

Data Visualization in Python

**Matplotlib
and Pandas**

David Landup

Practical, hands-on, straightforward guide
to understanding and performing
Data Visualization and Exploration using
Python, Pandas and Matplotlib

StackAbuse

Data Visualization in Python with Pandas and Matplotlib

StackAbuse

© 2021 StackAbuse

Copyright © by StackAbuse.com

Authored by David Landup

Cover design by Jovana Ninković

The images in this book, unless otherwise noted, are the copyright of StackAbuse.com.

The scanning, uploading, and distribution of this book without permission is a theft of the content owner's intellectual property. If you would like permission to use material from the book (other than for review purposes), please contact scott@stackabuse.com. Thank you for your support!

First Edition: June 2021

Published by StackAbuse.com, a subsidiary of Unstack Software LLC.

The publisher is not responsible for links, websites, or other third-party content that are not owned by the publisher.

Contents

Chapter 1. - Introduction	1
Chapter 2. - Installation and Setup	3
Windows	4
MacOS	6
Linux	7
Chapter 3. - Getting Started with Pandas	9
Pandas Data Structures	9
Series	9
Creating Series from Python Array	10
Creating Series from Python Dictionary	11
Creating Series from Numpy Array	12
Accessing Series Elements	13
Accessing Custom Indices	14
Series.head()	17
Series.tail()	17
DataFrames	18
Creating DataFrames	18
Manipulating DataFrames	23
Descriptive Statistics	37
<i>DataFrame.head()</i> and <i>DataFrame.tail()</i>	38
Reshaping DataFrames	39
Transposing a DataFrame	40
Stacking a DataFrame	41
Unstacking a DataFrame	43
Pivoting a DataFrame	44

CONTENTS

Melting a DataFrame	46
Melting and Unmelting a DataFrame	48
How to Iterate Over Rows	50
Iterating DataFrames with <i>items()</i>	50
Iterating DataFrames with <i>iterrows()</i>	52
Iterating DataFrames with <i>itertuples()</i>	54
Iteration Performance with Pandas	55
Speed Comparison	55
Merging DataFrames	57
Merge DataFrames Using <i>merge()</i>	58
Merge DataFrames Using <i>join()</i>	63
Merge DataFrames Using <i>append()</i>	65
Merge DataFrames Using <i>concat()</i>	66
Merge DataFrames Using <i>combine_first()</i> and <i>update()</i>	68
Handling Missing Data	70
Data Inspection	71
Customizing Missing Data Values	73
Removing Rows With Missing Values	75
Filling out Missing Values	77
Reading and Writing CSV Files	81
What is a CSV File?	81
Reading CSV Files with <i>read_csv()</i>	82
Writing CSV Files with <i>to_csv()</i>	86
Handling Missing Values	88
Chapter 4. - Getting Started with Matplotlib	90
What is Matplotlib?	90
The PyPlot Interface	91
Anatomy of Matplotlib Plots	98
Object-Oriented Plotting	100
PyPlot API vs. Object-Oriented API	110
Chapter 5. - Basic Matplotlib Customization	113
Changing the Figure Size	113
Setting the <i>figsize</i> Argument	115
Setting the Height and Width of a Figure in Matplotlib	119

CONTENTS

Understanding Matplotlib Text (Titles, Labels, Annotations)	121
Add Legend	128
Customizing A Legend in Matplotlib	131
Adding a Legend Outside of Axes	132
Changing the Legend Size	134
Changing the Font Size	136
Changing the Font Size using <i>fontsize</i>	137
Changing the Font Size Globally	138
Save Plot as Image	140
Setting the Image DPI	142
Save Transparent Image with Matplotlib	143
Changing Plot Colors	144
Setting Image Border Box	145
Set Axis Range (<i>xlim</i> , <i>ylim</i>)	147
Setting the X-Limit (<i>xlim</i>) for Axes	148
Setting the Y-Limit (<i>ylim</i>) for Axes	150
Change Tick Frequency	151
Setting Figure-Level Tick Frequency	153
Setting Axis-Level Tick Frequency	154
Rotate Axis Tick Labels	156
Rotating X-Axis Tick Labels	157
Rotate Y-Axis Tick Labels	160
Rotate Dates to Fit Automatically	161
Draw Vertical Lines on Plot	163
Drawing Vertical Lines with <i>PyPlot.vlines()</i>	164
Drawing Vertical Lines with <i>PyPlot.axvline()</i>	167
Chapter 6. - Data Visualization with Matplotlib	171
Line Plot	172
Plotting a Line Plot	172
Plotting a Line Plot Logarithmically	175
Customizing Line Plots in Matplotlib	179
Plot Multiple Line Plots with Different Scales	181
Plot Multiple Line Plots with Multiple Y-Axes	185
Bar Plot	186
Plotting a Bar Plot	186

CONTENTS

Bar Plot with Error Bars in Matplotlib	193
Changing Bar Plot Colors	195
Plotting Horizontal Bar Plots	197
Sorting Bar Order	197
Pie Chart	200
Plotting a Pie Chart	200
Customizing Pie Charts	202
Scatter Plot	213
Plotting a Scatter Plot	214
Customizing Scatter Plot in Matplotlib	219
Change Marker Size in Matplotlib Scatter Plot	222
Plotting Bubble Plots	223
Exploring Relationships with Scatter Plots	228
Histogram Plot	234
Plotting a Histogram Plot	235
Changing a Histogram's Bin Size	237
Plotting Histogram with Density Data	240
Histogram Plot with KDE	245
Customizing Histogram Plots	247
Box Plot	249
Importing Data	251
Plotting a Box Plot	252
Customizing Box Plots	258
Exploring Relationships with Scatter Plots, Histograms and Box Plots	263
Scatter Plot with Marginal Distributions (Joint Plot)	269
Importing Data	270
Plotting a Joint Plot with Single-Class Histograms	271
Plotting a Joint Plot with Multiple-Class Histograms	273
Joint Plots with Box Plots	278
Violin Plots	279
Importing Data	280
Plotting a Violin Plot	281
Customizing Violin Plots	288
Scatter Plots with Violin Plots	292
Stack Plots	294
Importing Data	294

CONTENTS

Plotting a Stack Plot	295
Heatmaps	301
Importing Data	302
Plotting a Heatmap	303
Ridge Plots (Joy Plots)	310
Plotting a Ridge Plot	313
Spectrogram Plots	322
Plotting a Spectrogram	324
Matplotlib - 3D Support	328
3D Scatter Plots and Bubble Plots	330
Combining 2D and 3D Plots	334
3D Surface Plots and Wireframe Plots	336
Projecting Surface Plots with Contour Plots	343
Plotting a Contour Plot	344
Plotting a Surface Plot with Contour Plots	347
3D Line Plots and CP1919 Ridge Plot	350
Exploring EEG (Brainwave) Channel Data with Line Plots, Surface Plots and Spectrograms	352
Importing Data	352
What is EEG?	353
Pre-Processing Data	355
Plotting Surface Plots of EEG Channels	360
Plotting Surface Plots for Different Stimuli	361
Plotting Neuron Group Activations for Individuals via Surface Plots .	368
Plotting Channels with Line Plots and Spectrograms	371
Chapter 7. - Advanced Matplotlib Customization	379
Understanding Matplotlib Stylesheets	379
Matplotlib Runtime Configuration (rc) Parameters	384
Understanding Matplotlib Colors and Colormaps	387
Colormaps	389
Customizing Layouts with GridSpec	392
Chapter 8 - Data Visualization with Pandas	403
The <i>DataFrame.plot()</i> Function	404
Pandas' plotting Module	406

CONTENTS

Bootstrap Plot	406
Autocorrelation Plot	407
Scatter Matrices	409
Chapter 9. - Matplotlib Widgets	411
Adding Buttons	412
Adding Radio Buttons and Check Boxes	416
Adding Textboxes	421
Adding Span Selectors	424
Adding Sliders	427
Thank You for Supporting Online Education	432

Chapter 1. - Introduction

Welcome to *Data Visualization in Python with Matplotlib and Pandas*. This book is designed to take absolute beginners to Pandas and Matplotlib, with basic Python knowledge, and allow them to build a strong foundation for advanced work with these libraries - from simple plots to 3D plots and interactive buttons.

We'll start out with the installation and setup of the environment you need to start working on Data Visualization projects, followed by a *crash-course* on Pandas - from the very foundations and upwards, covering everything you'll need to know to follow this book, and to start working on your own visualization projects.

Then, we'll dive into the basics of Matplotlib, the anatomy of plots, the APIs it offers, and basic plot customization, including common tasks like changing font size, setting axis ranges, changing tick frequency, adding legends and plotting basic plots. We'll also cover some fundamental classes like the `Text` class and how annotations work.

Once we've gotten the hang of Matplotlib's APIs, plot anatomy and how to customize them - we'll jump into the meat of the book - Data Visualization with Matplotlib. This is where we'll cover the quintessential plots that should be in the arsenal of any Data Scientist, like Scatter Plots, Box Plots, and Histograms. This is also the chapter in which we'll dive into custom plot types, such as Joint Plots and Ridge Plots that aren't part of the standard library, followed by 3D plots and an exploration section, where we'll explore an EEG dataset.

Note: For each plot type, we'll aim to use a different dataset, which necessitates different pre-processing. In some cases, the data will be very fit for the plot type we're using already - since we'll be choosing the right plot type for the job. In other cases, though, we'll have to perform pre-processing with Pandas. All of the datasets will be available publicly, and downloadable for your convenience and the links to each will be provided in the footnotes. In many cases, we'll be able to pose hypothesis on the correlations between certain features, as well as test these hypothesis through Exploratory Data Analysis.

At this point, we'll jump into more advanced customization, overcoming potential issues with more advanced and custom plot types. Before wrapping up with interactive

buttons, we'll take a brief look at Pandas' plotting abilities as well.

Finally, Matplotlib isn't only for static plots. While GUIs are typically created with GUI libraries and frameworks such as [PyQt¹](#), [Tkinter²](#), [Kivy³](#) and [wxPython⁴](#), and while Python does have excellent integration with PyQt, Tkinter *and* wxPython - there's no need to use any of these for some basic GUI functionality, through *Matplotlib Widgets*:

- [**Chapter 1. - Introduction**](#)
- [**Chapter 2. - Installation and Setup**](#)
- [**Chapter 3. - Getting Started with Pandas**](#)
- [**Chapter 4. - Getting Started with Matplotlib**](#)
- [**Chapter 5. - Basic Matplotlib Customization**](#)
- [**Chapter 6. - Data Visualization with Matplotlib**](#)
- [**Chapter 7. - Advanced Matplotlib Customization**](#)
- [**Chapter 8 - Data Visualization with Pandas**](#)
- [**Chapter 9. - Matplotlib Widgets**](#)

¹<https://riverbankcomputing.com/software/pyqt/>

²<https://docs.python.org/3/library/tkinter.html>

³<https://kivy.org/#home>

⁴<https://www.wxpython.org/>

Chapter 2. - Installation and Setup

We'll be working with several tools throughout the book. Dabbling in Python almost warrants that you've already used *some* of these before, if not most, since these are fairly popular libraries present in a large amount of projects. Namely, we'll be using:

- Python⁵
- Matplotlib⁶
- Pandas⁷
- Numpy⁸

We'll rely on Numpy sparingly, so if you haven't worked with it before - there's no need to worry. Even though Matplotlib uses Numpy Arrays under the hood, even without prior experience with it, you'll be able to follow the book without a problem given the intuitive API and little need to use it manually.

Assuming no prior knowledge of Pandas, we'll first be jumping into *Chapter 3 - Getting Started with Pandas*. It's extensively used with Matplotlib, and we'll be using it throughout the book to pre-process and wrangle data into the formats most fit for our visualization needs. We'll start at the foundations and building blocks of Pandas to common tasks and operations you'll be performing as well as *data reshaping*, giving you an solid introduction to the library and how we'll be using it in the book.

In the case you don't already have these tools installed on your local machine, let's quickly set them up.

⁵<https://www.python.org/>

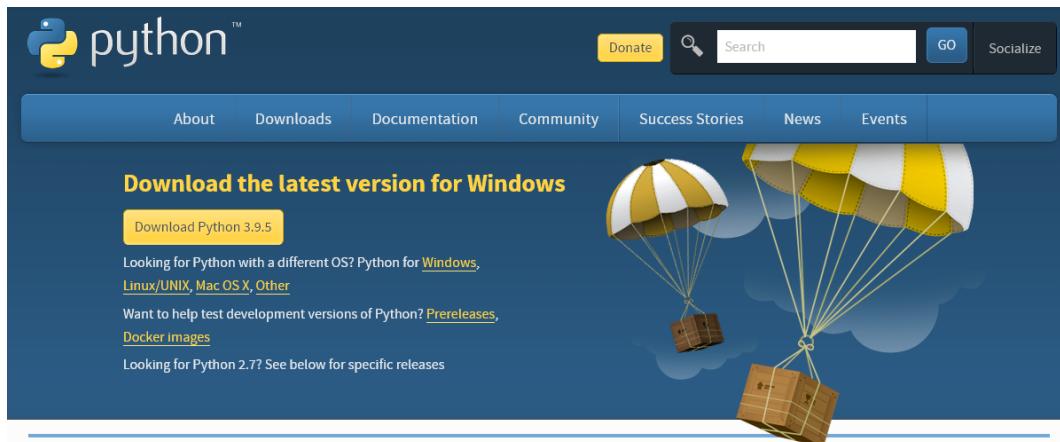
⁶<https://matplotlib.org/>

⁷<https://pandas.pydata.org/>

⁸<https://numpy.org/>

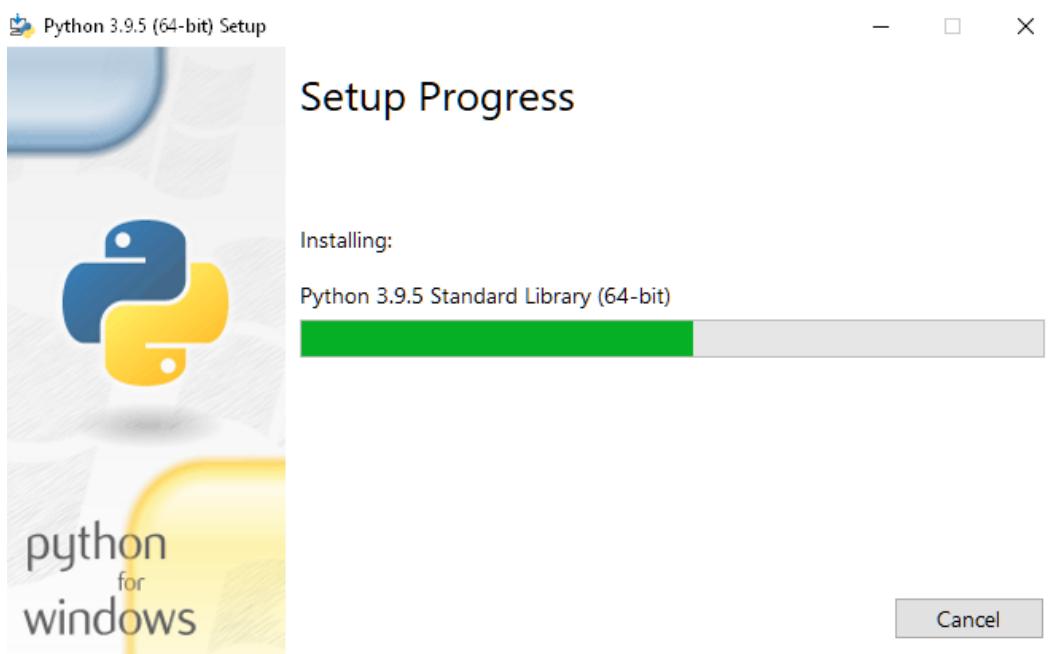
Windows

Let's first go ahead and install *Python*. Navigate to the [Downloads⁹](#) page, and choose your preferred version:



The installer is very straightforward, and going with the default settings will get us a long way:

⁹<https://www.python.org/downloads/>



Since Python 3.4, the [pip¹⁰](#) command-line utility has become a part of the standard package, so there's no need to install it separately.

It's good practice to move into a new directory for new projects, and instantiate a virtual environment within it to contain the installed dependencies and their versions. Let's create a directory that we'll be using to store our datasets and scripts, and instantiate a virtual environment there. Open up the terminal and input:

```
1 $ mkdir data_visualization
2 $ cd data_visualiation
3 $ pip install virtualenv
4 Collecting virtualenv
5   Downloading virtualenv-20.4.7-py2.py3-none-any.whl (7.2 MB)
6     |████████████████████████████████████████████████| 7.2 MB 6.4 MB/s
7 ...
8   Successfully installed appdirs-1.4.4 distlib-0.3.2 filelock-3.0.12 six-1.16.0 virtualen\
9 v-20.4.7
10
11 $ virtualenv env
12   created virtual environment CPython3.9.5.final.0-64 in 786ms
13 ...
14
15 $ cd env/Scripts
```

¹⁰<https://pip.pypa.io/en/stable/installing/>

```
16 $ activate  
17 $ (env) PATH\data_visualization\env\Scripts>
```

We've created a directory for our project, moved into it, downloaded `virtualenv` via `pip`, which is built into Python and successfully created a virtual environment for our project. Now, when we install other dependencies, they won't clash with other versions, since we're installing them in a virtual environment only.

Let's install Matplotlib:

```
1 (env) PATH\data_visualization> $ pip install matplotlib  
2 Collecting matplotlib  
3 ...  
4   Successfully installed cycler-0.10.0 kiwisolver-1.3.1 matplotlib-3.4.2 numpy-1.20.3 pillow-  
5 w-8.2.0 pyparsing-2.4.7 python-dateutil-2.8.1 six-1.16.0
```

It's worth noting that *Matplotlib* installs *Numpy* by default, so there's no need to install it manually. Finally, we should also install Pandas:

```
1 (env) PATH\data_visualization> $ pip install pandas  
2 Collecting pandas  
3 ...  
4   Successfully installed pandas-1.2.4 pytz-2021.1
```

MacOS

When it comes to MacOS - there are two ways to go about installing Python - through an installer on the official website, or through *Homebrew*. Navigate to the [downloads page](#)¹¹ and select the preferred version:

¹¹<https://www.python.org/downloads/mac-osx/>

The screenshot shows the Python.org homepage with a blue header. The Python logo is on the left, followed by the word "python™". To the right are buttons for "Donate", "Search" (with a magnifying glass icon), "GO", and "Socialize". Below the header is a navigation bar with links: "About", "Downloads", "Documentation", "Community", "Success Stories", "News", and "Events". The "Downloads" link is highlighted. Below the navigation bar, the URL "Python >>> Downloads >>> Mac OS X" is shown. The main content area has a title "Python Releases for Mac OS X" and two sections: "Stable Releases" (listing "Latest Python 3 Release - Python 3.9.5" and "Latest Python 2 Release - Python 2.7.18") and "Pre-releases" (listing "Python 3.10.0b2 - May 31, 2021").

Alternatively, if you don't already have Homebrew installed, you can fetch it from the official GitHub repository via curl:

```
1 $ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

Then, installing Python is as easy as:

```
1 $ brew install python3
2 $ pip install virtualenv
```

Of course, followed by creating a directory for our project, instantiating a virtual environment, and installing our dependencies:

```
1 $ mkdir data_visualization
2 $ cd data_visualization
3 $ virtualenv env
4 $ source env/bin/activate
5 (env) $ pip install matplotlib, pandas
```

Linux

Finally, Linux makes the installation process fairly easy as well. Let's first install Python:

```
1 $ sudo apt-get install python
```

Followed by creating a directory for our project:

```
1 $ mkdir data_visualziation  
2 $ cd data_visualization
```

And initiating the virtual environment:

```
1 $ virtualenv env  
2 $ source venv/bin/activate  
3 (env) $ pip install matplotlib pandas
```

Chapter 3. - Getting Started with Pandas

Pandas is an open-source Python package that provides numerous tools for data analysis. The package comes with several data structures that can be used for many different data manipulation tasks. It also has a variety of methods that can be invoked for data analysis, which come in handy when working on Data Science and Machine Learning problems.

It can present data in a way that is very intuitive and suitable for data analysis, via its `Series` and `DataFrame` data structures. The `DataFrame` is a fundamental and key data structure in the framework, and you'll spend a lot of time working with them.

Additionally, Pandas has a variety of ways to work with different types of I/O operations very seamlessly. It can read data from a variety of formats, such as CSV, XSLX, JSON, etc.

Pandas Data Structures

Pandas has two main data structures for data storage:

1. `Series`
2. `DataFrame`

Let's go over those two first.

Series

A series is similar to a one-dimensional array. It can store data of any type. The values of a Pandas `Series` are mutable but the size of a `Series` is immutable and cannot be changed.

The first element in the series is assigned the index of 0 , while the last element is at index $N-1$, where N is the total number of elements in the series.

Before creating a Pandas Series, we must first import the Pandas package via the Python's `import` command:

```
1 import pandas as pd
```

To create the Series, we invoke the `pd.Series()` method and pass in some data:

```
1 series = pd.Series(data)
```

This data can be a Python array, a Python dictionary, a scalar value or an `ndarray` from Numpy.

Additionally, there's an optional `index` argument that you can use to specify the indices of the values in the series, if you don't want them to automatically span $[0..N-1]$:

```
1 series = pd.Series(data, index)
```

Creating Series from Python Array

Let's first simply use a Python array:

```
1 series = pd.Series([1, 2, 3, 4])
```

Now, let's `print()` this series to check the contents:

```
1 print(series)
```

This results in:

```
1 0    1
2 1    2
3 2    3
4 3    4
5 dtype: int64
```

You can see that we have two columns, the first one with numbers starting from the index `0` and the second one with the elements that were added to the series.

The first column denotes the indexes for the elements, and by default, they start at `0` and increment. Let's define our own indices:

```
1 series1 = pd.Series([1, 2, 3, 4], index=[5, 7, 3, 5])
```

And now, let's print() the `series1`:

```
1 print(series1)
```

Which results in:

```
1 5    1
2 7    2
3 3    3
4 5    4
5 dtype: int64
```

The default indices were replaced with the ones we've supplied to the `index` argument.

Creating Series from Python Dictionary

Now, let's use a Python dictionary to populate a Series:

```
1 series = pd.Series({"key1":1, "key2":2, "key3":3})
2 print(series)
```

This results in:

```
1 key1      1
2 key2      2
3 key3      3
4 dtype: int64
```

The order of the dictionary is reflected in the populated `Series`, so if we jumbled the keys around in the dictionary, that order would've been preserved here.

Here, if we supply the `index` argument, we can rearrange the existing key-value pairs, instead of just overriding the ones we've put in initially:

```
1 series = pd.Series({"key1":1, "key2":2, "key3":3},
2                      index=["key3", "key1", "key2", "invalidkey"])
```

We've also put in an "invalidkey" string in the list passed to the `index`. Since no index corresponds to this, it'll be mapped to a `NaN` (Not a Number) value, which is the default missing value in Pandas.

Now, let's print the `series` again:

```
1 print(series)
```

And this results in:

```
1 key3      3.0
2 key1      1.0
3 key2      2.0
4 invalidkey  NaN
```

Creating Series from Numpy Array

A `Series` may also be created from a Numpy array. Let us create a Numpy array then convert it into a Pandas `Series`:

```
1 import pandas as pd
2 import numpy as np
3
4 fruits = np.array(['apple', 'orange', 'mango', 'pear'])
5 series = pd.Series(fruits)
6 print(series)
```

This results in:

```
1 0    apple
2 1    orange
3 2    mango
4 3    pear
5 dtype: object
```

We've used the `array()` function on the `np` module to create a Numpy array of fruits. Then, as usual, using the `Series()` constructor, we've passed the array in and constructed a `Series` instance.

Accessing Series Elements

Now that we know how to create and populate a `Series`, let's take a look at how we can access individual elements. Much the same way you can access list elements, or `ndarray` elements, accessing `Series` elements follows the same notation:

```
1 element = series[0]
2 print(element)
```

Here, we've chosen the element at the position of `0`, and printed it using `print()`:

```
1 apple
```

Note: The *index* associated with the entry isn't necessarily the *position* of the element. We'll cover this distinction shortly.

Alternatively, you can use the *slice* notation:

```
1 fruits = ['apple', 'orange', 'mango', 'pear', 'kiwi', 'pineapple']
2 series = pd.Series(fruits)
3
4 elements = series[2:5]
5 print(elements)
```

This would select all elements between indices 2 (inclusive) and 5 (exclusive), and save them into the `elements` variable.

This code results in:

```
1 2      mango
2 3      pear
3 4      kiwi
4 dtype: object
```

It's worth noting that `elements` is also a `Series`, since we've selected a sub-`Series` of elements here. Using the slice notation, you can also select the first or last n elements:

```
1 first_two_elements = series[:2]
2 last_two_elements = series[-2:]
3
4 print(first_two_elements)
5 print(last_two_elements)
```

This results in:

```
1 0      apple
2 1      orange
3 dtype: object
4 4      kiwi
5 5      pineapple
6 dtype: object
```

Accessing Custom Indices

However, what happens if we shuffle the indices around? Let's modify this `fruits` Series and add the optional `index` argument:

```
1 fruits = ['apple', 'orange', 'mango', 'pear', 'kiwi', 'pineapple']
2 series = pd.Series(fruits, index=[2, 5, 3, 6, 1, 9])
3
4 first_two_elements = series[:2]
5 last_two_elements = series[-2:]
6
7 print(first_two_elements)
8 print(last_two_elements)
```

This results in:

```
1 2      apple
2 5      orange
3 dtype: object
4 1      kiwi
5 9      pineapple
6 dtype: object
```

Here, it becomes clear that the positions of the values in the list don't necessarily have to reflect the indices. The first two elements have 2 and 5 as their indices, but the last two have 1 and 9 respectively.

If you'd like to access elements, based on their *index* instead of *position*, you can simply pass in one index:

```
1 element1 = series[1]
2
3 print(element1)
```

When directly selecting an element via their index, Pandas will search by index, and not by position. In our case, this should print `kiwi`, which has an index of 1, but is on the 4th position:

```
1 kiwi
```

Alternatively, if you're using some other data type for the labels, you can directly access them as well:

```
1 decimal_values = [80, 121, 116, 104, 111, 110]
2 series = pd.Series(decimal_values, index=['P', 'y', 't', 'h', 'o', 'n'])
3 print(series)
```

Here, we've mapped the decimal values of each character constituents of the word `Python` and used their respective characters as indices. Printing this series results in:

```
1 P      80
2 y      121
3 t      116
4 h      104
5 o      111
6 n      110
7 dtype: int64
```

Now, let's access the value of the `y` index, as well as the value at the position of 4, since our `index` is now comprised of characters:

```
1 element1 = series['y']
2 element2 = series[4]
3 print(element1, element2)
```

This results in:

```
1 121 111
```

Here, Pandas inferred that we're searching by position, not index, since there is no index of 4 - we're using characters, not integers. Though, selecting elements one by one isn't very efficient. You can select multiple elements with a single statement:

```
1 # Positional search
2 elements_by_position = series[[1, 4]]
3
4 # Label search
5 elements_by_labels = series[['t', 'n']]
6
7 print(elements_by_position)
8 print(elements_by_labels)
```

This results in:

```
1 y      121
2 o      111
3 dtype: int64
4 t      116
5 n      110
6 dtype: int64
```

Note: You cannot mix-and-match labels and positions with this approach.

Series.head()

Now, let's take a look at a really handy method for printing values in a Series. The `head()` method prints the first 5 elements, by default, and is most commonly used to print out the first few values so you verify that it's been constructed correctly, or just take a peek at the data inside.

The method also accepts an optional argument, which you can use to specify how many elements you'd like to print. Let's for example print the first 10, even though our series only has 6 entries:

```
1 print(series.head(10))
```

This results in:

```
1 P      80
2 y     121
3 t     116
4 h     104
5 o     111
6 n     110
7 dtype: int64
```

Series.tail()

Finally, there's the `tail()` method, which does the exact opposite of `head()` - it prints the *last* n elements of a Series, and n=5 by default:

```
1 print(series.tail(3))
```

This will print the last three items:

```
1 h     104
2 o     111
3 n     110
4 dtype: int64
```

DataFrames

While `Series`' are essentially one-dimensional labeled arrays of any type of data, `DataFrames` are two-dimensional, heterogenous, labeled arrays of any type of data. Heterogenous means that not all “rows” need to be of equal size.

In this section, we'll take a look at some ways you can create a `DataFrame` and methods you can use to change their structure and contents.

We'll interchangeably be using an IDE and a [Jupyter Notebook](#)¹² to print out the `DataFrames` since Jupyter Notebooks offer a nice visual representation of `DataFrames`, when there are many values. Though, any IDE or text editor will also do the job, just by calling a `print()` statement on the `DataFrame` object, very much the same way we printed `Series` objects in the previous section.

Creating `DataFrames`

Whenever you create a `DataFrame`, whether you're creating one manually or generating one from a data source such as a file - the data *has* to be ordered in a tabular fashion, as a sequence of rows containing data. This implies that the rows share the same order of fields, i.e. if you want to have a `DataFrame` with information about a person's name and age, you want to make sure that all your rows contain the information in the same order.

For example, here's a table with two people, however, the order of the information is reversed:

Name	Age
John Doe	21
21	Jane Doe

This discrepancy causes the `DataFrame` to be faulty. Although it will happily save these values into a `DataFrame` instance, you'd run into issues when trying to work with it due to an unexpected string value in the `Age` column.

¹²<https://jupyter.org/>

Creating an Empty DataFrame

Creating an empty DataFrame is as simple as:

```
1 import pandas as pd
2 dataframe = pd.DataFrame()
```

This is again, much the same way you'd create an empty Series. DataFrames accept the same two arguments at creation time as Series':

```
1 dataframe = pd.DataFrame(data, index)
```

Both of which are optional and of the same valid data types as with Series'.

Note: In practice, most people abbreviate `dataframe` to just `df`.

Creating a DataFrame From Lists

Following the “sequence of rows with the same order of fields” principle, you can create a DataFrame from a list of lists that contains such a sequence, or from multiple lists `zip()`-ed together in such a way that they generate a table-like structure:

```
1 import pandas as pd
2
3 pepper_list = [
4     [50, "Bell pepper", "Not even spicy"],
5     [5000, "Espelette pepper", "Uncomfortable"],
6     [50000, "Chocolate habanero", "Practically ate pepper spray"]
7 ]
8
9 dataframe = pd.DataFrame(pepper_list)
10
11 print(dataframe)
```

	0	1	2
2	0 50	Bell pepper	Not even spicy
3	1 5000	Espelette pepper	Uncomfortable
4	2 50000	Chocolate habanero	Practically ate pepper spray

The same effect could've been achieved by having the data in multiple lists and `zip()`-ing them together:

```
1 import pandas as pd
2
3 scoville_values = [50, 5000, 500000]
4 pepper_names = ["Bell pepper", "Espellete pepper", "Chocolate habanero"]
5 pepper_descriptions = ["Not even spicy", "Uncomfortable", "Practically ate pepper spray"]
6
7 pepper_list = zip(scoville_values, pepper_names, pepper_descriptions)
8
9 dataframe = pd.DataFrame(pepper_list)
10
11 print(dataframe)
```

Here, we've got columns specified in individual lists, which we've `zip()`-ed together and constructed a `Dataframe`:

```
1      0           1           2
2 0    50   Bell pepper   Not even spicy
3 1  5000   Espellete pepper   Uncomfortable
4 2 500000 Chocolate habanero  Practically ate pepper spray
```

You may have noticed that the column and row labels aren't very informative in the `DataFrame` we've created.

`DataFrames` have additional optional arguments, such as the `columns` argument, which lets us specify the column names, with a simple list. Let's modify the code to add some column names when creating the `DataFrame`:

```
1 import pandas as pd
2
3 scoville_values = [50, 5000, 500000]
4 pepper_names = ["Bell pepper", "Espellete pepper", "Chocolate habanero"]
5 pepper_descriptions = ["Not even spicy", "Uncomfortable", "Practically ate pepper spray"]
6
7 pepper_list = zip(scoville_values, pepper_names, pepper_descriptions)
8
9 dataframe = pd.DataFrame(pepper_list, columns = ['Scoville', 'Name', 'Feeling'])
10
11 print(dataframe)
```

Which would give us the same output as before, with more meaningful column names:

```

1   Scoville           Name           Feeling
2   0      50       Bell pepper     Not even spicy
3   1     5000    Espellete pepper Uncomfortable
4   2    500000  Chocolate habanero Practically ate pepper spray

```

Creating a DataFrame From Dictionaries

Dictionaries are another way of providing data in column-wise fashion. Every column is given a list of values rows contain for it, in order:

```

1 pepper_dictionary = {
2     'columnName1' : [valueForRow1, valueForRow2, valueForRow3...],
3     'columnName2' : [valueForRow1, valueForRow2, valueForRow3...],
4     ...
5 }

```

Let's fill in our pepper-related data into a dictionary and construct a DataFrame from it:

```

1 import pandas as pd
2 pepper_dictionary = {
3     'Scoville' : [50, 5000, 500000],
4     'Name' : ["Bell pepper", "Espelette pepper", "Chocolate habanero"],
5     'Feeling' : ["Not even spicy", "Uncomfortable", "Practically ate pepper spray"]
6 }
7
8 dataframe = pd.DataFrame(pepper_dictionary)
9
10 print(dataframe)

```

Running this code, again, results in:

```

1   Scoville           Name           Feeling
2   0      50       Bell pepper     Not even spicy
3   1     5000    Espellete pepper Uncomfortable
4   2    500000  Chocolate habanero Practically ate pepper spray

```

Additionally, we can use the `DataFrame.from_dict()` function:

```
1 dataframe = pd.DataFrame.from_dict(pepper_dictionary)
```

Creating DataFrames from List of Dictionaries

You can also provide the data as a list of dictionaries in the following format:

```
1 listPepper = [
2     { columnName1 : valueForRow1, columnName2: valueForRow1, ... },
3     { columnName1 : valueForRow2, columnName2: valueForRow2, ... },
4     ...
5 ]
```

When we fill in our values, the pepper_list looks something like:

```
1 pepper_list = [
2     {'Scoville' : 50, 'Name' : 'Bell pepper', 'Feeling' : 'Not even spicy'},
3     {'Scoville' : 5000, 'Name' : 'Espelette pepper', 'Feeling' : 'Uncomfortable'},
4     {'Scoville' : 500000, 'Name' : 'Chocolate habanero', 'Feeling' : 'Practically ate pepper'},
5     r spray},
6 ]
```

And we would create the DataFrame in the same way as before:

```
1 dataframe = pd.DataFrame(pepper_list)
```

Reading a DataFrame From a File

Realistically, you'll rarely be creating DataFrames by hand like this, unless you're generating one to save it for later. In most cases, you'll be *reading/importing* data into a DataFrame from an external dataset you'd like to explore.

Pandas makes this task seamless and as easy as it can get, as it's one of the core functionalities and most commonly used features of the library.

There are many file types supported for reading and writing DataFrames. Each respective filetype function follows the same syntax `read_filetype()`, such as `read_csv()`, `read_excel()`, `read_json()`, `read_html()`, etc...

Note: We'll cover reading and writing CSV files in more detail in later parts of this chapter, including good practices and some caveats. Though, you don't need to know all the details to simply import a valid file like this.

A very common filetype for data storage and sharing is `.csv` (Comma-Separated-Values). The rows are provided as lines, with the values they contain separated by a delimiter (most often a comma). This is, intuitively, why they're named Comma-Separated-Values.

When reading CSV files, the default separator will be a `,`. You can set another delimiter via the `sep` argument. Let's save our pepper-related data into a `.csv` file:

```

1 Scoville,Name,Feeling
2 50,Bell pepper,Not even spicy
3 5.000,Espelette pepper,Uncomfortable
4 10.000,Serrano pepper,I regret this
5 60.000,Bird's eye chili,4th stage of grief
6 500.000,Chocolate habanero,Practically ate pepper spray
7 2.000.000,Carolina Reaper,Actually ate pepper spray

```

Note that the first line in the file are the column names, as with the vast majority of CSV files.

You can specify from which line Pandas starts reading the data, but, by default, it treats the *first line* of the CSV document as the *column name* line, assigning the values from there to the `columns` argument when constructing a DataFrame. When we load this CSV file in, it'll use `Scoville`, `Name`, and `Feeling` for the columns.

Then, the constituent rows of the DataFrame start at the *second* line:

```

1 import pandas as pd
2
3 dataframe = pd.read_csv('peppers.csv')
4 print(dataframe)

```

Which gives us the output:

	Scoville	Name	Feeling
0	50	Bell pepper	Not even spicy
1	5.000	Espelette pepper	Uncomfortable
2	10.000	Serrano pepper	I regret this
3	60.000	Bird's eye chili	4th stage of grief
4	500.000	Chocolate habanero	Practically ate pepper spray
5	2.000.000	Carolina Reaper	Actually ate pepper spray

Note: For other separators, such as ;, which isn't a really uncommon one, you can provide the `sep` argument:

```
1 dataframe = pd.read_csv('peppers.csv', sep=';')
```

When provided, each value between two ; symbols will be treated as a *-separated value*.

Manipulating DataFrames

Now that we've got DataFrame creation under our belt, we can take a look at how we can manipulate them and the data they contain, as well as access/locate their elements and explore what they've got in store.

Accessing Columns

Let's start with accessing individual, or multiple columns. Accessing columns is as simple as referencing it in a `DataFrame`, such as - `dataFrameName.ColumnName` or `dataFrameName['ColumnName']`. The second option is preferred since the column can have the same name as a pre-defined Pandas method, and using the first option, in that case, could cause unexpected behavior.

Let's select the `Name` column:

```
1 print(dataframe['Name'])
```

Or, we can use the slightly different notation:

```
1 print(dataframe.Name)
```

Both of these output:

```
1 0      Bell pepper
2 1      Espelette pepper
3 2      Chocolate habanero
4 Name: Name, dtype: object
```

What's the type of the resulting column? Let's check it out:

```
1 print(type(dataframe['Name']))
```

```
1 <class 'pandas.core.series.Series'>
```

The *underlying* mechanism that makes `DataFrames` work are Pandas' `Series` objects. Each column and row is actually just a `Series`. Selecting any of these will net you a `Series`, which you've already learned how to work with in the previous part of this chapter.

Now, let's select *two* columns, not just one. For example, we might want to truncate the `DataFrame` to just the `Name` and `Scoville` columns, omitting the `Feeling` column totally:

```
1 dataframe_truncated = dataframe[['Scoville', 'Name']]  
2  
3 print(dataframe_truncated)
```

This results in:

```
1      Scoville           Name  
2      0            50    Bell Pepper  
3      1          5000   Espellete pepper  
4      2        500000 Chocolate habanero
```

What we've done here is selected two columns (`Series`) and effectively created a new `DataFrame` with those two objects. We've saved it in a new `dataframe_truncated`, and then printed it out.

This object is of type `pandas.core.frame.DataFrame`, since it consists of multiple `Series` objects.

Although `DataFrame` rows are just `Series` objects as well, the primary intention of the `[]` notation is to select columns. To understand how we can access rows, we'll want to use the `loc[]` and `iloc[]` methods.

Accessing/Locating Elements

Pandas has two different ways of selecting individual elements - `loc[]` and `iloc[]`.

`loc[]` allows you to select rows and columns by using labels, like `row['Value']` and `column['Other Value']`. Meanwhile, `iloc[]` requires that you pass in the index of the entries you want to select, so you can only use numbers.

Let's see how this works in action:

```
1 # Location by label  
2 # Here, '5' is treated as the *label* of the index, not its value  
3 print(dataframe.loc[5])  
4 # Location by index  
5 print(dataframe.iloc[1])
```

This fetches and outputs the objects we're looking for in their entirety:

```
1 Scoville           2.000.000
2 Name              Carolina Reaper
3 Feeling            Actually ate pepper spray
4 Name: 5, dtype: object
5 Scoville           5.000
6 Name              Espelette pepper
7 Feeling            Uncomfortable
8 Name: 1, dtype: object
```

This also works for a group of rows, using the slice notation, such as from *0...n*:

```
1 print(dataframe.loc[:1])
```

This will retrieve all *rows* between the start of the DataFrame and 1:

```
1   Scoville          Name        Feeling
2  0     50      Bell pepper  Not even spicy
3  1    5.000    Espelette pepper  Uncomfortable
```

It's important to note that `iloc[]` always expects an integer. `loc[]` supports other data types as well. We can use an integer here too, though we can also use other data types such as strings.

You can also access specific values of rows, instead of retrieving the entire row. For example, we might want to access the 2nd row, though only return its `Name` value, not the entire Series:

```
1 print(dataframe.loc[2, 'Name'])
```

This returns:

```
1 Chocolate habanero
```

Columns can also be accessed by using `loc[]` and `iloc[]`. For example, we'll access all rows, from *0...n* where *n* is the number of rows and fetch the first column, by modifying the syntax a bit:

```
1 print(dataframe.iloc[:, 1])
2 # print(dataframe.loc[:, 'Name'])
```

```
1 0      Bell pepper
2 1      Espelette pepper
3 2      Serrano pepper
4 3      Bird's eye chili
5 4      Chocolate habanero
6 5      Carolina Reaper
7 Name: Name, dtype: object
```

While we're there, we can also access column names through indexing as well, since we can access all columns via `dataframe.columns`:

```
1 print('Columns: ', dataframe.columns)
2 print('Column at 1st index: ', dataframe.columns[1])

1 Columns: Index(['Scoville', 'Name', 'Feeling'], dtype='object')
2 Column at 1st index: Name
```

Manipulating Indices

Indices are row “labels” in a DataFrame, and we can reference them when we want to select rows as well. Since we didn’t change the default indices Pandas assigns to DataFrames upon their creation, all our rows have been labeled with integers from 0 and up. This is the same behavior you could’ve seen with default index values for Series objects and DataFrames before.

The first way we can change the indexing of our DataFrame is by using the `set_index()` method. Any Series works wonderfully here, and you can set any indices by passing in a series:

```
1 indices = pd.Series([25, 150, 123])
2 dataframe = dataframe.set_index(indices)
```

This would set the indices from 0...n to 25, 150, 123 like we’ve seen before. However, another use-case of this method is to set an *existing column* as the index. For example, we can set our Scoville column (Series) as the new index column.

Note: This method doesn’t change the DataFrame in-place, and returns a new DataFrame with the changed index. To persist the change, we can either re-assign it to the same variable, or use the `inplace` argument, set to True:

```

1 import pandas as pd
2
3 pepper_list = [
4     {'Scoville' : 50, 'Name' : 'Bell pepper', 'Feeling' : 'Not even spicy'},
5     {'Scoville' : 5000, 'Name' : 'Espelette pepper', 'Feeling' : 'Uncomfortable'},
6     {'Scoville' : 500000, 'Name' : 'Chocolate habanero', 'Feeling' : 'Practically ate pep\
7 per spray'},
8 ]
9
10 dataframe = pd.DataFrame(pepper_list)
11 dataframe = dataframe.set_index('Scoville')
12
13 print(dataframe)

```

Here, we've chosen to re-assign the `dataframe` to itself. You can also assign it to a new `DataFrame` if you'd like to keep *both*:

	Name	Feeling
Scoville		
50	Bell pepper	Not even spicy
5000	Espelette pepper	Uncomfortable
500000	Chocolate habanero	Practically ate pepper spray

This would work just as well:

```

1 dataframe = pd.DataFrame(pepper_list)
2 dataframe.set_index('Scoville', inplace=True)
3
4 print(dataframe)

```

Now that we already have a non-default index, let's further customize the index with `reindex()`. The `reindex()` function *conforms* (reorders to conform) the existing `DataFrame` to a new set of labels. For example, we've got the `Scoville` column as our index. We can modify this index with a new set of values using `reindex()`.

Additionally, Pandas will automatically fill the row values with `NaN` for every index that can't be matched with an existing row:

```

1 new_index = [50, 5000, 'New value not present in the data frame']
2 dataframe = dataframe.reindex(new_index)

```

This reindexes the `DataFrame`, and changes our `Scoville` index with the newly added values:

```

1                               Name      Feeling
2 Scoville
3 50                         Bell pepper Not even spicy
4 5000                        Espelette pepper Uncomfortable
5 New value not present in the data frame      NaN      NaN

```

Since no existing row corresponds to the string we've added, instead of having our Chocolate habanero, we've got two `NaN` values, indicating missing values. You can control what value Pandas uses to fill in the missing values by setting the optional parameter `fill_value`:

```
1 dataframe = dataframe.reindex(new_index, fill_value=0)
```

This time around, instead of `NaN` values, we've got 0s:

```

1                               Name      Feeling
2 Scoville
3 50                         Bell pepper Not even spicy
4 5000                        Espelette pepper Uncomfortable
5 New value not present in the data frame      0      0

```

Note: We'll cover "*Handling Missing Values in a DataFrame*" in a later part of this chapter in more detail.

Since we have set a new index for our DataFrame, `loc[]` now works with that index as well:

```

1 dataframe.loc[5000]
2 # dataframe.iloc[5000] outputs the same in this case

```

This results in:

```

1 Name      Espelette pepper
2 Feeling    Uncomfortable
3 Name: 5000, dtype: object

```

Resetting Index

We've played around with the index a bit. Now, let's turn it back to the default one, since we don't really want it to stay like this:

```
1 dataframe.reset_index(inplace=True)
2 print(dataframe)
```

`reset_index()` isn't in-place by default, and you can either re-assign the DataFrame to itself or just set the `inplace` flag to True:

```
1                               Scoville      Name      Feeling
2   0                           50    Bell pepper  Not even spicy
3   1                          5000 Espelette pepper Uncomfortable
4   2 New value not present in the data frame           0          0
```

The actual index is reset, and back to the default values, and we've still got a remnant of the old index values in a separate column now, `Scoville`. This isn't the index anymore, but a column. In some cases, where you assign a new index to a DataFrame, and then run `reset_index()`, you might end up with an unnecessary old *index column* that you don't need anymore. To avoid having it stay after we've reset the index, you'll want to use the `drop` flag:

```
1 dataframe.reset_index(inplace=True, drop=True)
2 print(dataframe)
```

Now, this results in:

```
1             Name      Feeling
2   0    Bell pepper  Not even spicy
3   1 Espelette pepper Uncomfortable
4   2           0          0
```

Manipulating Rows

With that done, let's take a look at one of the two building block of each DataFrame - rows, and how we can manipulate them.

Adding New Rows

Adding and removing rows becomes simple if you're comfortable with using `loc[]`. If you try setting a value to a row that doesn't exist, it's created, with that value:

```
1  dataframe = pd.DataFrame(pepper_list)
2
3  dataframe.loc[50] = [10000, 'Serrano pepper', 'I regret this']
4  print(dataframe)
```

When we print this `dataframe` now, it'll contain a new index, added to the end:

```
1      Scoville           Name          Feeling
2  0        50     Bell pepper    Not even spicy
3  1      5000   Espelette pepper  Uncomfortable
4  2  500000  Chocolate habanero  Practically ate pepper spray
5  50     10000    Serrano pepper       I regret this
```

Removing Rows

If you want to remove a row, you pass its index to the `drop()` function which accepts an optional parameter, `axis`. The `axis` can be set to `0/index` or `1/columns`. Depending on this, the `drop()` function either drops the row it's called upon, or the column it's called upon.

Not specifying a value for the `axis` parameter will delete the corresponding row by default, as `axis` is `0` by default. This operation, as a lot of other operations, isn't in-place by default as well:

```
1  dataframe.drop(1, inplace=True)
2  print(dataframe)
```

Now, let's see what the `dataframe` looks like:

```
1      Scoville           Name          Feeling
2  0        50     Bell pepper    Not even spicy
3  2  500000  Chocolate habanero  Practically ate pepper spray
```

Renaming Rows

You can also very easily rename rows that already exist in the table. The `rename()` function accepts a dictionary of changes you wish to make:

```

1 dataframe.rename({0:"First", 1:"Second"}, inplace=True)
2 print(dataframe)

```

Let's take a peek at the `dataframe` now:

```

1      Scoville           Name          Feeling
2 First     50    Bell pepper  Not even spicy
3 Second   5000  Espelette pepper Uncomfortable
4 2       500000 Chocolate habanero Practically ate pepper spray

```

Note that `drop()` and `rename()` both accept the optional parameter - `inplace`. Setting this to `True` (`False` by default) will tell Pandas to change the original `DataFrame` instead of returning a new one. If left unset, you'll have to pack the resulting `DataFrame` into a new one to persist the changes.

Dropping Duplicate Rows

Sometimes, you might have duplicate data in rows. This is better left removed, as it can skew your data summaries and most aggregation functions if there's excessive amounts of duplicate data.

To drop duplicates, we can simply use the `drop_duplicates()` helper function, which will find all duplicate rows and drop them. Let's first add two rows, with the same content:

```

1 dataframe.loc[3] = [60.000, "Bird's eye chili", "4th stage of grief"]
2 dataframe.loc[4] = [60.000, "Bird's eye chili", "4th stage of grief"]
3
4 print(dataframe)

```

This `dataframe` now consists of:

```

1      Scoville           Name          Feeling
2 0     50.0    Bell pepper  Not even spicy
3 1   5000.0  Espelette pepper Uncomfortable
4 2 500000.0 Chocolate habanero Practically ate pepper spray
5 3     60.0  Bird's eye chili  4th stage of grief
6 4     60.0  Bird's eye chili  4th stage of grief

```

Now, to remove the duplicate rows, let's call `drop_duplicates()`, setting `inplace` to `True`:

```
1 dataframe.drop_duplicates(inplace=True)
2 print(dataframe)
```

Now, all the duplicate rows are removed:

```
1      Scoville           Name          Feeling
2      0      50.0    Bell pepper    Not even spicy
3      1     5000.0   Espelette pepper  Uncomfortable
4      2  500000.0 Chocolate habanero Practically ate pepper spray
5      3      60.0   Bird's eye chili      4th stage of grief
```

Manipulating Columns

Now, let's take a look at the second building block of `DataFrames` - columns, and how we can manipulate them.

Adding New Columns

Adding columns is performed with the same approach used to add rows. If you set a value to an non-existing column, a new one is made with that value:

```
1 dataframe = pd.DataFrame(pepper_list)
2
3 dataframe['Color'] = ['Green', 'Bright Red', 'Brown']
4 print(dataframe)
```

Output:

```
1      Scoville           Name          Feeling      Color
2      0      50    Bell pepper    Not even spicy    Green
3      1     5000   Espelette pepper  Uncomfortable  Bright Red
4      2  500000 Chocolate habanero Practically ate pepper spray  Brown
```

Removing Columns

Also, similarly to rows, columns can be removed by calling the `drop()` function, the only difference being that you have to set the optional parameter `axis` to `1`:

```
1 dataframe.drop('Feeling', axis=1, inplace=True)
2 print(dataframe)
```

This removes our column from the `dataframe`:

```
1      Scoville           Name      Color
2    0       50     Bell pepper    Green
3    1     5000   Espelette pepper Bright Red
4    2   500000  Chocolate habanero    Brown
```

Renaming Columns

When renaming columns, we use the same `rename()` function as for rows. This time around, we'll specifically set the `columns` argument, and add a dictionary of the old value and the new value for the column name.

Let's change our `Feeling` column to the `Measure of Pain` column:

```
1 dataframe = pd.DataFrame(pepper_list)
2 dataframe.rename(columns={"Feeling": "Measure of Pain"}, inplace=True)
3 print(dataframe)
```

Now, when we print the `dataframe`, the old `Feeling` column will be renamed:

```
1      Scoville           Name      Measure of Pain
2    0       50     Bell pepper    Not even spicy
3    1     5000   Espelette pepper Uncomfortable
4    2   500000  Chocolate habanero Practically ate pepper spray
```

Again, same as with removing/renaming rows, you can set the optional parameter `inplace` to `True` if you want the original `DataFrame` to be modified instead of the function returning a new `DataFrame`.

DataFrame Shapes

At any given point, if you'd like to check the *shape* of a `DataFrame`, you can easily access the `shape` property of the instance. Given our usual `pepper_list`, we can check the `shape` by printing it the property:

```
1 dataframe = pd.DataFrame(pepper_list)
2 print(dataframe.shape)
```

This will give us a succinct, and clear indication of how its shaped:

```
1 (3, 3)
```

Grouping Data in DataFrames

Grouping data is a common task. It's the process of containerizing certain groups of data, based on some criteria, into categories. Our pepper_list example won't be of too much use here, since we don't have various peppers with the same Scoville level, which makes grouping obsolete.

Let's quickly make a new raw dataset:

```
1 import pandas as pd
2
3 students = {
4     'Name': ['John', 'John', 'Grace', 'Grace', 'Benjamin', 'Benjamin', 'Benjamin',
5             'Benjamin', 'John', 'Alex', 'Alex', 'Alex'],
6     'Position': [2, 1, 1, 4, 2, 4, 3, 1, 3, 2, 4, 3],
7     'Year': [2009, 2010, 2009, 2010, 2010, 2011, 2012, 2011, 2013, 2013, 2012],
8     'Marks':[408, 398, 422, 376, 401, 380, 396, 388, 356, 402, 368, 378]
9 }
10 dataframe = pd.DataFrame(students)
```

Here, we've created a DataFrame of students, where there are a lot of duplicate names, belonging to different people, in different years of college, as well as different years of enrollment.

Now, these years of enrollment aren't incremental in the DataFrame, we've got a sequence of 2009, 2010, 2009, 2010, etc... The order of insertion is preserved, as usual.

Now, let's *group* students together by the Year column:

```
1 group = dataframe.groupby('Year')
```

Now, what is this group? Let's check its type:

```
1 print(type(group))
```

This results in:

```
1 <class 'pandas.core.groupby.generic.DataFrameGroupBy'>
```

The `DataFrameGroupBy` object is a specific type of object, used to hold the result of the `groupby()` function. You can access different properties of this object, such as :

- `groups` - A dictionary of groups and their labels.
- `indices` - A dictionary of groups and their indices.

And it also offers a pretty handy method:

- `get_group()` - Returns a group, converted into a new `DataFrame` with the entries collected when grouping.

Let's print out the available groups:

```
1 print(group.groups)
```

This gives us information on the available groups:

```
1 {2009: [0, 2], 2010: [1, 3, 4, 5], 2011: [6, 8], 2012: [7, 11], 2013: [9, 10]}
```

Here, we've got a dictionary of `label:elements`. Each group label has a list of added students.

Let's take year `2010` from these and print it out:

```
1 print(group.get_group(2010))
```

This gets the group, which is “incidentally” a `DataFrame`:

```
1   Marks      Name  Position  Year
2   1     398    John        1  2010
3   3     376   Grace        4  2010
4   5     380  Benjamin      4  2010
```

You can group data based on *any* column. We could've grouped by Name or Position as well. Grouping data is a common and very powerful way to wrangle information into different formats.

Descriptive Statistics

When working with datasets that have various values, a common thing is to quickly take a look at some general descriptive statistics, such as the mean, median, maximum and minimum values, etc.

These aren't very detailed and don't provide in-depth understanding of the dataset, but are the first and one of the most fundamental views you can get of a dataset. Pandas DataFrames include a handy `describe()` method, which ignores all non-numerical columns, and calculates some basic information (basic description) of the numerical columns, formats it and returns a table-like view to the user.

Let's create a new DataFrame, that contains a few student names, as well as their respective scores in Math and English:

```
1 import pandas as pd
2
3 students = {
4     'Name': ['John', 'Alice', 'Joseph', 'Alex'],
5     'English': [64, 78, 68, 58],
6     'Maths': [76, 54, 72, 64]
7 }
8
9 dataframe = pd.DataFrame(students)
10 print(dataframe)
```

If we print this as usual, we'll see the entire dataframe:

```
1      English    Maths    Name
2  0        64       76   John
3  1        78       54  Alice
4  2        68       72 Joseph
5  3        58       64  Alex
```

Now, if we want to take a look at the descriptive statistics of the `English` and `Math` columns, we can call `dataframe.describe()`:

```
1 df.describe()
```

Calling this method returns:

```
1      English    Maths
2 count  4.000000  4.000000
3 mean   67.000000 66.500000
4 std    8.406347  9.712535
5 min    58.000000 54.000000
6 25%   62.500000 61.500000
7 50%   66.000000 68.000000
8 75%   70.500000 73.000000
9 max    78.000000 76.000000
```

As you can see, the `describe()` method completely ignored the “Name” column since it’s not numerical, which is what we want. This simplifies things since you don’t need to worry about removing non-numerical columns before calculating the numerical stats you want.

We now have a concise and clear view of some general statistics, such as the `count`, `mean`, `std`, a breakdown by quartiles, as well as the `min` and `max` values.

DataFrame.head() and DataFrame.tail()

The same as with `Series` objects, you can use `DataFrame.head()` and `DataFrame.tail()` to get truncated, efficient chunks of `DataFrames`. Let’s construct one and print the first and last 3 elements:

```
1 import pandas as pd
2
3 students = {
4     'Name': ['John', 'Alice', 'Joseph', 'Alex', 'Benjamin', 'Frankie'],
5     'English': [64, 78, 68, 58, 65, 78],
6     'Maths': [76, 54, 72, 64, 93, 88]
7 }
8
9 dataframe = pd.DataFrame(students)
10 print(dataframe.head(3))
11 print(dataframe.tail(3))
```

This results in:

```
1      Name  English  Maths
2  0    John       64     76
3  1   Alice       78     54
4  2  Joseph       68     72
5      Name  English  Maths
6  3    Alex       58     64
7  4 Benjamin       65     93
8  5  Frankie       78     88
```

Without providing the number of elements as we did, the default value is 5.

Reshaping DataFrames

Now that we've got `Series` and `DataFrame` classes under our belt, and we've covered how to create `DataFrames`, work with their contents, manipulate and customize them - we've got one last thing to cover with the `DataFrames` themselves, before we can jump into handling missing values, merging `DataFrames` and reading/writing file formats like CSV using Pandas.

We've covered some rudimentary reshaping operations before, like selecting specific rows and generating `DataFrames` using them, changing column/row names, etc.

In this section, we'll cover *DataFrame Reshaping*. Specifically, we'll take a look at 5 more advanced reshaping operations:

- Transpose
- Stack
- Unstack

- Melt
- Pivot

Before we apply any of these operations, let's first make an exemplary DataFrame:

```
1 import pandas as pd
2
3 students = {
4     'Name': ['John', 'John', 'Grace', 'Grace', 'Benjamin', 'Benjamin', 'Benjamin',
5             'Benjamin', 'John', 'Alex', 'Alex', 'Alex'],
6     'Year': [2009, 2010, 2009, 2010, 2010, 2010, 2011, 2012, 2011, 2013, 2013, 2012],
7     'Marks': [408, 398, 422, 376, 401, 380, 396, 388, 356, 402, 368, 378]
8 }
9 df = pd.DataFrame(students)
10
11 print(df)
```

This DataFrame now contains 11 students, their respective years and marks:

```
1      Name  Year  Marks
2  0      John  2009    408
3  1      John  2010    398
4  2     Grace  2009    422
5  3     Grace  2010    376
6  4   Benjamin  2010    401
7  5   Benjamin  2010    380
8  6   Benjamin  2011    396
9  7   Benjamin  2012    388
10 8      John  2011    356
11 9      Alex  2013    402
12 10     Alex  2013    368
13 11     Alex  2012    378
```

Transposing a DataFrame

Transposition, as the name implies, is the act of switching the places of two or more elements. To *transpose* a DataFrame means to *transpose* its *index* and *columns*.

To transpose a DataFrame, you call the `transpose()` function, which returns a new DataFrame that contains the transposed items. You can either assign it to a new reference variable or an old one, but the function doesn't change the DataFrame in-place.

Let's `transpose()` the DataFrame we've set up and save the results into `df_transposed`:

```

1 df_transposed = df.transpose()
2 print(df_transposed)

```

Now, the newly returned DataFrame looks like this:

```

1          0   1   2   3   4   5   6   7   8   9   10\
2      11
3 Name  John  John Grace Grace Benjamin Benjamin Benjamin John Alex Alex\
4     Alex
5 Year  2009 2010 2009 2010 2010 2010 2011 2012 2011 2013 2013\
6     2012
7 Marks  408  398  422  376  401  380  396  388  356  402  368\
8     378

```

The index of the DataFrame turned into the columns, while the columns turned into the index. Alternatively, instead of the transpose() function, you can simply call DataFrame.T:

```

1 df_transposed = df.T
2 print(df_transposed)

```

This also results in:

```

1          0   1   2   3   4   5   6   7   8   9   10\
2      11
3 Name  John  John Grace Grace Benjamin Benjamin Benjamin John Alex Alex\
4     Alex
5 Year  2009 2010 2009 2010 2010 2010 2011 2012 2011 2013 2013\
6     2012
7 Marks  408  398  422  376  401  380  396  388  356  402  368\
8     378

```

Stacking a DataFrame

The stack() function reshapes the DataFrame so that the columns become parts of multi-level indices. It *stacks* the columns similar to how you'd stack books on top of each other - Instead of having them side-by-side, they're now on top of each other.

The function doesn't alter the DataFrame in-place, and you'll have to assign the returned value to persist it. Let's stack() our DataFrame:

```
1 df_stacked = df.stack()  
2 print(df_stacked)
```

This results in a DataFrame with stacked columns:

```
1 0 Name John  
2 Year 2009  
3 Marks 408  
4 1 Name John  
5 Year 2010  
6 Marks 398  
7 2 Name Grace  
8 Year 2009  
9 Marks 422  
10 3 Name Grace  
11 Year 2010  
12 Marks 376  
13 4 Name Benjamin  
14 Year 2010  
15 Marks 401  
16 5 Name Benjamin  
17 Year 2010  
18 Marks 380  
19 6 Name Benjamin  
20 Year 2011  
21 Marks 396  
22 7 Name Benjamin  
23 Year 2012  
24 Marks 388  
25 8 Name John  
26 Year 2011  
27 Marks 356  
28 9 Name Alex  
29 Year 2013  
30 Marks 402  
31 10 Name Alex  
32 Year 2013  
33 Marks 368  
34 11 Name Alex  
35 Year 2012  
36 Marks 378
```

Now, each *instance*, which in our case is a student, has their own Name, Year and Marks fields, instead of these columns being effectively shared between them. Each of these is a Series, and we can extract each student via an index value:

```
1 student = df_stacked.loc[1]
```

This will get the instance at index 1 and assign it to student. Let's print the student and its type:

```
1 print(student)
2 print(type(student))
```

This results in:

```
1 Name      John
2 Year      2010
3 Marks     398
4 dtype: object
5 <class 'pandas.core.series.Series'>
```

You can also access the fields of specific instances. For example, let's access only the `Marks` field:

```
1 print(student.Marks)
```

This results in:

```
1 398
```

Unstacking a DataFrame

Unstacking is the reverse process of *stacking*. Let's unstack() the DataFrame we've previously stacked:

```
1 df_unstacked = df_stacked.unstack()
2 print(df_unstacked)
```

This results in:

```
1      Name  Year  Marks
2  0    John  2009   408
3  1    John  2010   398
4  2   Grace  2009   422
5  3   Grace  2010   376
6  4 Benjamin  2010   401
7  5 Benjamin  2010   380
8  6 Benjamin  2011   396
9  7 Benjamin  2012   388
10 8    John  2011   356
11 9    Alex  2013   402
12 10   Alex  2013   368
13 11   Alex  2012   378
```

It's successfully unstacked, and looks just like the original `df`!

Pivoting a DataFrame

Most of the time, we're working with *narrow-form* (tidy-form) data. These are also commonly known as *long-form* or *log-data* because the data is written as a log of observations, one beneath the other. In this type, there's a column for each variable/feature, and each row is a single instance/observation.

We've been working with this type of data so far. For example:

```
1      Name  Year  Marks
2  0    John  2009   408
3  1 Victoria  2010   398
```

Each instance is given an index, and each instance has values depending on the feature/column.

By contrast, *wide-form* (short-form) data has the values of the independent variables as the row and column headings - while the values of the dependent variables are contained in the cells.

For example, we could reshape the two `DataFrame` instances (`Series`) objects into wide-form:

```

1 Year      2009   2010
2 Name
3 John      408.0   NaN
4 Victoria  NaN     398.0

```

Here, we've chosen the `Year` values to be our columns, while we've used `Name` as the index of the new DataFrame, while the values are the `Marks`. John doesn't have a record of his `Marks` in 2010, so a `NaN` is inserted. Generally, when turning narrow-form data into wide-form data, you'll commonly see some missing values since datasets are rarely perfectly ready for wide-form.

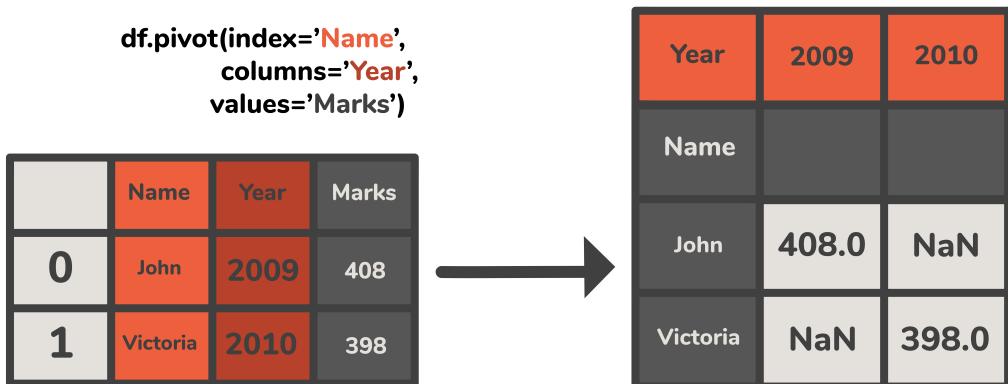
Let's simplify the DataFrame we've been working with and see how we can do this with the `pivot()` function:

```

1 students = {
2     'Name': ['John', 'Victoria'],
3     'Year': [2009, 2010],
4     'Marks': [408, 398]
5 }
6 df = pd.DataFrame(students)
7 df_pivoted = df.pivot(index='Name', columns='Year', values='Marks')
8 print(df_pivoted)

```

The `Marks` are dependent on the `Name` and `Year`, and each value will be *the mark* for that *year*, belonging to a *name*. The *mark* is shared between these two. Since this might be a bit difficult to visualize, here's an illustration:



And this code results in:

```
1 Year      2009   2010
2 Name
3 John      408.0   NaN
4 Victoria  NaN    398.0
```

This DataFrame is now in wide-form. Wide-form data is very commonly used for *heatmaps*, since they are, inherently, wide-form.

Additionally, you can turn a narrow-form dataset into a wide-form dataset to isolate certain features based on another feature. For example, you might have a dataset containing the stays in a hotel, from 2017 to 2020. If you'd like to plot Bar Charts, or Line Plots of the month-by-month occupancy, but also take the *year* of the reservation into consideration, you can turn the dataset into a wide-form dataset, pivoting the months and their values around the *year* feature.

Then, when plotting, you can have multiple plots, one for each year.

Melting a DataFrame

Melting a DataFrame is the process of reshaping it from wide-form to narrow-form. This is achieved by “melting” away, until there are only two columns - `variable` and `value`.

When melting, we discern two types of variable columns - `id_vars` and `value_vars`. The `id_vars` are identifier columns. This is typically just one column, but you can have multiple ones. The `value_vars` are the measurements of these identifiers, which are “unpivoted” to the row axis.

Let's use our original DataFrame, and `melt()` it:

```
1 import pandas as pd
2
3 students = {
4     'Name': ['John', 'Victoria'],
5     'Year': [2009, 2010],
6     'Marks':[408, 398]
7 }
8 df = pd.DataFrame(students)
9 # Check original DataFrame
10 print(df)
11
12 # Melt DataFrame with default settings
13 df_melted = df.melt()
14 # Check melted DataFrame
15 print(df_melted)
```

This code now results in:

```
1      Name  Year  Marks
2  0    John  2009    408
3  1 Victoria  2010    398
4
5      variable     value
6  0      Name       John
7  1      Name    Victoria
8  2      Year      2009
9  3      Year      2010
10     4      Marks      408
11     5      Marks      398
```

First, we've got a regular table-like DataFrame, which we've melted with the `melt()` function. It completely melted it, to the point where we've only got `variable` and `value` columns. The columns we've had previously - `Name`, `Year` and `Marks` became instances of the `variable` column, while their values are now under the `value` column. The instances of the `variable` column are identifiers, while the values under the `value` column are the values of those identifiers.

You can change the names of these new columns, by setting the `var_name` and `value_name` arguments:

```
1 df_melted = df.melt(var_name='Variables', value_name='Values')
2 print(df_melted)
```

This results in:

```
1      Variables    Values
2  0      Name      John
3  1      Name  Victoria
4  2      Year     2009
5  3      Year     2010
6  4      Marks     408
7  5      Marks     398
```

Though, probably more importantly, you can specify the `id_vars` and `value_vars` while melting the `DataFrame`. The column(s) passed to `id_vars` will be used as the new identifiers, while the column(s) passed to `value_vars` will be used for the new values. For example, let's use the `Name` column as the new identifiers, while using `Marks` as the values:

```
1 df_melted = df.melt(id_vars='Name', value_vars='Marks')
2 print(df_melted)
```

Now, when we print the melted `DataFrame`:

```
1      Name variable  value
2  0      John     Marks   408
3  1 Victoria     Marks   398
```

Instead of the variables, such as `Name`, `Year` and `Marks` being the identifiers, we've got instances of the `Name` column as the identifiers. Additionally, their values are specifically set to the values of the `Marks`.

Melting and Unmelting a DataFrame

Using `melt()` and `pivot()`, you can go back and forth between wide-form and narrow-form. Sadly, it's not as easy as just calling `DataFrame.melt()` and `DataFrame.pivot()` - you'll have to set a couple of arguments and reset the index.

Let's go through the process of constructing a `DataFrame`, melting it and then pivoting it back to the original state:

```
1 import pandas as pd
2
3 students = {
4     'Name': ['John', 'Victoria'],
5     'Year': [2009, 2010],
6     'Marks':[408, 398]
7 }
8 df = pd.DataFrame(students)
9 print('Original DataFrame: \n', df)
10
11 melted_df = df.melt(id_vars='Name', var_name='Variable', value_name='Value')
12 print('Melted DataFrame: \n', melted_df)
13
14 unmelted_df = melted_df.pivot(index='Name', columns='Variable')['Value'].reset_index()
15 unmelted_df.columns.name = None
16
17 print('Unmelted DataFrame: \n', unmelted_df)
```

Here, we've got our usual DataFrame. Then, we've melted it via the `melt()` function, providing the `Name` as the identifier variable, and setting the names of the new columns as `Variable` and `Value`.

Then, we've used `pivot()` to pivot the DataFrame around the `Name` variable, which is set as the new index. We've used the `Variable` column (created by melting the DataFrame) to construct a new set of columns. In our case, it turns the `Year` and `Marks` back from rows into columns. Then, since we're now using the `Name` column as the index, we'll want to run a `reset_index()`, and drop the stacked `Value` column.

Running this code results in:

```
1 Original DataFrame:
2         Name  Year  Marks
3  0      John  2009    408
4  1  Victoria  2010    398
5
6 Melted DataFrame:
7         Name  Variable  Value
8  0      John      Year  2009
9  1  Victoria      Year  2010
10 2      John      Marks   408
11 3  Victoria      Marks   398
12
13 Unmelted DataFrame:
14        Name  Marks  Year
15  0      John    408  2009
16  1  Victoria    398  2010
```

How to Iterate Over Rows

With the creation, basic reading as well as basic and more advanced operations, let's take a look at how we can iterate through rows of a DataFrame. In this section, we'll take a look at three different approaches:

- `items()`
- `iterrows()`
- `itertuples()`

Iterating DataFrames with `items()`

Let's set up a DataFrame with some data of fictional people:

```
1 import pandas as pd
2
3 dataframe = pd.DataFrame({
4     'first_name': ['John', 'Jane', 'Marry', 'Victoria', 'Gabriel', 'Layla'],
5     'last_name': ['Smith', 'Doe', 'Jackson', 'Smith', 'Brown', 'Martinez'],
6     'age': [34, 29, 37, 52, 26, 32]},
7     index=['id001', 'id002', 'id003', 'id004', 'id005', 'id006'])
```

Note that we are using custom IDs as our DataFrame's index. Let's take a look at how the DataFrame looks like:

```
1 print(dataframe)
```


	first_name	last_name	age
1	John	Smith	34
2	Jane	Doe	29
3	Marry	Jackson	37
4	Victoria	Smith	52
5	Gabriel	Brown	26
6	Layla	Martinez	32
7			

Now, to iterate over this DataFrame, we'll use the `items()` function:

```
1  dataframe.items()
```

This returns a generator:

```
1  <generator object DataFrame.items at 0x7f3c064c1900>
```

We can use this to generate pairs of `col_name` and `data`. These pairs will contain a column name and every row of data for that column. Let's loop through column names and their data:

```
1  for col_name, data in dataframe.items():
2      print("col_name:", col_name, "\ndata:", data)
```

This results in:

```
1  col_name: first_name
2  data:
3  id001      John
4  id002      Jane
5  id003      Marry
6  id004      Victoria
7  id005      Gabriel
8  id006      Layla
9  Name: first_name, dtype: object
10 col_name: last_name
11 data:
12 id001      Smith
13 id002      Doe
14 id003      Jackson
15 id004      Smith
16 id005      Brown
17 id006      Martinez
18 Name: last_name, dtype: object
19 col_name: age
20 data:
21 id001    34
22 id002    29
23 id003    37
24 id004    52
25 id005    26
26 id006    32
27 Name: age, dtype: int64
```

We've successfully iterated over all rows in each column. Notice that the index column stays the same over the iteration, as this is the associated index for the values. If you don't define an index, Pandas will enumerate the index column accordingly.

We can also print a particular row by passing the index number to the `data` as we do with Python lists:

```
1 for col_name, data in dataframe.items():
2     print("col_name:", col_name, "\n\tdata:", data[1])
```

As usual, lists are zero-indexed, so `data[1]` refers to the second row:

```
1 col_name: first_name
2 data: Jane
3 col_name: last_name
4 data: Doe
5 col_name: age
6 data: 29
```

We can also pass the index label to `data`:

```
1 for col_name, data in dataframe.items():
2     print("col_name:", col_name, "\n\tdata:", data['id002'])
```

The output would be the same as before:

```
1 col_name: first_name
2 data: Jane
3 col_name: last_name
4 data: Doe
5 col_name: age
6 data: 29
```

Iterating DataFrames with `iterrows()`

While `dataframe.items()` iterates over the rows in column-wise, doing a cycle for each column, we can use `iterrows()` to get the entire row-data of an index.

Let's try iterating over the rows with `iterrows()`:

```
1 for i, row in dataframe.iterrows():
2     print(f"Index: {i}")
3     print(f"\t{row}\n")
```

In the for loop, `i` represents the indices (our DataFrame has indices from `id001` to `id006`) and `row` contains the data for that specific index in all columns. This is a much more intuitive way to list through the `dataframe` and print each row's information.

Let's run this code:

```
1 Index: id001
2 first_name      John
3 last_name       Smith
4 age            34
5 Name: id001, dtype: object
6
7 Index: id002
8 first_name      Jane
9 last_name       Doe
10 age            29
11 Name: id002, dtype: object
12
13 Index: id003
14 first_name      Marry
15 last_name       Jackson
16 age            37
17 Name: id003, dtype: object
18
19 ...
```

Likewise, we can iterate over the rows in a certain column. Simply passing the *index number* or the *column name* to the `row`. For example, we can selectively print the first column of the row like this:

```
1 for i, row in dataframe.iterrows():
2     print(f"Index: {i}")
3     print(f"{row['0']}")
```

Or:

```
1 for i, row in dataframe.iterrows():
2     print(f"Index: {i}")
3     print(f"{row['first_name']}")
```

They both produce the same output:

```
1 Index: id001
2 John
3 Index: id002
4 Jane
5 Index: id003
6 Marry
7 Index: id004
8 Victoria
9 Index: id005
10 Gabriel
11 Index: id006
12 Layla
```

Iterating DataFrames with *itertuples()*

The `itertuples()` function will also return a generator, which generates row values in tuples. Let's try this out:

```
1 for row in dataframe.itertuples():
2     print(row)
```

Let's run this:

```
1 Pandas(Index='id001', first_name='John', last_name='Smith', age=34)
2 Pandas(Index='id002', first_name='Jane', last_name='Doe', age=29)
3 Pandas(Index='id003', first_name='Marry', last_name='Jackson', age=37)
4 Pandas(Index='id004', first_name='Victoria', last_name='Smith', age=52)
5 Pandas(Index='id005', first_name='Gabriel', last_name='Brown', age=26)
6 Pandas(Index='id006', first_name='Layla', last_name='Martinez', age=32)
```

The `itertuples()` method has two arguments: `index` and `name`. We can choose not to display index column by setting the `index` parameter to `False`:

```
1 for row in dataframe.itertuples(index=False):
2     print(row)
```

Our tuples will no longer have the index displayed:

```
1 Pandas(first_name='John', last_name='Smith', age=34)
2 Pandas(first_name='Jane', last_name='Doe', age=29)
3 Pandas(first_name='Marry', last_name='Jackson', age=37)
4 Pandas(first_name='Victoria', last_name='Smith', age=52)
5 Pandas(first_name='Gabriel', last_name='Brown', age=26)
6 Pandas(first_name='Layla', last_name='Martinez', age=32)
```

As you've already noticed, this generator yields *namedtuples* with the default name of `Pandas`. We can change this by passing `People` argument to the `name` parameter. You can choose any name you like, but it's always best to pick names relevant to your data:

```
1 for row in dataframe.itertuples(index=False, name='People'):
2     print(row)
```

Now our output would be:

```
1 People(first_name='John', last_name='Smith', age=34)
2 People(first_name='Jane', last_name='Doe', age=29)
3 People(first_name='Marry', last_name='Jackson', age=37)
4 People(first_name='Victoria', last_name='Smith', age=52)
5 People(first_name='Gabriel', last_name='Brown', age=26)
6 People(first_name='Layla', last_name='Martinez', age=32)
```

Iteration Performance with Pandas

The [official Pandas documentation](#)¹³ warns that iteration is a slow process. If you're iterating over a `DataFrame` to modify the data, vectorization would be a quicker alternative. Also, it's discouraged to modify data while iterating over rows as Pandas sometimes returns a copy of the data in the row and not its reference, which means that not all data will actually be changed.

For small datasets you can simply print the `DataFrame` as we've done before. For larger datasets that have many columns and rows, you can use `head(n)` or `tail(n)` methods to print out the first `n` rows of your `DataFrame` (the default value for `n` is 5).

Speed Comparison

To measure the speed of each particular method, we wrapped them into functions that would execute them for 1000 times and return the average time of execution.

¹³https://pandas.pydata.org/pandas-docs/stable/user_guide/basics.html#iteration

To test these methods, we will use both of the `print()` and `list.append()` functions to provide better comparison data and to cover common use cases. In order to decide a fair winner, we will iterate over DataFrame and use only 1 value to print or append per loop.

Here's how the return values look like for each method. For example, while `items()` would cycle column by column:

```

1 ('first_name',
2    id001      John
3    id002      Jane
4    id003     Marry
5    id004   Victoria
6    id005   Gabriel
7    id006    Layla
8 Name: first_name, dtype: object)

```

`iterrows()` would provide all column data for a particular row:

```

1 ('id001',
2  first_name      John
3  last_name      Smith
4  age            34
5 Name: id001, dtype: object)

```

And finally, a single row for the `itertuples()` would look like this:

```
1 Pandas(Index='id001', first_name='John', last_name='Smith', age=34)
```

Here are the average results in seconds:

Method	Speed (s)	Test Function
items()	1.349279541666571	print()
iterrows()	3.4104003086661883	print()
itertuples()	0.41232967500279	print()
Method	Speed (s)	Test Function
items()	0.006637570998767235	append()
iterrows()	0.5749766406661365	append()
itertuples()	0.3058610513350383	append()

Printing values will take more time and resources than appending in general and our

examples are no exceptions. While `itertuples()` performs better when combined with `print()`, `items()` method outperforms others dramatically when used for `append()` and `iterrows()` remains the last for each comparison.

Please note that these test results highly depend on other factors like OS, environment, computational resources, etc. The size of your data will also have an impact on your results. This isn't an issue for small `DataFrames`, but in *Chapter 6 - Data Visualization with Matplotlib*, we'll be dealing with a huge dataset near the end of the chapter, where execution times are significant - especially if you run the code multiple times.

Merging `DataFrames`

We've already seen how Pandas provides a huge range of methods and functions to manipulate data. One of the really useful operations that can also be performed is *merging `DataFrames`*. Merging `DataFrames` allows you to both create a new `DataFrame` without modifying the original data source or alter the original data source.

If you are familiar with the SQL or a similar type of tabular data, you probably are familiar with the term `join`, which means combining multiple `DataFrames` to form a new `DataFrame`. If you are a beginner it can seem daunting and a bit complicated to fully grasp the join types (*inner*, *outer*, *left*, *right*). In this section, we'll go over join types with practical examples to help you master them.

Merging `DataFrames` isn't an uncommon task in data pre-processing. Certain types of datasets purposefully save certain data in different files, so if you want to include that data as well, and take it into consideration while plotting, say, correlations - you'll have to merge the `DataFrames`.

Our main focus would be on using the `merge()` and `concat()` functions. However, we will discuss other merging methods to give you as many practical alternatives as possible.

Here are the methods we'll be taking a look at:

- Merge `DataFrames` Using `merge()`
- Merge `DataFrames` Using `join()`

- Merge DataFrames Using *append()*
- Merge DataFrames Using *concat()*
- Merge DataFrames Using *combine_first()* and *update()*

Merge DataFrames Using *merge()*

Let's start by setting up our DataFrames, which we'll use for the rest of the section. `df1` will include our imaginary user list with names, emails, and IDs:

```
1 import pandas as pd
2
3 df1 = pd.DataFrame(
4     {'user_id': ['id001', 'id002', 'id003', 'id004', 'id005', 'id006', 'id007'],
5      'first_name': ['Rivi', 'Wynnie', 'Kristos', 'Madalyn', 'Tobe', 'Regan', 'Kristin'],
6      'last_name': ['Valti', 'McMurty', 'Ivanets', 'Max', 'Riddich', 'Huyghe', 'Illis'],
7      'email': ['rvalti0@example.com',
8                 'wmcmurty1@example.com',
9                 'kivanets2@example.com',
10                'mmax3@example.com',
11                'triddich4@example.com',
12                'rhuyghe@example.com',
13                'killis4@example.com']
14    }
15 )
```

When designing databases, it's considered good practice to keep profile settings (like background color, avatar image link, font size etc.) in a separate table from the user data (email, date added, etc). These tables can then have a one-to-one relationship.

To simulate this scenario we will do the same by creating `df2` with image URLs and user IDs:

```
1 df2 = pd.DataFrame({'user_id': ['id001', 'id002', 'id003', 'id004', 'id005'],
2                     'image_url': ['http://example.com/img/id001.png',
3                                   'http://example.com/img/id002.jpg',
4                                   'http://example.com/img/id003.bmp',
5                                   'http://example.com/img/id004.jpg',
6                                   'http://example.com/img/id005.png']}
7 )
```

Here's how our two DataFrames look like:

```

1  # df1
2    user_id first_name last_name           email
3  0    id001      Rivi     Valti  rvalti0@example.com
4  1    id002     Wynnie   McMurry  wmcmurty1@example.com
5  2    id003    Kristos   Ivanets  kivanets2@example.com
6  3    id004    Madalyn      Max   mmax3@example.com
7  4    id005      Tobe   Riddich  triddich4@example.com
8  5    id006      Regan   Huyghe  rhuyghe@example.com
9  6    id007    Kristin    Illis  killis4@example.com
10
11 #df2
12    user_id           image_url
13  0    id001  http://example.com/img/id001.png
14  1    id002  http://example.com/img/id002.jpg
15  2    id003  http://example.com/img/id003.bmp
16  3    id004  http://example.com/img/id004.jpg
17  4    id005  http://example.com/img/id005.png

```

Looks good. We've got a `user_id` column that'll be used to match the `image_urls` with our users from the other DataFrame. Let's combine these DataFrames with the `merge()` function. The `merge()` function accepts a lot of optional arguments, and is called on the Pandas instance itself:

```

1 pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
2           left_index=False, right_index=False, sort=True,
3           suffixes=('_x', '_y'), copy=True, indicator=False,
4           validate=None)

```

Most of these options have a default value except for the `left` and `right`. These two parameters are the names of the DataFrames that we want to merge. The function itself will return a new DataFrame, so it's not in-place. We can re-assign this to any of the two existing DataFrames, though, we'll store it into a new `df3_merged` frame.

Let's avoid using any other arguments for now and just supply our two DataFrames:

```
1 df3_merged = pd.merge(df1, df2)
```

Since both of our DataFrames have the column `user_id` with the same name, the `merge()` function automatically joins the two tables matching on that key. If we had two columns with different names, we could use `left_on='left_column_name'` and `right_on='right_column_name'` to specify keys on both DataFrames explicitly.

Of course, the contents of these columns would have to match - but we *can* use different names, such as `user_id` and `userId` that way.

Let's print the newly created `df3_merged`:

```

1   user_id first_name last_name           email           image_url
2 0    id001      Rivi     Valti    rvalti0@example.com http://example.com/img/id001.png
3 1    id002     Wynnie   McMurty  wmcmurty1@example.com http://example.com/img/id002.jpg
4 2    id003    Kristos   Ivanets  kivanets2@example.com http://example.com/img/id003.bmp
5 3    id004   Madalyn     Max    mmax3@example.com http://example.com/img/id004.jpg
6 4    id005      Tobe   Riddich triddich4@example.com http://example.com/img/id005.png

```

You'll notice that `df3_merged` has only 5 rows while the original `df1` had 7. Why is that?

When the default value of the `how` parameter is set to `inner`, a new DataFrame is generated from the *intersection* of the left and right DataFrames. Therefore, if a `user_id` is missing in one of the tables, the row corresponding to that `user_id` would not be present in the merged DataFrame.

This would stay true even if swapped places of the left and right rows:

```
1 df3_merged = pd.merge(df2, df1)
```

The results are still:

```

1   user_id           image_url first_name last_name           email
2 0    id001  http://example.com/img/id001.png      Rivi     Valti    rvalti0@example.com
3 1    id002  http://example.com/img/id002.jpg     Wynnie   McMurty  wmcmurty1@example.com
4 2    id003  http://example.com/img/id003.bmp    Kristos   Ivanets  kivanets2@example.com
5 3    id004  http://example.com/img/id004.jpg   Madalyn     Max    mmax3@example.com
6 4    id005  http://example.com/img/id005.png      Tobe   Riddich triddich4@example.com

```

Users with IDs '`id006`' and '`id007`' are not part of the merged DataFrames since they do not intersect on both tables. This is the so-called *inner* join:

However, there are times we want to use one of the DataFrames as the “main” DataFrame and include all the rows from it, even if they don’t all intersect with another DataFrame. We wouldn’t *merge them together*, we’d *merge one into another*, where the “main” frame sets the fundamental values.

That is to say, to have all of our users, while the `image_url` is optional.

When using `merge()`, the `how` argument lets us specify how the merge should be performed. In this case, we’ll want to do a `left` join of `df2` into `df1`:

```

1 df_left_merge = pd.merge(df1, df2, how='left')
2
3 print(df_left_merge)

```

With a *left join*, we've included all elements of the left DataFrame (df1) and every element of the right DataFrame (df2). Running the above code would display this:

	user_id	first_name	last_name	email	image_url
0	id001	Rivi	Valti	rvalti0@example.com	http://example.com/img/id001.png
1	id002	Wynnie	McMurty	wmcmurty1@example.com	http://example.com/img/id002.jpg
2	id003	Kristos	Ivanets	kivanets2@example.com	http://example.com/img/id003.bmp
3	id004	Madalyn	Max	mmax3@example.com	http://example.com/img/id004.jpg
4	id005	Tobe	Riddich	triddich4@example.com	http://example.com/img/id005.png
5	id006	Regan	Huyghe	rhuyghe@example.com	NaN
6	id007	Kristin	Illis	killis4@example.com	NaN

Cells that don't have any matching values with the left DataFrame are filled with NaN. Why don't we try a *right join*? Let's go ahead and pass 'right' to the how argument:

```

1 df_right_merge = pd.merge(df1, df2, how='right')
2
3 print(df_right_merge)

```

As you may have expected, the *right join* would return every value from the left DataFrame that matches the right DataFrame:

	user_id	first_name	last_name	email	image_url
0	id001	Rivi	Valti	rvalti0@example.com	http://example.com/img/id001.png
1	id002	Wynnie	McMurty	wmcmurty1@example.com	http://example.com/img/id002.jpg
2	id003	Kristos	Ivanets	kivanets2@example.com	http://example.com/img/id003.bmp
3	id004	Madalyn	Max	mmax3@example.com	http://example.com/img/id004.jpg
4	id005	Tobe	Riddich	triddich4@example.com	http://example.com/img/id005.png

As every row in df2 has a value in df1, this right join is similar to the inner join, in this case.

Let's have a look at outer joins. To best illustrate how they work, let's swap places of our DataFrames and create 2 new variables for both left and outer joins:

```
1 df_left = pd.merge(df2, df1, how='left', indicator=True)
2 df_outer = pd.merge(df2, df1, how='outer', indicator=True)
3
4 # Printing in Jupyter for readability
5 df_left
6 df_outer
7
8 # For non-Jupyter environments, use `print()`
9 # print(df_left)
10 # print(df_outer)
```

Keep in mind that our left DataFrame is df2 and the right DataFrame is df1. Using how='outer' merges DataFrames matching on the key *but also* includes the values that are missing or don't match.

We also added the indicator flag and set it to True so that Pandas adds an additional column _merge to the end of our DataFrame. This column tells us if a row was found in the left, right or both DataFrames.

Note: We'll switch to displaying these in a Jupyter Notebook, since they provide a nice visualization of the tables. When you have several columns, and if they have long values (like URLs), formatting tends to get a bit hectic in text-form. We'll use a Jupyter Notebook and regular Python environment interchangeably in this section.

The df_left variable looks like this:

	user_id	image_url	first_name	last_name	email	_merge
0	id001	http://example.com/img/id001.png	Rivi	Valti	rvalti0@example.com	both
1	id002	http://example.com/img/id002.jpg	Wynnie	McMurty	wmcmurty1@example.com	both
2	id003	http://example.com/img/id003.bmp	Kristos	Ivanets	kivanets2@example.com	both
3	id004	http://example.com/img/id004.jpg	Madalyn	Max	mmax3@example.com	both
4	id005	http://example.com/img/id005.png	Tobe	Riddich	triddich4@example.com	both

However, df_outer has this data:

	user_id	image_url	first_name	last_name	email	_merge	
0	id001	http://example.com/img/id001.png	Rivi	Valti	rvalti0@example.com	both	
1	id002	http://example.com/img/id002.jpg	Wynnie	McMurty	wmcmurty1@example.com	both	
2	id003	http://example.com/img/id003.bmp	Kristos	Ivanets	kivanets2@example.com	both	
3	id004	http://example.com/img/id004.jpg	Madalyn	Max	mmax3@example.com	both	
4	id005	http://example.com/img/id005.png	Tobe	Riddich	triddich4@example.com	both	
5	id006		NaN	Regan	Huyghe	rhuyghe@example.com	right_only
6	id007		NaN	Kristin	Illis	killis4@example.com	right_only

Notice that in the `df_outer` frame, `id006` and `id007` only exist in right DataFrame (in this case it's `df1`). If we would try to compare the left and outer joins without swapping the places, we would end up with the same results for both of them.

Merge DataFrames Using `join()`

Unlike `merge()` which is a method of the Pandas instance, `join()` is a method of the DataFrame itself. This means that we can use it like a static method:

```
1 DataFrame.join(other, on=None, how='left', lsuffix='', rsuffix='', sort=False)
```

The DataFrame we call `join()` from will be our *left* DataFrame. The DataFrame set to the `other` argument would be our *right* DataFrame.

The `on` parameter can take one or more (`['key1', 'key2' ...]`) arguments to define the matching key, while `how` parameter takes one of the handle arguments (`left`, `right`, `outer`, `inner`), and it's set to `left` by default.

The `join()` method, even though called on a DataFrame *isn't* in-place. It returns a new DataFrame with the performed operation. You'll want to either re-assign it back to the same DataFrame or save it into a new one.

Let's try to join `df2` to `df1`:

```

1 df_join = df1.join(df2, rsuffix='_right')
2
3 df_join
4 # print(df_join) for non-Jupyter environments and textual format

```

Like the `merge()` function, the `join()` function automatically tries to match the keys (columns) with the same name. In our case, it's the `user_id` key.

Running this code in a Jupyter Notebook would result in:

	user_id	first_name	last_name	email	user_id_right	image_url
0	id001	Rivi	Valti	rvalti0@example.com	id001	http://example.com/img/id001.png
1	id002	Wynnie	McMurty	wmcmurty1@example.com	id002	http://example.com/img/id002.jpg
2	id003	Kristos	Ivanets	kivanets2@example.com	id003	http://example.com/img/id003.bmp
3	id004	Madalyn	Max	mmax3@example.com	id004	http://example.com/img/id004.jpg
4	id005	Tobe	Riddich	triddich4@example.com	id005	http://example.com/img/id005.png
5	id006	Regan	Huyghe	rhuyghe@example.com	NaN	NaN
6	id007	Kristin	Illis	killis4@example.com	NaN	NaN

You probably noticed a “duplicate column” called `user_id_right`. If you don’t want to display that column, you can set the `user_id` columns as an index on both columns so it would join without a suffix:

```

1 df_join_no_duplicates = df1.set_index('user_id').join(df2.set_index('user_id'))
2
3 print(df_join_no_duplicates)

```

By doing so, we are getting rid of the `user_id` column and setting it as the index column instead. This provides us with a cleaner resulting DataFrame:

```

1      first_name last_name          email           image_url
2 user_id
3 id001      Rivi     Valti    rvalti0@example.com http://example.com/img/id001.png
4 id002      Wynnie   McMurry  wmcmurty1@example.com http://example.com/img/id002.jpg
5 id003      Kristos  Ivanets  kivanets2@example.com http://example.com/img/id003.bmp
6 id004      Madalyn    Max     mmax3@example.com  http://example.com/img/id004.jpg
7 id005        Tobe   Riddich  triddich4@example.com http://example.com/img/id005.png
8 id006        Regan  Huyghe   rhuyghe@example.com   NaN
9 id007      Kristin  Illis    killis4@example.com   NaN

```

Merge DataFrames Using *append()*

As the official Pandas documentation points, since `concat()` and `append()` methods return new copies of `DataFrames`, overusing these methods can affect the performance of your program.

Appending is very useful when you want to merge two `DataFrames` in row axis only. This means that instead of matching data on their columns, we want a new `DataFrame` that contains all the rows of 2 different `DataFrames`.

Let's append `df2` to `df1` and print the results:

```

1 df_append = df1.append(df2, ignore_index=True)
2
3 print(df_append)

```

Using `append()` will not match `DataFrames` on any keys. It will just add the other `DataFrame` to the first and return a copy of it. If the shapes of `DataFrames` do not match, Pandas will replace any unmatched cells with a `NaN`.

The output for appending the two `DataFrames` looks like this:

```

1      user_id first_name last_name          email           image_url
2  0      id001      Rivi     Valti    rvalti0@example.com   NaN
3  1      id002      Wynnie   McMurry  wmcmurty1@example.com   NaN
4  2      id003      Kristos  Ivanets  kivanets2@example.com   NaN
5  3      id004      Madalyn    Max     mmax3@example.com   NaN
6  4      id005        Tobe   Riddich  triddich4@example.com   NaN
7  5      id006        Regan  Huyghe   rhuyghe@example.com   NaN
8  6      id007      Kristin  Illis    killis4@example.com   NaN
9  7      id001        NaN      NaN           NaN
10 8      id002        NaN      NaN           NaN
11 9      id003        NaN      NaN           NaN
12 10     id004        NaN      NaN           NaN
13 11     id005        NaN      NaN           NaN

```

Most users choose `concat()` over the `append()` since it also provides the key matching and axis option.

Merge DataFrames Using `concat()`

Concatenation is a bit more flexible when compared to `merge()` and `join()` as it allows us to combine `DataFrames` either vertically (row-wise) or horizontally (column-wise).

The trade-off is that any data that doesn't match will be discarded. It's called on the `Pandas` instance, just like `merge()` and accepts several arguments:

```
1 pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None,
2           levels=None, names=None, verify_integrity=False, sort=False, copy=True)
```

Here are the most commonly used parameters for the `concat()` function:

- `objs` is the list of `DataFrame` objects (`[df1, df2, ...]`) to be concatenated
- `axis` defines the direction of the concatenation, `0` for row-wise and `1` for column-wise
- `join` can either be `inner` (intersection) or `outer` (union)
- `ignore_index` by default set to `False` which allows the index values to remain as they were in the original `DataFrames`, can cause duplicate index values. If set to `True`, it will ignore the original values and re-assign index values in sequential order
- `keys` allows us to construct a hierarchical index. Think of it as another level of the index that appended on the outer left of the `DataFrame` that helps us to distinguish indices when values are not unique

Let's create a new `DataFrame` with the same column types with the `df2`, but this one includes the `image_url` for `id006` and `id007`:

```
1 df2_addition = pd.DataFrame({'user_id': ['id006', 'id007'],
2                             'image_url': ['http://example.com/img/id006.png',
3                                           'http://example.com/img/id007.jpg']}
4 )
```

In order to join `df2` and `df2_addition` row-wise, we can pass them in a list as the `objs` parameter and assign the resulting `DataFrame` to a new variable:

```
1 df_row_concat = pd.concat([df2, df2_addition])
2
3 print(df_row_concat)
```

If we print this new DataFrame, we'll effectively fill in the missing values:

```
1   user_id          image_url
2  0   id001  http://example.com/img/id001.png
3  1   id002  http://example.com/img/id002.jpg
4  2   id003  http://example.com/img/id003.bmp
5  3   id004  http://example.com/img/id004.jpg
6  4   id005  http://example.com/img/id005.png
7  0   id006  http://example.com/img/id006.png
8  1   id007  http://example.com/img/id007.jpg
```

However, have a look at the indices in the left-most column. The 0 and 1 indices are repeating. To get entirely new and unique index values, we pass True to the ignore_index parameter:

```
1 df_row_concat = pd.concat([df2, df2_addition], ignore_index=True)
```

Now our df_row_concat has unique, correct index values:

```
1   user_id          image_url
2  0   id001  http://example.com/img/id001.png
3  1   id002  http://example.com/img/id002.jpg
4  2   id003  http://example.com/img/id003.bmp
5  3   id004  http://example.com/img/id004.jpg
6  4   id005  http://example.com/img/id005.png
7  5   id006  http://example.com/img/id006.png
8  6   id007  http://example.com/img/id007.jpg
```

As mentioned earlier, concatenation can work both horizontally and vertically. To join two DataFrames together column-wise, we will need to change the axis value from the default 0 to 1:

```
1 df_column_concat = pd.concat([df1, df_row_concat], axis=1)
2
3 # Displaying on Jupyter Notebook due to long values
4 df_column_concat
5 # print(df_column_concat) for non-Jupyter environments
```

You will notice that it doesn't work like merge(), matching two tables on a key:

	user_id	first_name	last_name		email	user_id	image_url
0	id001	Rivi	Valti		rvalti0@example.com	id001	http://example.com/img/id001.png
1	id002	Wynnie	McMurty	wmcmurty1@example.com		id002	http://example.com/img/id002.jpg
2	id003	Kristos	Ivanets	kivanets2@example.com		id003	http://example.com/img/id003.bmp
3	id004	Madalyn	Max	mmax3@example.com		id004	http://example.com/img/id004.jpg
4	id005	Tobe	Riddich	triddich4@example.com		id005	http://example.com/img/id005.png
5	id006	Regan	Huyghe	rhuyghe@example.com		id006	http://example.com/img/id006.png
6	id007	Kristin	Illis	killis4@example.com		id007	http://example.com/img/id007.jpg

If our right DataFrame didn't even have a `user_id` column, this concatenation still would return the same result. The `concat()` function *glues* two DataFrames together, taking each DataFrame's indices values and table shape into consideration

It doesn't do key matching like `merge()` or `join()`.

Merge DataFrames Using `combine_first()` and `update()`

In some cases, you might want to fill the missing data in your DataFrame by merging it with another DataFrame. By doing so, you will keep all the non-missing values in the first DataFrame while replacing all `NaN` values with available non-missing values from the second DataFrame (if there are any).

Let's create a DataFrame with some missing values, and a DataFrame that would nicely fit it to fill those missing values:

```

1 import pandas as pd
2
3 df_first = pd.DataFrame({'COL 1': ['X', 'X', pd.NA],
4                           'COL 2': ['X', pd.NA, 'X'],
5                           'COL 3': [pd.NA, 'X', 'X']},
6                           index=range(0, 3))
7
8 df_second = pd.DataFrame({'COL 1': [pd.NA, 'O', 'O'],
9                           'COL 2': ['O', 'O', 'O']},
10                          index=range(0, 3))
11
12 print(df_first)
13 print(df_second)
```

`df_first` has 3 columns and 1 missing value in each of them:

```

1   COL 1 COL 2 COL 3
2   0      X      X    <NA>
3   1      X    <NA>      X
4   2    <NA>      X      X

```

While `df_second` has only 2 columns and one missing value in the first column:

```

1   COL 1 COL 2
2   0    <NA>  0
3   1      0  0
4   2      0  0

```

We can use `df_second` to *patch* missing values in `df_first`:

```

1 df_tictactoe = df_first.combine_first(df_second)
2
3 print(df_tictactoe)

```

Using the `combine_first()` method will only replace `<NA>` values in the same location of another `DataFrame`. Since `df_first` has missing values in `COL 1` and `COL 2`, the missing values from `df_first` that are present in `COL 1` and `COL 2` of `df_second` will be patched:

```

1   COL 1 COL 2 COL 3
2   0      X      X    <NA>
3   1      X      0      X
4   2      0      X      X

```

On the other hand, if we wanted to overwrite the values in `df_first` with the corresponding values from `df_second` (regardless they are `<NA>` or not), we would use the `update()` method.

Let's first add a another `DataFrame` to our arsenal:

```

1 df_third = pd.DataFrame({'COL 1': ['O'], 'COL 2': ['O'], 'COL 3': ['O']})
2
3 print(df_third)

```

The shape is $(1, 3)$ - 1 row and three columns, excluding the index:

```
1   COL 1 COL 2 COL 3  
2  0     0     0     0
```

Now let's update the `df_first` with the values from `df_third`:

```
1 df_first.update(df_third)  
2 print(df_first)
```

Keep in mind that unlike `combine_first()`, `update()` does not return a new `DataFrame`. It modifies the `df_first` in-place, altering the corresponding values:

```
1   COL 1 COL 2 COL 3  
2  0     0     0     0  
3  1     X     <NA>    X  
4  2     <NA>   X     X
```

The `overwrite` parameter of the `update()` function is set to `True` by default. This is why it changes all corresponding values, instead of only `<NA>` values. We can change it to `False` to replace *only* `<NA>` values:

```
1 df_tictactoe.update(df_first, overwrite=False)  
2  
3 print(df_tictactoe)
```

Here's the final state of our `df_tictactoe`:

```
1   COL 1 COL 2 COL 3  
2  0     X     X     0  
3  1     X     0     X  
4  2     0     X     X
```

We've successfully updated the values, but by doing so, we also won the Tic-Tac-Toe match!

Handling Missing Data

So far, we've been dabbling a lot with hand-made `DataFrames`, which we can construct how we want. `DataFrames` are designed and tailor-made to hold a *metric ton* of data, and we usually don't create that data manually. We import it from files.

In the aforementioned metric ton of data, some of it is bound to be missing for various reasons - input errors, file corruption, not adhering to standards, etc. This results in a bunch of missing (`null/None/Nan`) values in our `DataFrame`.

Some datasets you can find online are clean and tidy, with little to none missing values, but this is the vast minority of them, which are popular enough to have people cleaning them up.

This is why, in this section, we'll take a look at how to handle missing data in a Pandas `DataFrame`. We'll use the simple `read_csv()` function we've used before, to load in some data. In the next three sections, we'll go into the nitty-gritty of reading and writing files with Pandas.

Data Inspection

Again, real-world datasets are rarely perfect. They may contain missing values, wrong data types, unreadable characters, erroneous lines, and other unexpected values that are just waiting to throw a wrench in your program.

The first step to any proper data analysis is cleaning and organizing the data we'll later be using. We'll be working with small employees dataset for this, which we custom-made to have several errors.

The `.csv` file looks like this:

```
1 First Name,Gender,Salary,Bonus %,Senior Management,Team  
2 Douglas,Male,97308,6.945,TRUE,Marketing  
3 Thomas,Male,61933,NaN,TRUE,  
4 Jerry,Male,NA,9.34,TRUE,Finance  
5 Dennis,n.a.,115163,10.125,FALSE,Legal  
6 ,Female,0,11.598,,Finance  
7 Angela,,,18.523,TRUE,Engineering  
8 Shawn,Male,111737,6.414,FALSE,na  
9 Rachel,Female,142032,12.599,FALSE,Business Development  
10 Linda,Female,57427,9.557,TRUE,Client Services  
11 Stephanie,Female,36844,5.574,TRUE,Business Development  
12 .....
```

Let's import it into a `DataFrame`, using the `read_csv()` method just like before:

```

1 df = pd.read_csv('out.csv')
2 # Displaying using Jupyter Notebook
3 df
4 # print(df) for non-Jupyter environments

```

This `df` now contains the data from the `.csv` file:

	First Name	Gender	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	97308.0	6.945	True	Marketing
1	Thomas	Male	61933.0	NaN	True	NaN
2	Jerry	Male	NaN	9.340	True	Finance
3	Dennis	n.a.	115163.0	10.125	False	Legal
4	NaN	Female	0.0	11.598	NaN	Finance
5	Angela	NaN	NaN	18.523	True	Engineering
6	Shawn	Male	111737.0	6.414	False	na
7	Rachel	Female	142032.0	12.599	False	Business Development
8	Linda	Female	57427.0	9.557	True	Client Services
9	Stephanie	Female	36844.0	5.574	True	Business Development
10	NaN	NaN	NaN	NaN	NaN	NaN

original dataframe pandas

Taking a closer look at the dataset, we note that Pandas automatically assigns `NaN` if the value for a particular column is an empty string `' '`, `NA` or `NaN`. However, there are cases where missing values are represented by a custom value, for example, the string `'na'` or `0` for a numeric column. These *technically* aren't missing values, as there's *something* there, but they're *functionally* missing values.

If we try using utility methods such as `dropna()`, these values won't be dropped, even though they're functionally missing. For example, the 6th row has a value of `na` for the `Team` column, while the 5th row has a value of `0` for the `Salary` column.

We'll want to first clean this up and categorize them properly, before we try to handle them as missing values.

Customizing Missing Data Values

In our dataset, we want to consider these as missing values:

1. A `0` value in the `Salary` column
2. An `na` value in the `Team` column

The easiest way to deal with missing values is to handle them at *import-time*.

This can be achieved by using the `na_values` argument of the `read_csv()` method. This argument accepts a dictionary where the keys represent a column name and the value represents the data values that are to be considered as missing.

When loading our dataset in, let's set `0` and `na` as *missing values* for the `Salary` and `Team` columns respectively:

```
1 # In the Salary column, 0 is considered a missing value.
2 # In the Team column, 'na' is considered a missing value.
3 df = pd.read_csv('out.csv', na_values={"Salary" : [0], "Team" : ['na']})
4
5 # Display using Jupyter Notebook
6 df
```

Now, when we load our data in, all instances of `0` and `na` will be turned into `NaN`, which is the correct data type for missing values:

	First Name	Gender	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	97308.0	6.945	True	Marketing
1	Thomas	Male	61933.0	NaN	True	NaN
2	Jerry	Male	NaN	9.340	True	Finance
3	Dennis	n.a.	115163.0	10.125	False	Legal
4	NaN	Female	NaN	11.598	NaN	Finance
5	Angela	NaN	NaN	18.523	True	Engineering
6	Shawn	Male	111737.0	6.414	False	NaN
7	Rachel	Female	142032.0	12.599	False	Business Development
8	Linda	Female	57427.0	9.557	True	Client Services
9	Stephanie	Female	36844.0	5.574	True	Business Development
10	NaN	NaN	NaN	NaN	NaN	NaN

customized df

Great! Now we don't have *hidden* missing values anymore. Or do we? There's a n.a. cell in the Gender column, on index 3. We'll want to add *more* matchers for missing values to solve this.

You can map a *list of values* which will be treated as missing globally, in all columns. Let's make a list of various "hidden" missing values and pass that list to the na_values argument:

```

1 missing_values = ["n.a.", "NA", "n/a", "na", 0]
2 df = pd.read_csv('out.csv', na_values = missing_values)
3 # Display using Jupyter Notebook
4 df

```

This results in:

	First Name	Gender	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	97308.0	6.945	True	Marketing
1	Thomas	Male	61933.0	NaN	True	NaN
2	Jerry	Male	NaN	9.340	True	Finance
3	Dennis	NaN	115163.0	10.125	False	Legal
4	NaN	Female	NaN	11.598	NaN	Finance
5	Angela	NaN	NaN	18.523	True	Engineering
6	Shawn	Male	111737.0	6.414	False	NaN
7	Rachel	Female	142032.0	12.599	False	Business Development
8	Linda	Female	57427.0	9.557	True	Client Services
9	Stephanie	Female	36844.0	5.574	True	Business Development
10	NaN	NaN	NaN	NaN	NaN	NaN

global missing values pandas

Now, the dataset is finally cleaned from the garbage values we had in the beginning, and we can actually work on handling missing values now. What's important is that you have a uniform/standardized form for missing values - they don't have to be NaN.

For example, some of the numeric columns in the dataset might need to treat 0 as a missing value while other columns may not. Therefore, you can use the first approach where you customize missing values based on columns.

Likewise, if we want to treat 0 as a missing value globally, we can utilize the second method and just pass an array of such values to the na_values argument. Now that we've put all missing values on the same page and identified them, let's take a look at how to handle them.

Removing Rows With Missing Values

One approach would be removing all the rows which contain missing values. This can easily be done with the dropna() function, specifically dedicated for this:

```
1 # Drops all rows with NaN values
2 df.dropna(axis=0,inplace=True)
```

This results in:

	First Name	Gender	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	97308.0	6.945	True	Marketing
7	Rachel	Female	142032.0	12.599	False	Business Development
8	Linda	Female	57427.0	9.557	True	Client Services
9	Stephanie	Female	36844.0	5.574	True	Business Development

drop nan values pandas

`inplace = True` makes all the changes in the existing `DataFrame` without returning a new one. Without it, you'd have to re-assign the `DataFrame` to itself or a new variable.

The `axis` argument specifies if you're working with rows or columns - `0` being rows, and `1` being columns.

You can control whether you want to remove the rows containing at least 1 `NaN` or all `NaN` values by setting the `how` parameter in the `dropna` method.

how :

- `any`: if *any* `NaN` values are present, drop the row
- `all`: if *all* values are `NaN`, drop the row

```
1 df.dropna(axis=0,inplace=True, how='all')
```

This would only remove the last row from the dataset since `how=all` would only drop a row if all of the values are missing from it.

Similarly, to drop columns containing missing values, just set `axis=1` in the `dropna` method.

Filling out Missing Values

It might not be the best approach to remove the rows containing missing values if such rows are abundant. They might contain valuable data in other columns and we don't want to skew the data towards an inaccurate state.

If there are 15 columns containing valuable information, and one significant or insignificant column has a missing value, it'd be a total waste to drop it. Only drop rows when you don't have a lot to lose.

There are several options for dealing with this type of issue, and the most common ones are:

- Fill NaNs with Mean, Median or Mode of the data
- Fill NaNs with a constant value
- Forward Fill or Backward Fill NaNs
- Interpolate Data and Fill NaNs

Fill Missing DataFrame Values with Column Mean, Median and Mode

Let's start out with the `fillna()` method. It fills the NaN-marked values with values you supply the method with. For example, you can use the `.median()`, `.mode()` and `.mean()` functions on a column, and supply those as the fill value:

```
1 # Using median
2 df['Salary'].fillna(df['Salary'].median(), inplace=True)
3
4 # Using mean
5 df['Salary'].fillna(int(df['Salary'].mean())), inplace=True)
6
7 # Using mode
8 df['Salary'].fillna(int(df['Salary'].mode())), inplace=True)
```

Now, if a salary is missing from a person's row, a mean, mode or median are used to fill that value. This way, you're not dropping these people from the dataset, and you're also not skewing the salary values by much.

While not perfect, this method allows you to introduce values that don't impact the overall dataset, since no matter how many averages you add, the average stays the same. However, you can mess up the distribution of data if you add many averages.

Fill Missing DataFrame Values with a Constant

You could also decide to fill the NaN-marked values with a constant value. For example, you can put in a special string or numerical value:

```
1 df['Salary'].fillna(0, inplace=True)
```

While this may seem like we're going back to square one, we aren't. Square one was having unstandardized missing values, and this is functionally equivalent to having NaNs. The difference is, you can use values like 0 and perform operations on them, that you can't perform on NaN.

Forward Fill Missing DataFrame Values

This method would fill the missing values with first non-missing value that occurs before it:

```
1 df['Salary'].fillna(method='ffill', inplace=True)
```

For example, Jerry would inherit the Salary from Thomas:

```
1   First Name Gender    Salary  Bonus % Senior Management      Team
2 ...
3   1     Thomas   Male  61933.0      NaN           True      NaN
4   2     Jerry   Male  61933.0    9.34           True  Finance
```

Backward Fill Missing DataFrame Values

This method would fill the missing values with first non-missing value that occurs after it:

```
1 df['Team'].fillna(method='bfill', inplace=True)
```

This works exactly the other way around from the previous approach, and Thomas will inherit Finance as his Team from Jerry, because it's the first non-missing value occurring after his:

```

1   First Name Gender    Salary  Bonus % Senior Management      Team
2   ...
3   1     Thomas   Male  61933.0       NaN            True  Finance
4   2     Jerry   Male      NaN     9.34            True  Finance

```

Fill Missing DataFrame Values with Interpolation

Finally, this method uses mathematical interpolation to determine what value would have been in the place of a missing value. The `interpolate()` function can be used to achieve this, and for the `polynomial` and `spline` methods, you'll also have to specify the `order` of that method:

```
1 df['Salary'].interpolate(method='polynomial', order=5, inplace=True)
```

	First Name	Gender	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	97308.000000	6.945	True	Marketing
1	Thomas	Male	61933.000000	NaN	True	NaN
2	Jerry	Male	108558.907129	9.340	True	Finance
3	Dennis	NaN	115163.000000	10.125	False	Legal
4	NaN	Female	85046.355671	11.598	NaN	Finance
5	Angela	NaN	73044.323666	18.523	True	Engineering
6	Shawn	Male	111737.000000	6.414	False	NaN
7	Rachel	Female	142032.000000	12.599	False	Business Development
8	Linda	Female	57427.000000	9.557	True	Client Services
9	Stephanie	Female	36844.000000	5.574	True	Business Development
10	NaN	NaN	NaN	NaN	NaN	NaN

```
1 df['Salary'].interpolate(method='spline', order=5, inplace=True)
```

	First Name	Gender	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	97308.000000	6.945	True	Marketing
1	Thomas	Male	61933.000000	NaN	True	NaN
2	Jerry	Male	108557.233181	9.340	True	Finance
3	Dennis	NaN	115163.000000	10.125	False	Legal
4	NaN	Female	85049.266288	11.598	NaN	Finance
5	Angela	NaN	73048.433135	18.523	True	Engineering
6	Shawn	Male	111737.000000	6.414	False	NaN
7	Rachel	Female	142032.000000	12.599	False	Business Development
8	Linda	Female	57427.000000	9.557	True	Client Services
9	Stephanie	Female	36844.000000	5.574	True	Business Development
10	NaN	NaN	991081.025048	NaN	NaN	NaN

```
1 df['Salary'].interpolate(method='linear', inplace=True)
```

	First Name	Gender	Salary	Bonus %	Senior Management	Team
0	Douglas	Male	97308.0	6.945	True	Marketing
1	Thomas	Male	61933.0	NaN	True	NaN
2	Jerry	Male	88548.0	9.340	True	Finance
3	Dennis	NaN	115163.0	10.125	False	Legal
4	NaN	Female	114021.0	11.598	NaN	Finance
5	Angela	NaN	112879.0	18.523	True	Engineering
6	Shawn	Male	111737.0	6.414	False	NaN
7	Rachel	Female	142032.0	12.599	False	Business Development
8	Linda	Female	57427.0	9.557	True	Client Services
9	Stephanie	Female	36844.0	5.574	True	Business Development
10	NaN	NaN	36844.0	NaN	NaN	NaN

Reading and Writing CSV Files

While you can read and write CSV files in Python using the built-in `open()` function, or the dedicated `csv14` module - you can also use Pandas.

In this section, we'll take a look at how to read and write CSV files with Pandas. We've briefly covered this functionality a bit earlier in this chapter, while being introduced to `DataFrames`, but now, we can explore this functionality in a bit more detail.

What is a CSV File?

Let's quickly recap what a CSV file is - nothing more than a simple text file, following a few formatting conventions. However, it is the most common, simple, and easiest method to store tabular data. This format arranges tables by following a specific structure divided into rows and columns. It is these rows and columns that contain your data.

A new line terminates each row to start the next row. Similarly, a delimiter, usually a comma, separates columns within each row. For example, we might have a table that looks like this:

1	City	State	Capital	Population	1
2	Philadelphia	Pennsylvania	No	1.581 Million	2
3	Sacramento	California	Yes	0.5 Million	3
4	New York	New York	No	8.623 Million	4
5	Austin	Texas	Yes	0.95 Million	5
6	Miami	Florida	No	0.463 Million	6
7					7

If we were to convert it into the CSV format, it'd look like this:

¹⁴<https://docs.python.org/3/library/csv.html>

```
1 City,State,Capital,Population
2 Philadelphia,Pennsylvania,No,1.581 Million
3 Sacramento,California,Yes,0.5 Million
4 New York,New York,No,8.623 Million
5 Austin,Texas,Yes,0.95 Million
6 Miami,Florida,No,0.463 Million
```

Although the name (Comma-Separated Values) inherently uses a comma as the delimiter, you can use other delimiters (separators) as well, such as the semicolon (;). Each row of the table is a new line of the CSV file and it's a very compact and concise way to represent tabular data.

Now, let's take a look at the `read_csv()` function we've used before with a bit more detail.

Reading CSV Files with `read_csv()`

Let's import the Titanic Dataset, which can be obtained on [GitHub](#)¹⁵:

```
1 import pandas as pd
2 titanic_data = pd.read_csv('titanic.csv')
```

Pandas will search for this file in the directory of the script, naturally, and we just supply the filepath to the file we'd like to parse as the one and only required argument of this method.

Let's take a look at the `head()` of this dataset to make sure it's imported correctly:

```
1 titanic_data.head()
```

This results in:

¹⁵<https://raw.githubusercontent.com/datasets/master/titanic.csv>

```
1   PassengerId  Survived  Pclass  ...      Fare Cabin Embarked
2  0            1        0     3  ...    7.2500   NaN      S
3  1            2        1     1  ...  71.2833  C85      C
4  2            3        1     3  ...    7.9250   NaN      S
5  3            4        1     1  ...  53.1000  C123      S
6  4            5        0     3  ...    8.0500   NaN      S
```

DataFrames are inherently tabular, and the process of converting tabular data into a DataFrame is both intuitive, natural and simple.

Alternatively, you can also read CSV files from online resources, such as GitHub, simply by passing in the URL of the resource to the `read_csv()` function. Let's read this same CSV file from the GitHub repository, without downloading it to our local machine first:

```
1 import pandas as pd
2
3 titanic_data = pd.read_csv(r'https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv')
4
5 print(titanic_data.head())
```

This also results in:

```
1   PassengerId  Survived  Pclass  ...      Fare Cabin Embarked
2  0            1        0     3  ...    7.2500   NaN      S
3  1            2        1     1  ...  71.2833  C85      C
4  2            3        1     3  ...    7.9250   NaN      S
5  3            4        1     1  ...  53.1000  C123      S
6  4            5        0     3  ...    8.0500   NaN      S
```

Customizing Headers

By default, the `read_csv()` method uses the first row of the CSV file as the column headers. Sometimes, these headers might have odd names, and you might want to use your own headers. You can set headers either after reading the file, simply by assigning the `columns` field of the DataFrame instance another list, or you can set the headers while reading the CSV in the first place.

Let's define a list of column names, and use those names instead of the ones from the CSV file:

```
1 import pandas as pd
2
3 col_names = ['Id',
4              'Survived',
5              'Passenger Class',
6              'Full Name',
7              'Gender',
8              'Age',
9              'SibSp',
10             'Parch',
11             'Ticket Number',
12             'Price', 'Cabin',
13             'Station']
14
15 titanic_data = pd.read_csv(r'titanic.csv', names=col_names)
16 print(titanic_data.head())
```

Let's run this code:

```
1      Id  Survived Passenger Class ...    Price  Cabin  Station
2  0  PassengerId  Survived          Pclass ...     Fare  Cabin  Embarked
3  1            1         0           3   ...    7.25   NaN     S
4  2            2         1           1   ...  71.2833   C85     C
5  3            3         1           3   ...    7.925   NaN     S
6  4            4         1           1   ...   53.1    C123     S
```

Hmm, now we've got our custom headers, but the *first* row of the CSV file, which was originally used to set the column names is also included in the DataFrame. We'll want to skip this line, since it no longer holds any value for us.

Skiping Rows While Reading CSV

Let's address this issue by using the `skiprows` argument:

```
1 import pandas as pd
2
3 col_names = ['Id',
4              'Survived',
5              'Passenger Class',
6              'Full Name',
7              'Gender',
8              'Age',
9              'SibSp',
10             'Parch',
11             'Ticket Number',
12             'Price', 'Cabin',
13             'Station']
14
15 titanic_data = pd.read_csv(r'titanic.csv', names=col_names, skiprows=[0])
16 print(titanic_data.head())
```

Now, let's run this code:

```
1      Id  Survived  Passenger Class ...    Price Cabin Station
2  0      1         0                 3   ...  7.2500  NaN     S
3  1      2         1                 1   ... 71.2833  C85     C
4  2      3         1                 3   ...  7.9250  NaN     S
5  3      4         1                 1   ... 53.1000  C123     S
6  4      5         0                 3   ...  8.0500  NaN     S
```

Works like a charm! The `skiprows` argument accepts a list of rows you'd like to skip. You can skip, for example, `0, 4, 7` if you'd like as well:

```
1 titanic_data = pd.read_csv(r'titanic.csv', names=col_names, skiprows=[0, 4, 7])
2 print(titanic_data.head(10))
```

This would result in a DataFrame that doesn't have some of the rows we've seen before:

```
1      Id  Survived  Passenger Class ...    Price Cabin Station
2  0      1         0                 3   ...  7.2500  NaN     S
3  1      2         1                 1   ... 71.2833  C85     C
4  2      3         1                 3   ...  7.9250  NaN     S
5  3      5         0                 3   ...  8.0500  NaN     S
6  4      6         0                 3   ...  8.4583  NaN     Q
7  5      8         0                 3   ... 21.0750  NaN     S
8  6      9         1                 3   ... 11.1333  NaN     S
9  7     10        1                 2   ... 30.0708  NaN     C
10 8     11        1                 3   ... 16.7000  G6      S
11 9     12        1                 1   ... 26.5500  C103     S
```

Keep in mind that skipping rows happens *before* the DataFrame is fully formed, so you won't be missing any indices of the DataFrame itself, though, in this case, you can see that the `Id` field (imported from the CSV file) is missing IDs 4 and 7.

Removing Headers

You can also decide to remove the header completely, which would result in a DataFrame that simply has `0...n` header columns, by setting the `header` argument to `None`:

```
1 titanic_data = pd.read_csv(r'titanic.csv', header=None, skiprows=[0])
```

You'll also want to skip the first row here, since if you don't, the values from the first row will be actually be included in the first row:

```

1      0   1   2           3   4   ... 7   \
2          8   9
3  0   1   0   3       Braund, Mr. Owen Harris   male   ...   0   \
4  A/5 21171  7.2500
5  1   2   1   1 Cumings, Mrs. John Bradley (Florence Briggs Th... female   ...   0   \
6  PC 17599  71.2833
7  2   3   1   3       Heikkinen, Miss. Laina  female   ...   0   STON/O\
8  2. 3101282  7.9250
9  3   4   1   1  Futrelle, Mrs. Jacques Heath (Lily May Peel) female   ...   0   \
10 113803   53.1000
11 4   5   0   3       Allen, Mr. William Henry   male   ...   0   \
12 373450   8.0500

```

Specifying Delimiters

As stated earlier, you'll eventually probably encounter a CSV file that doesn't actually use commas to separate data. In such cases, you can use the `sep` argument to specify other delimiters:

```
1 titanic_data = pd.read_csv(r'titanic.csv', sep=';')
```

Writing CSV Files with `to_csv()`

Again, DataFrames are tabular. Turning a DataFrame into a CSV file is as simple as turning a CSV file into a DataFrame - we call the `write_csv()` function on the DataFrame instance.

When writing a DataFrame to a CSV file, you can also change the column names, using the `columns` argument, or specify a delimiter via the `sep` argument. If you don't specify either of these, you'll end up with a standard Comma-Separated Value file.

Let's play around with this:

```

1 import pandas as pd
2 cities = pd.DataFrame([['Sacramento', 'California'], ['Miami', 'Florida']], columns=['City',
3 'State'])
4 cities.to_csv('cities.csv')

```

Here, we've made a simple DataFrame with two cities and their respective states. Then, we've gone ahead and saved that data into a CSV file using `to_csv()` and providing the filename.

This results in a new file in the working directory of the script you're running, which contains:

```
1 ,City,State
2 0,Sacramento,California
3 1,Miami,Florida
```

Though, this isn't really well-formatted. We've still got the indices from the DataFrame, which also puts a weird missing spot before the column names. If we re-imported this CSV back into a DataFrame, it'd be a mess:

```
1 df = pd.read_csv('cities.csv')
2 print(df)
```

This results in:

```
1      Unnamed: 0      City      State
2 0      Sacramento  California
3 1          Miami    Florida
```

The indices from the DataFrame ended up becoming a new column, which is now Unnamed. When saving the file, let's make sure to *drop* the index of the DataFrame, since it's not really a part of the data - it's a part of the DataFrame:

```
1 import pandas as pd
2 cities = pd.DataFrame([['Sacramento', 'California'], ['Miami', 'Florida']], columns=['City', 'State'])
3 cities.to_csv('cities.csv', index=False)
```

Now, this results in a file that contains:

```
1 City,State
2 Sacramento,California
3 Miami,Florida
```

Works like a charm! If we re-import it and print the contents, the DataFrame is constructed well:

```
1 df = pd.read_csv('cities.csv')
2 print(df)
```

This results in:

```
1      City      State
2  0  Sacramento  California
3  1        Miami    Florida
```

Customizing Headers

Let's change the column headers from the default ones:

```
1 import pandas as pd
2 cities = pd.DataFrame([['Sacramento', 'California'], ['Miami', 'Florida']], columns=['City', 'State'])
3 new_column_names = ['City_Name', 'State_Name']
4 cities.to_csv('cities.csv', index=False, header=new_column_names)
```

We've made a new_header list, that contains different values for our columns. Then, using the header argument, we've set these instead of the original column names. This generates a cities.csv with these contents:

```
1 City_Name,State_Name
2 Sacramento,California
3 Miami,Florida
4 Washington DC,Unknown
```

Customizing Delimiter

Let's change the delimiter from the default (,) value to a new one:

```
1 import pandas as pd
2 cities = pd.DataFrame([['Sacramento', 'California'], ['Miami', 'Florida']], columns=['City', 'State'])
3 cities.to_csv('cities.csv', index=False, sep=';')
```

This results in a cities.csv file that contains:

```
1 City;State
2 Sacramento;California
3 Miami;Florida
```

Handling Missing Values

Sometimes, DataFrames have missing values that we've left as NaN or NA. In such cases, you might want to format these when you write them out into a CSV file. You can use the na_rep argument and set the value to be put instead of a missing value:

```
1 import pandas as pd
2 cities = pd.DataFrame([['Sacramento', 'California'], ['Miami', 'Florida'], ['Washington D\
3 C', pd.NA]],
4 columns=['City', 'State'])
5 cities.to_csv('cities.csv', index=False, na_rep='Unknown')
```

Here, we've got two valid city-state pairs, but Washington DC is missing its state. If we run this code, it'll result in a `cities.csv` with the following contents:

```
1 City,State
2 Sacramento,California
3 Miami,Florida
4 Washington DC,Unknown
```

Chapter 4. - Getting Started with Matplotlib

Now that we've covered everything you need to know about Pandas, its data structures, how to manipulate create and export them from and to various data types, as well as gotten a good view of Pandas' own data visualization capabilities, it's finally time to jump into the famed Matplotlib library.

What is Matplotlib?

Matplotlib is the de-facto most popular visualization engine. Note the usage of “visualization engine” here.

Matplotlib isn't just a standalone library for itself - it carries much more on its shoulders. Other libraries, such as Pandas and Seaborn rely on Matplotlib to perform the actual visualizations. Seaborn can construct and create beautiful plots, but ultimately relies on Matplotlib to actually visualize it.

GeoPandas is another library, specialized for creating, manipulating and visualizing geospatial data, based on Pandas, and thus, Matplotlib.

Originally, Matplotlib was released back in 2003, and has seen worldwide adoption since, with regular updates to this day. Year-over-year, it's cemented itself as one of the key and core libraries for visualization, and isn't likely to be dethroned soon, given how deeply engrained it is with other extremely popular libraries, alongside its own popularity.

During this time, the team behind Matplotlib, including the community, has expanded it to include a plethora of visualization tools and plot types - from simple static 2D plots to more advanced, animated 3D plots, widgets and event-handling. It even offers support for integration with the popular PyQt and TKinter frameworks, used to create GUI applications in Python, allowing developers to integrate powerful visualizations in their applications.

Although it has support for such a wide variety of tools and plots, it's not the most friendly for some - such as 3D plots, interactivity and animated plots. In general, there are libraries, such as Bokeh, Plotly and VisPy that are more suited for these tasks. They are, by no means, required to plot 3D and animated plots, though, the results usually look better and are easier to achieve.

One of the key features of Matplotlib is the fact that it offers both a high-level API, that lets you easily plot most things without batting an eye, but *also* offers a granular, fully customizable low-level API if you'd like to tweak anything. Practically *everything* is customizable when using Matplotlib. Certain libraries, like Seaborn, offer an even higher-level API, which makes it *even easier* to plot, though, when you want to customize Seaborn plots, you'll use the low-level Matplotlib configurations.

Matplotlib supports two expression styles - **MATLAB-style** and **Object-Oriented-style**.

The PyPlot Interface

Let's first take the MATLAB-style into consideration. Matplotlib was inspired by MATLAB, to a degree, and it was built to bring that functionality to Python. The collection of methods that were made to achieve this are brought together in the *PyPlot* interface. This is what we import through:

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

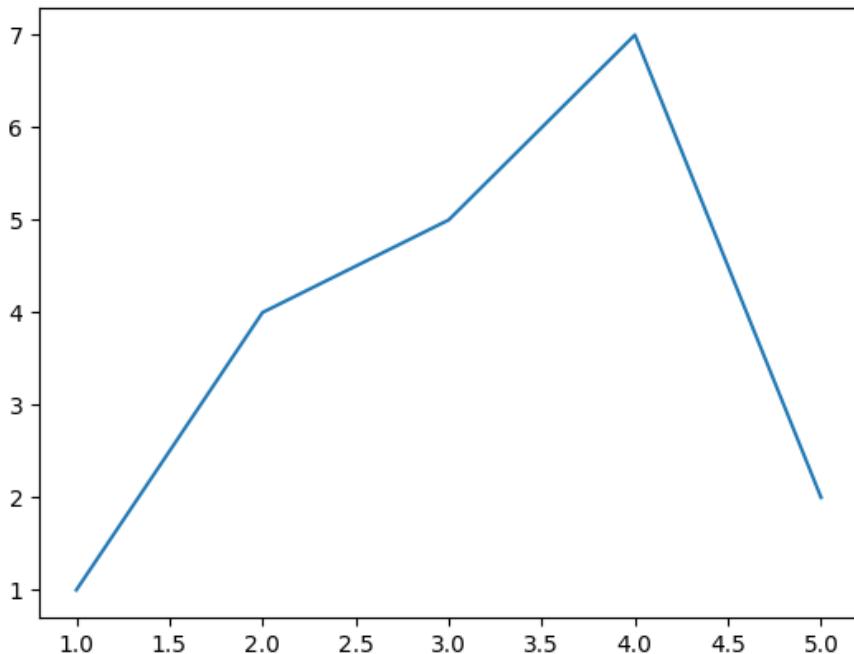
In essence, PyPlot is the interface of Matplotlib. The functions we call from it, such as `plt.plot()`, which is the general-purpose plotting function, all alter a figure, and thus change its *state*. This state is saved and carried across function calls so calling multiple functions will essentially *build on top* of the state left from the previous function.

Calling `plt.plot()` multiple times will plot multiple plots on top of each other, after which you can `plt.show()` them. The general-purpose `plot()` function, of the PyPlot interface, can be used to construct Scatter Plots, or Line Plots, even though you can use the dedicated `scatter()` function, for example. It plots the `y` feature versus the `x` features, and depending on their data you provide, it'll construct either a line plot or a scatter plot.

Let's go ahead and use the MATLAB-style to construct a simple Line Plot:

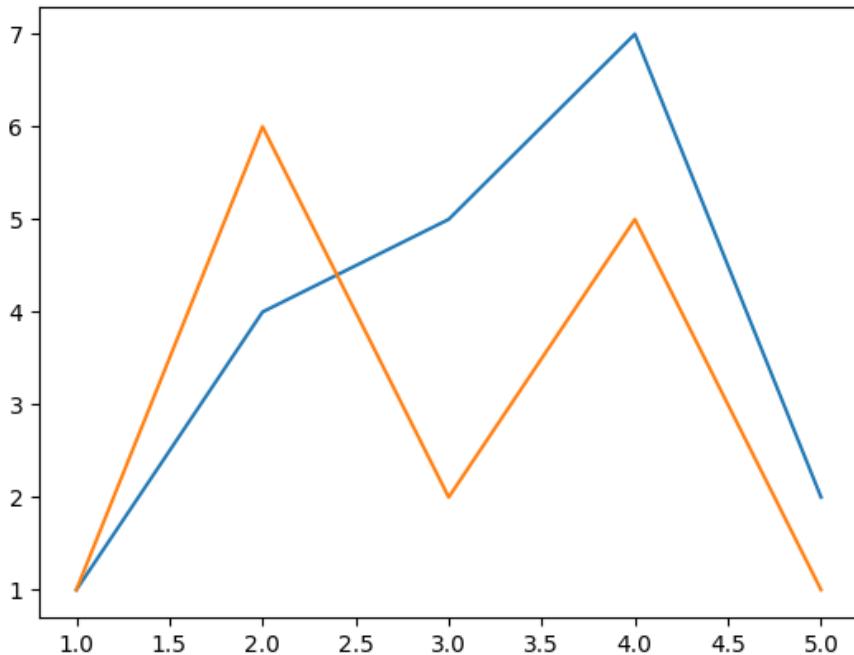
```
1 import matplotlib.pyplot as plt
2
3 x = [1, 2, 3, 4, 5]
4 y = [1, 4, 5, 7, 2]
5 plt.plot(x, y)
6 plt.show()
```

This will open up a window with a line plot:



Now, let's call the `plt.plot()` function again, this time, on the same X-axis, but setting new values on the Y-axis:

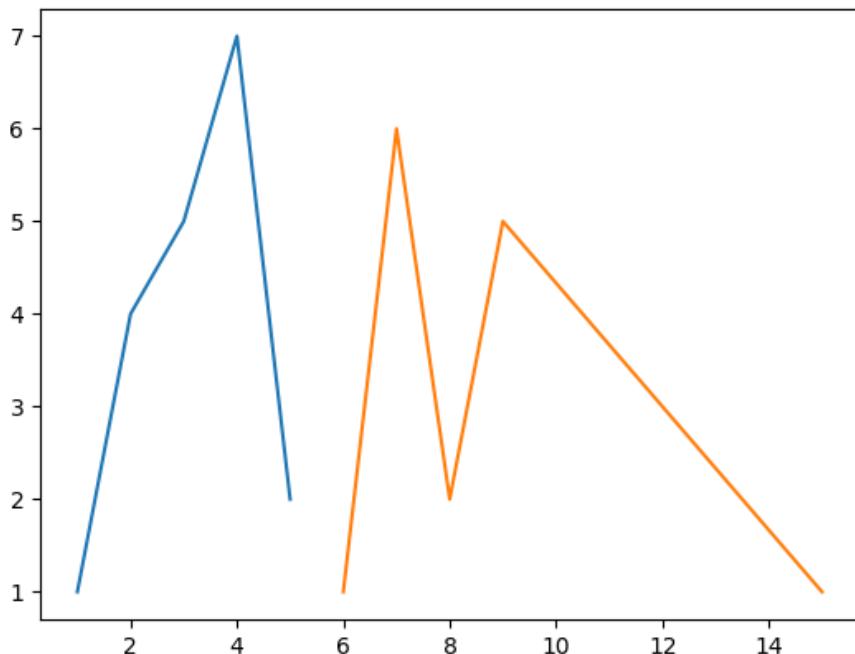
```
1 import matplotlib.pyplot as plt
2
3 x = [1, 2, 3, 4, 5]
4 y = [1, 4, 5, 7, 2]
5 z = [1, 6, 2, 5, 1]
6 plt.plot(x, y)
7 plt.plot(x, z)
8 plt.show()
```



We can also go ahead and plot two different lines, on the same figure, with totally different X-values:

```
1 import matplotlib.pyplot as plt
2
3 x_1 = [1, 2, 3, 4, 5]
4 y_1 = [1, 4, 5, 7, 2]
5 x_2 = [6, 7, 8, 9, 15]
6 y_2 = [1, 6, 2, 5, 1]
7
8 plt.plot(x_1, y_1)
9 plt.plot(x_2, y_2)
10 plt.show()
```

Running this code results in:



In fact, we can even make the X-axis of one line numerical, while the X-axis of the other line is categorical. We'll need to make sure they're the same *shape* though (contain the same number of elements):

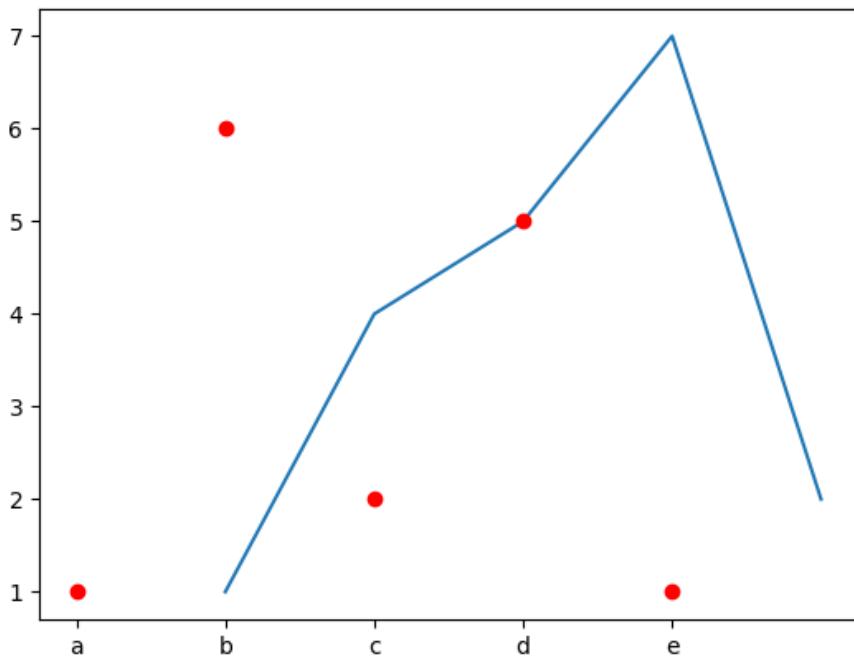
```
1 import matplotlib.pyplot as plt
2
3 x_1 = [1, 2, 3, 4, 5]
4 y_1 = [1, 4, 5, 7, 2]
5
6 x_2 = ['a', 'b', 'c', 'd', 'e']
7 y_2 = [1, 6, 2, 5, 1]
8 plt.plot(x_1, y_1)
9 plt.plot(x_2, y_2, 'ro')
10 plt.show()
```

Since the categorical values now clash with the numerical ones, the ones plotted the latest prevail. In our case, it means that our spine will have categorical values of `x_2`.

We've also used '`ro`' in the second `plot()` call, which is an optional argument and indicates the type of line you want to use. This is another inspiration from MATLAB, where '`ro`' represents red circles:

- `r` - Color (`r` is red, `g` is green, `b` is blue...)
- `o` - Shape (`o` is circle, `-` is line...)

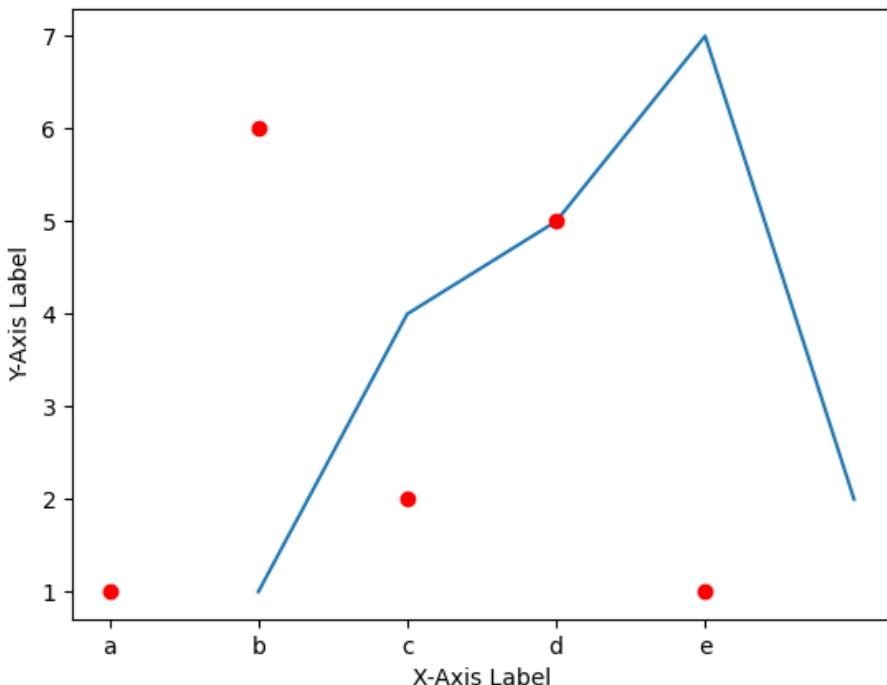
The default blue line we see maps to the `b-` argument, while we've also seen `o-` (orange line) in the previous plot. Knowing this, we can expect red circles on the `a`, `b`, `c`, `d` and `e` ticks, at the `1`, `6`, `2`, `5`, and `1` Y-values:



Using the PyPlot instance, we can also access and customize various aspects of the plots, such as the labels:

```
1 import matplotlib.pyplot as plt
2
3 x_1 = [1, 2, 3, 4, 5]
4 y_1 = [1, 4, 5, 7, 2]
5
6 x_2 = ['a', 'b', 'c', 'd', 'e']
7 y_2 = [1, 6, 2, 5, 1]
8 plt.plot(x_1, y_1)
9 plt.plot(x_2, y_2, 'ro')
10 plt.ylabel('Y-Axis Label')
11 plt.xlabel('X-Axis Label')
12 plt.show()
```

This is the same plot as before, but we've also added a label to both the X-axis and Y-axis:



Note: We'll go into detail on how labels and text work in figures, and what tools you have at disposal to work with them in the next chapter.

Now, it's becoming more evident how plotting and working with Matplotlib in the MATLAB-style looks like - creating/importing your data, and calling various `plt` functions that alter the state of the figure.

This is amazing for quick, simple plots, where you can write literally a few lines of code and have a working visualization, letting Matplotlib auto-configure the elements of the plot.

Although you can access every single aspect of Matplotlib through the `plt` instance, perform any customization, and visualize any plot - the functional style of this approach might be counter-intuitive for practitioners who haven't used MATLAB before, or are used to a more object-oriented format.

This approach also abstracts the inner workings of Matplotlib, since you don't bother

yourself with what the `Figure` object is, how the `Axes` is being filled up, etc.

On the other hand, if you're looking to directly work with these objects, and if you're more familiar with the concepts of Object-Oriented programming, you'll be delighted to know that you can fully switch to the OO paradigm with Matplotlib.

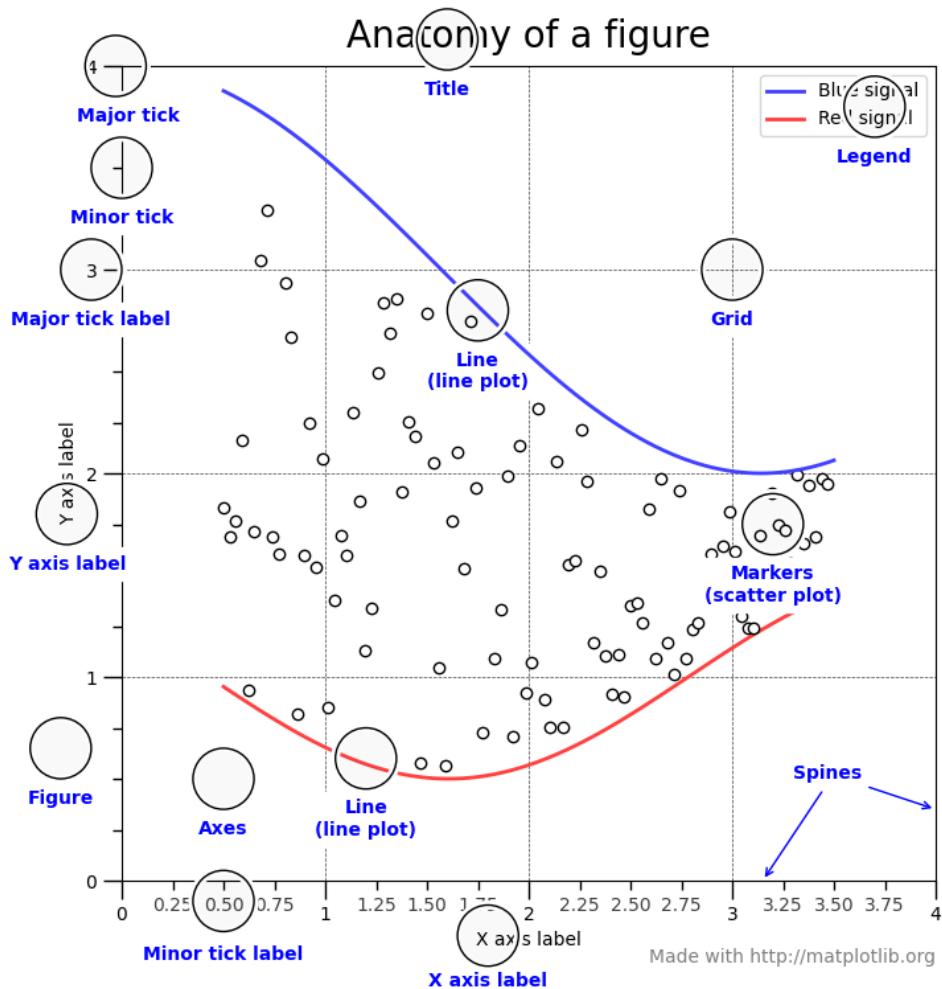
Though, switching over to it will require you to know the *anatomy of Matplotlib plots*, and how it works under the hood of the `plt` instance. Let's take a look at that first.

Anatomy of Matplotlib Plots

Let's take a look at the constituent parts of a typical Matplotlib plot:

- **Figure** - The figure that contains *everything* that we'll be seeing within it. Each `Figure` object can have one or more `Axes` objects.
- **Axes** - Although the name `Axes` implies the actual axes of the plot, the `Axes` object can practically be seen as *the plot itself*. An `Axes` sits snug in the `Figure` and contains elements such as `Titles`, `Legends`, `Grids`, etc. Since a `Figure` can have multiple `Axes` objects, each would actually be a plot for itself. Keep in mind that in the previous example, where we've used `plot()` two times, we haven't created multiple `Axes` objects. Both of those lines were plotted on the same `Axes` object, as `plot()` doesn't *create a plot*, it *plots*.
- **Title** - The title of the `Axes` object.
- **Legend** - The legend of the `Axes` object.
- **Ticks** - Sub-divided into *major ticks* and *minor ticks*. These are the ticks on the X-axis and Y-axis we've seen in the examples above.
- **Labels** - Labels can be set for the X and Y-axis, or for ticks. They're used to, well, label certain elements of the plot for a finer user experience.
- **Grids** - Optional lines in the background of the plot, that help the interpreter to distinguish between similar X and Y values, based on the frequency of grid lines.
- **Lines/Markers** - The actual lines/markers that are used to express records/data of a plot. Most of the time, you'll use lines to plot continuous data, while you'll use markers for discrete data.

The Matplotlib team went through the effort of creating a *wonderful* visualization of these elements, and marked them:



Plot Anatomy, created in Matplotlib by the Matplotlib team at matplotlib.org

Credit: [Matplotlib.org](http://matplotlib.org)

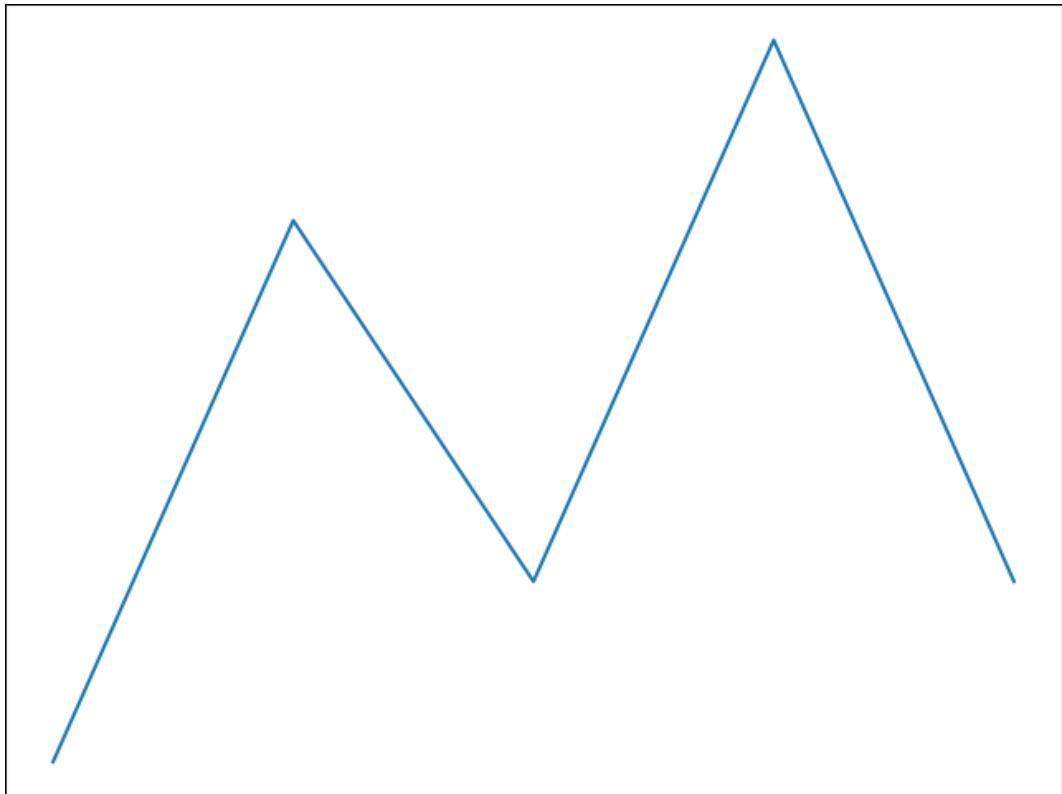
Object-Oriented Plotting

Now, keeping the constituent elements of a Matplotlib plot in mind, we're able to work with these elements in an Object-Oriented fashion. Let's import the PyPlot interface and access some of the objects we'll be using frequently:

```
1 import matplotlib.pyplot as plt
2
3 figure = plt.figure()
4 ax = figure.add_axes([0, 0, 1, 1])
5
6 x = [1, 2, 3, 4, 5]
7 y = [1, 7, 3, 9, 3]
8
9 ax.plot(x, y)
10 plt.show()
```

Here, we've instantiated a `Figure` object via the `plt.figure()` call, and saved a reference to it as `figure`. Then, we've added an `Axes` object to the `Figure`, on the `[0, 0, 1, 1]` position. Specifically, these are the `left`, `bottom`, `width` and `height` of the `Axes` object. The numbers are fractions of the figure the `Axes` object belongs to, so we've specified it to start at the bottom-left point (`0` for `left` and `0` for `bottom`) and to have the same height and width of the parent figure (`1` for `width` and `1` for `height`).

Let's take a look at the plot generated by this code:



It's bare-bones. We don't have any spines, ticks, or labels here, like we had before, when Matplotlib auto-configured values for us.

In general, you probably won't use the `add_axes()` function all too commonly. You'll only use it if you want to specifically add bare-bones `Axes` instances to a `Figure` while making your own layout, and customize how they look like. A more popular option is the `add_subplot()` function that adds a subplot (`Axes`), or even the `subplots()` function that can be used to add multiple subplots at once.

Let's change this function out with the `add_subplot()` function, so we don't have to manually add the spines and ticks:

```
1 import matplotlib.pyplot as plt
2
3 figure = plt.figure()
4 ax = figure.add_subplot(111)
5
6 x = [1, 2, 3, 4, 5]
7 y = [1, 7, 3, 9, 3]
8
9 ax.plot(x, y)
10 plt.show()
```

Now, we've got another possibly weird argument here - `111`. When adding Axes as subplots like this, they're part of a *subplot grid*. When using `add_axes()`, you can put two Axes one on top of each other, since you manually specify the absolute position. Here, we're specifying in which position on the grid we want our Axes to be added to.

Additionally, when calling this method, we *construct* the grid. So we don't only supply the index of the Axes we're adding - we're also letting Matplotlib know how many columns and rows the grid will have.

In other words, the `111` can be broken down into:

- 1 - Grid has one row
- 1 - Grid has one column
- 1 - Index of the subplot we're adding

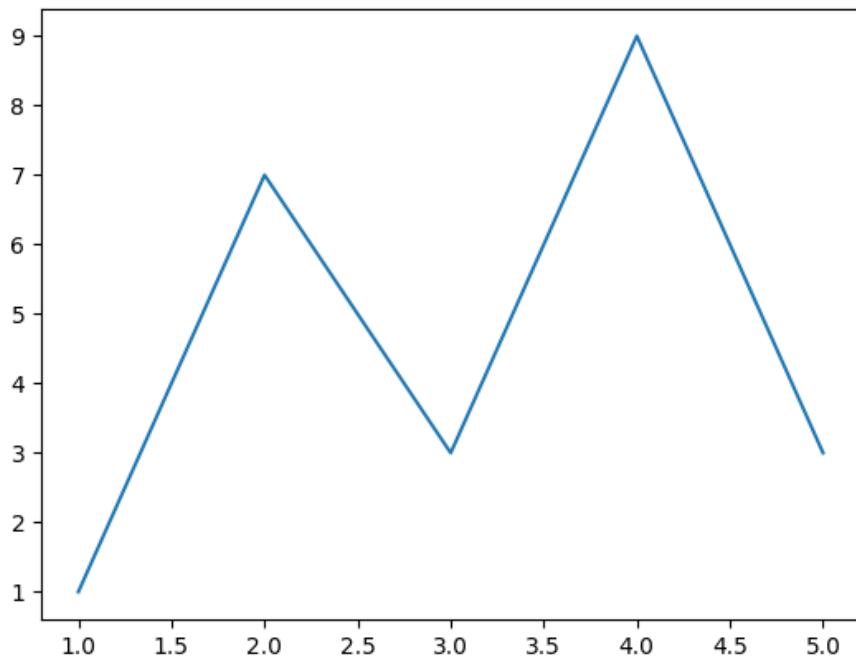
These can also be added as separate integers:

```
1 ax = figure.add_subplot(1, 1, 1)
```

Which allows us to insert a counter during iteration, if we're adding multiple subplots. Alternatively, you can also use keywords until you get used to the positional arguments:

```
1 ax = figure.add_subplot(nrows=1, ncols=1, index=1)
```

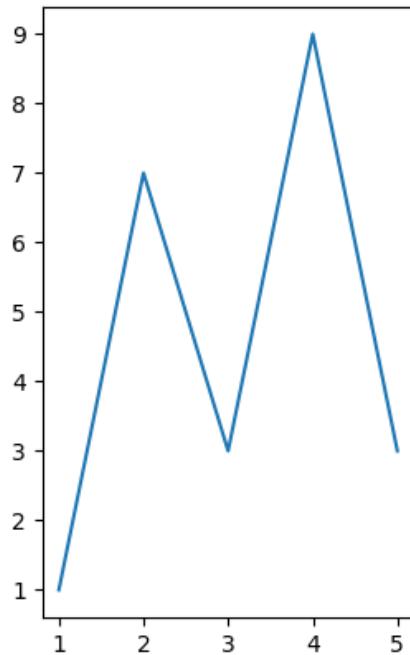
Following the same logic, Axes added to `121` and `122` would be in the same row, but in different columns - one next to the other. Let's see what this code produces:



Looks much better! And now let's set the position to 122 instead:

```
1 import matplotlib.pyplot as plt
2
3 figure = plt.figure()
4 ax = figure.add_subplot(122)
5
6 x = [1, 2, 3, 4, 5]
7 y = [1, 7, 3, 9, 3]
8
9 ax.plot(x, y)
10 plt.show()
```

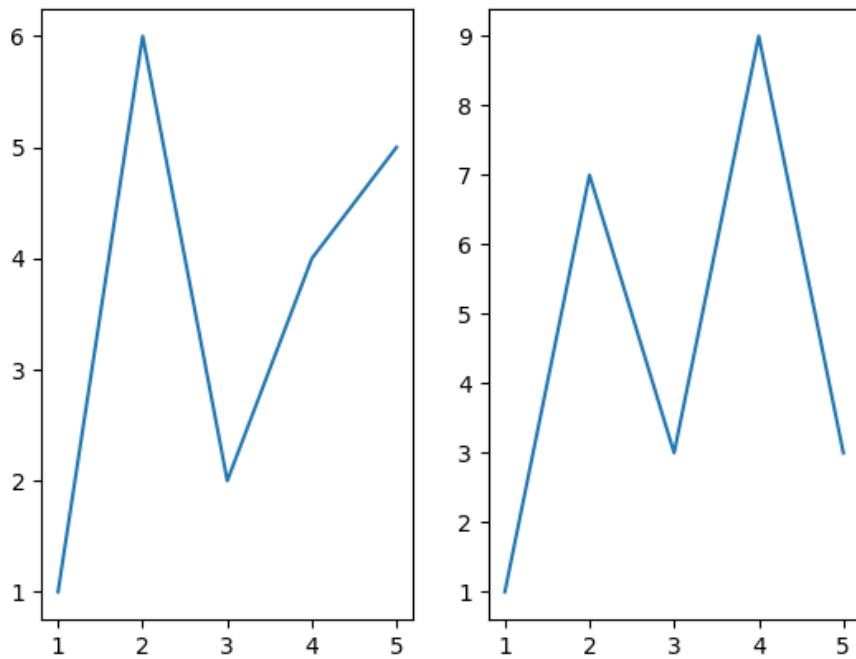
This will result in one row, two columns and an Axes in the second column:



The spot reserved for `121` is empty, since we haven't put anything there, but the `122` spot is taken by our `Axes`. Let's populate the position on `121` too:

```
1 import matplotlib.pyplot as plt
2
3 figure = plt.figure()
4 ax = figure.add_subplot(122)
5 ax2 = figure.add_subplot(121)
6
7 x = [1, 2, 3, 4, 5]
8 y = [1, 7, 3, 9, 3]
9 z = [1, 6, 2, 4, 5]
10
11 ax.plot(x, y)
12 ax2.plot(x, z)
13 plt.show()
```

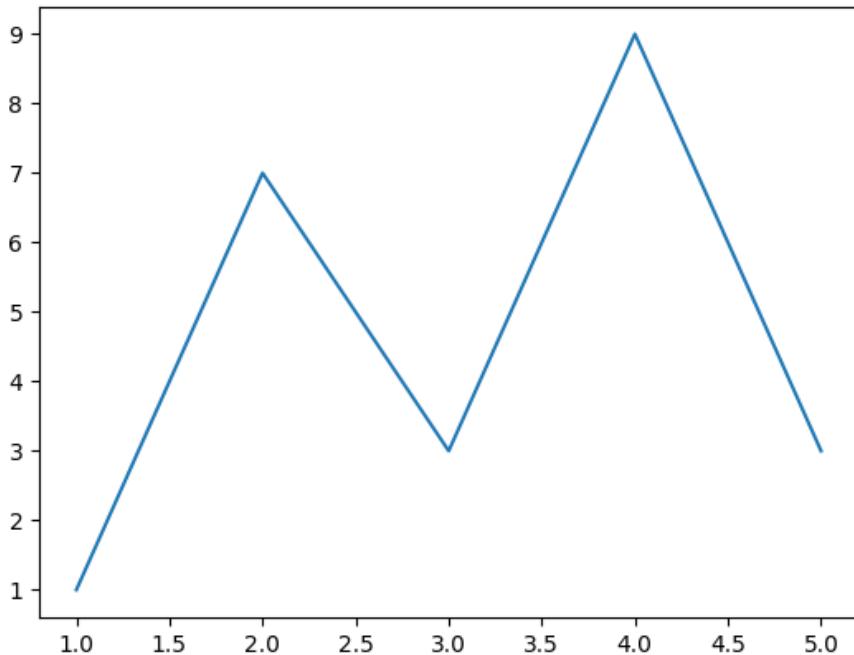
Another `Axes` is created, on the `121` position, which will put it in the same row as the previous one, but in the first column instead:



Alternatively, we can use the `subplots()` function instead, to create one or multiple subplots, as well as the `Figure` object. This is a really popular way of instantiating them, since you can do it all in one line:

```
1 import matplotlib.pyplot as plt
2
3 fig, ax = plt.subplots()
4
5 x = [1, 2, 3, 4, 5]
6 y = [1, 7, 3, 9, 3]
7
8 ax.plot(x, y)
9
10 plt.show()
```

This results in:

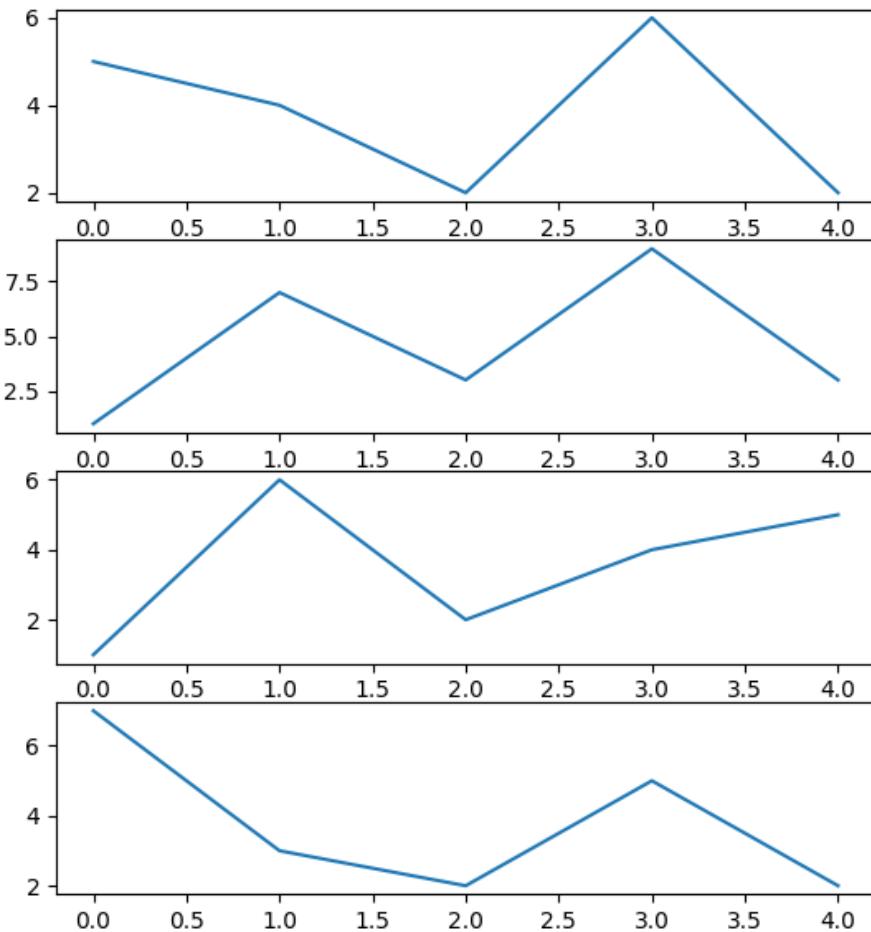


Or, if you'd like to work with more than one subplot, you simply chuck in the number of them into the `subplots()` call. Let's create 4 `Axes` objects, and plot 4 different plots on a single `Figure`:

```
1 import matplotlib.pyplot as plt
2
3 fig, ax = plt.subplots(4)
4
5 x = [5, 4, 2, 6, 2]
6 y = [1, 7, 3, 9, 3]
7 z = [1, 6, 2, 4, 5]
8 n = [7, 3, 2, 5, 2]
9
10 ax[0].plot(x)
11 ax[1].plot(y)
12 ax[2].plot(z)
13 ax[3].plot(n)
14
15 plt.show()
```

This time around, the returned `ax` is a Numpy array of `Axes` objects. We can access

them each through the usual notation `array[index]`. This creates 4 `Axes` objects and plots them in a row:



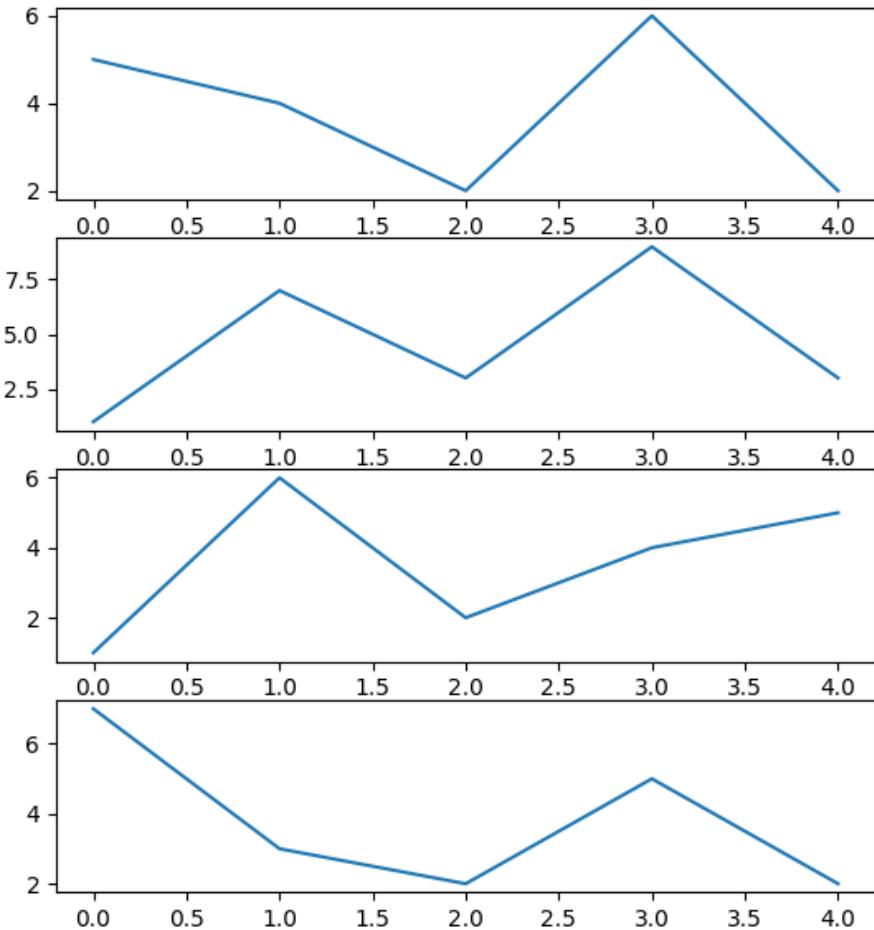
This also allows you to use a `for` loop to access all `Axes` of a `Figure`, incrementing the index on each call:

```
1 for i in range(4):
2     ax[i].plot(x)
```

On the other hand, if you want to have them as separate objects, each with a unique reference, you can:

```
1 import matplotlib.pyplot as plt
2
3 fig, (ax1, ax2, ax3, ax4) = plt.subplots(4)
4
5 x = [5, 4, 2, 6, 2]
6 y = [1, 7, 3, 9, 3]
7 z = [1, 6, 2, 4, 5]
8 n = [7, 3, 2, 5, 2]
9
10 ax1.plot(x)
11 ax2.plot(y)
12 ax3.plot(z)
13 ax4.plot(n)
14
15 plt.show()
```

This also results in:



Now, creating these 4 plots is a bit of a squeeze for the default figure size, which is 6.4×4.8 inches. The images inserted into the book showcase a stretched window, since Matplotlib windows are interactive to a degree - we can easily resize them using the mouse.

Though, you can also do that programmatically. In the following chapter, with the

basic usage of Matplotlib elements under your belt, we'll take a look at how to customize plots - add text/labels, rotate it, change its size, save plots as images, change figure size, add legends and change their size, change tick frequency, and many other options.

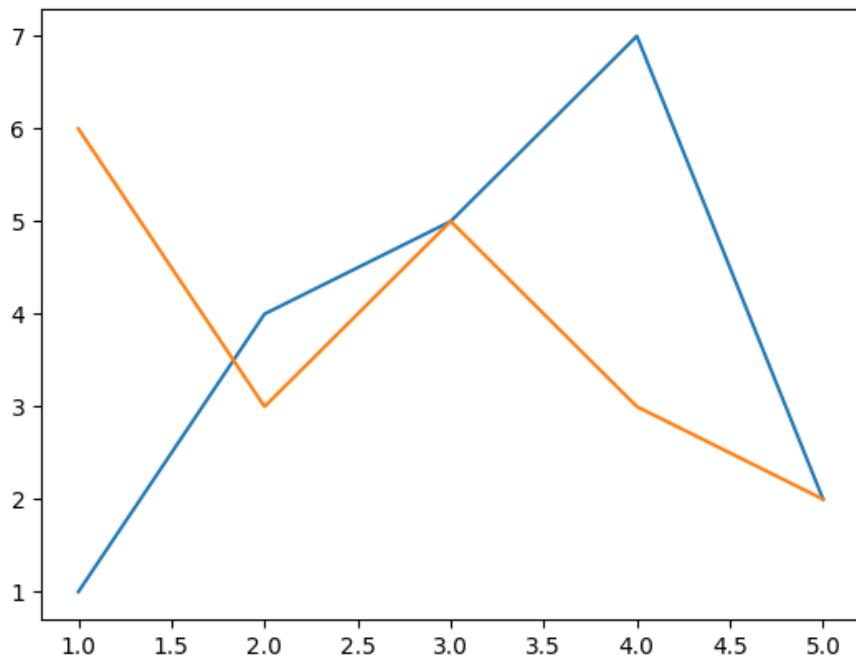
PyPlot API vs. Object-Oriented API

Choosing the API/plotting style mainly boils down to the style you prefer, and the specific plots you're creating. Both are equally customizable, but the Object-Oriented API has a more convenient customization approach, since you can directly work with objects and their parameters. The PyPlot API autoconfigures a lot of the commonly used options, so you don't have to worry about them - to a degree, it abstracts the inner-workings of Matplotlib, without locking you out from accessing the low-level parameters.

You'll use *both* of these pretty much interchangeably, all the time. In fact, you can mix both paradigms - plot on a `Figure` using the `plt` instance, and then extract the `Axes` and `Figure` objects from it:

```
1 import matplotlib.pyplot as plt
2
3 x_1 = [1, 2, 3, 4, 5]
4 y_1 = [1, 4, 5, 7, 2]
5 y_2 = [6, 3, 5, 3, 2]
6
7 plt.plot(x_1, y_1)
8
9 # Get current `Axes`
10 ax = plt.gca()
11 # Get current `Figure`
12 fig = plt.gcf()
13
14 ax.plot(x_1, y_2)
15
16 plt.show()
```

The `gca()` method stands for *Get Current Axes*, and it fetches the currently instantiated and used `Axes` object. Similarly, the `gcf()` method stands for *Get Current Figure*. Here, we've used the PyPlot API to plot a simple line plot, and then extracted both the `Axes` and `Figure` objects which we can also then use to plot further:



Note: As a general convention when it comes to customizing aspects of Matplotlib plots - there's a slight difference between the PyPlot API and the Object-Oriented API, which is very consistent.

When accessing elements, such as labels, titles, figure sizes, etc - there are two access conventions. In PyPlot's API, you directly access them, like how you'd access `xlabel()` for example. In the Object-Oriented API, you access them through getter and setter methods, such as `set_xlabel()`.

We've set some labels in the previous plots, using:

```
1 import matplotlib.pyplot as plt
2
3 # ...
4 plt.ylabel('Y-Axis Label')
5 plt.xlabel('X-Axis Label')
6 # ...
```

If we were to use the Object-Oriented approach, and were dealing with an `Axes` instance - we'd have to use the setter functions:

```
1 import matplotlib.pyplot as plt
2
3 # ...
4 ax.set_ylabel('Y-Axis Label')
5 ax.set_xlabel('X-Axis Label')
6 # ...
```

Additionally, the Object-Oriented API has two ways to *use setters*:

- The generic `set()` function
- Specific `set_x()` functions

The former allows you to write one line for multiple arguments while the latter provides you with code that's more readable if you change a large number of parameters. You'll see this same logic apply to practically all `set_x()` functions, where you can use `set(xlabel = 'n', ylabel = 'm', otheroption = x)` or set them all individually:

```
1 ax.set(xlabel = 'X-Axis Label', ylabel = 'Y-Axis Label')
2 # OR
3 ax.set_ylabel('Y-Axis Label')
4 ax.set_xlabel('X-Axis Label')
```

These all set the exact same parameters, so there's no real difference in the results - just the way you're approaching it.

Chapter 5. - Basic Matplotlib Customization

A good portion of Matplotlib's popularity comes from its customizability. Naturally, everyone working with Matplotlib will benefit significantly from being aware of these options, even if they're not really being used all the time.

In this chapter, we'll use the knowledge we've gained so far, alternating between the MATLAB/PyPlot-style approach to plotting and the Object-Oriented-style, and customize plots in a panoply of ways.

We'll explore common operations, such as changing the figure and font size, rotating text to make it fit better, saving images, setting axis ranges, adding multiple different-sized subplots, changing tick frequency, changing plot backgrounds, changing scales, but also dive into understanding *Matplotlib Text*.

There's an *abundance* of options to tweak and change with Matplotlib. In this chapter, we'll focus on the ones you'll frequently be using. In *Chapter 7. - Advanced Matplotlib Customization*, we'll focus on some of the less frequently used options, which are very useful and important nonetheless, such as understanding *Matplotlib Stylesheets*, *Matplotlib Colors and Colormaps*, and the *GridSpec*.

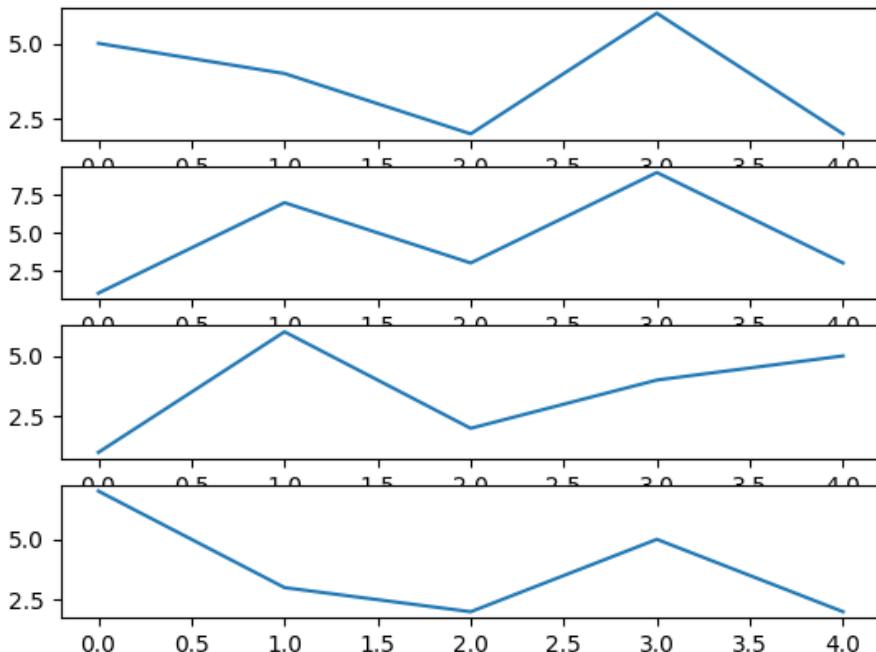
Since we've had an issue of fitting the four `Axes` objects in the last chapter, let's start off with changing the figure size.

Changing the Figure Size

Let's re-create the plot from the previous chapter, that couldn't fit very well into our Figure:

```
1 import matplotlib.pyplot as plt
2
3 fig, (ax1, ax2, ax3, ax4) = plt.subplots(4)
4
5 x = [5, 4, 2, 6, 2]
6 y = [1, 7, 3, 9, 3]
7 z = [1, 6, 2, 4, 5]
8 n = [7, 3, 2, 5, 2]
9
10 ax1.plot(x)
11 ax2.plot(y)
12 ax3.plot(z)
13 ax4.plot(n)
14
15 plt.show()
```

Without resizing the window, this plot looks something like this:



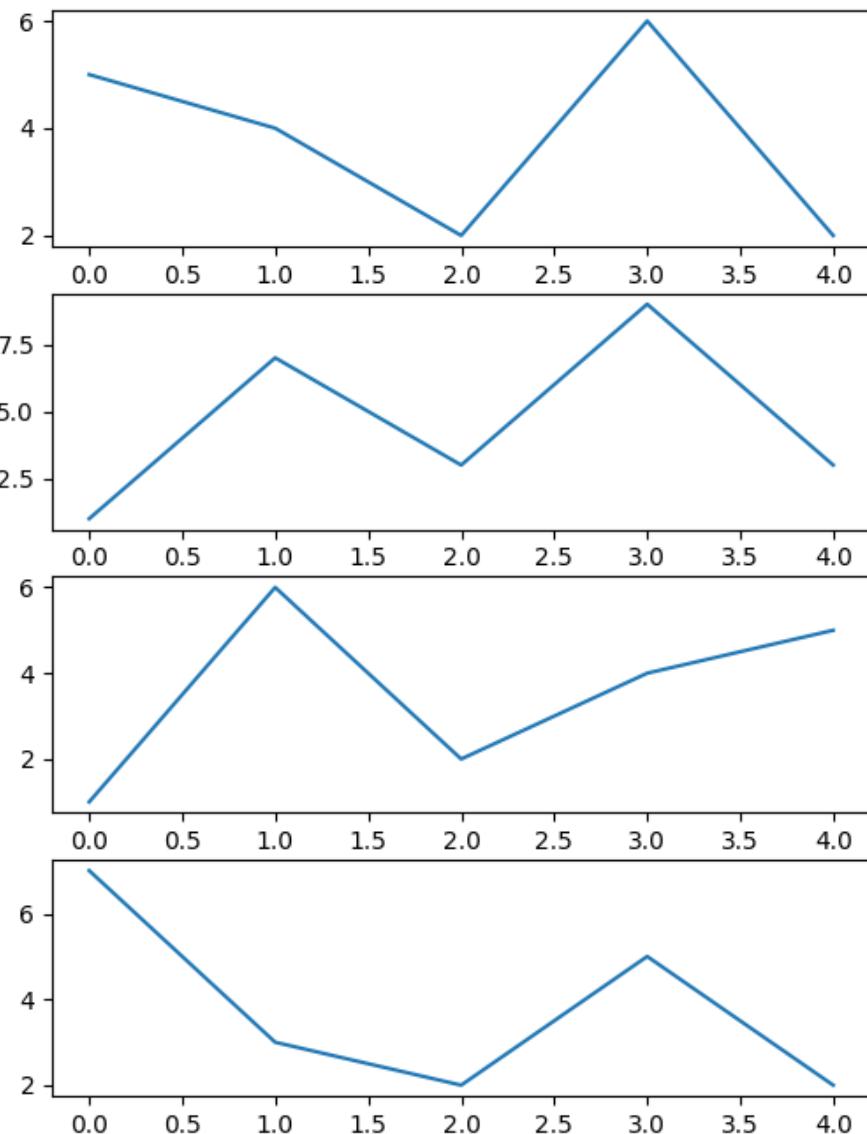
This doesn't look good. Let's take a look at how we can change the figure size to allow all four of these `Axes` objects to fit nicely.

Setting the `figsize` Argument

First off, the easiest way to change the size of a figure is to use the `figsize` argument. You can use this argument while instantiating a `Figure` object, or you can alter the height and width of a `Figure` post-festum. Let's first set the size during the initialization:

```
1 import matplotlib.pyplot as plt
2
3 fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, figsize=(6,8))
4
5 x = [5, 4, 2, 6, 2]
6 y = [1, 7, 3, 9, 3]
7 z = [1, 6, 2, 4, 5]
8 n = [7, 3, 2, 5, 2]
9
10 ax1.plot(x)
11 ax2.plot(y)
12 ax3.plot(z)
13 ax4.plot(n)
14
15 plt.show()
```

Here, while creating the `subplots()`, we've set the `figsize` argument to a tuple of `(6, 8)`. Note that the size is defined in *inches*, not pixels, which is a fairly intuitive way to imagine the size of plots. This will result in a figure that's 6in by 8in in size:



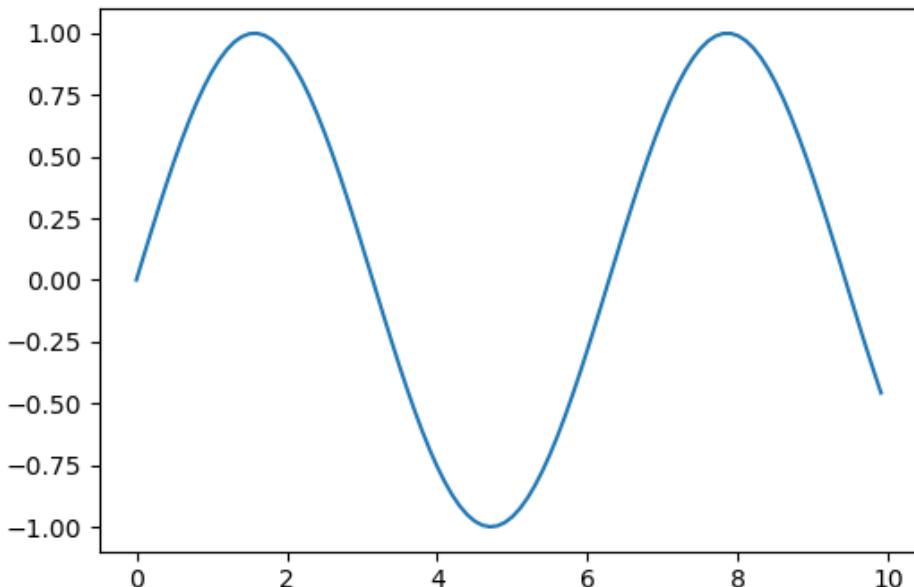
Matplotlib doesn't currently support the metric scale, though, it's easy to write a helper function to convert between the two:

```
1 def cm_to_inch(value):
2     return value/2.54
```

And then adjust the size of the plot by using the function to convert any values you put in, into inches:

```
1 plt.figure(figsize=(cm_to_inch(15),cm_to_inch(10)))
2
3 x = np.arange(0, 10, 0.1)
4 y = np.sin(x)
5 plt.plot(x, y)
6 plt.show()
```

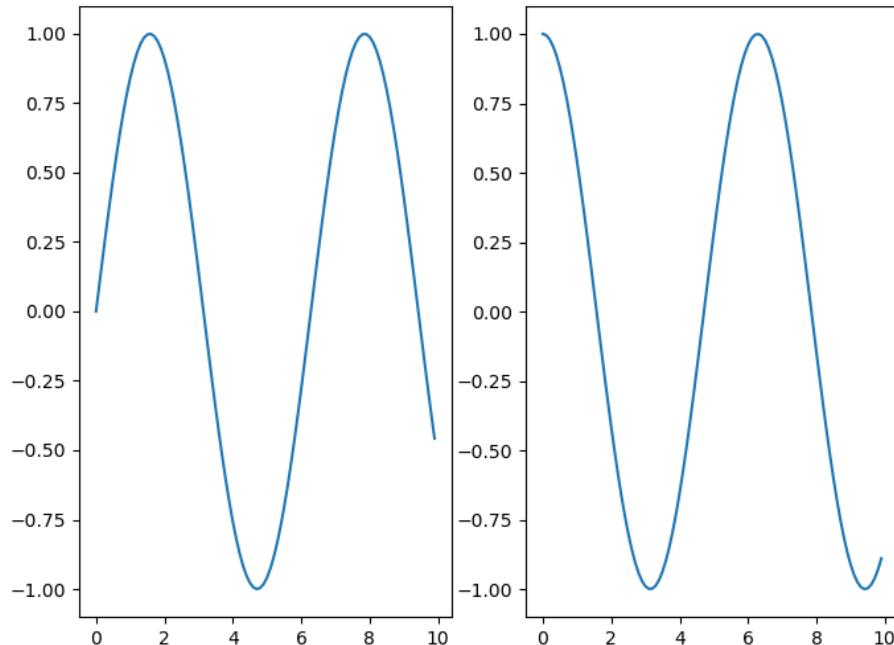
This would create a plot with the size of 15cm by 10cm:



Here, we've plotted a sine function, starting at 0 and ending at 10 with a step of 0.1, with the help of Numpy's `arange()`. Alternatively, if you're creating a `Figure` object

for your plot separately from your `Axes` object(s), you can assign the size at that time, in the `figure()` call:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0, 10, 0.1)
5 y = np.sin(x)
6 z = np.cos(x)
7
8 fig = plt.figure(figsize=(8, 6))
9
10 # Adds subplot on position 1
11 ax = fig.add_subplot(121)
12 # Adds subplot on position 2
13 ax2 = fig.add_subplot(122)
14
15 ax.plot(x, y)
16 ax2.plot(x, z)
17 plt.show()
```

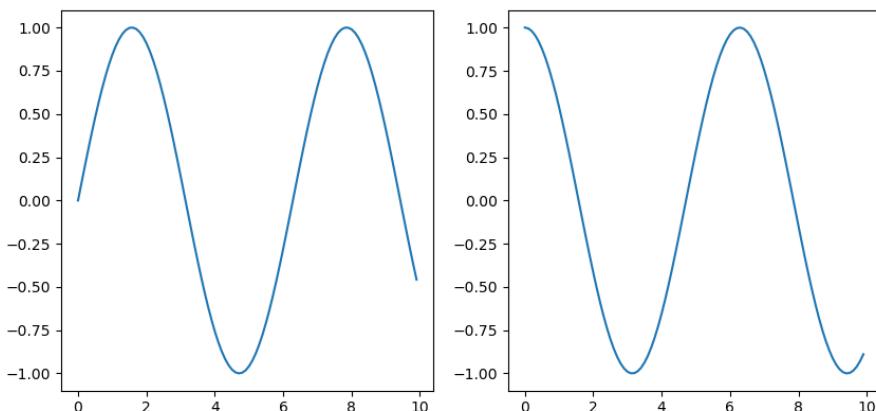


Setting the Height and Width of a Figure in Matplotlib

Instead of the `figsize` argument, we can also *set* the height and width of a figure. These can be done either via the `set()` function with the `figheight` and `figwidth` argument, or via the `set_figheight()` and `set_figwidth()` functions. Let's go with the second option:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0, 10, 0.1)
5 y = np.sin(x)
6 z = np.cos(x)
7
8 fig = plt.figure()
9
10 fig.set_figheight(5)
11 fig.set_figwidth(10)
12
13 # Adds subplot on index 1
14 ax = fig.add_subplot(121)
15 # Adds subplot on index 2
16 ax2 = fig.add_subplot(122)
17
18 ax.plot(x, y)
19 ax2.plot(x, z)
20 plt.show()
```

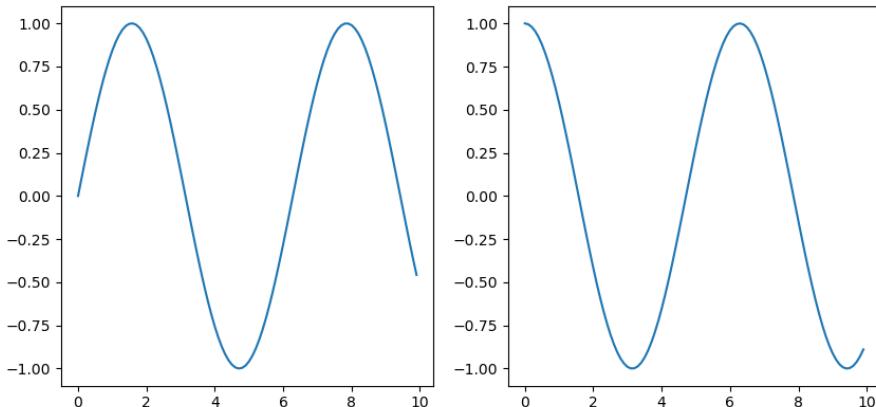
This code results in:



Finally, you can also use the `set_size_inches()` of the `Figure` object, which accepts two integers - the width and height:

```
1 fig = plt.figure()
2 fig.set_size_inches(10, 5)
3
4 # Adds subplot on position 1
5 ax = fig.add_subplot(121)
6 # Adds subplot on position 2
7 ax2 = fig.add_subplot(122)
8
9 ax.plot(x, y)
10 ax2.plot(x, z)
11 plt.show()
```

And this performs the same as setting the `figsize` argument or using the `set()`/`set_x()` functions:



Depending on the approach you use to instantiate your plot elements, such as the `Figure` and `Axes`, you'll choose approaches to change sizes accordingly. There's no *right or wrong* approach here - Matplotlib allows you to customize the figures in a myriad of ways, because it's anticipated that you might want to change the parameters in a myriad of ways.

Sometimes, you might want to change just the height, but leave the width unchanged - great case for using the `fig.set_figheight()` method. Want to change both? It'll be easier to use `fig.set_size_inches()` since you can supply both values to the method.

Understanding Matplotlib Text (Titles, Labels, Annotations)

Adding text, in any form, to plots is a common task. These can be labels for axes, titles for plots, or even values of certain markers in the form of tooltips. We use *text* to give *further context* to the numerical and visual data on the plots.

Matplotlib has great support for text, and a well-developed API for working with text. One of the key classes here is the `Text` class, not surprisingly. The class itself is what takes care of the parsing, storing and drawing of textual data on plots, given certain coordinates - and all of the methods you'll use to add labels, titles, etc. rely on the functionality of this class.

For example, when we called:

```
1 ax.set_ylabel('Y-Axis Label')
```

The method constructed a `Text` instance, with the default parameters, and set the `label` property to `'Y-Axis Label'`.

You can change these properties through the corresponding getter and setter methods, or through the `PyPlot` instance. Some of these properties are classes of their own, such as the `FontProperties` class that lets you manipulate the properties of fonts in a granular and detailed manner.

You'll rarely work manually with `Text` instances, and most of the time, you'll be using helper methods that construct instances and assign them appropriate positions. The naming conventions of the methods depends on the style you're using - keep the `set_x()` convention in mind.

Some of the common text-related commands you'll be using to add text like labels and titles to your plots are:

- `plt.text()` or `ax.text()`
- `plt.annotate()` or `ax.annotate()`
- `plt.xlabel()`/`plt.ylabel` or `ax.set_xlabel()`/`ax.set_ylabel()`
- `plt.title()` or `ax.set_title()`
- `plt.suptitle()` or `fig.suptitle()`

Whether you'll be invoking the `plt` instance or relevant `Axes` instances depends on whether you're using the PyPlot API or the Object-Oriented API.

Let's make a blank plot and add some text:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots()
5
6 fig.suptitle('This is the Figure-level Suptitle')
7 ax.set_title('This is the Axes-level Title')
8 ax.set_xlabel('X-Label')
9 ax.set_ylabel('Y-Label')
10 ax.text(0.5, 0.5, 'This is generic text')
11 ax.annotate('This is an annotation, with an arrow between \n itself and generic text',
12             xy=(0.625, 0.5),
13             xytext=(0.25, 0.25),
14             arrowprops=dict(arrowstyle='<->',
15                             connectionstyle='arc3, rad=0.15'))
16
17 plt.show()
```

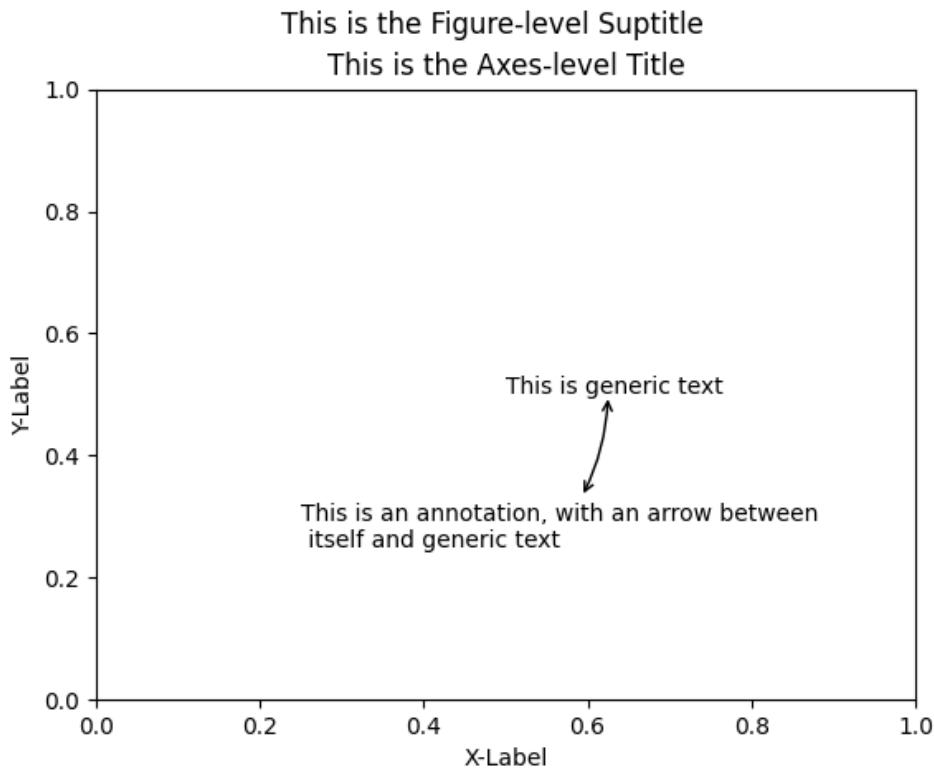
The `suptitle()` is added at the Figure-level, and is above all of its subplots. The `title` and labels can be set on the Axes-level, where each `Axes` can have separate titles and labels.

Finally, we've used the generic `text()` method and supplied the positional `x` and `y` values. Keep in mind that `x` and `y` here refer to the *actual* values on the plots - not percentages. Their positions will depend on the data you're plotting and the scale they are in. By default, Matplotlib creates a 1×1 plot (the X-axis has $0..1$ values, and the Y-axis has $0..1$ values as well). This practically means that we've set our text to start (but not end) in the very center of the plot. If we had a 10×10 plot, the text would've been plotted in the bottom-left corner instead, given the values we've supplied.

Finally, we've used the `annotate()` method, which accepts several arguments such as the `xy` tuple, `xytext` tuple, and `arrowprops`. The `xy` tuple is the *end-point* of the annotation - to what it's pointing. We've conveniently placed the arrow to point to the text here. The `xytext` tuple specifies where the text of the annotation will be positioned. We've simply moved it to $0.25, 0.25$ as to not interfere with the other text.

Note: Text instances support the use of a newline (`\n`) to break text into new lines.

Finally, the `arrowprops` accepts a dictionary with various properties you can use to customize arrows, such as the `arrowstyle` and `connectionstyle`. We'll focus on these next, but for now, let's take a look at the plot generated by this code:



Annotations are super useful when it comes to pointing things out on a plot. While you can use the `text()` method for this, annotations have more customizable features that simply make it easier to *really* point something out. The `arrowprops` dictionary is where you really customize them and you can set various properties there:

- `width` - width of the arrow
- `headwidth` - width of the head
- `shrink` - shrinking the arrow to allow for space between the annotation text and annotated point
- `boxstyle` - allows you to set the box style of the arrows

- `arrowstyle` - allows you to choose between types of arrows
- `connectionstyle` - allows you to set the connection style
- etc.

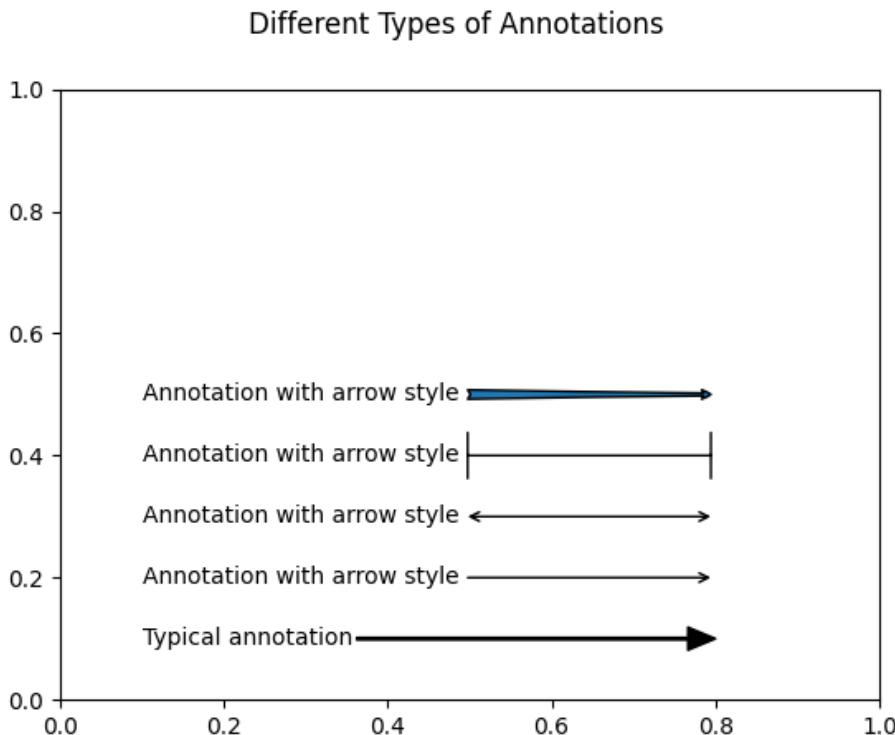
Each of these keys accepts one or more values. For example, `width` and `headwidth` accept scalar values, such as 5 and 10 (in pixels). On the other hand, keys like `boxstyle` and `connectionstyle` can get more tricky.

Let's clear the plot and add several annotations, with varying styles and types:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots()
5
6 fig.suptitle('Different Types of Annotations')
7
8 ax.annotate('Typical annotation',
9             xy=(0.8, 0.1),
10            xytext=(0.1, 0.1),
11            arrowprops=dict(facecolor='black', width=1, headwidth=10),
12            verticalalignment='center')
13
14 ax.annotate('Annotation with arrow style',
15             xy=(0.8, 0.2),
16             xytext=(0.1, 0.2),
17             arrowprops=dict(arrowstyle='->'),
18             verticalalignment='center')
19
20 ax.annotate('Annotation with arrow style',
21             xy=(0.8, 0.3),
22             xytext=(0.1, 0.3),
23             arrowprops=dict(arrowstyle='<->'),
24             verticalalignment='center')
25
26 ax.annotate('Annotation with arrow style',
27             xy=(0.8, 0.4),
28             xytext=(0.1, 0.4),
29             arrowprops=dict(arrowstyle='|-|'),
30             verticalalignment='center')
31
32 ax.annotate('Annotation with arrow style',
33             xy=(0.8, 0.5),
34             xytext=(0.1, 0.5),
35             arrowprops=dict(arrowstyle='fancy'),
36             verticalalignment='center')
37
38 plt.show()
```

Here, we've set the `verticalalignment` argument to `center` so the arrows get aligned with the center line of the annotation text. You can set it to `top` and `bottom` as well, though, the arrows will be tilted upward or downward depending on the font size.

This code results in:



In the example before the last, we've curved the arrow by adding an `arc`. Having straight lines is perfectly fine, though, for styling purposes (or not to interfere with some *other* important element on the plot), you can also curve them.

Let's plot a plot with several markers (dots) on the Y-axis, for the same X-value, resulting in a few vertical markers. We'll utilize the `scatter()` function for this - we'll talk about it in more detail in the next chapter. Suffice to say - it plots a marker for each X-Y pair. Where $xy=(1, 1)$, a marker will be placed on the $(1, 1)$ coordinates.

Then, we'll have 4 annotations, for each of these markers. Since the markers won't be in the same plane as the annotations (different Y-value), we'll curve them to their respective scatter marker:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots()
5
6 fig.suptitle('Different Types of Annotations')
7
8 # X-axis values
9 x = [1, 2]
10 # Y-axis values, empty for X=1, with several values for X=2
11 y = [(,), (1, 2, 3, 4)]
12
13 # Plot a scatter plot for each x-y pair
14 # resulting in 1-marker additions for each iteration
15 # Scatter Plots are covered in detail later
16 for xe, ye in zip(x, y):
17     ax.scatter([xe] * len(ye), ye)
18
19 # Expand view of the plot, by setting X-limit (further explained in following sections)
20 plt.xlim(0, 2)
21
22 ax.annotate('Connection style with Angle',
23             xy=(2, 1),
24             xytext=(0, 4),
25             arrowprops=dict(arrowstyle='->',
26                             connectionstyle='angle, angleA=120, angleB=0'),
27             verticalalignment='center')
28
29 ax.annotate('Connection style with Angle3',
30             xy=(2, 3),
31             xytext=(0, 2),
32             arrowprops=dict(arrowstyle='->',
33                             connectionstyle='angle3, angleA=90, angleB=0'),
34             verticalalignment='center')
35
36 ax.annotate('Connection style with Arc',
37             xy=(2, 2),
38             xytext=(0, 3),
39             arrowprops=dict(arrowstyle='->',
40                             connectionstyle='arc, angleA=0, angleB=0, armA=0, armB=45'),
41             horizontalalignment='left')
42
43 ax.annotate('Connection style with Arc3',
44             xy=(2, 4),
45             xytext=(0, 1),
46             arrowprops=dict(arrowstyle='->',
47                             connectionstyle='arc3, rad=0.25'),
48             horizontalalignment='left')
49
50 plt.show()
```

Unlike in the previous example, a key property is used - the `connectionstyle`. The property is used to, well, add *style* to a *connection*. There are four main *styles* you can apply: `angle`, `angle3`, `arc` and `arc3`.

Each of these has a few properties and their default values - namely:

Name	Possible Attributes	Class
angle	angleA=90,angleB=0,rad=0.0	Angle
angle3	angleA=90,angleB=0	Angle3
arc	angleA=0,angleB=0,armA=None,armB=None,rad=0.0	Arc
arc3	rad=0.0	Arc3

The Angle is used to create a [quadratic Bezier path](#)¹⁶ between the points referenced to by `xy` and `xytext`. The break-point is placed at the intersection of the the lines that point to the start and end point, each at a certain slope - `angleA` and `angleB`. If `angleA` is at 90 degrees of the starting point, and `angleB` is at 45 of the ending point - the lines that follow these angles *intersect* at one point. This is where the break occurs.

For this to work the difference between these two can't differ by 180 or more degrees, otherwise they will never intersect.

The optional `rad` argument rounds the edge(s) created by this break.

The Angle3 is used much for the same purpose - though, it has 3 control points, hence the name. For simple lines, you won't see a difference. For some fancy arrow styles though, you'll *have to* use Angle3 as opposed to Angle since they can only work with 3 control points. These styles are: `fancy`, `simple` and `wedge`. They can only work with Angle3 and Arc3 connection style variants.

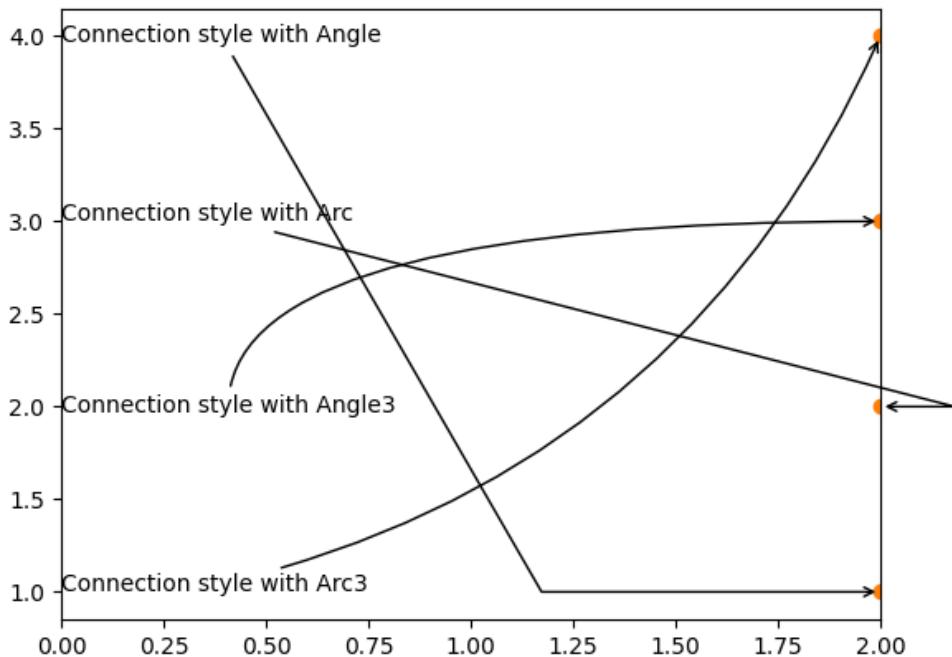
The Arc allows us to have two additional *arms* at the start and end of the path. These can be used to introduce new breaking angles that supplement the path that already exists. The optional `rad` argument rounds the edges off.

Arc3 creates a regular, simple, Bezier curve between two points and the middle control point is placed in the middle of the path. The `rad` argument is used to calculate the distance between this central control point and the straight line between the starting and ending point. In essence - the higher the `rad` value is, the more further away the central point of the curve will be - in effect, curving it more.

Let's run the code snippet and see what we end up with:

¹⁶https://en.wikipedia.org/wiki/B%C3%A9zier_curve

Different Types of Annotations



Add Legend

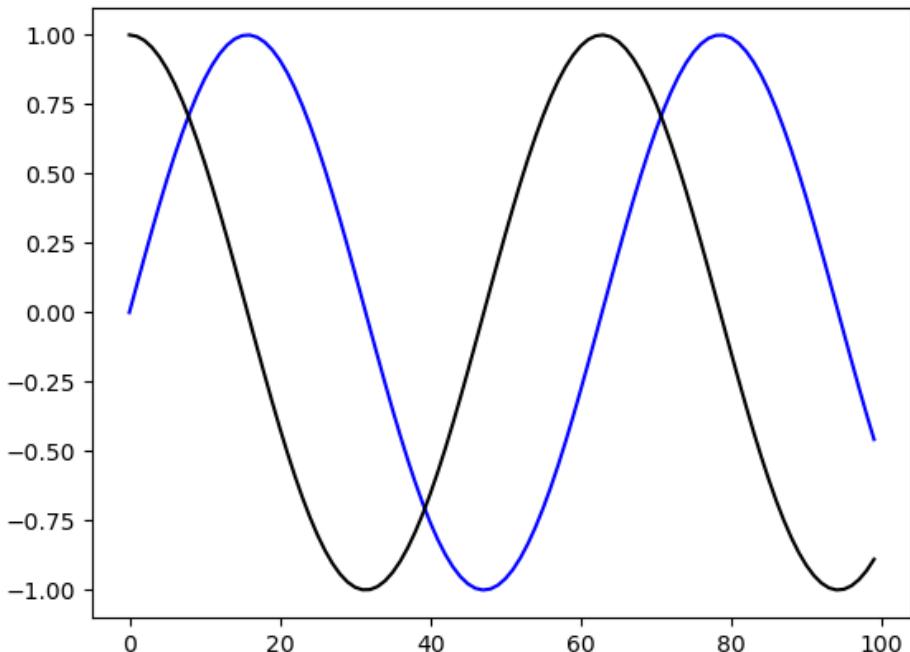
Annotations are not the only way to provide observers with a way to differentiate and interpret the data on the plot. We commonly color-code certain variables, so that they can easily be differentiated with a simple glance.

You can add annotations for these - the *green* line is the *age* variable, while the *red* line is the *population* variable, though, this becomes unwieldy. Also, it's a bit weird to annotate *features*. Annotations are typically used to point out a certain *observation*, rather than entire features.

For this, we typically look to a legend, which has a list of colors, and a list of labels for those colors. Let's first create a simple plot with two variables, each with a different color:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots()
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 ax.plot(y, color='blue')
11 ax.plot(z, color='black')
12
13 plt.show()
```

Here, we've plotted a sine function, starting at 0 and ending at 10 with a step of 0.1, as well as a cosine function in the same interval and step. Running this code results in:



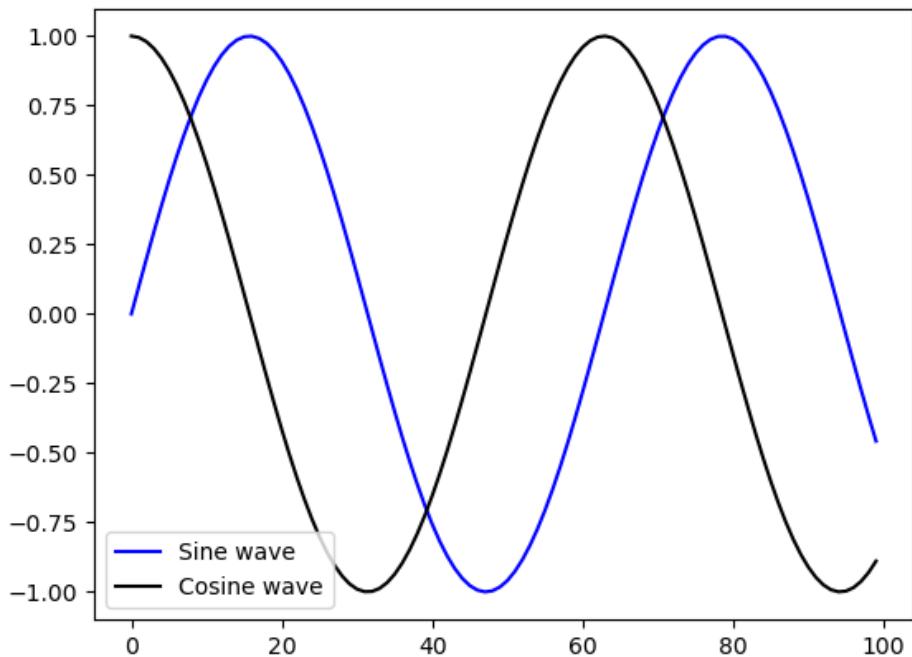
Now, let's add a legend to this plot. Firstly, we'll want to `label` these variables, so that we can refer to those labels in the legend. These labels aren't visible or appended

to the plot itself, unless we add them in the legend. Once labeled, all we have to do is call the `legend()` function on the `Axes` instance. This call accepts a few arguments for customization off the bat, such as setting a title by passing in a string to the `title` argument.

The `Legend` object is returned from this call, so we can use it to further customize it if need be - though, just calling the function will enable a legend, that's auto-placed by Matplotlib, with the *labels* we've provided to our variables:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots()
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 ax.plot(y, color='blue', label='Sine wave')
11 ax.plot(z, color='black', label='Cosine wave')
12 leg = ax.legend()
13
14 plt.show()
```

Now, if we re-run the code, a legend can be seen in the bottom-left:



Notice how the legend was automatically placed in the only free space where the waves won't run over it - other than the small edge being cut into. Matplotlib does its best to fit the legend in a place where it'll obstruct the least of the plot. In this case, it was placed in the bottom-left of the plot.

Customizing A Legend in Matplotlib

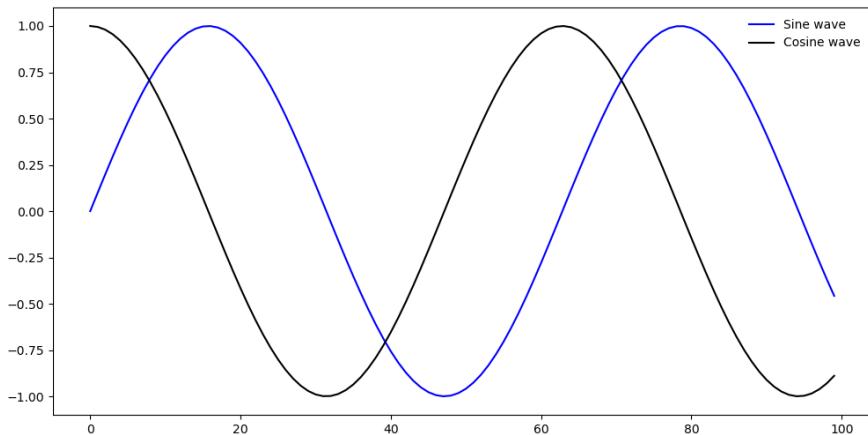
Sometimes, several locations with the same space can exist - and we might not like where Matplotlib has placed the legend at. In this case, we might want to put the legend at the top-right corner, since it's more visible that way, and a bit more natural for the human eye to notice. Let's remove the border around the legend, since we really don't need it, and place it at the top-right corner of the plot.

Additionally, we can change the plot size to allow for a bit more space:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots(figsize=(12, 6))
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 ax.plot(y, color='blue', label='Sine wave')
11 ax.plot(z, color='black', label='Cosine wave')
12 leg = ax.legend(loc='upper right', frameon=False)
13
14 plt.show()
```

The `loc` argument is used to specify the location of a legend, and it's fairly intuitive - a combination of upper, lower or center with left, right, or center. `center` alone (since `center center` doesn't make much sense) puts the legend in the center of the plot, naturally. Additionally, you can use `best` (the default value) to let Matplotlib figure this out for you.

We've also turned off the borders (frame) of the legend via the `frameon` argument:



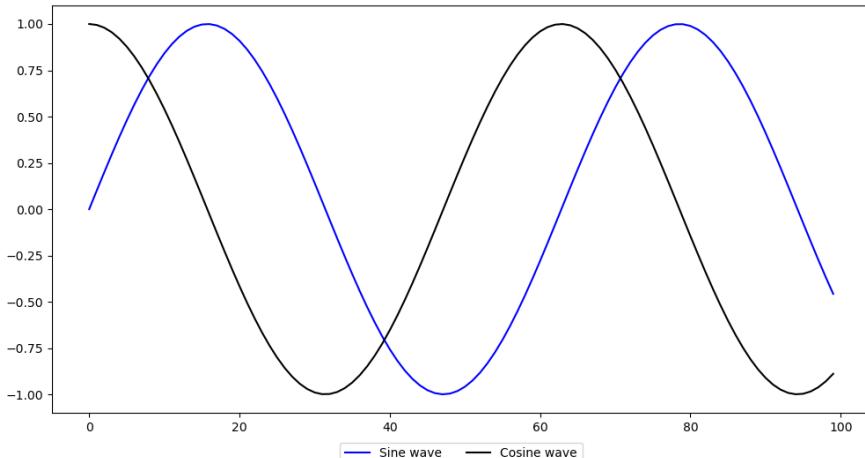
Adding a Legend Outside of Axes

Sometimes, it's tricky to place the legend within the border box of a plot. Perhaps, there are many elements going on and the entire box is filled with important data. In

such cases, you can place the legend *outside* of the axes, and away from the elements that constitute it. This is done via the `bbox_to_anchor` argument, which specifies where we want to *anchor* the legend to:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots(figsize=(12, 6))
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 ax.plot(y, color='blue', label='Sine wave')
11 ax.plot(z, color='black', label='Cosine wave')
12 leg = ax.legend(loc='center', bbox_to_anchor=(0.5, -0.10), shadow=False, ncol=2)
13
14 plt.show()
```

This results in:

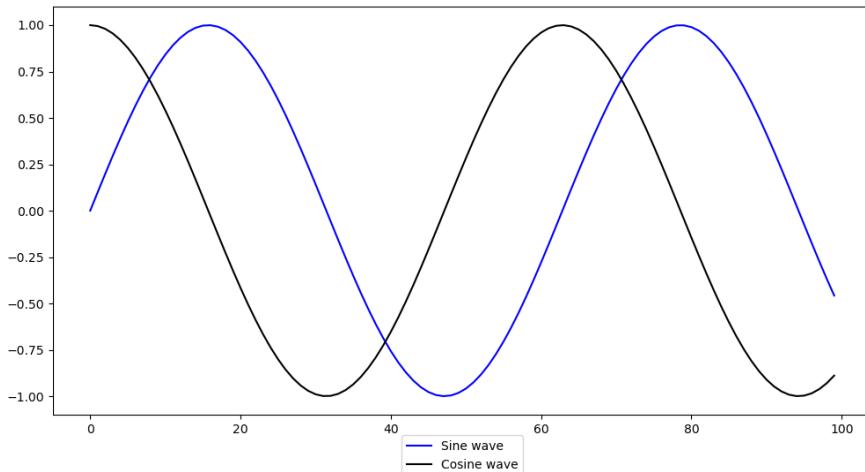


The `bbox_to_anchor` argument accepts a few arguments itself. Firstly, it accepts a tuple, which allows up to 4 elements. Here, we can specify the `x`, `y`, `width` and `height` of the legend.

We've only set the `x` and `y` values, to displace it `-0.10` below the axes, and `0.5` from the left side (`0` being the lefthand of the box and `1` the righthand side). By tweaking these, you can set the legend at any place. Within or outside of the box.

Then, we've set the `shadow` to `False`. This is used to specify whether we want a small shadow rendered below the legend or not.

Finally, we've set the `ncol` argument to 2. This specifies the number of labels in a column. Since we have two labels and want them to be in one column, we've set it to 2. If we changed this argument to 1, they'd be placed one above the other:



Note: The `bbox_to_anchor` argument is used *alongside* the `loc` argument. The `loc` argument will put the legend based on the `bbox_to_anchor`. In our case, we've put it in the center of the new, displaced, location of the *border box* we've anchored the legend to.

Changing the Legend Size

While we have successfully changed both the *location* and *anchor* of the legend, placing it outside the Axes we're plotting on, we've also made it disproportionately small compared to the plot above. This might be what you're aiming for, but it might also be the opposite of what you're aiming for.

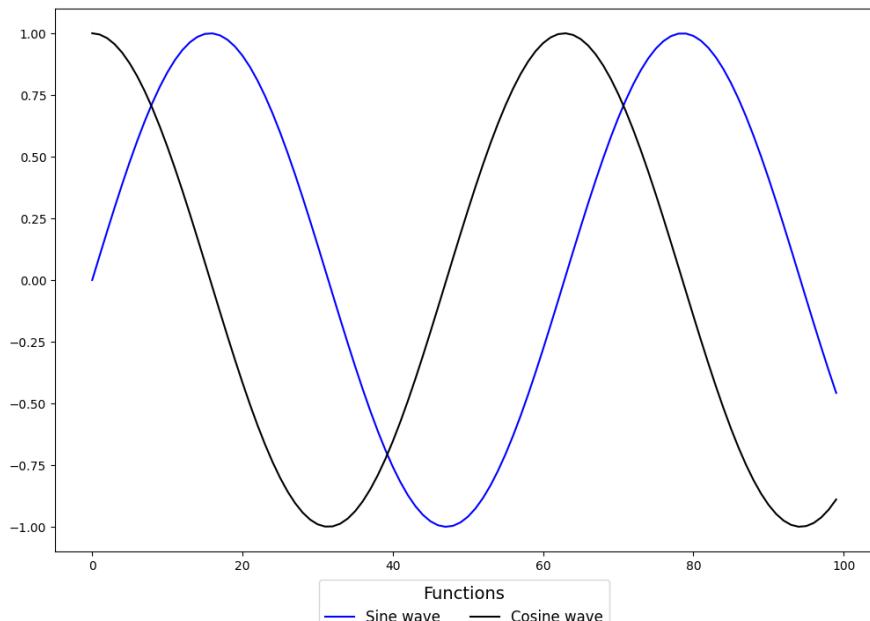
If we've taken the liberty of changing the location and anchor, we might as well also take the liberty of changing the size of the legend. Legends *scale* by the size of the text within them. Changing the font size of the text within will, therefore, change

the size of the legend as well. This is most easily achieved by passing in the `fontsize` argument when creating a legend.

Additionally, tweaking the size of the title of the legend will also enlarge it:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots(figsize=(12, 6))
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 ax.plot(y, color='blue', label='Sine wave')
11 ax.plot(z, color='black', label='Cosine wave')
12 leg = ax.legend(title='Functions', fontsize=12, title_fontsize=14,
13                  loc='center', bbox_to_anchor=(0.5, -0.10), shadow=False, ncol=2)
14
15 plt.show()
```

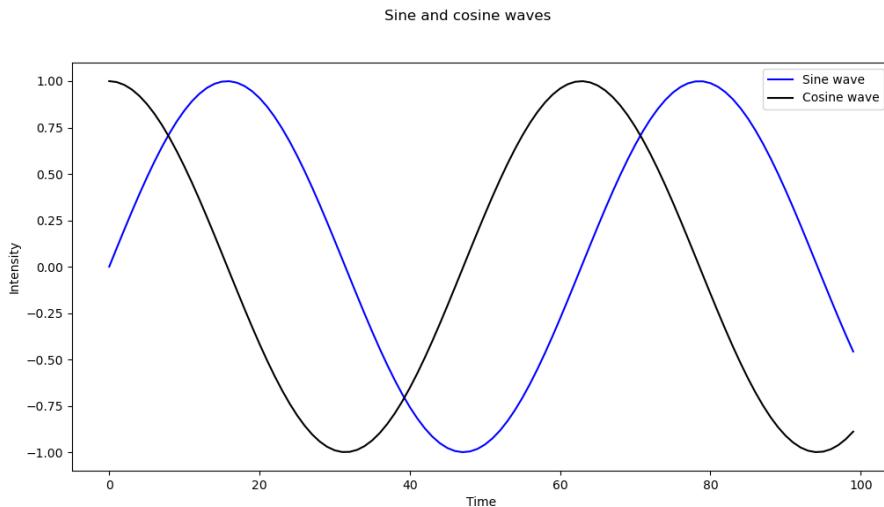
This results in:



Changing the Font Size

There are a couple of ways you can go about changing the size of fonts for text in Matplotlib. You can set the `fontsize` argument where supported, or change how Matplotlib treats fonts in general. Let's first add X and Y labels to our previous plot, as well as a title to the Axes:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots(figsize=(12, 6))
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 ax.plot(y, color='blue', label='Sine wave')
11 ax.plot(z, color='black', label='Cosine wave')
12 ax.set_title('Sine and cosine waves')
13 ax.set_xlabel('Time')
14 ax.set_ylabel('Intensity')
15 leg = ax.legend()
16
17 plt.show()
```



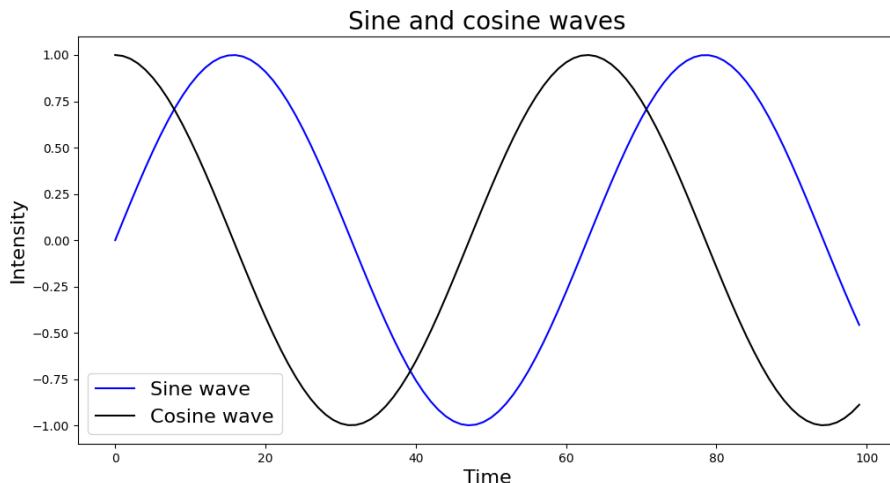
This plot looks nice, but since it's fairly big (12x6 inches) and the text now appears to be somewhat small. While sharing this visualization in a paper, or on a presentation, the attendees would probably have a bit of trouble making out the small text here.

Changing the Font Size using `fontsize`

Let's try out the simplest option. Every function that deals with text, such as `Title`, `labels` and all other textual functions accept an argument - `fontsize`. Let's revisit the code from before and specify a `fontsize` for these elements:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots(figsize=(12, 6))
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 ax.plot(y, color='blue', label='Sine wave')
11 ax.plot(z, color='black', label='Cosine wave')
12 ax.set_title('Sine and cosine waves', fontsize=20)
13 ax.set_xlabel('Time', fontsize=16)
14 ax.set_ylabel('Intensity', fontsize=16)
15 leg = ax.legend(fontsize=16)
16
17 plt.show()
```

Here, we've set the `fontsize` for the title as well as the labels for time and intensity. Running this code yields:



We can also change the size of the font in elements by passing in the `prop` argument and assigning a font size value to the "size" parameter:

```
1 leg = ax.legend(prop={"size":16})
```

Though, a more high-level and user-friendly approach is by simply using the `fontsize` argument. However, while we can set each font size like this, if we have many textual elements, and just want a uniform, general size - this approach is repetitive.

In such cases, we can turn to setting the font size *globally*.

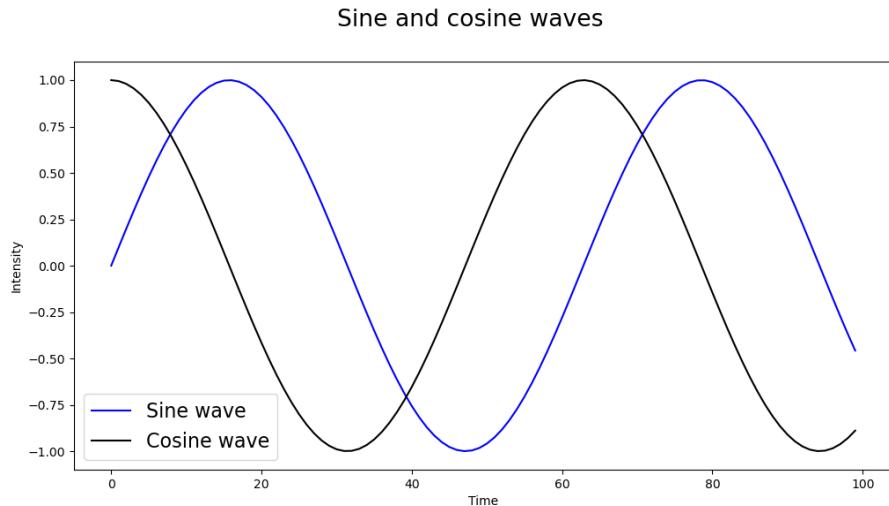
Changing the Font Size Globally

There are two ways we can set the font size globally. We'll want to set the global `font_size` parameter to a new size. We can get to this parameter via `rcParams['font.size']`. We'll be talking about *Runtime Configuration Parameters* (rc params) in more detail in *Chapter 7 - Advanced Customization in Matplotlib*. For now, suffice to say that these are the global configuration parameters, that we can always access and change:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots(figsize=(12, 6))
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 plt.rcParams['font.size'] = '16'
11
12 ax.plot(y, color='blue', label='Sine wave')
13 ax.plot(z, color='black', label='Cosine wave')
14 plt.xlabel('Time')
15 plt.ylabel('Intensity')
16 fig.suptitle('Sine and cosine waves')
17 leg = ax.legend()
18
19 plt.show()
```

You have to set these *before* the `plot()` function call since if you try to apply them afterwards, no change will be made. These are *runtime configurations*. This approach will change every object belonging to the `font` group.

However, when we run this code, it's obvious that the X and Y ticks, nor the X and Y labels didn't change in size:

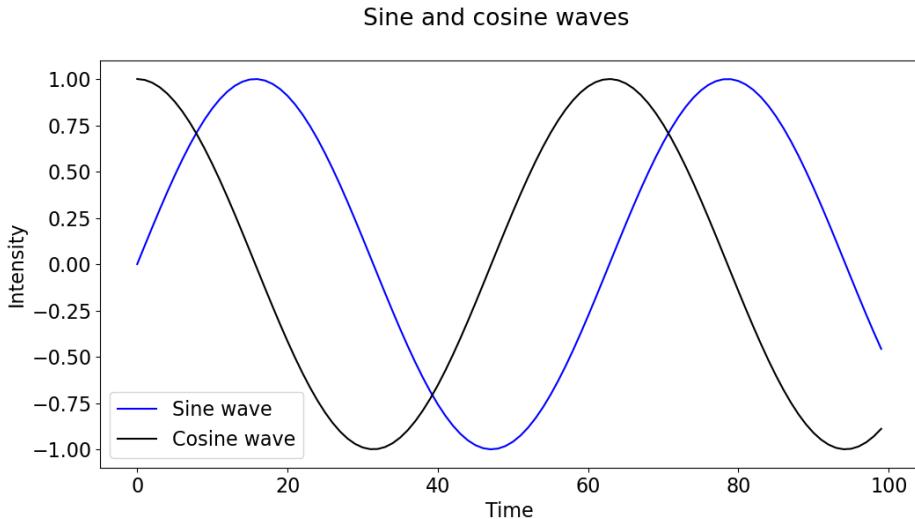


Depending on the Matplotlib version you're running, you won't be able to change these by changing the font group. You'd use `axes.labelsize` and `xtick.labelsizes/ytick.labelsizes` for them respectively, since the parameters defining the size of labels belong to the `xtick` and `ytick` groups.

If setting these doesn't change the size of labels, you can use the `set()` function passing in a `fontsize` or use the `set_fontsize()` function:

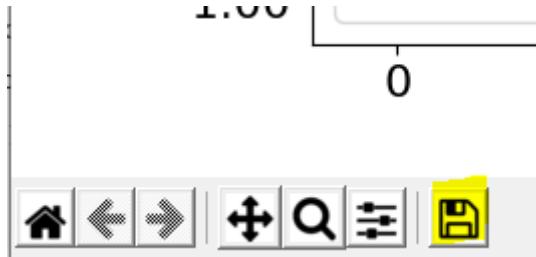
```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots(figsize=(12, 6))
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 # Set general font and tick font size
11 plt.rcParams['font.size'] = '16'
12 plt.rcParams['xtick.labelsize'] = '16'
13 plt.rcParams['ytick.labelsize'] = '16'
14
15 ax.plot(y, color='blue', label='Sine wave')
16 ax.plot(z, color='black', label='Cosine wave')
17 plt.xlabel('Time', fontsize=16)
18 plt.ylabel('Intensity', fontsize=16)
19
20 fig.suptitle('Sine and cosine waves')
21 leg = ax.legend()
22
23 plt.show()
```

This results in:



Save Plot as Image

Once you're happy with how your plot looks like, you can save it as an image, which is easily sharable with others. This can be done either by clicking the "Save" button, denoted by a floppy disk on the window opened by `plt.show()`:



Or, it can be done programmatically. Let's re-use the existing plot from the previous example and save it as an image. In all previous examples, we've generated the plot via the `plot()` function (or other plotting functions such as `scatter()`) and passed in the data we'd like to visualize.

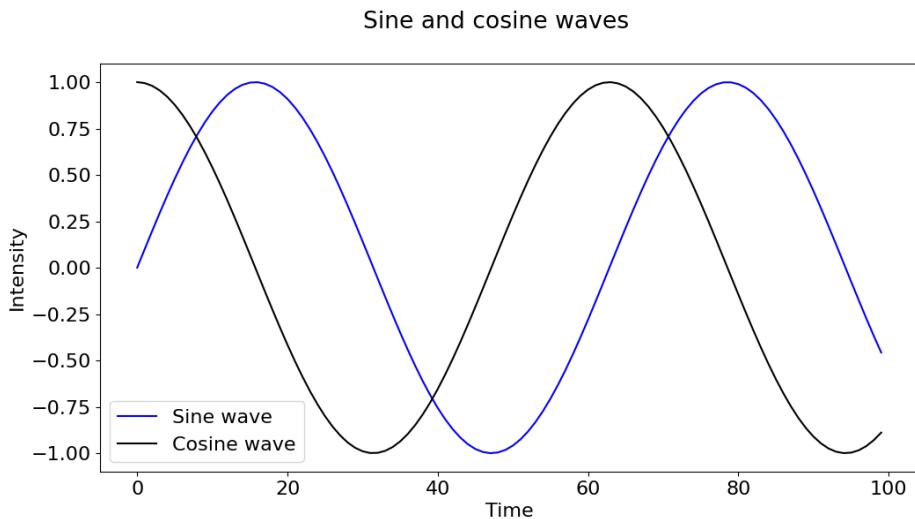
This plot is generated, but isn't shown to us, unless we call the `show()` function. The

`show()` function, as the name suggests, *shows* the generated plot to the user in a window.

Once generated, we can also *save* this figure/plot as a file instead of showing it (or both) - using the `savefig()` function:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots(figsize=(12, 6))
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 # Set font sizes
11 # plt.rcParams['font.size'] = ...
12
13 ax.plot(y, color='blue', label='Sine wave')
14 ax.plot(z, color='black', label='Cosine wave')
15 plt.xlabel('Time', fontsize=16)
16 plt.ylabel('Intensity', fontsize=16)
17
18 fig.suptitle('Sine and cosine waves')
19 leg = ax.legend()
20
21 plt.savefig('saved_figure')
```

Now, when we run the code, instead of a window popping up with the plot, we've got a file (`saved_figure.png`) in our project's directory. This file contains the exact same image we'd be shown in the window:



It's worth noting that the `savefig()` function isn't unique to the `plt` instance. You can also use it on a `Figure` instance (but not the `Axes` instance) as well:

```
1 fig.savefig('saved_figure')
```

The `savefig()` function accepts a mandatory `filename` argument. Here, we've specified the filename and format. Additionally, it accepts other options, such as `dpi`, `transparent`, `bbox_inches`, `quality`, etc.

Setting the Image DPI

The DPI parameter defines the number of dots (pixels) per inch. This is essentially the resolution of the image we're producing. A higher DPI can be used to produce larger files, with higher resolution (for larger screens) where resolution is important. On the other hand, where it isn't important, you'll be just fine with smaller resolutions:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig = plt.figure()
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8
9 plt.plot(x, y)
10 fig.savefig('saved_figure-50dpi.png', dpi = 50)
11 fig.savefig('saved_figure-100dpi.png', dpi = 100)
12 fig.savefig('saved_figure-1000dpi.png', dpi = 1000)
```

This results in three new image files on our local machine, each with a different DPI. You can see the dimensions in pixels ranging from *600x300* all the way up to *12000x6000*, and the size ranging from *27.1KB* to *1.07MB*:

	<code>saved_figure-50dpi.png</code>	Type: PNG File Dimensions: 600 x 300	Size: 27.1 KB
	<code>saved_figure-100dpi.png</code>	Type: PNG File Dimensions: 1200 x 600	Size: 68.5 KB
	<code>saved_figure-1000dpi.png</code>	Type: PNG File Dimensions: 12000 x 6000	Size: 1.07 MB

The default value is *100*.

Save Transparent Image with Matplotlib

The *transparent* argument can be used to create a plot with a transparent background. This is useful if you'll use the plot image in a presentation, on a paper or would like to present it in a custom design setting. All elements will be visible, but the background will be transparent, allowing any underlying pattern or color to be seen:

```
1 fig.savefig('saved_figure-transparent.png', transparent=True)
```

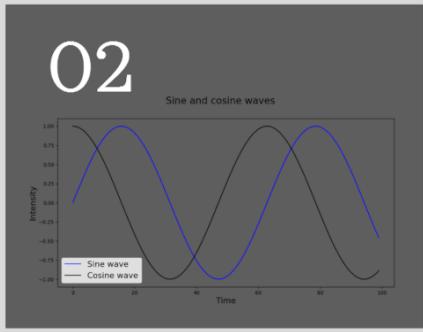
If we put this image on top of any other image or use it in a program such as *Microsoft Powerpoint*, we'll be able to integrate it seamlessly with many themes. For example, let's add it to a theme such as *Monochromatic Horizon*:

Important Company Meeting

01

If we take a look at the graph on the right...

02

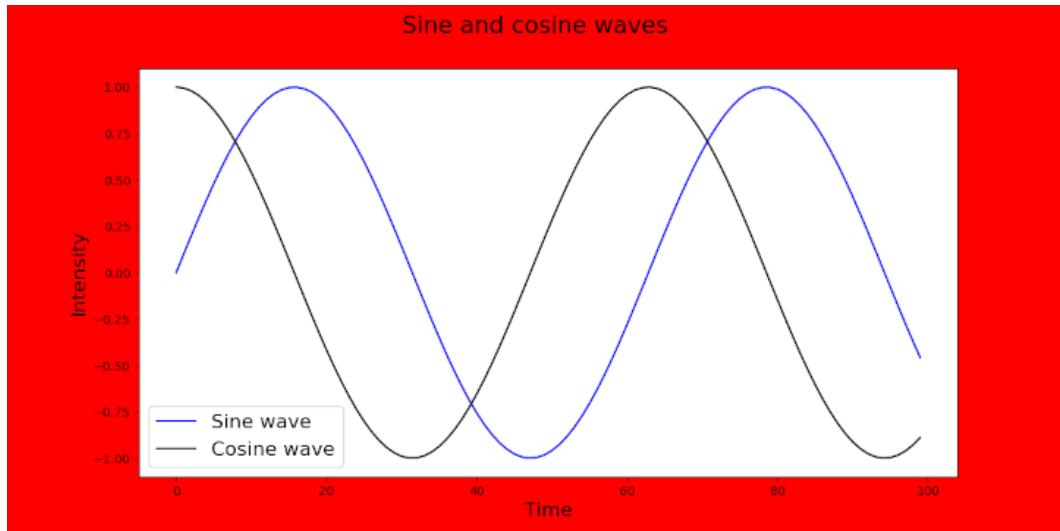


Changing Plot Colors

We can easily change the face color by setting a color for the `facecolor` argument. It accepts a `color` and defaults to `white`. Let's change it to `red`:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0, 10, 0.1)
5 y = np.sin(x)
6
7 plt.plot(x, y)
8 plt.savefig('saved_figure-colored.png', facecolor = 'red')
```

This results in:

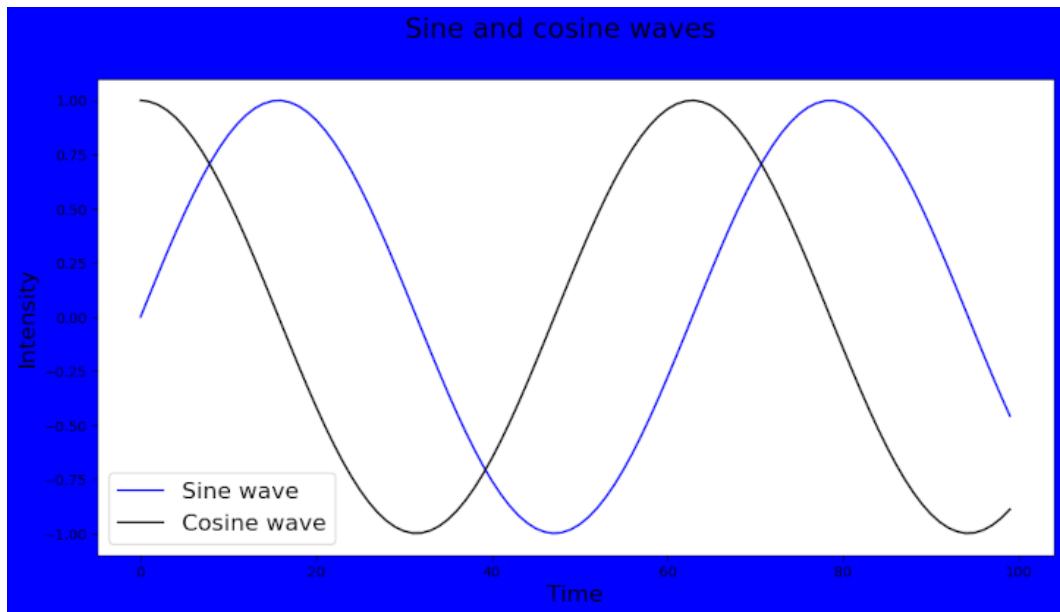


Setting Image Border Box

The `bbox_inches` argument accepts a string and specifies the border around the box we're plotting. If we'd like to set it to be `tight`, i.e. to crop around the box as much as possible, we can set the `bbox_inches` argument to '`tight`':

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.arange(0, 10, 0.1)
5 y = np.sin(x)
6
7 plt.plot(x, y)
8 plt.savefig('saved_figure-tight.png', bbox_inches = 'tight', facecolor='red')
```

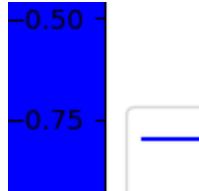
This results in a tightly packed box which is easier to see if we color the face with a different color:



Instead of a string, we can also pass in a `Bbox`. By default, the `Bbox` for your plot encompasses the entire plot, with some padding on each side. When we set the `Bbox` to `tight`, the padding was reduced.

You can also set the size of the `Bbox` to any arbitrary value by setting its starting and ending coordinates (really, *inches*). For example, we can set the `Bbox` to `start` at the 1-inch mark on the X-axis, and 1-inch mark on the Y-axis. And then, to `end` at the 2-inch mark on the X-axis, and 2-inch mark on the Y-axis:

```
1 fig.savefig('saved_figure-bbox.png', bbox_inches=Bbox([[1, 1], [2, 2]]), facecolor = 'blue')
2 e')
```



This creates a 1x1 inch box, starting from the bottom left - cropping our view quite a bit.

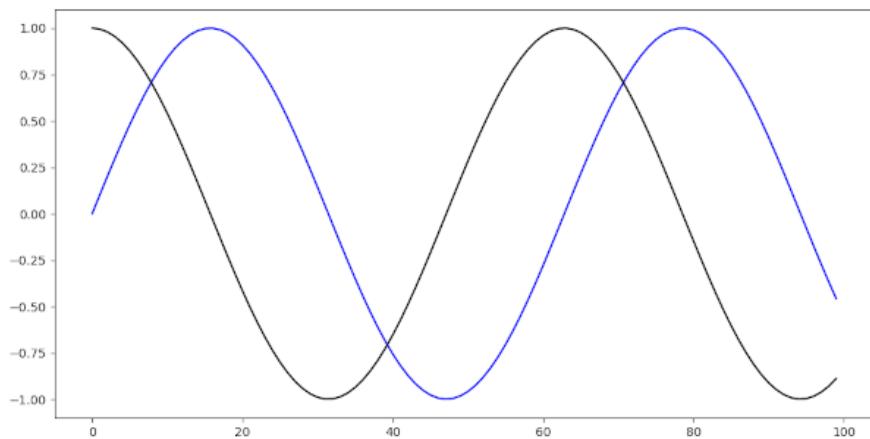
Set Axis Range (`xlim`, `ylim`)

In the previous example, we cropped the plot while we were saving it. We can also crop and focus on certain parts of plots by setting the *axis range* - also known as X-limit and Y-limit.

If we limit the axis to show only a certain range of values - we can effectively crop the range of the data shown in the plot. The same goes the other way around - we can *expand* the range of data by setting a higher range than we have data for. Matplotlib automatically sets the axis range based on the provided data - if you have 5 values on the X-axis, it'll set the X-limit (with a bit of padding) to show those 5 values. Same goes for the Y-axis.

Of course, we're free to change these parameters as we see fit. Let's strip down our plot example down for code brevity:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots(figsize=(12, 6))
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 ax.plot(y, color='blue', label='Sine wave')
11 ax.plot(z, color='black', label='Cosine wave')
12
13 plt.show()
```



The range of this X-axis currently goes from 0 to 100, with a bit of padding left and right. The range of the Y-axis goes from -1 to 1. Both of these conform to the data we've generated with the `np.arange()` function.

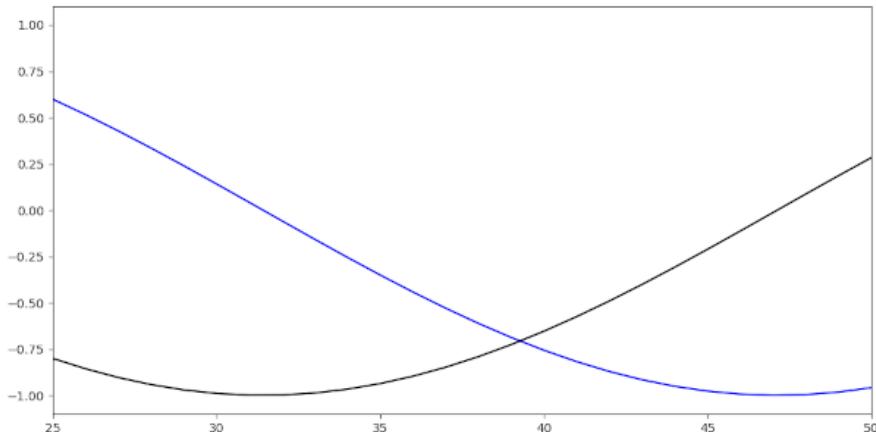
These limits can be accessed either through the PyPlot instance, or the `Axes` instance.

Setting the X-Limit (`xlim`) for `Axes`

Let's first set the X-limit, using either the PyPlot and `Axes` instances. Both of these methods accept a tuple - the left and right limits. So, for example, if we wanted to truncate the view to only show the data in the range of 25-50 on the X-axis, we'd use `xlim([25, 50])`:

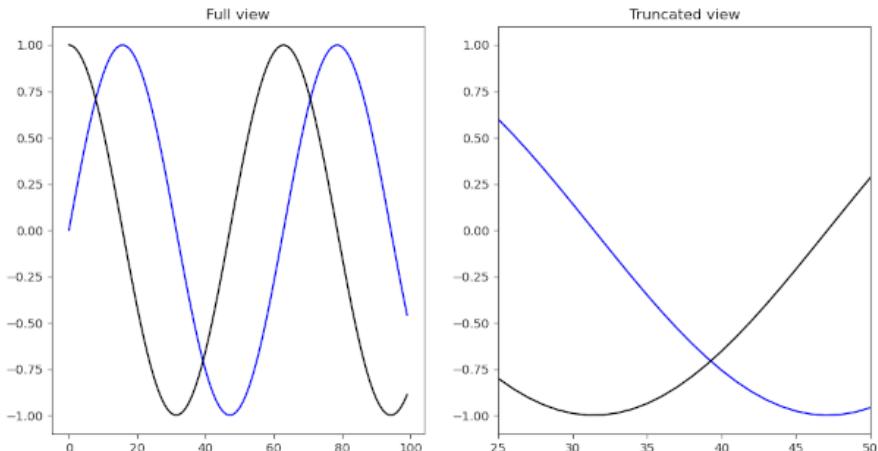
```
1 fig, ax = plt.subplots(figsize=(12, 6))
2
3 x = np.arange(0, 10, 0.1)
4 y = np.sin(x)
5 z = np.cos(x)
6
7 ax.plot(y, color='blue', label='Sine wave')
8 ax.plot(z, color='black', label='Cosine wave')
9
10 plt.xlim([25, 50])
11 plt.show()
```

This limits the view on the X-axis to the data between 25 and 50 and results in:



This same effect can be achieved by setting these via the `ax` object. This way, if we have multiple `Axes`, we can set the limit for them separately:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig = plt.figure(figsize=(12, 6))
5
6 x = np.arange(0, 10, 0.1)
7 y = np.sin(x)
8 z = np.cos(x)
9
10 ax = fig.add_subplot(121)
11 ax2 = fig.add_subplot(122)
12
13 ax.set_title('Full view')
14 ax.plot(y, color='blue', label='Sine wave')
15 ax.plot(z, color='black', label='Cosine wave')
16
17 ax2.set_title('Truncated view')
18 ax2.plot(y, color='blue', label='Sine wave')
19 ax2.plot(z, color='black', label='Cosine wave')
20
21 ax2.set_xlim([25, 50])
22
23 plt.show()
```



Setting the Y-Limit (ylim) for Axes

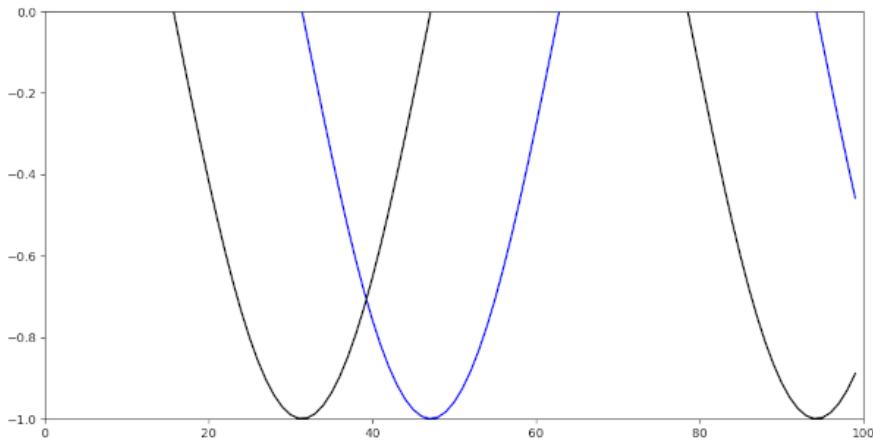
Now, let's set the Y-limit. This can be achieved with the same two approaches:

```
1 ax.plot(y, color='blue', label='Sine wave')
2 ax.plot(z, color='black', label='Cosine wave')
3
4 plt.ylim([-1, 0])
```

Or:

```
1 ax.plot(y, color='blue', label='Sine wave')
2 ax.plot(z, color='black', label='Cosine wave')
3
4 ax.set_ylim([-1, 0])
```

Both of which result in:



Change Tick Frequency

So far, we've been plotting mainly functions without much noise. This is fairly rare in the real world. The difference between two ticks isn't *too* important since there isn't a lot of change in the value along that axis.

When we're dealing with data that varies more than sine and cosine functions - we might want to tweak the number of ticks shown, which will in turn allow us to read the data much more accurately. When dealing with dates, you might also want to

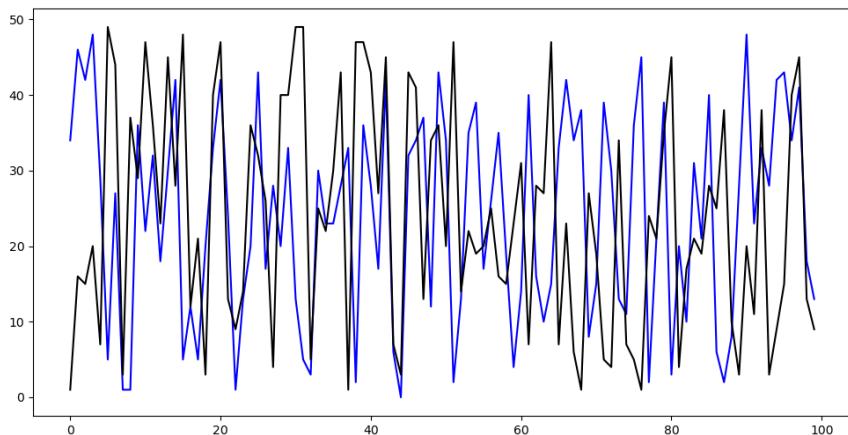
specify the frequency of the dates being shown - every day, month or year warranting a tick of its own.

Matplotlib, like usual, automatically tries to set the tick frequency to match the data you provide. Where it sees fit - the tick frequency is increased, and vice versa. Though, this doesn't always work as you had envisioned the plot to be like, even though it's satisfactory most of the time.

Let's create a plot with a *random* set of numbers, which will be much more erratic than our usual plots so far:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig = plt.subplots(figsize=(12, 6))
5
6 x = np.random.randint(low=0, high=50, size=100)
7 y = np.random.randint(low=0, high=50, size=100)
8
9 plt.plot(x, color='blue')
10 plt.plot(y, color='black')
11
12 plt.show()
```

x and y range from 0-50, and the length of these arrays is 100. This means, we'll have 100 data points for each of them:



The frequency of the ticks on the X-axis is `20`. What if we wanted to have a tick on every 5 steps, not 20? The same goes for the Y-axis. What if the distinction on this axis is even more crucial, and we'd want to have each tick on *every* step?

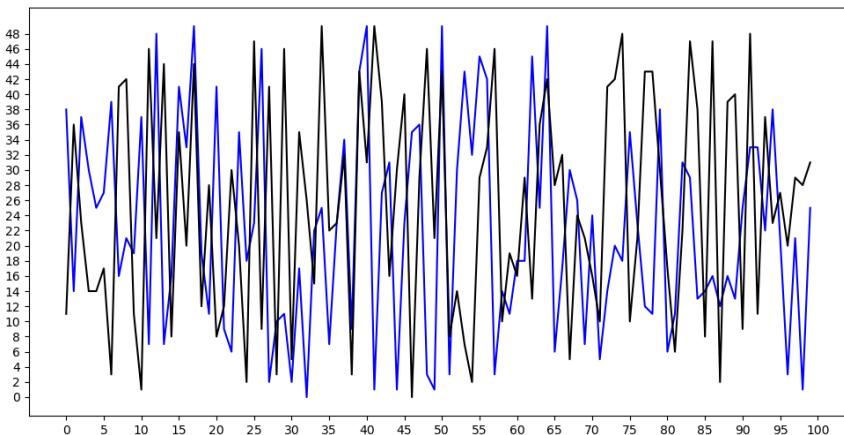
Setting Figure-Level Tick Frequency

As usual - you can set the tick frequency on a *Figure-level* and *Axes-level*. Let's change the figure-level tick frequency. If we have multiple `Axes` on it, the ticks on all of these will be uniform and will have the same frequency:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig = plt.subplots(figsize=(12, 6))
5
6 x = np.random.randint(low=0, high=50, size=100)
7 y = np.random.randint(low=0, high=50, size=100)
8
9 plt.plot(x, color='blue')
10 plt.plot(y, color='black')
11
12 plt.xticks(np.arange(0, len(x)+1, 5))
13 plt.yticks(np.arange(0, max(y), 2))
14
15 plt.show()
```

You can use the `xticks()` and `yticks()` functions and pass in an array denoting the *actual ticks*. On the X-axis, this array starts on `0` and ends at the length of the `x` array. On the Y-axis, it starts at `0` and ends at the max value of `y`.

The final argument is the `step`. This is where we define how large each step should be. We'll have a tick at every 5 steps on the X-axis and a tick on every 2 steps on the Y-axis:



It's worth noting that the `yticks()` and `xticks()` functions don't change the frequency. They *set the ticks*, and the `step` argument helps us in displaying them at a given frequency.

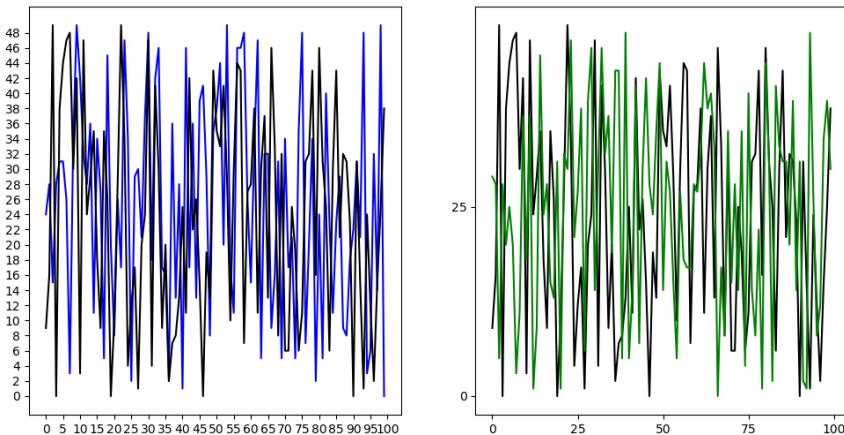
Setting Axis-Level Tick Frequency

If you have multiple plots going on, you might want to change the tick frequency on the `Axes`-level. For example, you want infrequent ticks on one graph, with frequent ticks on the other.

You can use the `set_xticks()` and `set_yticks()` functions on the returned `Axes` instance when adding subplots to a `Figure`. Let's create a `Figure` with two axes and change the tick frequency on them separately:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig = plt.figure(figsize=(12, 6))
5
6 ax = fig.add_subplot(121)
7 ax2 = fig.add_subplot(122)
8
9 x = np.random.randint(low=0, high=50, size=100)
10 y = np.random.randint(low=0, high=50, size=100)
11 z = np.random.randint(low=0, high=50, size=100)
12
13 ax.plot(x, color='blue')
14 ax.plot(y, color='black')
15 ax2.plot(y, color='black')
16 ax2.plot(z, color='green')
17
18 ax.set_xticks(np.arange(0, len(x)+1, 5))
19 ax.set_yticks(np.arange(0, max(y), 2))
20 ax2.set_xticks(np.arange(0, len(x)+1, 25))
21 ax2.set_yticks(np.arange(0, max(y), 25))
22
23 plt.show()
```

Now, this results in:



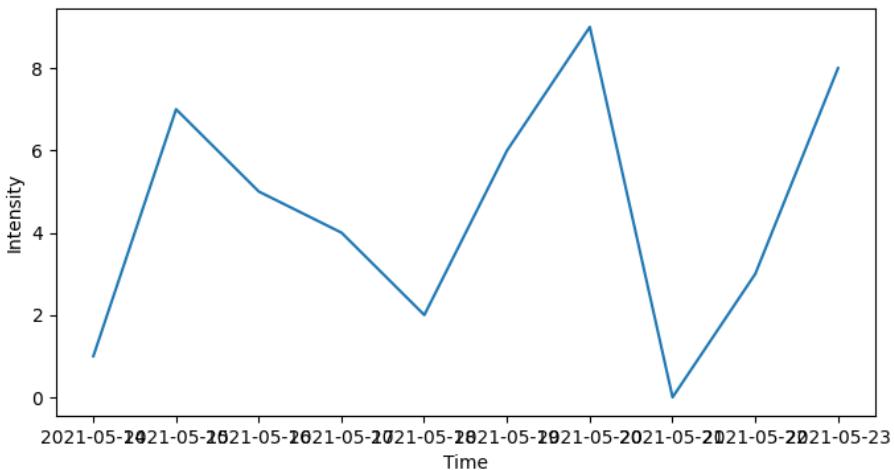
These have the same type of data - and we've set a different tick frequency for them. A thing to keep in mind is that if they're *too frequent*, the ticks will overlap. Thankfully, we can *rotate them* to allow for more space.

Rotate Axis Tick Labels

Tick labels, if numerical, typically fit nicely in most visualizations, unless there's a bunch of them. Though, when working with categorical data, *especially* with timeseries data which includes dates - these oftentimes overlap.

Let's take a look at a simple plot that visualizes a variable against dates:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 from datetime import datetime
4 import random
5
6 fig, ax = plt.subplots(figsize=(8, 4))
7
8 # [Timestamp('2021-05-14 00:05:14.652993', freq='D'),
9 # Timestamp('2021-05-15 00:05:14.652993', freq='D')...
10 x = pd.date_range(datetime.today(), periods=10).tolist()
11 # [1, 7, 5, 4, 2, 6, 9, 0, 3, 8]
12 y = random.sample(range(0, 10), 10)
13
14 ax.plot(x, y)
15 plt.xlabel('Time')
16 plt.ylabel('Intensity')
17
18 plt.show()
```



Dates are typically long - on average, longer than most other categorical variables. This plot doesn't have a lot of data at all - only 10 entries. 10 entries are plenty enough to get dates to overlap. We could fix this easily by enlarging the plot - though, even this becomes obsolete with a larger number of dates, and it isn't a very elegant solution, since this plot might very well be a *subplot* on a Figure with other plots, in which case, we can't just enlarge it.

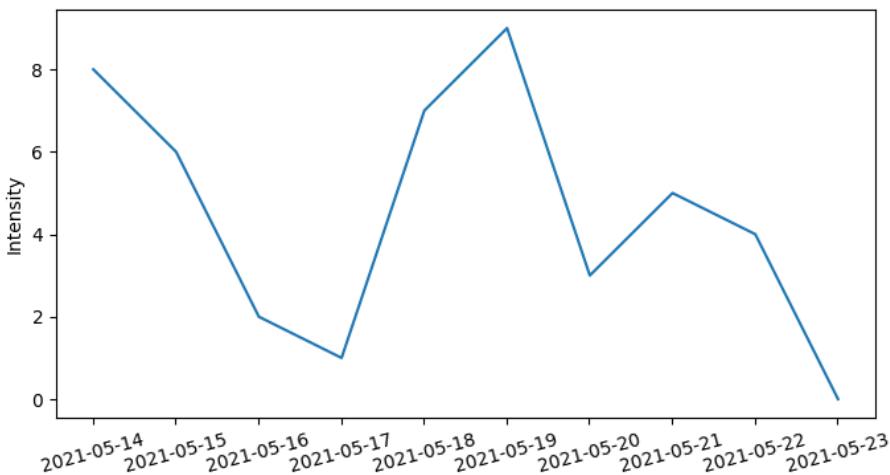
Naturally, we can rotate these, making space between them.

Rotating X-Axis Tick Labels

As usual - there are two ways to go about rotating X-Axis tick labels - rotate them on the Figure-level using `plt.xticks()` or rotate them on an Axes-level by using `tick.set_rotation()` individually, or even by using `ax.set_xticklabels()` and `ax.xaxis_params()`. Let's start off with the first option:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import pandas as pd
4 from datetime import datetime
5 import random
6
7 fig, ax = plt.subplots(figsize=(8, 4))
8
9 x = pd.date_range(datetime.today(), periods=10).tolist()
10 y = random.sample(range(0, 10), 10)
11
12 ax.plot(x, y)
13 plt.xlabel('Time')
14 plt.ylabel('Intensity')
15 plt.xticks(rotation = 15)
16 plt.show()
```

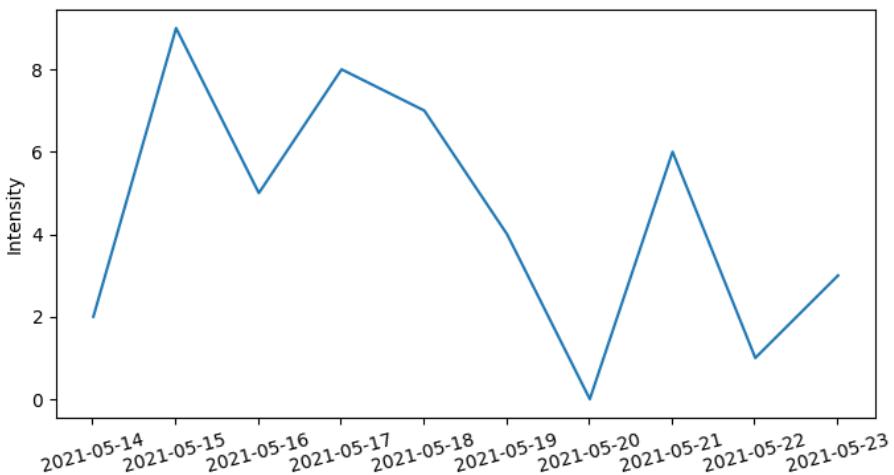
Here, we've set the rotation of `xticks` to 15, signifying a 15-degree tilt, counter-clockwise:



Alternatively, we could've iterated over the ticks in the `ax.get_xticklabels()` list. Then, we can call `tick.set_rotation()` on each of them:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import pandas as pd
4 from datetime import datetime
5 import random
6
7 fig, ax = plt.subplots(figsize=(8, 4))
8 x = pd.date_range(datetime.today(), periods=10).tolist()
9 y = random.sample(range(0, 10), 10)
10
11 ax.plot(x, y)
12 plt.draw()
13
14 for tick in ax.get_xticklabels():
15     tick.set_rotation(15)
16
17 plt.xlabel('Time')
18 plt.ylabel('Intensity')
19 plt.show()
```

This also results in:

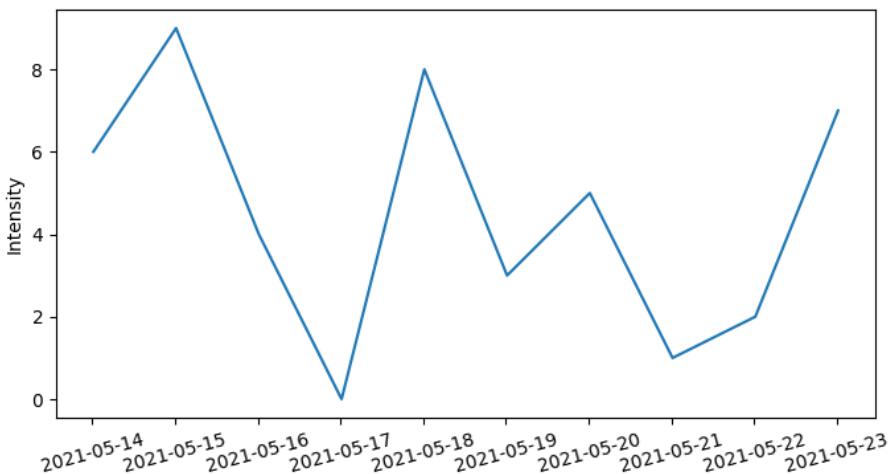


Note: For this approach to work, you'll need to call `plt.draw()` before accessing or setting the X-tick labels. This is because the labels are populated after the plot is drawn, otherwise, they'll return empty text values.

And finally, you can use the `ax.tick_params()` function and set the label rotation there:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import pandas as pd
4 from datetime import datetime
5 import random
6
7 fig, ax = plt.subplots(figsize=(8, 4))
8
9 x = pd.date_range(datetime.today(), periods=10).tolist()
10 y = random.sample(range(0, 10), 10)
11
12 ax.plot(x, y)
13 ax.tick_params(axis='x', labelrotation = 15)
14
15 plt.xlabel('Time')
16 plt.ylabel('Intensity')
17 plt.show()
```

This also results in:



Rotate Y-Axis Tick Labels

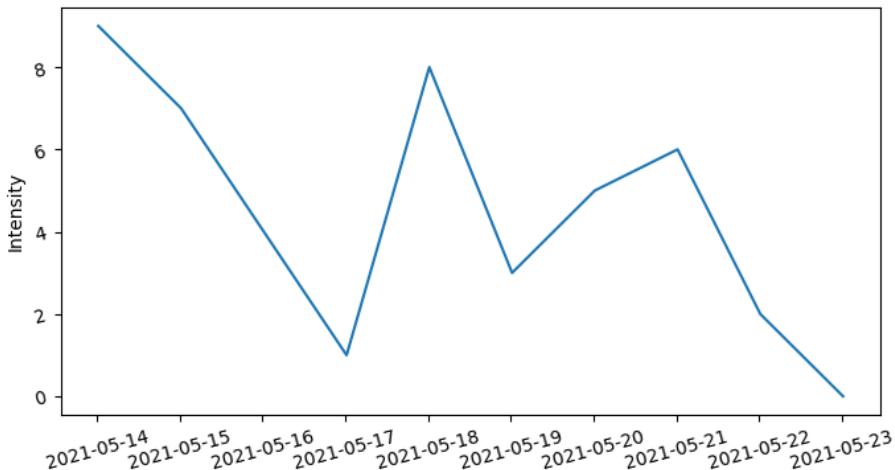
The exact same steps can be applied for the Y-Axis labels. Firstly, you can change it on the Figure-level with `plt.yticks()`, or on the Axes-level by using `tick.set_rotation()` or by manipulating the `ax.tick_params()`.

Let's start off with the first option:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 import pandas as pd
4 from datetime import datetime
5 import random
6
7 fig, ax = plt.subplots(figsize=(8, 4))
8
9 x = pd.date_range(datetime.today(), periods=10).tolist()
10 y = random.sample(range(0, 10), 10)
11
12 ax.plot(x, y)
13
14 # Approach 1
15 plt.yticks(rotation=15)
16 plt.xticks(rotation=15)
17
18 # Approach 2
19 plt.draw()
20
21 for tick in ax.get_yticklabels():
```

```
22     tick.set_rotation(15)
23
24 # Approach 3
25 ax.tick_params(axis='y', labelrotation = 15)
26
27 plt.xlabel('Time')
28 plt.ylabel('Intensity')
29 plt.show()
```

Using *either* of these approaches results in the same plot:



Rotate Dates to Fit Automatically

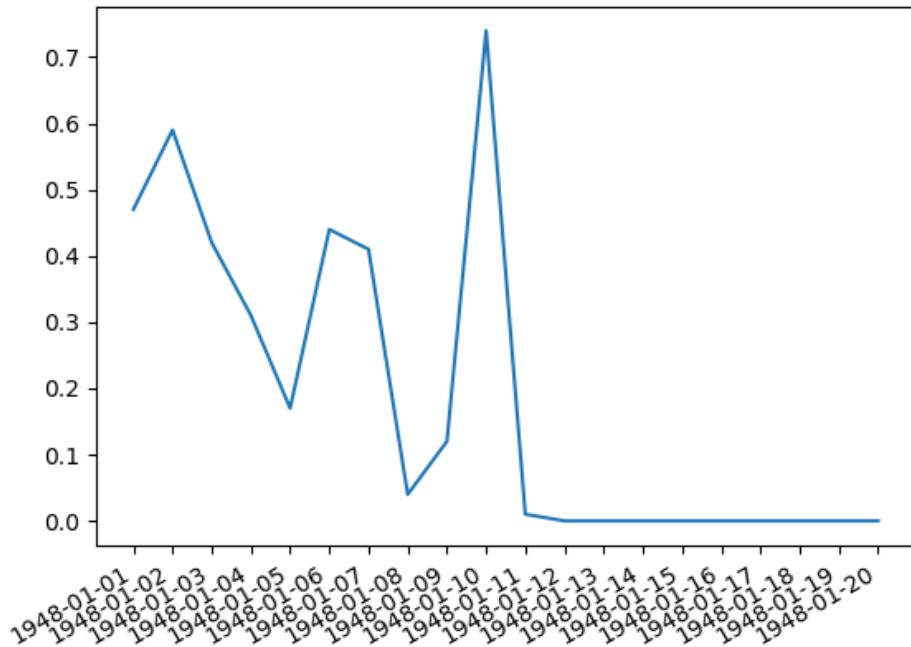
There's another option for rotating and fixing dates in Matplotlib, which is even easier than the previous methods - `fig.autofmt_xdate()`. This function can be used either as `fig.autofmt_xdate()` or `fig.autofmt_ydate()` for the two different axes. It was created *specifically* for auto-formatting dates, since they're a common nuisance.

Let's take a look at how we can use it on the [Seattle Weather Dataset¹⁷](https://www.kaggle.com/ratman/did-it-rain-in-seattle-19482017), providing the ['DATE'] and ['PRCP'] features as the X and Y values:

¹⁷<https://www.kaggle.com/ratman/did-it-rain-in-seattle-19482017>

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 weather_data = pd.read_csv("seattleWeather.csv")
5
6 fig = plt.figure()
7 plt.plot(weather_data['DATE'], weather_data['PRCP'])
8 fig.autofmt_xdate()
9 plt.show()
```

This results in:



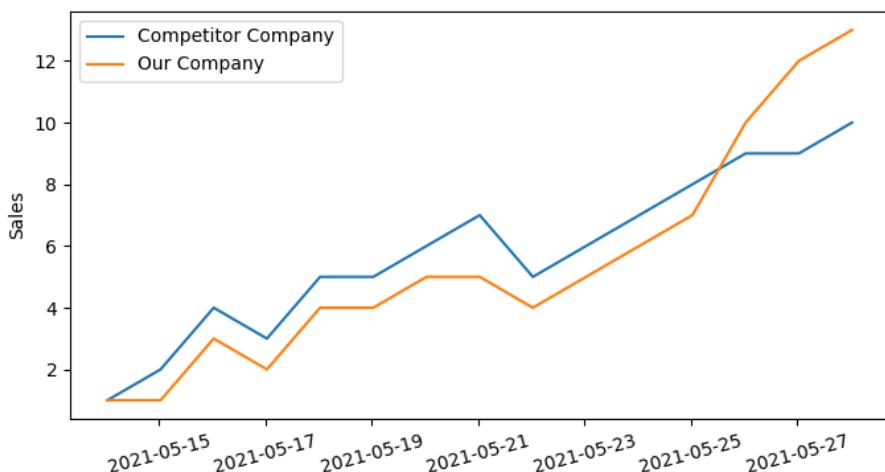
The dates are formatted automatically so that they both fit the X-axis, and have some padding between them.

Draw Vertical Lines on Plot

Similar to how we might want to point out a certain data point, using an annotation or by zooming in (limiting the Axis range), there are situations where we might want to corner off a *segment* of data. This is typically done when we'd like to point out, say, the results on a certain event.

For example, let's imagine a company making a report on their sales:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 from datetime import datetime
4
5 fig, ax = plt.subplots(figsize=(8, 4))
6
7 dates = pd.date_range(datetime.today(), periods=15).tolist()
8 y_1 = [1, 2, 4, 3, 5, 5, 6, 7, 5, 6, 7, 8, 9, 9, 10]
9 y_2 = [1, 1, 3, 2, 4, 4, 5, 5, 4, 5, 6, 7, 10, 12, 13]
10
11 ax.plot(dates, y_1, label='Competitor Company')
12 ax.plot(dates, y_2, label='Our Company')
13
14 plt.xticks(rotation=15)
15
16 plt.ylabel('Sales')
17 plt.legend()
18 plt.show()
```



They have data on their competitor, and at one point - pass the competitor's sales. To highlight this, they might put an annotation at the spot, marking it as an important one. Though, given their general trend of growth - they feel confident that it'll only get better after this point.

Instead, they draw a vertical line, making a clear distinction between the “older days”, and “new days”.

There are two ways we can draw lines, using the `vlines()` or `axvline()` functions of the PyPlot instance. Naturally, you can also call these methods on the `Axes` object.

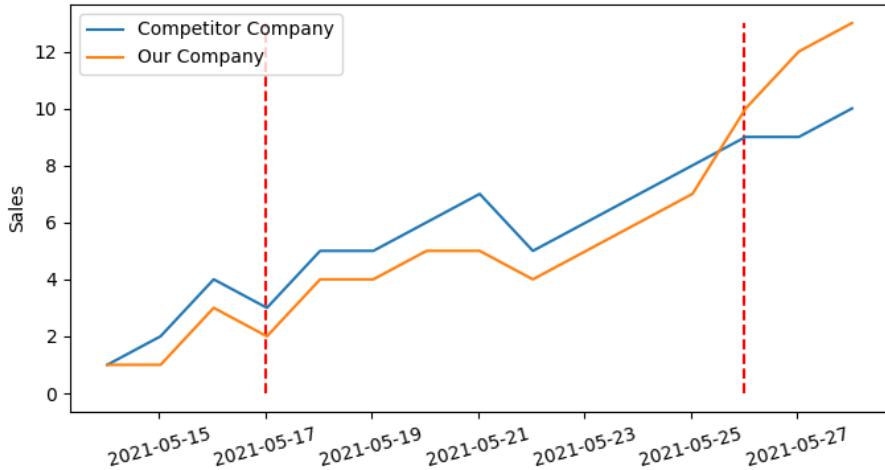
Drawing Vertical Lines with `PyPlot.vlines()`

Let's start off with the `vlines()` function:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 from datetime import datetime
4
5 fig, ax = plt.subplots(figsize=(8, 4))
6
7 dates = pd.date_range(datetime.today(), periods=15).tolist()
8 y_1 = [1, 2, 4, 3, 5, 5, 6, 7, 5, 6, 7, 8, 9, 9, 10]
9 y_2 = [1, 1, 3, 2, 4, 4, 5, 5, 4, 5, 6, 7, 10, 12, 13]
10
11 ax.plot(dates, y_1, label='Competitor Company')
12 ax.plot(dates, y_2, label='Our Company')
13
14 ax.vlines(['2021-05-17', '2021-05-26'], 0, 15, linestyles='dashed', colors='red')
15
16 plt.xticks(rotation=15)
17
18 plt.ylabel('Sales')
19 plt.legend()
20 plt.show()
```

The `vlines()` function accepts a few arguments - a scalar, or 1D array of X-values that you'd like to draw lines on. We've supplied `['2021-05-17', '2021-05-26']`, marking two points (dates), though you can go from `0..n` points here. Then, the `ymin` and `ymax` arguments - these are the *starting and ending points* of the lines. We've set them to be from `0` to `max(y_2)`, since we don't want the lines to go higher or lower than the maximum value on the plot. Then, you can set styles, such as `linestyles` or `colors`, which accept the typical Matplotlib styling options. We'll go into the nitty gritties of styling options like these in *Chapter 7 - Advanced Matplotlib Customization*.

Running this code will result in:



We've got two vertical lines, which are dashed, in red color, at the 2021-05-17 and 2021-05-26 points on the X-axis.

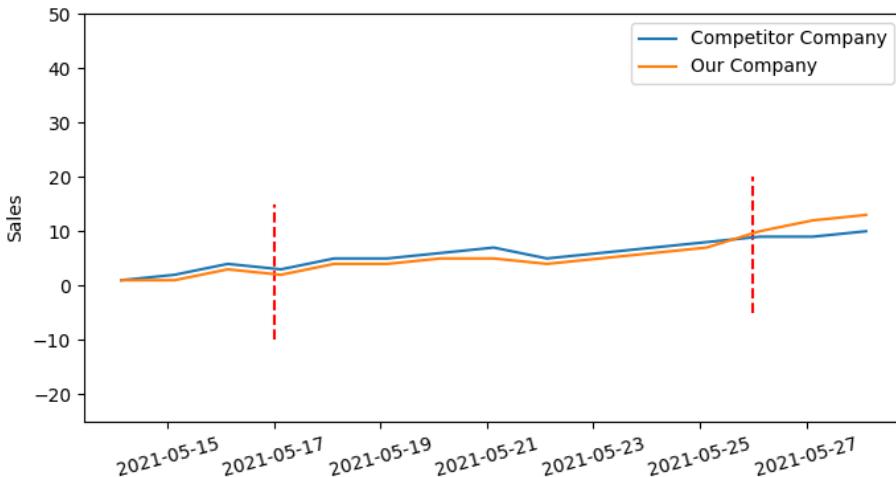
This function allows us to set the `ymin` and `ymax` in concrete values, while `axvline()` lets us choose the height percentage-wise, or we simply let it plot from the bottom to the top by default.

Let's change the range of our Y-axis, to include the view from -25 to 50, instead of 0 and 13. Our data is still in the same range as before so we'll have a view from a different perspective:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 from datetime import datetime
4
5 fig, ax = plt.subplots(figsize=(8, 4))
6
7 dates = pd.date_range(datetime.today(), periods=15).tolist()
8 y_1 = [1, 2, 4, 3, 5, 5, 6, 7, 5, 6, 7, 8, 9, 9, 10]
9 y_2 = [1, 1, 3, 2, 4, 4, 5, 5, 4, 5, 6, 7, 10, 12, 13]
10
11 ax.plot(dates, y_1, label='Competitor Company')
12 ax.plot(dates, y_2, label='Our Company')
13
14 ax.set_ylim(-25, 50)
15 ax.vlines(['2021-05-17'], -10, 15, linestyles='dashed', colors='red')
16 ax.vlines(['2021-05-26'], -5, 20, linestyles='dashed', colors='red')
```

```
17 plt.xticks(rotation=15)
18 plt.ylabel('Sales')
19 plt.legend()
20 plt.show()
```

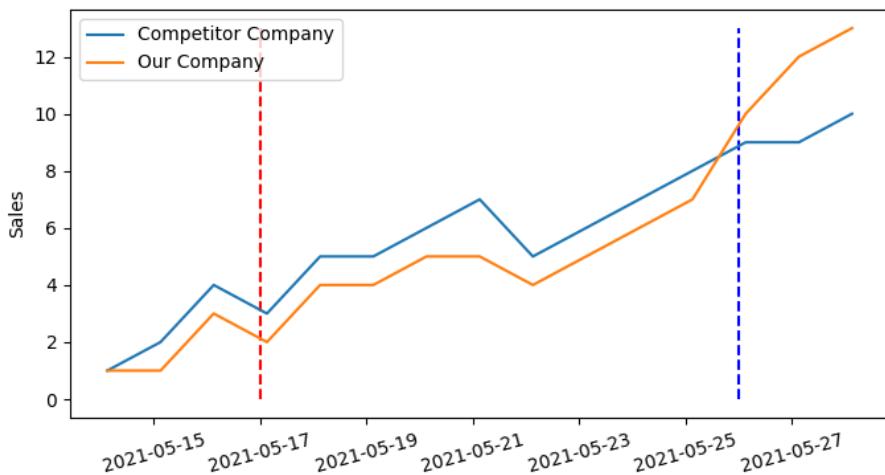
Here, we've called `vlines()` twice, providing a different `ymin` and `ymax` for each line. Since the data is generally trending upwards - we aligned the vertical lines this way:



Additionally, you can set *multiple* colors for these lines, by passing in a list of colors instead of just one:

```
1 ax.vlines(['2021-05-17', '2021-05-26'], 0, max(y_2), linestyles='dashed', colors=['red', \
2 'blue'])
```

Removing the `set_ylim()` call, and running the script - we're now greeted with the all-too-familiar plot, with the sequence of colors mapped to the vertical lines in order:



Drawing Vertical Lines with `PyPlot.axvline()`

Now, let's take a look at the alternative `axvline()` function. As the name implies - it's used to draw *a line*, not *lines*. Namely, `axvline()` accepts a single X-axis variable to draw a line on. When working with strings as dates - there might be a mismatch in the data type. Specifically, we're working with `<class 'pandas._libs.tslibs.timestamps.Timestamp'>` objects - not `datetime` objects here.

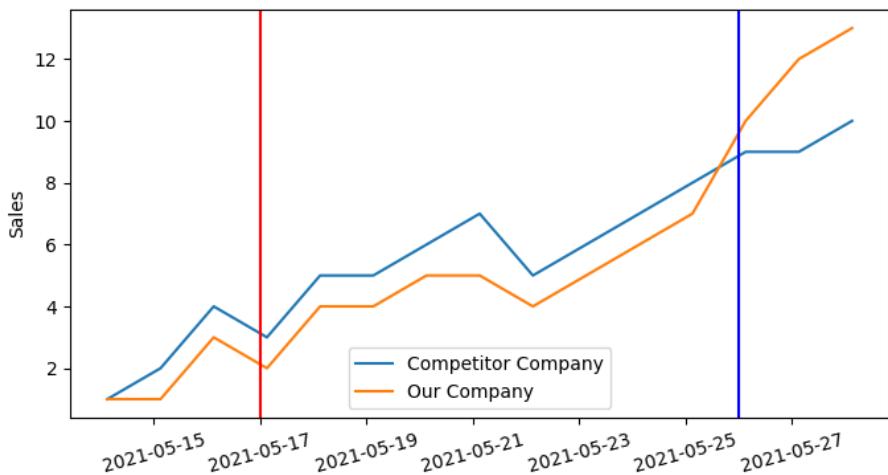
Matplotlib can't convert between these automatically in this case (even though it has no difficulty using their string value as labels). We can use Pandas' `to_datetime()` function to convert a *string* to another `<class 'pandas._libs.tslibs.timestamps.Timestamp'>` explicitly, which Matplotlib gladly uses as our X-axis point to draw a vertical line on. If you're not working with Pandas' `Timestamp`. Working with different representations of `datetime` will inevitably cause *some* conversion errors like these - so keep that in mind as you're working on visualizing data.

Alternatively, you can *put the `Timestamp` in a list*, which resolves this conversion issue:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 from datetime import datetime
4
5 fig, ax = plt.subplots(figsize=(8, 4))
6
7 dates = pd.date_range(datetime.today(), periods=15).tolist()
8 y_1 = [1, 2, 4, 3, 5, 5, 6, 7, 5, 6, 7, 8, 9, 9, 10]
9 y_2 = [1, 1, 3, 2, 4, 4, 5, 5, 4, 5, 6, 7, 10, 12, 13]
10
11 ax.plot(dates, y_1, label='Competitor Company')
12 ax.plot(dates, y_2, label='Our Company')
13
14 ax.axvline(pd.to_datetime('2021-05-17'), color='red')
15 ax.axvline(pd.to_datetime('2021-05-26'), color='blue')
16 # OR
17 ax.axvline(['2021-05-17'], color='red')
18 ax.axvline(['2021-05-26'], color='blue')
19
20 plt.xticks(rotation=15)
21
22 plt.ylabel('Sales')
23 plt.legend()
24 plt.show()
```

Being able to only plot on a single point at a time means that if we want to plot on multiple points, such as 2021-05-17 and 2021-05-26, we'll have to call the function multiple times.

It also doesn't really let us specify the `linestyle` like `vlines()` let us, though, it doesn't require the `ymin` and `ymax` arguments by default. If you omit them, like we have, they'll simply be from the top to the bottom of the Axes:

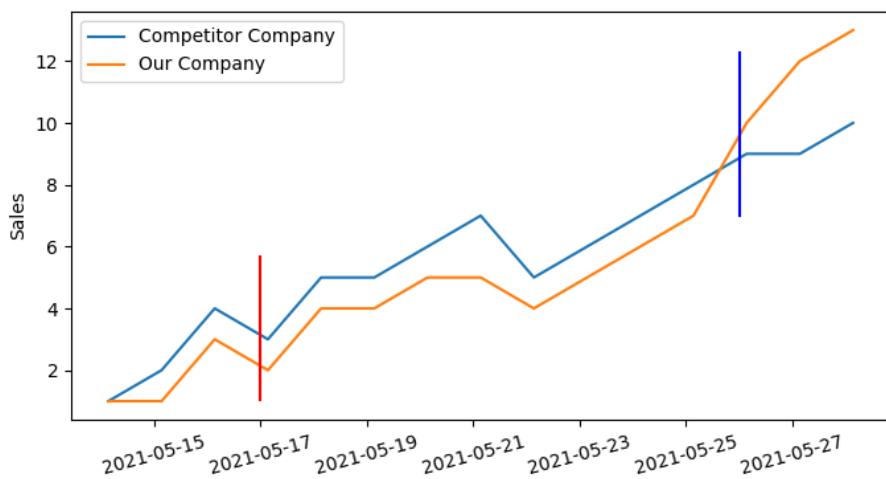


However, you can change the height - this time around though, you'll change the height in terms of percentages. These percentages take the top and bottom of the Axes into consideration so 0% will be at the very bottom, while 100% will be at the very top.

Let's draw the first line as spanning from 5% to 40%, and the second line as spanning from 50% to 90%:

```
1 ax.axvline(pd.to_datetime('2021-05-17'), 0.4, 0.05, color='red')
2 ax.axvline(pd.to_datetime('2021-05-26'), 0.9, 0.5, color='blue')
```

This produces:



Chapter 6. - Data Visualization with Matplotlib

In *Chapter 4. - Getting Started with Matplotlib* we got acquainted with the anatomy of Matplotlib plots, how to utilize the generic `plot()` function with simple data and got familiar with the APIs we can use to work with Matplotlib.

In *Chapter 5. - Basic Matplotlib Customization*, we explored some customization operations which are commonly used, such as changing the figure and font size, setting axis ranges, changing tick frequency, adding legends, and plotting vertical lines. We've also jumped into Matplotlib text, and how it can be used, including how to add and style annotations to point out certain parts of your plots.

These operations are a primer on what we'll be generally using, but aren't the only customization options you'll *ever* be using. While we'll explore those in the next chapter - these should be more than enough to get you through most of your plotting needs.

In this chapter, armed with knowledge of how Matplotlib works and how we can tweak plots - let's jump into *Data Visualization with Matplotlib*.

We'll be covering some of the most commonly used plot types, such as *Scatter Plots*, *Bar Plots* and *Box Plots*, but we'll also be utilizing some more rarely used plot types such as *Ridge Plots* and the story of how they were conceived. Additionally, we'll be creating a *custom plot* such as a *Joint Plot* which isn't built into the Matplotlib library, but is a popularized plot type from another Data Visualization library - Seaborn.

Note: For each plot type, we'll be using a different dataset, which necessitates different pre-processing. In some cases, the data will be very fit for the plot type we're using already - since we'll be choosing the right plot type for the job. In other cases, though, we'll have to perform pre-processing with Pandas.

All of the datasets will be available publicly, and downloadable for your convenience and the links to each will be provided in the footnotes.

Finally, we'll explore a fun, large dataset containing EEG readings (brainwaves) of two groups when shown different stimuli. We'll be creating Surface Plots of neuron group activations, as well as recreate the face of an EEG viewer application.

Line Plot

Let's first take a look at *Line Plots* - one of the most basic types of plots.

Line Plots display numerical values one on one axis, and categorical values on the other. They can typically be used in much the same way Bar Plots can be used, though, they're more commonly used to keep track of changes over time.

We've already been using Line Plots so far, since they're very intuitive and simple.

Plotting a Line Plot

To plot a line plot in Matplotlib, you use the generic `plot()` function from the PyPlot instance, as we've seen in the previous two chapters. There's no specific `lineplot()` function - the generic one automatically plots using lines *or* markers depending on the input data.

Let's ditch using lists and import a dataset to work with for this. We'll use the [Country, Regional and World GDP](#)¹⁸dataset from Kaggle. It contains data on country, regional and world GDP data, in US dollars (\$). Let's import the CSV file and take a look at what it looks like:

```
1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('gdp_csv.csv')
5 print(df)
```

This results in:

¹⁸<https://www.kaggle.com/tunguz/country-regional-and-world-gdp>

```

1      Country Name Country Code Year      Value
2  0      Arab World      ARB 1968  2.576068e+10
3  1      Arab World      ARB 1969  2.843420e+10
4  2      Arab World      ARB 1970  3.138550e+10
5  3      Arab World      ARB 1971  3.642691e+10
6  4      Arab World      ARB 1972  4.331606e+10
7  ...
8  11502    Zimbabwe      ZWE 2012  1.424249e+10
9  11503    Zimbabwe      ZWE 2013  1.545177e+10
10 11504    Zimbabwe      ZWE 2014  1.589105e+10
11 11505    Zimbabwe      ZWE 2015  1.630467e+10
12 11506    Zimbabwe      ZWE 2016  1.661996e+10

```

The dataset has 11507 entries, though, their categories are totally mixed. There are a lot of values such as `Arab World` which refers to several countries, just like `European Union` refers to multiple countries. We want to create a couple of smaller pieces of data for us to work with. Say we want to visualize the GDP of the EU and the US, year over year. We'll filter this dataset to find all the relevant entries, and then plot using those:

```

1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('gdp_csv.csv')
5 df_eu = df.loc[df['Country Name'] == 'European Union']
6 df_na = df.loc[df['Country Name'] == 'North America']
7 df_sa = df.loc[df['Country Name'] == 'South Asia']
8 df_ea = df.loc[df['Country Name'] == 'East Asia & Pacific']
9
10 print(df_eu.head())
11 print(df_na.head())
12 print(df_sa.head())
13 print(df_ea.head())

```

Now, 'Country Name' might be a bit misleading here, since these are not *country names*. One refers to a select set of countries within the *European Union*, one refers to the continent of *North America* and all the countries within it, one refers to the region of *South Asia* and the fourth one refers to *East Asia and the Pacific*, but this is how the dataset was formed.

In any case - printing the heads of the four new DataFrames gives us a clue as to whether we've filtered these correctly:

```

1      Country Name Country Code Year      Value
2  531 European Union          EUU 1960  3.589415e+11
3  532 European Union          EUU 1961  3.907915e+11
4  533 European Union          EUU 1962  4.269104e+11
5  534 European Union          EUU 1963  4.702995e+11
6  535 European Union          EUU 1964  5.212003e+11
7      Country Name Country Code Year      Value
8  1543 North America         NAC 1960  5.844779e+11
9  1544 North America         NAC 1961  6.041572e+11
10 1545 North America         NAC 1962  6.471730e+11
11 1546 North America         NAC 1963  6.833535e+11
12 1547 North America         NAC 1964  7.347905e+11
13      Country Name Country Code Year      Value
14 1907 South Asia            SAS 1960  4.653588e+10
15 1908 South Asia            SAS 1961  4.963927e+10
16 1909 South Asia            SAS 1962  5.296996e+10
17 1910 South Asia            SAS 1963  5.956412e+10
18 1911 South Asia            SAS 1964  6.820724e+10
19      Country Name Country Code Year      Value
20 190 East Asia & Pacific    EAS 1960  1.536120e+11
21 191 East Asia & Pacific    EAS 1961  1.540584e+11
22 192 East Asia & Pacific    EAS 1962  1.576342e+11
23 193 East Asia & Pacific    EAS 1963  1.759024e+11
24 194 East Asia & Pacific    EAS 1964  2.020997e+11

```

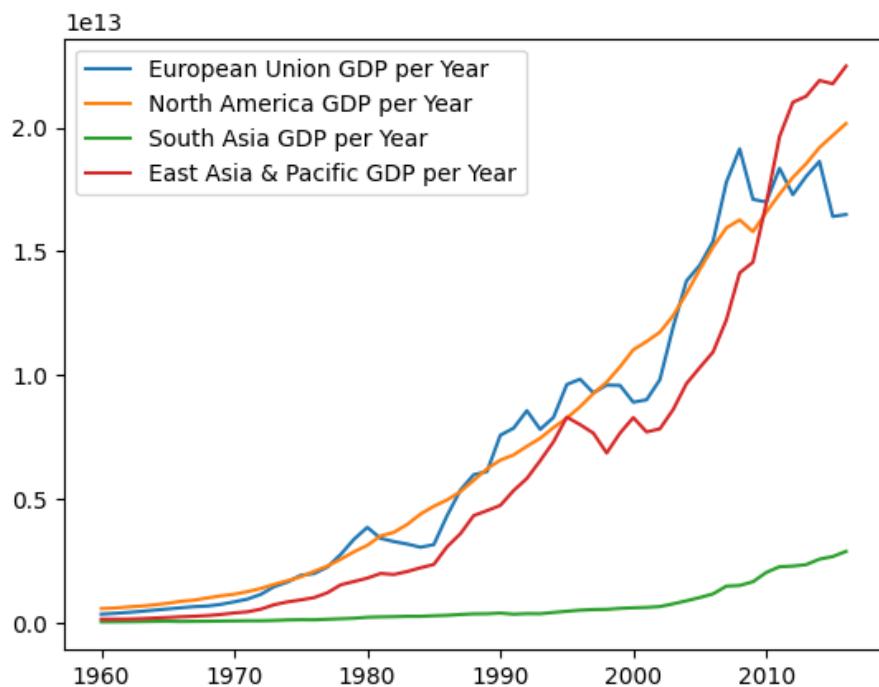
Looks good! Let's create a line plot for each of these. We'll be using the `Value` for our Y-feature, and the `Year` as our X-feature:

```

1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('gdp_csv.csv')
5 df_eu = df.loc[df['Country Name'] == 'European Union']
6 df_na = df.loc[df['Country Name'] == 'North America']
7 df_sa = df.loc[df['Country Name'] == 'South Asia']
8 df_ea = df.loc[df['Country Name'] == 'East Asia & Pacific']
9
10 fig, ax = plt.subplots()
11 ax.plot(df_eu['Year'], df_eu['Value'], label = 'European Union GDP per Year')
12 ax.plot(df_na['Year'], df_na['Value'], label = 'North America GDP per Year')
13 ax.plot(df_sa['Year'], df_sa['Value'], label = 'South Asia GDP per Year')
14 ax.plot(df_ea['Year'], df_ea['Value'], label = 'East Asia & Pacific GDP per Year')
15
16 ax.legend()
17 plt.show()

```

This results in:



Our scale is 10^{13} , as denoted by $1e13$ at the top left corner.

Plotting a Line Plot Logarithmically

When dealing with datasets that have progressively larger numbers, and especially if their distribution leans towards being exponential, it's common to plot a line plot on a logarithmic scale.

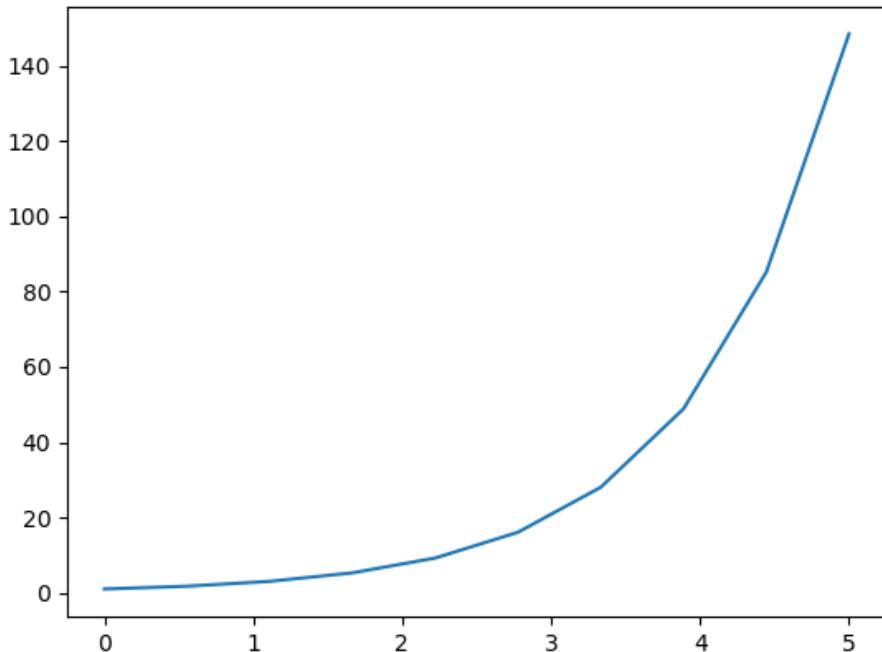
Instead of the Y-axis ticks being uniformly linear, this will change each interval/tick to be exponentially larger than the last one.

This results in exponential functions being plotted essentially as straight lines. When dealing with this type of data, it's hard to wrap your mind around exponential numbers, and you can make it much more easy to digest by plotting the data logarithmically.

Let's use Numpy to generate an exponential function and plot it linearly, like we did before:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 5, 10) # [0, 0.55, 1.11, 1.66, 2.22, 2.77, 3.33, 3.88, 4.44, 5]
5 y = np.exp(x) # [1, 1.74, 3.03, 5.29, 9.22, 16.08, 28.03, 48.85, 85.15, 148.41]
6
7 plt.plot(x, y)
8 plt.show()
```

This creates an array, that's 10 in length, and contains values between $0..5$. We've then used the `exp()` function from Numpy to calculate the exponential values of these elements, resulting in an exponential function on a linear scale:



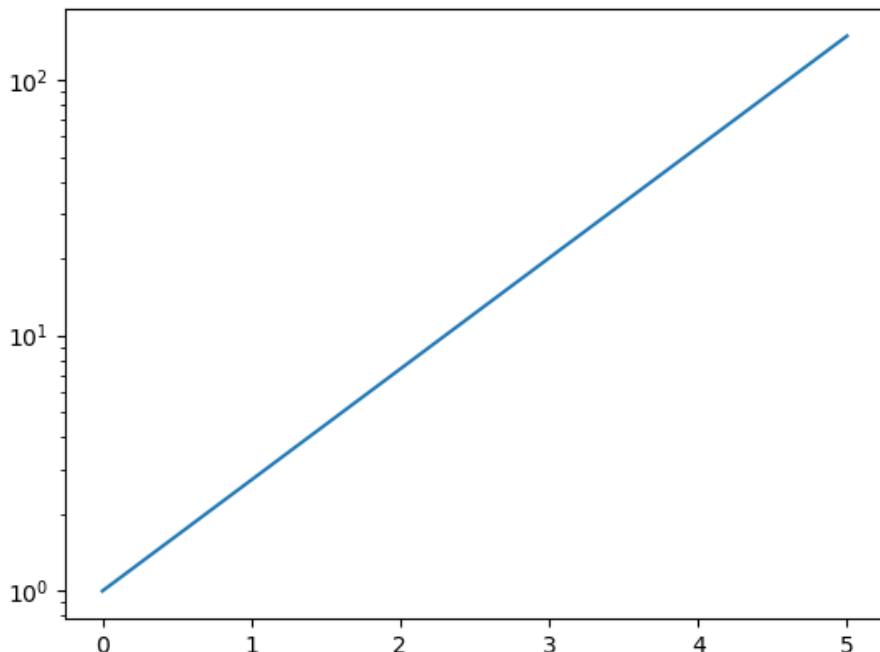
This sort of function, although simple, is hard for humans to conceptualize, and small changes can easily go unnoticed, when dealing with large datasets. Now, let's change

the scale of the Y-axis to logarithmic:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 x = np.linspace(0, 5, 10)
5 y = np.exp(x)
6
7 plt.yscale('log')
8 plt.plot(x, y)
9 plt.show()
```

Using the PyPlot instance, `plt`, we can set the scale of the X and Y axes. Here, we've set the Y-Axis on a logarithmic scale, via the `yscale()` function. Here, we could've also used `linear`, `log`, `logit` and `symlog`. The default is `linear`.

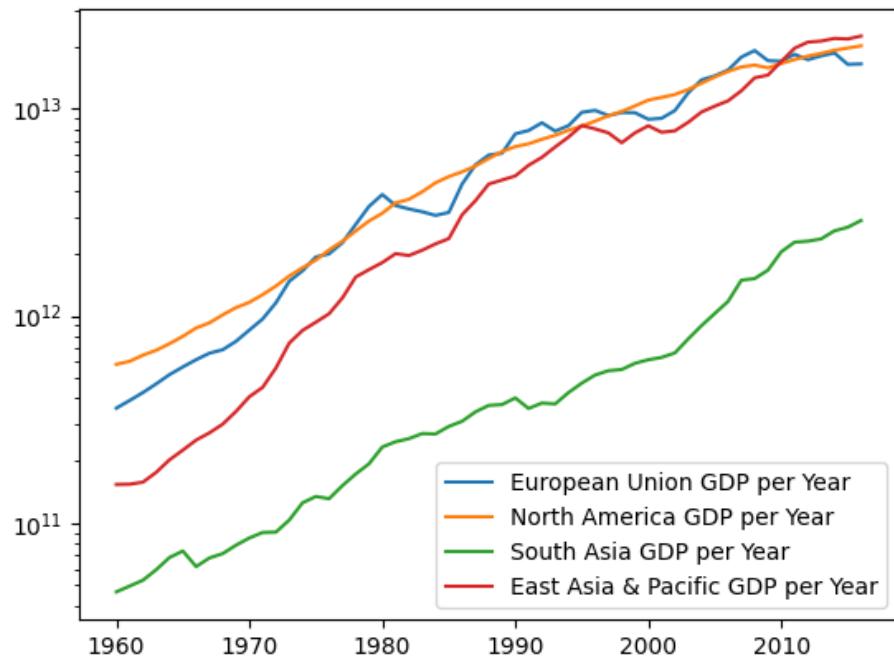
Running this code results in:



Now, applying this to our *GDP Dataset*, we have seen growth in certain regions that resembles exponential. Let's visualize these on a logarithmic scale:

```
1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('gdp_csv.csv')
5 # Filter DataFrame into smaller sections...
6
7 fig, ax = plt.subplots()
8 ax.plot(df_eu['Year'], df_eu['Value'], label = 'European Union GDP per Year')
9 ax.plot(df_na['Year'], df_na['Value'], label = 'North America GDP per Year')
10 ax.plot(df_sa['Year'], df_sa['Value'], label = 'South Asia GDP per Year')
11 ax.plot(df_ea['Year'], df_ea['Value'], label = 'East Asia & Pacific GDP per Year')
12
13 plt.yscale('log')
14 ax.legend()
15 plt.show()
```

The growth pattern isn't *quite* exponential, but the growth of *East Asia & Pacific* does resemble exponential growth more than, say, the *European Union*. Additionally, although *South Asia* appeared to be way below other regions in terms of GDP - on this scale, we can see that the *growth rate* is actually also nearing an exponential, which means it's actually much closer time-wise than it might appear on first sight:



A few more doublings down the line, and it'll reach the same values as the other lines in this plot.

Customizing Line Plots in Matplotlib

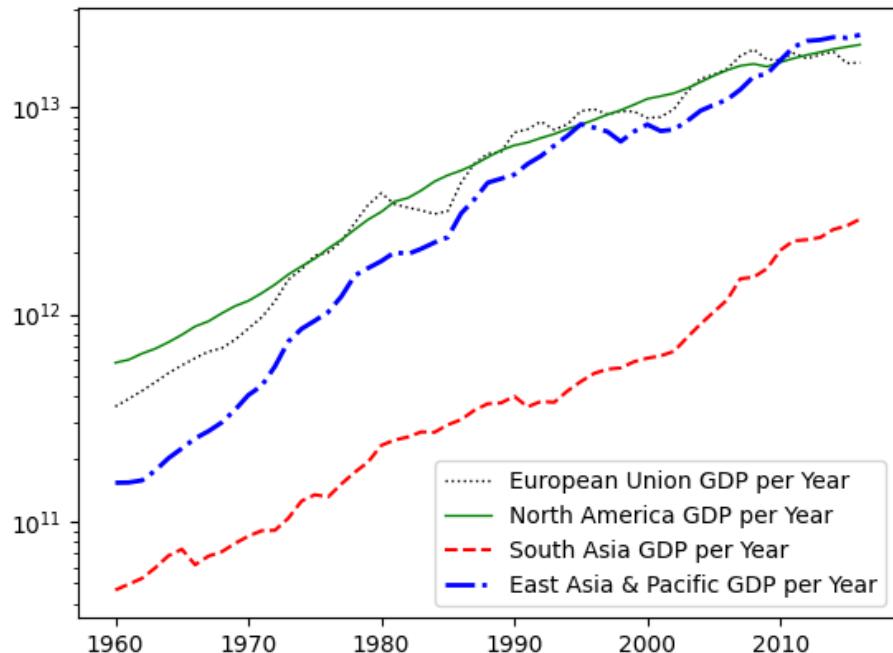
You can easily customize line plots by passing in arguments to the `plot()` function. These will typically be arguments such as `linewidth`, `linestyle` or `color`:

```

1 # Labels shortened for brevity
2 ax.plot(df_eu['Year'], df_eu['Value'], label = '#', linestyle='dotted', color='k', linewidth=1)
3 ax.plot(df_na['Year'], df_na['Value'], label = '#', linestyle='solid', color='g', linewidth=1)
4 ax.plot(df_sa['Year'], df_sa['Value'], label = '#', linestyle='dashed', color='r', linewidth=1.5)
5 ax.plot(df_ea['Year'], df_ea['Value'], label = '#', linestyle='dashdot', color='b', linewidth=2)

```

There are four basic types of linestyles - dotted, solid, dashed and dashdot. There are [other types¹⁹](#) as well though. The color argument can be any valid Matplotlib color and the linewidth can be a floating-point integer, denoting the width of each line:

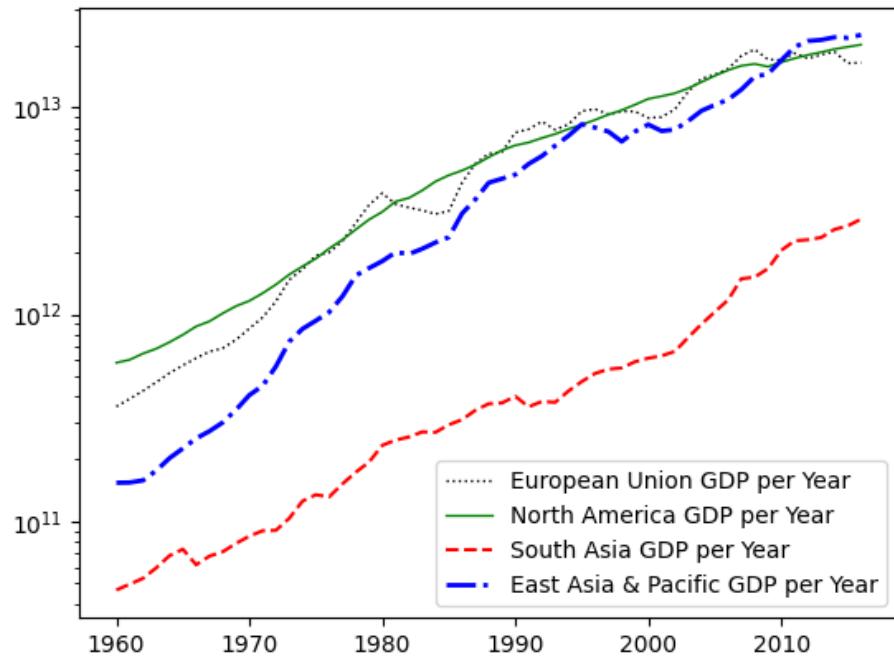


Instead of the dashed, solid, dotted and dashdot values - we could've also used special characters such as : , - , -- and - . :

¹⁹https://matplotlib.org/stable/gallery/lines_bars_and_markers/linestyles.html

```
1 # Labels shortened for brevity
2 ax.plot(df_eu['Year'], df_eu['Value'], label='*', linestyle=':', color='k', linewidth=1)
3 ax.plot(df_na['Year'], df_na['Value'], label='*', linestyle='-', color='g', linewidth=1)
4 ax.plot(df_sa['Year'], df_sa['Value'], label='*', linestyle='--', color='r', linewidth=1.\n5)
6 ax.plot(df_ea['Year'], df_ea['Value'], label='*', linestyle='-.', color='b', linewidth=2)
```

This results in the same plot, though, this representation of the linestyles might be a bit more intuitive:



Plot Multiple Line Plots with Different Scales

Sometimes, you might have two datasets, fit for line plots, but their values are significantly different, making it hard to compare both lines. For example, if `line_1` had an exponentially increasing sequence of numbers, while `line_2` had a linearly

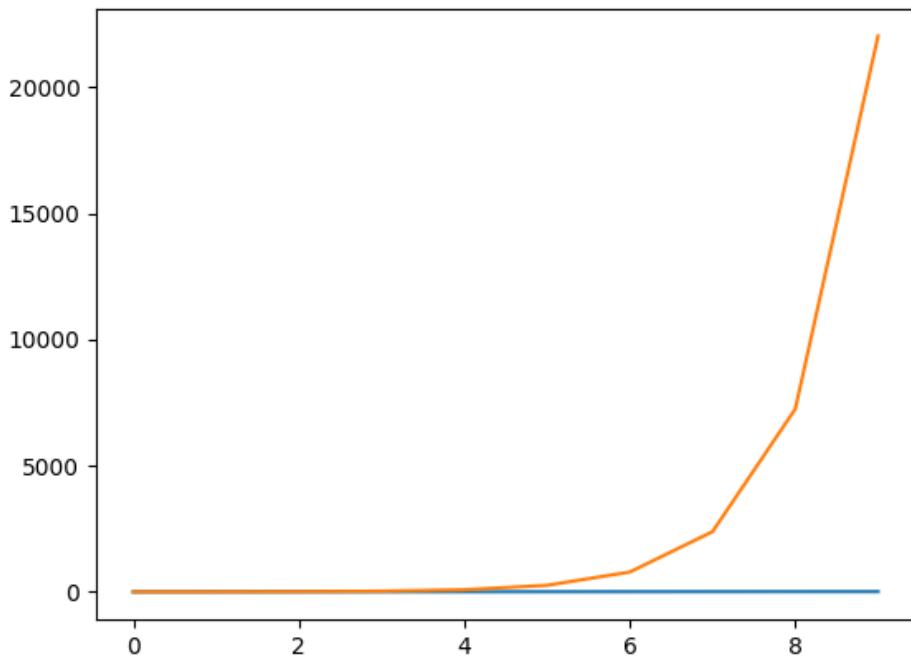
increasing sequence - surely and quickly enough, `line_1` would have values so much larger than `line_2`, that the latter fades out of view.

`line_2` goes 1, 2, 3. While `line_1` goes 1, 2, 4. Doesn't seem all to different in the beginning, but on step 30 - `line_2` is at 30, while `line_1` is at 1,073,741,824.

Let's use Numpy to make an exponentially increasing sequence of numbers, and plot it next to another line on the same Axes, linearly:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 linear_sequence = np.linspace(0, 10, 10)
5 # [0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 10]
6 exponential_sequence = np.exp(linear_sequence)
7 # [1.00e+00, 3.03e+00, 9.22e+00, 2.80e+01, 8.51e+01,
8 # 2.58e+02, 7.85e+02, 2.38e+03, 7.25e+03, 2.20e+04]
9
10 fig, ax = plt.subplots()
11
12 ax.plot(linear_sequence)
13 ax.plot(exponential_sequence)
14 plt.show()
```

Running this code results in:



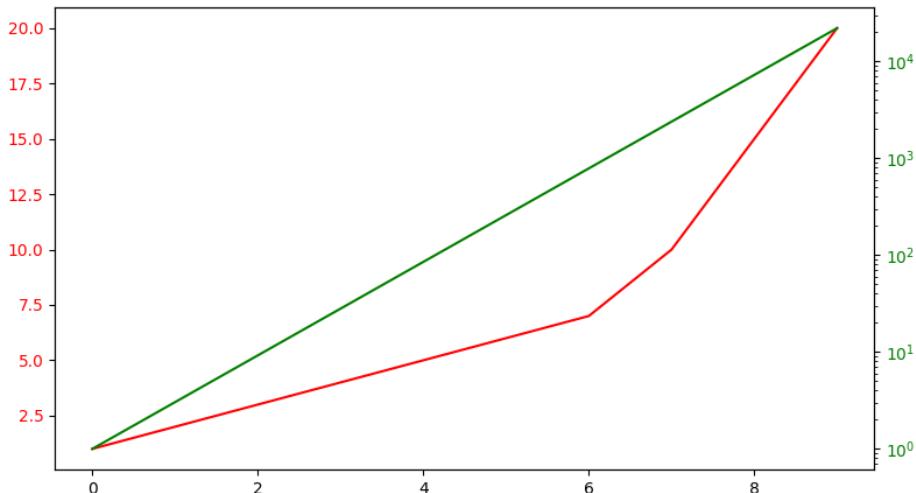
The exponential growth in the `exponential_sequence` goes out of proportion very fast, and it looks like there's absolutely no difference in Y-axis values of the `linear_sequence`, since it's so minuscule relative to the exponential trend of the other sequence.

Now, let's plot the `exponential_sequence` on a logarithmic scale, which will produce a visually straight line, since the Y-scale will exponentially increase. If we plot it on a logarithmic scale, and the `linear_sequence` just increases by the same constant, we'll have two overlapping lines and we will only be able to see the one plotted after the first.

Let's change up the `linear_sequence` a bit to make it observable once we plot both:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 # Sequences
5 linear_sequence = [1, 2, 3, 4, 5, 6, 7, 10, 15, 20]
6 exponential_sequence = np.exp(np.linspace(0, 10, 10))
7
8 fig, ax = plt.subplots()
9
10 # Plot linear sequence, and set tick labels to the same color
11 ax.plot(linear_sequence, color='red')
12 ax.tick_params(axis='y', labelcolor='red')
13
14 # Generate a new Axes instance, on the twin-X axes (same position)
15 ax2 = ax.twinx()
16
17 # Plot exponential sequence, set scale to logarithmic and change tick color
18 ax2.plot(exponential_sequence, color='green')
19 ax2.set_yscale('log')
20 ax2.tick_params(axis='y', labelcolor='green')
21
22 plt.show()
```

One Axes has one scale, so we create a new one, in the same position as the first one, and set its scale to a logarithmic one, and plot the exponential sequence. This results in:



We've also changed the tick label colors to match the color of the line plots themselves,

otherwise, it'd be hard to distinguish which line is on which scale.

Plot Multiple Line Plots with Multiple Y-Axes

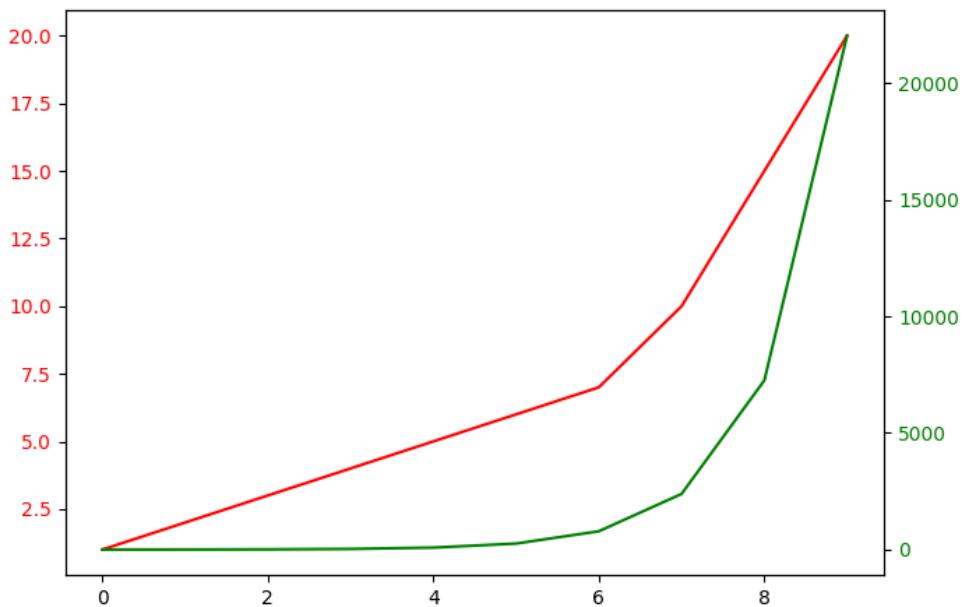
Finally, we can apply the same scale (linear, logarithmic, etc), but have different values on the Y-axis of each line plot. This is achieved through having multiple Y-axis, on different Axes objects, in the same position.

For example, the `linear_sequence` won't go above 20 on the Y-axis, while the `exponential_sequence` will go up to 20000. We can plot them both *linearly*, simply by plotting them on different Axes objects, in the same position, each of which set the Y-axis ticks automatically to accommodate for the data we're feeding in:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 # Sequences
5 linear_sequence = [1, 2, 3, 4, 5, 6, 7, 10, 15, 20]
6 exponential_sequence = np.exp(np.linspace(0, 10, 10))
7
8 fig, ax = plt.subplots()
9
10 # Plot linear sequence, and set tick labels to the same color
11 ax.plot(linear_sequence, color='red')
12 ax.tick_params(axis='y', labelcolor='red')
13
14 # Generate a new Axes instance, on the twin-X axes (same position)
15 ax2 = ax.twinx()
16
17 # Plot exponential sequence, set scale to logarithmic and change tick color
18 ax2.plot(exponential_sequence, color='green')
19 ax2.tick_params(axis='y', labelcolor='green')
20
21 plt.show()
```

We've again, created another Axes in the same position as the first one, so we can plot on the same place in the Figure but different Axes objects, which allows us to set values for each Y-axis individually.

Without setting the Y-scale to logarithmic this time, both will be plotted linearly:



Bar Plot

Another fairly simple plot type are Bar Plots.

Bar Plots display numerical quantities on one axis and categorical variables on the other, letting you see how many occurrences there are for the different categories. On the other hand, you don't have to plot the *number of occurrences* - you can use any *estimator function* to aggregate data.

Bar charts can be used for visualizing a time series, as well as just categorical data.

Plotting a Bar Plot

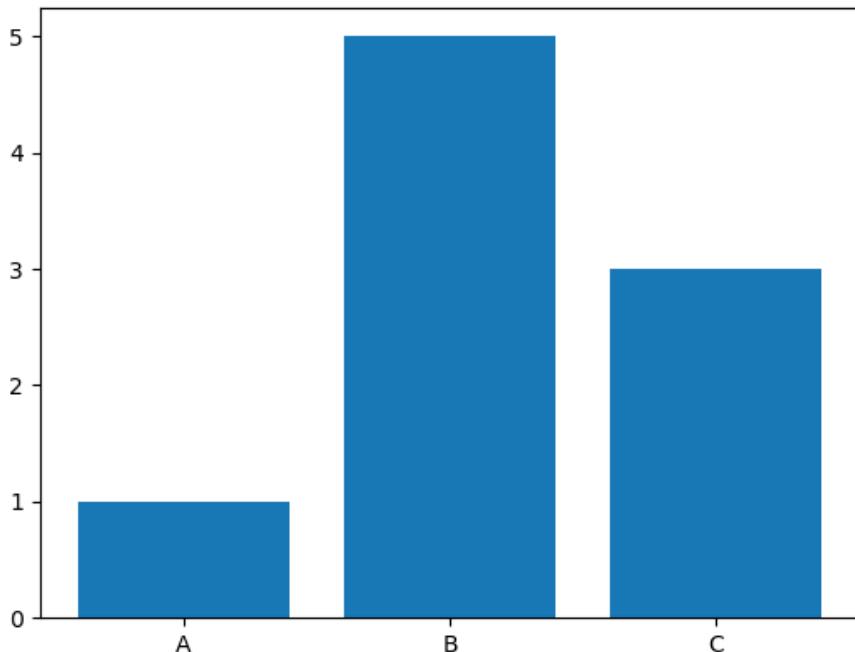
Plotting a Bar Plot in Matplotlib is as easy as calling the `bar()` function on the PyPlot or Axes instance, and passing in the categorical and numerical variables that we'd

like to visualize:

```
1 import matplotlib.pyplot as plt
2
3 x = ['A', 'B', 'C']
4 y = [1, 5, 3]
5
6 plt.bar(x, y)
7 plt.show()
```

Here, we've got a few categorical variables in a list - A, B and C. We've also got a couple of continuous variables in another list - 1, 5 and 3. The relationship between these two is then visualized in a Bar Plot by passing these two lists to `plt.bar()`.

This results in a clean and simple bar graph:



Let's ditch manual lists and find a dataset that we can use to visualize. We'll use

the [World Happiness Report 2021²⁰](#) dataset from Kaggle. It has various “scores” for different factors that might impact human happiness, and we can easily plot their relationships as well as standalone features. The data is aggregated from the [Gallup World Poll²¹](#), which amongst other things tracks other issues such as food access, employment and human well-being.

Let’s import the dataset and take a look at the contents inside:

```

1      Country name  Regional indicator  ...   Explained by: Perceptions of corruption
2  0      Finland      Western Europe  ...
3  1      Denmark      Western Europe  ...
4  2      Switzerland  Western Europe  ...
5  3      Iceland      Western Europe  ...
6  4      Netherlands  Western Europe  ...
7  ...
8  144     Lesotho    Sub-Saharan Africa ...
9  145     Botswana   Sub-Saharan Africa ...
10 146     Rwanda     Sub-Saharan Africa ...
11 147     Zimbabwe   Sub-Saharan Africa ...
12 148 Afghanistan   South Asia ...
13
14 [149 rows x 20 columns]

```

The dataset has 149 rows, with various features measured by Gallup, such as the *Logged GDP per capita*, *Healthy life expectancy*, *Freedom to make life choices*, *Generosity*, *Perceptions of corruption*, and *Social support*.

Each *Country name* belongs to a *Regional indicator*. Let’s go ahead and plot the *Freedom to make life choices* by the *Regional indicator*:

```

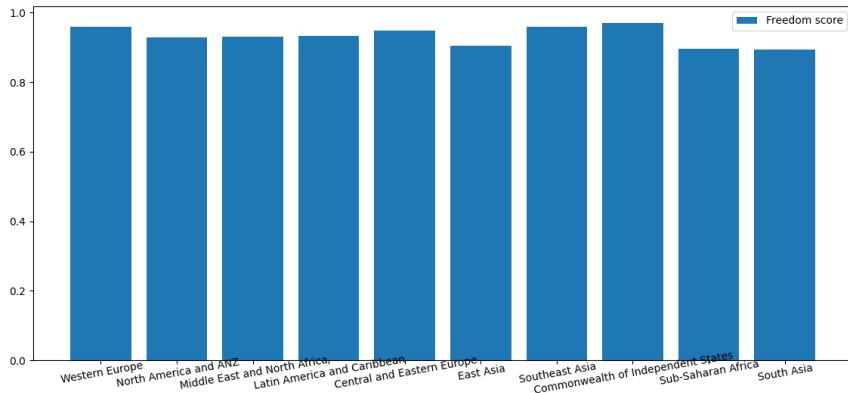
1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('world-happiness-report-2021.csv')
5
6 fig, ax = plt.subplots()
7 ax.bar(df['Regional indicator'], df['Freedom to make life choices'],
8        label='Freedom score')
9 ax.legend()
10
11 plt.xticks(rotation=10, wrap=True)
12 plt.show()

```

Since there are some fairly long regional indicator names, such as “Commonwealth of Independent States”, we’ve decided to *wrap* the text to make it a bit easier to fit:

²⁰<https://www.kaggle.com/ajaypalsinghlo/world-happiness-report-2021>

²¹<https://www.gallup.com/178667/gallup-world-poll-work.aspx>



The freedom score seems to be about equal between the regions, if we go ahead and `describe()` the feature:

```
1 print(df['Freedom to make life choices'].describe())
```

```
1 count      149.000000
2 mean       0.791597
3 std        0.113332
4 min        0.382000
5 25%        0.718000
6 50%        0.804000
7 75%        0.877000
8 max        0.970000
```

This paints a different picture though. Although it visually appears that these are about in the same ballpark - the *mean* of the feature is 0.79. Yet, we don't see any bars around the 0.8 mark. Why is that?

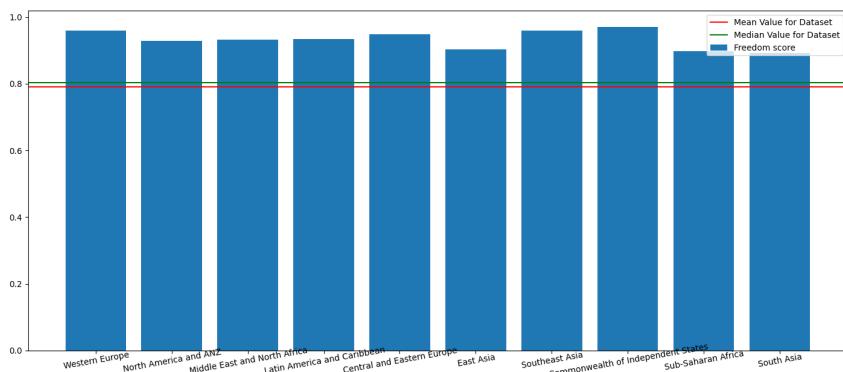
Taking a look at our dataset, we've got a bunch of rows with differing values for "*Regional indicator*". When we plot data like this - we plot the "*Freedom to make life choices*" feature for *each "Regional indicator"*. This means that if "*Western Europe*" has 7 countries - we have 7 *overlaid* bars on top of each other. Visually, since they're flush, we see the highest bar as the top value shown for each region, which seems to be around 0.9 to around 0.95 for most of the regional indicators. There are other bars on *top* of this bar, but since their values are less than the highest bar's value, we simply don't notice them.

What we *want* is to plot the *mean* or *median* value for *each individual region*, and then plot those values as the bars. Let's add the median and mean lines to our plot, to see where they are. Much like we can plot *vertical lines*, we can plot *horizontal lines* as well:

```

1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('world-happiness-report-2021.csv')
5 mean = df['Freedom to make life choices'].mean()
6 median = df['Freedom to make life choices'].median()
7
8 fig, ax = plt.subplots()
9 ax.bar(df['Regional indicator'], df['Freedom to make life choices'],
10        label = 'Freedom score')
11 ax.axhline(mean, 0, 1, color='red', label='Mean Value for Dataset')
12 ax.axhline(median, 0, 1, color='green', label='Median Value for Dataset')
13 ax.legend()
14
15 plt.xticks(rotation=10, wrap=True)
16 plt.show()
```

We've used Pandas to calculate the `mean()` and `median()` values for the two Series objects containing our freedom score data. Then, we can use these values to plot an `axhline()`, giving it a label for the legend:



When calculating the *estimator* for the bars like this - we can use a mean, median, mode or really, any statistical function. In our case, the mean and median aren't that far apart either. We'll go with a *mean*, as this is a fairly common estimator.

We'll group the dataset by the regional indicators, and calculate the *mean* for *each*

group - storing the results in a new DataFrame - df_means. This one will contain the grouped mean values of each region:

```

1 df = pd.read_csv('world-happiness-report-2021.csv')
2 df_means = df.groupby("Regional indicator").mean()
3 print(df_means)
4 print(df_means["Freedom to make life choices"])

```

This results in:

	Ladder score	...	Dystopia + residual
1 Regional indicator		...	
2 Central and Eastern Europe	5.984765	...	2.490000
3 Commonwealth of Independent States	5.467000	...	2.225750
5 East Asia	5.810333	...	2.117833
6 Latin America and Caribbean	5.908050	...	2.733000
7 Middle East and North Africa	5.219765	...	2.210588
8 North America and ANZ	7.128500	...	2.650500
9 South Asia	4.441857	...	1.964000
10 Southeast Asia	5.407556	...	2.033444
11 Sub-Saharan Africa	4.494472	...	2.507722
12 Western Europe	6.914905	...	2.628762
13 Regional indicator		...	
15 Central and Eastern Europe	0.797059		
16 Commonwealth of Independent States	0.816917		
17 East Asia	0.763500		
18 Latin America and Caribbean	0.831750		
19 Middle East and North Africa	0.716471		
20 North America and ANZ	0.898750		
21 South Asia	0.765000		
22 Southeast Asia	0.909000		
23 Sub-Saharan Africa	0.723194		
24 Western Europe	0.858714		

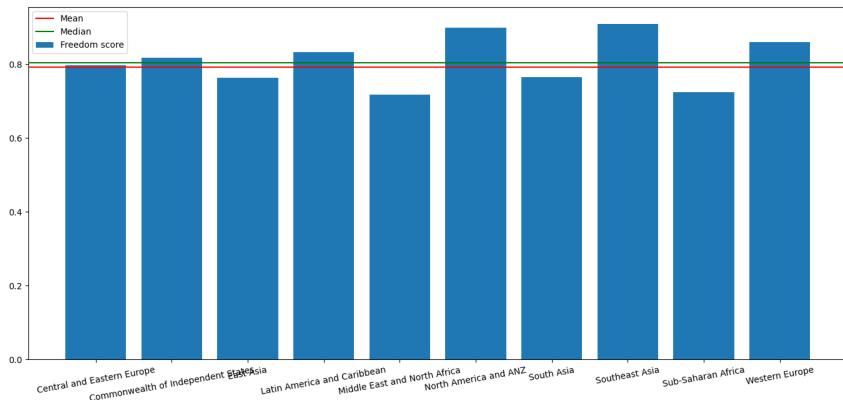
This is the data we *wanted* to plot originally - the aggregated data for each region. Let's use this new dataset to plot the values. This time around, our "Regional indicator" is no longer a *value* - it's our *index*. Instead of providing df['Regional indicator'] to the bar() function as the X-variable, we'll be providing the df_means.index:

```

1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('world-happiness-report-2021.csv')
5
6 df_means = df.groupby("Regional indicator").mean()
7 mean = df['Freedom to make life choices'].mean()
8 median = df['Freedom to make life choices'].median()
9
10 fig, ax = plt.subplots()
11
12 ax.bar(df_means.index, df_means['Freedom to make life choices'],
13         label = 'Freedom score')
14
15 ax.axhline(mean, 0, 1, color='red', label='Mean')
16 ax.axhline(median, 0, 1, color='green', label='Median')
17
18 ax.legend()
19 plt.xticks(rotation=10, clip_on=True)
20
21 plt.show()

```

This results in:



This looks much more accurate. We've got each region's mean plotted as a bar, and we've also got two lines signifying the *overall* mean and median for the entire dataset. Then again, there's another thing to consider - each of these bars is *an aggregate* of the constituent countries. Hypothetically speaking, one country may have the score of 0.0 , while another one may have the score of 1.0 and the *mean* would be 0.5 . The mean would not represent *either* of the countries at all.

This is where *error bars* come into play.

Bar Plot with Error Bars in Matplotlib

When you're plotting mean values of lists, which is a common application for Bar Plots, you'll have some error space. It's very useful to plot error bars to let other observers, and yourself, know how *truthful* these means are and which *deviation* is expected.

There are two estimates you typically use for *error bars*:

- Standard Deviation²²
- Standard Error²³

Standard deviation is pretty common, and we'll be using that to visualize error bars of our *mean values*, to let us know how truthful they are and what sort of deviations we may encounter in *each country's Freedom Score*.

Thankfully, we can easily calculate the standard deviation for a Series or DataFrame using Pandas:

```

1 df = pd.read_csv('world-happiness-report-2021.csv')
2
3 df_stds = df.groupby("Regional indicator").std()
4 # Print entire DataFrame
5 print(df_stds)
6 # Print only the Series we'll be using
7 print(df_stds['Freedom to make life choices'])

```

	Ladder score	...	Dystopia + residual
1 Regional indicator		...	
2 Central and Eastern Europe	0.493325	...	0.359212
3 Commonwealth of Independent States	0.438116	...	0.379774
4 East Asia	0.439913	...	0.507088
5 Latin America and Caribbean	0.693467	...	0.381805
6 Middle East and North Africa	0.999259	...	0.434674
7 North America and ANZ	0.138057	...	0.104914
8 South Asia	0.993462	...	0.664725
9 Southeast Asia	0.606271	...	0.466095
10 Sub-Saharan Africa	0.654892	...	0.707694
11 Western Europe	0.656519	...	0.325763
12 Regional indicator			
13 Central and Eastern Europe	0.070521		
14 Commonwealth of Independent States	0.090477		

²²https://en.wikipedia.org/wiki/Standard_deviation

²³https://en.wikipedia.org/wiki/Standard_error

```
17 East Asia          0.083464
18 Latin America and Caribbean 0.088115
19 Middle East and North Africa 0.138409
20 North America and ANZ      0.041732
21 South Asia          0.178477
22 Southeast Asia       0.029698
23 Sub-Saharan Africa    0.087100
24 Western Europe        0.098594
```

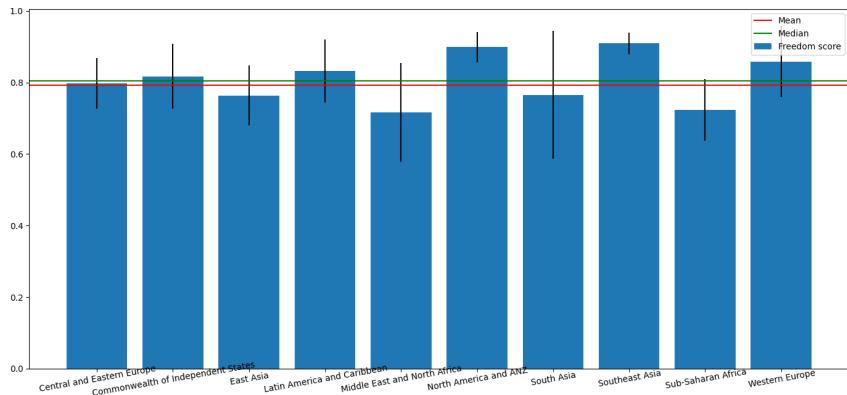
We've grouped the DataFrame by the regional indicator variable and calculated the standard deviation for each one. This calculates the standard deviation for *all variables*. We'll only be using the "*Freedom to make life choices*" though.

To visualize error bars, we call the regular `bar()` function, passing in the `df_means.index` (categorical values) and `df_means['Freedom to make life choices']` (numerical values), alongside the `yerr` argument. Since we're plotting vertically, we're using the `yerr` argument. If we were plotting horizontally, we'd use the `xerr` argument. These both denote the *error bars* that are to be added to each bar in the plot.

This adds the error bars to *each bar* on the plot we're creating:

```
1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('world-happiness-report-2021.csv')
5
6 df_means = df.groupby("Regional indicator").mean()
7 df_stds = df.groupby("Regional indicator").std()
8
9 mean = df['Freedom to make life choices'].mean()
10 median = df['Freedom to make life choices'].median()
11
12 fig, ax = plt.subplots()
13
14 ax.bar(df_means.index, df_means['Freedom to make life choices'],
15         label = 'Freedom score',
16         yerr=df_stds['Freedom to make life choices'])
17
18 ax.axhline(mean, 0, 1, color='red', label='Mean')
19 ax.axhline(median, 0, 1, color='green', label='Median')
20
21 ax.legend()
22 plt.xticks(rotation=10, wrap=True)
23 plt.show()
```

This results in:



The differences here are stark - “*Southeast Asia*” doesn’t have much deviation here, and most of the freedom scores you’ll encounter are in the ball-park of 0.9, give or take. On the other hand, in “*South Asia*”, the freedom score deviates between 0.9 and 0.6.

Now *this* bar plot does tell us something - it tells us the mean freedom score per region, the standard deviation per region indicating where policies are widespread and where they’re individualized, and it tells us the mean and median values for the entire dataset. Although Bar Plots are simple - that’s a solid amount of information from a single image.

Changing Bar Plot Colors

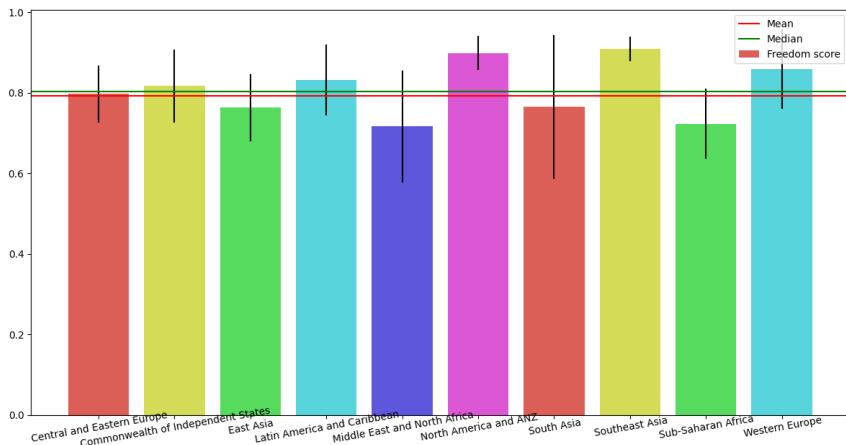
All of our bars are blue - while that’s fine, it’s nice to have a change of color as well. We can change the color of bars in a bar plot by supplying a list of colors that it’ll cycle through. If we have less colors than bars, once they’ve ran out - they’ll just cycle back from the beginning. If we have one color, it’ll be repeated for all bars:

```

1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('world-happiness-report-2021.csv')
5
6 # Calculating means and stds
7
8 fig, ax = plt.subplots()
9
10 colors=['#db5f57', '#d3db57', '#57db5f', '#57d3db', '#5f57db', '#db57d3']
11 ax.bar(df_means.index, df_means['Freedom to make life choices'], label = 'Freedom score', \
12        yerr=df_stds['Freedom to make life choices'], color=colors)
13
14 # Drawing lines and settings ticks
15
16 plt.show()

```

Here, we've made a list of HTML codes for colors, which is actually borrowed from *Seaborn's color palette - HLS*²⁴. We've supplied this list of colors to the color argument, which then colors each bar respectively:



We'll be covering *Matplotlib Colors and Stylesheets* in much more detail in *Chapter 7 - Advanced Matplotlib Customization*.

²⁴https://seaborn.pydata.org/tutorial/color_palettes.html

Plotting Horizontal Bar Plots

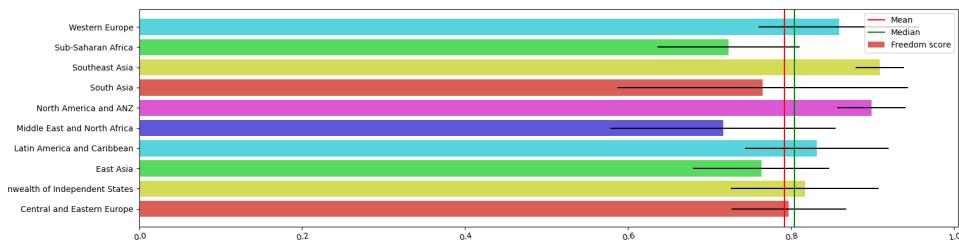
Since the really long names really do impede on the X-axis, it's high time we've done something about them. Unfortunately, Matplotlib doesn't have a built-in, smart text-wrap option which breaks it if it's too long to fit. We can manually set these labels, and include newlines (\n) or shorten them, though, there's an easier fix for most situations - *plotting the Bar Plot horizontally*.

To plot a Bar Plot horizontally - all we have to do is use the `barh()` function instead of `bar()`. It accepts the *same* arguments, though, we'll have to change our `yerr` to `xerr`, and our `axhline()` to `axvline()`:

```

1 colors=['#db5f57', '#d3db57', '#57db5f', '#57d3db', '#5f57db', '#db57d3']
2 ax.barh(df_means.index, df_means['Freedom to make life choices'],
3         label = 'Freedom score',
4         xerr=df_stds['Freedom to make life choices'],
5         color=colors)
6
7 ax.axvline(mean, 0, 1, color='red', label='Mean')
8 ax.axvline(median, 0, 1, color='green', label='Median')
```

We, don't actually have to change the order of `df_means.index` and `df_means['Freedom to make life choices']`, even though intuitively, it looks like they're in the wrong positions now. This code results in:



Sorting Bar Order

Matplotlib doesn't let us sort the order of the bars through its API. It gets the data as-is and plots it. What we can do however, is sort the data before we supply it. Thankfully, doing this with Pandas is a breeze:

```
1 df = pd.read_csv('world-happiness-report-2021.csv')
2 df_means = df.groupby("Regional indicator").mean()
3 df_means.sort_values("Freedom to make life choices", inplace=True)
```

This sorts the `df_means` DataFrame by the values of the “*Freedom to make life choices*” feature.

Note: Beware of sorting by values if you have *more than one* order dependent on these values. In our case, if we sort `df_means` by values, we also have to sort `df_stds` by values. Then again, there’s no guarantee that these will then be in sync, since the standard deviations between each categorical variable, and the means between them don’t have to follow the same trend. Sorted by value - the regional indicators can get mixed up fairly easily. If we sort them both individually and plot, it’ll look right - but the mappings between these could be totally wrong.

When you’d like to sort in a case like this - you have to map the standard deviation values to each regional indicator, and the mean values to each regional indicator, and *then sort*, when we can preserve their relationship.

Let’s rename the columns so that it’s a bit more clear what we’re referring to, and so that we can merge the `df_means` and `df_stds` feature in a single DataFrame which we can then sort:

```
1 df_means = df.groupby("Regional indicator").mean()
2 df_stds = df.groupby("Regional indicator").std()
3
4 df_stds.rename(columns = {'Freedom to make life choices': 'Freedom std'}, inplace=True)
5 df_means.rename(columns = {'Freedom to make life choices': 'Freedom mean'}, inplace=True)
6
7 df_merged = pd.merge(df_stds['Freedom std'], df_means['Freedom mean'],
8                      right_index=True, left_index=True)
9 print(df_merged)
```

We’ve only merged the two Series’ referring to “*Freedom std*” and “*Freedom mean*”, for brevity’s sake. We could’ve merged all other variables as well:

		Freedom std	Freedom mean
1	Regional indicator		
2	Central and Eastern Europe	0.070521	0.797059
3	Commonwealth of Independent States	0.090477	0.816917
4	East Asia	0.083464	0.763500
5	Latin America and Caribbean	0.088115	0.831750
6	Middle East and North Africa	0.138409	0.716471
7	North America and ANZ	0.041732	0.898750
8	South Asia	0.178477	0.765000
9	Southeast Asia	0.029698	0.909000
10	Sub-Saharan Africa	0.087100	0.723194
11	Western Europe	0.098594	0.858714
12			

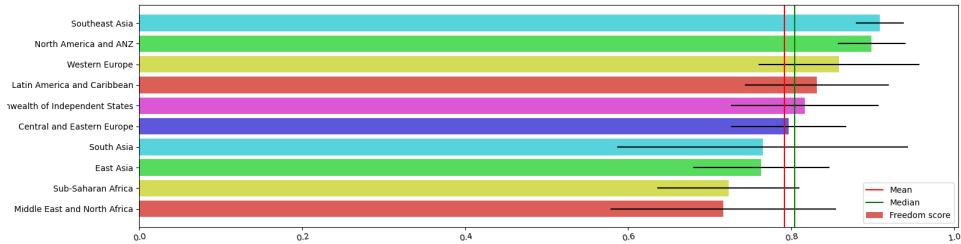
Now we can refer to this one DataFrame instead of the two individual ones, and sort these values which will remain mapped to their respective “*Regional indicator*”. Let’s sort this DataFrame by value and plot it again, tweaking our `barh()` function call to include the new DataFrame instead:

```

1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('world-happiness-report-2021.csv')
5
6 # Group by regions
7 df_means = df.groupby("Regional indicator").mean()
8 df_stds = df.groupby("Regional indicator").std()
9
10 # Rename columns so we can merge
11 df_stds.rename(columns = {'Freedom to make life choices': 'Freedom std'}, inplace=True)
12 df_means.rename(columns = {'Freedom to make life choices': 'Freedom mean'}, inplace=True)
13
14 # Merge and sort
15 df_merged = pd.merge(df_stds['Freedom std'], df_means['Freedom mean'],
16                      right_index=True, left_index=True)
17 df_merged.sort_values("Freedom mean", inplace=True)
18
19 # Calculate mean and median for vertical lines
20 mean = df['Freedom to make life choices'].mean()
21 median = df['Freedom to make life choices'].median()
22
23 fig, ax = plt.subplots()
24
25 # Plot bar plot
26 colors=['#db5f57', '#d3db57', '#57db5f', '#57d3db', '#5f57db', '#db57d3']
27 ax.barh(df_merged.index, df_merged['Freedom mean'],
28         label = 'Freedom score',
29         xerr=df_merged['Freedom std'], color=colors)
30
31 # Draw vertical lines
32 ax.axvline(mean, 0, 1, color='red', label='Mean')
33 ax.axvline(median, 0, 1, color='green', label='Median')
34
35 ax.legend()
36 plt.xticks(rotation=10, wrap=True)
37 plt.show()

```

Now, once we've sorted the DataFrame by a variable, whilst preserving the order of other dependent variables - we can run the code, which results in:



The bars are sorted by the mean value of each region, while their standard deviations each correspond to the region.

Pie Chart

After finishing up with Bar Plots, that show us the relationship between the numerical and categorical variables - we've got another, similar type of plot - Pie Charts.

Pie Charts represent data broken down into categories/labels. They're an intuitive and simple way to visualize proportional data - such as percentages.

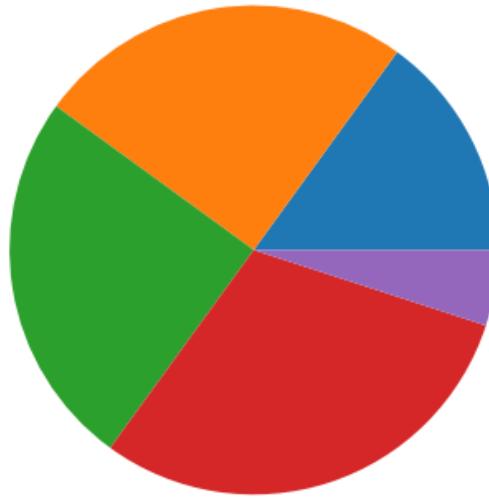
The main difference is that Pie Charts are *mainly* used with proportional data, where the sum of all the categories add up to 100% (or 1), instead of plotting the number of instances or aggregations of data like Bar Plots do. While you can tweak Bar Plots to display proportional data - Pie Charts are more visually fit for this task. This shouldn't stop you from making proportional Bar Plots, though.

Plotting a Pie Chart

To plot a pie chart in Matplotlib, we can call the `pie()` function of the PyPlot or Axes instance. The only mandatory argument is the data we'd like to plot, such as a feature from a dataset:

```
1 import matplotlib.pyplot as plt
2
3 x = [15, 25, 25, 30, 5]
4
5 fig, ax = plt.subplots()
6 ax.plot(x)
7 plt.show()
```

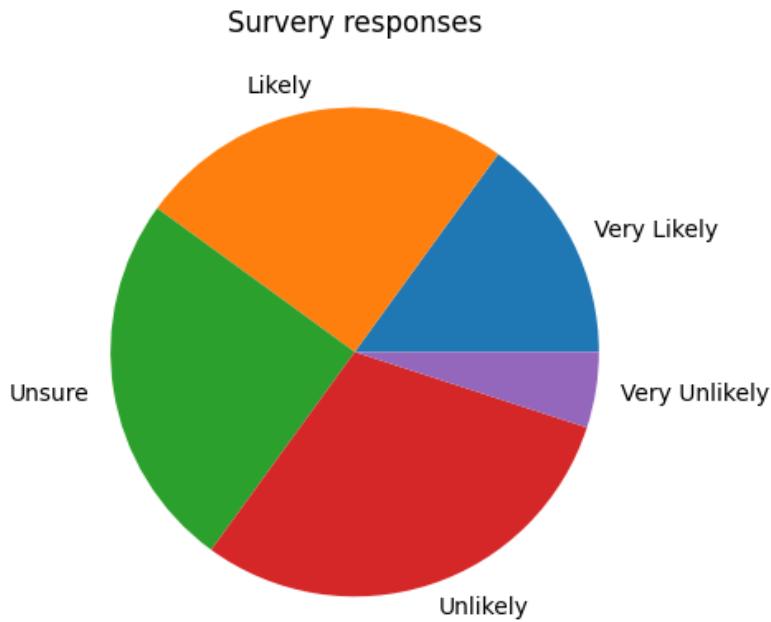
This generates a rather simple, but plain, Pie Chart with each value being assigned to a proportionally large slice of the pie:



Let's add some labels, so that it's easier to distinguish what's what here:

```
1 import matplotlib.pyplot as plt
2
3 x = [15, 25, 25, 30, 5]
4 labels = ['Very Likely', 'Likely', 'Unsure', 'Unlikely', 'Very Unlikely']
5
6 fig, ax = plt.subplots()
7 ax.pie(x, labels = labels)
8 ax.set_title('Survery responses')
9 plt.show()
```

Now, the Pie Chart will have some additional data that allows us to interpret it a bit easier:



Customizing Pie Charts

When preparing data visualizations for presentations, papers or simply to share them around with your peers - you might want to stylize and customize them a little bit,

such as using different colors, that correlate to the categories, showing percentages on slices, instead of just relying on the visual perception, or exploding slices to highlight them.

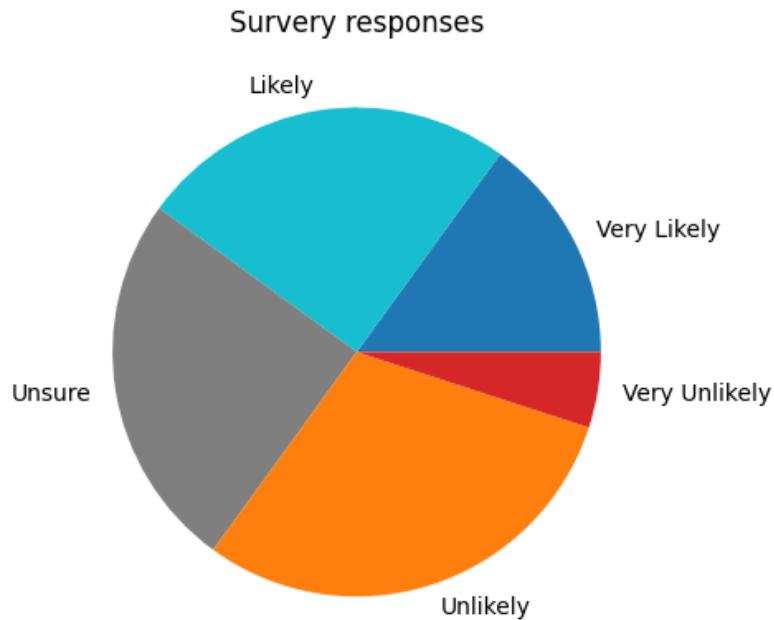
Changing Pie Chart Colors

To change the colors of a Pie Chart in Matplotlib, we'll need to supply an array of colors to the `colors` argument, while plotting it:

```
1 import matplotlib.pyplot as plt
2
3 x = [15, 25, 25, 30, 5]
4 labels = ['Very Likely', 'Likely', 'Unsure', 'Unlikely', 'Very Unlikely']
5 colors = ['tab:blue', 'tab:cyan', 'tab:gray', 'tab:orange', 'tab:red']
6
7 fig, ax = plt.subplots()
8 ax.pie(x, labels = labels, colors = colors)
9 ax.set_title('Survey responses')
10 plt.show()
```

Here, we've created a really simple correlation between the responses and the colors they're assigned. `Very Likely` will be blue in the Tableau Palette, while `Very Unlikely` will be red.

Running this code results in:



Showing Percentages on Slices

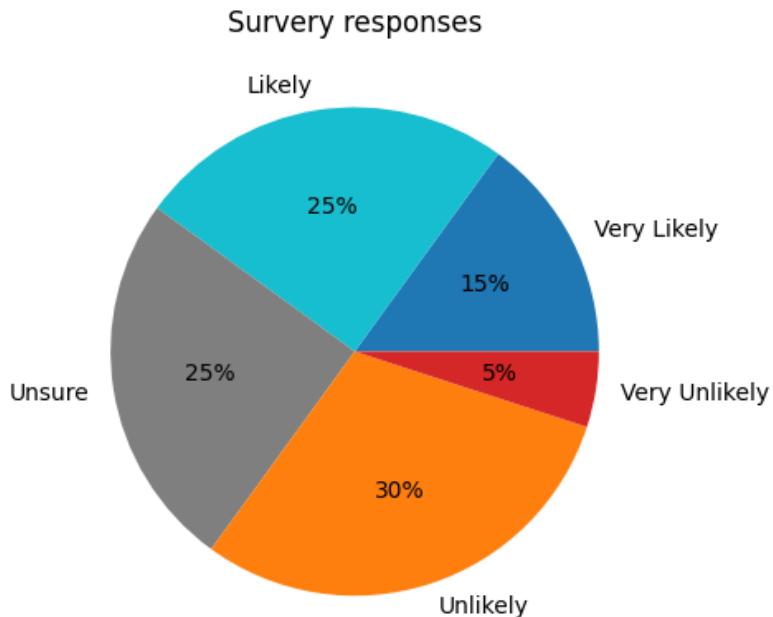
Looking at the Pie Chart we've made so far, it's clear that there are more `Unsure` and `Likely` respondents than other categories individually. Though, it's oftentimes easier for us to *both* interpret a Pie Chart visually, and numerically.

To add numerical percentages to each slice, we use the `autopct` argument. It automatically sets the percentages in each wedge/slice, and accepts the standard Python string formatting notation:

```
1 import matplotlib.pyplot as plt
2
3 x = [15, 25, 25, 30, 5]
4 labels = ['Very Likely', 'Likely', 'Unsure', 'Unlikely', 'Very Unlikely']
5 colors = ['tab:blue', 'tab:cyan', 'tab:gray', 'tab:orange', 'tab:red']
6
7 fig, ax = plt.subplots()
8 ax.pie(x, labels = labels, colors = colors, autopct='%.0f%%')
9 ax.set_title('Survery responses')
10 plt.show()
```

By setting `autopct` to `%.0f%%`, we've chosen to format the percentages with 0 decimal places (only whole numbers), and added a % sign at the end. If we had omitted the surrounding `%.0f%%` symbols, the strings wouldn't be formatted as percentages, but as literal values.

Running this code results in:



Explode/Highlight Wedges

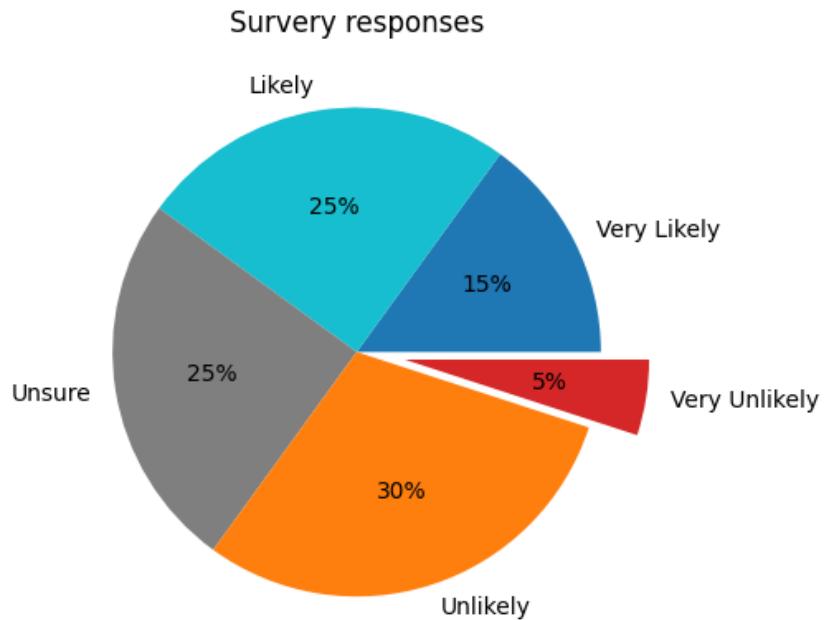
Sometimes, it's important to highlight certain entries. For example, in our survey, a really small percentage of the respondents feel like the advent of something in question is `Very Unlikely`. Assuming that we'd want to point out the fact that most people *don't think it's unlikely*, we can *explode* the wedge:

```
1 import matplotlib.pyplot as plt
2
3 x = [15, 25, 25, 30, 5]
4 labels = ['Very Likely', 'Likely', 'Unsure', 'Unlikely', 'Very Unlikely']
5 colors = ['tab:blue', 'tab:cyan', 'tab:gray', 'tab:orange', 'tab:red']
6 explode = [0, 0, 0, 0, 0.2]
7
8 fig, ax = plt.subplots()
9 ax.pie(x, labels = labels, colors = colors, autopct='%.0f%%', explode = explode)
10 ax.set_title('Survey responses')
11 plt.show()
```

The `explode` argument accepts an array of values, from `0..1`, where the values themselves define how further away the wedge is from the center. By default, all wedges have an `explode` value of `0`, so they're all connected to the center.

Setting this value to `1` would offset it by *a lot*, relative to the chart, so usually, you'll explode wedges by `0.1`, `0.2`, `0.3`, and similar values. You can *explode* as many of them as you'd like, with different values to highlight different categories.

Running this code results in:

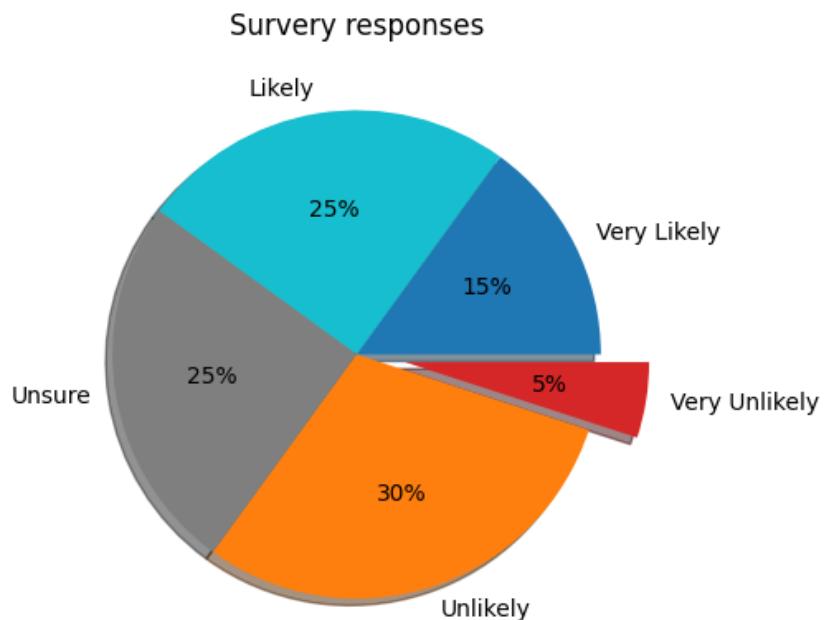


Adding a Shadow

To add a shadow to a Matplotlib pie chart, all you have to do is set the `shadow` argument to `True`:

```
1 import matplotlib.pyplot as plt
2
3 x = [15, 25, 25, 30, 5]
4 labels = ['Very Likely', 'Likely', 'Unsure', 'Unlikely', 'Very Unlikely']
5 colors = ['tab:blue', 'tab:cyan', 'tab:gray', 'tab:orange', 'tab:red']
6 explode = [0, 0, 0, 0, 0.2]
7
8 fig, ax = plt.subplots()
9 ax.pie(x, labels=labels,
10        colors=colors,
11        autopct='%.0f%%',
12        explode=explode,
13        shadow=True)
14
15 ax.set_title('Survery responses')
16 plt.show()
```

This results in:

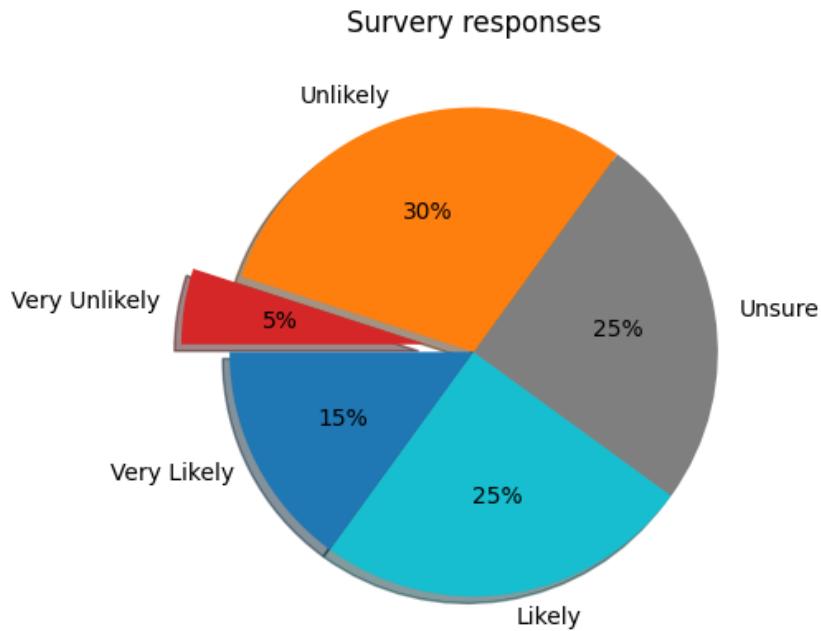


Rotating Pie Chart

Finally, you can also rotate the chart, by setting the starting angle. So far, it starts on 0 degrees (right-hand), and populated wedges counter-clockwise. By setting the startangle argument to a number between 0..360, you can make a full circle:

```
1 import matplotlib.pyplot as plt
2
3 x = [15, 25, 25, 30, 5]
4 labels = ['Very Likely', 'Likely', 'Unsure', 'Unlikely', 'Very Unlikely']
5 colors = ['tab:blue', 'tab:cyan', 'tab:gray', 'tab:orange', 'tab:red']
6 explode = [0, 0, 0, 0, 0.2]
7
8 fig, ax = plt.subplots()
9 ax.pie(x, labels=labels,
10         colors=colors,
11         autopct='%.0f%%',
12         explode=explode,
13         shadow=True,
14         startangle=180)
15
16 ax.set_title('Survey responses')
17 plt.show()
```

This results in a Pie Chart, rotated by 180 degrees, effectively flipping it to the other side:



Let's apply the dataset from the previous section to Pie Charts. This time around, we'll be visualizing the "*Perceptions of corruption*" which is the *subjective perception of corruption* by the people living in each country. We'll calculate the *mean* of the perception of each region in an attempt to visualize in which regions people perceive the most corruption.

Please keep in mind that this *would be skewed* by the number of countries - not the number of people if we didn't use an aggregation function such as `mean()`. We don't have access to individual responses here, so we can't summarize the *human-perceived corruption* clearly. Instead, a region with many countries will likely have a higher sum than another region with less countries - *even if the average perceived corruption is lower in every single country of the first region*:

```
1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('world-happiness-report-2021.csv')
5 df_mean = df.groupby("Regional indicator").mean()
6 print(df_mean['Perceptions of corruption'])
```

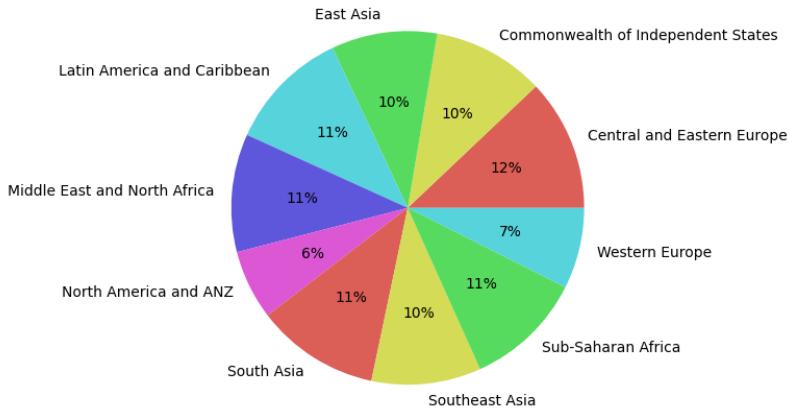
We've also printed the count of countries per region here, just so we get a sense of where the *means* are:

1	Regional indicator	
2	Central and Eastern Europe	0.850529
3	Commonwealth of Independent States	0.725083
4	East Asia	0.683333
5	Latin America and Caribbean	0.792600
6	Middle East and North Africa	0.762235
7	North America and ANZ	0.449250
8	South Asia	0.797429
9	Southeast Asia	0.709111
10	Sub-Saharan Africa	0.765944
11	Western Europe	0.523095

Now, let's use this data to form a Pie Chart:

```
1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('world-happiness-report-2021.csv')
5 df_sum = df.groupby("Regional indicator").sum()
6
7 fig, ax = plt.subplots()
8
9 colors=['#db5f57', '#d3db57', '#57db5f', '#57d3db', '#5f57db', '#db57d3']
10 ax.pie(df_sum['Perceptions of corruption'],
11         labels=df_sum.index,
12         colors=colors, autopct='%.0f%%')
13 plt.show()
```

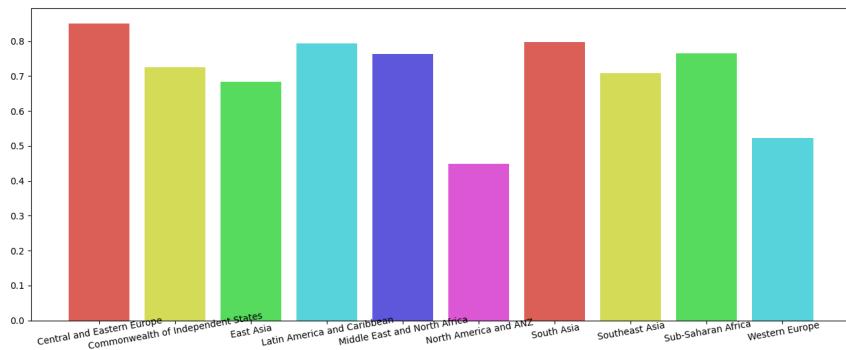
This results in:



Though, even if this chart displays the exact same data as if we were to plot it on a Bar Chart - it actually does a disservice to this dataset. Given several labels, and without focusing on what's on the Pie Chart, these wedges don't appear to be too different. Without the percentages on screen, it would be hard to tell whether *Western Europe* has a higher perceived corruption score than *North America and ANZ*.

Additionally, what happens with the standard deviations and confidence in this data? We've previously seen how *means* can actually be fairly inaccurate. If we were to switch the plot type here, and use a Bar Plot instead:

```
1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('world-happiness-report-2021.csv')
5 df_mean = df.groupby("Regional indicator").mean()
6
7 fig, ax = plt.subplots()
8
9 colors=['#db5f57', '#d3db57', '#57db5f', '#57d3db', '#5f57db', '#db57d3']
10 ax.bar(df_mean.index, df_mean['Perceptions of corruption'], color=colors)
11 plt.xticks(rotation=10, wrap=True)
12 plt.show()
```



Here, the distinction between values is much more apparent. Here, we can also add error bars - and plot the DataFrame level statistics. These are some of the reasons why many people prefer Bar Charts over Pie Charts, and why many people make a strong case against them.

This isn't to say that Pie Charts don't have a place or shouldn't be used - Pie Charts represent the same data as Bar Plots, it's just a matter of choosing when to use which one to more accurately portray the data.

Scatter Plot

With some of the more common and familiar plot types out of the way - let's take a look at *Scatter Plots* - one of the most important and commonly used plot type.

Scatter Plots visualize the relationship between two numerical features. These variables can be dependent, or independent of each other.

What Scatter Plots are really useful for is their ability to visualize *relationships and correlation* between multiple variables, denoted by *markers*. Markers can be any shape (even custom images), though, we commonly use circles. We can easily recognize *positive*, *negative* or *null* correlations between features, such as, for example - *Price* and *Quality*.

Note: There are other ways of visualizing correlation. A notable one is creating a *Correlation Heatmap* or *Correlation Matrix* and calculating the correlation for each feature against the feature we're comparing with. We'll cover these in later sections.

A variation of Scatter Plots, called Bubble Plots can include the size of a third variable into account by manipulating the size of the markers used to represent the relationship between two other variables. You can also visualize the relationship with a *line* instead of markers in some cases.

Scatter Plots are very versatile and can be used with a wide variety of datasets, so we'll be using a few in this section. Let's start off with the [Ames Housing²⁵](#) dataset and visualize correlations between features from it. The dataset contains 80 features that focus on the physical properties of houses in the city of Ames, Iowa:

```

1 import pandas as pd
2
3 df = pd.read_csv('AmesHousing.csv')
4 print(df)

```

	Order	PID	MS	SubClass	Lot	Frontage	...	Sold	Yr	Sale	Condition	SalePrice
2	0	1	526301100		20	141.0	...	2010		Normal		215000
3	1	2	526350040		20	80.0	...	2010		Normal		105000
4	2	3	526351010		20	81.0	...	2010		Normal		172000
5	3	4	526353030		20	93.0	...	2010		Normal		244000
6	4	5	527105010		60	74.0	...	2010		Normal		189900
7
8	2925	2926	923275080		80	37.0	...	2006		Normal		142500
9	2926	2927	923276100		20	Nan	...	2006		Normal		131000
10	2927	2928	923400125		85	62.0	...	2006		Normal		132000
11	2928	2929	924100070		20	77.0	...	2006		Normal		170000
12	2929	2930	924151050		60	74.0	...	2006		Normal		188000

This dataset has lots of numerical features and we can hypothesize that some features will affect other features. For example, the *Condition*, *Quality*, *Lot Frontage* and *Year Built* will certainly affect the *SalePrice* feature. On the other hand, the *Roof Style* will probably affect it a bit less, and since the style of a roof depends on the subjective preference of a buyer - we don't really know how much this feature could affect the *SalePrice*.

Plotting a Scatter Plot

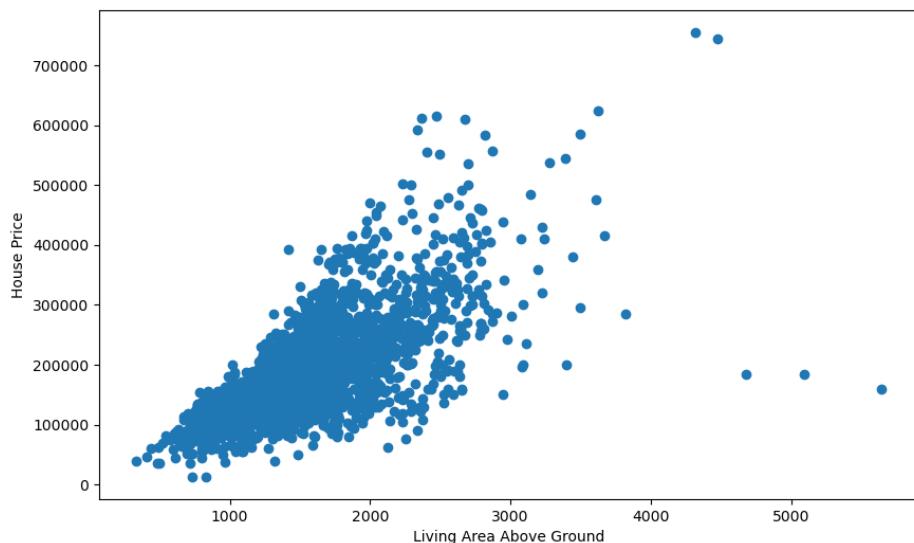
Let's start testing our hypothesis by visualizing a variable, such as "*Gr Liv Area*", which refers to the living area above ground, and the *SalePrice* variable, and supplying those as the *x* and *y* arguments of the *scatter()* function:

²⁵<https://www.kaggle.com/prevek18/ames-housing-dataset>

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('AmesHousing.csv')
5
6 fig, ax = plt.subplots(figsize=(10, 6))
7 ax.scatter(x = df['Gr Liv Area'], y = df['SalePrice'])
8 plt.xlabel("Living Area Above Ground")
9 plt.ylabel("House Price")
10
11 plt.show()
```

Note: You can simply supply the array-like data directly into the `scatter()` function like we usually did so far. Alternatively, you can specify that these as *keyword arguments*, explicitly setting `x` and `y`. Specifying them explicitly removes possible ambiguity. Some other libraries, like Seaborn have recently updated their API to *request adding these arguments*, since ambiguity may lead to unexpected visualizations in *some* cases. Throughout the book, we'll generally use *keyword arguments* instead of just *positional arguments*, but we'll switch between these approaches at our own leisure.

Running this code results in:



We've also set the X and Y labels to indicate what the variables represent. There's a clear positive correlation between these two variables. The more area there is above ground-level, the higher the price of the house was. This makes sense, since people generally prefer to be above the ground-level, where they have more privacy from the outside. There *are* a few outliers, but the vast majority follows this hypothesis.

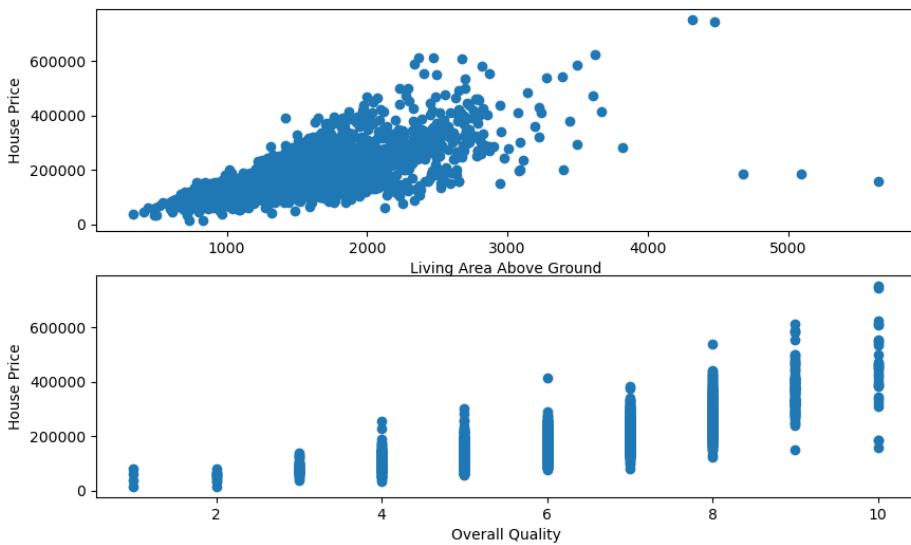
If you'd like to compare more than one variable against another, such as - check the correlation between the overall quality of the house against the sale price, as well as the area above ground level - there's no need to make a 3D plot for this.

While 2D plots that visualize correlations between more than two variables exist, such as Contour Plots, we'll simply plot multiple Scatter Plots, and we're saving Contour Plots for a later section in this chapter.

In one plot, we'll visualize the area above ground level against the sale price, in the other, we'll plot the overall quality against the sale price:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('AmesHousing.csv')
5
6 fig, ax = plt.subplots(2, figsize=(10, 6))
7 ax[0].scatter(x = df['Gr Liv Area'], y = df['SalePrice'])
8 ax[0].set_xlabel("Living Area Above Ground")
9 ax[0].set_ylabel("House Price")
10
11 ax[1].scatter(x = df['Overall Qual'], y = df['SalePrice'])
12 ax[1].set_xlabel("Overall Quality")
13 ax[1].set_ylabel("House Price")
14
15 plt.show()
```

We've created two axes, through `subplots(2)`, and saved them in a single array of `Axes` instances, accessing them through `ax[0]` and `ax[1]`. Then, we've called `scatter()` on each of them, producing a Figure with two plots:



The *Overall Quality* also obviously has a positive correlation with the *SalePrice*. There are, however, houses of quality 9 that sold for the same price as houses of quality 3. This doesn't really tell us much - since we don't know what the other features are, a really small, high quality house might sell for the same price as a really large, lower-quality house.

To explore these possibilities, we'd have to plot three variables at the same time - which again, requires us to use another type of plot or to plot in 3D where we can also go into the third dimension with our scattered markers. We'll cover both of these in the oncoming sections, though for now, let's settle for regular 2D Scatter Plots.

Speaking of which, the second plot here, which has only a few X-axis variables, when modified a bit, has another name - a *Strip Plot*. With a small number of categorical variables like this, there's a distinctive candle-like structure of the Scatter Plot. Since the markers here are relatively large for the number of houses in the candles - a lot of these look like filled straight lines. If we modified the markers here to occupy less space - we could see more of a distinction between individual houses. Now - we can't say if there's 5 or 500 houses in any of these candles.

This is what a *Strip Plot* is. Thankfully, we can change the markers for a Scatter Plot

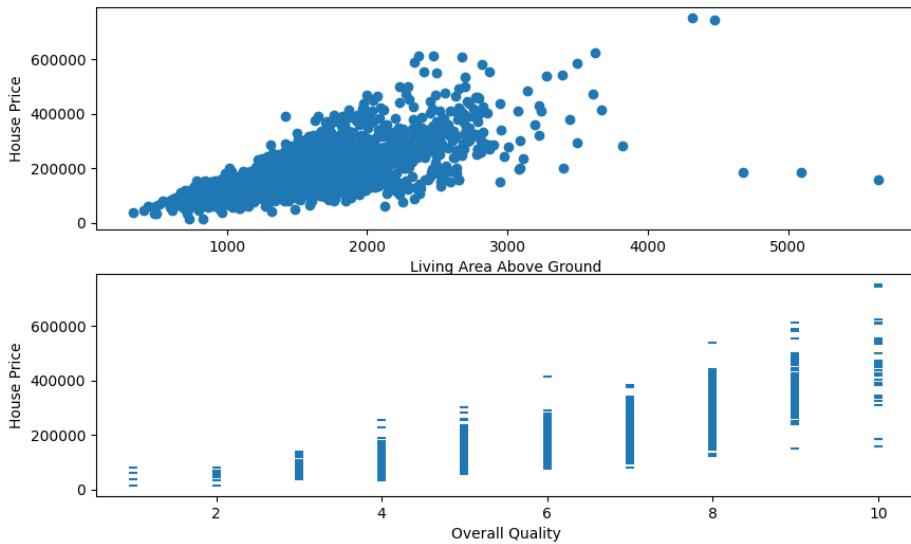
easily, and there are quite a few of them²⁶. Let's change out the circle with a `_` (*hline*) marker, which is significantly smaller in size:

```

1 # Import dataset, instantiate Figure and Axes, plot first plot
2
3 # Change marker type
4 ax[1].scatter(x = df['Overall Qual'], y = df['SalePrice'], marker='_')
5 ax[1].set_xlabel("Overall Quality")
6 ax[1].set_ylabel("House Price")
7
8 plt.show()

```

This now results in:



Matplotlib doesn't have a `strip()` function, or anything of the sort. People simply changed the marker type to gain more clarity for their Scatter Plots. At one point, the name stuck due to the look of the plot - which was then transferred as a built-in function in another popular data visualization library - Seaborn.

Oftentimes, Strip Plots also have a *hue* - different instances are colored based on another variable. For example, if this was a general *Real Estate dataset*, and contained

²⁶https://matplotlib.org/stable/api/markers_api.html

Apartments and *Houses*, we could color apartments in one color and houses in another, making the *strips* in the Strip Plot more informative.

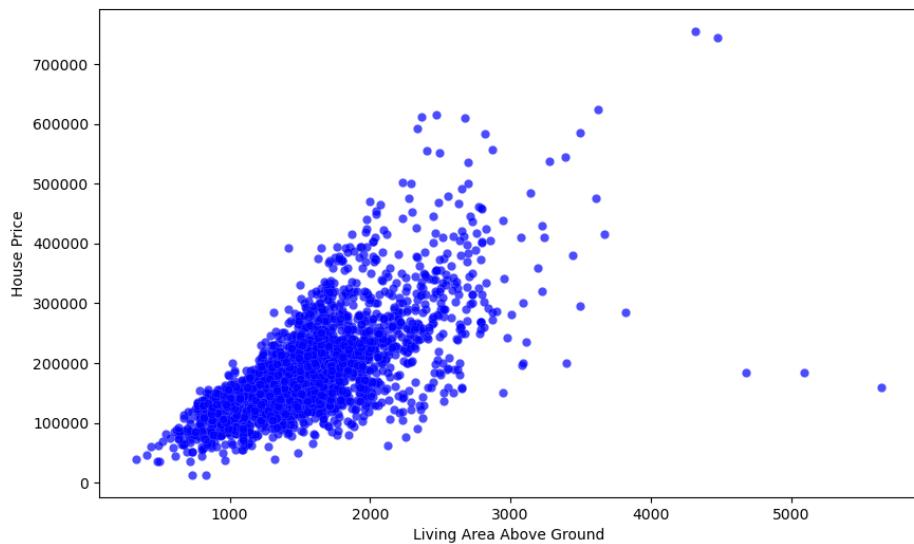
A good candidate for this is the *Sale Condition* variable, which denotes the condition under which the property is to be sold, and can be “Normal”, “Abnormal”, “Family”, “Partial”, “Alloca” and “AdjLand”. We’ll explore plotting both a Scatter Plot and Strip Plot with marker colors depending on another feature, amongst other options, in the next section.

Customizing Scatter Plot in Matplotlib

Let’s change things up a bit and customize how our Scatter Plot looks like. This can be for styling purposes, but also for *informative purposes*. As usual, you can set some common arguments such as `color` and `alpha` as arguments:

```
1 ax.scatter(x = df['Gr_Liv_Area'], y = df['SalePrice'],
2             color = "blue",
3             edgecolors = "white",
4             linewidths = 0.1,
5             alpha = 0.7)
```

Running this code would result in:



But we can also inject more information here through color-coding a certain class/feature of each house in the dataset. Matplotlib lets us change the *color* of each marker, even though it doesn't seem that way if we look at the code before this. We can map the feature values to certain colors, and then depending on the feature value, use the appropriate color while plotting.

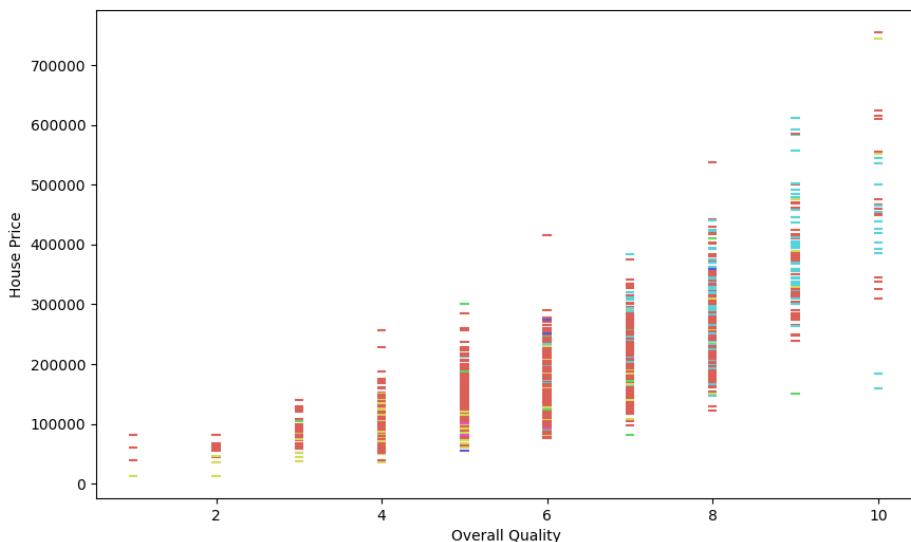
Let's define a `colors` dictionary, which maps the classes we'd like to color, and their respective colors:

```
1 colors = {'Normal' : '#db5f57',
2           'Abnorml' : '#d3db57',
3           'Family' : '#57db5f',
4           'Partial' : '#57d3db',
5           'Alloca' : '#5f57db',
6           'AdjLand' : '#db57d3'
7 }
```

These are, again, the colors of the HLS Color Palette, like last time. Now, when plotting the Scatter Plot (Strip Plot), we can *apply* the HTML color codes to the `color` argument of each marker:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('AmesHousing.csv')
5
6 fig, ax = plt.subplots(figsize=(10, 6))
7
8 colors = {'Normal' : '#db5f57',
9            'Abnorml' : '#d3db57',
10           'Family' : '#57db5f',
11          'Partial' : '#57d3db',
12         'Alloca' : '#5f57db',
13        'AdjLand' : '#db57d3'}
14
15 ax.scatter(x = df['Overall Qual'], y = df['SalePrice'],
16             marker='_',
17             color=df['Sale Condition'].apply(lambda x: colors[x]))
18
19 ax.set_xlabel("Overall Quality")
20 ax.set_ylabel("House Price")
21
22 plt.show()
```

This results in a color-coded Strip Plot:



There does seem to be some correlation in the colors here - there are more blue-coded houses in the higher-quality section of the market. Naturally, these sell for a higher

price in general. This color maps to #57d3db, which is the *Partial* sale condition. This could imply that house-owners fix up houses with high-quality materials, and then sell parts of the house as smaller standalone units, similar to apartments. This is most certainly not an unheard of strategy for rental properties - it might be a strategy here too.

This isn't conclusive evidence - but there does seem to be some correlation there.

Change Marker Size in Matplotlib Scatter Plot

Just like how we can change the marker shape and color - we can also change their size. Manipulating the size of a Scatter Plot marker, can in certain cases lead to a new type of plot - a *Bubble Plot*. The name arises from the fact that circular markers, if large, next to smaller markers of the same shape look like bubbles.

This is, essentially still a Scatter Plot, similar to how a Strip Plot is still, essentially a Scatter Plot.

If you want to scale the size of your markers by another variable - you'll want to do so with a numerical variable since you'll be passing those variables as the `size` argument. This, of course, means that we'll be using reasonable data - you don't really want to assign the size of 2058290505 to the marker. Generally, if you're working with variables on the order of tens, hundreds, or higher - you'll want to scale the data down. Similarly, if your data ranges from 0..1, you'll want to scale it up.

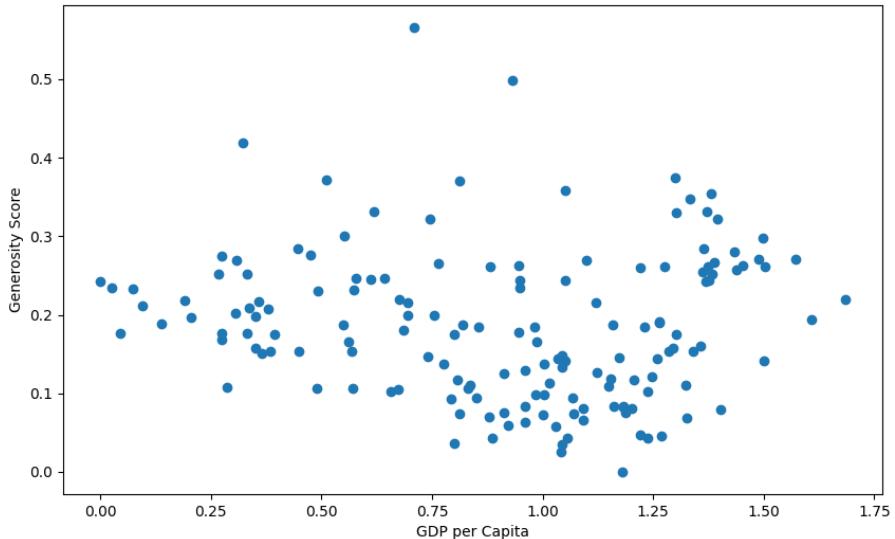
Let's switch our efforts to another dataset - the [World Happiness Dataset²⁷](#), for this, since we've got more fitting values there. It's very similar to the *World Happiness Report 2021* dataset, but also includes a *Happiness Score*, which isn't present in the latter.

We'll visualize the *GDP per Capita* and *Generosity* scores:

²⁷<https://www.kaggle.com/unssdsn/world-happiness>

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('worldHappiness2019.csv')
5
6 fig, ax = plt.subplots(figsize=(10, 6))
7 ax.scatter(x = df['GDP per capita'], y = df['Generosity'])
8 plt.xlabel("GDP per Capita")
9 plt.ylabel("Generosity Score")
10
11 plt.show()
```

This results in:



Not much correlation here at all. In fact, it looks like there's a bit less generosity with higher GDP per Capita, according to this dataset.

Plotting Bubble Plots

Bubble Plots are called Bubble Plots because they resemble a swarm of bubbles. They're typically each differently sized and transparent to a degree.

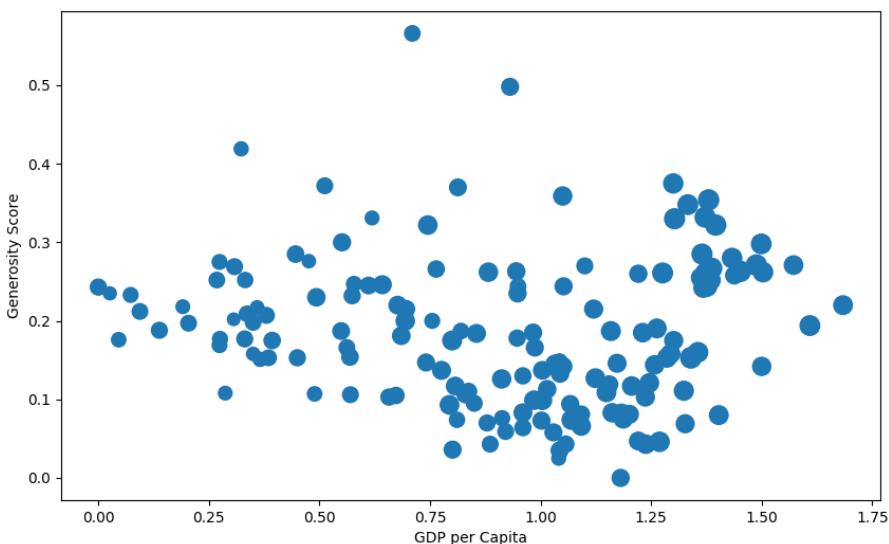
Let's increase the size of each marker, based on a third variable - the perceived *Happiness* of the inhabitants of that country. To change the size of the markers, we'll

set the `s` argument to a size. The same as with colors - it doesn't actually have a single size. We can supply a list of sizes, the length of the input.

For each entry, we'll have a different size - which is only feasible to do with a function or other calculation of that list. Let's set the `s` argument to reflect the *Happiness Score*, and multiply it by 25 since the *Happiness Score* ranges from 0..1. This would produce extremely small markers, which would be hard to interpret:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('worldHappiness2019.csv')
5
6 fig, ax = plt.subplots(figsize=(10, 6))
7 ax.scatter(x = df['GDP per capita'], y = df['Generosity'], s = df['Score']*25)
8 plt.xlabel("GDP per Capital")
9 plt.ylabel("Generosity Score")
10
11 plt.show()
```

This now results in:



This didn't help too much. These are of different sizes, though, they're not different enough for us to really know the difference between some of these. We don't want

the sizes to be calculated linearly, such as multiplying everything by a constant - 25 in this case.

Let's change this up by creating a new list, based on the values of the *Happiness Score* feature, but introduce an exponent. We'll multiply the score first, and then raise it to the power of 2. This introduces us to a non-linear growth pattern. Really small values will get *smaller*, while higher values will get *bigger*.

Let's convert the score into an array and calculate a new list of values:

```
1 size = df['Score'].to_numpy()
2 s = [3*s**2 for s in size]
3 print(s)
```

We're using `to_numpy()` here to create a Numpy array since we can easily iterate over it and create a new list. For each element in the list, we've multiplied it by 3 and raised it to the power of two, resulting in a new `s` list of:

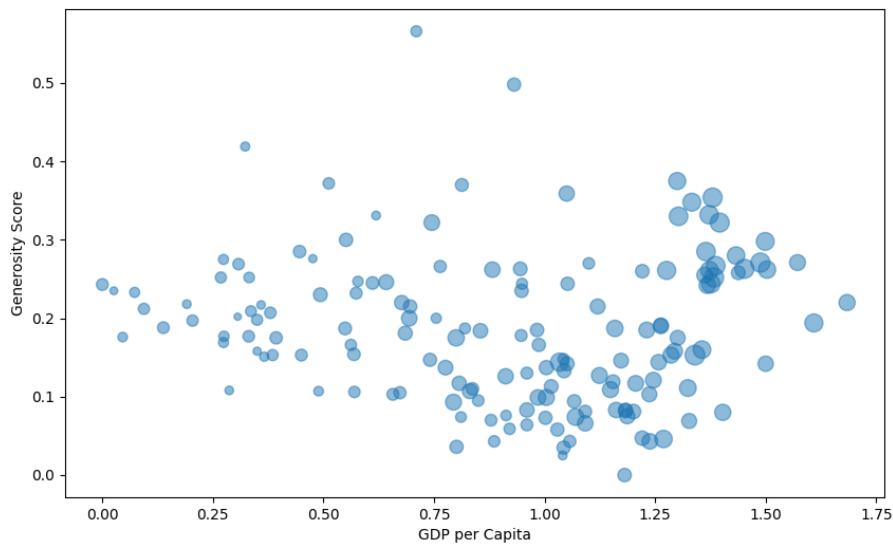
```
1 [181.0720830000002, 173.28, 171.188748, 168.480108, 168.2104320000003...]
```

While these may seem excessive - keep in mind that the `s` (size) of each marker is its *area*. This means that to double the height/width of a marker, you have to multiply `s` by 4. This mechanic is actually exactly the reason we *can* use this approach to scaling Scatter Plot markers.

Once we go back and use this new size list in the plot:

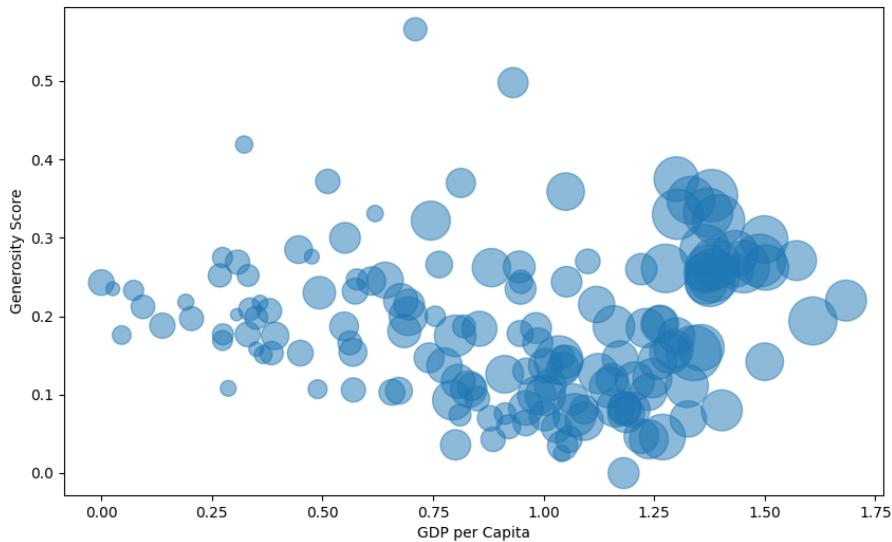
```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('worldHappiness2019.csv')
5
6 size = df['Score'].to_numpy()
7 s = [3*s**2 for s in size]
8
9 fig, ax = plt.subplots(figsize=(10, 6))
10 ax.scatter(x = df['GDP per capita'], y = df['Generosity'], s = s, alpha=0.5)
11 plt.xlabel("GDP per Capita")
12 plt.ylabel("Generosity Score")
13
14 plt.show()
```

We're greeted with a new plot:



Or, we could raise s to the power of 3:

```
1   s = [3*s**3 for s in size]
```

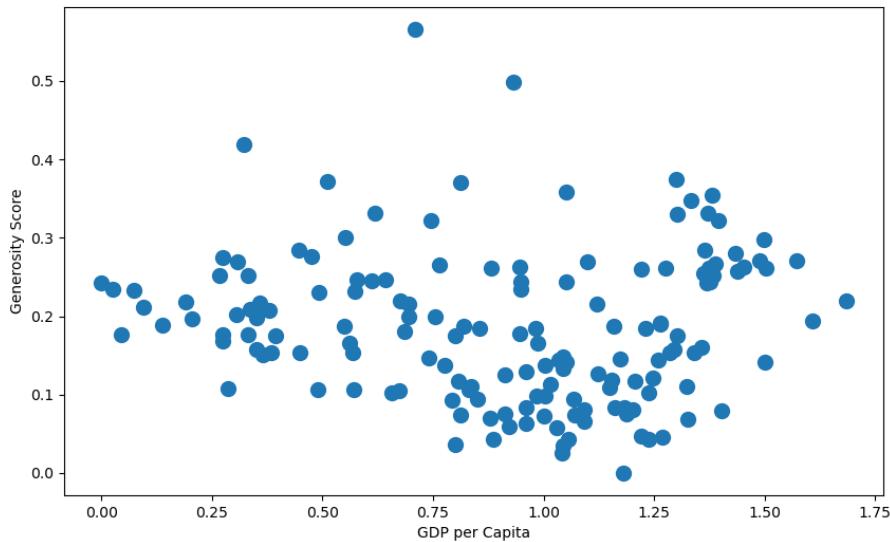


This looks more along the lines of what you'd expect when plotting a Bubble Plot. Although the *Generosity* does seem to *slightly* decrease with higher *GDP per Capita*, the *Happiness Score* appears to increase.

If you'd like to detach the marker size from another variable, and would just like to set a standard, global size of markers in the scatter plot, you can simply pass in a single value for *s*:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('worldHappiness2019.csv')
5
6 fig, ax = plt.subplots(figsize=(10, 6))
7 ax.scatter(x = df['GDP per capita'], y = df['Generosity'], s = 100)
8 plt.xlabel("GDP per Capital")
9 plt.ylabel("Generosity Score")
10
11 plt.show()
```

This now results in:



Exploring Relationships with Scatter Plots

Now, with customization in hand, we can go ahead and visualize a couple of relationships we're interested in. For example, let's take a look if the *Lot Area* has increased over the years. Also, we could take a look at the trend of *Total Basement Area* over the years as well. We can change the marker size of the *Living Area* against *SalePrice* Scatter Plot as well:

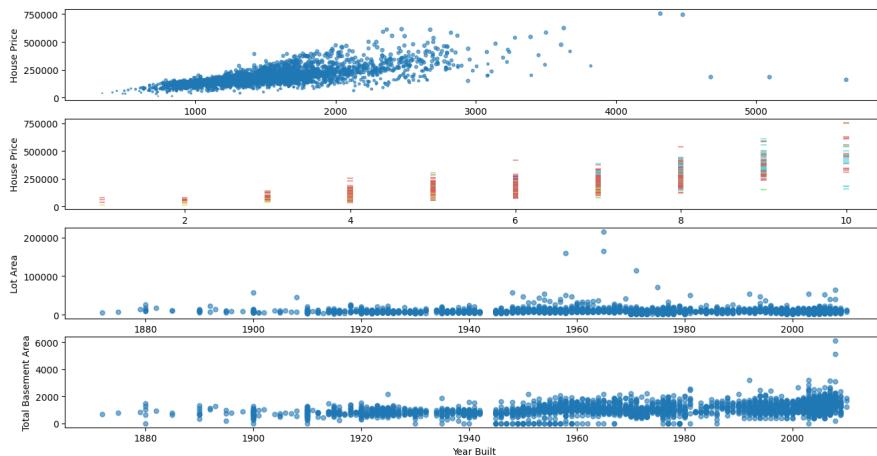
```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('AmesHousing.csv')
5
6 fig, ax = plt.subplots(4, figsize=(10, 6))
7
8 size = df['Overall Qual'].to_numpy()
9 s = [s**1.2 for s in size]
10
11 ax[0].scatter(x = df['Gr Liv Area'], y = df['SalePrice'], s = s)
12 ax[0].set_xlabel("Living Area Above Ground")
13 ax[0].set_ylabel("House Price")
14
15 colors = {'Normal' : '#db5f57',
16           'Abnormal' : '#d3db57',
```

```

17     'Family' : '#57db5f',
18     'Partial' : '#57d3db',
19     'Alloca' : '#5f57db',
20     'AdjLand' : '#db57d3'}
21
22 ax[1].scatter(x = df['Overall Qual'], y = df['SalePrice'],
23                 marker='_',
24                 color=df['Sale Condition'].apply(lambda x: colors[x]))
25
26 ax[1].set_xlabel("Overall Quality")
27 ax[1].set_ylabel("House Price")
28
29 ax[2].scatter(x = df['Year Built'], y = df['Lot Area'], alpha = 0.6, s = 25)
30 ax[2].set_xlabel("Year Built")
31 ax[2].set_ylabel("Lot Area")
32
33 ax[3].scatter(x = df['Year Built'], y = df['Total Bsmt SF'], alpha = 0.6, s = 25)
34 ax[3].set_xlabel("Year Built")
35 ax[3].set_ylabel("Total Basement Area")
36
37 plt.show()

```

This results in:



There doesn't seem to be much of an increase in the *Lot Area* for most houses. There is a *slight* increase, but for the most part - houses built in the 1880s and 1990s have a fairly similar lot area. However, on the other hand, the *Total Basement Area* definitely has increased over time. This makes sense - if you have a house next to yours, you can't expand the lot area. The entirety of the city might lean towards slightly larger

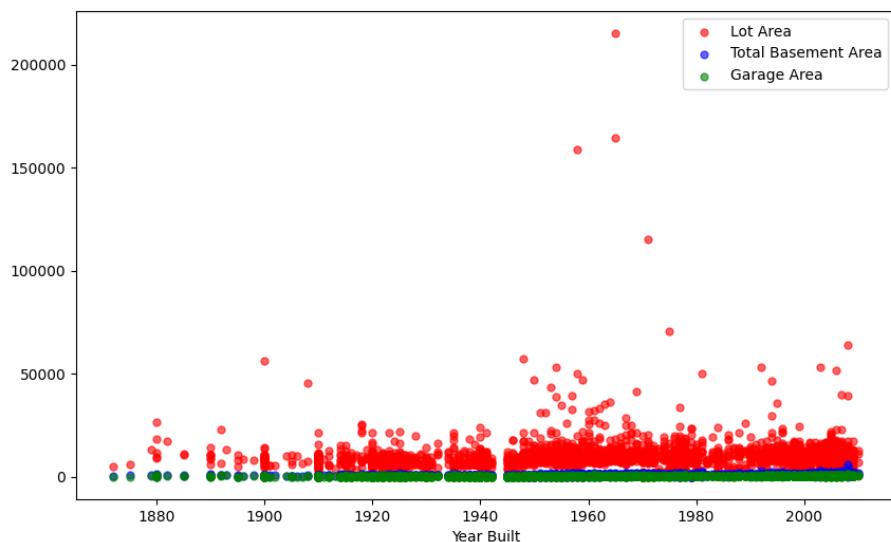
lot sizes when building houses, but at least this data implies that the trend wasn't too significant, other than several outliers.

On the other hand - you *can* expand the basement, as you won't intrude on your neighbor's lawn. Either by going down multiple levels, or by using more of the available space beneath the houses - increasing the *Basement Area* is possible, and it has increased over the years.

While we're already here, we can take a look at the *Total Basement Area* and *Garage Area* over the years on the same Axes, since they share the same X-axis values.

Since this is already *4 different Scatter Plots* on a single Axes, it'll most likely be crowded, so we'll reserve the entire Figure for this one Axes, and we'll preemptively reduce the size of the markers in anticipation of a huge number of them:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('AmesHousing.csv')
5
6 fig, ax = plt.subplots(figsize=(10, 6))
7
8 ax.scatter(x = df['Year Built'], y = df['Lot Area'],
9             alpha = 0.6, s = 25, color='red', label = 'Lot Area')
10
11 ax.scatter(x = df['Year Built'], y = df['Total Bsmt SF'],
12             alpha = 0.6, s = 25, color='blue', label = 'Total Basement Area')
13
14 ax.scatter(x = df['Year Built'], y = df['Garage Area'],
15             alpha = 0.6, s = 25, color='green', label='Garage Area')
16
17 ax.set_xlabel("Year Built")
18 ax.legend()
19
20 plt.show()
```



Not as helpful as planned. The few outliers of the *Lot Area* feature blew this out of proportion, and all other values are just jumbled at the bottom of the plot. Naturally, the *Lot Area* will be larger than *Garage Areas* and *Basement Areas*, though, at least it's not *this bad*. A relatively easy way to remove outliers, after we've visually identified them like this is to limit the *quantiles of a DataFrame*. We can cut off the top X-quantile, for example, removing some of the more offending outliers here.

An alternative approach would be to calculate the *Z-score (Standard Score)* of columns, and drop them based on the result:

The Z-score is the signed number of standard deviations by which the value of an observation or data point is above the mean value of what is being observed or measured.

A Z-score can be calculated through the `stats` module of Scipy, a Python library dedicated for Science, Mathematics and Engineering, mainly used for scientific and technical computing. Though, since the calculation of the Z-score is fairly straightforward, we can go ahead and calculate it ourselves as to not import a new library:

$$z = \frac{x - \mu}{\sigma}$$

Where μ is the mean, and σ is the standard deviation, and x is the value we're calculating z for

That being said, let's truncate the DataFrame and filter it out. Depending on the dataset, we'll define what outliers are. This is commonly a trial-and-error phase, where you tweak the threshold at which the data is being excluded from the DataFrame. In most cases, the Z-score will be between -3 and 3 , which means that outliers will most likely be *above* the 3 mark. This is a common threshold for treating something as an outlier - but keep in mind that *you* choose what an outlier is.

In our case, since we have only a few variables we'd like to get rid of, 3 should be fine:

```

1 df = pd.read_csv('AmesHousing.csv')
2 # Truncating DataFrame to just the features we're visualizing, since
3 # all need to be numerical for us to calculate the Z-score
4 df = df[['Year Built', 'Lot Area', 'Total Bsmt SF', 'Garage Area']].copy()
5
6 # Print df's shape
7 print(df.shape)
8 # Remove outliers by calculating the Z-score and removing
9 # all entries that have a Z-score higher than 2
10 df = df[df.apply(lambda x: np.abs(x - x.mean()) / x.std() < 3).all(axis=1)]
11 # Print df's shape again
12 print(df.shape)
```

Here, we've calculated the Z-score for each column and removed all entries that have a Z-score higher than 3 . Printing the shape of the DataFrame before and after this operation will show us how many rows we've removed:

```

1 (2930, 4)
2 (2859, 4)
```

Lost about 70 entries - sounds about right.

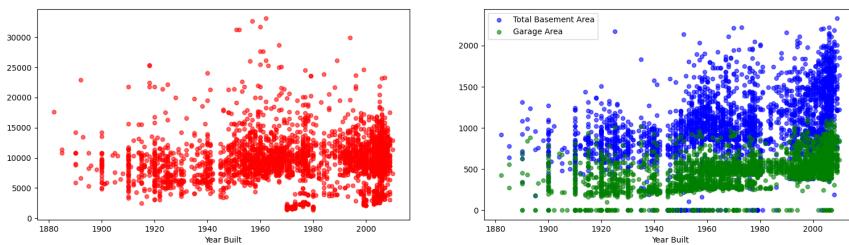
Since the *Lot Area* will generally be larger than both *Garage Areas* and *Basement Areas*, our plan to re-use the same X-axis might have to go down the drain. Let's plot the truncated DataFrame on two Axes, keeping the more similar features like *Garage Areas* and *Basement Areas* on one Axes, while plotting the *Lot Area* on the other:

```

1 import matplotlib.pyplot as plt
2 import pandas as pd
3 from scipy import stats
4 import numpy as np
5
6 df = pd.read_csv('AmesHousing.csv')
7 df = df[['Year Built', 'Lot Area', 'Total Bsmt SF', 'Garage Area']].copy()
8 df = df[df.apply(lambda x: np.abs(x - x.mean()) / x.std() < 3).all(axis=1)]
9
10 # ncols to create plots on the left and right, not one on top of the other
11 fig, ax = plt.subplots(ncols=2)
12
13 # Plot Scatter plots
14 ax[0].scatter(x = df['Year Built'], y = df['Lot Area'],
15                 alpha = 0.6, s = 25, color='red', label = 'Lot Area')
16
17 ax[1].scatter(x = df['Year Built'], y = df['Total Bsmt SF'],
18                 alpha = 0.6, s = 25, color='blue', label = 'Total Basement Area')
19
20 ax[1].scatter(x = df['Year Built'], y = df['Garage Area'],
21                 alpha = 0.6, s = 25, color='green', label='Garage Area')
22
23 # Set labels and enable legend
24 ax[0].set_xlabel("Year Built")
25 ax[1].set_xlabel("Year Built")
26 ax[1].legend()
27
28 plt.show()

```

This results in:



If we were to share the *Axes* on this, the *Lot Area* values would be significantly higher than the other values, and they'd sink to the bottom of the plot making them hard to interpret.

As we've seen before, the general *Lot Area* hasn't really increased much over the years. It's been much the same since the beginning of the dataset. There are some higher values than usual in the 1960s, but they drop off in the 1980 with a rise in the 2000s again.

On the other hand, both the *Total Basement Area* and *Garage Area* have a positive correlation with the years that have passed. They're definitely increasing over time. One explanation, as we've explored it previously, is that people tend to desire more land over time, but can't expand on the ground-level due to the fact that there are physical houses next to their lots. A solution is to increase the basement area, since you can expand into the ground without impeding on someone else's land.

Cars have also become more common in families over the years, as well as on the roads. Entire city infrastructures change to accommodate for the new cars on the roads, since they're produced in the tens of millions each year. According to [Statista²⁸](#), in 2018, we've produced *97 million cars*. The number dropped off in 2020, most likely due to the effects of the SARS-CoV-2 pandemic.

Also according to [Statista²⁹](#) the average number of vehicles per capita, in 2017, in the US is *1.88*, though, the data only goes back to 2001. unfortunately. Given the general increase of cars on the roads and households - it's only natural that *Garage Areas* are increasing, to accommodate.

Histogram Plot

In this section, we'll take a look at *Histogram Plot*. Histogram plots are a great way to visualize distributions of data - In a Histogram, each bar represents input data placed into *ranges*, oftentimes called *bins* or *buckets*. Taller bars show that more data falls in that range, in percentages or count.

Histograms are very common in Data Science as the first and foremost (but not only) way to display distributions of data. They're oftentimes used *alongside Kernel Density Estimations³⁰* since they represent the same data in a different format. KDE Lines are smoothed out, due to the smoothing parameter/bandwidth, while Histograms are a bit more crude. You'll commonly see Histogram Plots overlaid with a KDE Line.

This is yet another case of people simply combining different plot types or tweaking them to get new results. Thus - there's no built-in function in Matplotlib to "turn on" the KDE Line over a Histogram.

²⁸<https://www.statista.com/statistics/262747/worldwide-automobile-production-since-2000>

²⁹<https://www.statista.com/statistics/551403/number-of-vehicles-per-household-in-the-united-states/>

³⁰<https://stackabuse.com/kernel-density-estimation-in-python-using-scikit-learn>

Though, we *can* calculate the KDE ourselves and plot it on top of the Histogram manually. Or better yet - we can utilize *Pandas* to plot the KDE for us.

A Histogram displays the shape and spread of numerical data.

Plotting a Histogram Plot

Let's start off with finding a dataset ripe for Histograms. Again - Histograms are used to display the distribution of numerical data. Any numerical data works for us here. We'll use a couple of different datasets, including *Ames Housing Dataset* to visualize the *distribution* of certain house features.

We can begin with the [Netflix Movies and TV Shows³¹](#) dataset. It contains a list of Netflix movies and shows obtained through a third-party search engine. There's data up until 2019, which isn't the *freshest* dataset, but it'll work nonetheless. Let's import it and take a look at what's inside:

```

1 import pandas as pd
2
3 df = pd.read_csv('netflix_titles.csv')
4 print(df)
5 print(df.columns)

1      show_id    ...
2      0           s1   ...
3      1           s2   ...
4      2           s3   ...
5      3           s4   ...
6      4           s5   ...
7      ...        ...
8      7782       s7783 ...
9      7783       s7784 ...
10     7784       s7785 ...
11     7785       s7786 ...
12     7786       s7787 ...
13
14 [7787 rows x 12 columns]
15 Index(['show_id', 'type', 'title', 'director', 'cast', 'country', 'date_added',
16         'release_year', 'rating', 'duration', 'listed_in', 'description'],
17         dtype='object')
```

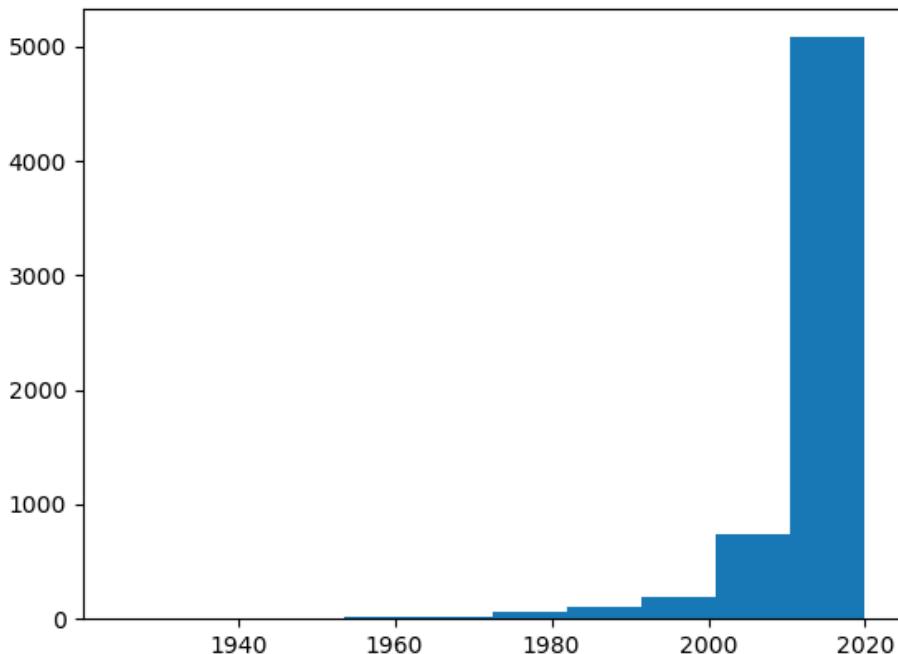
We've got 7787 rows with 12 columns, including the *Cast, Date of Addition, Release Year, Rating*, etc.

³¹<https://www.kaggle.com/shivamb/netflix-shows>

Let's visualize the distribution of the `release_year` feature, to get a feel of how Netflix's expansion has been going up until 2019 and how *novel* are the movies/shows they've been adding. To plot a *Histogram*, we use the `hist()` function on either the PyPlot instance or an `Axes`. It accepts the values on the X-axis, and several arguments that customize the plot. In its bare-bones form - it'll bin the values from `x` into, well, `bins` and plot the frequency of entries in those bins as bars:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('netflix_titles.csv')
5 plt.hist(df['release_year'])
6
7 plt.show()
```

By default, the generated histogram will contain *the count/frequency* of occurrences of these years, and plot bars based on that data. Running this code results in:



Here, the movie *bins* (ranges) are set to 10 years. Each bar here includes all shows/movies in batches of 10 years. For example, we can see that around 750 shows were released between 2000. and 2010. At the same time, 4500 were released between 2010. and 2020.

These are pretty big ranges for the movie industry. It makes more sense to visualize this for ranges smaller than 10 years. Additionally, the tick frequency is a bit off, though, it's not too bad. Let's change the bin size, which will also inherently change the tick frequency.

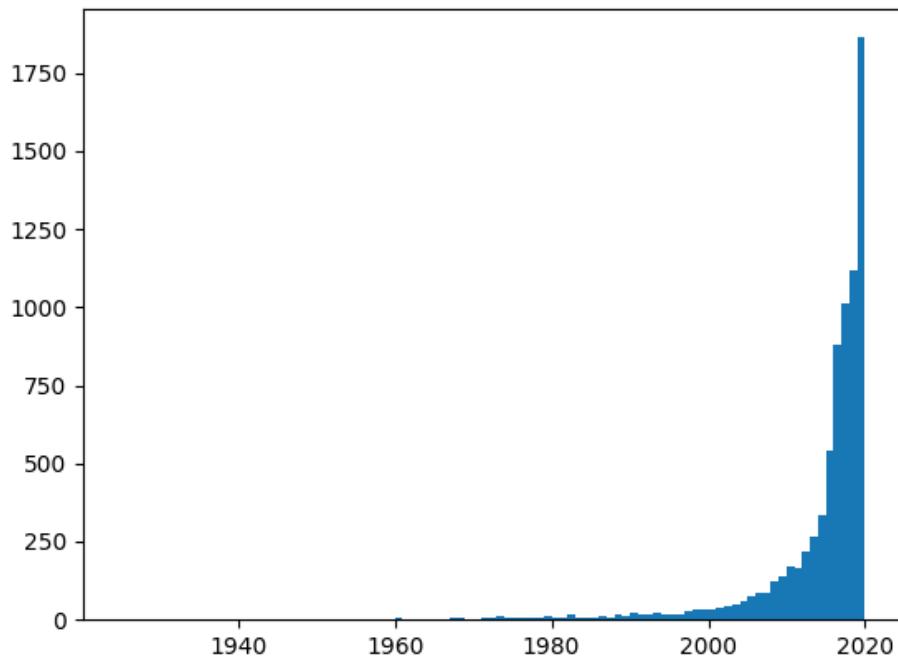
Changing a Histogram's Bin Size

Say, let's visualize a Histogram Plot in batches of 1 year, since this is a much more realistic time-frame for movie and show releases. We'll import `numpy`, as it'll help us calculate the size of the bins:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 df = pd.read_csv('netflix_titles.csv')
6 data = df['release_year']
7
8 plt.hist(data, bins = np.arange(min(data), max(data), 1)) # [1925 ... 2019 2020 2021]
9
10 plt.show()
```

This time around, we've extracted the `DataFrame` column into a `data` variable, just to make it a bit cleaner to look at and passed the `data` to the `hist()` function. The `bins` argument accepts a list of *bins*, which you can set manually, if you'd like, especially if you want a non-uniform bin distribution.

Since we'd like to pool these entries each in the same time-span (1 year), we'll create a Numpy array, that starts with the lowest value (`min(data)`), ends at the highest value (`max(data)`) and goes in increments of 1. This time around, running this code results in:

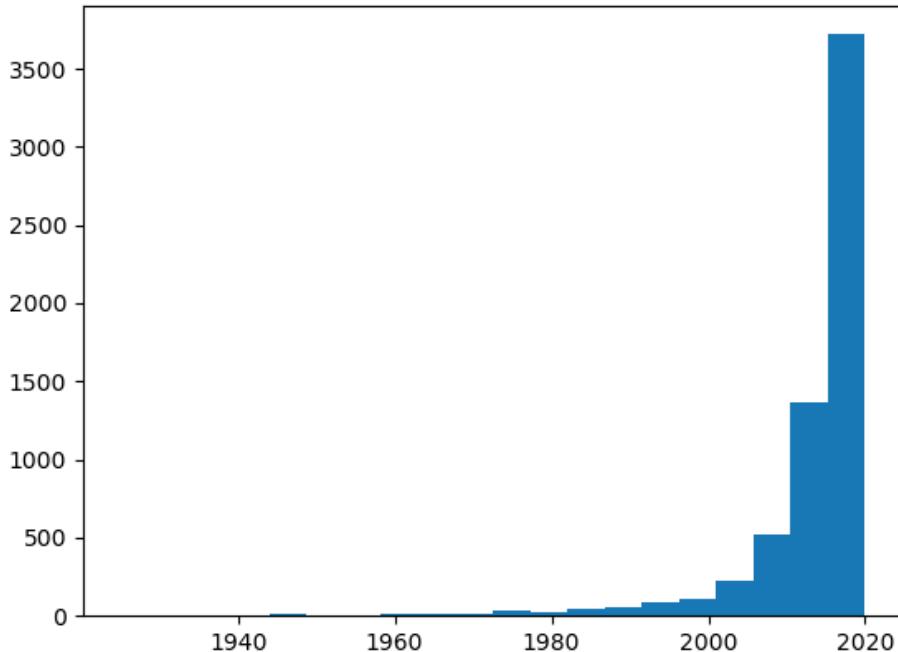


They've got a whole lot of new movies and shows, released between 2010 and 2020. A significant portion of them are *from 2020*, signifying that Netflix focused on released a lot of *novel content*, probably to hedge themselves as the leader in the industry with the newest movies and shows.

If you don't want to dabble with lists for the `bins` value, you can also assign a scalar value which denotes the total number of bins in the plot. Using 1 bin will result in 1 bar for the entire plot. Say, we want to have 20 bins, we'd use:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 df = pd.read_csv('netflix_titles.csv')
6 data = df['release_year']
7
8 plt.hist(data, bins = 20)
9
10 plt.show()
```

This results in 20 equal bins, with data within those bins pooled and visualized in their respective bars:



This results in 5-year intervals, considering we've got 100 years worth of data. Splitting it up in 20 bins means that each will include 5 years worth of data. Though, knowing how big the intervals will be depends on us already knowing how many years of data we've got for this specific dataset.

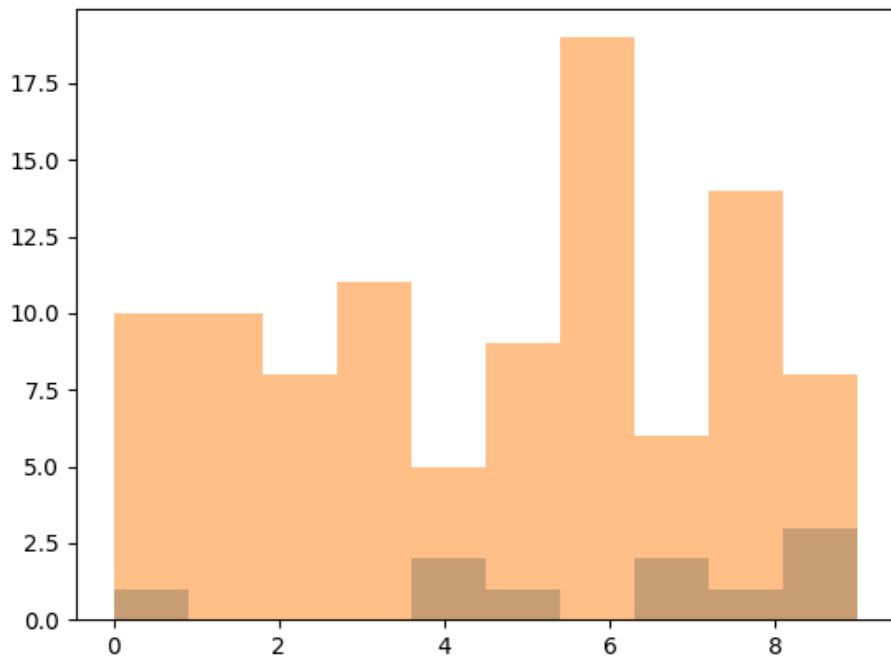
Plotting Histogram with Density Data

Sometimes, instead of the count of the features, we'd want to check what the density of each bar/bin is. That is, how common it is to see a range within a given dataset. Since we're working with 1-year intervals, this'll result in the probability of a movie/show being released in that year, for our dataset.

When plotting with density data - all bins should sum up to 1 (100%) and this gives us a proportional/relative view of the data. For example, if we're comparing the distribution of two datasets, and one has a count of 10000 in its highest bar, while the other has a count of 100 in its highest bar - the second plot will be proportionally small:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 # Create 10 random integers between 0 and 10
6 x_1 = np.random.randint(0, 10, 10)
7 # Create 100 random integers between 0, 10
8 x_2 = np.random.randint(0, 10, 100)
9
10 plt.hist(x_1, alpha=0.5)
11 plt.hist(x_2, alpha=0.5)
12
13 plt.show()
```

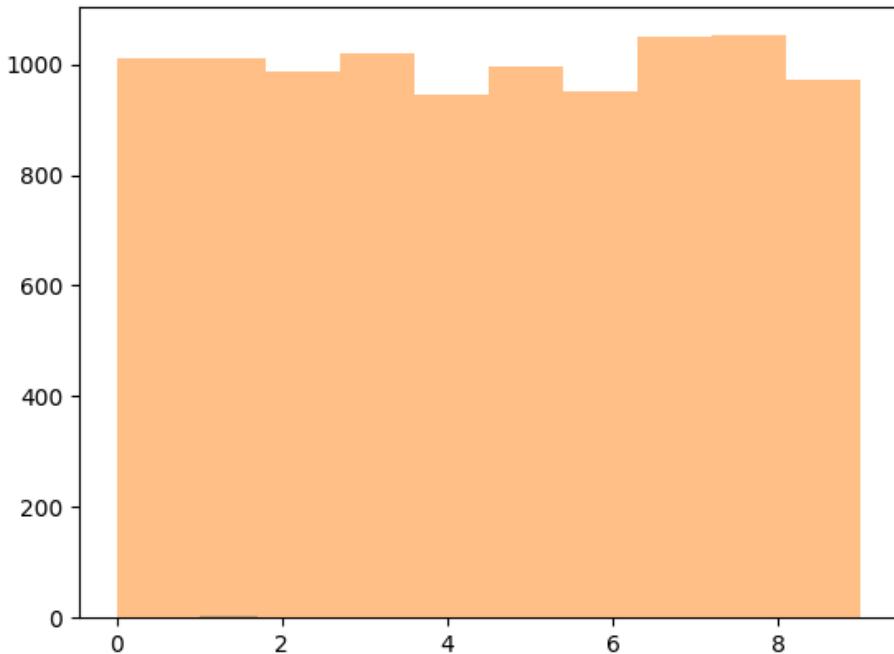
Histograms accept the classic customization options, such as `alpha` to set the transparency. If we run this code, we'll be greeted with:



The much smaller dataset at the bottom is still observable and we can interpret it. Though, what happens when we up the count from 100 to 10000?

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 x_1 = np.random.randint(0, 10, 10)
6 x_2 = np.random.randint(0, 10, 10000)
7
8 plt.hist(x_1, alpha=0.5)
9 plt.hist(x_2, alpha=0.5)
10
11 plt.show()
```

Suddenly, `x_1` becomes so minuscule that we practically can't see it, and even though a *slight* nudge can be seen at $X=1$, it's so small that we can't really interpret anything meaningful:

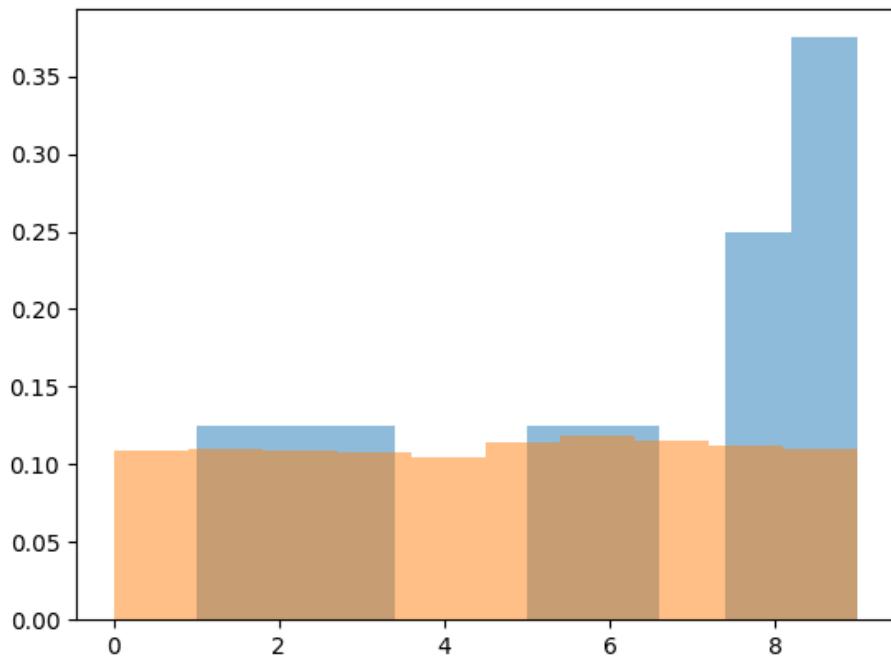


If we're comparing pears and apples like this - it's useful to actually plot the *density*, and not the *count*. This way, we take a look at the *relative data* instead of *absolute data* and even smaller datasets can be compared to larger ones this way. To plot the density, we just set the `density` flag to `True` in the `hist()` function:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 x_1 = np.random.randint(0, 10, 10)
6 x_2 = np.random.randint(0, 10, 10000)
7
8 plt.hist(x_1, alpha=0.5, density=True)
9 plt.hist(x_2, alpha=0.5, density=True)
10
11 plt.show()
```

Now, we can increase `x_2` by orders of magnitude if we'd like - we'll still be able to plot both of these and visually compare them. The densities shown are, of course, the

densities in their respective datasets - `x_1` has a higher *density* of elements with the value of 8, and 7 than `x_2`. This makes sense since with small random samples, we see much more diversity than with larger ones which eventually even out to a large degree, due to the [Law of Large Numbers](#)³²:

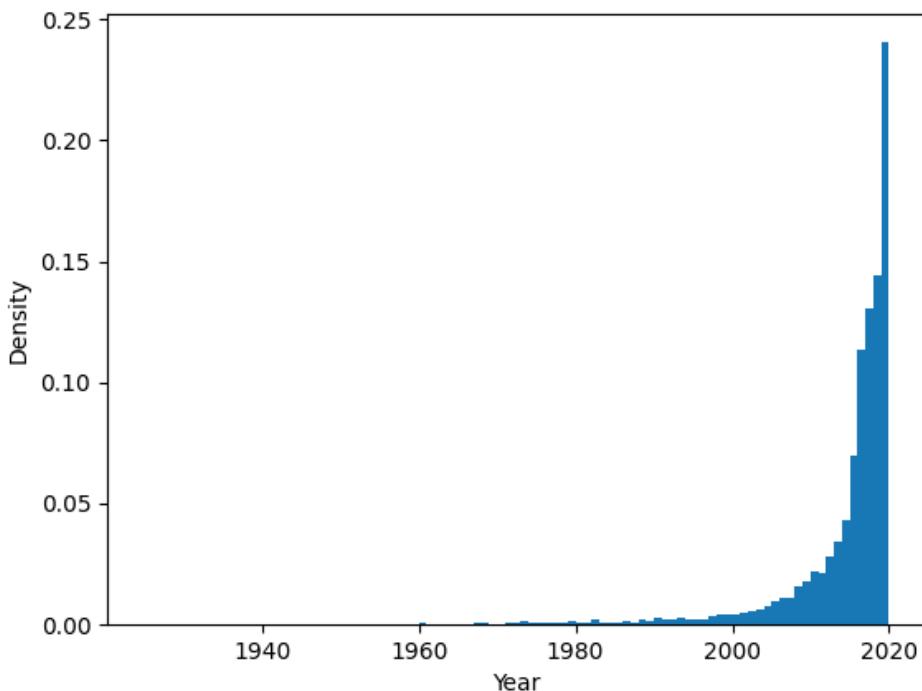


Let's go back to our Netflix dataset, and plot the distribution of `release_year` through a density lens:

³²https://en.wikipedia.org/wiki/Law_of_large_numbers

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 df = pd.read_csv('netflix_titles.csv')
6 data = df['release_year']
7 bins = np.arange(min(data), max(data), 1)
8
9 plt.hist(data, bins = bins, density = True)
10 plt.ylabel('Density')
11 plt.xlabel('Year')
12
13 plt.show()
```

This results in much the same plot, but scaled down to display proportional data, if we were to compare it with a different dataset *or* if we simply wanted to take a look at the density distribution without comparison to another plot;



We can see that around 23% of the entries were released in 2020, followed by 15% in 2019 and 14% in 2018. These three years alone constitute 50% of the shows you'll

encounter on Netflix. Needless to say - they're up to date.

Histogram Plot with KDE

Kernel Density Estimation (KDE) attempts to estimate the density function of a variable. It generally produces a fairly similar shape to a Density Histogram. Due to how common it is to plot these two together, libraries like Seaborn have a simple `kde` flag that you can turn on to display the KDE line alongside the Histogram you're plotting.

Again - this isn't a built-in feature of Matplotlib, and people have been simply creating these two plots together, which is the reason Seaborn even has the feature. Thankfully, Pandas *does* have a `kde()` function.

It might not be what you're thinking of though - it doesn't just compute the KDE - it *visualizes it*. Pandas actually integrates *with Matplotlib*. The `DataFrame` class has various visualization functions, which rely on Matplotlib under the hood for the most part, and require us to use the `PyPlot` instance to actually show the visualizations.

We'll go into more detail on how to plot with Pandas in a later chapter, since it's a topic of its own. For now - the `DataFrame` class has many plotting functions such as `bar()`, `hist()`, `area()`, etc. These methods inherently rely on the `pandas.plotting` module, which we'll look into later. Suffice to say - one of the rare functions that exists in Pandas' plotting module, but *doesn't* exist in Matplotlib is `kde()`.

We can utilize the wonderful integration between Pandas and Matplotlib and use them hand-in-hand to create a Density Histogram with a KDE Line:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 df = pd.read_csv('netflix_titles.csv')
6
7 fig, ax = plt.subplots()
8
9 # Invoke the `plotting` module
10 # And plot a KDE for 'release_year' on `ax`
11 df['release_year'].plot.kde(ax=ax)
12
13 # Invoke the `plotting module`
14 # And plot a Histogram on `ax` with certain arguments set
15 df['release_year'].plot.hist(ax=ax, bins=25, density = True)
16
```

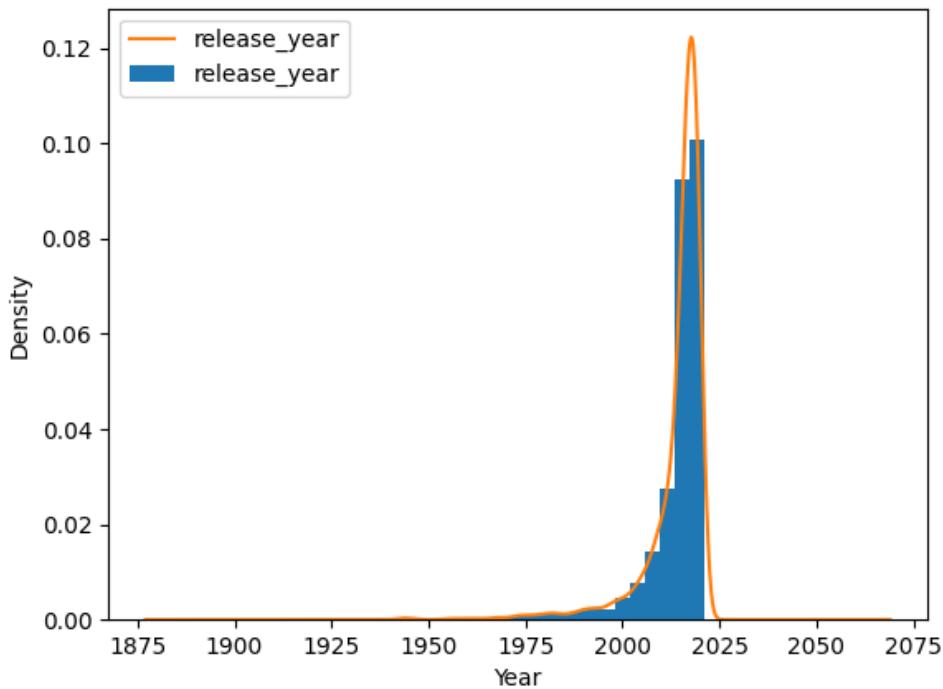
```
17 ax.set_ylabel('Density')
18 ax.set_xlabel('Year')
19
20 plt.show()
```

When plotting *from* a DataFrame, if we don't supply the feature we're plotting for, depending on the available features, Pandas will try to match the features that can be visualized with the function you invoke. In this case, since *only* release_year is fit for Histograms, Pandas would've matched that feature and we wouldn't have to specify it. Though, in practice, it's best to avoid ambiguity and simply declare which feature you'd like to plot.

We could've also used the Axes instance to plot the Histogram:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 df = pd.read_csv('netflix_titles.csv')
6
7 fig, ax = plt.subplots()
8
9 ax.hist(df['release_year'], bins=25, density = True)
10 df['release_year'].plot.kde(ax=ax)
11
12 ax.set_ylabel('Density')
13 ax.set_xlabel('Year')
14
15 plt.show()
```

This results in the same plot:



Customizing Histogram Plots

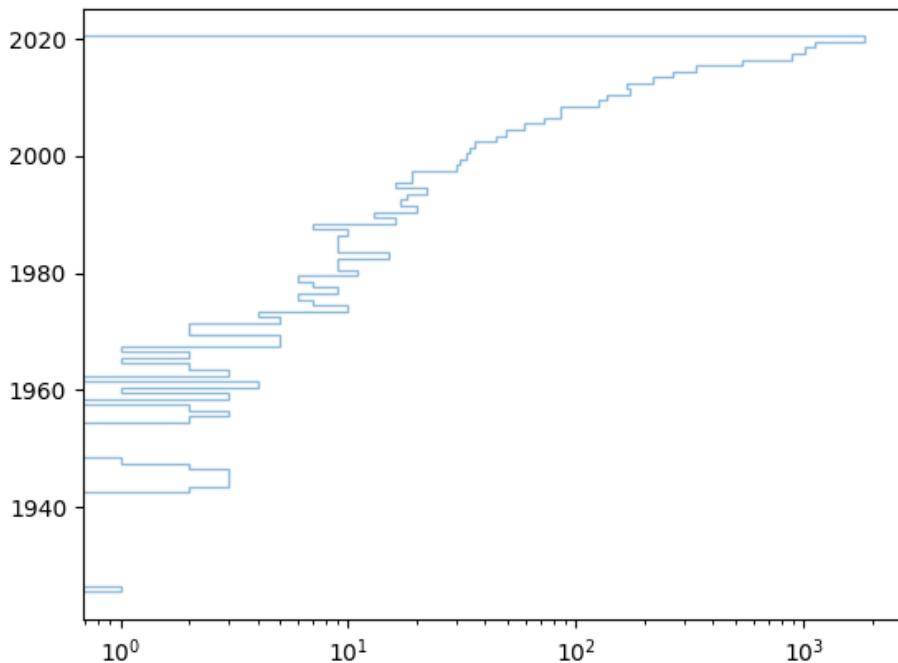
There are several customization options for Histograms, and some of these are the typical options, such as setting `alpha` to control transparency. Though, there are also some Histogram-specific options as well. Let's quickly take a look at some of them, without dragging them out too much:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 df = pd.read_csv('netflix_titles.csv')
6 data = df['release_year']
7 bins = np.arange(min(data), max(data), 1)
8
9 plt.hist(data, bins = bins, histtype = 'step', alpha = 0.5,
10          align = 'right', orientation = 'horizontal', log = True)
11
12 plt.show()
```

Here, we've defined several arguments:

- `bins` - Number of bins in the plot
- `density` - Whether PyPlot uses count or density to populate the plot
- `histtype` - The type of Histogram Plot (default is `bar`, though other values such as `step` or `stepfilled` are available)
- `alpha` - The alpha/transparency of the lines and bars
- `align` - To which side of the bins are the bars aligned, default is `mid`
- `orientation` - Horizontal/Vertical orientation, default is `vertical`
- `log` - Whether the plot should be put on a logarithmic scale or not

This now results in:



Since we've put the `align` to `right`, we can see that the bar is offset a bit, to the vertical right of the 2020 bin.

Box Plot

We've gotten familiar with data distributions through Histograms. Another very useful, albeit a bit less known plot type that can visualize distributions are *Box Plots*.

Box Plots are used to visualize summary statistics of a dataset, displaying attributes of the distribution like the data's range and distribution.

Although being used to visualize data distributions - they don't have the same job as Histograms. Box Plots are used to visualize *summaries* of the data through *quartiles*.

A *quartile* is just a type of *quantile*, which represents one *fourth* of the data. A well-known *quantile* is the *percentile* which represents one *hundredth* of the data, and a *decile* which represents one *tenth* of the data.

That is to say - a *quartile* represents a step at 25% of the dataset. If we divide the dataset into *quartiles*, they're divided into Q_1 (25%), Q_2 (50%), Q_3 (75%). Q_2 is effectively the *median* of a dataset, while Q_1 is the middle value between Q_2 and the minimum value, and Q_3 is the middle value between Q_2 and the maximum value.

The *quartiles* at 0% and 100% are simply called the “*minimum*” and “*maximum*” values of the dataset, and although they’re quartiles as well - they don’t usually have a Q_n name.

The range between Q_1 and Q_3 is known as the *Interquartile Range* (IQR).

Box Plots get their name from taking the shape of a *box*, with the distribution data within the boxes. The boxes are bound from Q_1 to Q_3 - essentially, the boxes *are* the interquartile range.

Additionally, they typically have *whiskers* - lines extending from the box that let us know what variance we can expect beyond Q_3 and below Q_1 , culminating in the “*minimum*” and “*maximum*” of the dataset, which are usually marked at the ends of the whiskers.

We’ve put “*minimum*” and “*maximum*” under quotation marks for a reason - they aren’t the *actual* minimum and maximum of the dataset. They’re the *minimum and maximum* left *after* we’ve taken out the outliers, which will typically be the actual minimums and maximums.

Outliers are detected using the *IQR Method*, namely - if they’re located $1.5 \times \text{IQR}$ below Q_1 or $1.5 \times \text{IQR}$ above Q_3 :

$$\text{IQR} = Q_3 - Q_1$$

$$\text{Lower outliers} < Q_1 - 1.5 \times \text{IQR}$$

$$\text{Upper outliers} < Q_3 + 1.5 \times \text{IQR}$$

Why 1.5? The [68–95–99.7 rule](#)³³ postulates that any data that fits within a normal distribution will be within three standard deviations from the mean. That is, 68%

³³https://en.wikipedia.org/wiki/68%25-80%25-90%25_rule

of the data will be present in the first standard deviation, 95% within the first and second and 99.7% within the first, second and third standard deviation of the mean. 99.7% of the data is close enough to 100% to safely say that the remaining 0.3% are most likely outliers.

When calculating the “maximum” and “minimum” these 0.3% are ignored. Outliers are very easy to spot with Box Plots, as they’re isolated outside the boxes themselves, and extend beyond the whiskers.

Box Plots are really the most useful for comparison purposes, since they put things in the same perspective. You’ll usually plot more than one Box Plot of several features to compare their summaries (outliers, interquartile ranges, etc). Then, you can go further with other plot types.

Importing Data

Since Box Plots provide us with summary statistics - we’ll need *numerical data* to work with. We’ll be working with the [Red Wine Quality](#)³⁴ dataset from Kaggle. It contains data on Portuguese “Vinho Verde” wine, such as its *acidity*, *chlorides*, etc., as well as a subjective *quality* mark given by subjects who tested the wine.

Let’s import the dataset and take a peek at what’s inside, checking for `null` values just in case any have crepted in:

```
1 import pandas as pd
2
3 df = pd.read_csv("winequality-red.csv")
4 print(df)
5 print(df.isnull().values.any())
```

³⁴<https://www.kaggle.com/uciml/red-wine-quality-cortez-et-al-2009>

```

1      fixed acidity  volatile acidity  citric acid  ...  sulphates  alcohol  quality
2  0            7.4            0.700        0.00  ...        0.56      9.4       5
3  1            7.8            0.880        0.00  ...        0.68      9.8       5
4  2            7.8            0.760        0.04  ...        0.65      9.8       5
5  3           11.2            0.280        0.56  ...        0.58      9.8       6
6  4            7.4            0.700        0.00  ...        0.56      9.4       5
7  ...
8  1594          ...            ...        ...  ...        ...      ...     ...
9  1595          6.2            0.600        0.08  ...        0.58     10.5       5
10 1596          5.9            0.550        0.10  ...        0.76     11.2       6
11 1597          6.3            0.510        0.13  ...        0.75     11.0       6
12 1598          5.9            0.645        0.12  ...        0.71     10.2       5
13
14 [1599 rows x 12 columns]
15 False

```

The second print statement returns `False`, which means that there isn't any missing data. We've got 1599 rows with 12 columns, culminating with a `quality` feature. This does call for relationship and correlation analysis such as the ones we can do with Scatter Plots. As a matter of fact - since we've gotten the chance to work with Scatter Plots, Histograms and Kernel Density Estimations - we'll be utilizing those here as well to explore the dataset alongside Box Plots.

Plotting a Box Plot

Since we'll be working with several features here, both for Box Plots and later on for other plot types - since we don't want to repeatedly reference the names through a `DataFrame`, let's extract a few of them beforehand so we can reference them without a bulky `DataFrame` call each time:

```

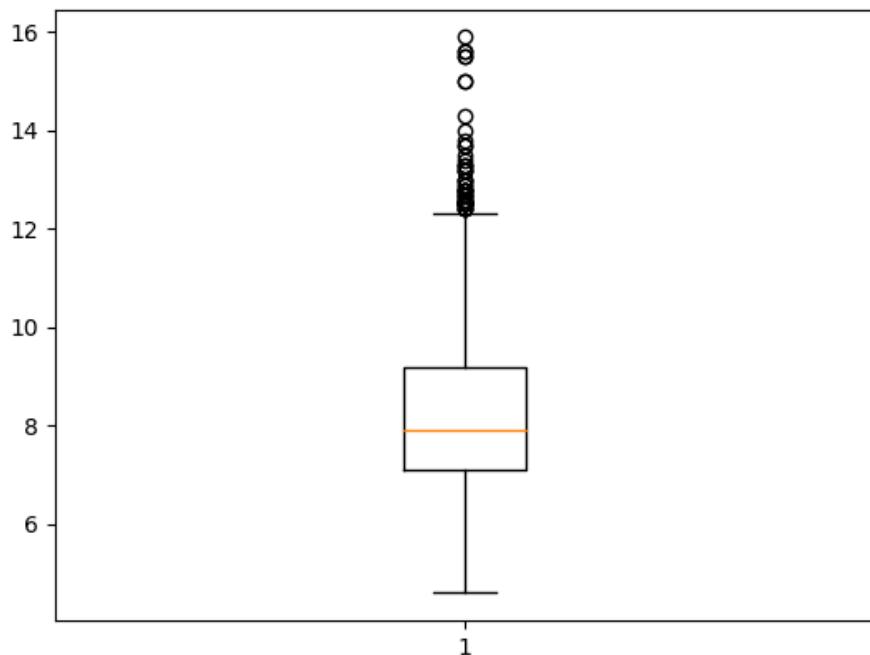
1 fixed_acidity = df["fixed acidity"]
2 free_sulfur_dioxide = df['free sulfur dioxide']
3 total_sulfur_dioxide = df['total sulfur dioxide']
4 alcohol = df['alcohol']

```

As usual, we can call plotting functions on the `PyPlot` instance (`plt`) or `Axes` instance. To plot a Box Plot, we use the `boxplot()` function and pass in the data. Since it requires just one numerical feature, we can either assign it to `x` or just pass in the feature as is since there can't be much ambiguity as to what it might be:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv("winequality-red.csv")
5
6 fixed_acidity = df["fixed acidity"]
7 free_sulfur_dioxide = df['free sulfur dioxide']
8 total_sulfur_dioxide = df['total sulfur dioxide']
9 alcohol = df['alcohol']
10
11 fig, ax = plt.subplots()
12 ax.boxplot(fixed_acidity)
13 plt.show()
```

This results in a simple, yet effective and informative Box Plot:



The Box Plot shows the median of the dataset (the vertical line in the middle), as well as the interquartile ranges (the ends of the boxes) and the minimum and maximum values of the chosen dataset feature (the far end of the “whiskers”).

When plotting, we can save the returned object of the plotting methods which contains various data on the plot itself. When we're dealing with something like a Box Plot, which has several distinct *parts*, there are a few we can extract from this object. You typically won't be accessing these, especially since it's not really meant to be accessed. There's simply no need to access these, especially since they're plotted right in front of your eyes.

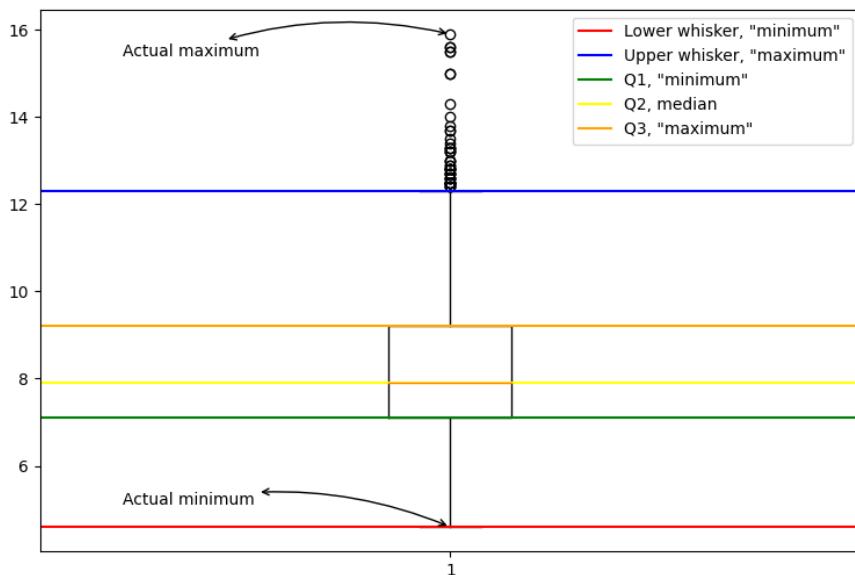
Then again, there's nothing stopping us from dissecting this object, and using the returned values to draw lines and add annotations to explain what we're seeing a bit more:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv("winequality-red.csv")
5
6 fixed_acidity = df["fixed acidity"]
7 free_sulfur_dioxide = df['free sulfur dioxide']
8 total_sulfur_dioxide = df['total sulfur dioxide']
9 alcohol = df['alcohol']
10
11 fig, ax = plt.subplots()
12
13 # Return the Box that we've plotted
14 # This object contains various data that we can extract, like the values
15 # of the lower and upper whiskers, quartiles, etc...
16 box = ax.boxplot(fixed_acidity)
17
18 # Extract the median value of the Box Plot as a list, so we truncate it to a single value
19 median = box['medians'][0].get_ydata()[0]
20
21 # Extract the upper and lower whiskers of the Box Plot
22 upper_whisker = box['whiskers'][1].get_ydata()[1]
23 lower_whisker = box['whiskers'][0].get_ydata()[1]
24
25 # Extract the new minimums and maximums, after removing outliers
26 new_min = box['boxes'][0].get_ydata()[1]
27 new_max = box['boxes'][0].get_ydata()[2]
28
29 # Calculate the actual minimum and maximum
30 min = fixed_acidity.min()
31 max = fixed_acidity.max()
32
33 ax.axhline(lower_whisker, 0, 1, label='Lower whisker, "minimum"', color='red')
34 ax.axhline(upper_whisker, 0, 1, label='Upper whisker, "maximum"', color='blue')
35
36 ax.axhline(new_min, 0, 1, label='Q1, "minimum"', color='green')
37 ax.axhline(median, 0, 1, label='Q2, median', color='yellow')
38 ax.axhline(new_max, 0, 1, label='Q3, "maximum"', color='orange')
39
40 ax.annotate('Actual maximum',
41             xy=(1, max),
42             xytext=(0.6, max-0.5),
43             arrowprops=dict(arrowstyle='->')
```

```
44                                         connectionstyle='arc3, rad=-0.15'))  
45  
46 ax.annotate('Actual minimum',  
47             xy=(1, min),  
48             xytext=(0.6, min+0.5),  
49             arrowprops=dict(arrowstyle='<->',  
50                           connectionstyle='arc3, rad=-0.15'))  
51  
52 ax.legend()  
53 plt.show()
```

The box['key'] calls all return lists, so we're just accessing the adequate elements in those lists that interest us. The get_ydata() functions of these *also* return lists, so we're again, accessing elements that we're interested in. Again, this approach is fairly crude and we're dissecting an object for its values in a non-elegant way - but this isn't something you'd typically be doing anyway.

Running this code results in:



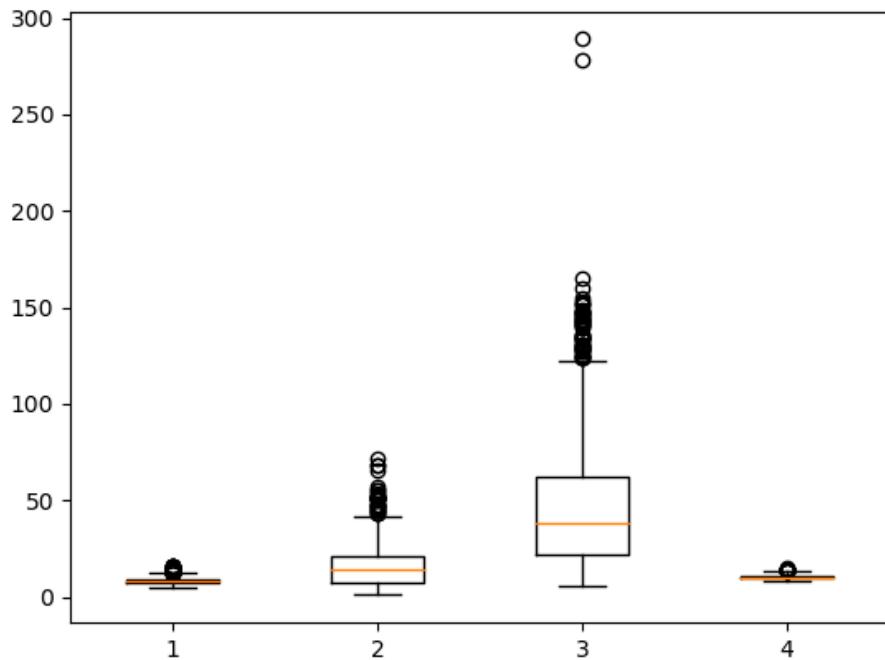
There aren't any outliers below the "minimum" - the actual minimum is the "minimum". On the other hand, the actual maximum is way above the "maximum",

which we can also see from the individually plotted entries above the upper whisker.

We can also plot multiple features on one figure, simply by providing a list of them. This again, can be done on either the `plt` instance, the `fig` object or the `ax` object:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 dataframe = pd.read_csv("winequality-red.csv")
5
6 fixed_acidity = dataframe["fixed acidity"]
7 free_sulfur_dioxide = dataframe['free sulfur dioxide']
8 total_sulfur_dioxide = dataframe['total sulfur dioxide']
9 alcohol = dataframe['alcohol']
10
11 columns = [fixed_acidity, free_sulfur_dioxide, total_sulfur_dioxide, alcohol]
12
13 fig, ax = plt.subplots()
14 ax.boxplot(columns)
15 plt.show()
```

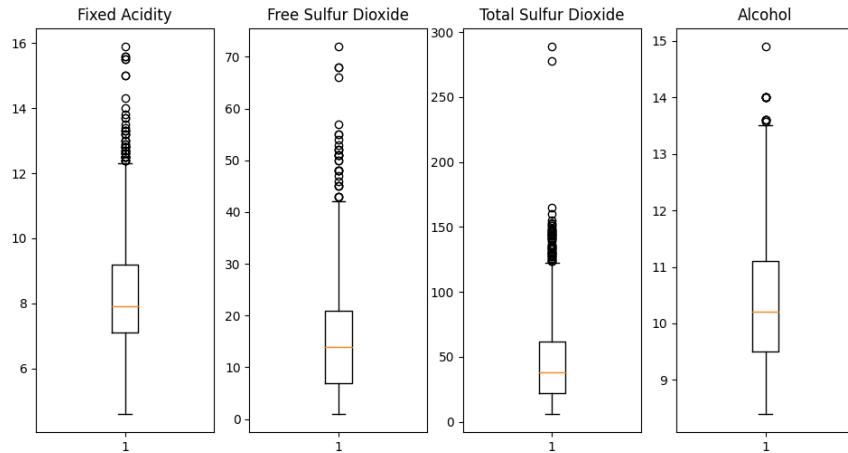
This results in:



Then again, we need to be careful to plot numerical features which have a similar range of values, otherwise some features, like the `alchohol` feature will be too small for us to interpret. This is a common scenario - since we have no guarantee that these features will be on the same scale at all.

In such cases, you'll have to plot these in different Axes instances:

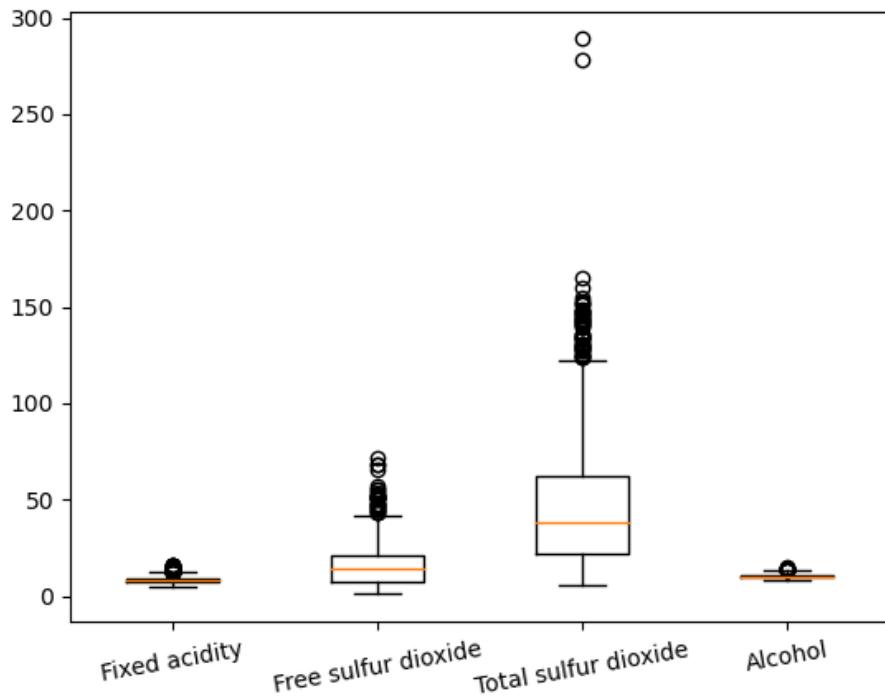
```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 dataframe = pd.read_csv("winequality-red.csv")
5
6 fig, ax = plt.subplots(nrows=1, ncols=4)
7
8 fixed_acidity = dataframe["fixed acidity"]
9 free_sulfur_dioxide = dataframe['free sulfur dioxide']
10 total_sulfur_dioxide = dataframe['total sulfur dioxide']
11 alcohol = dataframe['alcohol']
12
13 ax[0].boxplot(fixed_acidity)
14 ax[0].set_title('Fixed Acidity')
15 ax[1].boxplot(free_sulfur_dioxide)
16 ax[1].set_title('Free Sulfur Dioxide')
17 ax[2].boxplot(total_sulfur_dioxide)
18 ax[2].set_title('Total Sulfur Dioxide')
19 ax[3].boxplot(alcohol)
20 ax[3].set_title('Alcohol')
21
22 plt.show()
```



Customizing Box Plots

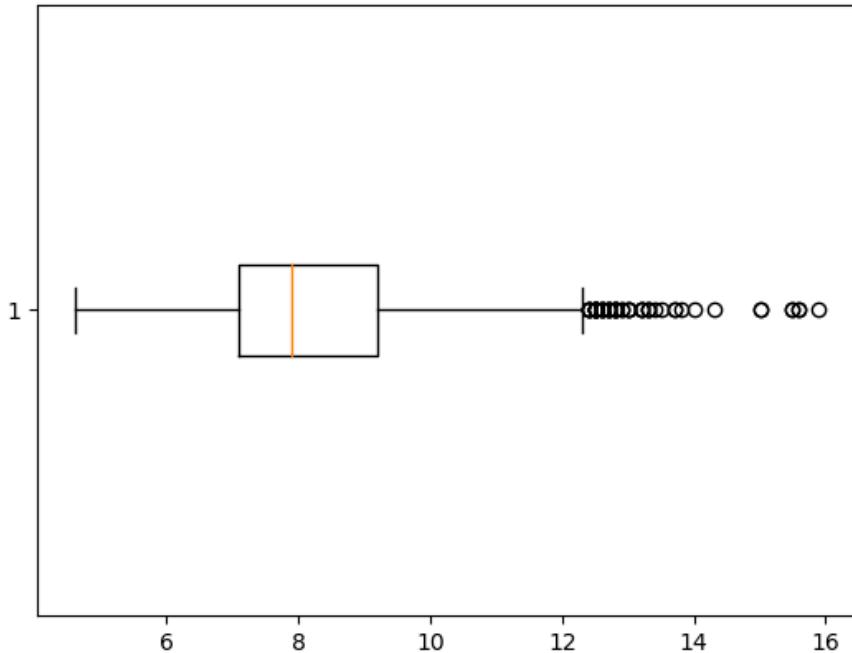
While the plots have successfully been generated, without tick labels on the X and Y-axis, it is difficult to interpret the graph. We can customize the plot and add labels to the X-axis by using the `xticks()` function. Let's pass in the number of labels we want to add and then the labels for each of those columns:

```
1 fig, ax = plt.subplots()
2 ax.boxplot(columns)
3 plt.xticks([1, 2, 3, 4],
4            ["Fixed acidity", "Free sulfur dioxide", "Total sulfur dioxide", "Alcohol"],
5            rotation=10)
6 plt.show()
```



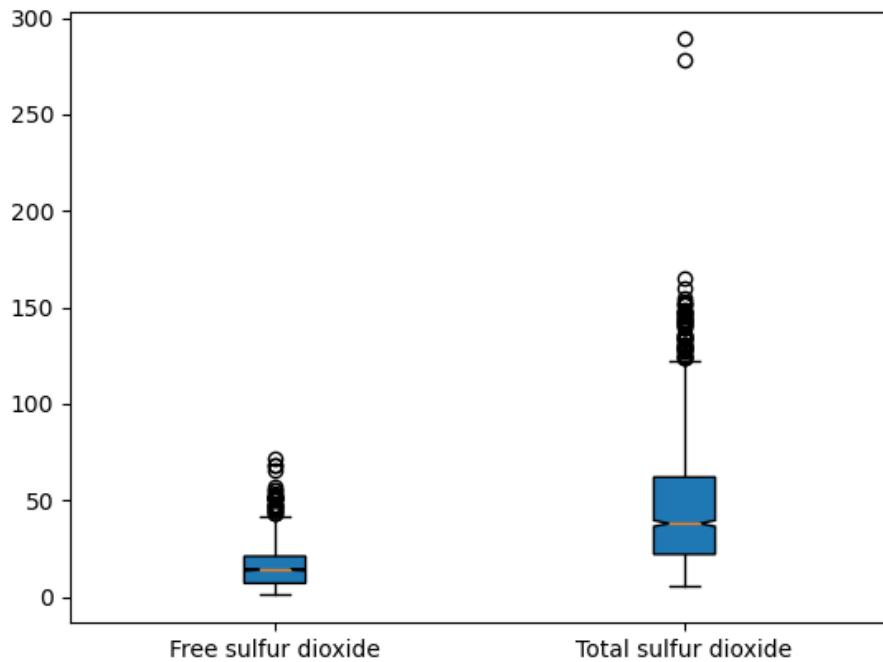
If we wanted to we could also change the orientation of the plot by altering the `vert` parameter. `vert` controls whether or not the plot is rendered vertically and it is set to 1 by default:

```
1 fig, ax = plt.subplots()
2 ax.boxplot(fixed_acidity, vert=0)
3 plt.show()
```



The `notch=True` attribute creates the notch format to the box plot, `patch_artist=True` fills the boxplot with colors:

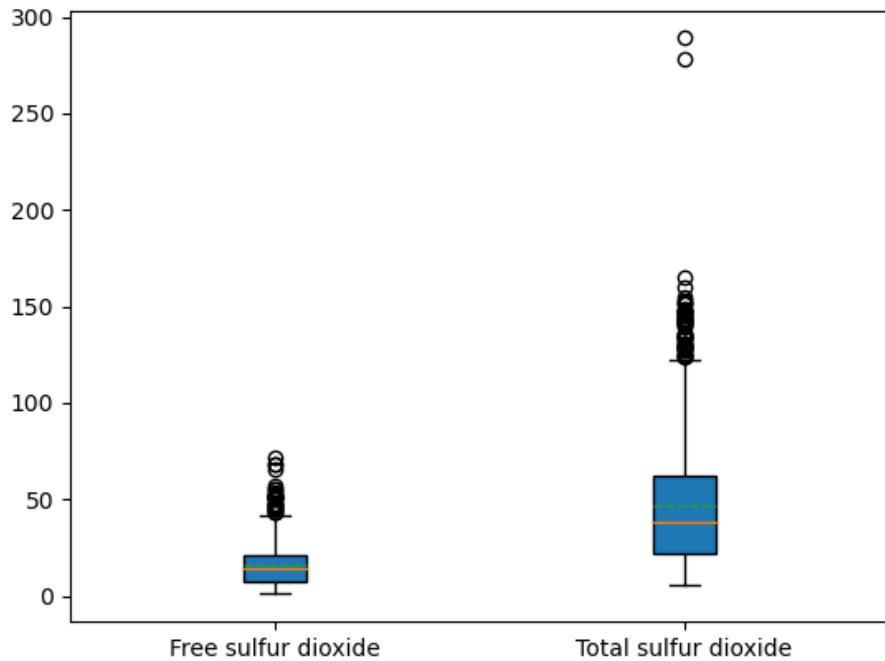
```
1 fig, ax = plt.subplots()
2 columns = [free_sulfur_dioxide, total_sulfur_dioxide]
3 ax.boxplot(columns, notch=True, patch_artist=True)
4 plt.xticks([1, 2], ["Free sulfur dioxide", "Total sulfur dioxide"])
5 plt.show()
```



We can make use of the `meanline` argument to render the mean on the box, although this should be avoided if we are also showing notches, since they can conflict.

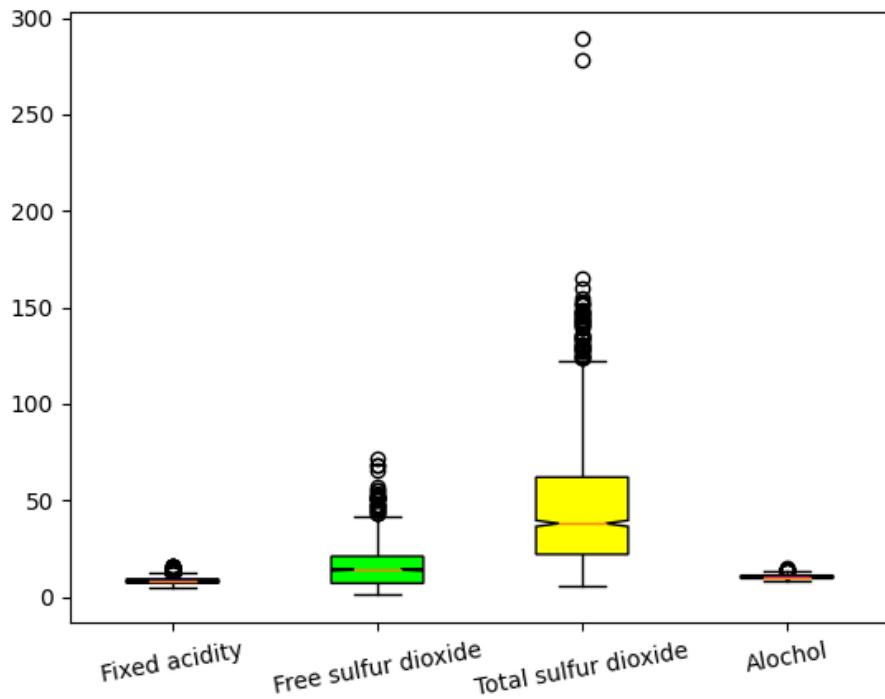
This must be combined with the `showmeans` parameter. If possible, the mean will be visualized as a line that runs all the way across the box. If not possible, the mean will be shown as points:

```
1 fig, ax = plt.subplots()
2 columns = [free_sulfur_dioxide, total_sulfur_dioxide]
3 ax.boxplot(columns, patch_artist=True, meanline=True, showmeans=True)
4 plt.xticks([1, 2], ["Free sulfur dioxide", "Total sulfur dioxide"])
5 plt.show()
```



We can color the Box Plots for different feature columns by creating a list of color values and using the `set_facecolor()` function. In the example below, we `zip()` the `boxes` element of the `box` object together with the colors we want to use and then set the face color for each of those boxes:

```
1  columns = [fixed_acidity, free_sulfur_dioxide, total_sulfur_dioxide, alcohol]
2  fig, ax = plt.subplots()
3  box = ax.boxplot(columns, notch=True, patch_artist=True)
4  plt.xticks([1, 2, 3, 4],
5           ["Fixed acidity", "Free sulfur dioxide", "Total sulfur dioxide", "Alcohol"],
6           rotation=10)
7
8  colors = ['#0000FF', '#00FF00',
9            '#FFFF00', '#FF00FF']
10
11 for patch, color in zip(box['boxes'], colors):
12     patch.set_facecolor(color)
13
14 plt.show()
```

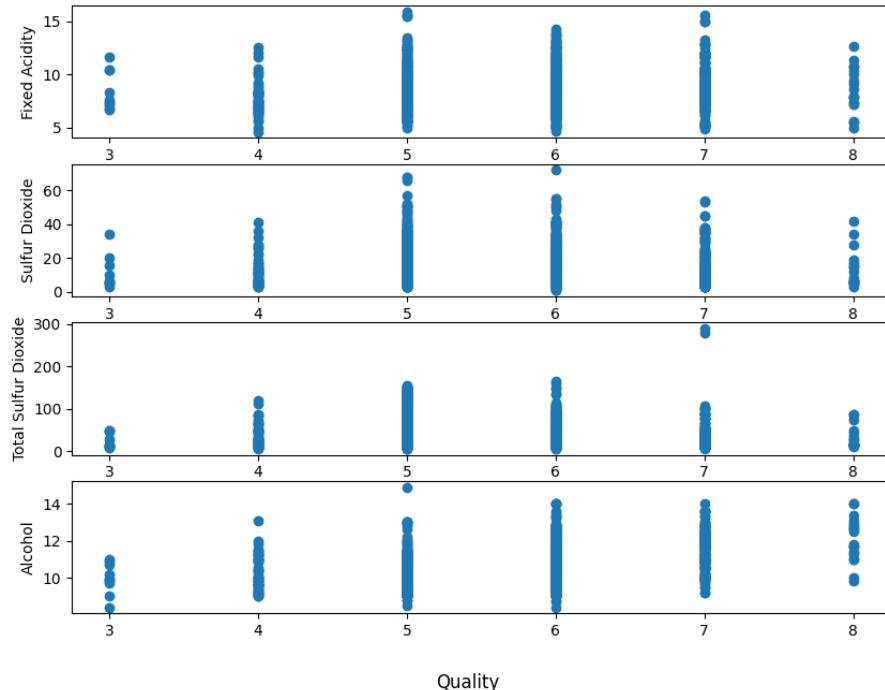


Exploring Relationships with Scatter Plots, Histograms and Box Plots

Let's pick out a feature and perform a relationship analysis between it and some other feature, as well as its distribution. For example, let's see test a few features for correlation with the *Quality* feature. If we do notice some positive correlation, we'll explore that feature further:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv("winequality-red.csv")
5
6 fixed_acidity = df["fixed acidity"]
7 free_sulfur_dioxide = df['free sulfur dioxide']
8 total_sulfur_dioxide = df['total sulfur dioxide']
9 alcohol = df['alcohol']
10 quality = df['quality']
11
12 fig, ax = plt.subplots(nrows=4)
13
14 fig.supxlabel('Quality')
15
16 ax[0].scatter(x=quality, y=fixed_acidity)
17 ax[0].set_ylabel('Fixed Acidity')
18
19 ax[1].scatter(x=quality, y=free_sulfur_dioxide)
20 ax[1].set_ylabel('Sulfur Dioxide')
21
22 ax[2].scatter(x=quality, y=total_sulfur_dioxide)
23 ax[2].set_ylabel('Total Sulfur Dioxide')
24
25 ax[3].scatter(x=quality, y=alcohol)
26 ax[3].set_ylabel('Alcohol')
27
28 plt.show()
```

This will show us the relationship between these 4 features and the *Quality* feature, giving us a peek at what *might* be impacting the subjective experience of drinking Portuguese “Vinho Verde” wine:



Not much correlation can be seen in the first three plots, indicating absolutely no connection between these features and the *Quality*. However, when it comes to the *Alcohol* feature, there seems to be a relatively weak correlation for higher-quality wine. Does a higher percentage of alcohol *produce higher quality wine*? Or does higher-quality wine usually have more alcohol than lower-quality wine?

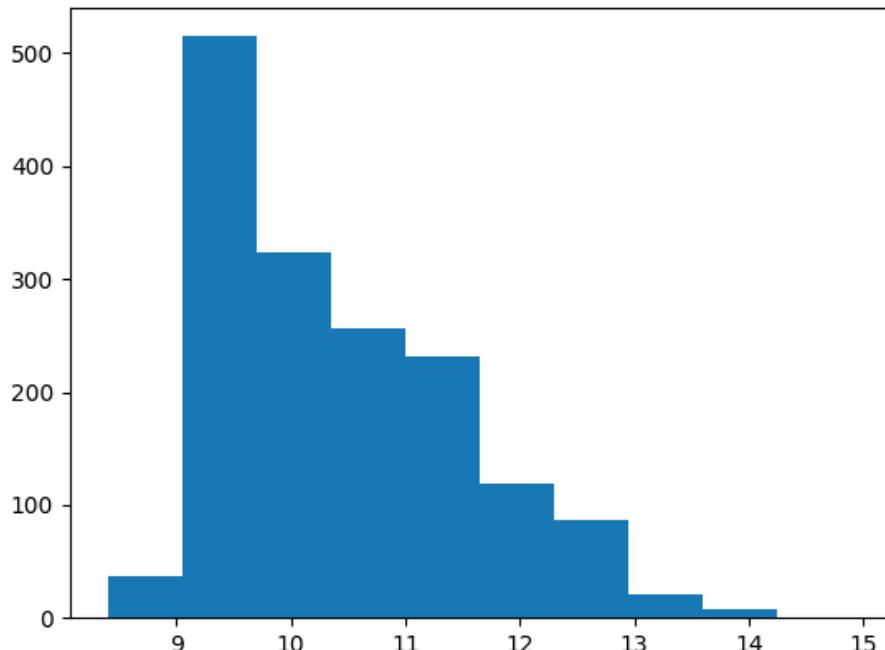
We can't talk much about causality here, but there does seem to be some correlation between the two. The *Alcohol* feature is the feature we haven't seen much due to the range of its values.

Let's explore this feature through a Histogram as well. If it's common to have higher alcohol percentages, then this correlation is just due to fluke coincidence. Though, if the higher percentage of alcohol is relatively *rare*, and this type of wine got a higher quality rating than others, then we might actually find more connection between these features. Now, we can already see some distribution of these visually, through the Scatter Plot (a bunch of markers on Y=6, for example), though, let's not eye-ball

and visualize this for good measure:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv("winequality-red.csv")
5
6 alcohol = df['alcohol']
7 fig, ax = plt.subplots()
8 ax.hist(alcohol)
9
10 plt.show()
```

Now, let's see a more accurate representation of the distributions:



We might be onto something. While a good portion of wine does have over 12% of alcohol, it's still fairly less common than the wine that has less than 12%, which is the majority of it.

Now, we don't exactly see the quality here anymore, so we should refer to the previous plot for this, or we can plot them together on the same Figure for easier viewing. This is actually the basis of a *Joint Plot*, which we'll be covering next. It's not too uncommon to plot a Scatter Plot with marginal Histograms that give us an accurate view of the distributions of the data whose relationships we're plotting instead of judging the distributions by eye. Our example is *exactly* a case where this would be really useful. Before fixing this issue with an elegant solution - let's make do with what we've got right now.

We'll also plot a horizontal Box Plot, to match our Histogram. This Box Plot will allow us to visually gauge where the median is, as well as the quartiles of this distribution/Histogram. In a sense, we'll be supplementing the Histogram with the Box Plot, obtaining the distribution summary from the Box Plot which we wouldn't have from a Histogram otherwise. We'll also be supplementing the Box Plot with the Histogram, since Box Plots show us the quartiles, but not the distribution *within them* - i.e. the bars of a Histogram.

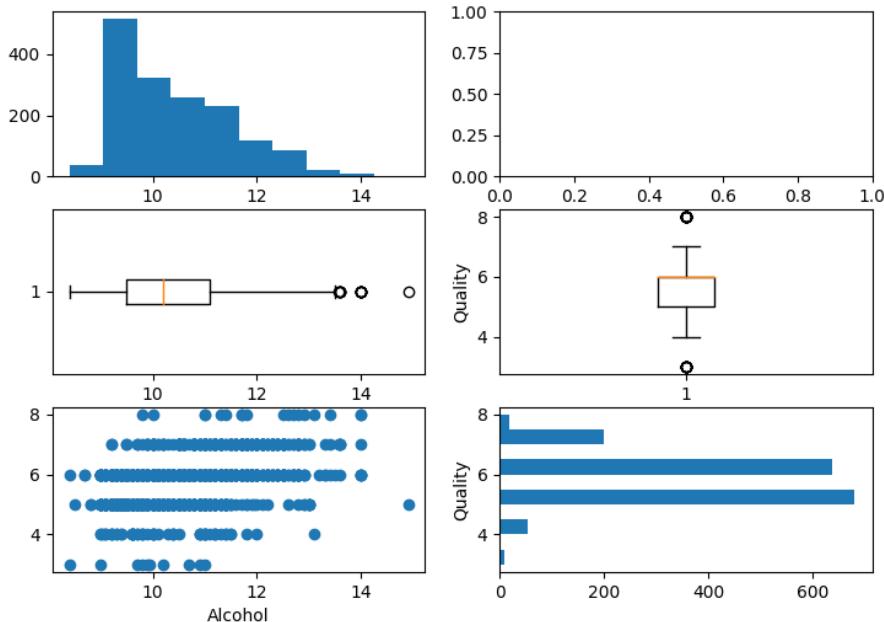
This symbiosis of Histograms and Box Plots actually gave birth to a new plot type - *Violin Plots*, which combine the strengths of **both**. We'll be covering Violin Plots after Joint Plots.

Though, let's not get ahead of ourselves - both of these new plot types are very intuitive when visualized - especially if we understand the constituent parts - Histograms, Scatter Plots and Box Plots:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv("winequality-red.csv")
5
6 alcohol = df['alcohol']
7 quality = df['quality']
8
9 fig, ax = plt.subplots(3, 2)
10
11 ax[0][0].hist(alcohol)
12 ax[1][0].boxplot(alcohol, vert=0)
13 ax[2][0].scatter(x = alcohol, y = quality)
14 ax[2][0].set_xlabel('Alcohol')
15
16 ax[1][1].boxplot(quality)
17 ax[2][1].hist(quality, orientation='horizontal')
18 ax[2][1].set_xlabel('Quality')
19
20 plt.show()
```

This isn't exactly how you'd want to visualize this - for this, we use Joint Plots (covered next), since the alternative we're using has a clunky user experience, an empty plot and just doesn't look good. Not to mention - plotting the data like this makes it non-optimal for us to visually gauge the correlations between certain plots, but it's better than nothing.

All of these issues will be rectified soon enough, but before that, let's take a look at what's generated by this code:



While admittedly ugly, we can still work out some details here. The left side of the Figure is reserved for the alcohol percentages. The top plot is the distribution of the *Alcohol* feature. The second plot is a distribution summary of the *Alcohol* feature. The third plot is the relationship between *Alcohol* and *Quality* features.

Since these three share the same X-axis, we can compare them directly. If we draw a line on, say, X=10 on the Scatter Plot up to the Histogram - we'll see the number of occurrences for that quality. When looking at the Scatter Plot as-is, this isn't that clear, since we just see a lot of markers overlapping. When paired with a Histogram,

we've got a much better idea of what's going on.

At the same time, we can see the *median* of the `alcohol` feature through the Box Plot between these - as well as the outliers and maximum/minimum value when we take the outliers away.

25% of the wine falls into the category *after Q3* (as seen on the Box Plot). This means that a smaller portion of the wine contains higher alcohol percentages. Coupling that with the “maximum” value, omitting the outliers, we can, with a certain degree of confidence, say that there is a correlation between the `alcohol` feature and the `quality` feature, though, it's not strong.

This *still* doesn't speak much about causality, but at least we've proven that it isn't fluke coincidence by taking the distribution of the `alcohol` feature as well as the outliers and quartiles into account as well.

The right side of the plot contains a Histogram and Box Plot that both visualize the distribution of the `quality` variable. This is important because we can observe that the median quality is 6, and that only a small amount of wine got a mark of 8.

This just further proves that the weak correlation isn't total coincidence, but that there is some genuine correlation there.

Now, since this plot really isn't really nicely plotted, let's explore the *Joint Plot* - which organizes plots like these in a much nicer fashion.

Scatter Plot with Marginal Distributions (Joint Plot)

Joint Plots are used to explore relationships between bivariate data, as well as their distributions at the same time.

Really, Joint Plots are just Scatter Plots with accompanying Distribution Plots (Histograms, Box Plots, Violin Plots) on both axes of the plot, to explore the distribution of the variables that constitute the Scatter Plot itself. The name *Joint Plot* was actually coined by the Seaborn team, when they introduced the convenience plot type that combines multiple plots on the same Figure. Following the popularity of Joint Plots - other libraries such as Bokeh and Pandas also introduced them.

Note: This sort of task is *much more fit* for libraries such as Seaborn, which has a built-in `jointplot()` function. With Matplotlib, we'll construct a Joint Plot manually, using `GridSpec` and multiple `Axes` objects, instead. The `GridSpec` is used for advanced customization of the subplot grid on a `Figure`. We'll be covering it in more detail in the next chapter.

You can put any distribution plot on these axes - we'll opt for Histograms and Box Plots, since Violin Plots are a combination of the two and can be used to replace *both*.

Importing Data

We'll use the famous [Iris Dataset³⁵](#), which contains data on the Iris genus, such as `SepalWidthCm` and `SepalLengthCm`, as well as their `Species`.

Let's import the dataset and take a peek:

```
1 import pandas as pd
2
3 df = pd.read_csv('iris.csv')
4 print(df.head())
```

This results in:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
1	0	5.1	3.5	1.4	0.2	Iris-setosa
2	1	4.9	3.0	1.4	0.2	Iris-setosa
3	2	4.7	3.2	1.3	0.2	Iris-setosa
4	3	4.6	3.1	1.5	0.2	Iris-setosa
5	4	5.0	3.6	1.4	0.2	Iris-setosa

We'll be exploring the bivariate relationship between the `SepalLengthCm` and `SepalWidthCm` features here, but also their distributions. We can approach this in two ways - with respect to their `Species` or not.

We can totally disregard the `Species` feature, and simply plot the distributions of each flower instance. On the other hand, we can color-code and plot distribution plots of each flower instance, highlighting the difference in their `Species` as well.

We'll explore *both* options here, starting with the simpler one - disregarding the `Species` altogether.

³⁵<https://www.kaggle.com/uciml/iris>

Plotting a Joint Plot with Single-Class Histograms

In the first approach, we'll just load in the flower instances and plot them as-is, with no regard to their Species.

We'll be using the `GridSpec` to customize our figure's layout, to make space for three different plots and `Axes` instances. To invoke the `GridSpec` constructor, we'll want to import it alongside the `PyPlot` instance:

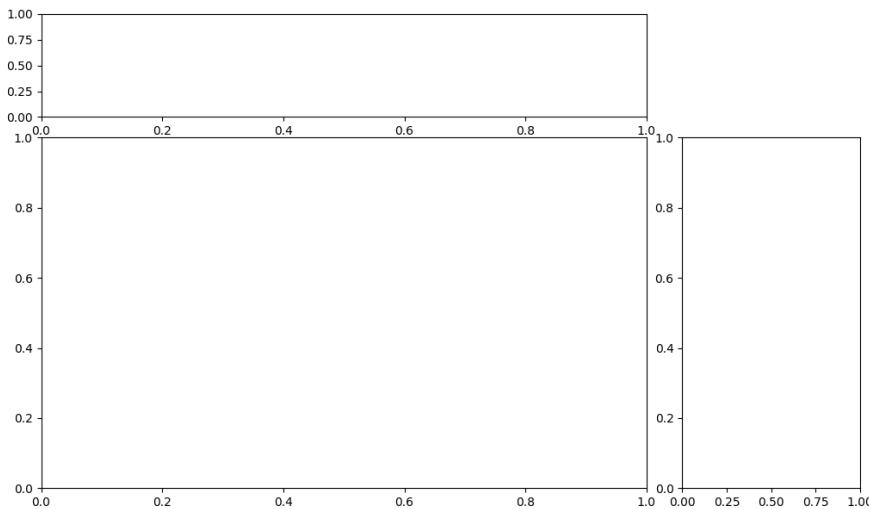
```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from matplotlib.gridspec import GridSpec
```

Note: Using the `GridSpec` allows us to control the way we organize axes and plots inside of a `Figure`. This is considered to be more on the advanced side of customization, but still an important tool nonetheless. We'll be briefly touching on the `GridSpec` in this chapter, but we'll explore its usage in more detail in *Chapter 7. - Advanced Matplotlib Customization*.

Now, let's create our `Figure` and create the `Axes` objects:

```
1 df = pd.read_csv('iris.csv')
2
3 fig = plt.figure()
4 gs = GridSpec(4, 4)
5
6 ax_scatter = fig.add_subplot(gs[1:4, 0:3])
7 ax_hist_y = fig.add_subplot(gs[0, 0:3])
8 ax_hist_x = fig.add_subplot(gs[1:4, 3])
9
10 plt.show()
```

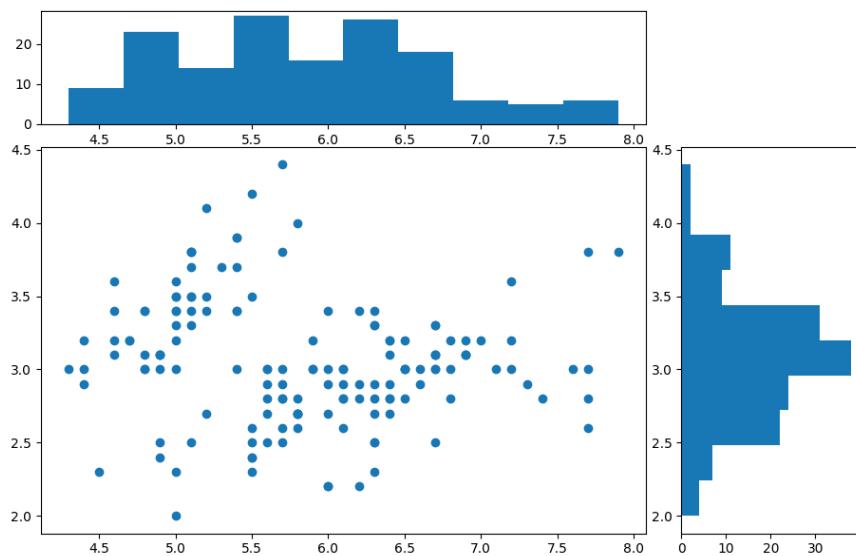
We've created 3 `Axes` instances, by adding subplots to the figure, using our `GridSpec` instance to position them. This results in a `Figure` with 3 empty `Axes` instances:



Now that we've got the layout and positioning in place, all we have to do is plot the data on our Axes. Let's update the script so that we plot the SepalLengthCm and SepalWidthCm features through a Scatter Plot, on our ax_scatter axes, and the distributions of each of these features on the ax_hist_y and ax_hist_x axes:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from matplotlib.gridspec import GridSpec
4
5 df = pd.read_csv('iris.csv')
6
7 fig = plt.figure()
8 gs = GridSpec(4, 4)
9
10 ax_scatter = fig.add_subplot(gs[1:4, 0:3])
11 ax_hist_x = fig.add_subplot(gs[0, 0:3])
12 ax_hist_y = fig.add_subplot(gs[1:4, 3])
13
14 ax_scatter.scatter(df['SepalLengthCm'], df['SepalWidthCm'])
15
16 ax_hist_x.hist(df['SepalLengthCm'])
17 ax_hist_y.hist(df['SepalWidthCm'], orientation = 'horizontal')
18
19 plt.show()
```

This results in:



This results in a Joint Plot of the relationship between the `SepalLengthCm` and `SepalWidthCm` features, as well as the distributions for the respective features. This plot contains two simple plot types within itself, and represents a wonderfully intuitive symbiosis between them. This dataset is fairly small, so we don't have a lot of markers to clutter the view.

Plotting a Joint Plot with Multiple-Class Histograms

Now, another case we might want to explore is the distribution of these features, with respect to the `Species` of the flower, since it could very possibly affect the range of sepal lengths and widths.

For this, we won't be using just one Histogram for each axis, where each contains *all flower instances*, but rather, we'll be overlaying a Histogram for each Species on both axes.

To do this, we'll first have to dissect the `DataFrame` we've been using before, by the flower `Species`:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from matplotlib.gridspec import GridSpec
4
5 df = pd.read_csv('iris.csv')
6
7 setosa = df[df['Species']=='Iris-setosa']
8 virginica = df[df['Species']=='Iris-virginica']
9 versicolor = df[df['Species']=='Iris-versicolor']
10 species = df['Species']
11 colors = {
12     'Iris-setosa' : 'tab:blue',
13     'Iris-versicolor' : 'tab:red',
14     'Iris-virginica' : 'tab:green'
15 }
```

Here, we've just filtered out the `DataFrame`, by the `Species` feature into three separate datasets. The `setosa`, `virginica` and `versicolor` datasets now contain only their respective instances.

We'll *also* want to color each of these instances with a different color, based on their `Species`, both in the Scatter Plot and in the Histograms. For that, we've simply cut out a `Series` of the `Species` feature, and made a `colors` dictionary, which we'll use to `map()` the `Species` of each flower to a color later on.

Now, let's make our `Figure`, `GridSpec` and `Axes` instances:

```
1 fig = plt.figure()
2 gs = GridSpec(4, 4)
3
4 ax_scatter = fig.add_subplot(gs[1:4, 0:3])
5 ax_hist_y = fig.add_subplot(gs[0,0:3])
6 ax_hist_x = fig.add_subplot(gs[1:4, 3])
```

Finally, we can plot out the Scatter Plot and Histograms, setting their colors and orientations accordingly:

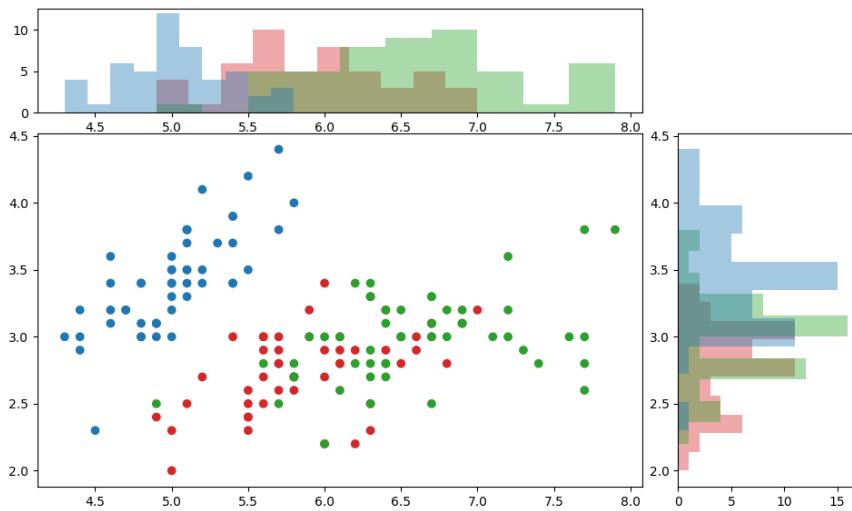
```
1 ax_scatter.scatter(df['SepalLengthCm'], df['SepalWidthCm'], c=species.map(colors))
2
3 ax_hist_y.hist(versicolor['SepalLengthCm'], color='tab:red', alpha=0.4)
4 ax_hist_y.hist(virginica['SepalLengthCm'], color='tab:green', alpha=0.4)
5 ax_hist_y.hist(setosa['SepalLengthCm'], color='tab:blue', alpha=0.4)
6
7 ax_hist_x.hist(versicolor['SepalWidthCm'],
8                 orientation = 'horizontal', color='tab:red', alpha=0.4)
9
10 ax_hist_x.hist(virginica['SepalWidthCm'],
11                  orientation = 'horizontal', color='tab:green', alpha=0.4)
12
13 ax_hist_x.hist(setosa['SepalWidthCm'],
14                  orientation = 'horizontal', color='tab:blue', alpha=0.4)
15
16 plt.show()
```

The `map()` call results in a Series of colors, each according to the *Iris* instance in question:

```
1 0      tab:blue
2 1      tab:blue
3 2      tab:blue
4 3      tab:blue
5 4      tab:blue
6 ...
7 145    tab:green
8 146    tab:green
9 147    tab:green
10 148   tab:green
11 149   tab:green
```

For the Histograms, we've simply plotted three plots, one for each Species, with their respective colors. You can opt for a step Histogram here as well, and tweak the `alpha` value to create different-looking distributions.

Running this code results in:



Now, each `Species` has its own color and distribution, plotted separately from other flowers. Furthermore, they're color-coded with the Scatter Plot so it's a really intuitive plot that can easily be read and interpreted.

Note: If you find the overlapping colors, such as the orange that comprises of the red and blue Histograms distracting, setting the `histtype` to `step` will remove the filled colors:

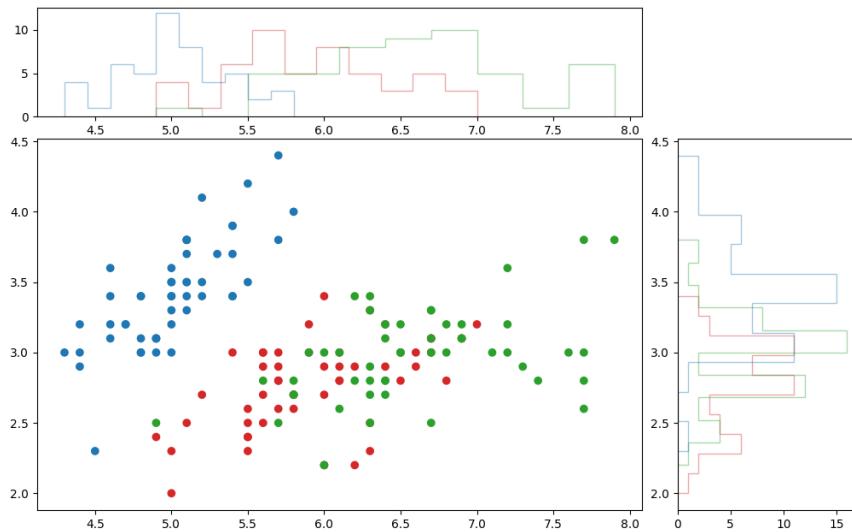
```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from matplotlib.gridspec import GridSpec
4
5 df = pd.read_csv('iris.csv')
6
7 setosa = df[df['Species']=='Iris-setosa']
8 virginica = df[df['Species']=='Iris-virginica']
9 versicolor = df[df['Species']=='Iris-versicolor']
10 species = df['Species']
11
12 colors = {'Iris-setosa' : 'tab:blue',
13           'Iris-versicolor' : 'tab:red',
14           'Iris-virginica' : 'tab:green'}
15
16 fig = plt.figure()
17 gs = GridSpec(4, 4)
18
19 ax_scatter = fig.add_subplot(gs[1:4, 0:3])

```

```
20 ax_hist_y = fig.add_subplot(gs[0,0:3])
21 ax_hist_x = fig.add_subplot(gs[1:4, 3])
22
23 ax_scatter.scatter(df['SepalLengthCm'], df['SepalWidthCm'], c=species.map(colors))
24
25 ax_hist_y.hist(versicolor['SepalLengthCm'], histtype='step', color='tab:red', alpha=0.4)
26 ax_hist_y.hist(virginica['SepalLengthCm'], histtype='step', color='tab:green', alpha=0.4)
27 ax_hist_y.hist(setosa['SepalLengthCm'], histtype='step', color='tab:blue', alpha=0.4)
28
29 ax_hist_x.hist(versicolor['SepalWidthCm'],
30                 histtype='step', orientation = 'horizontal',
31                 color='tab:red',
32                 alpha=0.4)
33
34 ax_hist_x.hist(virginica['SepalWidthCm'],
35                 histtype='step',
36                 orientation = 'horizontal',
37                 color='tab:green',
38                 alpha=0.4)
39
40 ax_hist_x.hist(setosa['SepalWidthCm'],
41                 histtype='step',
42                 orientation = 'horizontal',
43                 color='tab:blue',
44                 alpha=0.4)
45
46 plt.show()
```

This results in:



Joint Plots with Box Plots

Let's go back to our *Red Wine Quality* dataset. We've created a relatively ugly plot there to pack in a Box Plot, Histogram and Scatter Plot on the same figure. Now, let's rewrite that in a much nicer way, using the `GridSpec`:

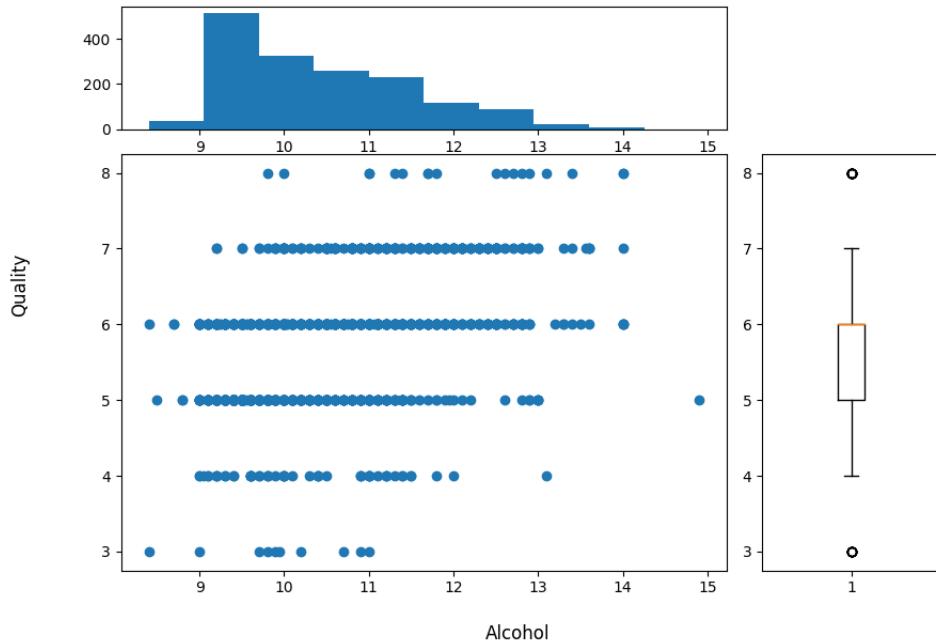
```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from matplotlib.gridspec import GridSpec
4
5 df = pd.read_csv("winequality-red.csv")
6
7 alcohol = df['alcohol']
8 quality = df['quality']
9
10 fig = plt.figure()
11 gs = GridSpec(4, 4)
12
13 ax_scatter = fig.add_subplot(gs[1:4, 0:3])
14 ax_box_y = fig.add_subplot(gs[0, 0:3])
15 ax_hist_x = fig.add_subplot(gs[1:4, 3])
16
17 ax_scatter.scatter(x = alcohol, y = quality)
18 ax_box_y.boxplot(quality)

```

```
19 ax_hist_y.hist(alcohol)
20
21 fig.supxlabel('Alcohol')
22 fig.supylabel('Quality')
23
24 plt.show()
```

This results in:



This is *much cleaner* both in terms of code and the resulting plot and allows us to interpret the data much easier than what we originally did.

Violin Plots

We've taken some time to get familiar with distribution plots like Histograms, as well as Box Plots. These are really useful by themselves, and in fact - commonly used together like we've used them previously. Box Plots give us a glimpse into the

summary statistics - the spread of a dataset, outliers, as well as the interquartile range and the median. Histograms give us a better glimpse into the distribution of a variable - letting us know the frequency of certain instances, displayed as a count or density (proportion) of the instance in the overall data.

Both Box Plots and Histograms can be used to analyze data distributions, but they're used for different facets of distribution. Thus - they're supplementary. Box Plots give us summaries that Histograms don't, and Histograms give us details that Box Plots don't.

This is where *Violin Plots* come into play:

Violin plots are used to visualize data distributions, combining the best of Histograms and Box Plots. Conceptually, they're Box Plots, but *don't have a fixed shape*. Instead, they use the Kernel Density Estimations (Histograms) to *shape the body of the plot*, in such a way to give us a glimpse into the frequency of elements in a distribution.

Violin plots show the same summary statistics as box plots, but they also include *Kernel Density Estimations* that represent the shape/distribution of the data - they're what we get when we plot a Box Plot next to a Histogram and stich them together.

Joint Plots usually combine Scatter Plots with one or more types of distribution plots - and since Violin Plots are a 2-in-1 deal - they're also a common pair, telling us more in the same space.

Importing Data

Let's visualize distributions of data from the [Gapminder World Dataset³⁶](#). It collects a few metrics on the human population, in different countries such as the *Life Expectancy*, *Population*, and *GDP per Capita*. An important thing to note is that it collects this data on a year-by-year basis, allowing us to plot timeseries plots.

The *Gapminder World Dataset* belongs to the *Gapminder Organization* which offers multiple staple datasets, with widespread use in the Data Science world, due to its great documentation and interesting data. From global warming, plastic in the

³⁶<https://www.kaggle.com/tklimonova/gapminder-datacamp-2007>

oceans, import taxes and social inequality - they've got a wide variety of datasets available for exploration.

We'll be using a couple of them in this chapter going forward. Let's start off with the *Gapminder World Dataset*:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 dataframe = pd.read_csv("gapminder_full.csv", error_bad_lines=False)
5 print(dataframe)
6 print(dataframe.isnull().values.any())
```

We'll check to make sure that there are no missing data entries and print out the head of the dataset to ensure that the data has been loaded correctly:

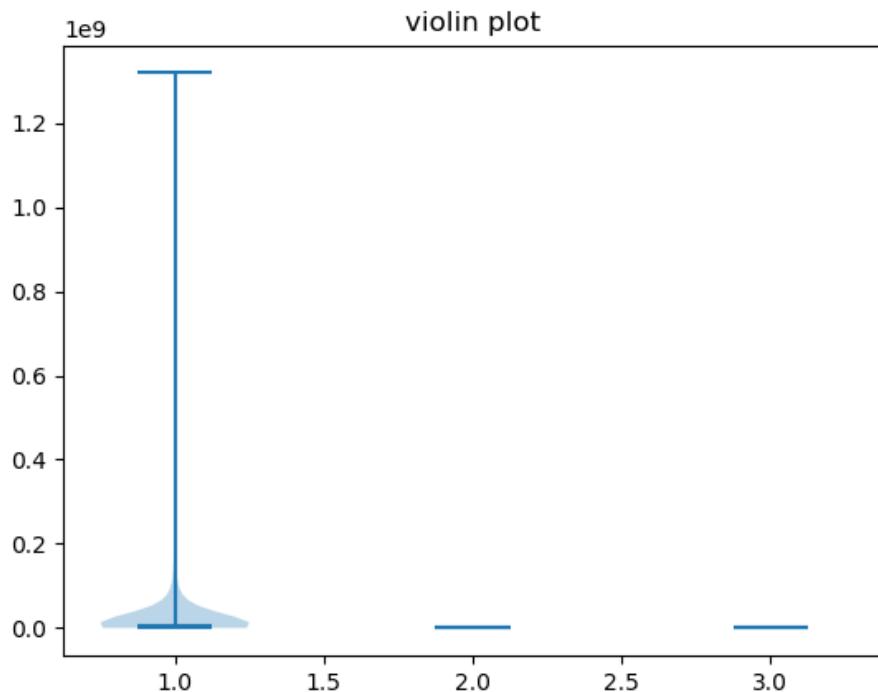
```
1      country  year  population continent  life_exp    gdp_cap
2  0  Afghanistan  1952     8425333     Asia   28.801  779.445314
3  1  Afghanistan  1957     9240934     Asia   30.332  820.853030
4  2  Afghanistan  1962    10267083     Asia   31.997  853.100710
5  3  Afghanistan  1967    11537966     Asia   34.020  836.197138
6  4  Afghanistan  1972    13079460     Asia   36.088  739.981106
7  ...
8  1699  Zimbabwe  1987    9216418     Africa  62.351  706.157306
9  1700  Zimbabwe  1992   10704340     Africa  60.377  693.420786
10 1701  Zimbabwe  1997   11404948     Africa  46.809  792.449960
11 1702  Zimbabwe  2002   11926563     Africa  39.989  672.038623
12 1703  Zimbabwe  2007   12311143     Africa  43.487  469.709298
13
14 [1704 rows x 6 columns]
15 False
```

1704 rows, without any null values. Let's get to visualizing the distributions.

Plotting a Violin Plot

To create a Violin Plot in Matplotlib, we call the `violinplot()` function on either the `Axes` instance, or the `PyPlot` instance itself. You can pass in *one* feature, which results in one Violin Plot or multiple columns, resulting in multiple plots. This is also a common usage of Violin Plots, given their summary nature, the same as Box Plots:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 dataframe = pd.read_csv("gapminder_full.csv", error_bad_lines=False)
5
6 population = dataframe.population
7 life_exp = dataframe.life_exp
8 gdp_cap = dataframe.gdp_cap
9
10 fig, ax = plt.subplots()
11
12 ax.violinplot([population, life_exp, gdp_cap])
13
14 ax.set_title('Violin Plot')
15 plt.show()
```



When we create the first plot, we can see the distribution of our data, but we will also notice some problems. Because the scale of the features are so different, it's practically impossible the distribution of the *Life expectancy* and *GDP* columns. This is a very similar issue as the one we had in our Box Plot example. Let's plot these in separate

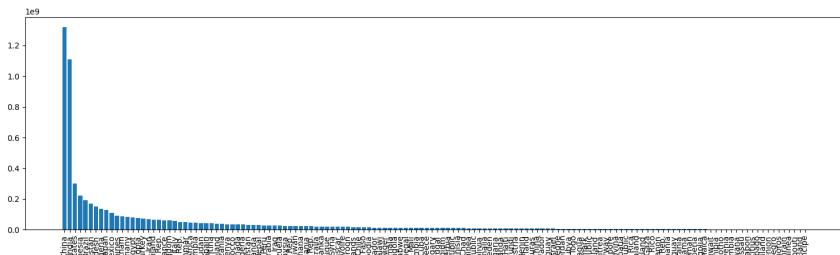
Axes.

We'll do a little sorting and slicing of the DataFrame to make the comparison of the dataset columns easier. We'll group the DataFrame by "country", and select just the most recent/last entries for each of the countries.

We'll then sort by population and drop the entries with the largest populations, so that the rest of the DataFrame is in a more similar range and comparisons are easier. This is because the population counts don't rise linearly:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 dataframe = pd.read_csv("gapminder_full.csv", error_bad_lines=False)
5
6 dataframe = dataframe.sort_values(by=["population"], ascending=False)
7 plt.bar(dataframe.country, dataframe.population)
8 plt.xticks(rotation=90)
9 plt.show()
```

This results in:



The top several countries that have a significantly higher population count make some other countries practically invisible on this plot. We'll want to drop them to normalize the data a bit, keeping in mind that we are, in fact, dropping a huge portion of the dataset and that the distributions henceforth will be distorted:

```
1 dataframe = dataframe.groupby("country").last()
2 dataframe = dataframe.sort_values(by=["population"], ascending=False)
3 dataframe = dataframe.iloc[10:]
4 print(dataframe)
```

Now, the DataFrame looks something along the lines of:

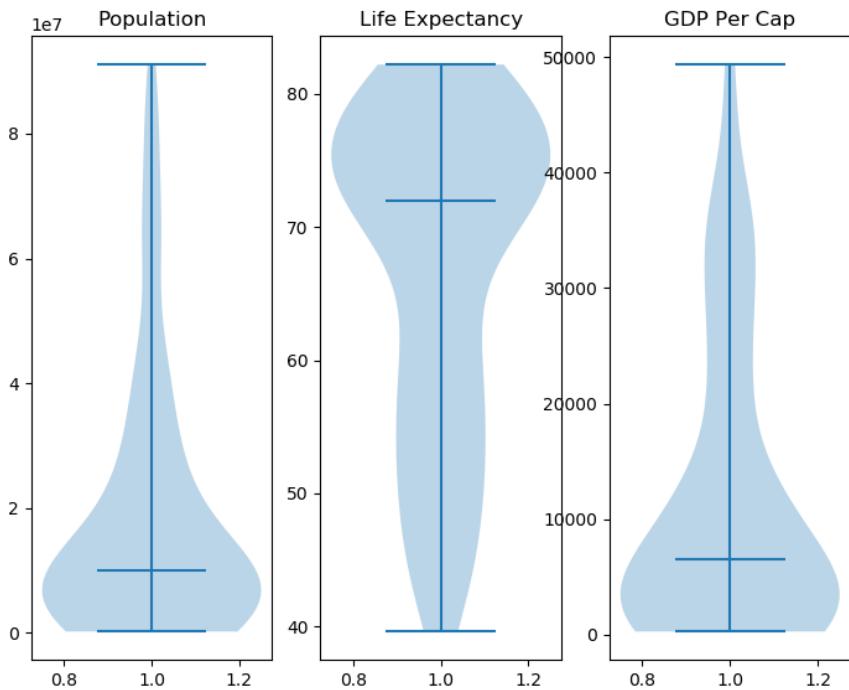
```
1          country      year  population continent  life_exp      gdp_cap
2 Philippines  2007    91077287     Asia   71.688  3190.481016
3 Vietnam     2007   85262356     Asia   74.249  2441.576404
4 Germany     2007  82400996   Europe  79.406  32170.374420
5 Egypt       2007  80264543   Africa  71.338  5581.180998
6 Ethiopia    2007  76511887   Africa  52.947   690.805576
7 ...
8 ...
9 Montenegro  2007   684736   Europe  74.543  9253.896111
10 Equatorial Guinea 2007  551201   Africa  51.579  12154.089750
11 Djibouti    2007   496374   Africa  54.791  2082.481567
12 Iceland     2007   301931   Europe  81.757  36180.789190
13 Sao Tome and Principe 2007  199579   Africa  65.528  1598.435089
14
15 [132 rows x 5 columns]
```

Great! Now we can create a figure and three axes objects with the `subplots()` function. Each of these axes will have a Violin Plot. Since we're working on a much more manageable scale now, let's also turn on the `showmedians` argument by setting it to `True`.

This will strike a horizontal line in the median of our Violin Plots:

```
1 population = dataframe.population
2 life_exp = dataframe.life_exp
3 gdp_cap = dataframe.gdp_cap
4
5 fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3)
6
7 ax1.violinplot(dataframe.population, showmedians=True)
8 ax1.set_title('Population')
9
10 ax2.violinplot(life_exp, showmedians=True)
11 ax2.set_title('Life Expectancy')
12
13 ax3.violinplot(gdp_cap, showmedians=True)
14 ax3.set_title('GDP Per Cap')
15
16 plt.show()
```

Running this code now yields us:



Now we can get a good idea of the distribution of our data. The central horizontal line in the Violins is where the median of our data is located, and minimum and maximum values are indicated by the line positions on the Y-axis.

For comparison with Box Plots, let's actually plot a Box Plot next to our Violin Plot, and annotate the constituent elements:

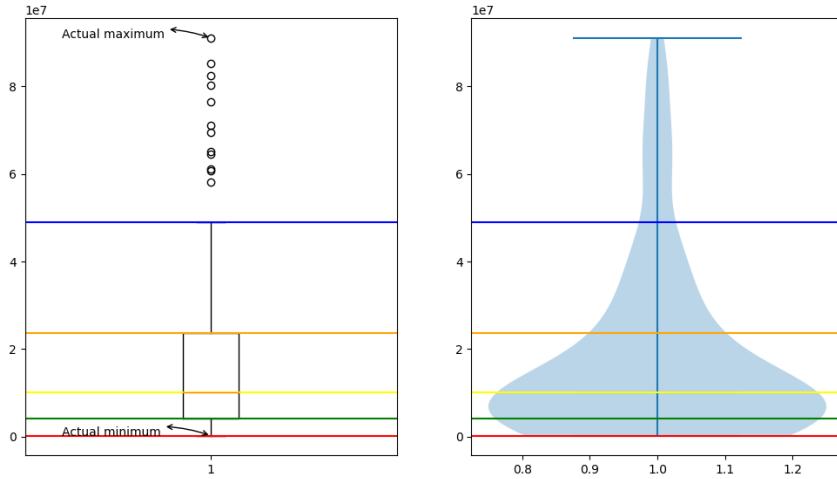
```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 dataframe = pd.read_csv("gapminder_full.csv", error_bad_lines=False)
5
6 dataframe = dataframe.groupby("country").last()
7 dataframe = dataframe.sort_values(by=["population"], ascending=False)
8 dataframe = dataframe.iloc[10:]
9
10 population = dataframe.population
11
12 fig, ax = plt.subplots(nrows=1, ncols=2)
13
14 # Return the Box and Violin objects for dissecting
15 box = ax[0].boxplot(population)
16 violin = ax[1].violinplot(population, showmedians=True)
17
18 # Extract the median value of the Box Plot as a list, so we truncate it to a single value
19 median = box['medians'][0].get_ydata()[0]
20
21 # Extract the upper and lower whiskers of the Box Plot
22 upper_whisker = box['whiskers'][1].get_ydata()[1]
23 lower_whisker = box['whiskers'][0].get_ydata()[1]
24
25 # Extract the new minimums and maximums, after removing outliers
26 new_min = box['boxes'][0].get_ydata()[1]
27 new_max = box['boxes'][0].get_ydata()[2]
28
29 # Calculate the actual minimum and maximum
30 min = population.min()
31 max = population.max()
32
33 ax[0].axhline(lower_whisker, 0, 1, label='Lower whisker, "minimum"', color='red')
34 ax[0].axhline(upper_whisker, 0, 1, label='Upper whisker, "maximum"', color='blue')
35
36 ax[0].axhline(new_min, 0, 1, label='Q1, "minimum"', color='green')
37 ax[0].axhline(median, 0, 1, label='Q2, median', color = 'yellow')
38 ax[0].axhline(new_max, 0, 1, label='Q3, "maximum"', color='orange')
39
40 ax[1].axhline(lower_whisker, 0, 1, label='Lower whisker, "minimum"', color='red')
41 ax[1].axhline(upper_whisker, 0, 1, label='Upper whisker, "maximum"', color='blue')
42
43 ax[1].axhline(new_min, 0, 1, label='Q1, "minimum"', color='green')
44 ax[1].axhline(median, 0, 1, label='Q2, median', color = 'yellow')
45 ax[1].axhline(new_max, 0, 1, label='Q3, "maximum"', color='orange')
46
47 ax[0].annotate('Actual maximum',
48                 xy=(1, max),
49                 xytext=(0.6, max-0.5),
50                 arrowprops=dict(arrowstyle='<->',
51                                 connectionstyle='arc3, rad=-0.15'))
52
53 ax[0].annotate('Actual minimum',
54                 xy=(1, min),
55                 xytext=(0.6, min+0.5),
56                 arrowprops=dict(arrowstyle='<->',
57                                 connectionstyle='arc3, rad=-0.15'))
58
59

```

```
60 plt.show()
```

This hefty chunk of code results in:



Unfortunately, we can't plot a KDE line vertically and match it next to the Box Plot. Pandas' plotting module simply can't do this. We can, however, plot them all horizontally - this time, without the extra lines for brevity's sake:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 dataframe = pd.read_csv("gapminder_full.csv", error_bad_lines=False)
5
6 dataframe = dataframe.groupby("country").last()
7 dataframe = dataframe.sort_values(by=["population"], ascending=False)
8 dataframe = dataframe.iloc[10:]
9
10 population = dataframe.population
11 life_exp = dataframe.life_exp
12 gdp_cap = dataframe.gdp_cap
13
14 fig, ax = plt.subplots(nrows=3, ncols=1)
15
16 population.hist(ax = ax[0], density=True, alpha = 0.1)
17 population.plot.kde(ax = ax[0])
18
19 ax[1].boxplot(population, vert=False)
20 ax[2].violinplot(population, showmedians=True, vert=False)
21
```

```
22 for axes in ax:  
23     axes.set_xlim(0, population.max())  
24  
25  
26 plt.show()
```

Here, we've used Pandas' plot module to plot a KDE line as well as a Histogram (with is almost transparent), and rotated the `boxplot()` and `violinplot()`, plotting the distribution of the *Population* feature on all of them. Since Pandas' KDE would've included a broader X-axis range - we've set the same X-limit on each of them:



Here, it's clearer than ever what a Violin Plot *really is*. The exact shape of the KDE line is appended to the Box Plot to achieve a Violin Plot. Since this approach is fairly cumbersome, and quite frankly *annoying* - the `violinplot()` function came to be.

That being said - much the same concepts of Box Plots and Histograms apply to Violin Plots. We're already familiar with them, in a different form.

Customizing Violin Plots

Now, with an understanding of what Violin Plots are, let's take a look at how we can customize them.

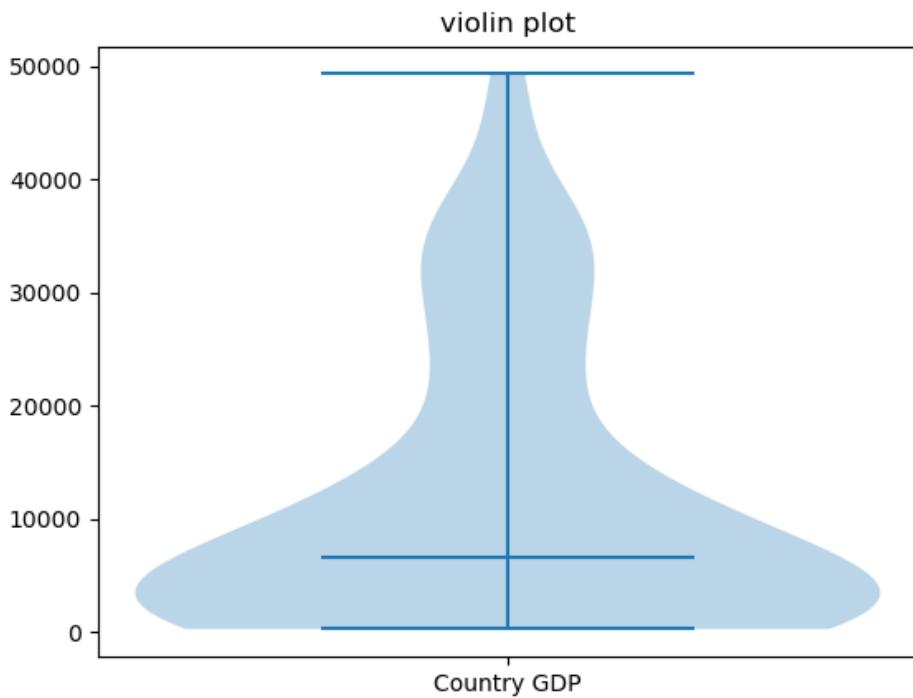
Adding X and Y Ticks

While multiple Violin Plots have successfully been generated previously, without tick labels on the X and Y-axis it can get difficult to interpret the graph. Humans interpret categorical values much more easily than numerical values, which in an absence of categorical values span from $0\dots 1$.

Since it has multiple ticks, for various decimal values between 0 and 1 , let's set the `xticks` to 1 , and assign a *categorical value/label* to it:

```
1 fig, ax = plt.subplots()
2 ax.violinplot(gdp_cap, showmedians=True)
3 ax.set_title('violin plot')
4 ax.set_xticks([1])
5 ax.set_xticklabels(["Country GDP"])
6 plt.show()
```

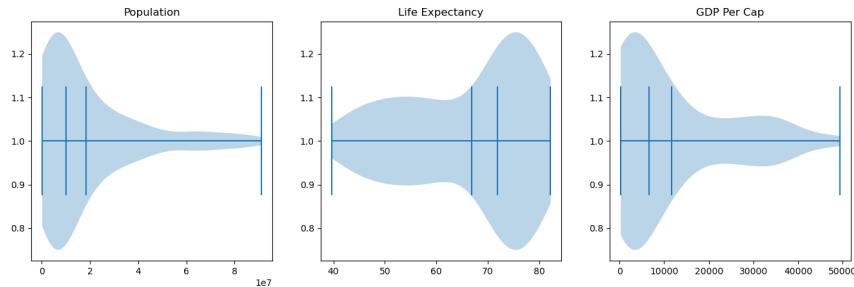
This results in:



Showing Dataset Means and Medians in Violin Plots

By default, unlike Box Plots - Violin Plots don't show a median line. Though, you can easily turn them on, as well as mean lines, with the `showmeans` and `showmedians` arguments:

```
1 fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3)
2 ax1.violinplot(population, showmedians=True, showmeans=True, vert=False)
3 ax1.set_title('Population')
4
5 ax2.violinplot(life_exp, showmedians=True, showmeans=True, vert=False)
6 ax2.set_title('Life Expectancy')
7
8 ax3.violinplot(gdp_cap, showmedians=True, showmeans=True, vert=False)
9 ax3.set_title('GDP Per Cap')
10 plt.show()
```



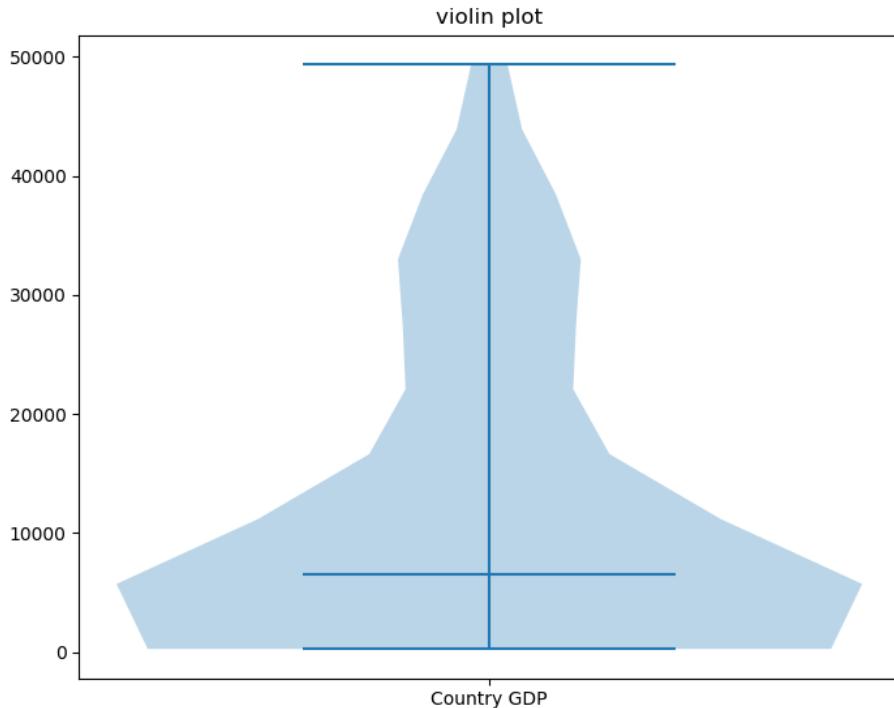
Though, please note that since the medians and means essentially look the same, it may become unclear which vertical line here refers to a median, and which to a mean, so it's not advisable to use them both at the same time.

Customizing Kernel Density Estimation for Violin Plots

Since the shape of the Violin Plot is really just a Kernel Density Estimation of the feature's distribution - we can tweak the estimation itself, thus tweaking the Violin Plot as a whole. For example, we can alter how many data points the model considers when creating the Kernel Density Estimations, by altering the `points` parameter.

The number of points considered is 100 by default. By providing the function with fewer data points to estimate from, we may get a less representative data distribution. Let's change this number to, say, 10:

```
1 fig, ax = plt.subplots()
2 ax.violinplot(gdp_cap, showmedians=True, points=10)
3 ax.set_title('violin plot')
4 ax.set_xticks([1])
5 ax.set_xticklabels(["Country GDP",])
6 plt.show()
```



Notice that the shape of the violin is less smooth since fewer points have been sampled for the estimation.

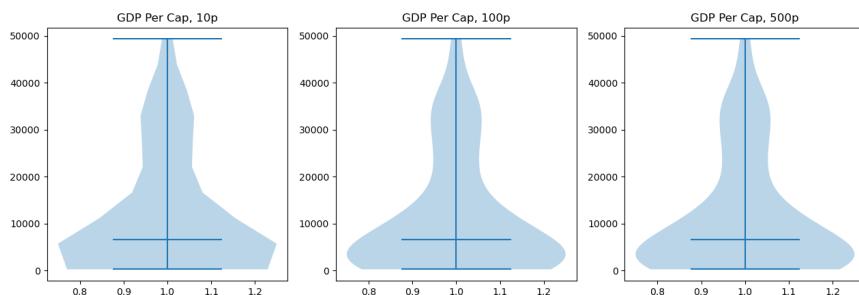
Typically, you would want to increase the number of points used to get a better sense of the distribution. This might not always be the case, if 100 is simply enough. Lets plot a 10-point, 100-point and 500-point sampled Violin Plot:

```

1 fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3)
2 ax1.violinplot(gdp_cap, showmedians=True, points=10)
3 ax1.set_title('GDP Per Cap, 10p')
4
5 ax2.violinplot(gdp_cap, showmedians=True, points=100)
6 ax2.set_title('GDP Per Cap, 100p')
7
8 ax3.violinplot(gdp_cap, showmedians=True, points=500)
9 ax3.set_title('GDP Per Cap, 500p')
10 plt.show()

```

This results in:



There isn't any obvious difference between the second and third plot, though, there's a significant one between the first and second. Generally speaking - you *should* be fine with the default setting most of the time.

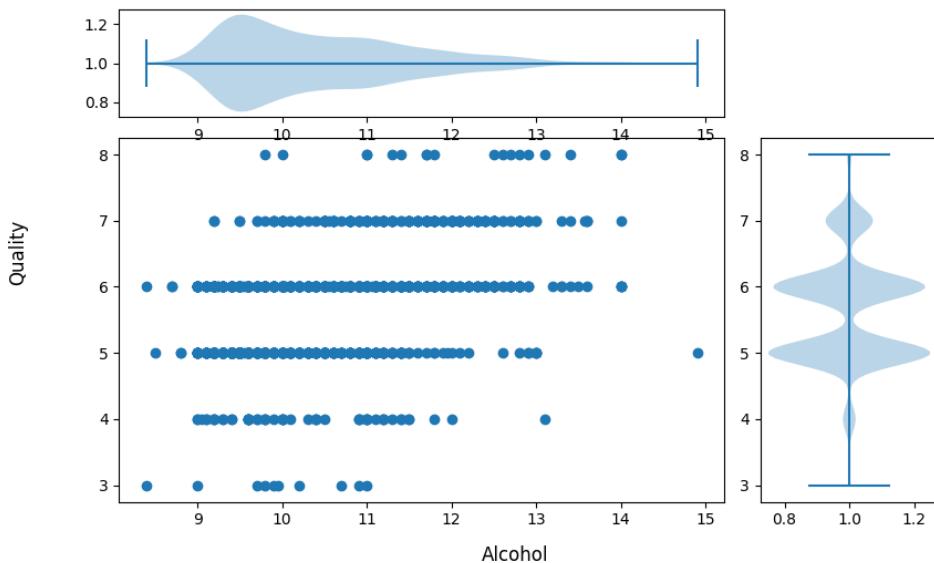
Scatter Plots with Violin Plots

Proving that Violin Plots are just Box Plots with a KDE-line opens up a new possibility. We've seen Joint Plots with Histograms/KDE lines, and Joint Plots with Box Plots. Why not just substitute them fully with Violin Plots, which have the informative value of both?

There's no reason as to why not. In fact - it's not uncommon either. Some libraries, like Plotly, for example, have built-in support that allows you to turn on marginal Violin/Box/Histogram plots attached to a Scatter Plot. Let's tweak our old Joint Plot example to use a Violin Plot on both margins instead:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from matplotlib.gridspec import GridSpec
4
5 df = pd.read_csv("winequality-red.csv")
6
7 alcohol = df['alcohol']
8 quality = df['quality']
9
10 fig = plt.figure()
11 gs = GridSpec(4, 4)
12
13 ax_scatter = fig.add_subplot(gs[1:4, 0:3])
14 ax_violin_y = fig.add_subplot(gs[0, 0:3])
15 ax_violin_x = fig.add_subplot(gs[1:4, 3])
16
17 ax_scatter.scatter(x = alcohol, y = quality)
18 ax_violin_x.violinplot(quality)
19 ax_violin_y.violinplot(alcohol, vert=0)
20
21 fig.suptitle('Alcohol')
22 fig.supylabel('Quality')
23
24 plt.show()
```

This results in:



Stack Plots

We've talked about distribution plots for quite a while - Histograms, Box Plots, Joint Plots, Violin Plots... Let's take a step back and take a look at a reimagined version of a *Line Plot*.

Again, Line Plots are very versatile, and can be used to plot any categorical and numerical feature pair. They can actually even work with two numerical features, such as with functions.

Combining multiple Line Plots, we can get a new type - *Stack Plots*:

Stack Plots are used to plot linear data, in a vertical order, stacking each linear plot on another. Typically, they're used to generate cumulative plots.

They're very useful when you'd like to contextualize the proportions of certain feature values over others. For example, your *Expenses* might be on a time-series.

These *Expenses* break down into *Household Expenses*, *Professional Expenses* and *Personal Expenses*. Each of these can be plotted as a time-series Line Plot of its own. Instead of having them plotted together - we stack them *on top of each other* so that their *sum* is equal to the *Expenses*. This is why they're so commonly used for cumulative plots.

Importing Data

We'll be using a dataset on Covid-19 vaccinations, from [Our World in Data³⁷](https://ourworldindata.org), specifically, the dataset that contains the cumulative vaccinations per country - the [Cumulative Covid Vaccinations Dataset³⁸](https://ourworldindata.org/grapher/cumulative-covid-vaccinations).

We'll begin by importing all the libraries and dataset - we'll need Numpy to produce a the days for the X-axis, since we won't be using the *Day* feature as a date, but rather as the *number of days* since the first vaccination:

³⁷<https://ourworldindata.org>

³⁸<https://ourworldindata.org/grapher/cumulative-covid-vaccinations>

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5 dataframe = pd.read_csv('cumulative-covid-vaccinations.csv')
6 print(dataframe)
```

We're interested in the Entity and total_vaccinations. While we could use the Date feature as well, to gain a better grasp of how the vaccinations are going *day-by-day*, we'll treat the first entry as *Day 0* and the last entry as *Day N*:

```
1          Entity    Code      Day  total_vaccinations
2  0      Afghanistan  AFG  2021-02-22                  0
3  1      Afghanistan  AFG  2021-02-28                8200
4  2      Afghanistan  AFG  2021-03-16              54000
5  3      Afghanistan  AFG  2021-04-07            120000
6  4      Afghanistan  AFG  2021-04-22            240000
7  ...
8  12177      Zimbabwe  ZWE  2021-05-12            730365
9  12178      Zimbabwe  ZWE  2021-05-13            752020
10 12179      Zimbabwe  ZWE  2021-05-14            775241
11 12180      Zimbabwe  ZWE  2021-05-15            793311
12 12181      Zimbabwe  ZWE  2021-05-16            796947
13
14 [12182 rows x 4 columns]
```

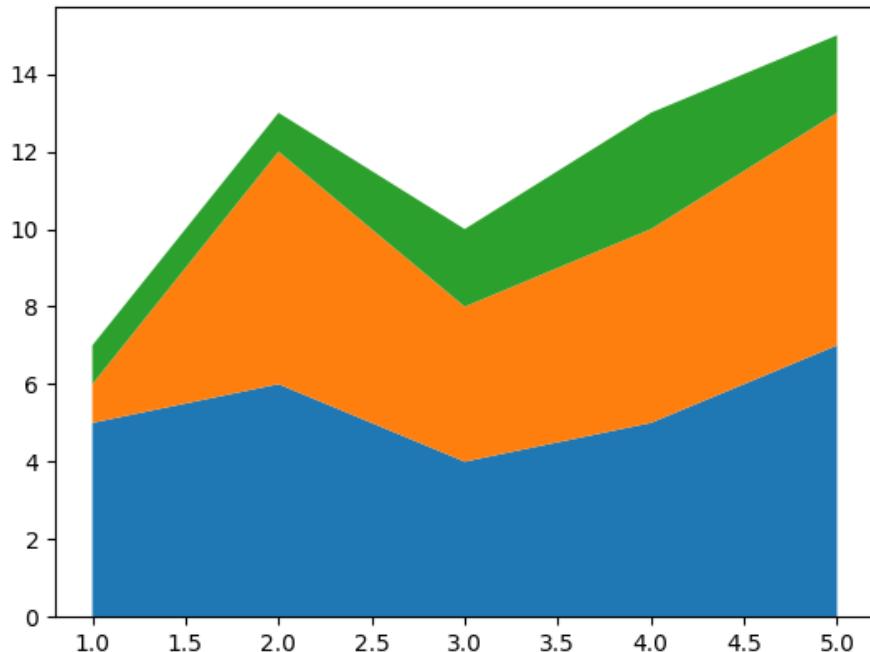
This dataset will require some pre-processing. Though, before pre-processing it, let's get acquainted with how Stack Plots are generally plotted.

Plotting a Stack Plot

Stack Plots are used to visualize multiple linear plots, stacked on top of each other. With a regular Line Plot, you'd plot the relationship between X and Y. Here, we're plotting multiple Y features on a shared X-axis, one on top of the other:

```
1 import matplotlib.pyplot as plt
2 x = [1, 2, 3, 4, 5]
3 y1 = [5, 6, 4, 5, 7]
4 y2 = [1, 6, 4, 5, 6]
5 y3 = [1, 1, 2, 3, 2]
6
7 fig, ax = plt.subplots()
8 ax.stackplot(x, y1, y2, y3)
9 plt.show()
```

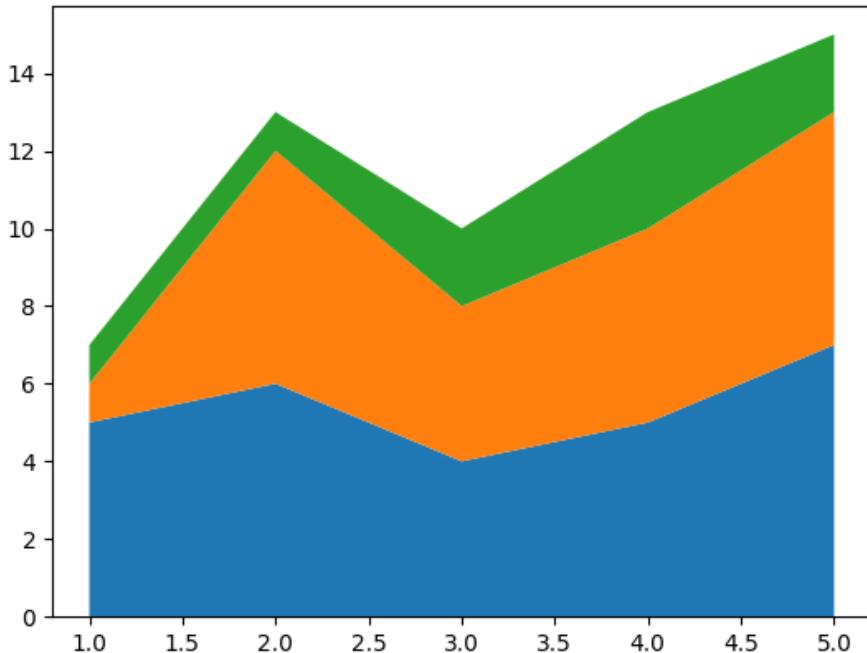
This results in:



Since it's a bit unwieldy to deal with multiple lists like this, you can simply use a dictionary, where each y_n feature is an entry:

```
1 import matplotlib.pyplot as plt
2 x = [1, 2, 3, 4, 5]
3
4 y_values = {
5     "y1": [5, 6, 4, 5, 7],
6     "y2": [1, 6, 4, 5, 6],
7     "y3": [1, 1, 2, 3, 2]
8 }
9
10 fig, ax = plt.subplots()
11 ax.stackplot(x, y_values.values())
12 plt.show()
```

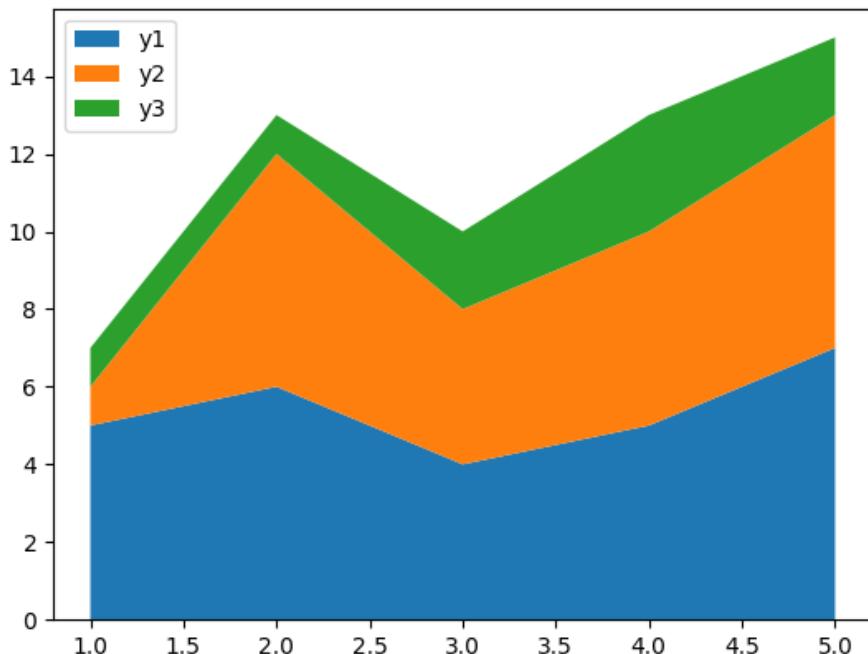
This results in:



Since this type of plot can easily get you lost in the stacks, it's really helpful to add labels attached to the colors, by setting the `keys()` from the `y_values` dictionary as the `labels` argument, and adding a legend to the plot:

```
1 import matplotlib.pyplot as plt
2 x = [1, 2, 3, 4, 5]
3
4 y_values = {
5     "y1": [5, 6, 4, 5, 7],
6     "y2": [1, 6, 4, 5, 6],
7     "y3": [1, 1, 2, 3, 2]
8 }
9
10 fig, ax = plt.subplots()
11 ax.stackplot(x, y_values.values(), labels=y_values.keys())
12 ax.legend(loc='upper left')
13 plt.show()
```

Now, this results in:



Note: The length of these lists *has to be the same*. You can't plot y1 with 3 values, and y2 with 5 values. If there's a mismatch of shapes, you'll have to conjure empty values to synchronize them.

This brings us to our Covid-19 vaccination dataset. We'll pre-process the dataset to take the form of a dictionary like this, and plot the cumulative number of vaccines given to the general population.

Let's start off by grouping the dataset by Entity and total_vaccinations, since each Entity currently has numerous entries. Also, we'll want to drop the entities named World and European Union, since they're convenience entities, added for cases where you might want to plot just a single cumulative line.

In our case, it'll effectively more than double the total_vaccination count, since they include already plotted values of each country, as single entities:

```
1  dataframe = pd.read_csv("cumulative-covid-vaccinations.csv")
2  indices = dataframe[(dataframe['Entity'] == 'World')
3                      | (dataframe['Entity'] == 'European Union')
4                      | (dataframe['Entity'] == 'High income')].index
5  dataframe.drop(indices, inplace=True)
6
7  countries_vaccinations = dataframe.groupby('Entity')[['total_vaccinations']].apply(list)
8  print(countries_vaccinations)
```

This results in a completely different shape of the dataset - instead of each entry having their own Entity/total_vaccinations entry, each Entity will have a *list* of their total vaccinations through the days:

```
1  Entity
2  Albania      [0, 128, 188, 266, 308, 369, 405, 447, 483, 51...
3  Algeria     [0, 30, 75000]
4  Andorra      [576, 1036, 1291, 1622, 2141, 2390, 2526, 3611...
5  ...
6  Croatia      [7864, 12285, 13798, 20603, 24985, 30000, 3455...
7  Cyprus        [3901, 6035, 10226, 17739, 25519, 32837, 44429...
8  Czechia      [1261, 3560, 7017, 10496, 11813, 12077, 13335,...
```

Now, let's convert this Series into a dictionary and see what it looks like:

```
1  cv_dict = countries_vaccinations.to_dict()
2  print(cv_dict)
```

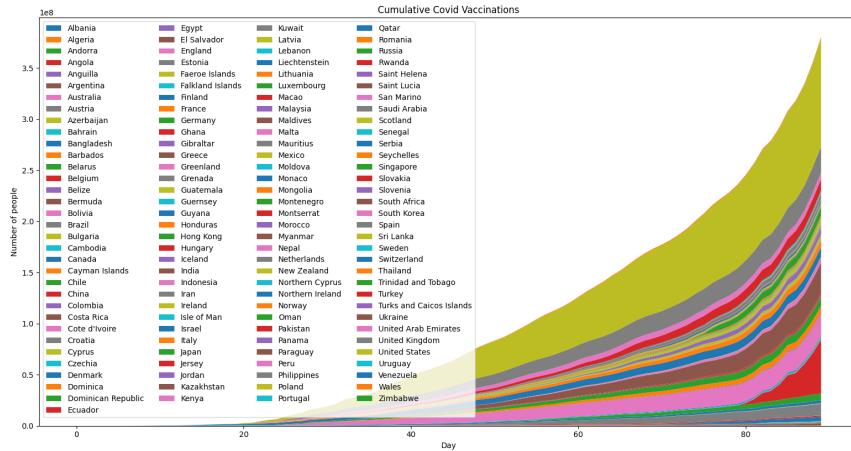
This results in:


```
1 print(max_key, len(max_value)) # Canada 90
```

Now that we've fully prepared our dataset, and we can plot it as we've plotted the Stack Plots before it, let's generate the days and plot:

```
1 dates = np.arange(0, len(max_value))
2
3 fig, ax = plt.subplots()
4 ax.stackplot(dates, cv_dict_full.values(), labels=cv_dict_full.keys())
5 ax.legend(loc='upper left', ncol=4)
6 ax.set_title('Cumulative Covid Vaccinations')
7 ax.set_xlabel('Day')
8 ax.set_ylabel('Number of people')
9
10 plt.show()
```

Since there's a lot of countries in the world, the legend will be fairly crammed, so we've put it into 4 columns to at least fit in the plot:



Heatmaps

A plot type you're most likely familiar with already are *Heatmaps*:

Heatmaps color-code variables, based on the value of other variables.

The color-coding of variables is usually ascribed to an *intensity* variable, which changes the color of the cell it belongs to, on a color spectrum. As the name implies, the colormaps of Heatmaps are typically from *cold colors* like *blue* to *warm colors* like *red*, giving us an intuitive feel of the intensity of a variable.

Heatmaps can be used to represent various things - for example, you could assign the *altitude* of a mountain to the intensity variable, and color each part of a map according to the height. Similarly enough, you can assign the *correlation* of a feature to the intensity, and create a correlation Heatmap. *Correlation Heatmaps* are commonly used in the Exploratory Data Analysis (which includes Data Visualization) step of Data Science.

These are, essentially, *Correlation Matrices*, stylized in an aesthetically pleasing way. Correlation Matrices are typically calculated with external libraries, and are surprisingly enough - not built into Matplotlib, though, we can make do with alternative methods.

Let's revisit an older example - from the *Ames Housing Dataset*. We've been speculating and visualizing the correlation between the *SalePrice* feature and some other features. What we can do, is actually calculate the correlation between these - and then turn that into a *Correlation Matrix/Heatmap*.

Importing Data

Let's import the *Ames Housing Dataset* and take a refreshing look on what it contains:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('AmesHousing.csv')
5
6 print(df)
7 print(df.columns)
```

This results in:

```

1      Order      PID  MS SubClass  ... Sale Type  Sale Condition  SalePrice
2      0          1  526301100        20  ...    WD      Normal     215000
3      1          2  526350040        20  ...    WD      Normal     105000
4      2          3  526351010        20  ...    WD      Normal     172000
5      3          4  526353030        20  ...    WD      Normal     244000
6      4          5  527105010        60  ...    WD      Normal     189900
7      ...
8      2925      2926  923275080        80  ...    WD      Normal     142500
9      2926      2927  923276100        20  ...    WD      Normal     131000
10     2927      2928  923400125        85  ...    WD      Normal     132000
11     2928      2929  924100070        20  ...    WD      Normal     170000
12     2929      2930  924151050        60  ...    WD      Normal     188000
13
14 [2930 rows x 82 columns]
15 Index(['Order', 'PID', 'MS SubClass', 'MS Zoning', 'Lot Frontage', 'Lot Area',
16 'Street', 'Alley', 'Lot Shape', 'Land Contour', 'Utilities',
17 'Lot Config', 'Land Slope', 'Neighborhood', 'Condition 1',
18 'Condition 2', 'Bldg Type', 'House Style', 'Overall Qual',
19 'Overall Cond', 'Year Built', 'Year Remod/Add', 'Roof Style',
20 'Roof Matl', 'Exterior 1st', 'Exterior 2nd', 'Mas Vnr Type',
21 'Mas Vnr Area', 'Exter Qual', 'Exter Cond', 'Foundation', 'Bsmt Qual',
22 'Bsmt Cond', 'Bsmt Exposure', 'BsmtFin Type 1', 'BsmtFin SF 1',
23 'BsmtFin Type 2', 'BsmtFin SF 2', 'Bsmt Unf SF', 'Total Bsmt SF',
24 'Heating', 'Heating QC', 'Central Air', 'Electrical', '1st Flr SF',
25 '2nd Flr SF', 'Low Qual Fin SF', 'Gr Liv Area', 'Bsmt Full Bath',
26 'Bsmt Half Bath', 'Full Bath', 'Half Bath', 'Bedroom AbvGr',
27 'Kitchen AbvGr', 'Kitchen Qual', 'TotRms AbvGrd', 'Functional',
28 'Fireplaces', 'Fireplace Qu', 'Garage Type', 'Garage Yr Blt',
29 'Garage Finish', 'Garage Cars', 'Garage Area', 'Garage Qual',
30 'Garage Cond', 'Paved Drive', 'Wood Deck SF', 'Open Porch SF',
31 'Enclosed Porch', '3Ssn Porch', 'Screen Porch', 'Pool Area', 'Pool QC',
32 'Fence', 'Misc Feature', 'Misc Val', 'Mo Sold', 'Yr Sold', 'Sale Type',
33 'Sale Condition', 'SalePrice'],
34 dtype='object')

```

This dataset contains quite a bit of columns. These are all features that might impact the *SalePrice*. While we *can* try and visualize each one and explore the relationship manually - this defeats the purpose.

Plotting a Heatmap

A really easy way to calculate the correlation of features in a DataFrame comes packaged with Pandas - the `corr()` function of a DataFrame:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('AmesHousing.csv')
5
6 saleprice_corr = df.corr()[['SalePrice']]
7 print(saleprice_corr)
```

The `corr()` function calculates the correlation between all numerical features and the target feature we've selected - in this case, the `SalePrice` feature. It ignores all null-fields and calculates the *Pearson Correlation Coefficient*³⁹.

The *Pearson Correlation Coefficient* measures the linear association between two variables - and depending on the value returned, from -1 to 1 it can interpreted as:

Value	Interpretation
+1	Complete positive correlation
+0.8	Strong positive correlation
+0.6	Moderate positive correlation
0	No correlation whatsoever
-0.6	Moderate negative correlation
-0.8	Strong negative correlation
-1	Complete negative correlation

Other than the default `pearson` value, we can specify the `method` as `kendall` or `spearman` - signifying the *Kendall Rank Correlation Coefficient*⁴⁰ and *Spearman's Rank Correlation Coefficient*⁴¹ respectively.

Where Pearson calculates the correlation as *a linear relationship* between two variables, Spearman calculates *the monotonic relation* between a pair of variables, while Kendall calculates *the ordinal association* of two variables.

The `corr()` function returns a `DataFrame`, so we can easily take a look at what's inside:

³⁹[https://stackabuse.com/calculating-pe](https://stackabuse.com/calculating-pearson-correlation-coefficient-in-python-with-numpy/)rson-correlation-coefficient-in-python-with-numpy/

⁴⁰[https://en.wikipedia.org/wiki/Kendall_rank](https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient) correlation coefficient

⁴¹[https://stackabuse.com/calculating-spe](https://stackabuse.com/calculating-spearman-correlation-coefficient-in-python-with-numpy/)rman-correlation-coefficient-in-python-with-numpy/

```

1 import pandas as pd
2
3 df = pd.read_csv('AmesHousing.csv')
4
5 # Optional argument, default method is 'pearson'
6 pearson_corr = df.corr(method='pearson')
7 print(pearson_corr)

```

Without specifying a column for *which* we're calculating the correlation of other variables for - Pandas will calculate the correlations between all features and all other features. Each feature has a *1.0* correlation with itself, which results in a 2D Correlation Matrix with a diagonal line of *1.0* correlations:

	Order	PID	MS SubClass	...	Mo Sold	Yr Sold	SalePrice
1 Order	1.000000	0.173593	0.011797	...	0.133365	-0.975993	-0.031408
2 PID	0.173593	1.000000	-0.001281	...	-0.050455	0.009579	-0.246521
3 MS SubClass	0.011797	-0.001281	1.000000	...	0.000350	-0.017905	-0.085092
4 Lot Frontage	-0.007034	-0.096918	-0.420135	...	0.011085	-0.007547	0.357318
5 Lot Area	0.031354	0.034868	-0.204613	...	0.003859	-0.023085	0.266549
6 Overall Qual	-0.048500	-0.263147	0.039419	...	0.031103	-0.020719	0.799262
7 ...							
8 Yr Sold	-0.975993	0.009579	-0.017905	...	-0.155554	1.000000	-0.030569
9 SalePrice	-0.031408	-0.246521	-0.085092	...	0.035259	-0.030569	1.000000

We could limit the correlation calculation to a single variable against all other variables as well, or any subset:

```
1 pearson_corr = df.corr(method='pearson')[['SalePrice']]
```

	SalePrice
1 Order	-0.031408
2 PID	-0.246521
3 ...	
4 Yr Sold	-0.030569
5 SalePrice	1.000000

Or, with several columns:

```

1  pearson_corr = df.corr(method='pearson')[['SalePrice',
2                                              'Overall Qual',
3                                              'Gr Liv Area',
4                                              'Garage Cars',
5                                              'Garage Area']]
6

7          SalePrice  Overall Qual  Gr Liv Area  Garage Cars  Garage Area
8 Order      -0.031408    -0.048500   -0.009342   -0.036185   -0.035435
9 PID       -0.246521    -0.263147   -0.107579   -0.237484   -0.210606
10 ...
11 Yr Sold   -0.030569    -0.020719   -0.026489   -0.022488   -0.013018
12 SalePrice  1.000000     0.799262    0.706780    0.647877    0.640401

```

In any case - as long as we have a 2D array, we can plot a Heatmap. Matplotlib doesn't have a built-in `heatmap()` function, and Heatmaps are plotted via the `imshow()` function - which is used for plotting images. When provided with 2D scalar data, such as this - it's rendered as a pseudocolor image. If tweak the generated plot a bit, by forcing `aspect` to be `'equal'`, the `interpolation` to `'nearest'` and `origin` to `'upper'`, we can achieve the look of a Heatmap.

Thankfully, since enough people were using this commonly - the Matplotlib team made a convenience function - `matshow()`, short for *Matrix Show*, which is just a wrapper for the `imshow()` function, with these arguments set to the right values to produce a Heatmap/Correlation Matrix:

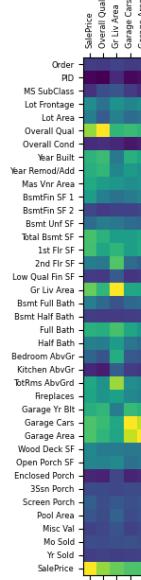
```

1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('AmesHousing.csv')
5 pearson_corr = df.corr(method='pearson')[['SalePrice',
6                                              'Overall Qual',
7                                              'Gr Liv Area',
8                                              'Garage Cars',
9                                              'Garage Area']]
10
11 fix, ax = plt.subplots()
12 ax.matshow(pearson_corr)
13
14 plt.yticks(range(0, len(pearson_corr.index)),
15            pearson_corr.index, fontsize=6)
16
17 plt.xticks(range(0, len(pearson_corr.columns)),
18            pearson_corr.columns, fontsize=6, rotation=90)
19
20 plt.show()

```

We'll also want to set the Y-ticks and X-ticks to show the feature names. Otherwise, they'll be labeled as numbers, which makes it impossible for us to figure out which

is which. On the Y-axis, we'll use the `index` of the `DataFrame` for the labels, and for the X-axis, we'll use the `columns`, rotating them by 90 degrees to make them fit:



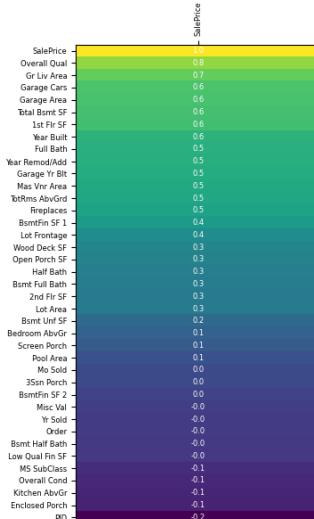
We can limit ourselves to plotting a *single* feature, such as the `SalePrice` against all other features, and sort them by their coefficient value, giving us a view of which features have the highest correlation with it. While we're there, since we're plotting just a single feature - the Heatmap will be very narrow. We can tweak the `aspect` argument, which also accepts a float. If the `aspect` is 1 (default value, same as `equal`), for each Y-pixel, an X-pixel will also be added. The closer the `aspect` is to 0, the more X-pixels there will be for each Y-pixel, and vice versa.

By setting the `aspect` to `0.05`, we'll widen the Heatmap.

We can, without text, interpret the Heatmap - that's what it's for. Though, sometimes, it's also nice to have numerical values of the correlations displayed as well. Let's sort the resulting `DataFrame`, change the `aspect` and add numerical values:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('AmesHousing.csv')
5
6 pearson_corr = df.corr(method='pearson')[['SalePrice']].sort_values(by='SalePrice',
7                                         ascending=False)
8
9 fix, ax = plt.subplots()
10
11 ax.matshow(pearson_corr, aspect=0.05)
12
13 plt.yticks(range(0, len(pearson_corr.index)),
14            pearson_corr.index, fontsize=6)
15 plt.xticks(range(0, len(pearson_corr.columns)),
16            pearson_corr.columns, fontsize=6, rotation=90)
17
18 for y in range(pearson_corr.shape[0]):
19     plt.text(0, y, '%.1f' % pearson_corr.iloc[y],
20              horizontalalignment='center',
21              verticalalignment='center',
22              fontsize=6
23      )
24
25 plt.show()
```

This results in a Correlation Matrix, with the *SalePrice* feature and its correlation with other features plotted vertically, and sorted:

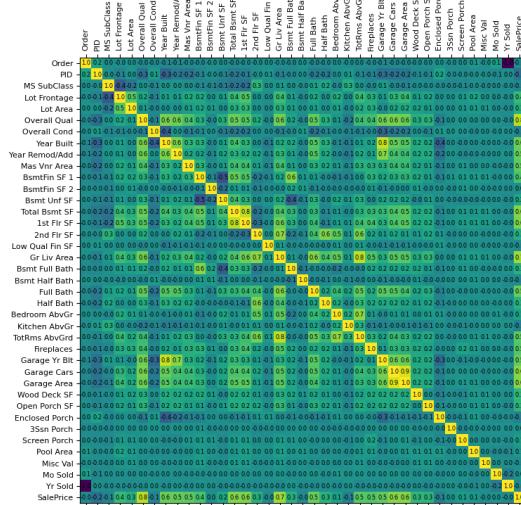


Here, we can see that the *Overall Qual* feature has the largest correlation with the *SalePrice*, even above the *Gr Liv Area* feature. At the very bottom, with a slight negative correlation, we've got *Kitchen AbvGr*, *Overall Cond*, etc. This might actually come as a surprise - it's reasonable to think that the overall condition of the property does have a major effect on the price. Surprisingly enough, the *Pool Area* only has a correlation value of *0.1*. It looks like many people *dream* of the luxury of having a pool, but when faced with an actual decision of buying a property with or without one - it's not really an important feature.

Let's widen the scope of the correlation to *all features*:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3
4 df = pd.read_csv('AmesHousing.csv')
5
6 pearson_corr = df.corr(method='pearson')
7
8 fix, ax = plt.subplots()
9 ax.matshow(pearson_corr)
10
11 plt.yticks(range(0, len(pearson_corr.index)),
12             pearson_corr.index, fontsize=6)
13 plt.xticks(range(0, len(pearson_corr.columns)),
14             pearson_corr.columns, fontsize=6, rotation=90)
15
16 for y in range(pearson_corr.shape[0]):
17     for x in range(pearson_corr.shape[1]):
18         plt.text(x, y, '%.1f' % pearson_corr.iloc[x, y],
19                   horizontalalignment='center',
20                   verticalalignment='center',
21                   fontsize=6
22         )
23
24 plt.show()
```

This results in a much larger, much more populated and busy Heatmap:



Though, since we have *a lot* of features here, the Heatmap might be a bit overwhelming. We can see the same correlation values between the *SalePrice* and other features, at the end - but we also see all the other features here.

Ridge Plots (Joy Plots)

Ridge Plots, also known as *Joy Plots*, are a vertical multi-Axes plot type that combines several, usually distribution plots - namely, Histograms, KDE plots and Violin Plots. Though, they can also be used for Line Plots, not just distribution plots.

What's the difference between vertically arranged Axes' with distribution/linear plots and Ridge Plots?

The Axes on Ridge Plots overlap, and form something that looks like a 3D image of *ridges*, even though they're really 2D. A 3D variation of Ridge Plots does exist, as well.

The alternative name of “*Joy Plots*” comes from pop culture - and surprisingly enough, from the English rock band [Joy Division](#)⁴². The story of Ridge Plots carries an

⁴²https://en.wikipedia.org/wiki/Joy_Division

interesting, quirky story, starting before the band was formed, albeit, not too long ago. Namely, back in 1967, [Jocelyn Bell Burnell](#)⁴³, who was a Cambridge student at the time and now holds a Nobel Prize for the discovery, noticed a very specific, seemingly artificial pattern of radio frequencies originating from *outer space*, marked down by her [chart recorder](#)⁴⁴. The tool receives electromagnetic input and charts the amplitude and frequency of the electromagnetic signals down on paper. A similar technology is used in *polygraphs* to this day.

The pattern of radio signals repeated *each 1.337 seconds* and had a width of *0.04* seconds, which seemed too precise to *not* be deliberate at the time. Jokingly, the signal was named *LGM-1*, standing for *Little Green Men-1*, alluding to the fact that the radio signal seemed, by all accounts, to be originating from extraterrestrial intelligent life. Quickly enough, a similar signal was detected coming from another location and the probability that we've encountered *two* extraterrestrial species, using similar technology, so far apart, *at the same time* was low enough that Jocelyn ruled out a deliberate extraterrestrial contact attempt, and ascribed the radio frequencies to a natural phenomenon.

Later on, it turned out that the radio signals were coming from a *pulsar* - a highly magnetized neutron star that emits *strong beams* of electromagnetic radiation (in a frequency range of radio waves) from its magnetic poles. Due to the fact that the radiation was emitted from the *poles* and was very directional, was the reason for the very specific pulse frequency, as well as the pulse width. Only when one of the pulsar's poles were pointing directly enough at Earth would the chart recorder pick up the signal and draw it on a piece of paper. The frequency of the signal was tied to the rotation of the star, which was fairly constant.

Upon this realization - the pulsar was accordingly named CP 1919 (*Cambridge Pulsar 1919*) due to its celestial coordinates of 19h and 19m of right ascension and +21 degrees of declination. Later on, it was renamed to *PSR B1919+21*.

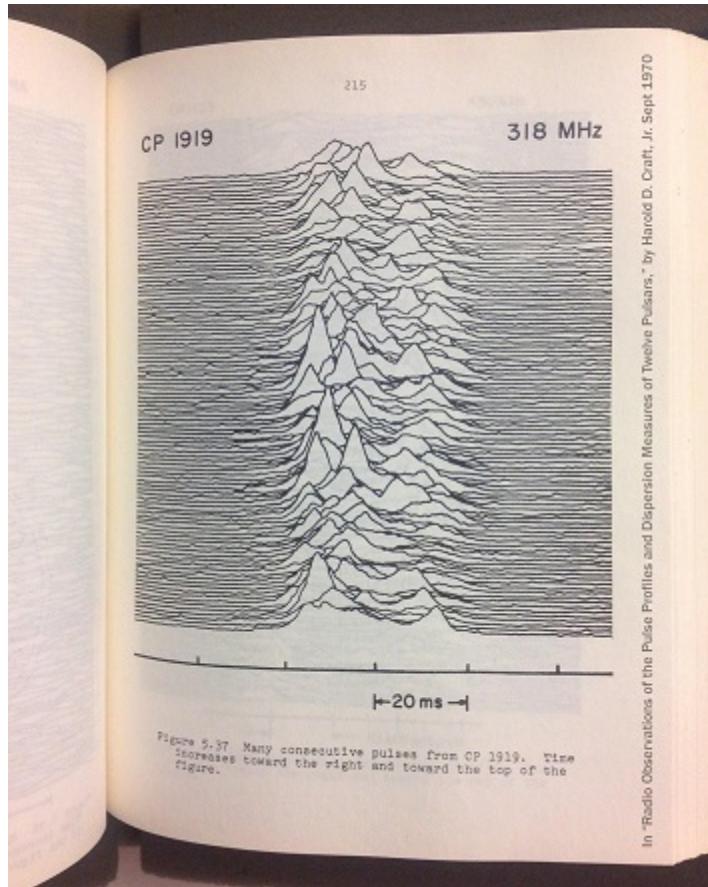
In 1979, the English band *Joy Division* was on the lookout to release a new album - "*Unknown Pleasures*". [Peter Saville](#)⁴⁵, the artist working with them came up with a brilliant idea for their album cover art. He referenced the work of *Harold D. Craft, Jr.* who created a wonderful data visualization rendition of the then-existing pulse

⁴³https://en.wikipedia.org/wiki/Jocelyn_Bell_Burnell

⁴⁴https://en.wikipedia.org/wiki/Chart_recorder

⁴⁵[https://en.wikipedia.org/wiki/Peter_Saville_\(graphic_designer\)](https://en.wikipedia.org/wiki/Peter_Saville_(graphic_designer))

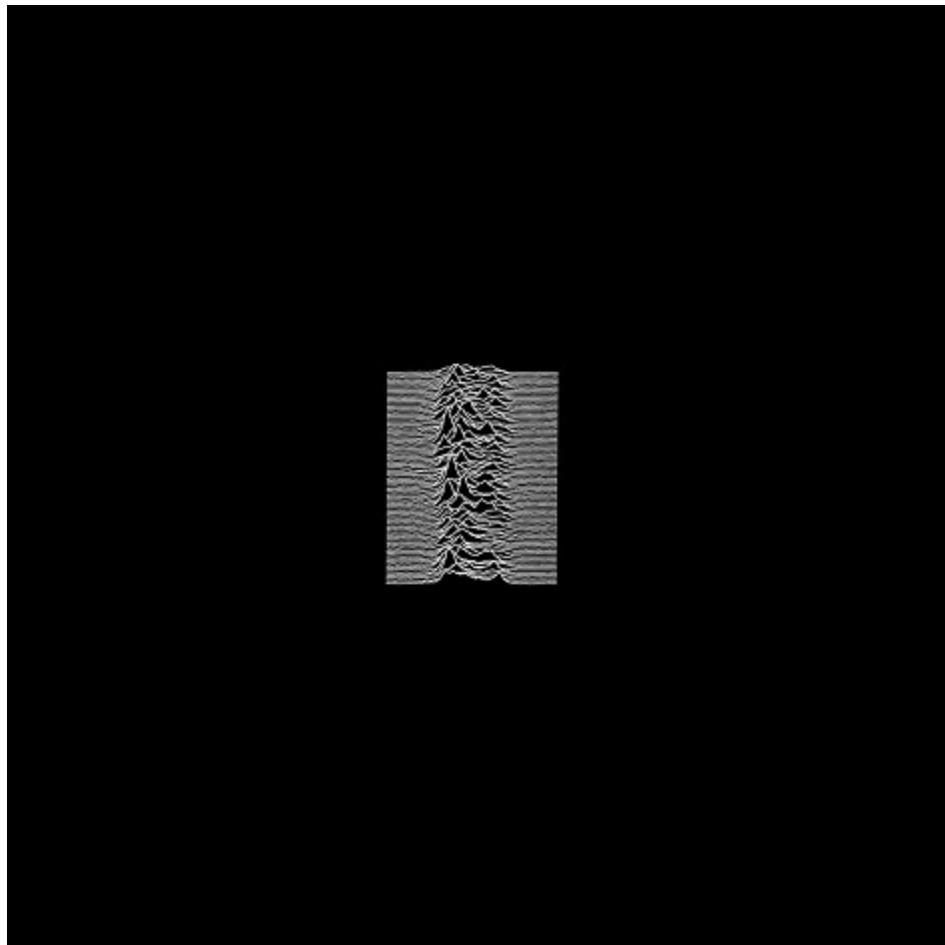
data of CP 1919, amongst other pulsars. It was released as a part of his [PhD Thesis in 1970⁴⁶](#), just 3 years after they were recorded by Jocelyn's chart recorder:



In “Radio Observations of the Pulse Profiles and Dispersion Measures of Twelve Pulsars”, by Harold D. Craft, Jr.

Harold’s work was a wonderful symbiosis of art and data – what Data Visualization really is. The plot was repurposed and released in its much popularized version with white lines and a black background as the *Joy Division*’s “*Unknown Pleasures*” album cover by Peter Saville:

⁴⁶<https://www.worldcat.org/title/radio-observations-of-the-pulse-profiles-and-dispersion-measures-of-twelve-pulsars-by-harold-d-craft-jr/oclc/657635342>



Unknown Pleasures Album Cover by Peter Saville

Plotting a Ridge Plot

Unfortunately, the *original data*, being marked on a piece of paper, was lost. We don't currently have a direct digital copy of the original pulses. However, due to the popularity of the album, and subsequently the plot itself - many people have *recreated the data*⁴⁷ based on the image. We've created a [fork of a GitHub repository](#)⁴⁸

⁴⁷<https://github.com/coolbutuseless/CP1919/blob/master/data-raw/clean.csv>

⁴⁸<https://github.com/StackAbuse/CP1919/blob/master/data-raw/clean.csv>

that contains exactly this data, in case the original repository goes out of commission, for posterity.

The data can be obtained from the `clean.csv` file, which contains the X and Y features for each *line*, of which there are 80. Let's go ahead and load in the dataset and draw Line Plots to visualize the distinctive waves:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv(r"https://raw.githubusercontent.com/StackAbuse/CP1919/master/data-raw/clean.csv")
5
6 print(df)
```

This results in:

```
1      line      x      y
2  0      80  0.378850  2.777900
3  1      80  11.831673  2.525300
4  2      80  20.417973  2.525300
5  3      80  30.014473  1.262700
6  4      80  43.398973  0.000000
7  ...
8  5058     1  565.647790  770.493813
9  5059     1  576.759470  771.503973
10 5060     1  603.023430  770.998893
11 5061     1  608.579270  770.998893
12 5062     1  623.226480  770.493813
13
14 [5063 rows x 3 columns]
```

Now, since the data is in tidy-form, where each *line* has multiple entries, and is supposed to be used to plot *one Line Plot*, we'll group the data by the *line* feature:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv(r"https://raw.githubusercontent.com/StackAbuse/CP1919/master/data-raw/clean.csv")
5
6 groups = df.groupby(['line'])
7
8 print(groups.get_group(1))
```

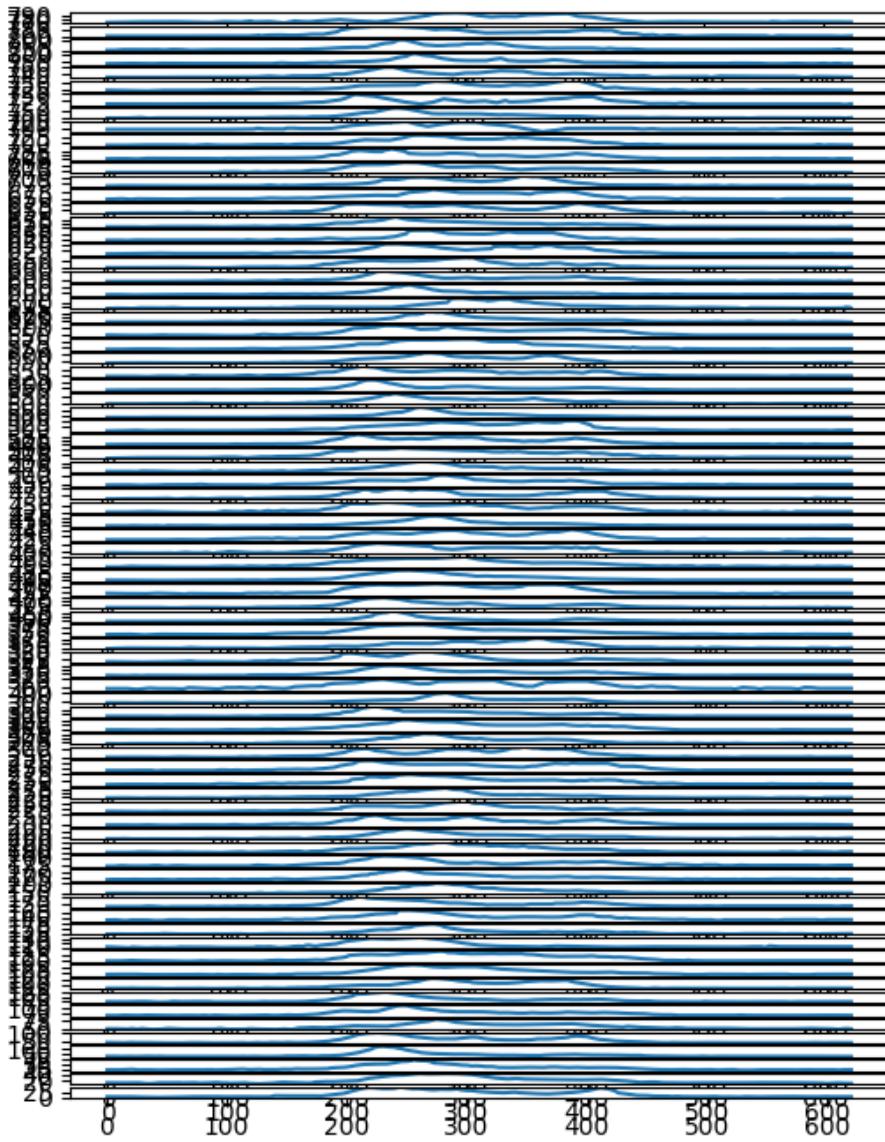
Now, just to take a peak, each *group* will contain the relevant data for the adequate line only. The first group will contain the *line*, *x* and *y* features representing the first line in the plot:

```
1      line      x      y
2  5011    1  0.000000  769.483663
3  5012    1  29.256773  770.998893
4  5013    1  58.551253  771.503973
5  5014    1  86.835453  769.483663
6 ...
```

Now, all that's left to do is plot a *Line Plot* for each group. The groups is of type `DataFrameGroupBy`, and each group in it is a *tuple* of a *Group Number* (`int`) and *Group Content* (`DataFrame`). While we loop through the groups, we'll want to reference the `DataFrame` for that group, which holds our data:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv(r"https://raw.githubusercontent.com/StackAbuse/CP1919/master/data-raw/cl\
5 ean.csv")
6 groups = df.groupby(['line'])
7 num_of_groups = len(groups.groups)
8
9 fig, ax = plt.subplots(num_of_groups, 1, figsize=(6, 8))
10
11 i = 0
12 for group in groups:
13     # Get DataFrame, plot X and Y from it in each Axes
14     ax[i].plot(group[1]['x'], group[1]['y'])
15     i += 1
16
17 plt.show()
```

This results in:



The first step is there - the *general lines* are there, but this plot is not stylized at all. The album cover had a black background, white lines, and the amplitudes of each line were much stronger. The *borders* of each `Axes` instance here are boxing the lines between each other. *Ridge Plots overlap*.

The `hspace` parameter defines the space between two `Axes` instances on a `Figure`. We can use the `subplots_adjust()` function of the `Figure` instance to change the `hspace` argument, just as we can change the `wspace` argument as well. Let's change the `hspace` to a *negative* float - which means that each `Axes` will impede on the axes above it and they'll overlap. To make this work, visually, we'll also want to remove the spines of each `Axes` instance. This can be done individually, by setting them to be invisible - or, we can simply turn them fully off, leaving *only* the line present, via the `axis()` function.

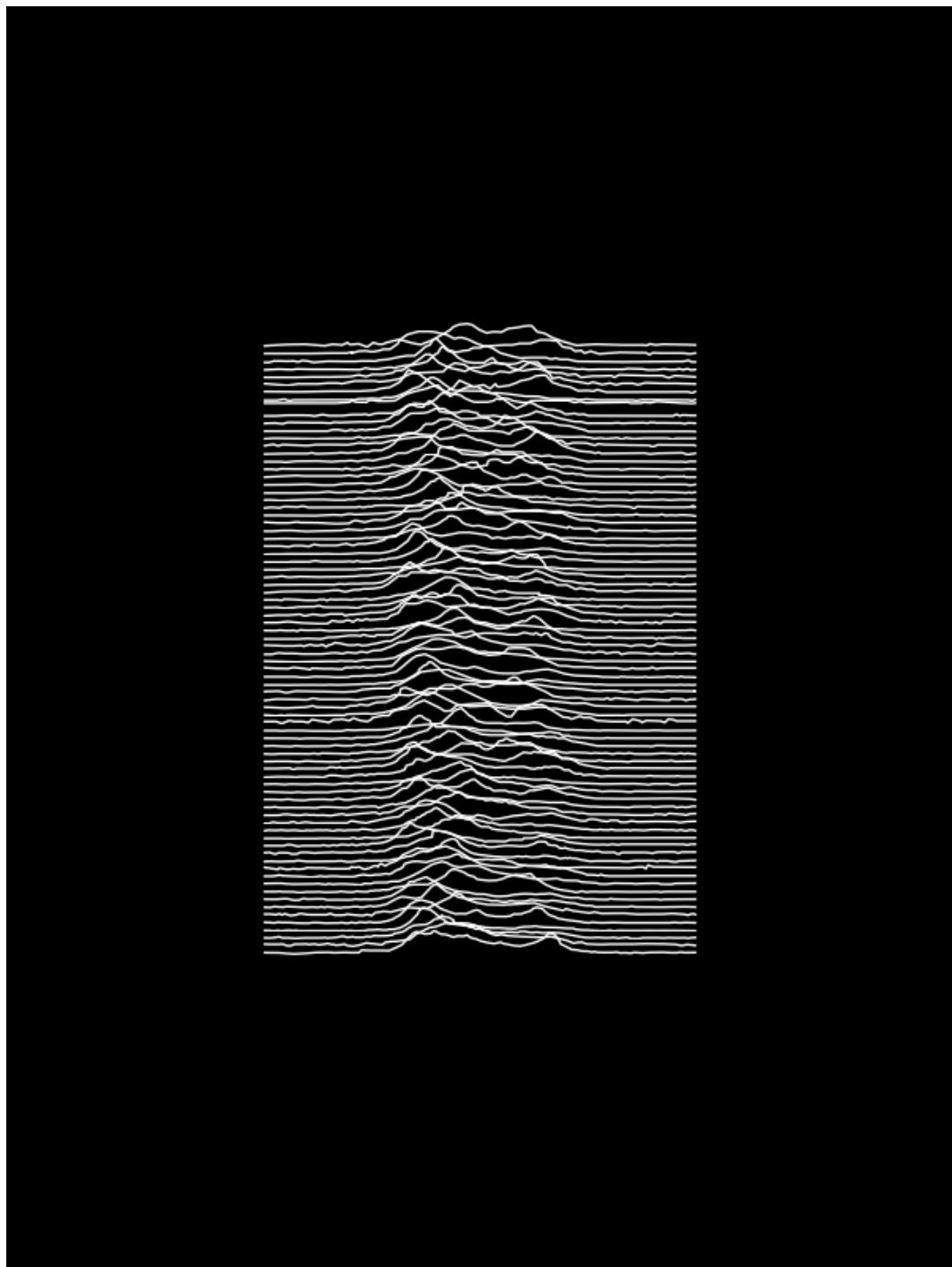
Additionally, the plot isn't stretched around the entire figure - it's in the center, smaller than the `Figure` would be. We can use the `left`, `right`, `top` and `bottom` arguments of the `subplots_adjust()` function to tweak these. Keep in mind - `left` and `bottom` *reduce the size* of the plot. The higher their value is, the smaller our plot will be in the `Figure`. By contrast, `right` and `top` *enlarge* the plot, so higher values result in a bigger plot. `left` and `bottom` cannot be equal to or larger than `right` and `top`, lest the image be flipped around - and to ensure that we maintain the ratio and keep the plot in the center, we'll make sure that the *sum* of each pair is equal to 1 - `left=0.25`, `bottom=0.25`, `right=0.75`, `top=0.75`.

We can set the background color (the color of the *face*) via the `facecolor` argument, as we've covered in *Chapter 5 - Basic Matplotlib Customization*, as well as the color of the lines via the `color` argument of the `plot()` function.

Tying that all together then boils down to just adding a few styling options to the code above:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv(r"https://raw.githubusercontent.com/StackAbuse/CP1919/master/data-raw/clean.csv")
5 groups = df.groupby(['line'])
6 num_of_groups = len(groups.groups)
7
8 fig, ax = plt.subplots(num_of_groups, 1, figsize=(6, 8), facecolor='black')
9 fig.subplots_adjust(hspace=-0.7, left=0.25, bottom=0.25, right=0.75, top=0.75)
10
11 i = 0
12 for group in groups:
13     ax[i].plot(group[1]['x'], group[1]['y'], color='white', linewidth=1)
14     ax[i].axis("off")
15     i += 1
16
17
18 plt.show()
```

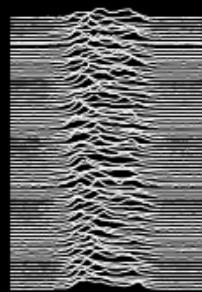
Running this code results in:



If you *really* want to make it closer to the original, we can tweak the `subplots_adjust()` arguments and the `linewidth` further:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv(r"https://raw.githubusercontent.com/StackAbuse/CP1919/master/data-raw/c1\
5 ean.csv")
6 groups = df.groupby(['line'])
7 num_of_groups = len(groups.groups)
8
9 fig, ax = plt.subplots(num_of_groups, 1, figsize=(6, 8), facecolor='black')
10 fig.subplots_adjust(hspace=-0.7, left=0.4, bottom=0.4, right=0.6, top=0.6)
11
12 i = 0
13 for group in groups:
14     ax[i].plot(group[1]['x'], group[1]['y'], color='white', linewidth=0.7)
15     ax[i].axis("off")
16     i += 1
17
18 plt.show()
```

This finally results in:

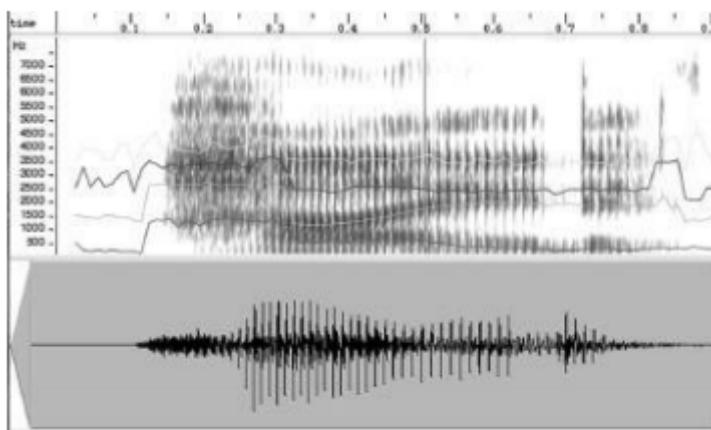


Now *this* is pretty close to the original.

Spectrogram Plots

Spectrograms are visual representations of the *spectrum of frequencies* of a signal in time-series. They are most commonly applied to audio signals, and are sometimes known as *sonographs* or *voicegrams* - though, the term *spectrogram* is much more common.

Spectrograms have several use-cases, but the most notable ones revolve around audio signals. As Ray Kurzweil, a famed inventor, futurist and best-selling author, notes in his 2012 book on “*How to Create a Mind*”⁴⁹ - in the early 1980s, his company who was developing a speech recognition model performed various transformations on the waveforms of speech data. The results of these transformations could be plotted as spectrograms, which allowed them to further analyze human speech and identify certain characteristics of phonemes:



A spectrogram of a person saying the word ‘hide’, as taken from Ray Kurzweil’s book - “How to Create a Mind”

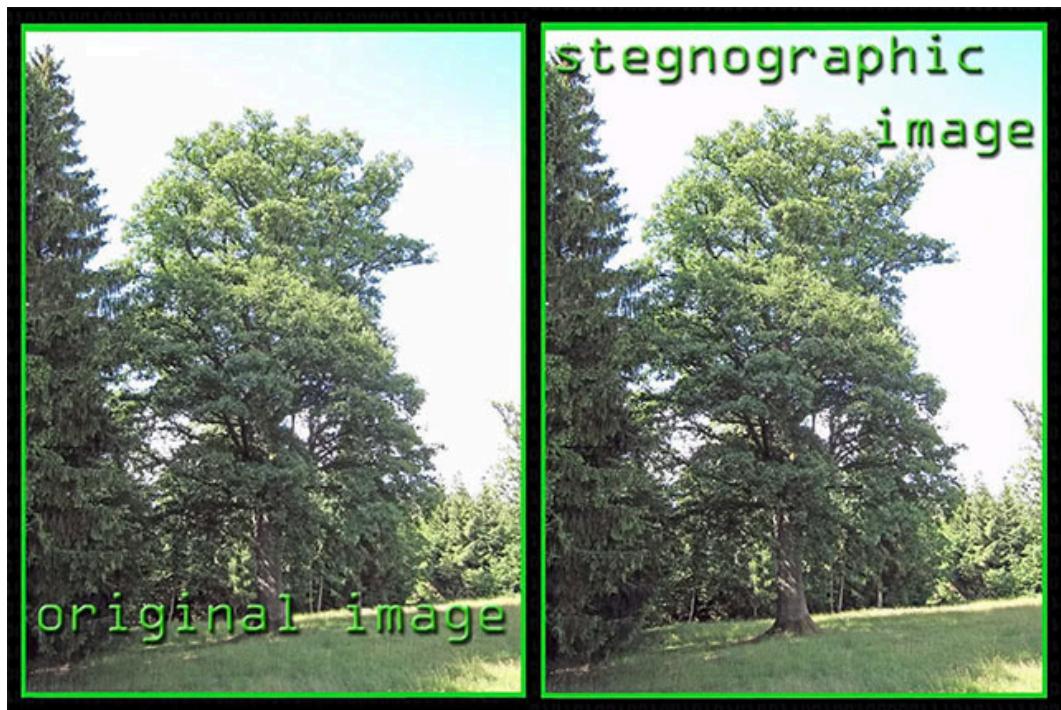
Although they’ve eventually moved to another model - a self-learning model using **Hidden Hierarchical Markov Models**⁵⁰ (HHMMs) - this analysis allowed them to gain a better understanding of the domain problem.

⁴⁹https://en.wikipedia.org/wiki/How_to_Create_a_Mind

⁵⁰https://en.wikipedia.org/wiki/Hierarchical_hidden_Markov_model

Spectrograms also have a use in decoding *steganography* - the art of hiding messages and data within other data or physical objects. Images, audio files, and regular files can have huge amounts of hidden data stored within them. As noted by *Dr. Mike Pound of the University of Nottingham*, in a [video released by Computerphile](#)⁵¹, it's remarkable how much data we can store in mediums such as images without having anyone notice it.

For reference, the image on the left is the raw, original image, while the image on the right contains hidden data, embedded in the data that constitutes the image itself:



Steganographic image vs Original Image by Dr. Mike Pound and Computerphile

How much data? The *entire works of William Shakespeare*. 39 plays, 154 sonnets and 3 poems. According to *Statista*, "*Hamlet*" alone has over 30,000 words⁵². All of this data was added simply by changing the two last (least) significant bits of a 8-bit per channel image. Although we don't perceive much of a difference between these images, due to the already varying intensities of colors and shapes - the data is

⁵¹<https://www.youtube.com/watch?v=TWEXCYQKyDc>

⁵²<https://www.statista.com/topics/7681/william-shakespeare/#dossierSummary>

definitely there.

In a similar manner - music can also hold hidden data. Though, it's easier to *hear* when something's off, than to notice a really slight change in one of the 16.7 million colors in an image we get from 8-bits per channel. We won't be able to hide as much data in spectrograms, though, it's fully possible.

Some music is created just *because* it holds some hidden data and images. This music typically doesn't really sound like music - more like a random set of noises and sounds.

That's where spectrograms can come in handy - they can display what we can't really see or notice, through the spectrum of frequencies.

Spectrograms plot the frequencies on the Y-axis, in ascending order, and time-series on the X-axis. Their Y-axis can be plotted as logarithmic or linear - depending on the use-case. Commonly (but not always), spectrograms of audio data are plotted logarithmically. This makes sense - since the *decibel* (unit of measurement for sound amplitude) is logarithmic in nature. A jet plane takeoff (120 dB) isn't only 4 times louder than whispering (30 dB).

On the other hand, tonal frequencies of human speech, which are typically ranging from around 85-180Hz for men and around 165-255Hz for women are usually plotted on a linear scale, since the ranges and their differences are so small.

Plotting a Spectrogram

Plotting a Spectrogram in Matplotlib can be achieved through the `specgram()` function of the `Axes` or `plt` instances. It accepts a 1D array of data, and computes a spectrogram to be plotted. You can tweak the computation via `NFFT`, `window` and `Fs`.

- `NFFT` - is the number of data points used in each block for the *Fast Fourier Transform (FFT)*⁵³.
- `Fs` - is the sampling frequency, used to calculate Fourier frequencies.
- `Window` - is the window function used in the FFT.

⁵³https://en.wikipedia.org/wiki/Fast_Fourier_transform

- *Scale* - is the Y-scale on which we're plotting, and can be `linear`, `dB` ($10^*\log_{10}$) or `default`. The `default` depends on the *mode* parameter.
- *Mode* - is the type of spectrum to use. The default is `psd` (power-spectral density⁵⁴), though other spectrums such as `magnitude` (magnitude spectrum⁵⁵), `angle` (phase spectrum without unwrapping) and `phase` (phase spectrum with unwrapping) exist.

Importing Data

We'll be using a music piece, called "*Look*"⁵⁶, created by *Venetian Snares* in the album "*Songs About My Cats 2001*". The music piece was created specifically to contain embedded data in the frequencies and due to a lack of other data, as well as the amount of embedded data - it's not really meant to be a piece of music. It sounds more like a science-fiction inspired set of sounds and electronic noises. In essence - it *sounds* like the *data that's embedded* since mostly only that data is actually present.

To load in the dataset, we'll be using *Scipy*'s `wavefile` module which allows us to extract data from a wavefile. Alternatively, you can access the data on [our GitHub](#)⁵⁷:

```

1 import matplotlib.pyplot as plt
2 from scipy.io import wavfile as wav
3 import numpy as np
4
5 rate, data = wav.read('Venetian-Snares-Look.wav')
6 print('2D data:', data)
7
8 data1, data2 = np.hsplit(data, 2)
# The hsplit() method returns a list with singular elements
# so we have to flatten it to produce a 1D array
11 data1 = data1.flatten()
12 data2 = data2.flatten()
13
14 print('Channel 1: ', data1)
15 print('Channel 2: ', data2)
```

The `read()` function returns the data itself, as well as its rate. This results in:

⁵⁴https://en.wikipedia.org/wiki/Spectral_density

⁵⁵https://wiki.seg.org/wiki/Magnitude_spectrum_and_phase_spectrum

⁵⁶<https://venetiansnares.bandcamp.com/track/look>

⁵⁷<https://github.com/StackAbuse/venetian-snares-look-data>

```
1 2D data: [[0 0]
2   [0 0]
3   [0 0]
4   ...
5   [0 0]
6   [0 0]
7   [0 0]]
8 Channel 1: [0 0 0 ... 0 0 0]
9 Channel 2: [0 0 0 ... 0 0 0]
```

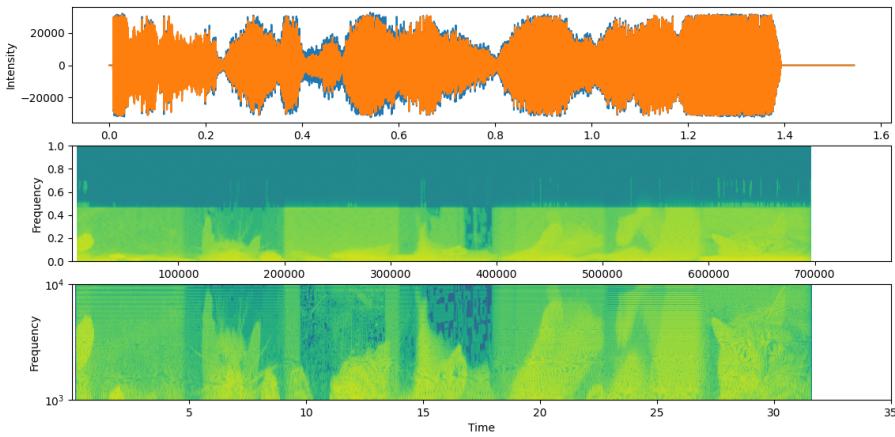
The start and end of the audio file are empty - they contain no sounds, so it's normal that the sneak peeks here don't really give us much. We've imported the waveform of the file via *Scipy*'s `wavfile` module, which results in a 2-channel 2D Numpy array.

To separate the channels, we've done a horizontal split (`hsplit()`) into two - resulting in the two channels. These hold the same data, so we can use only one if we'd like, but we'll plot both - setting a Y-limit on one of them to "zoom in" in the spectrogram. With 1D channels ready, we can plot the spectrograms, with the waveform in a separate Axes:

```
1 import matplotlib.pyplot as plt
2 from scipy.io import wavfile as wav
3 import numpy as np
4
5 rate, data = wav.read('Venetian-Snares-Look.wav')
6
7 data1, data2 = np.hsplit(data, 2)
8 data1 = data1.flatten()
9 data2 = data2.flatten()
10
11 fig, ax = plt.subplots(3, 1)
12
13 ax[0].plot(data)
14 ax[0].set_xlabel('Time')
15 ax[0].set_ylabel('Intensity')
16
17 ax[1].specgram(data1)
18 ax[1].set_xlabel('Time')
19 ax[1].set_ylabel('Frequency')
20
21 ax[2].specgram(data2, NFFT=1024, Fs=rate, noverlap=512, scale='default')
22 ax[2].set_xlabel('Time')
23 ax[2].set_ylabel('Frequency')
24 ax[2].set_yscale('symlog')
25 ax[2].set_ylim(100, 10000)
26
27 plt.show()
```

We've used the returned `rate` as the `Fs` (sampling frequency) for our spectrograms. For the first spectrogram, we haven't set any explicit settings and used the default

ones. For the other one, we've changed them up a bit to tweak the look of the spectrogram:



The first spectrogram isn't very clear, the data we're trying to showcase is at the bottom half of the frequency range and we're in the wrong scale. The ears of the cats get progressively longer up the Y-axis, since it's meant to be plotted on a logarithmic scale.

In the second plot, we've set the Y-scale to a logarithmic one, and limited the axis limits to focus on the interesting part. Additionally, we've tweaked some of the parameters used for the FFT, so that the images are a bit more clear.

It's amazing how the waveform right above these images actually contains the data to plot images of cats *hidden in its frequencies*. What's probably even more amazing is how cute a feline has to be to inspire an information-wielding human to embed their image in the frequency range of a song. The biological evolution of domesticated cats evolved certain features, with no foresight as to how far it'll get the species.

Maybe one day, we'll embed data like this onto *gold plates* on spacecraft, such as the [Voyager 2](#)⁵⁸, gaining Solar escape velocity, before setting it on a long journey through outer space. This certain set of changes made to the *Felis* genus might get data on itself far beyond the Solar System.

⁵⁸https://en.wikipedia.org/wiki/Voyager_2

Matplotlib - 3D Support

Matplotlib has 3D support. Several types of plots can be turned into their 3D variants fairly easily, and using what we've learned so far is very applicable to this use-case.

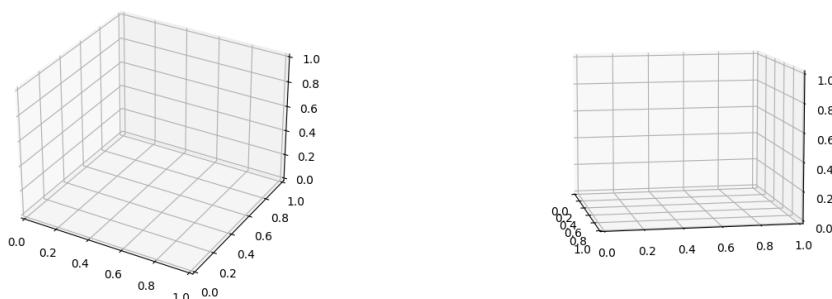
For 3D plotting, we don't use the regular `Axes` object. We use the `Axes3D` object. Thankfully, we don't have to really change anything ourselves - when creating the `Axes` instances, we can just pass in a `projection='3d'` as the argument to the `add_subplot()` function - which changes the returned `Axes` instance into an `Axes3D`.

It's worth noting that the `subplots()` function doesn't accept the `projection` argument. You'll have to create a `Figure` instance via the `figure()` call, and *then* add an `Axes3D` to it via the `add_subplot()` function and the `projection` argument. This is mainly done because we can combine multiple `Axes3D` and `Axes` instances on a single `Figure` - the `Figure` itself is projection-agnostic.

Note: Even though we don't actually reference it *anywhere*, we have to import `Axes3D` from the `mpl_toolkits.mplot3d` module, dedicated to plotting 3D plots:

```
1 import matplotlib.pyplot as plt
2 from mpl_toolkits.mplot3d import Axes3D
3
4 fig = plt.figure()
5 ax = fig.add_subplot(projection = '3d')
6
7 plt.show()
```

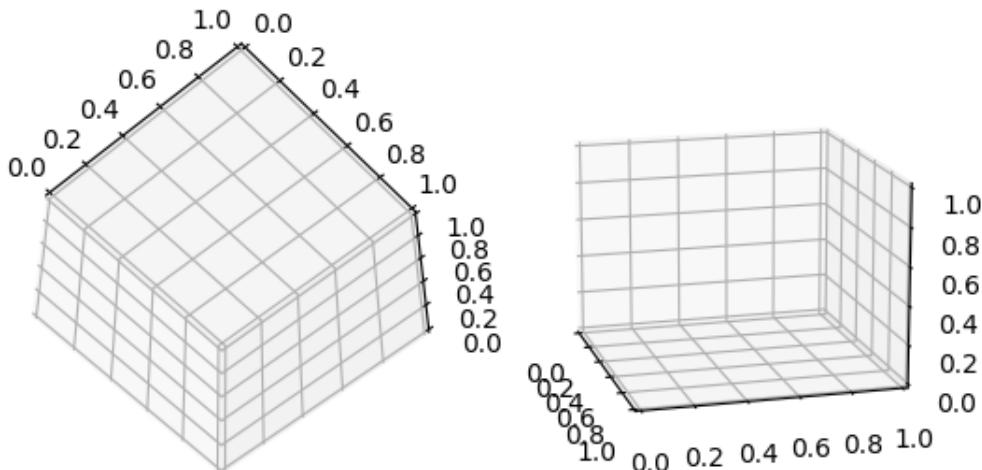
Running this piece of code will spin off a `Figure` with a 3D axes:



The `Axes3D` is interactive and can be panned and rotated in all three dimensions - the spines will adapt to the perspective you're looking from automatically. The image above displays the same `Axes3D` instance from two perspectives, combined into a single image.

Additionally, if you have two different `Axes3D` instances, you can rotate them totally independently of each other:

```
1 import matplotlib.pyplot as plt
2 from mpl_toolkits.mplot3d import Axes3D
3
4 fig = plt.figure()
5 ax = fig.add_subplot(121, projection = '3d')
6 ax2 = fig.add_subplot(122, projection = '3d')
7
8 plt.show()
```



Then, where we'd usually supply 2 dimensions (features) to plot - we'll now supply 3. What this allows us to do is to either create a synthetic dimension, such as a “time” dimension, or actually use any other dataset-provided feature, which might as well be a time-series feature. Let's revisit some of the plots we've covered before to see

what they look like plotted in 3D. We won't dedicate entire sections to them - since the only real addition is the Z-axis - the third feature.

3D Scatter Plots and Bubble Plots

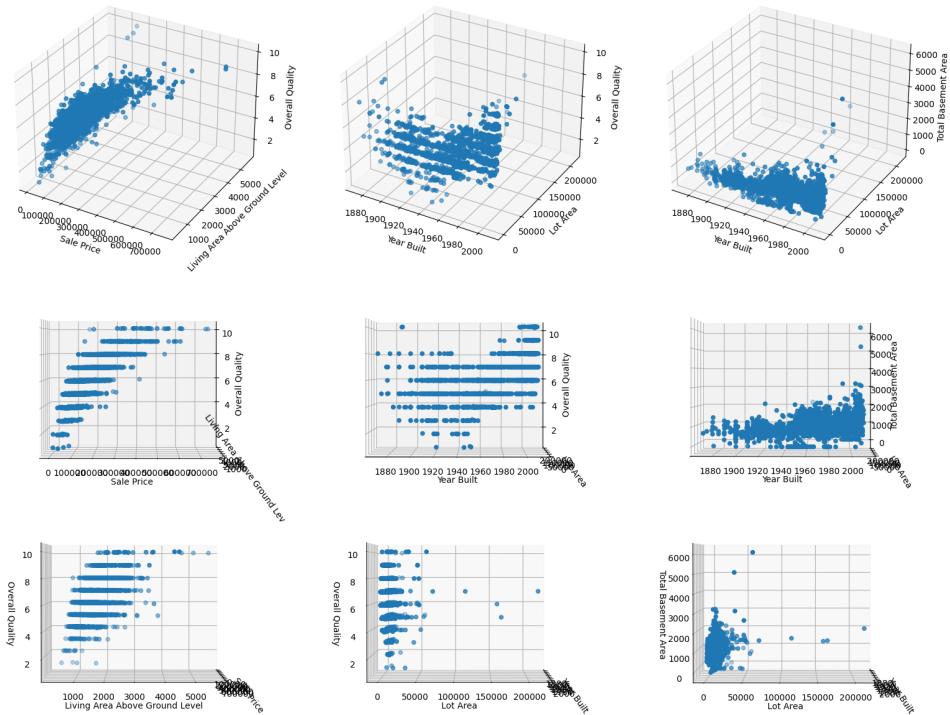
Let's start off with Scatter Plots - these are pretty common to do in 3 dimensions. This time around - we can explore the relationship of three features. The third one is oftentimes a time-related feature, so you can effectively visualize the relationship between two over time. However, this isn't a requirement.

Let's revisit the *Ames Housing Dataset* and do some of the older Scatter Plots, introducing a new feature:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 from mpl_toolkits.mplot3d import Axes3D
4
5 df = pd.read_csv('AmesHousing.csv')
6
7 fig = plt.figure()
8 ax = fig.add_subplot(131, projection = '3d')
9 ax2 = fig.add_subplot(132, projection = '3d')
10 ax3 = fig.add_subplot(133, projection = '3d')
11
12 sale_price = df['SalePrice']
13 gr_liv_area = df['Gr Liv Area']
14 overall_qual = df['Overall Qual']
15 lot_area = df['Lot Area']
16 total_bsmt_sf = df['Total Bsmt SF']
17 year_built = df['Year Built']
18
19 ax.scatter(sale_price, gr_liv_area, overall_qual)
20 ax.set_xlabel("Sale Price")
21 ax.set_ylabel("Living Area Above Ground Level")
22 ax.set_zlabel("Overall Quality")
23
24 ax2.scatter(year_built, lot_area, overall_qual)
25 ax2.set_xlabel("Year Built")
26 ax2.set_ylabel("Lot Area")
27 ax2.set_zlabel("Overall Quality")
28
29 ax3.scatter(year_built, lot_area, total_bsmt_sf)
30 ax3.set_xlabel("Year Built")
31 ax3.set_ylabel("Lot Area")
32 ax3.set_zlabel("Total Basement Area")
33
34 plt.show()
```

This results in a Figure with 3 Axes3D instances, each of which we can rotate in 3D (first row in the proceeding image). Matching two features together will effectively

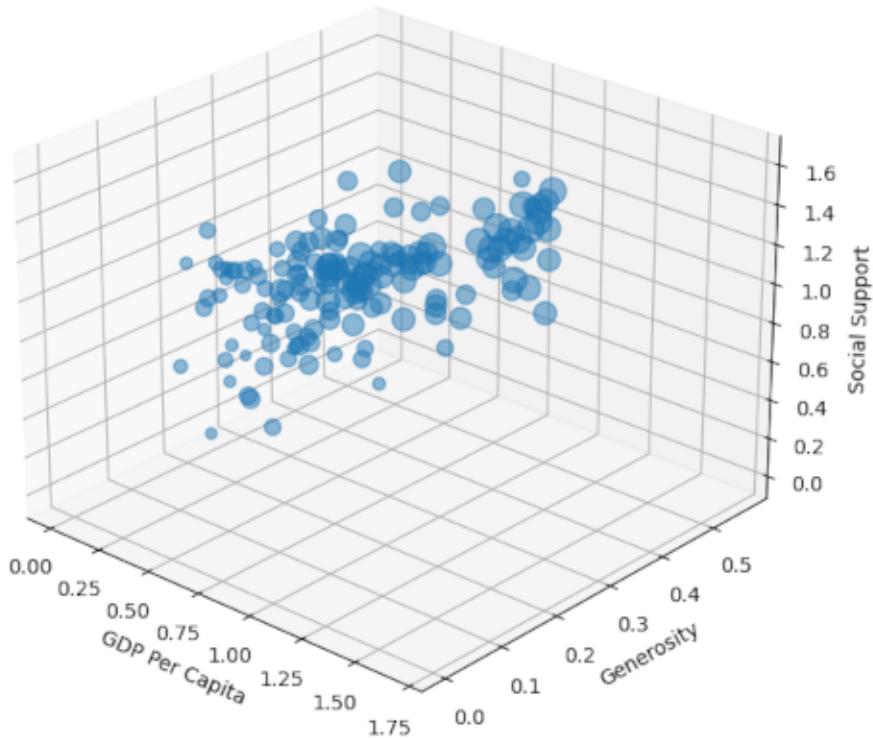
produce a 2D plot again between the first and second feature (second row) and we can rotate once again to get a 2D plot between the second and third feature (third row):



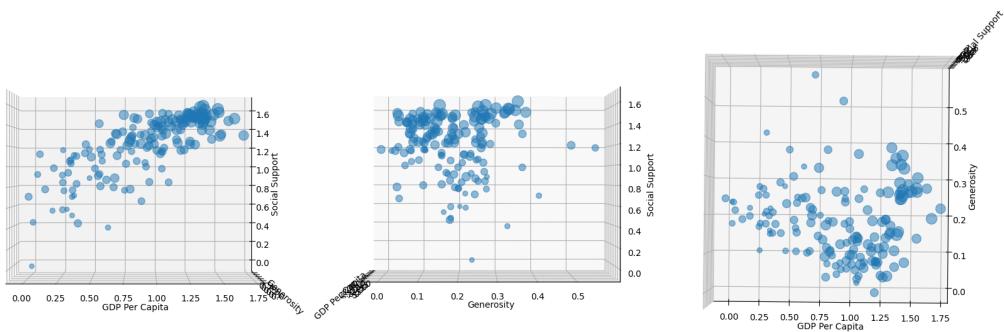
Changing the `s` parameter here can be used to generate a Bubble Plot, just like we did before. Let's revisit the *World Happiness Dataset*, in which we made a Bubble Plot. This time around, the Bubble Plot will likely look more like a plot of actual bubbles:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 from mpl_toolkits.mplot3d import Axes3D
4
5 df = pd.read_csv('worldHappiness2019.csv')
6
7 fig = plt.figure()
8 ax3d = fig.add_subplot(111, projection = '3d')
9 gdp = df['GDP_per_capita']
10 generosity = df['Generosity']
11 social_support = df['Social support']
12
13 size = df['Score'].to_numpy()
14 s = [3*s**2 for s in size]
15
16 ax3d.scatter(gdp, generosity, social_support, s = s, alpha=0.5)
17 ax3d.set_xlabel('GDP Per Capita')
18 ax3d.set_ylabel('Generosity')
19 ax3d.set_zlabel('Social Support')
20
21 plt.show()
```

This results in:



Let's explore the visualization a bit in 3D:



Here, we can draw more conclusions than we did before:

- *GDP per Capita* and *Social Support* seem to have a *strong positive correlation*.
- *Generosity* and *Social Support* don't seem to have any correlation at all.
- *Generosity* and *GDP per Capita* seems to have a *weak negative correlation*.

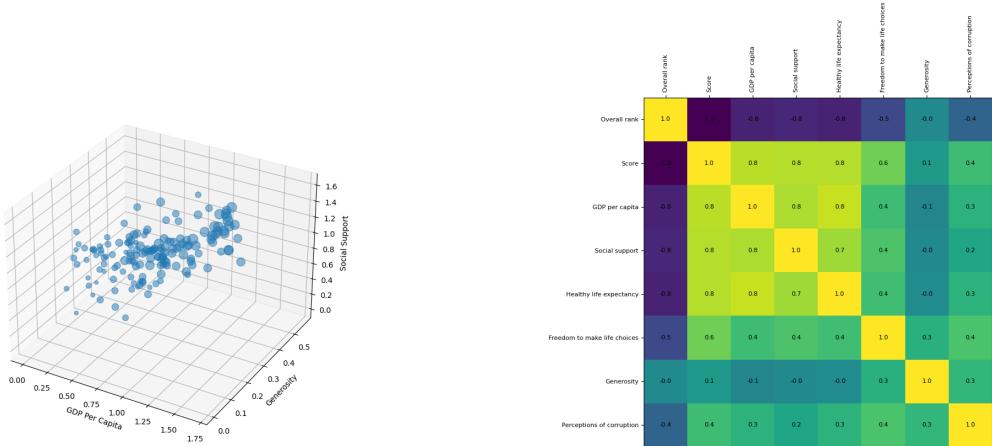
Though, we don't have to take a guess at this. If we've learned anything so far - there's a plot for just about anything. Heatmaps/Correlation Matrices are great especially for this, so let's go ahead and calculate the correlations between these features and plot them on a Heatmap.

Combining 2D and 3D Plots

Nothing stops us from creating a combination of 2D plots and 3D plots. Since `Figures` are projection-agnostic, we can have multiple `Axes3D` and `Axes` instances on a single `Figure`. Since we don't want to point and guess as to which correlations are strong or weak, positive or negative - let's have Pandas do the calculations for us and simply plot a Heatmap next to our Scatter Plot:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 from mpl_toolkits.mplot3d import Axes3D
4
5 df = pd.read_csv('worldHappiness2019.csv')
6 # Create a Figure
7 fig = plt.figure()
8
9 # Extract features in variables we can call more cleanly
10 gdp = df['GDP per capita']
11 generosity = df['Generosity']
12 social_support = df['Social support']
13 correlations = df.corr()
14
15 # Calculate the size for Scatter Plot markers to make a Bubble Plot
16 size = df['Score'].to_numpy()
17 s = [3*s**2 for s in size]
18
19 # Add Axes3D instance and plot on it
20 ax3d = fig.add_subplot(121, projection = '3d')
21 ax3d.scatter(gdp, generosity, social_support, s = s, alpha=0.5)
22 ax3d.set_xlabel('GDP Per Capita')
23 ax3d.set_ylabel('Generosity')
24 ax3d.set_zlabel('Social Support')
25
26 # Add Axes instance and plot on it
27 ax = fig.add_subplot(122)
28 ax.matshow(correlations)
29
30 # Add correlation values to the heatmap boxes
31 plt.yticks(range(0, len(correlations.index)),
32             correlations.index, fontsize=8, )
33 plt.xticks(range(0, len(correlations.columns)),
34             correlations.columns, fontsize=8, rotation=90)
35
36 for y in range(correlations.shape[0]):
37     for x in range(correlations.shape[1]):
38         plt.text(x, y, '%.1f' % correlations.iloc[x, y],
39                  horizontalalignment='center',
40                  verticalalignment='center',
41                  fontsize=8
42                  )
43 # Set layout to tight to avoid overlapping long X/Y-tick labels
44 # between the two axes
45 plt.tight_layout()
46
47 plt.show()
```

Now, we practically have a cheatsheet of the correlations on the right-hand side, and they confirm our earlier estimates:



- *GDP per Capita* and *Social Support* a *strong positive correlation* - **0.8**.
- *Generosity* and *Social Support* don't have any correlation at all - **0.0**.
- *Generosity* and *GDP per Capita* have a *weak negative correlation* - **-0.1**.

3D Surface Plots and Wireframe Plots

When we stop to think for a moment - Heatmaps could very well be 3D as well. They already have three features, actually - we could use the correlations here as an *intensity* on the Y-axis. Imagine the Heatmap as a *pliable surface*, much like a table cloth, being put on a clay tablet with certain terrain. The table cloth takes the shape of the clay beneath it.

In much the same way - we can *bend* and reshape a Heatmap based on another feature, like the correlation coefficients, for example. This type of plot is called a *Surface Plot*. A variant of it, *Wireframe Plots* exist too. Wireframe Plots convey the exact same data, but consist of a wire-grid without filled/colored rectangles.

To plot a *Surface Plot* in Matplotlib, we call the `plot_surface()` function, providing 3 features (X, Y and Z) and a color map for them. For a *Wireframe Plot*, we call the `plot_wireframe()` function, providing 3 features.

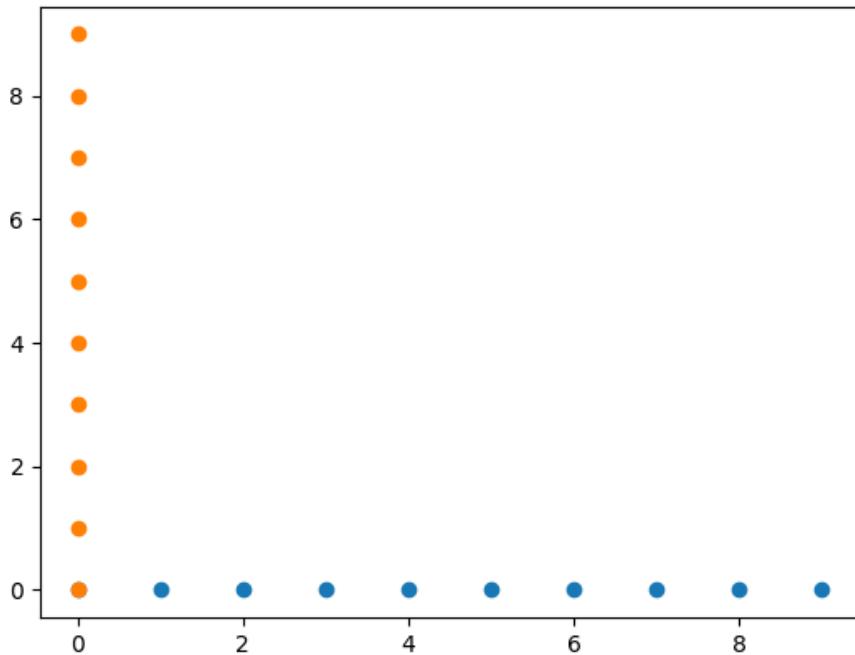
Let's re-plot the Heatmap from the previous section as a *Surface Plot*. We'll first have to work a bit with the data to provide a valid data-form to the `plot_surface()` function. The *X*, *Y* and *Z* features need to be 2D arrays - not 1D. So far, we've been mainly working with 1D data for features. And the features we want to visualize in the Surface Plot (*Generosity*, *GDP per capita*, etc.) are 1D.

To plot the relationship between *X* and *Y*, with the *Z* being the height above/below the *XY plane* - we'll need to create a *meshgrid* from the *X* and *Y* features. A *meshgrid* is created from two 1D arrays, and is itself a 2D array - a *grid* made from the two arrays. Since this is a book on Data Visualization, let's visualize this process, which should make it fairly clear.

To create a *meshgrid*, we'll use Numpy's `meshgrid()` function:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots()
5
6 nums = np.arange(0, 10, 1)
7 dummy = [0]*10
8
9 X = nums
10 Y = nums
11
12 ax.scatter(X, dummy)
13 ax.scatter(dummy, Y)
```

Our *X* and *Y* features are 1D arrays of data - simply spanning from [0..9]. We've added an arbitrary *dummy* feature which contains a series of 0s, since we don't want Matplotlib to auto-insert another [0..9] feature, resulting in a linear plot:



These are the two 1D features we'd like to visualize in a Surface Plot. This view is similar to what we'd have as a top-down view of the Surface Plot. Essentially, a top-down view of a Surface Plot can actually be equivalent to a Heatmap. Here, it's obvious what's missing - we've only got one row and one column filled with data.

Now, let's `meshgrid()` these two and see what shape they take:

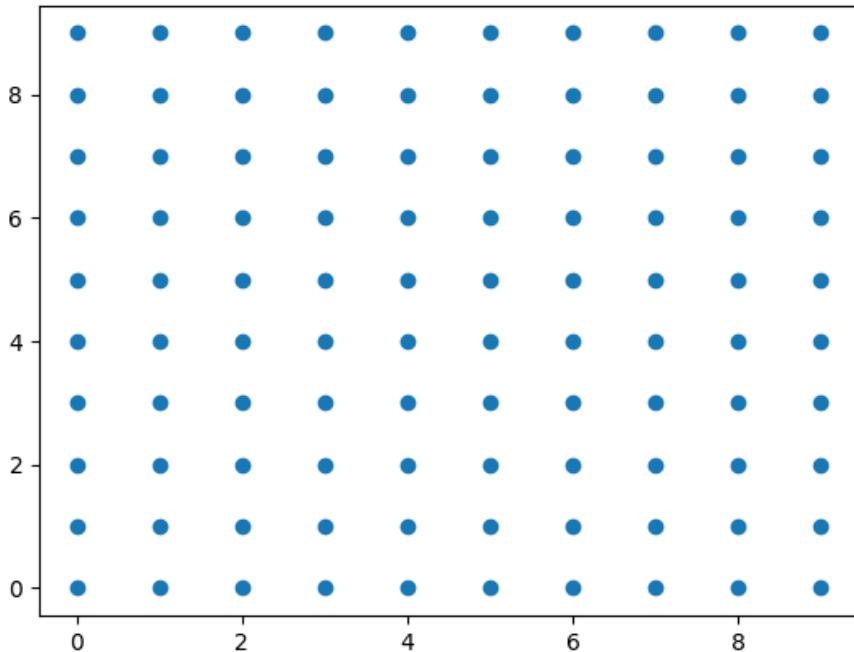
```
1 import numpy as np
2
3 nums = np.arange(0, 10, 1)
4 X = nums
5 Y = nums
6
7 print('X before meshgrid: ', X)
8 print('Y before meshgrid: ', Y)
9
10 X, Y = np.meshgrid(X, Y)
11
12 print('X after meshgrid: ', X)
13 print('Y after meshgrid: ', Y)
```

We've unpacked the return value of the `meshgrid()` function back to `X` and `Y` and added a couple of `print()` statements to compare them:

```
1 X before meshgrid: [0 1 2 3 4 5 6 7 8 9]
2 Y before meshgrid: [0 1 2 3 4 5 6 7 8 9]
3 X after meshgrid: [[0 1 2 3 4 5 6 7 8 9]
4 [0 1 2 3 4 5 6 7 8 9]
5 [0 1 2 3 4 5 6 7 8 9]
6 [0 1 2 3 4 5 6 7 8 9]
7 [0 1 2 3 4 5 6 7 8 9]
8 [0 1 2 3 4 5 6 7 8 9]
9 [0 1 2 3 4 5 6 7 8 9]
10 [0 1 2 3 4 5 6 7 8 9]
11 [0 1 2 3 4 5 6 7 8 9]
12 [0 1 2 3 4 5 6 7 8 9]]
13 Y after meshgrid: [[0 0 0 0 0 0 0 0 0 0]
14 [1 1 1 1 1 1 1 1 1 1]
15 [2 2 2 2 2 2 2 2 2 2]
16 [3 3 3 3 3 3 3 3 3 3]
17 [4 4 4 4 4 4 4 4 4 4]
18 [5 5 5 5 5 5 5 5 5 5]
19 [6 6 6 6 6 6 6 6 6 6]
20 [7 7 7 7 7 7 7 7 7 7]
21 [8 8 8 8 8 8 8 8 8 8]
22 [9 9 9 9 9 9 9 9 9 9]]
```

The `meshgrid()` function expands these into 2D arrays, filling in the gaps required to create a *grid* of values for these two specific arrays. Plotting `X` and `Y` would now return:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 fig, ax = plt.subplots()
5 nums = np.arange(0, 10, 1)
6 X = nums
7 Y = nums
8
9 X, Y = np.meshgrid(X, Y)
10
11 ax.scatter(X, Y)
12
13 plt.show()
```

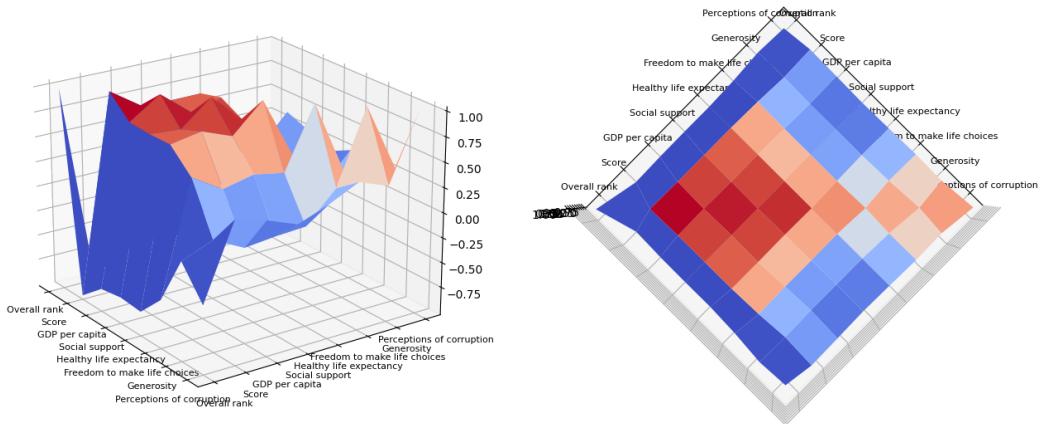


Great! We've turned two individual 1D arrays into a *meshgrid*, which is suitable for plotting Surface Plots. Each of these markers will be a “vertex”. Each vertex will be joined with 4 other vertices, with straight lines, forming “surfaces” between them. The height of each vertex will be determined by the Z feature - in our case, the correlation values.

Now, we just need to apply this exact same logic to our *World Happiness Dataset*:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4 from matplotlib import cm
5 from mpl_toolkits.mplot3d import Axes3D
6
7 df = pd.read_csv('worldHappiness2019.csv')
8 fig = plt.figure()
9
10 # Calculate correlations
11 correlations = df.corr()
12 # Create new values for the X and Y axis, since column names (strings)
13 # won't work here
14 X = Y = nums = np.arange(0, len(correlations.columns), 1)
15 # Create a *meshgrid* using X and Y
16 X, Y = np.meshgrid(X, Y)
17 # Extract the correlation values into Z (2D array)
18 Z = correlations.values
19
20 ax3d = fig.add_subplot(111, projection = '3d')
21 ax3d.plot_surface(X, Y, Z, cmap=cm.coolwarm)
22
23 plt.yticks(range(0, len(correlations.index)), correlations.index, fontsize=8, )
24 plt.xticks(range(0, len(correlations.columns)), correlations.columns, fontsize=8)
25
26 plt.show()
```

This results in a Surface Plot where the *correlation values* [-1..1] are the *height* of our surface plot vertices, compared to the XY plane. We've also imported the `cm` module from `matplotlib`, which allows us to set a `cmap` of the surfaces of the Surface Plot. We've chosen a `coolwarm` color map, since it goes along with the concept of the plot we're creating. Though - one thing to note here is that this will be different from the heatmap we've plotted before:

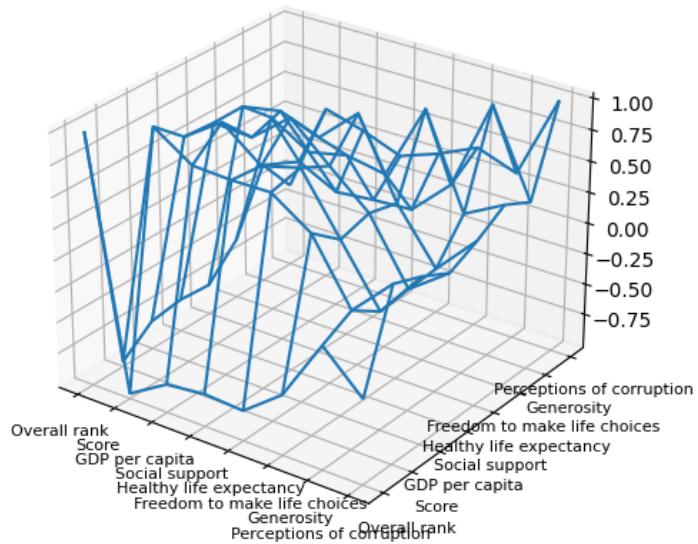


Here, the *height of the vertices* is defined by the correlation values. This doesn't mean that the *height of the surfaces* are. For instance, we can see that the diagonal correlation line is, as usual, filled with values of 1.0 . The vertices reflect this (as can be seen on the left-hand side image), however, the *surfaces between them* depend on other values as well. To this end, the surfaces between *Generosity* and *Generosity* are way below 1.0 , even though its *vertex* is right at 1.0 . Since we can't color a vertex (conceptually an infinitely small point) - if looking from the top-down, the Surface Plot can actually throw us off compared to the Heatmap.

Wireframe Plots are a *meshgrid*, where each marker ("vertex") is joined with 4 other vertices, but the surfaces *between them aren't filled*. They work in much the same way as *Surface Plots*, but don't have colored, surfaces. If we were to change the previous piece of code by just replacing `plot_surface()` with `plot_wireframe()`:

```
1 ax3d.plot_wireframe(X, Y, Z)
```

The code would generate:



Projecting Surface Plots with Contour Plots

Contour Plots are a type of 2D plot, used for visualizing 3D data. In essence - they're the *projection* from different perspectives of something like a Surface Plot. Contour Plots consist of *contours* (Z-axis slices). They allow us to plot the relationship between 3 features on a 2D plot - something that we so far have either used multiple 2D plots for, or a single 3D plot.

Contour Plots are usually, but not always, color-coded so that they can be more easily interpreted. The first Contour Plot most people see in their early school years is a *topographic map*⁵⁹ in Geography class. Topographic maps represent 2D maps of the world with the third dimension being the elevation compared to the world's water level. This third dimension is represented as lines (contours) on top the underlying

⁵⁹https://en.wikipedia.org/wiki/Topographic_map

2D map. These contours are essentially lines connecting points (vertices) of equal elevation. When put on a 3D surface, we can make out which parts of the surface are on a higher elevation level than others, and to which degree.

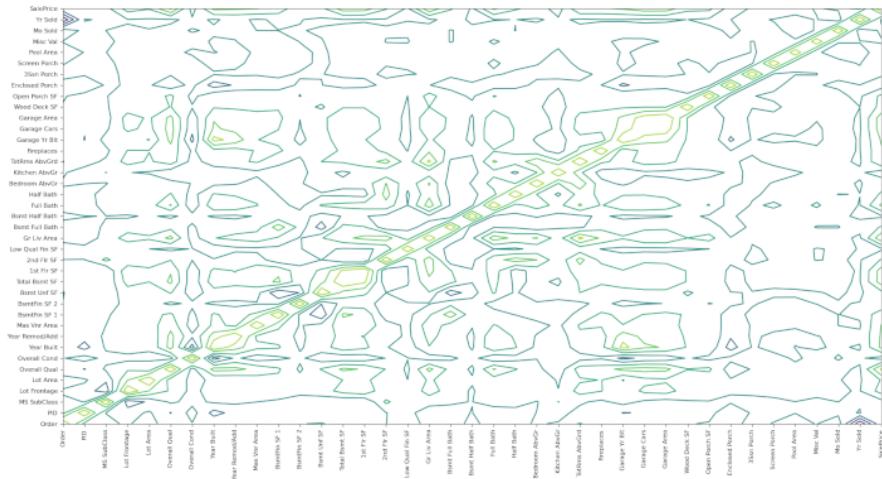
We get to see the X and Y relationship in 2D, and the colors on the plot (the frequency of *contours*) inform us about the Z feature and its values. Being practically a projection of a Surface Plot - Contour Plots use the very same data as Surface Plots.

Plotting a Contour Plot

In Matplotlib, to plot a Contour Plot, we use the `contour()` function on an `Axes` or `plt` instance, passing in the three features we'd like to plot. Let's plot a Contour Plot of the correlation of features in the *Ames Housing Dataset*:

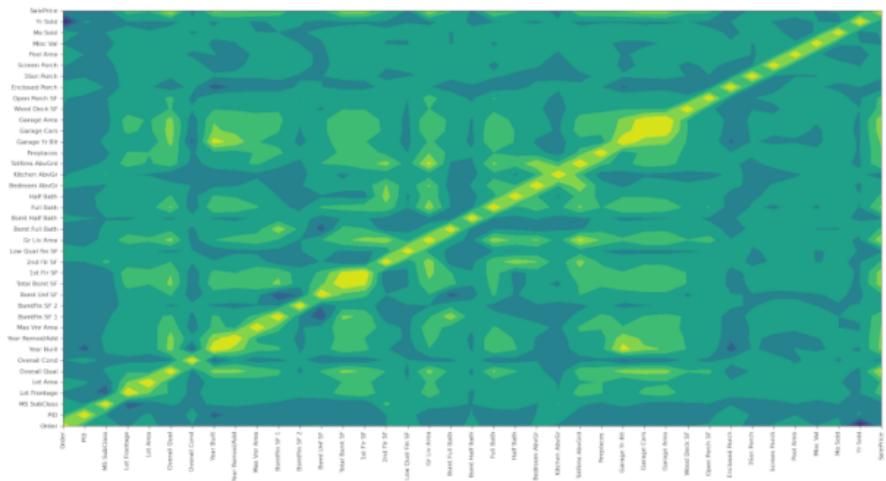
```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 df = pd.read_csv('AmesHousing.csv')
6 fig, ax = plt.subplots()
7 correlations = df.corr()
8
9 X = Y = range(0, len(correlations.columns))
10
11 X, Y = np.meshgrid(X, Y)
12 Z = correlations.values
13
14 ax.contourf(X, Y, Z)
15
16 plt.yticks(range(0, len(correlations.index)),
17             correlations.index, fontsize=8)
18 plt.xticks(range(0, len(correlations.columns)),
19             correlations.columns, fontsize=8, rotation=90)
20
21 plt.show()
```

Here, we've calculated the correlations between all the features, and plotted the contour of the surface they've formed. This will, to a large degree, look like a Heatmap, as we've covered before:



It's worth noting that the `contour()` function has a similar, overloaded partner - `contourf()`. It simply *fills in* the spaces between the contours, since it looks a bit "skinny" right now. It accepts the same set of parameters, so we can just switch them out when desired:

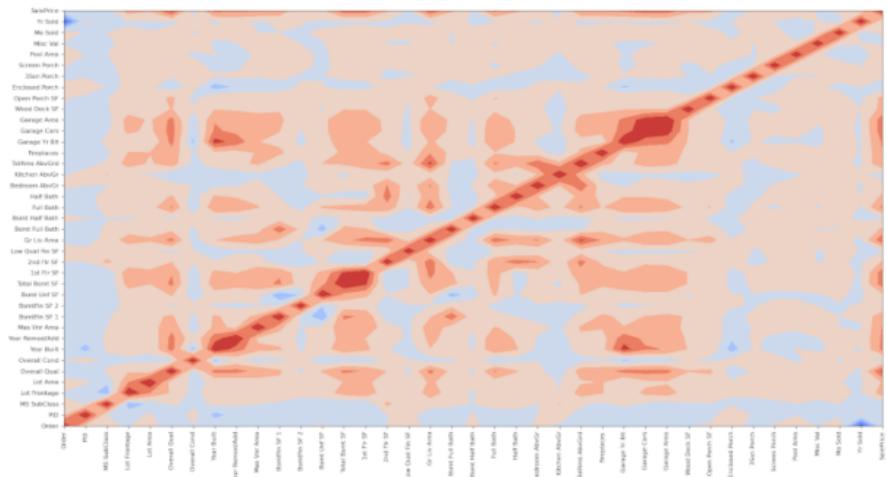
```
1 ax.contourf(X, Y, Z)
```



Contour Plots also accept a `cmap` argument, which can be used to change the colormap/scheme:

```
1 # imports...
2 from matplotlib import cm
3
4 ax.contourf(X, Y, Z, cmap=cm.coolwarm)
```

Which results in:



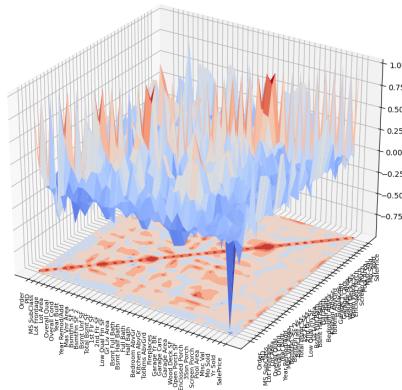
Plotting a Surface Plot with Contour Plots

Given that Contour Plots and Surface Plots work on the exact same data and one is a projection of the other - we can plot both on the same Axes3D instance, for the convenience of not having to pan and rotate much to view certain perspectives, while still retaining some other perspectives in our view.

When plotting Surface Plots like the one we've done in the previous section - we can “flatten” a contour plot on any of the spines as a projection of the surface plot from that perspective. Let's go ahead and plot a Surface Plot of these same three features coupled with a Contour Plot:

```
1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4 from matplotlib import cm
5 from mpl_toolkits.mplot3d import Axes3D
6
7 df = pd.read_csv('AmesHousing.csv')
8 fig = plt.figure()
9 correlations = df.corr()
10
11 X = Y = range(0, len(correlations.columns))
12
13 X, Y = np.meshgrid(X, Y)
14 Z = correlations.values
15
16 ax3d = fig.add_subplot(111, projection = '3d')
17
18 # Plot Surface Plot
19 ax3d.plot_surface(X, Y, Z, cmap=cm.coolwarm)
20 # Plot filled Contour Plot, on the Z-axis, offsetting by -1 to
21 # set it at the bottom of the Axes3D bounds
22 ax3d.contourf(X, Y, Z, zdir='z', offset=-1, cmap=cm.coolwarm)
23
24 plt.yticks(range(0, len(correlations.index)),
25            correlations.index, fontsize=8, rotation=90)
26 plt.xticks(range(0, len(correlations.columns)),
27            correlations.columns, fontsize=8, rotation=90)
28
29 plt.show()
```

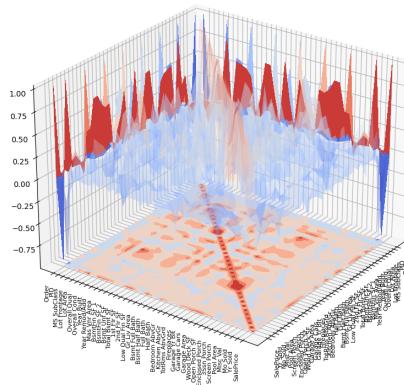
The `contour()` and `contourf()` functions accept a `zdir` argument. It defines the *direction* of the Z-axis. By setting it to `z` - we're plotting the Contour Plot on the bottom surface/spine. By default, it'll be plotted on the `Z=0`. Since our dataset's values range from `[-1..1]`, this is no good - the Contour Plot will be right in the very center, cutting into the Surface Plot. To alleviate this - we've set the `offset` to `-1`, plotting the Contour Plot just on the edge of the range. Depending on the range of your dataset, you'll set the offset accordingly as to not interfere with other elements:



We can see how the Surface Plot's vertices and their height is clearly reflected on the Contour Plot beneath it. The `-1` correlation between the *Order* and *SalePrice* features, which dips down way below all others almost touches the bottom of the 3D box, and we can see how the “temperature” is being reflected right beneath it.

We're not limited to plotting Contours just on the Z-axis - this we can do the same thing on the Y and X axes as well. With this specific Surface Plot, given the variance in intensity, it might be a bit hard to see them, so we'll adjust the `alpha` of the Surface Plot to largely make it transparent, making it easier to see the Contour Plots:

```
1 ax3d.plot_surface(X, Y, Z, cmap=cm.coolwarm, alpha=0.3)
2 ax3d.contourf(X, Y, Z, zdir='z', offset=-1, cmap=cm.coolwarm)
3 ax3d.contourf(X, Y, Z, zdir='x', offset=-1, cmap=cm.coolwarm)
4 ax3d.contourf(X, Y, Z, zdir='y', offset=-1, cmap=cm.coolwarm)
```



3D Line Plots and CP1919 Ridge Plot

We've recreated the *CP1919* pulsar data earlier in this chapter. The plot was done in 2D, but had a 3D effect, due to the fact that Ridge Plots are offset and overlap. This could've further been emphasized by adding a *perspective effect* to the plot by progressively shortening the X-axis and making lines progressively thicker down the line, to make it appear *closer* and *further*.

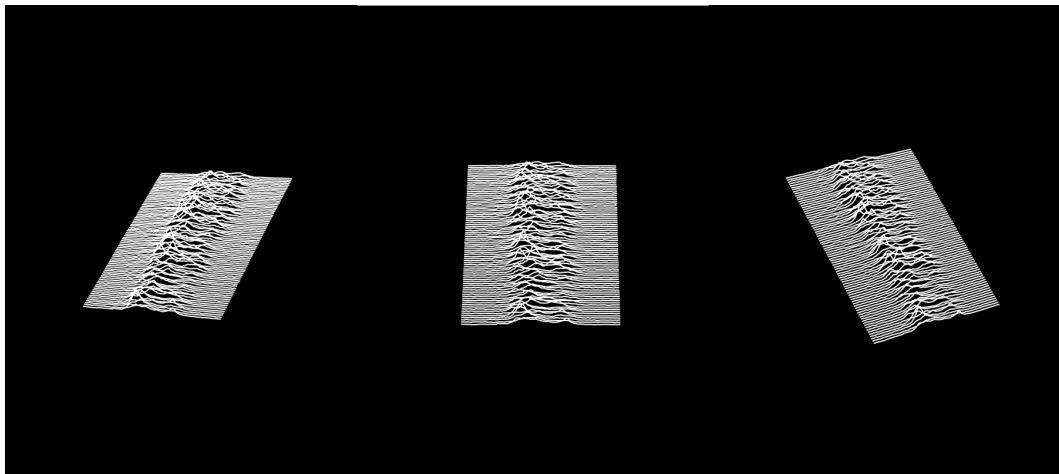
However, there's no better way to produce this effect than to actually plot it in 3D. When plotting this way, we won't have to create a bunch of `Axes` instances, though. There's simply no need - we can plot all lines on the same `Axes3D` instance, and offset them in space. It's worth noting that the Z-axis becomes the new amplitude axis - not the Y-axis. The Y-axis in this case would contain the `x` feature (time-series data on each pulse) and the X-axis will be a categorical feature - each line.

This makes plotting Ridge Plots much more concise. Other things we'll want to keep in mind is that we no longer have to fiddle around with the `hspace` parameter, and that setting the `facecolor` is no longer enough to get the desired look. The 3D axes will be *on top* of the face, just as usual, but turning off the splines on it won't make the 3D box transparent so we'll have a huge white box in the middle. We'll want to turn off the splines anyway, though.

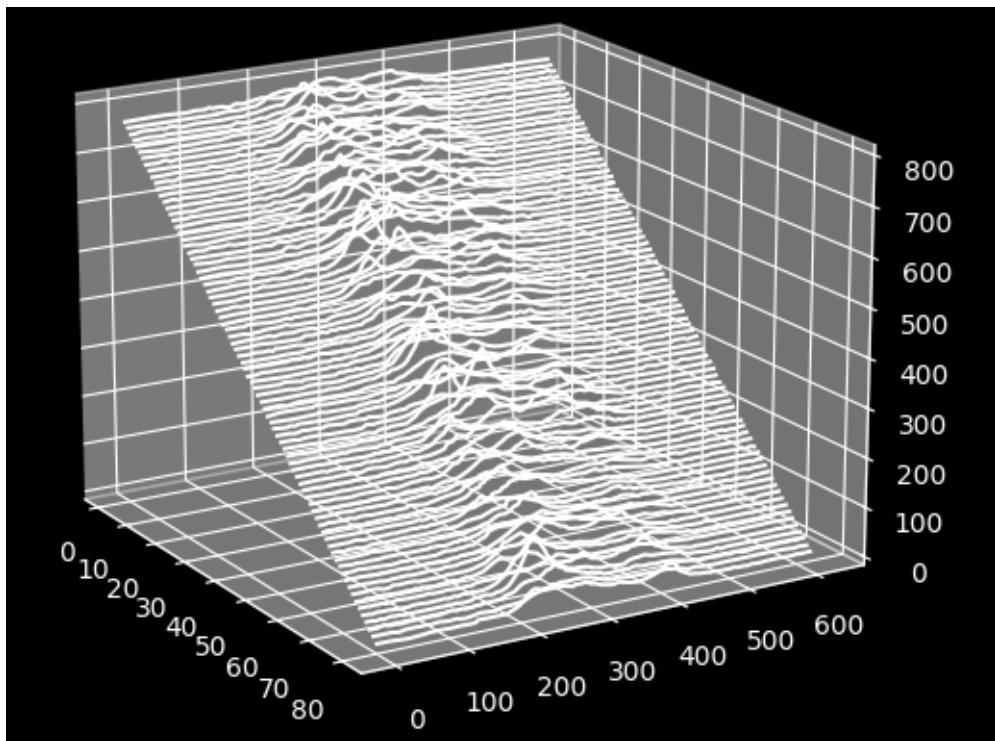
To combat this, we can use *Matplotlib Stylesheets*, which are covered in detail in the next chapter - *Chapter 7 - Advanced Matplotlib Customization*. For the time being - we'll set a 'dark_background' style via the plt instance, which sets the overall tone and style of the Figure:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from mpl_toolkits.mplot3d import Axes3D
4
5 df = pd.read_csv(r"https://raw.githubusercontent.com/StackAbuse/CP1919/master/data-raw/clean.csv")
6 groups = df.groupby(['line'])
7
8 plt.style.use('dark_background')
9
10 fig = plt.figure(figsize=(6, 8))
11
12 ax = fig.add_subplot(111, projection='3d')
13 ax.axis("off")
14
15 for group in groups:
16     ax.plot(group[1]['line'], group[1]['x'], group[1]['y'], color='white')
17
18 plt.show()
```

This results in a nice, interactive version of the earlier plot, which we can rotate to explore from different perspectives:



An interesting thing to note is that if we don't turn the axis off, a ghost-like spline can be seen, which gives a totally different feeling to the visualization:



We'll be exploring *Matplotlib Stylesheets* like this one in the next chapter.

Exploring EEG (Brainwave) Channel Data with Line Plots, Surface Plots and Spectrograms

Okay, let's have some fun now - we've covered all these different plot types, ranging from simple Line Plots to Surface Plots and Spectrograms. Believe it or not - we've got the knowledge required to do quite a lot at this point.

Importing Data

In this section - we'll be using several plot types to explore EEG data, provided to us by the [University of California, Irvine⁶⁰](https://archive.ics.uci.edu/ml/index.php). They host the *Machine Learning Repository*,

⁶⁰<https://archive.ics.uci.edu/ml/index.php>

which has *various* datasets collected from and for academic research. All of the datasets they host are public and can be accessed and downloaded by anyone with an internet connection. Certain datasets have a citation policy - so make sure to read the policy before publishing the findings found by exploring a dataset.

We'll be using the [*EEG Database Data Set*](#)⁶¹. It was formed during a large-scale study of 122 individuals, and the aim of the study was to examine EEG correlates of genetic predisposition to alcoholism.

The dataset is attributable to *Henri Begleiter* at the *Neurodynamics Laboratory* at the *State University of New York Health Center at Brooklyn*.

The individuals were separated into an *alcoholic group* and *control group*. To properly visualize data - we have to understand its domain. In this case, knowing the basics of EEG scanning will get us far - we can hardly work with an EEG dataset if we don't know what EEG scanning is, nor what the features are.

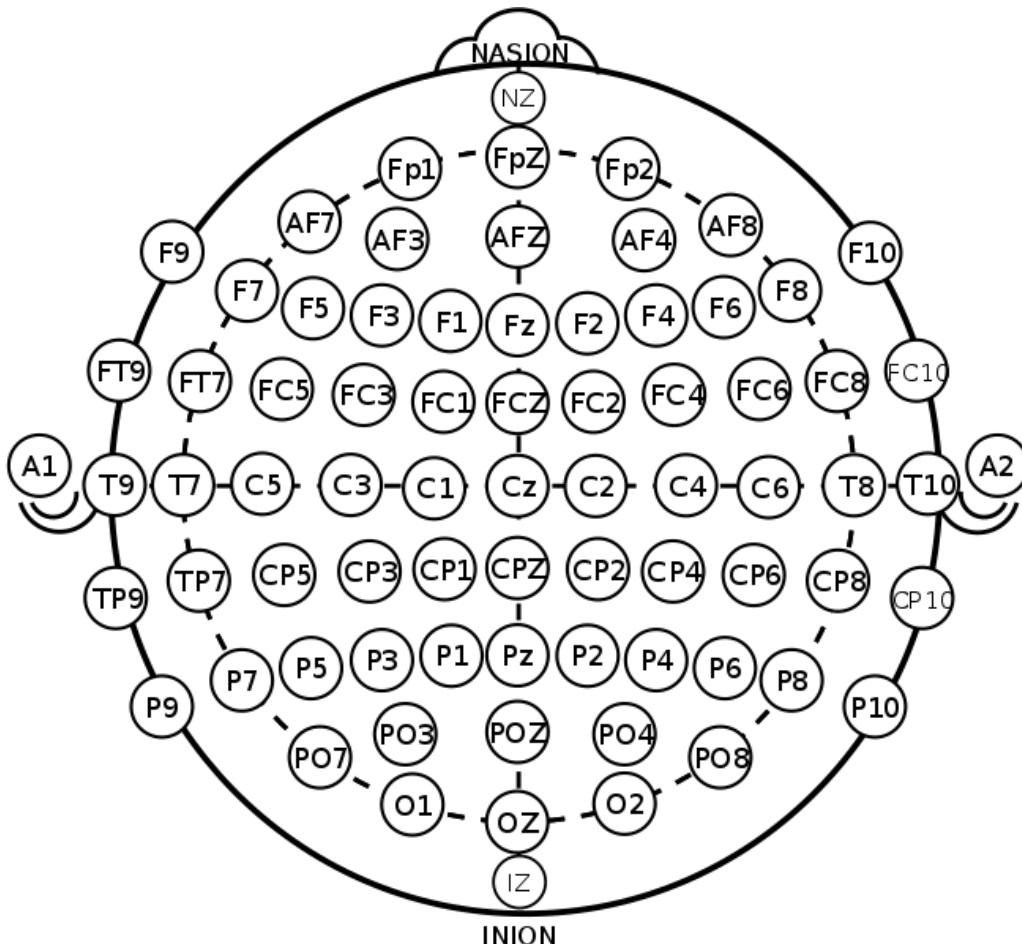
What is EEG?

Namely, ***Electroencephalography*** (EEG) is the process of recording an individual's brain activity - from a macroscopic scale. It's a non-invasive (external) procedure and collects aggregate, not individual data. This by all means *doesn't mean the procedure is of low quality or inaccurate*. What this means is that we see activation data of huge clumps of neurons, corresponding to a singular electrode placed in a certain area. To collect data on individual neurons, we'd need an invasive technology, which inserts channels directly into the brain, which are in physical contact with the neurons themselves. By using EEG and collecting data from a bunch of neurons that fire together - we've got a fairly effective way to correlate neuron activation to certain stimuli *without* having to perform invasive surgery on a patient.

These electrical signals are strong enough to actually travel through the skull, in a small range. By placing many electrodes on the human scalp - we can capture these signals. EEG allows us to also correlate these signals *through time*, which allows us to directly correlate certain stimuli with activations of groups of neurons.

⁶¹<https://archive.ics.uci.edu/ml/datasets/eeg+database>

There are various EEG electrode configurations - and the dataset we're working with has been generated by 64 electrodes:



Brylie Christopher Oxley, CC0, via Wikimedia Commons

Each subject in the trial was shown either one or two stimuli - named S_1 and S_2 . These stimuli were sampled from the [1980 Snodgrass and Vanderwart picture set](#)⁶². When two stimuli were shown - they could be matching or not ($S_1 = S_2$), and this feature was also written down in the dataset.

Note: Keep in mind - we're here for *Data Visualization*. Our task is to

⁶²<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.294.1979&rep=rep1&type=pdf>

visualize the signals provided in the dataset, and analyze it. Our task *isn't* to check the validity of the experiment's hypothesis. Whether there *is* or *isn't* genetic predisposition to alcoholism, and whether this experiment and dataset prove or disprove that predisposition *isn't* on us. It's on the research team performing the experiment.

Pre-Processing Data

That being said, let's download the dataset. A more readable version of it, which we can work with more easily is available on Kaggle - [EEG-Alcohol⁶³](#). It's worth noting that the dataset is *fairly large* - it contains almost 950 files, labeled DataX.csv in ascending order, through several folders.

Since it's meant to be used in Machine Learning projects, the dataset is already separated into two smaller datasets - SMNI_CMI_TEST and SMNI_CMI_TRAIN. Since these should, by definition, be functionally equivalent datasets, we'll work with only one of them to reduce the running time of our visualizations. Since this folder has a Train folder, alongside 468 DataX.csv files - we'll import them all and combine them into a single DataFrame, skipping over the Train folder. Each of these files contains a decent amount of data - 16451 rows each.

Once we've collected all of these files into a single DataFrame, we'll separate it into two distinct DataFrames. One DataFrame will be used to store the data for the *control group*, while the other DataFrame will be used to store the data on the *alcoholic group*:

```
1 import pandas as pd
2 import numpy as np
3 from matplotlib import pyplot as plt
4 import os
5
6 # List all of the filenames in the directory
7 filenames_list = os.listdir('datasets/SMNI_CMI_TRAIN/')
8 # Create an empty DataFrame to store each file's data in
9 df = pd.DataFrame({})
10
11 # Counter for convenience
12 i = 0
13 for filename in filenames_list:
14     if not filename == 'Train':
15         # Read each file into a DataFrame
16         temp_df = pd.read_csv('datasets/SMNI_CMI_TRAIN/' + filename)
17         # Append it to the main DataFrame
```

⁶³<https://www.kaggle.com/nnaier25/Alcoholics>

```

18         df = df.append(temp_df)
19         i += 1
20         print(f'Appended .csv file number {i}/{len(filenames_list)})')
21
22 # Drop redundant column
23 df = df.drop(['Unnamed: 0'], axis=1)
24
25 # Separate main DataFrame into the Alcoholic and Control groups
26 alcoholic_df = df[df['subject identifier'] == 'a']
27 control_df = df[df['subject identifier'] == 'c']
28
29 print(alcoholic_df)
30 print(control_df)

```

Note: This is a relatively expensive operation. There's a significant number of files here and the operation might very well take up to several minutes to run even on higher-end systems. If you'd like to make this project a bit easier on your CPU - you can pick any two DataX.csv files with opposing subject identifier features. Just make sure that the two DataFrames contain data on different subject identifiers, but do note that these are not *individuals*. Unfortunately, the Kaggle dataset doesn't provide a smaller version, though, the *original* dataset does contain a smaller version with just two individuals, one from each group (control and alcoholic).

The data obtained this way won't actually be representative of all the data (not individualized sets) and will contain just a small sample, so it can't be used to draw much conclusions - but the approach used for the smaller data sample is *the same* as using the entire dataset. Alternatively, you can create your own random two-individual sample by using, say, the *first 30* files and files *between 295 and 325* files which belong to the co2a0000364 (a group) and co2c0000340 (c group) individuals respectively, or any other *Control-Alcoholic* pair:

```

1 i = 0
2 for filename in filenames_list:
3     if i in range(0, 30) or i in range(295, 325):
4         if not filename == 'Train':
5             temp_df = pd.read_csv('datasets/SMNI_CMI_TRAIN/' + filename)
6             df = df.append(temp_df)
7             print(f'Appended .csv file number {i}/{len(filenames_list)})')
8     i += 1

```

We'll be using the entirety of the data available in the *SMNI_CMI_TRAIN* directory. Running this code results in:

```

1 Appended .csv file number 1/469
2 Appended .csv file number 2/469
3 ...
4 Appended .csv file number 468/469
5
6     trial number sensor position sample num ... channel name time
7 0             0       FP1        0 ... 0 co2a0000364 0.000000
8 1             0       FP1        1 ... 0 co2a0000364 0.003906
9 2             0       FP1        2 ... 0 co2a0000364 0.007812
10 3            0       FP1        3 ... 0 co2a0000364 0.011719
11 4            0       FP1        4 ... 0 co2a0000364 0.015625
12 ...
13 16379         12          Y      251 ... 63 co2a0000369 0.980469
14 16380         12          Y      252 ... 63 co2a0000369 0.984375
15 16381         12          Y      253 ... 63 co2a0000369 0.988281
16 16382         12          Y      254 ... 63 co2a0000369 0.992188
17 16383         12          Y      255 ... 63 co2a0000369 0.996094
18
19 [3850240 rows x 9 columns]
20     trial number sensor position sample num ... channel name time
21 0             0       FP1        0 ... 0 co2c0000337 0.000000
22 1             0       FP1        1 ... 0 co2c0000337 0.003906
23 2             0       FP1        2 ... 0 co2c0000337 0.007812
24 3             0       FP1        3 ... 0 co2c0000337 0.011719
25 4             0       FP1        4 ... 0 co2c0000337 0.015625
26 ...
27 16379         47          Y      251 ... 63 co2c0000345 0.980469
28 16380         47          Y      252 ... 63 co2c0000345 0.984375
29 16381         47          Y      253 ... 63 co2c0000345 0.988281
30 16382         47          Y      254 ... 63 co2c0000345 0.992188
31 16383         47          Y      255 ... 63 co2c0000345 0.996094
32
33 [3817472 rows x 9 columns]

```

Each row has 9 columns:

```

1 trial number,sensor position,sample num,sensor value,subject identifier,matching conditio\
2 n,channel,name,time

```

It's worth taking a moment to understand what these are columns denote:

- **trial number:** The number of the trial.
- **sensor position and channel:** Both of these features refer to the same thing - the position of the electrode on the scalp. The former is a numerical representation (1 .. 64), while the latter is a symbolic representation (AF1 .. Y).
- **sample num and time:** The number of the sample is the *time dimension*, where each sample is taken at a fixed time *after* the last (256 Hz). Alternatively, you can use the time measured in seconds.

- **sensor value**: The value recorded at a certain time for the given channel, in millivolts.
- **subject identifier**: The group of the subject - a for the *alcoholic group* and c for the *control group*.
- **matching condition**: The representation of what was shown to the individuals - S1 obj if a single object is shown, S2 match if two are shown and they're the same, S2 nomatch if two are shown and they're different.
- **name** - Name of the individual.

Now that we've successfully imported the data into a single DataFrame and then segregated it into the two groups we'll be working with - we'll want to change its shape from *long-form* to *wide-form*. Currently, *each measurement* is a different row, with 0..255 samples of sensor values for each channel/sensor position.

Let's go ahead and convert this data to a wide-form via Pandas' `pivot()` function, pivoting the data around the `sensor position` feature (categorical feature denoting the position of sensors, we can use `channel` here too), while setting the `sample num` as the columns (time dimension, we can use `time` here too) and the `sensor value` as the values.

This'll give us an overview of the `sensor values`, for *each sensor location/channel* individually, over *sample nums* (time):

```

1 alcoholic_df = pd.pivot_table(
2     alcoholic_df[['sensor position', 'sample num', 'sensor value']],
3     index='sensor position', columns='sample num', values='sensor value'
4 )
5
6 control_df = pd.pivot_table(
7     control_df[['sensor position', 'sample num', 'sensor value']],
8     index='sensor position', columns='sample num', values='sensor value'
9 )
10
11 print(alcoholic_df)
12
13 print('Alcoholic group min and max:\n', np.min(alcoholic_df.values), np.max(alcoholic_df.\nvalues))
14
15 print('Control group min and max:\n', np.min(control_df.values), np.max(control_df.values\
16 ))
```

Both of these DataFrames will be of the same shape (64×256) so let's print just one for brevity:

```

1 sample num      0      1      2      ...     253     254     255
2 sensor position
3 AF1           -0.045860 -0.064536 -0.056226 ... -0.249468 -0.095694 -0.025072
4 AF2           -0.260362 -0.177260 -0.067140 ... 0.028443  0.163481  0.329732
5 AF7           -0.135894 -0.144217 -0.158796 ... 1.312264  1.615651  1.981340
6 AF8           -0.417672 -0.378204 -0.259796 ... 1.595732  1.851289  2.090191
7 AFZ           -0.054826 -0.050677 -0.007043 ... 0.736826  0.846915  0.940417
8 ...
9 TP7           0.347438  0.112655 -0.070170 ... -4.558191 -4.574838 -4.639221
10 TP8          0.319043  0.181902  0.057230 ... -4.493115 -4.439128 -4.468204
11 X             0.097804  -0.328464 -0.575719 ... 6.625898  6.727698  6.617587
12 Y             0.356855  0.365145  0.304919 ... -3.466264 -3.281374 -3.347843
13 nd            0.175694  0.159072  0.173621 ... -0.622183 -0.771800 -0.788404
14
15 [64 rows x 256 columns]

```

This wide-form allows us to plot several types of plots. Namely, let's start out with Surface Plots which we can use to visualize the neuron (group) activations over time (*sample num*), with Contour Plots on the Z-axis to further help us get an idea of some trends, since the Surface Plots will have various intensities and will be relatively large.

Before plotting though, it's worth taking a look at the summary statistics of each DataFrame, such as the `min` and `max` values. If they're not in sync between the two DataFrames, the two surface plots will have different limits, making it hard to compare them visually. We could use the `describe()` function on the DataFrames, but we're not really interested in the summaries values other than the *sensor values*. Calling `describe()` on the pivoted DataFrames will return summary statistics for *each sample number*, which won't help us much since we'd have to go through them *again* to find the maximum and minimums there. To avoid that, we can use Numpy's `min()` and `max()` functions on the `DataFrame.values` of each DataFrame:

```

1 print('Alcoholic group min and max:\n', np.min(alcoholic_df.values), np.max(alcoholic_df.\nvalues))
2 print('Control group min and max:\n', np.min(control_df.values), np.max(control_df.values\
3 ))
4

```

These statements result in:

```
1 Alcoholic group min and max:  
2   -12.36796170212766 8.325544680851056  
3 Control group min and max:  
4   -10.598575107296146 12.822759656652364
```

The output proves why it's important to check statistics like this. The *Control Group* has a smaller *sensor value* range than the *Alcoholic Group*. It's in the range between (-10..12) while the *Alcoholic Group* is in the range of (-12..8). We'll want to set a *Z-axis limit* that encompasses both of these, and leaves a bit of room for breathing.

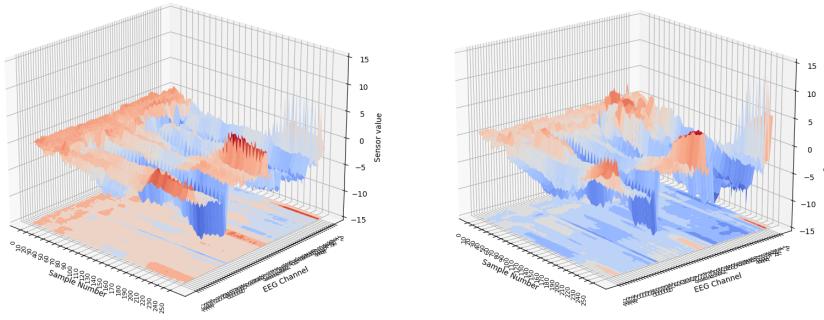
Plotting Surface Plots of EEG Channels

Since we've got two `DataFrames`, we'll put them both in a list to iterate over and plot in an `Axes3D` instance:

```
1 # Put both DataFrames in a list so we can iterate over them  
2 # and perform the same operations/visualizations without  
3 # having to repeat the code  
4 dataframes = [alcoholic_df, control_df]  
5 fig = plt.figure()  
6  
7 # Counter for DataFrames, to increase row-position in the `add_subplot()` function  
8 i = 0  
9 for dataframe in dataframes:  
10    # Add Axes3D instance for the respective group  
11    ax3d = fig.add_subplot(1,2,i+1, projection='3d')  
12    ax3d.set_zlim(-15, 15)  
13  
14    # Could be avoided if we were to use *channel* instead of *sensor position*  
15    # as these are already `0..63`  
16    X = range(0, len(dataframe.columns))  
17    Y = range(0, len(dataframe.index))  
18  
19    # Creating a meshgrid from the index and column  
20    X, Y = np.meshgrid(X, Y)  
21    # 2D array of values  
22    Z = dataframe.values  
23  
24    # Plot Surface Plot and set Labels  
25    ax3d.plot_surface(X, Y, Z, cmap=cm.coolwarm)  
26    ax3d.set_ylabel('EEG Channel')  
27    ax3d.set_xlabel('Sample Number')  
28    ax3d.set_zlabel('Sensor value')  
29  
30    # Plot filled Contour Plot on the Z-axis, offsetting to -15  
31    ax3d.contourf(X, Y, Z, zdir='z', offset=-15, cmap=cm.coolwarm)  
32  
33    # Set the Y-ticks with steps of 1 (setting tick frequency for channels)  
34    ax3d.set_yticks(np.arange(0, len(dataframe.index), 1))  
35    # Set the Y-tick labels (setting channel names as labels)  
36    ax3d.set_yticklabels(dataframe.index, rotation=90, fontsize=6)
```

```
37     # Set the X-ticks with a step of 10 (too many sample nums)
38     ax3d.set_xticks(np.arange(0, len(dataframe.columns), 10))
39     # Set the X-tick labels (changing the rotation and font size)
40     ax3d.set_xticklabels(ax3d.get_xticks(), rotation=90, fontsize=8)
41     i += 1
42
43 plt.show()
```

Running this code results in:



Though, we've made a cardinal mistake here - we haven't taken the *matching condition* into consideration at all. These are *integral* to the results of the experiment, since the stimuli should be the *cause* of certain neuron group activations. Let's rework the code so that we can provide a desired *matching condition*, which plots three surface plots for each group, for the three different *matching conditions*.

Plotting Surface Plots for Different Stimuli

Having 6 Surface Plots of this scale on a single Figure would be too much. We wouldn't be able to see much - so for each condition, we'll have a new Figure, containing two Surface Plots - one for each group. Additionally, there's another step we have to take during pre-processing. The dataset contains three types of matching conditions:

- S1 obj - One stimulus image is shown
- S2 nomatch - Two stimulus images are shown, and they're not identical
- S2 match - Two stimulus images are shown, and they're identical

However, the dataset isn't *clean*. Specifically, most likely during the process of porting the original dataset onto a version for *Kaggle*, some files contain a an "S2 nomatch," instead of S2 nomatch. The extra comma (,) caused the addition of quotation marks ("") to wrap the entry, since commas separate values in a CSV file. This is the only issue regarding the matching conditions in this dataset specifically, so we can easily fix that:

```
1 df = df.replace(to_replace='S2 nomatch,', value='S2 nomatch')
```

Since we'll be making three Figures and two Axes3D instances on each, we need to think about the Z-limits. If they're all over the place, it'll be hard to compare the plots. Visually, they might appear to be the same, and thus have the same heat (color-coded values), but if they're on different scales - this is misleading. If we were to leave each Axes3D to figure out the limits for itself, we'll have to suppress the *visual* data in lieu of the *numerical data* on the spines. This goes against the *point* of Data Visualization.

Naturally, the best thing to do is to match *all* of the instances to the same scale and limits. This way, we don't have to *think* about it, just compare the visuals produced by our code. Getting the minimum and maximum values of a DataFrame is easy:

```
1 DataFrame['column'].max()  
2 DataFrame['column'].min()
```

However, we're dealing with a couple of issues for this specific dataset. We're visualizing entire groups, not individuals. The aggregate data will differ. Some individuals in this trial had values going above 400 millivolts, which would bring up the scale so much that our Surface Plot looks similar to a flat plane or a 2D Heatmap.

Getting the min() and max() for each DataFrame we're plotting will then again, lead us back to the issue of having mixed limits. *Because* we're doing this for groups, not individuals, our hands are fairly tied when it comes to automating the minimum/maximum finding process. The best we can do in this situation is set a fixed limit.

We're visualizing *groups*, instead of *individuals* to try and explore whether there's a *generally different neural response* between two groups. Taking a look at just two individuals might not represent the entirety of the dataset. Again, the bigger the sample sizes, the smaller the impact of each outlier individual, due to the [Law of large numbers](#)⁶⁴. If we see a significant general difference between the groups, given the fact that the dataset is large enough, we can explore some individual cases and pick out *representative samples*.

Now - if we cut the *Control Group* in half, and plotted their results as two Surface Plots, would there be a difference? Most certainly so. Doesn't that discredit the difference between the Alcoholic and Control Groups? Most certainly not, if the difference is significant enough, and we have a large dataset.

As a sanity check, we'll revisit this concern again, a bit later on, to check if the potential differences between these groups are a fluke.

To get an idea of which Z-axis range/limit is suitable, let's define a list of conditions we'd like to explore and get an *Alcoholic DataFrame* and *Control DataFrame* for each of those conditions. Then, we'll print out the minimum and maximum of their *values*:

```

1 df = df.drop(['Unnamed: 0'], axis=1)
2 df = df.replace(to_replace='S2 nomatch,', value='S2 nomatch')
3
4 conditions = ['S1 obj', 'S2 nomatch', 'S2 match']
5
6 for condition in conditions:
7     alcoholic_df = df[(df['subject identifier'] == 'a') & (df['matching condition'] == condition)]
8
9     alcoholic_df = pd.pivot_table(
10         alcoholic_df[['sensor position', 'sample num', 'sensor value']],
11         index='sensor position', columns='sample num', values='sensor value'
12     )
13
14     control_df = df[(df['subject identifier'] == 'c') & (df['matching condition'] == condition)]
15     control_df = pd.pivot_table(
16         control_df[['sensor position', 'sample num', 'sensor value']],
17         index='sensor position', columns='sample num', values='sensor value'
18     )
19
20     print(condition + ' Alcoholic group min and max:\n', np.min(alcoholic_df.values), np.\max(alcoholic_df.values))
21     print(condition + ' Control group min and max:\n', np.min(control_df.values), np.min(\control_df.values))
22
23
24
25

```

⁶⁴https://en.wikipedia.org/wiki/Law_of_large_numbers

This results in:

```

1 S1 obj Alcoholic group min and max:
2   -9.525425000000002 14.776849999999992
3 S1 obj Control group min and max:
4   -11.08868749999997 -11.088687499999997
5
6 S2 nomatch Alcoholic group min and max:
7   -18.810906666666668 9.897586666666665
8 S2 nomatch Control group min and max:
9   -17.627878378378366 -17.627878378378366
10
11 S2 match Alcoholic group min and max:
12   -14.891837500000005 7.98706249999998
13 S2 match Control group min and max:
14   -21.071050632911394 -21.071050632911394

```

These range between -21 and 15. Allowing for some padding around that, setting the Z-limit between -25..25 should do the trick quite nicely. That being said, let's go ahead and plot a Figure for each *condition*, with an *Alcoholic Group* and *Control Group* on individual Axes3D:

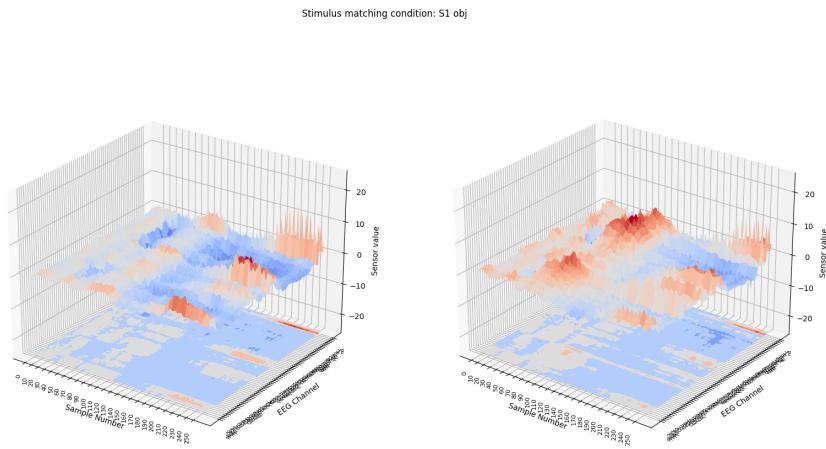
```

1 # Remove redundant feature and replace erroneous values
2 df = df.drop(['Unnamed: 0'], axis=1)
3 df = df.replace(to_replace='S2 nomatch', value='S2 nomatch')
4
5 # Specify conditions we want to explore
6 conditions = ['S1 obj', 'S2 nomatch', 'S2 match']
7
8 # Create a Figure with an Alcoholic and Control DataFrame for each condition
9 for condition in conditions:
10     alcoholic_df = df[(df['subject identifier'] == 'a') & (df['matching condition'] == condition)]
11     alcoholic_df = pd.pivot_table(
12         alcoholic_df[['sensor position', 'sample num', 'sensor value']],
13         index='sensor position', columns='sample num', values='sensor value'
14     )
15
16     control_df = df[(df['subject identifier'] == 'c') & (df['matching condition'] == condition)]
17     control_df = pd.pivot_table(
18         control_df[['sensor position', 'sample num', 'sensor value']],
19         index='sensor position', columns='sample num', values='sensor value'
20     )
21
22     # Put both DataFrames in a list so we can iterate over them
23     # and perform the same operations/visualizations without
24     # having to repeat the code
25     dataframes = [alcoholic_df, control_df]
26
27     fig = plt.figure()
28     # To help differentiate between the figures
29     fig.suptitle('Stimulus matching condition: ' + condition)
30
31
32

```

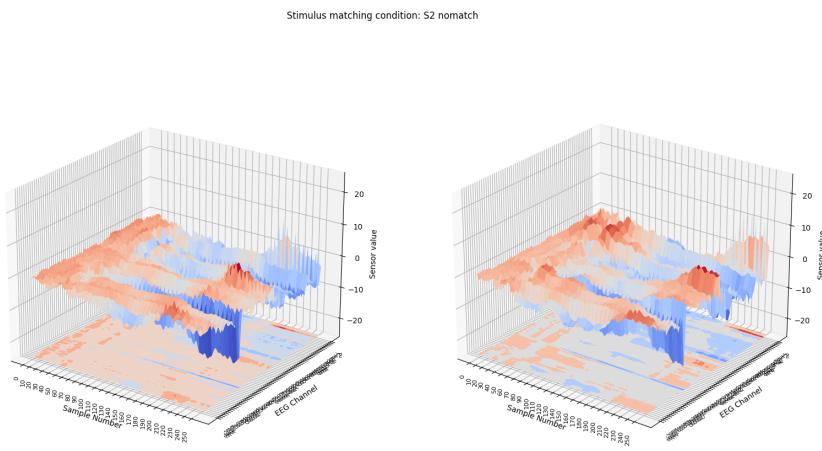
```
33 # DataFrame count, to increase row-position in `add_subplot()`
34 i = 0
35 for dataframe in dataframes:
36
37     # Add Axes3D instance for the respective group
38     ax3d = fig.add_subplot(1,2,i+1, projection='3d')
39     ax3d.set_zlim(-25, 25)
40
41     # Could be avoided if we were to use *channel* instead of
42     # *sensor position* as these are already `0..63`
43     X = range(0, len(dataframe.columns))
44     Y = range(0, len(dataframe.index))
45
46     # Creating a meshgrid from the index and column
47     X, Y = np.meshgrid(X, Y)
48     # 2D array of values
49     Z = dataframe.values
50
51     # Plot Surface Plot and set Labels
52     ax3d.plot_surface(X, Y, Z, cmap=cm.coolwarm)
53     ax3d.set_ylabel('EEG Channel')
54     ax3d.set_xlabel('Sample Number')
55     ax3d.set_zlabel('Sensor value')
56
57     # Plot filled Contour Plot on the Z-axis, offsetting to -10
58     ax3d.contourf(X, Y, Z, zdir='z', offset=-25, cmap=cm.coolwarm)
59
60     # Set the Y-ticks with steps of 1 (tick frequency for channels)
61     ax3d.set_yticks(np.arange(0, len(dataframe.index), 1))
62     # Set the Y-tick labels (setting channel names as labels)
63     ax3d.set_yticklabels(dataframe.index, rotation=90, fontsize=6)
64     # Set the X-ticks with a step of 10 (too many sample nums)
65     ax3d.set_xticks(np.arange(0, len(dataframe.columns), 10))
66     # Set the X-tick labels (changing the rotation and font size)
67     ax3d.set_xticklabels(ax3d.get_xticks(), rotation=90, fontsize=8)
68     i += 1
69
70 plt.show()
```

This results in three individual Figures with 2 plots each, showcasing the neural activity when faced with different stimuli. Starting with *S1 obj* matching condition, we've got the *Alcoholic Group* on the left, and the *Control Group* on the right:



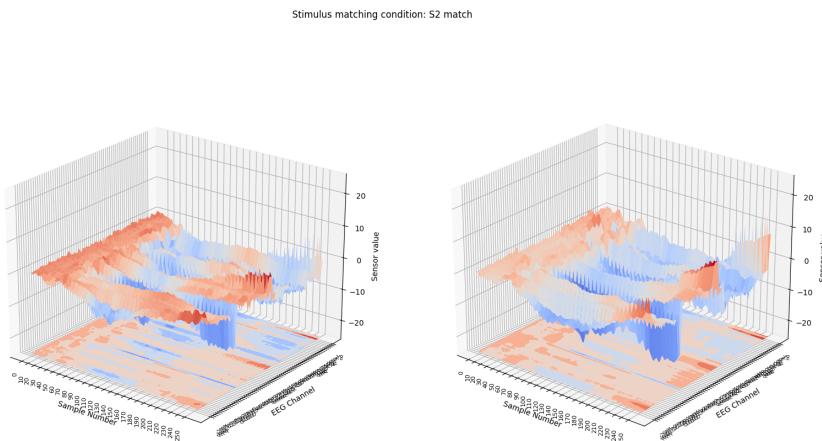
As evident from the Surface Plots here, the *Control Group* has a lot more activation in the channels around the *P* and *P0* channel groups (difficult to see from certain angles, rotating the plot helps). Taking a look back at the *channel locations* from the start of this section - we can map the activations to certain parts of the brain. There's a fairly significant difference in the activations between these two groups on average. We can see *some* common activations, such as the activations on the *AF channel group*, near the end on the X-axis (*sample num* feature), as well as some activations in the *FP channel group* and the *Y channel group* near the end.

Let's take a look at the other conditions as well, going forward with *S2 nomatch*:



The differences are not that stark here - while there are some, they don't pop out as much as on the previous condition. The main differences here can be noticed in the *P* and *P0* channel groups again, similar to last time. The *Control Group* has more activations in those areas than the *Alcoholic Group*.

And finally, let's explore the *S2 match* condition:



This time around, the activations in the *P and P0 channels groups* aren't too different, however, the activations in the *AF and FP channel groups* differ. It also appears that *all channels* have an initially higher reading in the early stages of the trials, as denoted by the more brightly red colored surface in near the beginning values of the *sample num* feature.

Plotting Neuron Group Activations for Individuals via Surface Plots

Let's try to limit ourselves to a much smaller dataset, and ditch the groups. We'll focus on two individuals, and plot their activations. The [original dataset⁶⁵](#) provided a *small, large* and *full* version. In the *small dataset version*, the team singled out two individuals from the groups as representative examples - `a_co2a0000364` , and `c_co2c0000337`. This saves us a bunch of trouble, since we'd have to search for representative samples ourselves if they haven't picked them out already. This would introduce the issue of *how do we identify representable samples?* Without prerequisite knowledge in the domain of the collected data, we'd be aiming fairly blindly.

Let's go ahead and filter out the full dataset to only focus on these two individuals, and plot their data side-by-side. The only thing we really have to change is the matching condition filter we did before. This time around, we'll be checking for either of the two names of these individuals. The original dataset had an `a_` and `c_` prefix for alcoholic and control individuals in their names. The *Kaggle version* doesn't, since these prefixes are essentially contained in the `subject identifier` feature:

⁶⁵<https://archive.ics.uci.edu/ml/datasets/eeg+database>

```

1 alcoholic_df = df[(df['name'] == 'co2a0000364')  

2                     & (df['matching condition'] == condition)]  

3 alcoholic_df = pd.pivot_table(  

4     alcoholic_df[['sensor position', 'sample num', 'sensor value']],  

5     index='sensor position', columns='sample num', values='sensor value'  

6 )  

7  

8 control_df = df[(df['name'] == 'co2c0000337')  

9                     & (df['matching condition'] == condition)]  

10 control_df = pd.pivot_table(  

11     control_df[['sensor position', 'sample num', 'sensor value']],  

12     index='sensor position', columns='sample num', values='sensor value'  

13 )

```

If you'd like to avoid going through all of these and appending them, just to filter out the vast majority of the values, since it can take some time on most local machines - you can single out the files in which the co2a0000364 and co2c0000337 are present, reducing the overall execution time.

Let's check the ranges of values again:

```

1 print(condition + ' Alcoholic group min and max:\n', np.min(alcoholic_df.values), np.max(\  

2 alcoholic_df.values))  

3 print(condition + ' Control group min and max:\n', np.min(control_df.values), np.min(cont\  

4 rol_df.values))

```

1 S1 obj Alcoholic group min and max:
2 -20.1425 53.299
3 S1 obj Control group min and max:
4 -10.927400000000002 -10.927400000000002
5
6 S2 nomatch Alcoholic group min and max:
7 -42.31879999999996 55.22250000000004
8 S2 nomatch Control group min and max:
9 -27.1444 -27.1444
10
11 S2 match Alcoholic group min and max:
12 -36.16229999999995 29.06789999999998
13 S2 match Control group min and max:
14 -28.614300000000004 -28.614300000000004

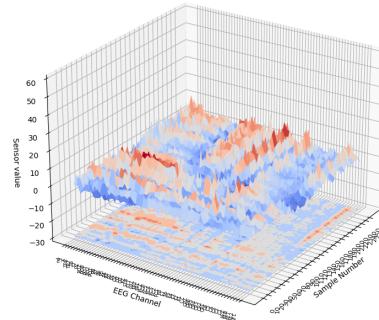
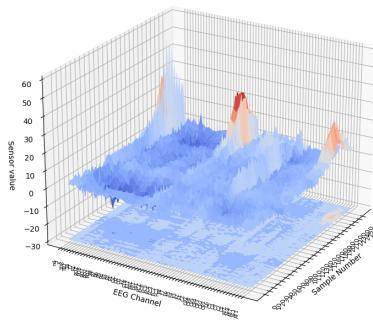
A -50..60 range would encompass everything here. Though, that's a range of 110 millivolts. Last time, we had a range of 50. Changing the scale this way will make it harder to interpret the results, as finer details will appear to be less impacting. The *lack* of activations here is a bit less important than the activations themselves. We can fairly safely take a chip from the lower-end here, and reduce the overall range for the sake of keeping the scale in a spot that we can interpret.

Let's adjust the Z-limit and the offset of the contour:

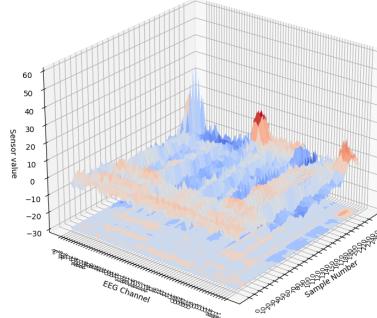
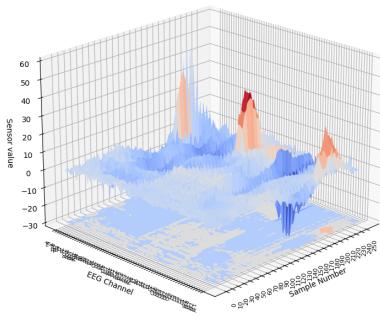
```
1 # ...
2 ax3d.set_zlim(-30, 60)
# ...
4 ax3d.contourf(X, Y, Z, zdir='z', offset=-30, cmap=cm.coolwarm)
```

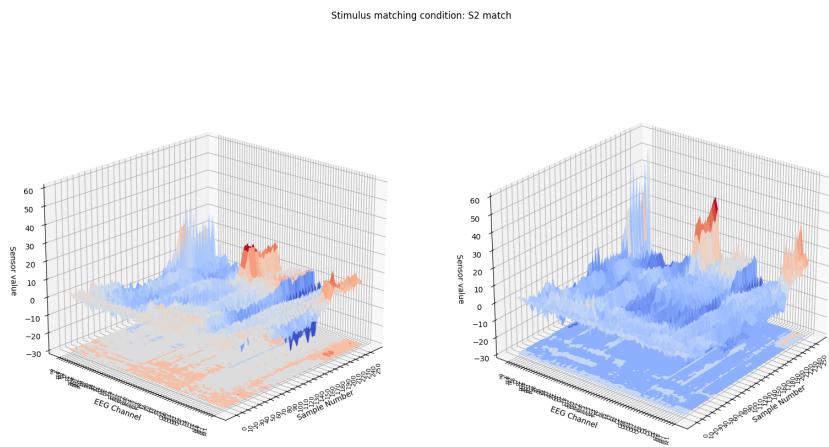
Now, we'll have two hand-picked individuals, and their activations for different conditions:

Stimulus matching condition: S1 obj

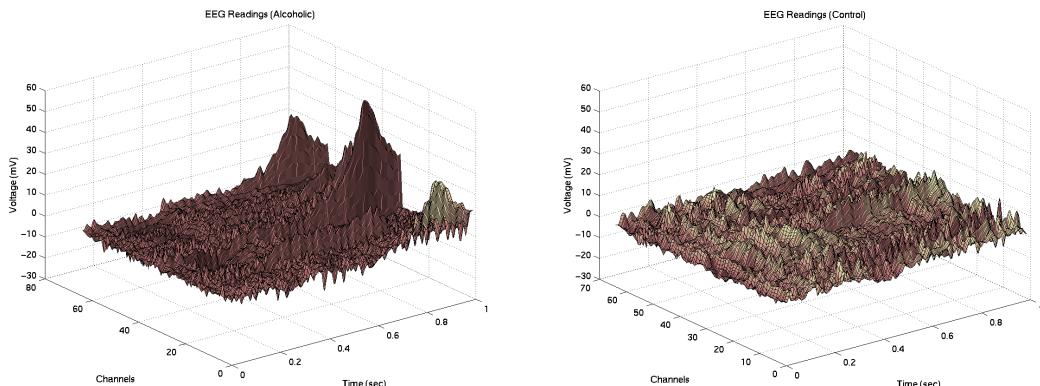


Stimulus matching condition: S2 nomatch





If we were to compare our own findings and visualizations to the exemplary visualizations on the [dataset page itself⁶⁶](#), we'll notice that the recreation is very faithful. The plots created there are courtesy of *Roger Gabriel*:



Surface Plots courtesy of Roger Gabriel

Plotting Channels with Line Plots and Spectrograms

Let's explore the dataset through a different lens. Even in the Surface Plots, each channel is constituted of a list of *sensor values*. The most basic, yet versatile plot

⁶⁶<https://archive.ics.uci.edu/ml/datasets/eeg+database>

we think of when we think of linear data are *Line Plots*. Let's (crudely) recreate the face of some of the EEG Viewer applications, which allow users to hook up an EEG scanner, whose channels are then plotted in real-time, allowing the user to check various information about each of those signals.

A great example is the *pbrain*⁶⁷ software, which uses Matplotlib, embedded into **GTK**⁶⁸. It's an analysis package for EEG and structural data, created by John Hunter for the University of Chicago. Micheal Castelle and Eli Albert, who works for the *Dr. Towle's Lab at the University of Chicago Hospital*, are now maintaining and expanding it.

It's a beautiful piece of software, which is incidentally also present in Matplotlib's gallery of projects:

There are various EEG viewers published online, and *pbrain* is one of them. We'll be aiming to recreate the inner, embedded plot of *pbrain*, though, don't mistake the simplicity of the plot as a sign of the simplicity of the software. We also won't be dealing with real-time data, but rather the dataset we've been using so far. Since we can't exactly plot a Spectrogram for *all of these* (we could, but it would defeat the purpose), we'll plot a Spectrogram for the *first* channel only. *pbrain* allows you to choose the channel you're plotting the Spectrogram for.

Let's load in the dataset for the first individual (first 30 files) first:

```

1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import os
4
5 filenames_list = os.listdir('datasets/SMNI_CMI_TRAIN/')
6 df = pd.DataFrame({})
7
8 i = 0
9 for filename in filenames_list:
10     if i in range(0, 30):
11         if not filename == 'Train':
12             temp_df = pd.read_csv('datasets/SMNI_CMI_TRAIN/' + filename)
13             df = df.append(temp_df)
14             print(f'Appended .csv file number {i}/{len(filenames_list)}')
15     i += 1
16
17 df = df.drop(['Unnamed: 0'], axis=1)
18 print(df.head())

```

Let's just check if the data is loaded properly first:

⁶⁷<https://github.com/nipy/pbrain>

⁶⁸<https://www.gtk.org>

```

1   trial number sensor position sample num ... channel name time
2   0           0       FP1        0 ...      0 co2a0000364 0.000000
3   1           0       FP1        1 ...      0 co2a0000364 0.003906
4   2           0       FP1        2 ...      0 co2a0000364 0.007812
5   3           0       FP1        3 ...      0 co2a0000364 0.011719
6   4           0       FP1        4 ...      0 co2a0000364 0.015625

```

Now, we'll group the data by each *channel* and *sensor position*, turning the corresponding *sensor values* into a *list*. This will collect all the values belonging to each *sensor position*, and turn them into a list that belongs to that row:

```

1 channels = df.groupby(['channel', 'sensor position'])['sensor value'].apply(list).reset_i\
2 ndex()
3
4 num_of_channels = len(channels.index)
5 print(channels)

```

The `apply()` function returns a Series or DataFrame - in our case, given several columns, it returns a DataFrame, where all sensor values are turned into a list, and assigned to the appropriate sensor position and channel. Once there, we reset the index, since there is now none. Running this piece of code results in:

```

1   channel sensor position          sensor value
2   0       0       FP1 [-8.921, -8.433, -2.574, 5.239, 11.587, 14.028...
3   1       1       FP2 [0.834, 3.276, 5.717, 7.67, 9.623, 9.623, 8.64...
4   2       2       F7  [-19.847, -12.522, 1.149, 14.821, 20.681, 17.2...
5   3       3       F8  [8.148, 1.801, -2.594, -4.547, -5.035, -5.524, ...
6   4       4       AF1 [-2.146, -2.146, -1.658, -0.682, 2.248, 5.178, ...
7   ..     ...
8   59      59       P2  [-2.421, -3.886, -4.862, -3.886, -1.933, -0.46...
9   60      60       P1  [-4.313, -5.29, -5.29, -4.313, -2.36, -0.407, ...
10  61      61       CPZ [-0.478, -0.966, -0.966, -0.966, -0.478, 0.01, ...
11  62      62       nd  [-8.901, -7.924, -3.042, 4.771, 11.607, 14.048...
12  63      63       Y   [-5.636, -2.706, 1.689, 5.595, 9.013, 10.478, ...

```

This turns our data into a format similar to the one we'll be visualizing. For each sensor position, we'll plot its sensor value list, and add it to an Axes. Now, all that's left is to actually visualize this:

```

1 # Specify height ratios, so the upper plot takes up 4/5 of the Figure
2 fig, ax = plt.subplots(2, 1, gridspec_kw={'height_ratios': [4, 1]})
3
4 # Plot each channel, and offset the next one by 50*channel index
5 # producing offset lines on the same Axes
6 for channel in channels.index:
7     ax[0].plot(np.array(channels['sensor value'].iloc[channel]).T + channel*50, linewidth\
8 =0.7)
9
10 # Set the Y-ticks in 50-tick steps to 50*num_of_channels to match the plotted lines
11 ax[0].set_yticks(np.arange(0, num_of_channels*50, 50))
12 # Set tick labels for each Y-tick
13 ax[0].set_yticklabels(channels['sensor position'], fontsize=8)
14 # Set the limit to remove padding, any list's length works here
15 ax[0].set_xlim(0, len(channels['sensor value'].iloc[0]))
16
17 # Plot spectrogram and add title for the user
18 ax[1].specgram(channels['sensor value'].iloc[0])
19 ax[1].set_title('Spectrogram for sensor: ' + channels['sensor position'].iloc[0])
20
21 plt.show()

```

We've performed a couple of new things here - first off, we've specified the `gridspec_kw`. We can set `width_ratios` and `height_ratios` if we'd like to add two Axes of different size to a Figure. Here, since we want the Line Plots to take up most of the space - we've set this to be 4:1.

Plotting 64 `Axes instances` here would get messy, real quick. It would raise questions about how to set the Y-tick labels, how to handle spines, without removing everything, and how we can add a Spectrogram at the bottom. To avoid this - we'll do all the plotting on a *single Axes* and have a separate one for the Spectrogram.

Naturally, we can call the `plot()` function many times on an `Axes`, but this would result in many overlapping Line Plots. We need to find a way to *offset them* - on each `plot()`, we'll offset the next one by some number. This can be achieved through Numpy's *Array Broadcasting* feature.

Namely, to showcase it on a smaller example before we execute the previous code:

```

1 array = np.array([1, 1, 1, 1, 1])
2 print(array.T)

```

This results in the *transposed array*:

```
1 [1 1 1 1 1]
```

Why is this useful? Because using *broadcasting*, we can increase every individual element by a constant here:

```
1 offset_array = array.T + 10
2 print(offset_array)
```

```
1 [11 11 11 11 11]
```

If we exchange the constant with a dynamic range such as:

```
1 offset_array = array.T + 10*np.arange(0, 5, 1)
2 print(offset_array)
```

Then our `offset_array` will contain much different values:

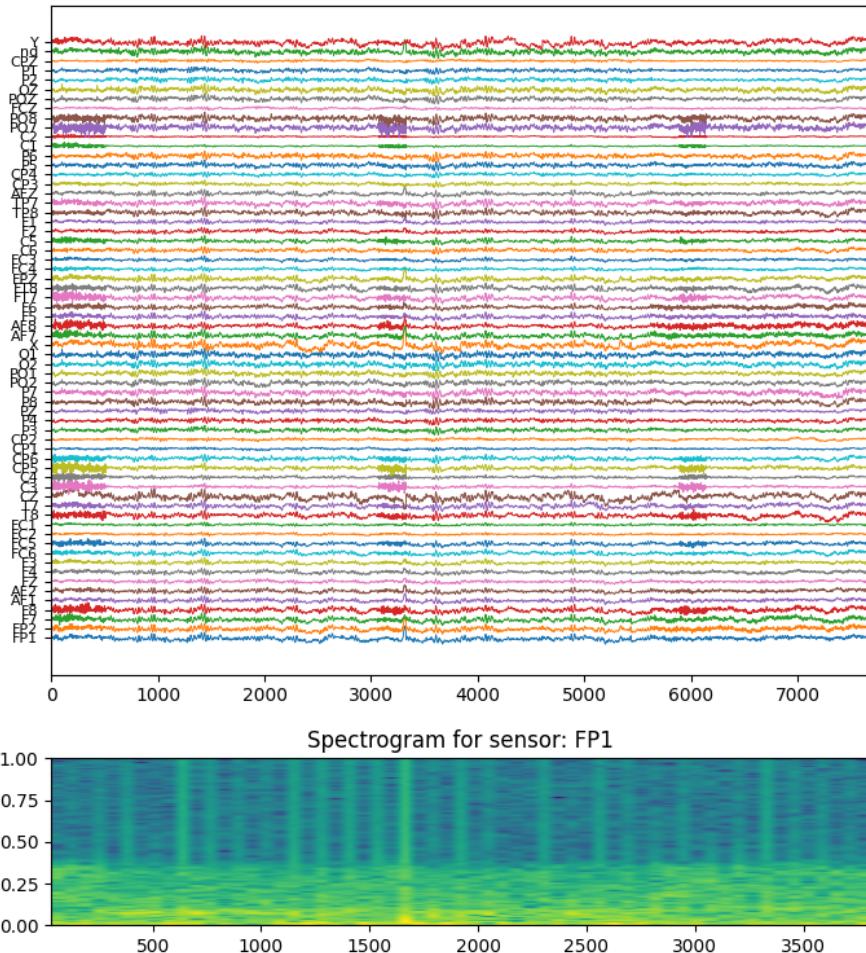
```
1 [ 1 11 21 31 41]
```

This is *perfect* for artificially offsetting the line plots. We'll increase/decrease each value of the Y-axis by some constant value, preserving the line paths, but offsetting them on the Y-axis. For this, we've chosen to offset the transposed values by the channel (0..63) multiplied by 50. This will typically be a trial-and-error phase where you test out different values and settle for one that works well for you.

To be able to transpose these values we've wrapped the result of `iloc[]` into an `np.array()`. In effect, this changes our regular channel value lists to lists with offset values:

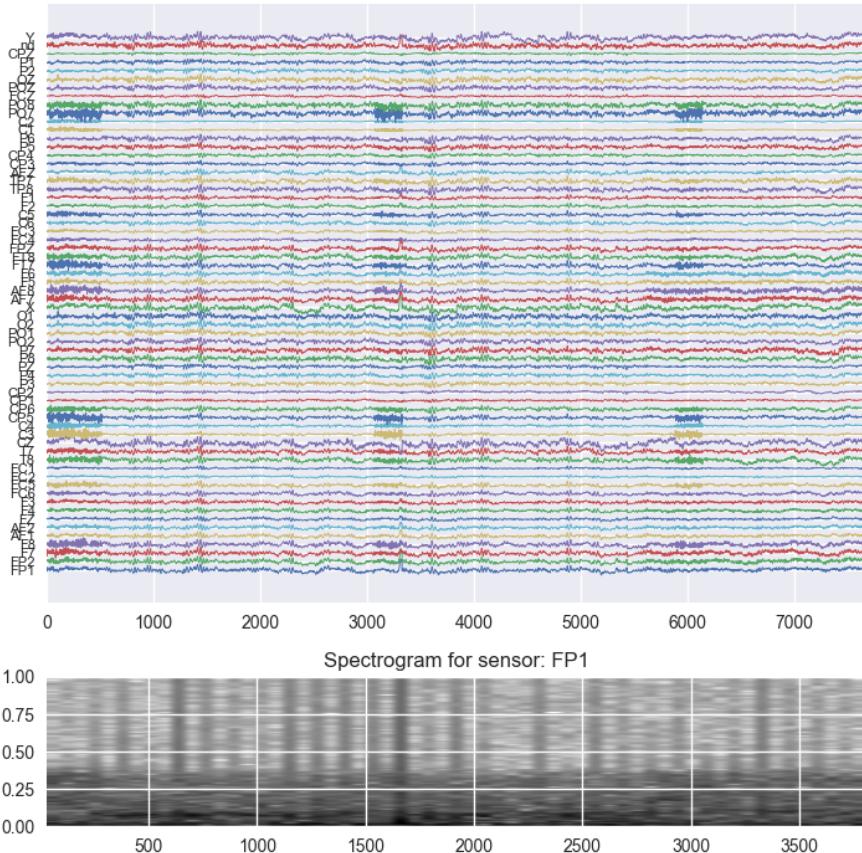
```
1 ax[0].plot(np.array(channels['sensor value'].iloc[channel]).T + channel*50)
```

Once we run this piece of code, we'll be greeted with:



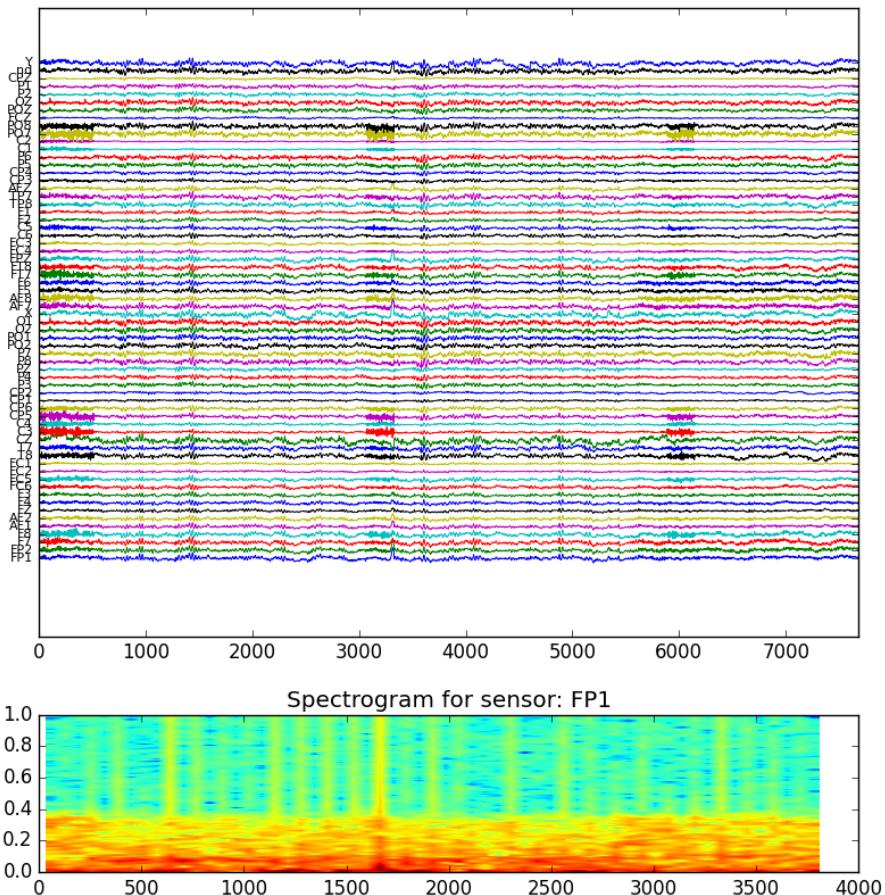
We could also try this plot out with different styles, just to check how it'd look like - *Stylesheets* are covered in the next chapter:

```
1 plt.style.use('seaborn')
```



Though, it looks like *pbrain* is using a more *classic* look of Matplotlib:

```
1 plt.style.use('classic')
2 plt.rcParams['figure.facecolor'] = 'grey'
```



Chapter 7. - Advanced Matplotlib Customization

In *Chapter 5*, we've taken a look at some of the basic customization we'll be doing fairly commonly when doing Data Visualization. These included the basic operations of changing figure sizes, tick frequency and scales. These features have gotten us to a point, though, even in the last chapter - these weren't enough for some of the plots we've been producing.

When recreating the *Joy Division* album cover, using a Ridge Plot in 3D - we couldn't just set the background to black. When creating a Joint Plot, simply adding multiple `Axes` instances looked bad so we opted to use a `GridSpec` instead. We've also used `Colormaps` in several Surface Plot examples.

We haven't fully delved into these or explained how they work - they were covered on a need-to-know basis. Now, this is the chapter in which we'll be exploring these functionalities in detail.

Understanding Matplotlib Stylesheets

First off, a relatively easy, but still really nice feature - *Matplotlib Stylesheets*. A stylesheet contains a set of parameters that change the look of Matplotlib's elements. These were hand-crafted by the Matplotlib team and are designed to have colors and palettes that work together. The `default` stylesheet doesn't look bad - but it could most certainly look better, slicker.

Each Matplotlib stylesheet is really just a file with a bunch of parameters tuned, just like a CSS file. These include various elements such as `axes.spines.linewidth`, `grid.linestyle`, `xtick.labelsize`, `text.color`, etc. These are all elements you can manually set through *setter* functions or by changing the *Runtime Configuration* (`rc`) parameters:

```
1 lines.linestyle: --
2 xtick.labelsize: 10
3 text.color: red
```

All of these settings reside in `style_name.mplstyle` files, and you can even create your own, and use them. If we were to save these parameters in a file such as `my_style.mplstyle`, we could import it and style our plots with it. As a matter of fact - let's save these in a file that we can import later on. We've mainly been working with the default stylesheet so far. Changing between stylesheets is as easy as using a single function, so we don't have to style each element manually, even though we can.

Let's go ahead and print out the available stylesheets from the `plt.style` module:

```
1 import matplotlib.pyplot as plt
2
3 for style in plt.style.available:
4     print(style)
```

This results in:

```
1 Solarize_Light2
2 _classic_test_patch
3 bmh
4 classic
5 dark_background
6 fast
7 fivethirtyeight
8 ggplot
9 grayscale
10 seaborn
11 seaborn-bright
12 seaborn-colorblind
13 seaborn-dark
14 seaborn-dark-palette
15 seaborn-darkgrid
16 seaborn-deep
17 seaborn-muted
18 seaborn-notebook
19 seaborn-paper
20 seaborn-pastel
21 seaborn-poster
22 seaborn-talk
23 seaborn-ticks
24 seaborn-white
25 seaborn-whitegrid
26 tableau-colorblind10
```

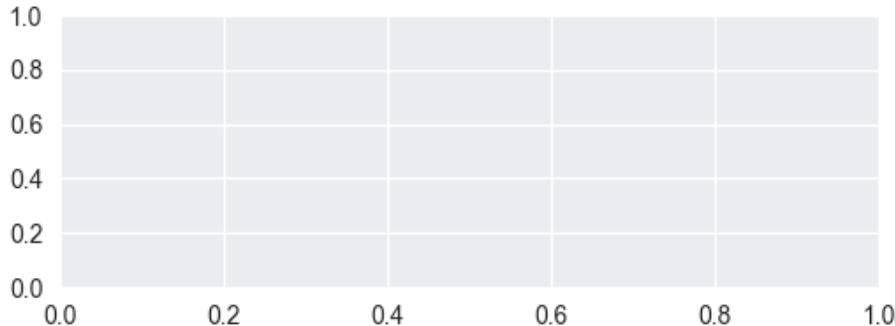
We've seen the `default` and `dark_background` styles so far - though, there's a bunch more. The `classic` stylesheet refers to an older version of Matplotlib, which was

popular throughout the ages, so the old design still stands out. *A lot* of these are inspired by Seaborn, another Data Visualization library that's built *on top* of Matplotlib, providing convenience functions and nicer styling. One of the key takeaways of Seaborn is how it makes complex function calls and customization easy by providing wrappers.

Instead of manually creating a Joint Plot, you can call the `jointplot()` function which takes care of the `GridSpec` for you. Let's take a look at some of the older visualizations we did, and take them through a few stylesheets. You can set the style on the `Figure` level via the `plt` instance - so you can't have two `Axes` instances on a `Figure` with differing styles.

Another takeaway is how it makes plotting, well, *nicer*. The more slick and well-balanced color palettes, mixed with softer contrasts make Seaborn plots simply look more appealing. Just changing the `default` style to `seaborn` can oftentimes improve the visual representation of a plot:

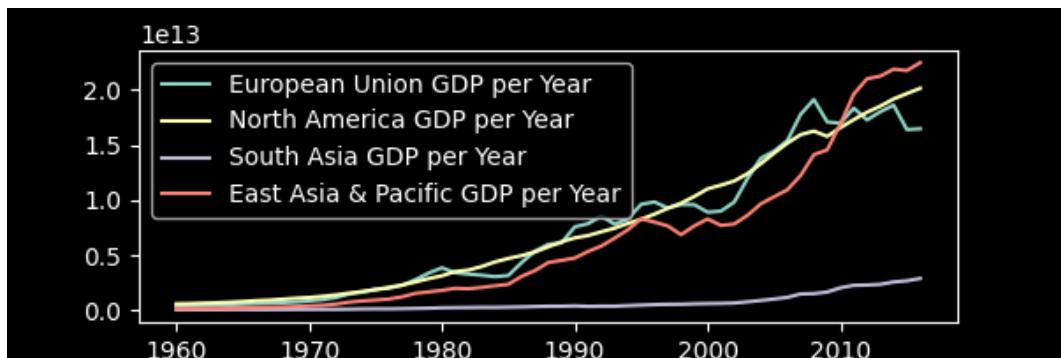
```
1 import pandas as pd
2 from matplotlib import pyplot as plt
3 plt.style.use('seaborn')
4 fig, ax = plt.subplots(figsize=(6, 2))
5 plt.show()
```



Now, let's re-do one of the older datasets and change the style:

```
1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('gdp_csv.csv')
5
6 plt.style.use('dark_background')
7
8 df_eu = df.loc[df['Country Name'] == 'European Union']
9 df_na = df.loc[df['Country Name'] == 'North America']
10 df_sa = df.loc[df['Country Name'] == 'South Asia']
11 df_ea = df.loc[df['Country Name'] == 'East Asia & Pacific']
12
13 fig, ax = plt.subplots(figsize=(6, 2))
14 ax.plot(df_eu['Year'], df_eu['Value'], label = 'European Union GDP per Year')
15 ax.plot(df_na['Year'], df_na['Value'], label = 'North America GDP per Year')
16 ax.plot(df_sa['Year'], df_sa['Value'], label = 'South Asia GDP per Year')
17 ax.plot(df_ea['Year'], df_ea['Value'], label = 'East Asia & Pacific GDP per Year')
18
19 ax.legend()
20 plt.show()
```

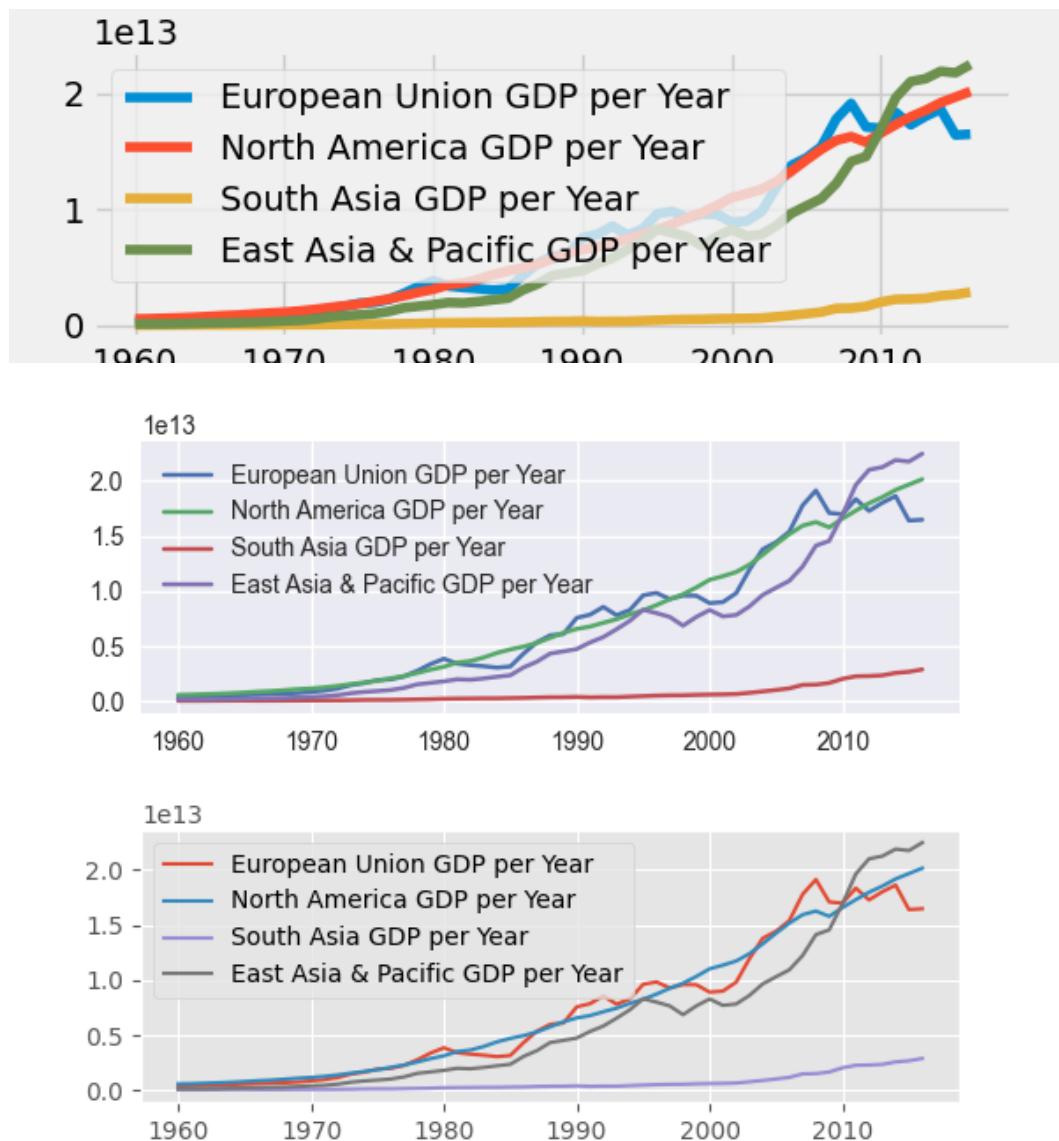
This results in:



Or, if we were to change the style to any of these:

```
1 plt.style.use('fivethirtyeight')
2 plt.style.use('seaborn')
3 plt.style.use('ggplot')
```

The results would be:



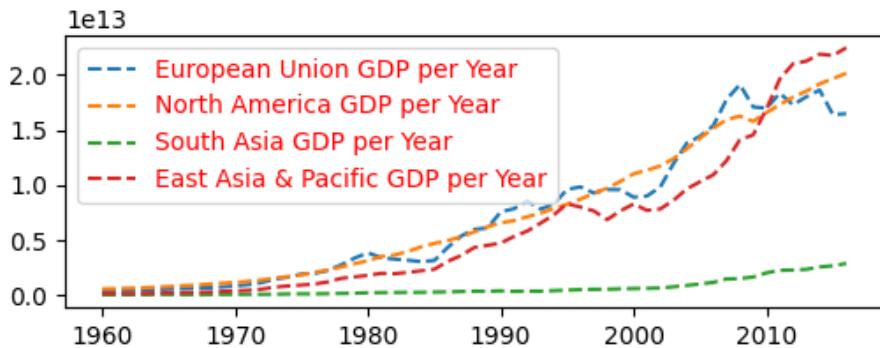
Or, we could use our own `my_style.mplstyle` file here as well:

```

1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('gdp_csv.csv')
5 # The .mplstyle file is located in the project's directory
6 plt.style.use('my_style.mplstyle')
7
8 # Pre-process data...
9 fig, ax = plt.subplots(figsize=(6, 2))
# Plot data...
10
11 ax.legend()
12 plt.show()
13

```

Which applies our own stylesheet to the created plot:



Matplotlib Runtime Configuration (rc) Parameters

We've mentioned *Runtime Configuration Parameters* (rc params) in the previous section. These are the parameters that control various aspects of Matplotlib plots and Figures. These include configurations such as *line width*, *face colors*, *edge colors*, *line styles*, etc. What happens when you use any sort of customization *setter* function, under the hood, is the changing of runtime configuration parameters.

You can access them via the `plt` instance:

```
1 plt.rc('group', **kwargs)
```

Or, via:

```
1 plt.rcParams['group.param'] = value
```

If you'd like to return them to the default values, reseting them is as easy as:

```
1 plt.rcdefaults()
2 # OR
3 plt.style.use('default')
```

Changing the parameters through code like this allows you to customize some specific parts of plots, and to essentially define your own *stylesheet*, without having to create a file. What stylesheets ultimately do is just set the *Runtime Configuration Parameters* themselves anyway.

Let's take a look at all the available keys we can play around with:

```
1 print('Number of keys: ', len(plt.rcParams.keys()))
2 print(plt.rcParams.keys())
```

This results in 310 keys, as well as their default values:

```
1 Number of keys: 310
2 KeysView(RcParams({'_internal.classic_mode': False,
3     'axes.axisbelow': 'line',
4     'axes.edgecolor': 'black',
5     'axes.grid': False,
6     'axes.grid.axis': 'both',
7     'axes.grid.which': 'major',
8     'axes.labelcolor': 'black',
9     'axes.labelpad': 4.0,
10    '# ...
```

Let's set a few runtime configuration parameters on the fly, in the code itself:

```
1 import pandas as pd
2 from matplotlib import pyplot as plt
3 df = pd.read_csv('gdp_csv.csv')
4
5 # Setting them directly
6 plt.rcParams['axes.linewidth'] = 2
7 plt.rcParams['axes.grid'] = True
8
9 # Multiple settings for group, such as 'lines'
10 plt.rc('lines', linestyle='--', color='red')
11
12 # Making a dictionary of settings
13 settings = {
14     'color':'g'
15 }
16 # Unpacking dictionary into multiple settings for 'text' group
17 plt.rc('text', **settings)
18
19 df_eu = df.loc[df['Country Name'] == 'European Union']
20 fig, ax = plt.subplots(figsize=(6, 2))
21 ax.plot(df_eu['Year'], df_eu['Value'], label = 'European Union GDP per Year')
22
23 ax.legend()
24 plt.show()
```

If you're setting just one parameter, the first approach is probably the easiest. Though, if you have many of them that you'd like to customize - you'll likely prefer to containerize the settings into shared groups and use `plt.rc()` with several arguments. Though, if this leads to a bunch of customization code, you might as well just transfer all of the settings into an `.mplstyle` file and read it from there via `plt.style.use()`.

Running this code results in:



Understanding Matplotlib Colors and Colormaps

We've been setting colors via various arguments on our plots so far. Typically, these will be something along the lines of `c`, `color` or `colors`, where many plots accept a list of colors, creating a color palette for that specific plot.

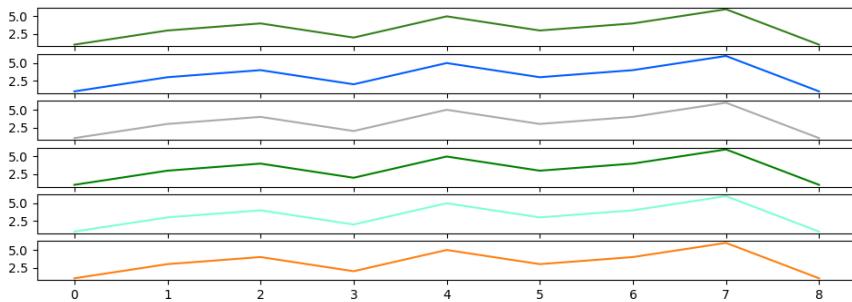
In the section on 3D visualizations and Surface Plots, we also used the `cm` module to take advantage of a few colormaps.

The `matplotlib.colors` module is the module that takes care of mapping numbers and color arguments to *RGB* and *RGBA* values. The `colors` API accepts various formats, and converts them into the underlying RGB values:

- **RGB tuples** - `(0.0, 0.0, 0.0)`, denoting the RGB values
- **Hex strings** - `#49CE3D`, `#005dff`, etc...
- **String representations of float values** - `0` for black, `1` for white, and `0. [0-99]` for values in-between.
- **Character representations of colors** - `r`, `g`, `b`, `c`, `m`, `y`, `k`, `w`.
- **X11/CSS names**: `burlywood`, `bisque`, `black`, `darkcyan`, etc...
- **T10/Tableau colors**: `tab:blue`, `tab:orange`, `tab:gray`, etc...

We've mainly been using *hex strings*, *character representations of colors*, as well as the *X11/CSS names*:

```
1 import matplotlib.pyplot as plt
2
3 fig, ax = plt.subplots(6, 1)
4 X = [1, 3, 4, 2, 5, 3, 4, 6, 1]
5
6 # Tuple RGB representation
7 ax[0].plot(X, color=(0.2, 0.5, 0.1))
8 # Hex String representation
9 ax[1].plot(X, color="#005dff")
10 # String-float representation
11 ax[2].plot(X, color='0.67')
12 # Character representation
13 ax[3].plot(X, color='g')
14 # X11/CSS names
15 ax[4].plot(X, color='aquamarine')
16 # T10/Tableau colors
17 ax[5].plot(X, color='tab:orange')
18
19 plt.show()
```



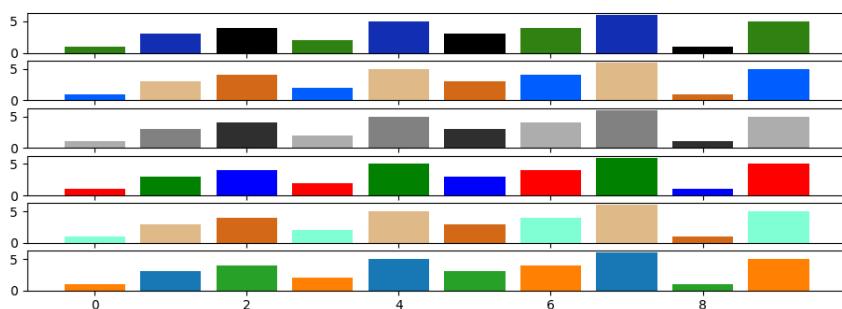
Or, if the plot type allows for multiple values:

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 fig, ax = plt.subplots(6, 1)
5
6 X = range(0, 10)
7 Y = [1, 3, 4, 2, 5, 3, 4, 6, 1, 5]
8
9 ax[0].bar(X, Y, color=[(0.2, 0.5, 0.1), (0.1, 0.2, 0.7), (0, 0, 0)])
10 ax[1].bar(X, Y, color=['#005DFF', '#DEB887', '#D2691E'])
11 ax[2].bar(X, Y, color=['0.67', '0.5', '0.2'])
12 ax[3].bar(X, Y, color=['r', 'g', 'b'])
13 ax[4].bar(X, Y, color=['aquamarine', 'burlywood', 'chocolate'])
14 ax[5].bar(X, Y, color=['tab:orange', 'tab:blue', 'tab:green'])
15
16 plt.show()

```

A *list* of colors (colormap) can be supplied. Once the list reaches the end, if there are more values - it'll just repeat:



Colormaps

Additionally, the `cm` module is simply a container for some of the built-in colormaps, so that we don't have to define our own. For example, the `coolwarm` colormap is a common one for Heatmaps and Surface Plots. Choosing a colormap isn't done at random - it depends on the dataset you're visualizing.

If you're visualizing a Heatmap/Surface Plot, naturally, you'll choose `coolwarm` or a similar colormap, due to the nature of the plot, where higher values are assigned warmer colors. For the most part, you'll be using *perceptually uniform* colormaps.

In perceptually uniform colormaps, steps in data are equally reflected as steps in color.

The *lightness* of a color is perceptually correlated with the underlying data. On the other hand, the *hue* is much less so. Because of this, most perceptually uniform colormaps rely mainly on the increase of the lightness, and less so on other parameters.

There are four types of colormaps:

- Sequential
- Diverging
- Cyclic
- Qualitative

Sequential colormaps are typically used for ordered data, that follows a range - 0 .. N. They linearly/sequentially increase the lightness and/or hue.

Diverging colormaps are typically used for data revolving around a certain value - such as a zero or peak. They contain two colors/hues that are intense on the ends, but reach an unsaturated point in the middle, that's shared. The `coolwarm` colormap is a *diverging colormap*, since on one end, we've got hues of *blue*, on the other end, we've got hues of *red*, and in the center - we've got an unsaturated grey.

Cyclic colormaps start and end with the same color, allowing us to repeat them over multiple values seamlessly when confronted with cyclic data, such as days.

Qualitative colormaps are used for unordered, qualitative data, where the changes in data don't reflect an equal change in the colors.

Sequential, *diverging* and *cyclic* colormaps can, but not always, be *perceptually uniform*, depending on the specific colormap itself. *Qualitative* colormaps aren't meant to be perceptually uniform, since the data they're being used on is typically hectic and doesn't have order.

It might be a bit difficult to visualize some of this in your mind, especially if you haven't dabbled much with *color theory* before. The easiest way to digest this information is to visualize them:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 sequential_colormaps = ['Greys', 'Blues', 'Reds', 'Greens', 'GnBu', 'PuBu']
5 diverging_colormaps = ['PiYG', 'PuOr', 'RdGy', 'coolwarm', 'seismic', 'Spectral']
6 cyclic_colormaps = ['twilight', 'hsv']
7 qualitative_colormaps = ['Pastel1', 'Pastel2', 'tab10', 'tab20', 'Paired']
8
9 colormaps = [sequential_colormaps,
10              diverging_colormaps,
11              cyclic_colormaps,
12              qualitative_colormaps]
```

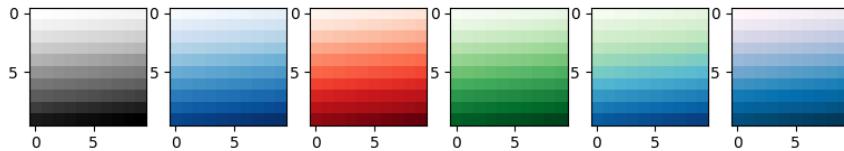
Here, we've made a few lists for different types of colormaps, and packed them all into one list that we'll iterate over. These are *not all* of the colormaps that Matplotlib has to offer⁶⁹. Now, let's create a 10x10 set of values, increasing linearly:

```

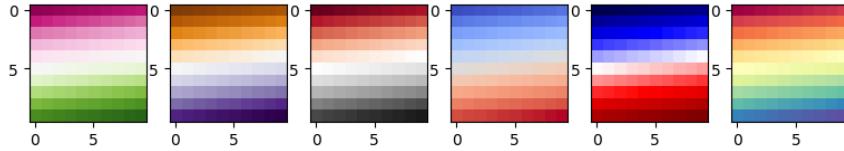
1 # Set the range from 0 to 50
2 # Create 100 values in this range
3 # Reshape it into a 10x10 matrix
4 X = np.linspace(0, 50, 100).reshape((10, 10))
5
6 for colormap_list in colormaps:
7     fig = plt.figure()
8     i = 0
9     for colormap_name in colormap_list:
10         ax = fig.add_subplot(1, len(colormap_list), i+1)
11         ax.imshow(X, cmap = colormap_name)
12         i += 1
13
14 plt.show()
```

For each colormap type, we've created a new `Figure`, in which we've got several `Axes` - one for each colormap in this list. First up, we've got the *sequential* colormaps, which just sequentially increase:

⁶⁹https://matplotlib.org/stable/gallery/color/color_map_reference.html



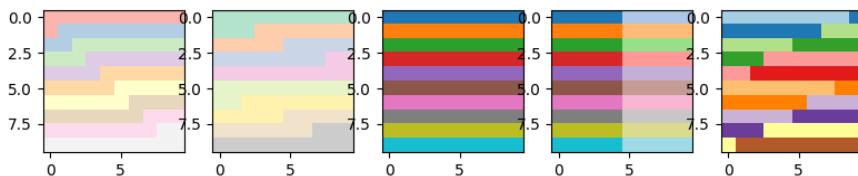
Then, we've got the *diverging* colormaps, including the all-too-familiar *coolwarm* colormap:



The *cyclic* colormaps repeat on each row, similar to *diverging colormaps* in this setup, but don't have a central unsaturated color in the middle. The end of each row is the predecessor of the start of the next one:



And finally, *qualitative* colormaps:



Similar to how `red` is mapped to a certain RGB value - each string representation of a colormap is mapped to a `cm.map_name`. "coolwarm" is mapped to `cm.coolwarm`, etc. You can choose either of these approaches to defining the colormap for your plot.

Customizing Layouts with GridSpec

We've mainly been adding `Axes` instances to `Figures` through a rudimentary process:

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

Alternatively, we've added `Axes` instances to `Figures` through the `add_subplot()` function:

```
1 fig = plt.figure()
2 ax = fig.add_subplot(121)
3 # OR
4 ax = fig.add_subplot(1, 2, 1)
```

The former is a nice and easy way to add axes to a figure, with a simple layout in mind, and the latter is a nice way to add them dynamically, since we can replace some of these parameters with counters from loops.

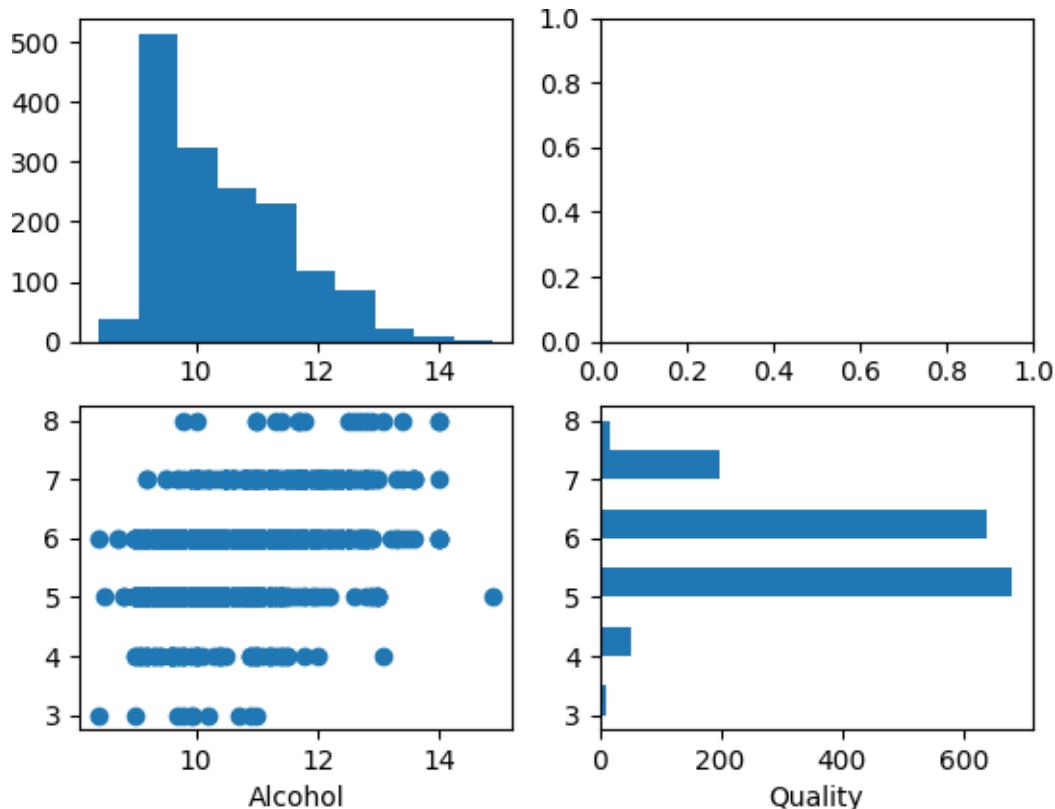
Though, this approach is fairly limited - and this becomes evident when we try to work with many `Axes` instances, and *especially* if they're not uniform. For example, we might want to have many `Axes` stacked vertically, but then have a large one at the bottom, displaying some aggregate data. Or, we might want one `Axes` to have an aspect of 16:9, while the other one having an aspect of, say, 5:4. We might want to have one `Axes` spanning through the entire `Figure`, left to right, while the next `Axes` to only span half of that distance. On the other hand, we might want to have the first

Axes fairly short, but the second Axes to stretch from the first one to the bottom of the Figure.

In all of these cases - we can't just rely on the `subplots()` and `add_subplot()` functions to get us through. A problem we encountered a long time ago in the book were *Joint Plots*. Let's get a refresher on the first attempt at making them, using only the `add_subplot()` function:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 df = pd.read_csv("winequality-red.csv")
5
6 alcohol = df['alcohol']
7 quality = df['quality']
8
9 fig, ax = plt.subplots(2, 2)
10
11 ax[0][0].hist(alcohol)
12 ax[1][0].scatter(x = alcohol, y = quality)
13 ax[1][0].set_xlabel('Alcohol')
14
15 ax[1][1].hist(quality, orientation='horizontal')
16 ax[1][1].set_xlabel('Quality')
17
18 plt.show()
```

We're creating a Figure with a few Axes instances, plotting a *Scatter Plot* at the bottom left and two *Histograms* on its margins:

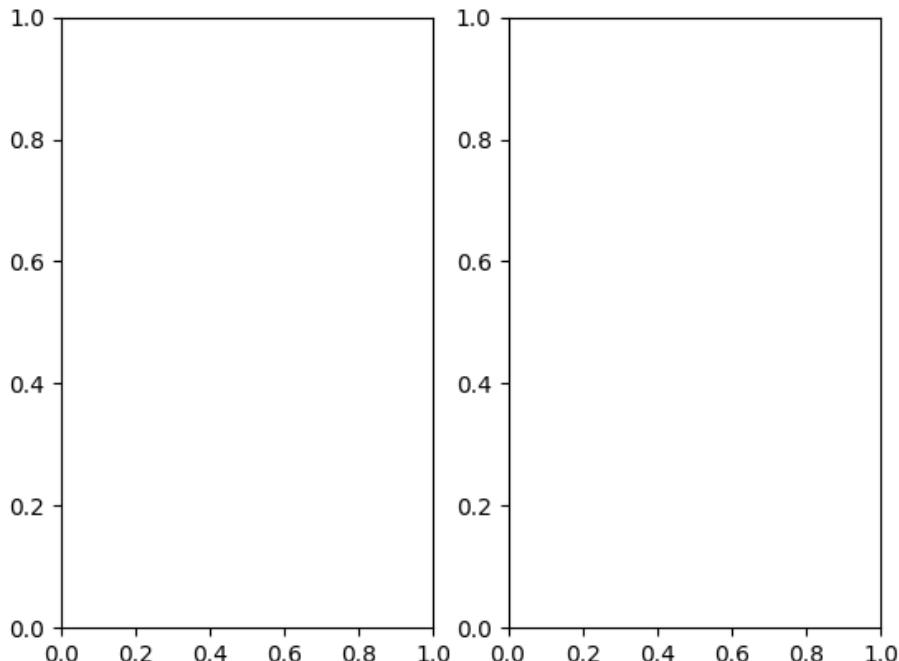


Since we define the rows and columns as just two numbers - they'll always form a rectangle or square. We can't have three plots shaped in an L-shape, which makes the top-right Axes totally useless. We can *turn it off* to make it invisible, but we won't really gain too much there, since the other plots are still awkwardly shaped and sized. The Histograms really don't need to be of the same size as the Scatter Plot, and if the Scatter Plot is the main focus - we're doing the user a disservice.

To mitigate these issues - we're introduced to `GridSpec`. It's used to specify a *layout* into which we can place subplots - `Axes` instances. The `GridSpec` instance is created separately from the `Figure`. Additionally, it's used to create an `Axes` while adding its respective `Figure`:

```
1 import matplotlib.pyplot as plt
2 # Importing GridSpec
3 import matplotlib.gridspec as gridspec
4
5 fig = plt.figure()
6 # Creating a GridSpec instance
7 gridspec = gridspec.GridSpec(ncols=2, nrows=1)
8
9 # Adding subplots via GridSpec, as
10 ax1 = fig.add_subplot(gridspec[0, 0])
11 ax2 = fig.add_subplot(gridspec[0, 1])
12
13 plt.show()
```

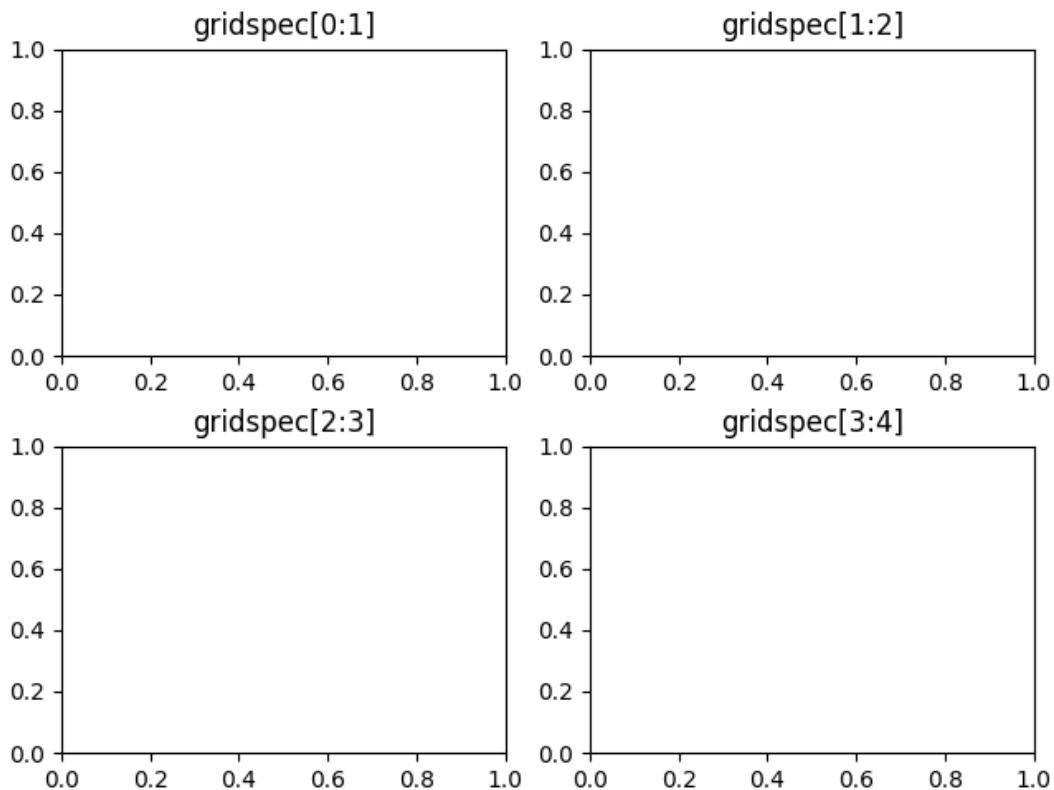
While we also define `ncols` and `nrows` - we don't have to occupy them, allowing us to shapes of Axes structures other than rectangles and squares. Via `GridSpec`, we've defined the positions of `ax1` and `ax2`, and achieved what we could've achieved with a simple `plt.subplots(2, 1)`:



When dabbling with the layout, there's a high probability that some elements will overlap between Axes, so it's a good idea to use a `constrained_layout` for your Figure. When setting the `constrained_layout` to True, we also have to define the optional `figure` argument of the `GridSpec` constructor to our `fig`.

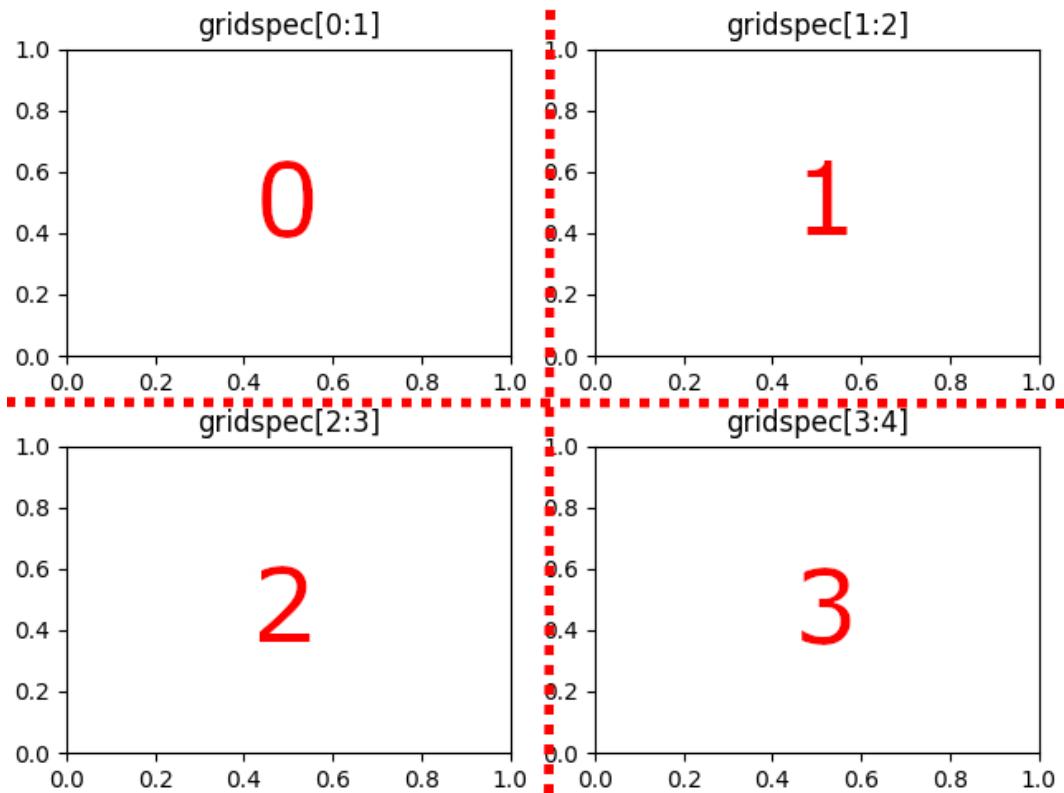
Let's get a feel for how `GridSpec` lays things out:

```
1 import matplotlib.pyplot as plt
2 import matplotlib.gridspec as gridspec
3
4 fig = plt.figure(constrained_layout=True)
5 gridspec = gridspec.GridSpec(2, 2, figure=fig)
6
7 ax1 = fig.add_subplot(gridspec[0:1])
8 ax1.set_title('gridspec[0:1]')
9
10 ax2 = fig.add_subplot(gridspec[1:2])
11 ax2.set_title('gridspec[1:2]')
12
13 ax3 = fig.add_subplot(gridspec[2:3])
14 ax3.set_title('gridspec[2:3]')
15
16 ax4 = fig.add_subplot(gridspec[3:4])
17 ax4.set_title('gridspec[3:4]')
18
19 plt.show()
```



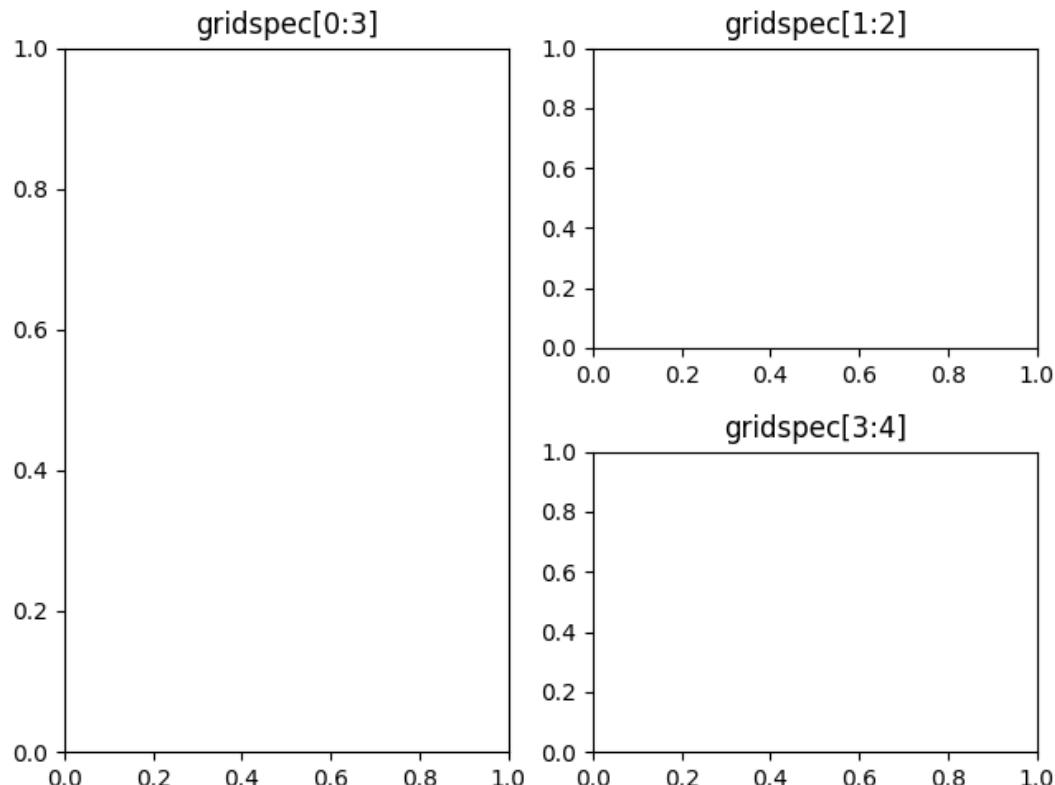
There are *four spaces*, since we've created two rows and two columns. The first Axes here is taking the free space between 0 and 1, the second is taking the free space between 1 and 2, etc.

It's worth taking a look at this Figure as a *grid*:



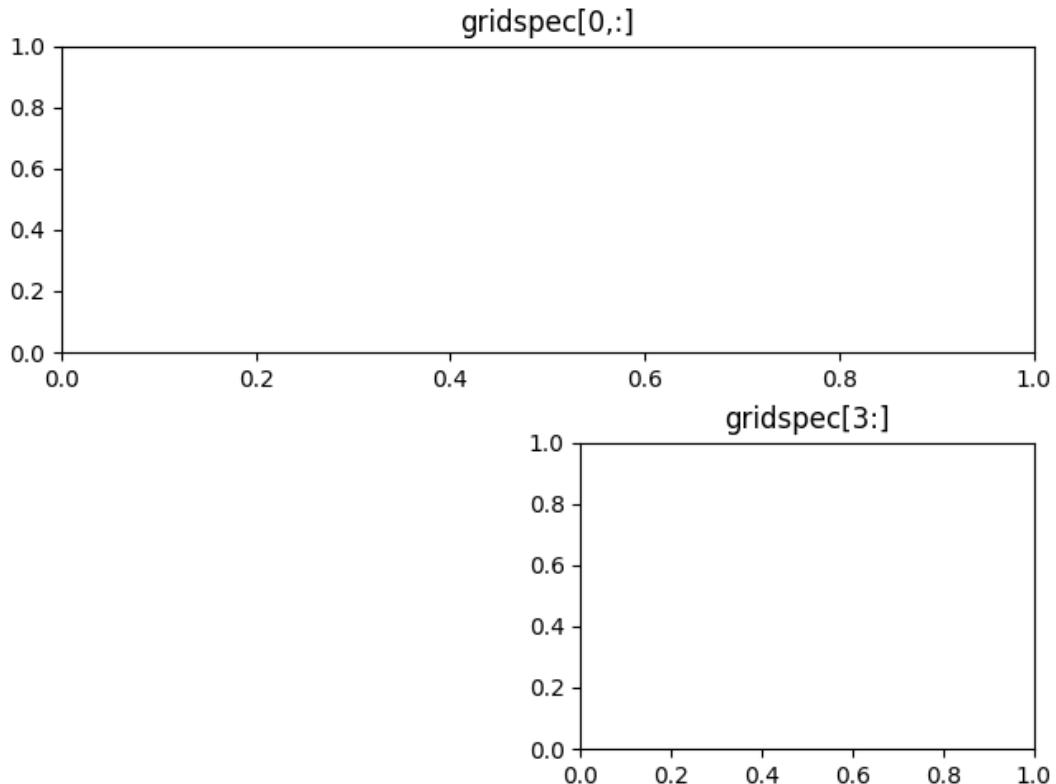
If this was our goal - it'd have been easier to use the `subplots()` function. Though, if we'd like to define *spans* of `Axes` instances, the only way to do that is through `GridSpec`. Let's change the code up so that `ax1` doesn't take the slots from `0..1`, but rather, from `0..3`. In that case, it'll take up the left-hand side of the `Figure`, since `0..3` would be as if we had combined `[0:1]` and `[2:3]`:

```
1 import matplotlib.pyplot as plt
2 import matplotlib.gridspec as gridspec
3
4 fig = plt.figure(constrained_layout=True)
5 gridspec = gridspec.GridSpec(2, 2, figure=fig)
6
7 ax1 = fig.add_subplot(gridspec[0:3])
8 ax1.set_title('gridspec[0:3]')
9
10 ax2 = fig.add_subplot(gridspec[1:2])
11 ax2.set_title('gridspec[1:2]')
12
13 ax4 = fig.add_subplot(gridspec[3:4])
14 ax4.set_title('gridspec[3:4]')
15
16 plt.show()
```



We can also use the *slicing notation* to define a *range* of positions in the layout:

```
1 import matplotlib.pyplot as plt
2 import matplotlib.gridspec as gridspec
3
4 fig = plt.figure(constrained_layout=True)
5 gridspec = gridspec.GridSpec(ncols=2, nrows=2, figure=fig)
6
7 # `0,:` slices the 0th subarray, i.e. the first row
8 # `0:` slices from 0..max
9 ax1 = fig.add_subplot(gridspec[0,:])
10 ax1.set_title('gridspec[0,:]')
11
12 # Slicing from 3 and onward
13 ax2 = fig.add_subplot(gridspec[3:])
14 ax2.set_title('gridspec[3:]')
15
16 plt.show()
```



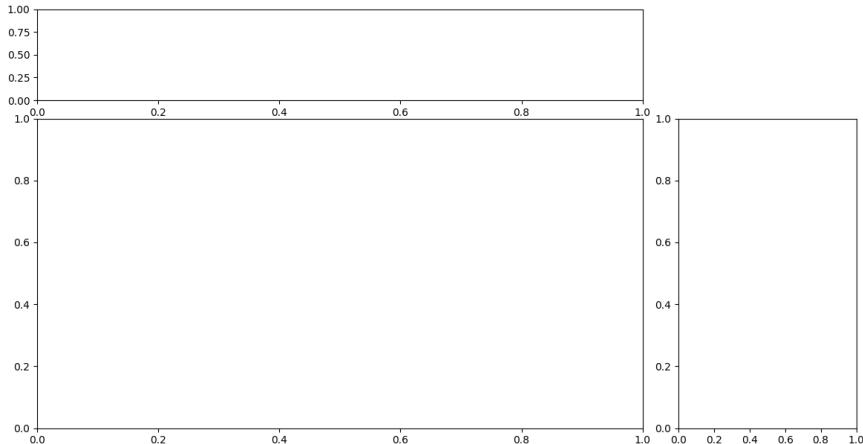
We can also use *negative indexing* to signify these slots in reverse order - in our case `gridspec[-1:]` and `gridspec[3:]` evaluate to the same slots, since this is the last slot.

With this in mind, we can revisit the *Joint Plot* from before:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import matplotlib.gridspec as gridspec
4
5 df = pd.read_csv('iris.csv')
6
7 fig = plt.figure()
8 gridspec = gridspec.GridSpec(4, 4)
9
10 # From 1 to 4, and 0 to 3, allowing for a slot
11 # above it and to its right
12 ax_scatter = fig.add_subplot(gridspec[1:4, 0:3])
13 # Axes above `ax_scatter`, on 0, but spanning from 0 to 3
14 # just like the scatter plot
15 ax_hist_y = fig.add_subplot(gridspec[0,0:3])
16 # Axes to the right, spanning only one slot
17 ax_hist_x = fig.add_subplot(gridspec[1:4, 3])
18
19 plt.show()
```

The `ax_scatter`'s height occupies slots from `1:4`, while the Figure's overall height is `0:4`. This leaves one slot on the top for another plot. The `ax_scatter`'s width occupies `0:3`, while the Figure's overall width is `0:4`, leaving a slot on the right.

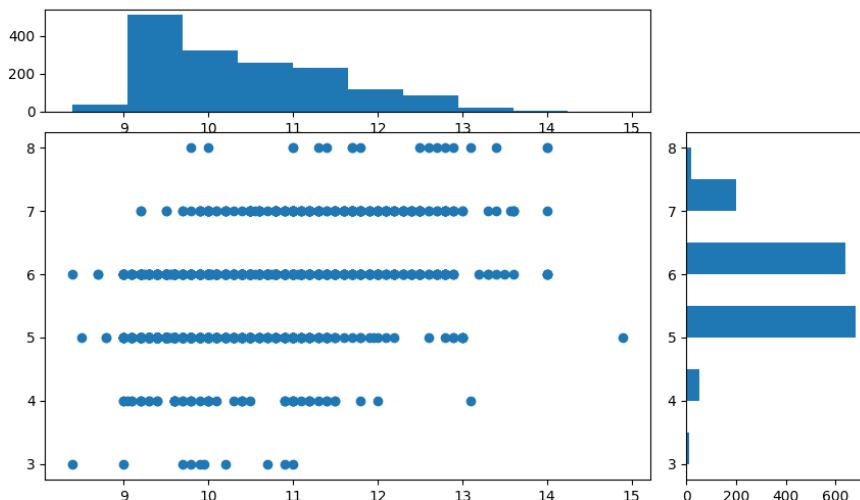
The `ax_hist_y`'s height occupies just the top slot, which was left free beforehand, but its width spans `0:3`, just like `ax_scatter`. Similarly, `ax_hist_x`'s height spans from `1:4`, leaving the top free, and its width spans only on the right-most free slot:



And if we insert the same data as in the first, failed Joint Plot attempt:

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import matplotlib.gridspec as gridspec
4
5 df = pd.read_csv("winequality-red.csv")
6
7 alcohol = df['alcohol']
8 quality = df['quality']
9
10 fig = plt.figure()
11 gs = gridspec.GridSpec(4, 4)
12
13 ax_scatter = fig.add_subplot(gs[1:4, 0:3])
14 ax_hist_y = fig.add_subplot(gs[0, 0:3])
15 ax_hist_x = fig.add_subplot(gs[1:4, 3])
16
17 ax_scatter.scatter(x = alcohol, y = quality)
18 ax_hist_y.hist(alcohol)
19 ax_hist_x.hist(quality, orientation='horizontal')
20
21 plt.show()
```

We'll be greeted with:



Chapter 8 - Data Visualization with Pandas

Pandas has been aiding us so far in the phase of Data Preprocessing. Though, in one instance, while creating Histograms, we've also utilized another module from Pandas - plotting.

We've purposefully avoided it so far, because introducing it earlier would raise more questions than it answered. Namely, Pandas *and* Matplotlib were such a common and ubiquitous duo, that Pandas has started integrating Matplotlib's functionality. It *heavily* relies on Matplotlib to do any actual plotting, and you'll find many Matplotlib functions wrapped in the source code. Alternatively, you can use other backends for plotting, such as *Plotly* and *Bokeh*.

However, Pandas also introduces us to a couple of plots that *aren't* a part of Matplotlib's standard plot types, such as *KDEs*, *Andrews Curves*, *Bootstrap Plots* and *Scatter Matrices*.

The `plot()` function of a Pandas `DataFrame` uses the backend specified by `plotting.backend`, and depending on the `kind` argument - generates a plot using the given library. Since a lot of these overlap - there's no point in covering plot types such as `line`, `bar`, `hist` and `scatter`. They'll produce much the same plots with the same code as we've been doing so far with Matplotlib.

We'll only briefly take a look at the `plot()` function since the underlying mechanism has been explored so far. Instead, let's focus on some of the plots that we *can't* already readily do with Matplotlib.

This will be a very short chapter, as Pandas' plotting and visualization capabilities pale in comparison to Matplotlib - but it's still useful to know of some of these, as well as be aware of the ability to plot from `DataFrames` directly.

The `DataFrame.plot()` Function

The `plot()` function accepts `x` and `y` features, and a `kind` argument. Alternatively, to avoid the `kind` argument, you can also call `DataFrame.plot.kind()` instead, and pass in the `x` and `y` features.

The accepted values for the `kind` argument are: `line`, `bar`, `barh` (horizontal bar), `hist`, `box`, `kde`, `density` (synonym for `kde`), `area`, `pie`, `scatter` and `hexbin` (similar to a Heatmap).

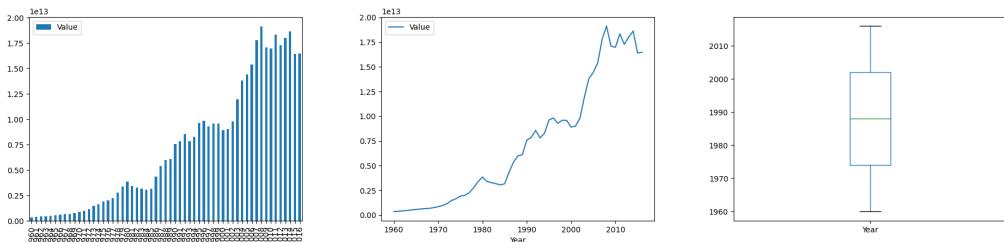
```

1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('gdp_csv.csv')
5 df_eu = df[df['Country Name'] == 'European Union']
6
7 df_eu.plot.bar(x = 'Year', y = 'Value')
8 df_eu.plot.line(x = 'Year', y = 'Value')
9 df_eu.plot.box(x='Value')
10
11 plt.show()

```

Instead of providing the `Series` instances like we do for Matplotlib - it's enough to provide the *column names*, and since you're calling the `plot()` function on the `DataFrame` you're visualizing, it's easy for Pandas to map these string values to the appropriate column names.

Each of these calls makes a new `Figure` instance and plots the appropriate values on it:



To plot multiple axes on the same `Figure`, you can make a `Figure` and one or more `Axes` via `plt.subplots()` and assign the appropriate `Axes` to the `ax` argument of each `plot.plot_type()` call:

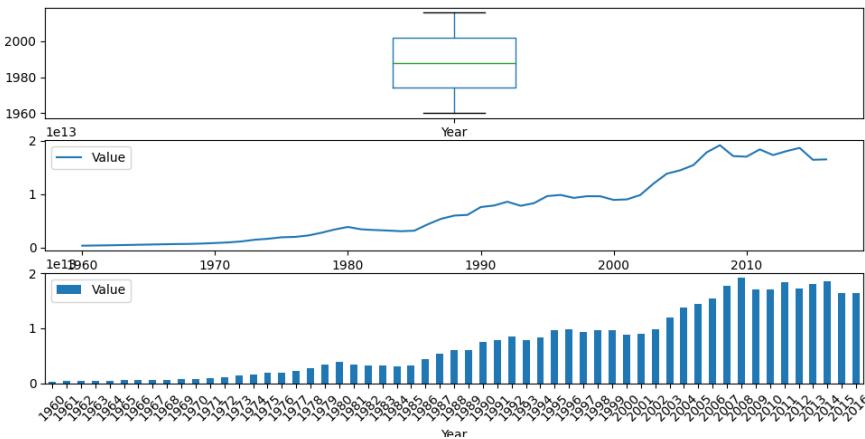
```

1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('gdp_csv.csv')
5 df_eu = df.loc[df['Country Name'] == 'European Union']
6
7 fig, ax = plt.subplots(3, 1)
8
9 df_eu.plot.box(x='Year', ax=ax[0])
10 df_eu.plot.line(x = 'Year', y = 'Value', ax = ax[1])
11 df_eu.plot.bar(x = 'Year', y = 'Value', rot = 45, ax = ax[2])
12
13 plt.show()

```

Some of the standard arguments, such as the `rotation` argument are actually different in the `plot()` calls. For example, `rotation` is shortened to `rot`. This makes it a bit tricky to *just switch* between Matplotlib and Pandas, as you'll most likely end up in the documentation pages, just checking which arguments can be applied and which can't.

Now, instead of creating a new `Figure` or `Axes` instance, each of these plots will be stationed in the appropriate `Axes` instances we've supplied them with:



In general, plotting with Pandas is convenient and quick - but even so, for plotting Bar Plots, Line Plots and Box Plots, you'll probably want to go with Matplotlib. It's both the underlying engine Pandas inevitably uses, and also has more customization options, and you won't have to remember a new set of arguments that you can use with Pandas plots.

That being said, for some plots, you might want to prefer *Pandas*, since they'd have to be manually made in Matplotlib, and some of them are such a hassle to make that it's not worth the effort, such as KDE lines.

Pandas' *plotting* Module

What `DataFrames` have to offer in terms of visualization isn't too new to us. However, the underlying module they call, `pandas.plotting` does. The plotting module has several functions that we can use, such as `autocorrelation_plot()`, `bootstrap_plot()`, and `scatter_matrix()`.

Each of these accept either a `Series` or a `DataFrame`, depending on the type of visualization they're producing, as well as certain parameters for plotting specification and styling purposes.

Bootstrap Plot

*Bootstrapping*⁷⁰ is the process of randomly sampling (with replacement) a dataset, and calculating measures of accuracy such as *bias*, *variance* and *confidence intervals* for the random samples. “With replacement”, in practical terms, means that each randomly selected element can be selected again. *Without replacement* means that after each randomly selected element, it's removed from the pool for the next sample.

A *Bootstrap Plot*, created by Pandas bootstraps the mean, median and mid-range statistics of a dataset, based on the sample `size`, after which the plot is subsequently shown via `plt.show()`. The default arguments for `size` and `samples` are 50 and 500 respectively.

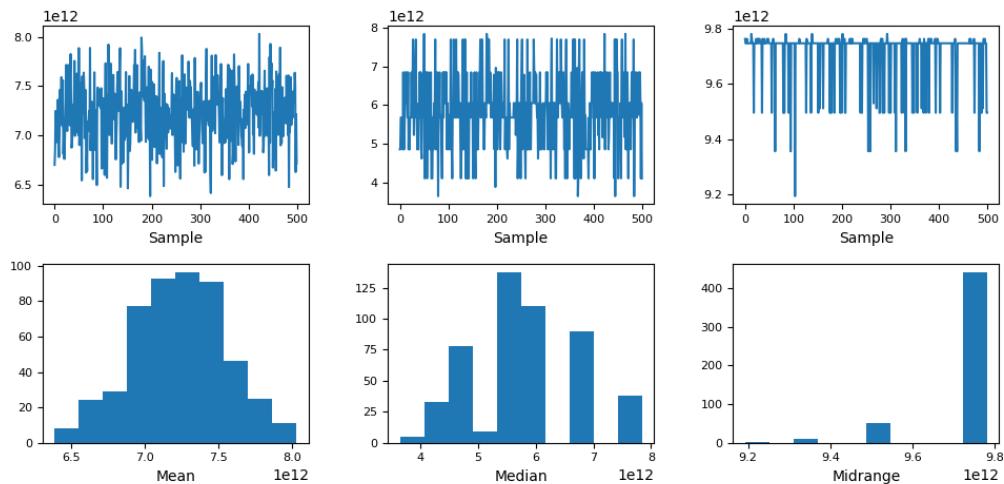
This means that for a feature, we sample 50 values. Then, a 50-element sub-sample is generated (synthetic data) for those 50 values and a summary statistic (mean/median/mid-range) is calculated for them. This process is repeated 500 times, so in the end, we've got 500 summary statistics:

⁷⁰[https://en.wikipedia.org/wiki/Bootstrapping_\(statistics\)](https://en.wikipedia.org/wiki/Bootstrapping_(statistics))

```

1 import pandas as pd
2 from matplotlib import pyplot as plt
3
4 df = pd.read_csv('./datasets/gdp_csv.csv')
5 df_eu = df.loc[df['Country Name'] == 'European Union']
6
7 pd.plotting.bootstrap_plot(df_eu['Value'])
8
9 plt.show()

```



Autocorrelation Plot

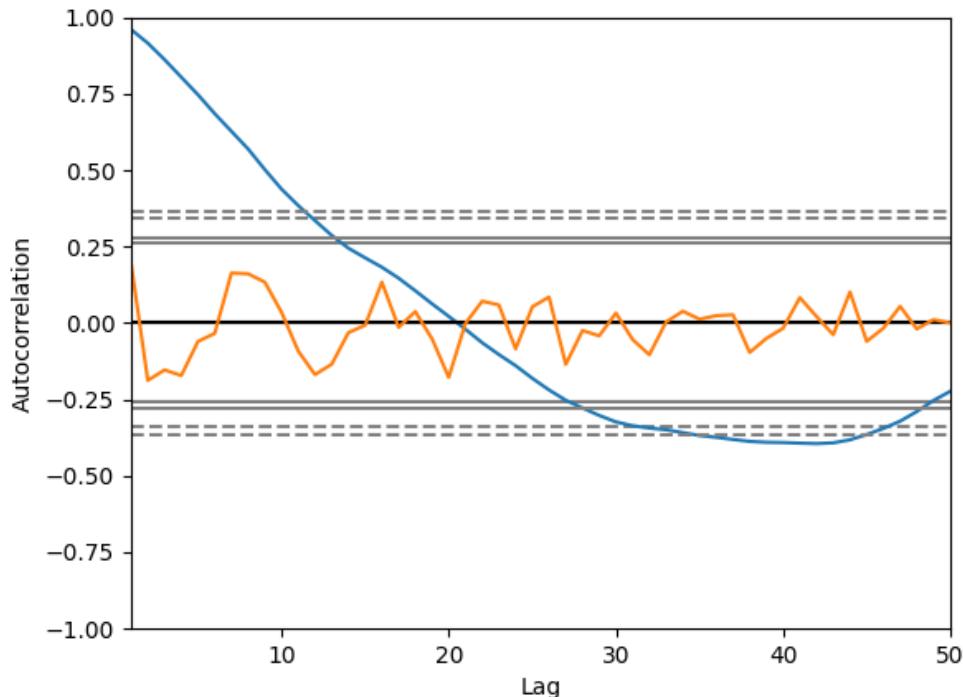
Autocorrelation Plots are used to check for data randomness, for time-series data. Multiple [autocorrelations⁷¹](#) are calculated for differing timestamps, and if the data is truly random - the correlation will be near zero. If not - the correlation will be larger than zero.

Let's plot two Autocorrelation Plots - one with our *Value* feature, and one with a Series filled with random values:

⁷¹<https://en.wikipedia.org/wiki/Autocorrelation>

```
1 import pandas as pd
2 import numpy as np
3 from matplotlib import pyplot as plt
4
5 df = pd.read_csv('gdp_csv.csv')
6 # Filter DataFrame for the EU
7 df_eu = df.loc[df['Country Name'] == 'European Union']
8 # Generate 50 random integers between 0 and 100, and turn into a Series
9 random_series = pd.Series(np.random.randint(0, 100, size=50))
10
11 # Plot Autocorrelation Plot for the *Value* feature
12 pd.plotting.autocorrelation_plot(df_eu[['Value']])
13 # Plot Autocorrelation Plot for the *random_series*
14 pd.plotting.autocorrelation_plot(random_series)
15
16 plt.show()
```

The Autocorrelation Plot for the `random_series` should revolve around 0, since it's random data, while the plot for the `Value` feature won't:



It's worth noting that Autocorrelation measures one *form* of randomness, as uncorrelated, but non-random data does exist. If it's non-random, but doesn't have any significant correlations - the Autocorrelation Plot would indicate that the data is random.

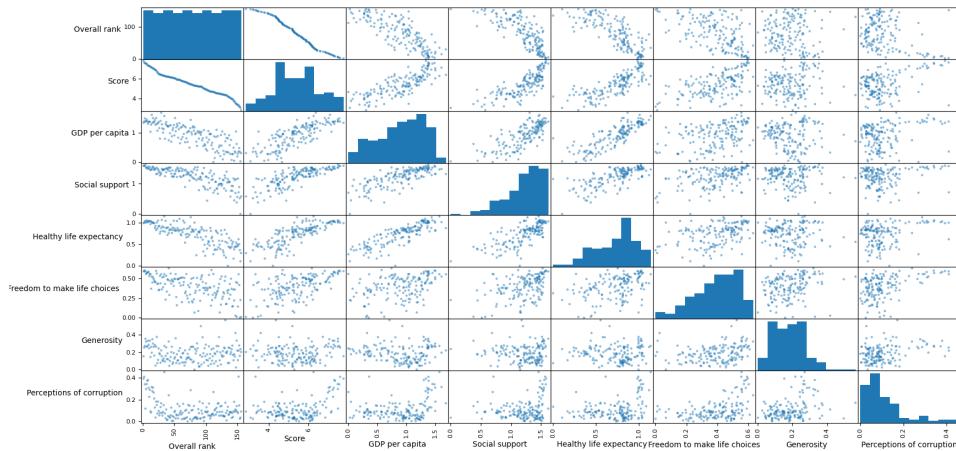
Scatter Matrices

Scatter Matrices plot a *grid* of Scatter Plots for all features against all features. Since this inevitably compares each feature with itself, as well - the *diagonal* where this happens is typically replaced with a *Histogram* of that feature, rather than a Scatter Plot. *Scatter Matrices* are also known as *Pair Plots*, and Seaborn offers a `pairplot()` function just for this.

The `scatter_matrix()` function accepts a `DataFrame` and produces a Scatter Matrix for all of its numerical features, and returns a 2D array of `Axes` instances that comprise the Scatter Matrix. To tweak anything about them, you'll want to iterate through them:

```
1 import pandas as pd
2 import numpy as np
3 from matplotlib import pyplot as plt
4
5 df = pd.read_csv('worldHappiness2019.csv')
6
7 axes = pd.plotting.scatter_matrix(df, diagonal='hist')
8
9 for ax in axes.flatten():
10     # Rotate back to 0 degrees since they're automatically rotated by 90
11     ax.yaxis.label.set_rotation(0)
12     # As to not overlap with the Axes instances, set the ticklabel
13     # alignment to 'right'
14     ax.yaxis.label.set_ha('right')
15
16 plt.show()
```

This results in a rather large Scatter Matrix of all the features against all other features:



You can also pass in the `diagonal` argument, which accepts 'hist' or 'kde' to specify what type of distribution plot you'd like to plot on the diagonal, as well as `alpha`, specifying the translucency of the markers in the Scatter Plots.

Chapter 9. - Matplotlib Widgets

Matplotlib isn't only for static plots. While GUIs are typically created with GUI libraries and frameworks such as [PyQt⁷²](#), [Tkinter⁷³](#), [Kivy⁷⁴](#) and [wxPython⁷⁵](#), and while Python does have excellent integration with PyQt, Tkinter and wxPython - there's no need to use any of these for some basic GUI functionality, through *Matplotlib Widgets*.

The `matplotlib.widgets` module has several classes, including the `AxesWidget`, out of which `Buttons`, `CheckButtons`, `Sliders`, `TextBoxes`, etc are derived. These all accept the `Axes` they're being added to as the one and only mandatory constructor argument, and their positioning has to be manually set. A thing to note is that the *widget is the axes*, so you'll create an `Axes` instance for each widget.

Another thing to note is that *you have to keep references to the widgets* otherwise, they might get garbage collected.

Each of them can also be disabled by setting `active` to `False`, in which case, they won't respond to any events, such as being clicked on. That being said, we can introduce a new type of interactivity to our plots, through various GUI elements and components.

Note: Matplotlib isn't meant to be used for high-quality GUI creation, nor user-friendly systems. These widgets are rudimentary, don't really look great and have limited functionality. They're meant as a way to prototype and test things out, rather than actually ship them.

If you've worked with PyQt before - you might notice that the general syntax and approach to adding these widgets, as well as connecting them to event handlers is fairly familiar.

⁷²<https://riverbankcomputing.com/software/pyqt/>

⁷³<https://docs.python.org/3/library/tkinter.html>

⁷⁴<https://kivy.org/#home>

⁷⁵<https://www.wxpython.org/>

Adding Buttons

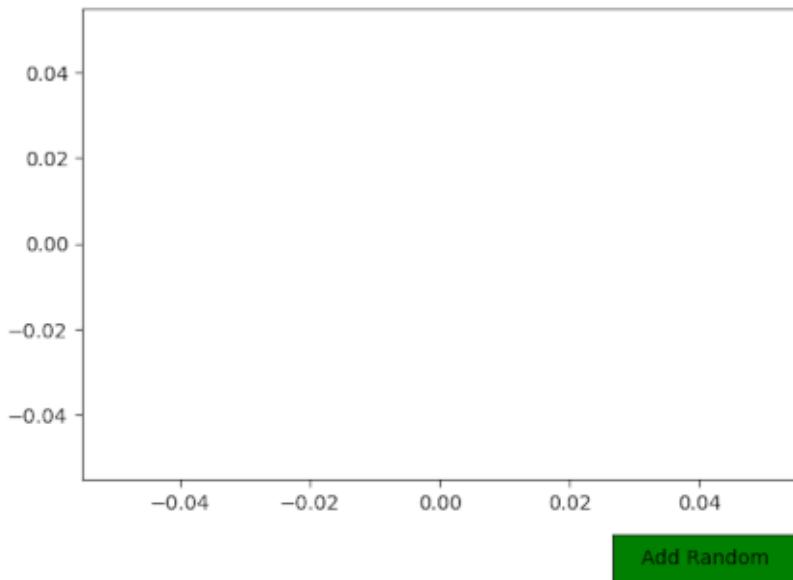
Let's start out with *buttons* - the `matplotlib.widgets` module defines a `Button` class. To connect to it, we call the `on_clicked()` function, which executes the function we supply. Once a click has been detected, the function executes.

While creating the button, we assign an `Axes` to it, used for positioning. We can also pass in a `label` at that time, to add some text and annotate it for a user. The `color` and `hovercolor` arguments define the color of the button before and after it's being hovered over.

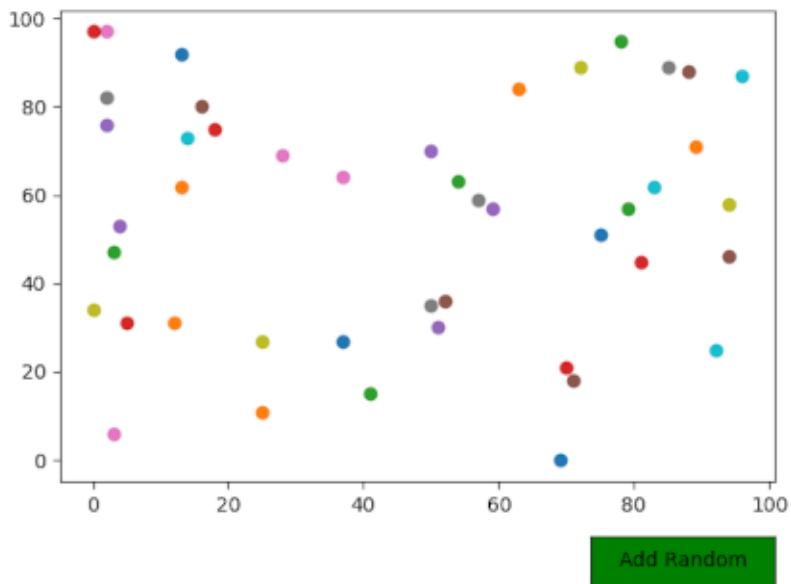
Since we take care of the positioning and space for all widgets - let's create a `Figure` and `Axes`, allow for some spacing at the bottom to add a button, and plot an empty Scatter Plot. Then, we'll define an `EventHandler` class, that has a single method `add_random()`. The method generates two random numbers, and plots a marker for them on the `Axes` we've created before and calls `plt.draw()`, which re-draws the `Figure`. When updating plots, we'll always have to call `plt.draw()` again to actually update it:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib.widgets import Button
4
5 fig, ax = plt.subplots()
6 fig.subplots_adjust(bottom=0.2)
7 plot = ax.scatter([], [])
8
9 class EventHandler:
10     def add_random(self, event):
11         x = np.random.randint(0, 100)
12         y = np.random.randint(0, 100)
13         ax.scatter(x, y)
14         plt.draw()
15
16 # Axes for the Button and positioning
17 # xposition and yposition in percentages, width, height
18 button_ax = plt.axes([0.7, 0.05, 0.2, 0.07])
19 # Create Button and assign it to `button_ax` with label
20 button = Button(button_ax, 'Add Random', color='green', hovercolor='red')
21 # On a detected click, execute add_random()
22 button.on_clicked(EventHandler().add_random)
23
24 plt.show()
```

This results in a `Figure`, with an empty `Axes` inside of it and a button on the top right-hand corner of the screen, in its own `Axes`:



And after pressing the button a couple dozen times, our `ax` will be populated with random markers:



In more practical terms, we could create a *cycle* of features to get plotted on each button press. This requires a few tweaks to the `EventHandler`, as well as another button to go back through that cycle.

Let's use the [Red Wine Quality](#)⁷⁶ dataset again, and visualize several features against the *Alcohol* feature. Since we can't be bothered to plot these individually by writing the code to plot one feature against the other, and then modifying that code to plot another feature against the other.

Creating a Scatter Matrix might help us here, but if the dataset has a lot of features, it'll be fairly unreadable and we won't get far. If you'd like to have *both* large-scale plots that you can easily view and interpret, *as well* as having multiple features cycling through without any extra effort - you can automate this process with buttons:

⁷⁶<https://www.kaggle.com/uciml/red-wine-quality-cortez-et-al-2009>

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from matplotlib.widgets import Button
5
6 fig, ax = plt.subplots()
7 fig.subplots_adjust(bottom=0.2)
8
9 df = pd.read_csv('winequality-red.csv')
10 plot = ax.scatter([], [])
11
12 class EventHandler:
13     i = 0
14     # Find and plot next feature, re-draw the Axes
15     def next_feature(self, event):
16         # If the counter is at the end of the columns
17         # Revert it back to 0 to cycle through again
18         if self.i >= len(df.columns):
19             self.i = 0
20         # Clear Axes from last plot
21         ax.cla()
22         # Plot a feature against a feature located on the `i` column
23         ax.scatter(df['alcohol'], df.iloc[:,self.i])
24         # Set labels
25         ax.set_xlabel('Alcohol')
26         ax.set_ylabel(df.columns[self.i])
27         # Increment i
28         self.i += 1
29         # Update Figure
30         plt.draw()
31
32     def previous_feature(self, event):
33         # If the counter is at the start of the columns
34         # Revert it back to the last column to cycle through
35         if self.i <= 0:
36             self.i = len(df.columns)-1
37         ax.cla()
38         ax.scatter(df['alcohol'], df.iloc[:,self.i])
39         ax.set_xlabel('Alcohol')
40         ax.set_ylabel(df.columns[self.i])
41         self.i -= 1
42         plt.draw()
43
44     # Add buttons
45 button1_ax = plt.axes([0.7, 0.02, 0.2, 0.07])
46 next_button = Button(button1_ax, 'Next Feature')
47 next_button.on_clicked(EventHandler().next_feature)
48
49 button2_ax = plt.axes([0.45, 0.02, 0.2, 0.07])
50 previous_button = Button(button2_ax, 'Previous Feature')
51 previous_button.on_clicked(EventHandler().previous_feature)
52
53 plt.show()
```

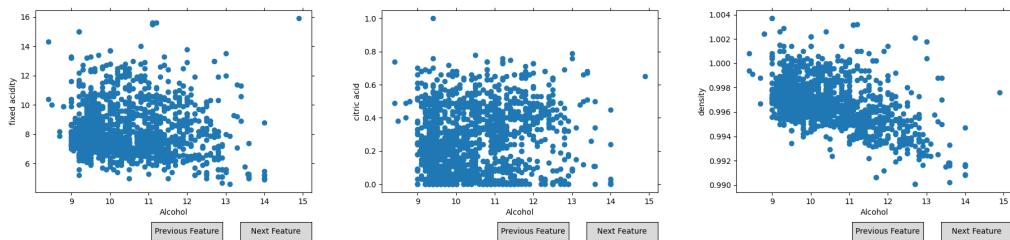
The EventHandler class now has two methods - `next_feature()` and `previous_feature()`. Both of these check whether the counter `i` has reached the end or start of the column list - and to avoid an `IndexError`, we reset the index to the opposite value

and simulate a *cycle*. Going below 0 will get us back to the *end* of the column list, and going above the last column will revert us back to the first.

After ascertaining where we're located - we *clear the Axes*, since we'd be plotting again on top of an existing plot without clearing it via `c1a()` (*clear axes*). You could alternatively pile up feature relations as well, by plotting on top of each other and use the `c1a()` statement when resetting the index at the end/start of the cycle.

After clearing the *Axes* - we've got a cleared canvas to paint on with the `ax.scatter()` function. In this example, the fixed feature is *Alcohol*, so it's present at all times. The other feature varies, and can be accessed through `i1oc[]`, passing in the index of the column. This returns a *Series* that we can use in this plot. Similarly, we can access *column names* through their index as well - `df.columns[index]`, which is used to set the Y-axis label.

Finally, we increase/decrease the counter and call `plt.draw()` to update the *Figure*:



Once we click on the *Next Feature* button, the next feature in the list of columns will be plotted against *Alcohol*, and the *Figure* will be appropriately updated - the labels, markers and scale. The same goes the other way around - *Previous Feature* will traverse the list in the opposite direction, allowing us to cycle back and forth, with a safety mechanism that resets our counter each time we get to the end or beginning of the cycle.

Adding Radio Buttons and Check Boxes

Radio Buttons are used to allow a user to select *one value* out of *several values*. Only one radio button can be selected at a time, and they typically represent a choice. *Check Boxes* can be used if you'd like to let the user select multiple options at once.

Note: There is very limited ability to check whether a checkbox is *on* or *off*. In fact, there's none out of the box. You can only ever check if the box is *pressed* or *not*, which poses a serious limitation to how it can be used since we have no idea in which state it was before that. The only alternative is to keep your own counter/check as to the current state of the box with a boolean, and alter the logic based on that.

This would allow you to, for example, add a checkbox for each *customization argument* of a certain plot, allowing the user to set them True or False (checked, or unchecked), or any other non-conflicting mapping based on these states.

Though, since the API is limited itself, we'll limit ourselves to the intended usage as well - turning things on and off. We'll have two features, that we can turn *on* and *off* via a Checkbox. Note that even this functionality is limited to objects for which you can check if they're visible or not.

On the other hand, we don't want to allow the user to apply two scales at once, or to set two X-limits at once, since only the statement called second in the sequence would be applied. For these - we'd use Radio Buttons.

Let's add a couple of Radio Buttons to let the user select the axis range via a couple of Radio Buttons, but also allow them turn feature visualizations *on* and *off*:

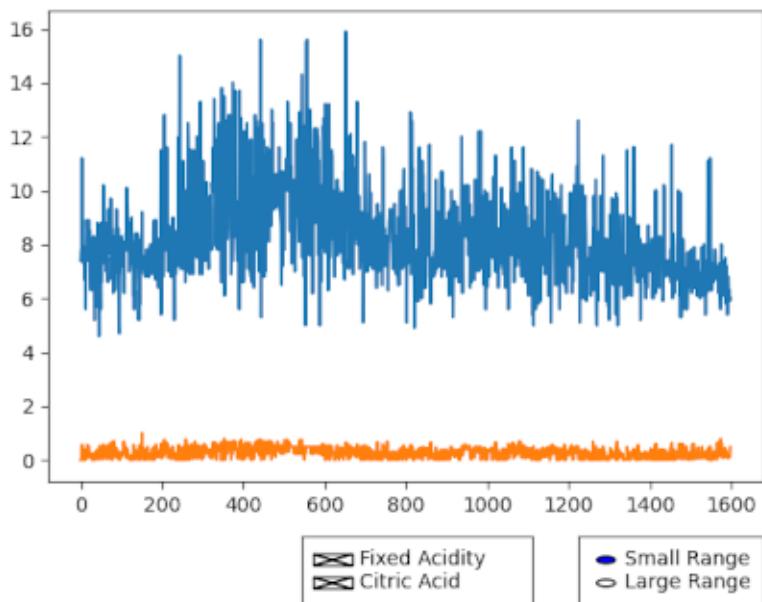
```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from matplotlib.widgets import CheckButtons
4 from matplotlib.widgets import RadioButtons
5
6 fig, ax = plt.subplots()
7 fig.subplots_adjust(bottom=0.2)
8
9 df = pd.read_csv('winequality-red.csv')
10
11 # Plot two line plots for two features, and turn them invisible
12 line1, = ax.plot(df['fixed acidity'], visible=False)
13 line2, = ax.plot(df['citric acid'], visible=False)
14
15 class EventHandler:
16     # set_range handler
17     def set_range(self, label):
18         if (label == 'Small Range'):
19             ax.set_xlim(0, 1600)
20             ax.set_ylim(0, 25)
21         else:
22             ax.set_xlim(0, 1600)
23             ax.set_ylim(0, 50)
24         plt.draw()
25
26 # Turn off, if on, and on if off
27 def apply_features(self, label):
```

```
28     if (label == 'Fixed Acidity'):
29         line1.set_visible(not line1.get_visible())
30     elif (label == 'Citric Acid'):
31         line2.set_visible(not line2.get_visible())
32     plt.draw()
33
34 # Add radio buttons and checkboxes
35 ranges_ax = plt.axes([0.7, 0.02, 0.2, 0.1])
36 range_radio_buttons = RadioButtons(ranges_ax, ('Small Range', 'Large Range'))
37 range_radio_buttons.on_clicked(EventHandler.set_range)
38
39 checkboxes_ax = plt.axes([0.4, 0.02, 0.25, 0.1])
40 checkboxes = CheckButtons(checkboxes_ax, ('Fixed Acidity', 'Citric Acid'))
41 checkboxes.on_clicked(EventHandler.apply_features)
42
43 plt.show()
```

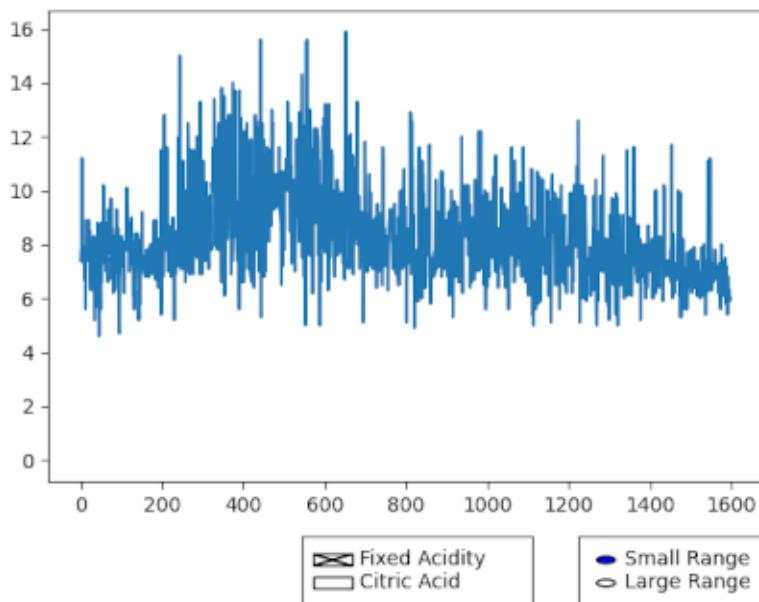
Again, we've got two methods in the `EventHandler()` class - `set_range()` and `apply_features()`. The `set_range()` method sets the range to either "small" or "large", by adjusting the Axes' X and Y-limits. The `apply_features()` function alters the `visible` field of the Line Plots we made earlier, based on their current `visible` status. If `visible == True`, we turn the Line Plot off, and vice versa.

We have to rely on the built-in ability to check the visibility of Line Plots, since we can't check if the checkbox was checked or not before. This same ability can be emulated with a `status` boolean in the scope of the `EventHandler()` class, which is set to `True` and `False` on every click, for plot types that don't support checking if they're visible out of the box.

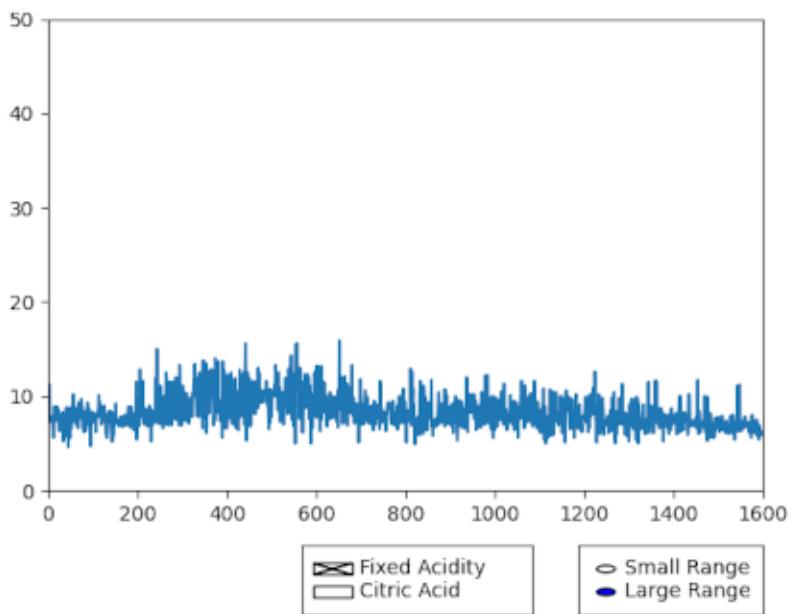
Running this code results in a Figure with two sets of buttons at the bottom. If we check both checkboxes, both Line Plots will appear:



We can turn them off individually:



And we can change the range of the Axes via the Radio Buttons:



Adding Textboxes

Textboxes are used to *collect* data from the user - and we can alter the plots based on this data. For example, we can ask a user to input the name of a feature, or to insert a function for our plot to visualize. Of course, working with user input can be tricky - there are always edge cases to look out for.

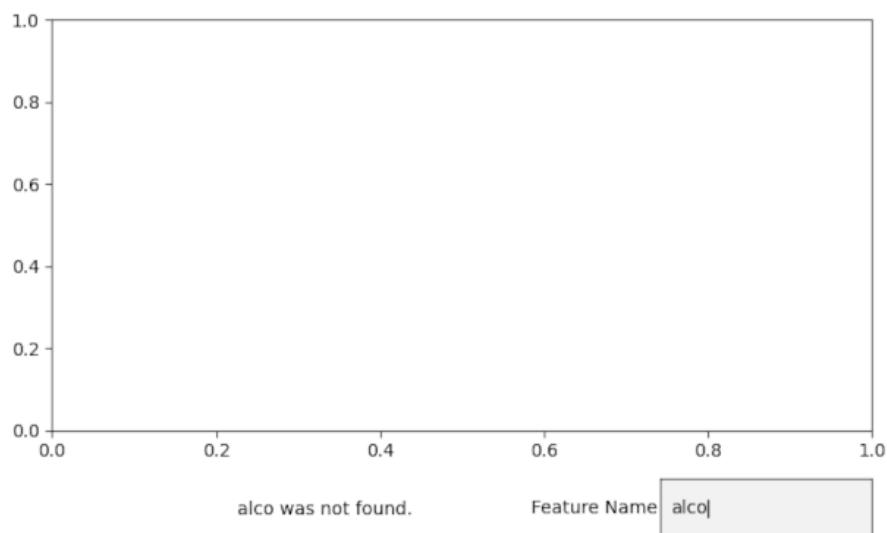
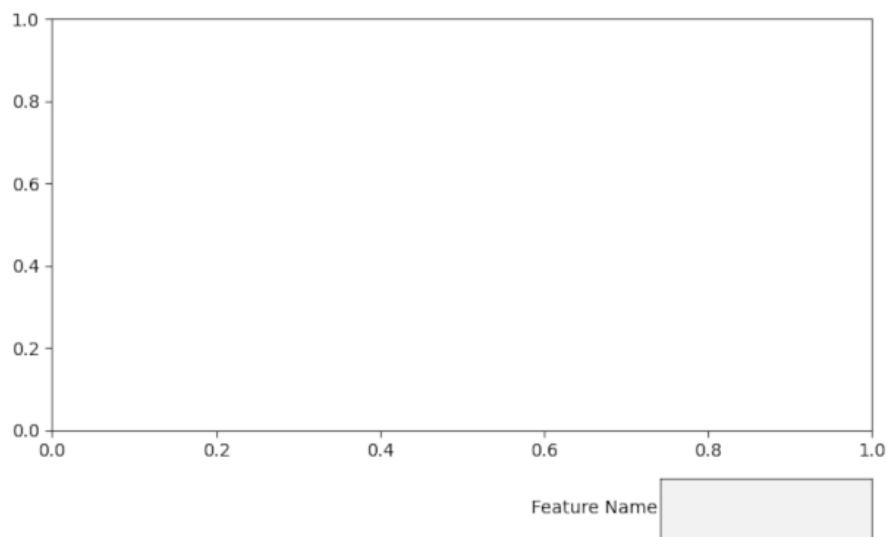
Let's write a script that allows the user to input a *feature name* of a dataset, and the Axes updates on each submission to reflect the input. For the convenience of the user, we'll let them know if the input couldn't be matched with a column name:

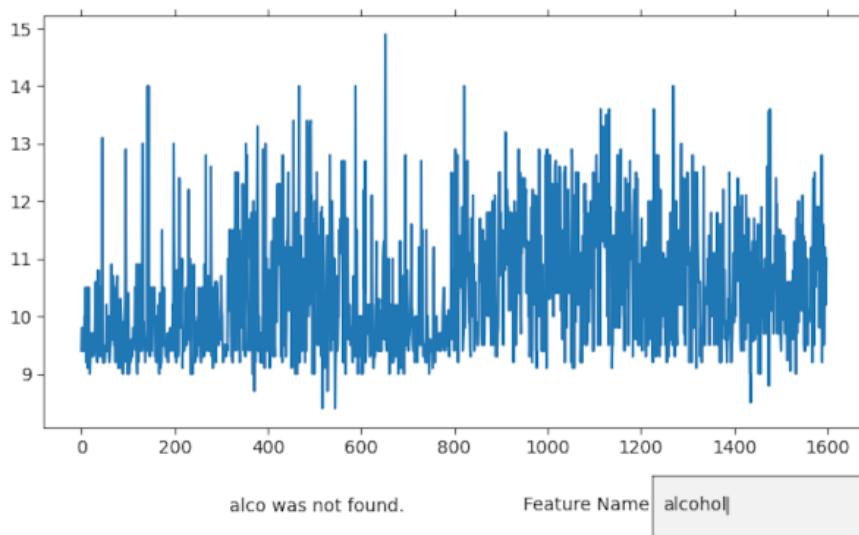
```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from matplotlib.widgets import TextBox
4
5 fig, ax = plt.subplots()
6 fig.subplots_adjust(bottom=0.2)
7
8 df = pd.read_csv('winequality-red.csv')
9
10 class EventHandler:
11     def submit(feature_name):
12         if feature_name != "" or feature_name != None:
13             if feature_name in df:
14                 ax.cla()
15                 ax.plot(df[feature_name])
16             else:
17                 if len(textbox_ax.texts) > 2:
18                     del textbox_ax.texts[-1]
19                 textbox_ax.text(-2, 0.4, feature_name + ' was not found.')
20         plt.draw()
21
22 textbox_ax = plt.axes([0.7, 0.02, 0.2, 0.1])
23 textbox = TextBox(textbox_ax, 'Feature Name')
24 textbox.on_submit(EventHandler.submit)
25
26 plt.show()
```

We have a simple check to see if the provided `feature_name` is blank or `None`, in which case, we don't do anything. If not, we check if the `feature_name` is present in the `DataFrame`, attaching a message that the feature wasn't found if it's not present. Before attaching the text though, we have to make sure that the previous message is removed, so that the new one doesn't overlap with it. The `Axes.texts` property is a list of all the `Text` instances on an `Axes`. Since the `Axes` already has a `Text` instance, belonging to our `TextBox`, we don't want to remove anything if there are 2 or less `Text` instances present - the error message and the `TextBox` label.

If above two, we've already got an error message, which should be removed.

If the feature *is* present in the `DataFrame`, though, we clear the `Axes` and plot it:





Adding Span Selectors

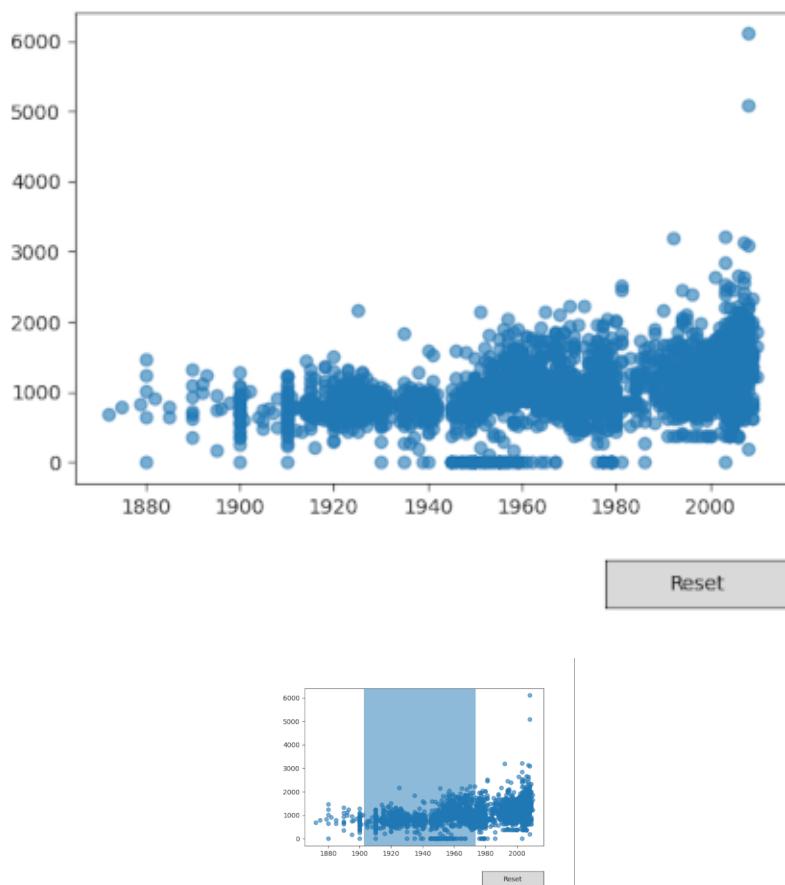
Span Selectors can be used to allow the user to select a span of data and focus on it, setting the axis limits based on that selection. By default, many libraries support this functionality, though unfortunately, Matplotlib doesn't and we'll have to do this manually. Additionally, we'll have to add an extra "*Reset*" button if we want to *zoom out* as well.

To add a *Span Selector*, we don't need to dedicate an entire new `Axes` for it - we can attach it to an existing one, which makes a lot of sense. When generating a `SpanSelector`, we supply the `Axes` it belongs to, as well as the event handler, followed by '`horizontal`' or '`vertical`', which rotates the `Axes` and *Span Selector both*.

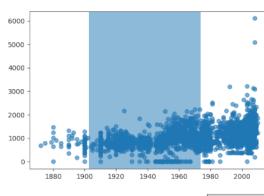
The `useblit` argument is typically set to `True` because it enhances performance on most backends. Additionally, we've added a few styling properties, such as setting the `alpha` of the rectangle created as a *Span Selector* to `0.5` and the `facecolor` to a nice `tab:blue`:

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from matplotlib.widgets import SpanSelector
5 from matplotlib.widgets import Button
6
7 fig, ax = plt.subplots()
8 fig.subplots_adjust(bottom=0.2)
9
10 df = pd.read_csv('AmesHousing.csv')
11
12 ax.scatter(x = df['Year Built'], y = df['Total Bsmt SF'], alpha = 0.6)
13
14 class EventHandler:
15     def select_horizontal(x, y):
16         ax.set_xlim(x, y)
17         plt.draw()
18
19     def reset(self):
20         ax.set_xlim(df['Year Built'].min(), df['Year Built'].max())
21         plt.draw()
22
23 span_horizontal = SpanSelector(ax, EventHandler.select_horizontal,
24                                 'horizontal', useblit=True,
25                                 rectprops=dict(alpha=0.5,
26                                               facecolor='tab:blue'))
27
28 button_ax = plt.axes([0.7, 0.02, 0.2, 0.07])
29 button = Button(button_ax, 'Reset')
30 button.on_clicked(EventHandler.reset)
31
32 plt.show()
```

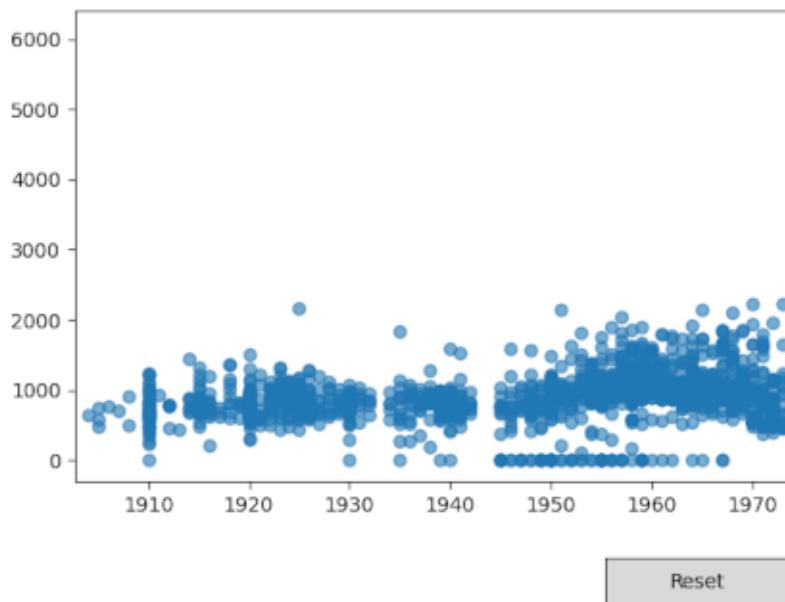
Running this generates a plot on which we can select spans and zoom in on them by setting the Axes-limits to the provided values:



Reset



Reset



Adding Sliders

Sliders allow users to select between many values intuitively by sliding a marker and selecting a value. Typically, sliders are used to continuously update some value on a plot, such as its range *or* even a feature. For example, you can adjust the value of a constant through a slider, which in turn affects a function that relies on that constant.

Let's write a script that allows us to change the Y and X-axis limits, through a slider, which will let us change the perspective from which we're viewing our data:

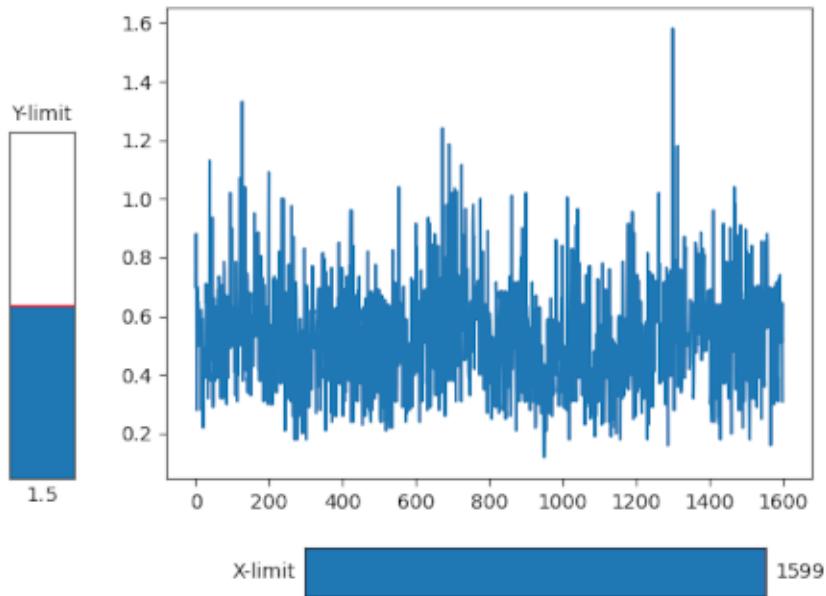
```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from matplotlib.widgets import Slider
4
5 fig, ax = plt.subplots()
6 fig.subplots_adjust(bottom=0.2, left=0.2)
7
8 df = pd.read_csv('winequality-red.csv')
9 plot, = ax.plot(df['volatile acidity'])
10
11 class EventHandler:
12     def update(val):
13         ax.set_ylim(0, yslider.val)
14         ax.set_xlim(0, xslider.val)
15         plt.draw()
16
17 xslider_ax = plt.axes([0.35, 0.03, 0.5, 0.07])
18 xslider = Slider(
19     ax=xslider_ax,
20     label="X-limit",
21     valmin=0,
22     valmax=len(df['volatile acidity']),
23     valinit=len(df['volatile acidity']),
24     orientation="horizontal"
25 )
26
27 yslider_ax = plt.axes([0.03, 0.2, 0.07, 0.5])
28 yslider = Slider(
29     ax=yslider_ax,
30     label="Y-limit",
31     valmin=0,
32     valmax=3,
33     valinit=1.5,
34     orientation="vertical"
35 )
36
37 xslider.on_changed(EventHandler.update)
38 yslider.on_changed(EventHandler.update)
39
40 plt.show()
```

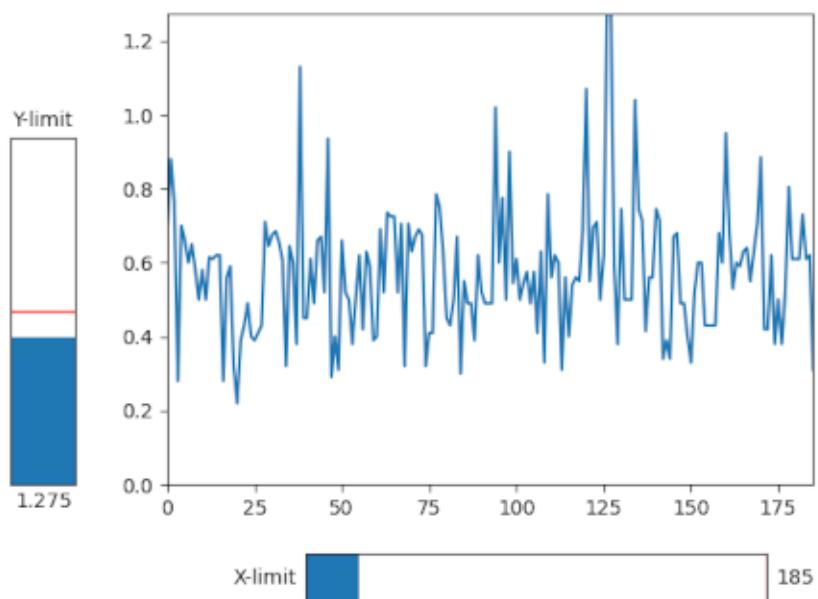
We've adjusted the padding to allow for a slider on the left and bottom of the Axes, and plotted a simple Line Plot. Adding a slider requires us to make an Axes for it, like with most other widgets, and assign it to the `ax` argument of the `Slider` through the constructor. Additionally, we can set the minimum, maximum and initial values of the slider. These will typically be dynamic ranges, based on the data you're plotting, but can also be manually set scalar values.

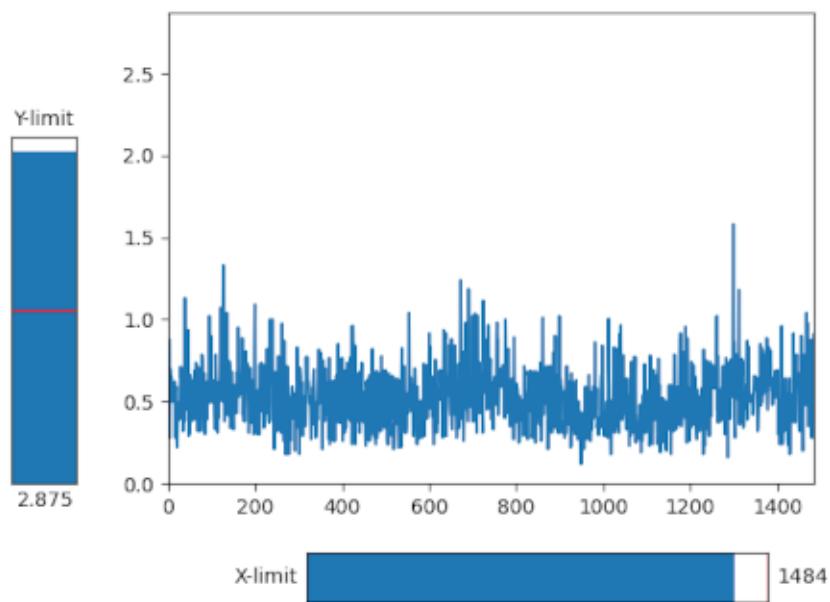
Finally, Sliders can be oriented horizontally or vertically. Since they're meant to be continually updated via a mouse-swipe - the `on_changed()` function is used to trigger a response when a user gives input. We've tweaked the `EventHandler` class with an `update()` function that simply adjusts the values of the X and Y-limits based on the

value of the respective sliders.

Running this code will produce a plot with two sliders, that we can use to change the scope of the Axes:







Thank You for Supporting Online Education

At StackAbuse, we believe that learning is not a one-stop time investment. It's *life-long*. Especially in the volatile and rapidly changing world of Computer Science and Software Engineering. So, we've pledged to update our eBooks, guides, and other upcoming material to keep the pace of progress in the field. Software is updating - it's only fitting that learning resources are updating as well. We'll issue updates to this eBook with Matplotlib updates, as well as new examples and fun mini-projects, like the section on exploring EEG data.

Thank you for purchasing *Data Visualization in Python with Pandas and Matplotlib!* We hope that it has brought a ton of value to you so far, and know that it will continue to do so as you dive further in to this topic.

Now, we'd like to ask you to **get involved** in improving the next version of the book and our *Data Visualization Bundle*.

We believe that high-quality resources and education is **community-driven** and that minor (or major) contributions from each member results in a wonderful learning oasis. For this, **feedback is crucial**.

Is there something you specifically liked in the book? Or more importantly, is there something you disliked? Did you feel like some examples could've been more detailed? Was something lacking? Were the code explanations clear and understandable? Do you feel more confident with these tools than before? Have you had the chance to use new Data Visualization skills for a project?

Any and all feedback is taken into consideration, and we're looking to incorporate everything we can into the next versions, which will, as usual, be pushed out for free to everyone who got their digital copy of the books. Just as importantly, it'll be there and ready for everyone who is yet to get their digital copies.

We'd really appreciate **an honest review on our Gumroad listing⁷⁷**! It helps spread the word and publishers like us to continue serving the community. If you have purchased this book as a *paperback* on Amazon, an honest review there means the world to publishers like us, and allows us to keep producing more content like this, as well as update already existing content.

Thank you for supporting StackAbuse and online education!

P.S. - Do you have an audience of Python developers, friends, or coworkers that may find this book or our bundle useful? Consider becoming an affiliate! We'll share 40% of each sale with you. Contact us to find out more.

⁷⁷<https://gumroad.com/l/data-visualization-in-python-book-bundle>