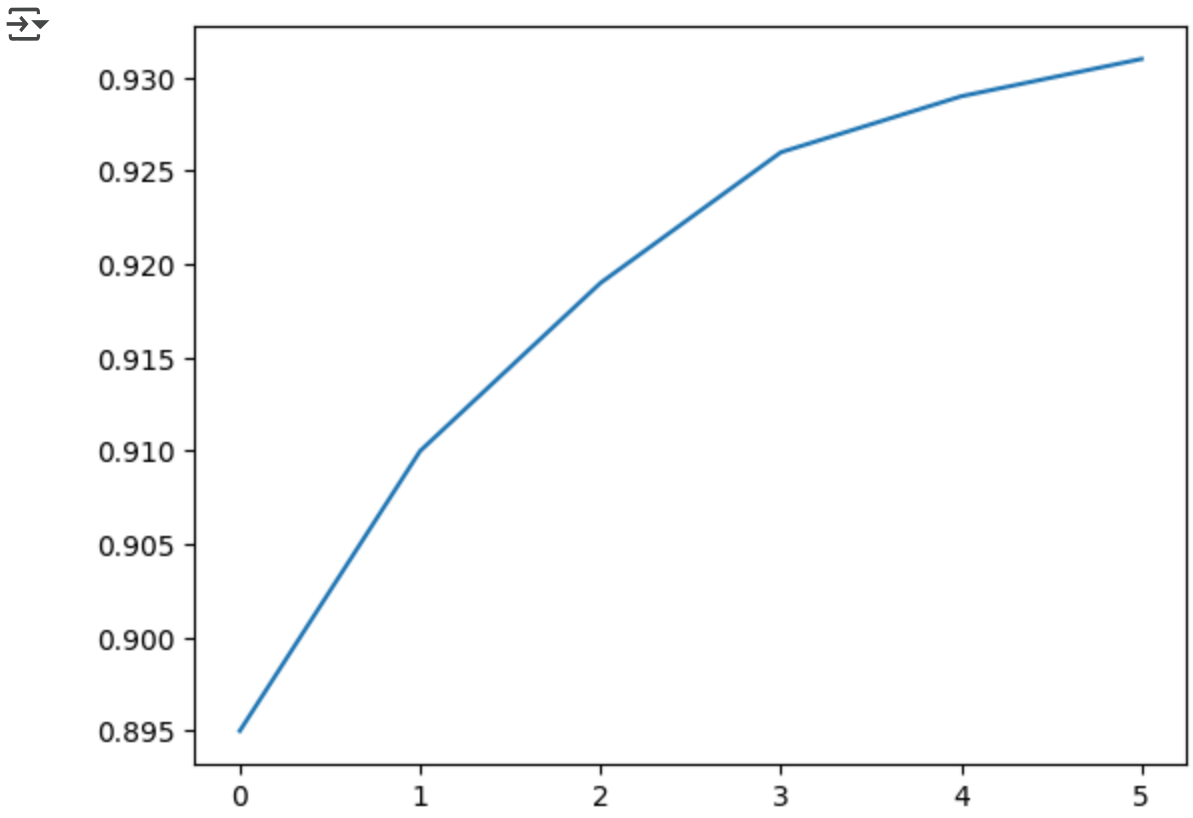
```
plt.plot(yield_apples)
```

```
[<matplotlib.lines.Line2D at 0x7cf3894eb950>]
```



Calling the `plt.plot` function draws the line chart as expected, and also returns a list of plots drawn `[<matplotlib.lines.Line2D at 0x7ff70aa20760>]` shown within the output. We can include a semicolon (`;`) at the end of the last statement in the cell to avoiding showing the output and just display the graph.

```
plt.plot(yield_apples);
```



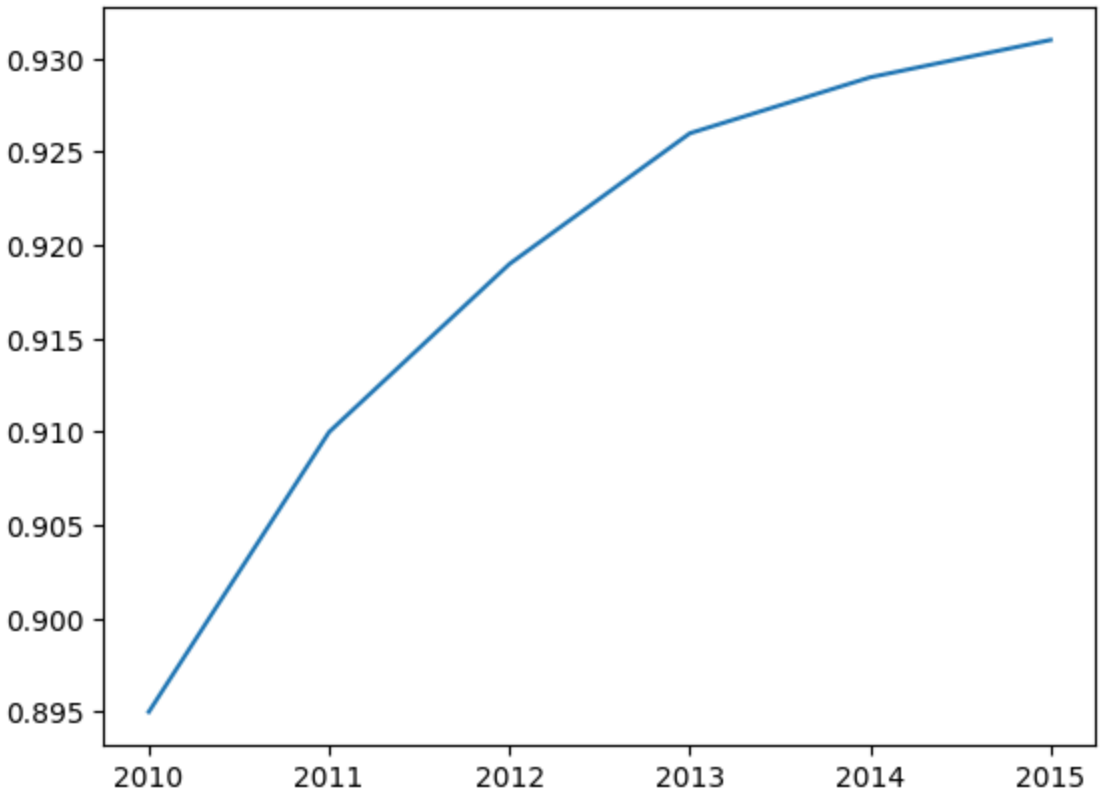
Let's enhance this plot step-by-step to make it more informative and beautiful.

Customizing the X-axis

The X-axis of the plot currently shows list element indexes 0 to 5. The plot would be more informative if we could show the year for which the data is being plotted. We can do this by two arguments `plt.plot`.

```
years = [2010, 2011, 2012, 2013, 2014, 2015]
yield_apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931]


plt.plot(years, yield_apples)
```

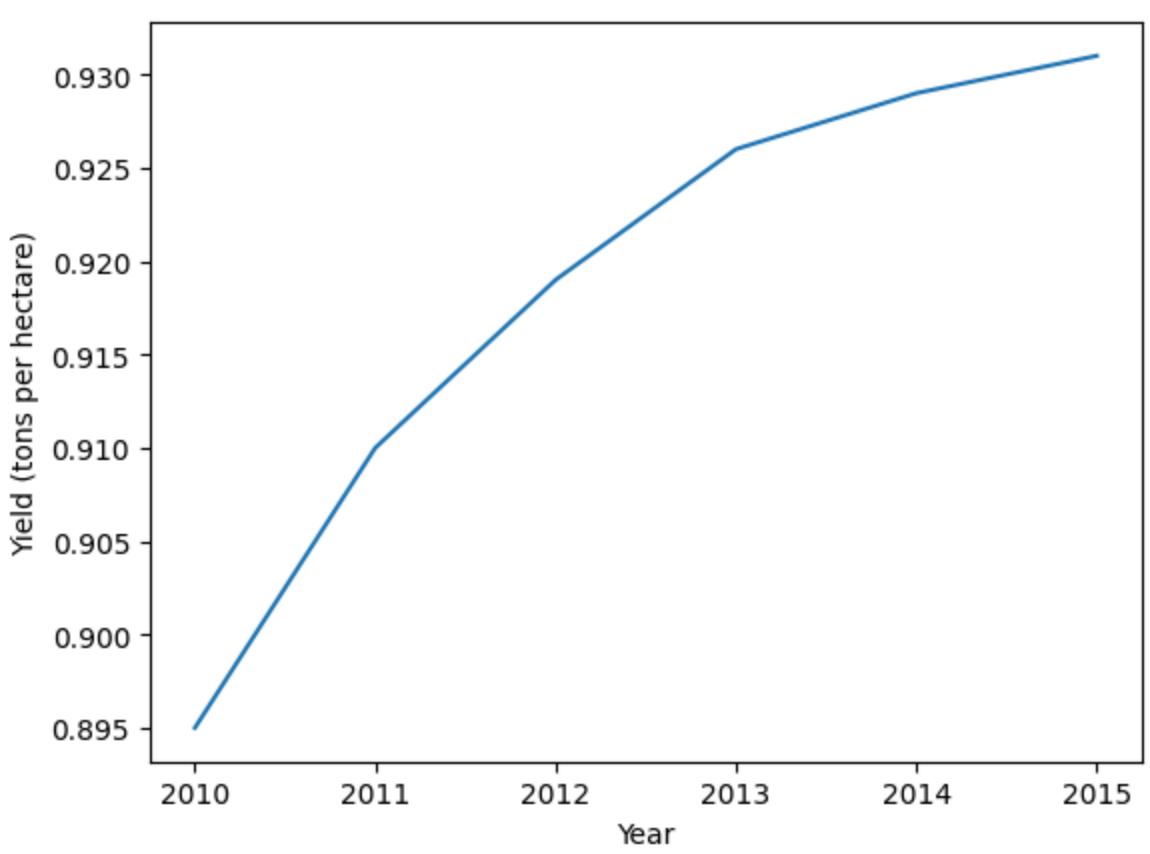
```
[<matplotlib.lines.Line2D at 0x7cf33b462010>]
```



Axis Labels

We can add labels to the axes to show what each axis represents using the `plt.xlabel` and `plt.ylabel` methods.

```
plt.plot(years, yield_apples)
plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)');
```



Plotting Multiple Lines

It's really easy to plot multiple lines in the same graph. Just invoke the `plt.plot` function multiple times. Let's compare the yields of apples vs. oranges in Kanto.

```
years = range(2000, 2012)
apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931, 0.934, 0.936, 0.937, 0.9375, 0.93
oranges = [0.962, 0.941, 0.930, 0.923, 0.918, 0.908, 0.907, 0.904, 0.901, 0.898, 0.9
```

```
plt.plot(years, apples)
plt.plot(years, oranges)
plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)');
```

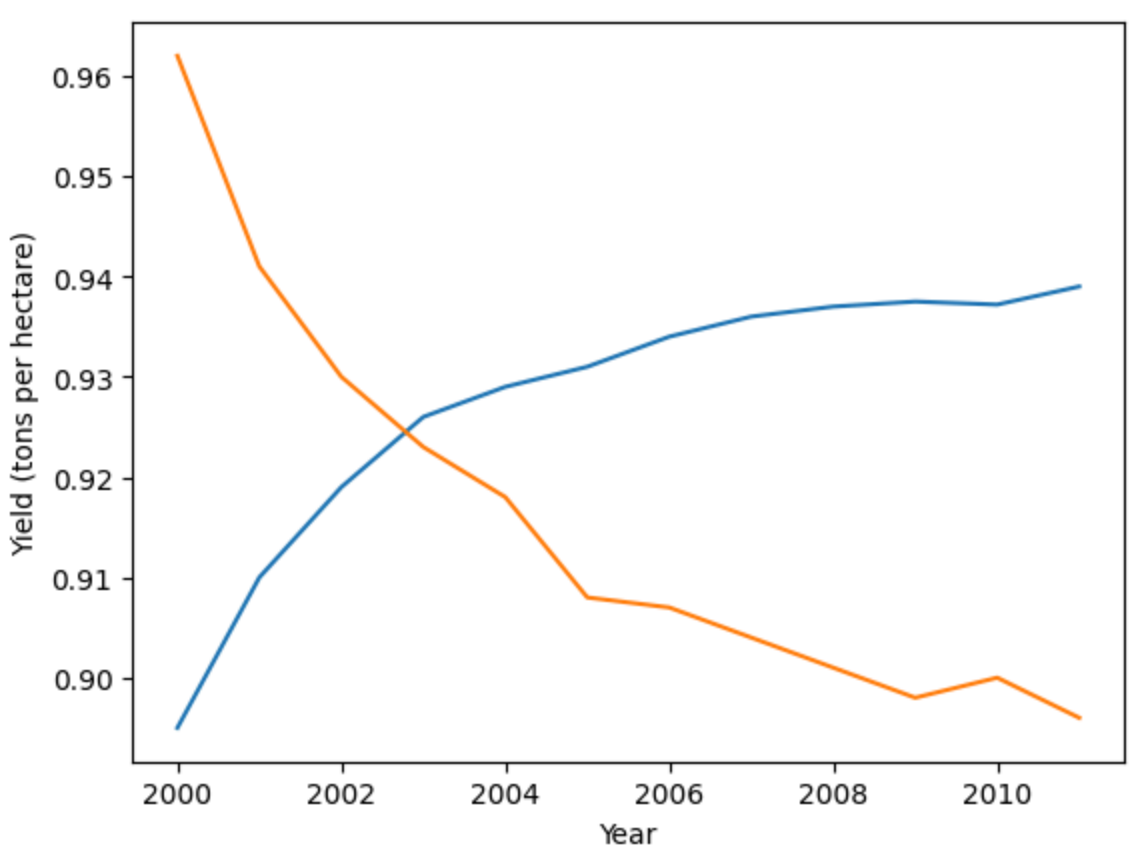
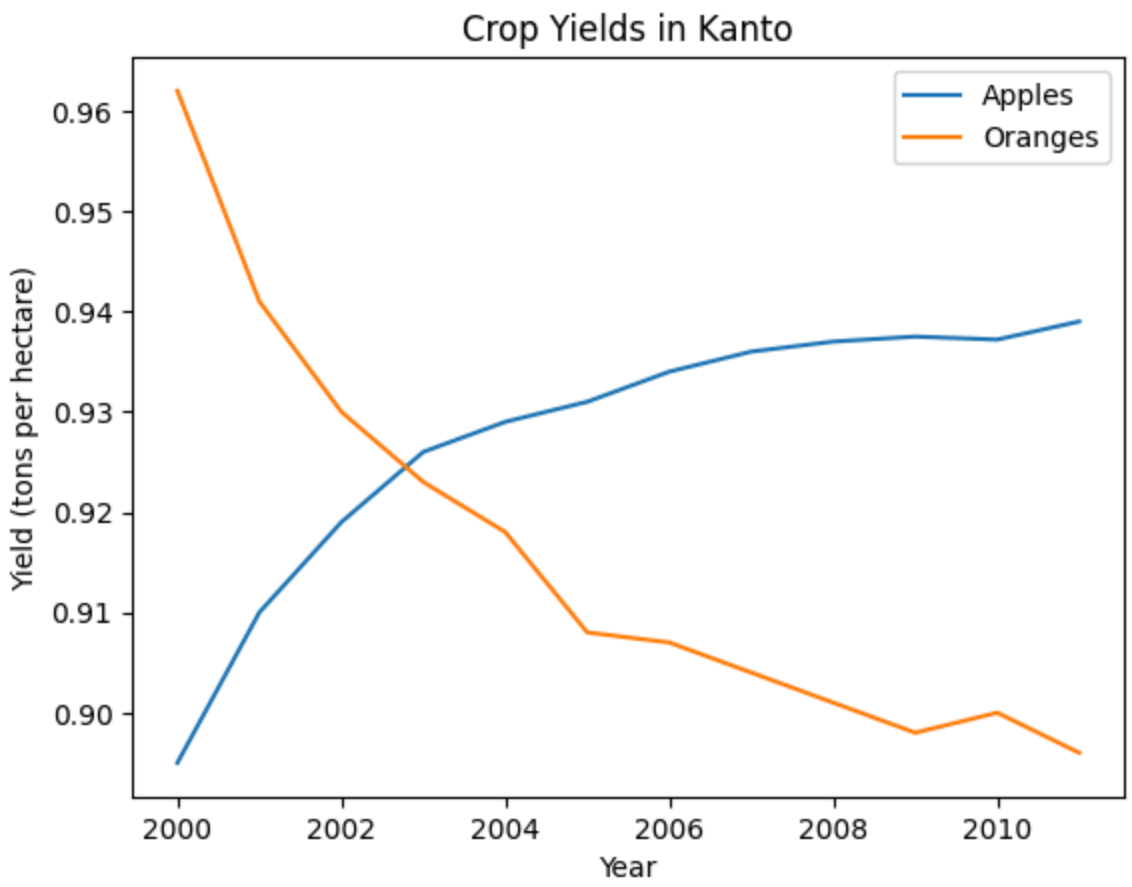


Chart Title and Legend

To differentiate between multiple lines, we can include a legend within the graph using the `plt.legend` function. We also give the entire chart a title using the `plt.title` function.

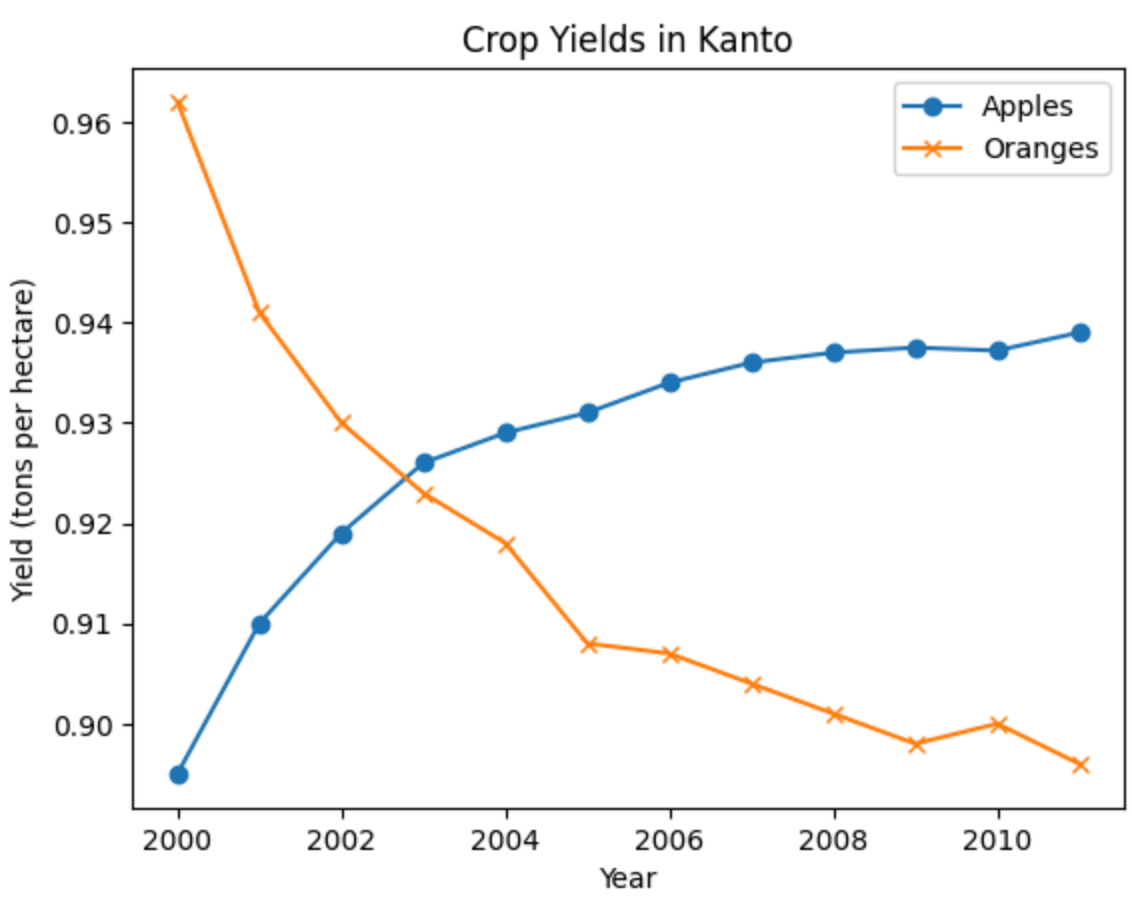
```
plt.plot(years, apples)
plt.plot(years, oranges)

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



Line Markers

We can also show markers for the data points on each line using the `marker` argument of `plt.plot`. Matplotlib supports many different types of markers like circle, cross, square, diamond etc. You can find the full list of marker types here: https://matplotlib.org/3.1.1/api/markers_api.html

```
plt.plot(years, apples, marker='o')
plt.plot(years, oranges, marker='x')

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



Styling lines and markers

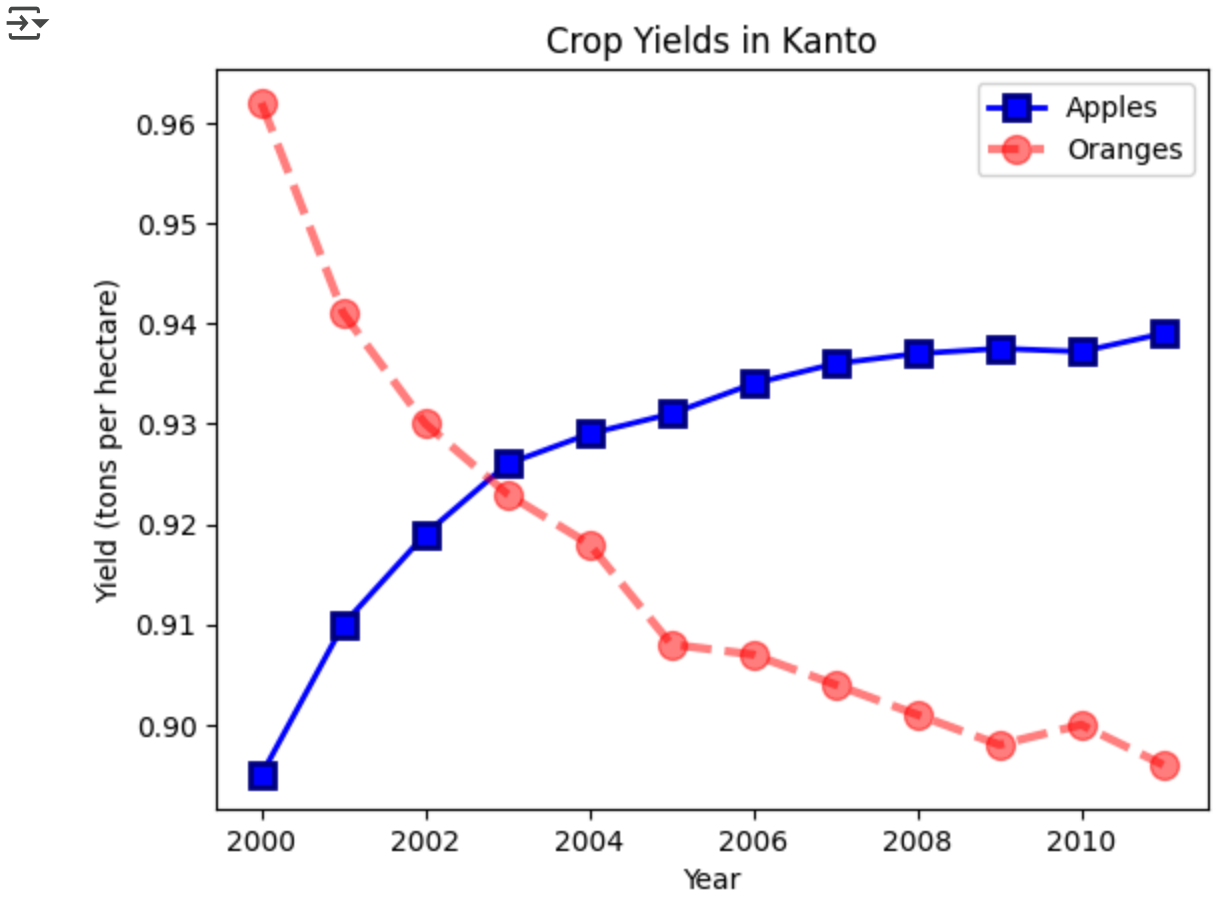
The `plt.plot` function supports many arguments for styling lines and markers:

- `color` or `c`: set the color of the line (supported colors)
- `linestyle` or `ls`: choose between a solid or dashed line
- `linewidth` or `lw`: set the width of a line
- `markersize` or `ms`: set the size of markers
- `markeredgecolor` or `mec`: set the edge color for markers
- `markeredgewidth` or `mew`: set the edge width for markers
- `markerfacecolor` or `mfc`: set the fill color for markers
- `alpha`: opacity of the plot

Check out the documentation for `plt.plot` to learn more:
https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot

```
plt.plot(years, apples, marker='s', c='b', ls='-', lw=2, ms=8, mew=2, mec='navy')
plt.plot(years, oranges, marker='o', c='r', ls='--', lw=3, ms=10, alpha=.5)

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')
```

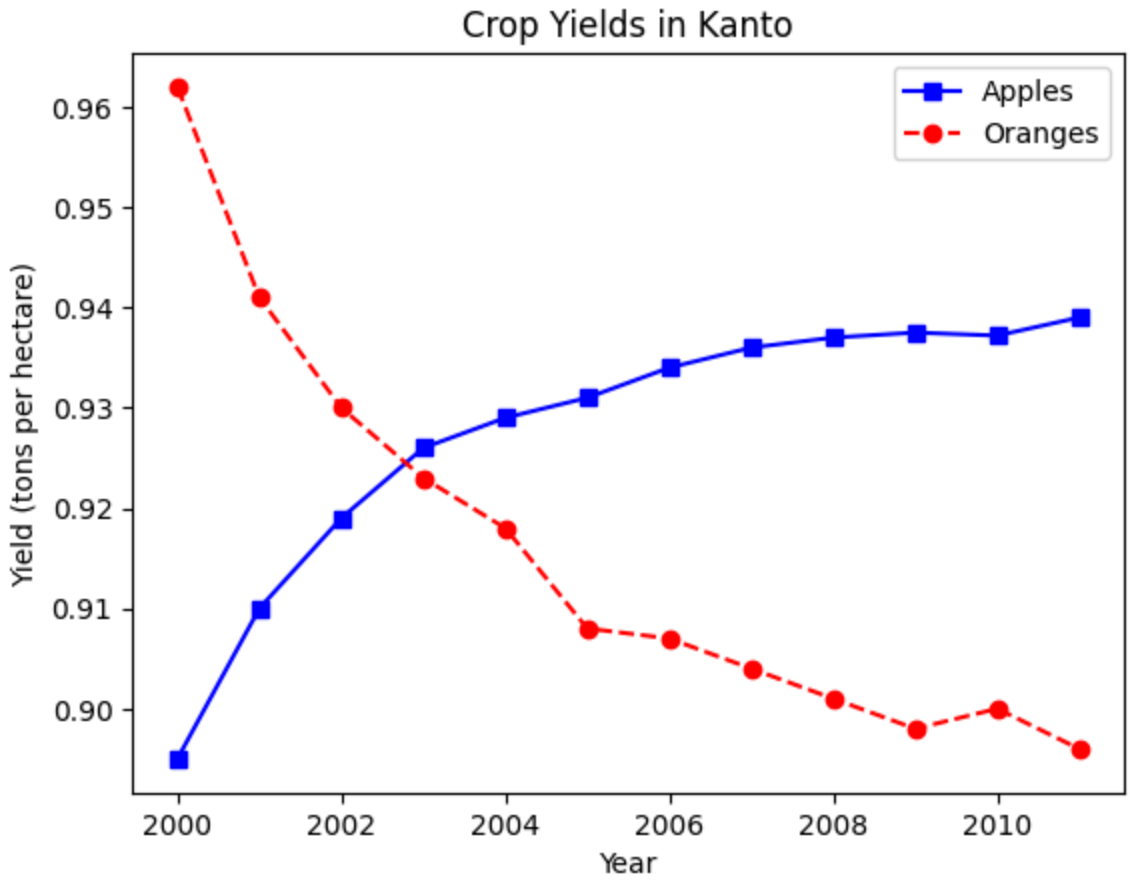
```
plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



The `fmt` argument provides a shorthand for specifying the line style, marker and line color. It can be provided as the third argument to `plt.plot`.

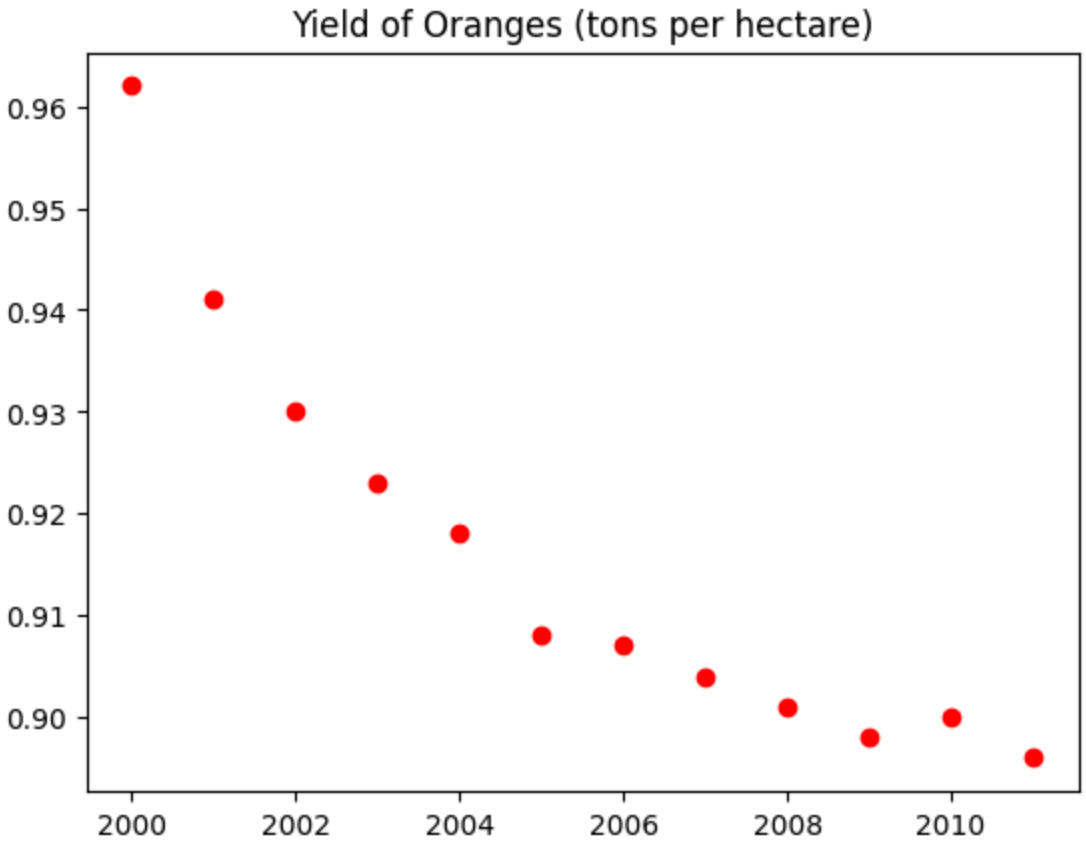
```
fmt = '[marker][line][color]'
```

```
plt.plot(years, apples, 's-b')
plt.plot(years, oranges, 'o--r')

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



If no line style is specified in `fmt`, only markers are drawn.

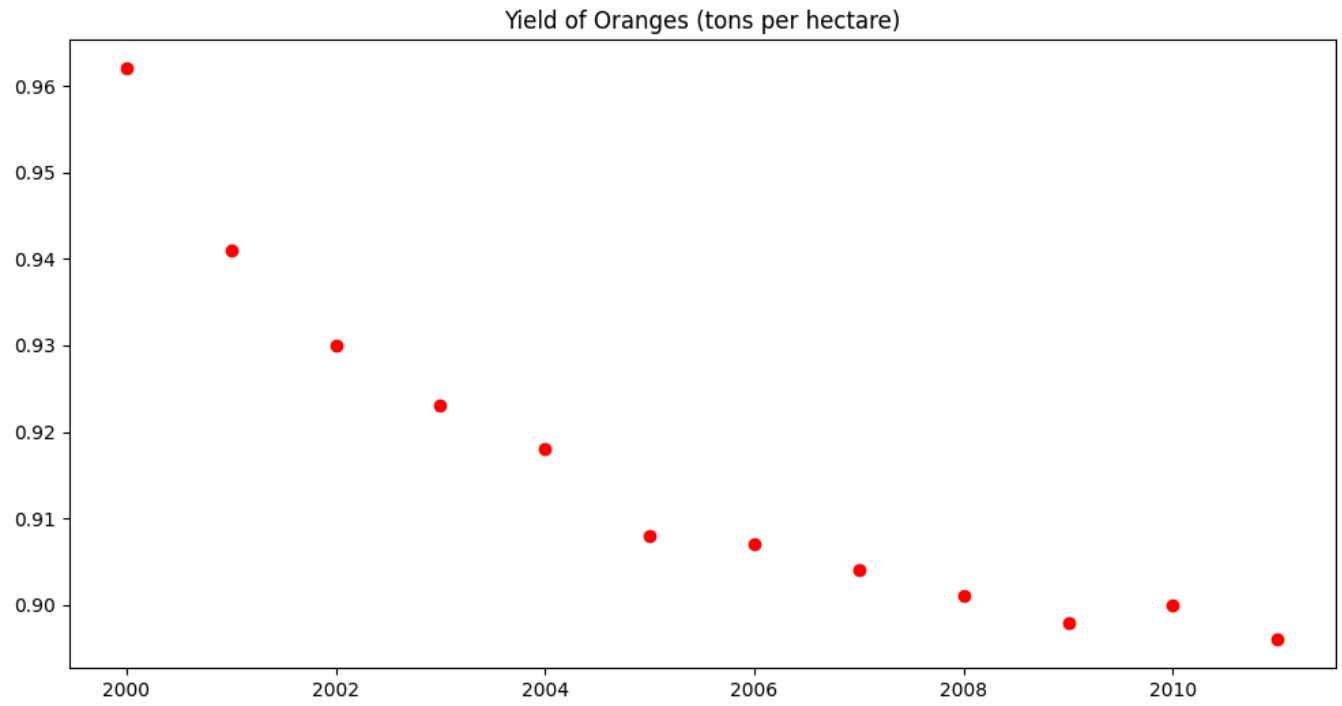
```
plt.plot(years, oranges, 'or')
plt.title("Yield of Oranges (tons per hectare)");
```



Changing the Figure Size

You can use the `plt.figure` function to change the size of the figure.

```
plt.figure(figsize=(12, 6))

plt.plot(years, oranges, 'or')
plt.title("Yield of Oranges (tons per hectare)");
```

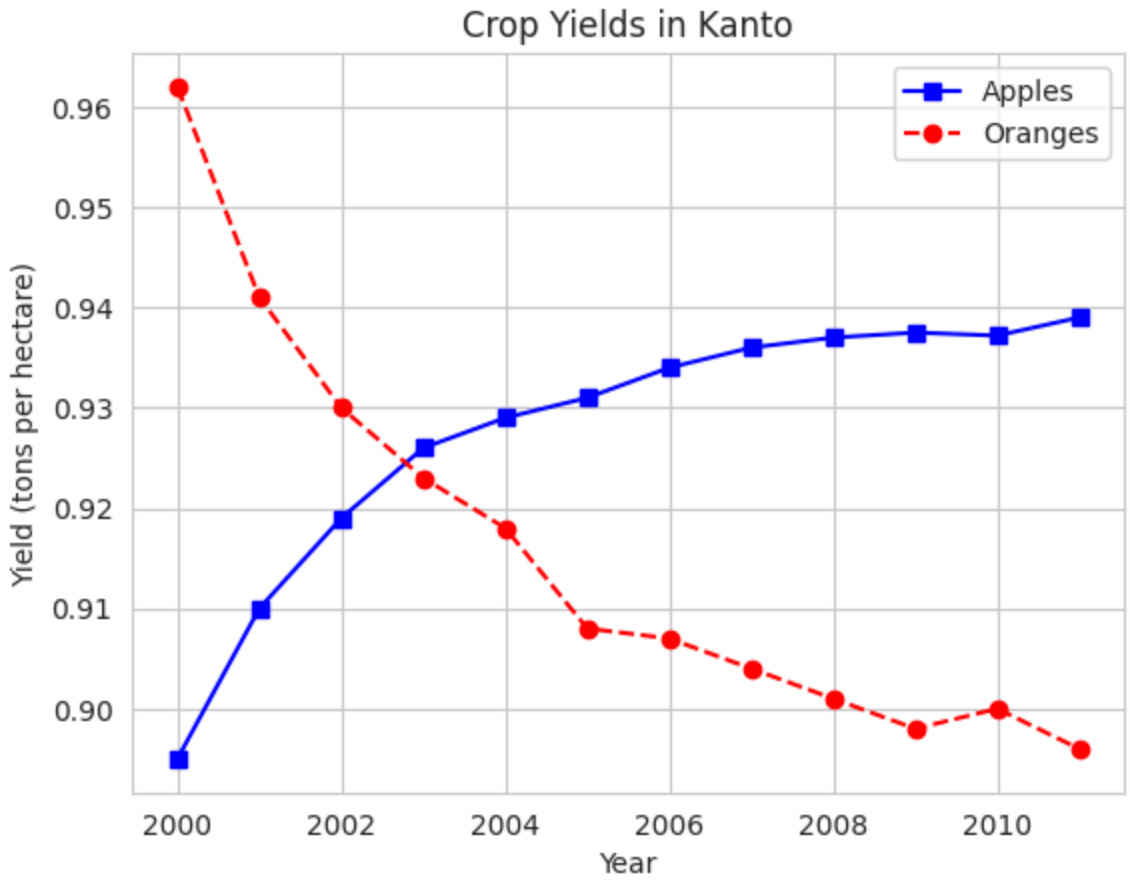


Improving Default Styles using Seaborn

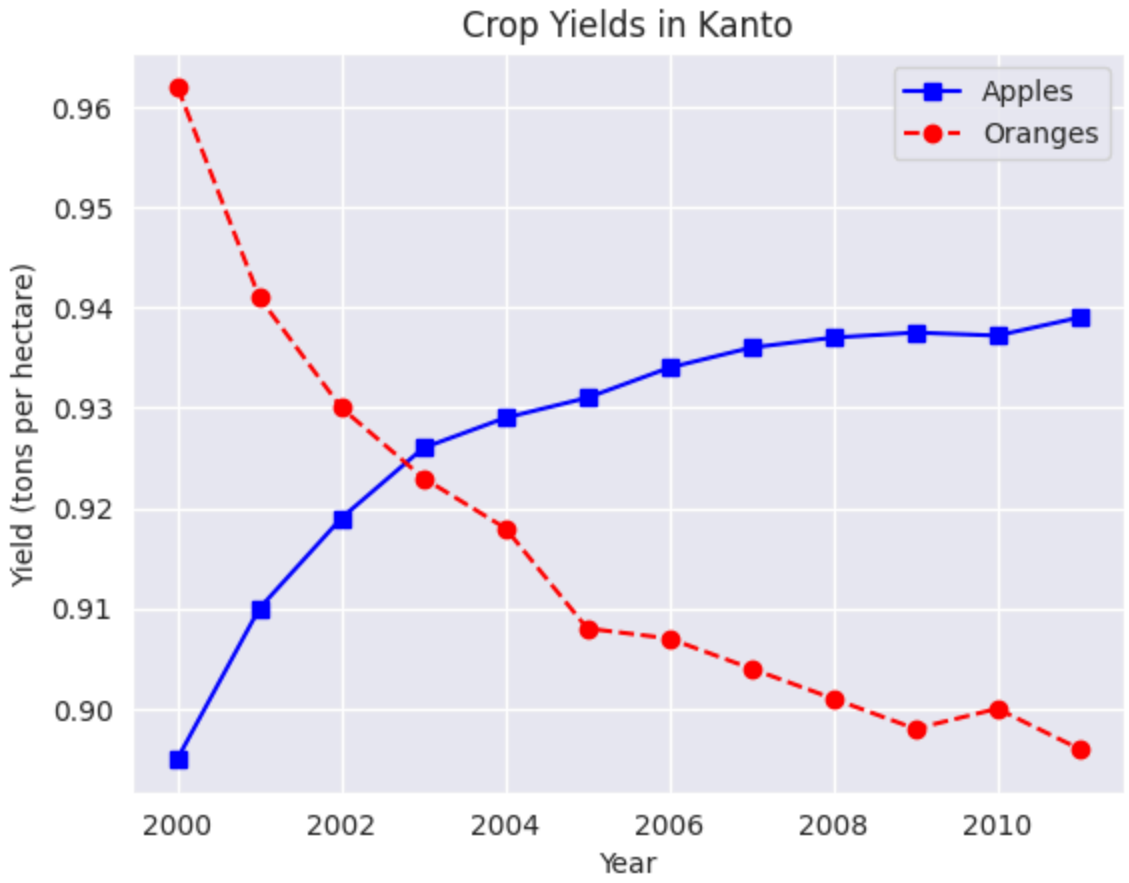
An easy way to make your charts look beautiful is to use some default styles provided in the Seaborn library. These can be applied globally using the `sns.set_style` function. You can see a full list of predefined styles here: https://seaborn.pydata.org/generated/seaborn.set_style.html

```
sns.set_style("whitegrid")
```

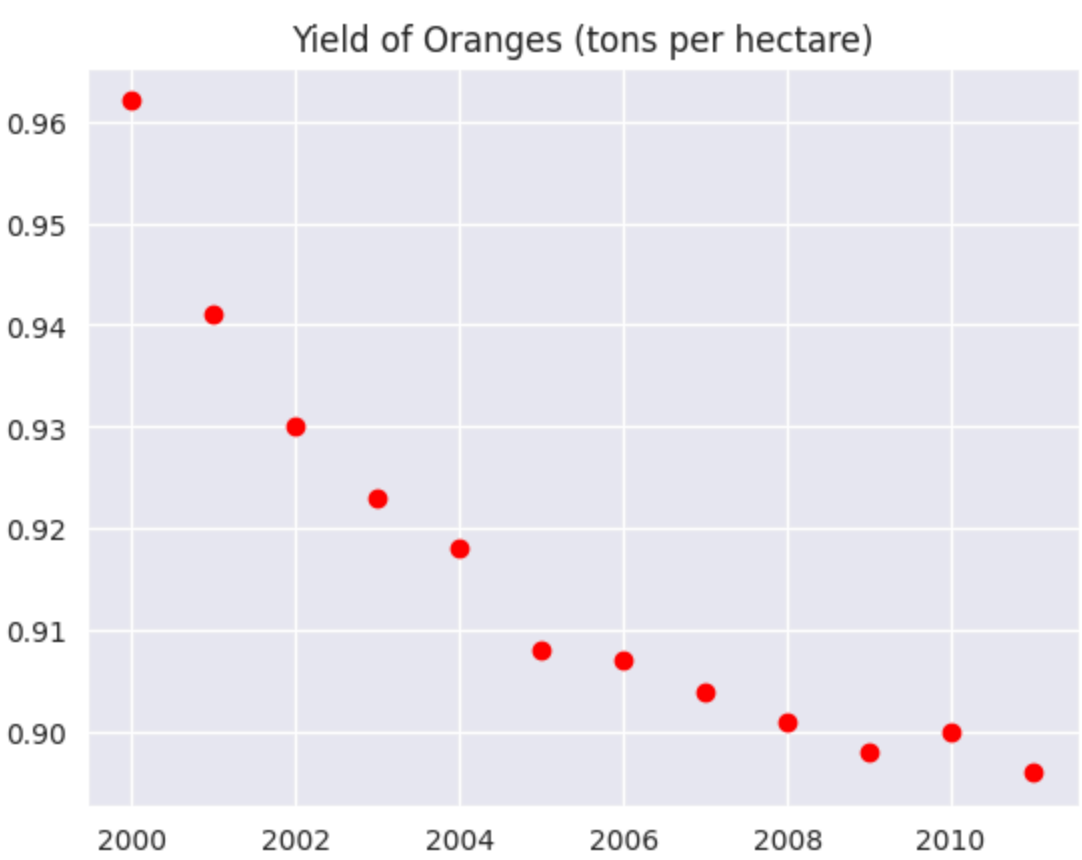
```
plt.plot(years, apples, 's-b')
plt.plot(years, oranges, 'o--r')

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```




```
sns.set_style("darkgrid")


plt.plot(years, apples, 's-b')
plt.plot(years, oranges, 'o--r')

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```




```
plt.plot(years, oranges, 'or')
plt.title("Yield of Oranges (tons per hectare)");
```



You can also edit default styles directly by modifying the `matplotlib.rcParams` dictionary. Learn more: https://matplotlib.org/3.2.1/tutorials/introductory/customizing.html#matplotlib-rcparams

```
import matplotlib
```

```
matplotlib.rcParams['font.size'] = 14
matplotlib.rcParams['figure.figsize'] = (9, 5)
matplotlib.rcParams['figure.facecolor'] = '#00000000'
```

Save and Upload

Whether you're running this Jupyter notebook on an online service like Binder or on your local machine, it's important to save your work from time, so that you can access it later, or share it online. You can upload this notebook to your Jovian.ml account using the `jovian` Python library.

```
# Install the library
!pip install jovian --upgrade --quiet
```

```
  Preparing metadata (setup.py) ... done
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 68.6/68.6 kB 5.3 MB/s eta 0:00:00
```

```
  Building wheel for uuid (setup.py) ... done
```

```
import jovian
```

```
jovian.commit(project='python-matplotlib-data-visualization')
```

```
[jovian] Detected Colab notebook...
[jovian] jovian.commit() is no longer required on Google Colab. If you ran this
then just save this file in Colab using Ctrl+S/Cmd+S and it will be updated on J
Also, you can also delete this cell, it's no longer necessary.
```

Exercise

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

Scatter Plot

In a scatter plot, the values of 2 variables are plotted as points on a 2-dimensional grid. Additionally, you can also use a third variable to determine the size or color of the points. Let's try out an example.

The Iris flower dataset provides samples measurements of sepals and petals for 3 species of flowers. The Iris dataset is included with the Seaborn library, and can be loaded as a Pandas data frame.

```
# Load data into a Pandas dataframe
flowers_df = sns.load_dataset("iris")
```

```
flowers_df
```

	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
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

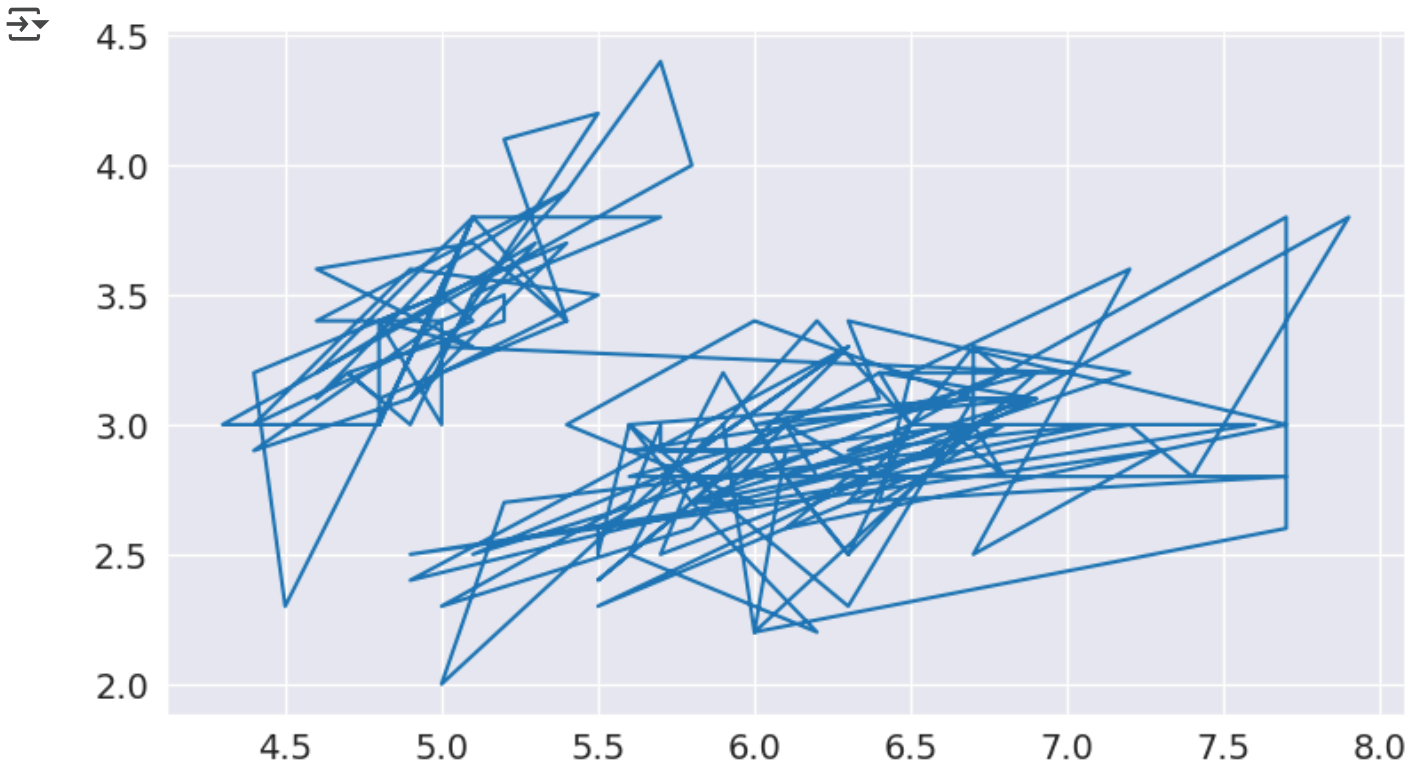
150 rows × 5 columns

```
flowers_df.species.unique()
```

```
array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

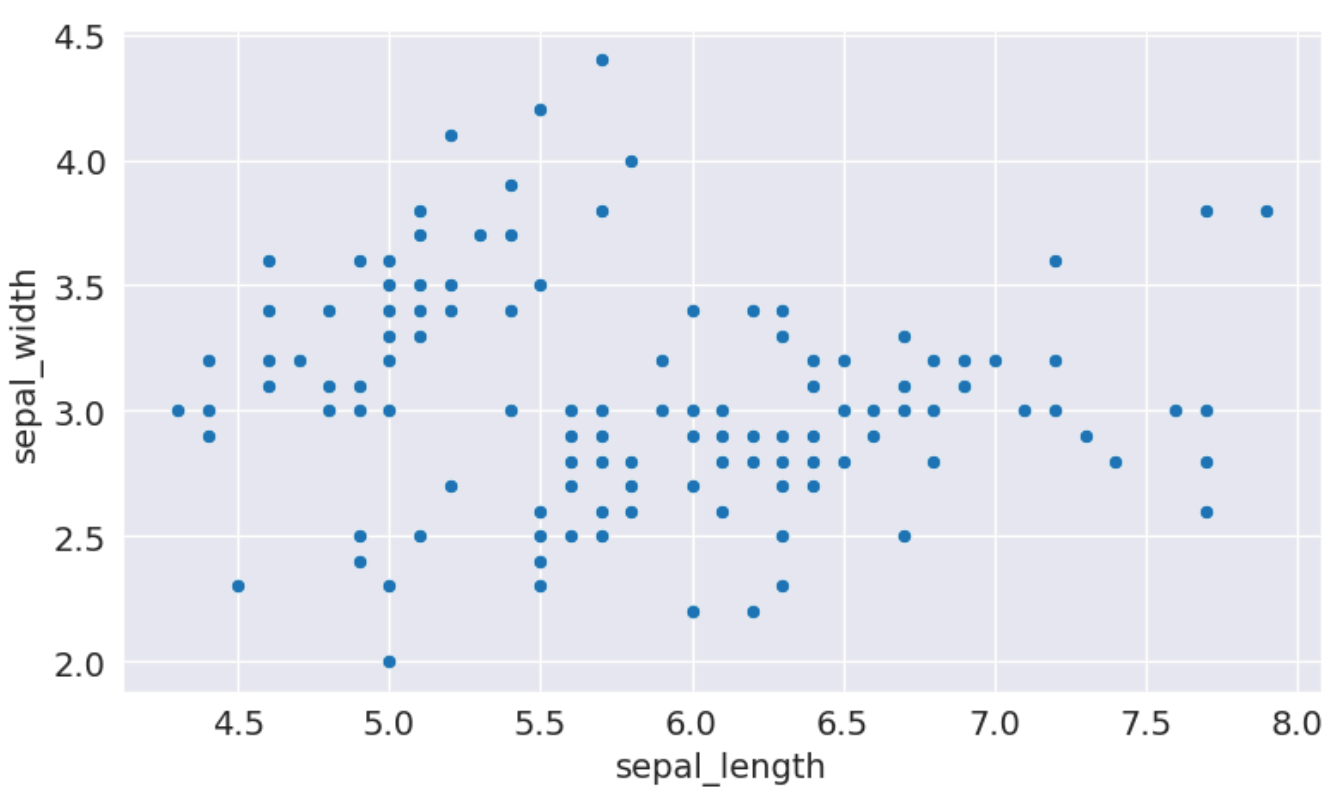
Let's try to visualize the relationship between sepal length and sepal width. Our first instinct might be to create a line chart using `plt.plot`. However, the output is not very informative as there are too many combinations of the two properties within the dataset, and there doesn't seem to be simple relationship between them.

```
plt.plot(flowers_df.sepal_length, flowers_df.sepal_width);
```



We can use a scatter plot to visualize how sepal length & sepal width vary using the `scatterplot` function from `seaborn` (imported as `sns`).

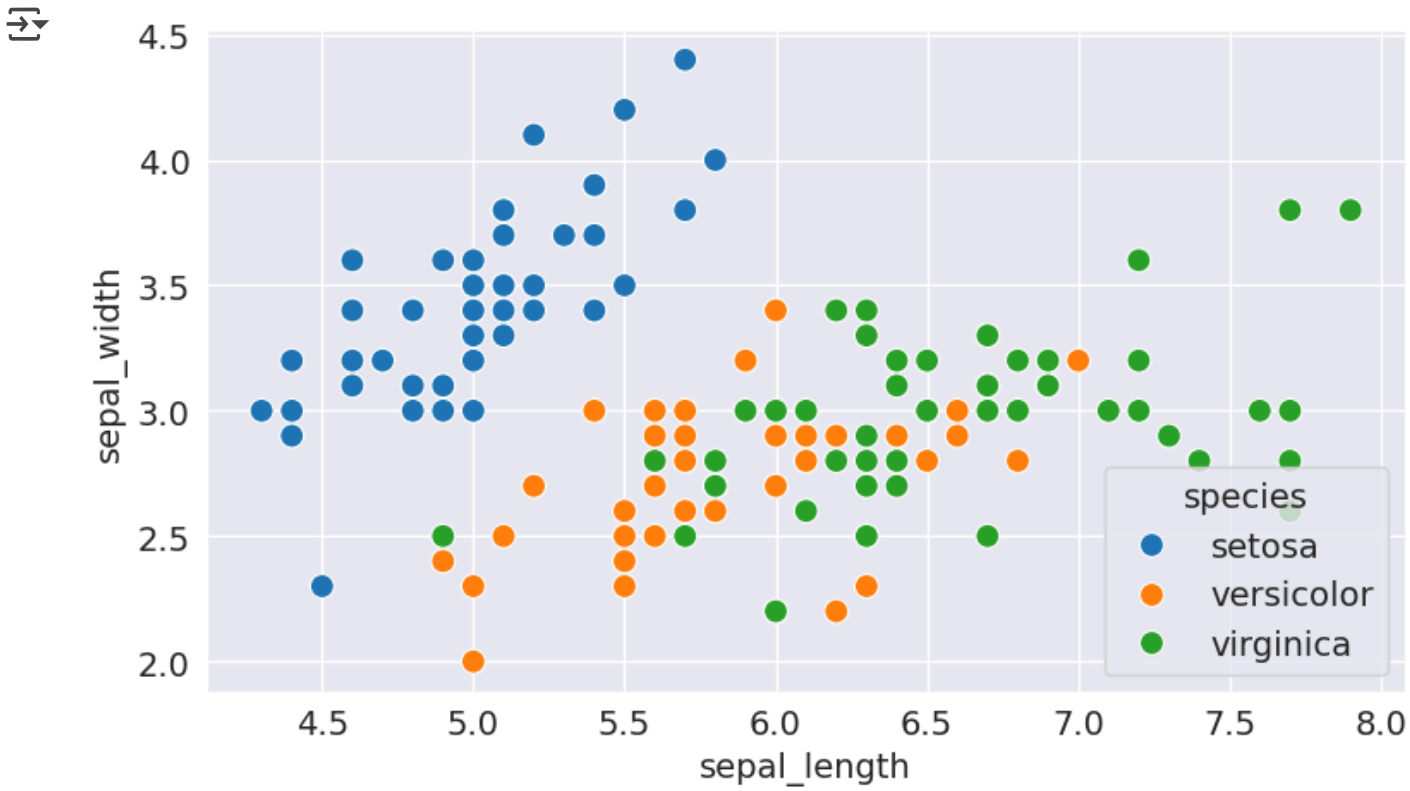
```
sns.scatterplot(x='sepal_length', y='sepal_width', data=flowers_df);
```



Adding Hues

Notice how the points in the above plot seem to form distinct clusters with some outliers. We can color the dots using the flower species as a `hue`. We can also make the points larger using the `s` argument.

```
sns.scatterplot(x='sepal_length', y='sepal_width', hue='species', s=100, data=flower
```

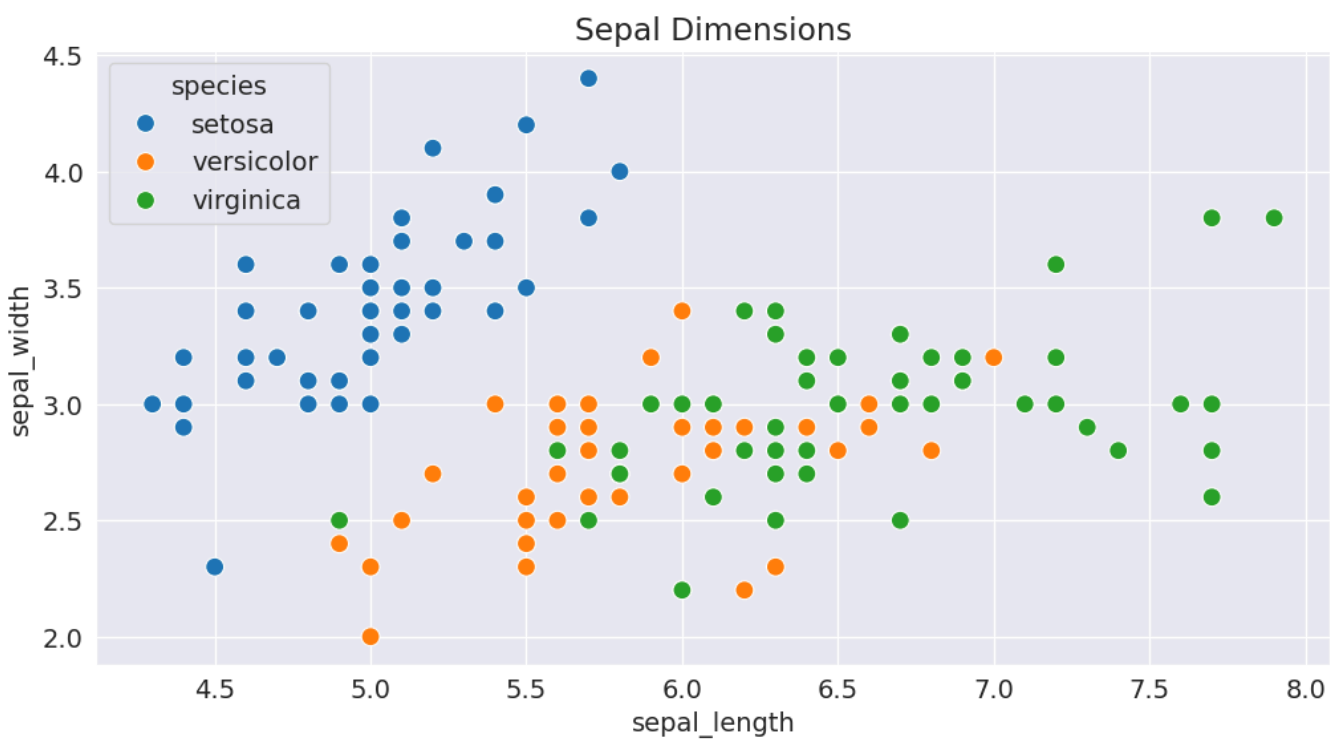


Adding hues makes the plot more informative. We can immediately tell that flowers of the Setosa species have a smaller sepal length but higher sepal widths, while the opposite holds true for the Virginica species.

Customizing Seaborn Figures

Since Seaborn uses Matplotlib's plotting functions internally, we can use functions like `plt.figure` and `plt.title` to modify the figure.

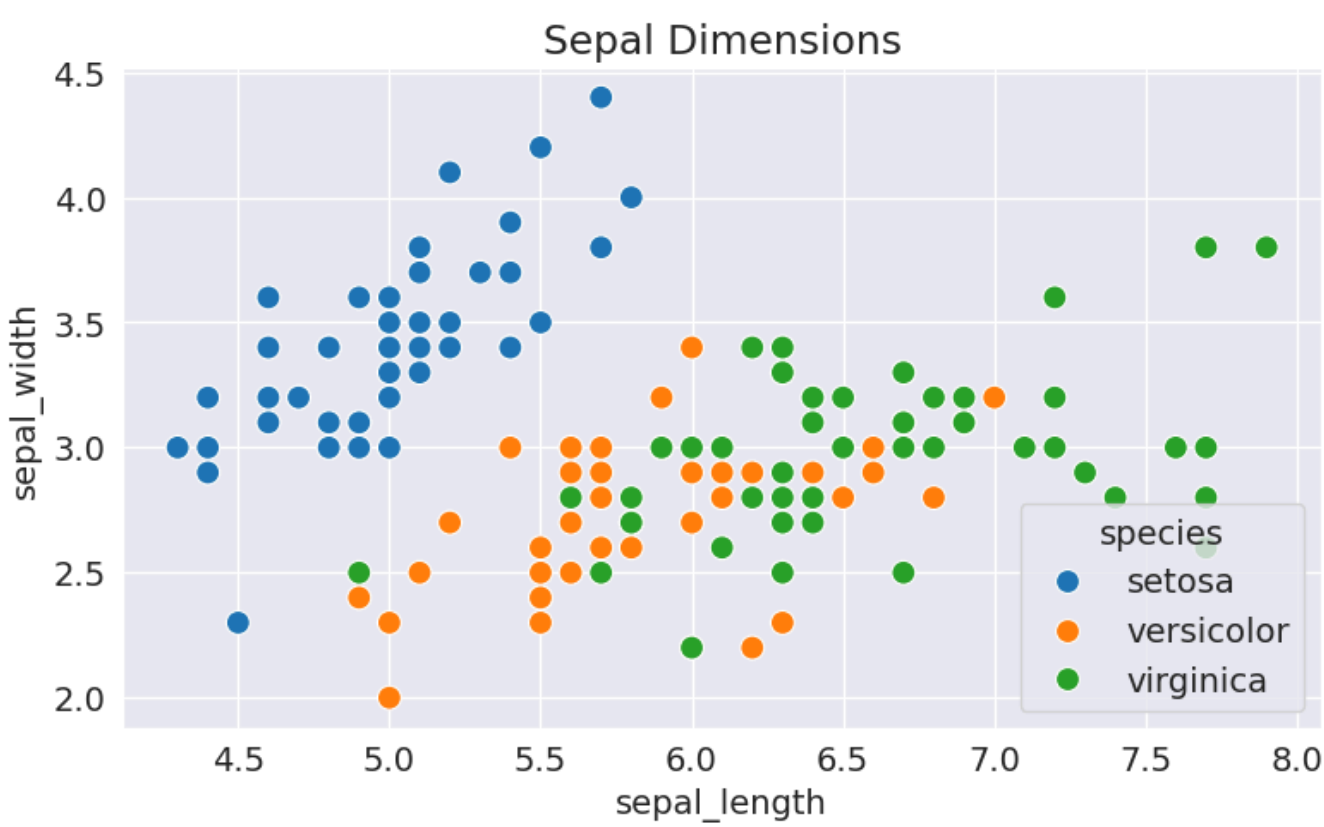
```
plt.figure(figsize=(12, 6))
plt.title('Sepal Dimensions')

sns.scatterplot(x=flowers_df.sepal_length,
                y=flowers_df.sepal_width,
                hue=flowers_df.species,
                s=100);
```



Plotting using Pandas Data Frames

Seaborn has in-built support for Pandas data frames. Instead of passing each column as a series, you can also pass column names and use the `data` argument to pass the data frame.

```
plt.title('Sepal Dimensions')
sns.scatterplot(x='sepal_length',
                y='sepal_width',
                hue='species',
                s=100,
                data=flowers_df);
```



Let's save and upload our work before continuing.

```
import jovian
```

```
jovian.commit()
```

Exercise - TODO

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

Histogram

A histogram represents the distribution of data by forming bins along the range of the data and then drawing bars to show the number of observations that fall in each bin.

As an example, let's visualize the how the values of sepal width in the flowers dataset are distributed. We can use the `plt.hist` function to create a histogram.

```
# Load data into a Pandas dataframe
flowers_df = sns.load_dataset("iris")
```

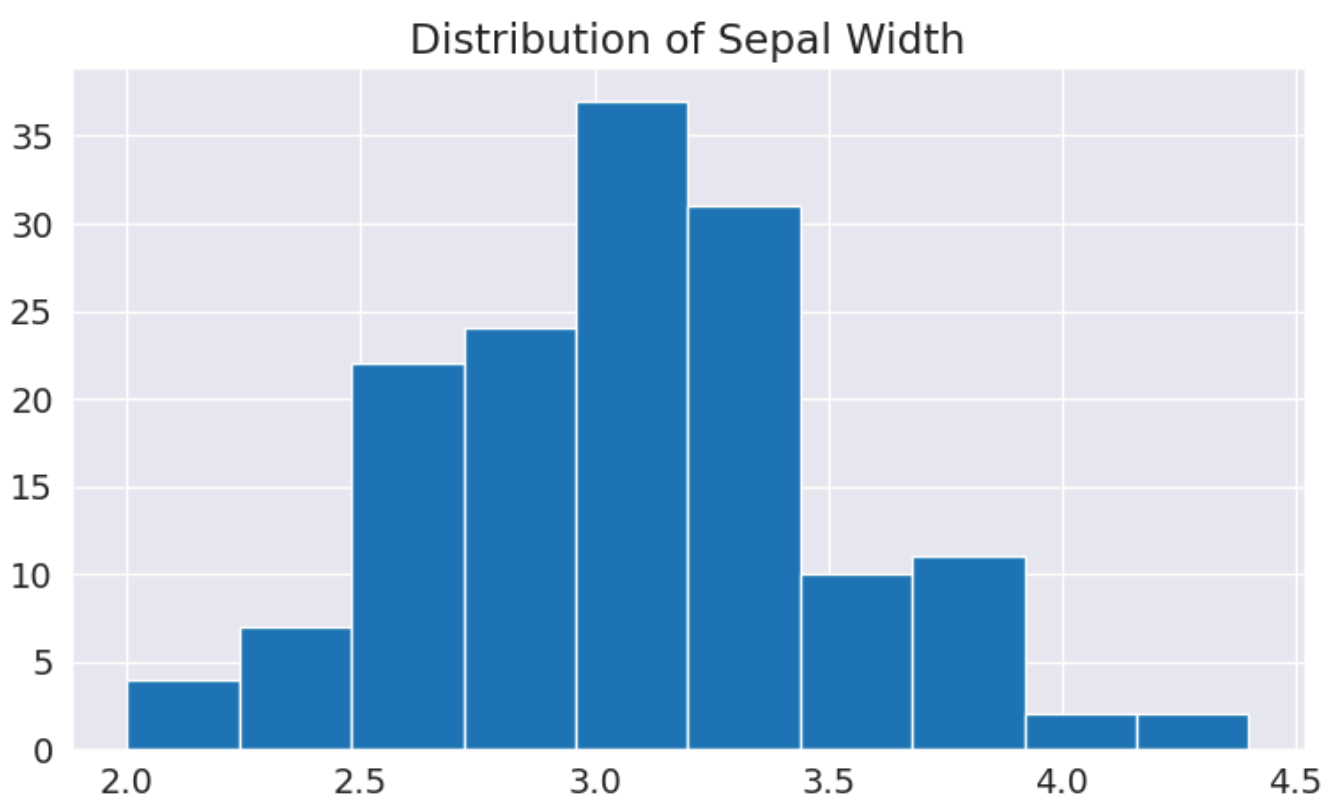
```
flowers_df.sepal_width
```

	sepal_width
0	3.5
1	3.0
2	3.2
3	3.1
4	3.6
...	...
145	3.0
146	2.5
147	3.0
148	3.4
149	3.0

150 rows × 1 columns

dtype: float64

```
plt.title("Distribution of Sepal Width")
plt.hist(flowers_df.sepal_width);
```

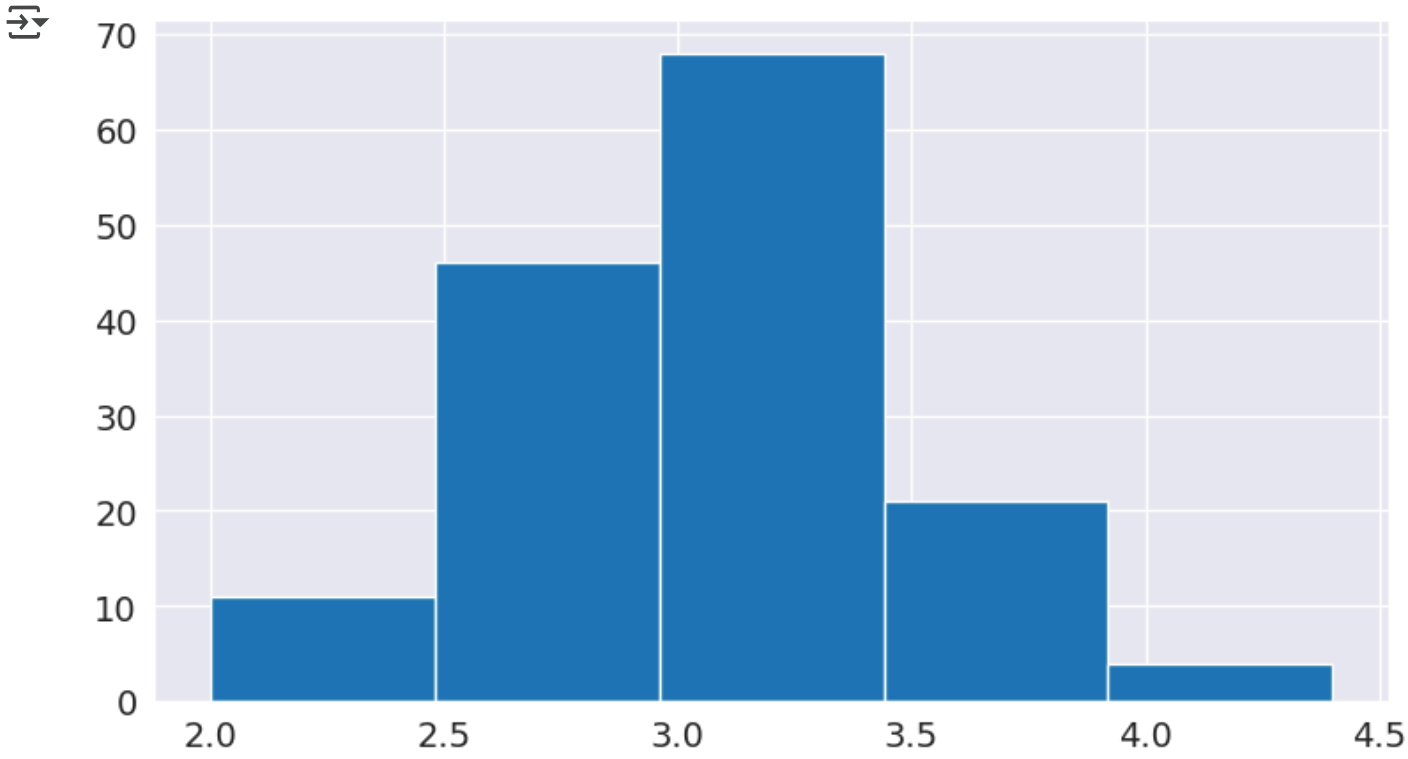


We can immediately see that values of sepal width fall in the range 2.0 - 4.5, and around 35 values are in the range 2.9 - 3.1, which seems to be the largest bin.

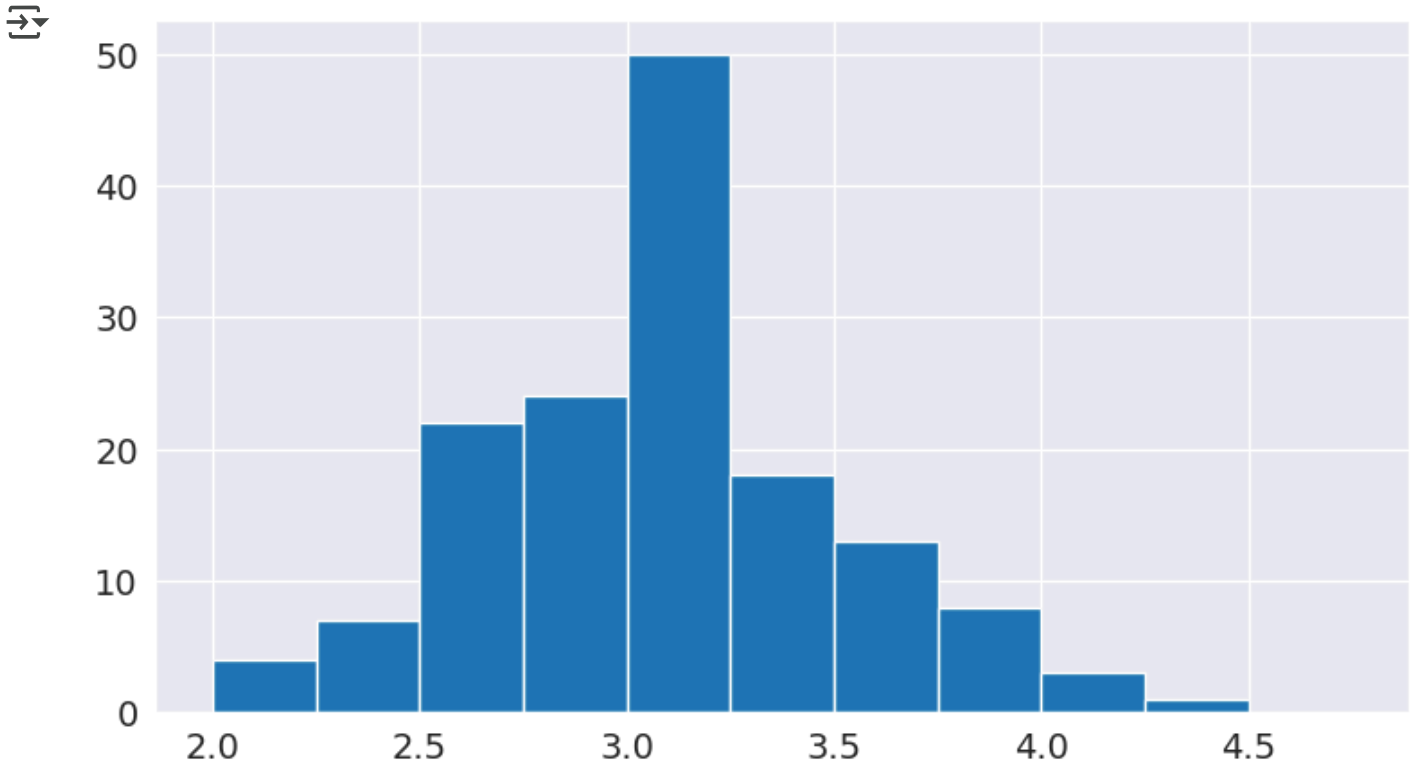
Controlling the size and number of bins

We can control the number of bins, or the size of each bin using the `bins` argument.

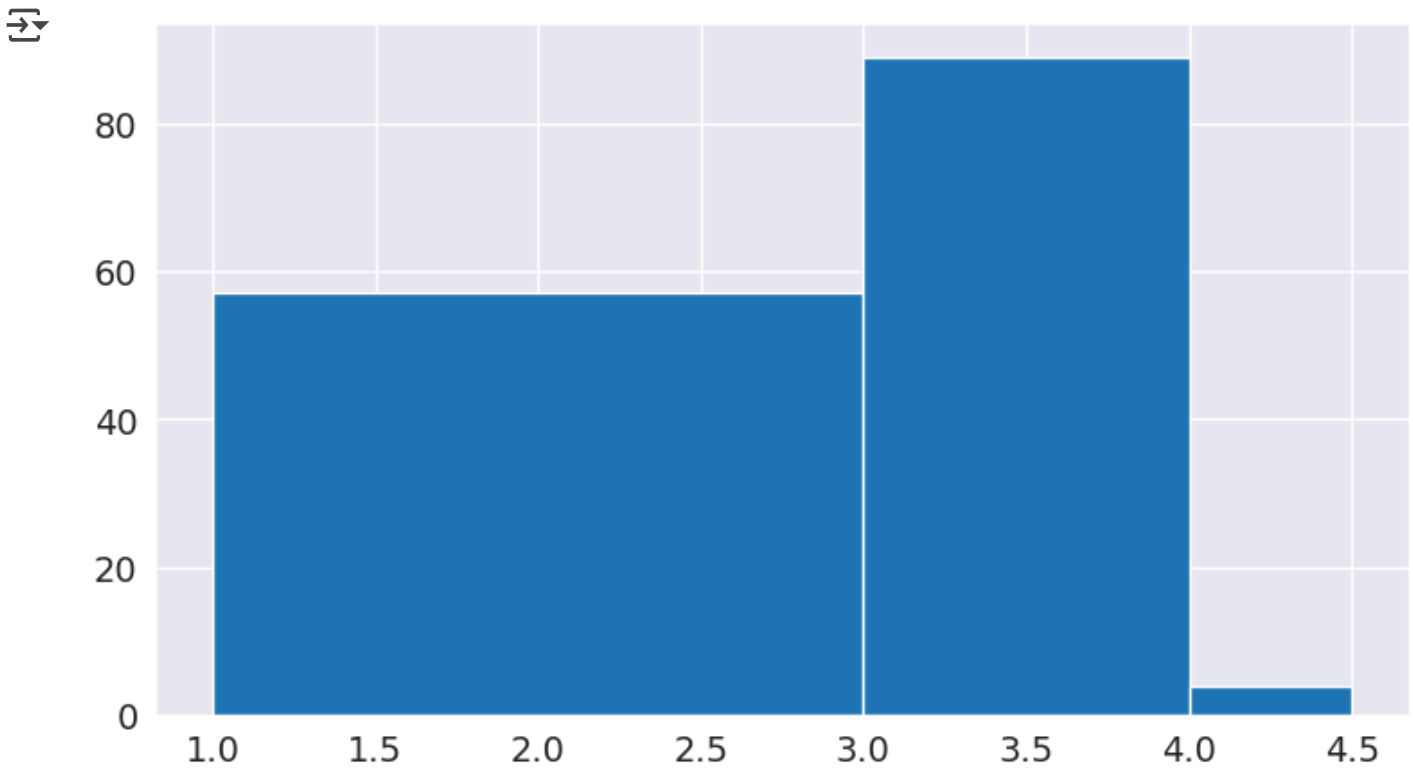
```
# Specifying the number of bins
plt.hist(flowers_df.sepal_width, bins=5);
```




```
import numpy as np

# Specifying the boundaries of each bin
plt.hist(flowers_df.sepal_width, bins=np.arange(2, 5, 0.25));
```




```
# Bins of unequal sizes
plt.hist(flowers_df.sepal_width, bins=[1, 3, 4, 4.5]);
```



Multiple Histograms

Similar to line charts, we can draw multiple histograms in a single chart. We can reduce the opacity of each histogram, so the the bars of one histogram don't hide the bars for others.

Let's draw separate histograms for each species of flowers.

```
setosa_df = flowers_df[flowers_df.species == 'setosa']
versicolor_df = flowers_df[flowers_df.species == 'versicolor']
virginica_df = flowers_df[flowers_df.species == 'virginica']
```

```
plt.hist(setosa_df.sepal_width, alpha=0.4, bins=np.arange(2, 5, 0.25));
plt.hist(versicolor_df.sepal_width, alpha=0.4, bins=np.arange(2, 5, 0.25));
```