Matplotlib

Matplotlib is a powerful **Python library for data visualization**, widely used for creating **line, bar, and scatter plots**. In this tutorial, we will use **2022 stock market data** to explore trends and patterns, helping you analyze and communicate insights effectively.

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. Developed by John D. Hunter, it provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits. Matplotlib is widely used for data visualization in scientific computing, engineering, and data analysis. It offers a variety of plots and charts, including line plots, scatter plots, bar charts, histograms, and more. The library is highly customizable, allowing users to adjust colors, fonts, and other aesthetic elements to create publication-quality graphics.

Matplotlib is a powerful and versatile Python library for creating a wide range of static, animated, and interactive visualizations. Its flexibility and extensive customization options make it a preferred choice for data scientists and analysts aiming to present data insights effectively.

Key Features of Matplotlib:

- **Diverse Plot Types:** Matplotlib supports various plot types, including line plots, scatter plots, bar charts, histograms, pie charts, and 3D plots, catering to a broad spectrum of data visualization needs.
- **High Customizability:** Users can tailor almost every aspect of a plot, such as colors, labels, line styles, and fonts, to create publication-quality visuals that align with specific requirements.
- **Integration Capabilities:** Matplotlib integrates seamlessly with other Python libraries like NumPy and Pandas, enhancing its functionality for numerical computations and data manipulation.

How To Install Matplotlib

To install Matplotlib, you can use the Python package manager, pip. Open your terminal or command prompt and run the following command:

```
pip install matplotlib
```

If you're using the Anaconda distribution, you can install Matplotlib using the conda package manager:

```
conda install matplotlib
```

After installation, you can verify it by importing Matplotlib in a Python script:

```
import matplotlib

print(matplotlib.__version__)
```

This will display the installed version of Matplotlib, confirming a successful installation

Why Use Matplotlib?

Matplotlib allows for **detailed and highly customizable** visualizations. You can modify **colors, markers, labels, titles, axes, legends, and grid styles** to suit your needs. It is widely used in **data analysis, research, and reporting** due to its flexibility in generating publication-quality charts.

Prerequisites

Basic knowledge of **NumPy arrays** and **pandas DataFrames** is recommended, as Matplotlib works seamlessly with these data structures. We will provide brief explanations when needed.

What You'll Learn

By the end of this tutorial, you'll be able to:

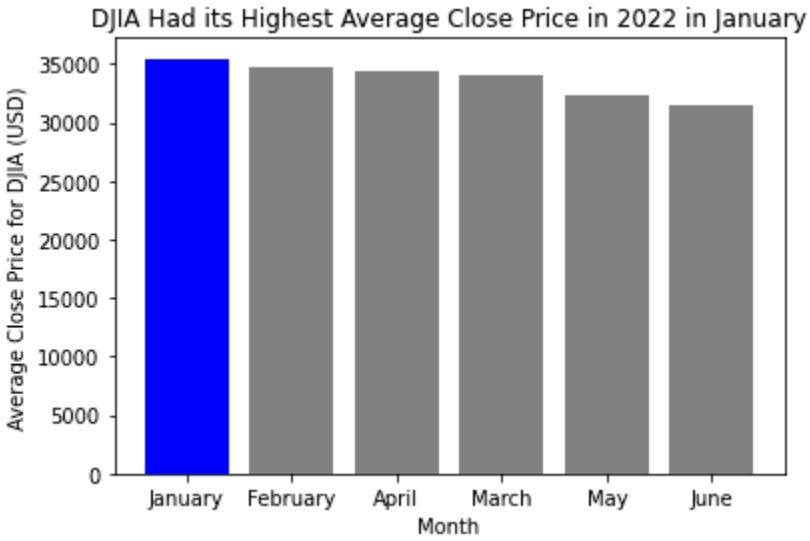
✓ Create **line, bar, and scatter plots** using Matplotlib

✓ Customize plots with **titles, labels, colors, and styles**

✓ Enhance visualizations for **better data storytelling**

Matplotlib Examples

By the end of this tutorial, you will be able to make great-looking visualizations in Matplotlib. We will focus on creating line plots, bar plots, and scatter plots. We will also focus on how to make customization decisions, such as the use of color, how to label plots, and how to organize them in a clear way to tell a compelling story.



The Dataset

Matplotlib seamlessly integrates with **NumPy arrays** and **pandas DataFrames**, making it easy to visualize tabular data. In this tutorial, we will use historical price data from the **Dow Jones Industrial Average (DJIA)** for the period **January 1, 2022 – December 31, 2022**.

To get the dataset:

1. Visit the DJIA historical data page.
2. Set the date range to **2022-01-01 to 2022-12-31**.
3. Click on the **"Download Spreadsheet"** button to save the file.

We will use the **pandas** library to load the CSV file (HistoricalPrices.csv) and display the first few rows using the .head() method:

```
import pandas as pd

djia_data = pd.read_csv('HistoricalPrices.csv')
djia_data.head()
```

Output:

	Date	Open	High	Low	Close
0	12/30/22	33121.61	33152.55	32847.82	33147.25
1	12/29/22	33021.43	33293.42	33020.35	33220.80
2	12/28/22	33264.76	33379.55	32869.15	32875.71
3	12/27/22	33224.23	33387.72	33069.58	33241.56
4	12/23/22	32961.06	33226.14	32814.02	33203.93

Analysis:

We see the data include 4 columns, a Date, Open, High, Low, and Close. The latter 4 are related to the price of the index during the trading day. Below is a brief explanation of each variable.

- **Date:** The day that the stock price information represents.
- **Open:** The price of the DJIA at 9:30 AM ET when the stock market opens.
- **High:** The highest price the DJIA reached during the day.
- **Low:** The lowest price the DJIA reached during the day.
- **Close:** The price of the DJIA when the market stopped trading at 4:00 PM ET.

As a quick clean up step, we will also need to use the rename() method in pandas as the dataset we downloaded has an extra space in the column names.

```
djia_data = djia_data.rename(columns = {' Open': 'Open', ' High': 'High', ' Low': 'Low', ' Close': 'Close'})
```

We will also ensure that the Date variable is a datetime variable and sort in ascending order by the date.

```
djia_data['Date'] = pd.to_datetime(djia_data['Date']) djia_data = djia_data.sort_values(by = 'Date')
```

Loading Matplotlib

Next, we will load the pyplot submodule of Matplotlib so that we can draw our plots. The pyplot module contains all of the relevant methods we will need to c

reate plots and style them. We will use the conventional alias plt. We will also load in pandas, numpy, and datetime for future parts of this tutorial.

```
import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

from datetime import datetime
```

Drawing Line Plots

The first plot we will create will be a line plot. Line plots are a very important plot type as they do a great job of displaying time series data. It is often important to visualize how KPIs change over time to understand patterns in data that can be actioned on.

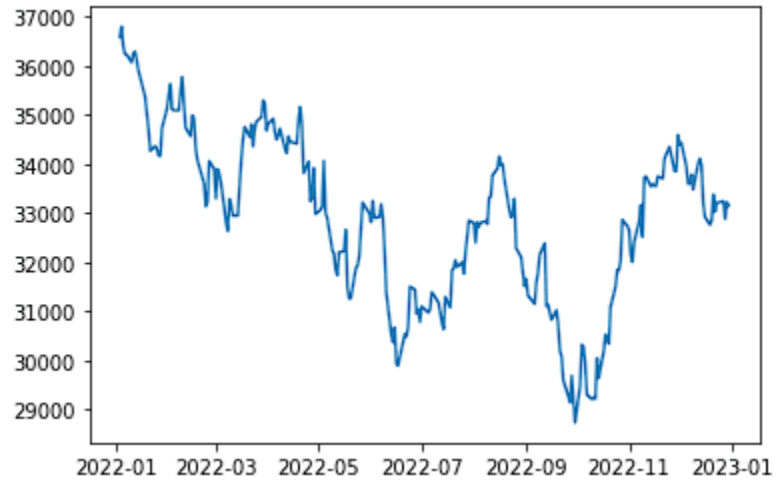
Line Plots with a Single Line

- Show how to draw a simple line plot with a single line.
    - Make sure to emphasize the use of plt.show() so the plot actually displays.
- Provide brief commentary on the plot, including interpretation.

We can create a line plot in matplotlib using the plt.plot() method where the first argument is the x variable and the second argument is the y variable in our line plot. Whenever we create a plot, we need to make sure to call plt.show() to ensure we see the graph we have created. We will visualize the close price over time of the DJIA.

```
plt.plot(djia_data['Date'], djia_data['Close'])

plt.show()
```

Output:



Over the year, the index price began at its peak, fluctuated throughout, and hit its lowest point around October before rebounding strongly toward the year's end.

Line Plots with Multiple Lines

We can visualize multiple lines on the same plot by adding another plt.plot() call before the plt.show() function.

```
plt.plot(djia_data['Date'], djia_data['Open']) plt.plot(djia_data['Date'], djia_data['Close'])

plt.show()
```

Output:

Throughout the year, the DJIA's open and close prices remained relatively close each day, with no consistent pattern of one being higher than the other.

Adding a Legend

If we want to distinguish which line represents which column, we can add a legend. This will create a color coded label in the corner of the graph. We can do this using plt.legend() and adding label parameters to each plt.plot() call.

```
plt.plot(djia_data['Date'], djia_data['Open'], label = 'Open')
plt.plot(djia_data['Date'], djia_data['Close'], label = 'Close')
plt.legend() plt.show()
```

Output:

A legend with the specified labels now appears in the default top-right location, which can be adjusted using the loc argument in plt.legend().

Drawing Bar Plots

Bar plots are useful for comparing numerical values across categories, making it easier to identify the largest and smallest groups.

In this section, we aggregate the data into monthly averages using pandas to compare the DJIA's performance in 2022. To enhance visualization, we focus on the first six months.

```
Import the calendar package

from calendar import month_name

Order by months by chronological order

djia_data['Month'] = pd.Categorical(djia_data['Date'].dt.month_name(),
month_name[1:])

Group metrics by monthly averages

djia_monthly_mean = djia_data

.groupby('Month')

.mean()

.reset_index()

djia_monthly_mean.head(6)
```

Output:

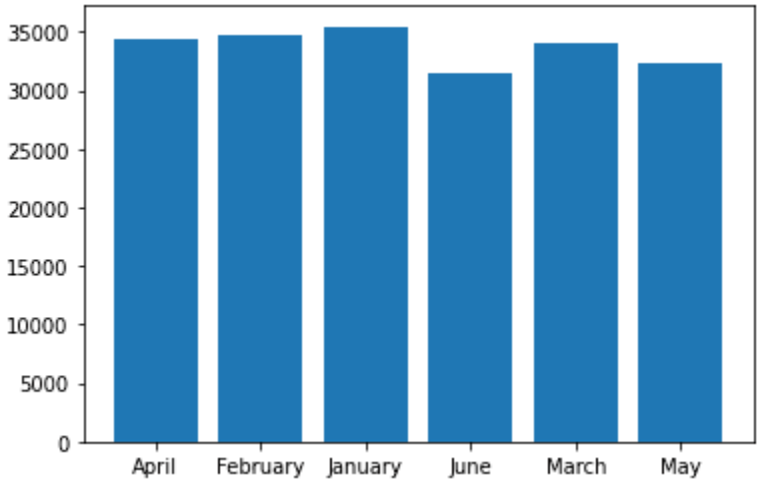
	Month	Open	High	Low	Close
0	January	35498.1305	35740.9105	35145.955	35456.145
1	February	34687.5163157895	34906.2042105263	34362.2436842105	34648.4805263158
2	March	34007.4982608696	34270.8908695652	33752.9634782609	34029.7404347826
3	April	34392.0945	34640.3675	34078.481	34314.99
4	May	32364.3271428571	32668.0280952381	31996.5128571429	32379.4628571429
5	June	31526.0333333333	31755.6371428571	31188.7880952381	31446.7128571429

Vertical Bar Plots

We will start by creating a bar chart with vertical bars. This can be done using the plt.bar() method with the first argument being the x-axis variable (Month) and the height parameter being the y-axis (Close). We then want to make sure to call plt.show() to show our plot.

```
plt.bar(djia_monthly_mean['Month'], height =
djia_monthly_mean['Close'])

plt.show()
```

Output



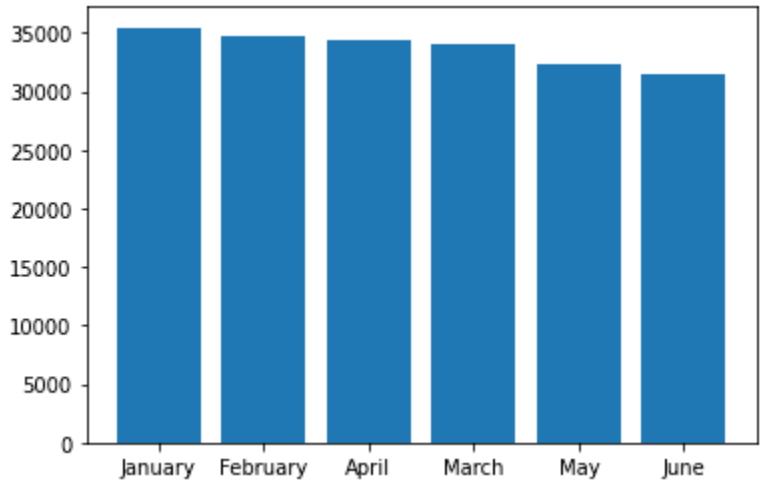
We see that most of the close prices of the DJIA were close to each other with the lowest average close value being in June and the highest average close value being in January.

Reordering Bars in Bar Plots

If we want to show these bars in order of highest to lowest Monthly average close price, we can sort the bars using the sort_values() method in pandas and then using the same plt.bar() method.

```
djia_monthly_mean_srtd = djia_monthly_mean.sort_values(by = 'Close', ascending = False)


plt.bar(djia_monthly_mean_srtd['Month'], height = djia_monthly_mean_srtd['Close'])
plt.show()
```

Output



As you can see, it is significantly easier to see which months had the highest average DJIA close price and which months had the lower averages. It is also easier to compare across months and rank the months.

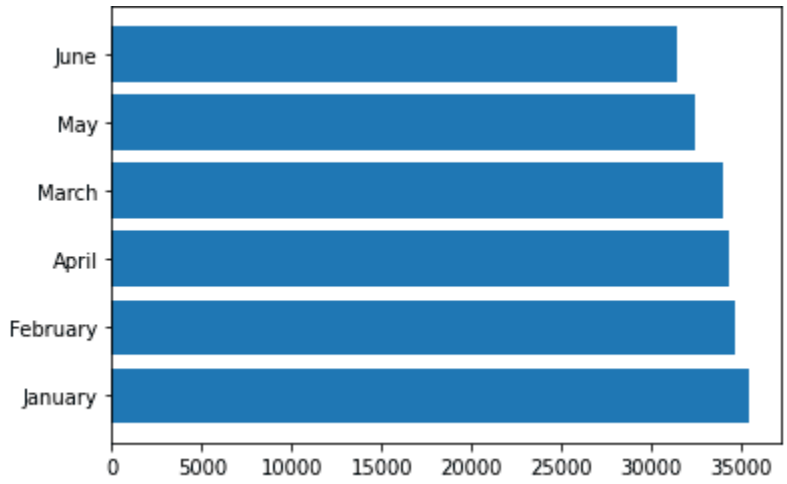
Horizontal Bar Plots

- Show how to swap the axes, so the bars are horizontal.
- Provide brief commentary on the plot, including interpretation.

It is sometimes easier to interpret bar charts and read the labels when we make the bar plot with horizontal bars. We can do this using the plt.hbar() method.

```
plt.barh(djia_monthly_mean_srtd['Month'], height = djia_monthly_mean_srtd['Close'])
plt.show()
```

Output:



As you can see, the labels of each category (month) are easier to read than when the bars were vertical. We can still easily compare across groups. This horizontal bar chart is especially useful when there are a lot of categories.

Drawing Scatter Plots

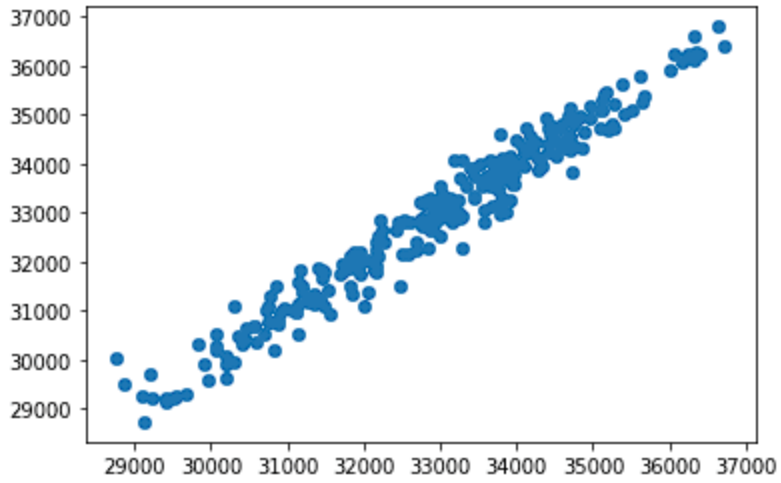
Scatterplots are very useful for identifying relationships between 2 numeric variables. This can give you a sense of what to expect in a variable when the other variable changes and can also be very informative in your decision to use different modeling techniques such as linear or non-linear regression.

Scatter Plots

Similar to the other plots, a scatter plot can be created using pyplot.scatter() where the first argument is the x-axis variable and the second argument is the y-axis variable. In this example, we will look at the relationship between the open and close price of the DJIA.

```
plt.scatter(djia_data['Open'], djia_data['Close'])

plt.show()
```

Output:



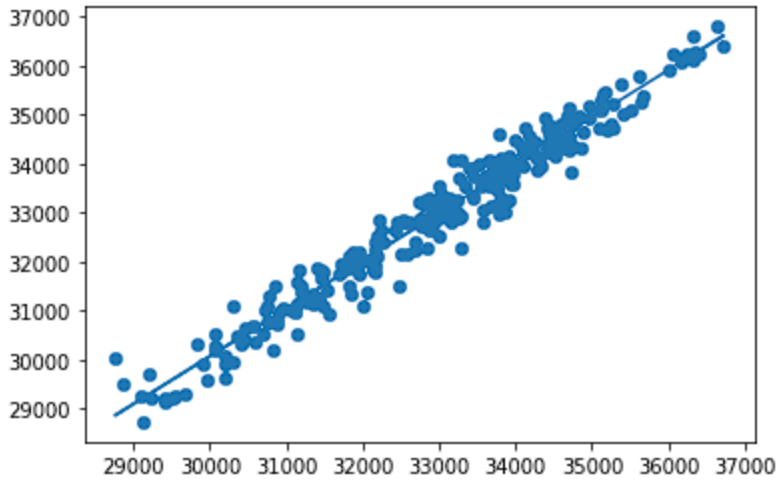
On the x-axis we have the open price of the DJIA and on the y-axis we have the close price. As we would expect, as the open price increases, we see a strong relationship in the close price increasing as well.

Scatter Plots with a Trend Line

Next, we will add a trend line to the graph to show the linear relationship between the open and close variables more explicitly. To do this, we will use the numpy polyfit() method and poly1d(). The first method will give us a least squares polynomial fit where the first argument is the x variable, the second variable is the y variable, and the third variable is the degrees of the fit (1 for linear). The second method will give us a one-dimensional polynomial class that we can use to create a trend line using plt.plot().

```
z = np.polyfit(djia_data['Open'], djia_data['Close'], 1)

p = np.poly1d(z)

plt.scatter(djia_data['Open'], djia_data['Close'])

plt.plot(djia_data['Open'], p(djia_data['Open']))

plt.show()
```

Output:



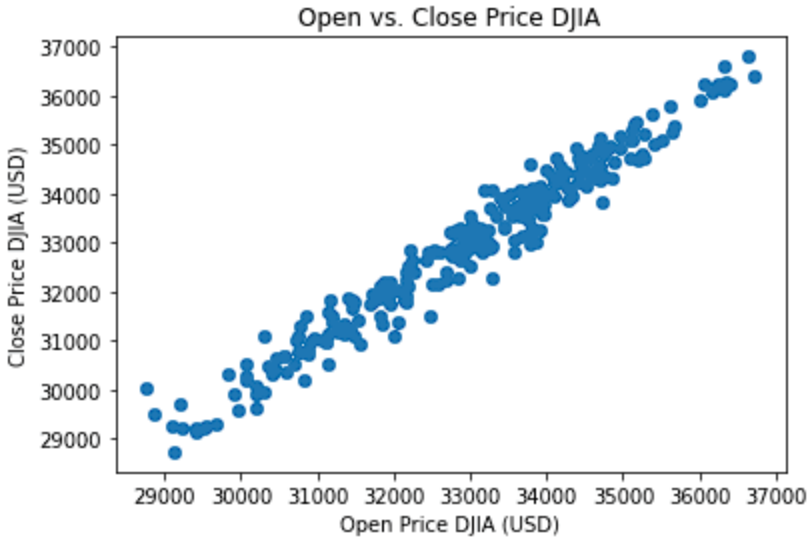
As we can see, the line in the background of the graph follows the trend of the scatterplot closely as the relationship between open and close price is strongly linear. We see that as the open price increases, the close price generally increases at a similar and linear rate.

Setting the Plot Title and Axis Labels

Plot titles and axis labels make it significantly easier to understand a visualization and allow the viewer to quickly understand what they are looking at more clearly. We can do this by adding more layers using plt.xtitle(), plt.ylabel() and plt.xlabel() which we will demonstrate with the scatterplot we made in the previous section.

```
plt.scatter(djia_data['Open'], djia_data['Close'])

plt.show()
```

Output:



Changing Colors

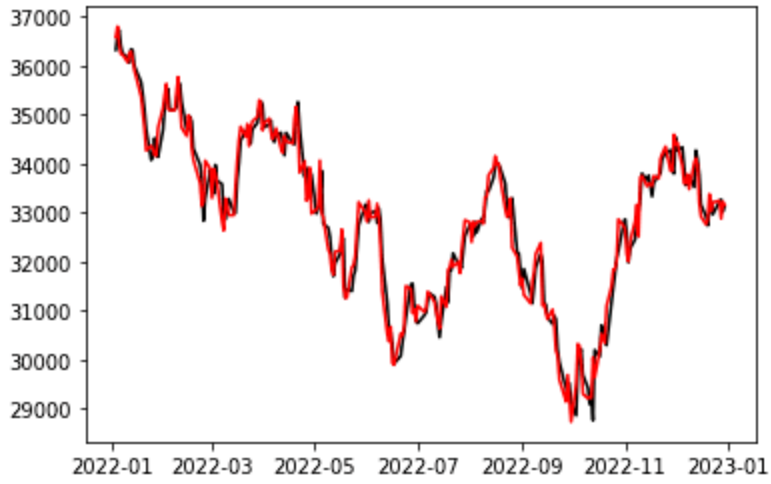
Color can be a powerful tool in data visualizations for emphasizing certain points or telling a consistent story with consistent colors for a certain idea. In Matplotlib, we can change colors using named colors (e.g. "red", "blue", etc.), hex code ("#f4db9a", "#383c4a", etc.), and red-green-blue tuples (e.g. (125, 100, 37), (30, 54, 121), etc.).

Lines

For a line plot, we can change a color using the color attribute in plt.plot(). Below, we change the color of our open price line to “black” and our close price line to “red.”

```
plt.plot(djia_data['Date'], djia_data['Open'], color = 'black')

plt.plot(djia_data['Date'], djia_data['Close'], color = 'red')

plt.show()
```

Output:

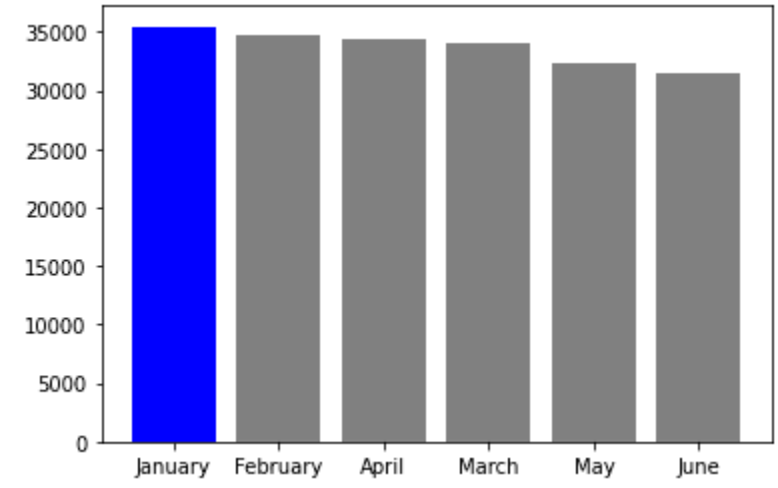


Bars

For bars, we can pass a list into the color attribute to specify the color of each line. Let's say we want to highlight the average price in January for a point we are trying to make about how strong the average close price was. We can do this by giving that bar a unique color to draw attention to it.

```
plt.bar(djia_monthly_mean_srtd['Month'], height =
djia_monthly_mean_srtd['Close'], color = ['blue', 'gray', 'gray', 'gray', 'gray',
'gray'])

plt.show()
```

Output:

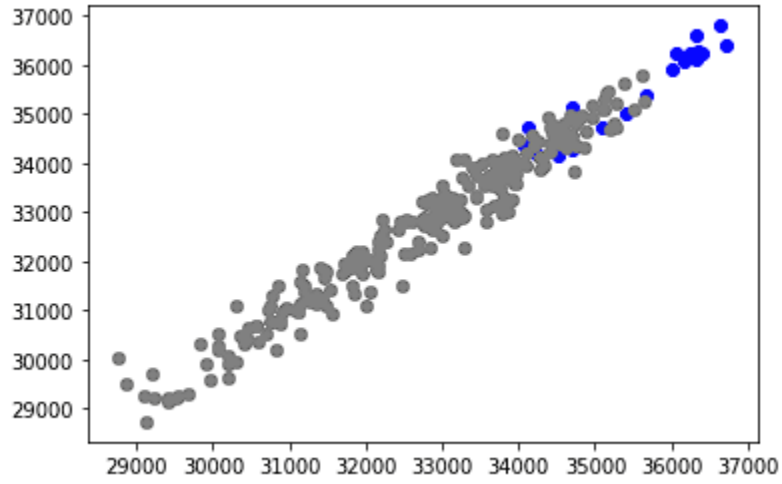


Points

Finally, for scatter plots, we can change the color using the color attribute of plt.scatter(). We will color all points in January as blue and all other points as gray to show a similar story as in the above visualization.

```
plt.scatter(djia_data[djia_data['Month'] == 'January']['Open'],
djia_data[djia_data['Month'] == 'January']['Close'], color = 'blue')

plt.scatter(djia_data[djia_data['Month'] != 'January']['Open'],
djia_data[djia_data['Month'] != 'January']['Close'], color = 'gray')

plt.show()
```

Output:



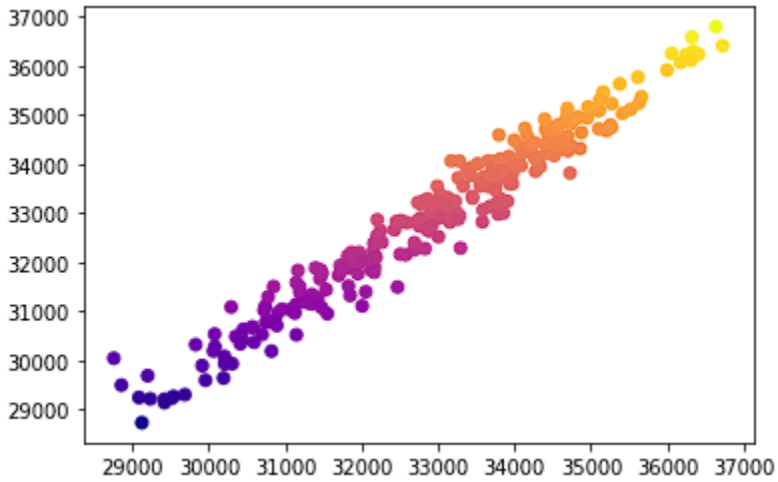
Using Colormaps

Colormaps are built-in Matplotlib colors that scale based on the magnitude of the value (**documentation here**). The colormaps generally aesthetically look good together and help tell a story in the increasing values.

We see in the below example, we use a colormap by passing the close price (y-variable) to the c attribute, and the plasma colormap through cmap. We see that as the values increase, the associated color gets brighter and more yellow while the lower end of the values is purple and darker.

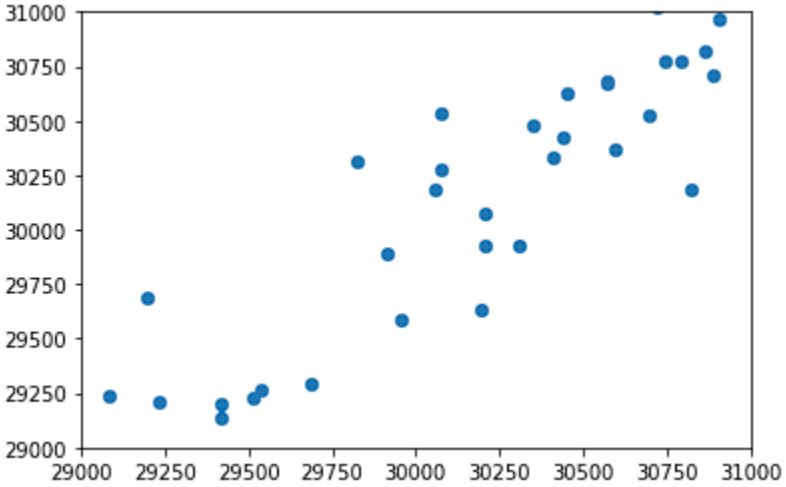
```
plt.scatter(djia_data['Open'], djia_data['Close'], c=djia_data['Close'], cmap = plt.cm.plasma)

plt.show()
```

Output:



Setting Axis Limits

Sometimes, it is helpful to look at a specific range of values in a plot. For example, if the DJIA is currently trading around $30,000, we may only care about behavior around that price. We can pass a tuple into the plt.xlim() and plt.ylim() to set x and y limits respectively. The first value in the tuple is the lower limit, and the second value in the tuple is the upper limit.



Saving Plots

Finally, we can save plots that we create in matplotlib using the plt.savefig() method. We can save the file in many different file formats including ‘png,’ ‘pdf,’ and ‘svg’. The first argument is the filename. The format is inferred from the file extension (or you can override this with the format argument).

```
plt.scatter(djia_data['Open'], djia_data['Close'])

plt.savefig('DJIA 2022 Scatterplot Open vs. Close.png')
```

Pandas

Pandas is a data manipulation package in Python for tabular data. That is, data in the form of rows and columns, also known as DataFrames. Intuitively, you can think of a DataFrame as an Excel sheet.

pandas' functionality includes data transformations, like sorting rows and taking subsets, to calculating summary statistics such as the mean, reshaping DataFrames, and joining DataFrames together. pandas works well with other popular Python data science packages, often called the PyData ecosystem, including

- NumPy for numerical computing
- Matplotlib, Seaborn, Plotly and other data visualization packages
- Scikit-Learn for machine learning

What is pandas used for?

pandas is used throughout the data analysis workflow. With pandas, you can:

- Import datasets from databases, spreadsheets, comma-separated values (CSV) files, and more.
- Clean datasets, for example, by dealing with missing values.
- Tidy datasets by reshaping their structure into a suitable format for analysis.
- Aggregate data by calculating summary statistics such as the mean of columns, correlation between them, and more.
- Visualize datasets and uncover insights.

pandas also contains functionality for time series analysis and analyzing text data.

Key benefits of the pandas package

Undoubtedly, pandas is a powerful data manipulation tool packaged with several benefits, including:

- Made for Python: Python is the world's most popular language for machine learning and data science.
- Less verbose per unit operations: Code written in pandas is less verbose, requiring fewer lines of code to get the desired output.
- Intuitive view of data: pandas offers exceptionally intuitive data representation that facilitates easier data understanding and analysis.
- Extensive feature set: It supports an extensive set of operations from exploratory data analysis, dealing with missing values, calculating statistics, visualizing univariate and bivariate data, and much more.

- Works with large data: pandas handles large data sets with ease. It offers speed and efficiency while working with datasets of the order of millions of records and hundreds of columns, depending on the machine.

Install pandas

Installing pandas is straightforward; just use the pip install command in your terminal.

```
pip install pandas
```

After installing pandas, it's good practice to check the installed version to ensure everything is working correctly:

```
import pandas as pd
print(pd.__version__)  # Prints the pandas version
```

This confirms that pandas is installed correctly and lets you verify compatibility with other packages.

Importing data in pandas

To begin working with pandas, import the pandas Python package as shown below. When importing pandas, the most common alias for pandas is pd

```
import pandas as pd
```

Importing CSV files

Use read_csv() with the path to the CSV file to read a comma-separated values file

```
df = pd.read_csv("diabetes.csv")
```

This read operation loads the CSV file diabetes.csv to generate a pandas Dataframe object df. Throughout this tutorial, you'll see how to manipulate such DataFrame objects.

Importing text files

Reading text files is similar to CSV files. The only nuance is that you need to specify a separator with the sep argument, as shown below. The separator argument refers to the symbol used to separate rows in a DataFrame. Comma (sep = ","), whitespace(sep = "\s"), tab (sep = "\t"), and colon(sep = ":") are the commonly used separators. Here \s represents a single white space character.

```
df = pd.read_csv("diabetes.txt", sep="\s")
```

Importing Excel files (single sheet)

Reading excel files (both XLS and XLSX) is as easy as the read_excel() function, using the file path as an input.

```
df = pd.read_excel('diabetes.xlsx')
```

Importing data from SQL databases

To load data from a relational database, use pd.read_sql() along with a database connection.

```
import sqlite3

# Establish a connection to an SQLite database
conn = sqlite3.connect("my_database.db")

# Read data from a table
df = pd.read_sql("SELECT * FROM my_table", conn)
```

Outputting data in pandas

Just as pandas can import data from various file types, it also allows you to export data into various formats. This happens especially when data is transformed using pandas and needs to be saved locally on your machine. Below is how to output pandas DataFrames into various formats.

A pandas DataFrame (here we are using df) is saved as a CSV file using the .to_csv() method. The arguments include the filename with path and index – where index = True implies writing the DataFrame's index.

```
df.to_csv("diabetes_out.csv", index=False)
```

Outputting a DataFrame into a text file

As with writing DataFrames to CSV files, you can call .to_csv(). The only differences are that the output file format is in .txt, and you need to specify a separator using the sep argument.

```
df.to_csv('diabetes_out.txt', header=df.columns, index=None, sep=' ')
```

Outputting a DataFrame into an Excel file

Call .to_excel() from the DataFrame object to save it as a ".xls" or ".xlsx" file.

```
df.to_excel("diabetes_out.xlsx", index=False)
```

Viewing and understanding DataFrames using pandas

After reading tabular data as a DataFrame, you would need to have a glimpse of the data. You can either view a small sample of the dataset or a summary of the data in the form of summary statistics.

How to view data using .head() and .tail()

You can view the first few or last few rows of a DataFrame using the .head() or .tail() methods, respectively. You can specify the number of rows through the n argument (the default value is 5).

```
df.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

First five rows of the DataFrame

```
df.tail(n = 10)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
758	1	106	76	0	0	37.5	0.197	26	0
759	6	190	92	0	0	35.5	0.278	66	1
760	2	88	58	26	16	28.4	0.766	22	0
761	9	170	74	31	0	44.0	0.403	43	1
762	9	89	62	0	0	22.5	0.142	33	0
763	10	101	76	48	180	32.9	0.171	63	0
764	2	122	70	27	0	36.8	0.340	27	0
765	5	121	72	23	112	26.2	0.245	30	0
766	1	126	60	0	0	30.1	0.349	47	1
767	1	93	70	31	0	30.4	0.315	23	0

First 10 rows of the DataFrame

Understanding data using .describe()

The .describe() method prints the summary statistics of all numeric columns, such as count, mean, standard deviation, range, and quartiles of numeric columns.

```
df.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Get summary statistics with .describe()

It gives a quick look at the scale, skew, and range of numeric data.

You can also modify the quartiles using the percentiles argument. Here, for example, we're looking at the 30%, 50%, and 70% percentiles of the numeric columns in DataFrame df.

```
df.describe(percentiles=[0.3, 0.5, 0.7])
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
30%	1.000000	102.000000	64.000000	8.200000	0.000000	28.200000	0.259000	25.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
70%	5.000000	134.000000	78.000000	31.000000	106.000000	35.490000	0.563700	38.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Get summary statistics with specific percentiles

You can also isolate specific data types in your summary output by using the include argument. Here, for example, we're only summarizing the columns with the integer data type.

```
df.describe(include=[int])
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	81.000000	1.000000

Get summary statistics of integer columns only

Understanding data using .info()

The .info() method is a quick way to look at the data types, missing values, and data size of a DataFrame. Here, we're setting the show_counts argument to True, which gives a few over the total non-missing values in each column. We're also setting memory_usage to True, which

shows the total memory usage of the DataFrame elements. When verbose is set to True, it prints the full summary from .info().

```
df.info(show_counts=True, memory_usage=True, verbose=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               768 non-null    int64  
 1   Glucose                   768 non-null    int64  
 2   BloodPressure             768 non-null    int64  
 3   SkinThickness             768 non-null    int64  
 4   Insulin                   768 non-null    int64  
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64  
 8   Outcome                   768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

Understanding your data using .shape

The number of rows and columns of a DataFrame can be identified using the .shape attribute of the DataFrame. It returns a tuple (row, column) and can be indexed to get only rows, and only columns count as output.

```
df.shape # Get the number of rows and columns
df.shape[0] # Get the number of rows only
df.shape[1] # Get the number of columns only
```

```
(768,9)
768
9
```

Get all columns and column names

Calling the .columns attribute of a DataFrame object returns the column names in the form of an Index object. As a reminder, a pandas index is the address/label of the row or column.

```
df.columns
```

```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

It can be converted to a list using a list() function.

```
list(df.columns)
```

```
['Pregnancies',
 'Glucose',
 'BloodPressure',
 'SkinThickness',
 'Insulin',
 'BMI',
 'DiabetesPedigreeFunction',
 'Age',
 'Outcome']
```

Checking for missing values in pandas with .isnull()

The sample DataFrame does not have any missing values. Let's introduce a few to make things interesting. The .copy() method makes a copy of the original DataFrame. This is done to ensure that any changes to the copy don't reflect in the original DataFrame. Using .loc (to be discussed later), you can set rows two to five of the Pregnancies column to NaN values, which denote missing values.

```
df2 = df.copy()
df2.loc[2:5,'Pregnancies'] = None
df2.head(7)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6.0	148	72	35	0	33.6	0.627	50	1
1	1.0	85	66	29	0	26.6	0.351	31	0
2	NaN	183	64	0	0	23.3	0.672	32	1
3	NaN	89	66	23	94	28.1	0.167	21	0
4	NaN	137	40	35	168	43.1	2.288	33	1
5	NaN	116	74	0	0	25.6	0.201	30	0
6	3.0	78	50	32	88	31.0	0.248	26	1

You can see, that now rows 2 to 5 are NaN

You can check whether each element in a DataFrame is missing using the .isnull() method.

```
df2.isnull().head(7)
```

Given it's often more useful to know how much missing data you have, you can combine .isnull() with .sum() to count the number of nulls in each column.

```
df2.isnull().sum()
```

```
Pregnancies                 4
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

You can also do a double sum to get the total number of nulls in the DataFrame.

```
df2.isnull().sum().sum()
```

Sorting, Slicing and Extracting Data in pandas

The pandas package offers several ways to sort, subset, filter, and isolate data in your DataFrames. Here, we'll see the most common ways.

Sorting data

To sort a DataFrame by a specific column:

```
df.sort_values(by="Age", ascending=False, inplace=True)
```

You can sort by multiple columns:

```
df.sort_values(by=["Age", "Glucose"], ascending=[False, True], inplace=True
```

Resetting the index

If you filter or sort a DataFrame, your index might become misaligned. Use .reset_index() to fix this:

```
df.reset_index(drop=True, inplace=True)
```

Filtering data using conditions

To extract data based on a condition:

```
df[df["BloodPressure"] > 100]
```

Isolating one column using [ ]

You can isolate a single column using a square bracket [ ] with a column name in it. The output is a pandas Series object. A pandas Series is a one-dimensional array containing data of any type, including integer, float, string, boolean, python objects, etc. A DataFrame is comprised of many series that act as columns.

```
df['Outcome']
```

```
0      1
1      0
2      1
3      0
4      1
      ..
763    0
764    0
765    0
766    1
767    0
Name: Outcome, Length: 768, dtype: int64
```

Isolating one column in pandas

Isolating two or more columns using [[ ]]

You can also provide a list of column names inside the square brackets to fetch more than one column. Here, square brackets are used in two different ways. We use the outer square brackets to indicate a subset of a DataFrame, and the inner square brackets to create a list.

```
df[['Pregnancies', 'Outcome']]
```

	Pregnancies	Outcome
0	6	1
1	1	0
2	8	1
3	1	0
4	0	1
...	...	...
763	10	0
764	2	0
765	5	0
766	1	1
767	1	0

768 rows × 2 columns

Isolating two columns in pandas

Isolating one row using [ ]

A single row can be fetched by passing in a boolean series with one True value. In the example below, the second row with index = 1 is returned. Here, .index returns the row labels of the DataFrame, and the comparison turns that into a Boolean one-dimensional array.

```
df[df.index==1]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
1	1	85	66	29	0	26.6	0.351	31	0

Isolating one row in pandas

Isolating two or more rows using [ ]

Similarly, two or more rows can be returned using the .isin() method instead of a == operator.

```
df[df.index.isin(range(2,10))]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	0	0	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	0	0	0	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	0	0	0.0	0.232	54	1

Isolating specific rows in pandas

Using .loc[] and .iloc[] to fetch rows

You can fetch specific rows by labels or conditions using .loc[] and .iloc[] ("location" and "integer location"). .loc[] uses a label to point to a row, column or cell, whereas .iloc[] uses the numeric position. To understand the difference between the two, let's modify the index of df2 created earlier.

```
df2.index = range(1,769)
```

The below example returns a pandas Series instead of a DataFrame. The 1 represents the row index (label), whereas the 1 in. iloc[] is the row position (first row).

```
df2.loc[1]
```

```
Pregnancies                 6.000
Glucose                   148.000
BloodPressure              72.000
SkinThickness              35.000
Insulin                     0.000
BMI                        33.600
DiabetesPedigreeFunction    0.627
Age                        50.000
Outcome                     1.000
Name: 1, dtype: float64
```

```
df2.iloc[1]
```

```
Pregnancies                 1.000
Glucose                    85.000
BloodPressure              66.000
SkinThickness              29.000
Insulin                     0.000
BMI                        26.600
DiabetesPedigreeFunction    0.351
Age                        31.000
Outcome                     0.000
Name: 2, dtype: float64
```

You can also fetch multiple rows by providing a range in square brackets.

```
df2.loc[100:110]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
100	1.0	122	90	51	220	49.7	0.325	31	1
101	1.0	163	72	0	0	39.0	1.222	33	1
102	1.0	151	60	0	0	26.1	0.179	22	0
103	0.0	125	96	0	0	22.5	0.262	21	0
104	1.0	81	72	18	40	26.6	0.283	24	0
105	2.0	85	65	0	0	39.6	0.930	27	0
106	1.0	126	56	29	152	28.7	0.801	21	0
107	1.0	96	122	0	0	22.4	0.207	27	0
108	4.0	144	58	28	140	29.5	0.287	37	0
109	3.0	83	58	31	18	34.3	0.336	25	0
110	0.0	95	85	25	36	37.4	0.247	24	1

Isolating rows in pandas with .loc[]

```
df2.iloc[100:110]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
101	1.0	163	72	0	0	39.0	1.222	33	1
102	1.0	151	60	0	0	26.1	0.179	22	0
103	0.0	125	96	0	0	22.5	0.262	21	0
104	1.0	81	72	18	40	26.6	0.283	24	0
105	2.0	85	65	0	0	39.6	0.930	27	0
106	1.0	126	56	29	152	28.7	0.801	21	0
107	1.0	96	122	0	0	22.4	0.207	27	0
108	4.0	144	58	28	140	29.5	0.287	37	0
109	3.0	83	58	31	18	34.3	0.336	25	0
110	0.0	95	85	25	36	37.4	0.247	24	1

Isolating rows in pandas with .iloc[]

You can also subset with .loc[] and .iloc[] by using a list instead of a range.

```
df2.loc[[100, 200, 300]]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
100	1.0	122	90	51	220	49.7	0.325	31	1
200	4.0	148	60	27	318	30.9	0.150	29	1
300	8.0	112	72	0	0	23.6	0.840	58	0

Isolating rows using a list in pandas with .loc[]

You can update/modify certain values by using the assignment operator =

```
df2.loc[df['Age']==81, ['Age']] = 80
```

Conditional slicing (that fits certain conditions)

pandas lets you filter data by conditions over row/column values. For example, the below code selects the row where Blood Pressure is exactly 122. Here, we are isolating rows using the brackets [ ] as seen in previous sections. However, instead of inputting row indices or column names, we are inputting a condition where the column BloodPressure is equal to 122. We denote this condition using df.BloodPressure == 122.

```
df[df.BloodPressure == 122]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
106	1	96	122	0	0	22.4	0.207	27	0

Isolating rows based on a condition in pandas

The below example fetched all rows where Outcome is 1. Here df.Outcome selects that column, df.Outcome == 1 returns a Series of Boolean values determining which Outcomes are equal to 1, then [] takes a subset of df where that Boolean Series is True.

```
df[df.Outcome == 1]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
2	8	183	64	0	0	23.3	0.672	32	1
4	0	137	40	35	168	43.1	2.288	33	1
6	3	78	50	32	88	31.0	0.248	26	1
8	2	197	70	45	543	30.5	0.158	53	1
...	...	...	...	...	...	...	...	...	...
755	1	128	88	39	110	36.5	1.057	37	1
757	0	123	72	0	0	36.3	0.258	52	1
759	6	190	92	0	0	35.5	0.278	66	1
761	9	170	74	31	0	44.0	0.403	43	1
766	1	126	60	0	0	30.1	0.349	47	1

268 rows × 9 columns

Isolating rows based on a condition in pandas

You can use a > operator to draw comparisons. The below code fetches Pregnancies, Glucose, and BloodPressure for all records with BloodPressure greater than 100

```
df.loc[df['BloodPressure'] > 100, ['Pregnancies', 'Glucose', 'BloodPressure
```

	Pregnancies	Glucose	BloodPressure
43	9	171	110
84	5	137	108
106	1	96	122
177	0	129	110
207	5	162	104
362	5	103	108
369	1	133	102
440	0	189	104
549	4	189	110
658	11	127	106
662	8	167	106
672	10	68	106
691	13	158	114

Isolating rows and columns based on a condition in pandas

Cleaning data using pandas

Data cleaning is one of the most common tasks in data science. pandas lets you preprocess data for any use, including but not limited to training machine learning and deep learning models. Let’s use the DataFrame df2 from earlier, having four missing values, to illustrate a few data cleaning use cases. As a reminder, here's how you can see how many missing values are in a DataFrame.

```
df2.isnull().sum()
```

```
Pregnancies                 4
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Dealing with missing data technique #1: Dropping missing values

One way to deal with missing data is to drop it. This is particularly useful in cases where you have plenty of data and losing a small portion won't impact the downstream analysis. You can use a .dropna() method as shown below. Here, we are saving the results from .dropna() into a DataFrame df3.

```
df3 = df2.copy()
df3 = df3.dropna()
df3.shape
```

```
(764, 9) # this is 4 rows less than df2
```

The axis argument lets you specify whether you are dropping rows, or columns, with missing values. The default axis removes the rows containing NaNs. Use axis = 1 to remove the columns with one or more NaN values. Also, notice how we are using the argument inplace=True which lets you skip saving the output of .dropna() into a new DataFrame.

```
df3 = df2.copy()
df3.dropna(inplace=True, axis=1)
df3.head()
```

	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DPF	Age	Outcome	STF
0	148	72	35	0	33.6	0.627	50	1	0.700000
1	85	66	29	0	26.6	0.351	31	0	0.935484
2	183	64	0	0	23.3	0.672	32	1	0.000000
3	89	66	23	94	28.1	0.167	21	0	1.095238
4	137	40	35	168	43.1	2.288	33	1	1.060606

Dropping missing data in pandas

You can also drop both rows and columns with missing values by setting the how argument to 'all'

```
df3 = df2.copy()
df3.dropna(inplace=True, how='all')
```

Dealing with missing data technique #2: Replacing missing values

Instead of dropping, replacing missing values with a summary statistic or a specific value (depending on the use case) maybe the best way to go. For example, if there is one missing row from a temperature column denoting temperature throughout the days of the week, replacing that missing value with the average temperature of that week may be more effective than dropping values completely. You can replace the missing data with the row, or column mean using the code below.

```
df3 = df2.copy()
# Get the mean of Pregnancies
mean_value = df3['Pregnancies'].mean()
# Fill missing values using .fillna()
df3 = df3.fillna(mean_value)
```

Dealing with Duplicate Data

Let's add some duplicates to the original data to learn how to eliminate duplicates in a DataFrame. Here, we are using the .concat() method to concatenate the rows of the df2 DataFrame to the df2 DataFrame, adding perfect duplicates of every row in df2.

```
df3 = pd.concat([df2, df2])
df3.shape
```

```
(1536, 9)
```

You can remove all duplicate rows (default) from the DataFrame using .drop_duplicates() method.

```
df3 = df3.drop_duplicates()
df3.shape
```

```
(768, 9)
```

Renaming columns

A common data cleaning task is renaming columns. With the .rename() method, you can use columns as an argument to rename specific columns. The below code shows the dictionary for mapping old and new column names.

```
df3.rename(columns = {'DiabetesPedigreeFunction':'DPF'}, inplace = True)
df3.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DPF	Age	Outcome
0	6.0	148	72	35	0	33.6	0.627	50	1
1	1.0	85	66	29	0	26.6	0.351	31	0
2	8.0	183	64	0	0	23.3	0.672	32	1
3	1.0	89	66	23	94	28.1	0.167	21	0
4	0.0	137	40	35	168	43.1	2.288	33	1

Renaming columns in pandas

You can also directly assign column names as a list to the DataFrame.

```
df3.columns = ['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI'
df3.head()
```

	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
2	183	64	0	0	23.3	0.672	32	1
3	89	66	23	94	28.1	0.167	21	0
4	137	40	35	168	43.1	2.288	33	1

Renaming columns in pandas

Data analysis in pandas

The main value proposition of pandas lies in its quick data analysis functionality. In this section, we'll focus on a set of analysis techniques you can use in pandas.

Summary operators (mean, mode, median)

As you saw earlier, you can get the mean of each column value using the .mean() method.

```
df.mean()
```

```
Pregnancies                 3.845052
Glucose                   120.894531
BloodPressure              69.105469
SkinThickness              20.536458
Insulin                    79.799479
BMI                        31.992578
DiabetesPedigreeFunction    0.471876
Age                        33.240885
Outcome                     0.348958
dtype: float64
```

Printing the mean of columns in pandas

A mode can be computed similarly using the .mode() method.

```
df.mode()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	1.0	99	70.0	0.0	0.0	32.0	0.254	22.0	0.0
1	NaN	100	NaN	NaN	NaN	NaN	0.258	NaN	NaN

Printing the mode of columns in pandas

Similarly, the median of each column is computed with the .median() method

```
df.median()
```

```
Pregnancies                 3.0000
Glucose                   117.0000
BloodPressure              72.0000
SkinThickness              23.0000
Insulin                    30.5000
BMI                        32.0000
DiabetesPedigreeFunction    0.3725
Age                        29.0000
Outcome                     0.0000
dtype: float64
```

Printing the median of columns in pandas

Create new columns based on existing columns

pandas provides fast and efficient computation by combining two or more columns like scalar variables. The below code divides each value in the column Glucose with the corresponding value in the Insulin column to compute a new column named Glucose_Insulin_Ratio.

```
df2['Glucose_Insulin_Ratio'] = df2['Glucose']/df2['Insulin']
df2.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome	Glucose_Insulin_Ratio
1	6.0	148	72	35	0	33.6	0.627	50	1	inf
2	1.0	85	66	29	0	26.6	0.351	31	0	inf
3	NaN	183	64	0	0	23.3	0.672	32	1	inf
4	NaN	89	66	23	94	28.1	0.167	21	0	0.946809
5	NaN	137	40	35	168	43.1	2.288	33	1	0.815476

Create a new column from existing columns in pandas

Counting using .value_counts()

Often times you'll work with categorical values, and you'll want to count the number of observations each category has in a column. Category values can be counted using the .value_counts() methods. Here, for example, we are counting the number of observations where Outcome is diabetic (1) and the number of observations where the Outcome is non-diabetic (0).

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

```
0    500
1    268
Name: Outcome, dtype: int64
```

Using .value_counts() in pandas

Adding the normalize argument returns proportions instead of absolute counts.

```
df['Outcome'].value_counts(normalize=True)
```

```
0    0.651042
1    0.348958
Name: Outcome, dtype: float64
```

Using .value_counts() in pandas with normalization

Turn off automatic sorting of results using sort argument (True by default). The default sorting is based on the counts in descending order.

```
df['Outcome'].value_counts(sort=False)
```

```
1    268
0    500
Name: Outcome, dtype: int64
```

Using .value_counts() in pandas with sorting

You can also apply .value_counts() to a DataFrame object and specific columns within it instead of just a column. Here, for example, we are applying value_counts() on df with the subset argument, which takes in a list of columns.

```
df.value_counts(subset=['Pregnancies', 'Outcome'])
```

```
Pregnancies  Outcome
1            0          106
2            0           84
0            0           73
3            0           48
4            0           45
0            1           38
5            0           36
6            0           34
1            1           29
3            1           27
7            1           25
4            1           23
8            1           22
5            1           21
7            0           20
2            1           19
9            1           18
6            1           16
8            0           16
10           0           14
             1           10
9            0           10
11           1            7
12           0            5
13           0            5
             1            5
11           0            4
12           1            4
14           1            2
15           1            1
17           1            1
dtype: int64
```

Using .value_counts() in pandas while subsetting columns

Aggregating data with .groupby() in pandas

pandas lets you aggregate values by grouping them by specific column values. You can do that by combining the .groupby() method with a summary method of your choice. The below code displays the mean of each of the numeric columns grouped by Outcome.

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

Outcome	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	3.298000	109.980000	68.184000	19.664000	68.792000	30.304200	0.429734	31.190000
1	4.865672	141.257463	70.824627	22.164179	100.335821	35.142537	0.550500	37.067164

Aggregating data by one column in pandas

.groupby() enables grouping by more than one column by passing a list of column names, as shown below.

```
df.groupby(['Pregnancies', 'Outcome']).mean()
```

Pregnancies	Outcome	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
0	0	111.945205	69.205479	21.054795	77.561644	31.727397	0.457055	27.095890
	1	144.236842	63.210526	24.605263	89.578947	39.213158	0.643368	28.578947
1	0	104.254717	66.830189	23.047170	84.320755	29.616038	0.451679	25.254717
	1	143.793103	71.310345	29.517241	151.137931	37.793103	0.613759	35.103448
2	0	105.214286	61.940476	20.107143	72.619048	29.679762	0.479881	25.892857
	1	135.473684	69.052632	28.210526	144.315789	34.578947	0.543737	32.947368
3	0	109.604167	65.708333	17.520833	62.020833	29.231250	0.358354	28.770833
	1	148.444444	68.148148	24.629630	132.666667	32.548148	0.563333	29.481481
4	0	117.555556	71.577778	18.422222	78.466667	31.255556	0.410511	30.066667
	1	139.913043	67.000000	10.913043	51.782609	33.873913	0.516478	38.086957
5	0	111.666667	74.666667	17.166667	46.861111	31.100000	0.359278	39.416667

Aggregating data by two columns in pandas

Any summary method can be used alongside .groupby(),
including .min(), .max(), .mean(), .median(), .sum(), .mode(), and more.

Pivot tables

pandas also enables you to calculate summary statistics as pivot tables. This makes it easy to draw conclusions based on a combination of variables. The below code picks the rows as unique values of Pregnancies, the column values are the unique values of Outcome, and the cells contain the average value of BMI in the corresponding group.

For example, for Pregnancies = 5 and Outcome = 0, the average BMI turns out to be 31.1.

```
pd.pivot_table(df, values="BMI", index='Pregnancies',
               columns=['Outcome'], aggfunc=np.mean)
```

Outcome	0	1
Pregnancies		
0	31.727397	39.213158
1	29.616038	37.793103
2	29.679762	34.578947
3	29.231250	32.548148
4	31.255556	33.873913
5	31.100000	36.780952
6	29.591176	31.775000
7	29.975000	34.756000
8	30.693750	32.204545
9	28.840000	33.300000
10	30.114286	31.380000
11	37.125000	39.385714
12	30.560000	34.575000
13	33.280000	36.720000
14	NaN	35.100000
15	NaN	37.100000
17	NaN	40.900000

Aggregating data by pivoting with pandas

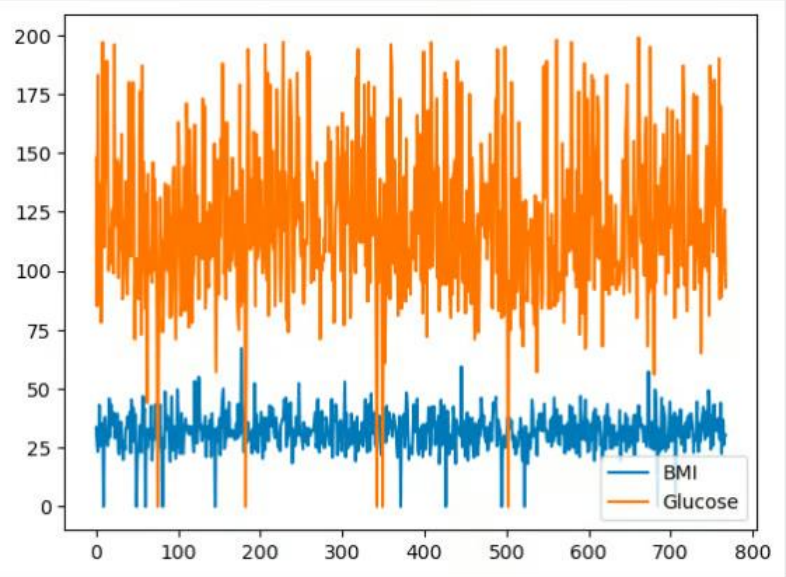
Data visualization in pandas

pandas provides convenience wrappers to Matplotlib plotting functions to make it easy to visualize your DataFrames. Below, you'll see how to do common data visualizations using pandas.

Line plots in pandas

pandas enables you to chart out the relationships among variables using line plots. Below is a line plot of BMI and Glucose versus the row index.

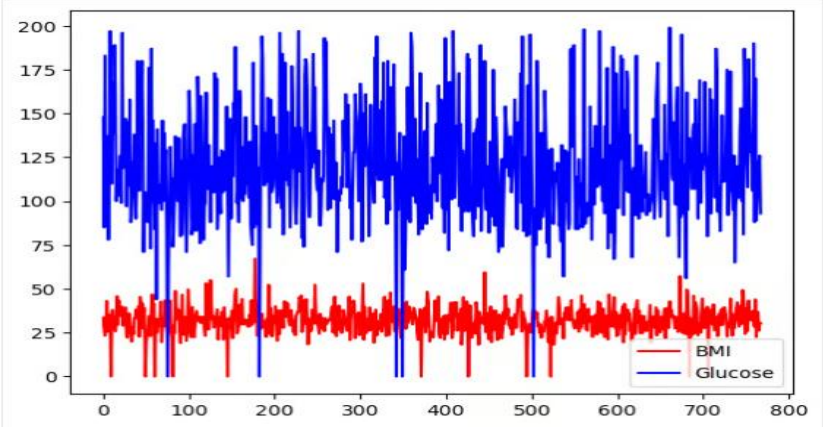
```
df[['BMI', 'Glucose']].plot.line()
```



Basic line plot with pandas

You can select the choice of colors by using the color argument.

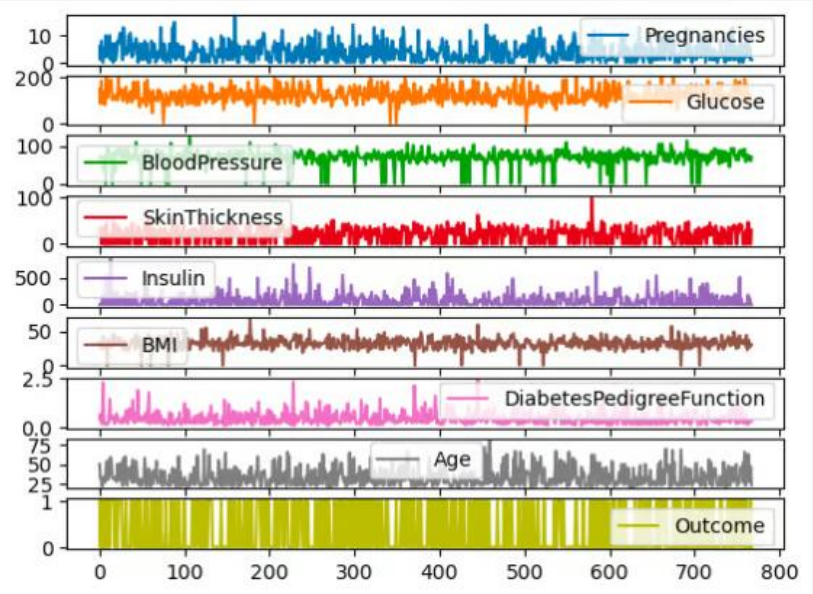
```
df[['BMI', 'Glucose']].plot.line(figsize=(20, 10),
                                 color={"BMI": "red", "Glucose": "blue"})
```



Basic line plot with pandas, with custom colors

All the columns of df can also be plotted on different scales and axes by using the subplots argument.

```
df.plot.line(subplots=True)
```

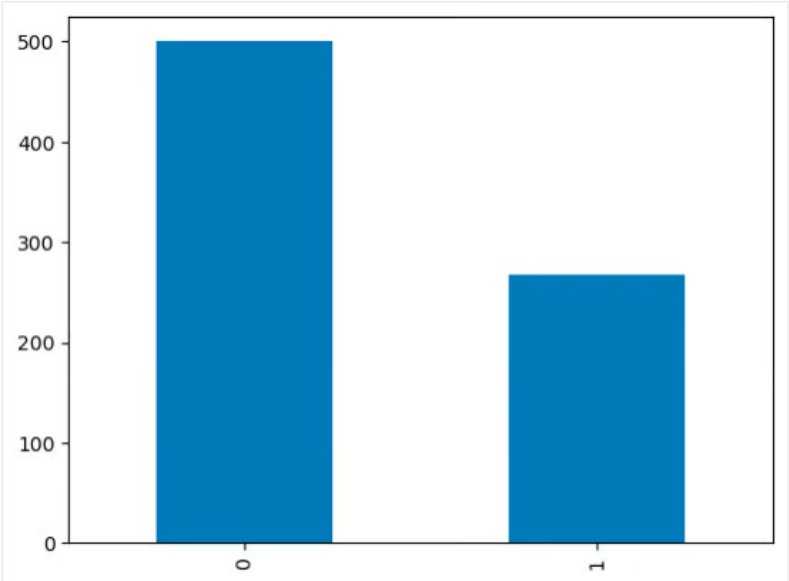


Subplots for line plots with pandas

Bar plots in pandas

For discrete columns, you can use a bar plot over the category counts to visualize their distribution. The variable Outcome with binary values is visualized below.

```
df['Outcome'].value_counts().plot.bar()
```

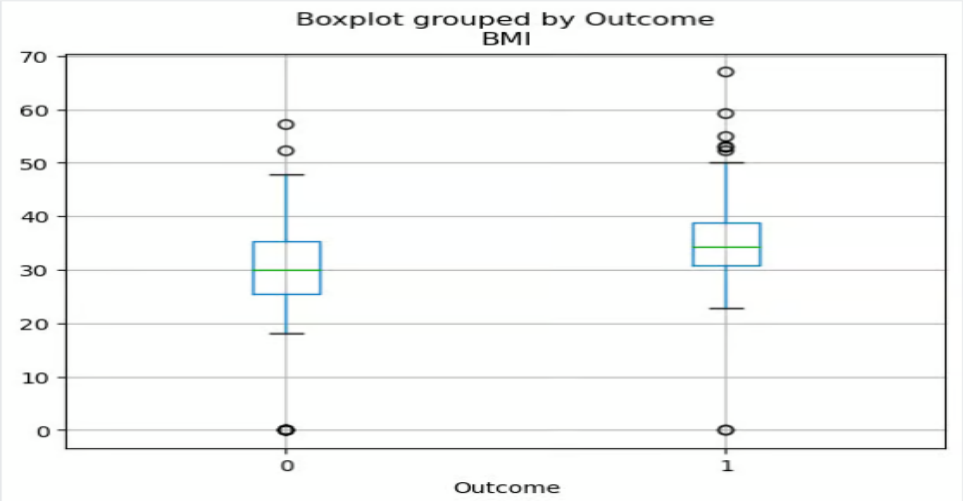


Barplots in pandas

Box plots in pandas

The quartile distribution of continuous variables can be visualized using a boxplot. The code below lets you create a boxplot with pandas.

```
df.boxplot(column=['BMI'], by='Outcome')
```



Boxplots in pandas

Comparison of Matplotlib and Pandas visualization based on ease of use, customization options, interactivity, and performance with large datasets:

1. Ease of Use

- **Matplotlib**:
    - Offers a **low-level** approach, which can be more complex for beginners.
    - Requires writing more lines of code to create basic plots.
    - Provides full control over every aspect of the plot, but this makes it harder for quick visualizations.
- **Pandas:**
    - Built on top of Matplotlib, making it **easier to use** with a more intuitive syntax.
    - Allows users to quickly create plots using .plot() without extensive customization.
    - Well-integrated with Pandas Data Frames, making it efficient for exploratory data analysis (EDA).

Winner: Pandas (for ease of use in simple plots), but **Matplotlib** is better for fine-tuned customizations.

---

2. Customization Options

- **Matplotlib**:
    - Highly flexible, allowing users to customize every aspect of a plot.
    - Supports advanced features like **subplotting, annotations, styles, and themes**.
    - Ideal for creating **publication-quality** visualizations.
- **Pandas**:
    - Limited customization as it is a wrapper around Matplotlib.
    - Users need to access Matplotlib methods for deeper modifications.
    - Best suited for quick, simple visualizations without extensive tweaks.

Winner: Matplotlib, as it provides significantly more control over customization.

---

3. Interactivity

- **Matplotlib**:

- By default, Matplotlib produces static plots.
- Supports **some interactivity** using mpl_toolkits or through integration with plt.show(block=False).
- Limited interactive support compared to libraries like Plotly.

- **Pandas**:
  - Since it is a wrapper over Matplotlib, it also produces **static plots**.
  - No built-in support for interactive visualizations.
  - Users must rely on **Plotly or Bokeh** for interactivity.

Winner: Neither (both are limited in interactivity without additional libraries).

---

4. Performance with Large Datasets

- **Matplotlib**:
  - Handles large datasets relatively well but can be slow for complex plots.
  - Performance degrades when rendering high-density scatter plots or 3D plots.
- **Pandas**:
  - Performs well with large datasets as it is optimized for DataFrames.
  - However, since it relies on Matplotlib, performance issues persist for very large datasets.
  - Can be improved by **downsampling data** before visualization.

Winner: Pandas (better integration with large DataFrames), but **Matplotlib** is still needed for detailed custom plots.

Final Verdict

Feature	Matplotlib	Pandas Visualization
Ease of Use	❌ Complex	✅ Simple, intuitive
Customization	✅ Highly customizable	❌ Limited
Interactivity	❌ Static (unless extended)	❌ Static
Large Dataset Handling	❌ Can be slow for big data	✅ Efficient with DataFrames

- **Use Pandas** when you need **quick exploratory** visualizations.
- **Use Matplotlib** when you need **full control** over plot appearance and customization.