

50 OF DATA DAYS ANALYSIS WITH PYTHON

The Ultimate Challenges Book for Beginners

pandas

numpy

MATPLOTLIB

seaborn

scikit-learn

BENJAMIN BENNETT ALEXANDER

50 OF DATA ANALYSIS WITH DAYS PYTHON

The Ultimate Challenges Book for Beginners

Copyright © 2023 by Benjamin Bennett Alexander

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law. For permission requests, please contact the publisher.

While every effort has been made to ensure the accuracy and completeness of the information in this book, the author does not warrant or represent its completeness or accuracy. The information provided is for general informational purposes only and should not be relied upon as professional or legal advice. Readers are advised to consult with appropriate professionals for advice specific to their circumstances. The author and publisher disclaim any liability for any loss or damage incurred by readers relying on information in this publication.

Feedback and Reviews

Dear Readers,

Thank you for choosing my book. As an independent writer, your support means a lot to me. If you enjoyed reading the book and found it valuable, I kindly request that you consider leaving a review, rating, and providing feedback on the platform where you purchased the book.

Your reviews and feedback play a crucial role in helping other readers make informed decisions about the book. They also provide me with valuable insights and encouragement to continue writing and improving my work.

Please take a moment to share your thoughts, opinions, and experiences with the book. Your honest feedback is highly appreciated and will contribute to the growth and success of future projects. You can contact me on:

benjaminbennettalexander@gmail.com

Once again, thank you for your support and for considering leaving a review. Your feedback is invaluable, and it helps me as an independent writer to reach a wider audience and create meaningful content.

Contents

Feedback and Reviews	3
About This Book	12
Getting Started.....	14
Day 1: Essentials of NumPy, Pandas, Matplotlib, Seaborn and Sklearn	16
1.0 NumPy	16
1.1 Creating NumPy Arrays	16
1.1.1 np.array()	16
1.1.2 np.arange()	17
1.1.3 np.zeros().....	18
1.1.4 np.ones().....	18
1.1.5 numpy.random.Generator.integers.....	18
1.1.6 numpy.random.Generator.random.....	19
1.2 Accessing Array Elements	19
1.2.1 Slicing	20
1.2.2 Fancy Indexing.....	20
1.2.3 Boolean Indexing	21
1.3 Array Manipulation	21
1.3.1 np.reshape()	22
1.3.2 np.concatenate()	23
1.3.3 np.split().....	23
1.3.4 np.transpose()	24
1.4 Mathematical Functions.....	25
1.4.1 np.add() and np.subtract()	25
1.4.2 np.multiply() and np.divide()	26

1.5 Statistical Functions	27
1.5.1 np.mean()	27
1.5.2 np.median().....	27
1.5.3 np.std().....	28
1.5.4 np.var().....	28
2.0 Pandas	29
2.1 Pandas Series	29
2.1.1 Series Index and Name.....	30
2.1.2 Series Data Type	30
2.2 Creating a Pandas DataFrame	32
2.3 Data Loading Functions	33
2.3.1 read_csv()	33
2.3.2 read_excel().....	34
2.3.3 read_sql()	34
2.4.1 .dropna()	34
2.4.2 fillna()	36
2.4.3 bfill and ffill().....	36
2.5.1 head()	37
2.5.2 tail()	38
2.5.3 info()	39
2.5.4 describe()	39
2.5.5 groupby()	40
2.5.6 merge()	41
2.6 Selecting Data.....	42
2.6.1 .loc	43
2.6.2 .iloc	44
2.7 Pandas Data Visualization Functions.....	45

2.7.1 Line Plot	45
2.7.2 Bar Plot	46
2.7.3 Box Plot.....	46
2.7.4 Hist Plot	48
2.8 Sorting Data	48
2.8.1 sort_values()	48
2.8.2 sort_index().....	49
2.8.3 nsmallest and nlargest().....	50
3.0 Matplotlib.....	51
3.1.1 plt.scatter()	51
3.1.2 plt.bar()	52
3.1.3 plt.hist().....	53
3.1.4 plt.imshow().....	54
3.1.5 plt.plot()	55
4.0 Seaborn	57
4.1.1 histplot()	57
4.1.2 lineplot()	59
4.1.3 pairplot()	59
4.1.4 regplot()	60
4.1.5 boxplot().....	61
5.0 Scikit-Learn.....	63
5.1.1 SimpleImputer().....	63
5.1.2 LabelEncoder().....	64
5.1.3 OneHotEncoder().....	65
5.1.4 StandardScaler()	67
5.1.5 train_test_split().....	69
5.1.6 Classification with Sklearn.....	71

5.1.7 accuracy_score()	72
5.1.8 precision Score()	73
5.1.9 recall_score()	74
5.2.1 f1_score()	76
5.2.2 Confusion_matrix()	76
5.2.3 Regression with Sklearn	78
5.2.4 Mean Squared Error (MSE)	80
5.2.5 Root Mean Squared Error (RMSE)	80
5.2.6 R2_score	82
6.0 Final Thoughts	83
Day 2: Creating and Manipulating Arrays	84
Day 2 - Answers	85
Day 3: Generating Random Arrays	90
Day 3 - Answers	91
Day 4: NumPy Arrays and Vector Operations	94
Day 4 - Answers	95
Day 5: Array Creation and Vector Operations	100
Day 5 - Answers	101
Day 6: Array Manipulation and Vector Operations	105
Day 6 - Answers	106
Day 7: Transpose and Swap Arrays	109
Day 7 - Answers	110
Day 8: Slicing NumPy Arrays	113
Day 8 - Answers	114
Day 9: Analyze a One-Dimensional Array	117
Day 9 - Answers	118
Day 10: The arange Function and Boolean Indexing	120

Day 10 - Answers	121
Day 11: Preprocessing, Analysis and Visualization	123
Day 11 - answers	124
Day 12: Array Sorting and Filtering	127
Day 12 - Answers	128
Day 13: Slicing and Analyzing Arrays.....	131
Day 13 - Answers	132
Day 14: Analyze Data with NumPy Part - 1	135
Day 14 - Answers	136
Day 15: Analyse Data with NumPy Part - 2	140
Day 15 - Answers	141
Day 16: Pandas Series Analysis	145
Day 16 - Answers	146
Day 17: Creating and Modifying DataFrames	150
Day 17 - Answers	151
Day 18: Runners Data Analysis –Part 1.....	154
Day 18 - Answers	155
Day 19: Runners Data Analysis – Part 2	159
Day 19 - Answers	160
Day 20: Explore Data with Pandas and Matplotlib.....	164
Day 20 - Answers	165
Day 21: Processing Data with Pandas.....	169
Day 21 - Answers	170
Day 22: Data Preprocessing and Analysis	175
Day 22 - Answers.....	177
Day 23: Preprocessing with Pandas and Matplotlib	182
Day 23 - Answers.....	183

Day 24: Business Data Analysis	187
Day 24 - Answers.....	188
Day 25: Retail Data Processing and Analysis - Part 1...	193
Day 25 - Answers.....	194
Day 26: Retail Data Processing and Analysis – Part 2 .	197
Day 26 - Answers.....	198
Day 27: Retail Data Processing and Analysis – Part 3 .	201
Day 27 - Answers.....	202
Day 28: Population Data Analysis.....	205
Day 28 - Answers	206
Day 29: Car Service Data Analysis	213
Day 29 - Answers.....	214
Day 30: Furniture Data Analysis.....	219
Day 30 - Answers	220
Day 31: Analyze Database Data with SQL	224
Day 31 - Answers	225
Day 32: Soccer Stricker's Data Analysis	230
Day 32 - Answers.....	231
Day 33: Website Data Analysis.....	236
Day 33 - Answers.....	237
Day 34: Income Data Analysis	243
Day 34 - Answers.....	244
Day 35: Runners And Income Data Analysis	248
Day 35 - Answers.....	249
Day 36: Social Media Data Analysis	253
Day 36 - Answers.....	254
Day 37: Stock Market Data Processing and Analysis ...	258

Day 37 - Answers.....	259
Day 38: Rental Car Data Analysis	264
Day 38 - Answers	265
Day 39: Analyze, Transform, and Shift Data	270
Day 39 - Answers.....	271
Day 40: Car Spare Parts Data Analysis	278
Day 40 - answers	280
Day 41: Population Data Analysis	293
Day 41 - Answers	294
Day 42: Toys Data Analysis	299
Day 42 - Answers.....	300
Day 43: Time Series Data Analysis.....	305
Day 43 - Answers.....	306
Day 44: Sports Data Analysis	311
Day 44 - Answers	312
Day 45: Medical Data Analysis.....	316
Day 45 - Answers.....	317
Day 46: Financial Data Analysis	320
Day 46 - Answers	321
Day 47: Text Data Preprocessing	325
Day 47 - answers	326
Day 48: Preprocess Data with Sklearn	330
Day 48 - Answers	331
Day 49: End-to-End Regression Challenge.....	335
Day 49 - Answers	337
Day 50: End-to-End Classification Challenge	349
Day 50 - Answers	352

What's Next?	377
Other Books By Author	380

About This Book

Welcome to "**50 Days of Data Analysis with Python: The Ultimate Challenges Book for Beginners**"! This book is designed to take you on an exciting journey through the world of data analysis using Python. Whether you're a novice programmer or someone with some coding experience, this book will challenge and enhance your skills while exploring key Python libraries such as NumPy, pandas, Seaborn, Sklearn and Matplotlib.

Challenge Yourself, Excel in Data Analysis

This book is not your typical Python guide. It is a collection of carefully crafted challenges, each designed to push your knowledge, problem-solving abilities, and understanding of data analysis to the next level. Each challenge is designed to be completed in a single day, making it an ideal resource for those looking for a structured learning experience. Some days may require more work than others.

Discover the Power of Python Libraries

Throughout this book, you'll explore the fundamental concepts of data analysis and gain hands-on experience with the essential Python libraries. You'll dive into the versatile NumPy library for efficient numerical computations, master the powerful pandas library for data manipulation and analysis, visualize data with the captivating Seaborn library, and create stunning plots and visualizations with Matplotlib. These libraries have numerous functions; however, in this book, we will concentrate on the functions that are mostly used in data analysis.

Learn by Doing, Solve Real-world Problems

The challenges in this book are carefully designed to simulate real-world scenarios, enabling you to apply your

newfound skills to practical data analysis tasks. You'll work with diverse datasets, explore data cleaning and preprocessing techniques, perform statistical analysis, create insightful visualizations, and draw meaningful conclusions from your data.

Embrace the Learning Journey

I encourage you to embrace the learning journey presented in this book. Make good use of the internet to tackle the challenges, but resist the temptation to seek immediate answers from external sources such as chatGPT. Instead, challenge yourself to think critically, explore different approaches, and leverage the guidance provided within the book. Each challenge comes with detailed explanations and hints to help you progress, ensuring you develop a solid understanding of Python's data analysis capabilities.

Get Ready to Transform into a Skilled Data Analyst

By the end of "50 Days of Data Analysis with Python," you will have acquired a strong foundation in Python programming, mastered key data analysis techniques, and gained the confidence to tackle complex data problems. This book will serve as a springboard for your future data analysis endeavors and open doors to exciting opportunities in various fields.

So, are you ready to embark on this thrilling 50-day journey? Get ready to unlock the full potential of Python for data analysis and become a proficient data analyst. So, dive in and embrace the challenges that await you!

Note: Remember, this book is designed for self-study and personal growth. Embrace the challenges, experiment, and let your creativity flourish as you tackle each task head-on.

Getting Started

In "50 Days of Data Analysis with Python," day one is a recap of the important functions of NumPy, pandas, and Matplotlib as these are the main libraries used in the book. If you are already familiar with the basic functions of these libraries, you can skip day one. In this book, I utilize code snippets from Jupyter Notebook to provide an interactive learning experience. Jupyter Notebook is a powerful tool that allows you to write and execute Python code in a web-based environment. I highly recommend using Jupyter Notebook or an equivalent platform like Google Colab for solving the challenges in the book.

Google Colab offers several advantages as it comes preinstalled with many essential libraries needed for data analysis. This means you can dive right into the practical aspects without the need for additional installations. It provides a convenient and accessible platform for running code, collaborating with others, and leveraging the power of cloud computing resources.

To install Jupyter Notebook on your local machine, you can follow these steps:

1. Install Python: If you don't have Python installed, visit the official Python website (<https://www.python.org>) and download the latest version compatible with your operating system. Follow the installation instructions provided.
2. Install Jupyter Notebook: Once Python is installed, open a command prompt or terminal and run the following command:

```
pip install jupyter
```

This will install Jupyter Notebook along with its dependencies.

If you choose to use Google Colab, you can access it through your web browser. Simply visit the Google Colab website (<https://colab.research.google.com>) and sign in with your Google account. You can create a new notebook and start writing code right away.

Please note that "**50 Days of Data Analysis with Python**" does not aim to teach Python from scratch. It assumes that the reader has some basic knowledge of Python programming. If you are new to Python, I recommend exploring introductory Python resources to familiarize yourself with the language before diving into data analysis.

The libraries used in this book are: pandas, NumPy, Matplotlib, Sklearn, and Seaborn for data analysis tasks. If you are using Jupyter Notebook, you can install these libraries by running the following command in a notebook cell:

```
!pip install pandas numpy matplotlib Sklearn seaborn
```

This command utilizes the pip package manager to install the specified libraries.

The datasets used in the book will be saved on GitHub. Here is the link below:

<https://github.com/Realbenjizo/50-Days-of-Data-Analysis-With-Python>. You can access and download these datasets from the provided GitHub repository to use for the challenges in the book. Download a zip file: **Datasets_50_Days_of_Data_Analysis**.

I also suggest that you create a GitHub account, where you can keep a record of the challenges that you tackle in this book. I hope that using Jupyter Notebook or Google Colab, along with the suggested libraries and datasets, will enhance your learning experience and empower you to become a proficient data analyst. Let's embark on this exciting journey together!

Day 1: Essentials of NumPy, Pandas, Matplotlib, Seaborn and Sklearn

We are going to look at the essential functions of NumPy, Pandas, Matplotlib, Seaborn and Sklearn for data analysis.

1.0 NumPy

NumPy is a Python library that is commonly used for scientific computing and data analysis. It stands for **Numerical Python** and provides fast and efficient numerical computation on large datasets. NumPy is built on top of the C language, which makes it faster than pure Python code.

We will cover some of the most important functions of NumPy that are used for data analysis. These functions will include:

1.1 Creating NumPy Arrays

NumPy arrays are similar to Python lists, but they are more efficient when it comes to numerical computation. They can be one-dimensional or multi-dimensional and can hold homogeneous elements. Here are some functions that can be used to create NumPy arrays:

1.1.1 np.array()

If we want to create an array from a list of tuples, we can use the `np.array()` function. Let's create a one-dimensional array below:

```
[1]: import numpy as np

lst = [1, 2, 3, 4, 5, 6]
# Creating an array from List
arr = np.array(lst)
arr
```



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

The array data type is inferred from the data. However, we can also set the data type using the `dtype` parameter. See below:

```
[2]: lst = [1, 2, 3, 4, 5, 6]

# Creating an array of floats from list
arr = np.array(lst, dtype = float)
arr

[2]: array([1., 2., 3., 4., 5., 6.])
```

We can also create a multi-dimensional array from a nested list. We are going to create an array and check how many dimensions it has using the `ndim` attribute. See below:

```
[3]: names = [['Jon', 'Mary', 'Paul'], ['Peter', 'Ben', 'Saul']]

arr1 = np.array(names)
arr1

[3]: array([['Jon', 'Mary', 'Paul'],
           ['Peter', 'Ben', 'Saul']], dtype='<U5')

[4]: # Checking for number of dimensions in the array
arr1.ndim

[4]: 2
```

1.1.2 np.arange()

This function creates an array with regularly spaced values within a given range. It works similarly to the Python `arange()` function when applied to numbers. Here is an example of how to use it to generate an array.

```
[5]: arr2 = np.arange(0, 50, 5)

arr2

[5]: array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45])
```

We have generated an array of values from 0 to 50 (50 excluded), spaced by 5.

1.1.3 np.zeros()

The **np.zeros()** function creates an array with all elements set to zero. It allows us to set the shape and data type of the array. Below, we generate an array of 2 rows and 3 columns of float **dtype**.

```
[6]: arr3 = np.zeros((2, 3), dtype = float)
arr3
```

```
[6]: array([[0., 0., 0.],
           [0., 0., 0.]])
```

1.1.4 np.ones()

The **np.ones()** function works similarly to the **np.zeros()** function. The only difference is that the **np.ones()** function creates an array with all elements set to one (1).

```
[7]: arr3 = np.ones((2, 3), dtype = float)
arr3
```

```
[7]: array([[1., 1., 1.],
           [1., 1., 1.]])
```

1.1.5 numpy.random.Generator.integers

The **numpy.random.Generator.integers** function is used by the NumPy library for generating random integers. It allows the generation of random integers within a specified range or from a specified set of values.

Let's say we want to create an array of random integers from 0 to 9, with 2 rows and 4 columns. First, we will create a random number generator, and then we will use the

generator to generate the array. Here is how we can use this function to achieve that:

```
[8]: # Create a random number generator
rng = np.random.default_rng()

# Generate random integers from 0 to 9
rng.integers(low=0, high=10, size=(2, 4))

[8]: array([[7, 0, 9, 1],
           [5, 1, 5, 6]], dtype=int64)
```

Note that the output will always vary because it is generated randomly.

1.1.6 numpy.random.Generator.random

This method creates an array of random floats. Let's create an array of random floats between 0 and 1 in the shape of 2 rows and 4 columns.

```
[9]: # Create a random number generator
rng = np.random.default_rng(seed = 24)

# Generate random floats between 0 and 1
rng.random((2, 4))

[9]: array([[0.33026884, 0.40517732, 0.57473782, 0.50639977],
           [0.56421251, 0.56968731, 0.87411653, 0.08643046]])
```

Since the output is random, to ensure that the output is reproducible, we set the seed parameter to 24. This means that the generated random state will be repeated every time you run the code.

1.2 Accessing Array Elements

Once you have created a NumPy array, you may want to access its elements. NumPy arrays are indexed using integers starting from zero (0). Here are some ways to access NumPy array elements:

1.2.1 Slicing

This is similar to Python lists, where you can access a range of elements using a colon (:). We are going to use the array of names to demonstrate how to use indexing to select elements from the array.

```
[10]: names = [[ "Jon", "Mary", "Paul"], [ "Peter","Ben", "Saul"]]

arr1 = np.array(names)
arr1

[10]: array([['Jon', 'Mary', 'Paul'],
       ['Peter', 'Ben', 'Saul']], dtype='|<U5')
```

Let's say we want to select the names "Mary" and "Paul" from the array. Here is how we can use slicing for such an operation:

```
[11]: select_mary_paul = arr1[0,1:]
select_mary_paul

[11]: array(['Mary', 'Paul'], dtype='|<U5')
```

We first access row [0], which has the names "Mary" and "Paul." Then we use the comma (,) to jump into the row. Since Mary is sitting on index 1, we select **arr1[0,1]**. We then use the colon (:) after 1 to tell the code that we want "Mary" and everything in the row after Mary to be selected (since the name "Paul" is the only name after Mary in the row). Putting it all together as **arr1[0, 1:]** allows us to achieve this selection.

1.2.2 Fancy Indexing

This allows you to select elements based on a list of indices. If we want to select "Peter" and "Paul" from the array; here is how we do it using fancy indexing:

```
[12]: # Creating an elements indices
select_peter = np.array([1, 0])
select_paul = np.array([0, 2])

# Using fancy indexing to select
select_peter_paul = arr1[select_peter, select_paul]
select_peter_paul

[12]: array(['Peter', 'Paul'], dtype='<U5')
```

Here, we created two arrays of indices, `select_peter` and `select_paul`, that contain the row and column indices of the elements we want to select from the original array, `arr1`. We then passed these arrays of indices to `arr1` to select the corresponding elements and create a new array. Note that the arrays of indices must have the same shape to use fancy indexing with multi-dimensional arrays.

1.2.3 Boolean Indexing

This allows you to select elements based on a Boolean array. It involves creating a Boolean array of the same shape as the original array, where the elements of the Boolean array are true or false based on a given condition. This Boolean array is then used to select the corresponding elements from the original array. Let's say we want to remove all the numbers from an array that are odd.

```
[13]: arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# create a boolean mask based on the condition
filter_array = arr % 2 != 0

# select the elements from the original array based on the filter
arr[filter_array]

[13]: array([1, 3, 5, 7, 9])
```

1.3 Array Manipulation

NumPy provides a wide range of functions to manipulate arrays. These include:

1.3.1 np.reshape()

We can change the shape of an array using the **np.reshape()** function. Here is how we can use reshape to flatten a multi-dimensional array.

```
[14]: # Creating an array
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Flatten the array using reshape
np.reshape(arr, 9)
```



```
[14]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In the code above, we create a 2-dimensional array and then use the **np.reshape()** function to flatten the array. The number 9 is the number of elements in the original array. By passing the number 9, we are telling the reshape function to return a 1-dimensional array.

Here is another example of how to reshape an array. We will reshape the array of names below:

```
[15]: names = [['Jon', 'Mary', 'Paul'], ['Peter', 'Ben', 'Saul']]

arr1 = np.array(names)
arr1
```



```
[15]: array([['Jon', 'Mary', 'Paul'],
           ['Peter', 'Ben', 'Saul']], dtype='<U5')
```



```
[16]: # reshaping array to a (3, 2) shape
new_array = np.reshape(arr1, (3, 2))
new_array
```



```
[16]: array([['Jon', 'Mary'],
           ['Paul', 'Peter'],
           ['Ben', 'Saul']], dtype='<U5')
```

Above, we create an **arr1** of names for shapes (2, 3). We then use the **reshape()** function to reshape the array to a (3, 2) shape and assign it to a new variable.

1.3.2 np.concatenate()

This function joins two or more arrays together. We can pick the axis on which we want the arrays to be joined. If we do not provide the axis, it will be joined on axis 0.

```
[17]: arr1 = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
arr2 = np.array([[100, 110, 120], [130, 140, 150]])

# Concatenate on axis-0
arr3 = np.concatenate((arr1, arr2), axis=0)
arr3
```



```
[17]: array([[ 10,  20,  30],
       [ 40,  50,  60],
       [ 70,  80,  90],
       [100, 110, 120],
       [130, 140, 150]])
```

Note that for concatenation to work, the arrays must have the same number of dimensions. If we were to try to join these two arrays on axis 1 (rows), we would get an error because the two arrays do not have an equal number of rows.

1.3.3 np.split()

This function splits an array into smaller arrays. Below, we use the **np.split()** function to split the array into two parts. You can see in the output that we have created two arrays from **arr1**: **split_one** and **split_two**.

```
[18]: names = [[ "Jon", "Mary", "Paul"], [ "Peter","Ben", "Saul"]]

# Creating array
arr1 = np.array(names)

# using split to split array into two parts
split_one, split_two = np.split(arr1, 2)
```

```
[19]: split_one
```

```
[19]: array([['Jon', 'Mary', 'Paul']], dtype='<U5')
```

```
[20]: split_two
```

```
[20]: array([['Peter', 'Ben', 'Saul']], dtype='<U5')
```

1.3.4 np.transpose()

This function transposes an array, switching its rows and columns. Transposing an array is especially useful when working with linear algebra operations such as matrix multiplication. In some cases, the multiplication of two matrices requires that one of the matrices be transposed in order for the operation to be performed correctly. For example, if we try to carry out a dot operation on the two arrays below, we get a value error because the arrays have different shapes.

```
[21]: arr1 = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])  
arr2 = np.array([[100, 110, 120], [130, 140, 150]])
```

```
# Performing a dot operation  
np.dot(arr1, arr2)
```

```
-----  
ValueError                                Traceback (most recent call last)  
Input In [21], in <cell line: 5>()  
      2 arr2 = np.array([[100, 110, 120], [130, 140, 150]])  
      4 # Performing a dot operation  
----> 5 np.dot(arr1, arr2)  
  
File <__array_function__ internals>:200, in dot(*args, **kwargs)  
  
ValueError: shapes (3,3) and (2,3) not aligned: 3 (dim 1) != 2 (dim 0)
```

So to make the dot operation possible, we have to transpose one of the arrays. We transpose **arr2**.

```
[22]: arr1 = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
arr2 = np.array([[100, 110, 120], [130, 140, 150]])

# Performing a dot operation and transposing arr2
np.dot(arr1, arr2.transpose())

[22]: array([[ 6800,  8600],
       [16700, 21200],
       [26600, 33800]])
```

You can see in the output that the operation was successful after transposing the array.

1.4 Mathematical Functions

NumPy provides a wide range of mathematical functions that can be used on arrays. These include:

1.4.1 np.add() and np.subtract()

The **np.add()** adds two arrays together (element-wise addition of two arrays, where corresponding elements from the two arrays are added together), and the **np.subtract()** function subtracts one array from another (element-wise). Here are the examples below:

```
[23]: arr1 = np.array([[1, 2, 4], [6, 7, 8]])
arr2 = np.array([[3, 3, 6], [4, 5, 7]])

# adding two arrays
np.add(arr1, arr2)

[23]: array([[ 4,  5, 10],
       [10, 12, 15]])

[24]: # Subtracting two arrays
np.subtract(arr1, arr2)

[24]: array([[-2, -1, -2],
       [ 2,  2,  1]])
```

1.4.2 np.multiply() and np.divide()

The **np.multiply()** function multiplies two arrays together. It performs element-wise multiplication of two arrays, where the corresponding elements from the two arrays are multiplied together. The **np.divide()** function divides one array by another (element-wise). Note that for these operations to work, the arrays must be broadcastable in a common shape.

```
[25]: arr1 = np.array([[1, 2, 4], [6, 7, 8]])
arr2 = np.array([[3, 3, 6], [4, 5, 7]])
```

```
# multiplying two arrays
np.multiply(arr1, arr2)
```

```
[25]: array([[ 3,  6, 24],
[24, 35, 56]])
```

```
[26]: # Dividing two arrays
np.divide(arr1, arr2)
```

```
[26]: array([[0.33333333, 0.66666667, 0.66666667],
[1.5           , 1.4           , 1.14285714]])
```

1. 4.3 np.power() and np.sqrt()

This **np.power()** function raises an array to a given power. The **np.sqrt()** function calculates the square root of each element in an array.

```
[27]: arr2 = np.array([[3, 3, 6], [4, 5, 7]])
# Raising array to power 2
np.power(arr2, 2)
```

```
[27]: array([[ 9,  9, 36],
[16, 25, 49]], dtype=int32)
```

```
[28]: arr2 = np.array([[3, 3, 6], [4, 5, 7]])  
  
# Calculating the square-root of array  
np.sqrt(arr2)
```



```
[28]: array([[1.73205081, 1.73205081, 2.44948974],  
           [2. , 2.23606798, 2.64575131]])
```

1.5 Statistical Functions

NumPy also provides a wide range of statistical functions that can be used on arrays. These include:

1.5.1 np.mean()

The `np.mean()` function calculates the mean of an array. By default, this function will flatten the array and calculate the mean of the flattened array. You can also specify the axis on which you want the mean to be calculated. Below, we calculate the mean on axis 1. This means we calculate the mean of each row.

```
[29]: arr2 = np.array([[3, 3, 6], [4, 5, 7]])  
  
# Calculating the mean of each row  
np.mean(arr2, axis=1)
```

```
[29]: array([4. , 5.33333333])
```

1.5.2 np.median()

This function works similar to the `np.mean()` function. In the example below, we calculate the median of each row:

```
[30]: arr2 = np.array([[3, 3, 6], [4, 5, 7]])  
  
# Calculating the median of each row  
np.median(arr2, axis=1)
```

```
[30]: array([3., 5.])
```

1.5.3 np.std()

Standard deviation is a measure of how spread out the data is. It tells you how much the data points typically vary from the average (mean). To calculate the standard deviation of an array, we can use the `np.std()` function. Below, we calculate the std of each column.

```
[31]: arr2 = np.array([[3, 3, 6], [4, 5, 7]])  
  
# Calculating the std of each column  
np.std(arr2, axis=0)
```

```
[31]: array([0.5, 1. , 0.5])
```

1.5.4 np.var()

Variance measures the average of the squared differences from the mean. The `np.var()` function calculates the variance of an array. Here is an example of how we can calculate var on axis 1 of the array:

```
[32]: arr2 = np.array([[3, 3, 6], [4, 5, 7]])  
  
# Calculating the var of each column  
np.var(arr2, axis=0)
```

```
[32]: array([0.25, 1. , 0.25])
```

1.5.5 np.min() and np.max()

This `np.min()` calculates the minimum value of an array, and this `np.max()` calculates the maximum value of an array. First, we calculate the minimum of each column, and in the second example, we calculate the mean of the whole array. See below.

```
[33]: arr2 = np.array([[3, 3, 6], [4, 5, 7]])  
  
# Calculating the min of each column  
np.min(arr2, axis=0)
```

```
[33]: array([3, 3, 6])
```

```
[34]: arr2 = np.array([[3, 3, 6], [4, 5, 7]])  
  
# Calculating the max of the whole array  
np.max(arr2, axis=None)
```

```
[34]: 7
```

2.0 Pandas

Pandas is an open-source data analysis and manipulation library in Python. It provides data structures for efficiently storing and manipulating large datasets, as well as tools for reading and writing data from various sources. In this section, we'll go over the basics of pandas and introduce some of their most important functions:

2.1 Pandas Series

A pandas Series is a one-dimensional array. Simply put, a series has one column and an index. In pandas, we can create a one-dimensional array using the `Series()` function. Here is an example:

```
[35]: import pandas as pd  
  
# Creating a Series  
fruits = pd.Series(["Orange", "Apple", "Mango"],  
                  name="fruits")  
fruits
```

```
[35]: 0    Orange  
1    Apple  
2    Mango  
Name: fruits, dtype: object
```

In the above code, we have made a one-dimensional array using pandas. We passed a list ["Orange," "Apple," "Mango"] as an argument to the function. The output on the far left (0, 1, 2) is the index of the Series. And on the right is a column of the data in the Series. By default, the index will start from (0, 1, 2,... n).

2.1.1 Series Index and Name

We can also specify the values of the index using the index parameter. The specified index must be the same length as the items (data) in the Series and does not have to be unique (meaning you can pass the same index value for all the data). In the example below, we specify the values of the index. You can see in the output that the series is now (a, b, c). We could have passed (c, c, c) as index values, and it would still be valid. However, it is good practice to have unique values for the index.

```
[36]: fruits = pd.Series(["Orange", "Apple", "Mango"],
                        name="fruits",
                        index = ['a', 'b', 'c'])

fruits
```



```
[36]: a    Orange
      b    Apple
      c    Mango
Name: fruits, dtype: object
```

The name of the Series is "fruits." This is the name that we have passed to the name parameter as an argument. The name parameter is optional.

2.1.2 Series Data Type

The **dtype** in the Series means "data type." By default, the data type is inferred from the data in the Series. The data type of our series above is object data type. If the series'

data is of the string type, the inferred data type will be object. Because the items in the list are strings, in the code above, that is why the data type is "object." If the data is a mixture of different data types, e.g., strings and integers, then the data type will be "object." Let's see what happens when we create a series of integers:

```
[37]: int_numbers = pd.Series([10, 20, 30], name="numbers")
int_numbers
```



```
[37]: 0    10
      1    20
      2    30
Name: numbers, dtype: int64
```

You can see from the output that the data type is now int64 (a 64-bit integer). This is because the data in our Series is in integers. What happens when we mix integers with strings? You can see below that the dtype has changed to "object." By default, heterogeneous data will be interpreted as object data.

```
[38]: int_str = pd.Series([10, 20, 30, 'Orange'], name="mixed")
int_str
```



```
[38]: 0      10
      1      20
      2      30
      3    Orange
Name: mixed, dtype: object
```

We can also specify the data type of the Series. The series function has a **dtype** parameter where we can pass a data type as an argument. Let's see an example below. Below, we specify that we want our dtype to be **int8**. You can see that even though our data would have been assigned the **int64** type by default, it is now **int8**.

```
[39]: int_str = pd.Series([10, 20, 30], name="mixed", dtype="int8")
      int_str
```

```
[39]: 0    10
      1    20
      2    30
      Name: mixed, dtype: int8
```

It is also possible to change the data type of a series after it is created. This can be done using the `astype()` method. Let's see an example.

```
[40]: num_series = pd.Series([10, 20, 30, 40])
      num_series
```

```
[40]: 0    10
      1    20
      2    30
      3    40
      dtype: int64
```

Now, let's change the `dtype` of the Series to `float64`

```
[41]: # Changing data type to float
      num_series = num_series.astype("float64")
      num_series
```

```
[41]: 0    10.0
      1    20.0
      2    30.0
      3    40.0
      dtype: float64
```

2.2 Creating a Pandas DataFrame

To create a pandas DataFrame, we can pass a dictionary or a list of lists as the argument to the `DataFrame` function. The names of the columns are passed as a separate argument to the `columns` parameter. Below we create a DataFrame from a dictionary:

```
[42]: data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
             'Age': [25, 32, 18, 47],
             'Salary': [50000, 80000, 20000, 120000]}

df = pd.DataFrame(data, columns=['Name', 'Age', 'Salary'])
df
```

	Name	Age	Salary
0	Alice	25	50000
1	Bob	32	80000
2	Charlie	18	20000
3	David	47	120000

We can also create a DataFrame from lists by passing them to the DataFrame function as a dictionary. See below:

```
[43]: names = ['Alice', 'Bob', 'Charlie', 'David']
age = [25, 32, 18, 47]
salary = [50000, 80000, 20000, 120000]

df = pd.DataFrame({"Names": names, "Age":age, "Salary": salary})
df
```

	Names	Age	Salary
0	Alice	25	50000
1	Bob	32	80000
2	Charlie	18	20000
3	David	47	120000

2.3 Data Loading Functions

Pandas provides several functions for loading data from various sources:

2.3.1 `read_csv()`

This function is used to read data from a CSV file into a pandas DataFrame. Since most tabular data is saved in this format, this is one of the most commonly used functions in pandas. It provides options to specify the delimiter, encoding, header row, and more. Let's say you have a file

called "Data" saved in CSV format. Here is how you would open the file using the `read_csv()` function.

```
[44]: try:  
    df = pd.read_csv("Data.csv")  
except FileNotFoundError:  
    print('File not found')  
  
File not found
```

Here, we get file not found because, the file does not exist. This just demonstrates how to open a CSV file with pandas.

2.3.2 `read_excel()`

You can also use the pandas `read_excel()` files to read data from an Excel file into a pandas DataFrame. It provides options to specify the sheet name, header row, and more.

2.3.3 `read_sql()`

This function is used to read data from a SQL database into a pandas DataFrame. It requires a connection to the database and a SQL query.

2.4 Data Cleaning

Data comes in many forms. Before we can start analyzing our data, we need to clean it up. Here are some of the most important functions for data cleaning:

2.4.1 `.dropna()`

If we have NaN values in the DataFrame, we can drop them using the `df.dropna()` method. Let's create a DataFrame with missing values and use the `dropna()` method to drop the missing values.

```
[45]: names = ['Alice', 'Bob', 'Charlie', 'David']
age = [25, None, 18, 47]
salary = [50000, 80000, None, 120000]

df = pd.DataFrame({"Names": names, "Age":age, "Salary": salary})
df
```

	Names	Age	Salary
0	Alice	25.0	50000.0
1	Bob	NaN	80000.0
2	Charlie	18.0	NaN
3	David	47.0	120000.0

The NaN values in the DataFrame represent missing values. Now, to drop all rows with missing values with the `dropna()` method, here is how we do it:

```
[46]: df.dropna()
```

	Names	Age	Salary
0	Alice	25.0	50000.0
3	David	47.0	120000.0

We can also drop all columns with missing values. Columns are on `axis 1`, so we pass `1` to the axis parameter. This will drop all the columns except for the "Names" column.

```
[47]: df.dropna(axis=1)
```

	Names
0	Alice
1	Bob
2	Charlie
3	David

You can see above that the "Age" and "Salary" columns have been dropped because they have missing values.

2.4.2 fillna()

This method is used to fill missing values in the DataFrame with a specified value or method. In the example below, we use the `fillna()` method to fill the missing value in the "Age" column with the mean of the values in the column, and we fill the missing value in the "Salary" column with the mode or the most frequent value of the column.

```
[48]: names = ['Alice', 'Bob', 'Charlie', 'David']
age = [25, None, 18, 47]
salary = [50000, 80000, None, 120000]

df = pd.DataFrame({"Names": names, "Age": age, "Salary": salary})

# Fill missing values in 'Age' with mean
df['Age'] = df['Age'].fillna(df['Age'].mean())
# Fill missing values in 'Salary' using mode
df['Salary'] = df['Salary'].fillna(df['Salary'].mode())
df
```

	Names	Age	Salary
0	Alice	25.0	50000.0
1	Bob	30.0	80000.0
2	Charlie	18.0	120000.0
3	David	47.0	120000.0

2.4.3 bfill and ffill()

In pandas we can also use the `bfill()` method (backward fill) to fill the missing value with the next non-missing value in the column. So, if a value is missing, `bfill()` looks down the column and replaces the missing value with the first valid entry that appears after it.

With `ffill()` method (forward fill), we fill the missing values with the previous non-missing value in the column. So, if a value is missing, `ffill()` looks up the column and replaces the missing value with the last valid entry that appeared before it. We use these two method to fill the missing values in the "Age" and "Salary" columns below:

```
[49]: names = ['Alice', 'Bob', 'Charlie', 'David']
age = [25, None, 18, 47]
salary = [50000, 80000, None, 120000]

df = pd.DataFrame({"Names": names, "Age": age, "Salary": salary})

# Fill missing values in 'Age' using backward fill
df['Age'] = df['Age'].bfill()
# Fill missing values in 'Salary' using forward fill
df['Salary'] = df['Salary'].ffill()
df
```

	Names	Age	Salary
0	Alice	25.0	50000.0
1	Bob	18.0	80000.0
2	Charlie	18.0	80000.0
3	David	47.0	120000.0

2.5 Data Manipulation in Pandas

Once data is loaded into a pandas DataFrame, there are several methods for manipulating and transforming the data.

2.5.1 head()

This method returns the first n rows of a DataFrame. By default, it returns the first five rows. This method is perfect for exploration. For example, you can use it to check if data has loaded or processed correctly or explore the number of rows and columns, as well as the column names and data types. Below, we create a DataFrame from a list and use it to view the first five rows of the DataFrame.

```
[50]: names = ['Alice', 'Bob', 'Charlie', 'David', 'John',
              'Mpho', 'Steve', 'Ben']
age = [25, 29, 33, 21, 57, 66, 50, 30]

# Creating a DataFrame
df = pd.DataFrame({"Names": names, "Age":age})
# Viewing the first five rows
df.head()
```

```
[50]:    Names   Age
0      Alice    25
1        Bob    29
2   Charlie    33
3     David    21
4      John    57
```

If we want to view only two rows, we would pass `2` to the `head()` method as an argument. See below:

```
[51]: # Creating a DataFrame
df = pd.DataFrame({"Names": names, "Age":age})
# Viewing the first 2 rows
df.head(2)
```

```
[51]:    Names   Age
0      Alice    25
1        Bob    29
```

2.5.2 tail()

The `tail()` method returns the last `n` rows of a DataFrame. By default, it returns the last 5 rows. It's a useful tool for quickly inspecting the ending portion of your data. See below:

```
[52]: # Creating a DataFrame
df = pd.DataFrame({"Names": names, "Age":age})
# Viewing the last 5 rows
df.tail()
```

```
[52]:   Names  Age
      3  David   21
      4  John    57
      5  Mpho    66
      6  Steve   50
      7    Ben    30
```

2.5.3 info()

The `info()` method provides a summary of the DataFrame, including the data types, number of non-null values, and memory usage. You can see below that our DataFrame is using 256 bytes of memory and the data types are 'object' and 'int64.'

```
[53]: # Creating a DataFrame
df = pd.DataFrame({"Names": names, "Age":age})
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8 entries, 0 to 7
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype  
---  -- 
 0   Names    8 non-null      object 
 1   Age      8 non-null      int64  
dtypes: int64(1), object(1)
memory usage: 256.0+ bytes
```

2.5.4 describe()

If we want a summary of the statistics for each column in the DataFrame, such as count, mean, standard deviation, minimum, and maximum, we can use the `describe()`

method. When we use the `describe()` method on the DataFrame, we can see the statistical summary of our DataFrame. For example, we can see that the average age of the people is 38.

```
[54]: df.describe()
```

```
[54]:          Age
count    8.000000
mean    38.875000
std     16.522171
min    21.000000
25%   28.000000
50%   31.500000
75%   51.750000
max   66.000000
```

2.5.5 groupby()

This method is used to group the data by one or more columns and perform aggregate operations on each group, such as sum, mean, count, and more. Let's use an example to demonstrate. First, we create a DataFrame of product sales.

```
[55]: Products = ['Computers', 'Phones', 'Shoes', 'Computers', 'Phones']
Sales = [2500, 3000, 1400, 2100, 2800]

# Creating a DataFrame
df = pd.DataFrame({"Products": Products, "Sales": Sales})
# viewing the first 2 rows
df.head(2)
```

```
[55]:      Products  Sales
0  Computers    2500
1    Phones     3000
```

Let's say we want to know how much revenue each product brings in. We can use the `groupby()` method to group the data by product and sum the sales values. See below:

```
[56]: # Grouping products by the "Products" column and summing the sales
groupby_products = df.groupby("Products")["Sales"].sum()
groupby_products
```



```
[56]: Products
      Computers    4600
      Phones       5800
      Shoes        1400
      Name: Sales, dtype: int64
```

Using the `groupby()` method is important when you have duplicate values in a column, e.g., duplicate products. Note that the `groupby()` method requires that an aggregate (mean, sum, count, min, max, etc.) function be used for the function to return a value. In the code above, if we did not use the `sum()` method, our code would return an object. See below:

```
[57]: groupby_products = df.groupby("Products")["Sales"]
groupby_products
```



```
[57]: <pandas.core.groupby.generic.SeriesGroupBy object at 0x00000256DB7A64A0>
```

2.5.6 merge()

This merge is used to join two DataFrames on a common column or index. Let's say we have two DataFrames. Here is the first DataFrame. This DataFrame has sales.

```
[58]: Products = ['Computers', 'Phones', 'Shoes']
Sales = [2500, 3000, 1400]

# Creating a DataFrame
df_sales = pd.DataFrame({"Products": Products, "Sales": Sales})
df_sales
```


	Products	Sales
0	Computers	2500
1	Phones	3000
2	Shoes	1400

Here is the other DataFrame, which holds costs.

```
[59]: Products = ['Computers', 'Phones', 'Shoes']
costs = [1800, 2300, 1000]

# Creating a DataFrame
df_costs = pd.DataFrame({"Products": Products, "Costs":costs})
df_costs
```

```
[59]:   Products  Costs
0  Computers    1800
1    Phones     2300
2      Shoes    1000
```

Now, if we want to merge the two DataFrames, we can use the `merge()` method. The common column between the two DataFrames is the "Products" column. We will merge the two DataFrames on this column.

```
[60]: # Merging the two DataFrames
merged_df = df_sales.merge(df_costs, how = 'left',
                           on = "Products")
merged_df
```

```
[60]:   Products  Sales  Costs
0  Computers    2500   1800
1    Phones     3000   2300
2      Shoes    1400   1000
```

You can see above that the two columns have been merged, and we have the "Sales" and "Costs" columns in one DataFrame.

2.6 Selecting Data

One of the main things you'll be doing in pandas is selecting and filtering data. Here are some of the most important functions for selecting data:

2.6.1 .loc

The **.loc** attribute is used to select rows and columns by label. You can use it to select specific rows or columns or to slice the DataFrame. If we want to select the first three rows from the DataFrame above, here is how we can do it using the **.loc** attribute.

```
[61]: names = ['Alice', 'Bob', 'Charlie', 'David']
      age = [25, 32, 18, 47]
      salary = [50000, 80000, 20000, 120000]

      df = pd.DataFrame({"Names": names, "Age":age, "Salary": salary})
      df
```

```
[61]:    Names   Age  Salary
      0   Alice    25    50000
      1     Bob    32    80000
      2  Charlie   18    20000
      3   David    47   120000
```

```
[62]: # Select rows by Label
      first_three_rows = df.loc[1:3]
      first_three_rows
```

```
[62]:    Names   Age  Salary
      1     Bob    32    80000
      2  Charlie   18    20000
      3   David    47   120000
```

We can also use the **.loc** attribute to select specific rows and columns from a DataFrame. Let's say we want to select the names Alice and David and their salaries from the DataFrame.

```
[63]: alice_and_david = df.loc[[0,3],['Names', 'Salary']]  
alice_and_david
```

```
[63]:    Names   Salary  
0 Alice    50000  
3 David   120000
```

We can also filter the DataFrame by a specific value in a column. Let's say we want to slice the DataFrame to return rows of everyone over 30 years. Here is how we do it:

```
[64]: over_30 = df.loc[df["Age"] > 30]  
over_30
```

```
[64]:    Names   Age   Salary  
1 Bob     32    80000  
3 David   47   120000
```

2.6.2 .iloc

The `.iloc` attribute is used to select rows and columns by index position. It works similarly to the `.loc` attribute but uses the index instead of labels. It works similarly to slicing lists. If we wanted to slice the first two rows of the DataFrame, that is, Alice and Bob, we would write:

```
[65]: alice_and_bob = df.iloc[0:2]  
alice_and_bob
```

```
[65]:    Names   Age   Salary  
0 Alice   25    50000  
1 Bob     32    80000
```

This attribute can also be used to select specific rows and columns from a DataFrame. In the example below, we select the first two rows and the "Names" and "Salary" columns.

```
[66]: # Selecting specific rows and columns
alice_bob_salaries = df.iloc[:2,[0,2]]
alice_bob_salaries
```

```
[66]:   Names  Salary
0  Alice    50000
1    Bob    80000
```

2.7 Pandas Data Visualization Functions

Pandas also provides several functions for visualizing data which works well with the matplotlib library. Here are some of the plots we can create with pandas:

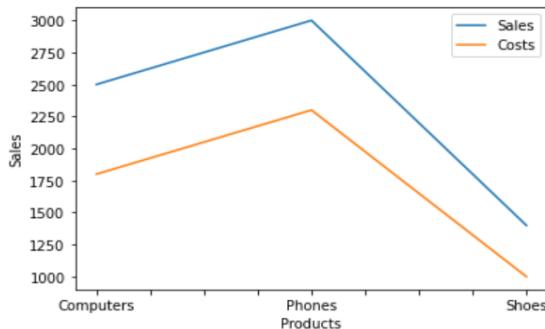
2.7.1 Line Plot

Using the pandas `plot()` method, we can create plots such as line plots, scatter plots, and histograms. By default, this method will create a line plot. Let's use the merged DataFrame from the previous section to demonstrate:

```
[67]: import matplotlib.pyplot as plt

# Merging the two DataFrames
merged_df = df_sales.merge(df_costs, how = 'left',
                           on = "Products")

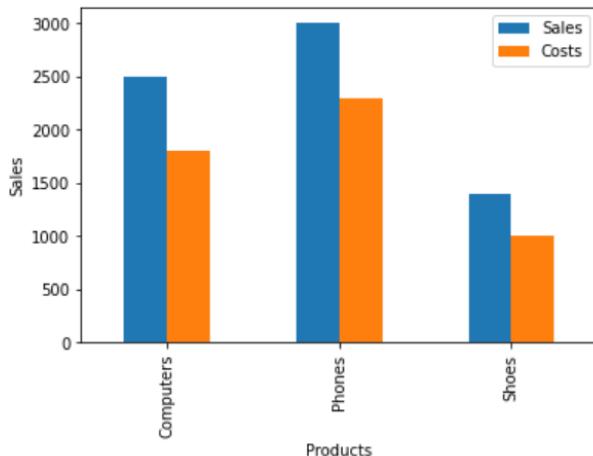
# Plotting a Line plot
merged_df.plot(x = 'Products', y = ["Sales", "Costs"], )
plt.xlabel("Products")
plt.ylabel("Sales")
plt.show()
```



2.7.2 Bar Plot

We can also create a **bar** plot using the **plot()** method. We will pass the "bar" argument to the "kind" parameter. We will use the bar plot to visualize the relationship between sales and costs for each product.

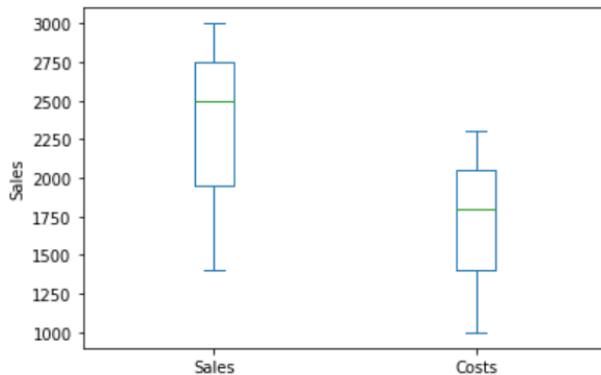
```
[68]: # Plotting a bar plot
merged_df.plot(kind = "bar",
               x = 'Products',
               y = ["Sales", "Costs"], )
plt.xlabel("Products")
plt.ylabel("Sales")
plt.show()
```



2.7.3 Box Plot

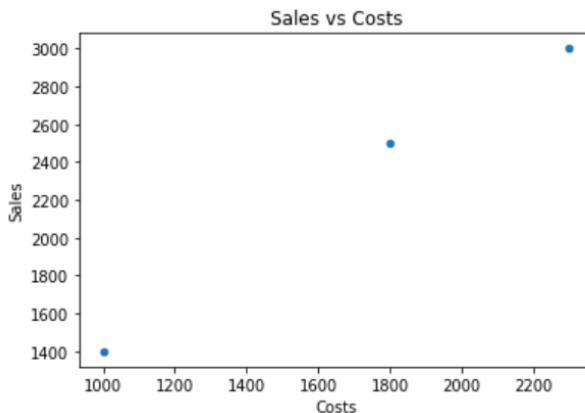
It is also easy to create a **box plot** with the pandas **plot()** method (Check the Seaborn section to learn how to interpret boxplots). We will pass "box" to the kind parameter. This will create a **box plot** of the "sales" and "costs" columns.

```
[69]: merged_df.plot(kind = "box",
                     x = "Products",
                     y = ["Sales", "Costs"])
plt.ylabel("Sales")
plt.show()
```



We can also create a **scatter** plot of two variables in a DataFrame. We will create a scatter plot of the "Sales" column and the "Costs" column. Since we only have three rows in our DataFrame, our plot will have 3 data points.

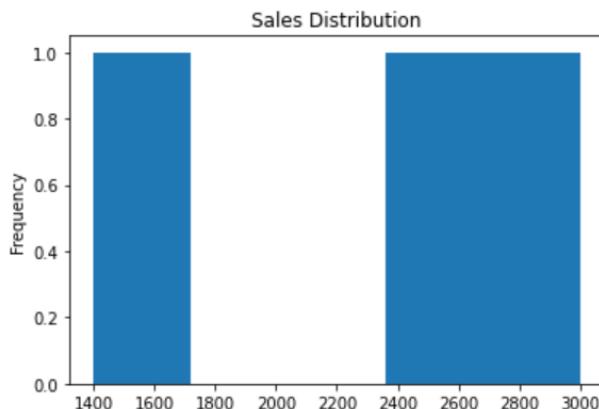
```
[70]: # Plotting a Scatter plot
merged_df.plot(kind = "scatter", x = "Costs", y = "Sales")
plt.title("Sales vs Costs")
plt.show()
```



2.7.4 Hist Plot

To view the distribution of data, we can use the pandas **plot()** function to create a **histogram** plot. Here is an example of the histogram plot in the sales column:

```
[71]: #plotting a hist plot  
merged_df["Sales"].plot(kind = "hist", bins = 5)  
plt.title("Sales Distribution")  
plt.show()
```



These are just a few plots that you can generate with pandas.

2.8 Sorting Data

The pandas library provides various functions to sort data in a DataFrame or a Series. These sorting functions can be used to sort data in ascending or descending order based on one or more columns. Let's look at some of the most commonly used sorting functions in pandas:

2.8.1 sort_values()

This method is used to sort the data in a DataFrame or a Series in ascending or descending order based on one or

more columns. The `by` parameter is used to specify the column(s) to sort by.

In the code below, we sort the DataFrame in ascending order by the "Sales" column.

```
[72]: Products = ['Computers', 'Phones', 'Shoes', 'Computers', 'Phones']
       Sales = [2500, 3000, 1400, 2100, 2800]

       # Creating a DataFrame
       df = pd.DataFrame({'Products': Products, "Sales":Sales})

       # Sorting the DataFrame by the sales column
       df_sorted = df.sort_values(by='Sales')
       df_sorted
```

	Products	Sales
2	Shoes	1400
3	Computers	2100
0	Computers	2500
4	Phones	2800
1	Phones	3000

By default, the `sort_values()` method sorts data in ascending order. If we want to sort data in descending order, we can set the `ascending` parameter to `False`.

2.8.2 sort_index()

This function is used to sort the data in a DataFrame or a Series based on the index. The `ascending` parameter is used to specify whether to sort in ascending or descending order. We are going to sort the DataFrame in descending order on the index. The `inplace` argument to `False` means that a new DataFrame of sorted data will be created. Here is the code below:

```
[73]: Products = ['Computers', 'Phones', 'Shoes', 'Computers', 'Phones']
Sales = [2500, 3000, 1400, 2100, 2800]

# Creating a DataFrame
df = pd.DataFrame({"Products": Products, "Sales":Sales})

# Sorting the DataFrame by the index
df_sorted_by_index = df.sort_index(ascending=False, inplace=False)
df_sorted_by_index
```

	Products	Sales
4	Phones	2800
3	Computers	2100
2	Shoes	1400
1	Phones	3000
0	Computers	2500

2.8.3 nsmallest and nlargest()

Let's say we want to return the two least profitable products from the DataFrame. Here is how we can do it with the **nsmallest()** function: We will pass the number of rows (2) we want to return and the column we want to sort ("Sales").

```
[74]: # Creating a DataFrame
df = pd.DataFrame({"Products": Products, "Sales":Sales})

# Getting two products with the Least sales
df.nsmallest(2, "Sales")
```

	Products	Sales
2	Shoes	1400
3	Computers	2100

We can do a similar thing if we want to return the two most profitable products from the DataFrame. This time, we will use the **nlargest()** function.

```
[75]: # Creating a DataFrame
df = pd.DataFrame({"Products": Products, "Sales":Sales})

# Getting two products with the Least sales
df.nlargest(2, "Sales")
```

	Products	Sales
1	Phones	3000
4	Phones	2800

3.0 Matplotlib

Matplotlib is a widely used data visualization library in Python that provides many powerful and flexible functions to create a wide range of plots and graphs. We are going to explore some of the essential Matplotlib plots used in visualization.

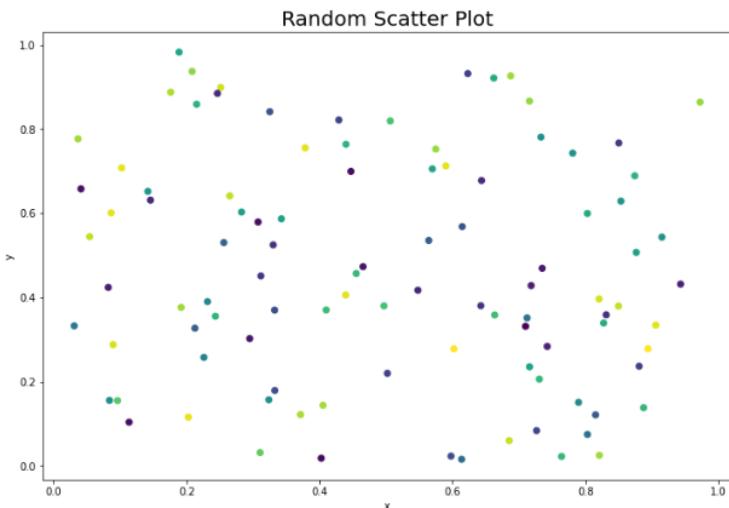
3.1.1 plt.scatter()

With Matplotlib, you can easily create scatter plots using the `plt.scatter()` function. The function is used to create scatter plots, where individual data points are represented as dots. It takes `x` and `y` values as input. Below, we generate random data using NumPy and use `plt.scatter()` to plot a scatter plot.

```
[76]: import matplotlib.pyplot as plt
import numpy as np

# Generating data using NumPy
rng = np.random.default_rng(seed = 24)
x = rng.random(100)
y = rng.random(100)
colors = rng.random(100)

# Plotting data
plt.figure(figsize=(12, 8))
plt.scatter(x, y, c = colors, alpha = 1.0)
plt.title('Random Scatter Plot', fontsize = 20)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



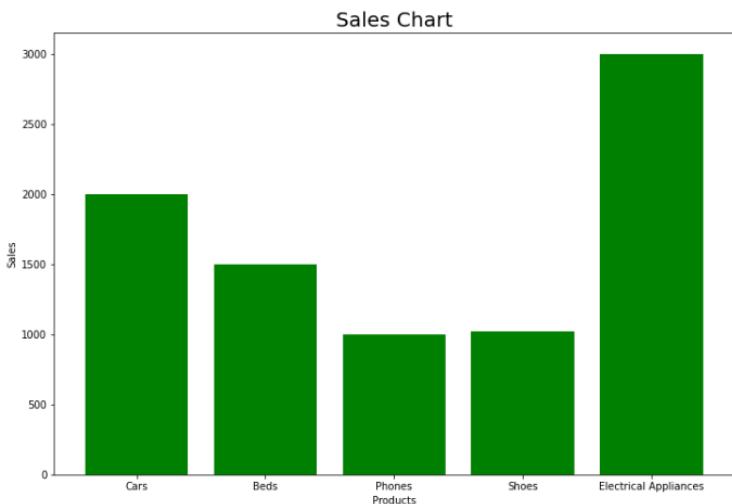
3.1.2 plt.bar()

This function is used to create bar plots. Bar plots require two values, **x** and **y**. The **x** value represents the categories, and the **y** value represents the values to be plotted. Here is a simple demonstration of how you can plot a bar plot with Matplotlib:

```
[77]: products = ['Cars', 'Beds', 'Phones', 'Shoes', 'Electrical Appliances']
sales = [2000, 1500, 1000, 1020, 3000]

plt.figure(figsize=(12, 8))
plt.bar(products, sales, color = "green")
plt.title('Sales Chart', fontsize= 20)
plt.xlabel('Products')
plt.ylabel('Sales')
plt.show()
```

In this example, the **products** list represents the x-values (categories), and the **sales** list represents the corresponding y-values. The **plt.bar()** function is used to create the bar plot using these values. The axes and adding a title, is done using **plt.title()**, **plt.ylabel()**, and **plt.xlabel()**.

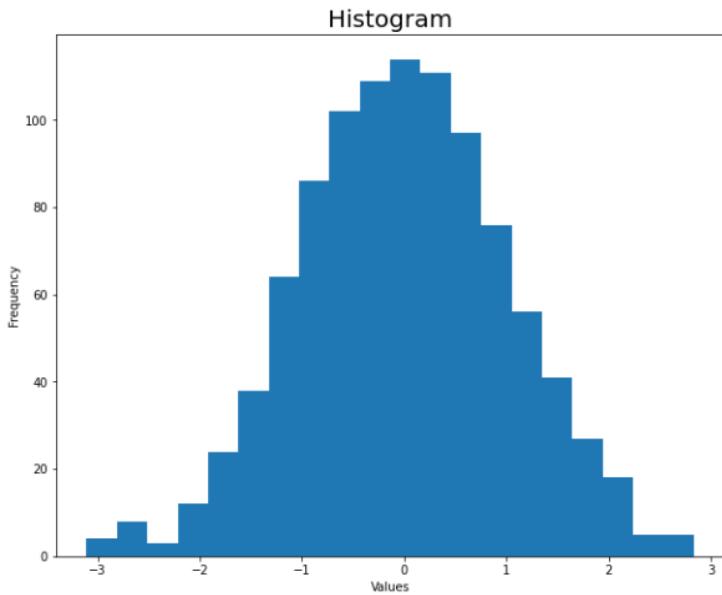


3.1.3 plt.hist()

To plot a distribution of continuous data, we can use the `plt.hist()` function. This function creates histograms, which represent the distribution of a set of continuous data. It takes a single array of values as input. Below, we generate standard normal distribution data using NumPy and create a histogram plot with Matplotlib.

```
[78]: # generating data using NumPy
rng = np.random.default_rng(seed = 24)
x = rng.standard_normal(1000)

plt.figure(figsize=(10,8))
plt.hist(x, bins=20)
plt.title('Histogram', fontsize = 20)
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.show()
```



3.1.4 plt.imshow()

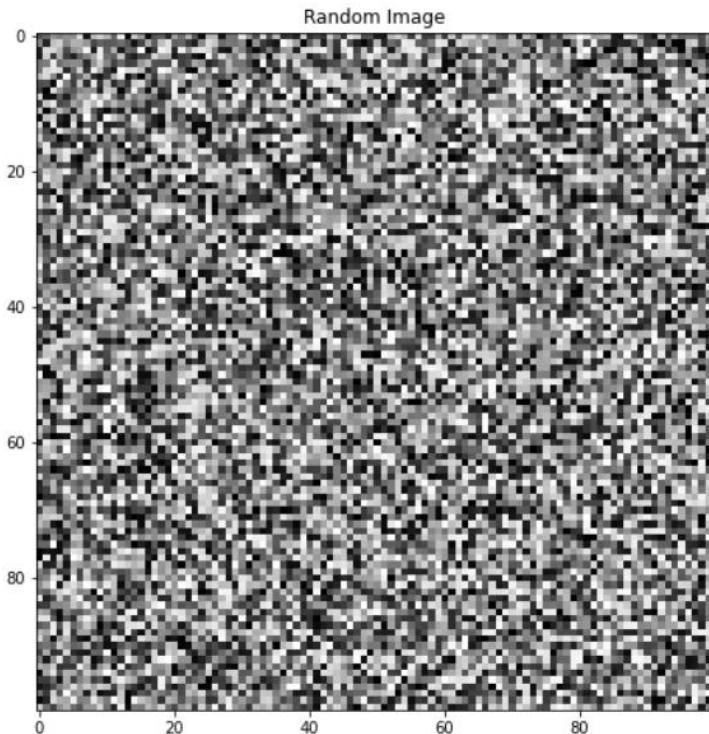
This function is used to display images. It takes a 2D array or a 3D array (for color images) as input. Below, we generate a black-and-white image using random data generated from NumPy.

```
[79]: rng = np.random.default_rng(seed = 24)
img = rng.random((100, 100))

plt.figure(figsize=(10,8))
plt.imshow(img, cmap='gray')
plt.title('Random Image')
plt.show()
```

In this example, a random 2D array of shape (100, 100) is generated, representing the grayscale intensities of each pixel. The `plt.figure()` set the size of the graph, The `cmap='gray'` parameter is used to display the image in grayscale. We customize the plot by adding a title. Finally,

`plt.show()` is used to display the image. See the output below:

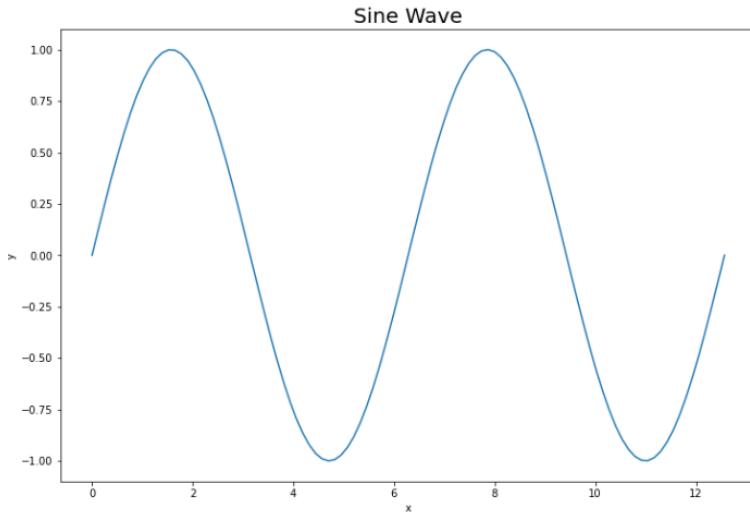


3.1.5 plt.plot()

This function is used to plot lines and/or markers to represent data. It takes x and y values as input and creates a line plot by default. In the code below, `x` and `y` are the arrays of values representing the data points. These arrays should have the same length.

```
[80]: import numpy as np
import matplotlib.pyplot as plt

# Generate an array of values from 0 to 4π with 100 evenly spaced points
x = np.linspace(0, 4*np.pi, 100)
# Compute the sine of each value in x
y = np.sin(x)
# Create a new figure for plotting and set its size to (12, 8)
plt.figure(figsize=(12, 8))
# Plot the x and y values on the figure
plt.plot(x, y)
# Set the title of the plot as 'Sine Wave' with font size of 20
plt.title('Sine Wave', fontsize=20)
# Label the x-axis as 'x' and y-axis as 'y'
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



These are just a few examples of the many Matplotlib functions available for data visualization. By using these functions in combination with NumPy and pandas functions, it is possible to create complex and informative visualizations that can help gain insights from data.

4.0 Seaborn

The Seaborn library is another popular library for visualizations in Python. It is also an open-source library. Seaborn is built on top of the Matplotlib library; as such, it integrates well with the functions of Matplotlib as well as the pandas library. The thing I like about Seaborn is that with just a few lines of code, we can make complex and meaningful plots for analyzing data. In this section, we are going to explore some of the essential functions of Seaborn that are frequently used in data analysis. Before we dive in, let's load the data that we are going to use for this section:

```
[81]: products = ['Cars', 'Beds', 'Phones', 'Shoes',
                 'computers', 'clothes', 'Books', 'Boats',
                 'Car oil', 'Scooter']
categories = ['Transportation', 'Household', 'Electronics',
              'Personal Items', 'Electronics','Personal Items',
              'Personal Items','Transportation', 'Transportation','Transportation' ]
sales = [2000, 1500, 1000, 1020, 3000, 1000, 1300, 1000, 2500]
costs = [1500, 1000, 700, 800, 2500, 700, 700, 1000, 900, 1100]
profit = [500, 500, 300, 220, 500, 300, 500, 300, 100, 1400]

# Creating a DataFrame from the four Lists above
df = pd.DataFrame({"Products": products, "Categories":categories,
                     "Sales":sales, "Costs": costs, "Profit": profit})
df.head()
```

	Products	Categories	Sales	Costs	Profit
0	Cars	Transportation	2000	1500	500
1	Beds	Household	1500	1000	500
2	Phones	Electronics	1000	700	300
3	Shoes	Personal Items	1020	800	220
4	computers	Electronics	3000	2500	500

We have created a DataFrame from five lists. Now let's analyze this data using some of the popular Seaborn functions:

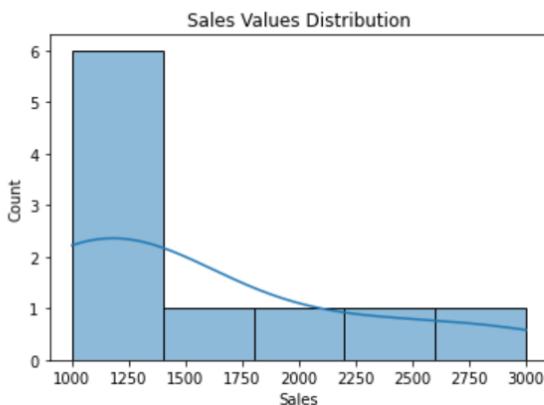
4.1.1 histplot()

The **histplot()** function plots a histogram of the data. This function is useful if you want to visualize the distribution of numerical data. Here is how we can use the function to

visualize the distribution of the "Sales" values from the DataFrame. First, we will import Seaborn as sns (it is convention to import Seaborn like that) and Matplotlib. We will use the `histplot()` function from Seaborn to create the plot. See below:

```
[82]: import seaborn as sns
import matplotlib.pyplot as plt

sns.histplot(df['Sales'], kde=True)
plt.title("Sales Values Distribution")
plt.show()
```

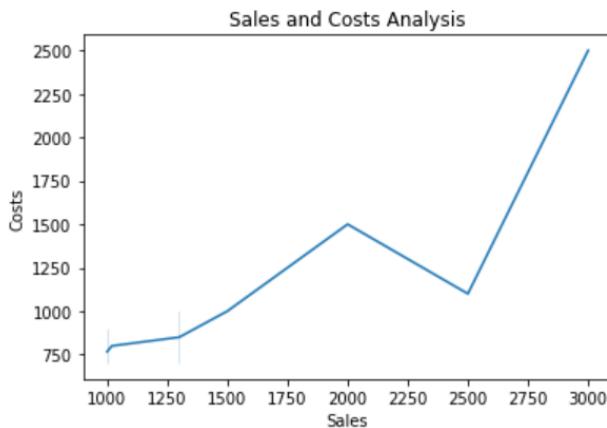


You can see from the plot that the most frequent sales values appear to be between 1000 and 1250. The sales appear to be right-skewed (positively skewed), as the majority of the sales values are concentrated on the lower end (closer to 1000–1250), with a long tail stretching towards higher values (2500–3000). The KDE line (in blue) helps smooth out the distribution, reinforcing the observation that the bulk of the data points lie on the left side (around 1000–1250) and taper off gradually. You can see from the code above that Seaborn integrates with Matplotlib.

4.1.2 lineplot()

Seaborn has a **lineplot()** function. The **lineplot()** function is used to visualize the relationship between two continuous variables. It helps in identifying trends, patterns, and potential correlations between the variables. Let's use it to see if there is a potential correlation between the "Sales" and "Costs" columns in the data. We are going to pass "Sales" to the x-axis and "Costs" to the y-axis. See the plot below:

```
[83]: # Creating a Line plot
sns.lineplot(x=df['Sales'], y=df["Costs"], data=df)
plt.title("Sales and Costs Analysis")
plt.show()
```



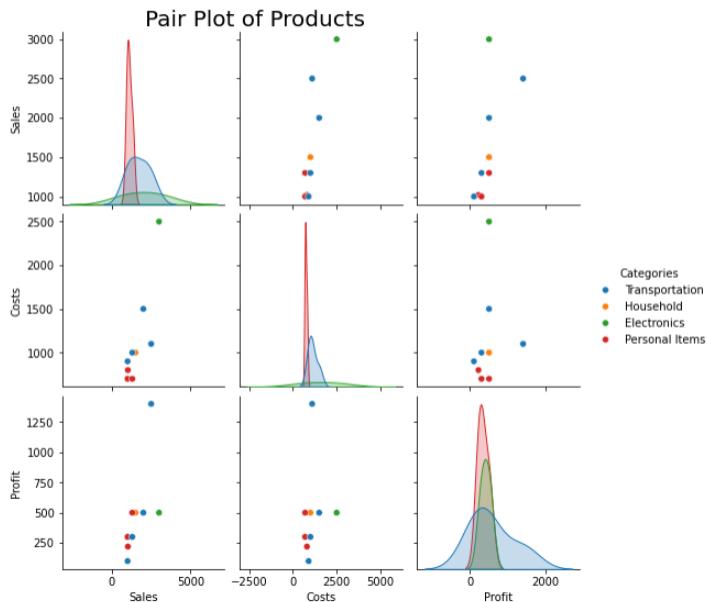
From the plot, we can conclude that the relationship between sales and costs is nonlinear. Costs initially increase gradually with sales, but there are significant jumps at certain sales values.

4.1.3 pairplot()

The Seaborn **pairplot()** allows you to visualize pairwise relationships in a dataset, especially useful for exploring correlations. Now let's use it to visualize the relationships

between multiple variables in the dataset, along with the distribution of each variable:

```
[84]: sns.pairplot(df, hue="Categories")
plt.title("Pair Plot of Products", y=3.10, x=-1, fontsize = 20)
plt.show()
```



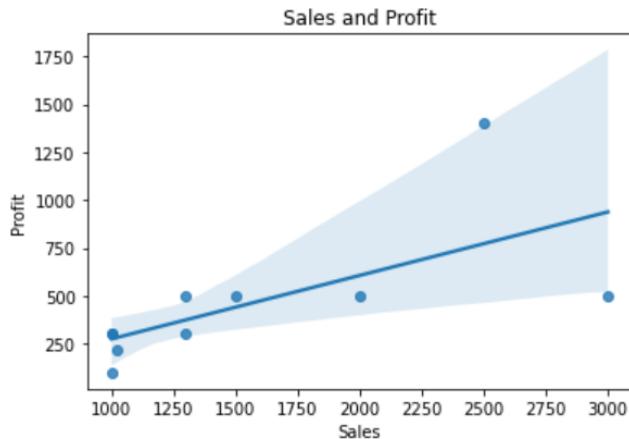
In this plot, we see pairwise relationships between variables and the role that different product categories play in shaping those relationships. The transportation category dominates across all metrics (sales, costs, profit).

4.1.4 regplot()

Another important Seaborn function is the **regplot()** function. This function is used to visualize the relationship between two numeric variables and fit a linear regression model to the data. It is particularly useful because it combines a scatter plot and a regression line in a single plot, providing a clear visual representation of the linear relationship. Let's look at an

example. Let's look at the relationship between 'Sales' and 'Profit' columns.

```
[85]: sns.regplot(x=df['Sales'], y=df["Profit"], data=df)
plt.title("Sales and Profit")
plt.show()
```

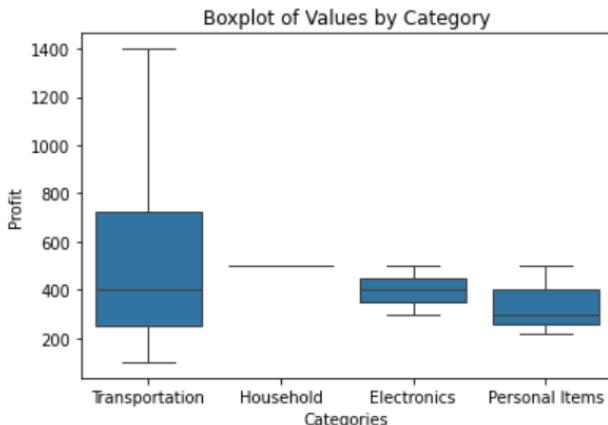


From the graph, we can generally conclude that there appears to be a positive linear relationship between sales and profit. As sales increase, profit tends to increase as well. This is indicated by the upward slope of the regression line. We can also see some outliers (data points that are significantly different from the rest), that is, sales of 2500 and 3000.

4.1.5 `boxplot()`

For statistical visualization and a summary of a dataset's distribution, Seaborn has the `boxplot()` function. This function is particularly useful for comparing distributions across different categories or groups. It is also perfect for catching outliers in categories. Let's look at an example:

```
[86]: sns.boxplot(x='Categories', y='Profit', data=df)
plt.title("Boxplot of Values by Category")
plt.show()
```



Here is how to read the plot: The line inside the box represents the median (the 50th percentile) of the data. The bottom part of the blue box represents the 25th quartile of data, and the top part of the blue box is the 75th quartile of the data in the category. The whiskers (the line extending lower and upper from the box) represent data points that lie within 1.5 IQRs of the lower and upper quartiles. Anything outside the whiskers is considered to be an outlier of the data. From this plot, we can see that the biggest junk of profit is from the transportation category.

5.0 Scikit-Learn

Scikit-learn (sklearn) is another important Python library for machine learning and data analysis. It provides simple and efficient tools for data analysis and building machine learning models. Let's look at some of the important functions for data analysis:

5.1.1 SimpleImputer()

This function is useful for preprocessing data. We can use it to deal with missing values in the dataset. This makes it a good alternative to pandas for this task. Let's create a simple dataset with some missing data:

```
[87]: import numpy as np
import pandas as pd

names = ["Joe", np.nan, "Ken", "Jos", "Luke"]
exam_marks = [120, np.nan, 100, np.nan, 85]

df = pd.DataFrame({"names":names,"marks": exam_marks})
df
```

```
[87]:      names  marks
          0     Joe  120.0
          1     NaN    NaN
          2     Ken  100.0
          3     Jos    NaN
          4     Luke   85.0
```

We have a dataset with some missing data in the "names" and "marks" columns. Now we are going to use the **SimpleImputer()** to fill the missing values. First, we will import **SimpleImputer** from the Sklearn library:

```
[88]: from sklearn.impute import SimpleImputer

# Creating an imputer to replace missing values with the mean for numeric column
imputer_numeric = SimpleImputer(missing_values=np.nan, strategy='mean')

# Creating an imputer to replace missing values with 'Rob' for non-numeric column
imputer_non_numeric = SimpleImputer(missing_values=np.nan, strategy='constant',
                                      fill_value='Rob')

# Fitting the imputer on the numeric 'marks' column
df[['marks']] = imputer_numeric.fit_transform(df[['marks']])

# Fitting the imputer on the non-numeric 'names' column
df[['names']] = imputer_non_numeric.fit_transform(df[['names']])
df
```

	names	marks
0	Joe	120.000000
1	Rob	101.666667
2	Ken	100.000000
3	Jos	101.666667
4	Luke	85.000000

Here we are using **SimpleImputer** with **strategy='mean'** to fill missing values in the 'marks' column with the **mean** of the existing values. For the missing value in the 'names' column, we use **SimpleImputer** with **strategy='constant'** and **fill_value='Rob'** to fill missing values in the 'names' column with the name 'Rob'. After this, we fit the imputer by applying the **fit_transform** method to transform the respective columns and updating the original DataFrame with the imputed values. You can see in the output that the missing data has been updated with values.

5.1.2 LabelEncoder()

The Sklearn **LabelEncoder()** is another preprocessing function that is used to convert categorical data into numeric labels. Why convert categorical data into numerical labels? Most machine learning models operate on numerical data, so it's necessary to represent categorical variables (e.g., "male" or "female," "red" or "blue") in a numerical format. Let's use it to convert the 'names' column into a numeric data type:

```
[89]: from sklearn.preprocessing import LabelEncoder

# Initializing the LabelEncoder
le = LabelEncoder()
# Apply LabelEncoder to the 'names' column
df['names'] = le.fit_transform(df['names'])
df
```

```
[89]:      names      marks
0        0  120.000000
1        4  101.666667
2        2  100.000000
3        1  101.666667
4        3   85.000000
```

You can see that the 'names' column is now in numerical labels. So, **LabelEncoder** encodes categorical labels as integer values. The transformed data is a single column of integers.

5.1.3 OneHotEncoder()

This is another data preprocessing function in Sklearn. Unlike the **LabelEncoder** that encodes categorical labels as integers, **OneHotEncoder** encodes categorical values as a set of binary columns (one column for each category). Each category is represented as a binary vector where only one element is 1 (indicating the presence of that category), and the rest are 0. Let's use it in the 'names' column. But first, we will load our dataset again:

```
[90]: import numpy as np
import pandas as pd

names = ["Joe", "Rob", "Ken", "Jos", "Luke"]
exam_marks = [120, 101, 100, 101, 85]

df = pd.DataFrame({"names":names,"marks": exam_marks})
df
```

	names	marks
0	Joe	120
1	Rob	101
2	Ken	100
3	Jos	101
4	Luke	85

Now let's use the **OneHotEncoder** to transform the 'names' column. We are going to import the encoder and then initialize it. We will set the **sparse_output** parameter to False to ensure that the output is a dense NumPy array instead of a sparse matrix. We will use the **fit_transform()** method to fit the encoder to the "names" column. Here is the code below:

```
[91]: from sklearn.preprocessing import OneHotEncoder

# Initializing encoder
encoder = OneHotEncoder(sparse_output=False)

# Fit and transform the data
one_hot_encoded = encoder.fit_transform(df[['names']])

# Converting into DataFrame for better readability
encoded_df = pd.DataFrame(one_hot_encoded, columns=encoder.categories_[0])
encoded_df
```

	Joe	Jos	Ken	Luke	Rob
0	1.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	1.0
2	0.0	0.0	1.0	0.0	0.0
3	0.0	1.0	0.0	0.0	0.0
4	0.0	0.0	0.0	1.0	0.0

You can see in the output that for each row in the 'names' column, the encoder places 1 in the column that corresponds to the original categorical value and 0 in the others. For example, Joe is in the first row (row 0) of the 'names' column, so in the 'Joe' column of the encoded DataFrame, row zero is given a 1, and the rest of the rows are given zeros (0). This is repeated for the rest of the names. So, unlike **LabelEncoder**, which inputs a single column, **OneHotEncoder** outputs multiple columns, each representing a category with 1 or 0.

5.1.4 StandardScaler()

Machine learning algorithms or models can be sensitive to the scale of data. This means that the model is likely to be biased towards the large values in the data. Consider a dataset with two features: 'Age' (ranging from 18 to 65) and 'Salary' (ranging from 20,000 to 200,000). Without scaling, the algorithm might give more weight to the 'Salary' feature due to its larger magnitude, even if 'Age' is more relevant for the task at hand. To address this scale sensitivity, we can standardize features using **StandardScaler()**. The **StandardScaler** will standardize features to have a mean of 0 and a standard deviation of 1. Here is an example:

```
[92]: age = [23, 45, 35, 68, 55]
       salary = [50000, 80000, 45000, 55000, 65000]

       df = pd.DataFrame({"Age": age, "Salary":salary})
       df
```

	Age	Salary
0	23	50000
1	45	80000
2	35	45000
3	68	55000
4	55	65000

Here, we have a 'salary' column that has a much larger range of values (50,000 to 65,000) compared to the 'Age' column (23 to 68). This difference in scale can significantly impact the performance of machine learning algorithms. We are going to use the **StandardScaler()** to standardize the features to have a mean of 0 and a standard deviation of 1. First, we are going to import StandardScaler from **sklearn.preprocessing**:

```
[93]: from sklearn.preprocessing import StandardScaler

# Initialize the Scaler
scaler = StandardScaler()
# Standardize the data
scaled_data = scaler.fit_transform(df)

# Convert the scaled data back to a DataFrame
scaled_df = pd.DataFrame(scaled_data, columns=['Age', 'Salary'])
scaled_df
```

```
[93]:      Age    Salary
0   -1.425422 -0.725241
1   -0.012842  1.692228
2   -0.654924 -1.128152
3    1.463947 -0.322329
4    0.629240  0.483494
```

To standardize the data with **StandardScaler**, first we initialize the Scaler, and then we use the **fit_transform()** method to fit the scaler to the DataFrame and then transform the data. This process calculates the mean and standard deviation of each feature and standardizes the data accordingly. You can see in the output that when we convert the standardized data back into a DataFrame, we get an output of the two columns with scaled data. This is beneficial for machine learning algorithms that are sensitive to feature scales and is likely to improve model performance and convergence.

5.1.5 train_test_split()

Once we have the data ready for training, it is a common practice to split data into training and testing sets to evaluate how well a machine learning model performs. This is similar to training students on one set of material and then testing them on another set of material that they have not seen to test their understanding of the training. The `train_test_split()` function from scikit-learn makes this process easy. Using this function, we can split the data into training and testing sets. Here is the data:

```
[94]: feature1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
       feature2 = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
       target = [1, 0, 0, 0, 1, 1, 0, 1, 0, 0]

       df = pd.DataFrame({"feature1":feature1, "feature2":feature2, "target": target})
       df
```

	feature1	feature2	target
0	1	10	1
1	2	20	0
2	3	30	0
3	4	40	0
4	5	50	1
5	6	60	1
6	7	70	0
7	8	80	1
8	9	90	0
9	10	100	0

Here we have this simple dataset with 'feature1', 'feature2', and 'target' columns. This dataset is for demonstration purposes only. In the real world, a much larger dataset will be required for the Sklearn functions

that we are going to use on this data. The task is to train our model using 'feature1' and 'feature2' variables to predict the 'target' column. We want to use 80% of the data for training the model (which is 8 rows) and 20% (which is 2 rows) for testing how well the model has been trained. So, we are going to split the data using the Sklearn `train_test_split()` function:

```
[95]: from sklearn.model_selection import train_test_split

# Split data into X and y variables
X = df.drop(columns= ['target'])
y = df[ "target"]

# Split the data (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)

#View training and testing data:
X_train, X_test
```



```
[95]: (   feature1  feature2
      5          6        60
      0          1        10
      7          8        80
      2          3        30
      9         10       100
      4          5        50
      3          4        40
      6          7        70,
   feature1  feature2
      8          9        90
      1          2        20)
```

In this code, first we import the `train_test_split()` function from scikit-learn. We split the data into features and target columns. The features columns ('feature1 and feature2) are assigned to the 'X' variable (these will be used to predict the target column). The 'target' column is assigned to the 'y' variable. Once we have the 'X' and 'y' variables, we use the `train_test_split()` function to split the X and y into training and test sets (random split). The split is done by passing the X and Y variables to the `train_test_split()` function, and we specify the size of the test data as `test_size = 0.2`, which specifies that 20% of the data should be allocated to the testing set. The `random_state = 42` sets a random seed for

reproducibility. This ensures that the same data points will be assigned to the training and testing sets each time the code is run. You can see in the output that the first 8 rows in the output represent 80% of the data that will be used for training (`X_train`, `y_train`), and the last 2 rows are data for testing(`X_test`, `y_test`).

5.1.6 Classification with Sklearn

In the previous section, we split the data into training and testing sets. You may have noticed that the 'target' column contains binary values (0 or 1), indicating that this is a classification problem. In classification, we aim to predict a binary outcome, and Scikit-learn makes it easy to build models for this. For example, we can use the `DecisionTreeClassifier` to train a model with the data above. Think of the model as a student: we provide 80% of the material to train this student. This material contains questions (`X_train`) and answers (`y_train`). The student must learn how the questions relate to the answers. Then, when given new, unseen questions (`X_test`), the student should be able to predict the correct answers (`y_test`).

```
[96]: from sklearn.tree import DecisionTreeClassifier  
  
# Create and train the model  
clf = DecisionTreeClassifier(random_state=42)  
clf.fit(X_train, y_train)
```

```
[96]: ▾ DecisionTreeClassifier ⓘ ⓘ  
DecisionTreeClassifier(random_state=42)
```

So first, we create an instance of the `DecisionTreeClassifier` called `clf`. The parameter `random_state=42` ensures that the results are the same each time we run the code, making it reproducible. We use the `fit()` method to train the classifier with the training data: `X_train` and `y_train`. Think of the `fit()` method as the tool we use to train the

student. After the model is trained, we want to check how well the student has learned. We give the student the test questions (**X_test**) and let the model predict the answers. Then, we compare these predictions to the correct answers (**y_test**). To do this, we use the **score()** method, which measures how accurately the model's predictions match the actual answers. This is done using the 20% of the data that was set aside for testing. See the code below:

```
[97]: # Checking accuracy on test set  
accuracy = clf.score(X_test, y_test)  
print(f"Accuracy on test set: {accuracy:.2f}")  
  
Accuracy on test set: 0.00
```

The **score()** method calculates the accuracy of the model on the test data. Accuracy is the proportion of correct predictions compared to the total predictions. A score of 1.00 means the model correctly predicted every sample in the test set, while a score of 0.0 means the model got all predictions wrong. In this case, the accuracy is 0.0 because the model failed to learn the relationship between the features and the target. There are several possible reasons for this: the training data may have been insufficient (since we only have 10 rows of data), there may be no real relationship between the features and target, or the model itself may not be appropriate for this problem. It's important to remember that a perfect accuracy score is rare in real-world datasets, and low accuracy can point to the need for better data, a more suitable model, or further tuning.

5.1.7 accuracy_score()

If we want more flexibility, we can predict the values first and then use the **accuracy_score()** function to calculate the accuracy. Before we use it, we have to import it from **sklearn**:

```
[98]: from sklearn.metrics import accuracy_score

y_pred = clf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy with accuracy_score: {accuracy:.2f}")

Accuracy with accuracy_score: 0.00
```

Here first, we predict the values using the `predict` method, and then we pass them to the `accuracy_score`. You can see here that the `accuracy_score` is also scoring 1.00. Both `score()` and `accuracy_score()` compute the same accuracy if you use them in their default forms for classification tasks. However, the main difference between the `score()` and `accuracy_score()` is that the `accuracy_score()` allows more control. For example, you can manually get the predictions, adjust them if needed, and then pass them to `accuracy_score()`.

5.1.8 precision Score()

Analyzing a model using only one metric, like accuracy, is often not enough. When you evaluate a student on a test, you don't just look at their overall score—you also want to know how they performed on individual questions or topics. The same principle applies when evaluating a machine learning model. Accuracy shows how often the classifier makes correct predictions overall, but precision gives a more focused view: it measures how many of the predictions that were labeled as 'positive' are actually correct.

To clarify, imagine you're building a model to predict 'Yes' (positive) and 'No' (negative) outcomes. The `precision_score()` tells you how many of the 'Yes' predictions are truly 'Yes.' This is particularly useful when false positives (incorrect 'Yes' predictions) are more problematic than false negatives. For example, in medical diagnoses, you want to be confident that when the model predicts someone has a disease (a 'Yes' prediction), it is

truly correct. Let's now use the `precision_score()` to evaluate our model:

```
[99]: from sklearn.metrics import precision_score

# Calculating precision score
precision = precision_score(y_test, y_pred, average='binary',
                             zero_division=0.0)
print(f"Precision: {precision:.2f}")

Precision: 0.00
```

Here, we get a precision score of 0.0 because the model did not make any true positive predictions. This means the model didn't correctly identify any of the positive class labels (1). In other words, whenever the model predicted a 'positive' (1), it was wrong. This is expected in this case, as the model didn't learn the patterns well and made incorrect predictions for all instances, resulting in no true positives. Here is the formula for calculating `precision_score`:

Precision = True Positives / (True Positives + False Positives)

So when the True Positives are zero, we get a results of zero.

5.1.9 `recall_score()`

The `recall_score()` evaluates the model's ability to identify all actual positive instances in the data. It measures how many of the true positive cases (1s in the correct answer set) the model correctly predicted. In other words, **recall** tells us what percentage of all the actual positives the model was able to find.

For example, if there are 10 positive cases (true positives) in the data, and the model correctly identifies 7 of them, the recall would be 70%. This score is useful when missing a positive prediction is more critical, such as in medical testing where failing to identify a disease could be harmful. Here is the formula for recall:

recall = True Positives/(True Positives + False Negatives)

Let's use it to evaluate the model using **recall_score**:

```
[100]: from sklearn.metrics import recall_score

# Recall evaluation
recall = recall_score(y_test, y_pred, average='binary',
                      zero_division=0)
print(f'Recall: {recall:.2f}')
```

Recall: 0.00

Since our model made no true positive predictions, it's expected that the **recall** score is 0.0. In fact, all the predictions by the model are positive (1), which is incorrect because there are no True Positives in the result set. We can confirm this by examining the output of **y_pred**, which contains the model's predictions on the test data.

```
[101]: y_pred
```

```
[101]: array([1, 1], dtype=int64)
```

You can see here that the model is predicting only ones (1), while the actual labels in **y_test** are all zeros (see below). This mismatch explains why both the **recall score** and **precision score** are 0.0. Our model has incorrectly predicted all test samples as positive.

```
[102]: y_test
```

```
[102]: 8    0
       1    0
Name: target, dtype: int64
```

5.2.1 f1_score()

The **f1_score** is the harmonic mean of precision and recall, providing a balance between the two metrics. Here is the formula for the score:

$$\text{F1-score} = \frac{2 * (\text{precision} * \text{recall})}{(\text{precision} + \text{recall})}$$

Since precision and recall are returning zero, we should expect similar results from **f1_score**. See below:

```
[103]: from sklearn.metrics import f1_score

# F1-Score evaluation
f1 = f1_score(y_test, y_pred, average='binary',
               zero_division=0)
print(f"F1-Score: {f1:.2f}")

F1-Score: 0.00
```

5.2.2 Confusion_matrix()

Another useful tool for evaluating a classification model is the **confusion_matrix()** function. It provides a detailed summary of the model's performance by showing the number of True Positives, True Negatives, False Positives, and False Negatives. In essence, it summarizes the model's predictions across all classes. Let's use it to evaluate the model:

```
[104]: from sklearn.metrics import confusion_matrix

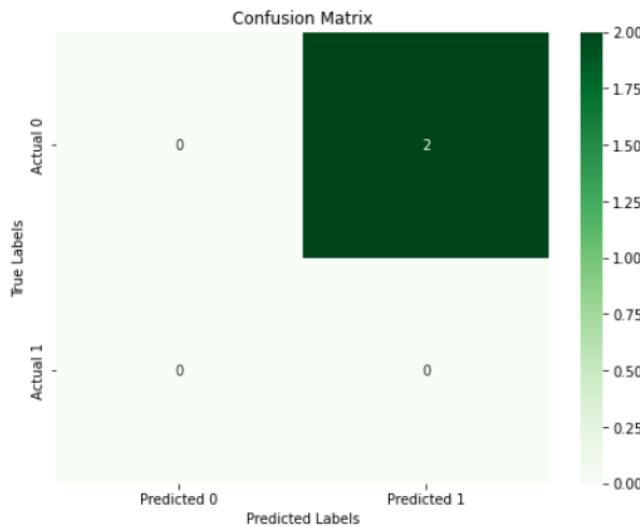
# Generating the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print(conf_matrix)

[[0 2]
 [0 0]]
```

The confusion matrix above shows that there are 2 False Positive predictions. This confirms that the model predicted the positive class (1) for both test samples, but those predictions were incorrect. There are no True

Positives, True Negatives, or False Negatives in this case, as the model consistently predicted the positive class. The best way to view the output of the confusion matrix is via a confusion matrix plot. Let's create one using the Seaborn heatmap:

```
[105]: # Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True,
            fmt="d", cmap="Greens",
            cbar=True,
            xticklabels=['Predicted 0', 'Predicted 1'],
            yticklabels=['Actual 0', 'Actual 1'])
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```



We can now clearly see the two False Positives predictions made by the model.

5.2.3 Regression with Sklearn

Sklearn also provides algorithms for regression problems. Unlike a classification that aims to predict a binary output, a regression predicts a continuous (numerical) value based on input data. For example, predicting a person's weight based on their height and age or predicting house prices based on features like the size of the house, number of rooms, and location. A regression problem follows similar steps to those of a classification problem. Sklearn provides many regression algorithms, such as Linear Regression, Ridge Regression, Lasso Regression, Decision Tree Regressor, and others. Let's look at a simple example on how to implement regression models with Scikit-learn, using **LinearRegression**. Let's load a simple dataset to demonstrate:

```
[106]: data = {  
    'Experience': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],  
    'Age': [25, 35, 35, 40, 42, 45, 50, 55, 60, 63],  
    'Salary': [45000, 50000, 60000, 65000, 70000, 75000,  
               80000, 85000, 90000, 95000]  
}  
df = pd.DataFrame(data)  
df
```

```
[106]:   Experience  Age  Salary  
0          1   25  45000  
1          2   35  50000  
2          3   35  60000  
3          4   40  65000  
4          5   42  70000  
5          6   45  75000  
6          7   50  80000  
7          8   55  85000  
8          9   60  90000  
9         10   63  95000
```

Here we have a simple dataset with two features: 'Experience' and 'Age'. We are going to use these columns to train the model to predict the 'Salary' column. First, we need to separate the data into training and testing data. We are going to do an 80-20 split (80 training and 20% testing).

```
[107]: from sklearn.model_selection import train_test_split

# Separate features (X) and target (y)
X = df[['Experience', 'Age']]
y = df['Salary']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=42)
```

Now that we have split the data, we can now use a **StandardScaler** to standardize the data. Remember, scaling ensures that all features contribute equally, especially when they are on different scales (age and experience).

```
[108]: from sklearn.preprocessing import StandardScaler

# Create a StandardScaler object to scale the features
scaler = StandardScaler()

# Fit the scaler on the training data test sets
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Here we use the **fit_transform()** method to learn the parameters necessary for the scaling of the data. The **transform()** method applies the transformation learned in a previous (**fit_transform()**) step to new data (test_data). Next, we can train the linear regression model on scaled data:

```
[109]: from sklearn.linear_model import LinearRegression

# Create a Linear Regression model
model = LinearRegression()

# Training the model on the scaled training data
model.fit(X_train_scaled, y_train)
```

```
[109]: ▾ LinearRegression ⓘ ⓘ
LinearRegression()
```

So we have successfully trained the model with scaled data. The next step is to evaluate the model.

5.2.4 Mean Squared Error (MSE)

To measure how well our model did, we can use the **Mean_squared_Error** from Sklearn. It calculates the average of the squared differences between the predicted and actual values. Generally, for this metric, a lower number indicates better model performance. Let's see how well the model is doing:

```
[110]: from sklearn.metrics import mean_squared_error

# Make predictions on the scaled test set
y_pred = model.predict(X_test_scaled)

# Evaluating the model
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
```

Mean Squared Error: 20529957.75375235

By looking at the magnitude of the output, we can conclude that our model predictions are off by a substantial margin on average.

5.2.5 Root Mean Squared Error (RMSE)

Since the target variable is salary, the error (MSE) is in squared dollars. This makes it difficult to interpret the

MSE directly in terms of the actual salary prediction error. To interpret the error in more meaningful units, we can use **Root Mean Squared Error** (RMSE). The Root Mean Squared Error (RMSE) measures how far off the model's predictions are from the actual values. Here is how it works:

- First, it finds the difference between each predicted value and the actual value. This tells us how far the prediction was from being correct.
- Then, it squares these differences so that all values are positive, making sure large errors stand out more.
- Next, it takes the average of all those squared differences.
- Finally, it takes the square root of that average to bring the units back to the original scale (like dollars, meters, etc.).

Now that you understand how it works, let's import it from Sklearn and use it to evaluate the model:

```
[111]: from sklearn.metrics import root_mean_squared_error

# Make predictions on the scaled test set
y_pred = model.predict(X_test_scaled)

# Evaluating the model using rmse
rmse = root_mean_squared_error(y_test, y_pred)
print(f"Root Mean Squared Error: {rmse}")
```

Root Mean Squared Error: 4530.999641773585

This output tells us that, on average, the model's predictions are off by approximately \$4530. This means if the salary is \$50,000, for example, the model may predict 54530 or 45469. This is still a pretty high error. In practice, you would typically aim to improve the model by adding more features, fine-tuning it, or considering a different model. However, this is easier to interpret because it's in the same units as the salary values.

Remember, the purpose of this demonstration is basically to show how you can use these Sklearn metrics to evaluate the performance of a model, so we will not concern ourselves much with the output.

5.2.6 R2_score

The **r2_score** is another useful Sklearn metric for evaluating how well the model is performing. It measures how much of the variation in the target variable (what you're trying to predict) can be explained by the features in the data. Here's a breakdown:

- **Perfect score:** This means the model's predictions perfectly match the actual values. The features used to train the model explain 100% of the target's behavior. The perfect score is 1.0.
- **Zero score:** This means the data features were not helpful at all in making predictions, and the model performs no better than simply predicting the average value of the target.
- **Negative values:** This means the model is performing even worse than simply guessing the average target value.

Basically, the closer the **r2_score** is to 1, the better the model is at making accurate predictions. Let's compute the **r2_score** for our model and see how well it performs:

```
[112]: from sklearn.metrics import r2_score

# Make predictions on the scaled test set
y_pred = model.predict(X_test_scaled)

# Evaluating the model using r2_score
r2 = r2_score(y_test, y_pred)
print(f"R-squared: {r2}")

R-squared: 0.9486751056156191
```

The output of 94% shows that model is doing a really good job of predicting the target (salary) based on the features (experience and age).

6.0 Final Thoughts

These are just some of the essential aspects of these libraries for data analysis. Each library offers a vast array of functions and capabilities, enabling you to perform sophisticated analyses efficiently. As you progress through the challenges in this book, you'll discover even more functions and features. I encourage you to adopt a spirit of curiosity and exploration as you encounter other powerful functions. Remember, mastering these tools will empower you to tackle real-world data problems with greater confidence and precision.

A word of caution: the libraries used in this book are constantly evolving, and future updates may cause some of the code to stop working as expected. While every effort will be made to keep the code up to date, it's always a good idea to stay informed about new releases and changes to these libraries.

Day 2: Creating and Manipulating Arrays

These challenges will help you solidify your understanding of NumPy arrays for data analysis in Python. By completing these tasks, you'll learn how to create arrays, how to check array size and shape, and how to manipulate arrays.

1. Write code to import NumPy into your script. Check the version of NumPy installed.
2. Here is a list below:
`list1 = [2, 3, 4, 6]`

Create a one-dimensional array from the list above. Write another line of code to check the **shape** and **size** of your array.

Write a function (Python function) that takes this array and returns a new array with all the values in the original array reversed.

3. `list1 = [2, 3, 4, 6]`
`list2 = [8, 10, 12, 14]`

Create an array from the **two** lists above. Then write code to check the shape, dimensions, and data type of the array.

4. `list1 = [2, 3, 4, 6]`
`list2 = [8, 10.1, 12, 14]`
`list3 = [16, 18, 20, 22.1]`

Create an array from the **three** (3) lists above. Take note that some of the lists contain floats. Create an array that will have an **int64** data type.

5. Is it possible to create an array of both floats and integers?

Day 2 - Answers

1. NumPy is not a Python built-in library, so it must be installed and then imported into the script. To install NumPy using pip, simply run: **pip install numpy**. This challenge asks that we import NumPy into our script. By convention, this is how you import NumPy into the script:

```
[1]: import numpy as np
```

To check the version of NumPy installed, you can use:

```
[2]: # Checking the version of NumPy  
np.__version__
```

```
[2]: '1.24.3'
```

This returns the version of NumPy on the machine at the time of running this code. Your code may return a different version. If you are using Conda, you can also check by typing:

conda list numpy

2. To create a one-dimensional array, we can use the **np.array()** function. We pass the list as an argument, and it is converted into an array. Here is the code below:

```
[3]: list1 = [2, 3, 4, 6]  
arr = np.array(list1)  
arr
```

```
[3]: array([2, 3, 4, 6])
```

We use the **shape** attribute to check the shape of the array.

```
[4]: # checking shape of array  
arr.shape
```

```
[4]: (4,)
```

The output shows that this is a one-dimensional array with four (4) items. A one-dimensional array simply means it has one axis.

We can use the `size` attribute to know the size of the array. This attribute will return the number of items in the array.

```
[5]: # Checking size of array
arr_size = np.size(arr)
print("The Number of Itemms in the array: ", arr_size)
```

The Number of Itemms in the array: 4

There are two ways you can reverse an array. The first way is by using a Python user-defined function. We define a function called `reverse_array` that takes an array and reverses it using `[::-1]`. You can see from the output that the array has been reversed.

```
[6]: # Using function to reverse array
def reverse_array(array):
    return array[::-1]

# Creating an array
arr = np.array(list1)

# Reversing array using function
arr_reversed = reverse_array(arr)
arr_reversed
```

[6]: array([6, 4, 3, 2])

Another way is to use the `np.flip()` function. This function will reverse an array along a given axis. Since our array has one axis, we pass `None` to the `axis` parameter. See below:

```
[7]: # Creating an array
arr = np.array(list1)

# Reversing array using flip
flip_arr = np.flip(arr, axis=None)
flip_arr

[7]: array([6, 4, 3, 2])
```

3. In the previous question, we created an array from a single list. Now we are required to create an array from two lists. So, how do we do it? We create an array using the **np.array()** function. This time, we pass the two lists as arguments. Since an array function can only take one argument, we enclose the two arrays in a list ([list1, list2]). We use the **np.shape** attribute to check the array shape, the **ndim** attribute to check the array dimensions, and **dtype** to check the array data type. Here is the code below:

```
[8]: list1 = [2, 3, 4, 6]
list2 = [8, 10, 12, 14]

# Creating array from two lists
arr = np.array([list1, list2])
arr
```

```
[8]: array([[ 2,  3,  4,  6],
           [ 8, 10, 12, 14]])
```

```
[9]: # Checking shape of the array
arr.shape
```

```
[9]: (2, 4)
```

```
[10]: # Checking dimensions of the array
arr.ndim
```

```
[10]: 2
```

```
[11]: # checking the dtype of array
arr.dtype
```

```
[11]: dtype('int32')
```

You can see from the outputs that the shape of the array will be $(2, 4)$, which means it has 2 rows and 4 columns. The dimensions of the array will be 2, indicating that it's a 2-dimensional array (arrays inside an array). The **dtype** (data type) is `int32`.

4. Since some of our lists have some floating-point numbers, when we create an array, the default data type will be `float64`. That is why we need to convert it to an `int64` data type. To change the **dtype**, we can create an array using the `np.array()` function and then use the `astype()` method to convert the array to the desired data type of `int64`.

```
[12]: list1 = [2, 3, 4, 6]
list2 = [8, 10.1, 12, 14]
list3 = [16, 18, 20, 22.1]

# Create an array from the three lists
arr = np.array([list1, list2, list3])

# Convert the array to int64 data type
arr = arr.astype(np.int64)
print(arr)

[[ 2  3  4  6]
 [ 8 10 12 14]
 [16 18 20 22]]
```

You can see from the output that our array has all integer numbers. This is because the `astype()` method has rounded the floats when converting them to `int64`.

Now, let's check our array **dtype**:

```
[13]: # Print array data type
print("New arr data type:", arr.dtype)

New arr data type: int64
```

Another method is to specify the data type when the array is created. The `array` function has a **dtype** parameter that we can

use to explicitly specify the data type of our array. Here is the code below:

```
[14]: list1 = [2, 3, 4, 6]
       list2 = [8, 10.1, 12, 14]
       list3 = [16, 18, 20, 22.1]

       arr = np.array([list1, list2, list3], dtype='int64')
       arr

[14]: array([[ 2,  3,  4,  6],
       [ 8, 10, 12, 14],
       [16, 18, 20, 22]], dtype=int64)
```

You can see in the output that the `dtype` is `int64` and the array has no floats in it.

5. In NumPy, an array is a homogeneous data structure, which means that it can only hold values of a single data type. Like we have seen above, if you have a list of floats and integers and you want to convert it to an array, by default it will be converted to the float data type. Here is an example:

```
[15]: # list of integers and floats
       lst = [1, 2.5, 3, 4, 2, 5]

       # Creating an array from a list of floats and integers
       my_array = np.array(lst)
       my_array.dtype

[15]: dtype('float64')
```

You can see that even though we have integers in the original list, the array has just one data type—`float64`.

Day 3: Generating Random Arrays

By tackling the challenges below, you will learn how to generate random numbers from a standard normal distribution, create arrays of a specific shape, randomly sample elements from an array, and create some visualizations using Matplotlib.

1. Using NumPy, create an array of random numbers between 0 and 1. The shape of the array must be (3, 4). Use **seed** to ensure that the results are reproducible.
2. Create an array of random integers between 0 and 10. The shape of the array must be (3, 4).
3. Using NumPy, generate an array of 1000 random numbers from a standard normal distribution with a **mean** of 0 and a **variance** of 1. Create a histogram with this array. Use the **seed** parameter to ensure the results are reproducible.
4. **arr = ["Orange", "Apple", "Pear"]**

Using **random.choice()** from NumPy, generate an array from the list above. The shape of the array must be (3, 4).

Day 3 - Answers

1. First, we import NumPy. We are going to use `np.random.default_rng` to construct a generator. We will use the generator to generate random numbers between 0 and 1. The shape of our array will be 3 rows and 4 columns. Now, when we run this code, we will get different random numbers every time. We will set the seed parameter to 42 to ensure that the results are reproducible.

```
[1]: import numpy as np

rng = np.random.default_rng(seed = 42)
arr = rng.random(size=(3, 4))
arr

[1]: array([[0.77395605, 0.43887844, 0.85859792, 0.69736803],
           [0.09417735, 0.97562235, 0.7611397 , 0.78606431],
           [0.12811363, 0.45038594, 0.37079802, 0.92676499]])
```

2. Now we are going to create an array of random integers between 0 and 10 with a shape of (3, 4). We are going to use `random.Generator.integers`. Here is how to implement this using NumPy.

```
[2]: rng = np.random.default_rng()
rng.integers(10, size=(3, 4))

[2]: array([[6, 8, 0, 5],
           [4, 6, 0, 1],
           [1, 8, 5, 8]], dtype=int64)
```

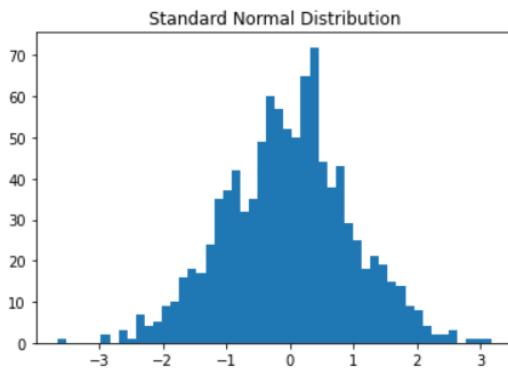
3. In this challenge, we will generate an array of random numbers from a standard normal distribution with a `mean` of 0 and variance of 1. We will use the `numpy.random.default_rng()` function to create a random generator with a specified seed to ensure reproducibility. Then, we will generate the random numbers using the `standard_normal()` method of the random generator. Finally,

we will use Matplotlib to plot a histogram of the generated values.

```
[3]: import matplotlib.pyplot as plt

# Instantiating a generator and setting seed
rng = np.random.default_rng(seed = 42)
arr2 = rng.standard_normal(size=(1000))

# plotting a hist plot
plt.hist(arr2, bins=50)
plt.title("Standard Normal Distribution")
plt.show()
```



A normal distribution will always return a bell-shaped histogram.

4. We are going to use the random generator from NumPy to generate an array from the list. We will use `numpy.random.Generator.choice` to randomly choose the elements from a given array. Here is the list below:

```
lst = ["orange", "apple", "Pear"]
```

We will set the shape to `(3, 4)`. Remember that because this is using random, the order of the elements will change every time you run the code. If we want the generated random state to remain constant, we can set the seed parameter (`seed = 24`). Here is the code below:

```
[4]: lst = ["Orange", "Apple", "Pear"]

# constructing a random generator
rng = np.random.default_rng(seed = 24)

arr = rng.choice(lst, size = (3, 4))
arr

[4]: array([['Apple', 'Orange', 'Pear', 'Apple'],
           ['Pear', 'Apple', 'Orange', 'Apple'],
           ['Apple', 'Apple', 'Apple', 'Apple']], dtype='|U6')
```

You can see that we have generated a 3x4 NumPy array containing randomly selected fruits from the list.

Day 4: NumPy Arrays and Vector Operations

In these challenges, you will create arrays and perform vector operations with them. These operations are an important feature of NumPy that makes it popular for scientific and numerical computing. Being able to perform vector operations is an important skill for all data analysts.

1. Create two arrays of random integers between 10 and 20. The shape of the array must be (2, 3) for the first array and (1, 3) for the second array. Check the shape of the arrays. Ensure that the results are reproducible.
2. Write a code to add the two arrays you just created in question 1. Check the shape of the resulting array. Explain why the resulting shape is that of the bigger array from the two arrays created in question 1.
3. Write another code to perform a dot operation on the arrays.
4. Using NumPy, create a 1-dimensional array of 100 random integers from 0 to 10. Use the array to create a histogram of the data and calculate the **median** and **mode**. Ensure that the results are reproducible.
5. Using NumPy, create a 3-dimensional array of 100 random floats between 0 and 1. Use the array to create a 3D scatter plot of the data. Ensure that your results are reproducible.
6. Create a an array from the list below:

lst =[[0, 1, 3, 0, 4], [8, 9, 0, 9, 6]]

Write code to flatten the list and return all non-zero numbers from the list.

Day 4 - Answers

1. In this challenge, we are required to create two arrays of random integers between 10 and 20 with the shapes (2, 3) and (1, 3). We are going to use the `np.random.default_rng()` function. It's worth noting that, due to the use of random numbers, running the code multiple times will produce different random values. However, by setting the seed value to 42, the generated random numbers will remain the same each time you run the code with the same seed.

```
[1]: import numpy as np

rng = np.random.default_rng(seed = 42)

array1 = rng.integers(low = 10, high = 21, size=(2, 3))
array2 = rng.integers(low = 10, high = 21, size=(1, 3))

print("Shape of array1:", array1.shape)
print("Shape of array2:", array2.shape)

Shape of array1: (2, 3)
Shape of array2: (1, 3)
```

2. In this challenge, we are required to add two arrays using the `np.add()` function. The `np.add()` function performs element-wise addition, adding the corresponding elements of the two arrays together. If the shapes of the arrays are different, the broadcasting rule is applied.

Under the broadcasting rule, the smaller array is expanded to match the shape of the larger array. In our case, the first array has a shape of (2, 3), and the second array has a shape of (1, 3). The smaller array (second array) will be replicated along the first axis to match the shape of the larger array (first array). The resulting shape after addition will be (2, 3).

```
[2]: array1

[2]: array([[10, 18, 17],
           [14, 14, 19]], dtype=int64)
```

```
[3]: array2  
[3]: array([[10, 17, 12]], dtype=int64)  
  
[4]: # Using numpy.add() function  
array3 = np.add(array1, array2)  
print(array3)  
[[20 35 29]  
 [24 31 31]]  
  
[5]: print(array3.shape)  
(2, 3)
```

3. This challenge requires that we perform a dot operation on the two arrays we created above. What is a dot operation? A dot operation, also known as a dot product or inner product, is a mathematical operation that takes two arrays and returns a **scalar value**. In the context of NumPy, the dot operation is performed using the `numpy.dot()` function. If the shape of the arrays are not the same, this operation will generate an error. The two arrays have different shapes (2, 3) and (1, 3). Now, for this operation to be performed on these arrays, the two inner numbers of the shape of the array must be equal. This basically means that if they are put side by side, the two arrays must look like this: (2, 3), (3, 1). You can see that the inner numbers for both arrays are 3. So, to achieve this, we must transpose the smaller array (array2). Here is the code below:

```
[6]: # Perform dot operation on the arrays  
dot_product = np.dot(array1, array2.transpose())  
dot_product  
  
[6]: array([[610],  
           [606]], dtype=int64)
```

Please note that your output will be different from the one above because we are using random numbers. However, the shape of the output array should be exactly the same.

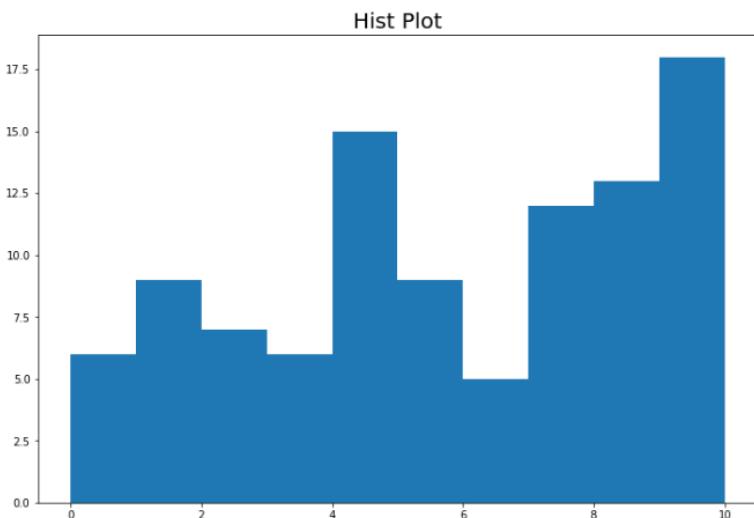
4. First, we will generate random data and plot a histogram plot using Matplotlib. We are going to use `random.Generator.integers()` to generate data. We will set the `seed` to ensure the results are reproducible. Since we are generating numbers to 10, we will pass 11 to the high parameter.

```
[7]: import matplotlib.pyplot as plt

# Generating data
rng = np.random.default_rng(seed = 42)
random_data = rng.integers(low = 0, high = 11, size=(100))

# Plotting the data
plt.figure(figsize=(12,8))
plt.hist(random_data, bins=10)
plt.title("Hist Plot", fontsize = 20)
plt.show()
```

The output of this will be:



Now let's calculate the median of the array using `np.median`. The median is the middle value of the data. We are going to pass the array as an argument.

```
[8]: # To calculate the median  
median = np.median(random_data)  
print("Median : ", median)
```

Median : 5.0

To calculate the mode, we are going to use **mode** from the **statistics** module.

```
[9]: from statistics import mode  
  
# To calculate the mode  
mode = mode(random_data)  
print("Mode : ", mode)
```

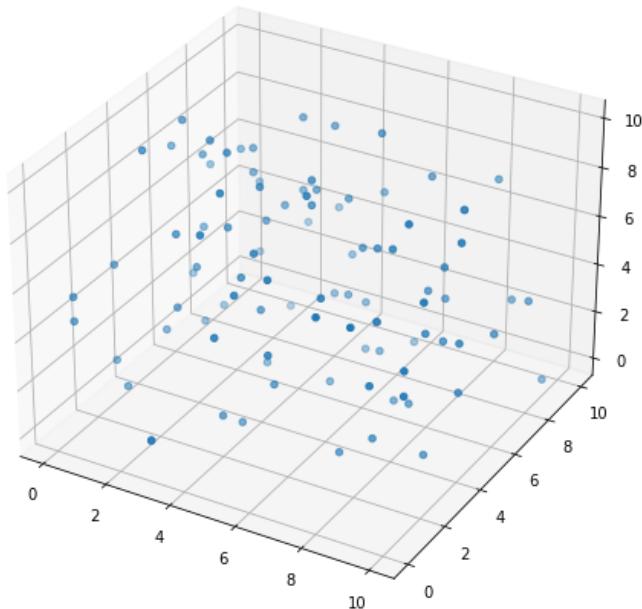
Mode : 4

5. Now, let's create a 3D scatter plot using 100 random numbers between 0 and 1. This will be a 3-dimensional array. We will use Matplotlib to plot a scatter plot using the generated random data.

```
[10]: import matplotlib.pyplot as plt  
  
# Generating data  
rng = np.random.default_rng(seed = 42)  
random_data = rng.integers(low = 0, high = 11, size=(100, 3))  
  
# Create the figure and 3D Axes object  
fig = plt.figure(figsize=(12,8))  
ax = fig.add_subplot(projection='3d')  
  
# Plot the data using the scatter() method  
ax.scatter(random_data[:,0],  
           random_data[:,1],  
           random_data[:,2])  
  
plt.title("3D Scatter Plot", fontsize = 20)  
plt.show()
```

This will output a 3D scatter plot of the data.

3D Scatter Plot



6. To flatten the array and return indices of non-zero numbers, we can use the **np.flatnonzero()** function. This function will flatten the array and return an array of non-zero numbers.

```
[11]: # Creating an array
lst = [[0, 1, 3, 0, 4], [8, 9, 0, 9, 6]]
array = np.array(lst)

# Flattening the list and returning indices of non-zero numbers
np.flatnonzero(array)
```

```
[11]: array([1, 2, 4, 5, 6, 8, 9], dtype=int64)
```

Day 5: Array Creation and Vector Operations

In these challenges, you will learn how to create multi-dimensional arrays from lists, how to extract specific elements or rows from arrays, and how to perform element-wise operations on arrays, such as multiplication.

```
list_numbers = [[12, 23, -45], [18, -77, -44]]
```

1. Using NumPy, create an array from the list. Write a code to return all the negative numbers in the list.
2. Using NumPy, write code to multiply each element in the array you created in question 1 by 2. Create a new variable for this array.
3. Write code to calculate the sum of the numbers on axis-1 of the array you created in question 2.
4. Write code to calculate the sum of numbers on axis-0 of the array you created in question 2.
5. Using the NumPy **arange()** function, create a NumPy array of numbers from 0 to 99. The shape of the array must be (2, 5, 10).
6. Write code to slice row [50, 51, 52, 53, 54, 55, 56, 57, 58, 59] from the array above (question 5).
7. Let's analyze the array you created in question 1 further. How many numbers in row 1 are greater than the numbers in row 2?

Day 5 - Answers

1. We are going to create an array and then use the `np.where()` function to filter the array for indexes of negative numbers in the array. We will use these indexes to access the numbers in the array.

```
[1]: import numpy as np

list_numbers = [[12, 23, -45], [18, -77, -44]]

# Creating an array
arr = np.array(list_numbers)

# Using where function to find indexes of negative numbers
negatives = np.where(arr < 0)
print(arr[negatives])

[-45 -77 -44]
```

2. NumPy has a `multiply` function that we can use to multiply each element of the array by 2. The function takes two arguments: the array and the number we want each element to be multiplied by. We are going to create a new array.

```
[2]: arr = np.array(list_numbers)

new_arr = np.multiply(arr, 2)
new_arr

[2]: array([[ 24,   46,  -90],
           [ 36, -154,  -88]])
```

You can see that every number has been multiplied by 2.

3. We are going to use the `new_arr` array we created in question 2. To add the numbers on axis-1, we will use the `np.sum()` function. We will have to specify which axis we want to add. Now, axis-0 is a column, and axis-1 is a row. So, here we are basically adding up the rows. So, in the code below, -20 is the sum of the first row (24, 46, -90), and -206 is the sum of the second row (36, -154, -188).

```
[3]: new_arr = np.multiply(arr, 2)

sum_axis_0 = np.sum(new_arr, axis=1)
sum_axis_0

[3]: array([-20, -206])
```

4. We do the same thing as in the previous question, but this time we set the axis to zero (0). This means that we are now adding up the columns. In the code below, 60 is the sum of columns 24 and 36, -108 is the sum of columns 46 and 154, and -178 is the sum of columns -90 and -88.

```
[4]: new_arr = np.multiply(arr, 2)

sum_axis_1 = np.sum(new_arr, axis=0)
sum_axis_1

[4]: array([ 60, -108, -178])
```

5. This challenge requires that we create a 3-dimensional array using the `arange()` function.

Here is a trick to use when creating arrays using the `arrange` function: To ensure that the code for creating arrays using `np.arange()` works correctly, it is important to ensure that the product of the shape dimensions equals the size of the array. If the product of the shape does not match the size, it will result in an error. In the example below, we create a NumPy array of numbers from 0 to 99 using `np.arange(100)`. We then reshape the array into a 3-dimensional array with the `shape=(2, 5, 10)`, using the

`reshape()` function. The `reshape()` function specifies the desired shape of the array.

```
[5]: array1 = np.arange(100).reshape(2, 5, 10)
array1
```

```
[5]: array([[[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
           [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
           [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
           [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
           [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]],
          [[50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
           [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
           [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
           [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
           [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

6. To answer this question, we must first understand the shape of the array we just created in question 5. The first number is 2. This number represents the number of sections in the array. The second number is 5. Five (5) is the number of rows in each section of the array. The third number is 10. Ten is the number of elements in each row. The numbers 0–49 are in Section 1, and the numbers 50–99 are in Section 2. If we use indexing, section 1 is index 0, and section 2 is index 1.

Here is how we create the code to slice the row: The row of the array that we are trying to slice (50–59) is sitting in section 2 (index 1), so we pass [1]. This row we want to slice is row number 1 in section 2 (row number 1 is index 0), so we pass [0]. In this row, we want to select all the numbers starting from index 0, so we pass [0:]. We combine these three to create [1, 0, 0]. Here is the code below:

```
[6]: array1[1,0,0:]
[6]: array([50, 51, 52, 53, 54, 55, 56, 57, 58, 59])
```

7. First, we need to slice rows 1 and 2 from the array we created in question 1. We are going to use the NumPy `greater()` function to return the numbers in row one that are

greater than the numbers in row two. This function will return an array of Boolean values. If a number in row one is greater than a number in row 2, it will return True, and if it is not, it will return False. We will use the `sum()` function to sum the True values.

```
[7]: # Selecting row one from the array
array_row_one = arr[0]
# Selecting row two from the array
array_row_two = arr[1]

# Using greater than function
nums_greater = np.greater(array_row_one, array_row_two)
print(f"The numbers in row 1 that are greater are: {sum(nums_greater)}")
```

The numbers in row 1 that are greater are: 1

Day 6: Array Manipulation and Vector Operations

In these challenges, you will create a NumPy array from a list of strings. These challenges aim to assess your proficiency in generating arrays and manipulating array data types as a means of enhancing data processing flexibility.

```
list_str = ["23", "12", "90", "28", "30"]
```

1. Create a NumPy array with the list above. Calculate the standard deviation of the array.
2. Write a code to change the data type of the array you created in question one (1) to a floating data type. Save this as a new variable.
3. Write a code to return the sum of values [90.0, 28.0, 30.0] from the array with floating data type you just updated in question (2).
4. Create a 2-dimensional array of random integers from 0 to 100 with the shape (5, 5). Use the array to find the minimum and maximum values, as well as the mean and standard deviation. Ensure that the results are reproducible.
5. Below are two arrays:

```
first_names = ["John", "Kenny"]
last_names = ["Smith", "Sakula"]
```

Using NumPy's `char.join()` function, create two arrays by joining the first name with the last name. Your first array should be: `array(['John', 'Smith'], dtype='U6')`. Your second array should be: `array(['Kenny', 'Sakula'], dtype='U6')`.

Day 6 - Answers

1. We create an array using the `np.array()` function.

```
[1]: import numpy as np  
  
list_str = ["23", "12", "90", "28", "30"]  
array1 = np.array(list_str)  
array1  
  
[1]: array(['23', '12', '90', '28', '30'], dtype='|U2')
```

Take note of the data type of the output. The "u2" data type means that each string in the array is 2 characters long.

To calculate the standard deviation (std), we can use the `np.std()` function. However, if we try to calculate the standard deviation on the array in question 1, we are going to get an error because the data type "U2" is that of a string, and standard deviation can only be calculated on numerical data types. To calculate the standard deviation, we will have to convert the array data type to an integer data type using the `astype()` function. Here is the code below:

```
[2]: # convert the array dtype to int64 and calculate std  
standard_deviation = array1.astype(np.int64).std()  
standard_deviation  
  
[2]: 27.419700946582186
```

2. To change the data type of the array in question one to a float, we use the `astype()` method. We used this function in the previous question. We are going to pass `float` as the argument. You can see in the output of the code below that the data type of the array has changed to floats.

```
[3]: # Converting array to int data type  
array2 = array1.astype(float)  
array2  
  
[3]: array([23., 12., 90., 28., 30.])
```

3. There are two ways we can approach this challenge:

The first way is to use indexing. We pass the indexes of the numbers we want, and they will be sliced from the array. The numbers we want are 90.0, 28.0, and 30.0. The indexes of these numbers are 2, 3, and 4. We will use the `sum()` method to add the numbers. Here is how we can implement this code:

```
[4]: result = array2[[2, 3, 4]].sum()
print(result)
```

148.0

The second way is to use NumPy's advanced indexing. We select the numbers between 28.0 and 90.0 and use the `sum()` method to get the total. Here is the code below:

```
[5]: result = array2[(array2 >= 28) & (array2 <= 90)].sum()
print(result)
```

148.0

You can see that both methods achieve the same results.

4. Here is the array below: We create a 2-dimensional array of integers from 0 to 100. The shape of the array is (5, 5). We are going to use NumPy's built-in functions `min()`, `max()`, `mean()`, and `std()` to find the minimum, maximum, and standard deviation of the array. Because this is random, your results will be different from what we have below. We are going to use `numpy.random.Generator.integers()` to generate the data, and we will use `seed` to ensure that the results are reproducible. Here is the code below, with the outputs.

```
[6]: # Generating data
rng = np.random.default_rng(seed = 42)
array3 = rng.integers(low = 0, high = 100, size=(5, 5))

min_val = array3.min()
max_val = array3.max()
mean = array3.mean()
std = array3.std()
print("Minimum value: ", min_val)
print("maximum value: ", max_val)
print("Mean: ", mean)
print("Std: ", std)

Minimum value:  8
maximum value:  97
Mean:  53.6
Std:  27.832355272236665
```

6. Since we are joining strings, we are going to use the **np.char.join()** function to do an element-wise join of the two lists. We will use slicing to slice two rows from the array to create two arrays.

```
[7]: first_names = ["John", "Kenny"]
last_names = ["Smith", "Sakula"]

# Using the join function to create an array
arr = np.char.join("", [first_names, last_names])

# Using slicing to return first and second arrays
arr1 = arr[:,0]
arr2 = arr[:,1]

arr1, arr2

[7]: (array(['John', 'Smith'], dtype='<U6'),
      array(['Kenny', 'Sakula'], dtype='<U6'))
```

Day 7: Transpose and Swap Arrays

The ability to transpose and swap arrays is important for manipulating and analyzing data, performing matrix operations, cleaning and sorting data, and visualizing data. By mastering these skills, you can improve your data manipulation and analysis capabilities and work more efficiently with arrays.

1. Using the `arange()` function, create an array of numbers from 0 and 10. The shape of the array must be (2, 5).
2. Write a code to change the shape of the array (question 1) to (5, 2). Create a new variable for this array. Now, add this array to the original array in question 1. The resulting array should have shapes (2, 5). Save this as a variable.
3. Swap the indexes of the array (question 2) using the `swapaxis()` method. Create a new variable for the swapped array.
4. Write a code to slice row 1 from the swapped array in question 3.
5. You are given the following data:

```
players = ["Robin", "Leo", "Pogba", "Diego", "Ronaldo"]
teams = ["Man UTD", "Barca", "Juve", "Napo", "RMadrid"]

goals = [[12, 15, 16, 15, 13],
         [26, 30, 31, 25, 24],
         [10, 12, 8, 6, 13],
         [18, 19, 17, 20, 21],
         [21, 32, 25, 21, 22]]
```

Each row in the "goals" list represents the goals scored by each player in the last 5 seasons. For example, Robin's goals are [12, 15, 16, 15, 13] and he plays for "Man UTD," and Leo's are [26, 30, 31, 25, 24] and he plays for "Barca." Using NumPy, Seaborn, and Matplotlib, create a heatmap of the goals and calculate the gradient of the data. Annotate your heat map. The **x-ticks** will be the names of the players, and the **y-ticks** will be the teams of the players.

Day 7 - Answers

1. We are going to create an array using the `np.arange()` function. We will use the reshape method to shape the array into (2, 5).

```
[1]: import numpy as np  
  
array1 = np.arange(0, 10).reshape(2, 5)  
array1  
  
[1]: array([[0, 1, 2, 3, 4],  
           [5, 6, 7, 8, 9]])
```

2. To reshape the array, we are going to use the `reshape()` method of NumPy. We will pass the 5 and 2 as arguments.

```
[2]: # Reshaping array using the shape method  
array2 = array1.reshape(5, 2)  
array2  
  
[2]: array([[0, 1],  
           [2, 3],  
           [4, 5],  
           [6, 7],  
           [8, 9]])
```

Now, if you try to add this array to the array above (question 1), we will get a broadcast error because of the different shapes. Since we want the resulting array to have a shape of (2, 5), we will have to transpose `array2`.

```
[3]: # Transposing and adding the two arrays  
array3 = array2.transpose() + array1  
array3  
  
[3]: array([[ 0,  3,  6,  9, 12],  
           [ 6,  9, 12, 15, 18]])
```

3. The `np.swapaxes()` function is used to swap two axes of an array, altering its layout. Unlike transpose, which can rearrange the entire axes order, `np.swapaxes()` specifically

focuses on swapping two axes. To use `np.swapaxes()`, you need to specify the axes you want to interchange. For example, `np.swapaxes(arr, axis1, axis2)` swaps axis-1 with axis-2 in the array, arr. Here is the code below:

```
[4]: swapped_array = np.swapaxes(array3, 0, 1)
swapped_array
```

```
[4]: array([[ 0,  6],
           [ 3,  9],
           [ 6, 12],
           [ 9, 15],
           [12, 18]])
```

4. We can use slicing to slice row-1. Here is how we write the code:

```
[5]: row_1 = swapped_array[0, :]
row_1
```

```
[5]: array([0, 6])
```

You can see from the output that the first row has been sliced. Since we want the first row, we pass 0 (the row we want is sitting at index 0) as the first number. The comma (,) means we are now going inside the row. We pass (:) because we want all of the elements in a row. We combine the two to form [0, :].

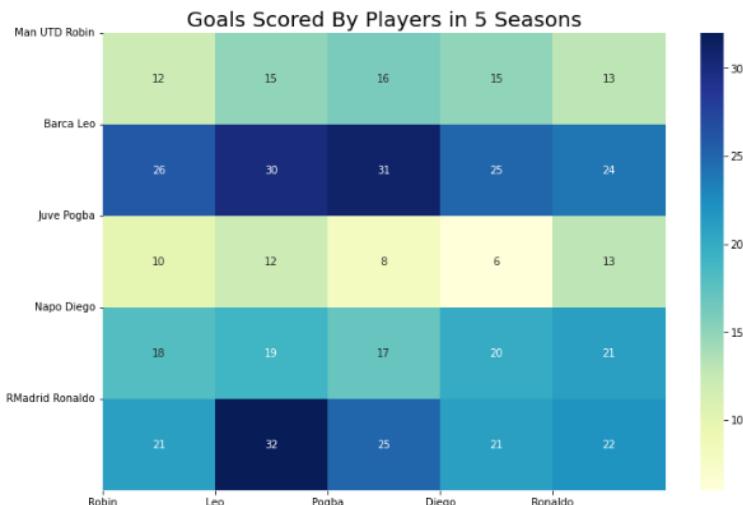
5. This challenge requires that we plot a heatmap of goals from the given data. First, we are going to create an array and then plot a heatmap graph. We are going to import Matplotlib for this challenge. We will use the `np.arange()` function to set `xticks` and `yticks` for the heatmap.

```
[6]: import seaborn as sns
import matplotlib.pyplot as plt

players = ["Robin", "Leo", "Pogba", "Diego", "Ronaldo"]
teams =[ "Man UTD", "Barca", "Juve", "Napo", "RMadrid"]

goals =[[12, 15, 16, 15, 13],
        [26, 30, 31, 25, 24],
        [10, 12, 8, 6, 13],
        [18, 19, 17, 20, 21],
        [21, 32, 25, 21, 22]
    ]

# Creating array from list of goals
array1 = np.array(goals)
# Combining player name with team Label
player_team_label = [f'{team} {player}' for team,
                     player in zip(teams, players)]
fig, ax = plt.subplots(figsize=(12,8))
sns.heatmap(array1, annot=True, cmap="YlGnBu")
ax.set_xticks(np.arange(len(players)), labels=players)
ax.set_yticks(np.arange(len(teams)), rotation='horizontal',
            labels=player_team_label)
plt.title("Goals Scored By Players in 5 Seasons", fontsize=20)
plt.show()
```



In this heatmap, the rows are matched to the columns. For example, Robin is in row one, and the goals and team he played for are in column one. Feel free to customize the heatmap to your preference.

Day 8: Slicing NumPy Arrays

These challenges will test your abilities to create and slice NumPy arrays. Learning to slice NumPy arrays is an important skill to have for anyone working with data because it allows you to extract and manipulate specific portions of an array. This is especially useful when working with large datasets, as it allows you to perform operations on a subset of the data rather than the entire array. This is what is called "dimensionality reduction."

1. Create a one-dimensional array with random integers between 0 and 20. The size of the array is 10. Ensure that the results are reproducible.
2. Using slicing, create a subset array of 3 integers from the array you created in question 1. The first integer is at index 2, the second integer is at index 4, and the third index is at index 7.
3. Create a two-dimensional array from the list below. Check the dimensions and shape of your array.

```
my_list = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10],  
           [11, 12, 13, 14, 15]]
```

Using slicing, write a code to access the numbers 3, 8, and 13. Your code should return [3, 8, 13].

4. Create a three-dimensional array from the nested list below:

```
lis1 = [[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10],  
           [11, 12, 13, 14, 15]]]
```

Using slicing, write code that creates a sublist from the array you created. Your code should return an array of numbers (4, 8, and 15).

Day 8 - Answers

1. To create an array of 10 randomly chosen integers between 0 and 20, we'll use the `random.default_rng()` function to create a random number generator. This code will produce 10 random integers between 0 and 20 each time it is executed. The result will always vary. We'll set the `seed` parameter to 42 to make sure the outcomes can be replicated.

```
[1]: import numpy as np

rng = np.random.default_rng(seed=42)
array1 = rng.integers(21, size=10)
array1
```



```
[1]: array([ 1, 16, 13,  9,  9, 18,  1, 14,  4,  1], dtype=int64)
```

2. The easiest way to answer this question is to create a list of indices that we want to slice from the array we created in question 1. We want to slice indices 2, 4, and 7. We will create a list of these numbers and use it to slice a subset from array 1.

```
[2]: subset_array = array1[[2, 4, 7]]
print(subset_array)
```



```
[13  9 14]
```

3. To understand how slicing works in NumPy, we have to first create the two-dimensional array from the nested list.

```
[3]: my_list = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]

# Convert the List to a NumPy array
array2 = np.array(my_list)
array2
```



```
[3]: array([[ 1,  2,  3,  4,  5],
           [ 6,  7,  8,  9, 10],
           [11, 12, 13, 14, 15]])
```

This will output a two-dimensional array. We can use the `ndim` attribute to check the number of dimensions in the array.

```
[4]: # Check dimensions  
array2.ndim
```

```
[4]: 2
```

Now, let's check the shape of this array using the `shape` attribute.

```
[5]: # Check shape  
array2.shape
```

```
[5]: (3, 5)
```

This output shows that our array has 3 rows, and each row has 5 elements. To access the items in the array, we must first access the row, then the element. For example, if we wanted to access the number 5 from the first row, we would pass [0, 4]. Zero(0) is for the row, and 4 is for the element index in row zero(0). Now, the challenge requires that we access numbers 3, 8, and 13. The number 3 is in row 0, at index 2. The number 8 is in row 1 at index 2, and the number 13 is in row 2 at index 2. Here is the full code:

```
[6]: my_list = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]  
  
# Convert the list to a NumPy array  
array2 = np.array(my_list)  
  
# Use slicing to access the desired elements  
sub_array = array2[0, 2], array2[1, 2], array2[2,2]  
print(np.array(sub_array))
```



```
[ 3  8 13]
```

4. This is similar to question 3. We will use the same method to slice the array. The only difference is that this is a three-dimensional array. So, instead of using a 2-element list to

slice the array, we use a 3-element list. This is because we need to access two lists in the array to get to the element.

```
[7]: my_list = [[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]]

# Convert the nested list to a NumPy array
array3 = np.array(my_list)

# Use slicing to create the sub list
sub_list = array3[0,0,3], array3[0,1,2], array3[0,2,4]
np.array(sub_list)

[7]: array([ 4,  8, 15])
```

Day 9: Analyze a One-Dimensional Array

In these challenges, you are going to create an array from a list. Why should you use an array instead of a Python list? The NumPy Series provides better performance, easier manipulation and indexing, and better interoperability with other scientific libraries, making them a preferred choice for scientific and data analysis applications. They also have better memory management than Python lists.

```
nums = [12, 4, 6, 7, 9, 19, 21, 67, 8]
```

1. Create a NumPy one-dimensional array with the list above.
2. Using NumPy, write code that returns the minimum number in the array.
3. Write a code that returns the index of the largest number in the array.
4. Using NumPy, calculate the average of the biggest and smallest numbers in the array.
5. Write a code to find the mean, median, and standard deviation of the array.
6. Write code to find the outlier in the array above.

Day 9 - Answers

- Creating a NumPy one dimensional requires that we use the **np.array()** function. We pass the list as an argument to the function.

```
[1]: import numpy as np  
  
nums = [12, 4, 6, 7, 9, 19, 21, 67, 8]  
nums_array = np.array(nums)  
nums_array  
  
[1]: array([12, 4, 6, 7, 9, 19, 21, 67, 8])
```

- NumPy has a **np.min()** function that returns the minimum number from a given array.

```
[2]: min_number = np.min(nums_array)  
print('The minimum of the array is', min_number)  
  
The minimum of the array is 4
```

- NumPy has a **np.argmax()** function that we can use to return the index of a number from an array.

```
[3]: nums = [12, 4, 6, 7, 9, 19, 21, 67, 8]  
nums_array = np.array(nums)  
  
max_index = np.argmax(nums_array)  
print(f'The index of the largest number is {max_index}')  
  
The index of the largest number is 7
```

- To calculate the average of the biggest and smallest numbers in the array, we can use the **np.min()** and **np.max()** functions, and then take the average:

```
[4]: nums = [12, 4, 6, 7, 9, 19, 21, 67, 8]  
nums_array = np.array(nums)  
  
min_num = np.min(nums_array)  
max_num = np.max(nums_array)  
average = np.array([min_num, max_num]).mean()  
print('The average of the min and max is', average)  
  
The average of the min and max is 35.5
```

5. NumPy is so powerful because it has all these functions that we can use to carry out mathematical operations. We are going to use the **np.mean()**, **np.median()**, and **np.std()** functions to answer this challenge:

```
[5]: mean = np.mean(nums_array)
median = np.median(nums_array)
std = np.std(nums_array)

print("Mean: ", mean)
print("Median: ", median)
print("Standard deviation: ", std)
```

```
Mean: 17.0
Median: 9.0
Standard deviation: 18.499249234015476
```

6. To find the outlier of the array using NumPy, you can calculate the lower and upper bounds of the data using the Interquartile Range (IQR) method. Outliers are defined as any values that are more than 1.5 times the IQR outside of the lower or upper quartile. We can use the **np.percentile()** function to find the lower and upper quartiles, and then use these values to calculate the IQR, and subsequently the lower and upper bounds. The IQR is the difference between the upper quartile and the lower quartile. It represents the spread of the middle 50% of the data. The lower bound is calculated as the lower quartile (q1) minus 1.5 times the IQR, and the upper bound is calculated as the upper quartile (q3) plus 1.5 times the IQR. Here is the code below:

```
[6]: # First sort the array
sorted_array = np.sort(nums_array)

def calculate_outlier(array):
    q1 = np.percentile(sorted_array, 25)
    q3 = np.percentile(sorted_array, 75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    outliers = [x for x in sorted_array if x < lower_bound or x > upper_bound]
    return outliers

calculate_outlier(sorted_array)
```

```
[6]: [67]
```

The outlier is 67. This is because 67 is way larger than all the numbers in the array.

Day 10: The arange Function and Boolean Indexing

In the challenges below, you will use the NumPy **arange()** function. It is similar to the built-in **range()** function in Python, but it returns a NumPy array. Boolean indexing, on the other hand, is a way of indexing NumPy arrays based on a set of Boolean conditions. It allows you to select elements from an array that meet certain criteria.

num = 50

1. Create an array from the number above using the **arange()** function of NumPy. Your array should start at 0 with a step of 5.
2. Using NumPy, write a code that returns all the numbers in the array that are greater than 25.
3. Here is some data below:

```
days = ["mon", "tue", "wed", "thu", "Fri"]  
hours_worked = [ 8, 8, 10, 11, 7]
```

Using Boolean indexing of NumPy arrays, return the days of the week where hours worked were 8.

4. Write another code (using Boolean indexing) to return the day of the week with the highest number of hours worked.
5. Write another code to return all the days of the week where over 9 hours of work were achieved.

Day 10 - Answers

1. The **np.arange()** function is similar to the **range()** function in Python. It takes three parameters: start, finish, and step. For this challenge, we want to set the start at 0, stop at 50, and step at 5. This means it will create an array with numbers between 0 and 50, spaced by 5.

```
[1]: import numpy as np  
  
num_array = np.arange(0, 50, 5)  
num_array  
  
[1]: array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45])
```

It is important to note that the *stop* parameter in **np.arange()** specifies the end value that is not included in the generated array. In this case, the stop value of 50 is **not** included in the output array.

2. The **np.where()** function makes it easy to filter an array for specific values. In this challenge, we are going to use the function to return numbers that are greater than 25 from an array. First, we are going to use the **np.where()** function to find indexes of numbers over 25. Then we will use the indexes to get the numbers from the array. Here is the code below:

```
[2]: # Getting indexes of numbers over 25  
over_25_indices = np.where(num_array > 25)  
# Using the index to get numbers over 25  
print(num_array[over_25_indices])  
  
[30 35 40 45]
```

3. Boolean indexing comes in handy when filtering data from multiple arrays. Remember, for this to work, the arrays must be the same length. You can see from the code below

that Monday and Tuesday were the days when 8 hours were worked.

```
[3]: days = ["mon", "tue", "wed", "thu", "Fri"]
hours_worked = [ 8, 8, 10, 11, 7]

# Creating arrays
days = np.array(days)
hours_worked = np.array(hours_worked)

# Using Boolean to get the 8 hour days.
days_8hrs = days[hours_worked == 8]
days_8hrs
```



```
[3]: array(['mon', 'tue'], dtype='<U3')
```

4. For this challenge, we are going to use the `np.argmax()` function, first to find the index of the maximum hours worked from the `hours_worked` array. We will then use the index to find the corresponding day in the `days` array.

```
[4]: # finding max value index
max_hours_index = np.argmax(hours_worked)
print(f'Day with the max hours worked is,{days[max_hours_index]}'')
Day with the max hours worked is,thu
```

So, Thursday is the day with the maximum hours worked

5. We are going to use Boolean indexing to solve this challenge. See the code below:

```
[5]: days_over_9hrs = days[hours_worked > 9]
days_over_9hrs
```



```
[5]: array(['wed', 'thu'], dtype='<U3')
```

In this code, `days[hours_worked > 9]` uses the Boolean mask to filter the days DataFrame. It selects only the rows where the mask is True, effectively extracting the days with more than 9 hours worked. You can see that Wednesday and Thursday are the days with over 9 hours. The results are returned as an array.

Day 11: Preprocessing, Analysis and Visualization

As part of data preprocessing, you have to identify different types in the array. You may be required to create subarrays of a specific data type in order to carry out, for example, mathematical operations and visualization. The challenges below will challenge you to preprocess different data types.

```
list1 = ["34", "name", "45", "is", "100"]
```

1. Using NumPy, create an array using the list above. Write a code to check how many non-numeric elements are in the list.
2. Using NumPy, write code to sum all the numeric elements in the list.
3. Using NumPy, write code to square all the numeric elements in the array you created in question 1. Return an array of squared elements.
4. You have the following lists:

```
student_id = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
attendance = [85, 92, 70, 95, 80, 60, 90, 75, 65, 100]
final_exam_score = [80, 89, 65, 56, 66, 50, 85, 70, 55, 100]
```

Using NumPy, create an array using the lists above. Each list must be a row in the array. Your array must have 3 rows and 10 columns. To visualize the correlation between **attendance** and the **final_exam_score**, create a scatter plot using Matplotlib. Set the **final_exam_score** the **x-axis** and **attendance** as the **y-axis** of your scatter plot. Use the NumPy array you just created as the source of your data for the plot.

- b. By visualizing your scatter plot (question 4a), what final score was the outlier of the data?

Day 11 - answers

1. We are going to create an array, and then we are going to use the `np.char.isnumeric()` function to check if the elements of the string array represent numeric values. We are going to return a list of these non-numeric items using the `list()` function. We will use the `len()` function to count the non-numeric items in the list.

```
[1]: import numpy as np

list1 = ["34", "name", "45", "is", "100"]

# Creating an array
arr = np.array(list1)
non_numeric = list(filter(lambda x: not np.char.isnumeric(x), arr))
len(non_numeric)
```

[1]: 2

You can see from the output that there are two(2) non-numeric items ("name" and "is") in the array.

2. To sum the numeric items in the array, we have to convert them to numeric data types first. We are going to first filter the array we created in question 1 for numeric characters using the `filter()` function and the `np.char.isnumeric()` function. Then we will convert them into integers. We will use the `sum()` function to sum the numeric items.

```
[2]: arr = np.array(list1)

# Access the numeric items from the arr array
numeric_list = list((filter(lambda x: np.char.isnumeric(x), arr)))
# Convert the numeric items to integers
numeric_sum = np.array([int(i) for i in numeric_list]).sum()
numeric_sum
```

[2]: 179

3. To square the numeric items in the array, we are going to use the same method we used in the previous question. However, this time we will square ($**2$) the numbers

instead of summing them. We will return an array with squared numbers.

```
[3]: arr = np.array(list1)

# Access the numeric items from the arr
numeric_list = list(filter(lambda x: np.char.isnumeric(x), arr))

# Convert the numeric items to integers
numeric_list = np.array([int(i)**2 for i in numeric_list])
numeric_list
```

[3]: array([1156, 2025, 10000])

4. This challenge requires that we create a NumPy array from the given data and use the array to plot a scatter plot. First, we create a NumPy array by passing the lists as arguments to the **np.array()** function. See the code below:

```
[4]: student_id = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
attendance = [85, 92, 70, 95, 80, 60, 90, 75, 65, 100]
final_exam_score = [80, 89, 65, 56, 66, 50, 85, 70, 55, 100]

arr = np.array([student_id, attendance, final_exam_score])
arr
```

[4]: array([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
 [85, 92, 70, 95, 80, 60, 90, 75, 65, 100],
 [80, 89, 65, 56, 66, 50, 85, 70, 55, 100]])

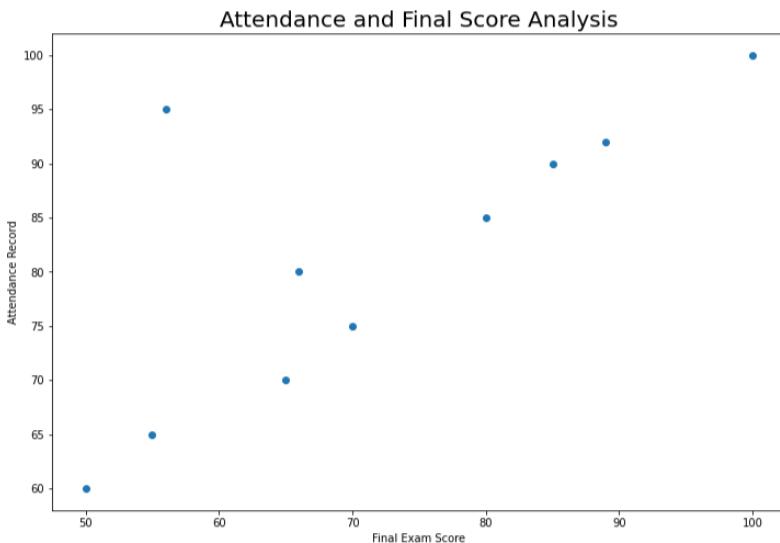
Now let's create a scatter plot from the array. We will slice the **attendance** row and the **final_exam_score** from the array. We will use the sliced data as the **x-axis** and **y-axis** of the scatter plot respectively.

To slice the attendance from the array, we will use `[1:2, 0:]` and `[2:3, 0:]` for the **final_exam_score**. We will use Matplotlib to plot the graph.

```
[5]: import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))
# Using slicing to get x-axis and y-axis data
plt.scatter(arr[2:3,0:], arr[1:2,0:])
plt.xlabel("Final Exam Score")
plt.ylabel("Attendance Record")
plt.title("Attendance and Final Score Analysis", fontsize=20)
plt.show()
```

This will output:



You can see that there is a link between attendance records and final exam scores. It looks like the students who have a high attendance record also have high exam results.

- A student who has an attendance record of 95 seems to be an outlier. This student had high attendance but low exam results. Which is unusual compared to the other data points. You can see that the 95th data point is separated from the other points.

Day 12: Array Sorting and Filtering

The skill to sort arrays in ascending or descending order and filter arrays to include only elements that meet certain criteria is a useful skill for organizing data or identifying patterns in the data. These skills will be tested with the challenges below:

list1 = [[12, 34, 56], [12, 13, 8], [6, 5, 10]]

1. Create an array using the list above. Write code to sort the array in ascending order: first sort by columns, then by rows. Create a new variable for each of the sorted arrays.
2. Using NumPy slicing, write a code that slices these numbers in this order [8, 12, 56] from the array sorted by columns (question 1). Create a new variable.
3. Create a 2-dimensional array of random integers between 100 and -100 in the shape (3, 3). Use the **np.where()** function to replace all negative values in the array with 0. Use a seed to save the random numbers generated.
4. Here is a list below:

list1 = [[12, 23, 34, 34], [13, 13, 20, 21]]

Create an array from the list above.

Using the **np.unique()** function, return the unique elements in the array and the counts of each unique element (how many times each element appears in the list).

5. Use the **np.flatten()** function to flatten the nested array you created in question 4.

Day 12 - Answers

1. In this challenge, we are creating a 2-dimensional array from a nested list. We are going to use the `np.sort()` function to sort the array in descending order. By default, this function sorts an array in ascending order. It creates a copy of the original array. After sorting, each column will have the smallest number at the top and the largest number at the bottom. To sort the array by columns, we are going to set the axis parameter to zero (0).

```
[1]: import numpy as np

list1 = [[12, 34, 56], [12, 13, 8], [6, 5, 10]]

# Creating array from list
array1 = np.array(list1)
# Sort the array in ascending order by columns
sorted_array_column = np.sort(array1, axis = 0)
sorted_array_column

[1]: array([[ 6,  5,  8],
           [12, 13, 10],
           [12, 34, 56]])
```

To sort the array by rows, we will use the `np.sort()` function and set the axis parameter to one (1). Each row will be sorted from the smallest to the largest number. Here is the code below:

```
[2]: list1 = [[12, 34, 56], [12, 13, 8], [6, 5, 10]]

# Creating array from list
array1 = np.array(list1)

# Sort array in ascending order by rows
sorted_array_row = np.sort(array1, axis=1)
sorted_array_row

[2]: array([[12, 34, 56],
           [ 8, 12, 13],
           [ 5,  6, 10]])
```

2. In this challenge, we are going to use NumPy slicing to slice numbers from the array sorted by columns (`sorted_array_column`) in question 1. Since we are slicing three numbers, we are going to create three slices. On each slice, the first number is the row, and the second number is the index of the number we want in the row. In the code below, `sorted_arr[0][2]`, 0 is the first row (6, 5, 8), and 2 is the index of the number 8. Then comes `sorted_arr[1][0]`, where 1 is the row ([12, 13, 10]) and 0 is the index of the number 12. So, we repeat that for the three numbers. Here is the full code.

```
[3]: # using slicing to create a new array
new_array = np.array([sorted_array_column[0][2],
                     sorted_array_column[1][0],
                     sorted_array_column[2][2]])
new_array
```



```
[3]: array([ 8, 12, 56])
```

3. Let's create an array of size (3, 3) of random integers between 100 and -100. Please note that your output will be different from the output below because we are using random. We set the seed parameter to 42 (you can use whatever number you want) to ensure that the random numbers generated when we run the code are saved. This means we will not get a different set of numbers every time we run the code.

```
[4]: # Generating data
rng = np.random.default_rng(seed = 42)
array2 = rng.integers(low = -100, high = 100, size=(3, 3))
array2
```



```
[4]: array([[ -83,   54,   30],
           [ -13,  -14,   71],
           [ -83,   39,  -60]], dtype=int64)
```

Now let's use the `np.where()` function to convert the negative numbers to zeros. Let's breakdown the `np.where`

function below. The first part of the function ($\text{array2} < 0$) is the condition. The next part (0) is what we want the function to do: replace all the numbers in the array that meet the condition ($\text{array2} < 0$) set by zero, and the last part (array2) is the array that the function will be applied to. You can see that the negative numbers generated above have been replaced by zeros.

```
[5]: array2 = np.where(array2 < 0, 0, array2)
array2

[5]: array([[ 0, 54, 30],
           [ 0,  0, 71],
           [ 0, 39,  0]], dtype=int64)
```

4. First, we are going to create an array from the list. We will use the `np.unique()` function to return an array with only unique elements. We also return the counts of each element in the array by setting the `return_counts` parameter to True. For instance, the element count for 13 is 3. This means that 13 appears three times in the `array3`.

```
[6]: list1 = [[12, 23, 34, 34, 35], [13, 13, 13, 20, 21]]

# Creating an array
array3 = np.array(list1)
# use np.unique to find the unique elements in the array
unique_elements, elements_count = np.unique(array3, return_counts=True)
print("Unique Elements:", unique_elements)
print("Elements count:", elements_count)

Unique Elements: [12 13 20 21 23 34 35]
Elements count: [1 3 1 1 1 2 1]
```

5. We are going to use the `flatten()` method to flatten the array we created in question 4. This will return a one-dimensional array.

```
[7]: # Using flatten method to flatten a nested array
array4 = array3.flatten()
array4

[7]: array([12, 23, 34, 34, 35, 13, 13, 13, 20, 21])
```

Day 13: Slicing and Analyzing Arrays

In this challenge, your objective is to demonstrate your proficiency in creating arrays from lists and extracting specific information from those arrays. You have the following data:

```
names = ["John", "Kelly", "Jos", "Peter", "Robert", "Piper"]
age = [21, 21, 56, 44, 56, 96]
gender = ['m', 'f', 'm', 'm', 'm', 'f']
```

1. Using the data above, create an array and transpose it. The creation of an array and transposition should be combined into one code. The code should return a transposed array.
2. Using slicing or indexing, write code to return the last row of the transposed array (question 1).
3. Using slicing or indexing, write code to return the first two rows of the transposed array (question 1).
4. Using slicing, write a code that returns the age of Peter.
Return Peter's age as an integer data type.
5. Write code to return an array of Kelly's age and gender using slicing (from the transposed array in question 1).
6. One of the uses of slicing is for data visualization. Slicing is often used in data visualization to extract specific portions of a dataset for plotting. Write code to create a subarray of the ages (only ages) in the transposed array using slicing. Using Matplotlib, plot a histogram of the age array. Your graph should have an **xlabel**, an **ylabel**, and a title.

Day 13 - Answers

- First, we create an array from the three lists (names, age and gender). We will pass the lists as arguments to the `np.array()` function to create a NumPy array. Transposing the data means that we will turn the columns into rows and the rows into columns. We use the `transpose()` method. Here is the code below:

```
[1]: import numpy as np

names = ["John", "Kelly", "Jos", "Peter", "Robert", "Piper"]
age = [21, 21, 56, 44, 56, 96]
gender = ['m', 'f', 'm', 'm', 'm', 'f']

# creating an array and transposing
array1= np.array([names, age, gender]).transpose()
array1

[1]: array([['John', '21', 'm'],
           ['Kelly', '21', 'f'],
           ['Jos', '56', 'm'],
           ['Peter', '44', 'm'],
           ['Robert', '56', 'm'],
           ['Piper', '96', 'f']], dtype='|<U11')
```

Another way would be to use the `T` attribute. This will return a transposed array exactly like the one above.

```
[2]: names = ["John", "Kelly", "Jos", "Peter", "Robert", "Piper"]
age = [21, 21, 56, 44, 56, 96]
gender = ['m', 'f', 'm', 'm', 'm', 'f']
# creating an array and transposing
array1= np.array([names, age, gender]).T
array1

[2]: array([['John', '21', 'm'],
           ['Kelly', '21', 'f'],
           ['Jos', '56', 'm'],
           ['Peter', '44', 'm'],
           ['Robert', '56', 'm'],
           ['Piper', '96', 'f']], dtype='|<U11')
```

You can see that both codes return a transposed array. You can use whichever method you prefer. I would recommend using the `transpose()` method because it is more readable.

2. The easiest way to access the last row of the transposed array is to use indexing. Since we are accessing the last row, we can use negative indexing.

```
[3]: # Slicing one row of the array
last_row_array = array1[-1]
last_row_array

[3]: array(['Piper', '96', 'f'], dtype='|<U11')
```

3. This question asks that we access the first and second rows of the transposed array. Here is the code below:

```
[4]: # slicing two rows of the array
slice_two_rows = array1[:2]
slice_two_rows

[4]: array([['John', '21', 'm'],
           ['Kelly', '21', 'f']], dtype='|<U11')
```

4. To access Peter, first we access the row that has the name "Peter" (row 3), and then in that row, we access his age. The age is on index 1. We combine the two as [3, 1]. We use the `int()` function to ensure that an array is converted to an integer data type. Here is the code below:

```
[5]: # Accessing Peter's age using slicing
peter_age = int(array1[3,1])
print(f'Peter is {peter_age} yrs old')
```

Peter is 44 yrs old

5. To access the age and gender of Kelly, we have to first access row 1. In that row, we want age, which is sitting on index 1, and gender, which is sitting on index 2. To access these two, we will use [1:3]. Now, we combine this with the row index to create [1, 1:3].

```
[6]: kelly_age_gender = array1[1, 1:3]
kelly_age_gender
```

```
[6]: array(['21', 'f'], dtype='|<U11')
```

You can see in the output that Kelly is 21 and female.

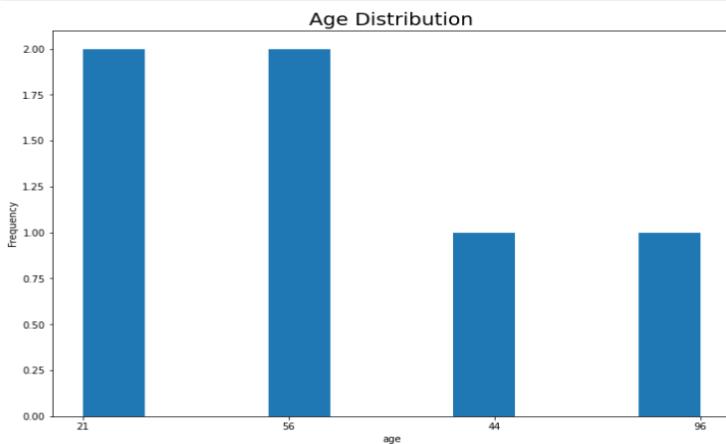
6. In this challenge, we are required to create an array of all the ages in the transposed array. First, we access the whole array using [:], then we access index 1 in each row. So we combine the two to create [:,1].

```
[7]: # Slice age from the array
age_array = array1[:,1]
age_array
```

```
[7]: array(['21', '21', '56', '44', '56', '96'], dtype='|<U11')
```

Let's create a histogram using Matplotlib.

```
[8]: import matplotlib.pyplot as plt
plt.figure(figsize=(12, 8))
plt.hist(age_array, rwidth = 2.9, bins= 10)
plt.xlabel("age")
plt.ylabel("Frequency")
plt.title("Age Distribution", fontsize= 20)
plt.show()
```



Day 14: Analyze Data with NumPy Part - 1

You are given the following data, which is in list format: Your task is to analyze and obtain insights from this data using NumPy (do not use pandas). Your first task is to create one array from the three (3) lists. You will use this array to answer the questions that follow.

```
names = ["Kelly", "Ben", "Jack", "Muhammad", "Jose", "Liz"]
grades = [ 45, 65, 75, 35, 85, 40]
classes = ["a", "c", "e", "g", "e", "f"]
```

1. Using NumPy, write code that returns all the names and the number of students from class "e" who got over 50 marks.
2. Your next task is to write a code that returns all the students who got less than 75 marks.
3. Write code that returns how many marks a student named Liz got.
4. Which student got the highest marks?
5. What is the longest name in the array, and what is its index?

Day 14 - Answers

1. In order to retrieve the values in the array, we must utilize Boolean indexing. Making an array from the three lists is the first thing that we are going to do.

```
[1]: import numpy as np

names = ["Kelly", "Ben", "Jack", "Muhammad", "Jose", "Liz"]
grades = [ 45, 65, 75, 35, 85, 40]
classes = ["a", "c", "e", "g", "e", "f"]

student_array = np.array([names, grades, classes])
student_array

[1]: array([['Kelly', 'Ben', 'Jack', 'Muhammad', 'Jose', 'Liz'],
       ['45', '65', '75', '35', '85', '40'],
       ['a', 'c', 'e', 'g', 'e', 'f']], dtype='|<U11')
```

We are going to slice three arrays from the array above. For Boolean indexing to work, the arrays must be of the same length. We will use the `np.where()` function to filter the arrays for students who got over 50 marks. The `where()` function will return the index of the names in class "e" who got over 50 marks. We use this index to access the names from the "**names**" array.

```
[2]: # Slicing names from array
names = student_array[0:1,0:]

# Slicing grades from array and converting array to int
grades = student_array[1:2,0:1].astype(int)
# Slicing classes from array
classes = student_array[2:3, 0:]

# find the index of students in class 'e' who got over 50
indices = np.where((classes == "e") & (grades > 50))
# Use the indices to get the number of the students
over_50_names = len(names[indices])

# Use the indices to get the names of the students
class_e_students_over_50 = names[indices]

print("Students from class 'e' with over 50 are:", class_e_students_over_50)
print(f"Number Students who got over 50 marks in class 'e'are: {over_50_names}")

Students from class 'e' with over 50 are: ['Jack' 'Jose']
Number Students who got over 50 marks in class 'e'are: 2
```

2. I will share two solutions to this problem. First, to find the names of the students that got over 75 marks, we can use the **where()** function. This will return a list of students who got less than 75 marks.

```
[3]: # Find the indices of students who got less than 75 marks
indices = np.where(grades < 75)

# Use the indices to get the names of the students
less_75_marks = names[indices]
print(less_75_marks)

['Kelly' 'Ben' 'Muhammad' 'Liz']
```

The second way would be to use Boolean indexing. See the code below:

```
[4]: # Get names over 75 marks
names_75 = names[grades < 75]

# Use the list function to return a list of names
print(list(names_75))

['Kelly', 'Ben', 'Muhammad', 'Liz']
```

3. This is similar to the previous question, only this time we are looking for a specific name. We can use the **np.where()** function. Using the **np.where()** function, here's how we can find the name: First, we will use the **np.where()** function to find the index of the female named "Liz." We will use this index to get Liz's grades from the "grades" array.

```
[5]: # Find the index of the student named "Liz"
index = np.where(names == "Liz")

# Use the index to get the student's marks
liz_marks = int(grades[index])
print(f'Liz got {liz_marks} marks')

Liz got 40 marks
```

4. First, we can use the `argmax()` function to return the index of the highest integer in the "grades" array, and then we can use that index to extract the name of the student who received the highest marks from the "names" array.

```
[6]: # Find the index of the highest grade
      highest_index = np.argmax(grades)

      # Use the index to find the corresponding name
      name = names[:,highest_index]
      print(f'The student who got the highest marks is, {" ".join(name)}')

The student who got the highest marks is, Jose
```

Another way would be to use Boolean indexing. This will return an array with the name of the student with the highest marks.

```
[7]: # Getting the name of the student
      name = names[grades == np.max(grades)]

      print(f'The student who got the highest marks is, {name[0]}')

The student who got the highest marks is, Jose
```

Both codes will return "Jose" as the student who got the highest marks.

5. We will use slicing to access a subarray of all the names in the `student_array`. To return the student with the longest name and their index, we will use the `np.char.str_len()` function to return a list of the lengths of all the strings in the sliced subarray. We will then use the `np.argmax()` function to return the index of the longest string in the array. The index will be used to fetch the longest word from the `student_array`. Here is the code below:

```
[8]: # find length of each word in the array
longest_word = np.char.str_len(student_array[0:1,0:])

# find the index of the longest word
index_of_longest_word = np.argmax(longest_word)

# find the longest name
longest_word = student_array[:,index_of_longest_word]

print(f'The longest name is sitting at index: {index_of_longest_word}')
print(f'The longest name is: {longest_word[0]}')

The longest name is sitting at index: 3
The longest name is: Muhammad
```

Muhammad is the longest name and is sitting at index 3. This challenge demonstrates the versatility of NumPy for working with both numerical and textual data.

Day 15: Analyse Data with NumPy Part - 2

For this challenge, you are going to work with the data below. You are going to use NumPy functionality to answer questions using this data. This challenge will test your ability to use slicing to extract specific columns or rows of the data and perform operations on them. You will import a CSV file called **names_age_sex_data**.

	A	B	C
1	Name	Sex	Age
2	Grace	Male	18
3	Charlie	Female	60
4	Olivia	Female	27
5	Frank	Male	55
6	Charlie	Male	53

1. Your task is to import the CSV file using NumPy's **genfromtxt()** function.
 - a. Transpose the array.
 - b. Using slicing, create three arrays: an array of all the **names** in the column, an array of the **age** column, and an array of the **gender** (sex) column.
2. Using NumPy, write code that returns all the names of people who are 56 years old.
3. How many people are aged 44 or under?
4. Write another code snippet to return an array of all the males in the dataset. How many males are in the dataset?
5. Calculate the average age of all the females in the table.
6. Calculate the average age of people named Olivia and Kate.

Day 15 - Answers

1. This challenge is about using NumPy to analyze data. The first thing to do is import the `names_age_sex_data` data using the `np.genfromtxt()` function. This function will import a CSV file. Ensure that you pass a delimiter so that your data can be separated.

```
[1]: import numpy as np

arr = np.genfromtxt("names_age_sex_data.csv",
                     delimiter=",", dtype ='U7')
arr[:5]

[1]: array([['Name', 'Sex', 'Age'],
           ['Grace', 'Male', '18'],
           ['Charlie', 'Female', '60'],
           ['Olivia', 'Female', '27'],
           ['Frank', 'Male', '55']], dtype='|<U7')
```

This code returns an array of the data in the CSV file. The `dtype` parameter set to "U7" specifies that the column should be treated as Unicode strings of length up to 7 characters.

- a. The next thing that we will do is transpose the data. Transposing simply means swapping the columns with the rows, effectively making columns become rows and vice versa. We are going to use the `transpose()` method.

```
[2]: transposed_arr = arr.transpose()
transposed_arr

[2]: array([['Name', 'Grace', 'Charlie', 'Olivia', 'Frank', 'Charlie',
           'Olivia', 'Mia', 'Jos', 'Peter', 'Olivia', 'Charlie', 'David',
           'Kate', 'Thomas', 'Alice', 'Rob', 'Piper', 'Kate', 'Liam',
           'Quinn', 'Charlie', 'David', 'Ryan', 'Kate', 'John', 'Kelly'],
           ['Sex', 'Male', 'Female', 'Female', 'Male', 'Male', 'Male',
            'Female', 'Male', 'Male', 'Female', 'Female', 'Female',
            'Female', 'Female', 'Male', 'Female', 'Female', 'Female',
            'Female', 'Male', 'Male', 'Female', 'Male', 'Male', 'Female'],
           ['Age', '18', '60', '27', '55', '53', '39', '60', '56', '44',
            '57', '41', '39', '49', '37', '63', '56', '96', '41', '19', '34',
            '54', '18', '49', '19', '21', '21']], dtype='|<U7')
```

- b. The next thing is to slice three arrays from the transposed array. First, we will slice the "names" from the array. Then we will proceed to slice "age" and "gender." Here is the code below:

```
[3]: # Slicing names
names = transposed_arr[0:1,1:]
names

[3]: array(['Grace', 'Charlie', 'Olivia', 'Frank', 'Charlie', 'Olivia',
       'Mia', 'Jos', 'Peter', 'Olivia', 'Charlie', 'David', 'Kate',
       'Thomas', 'Alice', 'Rob', 'Piper', 'Kate', 'Liam', 'Quinn',
       'Charlie', 'David', 'Ryan', 'Kate', 'John', 'Kelly'], dtype='|<U7')

[4]: # Slicing age
age = transposed_arr[2:3,1:]
age

[4]: array([['18', '60', '27', '55', '53', '39', '60', '56', '44', '57', '41',
       '39', '49', '37', '63', '56', '96', '41', '19', '34', '54', '18',
       '49', '19', '21', '21']], dtype='|<U7')

[5]: # Slicing sex
gender = transposed_arr[1:2,1:]
gender

[5]: array(['Male', 'Female', 'Female', 'Male', 'Male', 'Male',
       'Male', 'Male', 'Female', 'Female', 'Female', 'Female',
       'Female', 'Male', 'Female', 'Female', 'Female', 'Female',
       'Male', 'Female', 'Male', 'Male', 'Female'], dtype='|<U7')
```

2. We are going to use Boolean indexing to return the names of people who are 56 years old.

```
[6]: # Using Boolean slicing to access names
aged_56 = names[age == '56']
print(aged_56)

['Jos' 'Rob']
```

Jos and Rob are 56 years old.

3. We are going to use Boolean indexing again to find the names of people aged 44 and under. We will then use the NumPy shape attribute to find the number of people in the array. The shape attribute will return the number of rows (i.e., the number of people) who are 44 years old or less.

```
[7]: # Using Boolean slicing to access names  
aged_44_under = names[age <= '44']  
  
# Using the shape attribute to find people under 44  
aged_44_under_num = np.shape(aged_44_under)  
print("The number of people aged 44 and under is",  
      aged_44_under_num[0])
```

The number of people aged 44 and under is 14

In this code, the `names[age <= '44']` creates a NumPy array containing only the elements from the `names` array where the corresponding age values are less than or equal to '44'. The `np.shape` basically counts the number of people in the returned array, which is 14.

4. First, we are going to filter the "names" array to return an create an array of only males. Then we will use the `shape()` function to find how many males are in the array. The `shape()` will count the number of names in the 'males' array.

```
[8]: # Names of males in the array  
males = names[gender == 'Male']  
  
# Number of males in the array  
males_number_in_array = np.shape(males)  
print(males)  
print("The number of males is:", males_number_in_array[0])  
  
['Grace' 'Frank' 'Charlie' 'Olivia' 'Jos' 'Peter' 'Rob' 'Charlie' 'David'  
 'Kate' 'John']  
The number of males is: 11
```

5. First, we will convert the "age" array into an integer data type. This will make it possible for us to calculate the average. We are going to use the "gender" array to filter the "age" array for female names and calculate the mean.

```
[9]: # Converting age data into int data type  
age = age.astype(int)  
  
# Calculating the average age of females  
average_age = age[gender == 'Female'].mean()  
print("The average females age is ", average_age)
```

The average females age is 46.2

This code outputs 46.2 as the average age.

6. To calculate the average age of persons named **Olivia** and **Kate**, we will first filter the "age" array for the ages of people named Olivia and Kate. We will then use the ages to create an array and calculate the **mean**.

```
[10]: # Filter the age array and calculating mean  
olivia_age = age[names == 'Olivia']  
kate_age = age[names == 'Kate']  
  
# Create array and calculate the average age  
average = np.array([olivia_age, kate_age]).mean()  
print(f'Average age of persons named Olivia and Kate is: {average:.2f}')
```

Average age of persons named Olivia and Kate is: 38.67

Here we create array from the ages of Olivia and Kate and we use the **mean()** method to calculate the average age.
The average age is 38.67.

Day 16: Pandas Series Analysis

In the challenges below, you will create and analyze a pandas Serie and create a DataFrame from a pandas Series. You will use the data below to answer the challenges.

```
list1 = ["wood", "red", "red", "white", "blue", "red"]
```

1. Using pandas, create a pandas Series from the list above. Write a code to find how many times each item appears in the list.
2. Using pandas, write code to check if red, white, or black are in the Series.
3. Using pandas, write code to return all **unique** values from the Series.
4. Using the pandas **update()** method, write code to update the Series you created in question 1. Add **green** and **orange** to the series. The color green will be at index 0, and orange will be at index 2.
5. Write code to check the number of dimensions of your updated Series (question 4). Write another line of code to convert the Series into a DataFrame. Your DataFrame should have **one** column of colors. Check the shape and number of dimensions of your DataFrame. Your DataFrame must have a shape of (6, 1).

Day 16 - Answers

1. To create a Series using the pandas, we can use the `pandas.Series()` function. The list of colors will be passed as an argument to this function.

```
[1]: import pandas as pd

list1 = ["wood", "red", "red", "white", "blue", "red"]

# Creating a pandas series
colors_series = pd.Series(list1)
colors_series
```



```
[1]: 0      wood
1      red
2      red
3    white
4    blue
5      red
dtype: object
```

To know how many times each item appears in the Series we can use the `Series.value_counts()` method. This function returns how many times each value appears in the Series. The resulting Series is sorted in descending order based on the counts. See the code below:

```
[2]: colors_series.value_counts()
```



```
[2]: red      3
      wood     1
      white    1
      blue     1
Name: count, dtype: int64
```

You can see that red appears three times in the output, while the other colors appear once.

2. How do we check if a given item is in the Series? We use the **isin()** method. This function will return True if the item in the series is also in the list; otherwise, it will return False.

```
[3]: colors = ["red", "white", "black"]

# Checking if the colors in the list are in series
is_in_series = colors_series.isin(colors)
is_in_series
```



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

3. To return unique values in the Series we will use the **unique()** method. This method will remove duplicates from the Series.

```
[4]: # Checking for unique values in Series
unique_values = colors_series.unique()
unique_values
```



```
[4]: array(['wood', 'red', 'white', 'blue'], dtype=object)
```

4. To update the Series with the **update()** method, we will create a dictionary of colors and their indexes. The index is the key, and the color is the value. The **update()** method will insert each color at the stipulated index. Here is the update code below:

```
[5]: # Create a dictionary of updates  
updates = {0: "green", 2: "orange"}  
  
# Update the values in the Series  
colors_series.update(updates)  
colors_series
```

```
[5]: 0    green  
1    red  
2  orange  
3  white  
4  blue  
5    red  
dtype: object
```

You can see that green has been inserted at index 0 and orange at index 2. Basically the keys have become the indices.

5. We can use the **ndim** attribute to check the number of dimensions in the Series.

```
[6]: # Checking for dimensions in the series  
colors_series.ndim
```

```
[6]: 1
```

To update the series to a DataFrame, we can use the **Series.to_frame()** method. We will use the name of the column as the argument. We will write another code to check the number of dimensions in the DataFrame and the shape of the DataFrame. Here is the code below:

```
[7]: d_frame_colors= colors_series.to_frame("colors")
d_frame_colors
```

```
[7]: colors
_____
0    green
1     red
2   orange
3   white
4    blue
5     red
```

```
[8]: # Checking for dimensions in the DataFrame
d_frame_colors.ndim
```

```
[8]: 2
```

```
[9]: # Checking the shape of the DataFrame
d_frame_colors.shape
```

```
[9]: (6, 1)
```

You can see that the Series has been updated to a DataFrame. This DataFrame has 2 dimensions and has a shape of (6, 1); meaning it has 1 column and the column has 6 items, or 6 rows.

Day 17: Creating and Modifying DataFrames

Creating and modifying DataFrame is an essential skill for data analysts because it allows them to efficiently clean, transform, analyze, and report on data. It is a foundational skill that is used in many data-related tasks and workflows. In the challenges below, you will create and modify DataFrames.

```
list1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

1. Create a DataFrame from the list above with three columns [A, B, C].
2. Using the pandas `.at` attribute access the number 9 from your DataFrame.
3. Write another code using the `.at` attribute to change the number 9 to 10.
4. Create a DataFrame using the data below:

```
names = ["John", "Mary", "Peter"]
age = [27, 34, 47]
sex = [ "male", "female", "female"]
```

Using the `.loc` attribute, access the age of John from the DataFrame.

5. Using the `.loc` attribute change the gender of Peter from "female" to "male."
6. Write code to extract a subset of the DataFrame containing only male individuals. Write another code to retrieve the `name` and `age` of the oldest male from the modified DataFrame.

Day 17 - Answers

1. We will use the `pd.DataFrame()` function to create a DataFrame from a list. Notice we specify the names of the columns in the function.

```
[1]: import pandas as pd

list1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
# Creating a DataFrame from Lists
df = pd.DataFrame(list1, columns=["A", "B", "C"])
df
```

```
[1]:   A  B  C
      0  1  2  3
      1  4  5  6
      2  7  8  9
```

2. To access the number 9 using the `.at` attribute, we pass the index of the row and the column name to the attribute. The number 9 is on index 2 and in column C. The column is passed as a string.

```
[2]: df.at[2, "C"]
```

```
[2]: 9
```

3. To modify the DataFrame using the `.at` attribute, we pass the index of the row and the column of the number we want to change. We use the `equal` sign to assign a new value.

Below, we change 9 to 10. You can see from the output that the number 9 has changed to 10 (last row).

```
[3]: # Modifying the DataFrame  
df.at[2, "C"] = 10  
df
```

```
[3]:  
     A   B   C  
0   1   2   3  
1   4   5   6  
2   7   8  10
```

4. First, we create a DataFrame with the given data. We use the `loc` attribute to locate the name "John" and their age. The `loc` stands for location.

```
[4]: names = ["John", "Mary", "Peter"]  
age = [27, 34, 47]  
sex = ["male", "female", "female"]  
  
# Creating a DataFrame from Lists  
df = pd.DataFrame({"name": names, "age": age, "sex": sex})  
  
# Locating John's age using Loc  
john_age = df.loc[df["name"] == "John", "age"]  
  
print(f"John's age is: {john_age[0]}")
```

John's age is: 27

5. We use the `.loc` attribute to locate Peter and retrieve their gender from the DataFrame. Then, by using the assignment operator (`=`), we assign a new value to Peter's gender. When we print out the DataFrame, you can see that Peter's gender has been updated to male. See below:

```
[5]: # Changing Peter's gender using Loc  
df.loc[df["name"] == "Peter", "sex"] = "male"  
df
```

```
[5]:    name  age     sex  
0   John   27   male  
1   Mary   34 female  
2  Peter   47   male
```

6. This challenge requires us to return a subset of the DataFrame containing only the male entries. To accomplish this, we will utilize the `query()` method to filter the DataFrame based on a specified condition.

```
[6]: # using query method to filter the DataFrame  
male = "male"  
males = df.query('sex == @male')  
males
```

```
[6]:    name  age     sex  
0   John   27   male  
2  Peter   47   male
```

Now to return the age and name of the oldest male, we will sort the "males" DataFrame by age and then retrieve the oldest person from the DataFrame. We will use the `sort_values()` method and the `max()` function. See the code below:

```
[7]: # Sorting the DataFrame and returning the oldest male  
oldest = males.sort_values('age', axis=0, ascending=False).max()  
  
print(f'The oldest person is {oldest["name"]}'  
      f'and he is {oldest.age} yrs old.')
```

The oldest person is Peter and he is 47 yrs old.

Day 18: Runners Data Analysis –Part 1

In this challenge, you will import the "running_data" dataset, which is a CSV file. Here is the sample below:

	A	B	C	D
1	Name	Distance (km)	Time (hours)	
2	Wendy	9.1	4.24	
3	Yara	8.61	4.16	
4	Grace	7.17	2.94	
5	Grace	6.36	2.78	
6	David	7.96	3.46	

1. Import the **running_data** dataset above.
2. Create a copy of the DataFrame. Add another column to the DataFrame that will convert the kilometers run into miles.
3. Add another column to the DataFrame to calculate the speed per hour in kilometers.
4. You have come across some more information to be added to the DataFrame. Some runners stopped for a rest. This information is saved in a CSV dataset called "**runner_rest**." Here is the dataset sample below:

	A	B	C	D
1	Runners	Rest Time (Mins)		
2	Grace	45		
3	Chris	25		
4	Frank	58		
5	Alice	47		
6	Noah	105		

- Merge this dataset into your DataFrame above. Use the `pandas merge()` method. Ensure to drop the "**Runners**" column since you already have the "**Names**" column.
5. Convert the "**Rest Time (Mins)**" column into hours using the `apply()` method. Add a new column called "**Rest Time (Hrs)**."
 6. Using the `pandas.Series.map()` function, create a subset of the DataFrame with runners that covered over 5 miles. Reset the index of the resulting DataFrame and ensure to drop the index column. Which runner covered the most distance?

Day 18 - Answers

1. First, we are going to import the "running_data.csv" file and view 5 rows using the `head()` function.

```
[1]: import pandas as pd  
  
df = pd.read_csv("running_data.csv")  
df.head()
```

```
[1]:    Name  Distance (km)  Time (hours)  
0  Wendy          9.10        4.24  
1   Yara          8.61        4.16  
2  Grace          7.17        2.94  
3  Grace          6.36        2.78  
4  David          7.96        3.46
```

2. First, we will create a copy of the DataFrame. To add the "**Distance Mi**" column to the DataFrame, we will divide the "**Distance (km)**" column by the conversion factor of 1.60934. This will result in a DataFrame with a new column.

```
[2]: # Creating a copy of the DataFrame  
df_copy = df.copy()  
  
# Converting kms to miles and creating a new column  
df_copy["Distance (Mi)"] = df_copy["Distance (km)"] / 1.60934  
df_copy.head()
```

```
[2]:    Name  Distance (km)  Time (hours)  Distance (Mi)  
0  Wendy          9.10        4.24      5.654492  
1   Yara          8.61        4.16      5.350019  
2  Grace          7.17        2.94      4.455243  
3  Grace          6.36        2.78      3.951931  
4  David          7.96        3.46      4.946127
```

3. In this step, we will add an additional column called "**speed_in_kms (hr)**" to our dataset. This column will represent the running speed in kilometers per hour, calculated by dividing the distance run in kilometers by the time taken in hours.

```
[3]: # Calculating the speed in kms  
df_copy["speed_in_kms(hr)"] = df_copy["Distance (km)"] / df_copy["Time (hours)"]  
df_copy.head()
```

	Name	Distance (km)	Time (hours)	Distance (Mi)	speed_in_kms(hr)
0	Wendy	9.10	4.24	5.654492	2.146226
1	Yara	8.61	4.16	5.350019	2.069712
2	Grace	7.17	2.94	4.455243	2.438776
3	Grace	6.36	2.78	3.951931	2.287770
4	David	7.96	3.46	4.946127	2.300578

4. For this challenge, we are going to create a DataFrame with the new information that has become available. First, we will import the "**runners_rest**" dataset. Here is the new DataFrame below:

```
[4]: df2 = pd.read_csv("runner_rest.csv")  
df2.head()
```

	Runners	Rest Time (Mins)
0	Grace	45
1	Chris	25
2	Frank	58
3	Alice	47
4	Noah	105

We will use the `merge()` method to merge the two DataFrames on the "Names" and "Runners" columns. We will call the new column "**Rest Times (Mins)**." To avoid redundancy, we need to drop the "Runners" column since the "Names" column already exists in the DataFrame.

```
[5]: # New available information
df2 = pd.read_csv("runner_rest.csv")

# Merge the two DataFrames
df_merged = df_copy.merge(df2, left_on="Name",
                           right_on="Runners",
                           how="left").drop(columns=["Runners"])

df_merged.head()
```

	Name	Distance (km)	Time (hours)	Distance (Mi)	speed_in_kms(hr)	Rest Time (Mins)
0	Wendy	9.10	4.24	5.654492	2.146226	38.0
1	Yara	8.61	4.16	5.350019	2.069712	NaN
2	Grace	7.17	2.94	4.455243	2.438776	45.0
3	Grace	6.36	2.78	3.951931	2.287770	45.0
4	David	7.96	3.46	4.946127	2.300578	NaN

5. Now we are going to use the pandas `apply()` method to convert the "**Rest Time (Mins)**" column into hours. We are going to use a `lambda` function to divide each value in the column by 60. This lambda function will be passed to the `apply()` method and applied to all items in the "**Rest Time (Mins)**" column. The resulting values will be added to a new column named "**Rest Time (Hrs)**."

```
[6]: # Converting mins to hrs and adding a new column
df_merged["Rest Time(Hrs)"] = df_merged["Rest Time (Mins)"].apply(lambda x:
                                                               x/60 if x is not None else None)

df_merged.head()
```

	Name	Distance (km)	Time (hours)	Distance (Mi)	speed_in_kms(hr)	Rest Time (Mins)	Rest Time(Hrs)
0	Wendy	9.10	4.24	5.654492	2.146226	38.0	0.633333
1	Yara	8.61	4.16	5.350019	2.069712	NaN	NaN
2	Grace	7.17	2.94	4.455243	2.438776	45.0	0.750000
3	Grace	6.36	2.78	3.951931	2.287770	45.0	0.750000
4	David	7.96	3.46	4.946127	2.300578	NaN	NaN

6. To extract a subset of the DataFrame containing runners who covered over 5 miles, we can utilize the `map()` method. By calling the `map()` method on the "**Distance (Mi)**" column, we can extract a subset of the DataFrame with over 5 miles. We use the `reset_index()` method to reset the index and drop the index column.

```
[7]: over_5_miles_run = df_merged[df_merged["Distance (Mi)"]
                                 .map(lambda x: x) > 5].reset_index(drop=True)
over_5_miles_run.head()
```

	Name	Distance (km)	Time (hours)	Distance (Mi)	speed_in_kms(hr)	Rest Time (Mins)	Rest Time(Hrs)
0	Wendy	9.10	4.24	5.654492	2.146226	38.0	0.633333
1	Yara	8.61	4.16	5.350019	2.069712	NaN	NaN
2	Noah	8.95	3.51	5.561286	2.549858	105.0	1.750000
3	Uma	8.12	3.15	5.045547	2.577778	NaN	NaN
4	Xander	9.36	2.71	5.816049	3.453875	NaN	NaN

To find the person who covered the most distance, we will use the `idxmax()` method to retrieve the index of the maximum distance covered in miles. We will use the index to retrieve the name.

```
[8]: # retrieving the index with max miles
max_distance_index = over_5_miles_run["Distance (Mi)"].idxmax()

# Using index to retrieve name with max miles
name = over_5_miles_run["Name"][max_distance_index]
print("The longest distance was achieved by", name)
```

The longest distance was achieved by Bob

You can see the longest distance was run by Bob.

Day 19: Runners Data Analysis – Part 2

Shuffling and merging DataFrames are essential skills for data analysts because they enable them to combine and explore data, build predictive models, and communicate insights through data visualization.

```
names = ["Joe", "Phil", "Ken", "Jos", "Luke"]
miles_run = [120, 80, 100, 90, 85]
time_in_hours = [40, 38, 45, 50, 50]
```

1. Create a DataFrame from the lists above using pandas
2. More information has become available below. Create a new DataFrame with this information. Merge it to the DataFrame above using pandas **concat()** function.

```
# New information
names = ["Joe", "Phil", "Ken", "Jos", "Luke"]
Age = [45, 28, 21, 55, 62]
gender = ["M", "M", "F", "F", "M"]
```

3. Shuffling DataFrames can randomize the data. Write code to reshuffle the index of the DataFrame.
4. Explain why it may be important to reshuffle the index of the data for machine learning.
5. Using the pandas **melt()** function, reshape your DataFrame to a long format. The "names" column must remain in place.

Day 19 - Answers

1. To create a DataFrame we use the pd.DataFrame() function.

```
[1]: import pandas as pd

names = ["Joe", "Phil", "Ken", "Jos", "Luke"]
miles_run = [120, 80, 100, 90, 85]
time_in_hrs = [40, 38, 45, 50, 50]

df = pd.DataFrame({"Names": names, "Miles run": miles_run,
                    "Time in hrs": time_in_hrs})
df
```

```
[1]:    Names  Miles run  Time in hrs
0     Joe        120         40
1   Phil         80         38
2     Ken        100         45
3     Jos         90         50
4   Luke         85         50
```

2. In this challenge, we have new information that needs to be incorporated into an existing DataFrame. To achieve this, we will merge two DataFrames using the **concat()** function.

When merging the DataFrames, we need to be mindful of columns with identical names to avoid duplication. In this case, the new DataFrame contains a column called "Names" that already exists in the original DataFrame. To prevent redundancy, we will drop this column during the concatenation stage.

```
[2]: # New information
names = ["Joe", "Phil", "Ken", "Jos", "Luke"]
Age = [45, 28, 21, 55, 62]
gender = ["M", "M", "F", "F", "M"]

# Creating a new DataFrame.
new_df = pd.DataFrame({'Names': names, 'Age': Age, 'Gender': gender})

# Concatenate new_df to old_df and dropping the Names column
df = pd.concat([df, new_df.drop(columns="Names")], axis=1)
df
```

```
[2]:   Names Miles run Time in hrs  Age  Gender
0    Joe      120          40   45      M
1   Phil       80          38   28      M
2    Ken      100          45   21      F
3    Jos       90          50   55      F
4   Luke       85          50   62      M
```

3. To reshuffle the rows of a DataFrame, we will use the `sample()` method. This method allows us to randomly sample a fraction of the rows in the DataFrame and return them in a shuffled order.

To shuffle all the rows of the DataFrame, we will set the `frac` parameter to 1, indicating that we want to sample the entire DataFrame. By passing a value of 1, all rows will be returned in a random order. Here is the code below:

```
[3]: # Reshuffling the dataset
df = df.sample(frac=1)
```

```
[3]:   Names Miles run Time in hrs  Age  Gender
2    Ken      100          45   21      F
1   Phil       80          38   28      M
4   Luke       85          50   62      M
3    Jos       90          50   55      F
0    Joe      120          40   45      M
```

You can see in the output (index) that the DataFrame has been reshuffled. Please note that every time you run this code, it will output a different order of DataFrame.

4. In many machine learning algorithms, the order of the data in the training set can affect the final model. For example, if the data is ordered by the target variable, an algorithm that builds a model incrementally, like gradient descent, will have an easier time fitting the model to the data.

Shuffling the data can help prevent overfitting. If the data is ordered by the target variable, an algorithm that uses all the data to make predictions will tend to overfit the data because it will learn to predict the target variable based on the order of the data rather than the underlying patterns in the data.

5. To reshape a DataFrame from a wide format to a long format, we can use the `melt()` function in pandas. The `melt()` function is used to transform columns into rows, effectively consolidating multiple columns into a single column while retaining the corresponding values.

When applying the `melt()` function, all columns in the original DataFrame will be transformed into two output columns: one for the variable names and one for the corresponding values. This process effectively 'melts' the DataFrame from a wider representation to a longer one.

```
[4]: df_set_long = df.melt(id_vars=[ "Names" ],
                           var_name="Variable",
                           value_name="Values")  
  
df_set_long
```

You can see below that the DataFrame has been transformed in a long way. The columns have become rows.

Out[4]:

	Names	Variable	Values
0	Jos	Miles run	90
1	Luke	Miles run	85
2	Ken	Miles run	100
3	Phil	Miles run	80
4	Joe	Miles run	120
5	Jos	Time in hrs	50
6	Luke	Time in hrs	50
7	Ken	Time in hrs	45
8	Phil	Time in hrs	38
9	Joe	Time in hrs	40
10	Jos	Age	55
11	Luke	Age	62
12	Ken	Age	21
13	Phil	Age	28
14	Joe	Age	45
15	Jos	Gender	F
16	Luke	Gender	M
17	Ken	Gender	F
18	Phil	Gender	M
19	Joe	Gender	M

This long format can be useful for various data analysis tasks, such as statistical modeling, or data visualization.

Day 20: Explore Data with Pandas and Matplotlib

In this challenge, you are going to create a DataFrame that you will analyze with pandas, and generate visualizations using Matplotlib.

```
products = ["sugar", "Salt", "oil", "diapers", "rice"]
costs = [2450, 1989, 6745, 9807, 8743]
sales = [27908, 4508, 6734, 9976, 9000]
```

1. Create a pandas DataFrame from the lists above. Find the most profitable product. Using Matplotlib, create a bar plot to visualize product profitability.
2. Using just pandas, find the least profitable product.
3. What is the difference in profit between the most profitable item and the least profitable item?
4. Using pandas, and Matplotlib, create a line plot of the costs and profits of all products.
5. Using pandas, calculate the average cost per product.
6. Using pandas calculate the average profit per product.

Day 20 - Answers

1. First, let's create a DataFrame using pandas. Remember to import pandas if you have not yet.

```
[1]: import pandas as pd

products = ["sugar", "Salt", "oil", "diapers", "rice"]
costs = [2450, 1989, 6745, 9807, 8743]
sales = [27908, 4508, 6734, 9976, 9000]

df = pd.DataFrame({"products":products, "costs":costs,"sales":sales})
df
```

	products	costs	sales
0	sugar	2450	27908
1	Salt	1989	4508
2	oil	6745	6734
3	diapers	9807	9976
4	rice	8743	9000

To find the most profitable product, we will calculate the profit for each product by subtracting the "costs" from the "sales" column, and then use the `idxmax()` method, which will return the index of the maximum value in the "profit" column. Then we can use that index to find the name of the product with the largest profit using the `.loc` attribute.

```
[2]: # calculate the profit for each product
df["profit"] = df["sales"] - df["costs"]

# find the index of the most profitable product
most_profitable_index = df["profit"].idxmax()

# use the index to find the corresponding product
most_profitable_product = df.loc[most_profitable_index, "products"]
print(f'The most profitable product is {most_profitable_product}')

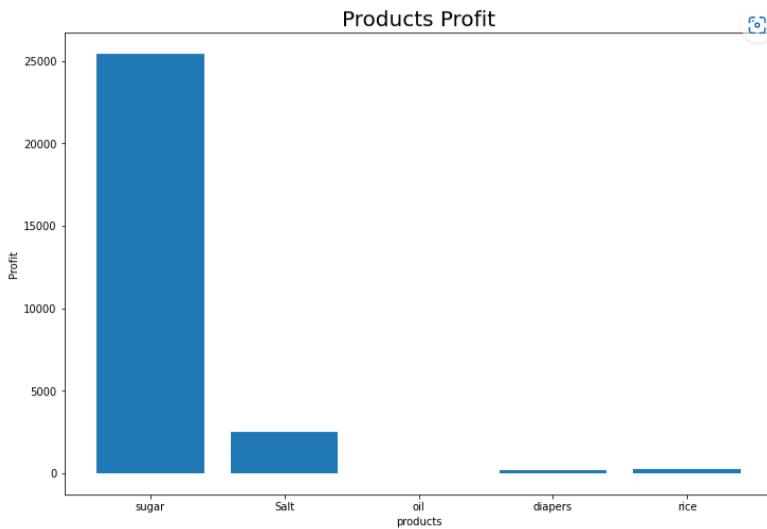
The most profitable product is sugar
```

We will use the Matplotlib library to visualize the most profitable product.

```
[3]: import matplotlib.pyplot as plt

plt.figure(figsize=(12,8))
plt.bar(df['products'], df['profit'])
plt.xlabel("products")
plt.ylabel("Profit")
plt.title("Products Profit", fontsize=20)
plt.show()
```

You can see from the plot below that the most profitable product is sugar.



- Now, to find the least profitable product using pandas, we can use the `idxmin()` method to find the index of the row with the lowest profit, and then use that index to find the corresponding product name using the `.loc` attribute.

```
[4]: # find the index of the Least profitable product
least_profit_index = df["profit"].idxmin()

# use the index to find the corresponding product
least_profit_product = df.loc[least_profit_index, "products"]
print(f'The least profitable product is {least_profit_product}')

The least profitable product is oil
```

In the code above, you must have noticed that we have passed two arguments to the `.loc` attribute: the `least_profit_index` and the `products` column. This is because we are using the `least_profit_index` to retrieve the product from the `products` column.

3. In this challenge, we are asked to find the difference in profit between the most profitable and the least profitable item. We will use the `most_profitable_index` we calculated above(question 2) to locate the profit amount of the most profitable product and the `least_profit_index` to locate the profit value of the least profitable product. In both instances, we will use the `.loc` attribute. Then, we will calculate the difference between the two values.

```
[5]: # calculate most profitable in profit
      most_profitable = df.loc[most_profitable_index, "profit"]
      # calculating the Least profitable in profit
      least_profitable = df.loc[least_profit_index, "profit"]

      # Difference between most profitable and Least profitable
      difference = most_profitable - least_profitable
      print(f'The profit difference is {difference}'')
```

The profit difference is 25469

4. Let's create a line plot of the costs and profits of the products using Matplotlib. On the `x-axis`, we will have the products, and on the `y-axis`, we will have the product costs and profit. We will use the `plot()` method to create a plot.

```
[6]: df.plot(x="products", y=["costs", "profit"],
            kind="line",
            figsize=(12, 8))
plt.xlabel("Products")
plt.ylabel("Cost and Profit")
plt.title("Cost and Profit Analysis", fontsize = 20)
plt.show()
```

The output will be this plot below.



5. The easiest way to calculate the average cost using pandas, is to use the **mean()** method on the **costs** column:

```
[7]: average_cost = df["costs"].mean()
print('The average costs is', average_cost)
```

The average costs is 5946.8

6. This is similar to the previous question. The difference is that we are going to apply the **mean()** method to the **profit** column. This will return the average profit.

```
[8]: average_profit = df["profit"].mean()
print('The average_profit per product is', average_profit)
```

The average_profit per product is 5678.4

Day 21: Processing Data with Pandas

For this challenge, you will import the `asset_data_analysis` file. Here is a sample of the data below saved as a JSON file:

	date	products	sales	costs
3	2021-11-07	cars	15900	10910
4	2021-12-06	boats	12087	7087
5	2021-12-09	cars	56897	40447
6	2021-11-10	cars	11879	5879
7	2021-12-06	houses	19345	13451

1. Import the JSON file using pandas. View the last 5 rows of the DataFrame.
2. Using pandas, create a copy of the DataFrame and convert the "date" column into a pandas datetime format. Set the date as the index of the DataFrame. The "date" column should not be deleted when it is set as an index.
3. Using Matplotlib, create a line trend plot of sales and date.
4. Using the original DataFrame (question 1), write another code to set a hierarchical index, with the date column as the outer index and the products column as the inner index. Save this as a new variable.
5. One of the benefits of setting a multiindex is that it makes it easy to filter data. Using the hierarchical index you set in question 4, calculate the profit of "Houses" on December 12, 2021.
6. Use the `apply()` method to apply a thousand separators to the "sales" and "cost" columns (DataFrame from question 4).
7. Count how many times each product appears in the "products" columns. Which product appears the most?

Day 21 - Answers

1. First we import the **asset_data_analysis** dataset using `pd.read_json()`. We use `tail()` method to view the last 5 rows.

```
[1]: import pandas as pd

df = pd.read_json("asset_data_analysis.json", orient= "split")
df.tail()
```

	date	products	sales	costs
3	2021-11-07	cars	15900	10910
4	2021-12-06	boats	12087	7087
5	2021-12-09	cars	56897	40447
6	2021-11-10	cars	11879	5879
7	2021-12-06	houses	19345	13451

2. First, it is recommended to create a copy of the original DataFrame to avoid modifying the original data unintentionally. We can use the `copy()` method to create a copy.

Next, we need to convert the "date" column to the pandas datetime format. This can be done using the `pd.to_datetime()` function, which converts the values to a datetime data type.

Once the date column is in the datetime format, we can set it as the index of the DataFrame. The `set_index()` method is used for this purpose. By default, the `set_index()` method removes the column being set as the index from the DataFrame. However, if we want to keep the "date" column in the DataFrame while setting it as the index, we need to set the `drop` parameter to False. This is what we do in the code below. This ensures that the column is retained even after setting it as the index.

```
[2]: # Creating a copy of the DataFrame
df_copy = df.copy()

# Setting date as index
df_copy['date'] = pd.to_datetime(df_copy['date'])
df_copy = df.set_index('date', drop=False)
df_copy.head()
```

			date	products	sales	costs
			date			
	2021-11-20	2021-11-20		cars	19234	12340
	2021-12-12	2021-12-12		boats	87598	67568
	2021-12-06	2021-12-06		houses	20989	11999
	2021-11-07	2021-11-07		cars	15900	10910
	2021-12-06	2021-12-06		boats	12087	7087

You can see from the output that we have the date set as the index, and we still have a **date** column in the DataFrame.

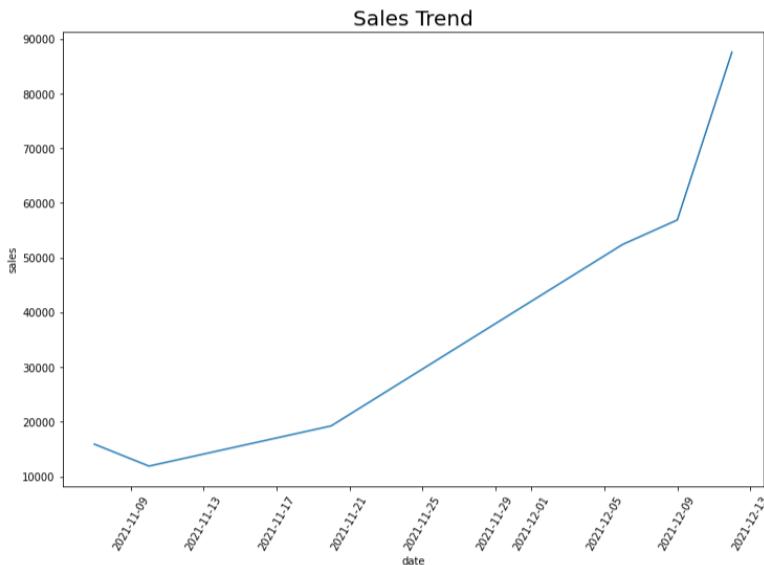
3. We are now going to use Matplotlib to create a line plot of the sales and dates. We will group the sales by the index (the index is the date) using the **groupby()** method.

```
[3]: import matplotlib.pyplot as plt

# Grouping by the index(date)
df_grouped = df_copy.groupby(df_copy.index).sum(numeric_only=True)

plt.figure(figsize=(12, 8))
plt.plot(df_grouped.index, df_grouped.sales)
plt.xlabel('date')
plt.xticks(rotation = 60)
plt.ylabel('sales')
plt.title("Sales Trend", fontsize=20)
plt.show()
```

This will output:



4. This challenge requires that we modify the original DataFrame (question 1) by setting a hierarchical index of the "date" and "product" columns. We are going to use the `set_index()` method. We will pass the "date" and "products" columns as arguments.

```
[4]: # Setting hierarchical index
df2 = df.set_index(['date', "products"])
df2
```

		sales	costs
		date	products
	2021-11-20	cars	19234 12340
	2021-12-12	boats	87598 67568
	2021-12-06	houses	20989 11999
	2021-11-07	cars	15900 10910
	2021-12-06	boats	12087 7087
	2021-12-09	cars	56897 40447
	2021-11-10	cars	11879 5879
	2021-12-06	houses	19345 13451

You can see that this outputs a DataFrame with a hierarchical index, with the **date** set as the main index and the "**products**" column as the inner index.

5. This challenge requires that we use the hierarchical index to calculate the profit on houses on June 12, 2021. Remember that our index has two columns: **date** and **products**. The first thing that we will pass under the date index is the date we are interested in (12-06-2021). Then, under the products index, we will pass the product "houses." We will pass this on to the **.loc** attribute to first filter sales and then filter costs based on the date (12-06-2021) and product (houses). We will use the **sum()** function to sum the sales and costs. Profit is the difference between the two. Here is the code below:

```
[5]: # Using multi-indexing to filter sales
sales_on_12 = df2.loc[( ["12-6-2021"], "houses"), "sales"].sum()

# Using multi-indexing to filter costs
costs_on_12 = df2.loc[( ["12-6-2021"], "houses"), "costs"].sum()

# Calculating the difference between costs and sales
profit_on_12 = sales_on_12 - costs_on_12
print('Profit on houses on the 12th of June is ', profit_on_12)
```

Profit on houses on the 12th of June is 14884

6. To add a thousand separators to the columns, we are going to use the **apply()** method. We are going to pass a function to the **apply()** function, and this function will be applied to the specific columns.

```
[6]: def thousand_separator(df, sales, costs):
        df[sales] = df[sales].apply(lambda x: '{:,}'.format(x))
        df[costs] = df[costs].apply(lambda x: '{:,}'.format(x))
    return df2
thousand_separator(df2, "sales", "costs")
```

		sales	costs
	date	products	
2021-11-20	cars	19,234	12,340
2021-12-12	boats	87,598	67,568
2021-12-06	houses	20,989	11,999
2021-11-07	cars	15,900	10,910
2021-12-06	boats	12,087	7,087
2021-12-09	cars	56,897	40,447
2021-11-10	cars	11,879	5,879
2021-12-06	houses	19,345	13,451

This code applies a thousand separators to the sales and cost columns. You can see that the columns are easily readable now.

- To find the count of each product, we can use the `groupby()` method. We can group the products and use the `size()` method to count how many times each product appears in the column. Since we are looking for the product that appears the most, we group by the `products` column and sort the DataFrame in descending order using the `sort_values()` method. We will return the top-most product.

```
[7]: df3 = df2.groupby(['products']).size().sort_values(ascending=False)
df3
```

products	count
cars	4
boats	2
houses	2

```
[8]: # Returning the product that appears the most
df3.index[0], df3.iloc[0]
```

```
[8]: ('cars', 4)
```

Day 22: Data Preprocessing and Analysis

Part of data preprocessing involves cleaning, transforming, and organizing data to make it suitable for analysis. It includes tasks such as removing duplicate entries, handling missing values, and transforming data types. The main goal of preprocessing is to improve the quality of the data, reduce noise and inconsistencies, and make it easier to analyze.

```
names = ["Carol", "Kate", "Jane", "Kuda", "Tito", "Kuku"]
age = [23, np.nan, 34, 56, np.nan, 44]
```

1. Using pandas create a DataFrame with the two lists above. Write a code that counts how many missing values are in each column.
2. Create a copy of the DataFrame in **question 1** and drop all the missing values.
3. You decided you don't want to drop the missing numbers, but replace them with the **mean** of the numbers in the column. Create another copy of the DataFrame (from the original DataFrame in question 1) and fill in the missing values with the **mean** of the column.
4. You have come across some more data:

```
gender_values =[ "F", "F", "F", "M", "M", "M"]
```

Update your DataFrame (question 3) with the gender column using the **gender_values** list above. Use the **melt()** method to reshape the DataFrame and create a new DataFrame with **three** columns: the "names" column, the "gender" and "age" columns will be combined, and their values will be in the **values** column. This will change the DataFrame from a wide format to a long format.

5. Using the pandas `where()` method, write a code to return a DataFrame of `names` and `gender` values only from the "melted" DataFrame (question 4). Your DataFrame should not have NaN values. Save this to the new variable.
6. Convert the column header of the DataFrame in question 5 to uppercase letters.

Day 22 - Answers

1. First, we are going to create a DataFrame using the `pd.DataFrame()` function. Then we will use the `isna()` method to check for missing values in the data. The `isna()` method returns Boolean values. We will use the `sum()` function to sum all True values.

```
[1]: import pandas as pd
      import numpy as np

      names = ["Carol", "Kate", "Jane", "Kuda", "Tito", "Kuku"]
      age = [23, np.nan, 34, 56, np.nan, 44]
      # Creating the DataFrame
      df = pd.DataFrame({"names": names, "age": age})
      df
```

	names	age
0	Carol	23.0
1	Kate	NaN
2	Jane	34.0
3	Kuda	56.0
4	Tito	NaN
5	Kuku	44.0

Now, let's get to the missing values. We are going to use the `isna()` method and the `sum()` method.

```
[2]: # Checking for missing values in the DataFrame
      df.isna().sum()
```

	names	0
age		2
	dtype:	int64

You can see the "names" column has no missing values, and the "age" columns has two (2) missing values.

2. The pandas **dropna()** method will drop missing values from a DataFrame.

```
[3]: # Creating a copy of the DataFrame  
df_copy = df.copy()  
# Using dropna to drop missing values  
df_copy = df_copy.dropna()  
df_copy
```

```
[3]:   names    age  
0  Carol  23.0  
2   Jane  34.0  
3   Kuda  56.0  
5   Kuku  44.0
```

You can see the output has no missing values.

3. Pandas has the **fillna()** method that we can use to fill in missing values. We are going to use the **mean()** method to fill in the missing values with the column average.

```
[4]: df_copy_2 = df.copy()  
  
# Filling missing values with mean of the column  
df_copy_2 = df_copy_2.fillna(df_copy_2["age"].mean(), inplace = False)  
df_copy_2
```

```
[4]:   names    age  
0  Carol  23.00  
1   Kate  39.25  
2   Jane  34.00  
3   Kuda  56.00  
4   Tito  39.25  
5   Kuku  44.00
```

4. The first thing we have to do is update the DataFrame with the "gender" column. Then we will reshape the DataFrame with the **melt()** method.

```
[5]: gender_values =["F", "F", "F", "M", "M", "M"]

# Add a gender column to the dataframe
df_copy_2['gender'] = gender_values
df_copy_2
```

```
[5]:   names    age  gender
0  Carol  23.00      F
1   Kate  39.25      F
2   Jane  34.00      F
3   Kuda  56.00      M
4   Tito  39.25      M
5   Kuku  44.00      M
```

To reshape the DataFrame using the `melt()` method in pandas, we can specify the columns that should remain unchanged by using the `id_vars` parameter. Below, we want to reshape the DataFrame by combining the `age` and `gender` columns while keeping the `names` column unchanged. We pass "names" as the value for the `id_vars` parameter.

```
[6]: melted_df = df_copy_2.melt(id_vars=["names"],
                                value_name="values",
                                var_name="ages_gender")
melted_df
```

The resulting DataFrame (`melted_df`) has three columns: `names`, `ages_gender` and `values`. The `names` column is unchanged, while the `age` and `gender` columns have been combined into the `ages_gender` column. The corresponding values have been stored in the `values` column. See the output below:

```
[6]:      names  ages_gender  values
          0   Carol           age    23.0
          1   Kate            age   39.25
          2   Jane            age   34.0
          3   Kuda            age   56.0
          4   Tito            age   39.25
          5   Kuku            age   44.0
          6   Carol           gender     F
          7   Kate            gender     F
          8   Jane            gender     F
          9   Kuda            gender     M
          10  Tito            gender     M
          11  Kuku            gender     M
```

5. The pandas `where()` method is a conditional method that allows you to perform a conditional operation on a DataFrame or Series. It basically takes a condition and applies it to the DataFrame. By default, the values that do not satisfy the condition are replaced by `NaN` values. In the code below, we want the `where()` method to go into the `ages_gender` column and return all rows that have the word "gender" in them. We will use the `dropna()` method to drop `NaN` values after applying the `where()` method.

```
[7]: names_gender = melted_df.where(melted_df["ages_gender"]=="gender").dropna()
      names_gender
```

	names	ages_gender	values
6	Carol	gender	F
7	Kate	gender	F
8	Jane	gender	F
9	Kuda	gender	M
10	Tito	gender	M
11	Kuku	gender	M

6. Since the column headers are in string format, to convert them into uppercase letters, we will use the `.str.upper()` method. Here is the code below:

```
[8]: # Converting the column headers into uppercase letters
names_gender.columns = names_gender.columns.str.upper()
names_gender
```

```
[8]:    NAMES   AGES_GENDER   VALUES
      6   Carol        gender       F
      7   Kate         gender       F
      8   Jane         gender       F
      9   Kuda         gender       M
     10   Tito         gender       M
     11   Kuku         gender       M
```

You can see that the three column headers have been converted into uppercase letters.

Day 23: Preprocessing with Pandas and Matplotlib

For the challenges below, you are going to use the data below. You will create a DataFrame from the three lists.

```
names = ["Kelly", np.nan, 'Jon', 'Ken', 'Tim', 'Pel']
grades = [30, 40, 30, 67, np.nan, 55]
age =[15, np.nan, 18, 17, np.nan, 16]
```

1. Create a pandas DataFrame from the lists above. Write a code to check if there are any missing values in the columns of the DataFrame.
2. Your supervisor has asked that you show them how to fill in the missing values using the library's Sklearn. Write code to fill in missing values in the DataFrame above using this library. Replace the missing value in the "Name" column with the name "Paul." Use the **mean** strategy for the "Grades" column and the **median** strategy for the "Age" column. Make a copy of the original DataFrame.
3. Using the original DataFrame (question 1) with missing values, identify and drop any columns that have more than 30% missing values. Save this as a new variable.
4. Using pandas, check for any duplicate values in the "Names" columns of your DataFrame (question 3).
5. Use the pandas **groupby()** method and **agg()** method to calculate the mean, minimum, and maximum of the "Grades" column, grouped by the "Names" column. Use a copy of the DataFrame you saved in question 2.

Day 23 - Answers

1. First, let's create a DataFrame from the given lists:

```
[1]: import pandas as pd
      import numpy as np

      names = ["Kelly", np.nan, 'Jon', 'Ken', 'Tim', 'Pel']
      grades = [30, 40, 30, 67, np.nan, 55]
      age = [15, np.nan, 18, 17, np.nan, 16]
      df = pd.DataFrame({'Names': names, 'Grades':grades, 'Age': age})
```

	Names	Grades	Age
0	Kelly	30.0	15.0
1	NaN	40.0	NaN
2	Jon	30.0	18.0
3	Ken	67.0	17.0
4	Tim	NaN	NaN
5	Pel	55.0	16.0

Now, let's check for missing values:

```
[2]: # Checking for missing values
      df.isnull().sum()
```

```
[2]: Names      1
      Grades     1
      Age        2
      dtype: int64
```

You can see that the **Names** and **Grades** columns have one missing value each. The **age** column has 2.

2. Sklearn is a library in Python that is commonly used for machine learning tasks. To fill in missing values in the DataFrame using this library, the first step is to install the library by running the command! **pip install -U Sklearn**. First, we will create a copy of the DataFrame. Next, we will

import the **SimpleImputer** class from the **sklearn.impute** module and create three instances of the class. We will use the "constant" strategy for filling the **Names** column with the name Paul, the "mean" strategy to fill the **Grades** column, and the "median" strategy to fill the **Age** column. Then we will fit the imputer to the DataFrame, and use the **transform** method to fill in missing values in the DataFrame.

Notice below that we use double brackets "[[]]" to select the columns we want to impute. This is because the **fit_transform** method of SimpleImputer expects a 2D array or DataFrame as input. When we select a single column of a DataFrame using the single bracket notation (e.g., `df['A']`), we get a Series object, which is a 1D array-like object. So, we use double quotes to convert a 1D array into a 2D array.

```
[3]: from sklearn.impute import SimpleImputer

# Create a copy of the DataFrame
df2 = df.copy()
# create an instance of SimpleImputer with different strategies for each column
imputer_names = SimpleImputer(strategy='constant', fill_value="Paul")
imputer_grades = SimpleImputer(strategy='mean')
imputer_age = SimpleImputer(strategy='median')

# fit and transform the imputer on selected columns
df2[['Names']] = imputer_names.fit_transform(df2[['Names']])
df2[['Grades']] = imputer_grades.fit_transform(df2[['Grades']])
df2[['Age']] = imputer_age.fit_transform(df2[['Age']])
df2
```

	Names	Grades	Age
0	Kelly	30.0	15.0
1	Paul	40.0	16.5
2	Jon	30.0	18.0
3	Ken	67.0	17.0
4	Tim	44.4	16.5
5	Pel	55.0	16.0

You can see in the output that the DataFrame does not have any missing data.

3. To identify and drop any columns in the original DataFrame that have more than 30% missing values, we will first check the percentage of missing values in each column using the `isnull()` method and the `mean()` method. We will then use Boolean indexing to select columns that have more than 30% missing values and drop them using the `drop()` method. The `age` column is dropped because it has more than 30% missing values.

```
[4]: # Check percentage of missing values in each column
missing_values = df.isnull().mean()

# Select columns that have more than 50% missing values
columns_to_drop = missing_values[missing_values > 0.30].index

# Drop columns from the dataframe
df_cleaned = df.drop(columns_to_drop, axis=1)
df_cleaned
```

```
[4]:
```

	Names	Grades
0	Kelly	30.0
1	NaN	40.0
2	Jon	30.0
3	Ken	67.0
4	Tim	NaN
5	Pel	55.0

4. To check for any duplicate values in the `Names` column, we use the `duplicated()` method with the `sum()` method. We will pass the `Names` column as the argument to the `duplicated()` method. If a row is duplicated, the `duplicated` method will return a True Boolean value. The `sum()` method will sum all the True values. If there are no duplicated values, it will return zero (0).

```
[5]: # Check for duplicated values
df_cleaned.duplicated("Names").sum()
```

```
[5]: 0
```

We have zero duplicated values in the DataFrame.

5. To calculate the **mean**, **minimum**, and **maximum** of the "**Grades**" column, grouped by the "**Names**" column, we can use the **groupby()** method to group the data. We use the **agg()** method to calculate the mean, min, and max of the **Grades** column.

The resulting "**sch_data_agg**" DataFrame will contain the calculated statistics, with the "**Names**" column serving as the index and the "**Grades**" column aggregated using the specified functions (mean, min, max). Here is the code below:

```
[6]: # calculate mean, min, and max of 'Grades' column grouped by 'Age' column
sch_data_agg = df2.groupby('Names').agg({'Grades': ['mean', 'min', 'max']})
sch_data_agg
```

```
[6]:          Grades
              mean   min   max
Names
Jon    30.0  30.0  30.0
Kelly  30.0  30.0  30.0
Ken    67.0  67.0  67.0
Paul   40.0  40.0  40.0
Pel    55.0  55.0  55.0
Tim    44.4  44.4  44.4
```

Day 24: Business Data Analysis

Can you analyze business data with pandas and Matplotlib? Business data analysis is an important skill to have because it helps businesses stay competitive and make data-driven decisions that can lead to improved performance, increased profits, and better outcomes for customers and stakeholders.

For this challenge, you are going to work with an Excel file called **Asset_sales_data**. Here is the sample data below:

	A	B	C
1	date	products	sales
2	11-20-2021	cars	19234
3	12-12-2021	boats	87598
4	12-06-2021	houses	20989
5	11-07-2021	cars	15900

1. Using pandas import the Excel file above. Write a code to view the first 5 rows of your DataFrame.
2. Write another code that will return the data types of all the columns in the DataFrame.
3. Using pandas, what month had the highest value of sales?
4. Write a code to return the value of sales between 11-20-2021 and 12-06-2021. Create a DataFrame and return the total sales value for this period.
5. Using Matplotlib, create a pie chart of the products and their sales values as percentages. Your chart should have labels and a title. Add explode (0, 0.1, 0) and a shadow.
6. Use pandas to create a pivot table and calculate the sum of the **sales** column grouped by the **products** column. This will give you the total sales for each product. Use pandas and Matplotlib to plot this on a bar plot. Your plot size must be: width = 12, height = 10. Your plot title will be "Total Sales Per Product."

Day 24 - Answers

1. We are going to import the "Asset_sales_data" xlsx file. Since this is an Excel file, we will use the `pd.read_excel()` function to open the file. We will use the `head()` method to view the first five rows of the DataFrame.

```
[1]: import pandas as pd

df = pd.read_excel("Asset_sales_data.xlsx")
df.head()
```

```
[1]:      date  products  sales
0  11-20-2021      cars   19234
1  12-12-2021     boats   87598
2  11-07-2021      cars   15900
3  12-06-2021     boats   12087
4  12-09-2021      cars   56897
```

2. To return the data types of the columns in the DataFrame, we can use the `dtype` attribute.

```
[2]: print(df.dtypes)

date        object
products    object
sales       int64
dtype: object
```

3. To return the month with the highest sales, we need to work with the `date` column. Like we saw in question 2, the date column data type is "object." This format is not recognized as a date by pandas. We need to convert the `date` column to a date format that pandas can recognize.

We will use the `pd.to_datetime()` function to convert the "date" column into pandas datetime format. Then we will create a "month" column from the date column (we will extract the month using the `dt.month` attribute). To find the

month with the highest sales, we will group sales by the month column using the pandas `groupby()` method.

```
[3]: # Convert date to pandas datetime object.  
df["month"] = pd.to_datetime(df['date']).dt.month  
df.head()
```

```
[3]:      date  products  sales  month  
0  11-20-2021      cars   19234    11  
1  12-12-2021     boats   87598    12  
2  11-07-2021      cars   15900    11  
3  12-06-2021     boats   12087    12  
4  12-09-2021      cars   56897    12
```

You can see above that the code converts the date to pandas datetime format and creates a new "month" column.

Now we can use the `groupby()` method with the `idxmax()` method to extract the month (index) with the highest value of sales.

```
[4]: # Use groupby and idxmax to find month with max sales  
max_sales_month = df.groupby(df['month']).sum(numeric_only=True).idxmax()  
max_sales_month
```



```
[4]: sales    12  
dtype: int32
```

You can see from the output that December had the highest sales value. We can remove the `idxmax()` method if we want to see each month's total sales values. The code below outputs the month and its total sales.

```
[5]: # Use groupby and idxmax to find column with max sales  
sales_per_month = df.groupby(df['month']).sum(numeric_only=True)  
sales_per_month
```



```
[5]:      sales  
month  
11  111736  
12  565688
```

4. To filter DataFrame by a date range, we can use the `.loc` attribute. We are going to save this to a new variable.

```
[6]: # Using Loc to filter DataFrame by Date
df2 = df.loc[(df['date'] >= '11-20-2021') & (df['date'] <= '12-06-2021')]
print(df2)
```

	date	products	sales	month
0	11-20-2021	cars	19234	11
3	12-06-2021	boats	12087	12
5	12-06-2021	houses	19345	12
6	11-20-2021	cars	19234	11
8	12-06-2021	houses	20989	12
10	12-06-2021	boats	12087	12
12	11-25-2021	cars	22234	11
14	12-06-2021	boats	14987	12
16	11-20-2021	cars	19234	11

You can see now that we have a DataFrame with the date range that we want. We can now sum the sales column using the `sum()` method. This will give us the total sales for this period.

```
[7]: df2["sales"].sum()
```



```
[7]: 159431
```

5. Now let's display the results on a **pie** plot using Matplotlib. Since some products appear in more than one row in the DataFrame, the first thing we should do is use the `groupby()` method to aggregate the sales by product. The sum of the sales will be used as the data for the pie chart. We will convert the sales data into percentages. Here is the code below:

```
[8]: import matplotlib.pyplot as plt

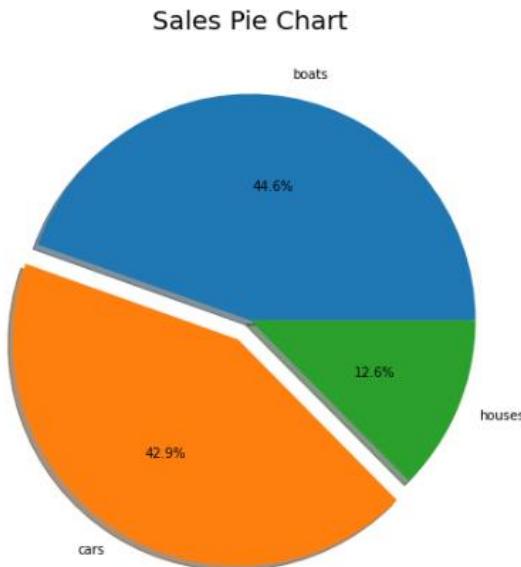
prod_sales = df.groupby(['products'])['sales'].sum()

explode = (0, 0.1, 0)

plt.figure(figsize=(12, 8))
plt.pie(prod_sales,
        explode=explode,
        labels=prod_sales.index,
        autopct='%1.1f%%',
        shadow=True)

plt.title("Sales Pie Chart", fontsize=20)
plt.show()
```

You can see from the output below that cars were the highest earners for this period of results. Houses brought in the least profit.

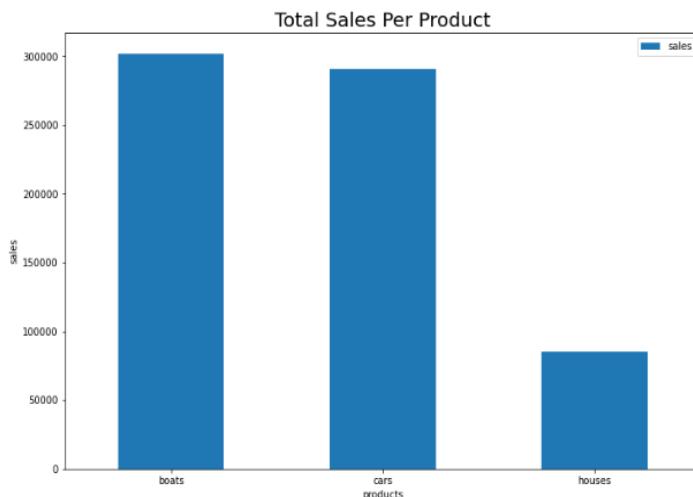


6. The `pivot_table()` method can be used to create a pivot table. This method will take the `products` column as the index and the sales column as the values. The sales values will be grouped by product. When we pivot the data, it will be easy to see how much revenue each product brings in.

```
[9]: sales_per_product = df.pivot_table(values='sales',
                                         index='products',
                                         aggfunc='sum')
```

```
[9]:      sales
products
    boats    301955
    cars    290427
    houses   85042
```

```
[10]: sales_per_product.plot(kind="bar", figsize=(12, 8))
plt.title("Total Sales Per Product", fontsize = 20)
plt.xticks(rotation = 0)
plt.ylabel("sales")
plt.show()
```



Day 25: Retail Data Processing and Analysis - Part 1

You are going to import the **retail_shop_data** CSV file below. You will preprocess and gain some insights into the data.

	A	B	C	D	E	F	G	H
1	Date	Customer	Product ID	Product Name	Cost	Price	Quantity	Total
2	1 01 2023	1001	101	T-Shirt	18	20	2	40
3	1 01 2023	1002	102	Jeans	35	50	1	50
4	2 01 2023	1003	103	Hoodie	25	30	3	90
5	3 01 2023	1004	104	Sneakers	55	70	2	140
6	3 01 2023	1005	105	Sunglasses	17	25	1	25
7	4 01 2023	1006	106	Backpack	40	45	2	90

1. Import the **retail_shop_data** file and view the first 5 rows. Write another code to check how many rows and columns are in the DataFrame. Are there any duplicates?
2. Create a copy of the DataFrame. Now, write another line of code to rename columns: "Total" to "Revenue," "Price" to "Price Per Product," and "Cost" to "Cost Per Product."
3. Calculate the total cost of each product and add this as a column to the DataFrame. Name this column: Costs. Calculate the difference between total revenue and total expenses. Name this column: Profit.
4. Add another column called "Filter." This column should check for all products that have a profit margin above \$15. If a product has a profit margin of over \$15, it should be given a value of True; otherwise, it should be False.
5. Which products have a profit margin of over \$15? Use pandas.

Day 25 - Answers

1. We are going to import the "retail_shop_data" dataset using the `pd.read_csv()` function. We will use the `head()` method to view the first five rows. We will use the `shape` attribute to check the number of rows and columns in the dataset. We will check for duplicates using the `duplicated()` method.

```
[1]: import pandas as pd  
  
df = pd.read_csv("retail_shop_data.csv")  
df.head()
```

	Date	Customer ID	Product ID	Product Name	Cost	Price	Quantity	Total
0	1 01 2023	1001	101	T-Shirt	18	20	2	40
1	1 01 2023	1002	102	Jeans	35	50	1	50
2	2 01 2023	1003	103	Hoodie	25	30	3	90
3	3 01 2023	1004	104	Sneakers	55	70	2	140
4	3 01 2023	1005	105	Sunglasses	17	25	1	25

```
[2]: # Checking the number of rows in the dataset  
df.shape[0]
```

```
[2]: 21
```

```
[3]: # Checking number of columns:  
df.shape[1]
```

```
[3]: 8
```

```
[4]: # Checking for duplicates  
df.duplicated('Product Name').sum()
```

```
[4]: 1
```

We have a duplicate in the "Product Names" column.

2. In this challenge, we are renaming the columns of the DataFrame using the `rename()` method. The first argument is a dictionary that maps the old column names to the new column names.

```
[5]: df.rename(columns={'Total': 'Revenue', 'Cost': 'Cost Per Product',
                      'Price': 'Price Per Product'}, inplace=True)
df.head()
```

	Date	Customer ID	Product ID	Product Name	Cost Per Product	Price Per Product	Quantity	Revenue
0	1 01 2023	1001	101	T-Shirt	18	20	2	40
1	1 01 2023	1002	102	Jeans	35	50	1	50
2	2 01 2023	1003	103	Hoodie	25	30	3	90
3	3 01 2023	1004	104	Sneakers	55	70	2	140
4	3 01 2023	1005	105	Sunglasses	17	25	1	25

3. First, we need to calculate the total costs by multiplying the "cost per product" column by the "quantity" column. This will create a "Costs" column. Now, let's add a profit column to the DataFrame. The value of this column will be the difference between the values in the "Revenue" and "Costs" columns.

```
[6]: # Adding the total costs column to the DataFrame
df['Costs'] = df['Cost Per Product'] * df['Quantity']

# Adding the profit column
df['Profit'] = df['Revenue'] - df['Costs']
df.head()
```

	Date	Customer ID	Product ID	Product Name	Cost Per Product	Price Per Product	Quantity	Revenue	Costs	Profit
0	1 01 2023	1001	101	T-Shirt	18	20	2	40	36	4
1	1 01 2023	1002	102	Jeans	35	50	1	50	35	15
2	2 01 2023	1003	103	Hoodie	25	30	3	90	75	15
3	3 01 2023	1004	104	Sneakers	55	70	2	140	118	30
4	3 01 2023	1005	105	Sunglasses	17	25	1	25	17	8

4. In this challenge, we are adding another column called "Filter" to the DataFrame. This column will return True if a product has a profit above \$15. If the product has a profit below \$15, it will return False.

```
[7]: df['Filter'] = df["Profit"] > 15
df.head()
```

[7]:	Date	Customer ID	Product ID	Product Name	Cost Per Product	Price Per Product	Quantity	Revenue	Costs	Profit	Filter
0	1 01 2023	1001	101	T-Shirt	18	20	2	40	36	4	False
1	1 01 2023	1002	102	Jeans	35	50	1	50	35	15	False
2	2 01 2023	1003	103	Hoodie	25	30	3	90	75	15	False
3	3 01 2023	1004	104	Sneakers	55	70	2	140	110	30	True
4	3 01 2023	1005	105	Sunglasses	17	25	1	25	17	8	False

5. We are going to use the pandas `query()` method to filter the DataFrame and return all products that have a profit of over \$15. We will filter the "Filter" column for True values. The `query()` method takes a string as an argument, which is a Boolean expression that is evaluated for each row in the DataFrame. This will filter the DataFrame, keeping only the rows where the expression is True.

```
[8]: # Using filter to filter True values from DataFrame
df.query("Filter == True")['Product Name']
```

```
[8]: 3      Sneakers
      7      Jacket
     13      Jacket
Name: Product Name, dtype: object
```

This returns a list of products that have a profit over \$15; in this case, it is "sneakers" and "jackets."

Another method would be to use Boolean filtering. This will return all product names that have a True value in the filter column.

```
[9]: print(df[df['Filter'] == True]['Product Name'])
```

```
3      Sneakers
      7      Jacket
     13      Jacket
Name: Product Name, dtype: object
```

Day 26: Retail Data Processing and Analysis – Part 2

You will continue to analyze the **retail_shop_data** CSV file below. You will use Matplotlib to create some visualizations.

	A	B	C	D	E	F	G	H
1	Date	Customer	Product ID	Product Name	Cost	Price	Quantity	Total
2	1 01 2023	1001	101	T-Shirt	18	20	2	40
3	1 01 2023	1002	102	Jeans	35	50	1	50
4	2 01 2023	1003	103	Hoodie	25	30	3	90
5	3 01 2023	1004	104	Sneakers	55	70	2	140
6	3 01 2023	1005	105	Sunglasses	17	25	1	25
7	4 01 2023	1006	106	Backpack	40	45	2	90

1. Using pandas, write code to check the difference in profit between jackets and sneakers. Return the absolute value of the profit.
2. Using pandas, what is the difference in costs between the most profitable product and the least profitable product.
3. Write code to access the total costs of jackets (using the `.loc` attribute).
4. Using Matplotlib, create a **bar stack plot** of the sales, costs, and profits of the 6 least profitable products. The bar plot should be sorted by profit in ascending order.

Day 26 - Answers

- Now, to calculate the difference in profit between jackets and sneakers, we have to group the data by "Products Name" column and sum the "Profit" column. Then we can use the `.loc` attribute to find the profit of jackets and the profit of sneakers from the grouped data and calculate the difference. We use the `abs()` function to return the absolute value of the number.

```
[10]: # Group data by products and sum profit
df_data = df.groupby("Product Name")["Profit"].sum()

jackets_sneakers_profit = abs(df_data.loc["Jacket"] - df_data.loc["Sneakers"])
print(f'The profit difference between Jackets and Sneakers {jackets_sneakers_profit}')

The profit difference between Jackets and Sneakers 21
```

- In this challenge, to find the difference in cost between the most and least profitable products, we can sort the DataFrame by profit in ascending order. The least profitable product will be in the first row, and the most profitable product will be in the last row. We will subtract the costs in the first row from the costs in the last row to find the difference in costs. We will use the `.iloc` attribute to locate the cost values.

```
[11]: # Sort DataFrame by profit
df.sort_values("Profit", inplace=True)
cost_diff = df.iloc[-1]["Costs"] - df.iloc[0]["Costs"]

print('The difference in costs between the most profitable',
      'product and least profitable is', cost_diff)

The difference in costs between the most profitable product and least profitable is 98
```

- To access the cost of jackets, since the jacket product appears in the DataFrame more than once, we will group the DataFrame by the "Products Name" using the `groupby()` method column and sum the "Costs" column using. We will use the `.iloc` attribute to access the costs of "Jackets."

```
[12]: # Group data by Product Name and sum costs
df_data = df.groupby("Product Name")["Costs"].sum()

# Use loc to find the product costs
df_costs_of_jeans = df_data.loc["Jacket"]

print("The costs of Jackets is", df_costs_of_jeans)
```

The costs of Jackets is 189

4. Now we are going to plot the data on a stack bar plot using Matplotlib. This will create a stacked bar plot with three bars for each product: one for sales, one for costs, and one for profit. We are looking for the least profitable products, so we are going to sort the DataFrame by profit in ascending order and return the 6 least profitable products using slicing.

```
[13]: import matplotlib.pyplot as plt

df_least_5_products = df.sort_values(by = "Profit", ascending=True)[0:6]

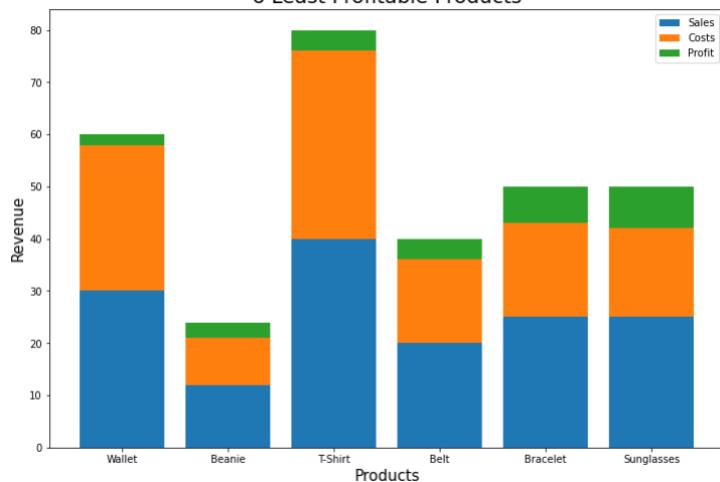
# Create data
products = df_least_5_products["Product Name"]
sales = df_least_5_products["Revenue"]
costs = df_least_5_products["Costs"]
profit = df_least_5_products["Profit"]

# Create figure and axes
fig, ax = plt.subplots(figsize=(12, 8))

# Plotting the data
ax.bar(products, sales, label="Sales")
ax.bar(products, costs, label="Costs", bottom = sales)
ax.bar(products, profit, label="Profit", bottom = sales + costs)

ax.set_ylabel("Revenue", fontsize = 15)
ax.set_xlabel("Products", fontsize = 15)
plt.xticks(rotation = 0)
ax.legend()
plt.title("6 Least Profitable Products", fontsize = 20)
plt.show()
```

6 Least Profitable Products



Day 27: Retail Data Processing and Analysis – Part 3

Learning `.loc` and `.iloc` attributes in pandas is important for data analysis because they provide a powerful and flexible way to select and manipulate data in a DataFrame. They are essential tools for any data analyst working with pandas. The challenges below require that you use these attributes in your analysis. You will continue to work with the `retail_shop_data` CSV file.

	A	B	C	D	E	F	G	H
1	Date	Customer Product ID	Product Name	Cost	Price	Quantity	Total	
2	1 01 2023	1001	101 T-Shirt	18	20	2	40	
3	1 01 2023	1002	102 Jeans	35	50	1	50	
4	2 01 2023	1003	103 Hoodie	25	30	3	90	
5	3 01 2023	1004	104 Sneakers	55	70	2	140	
6	3 01 2023	1005	105 Sunglasses	17	25	1	25	
7	4 01 2023	1006	106 Backpack	40	45	2	90	

1. Using `.loc` attribute, calculate the profit of the sunglasses.
2. Using `.loc` attribute, calculate the profit of hoodies.
3. Using `.iloc` attribute, return the names of the two least profitable products.
4. Using the `.loc` attribute, return a DataFrame subset of the most profitable product. Retrieve the product name and ID from the subset.
5. Using the Seaborn library, create a scatter plot to visualize the relationship between sales and costs for each product. Is there any noticeable correlation?

Day 27 - Answers

1. Now, "loc" is short for location. To calculate the profit of the sunglasses using "loc," we can use the column "Product Name" to locate the item, sunglasses. The `.loc` attribute will find the product values in the "Revenue" and "Costs" columns. The difference between the two is the profit of the sunglasses.

```
[14]: # Find the batteries sales value
sales_sunglasses = df.loc[df["Product Name"]=="Sunglasses", "Revenue"]
# Find the batteries costs values
costs_sunglasses = df.loc[df["Product Name"]=="Sunglasses", "Costs"]

# Calculate the difference to find profit.
profit_of_sunglasses = sales_sunglasses - costs_sunglasses
print(f"Profit on Sunglasses is {profit_of_sunglasses.iloc[0]} dollars")
```

Profit on Sunglasses is 8 dollars

2. With the `.loc` attribute, we access the product using the product name in the "Product Name" column. Once the `.loc` attribute locates the product name in the column, it will locate the profit of the "Hoodie" in the "Profit" column.

```
[15]: # Locating the profit of hoodies
profit_hoodies = df.loc[df["Product Name"] == "Hoodie", "Profit"]
print("The profit of hoodies is:", profit_hoodies.iloc[0])
```

The profit of hoodies is: 15

3. For this challenge, first we will sort the DataFrame by profit value and return the two least profitable products using the `.iloc` attribute. Since the `sort()` method sorts the data in ascending order by default, the least profitable products will be on top. So the argument to the `.iloc` attribute `[:2, 3]` means we want the first two rows (`:2`) of column number 4 (index 3).

```
[16]: # Sort products by profit in ascending order
df.sort_values("Profit", inplace=True)

# Use iloc to find 2 least profitable products.
least_profitable_products = df.iloc[:2,3]
least_profitable_products
```

```
[16]: 16    Wallet
12    Beanie
Name: Product Name, dtype: object
```

4. To find the most profitable product using the `.loc` attribute, we can locate the product by filtering the "Profit" column. First, we will return a Dataframe of the product that has the maximum profit in that column.

```
[17]: df_most_profit = df.loc[df["Profit"]==max(df["Profit"])]
df_most_profit
```

	Date	Customer ID	Product ID	Product Name	Cost Per Product	Price Per Product
13	9 01 2023	1008	108	Jacket	63	80

Now we can retrieve the product name and ID:

```
[18]: df_most_profit = df.loc[df["Profit"]==max(df["Profit"])]
# Getting product name and ID
df_most_profit[['Product ID', 'Product Name']]
```

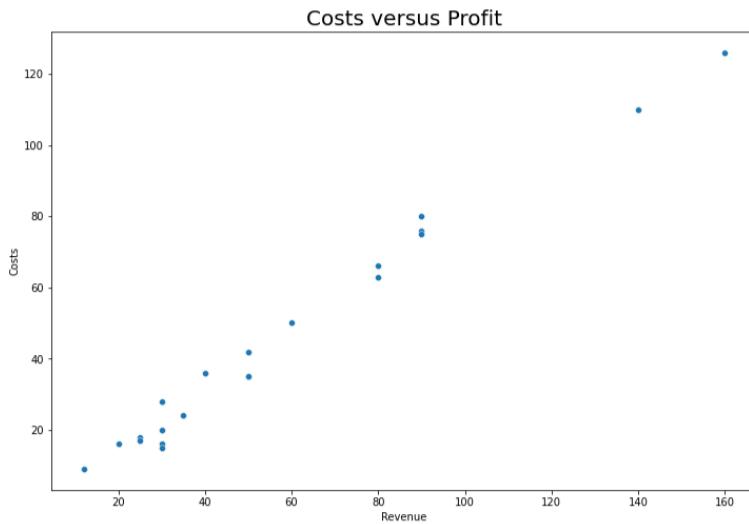
	Product ID	Product Name
13	108	Jacket

You can see that the "Jacket" is the most profitable product.

5. Now let's create a scatter plot using Seaborn. We will import Seaborn as `sns` as per convention.

```
[19]: import seaborn as sns

fig, ax = plt.subplots(figsize=(12, 8))
sns.scatterplot(x=df["Revenue"], y=df["Costs"], data=df)
plt.title('Costs versus Profit', fontsize=20)
plt.show()
```



You can see that the higher the sales, the higher the costs of the products. This is evidence of a correlation between revenue and costs.

Day 28: Population Data Analysis

For the challenges below, you will import and analyze the **countries_population_data** CSV file.

A	B	C	D	E
1	Country	Population	GDP_per_	Unemployment_rate
2	United States	331002651	65298	6.10%
3	China	1439323776	16708	3.80%
4	Japan	126476461	39058	2.80%
5	Germany	83783942	55803	3.20%
6	United Kingdom	67886011	44177	4.80%

1. Using pandas, import the CSV file of the country's data. Import the warnings module and use it to "ignore" the warnings.
2. Create a copy of the DataFrame. More information has become available. You have a CSV file that has the capital cities. Import the file called **countries_data_capital_cities**. Using the `merge()` method from pandas, add a capital city column to your DataFrame.
3. Using Matplotlib, create a bar plot to visualize the size of the population among the countries. The plot data must be in descending order.
4. Using NumPy, calculate the correlation between the population and GDP per capita for each country and create a scatter plot to visualize the relationship between the two variables (population and GDP per capita) using Matplotlib.
5. Using pandas, return a subset of the top three countries with the lowest GDP. Assign this to a variable.
6. Compare the unemployment rates of Switzerland, China, and the USA by plotting a bar plot of this data using Matplotlib.

Day 28 - Answers

1. The first thing is to import the "countries_population_data" dataset using `pd.read_csv()`. We will use `head()` to view the first rows. Pandas provides a warning system that alerts a user about potential problems or deprecated functionality that may affect the code. These warnings are designed to help identify potential issues and take appropriate action. By importing the warnings module and passing "ignore" to the `filterwarnings()` method, we are telling pandas not to display these messages.

```
[1]: import pandas as pd
import warnings
warnings.filterwarnings('ignore')

df = pd.read_csv("countries_population_data.csv")
df.head()
```

	Country	Population	GDP_per_capita	Unemployment_rate	Area
0	United States	331002651	65298	6.10%	9629091
1	China	1439323776	16708	3.80%	9640011
2	Japan	126476461	39058	2.80%	377975
3	Germany	83783942	55803	3.20%	357022
4	United Kingdom	67886011	44177	4.80%	242500

2. First, we will create copy of the original DataFrame.

```
[2]: # Creating a copy of the DataFrame
df_copy = df.copy()
```

Now, let's import the other dataset with capital city information and view the first 5 rows.

```
[3]: # Reading capital cities data
df2 = pd.read_csv("Countries_data_capital_cities.csv")
df2.head()
```

```
[3]:
```

	Country	Capital_City
0	United States	Washington, D.C.
1	China	Beijing
2	Japan	Tokyo
3	Germany	Berlin
4	United Kingdom	London

Now, we are going to merge the two DataFrames on the "Country" column. We use the "Country" column because this is the column that the two DataFrames have in common. We will use the `merge()` method.

```
[4]: # Using merge to merge DataFrame on 'Country'
merged_df = df_copy.merge(df2, on='Country')
merged_df.head()
```

```
[4]:
```

	Country	Population	GDP_per_capita	Unemployment_rate	Area	Capital_City
0	United States	331002651	65298	6.10%	9629091	Washington, D.C.
1	China	1439323776	16708	3.80%	9640011	Beijing
2	Japan	126476461	39058	2.80%	377975	Tokyo
3	Germany	83783942	55803	3.20%	357022	Berlin
4	United Kingdom	67886011	44177	4.80%	242500	London

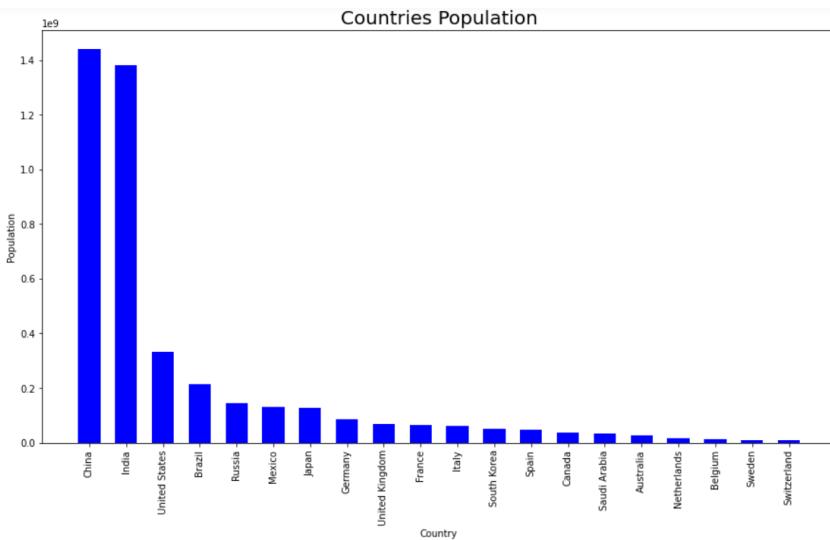
3. Plotting the distribution on a bar plot will make it easy to visualize the population of each country. We are going to use a bar plot from Matplotlib to create a plot. We are going to use the `sort_values()` method to sort the data in descending order.

```
[5]: import matplotlib.pyplot as plt

merged_df.sort_values(by = ["Population"], axis=0,
                      inplace=True,
                      ascending=False)
fig = plt.figure(figsize = (15, 8))
plt.bar(merged_df.Country, merged_df.Population,
        color = 'blue',
        width = 0.6)

plt.xlabel("Country")
plt.ylabel("Population")
plt.xticks(rotation = 90)
plt.title("Countries Population", fontsize = 20)
plt.show()
```

This will output a bar plot. We can easily see the countries with the biggest and smallest populations.



4. In this challenge, we are required to plot a scatter plot that visualizes the relationship between the population and the GDP. But first, we are required to calculate the correlation between the population and the GDP per capita for each country using NumPy. We are going to use the

`numpy.corrcoef()` function to calculate the correlation. We will pass the population and GDP as arguments, and it will calculate the correlation. We will use these results to plot a scatter plot.

```
[6]: import numpy as np

population = merged_df["Population"]
gdp_per_capita = merged_df["GDP_per_capita"]
correlation = np.corrcoef(population, gdp_per_capita)
correlation
```



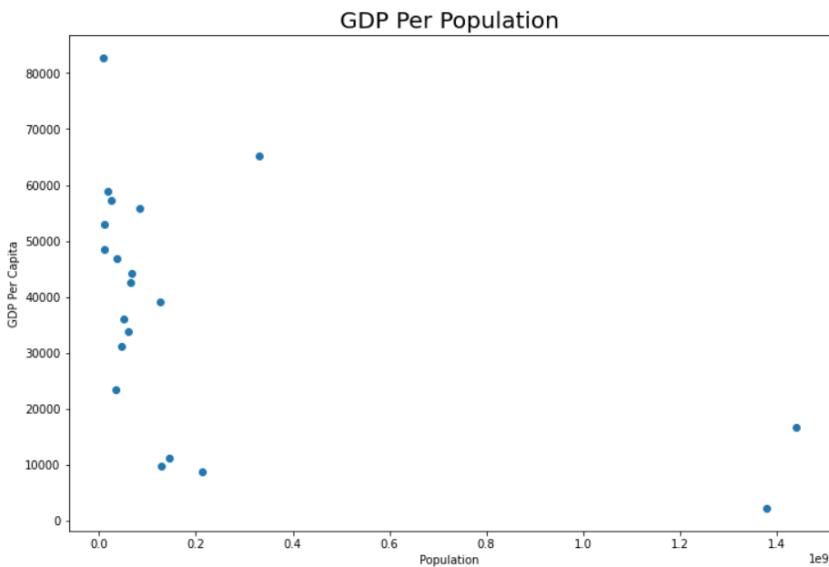
```
[6]: array([[ 1.          , -0.49159759],
           [-0.49159759,  1.          ]])
```

Now let's use this to plot a scatter plot.

```
[7]: import matplotlib.pyplot as plt

fig = plt.figure(figsize = (12, 8))
plt.scatter(population, gdp_per_capita)
plt.xlabel('Population')
plt.ylabel('GDP Per Capita')
plt.title("GDP Per Population", fontsize= 20)
plt.show()
```

You can see from the output that countries with small populations in the DataFrame have a higher GDP per capita.



- To return the top three countries with the lowest GDP, we need to sort the DataFrame in ascending order and use the `head()` method to return the 3 countries with the lowest GDP.

```
[8]: df_low_gdp_3 = merged_df.sort_values(by='GDP_per_capita', ascending=True,
                                         inplace=False).head(3)
df_low_gdp_3
```

	Country	Population	GDP_per_capita	Unemployment_rate	Area	Capital_City
16	India	1380004385	2252	6.10%	3287263	New Delhi
17	Brazil	212559417	8711	13.90%	8515767	Brasilia
10	Mexico	128932753	9877	3.60%	1964375	Mexico City

- First, we will use the `query()` method to filter the DataFrame. We will create a DataFrame for three countries: Switzerland, China, and the United States.

```
[9]: # Filtering the DataFrame for three countries
countries = ["Switzerland", "China", "United States"]

filtered_df = merged_df.query("Country==@countries" )
filtered_df
```

```
[9]:
```

	Country	Population	GDP_per_capita	Unemployment_rate	Area	Capital_City
1	China	1439323776	16708	3.80%	9640011	Beijing
0	United States	331002651	65298	6.10%	9629091	Washington, D.C.
13	Switzerland	8654622	82792	2.50%	41285	Bern

To plot the data, we will have to clean up the "**Unemployment_rate**" column. We will remove the "%" from the column and convert the column into a float data type.

```
[10]: # cleaning the unemployment_rate column
filtered_df["Unemployment_rate"] = filtered_df["Unemployment_rate"].str.replace("%", "").astype(float)
filtered_df
```

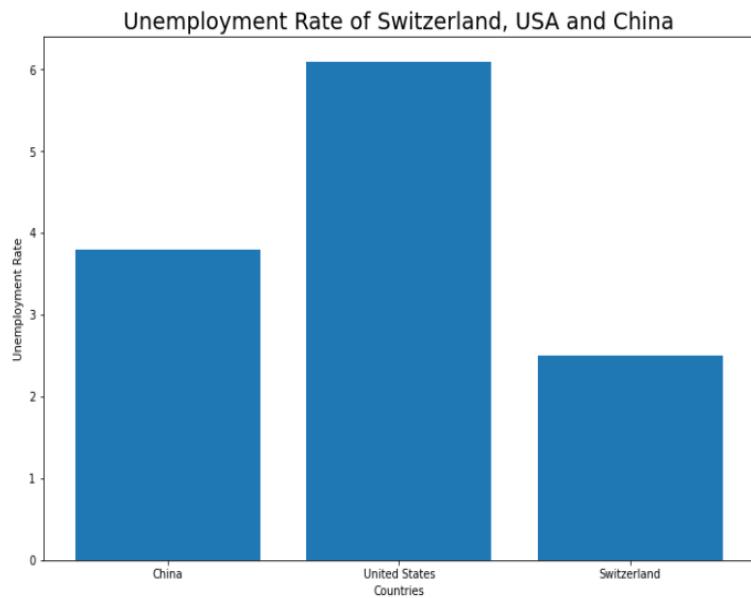
```
[10]:
```

	Country	Population	GDP_per_capita	Unemployment_rate	Area	Capital_City
1	China	1439323776	16708	3.8	9640011	Beijing
0	United States	331002651	65298	6.1	9629091	Washington, D.C.
13	Switzerland	8654622	82792	2.5	41285	Bern

You can see that the "**Unemployment_rate**" column has been cleaned.

Now, let's create a bar plot of the unemployment rate in Switzerland, the USA, and China using Matplotlib.

```
[11]: plt.figure(figsize=(12, 8))
plt.bar(filtered_df.Country, filtered_df.Unemployment_rate)
plt.xlabel("Countries")
plt.ylabel("Unemployment Rate")
plt.title("Unemployment Rate of Switzerland, USA and China", fontsize=20)
plt.show()
```



Day 29: Car Service Data Analysis

For this challenge, you are going to analyze the data of a car service business. You will import the `car_service_data` CSV file.

	A	B	C	D	E	F	G
1	Service ID	Service Type	Service Cost	Service Revenue	Number of Customers	Advertising Cost	Location
2	1	Oil Change	\$25	\$50	10	\$100	Urban
3	2	Brake Repair	\$200	\$500	5	\$50	Suburban
4	3	Tire Rotation	\$30	\$75	12	\$150	Rural
5	4	Transmission Repair	\$1000	\$2500	2	\$200	Urban

1. Import the `car_service_data` CSV file. Write a code to return all the column names in the DataFrame. Check the data types of the DataFrame.
2. Which location has the highest service costs? Plot a bar plot using Seaborn to visualize the service costs by location.
3. Which location has the highest number of customers? Using Matplotlib, plot a pie chart of the location and number of customers. Present the number of customers as percentages. Apply the `explode` parameter to the location with the highest number of customers. Add a shadow to your plot.
4. Using the pandas `groupby()` method, group the data by location and find the average profit margin per location. Format your output to 1 decimal place and present it as a percentage.
5. What is the advertising cost of each region as a percentage of total revenue? Using the `rank()` method, rank each region by the cost of advertising as a percentage of revenue in descending order (add a rank column to the grouped data). Which region has the lowest rank?

Day 29 - Answers

1. We are going to import the "car_service_data" dataset using `pd.read_csv()`. We will use the `head()` method to view the first five rows of the DataFrame.

```
[1]: import pandas as pd  
  
df = pd.read_csv("car_service_data.csv")  
df.head()
```

	Service ID	Service Type	Service Cost	Service Revenue	Number of Customers	Advertising Cost	Location
0	1	Oil Change	\$25	\$50	10	\$100	Urban
1	2	Brake Repair	\$200	\$500	5	\$50	Suburban
2	3	Tire Rotation	\$30	\$75	12	\$150	Rural
3	4	Transmission Repair	\$1000	\$2500	2	\$200	Urban
4	5	Battery Replacement	\$150	\$300	8	\$75	Suburban

To return all the columns in the DataFrame, we are going to use the `df.columns` attribute.

```
[2]: df.columns  
  
[2]: Index(['Service ID', 'Service Type', 'Service Cost', 'Service Revenue',  
           'Number of Customers', 'Advertising Cost', 'Location'],  
           dtype='object')
```

Let's check the data types in our DataFrame.

```
[3]: df.dtypes  
  
[3]: Service ID          int64  
      Service Type        object  
      Service Cost         object  
      Service Revenue      object  
      Number of Customers  int64  
      Advertising Cost     object  
      Location             object  
      dtype: object
```

2. The first thing that we are going to do is clean up the numeric columns: "Service Revenue," "Service Cost," and "Advertising Cost." The data type of these columns is "object." We cannot use this data type to create plots. We first have to remove the "\$" and convert it to a numeric data type. Below, we first create a copy of the DataFrame and use a function to clean up the columns.

```
[4]: # Creating a copy of the DataFrame
df2 = df.copy()

# Creating a function to clean the data
def modify_df(df):
    df["Service Revenue"] = df["Service Revenue"].str.replace("$", "").astype(int)
    df["Service Cost"] = df["Service Cost"].str.replace("$", "").astype(int)
    df["Advertising Cost"] = df["Advertising Cost"].str.replace("$", "").astype(int)
    return df

modify_df(df2)

df2.head()
```

	Service ID	Service Type	Service Cost	Service Revenue	Number of Customers	Advertising Cost	Location
0	1	Oil Change	25	50	10	100	Urban
1	2	Brake Repair	200	500	5	50	Suburban
2	3	Tire Rotation	30	75	12	150	Rural
3	4	Transmission Repair	1000	2500	2	200	Urban
4	5	Battery Replacement	150	300	8	75	Suburban

Let's group the data by location.

```
[5]: # Group by Location
group_by_location = df2.groupby("Location").sum()

# Sort data by Location in ascending order
group_by_location.sort_values(by="Service Cost", ascending=False)
```

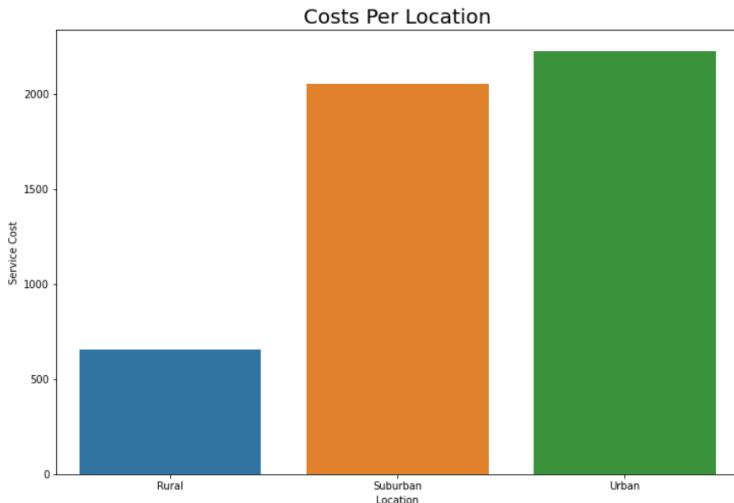
	Service ID	Service Type	Service Cost	Service Revenue	Number of Customers	Advertising Cost
Location						
Urban	70	Oil Change Transmission Repair AC Recharge Detail...	2225	5350	42	750
Suburban	77	Brake Repair Battery Replacement Suspension Repa...	2050	4950	67	625
Rural	63	Tire Rotation Engine Diagnostic Radiator Flush Fu...	655	1650	72	575

We will use the Seaborn library to plot a bar plot.

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

fig, ax = plt.subplots(figsize=(12, 8))
sns.barplot(x=group_by_location.index,
            y=group_by_location["Service Cost"],
            data=group_by_location)
ax.set_title('Costs Per Location', fontsize=20)
plt.show()
```

You can see below that the cost of location increases as you go from rural to urban areas.



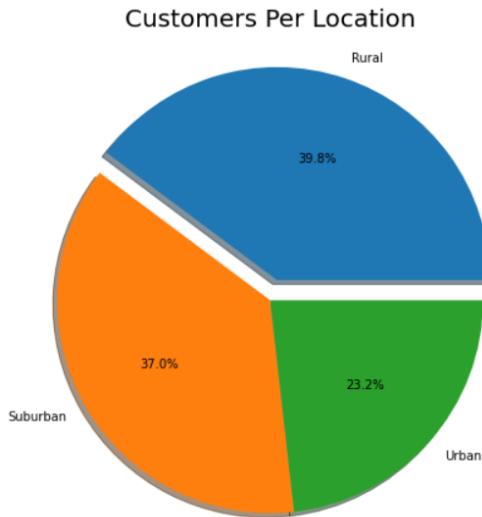
3. We are going to use the pie plot function from Matplotlib to visualize customer size per location. First, we will group the number of customers by location using the `groupby()` method. We will present the population size as percentages. Here is the full code below:

```
[7]: # Grouping the data by Location
df_group = df2.groupby("Location")["Number of Customers"].sum()

plt.figure(figsize=(12, 8))
plt.pie(df_group, explode=(0.1, 0, 0),
        labels=df_group.index,
        autopct='%1.1f%%',
        shadow=True,
        )

plt.title("Customers Per Location", fontsize=20)
plt.show()
```

You can see below that the largest group of customers is in rural areas, and the urban area has the smallest group of customers.



4. First, we are going to add a "Profit" column to the DataFrame. We will use the "profit" column to calculate the profit margin by dividing the "Profit" column by the "Service Revenue" column. The "Profit Margin" column will be added to the DataFrame. We will then use the `groupby()` method to group the DataFrame by the "Location" column and calculate the mean of the "Profit Margin" column. We use the `map()` method to format the output to 1 decimal point.

```
[8]: # calculate the profit
df2['Profit'] = df2['Service Revenue'] - df2['Service Cost']

# Calculate the profit margin
df2['Profit Margin'] = df2['Profit'] / df2['Service Revenue']*100

# Grouping data by products and calculating mean
df2.groupby('Location')['Profit Margin'].mean().map('{:.1f}%'.format)

[8]: Location
      Rural      60.4%
      Suburban    55.8%
      Urban       56.5%
Name: Profit Margin, dtype: object
```

5. In the code below, we use the `groupby()` method to group the data by location and sum the "Advertising Cost" and "Service Revenue" columns. We divide the advertising cost by the revenue to get the cost of advertising as a percentage of revenue.

```
[9]: # Using groupby to group by location and sum advertising and revenue columns
df_group2 = df2.groupby(["Location"])[["Advertising Cost",'Service Revenue']].sum()

# Calculating the advertising cost as percentage of revenue
(df_group2["Advertising Cost"]/df_group2["Service Revenue"] * 100).map('{:.1f}%'.format)

[9]: Location
      Rural      34.8%
      Suburban   12.6%
      Urban      14.0%
      dtype: object
```

To add the "rank" column, we are going to group the data by location and the cost of advertising as a percentage of revenue and apply the `rank()` method. We will apply the rank value in descending order. This means that the location with the highest advertising cost as a percentage of revenue will have the lowest rank. Here is the code below:

```
[10]: df_group2["rank"] = (df_group2["Advertising Cost"]/df_group2
                           ["Service Revenue"]).rank(ascending=False)
df_group2
```

	Advertising Cost	Service Revenue	rank
Location			
Rural	575	1650	1.0
Suburban	625	4950	3.0
Urban	750	5350	2.0

The rural region has the lowest rank because it has the highest cost of advertising as a percentage of its revenue.

Day 30: Furniture Data Analysis

For this challenge, you are going to use the **furniture_data** CSV file. You will clean the data and create visualizations.

	A	B	C	D	E	F
1	Product	Sale Price	Cost per P	Quantity	Total Revenue	
2	Sofa	800	600	32	25600	
3	Chair	300	200	12	3600	
4	Table	500	300	41	20500	
5	Bed	1000	700	23	23000	

1. Import the **furniture_data** CSV file using pandas. Check the length of the DataFrame. Find the sum of duplicates in the "Product" column. Using the pandas **pivot_table()** function, return a table that shows how many times each product appears in the "Product" column. This table will reveal which products are duplicated.
2. Use Seaborn **regplot()** to fit a linear regression model and visualize the relationship between the price of each item and total revenue. What conclusion can you draw from the plot?
3. What is the difference in revenue between "wardrobes" and "beds"?
4. Calculate the "Total_Cost" column and insert it in the DataFrame between the "Quantity" and "Total Revenue." columns. Use the pandas **insert()** function.
5. Which product has the least profit margin?

Day 30 - Answers

- First, we are going to import the "furniture_data" dataset using pandas `read_csv()`. We will use the `head()` method to check if the data has loaded properly.

```
[1]: import pandas as pd  
  
df = pd.read_csv("furniture_data.csv")  
df.head()  
  
[1]:   Product Sale Price Cost per Product Quantity Total Revenue  
0     Sofa      800        600       32    25600  
1   Chair      300        200       12     3600  
2   Table      500        300       41    20500  
3     Bed     1000        700       23    23000  
4 Wardrobe     800        600       19    15200
```

Now, let's check the length of the DataFrame.

```
[2]: # Finding Length of the DataFrame  
len(df)
```

```
[2]: 16
```

You can see that our DataFrame has 16 rows. Now, let's check for duplicates and sum them up.

```
[3]: df.Product.duplicated().sum()
```

```
[3]: 6
```

We have 6 duplicates in our dataset. Now, we are going to use the `pivot_table()` method on the "product" column to see the duplicated products. We will set the "Product" column as the index and set the `aggfunc` parameter to size.

```
[4]: # Counting duplicates in the product column
df_duplicates = df.pivot_table(index = ['Product'], aggfunc ='size')
print(df_duplicates)

Product
Bed           1
Bookshelf      2
Chair          1
Coffee Table   2
Dresser         2
Ottoman         2
Rug            2
Sofa            1
Table           1
Wardrobe        2
dtype: int64
```

You can see the 6 products that are duplicated.

2. Now let's plot a linear regression plot to visualize the relationship between the sale price of each item and the total revenue. We will use **regplot** from the Seaborn library.

```
[5]: import matplotlib.pyplot as plt
import seaborn as sns

fig = plt.figure(figsize=(15, 10))
sns.regplot(x=df["Sale Price"],
            y=df["Total Revenue"],
            data=df)
plt.title("Relationship Between Sale Price and Total Revenue", fontsize= 20)
plt.show()
```



Based on the plot above, there appears to be a positive correlation between sale price and total revenue. As the sale price increases, the total revenue tends to increase as well. This is indicated by the upward slope of the regression line. However the scatter around the line indicates that other factors besides sale price may also influence total revenue.

3. To calculate the difference between the wardrobes and beds, we can use the `groupby()` method to group data by products and sum the "Total Revenue" column. We have to group the products because they are duplicated in the column. The difference will be the total revenue of the wardrobes less the total revenue of the beds.

```
[6]: # Grouping data by product and summing the total revenue column
df_group = df.groupby("Product")["Total Revenue"].sum()

# Calculating the difference
wardrobes_less_bed_revenue = df_group.Wardrobe - df_group.Bed
print(wardrobes_less_bed_revenue)
```

2600

Another way to go about it is to use the `.loc` attribute. We will access each product using the `.loc` attribute and sum it up using the `sum()` method.

```
[7]: # using loc to locate wardrobe and bed
wardrobes_revenue = df.loc[df["Product"] == "Wardrobe", "Total Revenue"].sum()
beds_revenue = df.loc[df["Product"] == "Bed", "Total Revenue"].sum()
print(wardrobes_revenue - beds_revenue)
```

2600

You can see that both methods return 2600.

4. First, we will calculate the total cost by multiplying the "Cost per Product" column by the "Quantity" column. We will then use the `insert()` method to insert the column at index 4. Index 4 is between the "Quantity" column and the "Total Revenue" column. You can see in the output that a new column has been added.

```
[8]: # Calculating the total costs
Total_Costs = df["Cost per Product"] * df["Quantity"]

# Using insert to insert column on index 4
df.insert(4, 'Total_Costs', Total_Costs)
df.head()
```

	Product	Sale Price	Cost per Product	Quantity	Total_Costs	Total Revenue
0	Sofa	800	600	32	19200	25600
1	Chair	300	200	12	2400	3600
2	Table	500	300	41	12300	20500
3	Bed	1000	700	23	16100	23000
4	Wardrobe	800	600	19	11400	15200

5. To find the product with the least profit margin, first we will calculate the profit of each product and add a "Profit" column to the DataFrame. We will use the `groupby()` method to group the data by products and sum the "Total Revenue" and "Profit" columns. We then calculate the profit margin by dividing profit by revenue. The `sort_values()` method will be used to sort the DataFrame in ascending order and return the product in the first row.

```
[9]: # Calculating the profit for each product
df["Profit"] = df["Total Revenue"] - df["Total_Costs"]

# Grouping data by product and summing revenue and profit
df_group = df.groupby("Product")[["Total Revenue","Profit"]].sum()

# Calculating profit margin for each product
df_group["Profit_Margin"] = (df_group["Profit"]/df_group["Total Revenue"])*100

# Sorting data by profit margin and returning the first product
df_group.sort_values(by="Profit_Margin", ascending=True).head(1)
```

Product	Total Revenue	Profit	Profit_Margin
Dresser	53400	8900	16.666667

You can see that Dresser was the least profitable product.

Day 31: Analyze Database Data with SQL

In this challenge, you are going to import the **employees_record_data** dataset, which is a CSV file, and save it to a database. You will then analyze the data using pandas SQL commands. You will use Sqlite3.

	A	B	C	D	E
1	Name	Job Title	Department	Salary	HireDate
2	John Smith	Manager	Management	100000	1 01 2020
3	Jane Doe	Sales Associate	Sales	50000	1 06 2019
4	Bob Johnson	Assistant Manager	Management	75000	1 03 2018
5	Mary Johnson	Sales Associate	Sales	45000	1 02 2021
6	Kevin Lee	Analyst	Finance	80000	1 07 2020
7	Sophia Martinez	Technician	Customer Service	71000	12 09 2018

1. Import the **employees_record_data** dataset and save it as a table in SQL. Write code to fetch the table, first using `fetchall()` and then `read_sql_query()`.
2. Using pandas `sql_query`, what are the names and salaries of employees that make over \$60,000.
3. Write code to return the average salary for each department. Group it by department.
4. Which department has the highest-paid employee? What is their name?
5. Which employees were hired before 2020-01-01?

Day 31 - Answers

1. First, we are going to import the dataset using pandas, and view the first 5 rows.

```
[1]: import pandas as pd

df = pd.read_csv("employees_record_data.csv")
df.head()
```

	Name	Job Title	Department	Salary	HireDate
0	John Smith	Manager	Management	100000	1 01 2020
1	Jane Doe	Sales Associate	Sales	50000	1 06 2019
2	Bob Johnson	Assistant Manager	Management	75000	1 03 2018
3	Mary Johnson	Sales Associate	Sales	45000	1 02 2021
4	Kevin Lee	Analyst	Finance	80000	1 07 2020

To create a database from this DataFrame, we are going to use the `df.to_sql()` method. First, we will import the `sqlite3` library, set up a connection to sqlite3, and create a database. We will save this DataFrame as a table in this database. The database is "Employees_Records.db," and the name of the table is "MyEmployeesTable."

```
[2]: import sqlite3

# Connect to SQL and create a database
engine = sqlite3.connect('Employees_Records.db')
cur = engine.cursor()

# Save to SQL table
try:
    df.to_sql('MyEmployeesTable', con=engine, index = False)
except ValueError:
    print("Table already exists")
```

We can check if the saved table is in the database using the SELECT query. This query will be passed as an argument to the `execute()` method. By using the asterisk (*) operator, we can select all columns from the "MyEmployeesTable." Here is the code below:

```
[3]: df_data = cur.execute("""SELECT * FROM MyEmployeesTable""")
df_data.fetchall()

[3]: [('John Smith', 'Manager', 'Management', 100000, '1 01 2020'),
 ('Jane Doe', 'Sales Associate', 'Sales', 50000, '1 06 2019'),
 ('Bob Johnson', 'Assistant Manager', 'Management', 75000, '1 03 2018'),
 ('Mary Johnson', 'Sales Associate', 'Sales', 45000, '1 02 2021'),
 ('Kevin Lee', 'Analyst', 'Finance', 80000, '1 07 2020'),
 ('Sophia Martinez', 'Technician', 'Customer Service', 71000, '12 09 2018'),
 ('Emma Thomas', 'Assistant', 'Research', 59000, '30 11 2019'),
 ('Liam Taylor', 'Consultant', 'Finance', 72000, '22 07 2015'),
 ('Ava White', 'Specialist', 'Engineering', 67000, '18 09 2020'),
 ('Noah Martin', 'Coordinator', 'HR', 55000, '3 03 2019'),
 ('Isabella Hall', 'Officer', 'IT', 78000, '29 05 2018'),
 ('Ethan Garcia', 'Programmer', 'Marketing', 65000, '12 10 2016'),
 ('Mia Lopez', 'Associate', 'Operations', 72000, '8 08 2017'),
 ('Alexander Hill', 'Administrator', 'Sales', 60000, '5 04 2022'),
 ('Charlotte Clark', 'Designer', 'Research', 73000, '2 09 2019'),
 ('Daniel Lewis', 'Assistant', 'IT', 69000, '14 01 2018'),
 ('Harper Adams', 'Manager', 'Marketing', 85000, '7 06 2016')]
```

To display the table, we can also use the `pd.read_sql_query()` function. We pass a query "SELECT * FROM MyEmployeesTable" to the function, which selects all columns from the table. The result is stored in a DataFrame called "data_table". See the code below:

```
*[4]: data_table = pd.read_sql_query("""
    SELECT * FROM MyEmployeesTable
    """", con=engine)
data_table.head()
```

	Name	Job Title	Department	Salary	HireDate
0	John Smith	Manager	Management	100000	1 01 2020
1	Jane Doe	Sales Associate	Sales	50000	1 06 2019
2	Bob Johnson	Assistant Manager	Management	75000	1 03 2018
3	Mary Johnson	Sales Associate	Sales	45000	1 02 2021
4	Kevin Lee	Analyst	Finance	80000	1 07 2020

2. To find the employees who make over \$60k, we are going to use the SELECT query and select the "Name" and "Salary" columns. We will filter the "Salary" column using the WHERE query to return the names and salaries of those that get over \$60,000. We will pass this SQL query to the `pd.read_sql_query()` function. Here is the code below:

```
[5]: salary_over_60k = pd.read_sql_query("""
    SELECT Name, Salary
    FROM MyEmployeesTable
    WHERE Salary > 60000
""", con=engine)
salary_over_60k
```

```
[5]:      Name   Salary
0     John Smith  100000
1     Bob Johnson  75000
2     Kevin Lee  80000
3 Sophia Martinez  71000
4     Liam Taylor  72000
5     Ava White  67000
6 Isabella Hall  78000
7     Ethan Garcia  65000
8     Mia Lopez  72000
9 Charlotte Clark  73000
10    Daniel Lewis  69000
11    Harper Adams  85000
```

3. We will use the SELECT query to select the "Department" and "Salary" columns. We will calculate the average salary for each department using the `AVG()` function. We will group the results by the "Department" column. We will name the column that returns the average salary "Average_salary."

```
[6]: ave_salary = pd.read_sql_query("""
    SELECT Department, AVG(Salary) AS Average_salary
    FROM MyEmployeesTable
    GROUP BY Department
    """, con=engine)

ave_salary
```

```
[6]:      Department  Average_salary
0  Customer Service   71000.000000
1      Engineering   67000.000000
2        Finance   76000.000000
3          HR   55000.000000
4          IT   73500.000000
5     Management   87500.000000
6     Marketing   75000.000000
7     Operations   72000.000000
8       Research   66000.000000
9         Sales   51666.666667
```

4. To find the highest-paid employee and retrieve their name, department, and salary, we SELECT the "Name," "Department," and "Salary" columns from the "MyEmployeesTable". We will use the MAX function on the "Salary" column to return the row with the maximum value. This will ensure that we only retrieve the details of the highest-paid employee.

```
[7]: highest_salary = pd.read_sql_query("""
    SELECT Department, Name, MAX(Salary) AS Highest_salary
    FROM MyEmployeesTable
    """, con=engine)

highest_salary
```

```
[7]:      Department      Name  Highest_salary
0     Management  John Smith        100000
```

5. To find the employees that were hired before 2020-01-01, we are going to select the "Names" column and filter the table using the "HireDate" column by selecting dates before 2020-01-01. We will return the names of these employees.

```
[8]: hire_date_before_2020 = pd.read_sql_query("""  
    SELECT Name  
    FROM MyEmployeesTable  
    WHERE HireDate < "2020-01-01"  
""", con=engine)  
  
hire_date_before_2020
```

```
[8]:      Name  
0      John Smith  
1      Jane Doe  
2      Bob Johnson  
3      Mary Johnson  
4      Kevin Lee  
5  Sophia Martinez  
6      Ava White  
7      Ethan Garcia  
8  Charlotte Clark  
9      Daniel Lewis
```

Day 32: Soccer Stricker's Data Analysis

This challenge requires that you carry out some cleaning and preprocessing of the data, such as checking for duplicates, handling missing values, and manipulating `DataFrames`. You will also be required to provide some insights about the data and much more.

	A	B	C	D	E	F	G
1	Player Name	Chances Created	Assists	Goals	Club	Manager	Annual Salary
2	Cristiano Ronaldo	85	20	35	Manchester United	Ole Gunnar Solskjær	\$35,000,000
3	Lionel Messi	90	25	30	Paris Saint-Germain	Mauricio Pochettino	\$40,000,000
4	Robert Lewandowski	70	15	40	Bayern Munich	Julian Nagelsmann	\$25,000,000
5	Erling Haaland	60	10	35	Borussia Dortmund	Marco Rose	\$20,000,000
6	Kylian Mbappé	80	18	25	Paris Saint-Germain	Mauricio Pochettino	\$30,000,000

1. Import the `soccer_strickers` CSV file using pandas. Check for missing data, duplicates and column data types. If any duplicates, drop them.
2. Create a copy of the DataFrame. Using pandas, convert the "**Annual Salary**" column from object data type to integer data type.
3. Which player has the highest goal conversion rate (goals scored as a percentage of chances created)?
4. Create a hierarchical index for your DataFrame. Set the "**Player Name**" and the "**Club**" as index columns. Using the `.loc` attribute, filter the DataFrame to find the annual salary of Romelu Lukaku.
5. Using the pandas `unstack()` method, unstack the DataFrame with a hierarchical index (from question 4). The unstacked level should be the two columns: "Player Name" and "Club." The "Player Name" column is the outer index. Save this as a new variable.
6. How many chances were created by Karim Benzema? Use the unstack DataFrame from question 5.
7. What is the combined salary of Lionel Messi and Kylian Mbappé?

Day 32 - Answers

- First, we import the **soccer_strikers** dataset using `pd.read_csv()`.

```
[1]: import pandas as pd  
  
df = pd.read_csv("soccer_strikers.csv")  
df.head()
```

	Player Name	Chances Created	Assists	Goals	Club	Manager	Annual Salary
0	Cristiano Ronaldo	85	20	35	Manchester United	Ole Gunnar Solskjær	\$35,000,000
1	Lionel Messi	90	25	30	Paris Saint-Germain	Mauricio Pochettino	\$40,000,000
2	Robert Lewandowski	70	15	40	Bayern Munich	Julian Nagelsmann	\$25,000,000
3	Erling Haaland	60	10	35	Borussia Dortmund	Marco Rose	\$20,000,000
4	Kylian Mbappé	80	18	25	Paris Saint-Germain	Mauricio Pochettino	\$30,000,000

Now let's check for duplicates. This code will check if the DataFrame has duplicates. It will return a Boolean value of True if a row is duplicated and False if a row is not duplicated. The `sum()` method will sum all the True values.

```
[2]: # Checking for duplicates  
df.duplicated().sum()
```

```
[2]: 0
```

You can see that our dataset has no duplicates. Now let's check for missing values in each column.

```
[3]: # Check for missing values  
df.isna().sum()
```

```
[3]: Player Name      0  
Chances Created    0  
Assists             0  
Goals               0  
Club                0  
Manager             0  
Annual Salary       0  
dtype: int64
```

You can see that we do not have any missing values in the dataset. We can now check the data types in the dataset.

```
[4]: # Check for data types  
df.dtypes
```

```
[4]: Player Name      object  
Chances Created    int64  
Assists             int64  
Goals               int64  
Club                object  
Manager             object  
Annual Salary       object  
dtype: object
```

- When dealing with data, cleaning it up for data analysis and machine learning is crucial. Sometimes data is presented in such a format that it must be cleaned before it can be used. In this dataset, the "Annual Salary" column is of the object data type. We must remove the dollar sign "\$" and the commas "," from the column. First, we will create a copy of the DataFrame and then we will use a function to clean the column. See the code below:

```
[5]: # Creating a copy of the DataFrame  
df2 = df.copy()  
  
# Creating a function to clean the data  
def clean_df(df):  
    df["Annual Salary"] = df["Annual Salary"].str.replace("$", "")  
    df["Annual Salary"] = df["Annual Salary"].str.replace(",", "").astype(int)  
    return df  
  
clean_df(df2)
```

	Player Name	Chances Created	Assists	Goals	Club	Manager	Annual Salary
0	Cristiano Ronaldo	85	20	35	Manchester United	Ole Gunnar Solskjær	35000000
1	Lionel Messi	90	25	30	Paris Saint-Germain	Mauricio Pochettino	40000000
2	Robert Lewandowski	70	15	40	Bayern Munich	Julian Nagelsmann	25000000
3	Erling Haaland	60	10	35	Borussia Dortmund	Marco Rose	20000000
4	Kylian Mbappé	80	18	25	Paris Saint-Germain	Mauricio Pochettino	30000000

You can see that the "Annual salary" column has been cleaned.

3. To find the striker with the highest chance conversion rate, first we will calculate the chance conversion rate of all the strikers in the dataset by dividing the "Goals" column by the "Chances Create" column. We will create a new column called "Chance Conversion Rate." We will then sort the DataFrame by this added column in descending order using the `sort_values()` method. We will use the `.loc` attribute and the `head()` method to return the name of the player with the highest conversion rate.

```
[6]: # Adding a chance conversion rate column
df2["Chance Conversion Rate"] = df2["Goals"]/ df2["Chances Created"] *100

# Sorting Dataframe by chance conversion rate
sorted_df = df2.sort_values(by="Chance Conversion Rate", ascending=False)

# Get the player with highest conversion rate
player_name = sorted_df.loc[:, "Player Name"].head(1)
# Get the conversion rate
conversion_rate = sorted_df.loc[:, "Chance Conversion Rate"].head(1)

print(f'Player with highest chance conversion rate is {player_name.iloc[0]}')
      f' with a rate of {conversion_rate.iloc[0]:.2f}%)'
```

Player with highest chance conversion rate is Erling Haaland with a rate of 58.33%

You can see that Erling Haaland is the player with the highest chance conversion rate of 58.33%.

4. To create a hierarchical index (MultiIndex), we are going to use the `set_index()` method. We will pass the two columns ("Player Name" and "Club") as arguments to the method. These two will be set as hierarchical indexes for our DataFrame.

```
[7]: # Using Player Name and Club columns as hierarchical index
df2.set_index(['Player Name', 'Club'], inplace=True)
df2
```

	Player Name	Club	Chances Created	Assists	Goals	Manager	Annual Salary	Chance Conversion Rate
	Cristiano Ronaldo	Manchester United	85	20	35	Ole Gunnar Solskjær	35000000	41.176471
	Lionel Messi	Paris Saint-Germain	90	25	30	Mauricio Pochettino	40000000	33.333333
	Robert Lewandowski	Bayern Munich	70	15	40	Julian Nagelsmann	25000000	57.142857
	Erling Haaland	Borussia Dortmund	60	10	35	Marco Rose	20000000	58.333333
	Kylian Mbappé	Paris Saint-Germain	80	18	25	Mauricio Pochettino	30000000	31.250000
	Karim Benzema	Real Madrid	75	15	20	Carlo Ancelotti	28000000	26.666667
	Harry Kane	Tottenham Hotspur	65	20	30	Nuno Espírito	32000000	46.153846

To find the annual salary of Romelu Lukaku, we can filter our DataFrame using the `.loc` attribute. We will use a hierarchical index. Under the "Player Name" column, we will pass the name of the player (Romelu Lukaku), and under the "Club" column, we will pass the club (Chelsea) as an argument. We will add the "Annual Salary" column because we want to return the salary value. Here is the code below:

```
[8]: lukaku_salary = df2.loc[('Romelu Lukaku', 'Chelsea'), 'Annual Salary']
print(" The Annual salary for Lukaku is:",lukaku_salary)
```

```
The Annual salary for Lukaku is: 27000000
```

5. Unstacking essentially transforms a DataFrame from "long" to "wide" format by creating a new column for each unique value in the innermost level of the index. In the code below, the outer column is the "Player Name" column. The inner column is the "Club" column.

```
[9]: # Unstacking the DataFrame on "Player Name" and "Club" columns  
df3 = df2.unstack(level=['Player Name', 'Club'])  
df3.head()
```

```
[9]:          Player Name      Club  
Chances Created Cristiano Ronaldo Manchester United    85  
                  Lionel Messi Paris Saint-Germain    90  
                  Robert Lewandowski Bayern Munich     70  
                  Erling Haaland Borussia Dortmund    60  
                  Kylian Mbappé Paris Saint-Germain   80  
dtype: object
```

6. First, we will sort the index to enhance performance. Since the unstacked DataFrame is multi-indexed, we are going to pass two values to the index. We want to retrieve the number of chances created by Karim Benzima, so we will pass "**Chances created**" and "**Karim Benzema**" as arguments. See below:

```
[10]: # Sorting the index  
df3.sort_index(inplace=True)  
  
# Using multindex to access the value  
df3.loc[("Chances Created", "Karim Benzema")][0]
```

```
[10]: 75
```

You can see from the output that Benzema created 75 chances.

7. To find the combined salary of Messi and Mbappe, we will pass the "**Annual Salary**" column and their respective names as arguments to the `.loc` attribute. This will retrieve the annual salaries of the two players. We will add the two salaries to get the combined value.

```
[11]: # Messi Annual Salary  
messi_salary = df3.loc[("Annual Salary", "Lionel Messi")][0]  
  
# Mbappe Annual salary  
mbappe_salary = df3.loc[("Annual Salary", "Kylian Mbappé")][0]  
  
combined_salary = messi_salary + mbappe_salary  
print(f"The combined total of Messi and Mbappe salary is {combined_salary}")
```

```
The combined total of Messi and Mbappe salary is 70000000
```

Day 33: Website Data Analysis

For this challenge, you will analyze data using the pandas, Matplotlib, and Seaborn libraries. You are going to use the **website_data_analysis** CSV file. Here is a sample of the data:

	A	B	C	D	E	F	G	H
1	website	visits	bounce_rate	conversion_rate	days_of_week	unique_visitors	referral_source	revenue
2	boss.com	5000	20	3	Monday	3000	Google	1000
3	python.co	3500	25	2	Tuesday	2800	Facebook	800
4	wild.com	6000	15	4	Wednesday	3900	Twitter	1200
5	cat.com	4500	30	1.5	Thursday	3500	Instagram	700
6	kit.com	5500	18	3.5	Friday	3700	Yahoo	1000

1. Using pandas, import the **website_data_analysis** dataset and find the average number of visits per website.
2. Add another column to the DataFrame that calculates the number of website visits per unique visitor.
3. Using Matplotlib, create a bar plot to visualize the top 5 websites with the highest number of page views. The data plotted must be sorted in descending order.
4. Using pandas, calculate the average bounce rate for each day of the week. Retrieve the day with the highest average bounce rate and its rate.
5. Using Seaborn, create a line plot to show the trend of unique visitors over time. Group the data by the day of the week.
6. Using pandas, group the data by **"day_of_week"** and **"referral_source"** columns and find the average of the **visits** and **revenue** for each group.
7. Using pandas, calculate the revenue rate for each referral source and create a pie chart to visualize the breakdown of revenue rate by referral source. Which referral source brought it the most revenue?

Day 33 - Answers

1. We are going to import the `website_data_analysis` dataset using pandas:

```
[1]: import pandas as pd

df = pd.read_csv("website_data_analysis.csv")
df.head()

[1]:   website  visits  bounce_rate  conversion_rate  days_of_week  unique_visitors  referral_source  revenue
  0  boss.com    5000        20.0          3.0       Monday            3000      Google        1000
  1  python.com   3500        25.0          2.0      Tuesday            2800  Facebook         800
  2  wild.com    6000        15.0          4.0     Wednesday            3900     Twitter        1200
  3  cat.com     4500        30.0          1.5    Thursday            3500  Instagram         700
  4  kit.com     5500        18.0          3.5      Friday             3700      Yahoo        1000
```

To calculate the average visits per website, we have to use the `groupby()` method to group the data by the "website" column. We will then use the `mean()` method to calculate the average.

```
[2]: df_average_visit_per_site = df.groupby('website')['visits'].mean()
df_average_visit_per_site.head()

[2]: website
Childcare.net      1400.0
CodeCrushers.org   2900.0
CraftyCorner.com   3000.0
FinFit.net         3400.0
FoodieFrenzy.com   3200.0
Name: visits, dtype: float64
```

2. To find the number of visits per unique visitor, we have to divide the number of website visits by the number of unique visitors. We are going to create a new column called `views_per_unique_visitor`.

```
[3]: # Views per unique visitor
df["views_per_unique_visit"] = df["visits"] / df["unique_visitors"]
df.head()
```

	website	visits	bounce_rate	conversion_rate	days_of_week	unique_visitors	referral_source	revenue	views_per_unique_visit
0	boss.com	5800	20	3.0	Monday	3800	Google	1800	1.666667
1	python.com	3500	25	2.0	Tuesday	2800	Facebook	800	1.250000
2	wild.com	6000	15	4.0	Wednesday	3900	Twitter	1200	1.538462
3	cat.com	4500	30	1.5	Thursday	3500	Instagram	700	1.285714
4	kit.com	5500	18	3.5	Friday	3700	Yahoo	1800	1.486486

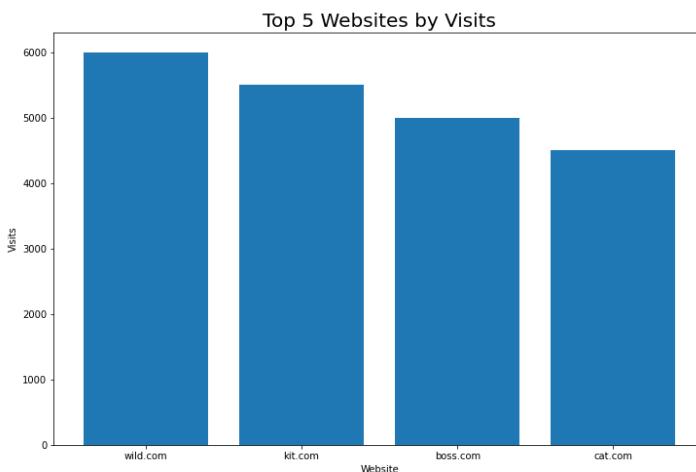
3. To answer this question, we must first sort the DataFrame by the number of visits in descending order using the `sort_values()` method. Then we will use the information to plot a bar graph using Matplotlib.

```
[4]: import matplotlib.pyplot as plt

# Sort the website by visits in descending order
top_5_websites = df.sort_values(by="visits", ascending=False).head(5)

plt.figure(figsize=(12, 8))
plt.bar(top_5_websites["website"], top_5_websites["visits"])
plt.xlabel("Website")
plt.ylabel("Visits")
plt.title("Top 5 Websites by Visits", fontsize=20)
plt.show()
```

This will output the following bar plot.



4. We are going to use the `groupby()` method to group the data by the "days_of_week" column, and we will calculate the **mean** of the "bounce rate" for each day of the week.

```
[5]: max_bounce_rate = df.groupby("days_of_week")["bounce_rate"].mean()
max_bounce_rate
```

```
[5]: days_of_week
Friday      15.666667
Monday      21.333333
Saturday    16.000000
Sunday      18.333333
Thursday    20.666667
Tuesday     18.000000
Wednesday   13.666667
Name: bounce_rate, dtype: float64
```

This gives us the average bounce rate per day of the week. Here is how we can retrieve the day with the highest average rate and its value:

```
[6]: max_bounce_rate = df.groupby("days_of_week")["bounce_rate"].mean().sort_values(
      ascending = False)

# retrieving the max bouncing rate
max_rate = max_bounce_rate.head(1).iloc[0]
# retrieving the day with max bouncing rate
max_day = max_bounce_rate.index[0]
print(f'The day with max bounce rate is {max_day} and rate is {max_rate:.2f}')
```

```
The day with max bounce rate is Monday and rate is 21.33
```

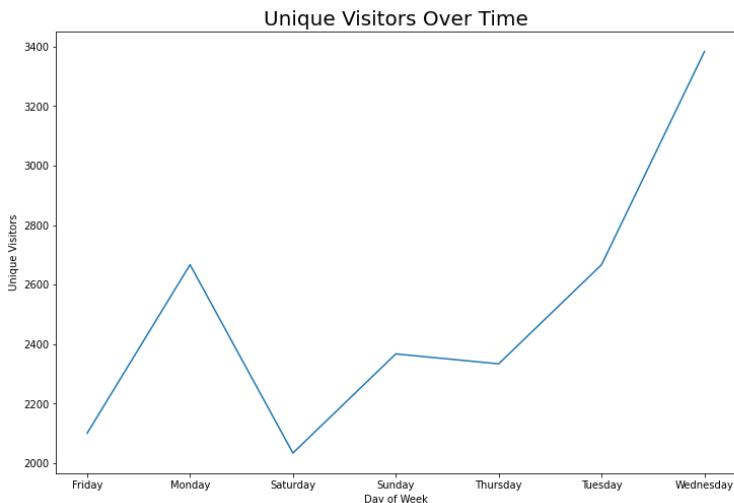
Here, first we sort the grouped data (`max_bounce_rate`) in descending order. This means the day with the highest average bounce rate will appear at the top. Next, we retrieve the first row from the sorted `max_bounce_rate`, which corresponds to the day with the highest bounce rate using the `.iloc` attribute. The `max_bounce_rate.index[0]` retrieves the day of the week that corresponds to the highest bounce rate.

5. We are going to create a line plot using Seaborn. The **x-axis** will be the days of the week, and the **y-axis** will be the unique visitors. But first, we have to group the data by the "days_of_week" column.

```
[7]: import seaborn as sns

# Group data by days of the week
df_group = df.groupby('days_of_week')['unique_visitors'].mean()

# Plot the data
plt.figure(figsize = (12, 8))
sns.lineplot(x = df_group.index, y = df_group)
plt.xlabel("Day of Week")
plt.ylabel("Unique Visitors")
plt.title("Unique Visitors Over Time", fontsize = 20)
plt.show()
```



6. This challenge requires that we group the data using the `groupby()` method. We will group data by the "days_of_week" and "referral source" columns and calculate the mean of the "visits" and "revenue" columns. The result will consist of groups based on unique combinations of each day of the week and the referral source (e.g., Monday-Google, Tuesday-Facebook, etc.). See the code below:

```
[8]: grouped_df = df.groupby(['days_of_week','referral_source'])[['visits','revenue']].mean()
grouped_df.head()
```

		visits	revenue
days_of_week	referral_source		
Friday	Instagram	2100.0	1450.0
	Yahoo	5500.0	1000.0
Monday	Google	5000.0	1000.0
	Instagram	2600.0	700.0
	Reddit	3200.0	900.0

This gives the average visits and revenue for each combination of days of the week and referral source.

- To calculate revenue by referral source, we are going to create a column for the revenue rate. We are going to divide the revenue brought in by each source by the total revenue. We are then going to use the `groupby()` method to group the referral source by revenue rate.

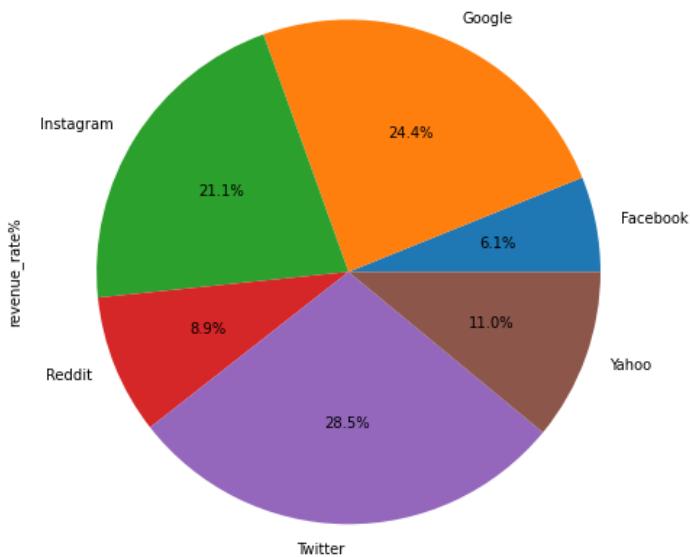
```
[9]: # Creating a new column for revenue per referral source
df["revenue_rate%"] = (df["revenue"]/(sum(df["revenue"])))*100

# Grouping data revenue % by referral source and summing it up
group_data = df.groupby("referral_source")["revenue_rate%"].sum()

# Plotting with pandas and matplotlib
plt.figure(figsize=(12, 8))
group_data.plot.pie(autopct='%.1f%%')
plt.title("Revenue Rate by Referral Source", fontsize = 20)
plt.show()
```

See the resulting plot on the next page:

Revenue Rate by Referral Source



You can see that the majority of revenue (28.5%) came through Twitter referrals.

Day 34: Income Data Analysis

For this challenge, you are going to preprocess and analyze income data. You will import a CSV file called **income_data**. Here is a sample of income data below:

	A	B	C	D	E
1	Name	Age	Height	Gender	Income
2	Alice	25	5.5	F	30000
3	Bob	35	5.9	M	40000
4	Charlie	45	6.2	M	50000
5	Debbie	55	6.6	F	60000
6	Edward	65	6.1	M	70000

1. Using pandas, write a code to display only 5 rows from the data. How many males and females are in the dataset?
2. What is Edward's height?
3. Write another code to create a subset DataFrame of only female names from the DataFrame above. Reset the index and drop it as a column.
4. What is the average income per female?
5. Your boss suspects that there is a correlation between the person's age and their income. She asks you to create a plot to show this correlation. Using Pandas and Matplotlib, create a scatter plot of age against income.
6. What is the average income of males over 50 compared to the average salary of females over 50? Plot a bar plot using Matplotlib.

Day 34 - Answers

1. To display the first 5 rows of the DataFrame we use the `head()` method. First, we import `income_data` using `pd.read_csv`. By default, the `head()` method displays 5 rows of a DataFrame. However, it can also take an argument of the number of rows we want displayed.

```
[1]: import pandas as pd  
  
df = pd.read_csv("Income_data.csv")  
df.head()
```

```
[1]:      Name  Age  Height  Gender  Income  
0     Alice   25      5.5      F    30000  
1       Bob   35      5.9      M    40000  
2  Charlie   45      6.2      M    50000  
3   Debbie   55      6.6      F    60000  
4   Edward   65      6.1      M    70000
```

We will use the `value_counts()` method to know how many males and females we have in our dataset.

```
[2]: df["Gender"].value_counts()
```

```
[2]: Gender  
M    6  
F    4  
Name: count, dtype: int64
```

We have 6 males and 4 females in the dataset.

2. To find Edward's height, first we will have to find where "Edward" is located (index). We will use the index with the `.loc` attribute to retrieve Edward's height from the DataFrame. Here is the code below:

```
[3]: # Find the index for Edward
index_edward = df[df["Name"]=="Edward"].index

# Use index to find Edward's height
edward_height = df.loc[index_edward, "Height"]
print("Edward's height:", float(edward_height.iloc[0]))
```

Edward's height: 6.1

3. We can use the `query()` method to filter the DataFrame for female names. We are going to use the `reset_index()` function to reset the index. We set the drop parameter to True so that it is not inserted in the DataFrame as a column. Here is the code below:

```
[4]: # Using query to filter the DataFrame
df_females = df.query('Gender == "F"')

#Resetting the index
df_females.reset_index(inplace=True, drop=True)
df_females
```

```
[4]:      Name  Age  Height  Gender  Income
0    Alice   25      5.5      F  30000
1  Debbie   55      6.6      F  60000
2   Greta   76      6.3      F  90000
3     Ivy   60      6.7      F 110000
```

4. In this challenge, we are required to find the average income per female. The first thing to do is filter the DataFrame for female names and then calculate their average income using the `mean()` method.

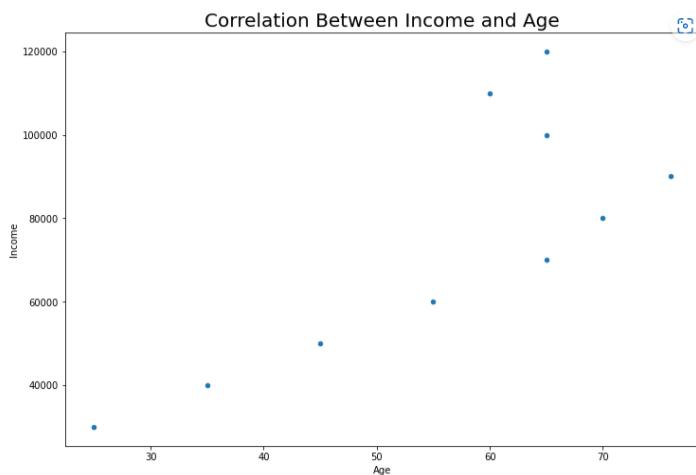
```
[5]: # Filtering the Dataframe and calculating mean()
ave_income_female = df[df['Gender'] == 'F']['Income'].mean()
print(f'The average female income is', ave_income_female)
```

The average female income is 72500.0

5. In this challenge, we are required to plot a scatter plot of the correlation between the "Age" and "Income" columns. We will use the pandas **plot()** method. The **x-axis** will be income, and the **y-axis** will be age.

```
[6]: import matplotlib.pyplot as plt

df.plot(x='Age', y='Income',
        kind='scatter',
        figsize=(12, 8))
plt.title("Correlation Between Income and Age", fontsize = 20)
plt.show()
```



It looks like an increase in age also results in an increase in income.

6. Now let's find the average income of males over 50 and the average income of females over 50 and plot a bar graph. We are going to filter the DataFrame for males and females over 50 years old and use the **mean()** method to find the average income. We are going to use the average income to plot a bar plot using Matplotlib.

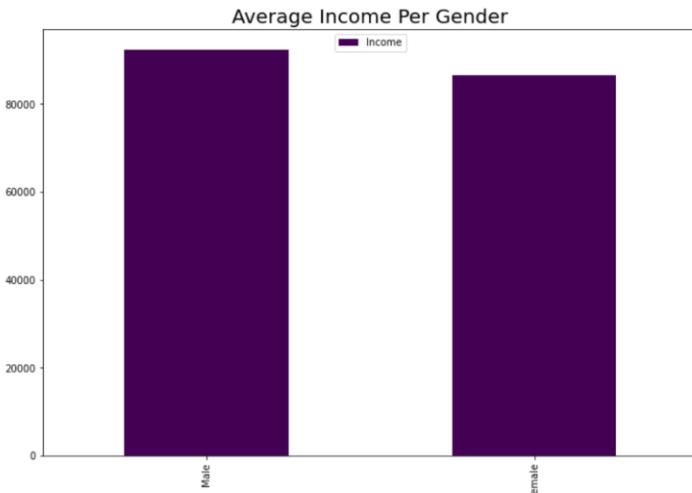
```
[7]: # Filter the Dataframe and calculate mean
male_over_50 = df[(df['Gender'] == 'M') & (df['Age'] > 50)]['Income'].mean()
female_over_50 = df[(df['Gender'] == 'F') & (df['Age'] > 50)]['Income'].mean()

print('Average income for males over 50 is', male_over_50)
print('Average income for females over 50 is', female_over_50)

# Creating a DataFrame of females and males ave income
data = {'Gender': ['Male', 'Female'],
        'Income': [male_over_50, female_over_50]}

# Plotting data
bar_df = pd.DataFrame(data)
bar_df.plot(x='Gender', y='Income', kind='bar',
            figsize=(12, 8), colormap = 'viridis')
plt.title("Average Income Per Gender", fontsize = 20)
plt.legend(loc='upper center')
plt.show()
```

Average income for males over 50 is 92500.0
 Average income for females over 50 is 86666.66666666667



Day 35: Runners And Income Data Analysis

In this challenge, you will preprocess and analyze runners data. You will import the `runners_and_income_data` file. This is a CSV file.

	A	B	C	D	E	F
1	Name	Miles Run	Time in hrs	Age	Income	Gender
2	Joe	45	40	23	50000	male
3	Phil	Nan	38	44	Nan	male
4	Ken	63	Nan	56	60000	female
5	Jos	36	50	Nan	75000	male
6	Luke	43	50	34	Nan	female

1. Load the dataset using pandas and view the first two rows of the dataset. You want to know how many rows and columns are in the DataFrame; write a code to check the number of rows and columns.
2. Write a line of code to check the total NaN values in the DataFrame. Drop rows that have all NaN values. Replace the remaining NaN values with the value 0.0 (use pandas).
3. What does the `describe()` method do? Use describe to return the **mean** of the "Miles Run" column.
4. Using the `np.sum`, what is the total number of miles run by all the names in the DataFrame?
5. How many people have an income of \$50000? Return a list of names.
6. Create a subset of the DataFrame by selecting the last two rows of the dataset. Reset the index and drop the other index column. Save the DataFrame as a CSV file. Name your subset "`runners_data_modified.csv`." Save it without the index.

Day 35 - Answers

1. Let's load the **runners_and_income_data** dataset using pandas and view the first 2 rows using the **head()** method.

```
[1]: import pandas as pd  
import numpy as np  
  
df = pd.read_csv('runners_and_income_data.csv')  
df.head(2)
```

```
[1]:    Name  Miles Run  Time in hrs   Age   Income  Gender  
0   Joe      45.0        40.0  23.0  50000.0    male  
1  Phil       NaN        38.0  44.0      NaN    male
```

The next thing is to check the number of rows and columns in the dataset. Pandas has a **shape** attribute that we can use to check both rows and columns. Rows are on **axis-0** and columns are on **axis-1**.

```
[2]: print("The number of rows: ", df.shape[0])  
print("The number of columns: ", df.shape[1])
```

```
The number of rows:  23  
The number of columns:  6
```

2. To check how many NaN values we have in the DataFrame, we can use the **isna()** method with the **sum()** method. The **sum() method** will return the total of all NaN values in the DataFrame.

```
[3]: print("Number of NaN values: ", df.isna().sum().sum())  
Number of NaN values:  50
```

To drop rows that have **all** NaN values, we can use the **dropna()** method. We will pass the parameter "all" to indicate all rows on **axis-0**.

```
[4]: # Drop rows with all NaN values  
df.dropna(axis = 0, how = 'all', inplace = True)
```

Now that we have dropped rows with all NaN values, we can replace the remaining NaN values with 0.00 using the pandas `fillna()` method.

```
[5]: # Replace NaN values with 0.00  
df.fillna(value= 0.0, inplace = True)
```

If we now check if the DataFrame has any NaN values, we get zero.

```
[6]: print("Number of NaN values: ", df.isna().sum().sum())  
Number of NaN values: 0
```

If we print out the DataFrame we can see that the NaN values have been replaced by 0.00 values.

```
[7]: df.head()
```

```
[7]:   Name Miles Run Time in hrs Age Income Gender  
0 Joe 45.0 40.0 23.0 50000.0 male  
1 Phil 0.0 38.0 44.0 0.0 male  
2 Ken 63.0 0.0 56.0 60000.0 female  
3 Jos 36.0 50.0 0.0 75000.0 male  
4 Luke 43.0 50.0 34.0 0.0 female
```

3. The `describe()` method in pandas is used to generate descriptive statistics for the DataFrame. It returns the count, mean, standard deviation, minimum, and maximum of the numeric columns in the DataFrame. It also provides the quantiles of the data. Let's use it to access the mean of the "Miles Run" column below:

```
[8]: # Mean of the "Miles Run" column  
df.describe()["Miles Run"]["mean"]
```

```
[8]: 29.470588235294116
```

4. NumPy has a `sum()` function that we can use to sum the "Miles Run" column to find the total miles run by all the runners. We are going to pass the "Miles Run" column as the argument to the `np.sum()` function.

```
[9]: print("Total miles run: ", np.sum(df["Miles Run"]))
```

```
Total miles run: 501.0
```

5. We are going to use the `query()` method to filter the data for names that have an income of \$5000. We will filter the "Income" column.

```
[10]: # Using query and iloc to filter the DataFrame for names  
names_of_5000_income = df.query("Income== 50000").iloc[0:,0]  
print(list(names_of_5000_income))  
['Joe', 'Teddy', 'Ira']
```

6. If we have a large DataFrame and we want to see the last two rows, we can use the `tail()` method. By default, the `tail()` method will return the last 5 rows of a DataFrame. We will use this method to access the last two rows of the dataset. We will use the `reset_index()` method to reset the index.

```
[11]: # Accessing the Last two rows of the DataFrame  
df2 = df.tail(2)  
# Reset the index  
df2.reset_index(inplace=True, drop= True)  
df2
```

```
[11]:      Name Miles Run Time in hrs   Age   Income Gender  
0    Barack      35.0       40.0  62.0  33000.0  male  
1  Vladamir      44.0       38.0  64.0  53000.0  male
```

We can save the DataFrame to a CSV file using the `to_csv()` method. We are going to save without the index. See the code below:

```
[12]: df2.to_csv("runners_data_modified.csv", index=False)
```

This will save the DataFrame to a file name "runners_data_modified.csv".

Day 36: Social Media Data Analysis

Below, you are going to analyze social media data. You will import the dataset below, which is saved in CSV format. The name of the file is **social_media**.

	A	B	C	D	E	F	G
1	username	age	gender	location	followers	posts	friends
2	user1	28	male	Philadelphia	5493	56	605
3	user2	22	female	Los Angeles	9530	25	683
4	user3	22	male	Philadelphia	7200	20	100
5	user4	32	female	New York	7164	90	364
6	user5	33	male	San Diego	4034	44	624
7	user6	27	male	Phoenix	1389	58	765
8	user7	28	male	Phoenix	1937	25	923
9	user8	34	female	Dallas	5344	70	452
10	user9	34	female	Chicago	8655	51	980
11	user10	27	female	Los Angeles	9825	60	575

1. Load the data using pandas. Which location has the most active users based on total posts? What is the total number of posts in this location?
2. Which gender has the least number of friends, and what is the total number of friends for this gender?
3. What is the gender, location, and age of the most active user based on posts?
4. Which city has the highest number of female users?
5. Using a pie plot type, compare the number of male and female users. Present your answer in percentages. Use only Matplotlib.

Day 36 - Answers

1. To solve this challenge, we will first load the `social_media` dataset into a pandas DataFrame using the `read_csv()` function. Then, we use the `groupby()` method to group the data by location and `sum` the number of posts per location. Then, we will use the `idxmax()` method to retrieve the location with the maximum number of posts.

```
[1]: import pandas as pd  
  
df = pd.read_csv('social_media.csv')  
df.head()
```

	username	age	gender	location	followers	posts	friends
0	user1	28	male	Philadelphia	5493	56	605
1	user2	22	female	Los Angeles	9530	25	683
2	user3	22	male	Philadelphia	7200	20	100
3	user4	32	female	New York	7164	90	364
4	user5	33	male	San Diego	4034	44	624

Now let's group the data using the `groupby()` method.

```
[2]: # Group the data by Location and sum the number of posts per Location  
location_counts = df.groupby('location').sum()['posts']  
  
# Find the Location with the maximum number of posts  
most_active_loc = location_counts.idxmax()  
print(f'The most active location based on posts is {most_active_loc}')  
print(f'The total number of posts is {location_counts.max()}')  
  
The most active location based on posts is New York  
The total number of posts is 90
```

2. To find the gender with the smallest number of friends, we need to use the `groupby()` method. We will group by gender and sum up the number of friends for each gender using the `sum()` method. We will use the `idxmin()` method to return the gender with the least number of friends.

```
[3]: # Group the data by gender and sum the number friends
gender_counts = df.groupby('gender').sum()['friends']

# Find the gender with the Least friends
least_active_gender = gender_counts.idxmin()
print(f'The gender with the least friends is {least_active_gender}')
print(f'The total number of friends is {gender_counts.male}')

The gender with the least friends is male
The total number of friends is 3017
```

You can see above that males have the least number of friends, and the total number of friends is 3017.

- Now, to find the age, gender, and location of the most active user, we will use the `sort_values()` method to sort the DataFrame by the "post" column in descending order. We will then use the `.loc` attribute with the `head()` method to return the age, gender, and location in the top row of the sorted DataFrame.

```
[4]: # Finding the age of most active user based on posts
age_of_user = df.sort_values(by="posts", ascending=False).loc[:, "age"].head(1)

# Finding the gender of most active user based on posts
gender_of_user = df.sort_values(by="posts", ascending=False).loc[:, "gender"].head(1)

# Finding the city of most active user based on posts
city_of_user = df.sort_values(by="posts", ascending=False).loc[:, "location"].head(1)

print(f'The age of the most active is {age_of_user.iloc[0]}')
print(f'The gender of the most active user is {gender_of_user.iloc[0]}')
print(f'The city with the most active user is {city_of_user.iloc[0]}')

The age of the most active is 32
The gender of the most active user is female
The city with the most active user is New York
```

- In order to find which city has the highest number of female users, we can use the `groupby()` method to group the users by their location and gender, and then use the `idxmax()` function to find the location with the highest number of female users. Here is the code below:

```
[5]: # Group the data by city and gender, and count the number of users in each group
city_gender_counts = df.groupby(["location", "gender"]).size().reset_index(name = "count")

# Filter the data to only include rows with female users
female_users = city_gender_counts[city_gender_counts["gender"] == "female"]

# Find the city with the highest number of female users
most_female_users_city = female_users["count"].idxmax()

# Return the city with the highest number of female
city = female_users.iloc[most_female_users_city][["location"]]

print(f'The city with the most female users is {city}')
```

The city with the most female users is Los Angeles

You can see that Los Angeles has the highest number of female users.

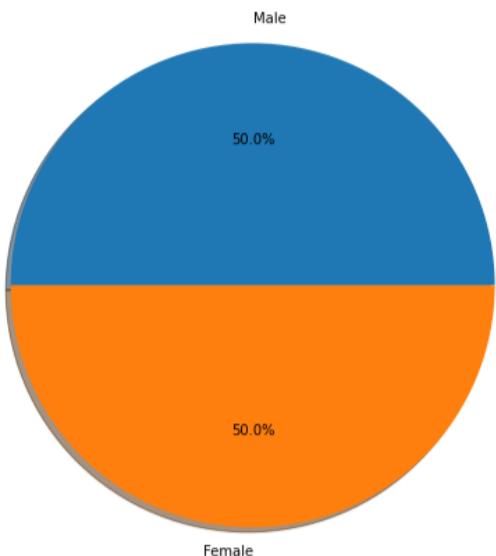
5. To answer this question, we will use pandas and Matplotlib. We will use the pandas `value_counts()` method on the `gender` column to count the number of male and female users. Then, we will use the `pie()` method of Matplotlib to create a pie plot of the gender distribution. We can set the `autopct` parameter to display the percentages on the chart.

```
[6]: import matplotlib.pyplot as plt

# Count the number of male and female users
gender_counts = df['gender'].value_counts()

# Create the pie chart
plt.figure(figsize=(12,8))
plt.pie(gender_counts, labels=['Male', 'Female'],
        autopct='%1.1f%%',
        shadow=True)
plt.title('Gender Distribution', fontsize= 20)
plt.show()
```

Gender Distribution



The gender distribution is 50-50

Day 37: Stock Market Data Processing and Analysis

Stock market data is typically time-series data, which means it's collected and recorded over time. For this challenge, you are going to import the Fusion Systems `fusion_stock_data` dataset. This is a CSV file.

	A	B	C	D
1	Date	Company	Price	Volume
2	1 01 2022	Fusion Systems	66	15698
3	2 01 2022	Fusion Systems	53	42461
4	3 01 2022	Fusion Systems	92	26775
5	4 01 2022	Fusion Systems	55	27965
6	5 01 2022	Fusion Systems	56	44423

1. Load the dataset. Write a code to check the data types of the columns in the dataset. Check the data for any missing values using the `isnull()` method.
2. Convert the date column into datetime format and set it as the index.
3. Using the pandas `plot()` method and Matplotlib, plot a line plot of the stock price over time. Ensure that your plot has axis labels and a title.
4. What was the volume on the day with the lowest stock price?
5. Calculate the daily returns of the stock price using pandas "`pct_change`" and plot a line plot using pandas.
6. What date had the highest price?

Day 37 - Answers

1. First, we import the `fusion_stock_data` dataset using `pd.read_csv()` function. We are going to display the first five rows using the `head()` method.

```
[1]: import pandas as pd  
  
df = pd.read_csv("fusion_stock_data.csv")  
df.head()
```

```
[1]:      Date      Company  Price  Volume  
0  1 01 2022  Fusion Systems    66   15698  
1  2 01 2022  Fusion Systems    53   42461  
2  3 01 2022  Fusion Systems    92   26775  
3  4 01 2022  Fusion Systems    55   27965  
4  5 01 2022  Fusion Systems    56   44423
```

To check the data types of the DataFrame, we will use the `pandas dtypes` attribute. This will return the data type of each column. Here is the code below:

```
[2]: df.dtypes  
  
[2]: Date        object  
      Company    object  
      Price      int64  
      Volume     int64  
      dtype: object
```

To check for missing values, we will use the `isnull()` method and `sum` the total of the missing values in the DataFrame.

```
[3]: df.isnull().sum()
```

```
[3]: Date      0  
Company    0  
Price      0  
Volume     0  
dtype: int64
```

2. We are going to use the pandas `to_datetime()` function to convert the date to pandas datetime format. We are setting the parameter, `inplace = True` to modify the original DataFrame. We will use the `set_index()` method to set the "Date" column as an index.

```
[4]: # Convert data types into date time  
df["Date"] = pd.to_datetime(df["Date"])  
  
# Set the date as the index  
df.set_index("Date", inplace=True)  
df.head()
```

```
[4]:
```

		Company	Price	Volume
	Date			
2022-01-01	Fusion Systems	66	15698	
2022-02-01	Fusion Systems	53	42461	
2022-03-01	Fusion Systems	92	26775	
2022-04-01	Fusion Systems	55	27965	
2022-05-01	Fusion Systems	56	44423	

You can see above that the "Date" column is now the index of the DataFrame.

3. We are going to use pandas and Matplotlib to analyze the stock price over time. The `xlabel` will be the date, and the `ylabel` will be the price. With this plot, we can see the stock price over a given period of time.

```
[5]: import matplotlib.pyplot as plt

df.plot(y='Price', figsize=(10,8))
plt.xlabel('Date')
plt.ylabel('Price')
plt.title('Stock Price Over Time', fontsize = 20)
plt.show()
```

This will output:



4. Using the `.loc` attribute and `idxmin()` method, we will find the volume amount on the day with the lowest price.

```
[6]: # Using Loc to find the volume on lowest price
volume_lowest_price = df.loc[df['Price'].idxmin()]["Volume"]

print(f'Volume on lowest price: {volume_lowest_price}')
```

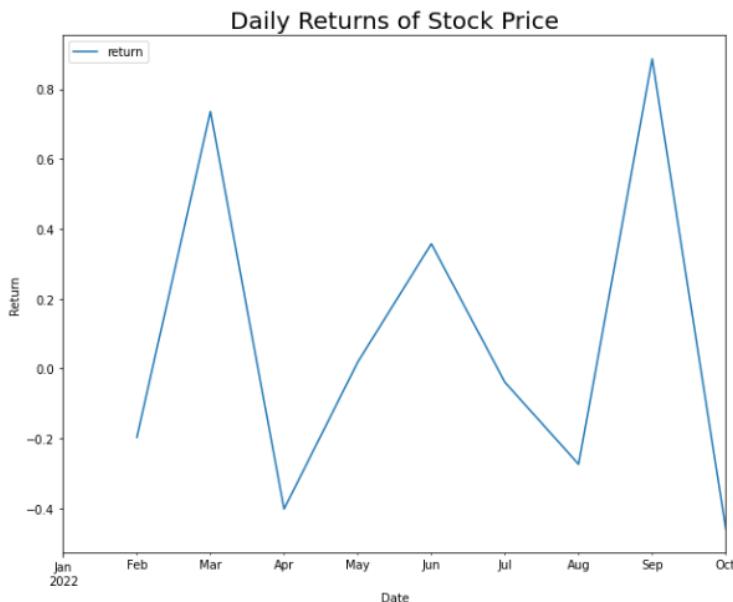
Volume on lowest price: 42461

5. The pandas `pct_change()` function returns a change between the current and prior periods. We are going to use this data to plot a line plot.

```
[7]: df["return"] = df["Price"].pct_change()

df.plot(y='return', figsize=(10,8))
plt.xlabel('Date')
plt.ylabel('Return')
plt.title('Daily Returns of Stock Price', fontsize = 20)
plt.show()
```

This will output:



6. To find the date with the highest price, first we need to access the "Price" column and retrieve the maximum value. We will use this maximum value to retrieve the date with the highest stock price.

```
[8]: # Finding the maximum price in the price column
max_price = df.loc[:, "Price"].max()

# Returning row with maximum price
max_price_row = df.loc[df["Price"] == max_price]

# Retrieving the date
date_with_max_price = max_price_row.index[0]
date_with_max_price
```

[8]: Timestamp('2022-09-01 00:00:00')

You can see that the highest price was on 22-09-01.

Day 38: Rental Car Data Analysis

For this challenge, you are going to preprocess and analyze a dataset of car rental services. You are going to use the **rental_car_analysis** dataset. Here is a snippet of the data below:

	A	B	C	D	E	F
1	Date	City	Car Type	Rental Price	Discount	Final Price
2	1 01 2022	New York	SUV			74
3	1 01 2022	Chicago	Van	179	0.123537522	156.8867835
4	1 01 2022	Los Angeles	Van		0	144
5	1 01 2022	Houston	Truck	93		93
6	1 01 2022	Phoenix	Truck		0	124

1. Write code to check how many rows are in the dataset.
Check the sum of duplicates in the "City" column.
2. Write code to find missing data in the generated DataFrame. Which rows have missing data?
3. Using pandas, fill in the missing values using the column **median** (numeric columns only).
4. Which city brought in the most revenue?
5. Which car type brought in the most money?
6. Which car type is the most expensive to rent?
7. Which two cities brought in the least amount of revenue?
8. Using pandas **plot()** function and Matplotlib, plot a bar plot of the total revenue brought in by each type of car.
9. What is the total revenue of this car rental business?
10. What would be the total revenue if no discount was given?

Day 38 - Answers

1. First, we import the dataset and view the last 5 rows, we use the `tail()` method.

```
[1]: import pandas as pd  
  
df = pd.read_csv("rental_car_analysis.csv")  
df.tail()  
  
[1]:
```

	Date	City	Car Type	Rental Price	Discount	Final Price
95	10 01 2022	Philadelphia	Sports	145.0	0.0	145.000000
96	10 01 2022	San Antonio	Sports	NaN	NaN	190.000000
97	10 01 2022	San Diego	Sedan	64.0	0.0	64.000000
98	10 01 2022	Dallas	Truck	NaN	0.0	112.000000
99	10 01 2022	San Jose	Van	172.0	NaN	98.393954

We will use the `shape` attribute of the DataFrame to check the number of rows in the dataset. This attribute returns a tuple of (number of rows, number of columns). Rows are on axis 0.

```
[2]: # Printing number of rows in the dataset.  
print(df.shape[0])
```

100

To check the number of duplicates in a column, we will use the `duplicated()` method and pass the column as an argument to the subset parameter.

```
[3]: df.duplicated(subset=["City"]).sum()
```

90

2. To find missing data in the generated DataFrame, we can use the `isnull()` method of the DataFrame. We can also use the `isna()` method. Both of these methods will return the same results. The `sum()` method will return the total of missing values in each column.

```
[4]: print(df.isnull().sum())
```

```
Date      0  
City      0  
Car Type  0  
Rental Price 50  
Discount   34  
Final Price 0  
dtype: int64
```

You can see that the "Rental Price" column has 50 missing values and the "Discount" column has 34 missing values.

- Now, to fill in the missing values using the column median, we will use the `fillna()` method of the DataFrame. We will select the numeric columns with missing values, and use the `median()` function to calculate the median of each column.

```
[5]: # Filling missing values with the median of the column  
df2 = df.fillna(df[["Rental Price", "Discount"]].median())  
  
# Checking for missing values  
df2.isnull().sum()
```

```
[5]: Date      0  
City      0  
Car Type  0  
Rental Price 0  
Discount   0  
Final Price 0  
dtype: int64
```

The output shows that we do not have any missing values in the DataFrame anymore.

- To find the city that brought in the most revenue, we will group the DataFrame by the "City" column and sum the "Final Price" column, and then use the `idxmax()` method on the resulting DataFrame to find the city that has the highest revenue.

```
[6]: city_most_revenue = df2.groupby('City')['Final Price'].sum().idxmax()  
print("The City with most revenue: ", city_most_revenue)
```

```
The City with most revenue: Phoenix
```

5. To find the car type that brought in the most money, we will group the DataFrame by "Car Type" and then use the `sum()` method on the "Final Price" column, and then use the `idxmax()` method on the resulting DataFrame to retrieve the car type that has the highest revenue.

```
[7]: most_revenue_car = df2.groupby('Car Type')['Final Price'].sum().idxmax()
print("Car Type with most revenue: ",most_revenue_car)
```

Car Type with most revenue: Van

6. We are going to group the DataFrame by "Car Type" and then use the `mean()` method on the "Rental Price" column and then use the `idxmax()` method on the resulting DataFrame to find the car type that has the highest mean rental price.

```
[8]: most_expensive_car = df2.groupby('Car Type')['Rental Price'].mean().idxmax()
print("The most expensive car type to rent: ",most_expensive_car)
```

The most expensive car type to rent: Van

7. To retrieve the two least profitable cities, we can group the DataFrame by the "City" column and sum the "Final Price." We will sort the DataFrame in descending order and use the `tail()` method to return the two least profitable cities.

```
[9]: # Group by city and sum the revenue
group_by_city = df2.groupby('City')['Final Price'].sum()

# Sorting the list in descending order return the two least profitable cities
least_profitable_car = group_by_city.sort_values(ascending=False).tail(2)
least_profitable_car
```

```
[9]: City
Houston      951.338397
San Antonio  870.153279
Name: Final Price, dtype: float64
```

You can see that the least profitable cities are Houston and San Antonio.

8. Let's plot the total revenue of each car. First, we will group the DataFrame by car type and sum the "Final Price"

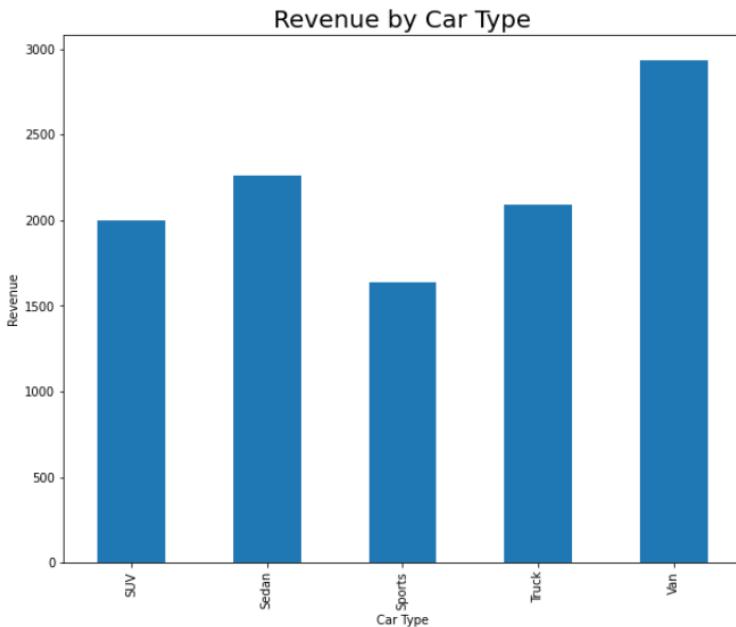
column. We are going to use the bar plot from Matplotlib to create the graph.

```
[10]: import matplotlib.pyplot as plt

# Group the data by car type and sum the revenue
car_revenue = df2.groupby('Car Type')['Final Price'].sum()

# Plot the revenue of each car type
plt.figure(figsize=(10, 8))
car_revenue.plot(kind='bar', x='car_type',
                  y='car_revenue')
plt.title("Revenue by Car Type", fontsize = 20)
plt.ylabel(ylabel= "Revenue")
plt.show()
```

This will output the plot below. You can see that the van brought in the most revenue, and the sports car brought in the least revenue.



9. To get the total revenue for the whole business, we can sum the "Final Price" column.

```
[11]: total_revenue = df2['Final Price'].sum()  
print(f'The total_revenue is {total_revenue:.2f}')  
  
The total_revenue is 10926.06
```

10. To calculate the revenue before discount, we have to sum the "Rental Price" column. This is because this column has the price before the discount is applied.

```
[12]: # Using the rental price column to find total before discount  
no_discount_revenue = df2['Rental Price'].sum()  
print(f'The total revenue before discount is {no_discount_revenue:.2f}')  
  
The total revenue before discount is 12542.00
```

Day 39: Analyze, Transform, and Shift Data.

In this challenge, you will analyze and transform data. You will import the `cars_and_careers` CSV file. Here is a sample of the dataset below:

	A	B	C	D
1	Name	Occupation	Age	Car
2	John	ENGINEER	32	Toyota Camry
3	Emily	TEACHER	28	Honda Civic
4	Michael	DOCTOR	45	Ford Mustang
5	Emma	LAWYER	37	BMW 3 Series

1. Import the `cars_and_careers` dataset. Using the pandas `describe()` method, what is the `mean` age of the "Age" column?
2. Using the pandas `transform()` method, convert the items in the "cars" column into lowercase letters.
3. You have come across some new information that must be added to your DataFrame. Using pandas `shift()` and `.iloc` attributes, insert a row into your DataFrame. This row will sit at index 0. The row is: `["Casy", "Ford", 31]`. The last row `["Ben", "Toyota", 55]` must be removed.
4. Using the `pandas.str.find()` method, write a code to confirm if the name "Ben" has been removed from the DataFrame.
5. What car does a person by the name of "Emily" drive? And what is her occupation?
6. Which car is driven by the oldest person? Using Matplotlib, plot a bar plot of the cars driven by the 5 oldest people and their ages in descending order.
7. For data to be used in machine learning algorithms, it must be converted to a numerical format. This is because machine learning algorithms can only understand numbers. Your task now is to write code that will convert the text columns into numeric data types for machine learning using first, pandas and then Sklearn.

Day 39 - Answers

1. First, we are going to import the "cars_and_careers" dataset using pandas.

```
[1]: import pandas as pd  
  
df = pd.read_csv("cars_andcareers.csv")  
df.head()
```

	Name	Occupation	Age	Car
0	John	ENGINEER	32	Toyota Camry
1	Emily	TEACHER	28	Honda Civic
2	Michael	DOCTOR	45	Ford Mustang
3	Emma	LAWYER	37	BMW 3 Series
4	David	ARTIST	41	Mercedes-Benz C-Class

The **describe()** method returns a summary of the descriptive statistics of the DataFrame. Here is how we find the mean of the "Age" column using the **describe()** method: We are going to select the "Age" column and call the describe method.

```
[2]: age_mean = df["Age"].describe()["mean"]  
print('The mean age is', age_mean)
```

The mean age is 34.75

2. The **transform()** method can be used to transform or modify values in a column or row using a function. We are going to use this method to convert the names of the cars to lowercase. The function will transform the data using the **lambda** function.

```
[3]: # Create a copy of DataFrame
df_copy = df.copy()

df_copy["Occupation"] = df_copy["Occupation"].transform(lambda x: x.capitalize())
df_copy.head()
```

```
[3]:
```

	Name	Occupation	Age	Car
0	John	Engineer	32	Toyota Camry
1	Emily	Teacher	28	Honda Civic
2	Michael	Doctor	45	Ford Mustang
3	Emma	Lawyer	37	BMW 3 Series
4	David	Artist	41	Mercedes-Benz C-Class

3. The `shift()` method is a data transformation method that shifts the index or data in a DataFrame or Series by a specified number of periods. By default, this method moves data up or down by one row. However, you can also specify a different number of periods to shift by.

We want to remove the last row in the DataFrame ["Grace," "Actor," 29, "Ford Explorer"] and insert a new row at the top of the DataFrame ["Casy," "Dancer," 31, "Tesla"]. When we apply the method to our DataFrame, the first row will shift from index 0 to index 1. This means that the last row will be pushed out of the DataFrame. We will use the `.iloc` attribute to insert the new row at index 0.

```
[4]: # Using shift to move row
df1 = df_copy.shift(periods=1, axis = "rows")

# Using iloc to insert new row
df1.iloc[0] = ["Casy", "Dancer", 31, "Tesla"]
df1.head()
```

```
[4]:
```

	Name	Occupation	Age	Car
0	Casy	Dancer	31.0	Tesla
1	John	Engineer	32.0	Toyota Camry
2	Emily	Teacher	28.0	Honda Civic
3	Michael	Doctor	45.0	Ford Mustang
4	Emma	Lawyer	37.0	BMW 3 Series

4. We can use the pandas `str.find()` method to check if a specific substring, such as "Grace," is present in the "Names" column of a DataFrame. This method searches each row in the column and returns the lowest index of the substring's occurrence. If the substring is not found in a row, it returns -1.

To check if "Grace" is not present in the DataFrame, we can compare the sum of all the values returned by `str.find()` to the length of the DataFrame. If the sum is equal to the negative length of the DataFrame, it confirms that "Grace" is not present.

```
[5]: # Check if "Grace" is not present in the DataFrame
is_grace_present = not (df1['Name'].str.find('Grace').sum() == -len(df1))
is_grace_present
```

```
[5]: False
```

You can see that it returns False. This means that "Grace" is not in the DataFrame."

5. First, we will use the `index` attribute to access the index or row of "Emily" in the "Name" column. We will use that index to access the car and occupation of Emily using the `.loc` attribute.

```
[6]: # Getting Emily's index in the DataFrame
emily_index = df1.index[df1["Name"]=="Emily"].tolist()

# Using index to get Emily's car
emily_car = df1.loc[emily_index, "Car"]

# Using index to get Emily's occupation
emily_occupation = df1.loc[emily_index, "Occupation"]
print(f'Emily drives a: {str(emily_car.tolist()[0])}')
print(f'Emily's occupation is: {emily_occupation.tolist()[0]}')

Emily drives a: Honda Civic
Emily's occupation is: Teacher
```

6. Below, we use `idxmax()` to find the index of the oldest person in the DataFrame. We use the `.loc` attribute to locate the car driven by this person.

```
[7]: # Finding the index of the oldest person
oldest_person_index = df1['Age'].idxmax()

# Using index to find the oldest car
name_car = df1.loc[oldest_person_index, "Car"]

print(f'The oldest person drives a {name_car}')
```

The oldest person drives a Ford Mustang

To plot the data of the cars and the age of the 5 oldest persons, first we need to sort the data in descending order by the "Age" column and return the top 5 rows from the DataFrame.

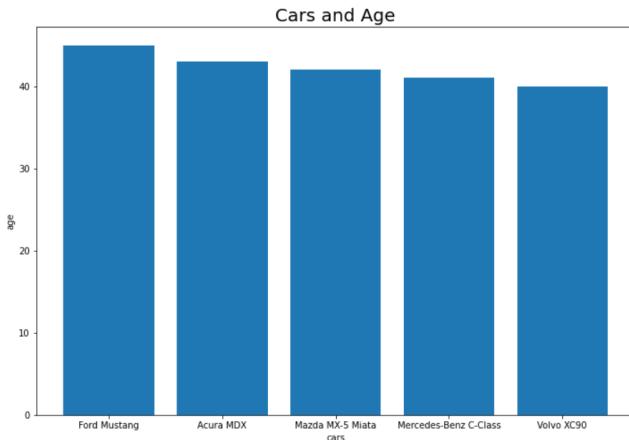
```
[8]: import matplotlib.pyplot as plt

# sorting data by the "Age" column
df1_sorted = df1.sort_values(by="Age", ascending=False).head()

# Getting the cars column
cars = df1_sorted["Car"]

# Getting the age column
age = df1_sorted["Age"]

plt.figure(figsize=(12, 8))
plt.bar(cars, age)
plt.xticks(rotation=90)
plt.xlabel("cars")
plt.ylabel("age")
plt.title("Cars and Age", fontsize=20)
plt.show()
```



7. Data must be transformed into a numerical format in order to be used in machine learning algorithms. This is due to the inability of machine learning algorithms to process non-numerical data types, which necessitates their conversion to numerical data types.

Pandas has a way we can easily convert non-numerical columns into numbers for machine learning. We can use the `pd.get_dummies()` function. This function will create dummy variables for each column we are converting. It creates a new DataFrame with binary (0 and 1) columns for each unique value in the specified columns of the original DataFrame.

```
[9]: # Create dummy variables
df_modified = pd.get_dummies(df1, columns=["Name", "Occupation", "Car"], dtype=int)
df_modified
```

	Age	Name_Alexander	Name_Amelia	Name_Andrew	Name_Ava	Name_Benjamin	Name_Casy	Nam
0	31.0	0	0	0	0	0	0	1
1	32.0	0	0	0	0	0	0	0
2	28.0	0	0	0	0	0	0	0
3	45.0	0	0	0	0	0	0	0
4	37.0	0	0	0	0	0	0	0

```
[10]: # Checking the number of columns  
df_modified.shape[1]
```

```
[10]: 61
```

The resulting DataFrame from this code will have a new column for every unique value in the columns ("Car," "Name," and "Occupation"). Notice that the resulting DataFrame has 61 columns, one column for each unique item in those original columns.

The resulting large DataFrame is one of the main disadvantages of using `pd.get_dummies` to convert non-numerical data if you have many columns with unique values. A large dataset will cause memory inefficiency.

The second method we can use to convert the non-numerical columns is from the Sklearn library. We can use the `LabelEncoder` class from `sklearn.preprocessing`.

The resulting DataFrame will have the same number of columns as the original DataFrame. The non-numeric items in the two columns have been converted to numbers. See the code below:

```
[11]: from sklearn.preprocessing import LabelEncoder

# Initialize label encoders
cars = LabelEncoder()
names = LabelEncoder()
occupation = LabelEncoder()

# Fit the label encoders to the data
df1["Car"] = cars.fit_transform(df["Car"])
df1["Name"] = names.fit_transform(df["Name"])
df1["Occupation"] = names.fit_transform(df["Occupation"])
df1.head()
```

```
[11]:   Name Occupation  Age  Car
      0      13          8  31.0  17
      1       8          18  32.0   7
      2      16          7  28.0   5
      3       9          11  45.0   2
      4       7          3  37.0  13
```

One of the advantages of using LabelEncoder is that it maintains the original number of columns and their categorical order in the numerical representation.

Day 40: Car Spare Parts Data Analysis

You are going to analyze the sales data of a spare parts business. You are going to use the `spare_parts_expanded.csv` dataset.

	A	B	C	D	E	F	G
1	spare_part	quantity	costs	sale_price	date	total_revenue	
2	Wheel	24	30	53	1 01 2020	\$1,272	
3	Tyre	20	80	110	2 01 2020	\$2,200	
4	Battery	46	85	95	3 01 2020	\$4,370	
5	Air Filter	37	36	48	4 01 2020	\$1,776	
6	Oil Filter	28	96	120	5 01 2020	\$3,360	

1. Load the CSV dataset above. Check the "date" and "revenue" data types. Write another code to check for any duplicates in the "spare_parts" column. Use a histogram to visualize the distribution of total sales for the entire store. Do you notice any outliers in the histogram?
2. Box plots are useful for identifying outliers, visualizing the median, quartiles, and range of the dataset, and comparing the distributions of different groups or categories within the dataset. Use a box plot of the Seaborn library to compare the distribution of total revenue for each product.
3. Use a Matplotlib scatter plot to visualize the relationship between products and "total_revenue" for each item. Can you identify on the plot which product brought in the most income?
4. Filter the DataFrame to return only columns with integer data types. Save this as a new variable.
5. What is the most profitable and least profitable product? What is the difference in profit between the most profitable and the least profitable product? By what percentage will the total profit drop if the least profitable product is dropped?
6. Use a 3D scatter plot to visualize the relationship between price, quantity, and cost for each item. Use Matplotlib.

7. The Seaborn **pairplot** is a very important tool for visualizing relationships between variables in a dataset. Use the pairplot to visualize the relationship of the 'total_revenue', "sale_price", "costs", "quantity" and "spare_parts" columns (hue="spare_parts").
8. Use a Seaborn **barplot** to visualize the distribution of quantities for each item.
9. Use a Seaborn **lineplot** to visualize the trend of total sales over time for all items.
10. Use a Seaborn **Implot** to fit a linear regression model and visualize the relationship between price and total revenue. Your plot should have a white grid style and a height of 8 inches. What can you conclude from the plot?

Day 40 - answers

1. The first thing that we have to do is import the dataset and inspect it. Check for missing data and data types. Once we are happy with the data, we can start the analysis.

```
[1]: import pandas as pd  
  
df = pd.read_csv("spare_parts_expanded.csv")  
df.head()
```

	spare_parts	quantity	costs	sale_price	date	total_revenue
0	Wheel	24	30	53	1 01 2020	\$1,272
1	Tyre	20	80	110	2 01 2020	\$2,200
2	Battery	46	85	95	3 01 2020	\$4,370
3	Air Filter	37	36	48	4 01 2020	\$1,776
4	Oil Filter	28	96	120	5 01 2020	\$3,360

Now let's check the data types of the "date" and "total_revenue" columns:

```
[2]: df[["date", "total_revenue"]].dtypes
```

```
[2]: date          object  
      total_revenue    object  
      dtype: object
```

Notice that the "date" and "total_revenue" columns are object data types. We will have to convert them to the appropriate data type for analysis purposes.

Now let's check for missing data.

```
[3]: df.isnull().sum()
```

```
[3]: spare_parts      0  
      quantity        0  
      costs           0  
      sale_price      0  
      date            0  
      total_revenue    0  
      dtype: int64
```

The check for missing data returns zero (0) missing data in all columns.

Let's check for duplicates in the "spare_parts" column:

```
[4]: df["spare_parts"].duplicated().sum()
```

```
[4]: 10
```

We have 10 duplicates in the column.

Now, let's clean our data. We will remove the comma and the "\$" sign from the "total_revenue" column.

```
[5]: # Remove comma from the total revenue column
df["total_revenue"] = df["total_revenue"].replace(",", "", regex=True)

# Remove $ from the total_revenue column
df["total_revenue"] = df["total_revenue"].str.replace("$", "").astype(int)
df.head()
```

```
[5]:   spare_parts  quantity  costs  sale_price      date  total_revenue
  0      Wheel       24     30        53 1 01 2020        1272
  1      Tyre        20     80        110 2 01 2020       2200
  2    Battery       46     85        95 3 01 2020       4370
  3   Air Filter     37     36        48 4 01 2020       1776
  4   Oil Filter     28     96        120 5 01 2020       3360
```

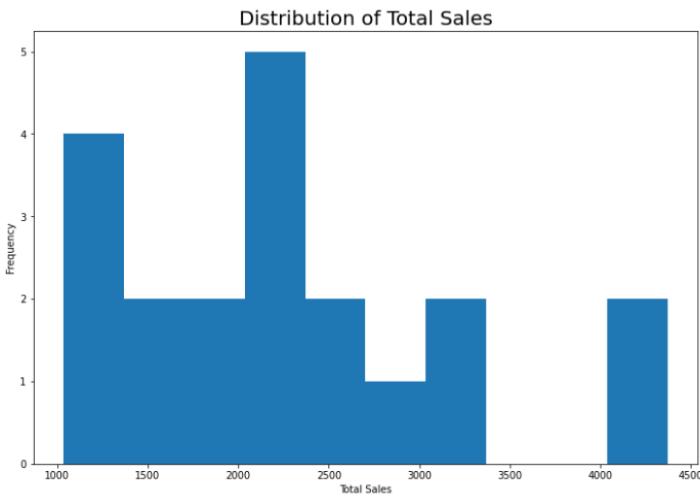
You can see that the "total_revenue" column has been cleaned.

To create a histogram to visualize the distribution of total sales for the entire store, we will use the Matplotlib library. We will create a histogram using the `hist()` method, passing in the "total_revenue" column as the argument. We will add a title and labels using the `title()`, `xlabel()`, and `ylabel()` methods. This histogram will return the frequency of the sales.

```
[6]: import matplotlib.pyplot as plt

plt.figure(figsize=(12, 8))
plt.hist(df['total_revenue'], bins = 10)
plt.xlabel("Total Sales")
plt.ylabel("Frequency")
plt.title("Distribution of Total Sales", fontsize = 20)
plt.show()
```

This will output:



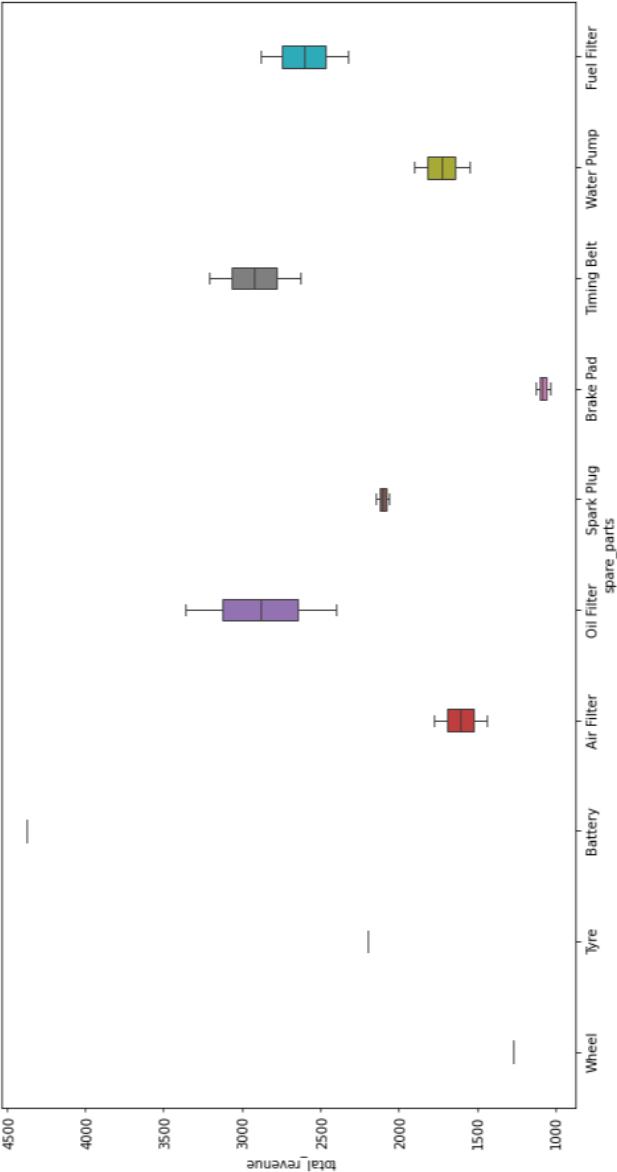
The sales between 4000 and 4500 are the outliers in the sales column.

2. We are going to use the Seaborn library to create a boxplot to compare the distribution of total sales. We will import Seaborn as **sns**. We will create a boxplot using the "spare_parts" column as the **x-axis** and the "total_revenue" column as the **y-axis**.

```
[7]: import seaborn as sns  
  
plt.subplots(figsize=(15, 8))  
sns.boxplot(x="spare_parts", y="total_revenue",  
            data=df,  
            width=0.2,  
            linewidth=1.0 )  
plt.title("Total Sales Distribution by Item", fontsize= 20)  
plt.show()
```

You will notice below that the boxplot consists of a box and two whiskers, which indicate the interquartile range (IQR) of the data. The box represents the middle 50% of the data, and the line inside the box represents the median value. The whiskers extend from the box to the minimum and maximum values that are within 1.5 times the IQR. Anything outside this range is considered an outlier.

Total Sales Distribution by Item

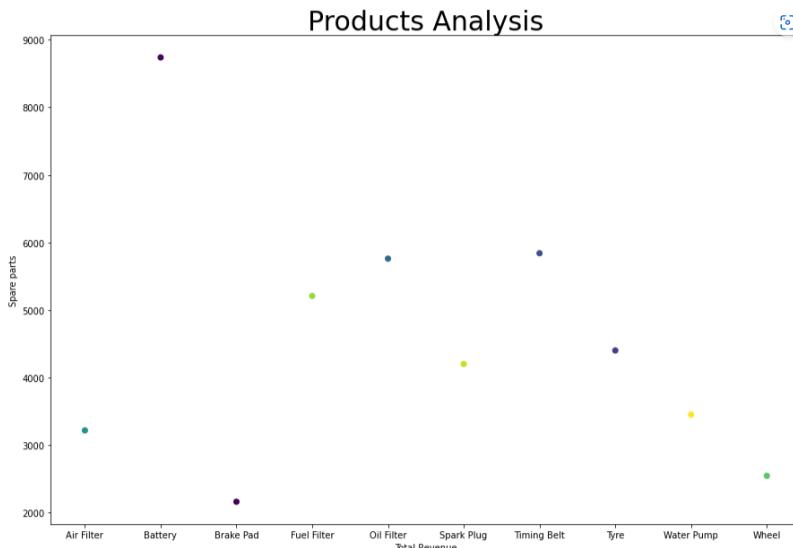


3. Let's create a scatter plot using Matplotlib. Because we have duplicates in the "spare_parts" column, we will group the DataFrame by the "spare parts" column, and sum the total revenue. We are going to use NumPy to generate random colors for the data points.

```
[8]: import numpy as np

# Group the dataframe by spare parts
grouped_sales = df.groupby("spare_parts")["total_revenue"].sum()

# Setting up the graph size
fig, ax = plt.subplots(figsize=(15, 10))
colors = np.random.rand(10)
plt.scatter( grouped_sales.index,grouped_sales, c=colors, alpha=1.0)
plt.xlabel(xlabel="Total Revenue")
plt.ylabel(ylabel= "Spare parts")
plt.title("Products Analysis", fontsize= 30)
plt.show()
```



From the plot, we can tell that the battery brought in the most revenue.

4. To filter the DataFrame for columns that have only integer **dtypes**, we can use the **df.select_dtypes()** method. We are

going to pass "int" as an argument to tell the function to return only columns that have integer data types. See the code below:

```
[9]: df2 = df.select_dtypes(include='int')
df2.head()
```

```
[9]:   quantity  costs  sale_price  total_revenue
      0        24     30          53        1272
      1        20     80         110        2200
      2        46     85          95        4370
      3        37     36          48        1776
      4        28     96         120        3360
```

Additionally, we can also use the **exclude** parameter to exclude specific data types from the selection.

5. First, we have to create a "total_costs" column by multiplying the "costs" column by the "quantity" column. We will subtract total costs from total revenue to create the "profit" column. Here is the modified DataFrame below:

```
[10]: df["total_costs"] = df["costs"] * df["quantity"]
df["profit"] = df["total_revenue"] - df["total_costs"]
df.head()
```

```
[10]:   spare_parts  quantity  costs  sale_price  date  total_revenue  total_costs  profit
      0      Wheel       24     30          53  1 01 2020        1272        720      552
      1      Tyre        20     80         110  2 01 2020        2200       1600      600
      2    Battery       46     85          95  3 01 2020        4370       3910      460
      3  Air Filter      37     36          48  4 01 2020        1776       1332      444
      4  Oil Filter       28     96         120  5 01 2020        3360       2688      672
```

Next, we are going to group data by the "spare parts" column and sum the "profit" column. We will use the **.loc** attribute with the **idxmax()** method and the **idxmin()** method to find the most and least profitable products.

```
[11]: # Grouping data by spare parts
df_group = df.groupby("spare_parts")[["profit"]].sum()

# getting the most profitable product
most_profitable_product = df_group.loc[:, "profit"].idxmax()

# getting the least profitable product
least_profitable_product = df_group.loc[:, "profit"].idxmin()

print(f'The most profitable product is: {most_profitable_product}')
print(f'The least profitable product is: {least_profitable_product}')

The most profitable product is: Timing Belt
The least profitable product is: Air Filter
```

We will use the `idxmax()` method to get the maximum profit amount and the `idxmin()` method to get the minimum profit amount. We use these amounts to calculate the difference.

```
[12]: # getting the most profitable product and amount
max_value_row = df_group.loc[df_group['profit'].idxmax()]

# getting the minimum profitable product and amount
min_value_row = df_group.loc[df_group['profit'].idxmin()]

difference_in_profit = max_value_row - min_value_row
print("Profit difference between the most profitable and least profitable "
      "product is", difference_in_profit.iloc[0])

Profit difference between the most profitable and least profitable product is 1556
```

Let's now calculate the drop in profit if the least profitable product is dropped. We will calculate the sum of the "profit" column and access the value of the least profitable product. We will divide the least profitable product value by the total profit to find the drop in profit.

```
[13]: # Sum the profit column
total_profit = df_group["profit"].sum()

# Find the Least profit value
least_profitable_profit = df_group["profit"].min()

# Calculate the percentage drop
percentage_drop = least_profitable_profit/total_profit*100
print(f'{percentage_drop:.2f}%')

5.77%
```

The drop in profit is 5.77%.

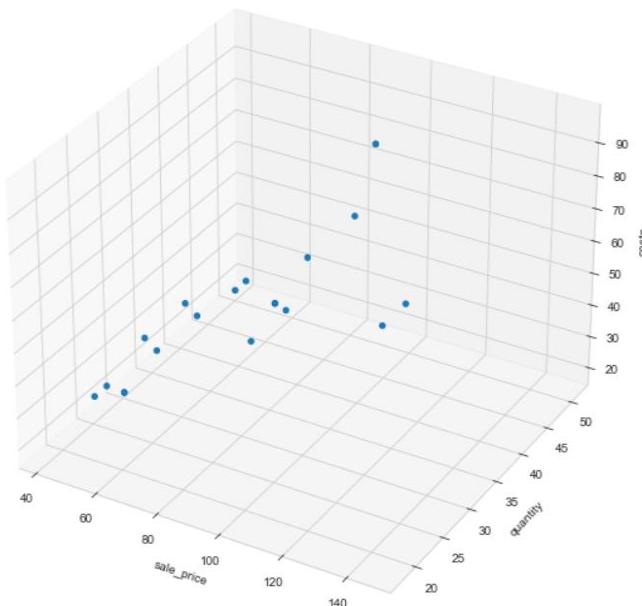
6. Unlike a 2D plot that requires two axes, a 3D plot will require three axes. We will use the three columns "sale_price," "quantity," and "costs" to create this plot using Matplotlib.

```
[14]: fig = plt.figure(figsize=(15, 10))
ax = fig.add_subplot(111, projection='3d')

# Plot the data using the scatter() method
ax.scatter(df['sale_price'],
           df['quantity'],
           df['costs'],
           alpha=1.0)

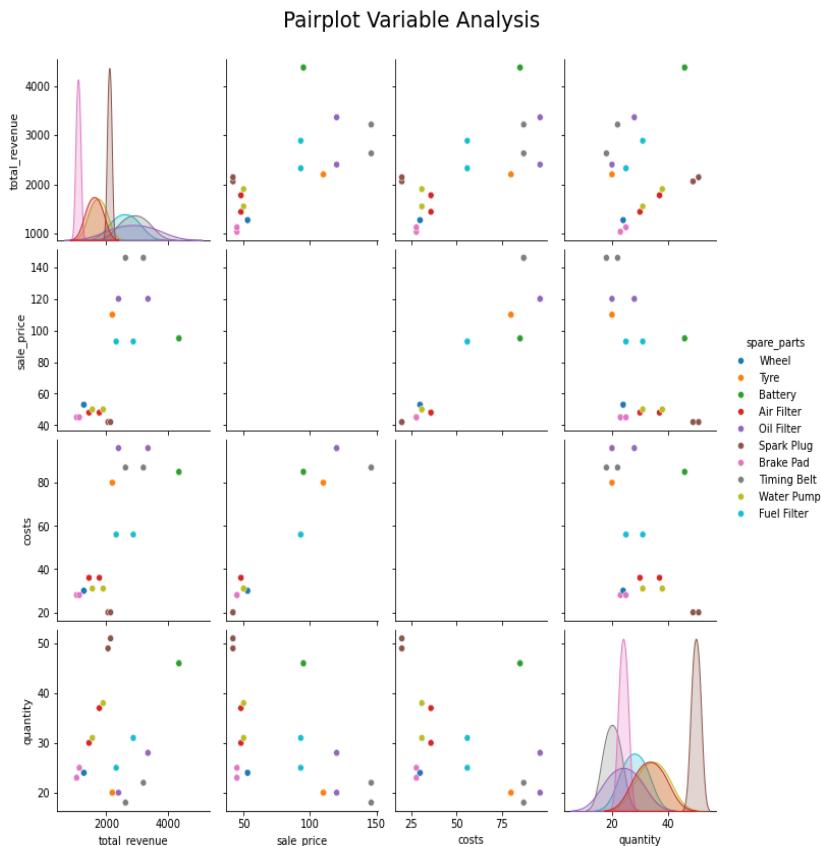
# Add Labels to the axes
ax.set_xlabel('sale_price')
ax.set_ylabel('quantity')
ax.set_zlabel('costs')
plt.title("Price vs Quantity vs Costs", fontsize= 20)
plt.show()
```

Price vs Quantity vs Costs



7. A pairplot is powerful because we can use it to visualize the relationships among variables in the dataset. We are going to pass the following variables "total_revenue", "sale_price", "costs", "quantity", and "spare_parts" columns to the Seaborn pairplot function. Here is the code and output:

```
[15]: sns.pairplot(df[['total_revenue', 'sale_price', 'costs', 'quantity', 'spare_parts']],
                  hue="spare_parts")
plt.title("Pairplot Variable Analysis", y=4.3, x=-1, fontsize=20)
plt.show()
```

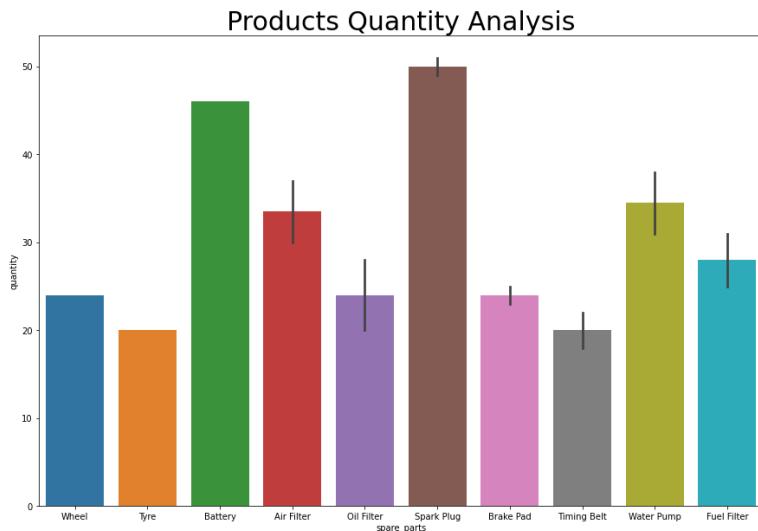


8. A Seaborn bar plot is a way to represent categorical data using bars. We are going to create a bar plot for the "spare parts" and "quantity" columns. This will help to visualize the quantity sold per product. Here is the code below:

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

fig = plt.figure(figsize=(15, 10))
sns.barplot(x="spare_parts",y = "quantity",
            hue="spare_parts", data = df)
plt.title("Products Quantity Analysis", fontsize = 30)
plt.show()
```

This will output the following plot:

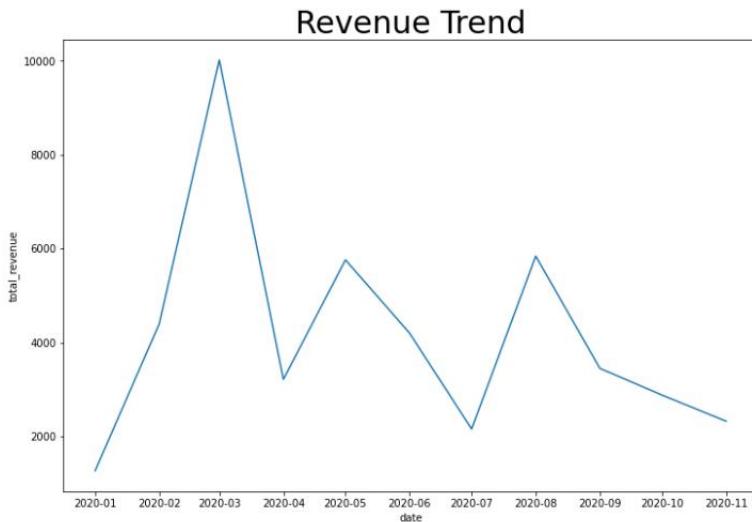


9. We are going to create a line plot that shows the revenue over a period of time. The **x-axis** represents the date, and the **y-axis** represents the total sales. Here is the code and plot below:

```
[17]: # Convert the date to datetime format
df["date"] = pd.to_datetime(df["date"])

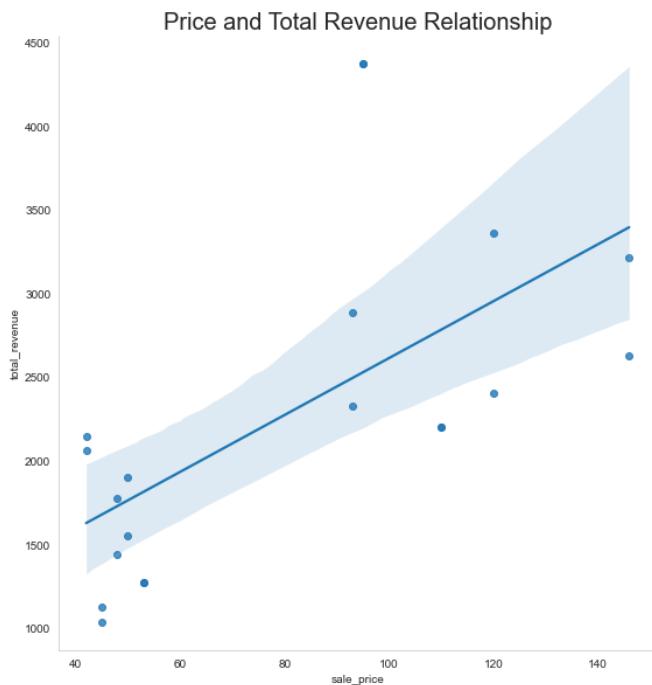
# Group the revenue by date
df_group = df.groupby("date")["total_revenue"].sum()

fig = plt.figure(figsize=(12, 8))
sns.lineplot(x = df_group.index, y = df_group)
plt.title("Revenue Trend", fontsize = 30)
plt.show()
```



10. The **sns.lmplot()** is a function in the Seaborn library that creates a scatter plot with a linear regression line fit to the data. It is used to visualize the relationship between two continuous variables. We want to visualize the relationship between the price of the items and total revenue.

```
[18]: sns.lmplot(x = "sale_price",
                 y = "total_revenue",
                 fit_reg=True,
                 height = 8,
                 data=df)
sns.set_style("whitegrid", {'axes.grid' : False})
plt.title("Price and Total Revenue Relationship", fontsize = 20)
plt.show()
```



This plot shows the relationship between `sale_price` and `total_revenue`, and from the upward sloping line, we can conclude that there is a positive correlation between these two variables. This means that as the sale price increases, the total revenue tends to increase as well. However, Some points deviate significantly from the line, indicating that while the general trend is positive, the relationship between sale price and total revenue may not be perfectly linear or there could be other influencing factors.

Day 41: Population Data Analysis

In this challenge, you are going to gain insights and preprocess fictitious population data. You will import the **population_data** file, which is saved in CSV format.

	A	B	C	D
1	Country	Population	Urban Population	Rural Population
2	China	144616277	62233489.02	82382787.98
3	India	161802610	93703729.99	68098880.01
4	United States	199096104	89266811.49	109829292.5
5	Indonesia	192423285	58981078.45	133442206.6

1. Import the dataset using pandas. View the first 5 columns. What is the data type of each column in the dataset? What is the shape of the DataFrame?
2. Are there any missing values in the dataset? If so, how many and in which columns?
3. Are there any duplicate rows in the "Country" column?
4. How many unique countries are there in the "Country" column?
5. What is the range of the "**Population**" column values in the dataset?
6. What are the **mean** and **median** of the "**Population**" column values?
7. Import the population dataset. What is the total population of all the countries in the dataset?
8. What is the average population of all the countries?
9. What is the average urban population of all the countries?
10. Which country has the biggest population?
11. Which country has the smallest urban population?
12. Using Matplotlib, plot a histogram of the population in the dataset. The number of bins for your plot must be 10. Your plot must have **axis** labels and a title. Your plot title's font size must be 20. Make your title bold.

Day 41 - Answers

1. We are going to import the **population_data** dataset using pandas **read_csv()** function.

```
[1]: import pandas as pd  
  
df = pd.read_csv("Population_data.csv")  
df.head()
```

	Country	Population	Urban Population	Rural Population
0	China	144616277	62233489.02	8.238279e+07
1	India	161802610	93703729.99	6.809888e+07
2	United States	199096104	89266811.49	1.098293e+08
3	Indonesia	192423285	58981078.45	1.334422e+08
4	Pakistan	122038765	78004077.98	4.403469e+07

We will use the **dtypes** attribute to view the data types of the columns in our dataset.

```
[2]: # Viewing the data types  
df.dtypes
```

[2]:	Country object
	Population int64
	Urban Population float64
	Rural Population float64
	dtype: object

We can use the **shape** attribute to check the shape of our DataFrame. The shape method will return a tuple of the number of rows and columns in the DataFrame.

```
[3]: # Checking the shape of the DataFrame  
df.shape
```

```
[3]: (20, 4)
```

We have 20 rows and 4 columns in the DataFrame.

2. The pandas `isnull()` method checks for missing values in a DataFrame. It returns a Boolean value of True if it finds a missing value in the column and False if there is one. We are going to use this with the `sum()` method. The `sum()` method will sum all True values in a column, if there are any.

```
[4]: # Checking for missing values  
df.isnull().sum()
```

```
[4]: Country      0  
Population      0  
Urban Population 0  
Rural Population 0  
dtype: int64
```

3. We can use the `duplicated()` method to check for duplicates in a column. The method returns a Boolean value of True if a value is duplicated. Since we want to check for duplicates in a specific column, we are going to pass the "Country" column as the argument to the subset parameter.

```
[5]: # Checking for duplicates in a column  
df.duplicated(subset=["Country"]).sum()
```

```
[5]: 0
```

There are zero duplicated rows in the column.

4. To find the number of unique values in a column, we can use the `pandas.Series.unique()` method.

```
[6]: # Find the number of unique values in a column  
print(f'The number of unique countries is {len(df.Country.unique())}')  
The number of unique countries is 20
```

5. To find the range of values in the column, we have to find the difference between the `max()` value and the `min()` value.

```
[7]: column_name = 'Population'

# Find the difference between the max value and min value
column_range = df[column_name].max() - df[column_name].min()
print(f'{column_range:,}')
```

159,664,488

6. Let's use the `mean()` and `median()` methods on the "population" column to calculate the **mean** and **median**.

```
[8]: # Finding the median of the population
median = df["Population"].median()
print(f"Median of data: {median:,}")

Median of data: 141,602,917.5
```

```
[9]: # Finding the mean of the population
mean = df["Population"].mean()
print(f"Mean of the data: {mean:,}")

Mean of the data: 127,368,421.75
```

7. To find the total population of all countries in the dataset, we sum up the "Population" column using the `sum()` method.

```
[10]: total_population = df["Population"].sum()
print(f'Total population is: {total_population:,}')

Total population is: 2,547,368,435
```

8. There are two ways we can find the average population of the countries: We can divide the total population by the number of countries

```
[11]: ave_population = (df["Population"].sum())/len(df["Country"])
print(f'Average population per country is: {ave_population:,}')

Average population per country is: 127,368,421.75
```

Another easy way is to use the `mean()` method.

```
In [12]: ave_population = df["Population"].mean()
print(f'Average population per country is: {ave_population:.2f}')

Average population per country is: 127,368,421.75
```

9. One way we can find the average urban population is by dividing the sum of the "Urban Population" column by the number of countries.

```
[13]: ave_urb_population = (df["Urban Population"].sum())/len(df["Country"])
print(f'Average urban population per country is : {ave_urb_population:.2f}')

Average urban population per country is : 56,629,809.43
```

We can also use the pandas `mean()` method to find the average of the column.

```
[14]: print(f'{df["Urban Population"].mean():,.2f}')

56,629,809.43
```

10. To find the country with the biggest population in the DataFrame we can use the `idxmax()` method on the "Population" column. This method will return the index of the country with the largest population. We will use the index to retrieve the country with the largest population.

```
[15]: # Finding the index of the country with biggest population
max_index = df["Population"].idxmax()

# Using index to find the country with the biggest population
max_population_country = df["Country"][max_index]
print(f'The country with the largest populations is: {max_population_country}')

The country with the largest populations is: United States
```

11. Now, to find the country with the smallest urban population, we can use the `idxmin()` method on the "Urban Population" column. This will return the index of the smallest number in the column. We will use the index to return the country name with the smallest population.

```
[16]: # Finding the index of the country with smallest urban population
min_index = df["Urban Population"].idxmin()

# Using index to find the country with the smallest urban population
min_urban_pop_country = df["Country"][min_index]
print(f'The country with the smallest urban populations is: {min_urban_pop_country}')

The country with the smallest urban populations is: Turkey
```

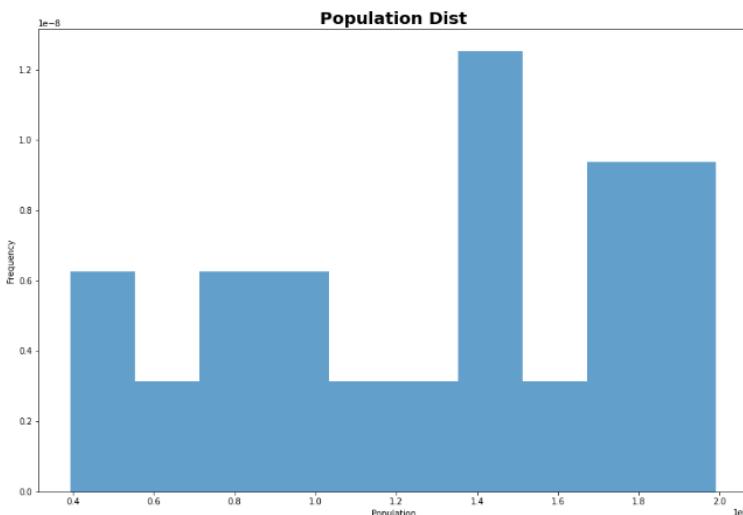
12. We are going to use Matplotlib to plot a histogram that will display the population distribution in the dataset.

```
[17]: import matplotlib.pyplot as plt

# Plot the distribution of the population values
plt.subplots(figsize=(15,10))

plt.hist(df['Population'],
         bins=10,
         density=1,
         alpha = 0.7,
         edgecolor=None)

plt.xlabel('Population')
plt.ylabel('Frequency')
plt.title('Population Dist', fontsize=20,
          fontweight= "bold")
plt.show()
```



Day 42: Toys Data Analysis

For this challenge, you are going to preprocess, analyze, and create visualizations of the `toys_sales_data` dataset. Here is a snippet of the dataset below:

	A	B	C	D
1	Date	Item	Quantity	Total Sales
2	1 01 2022	Stuffed animal	50	\$250
3	4 01 2022	Lego set	25	\$500
4	7 01 2022	Board game	30	\$450
5	13 01 2022	Doll	40	\$400
6	21 01 2022	Stuffed animal	40	\$200
7	24 01 2022	Lego set	35	\$600
8	31 01 2022	Board game	20	\$250

1. Load the CSV file using pandas. Check the data types of the "Date" and "Total Sales" columns.
2. Using Matplotlib, plot a bar plot of the sales value of each item. Your plot should have `axis` labels and a `title`. The plot size should be 10 inches by 8 inches.
3. Using pandas and Matplotlib, plot a pie chart showing the percentage of total sales for each item. Your plot should have a title.
4. Using Matplotlib subplots, plot both a `bar` plot and a `line` plot showing the total sales for each item. Your subplot must have one column and two rows. The line plot must be on top.
5. Use Seaborn to create a `scatter` plot showing the relationship between items and the quantity of each item.

Day 42 - Answers

1. First, let's import the `toys_sales_data` dataset and check the data types.

```
[1]: import pandas as pd  
  
df = pd.read_csv("toy_sales_data.csv")  
df.head()
```

```
[1]:          Date      Item  Quantity  Total Sales  
0   1 01 2022  Stuffed animal      50        $250  
1   4 01 2022       Lego set      25        $500  
2   7 01 2022     Board game      30        $450  
3  13 01 2022         Doll      40        $400  
4  21 01 2022  Stuffed animal      40        $200
```

Now let's check the data types of the "Date" and "Total Sales" columns:

```
[2]: df[["Date", "Total Sales"]].dtypes  
  
[2]: Date      object  
      Total Sales    object  
      dtype: object
```

2. Now, notice that our total sales column is of the object data type. This means that we will be unable to carry out calculations on this column unless we convert it to a numeric data type. You may also have noticed that our column has "\$" next to the numbers; this must be removed before we can carry out any calculations.

Before we can plot this data on a plot, we will remove the "\$" from the total sales column and convert the column to a numeric data type. We will then group the DataFrame by the items and sum the total sales column. We will use this data to create a plot.

```
[3]: # converting the total sales column to numeric datatype and droping "$"
df["Total Sales"] = df["Total Sales"].str.replace("$","").astype(int)
df.dtypes
```

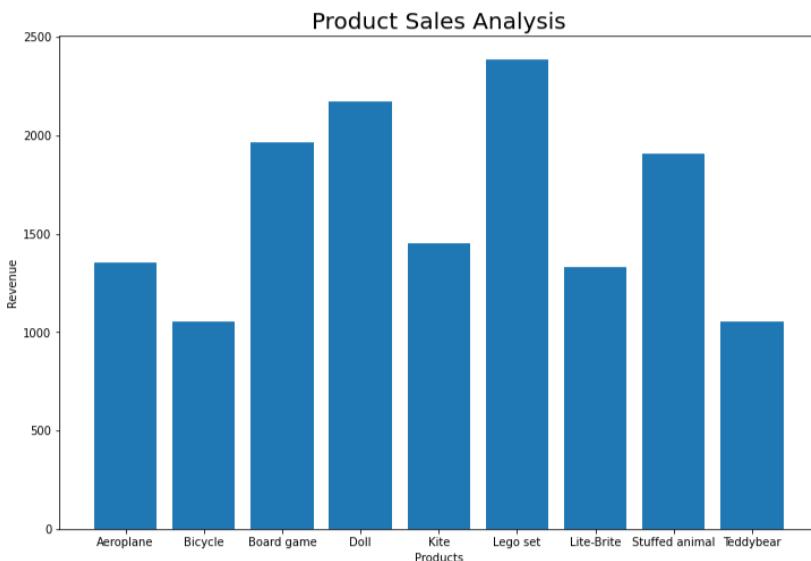
```
[3]: Date        object
      Item       object
      Quantity    int64
      Total Sales int32
      dtype: object
```

You can see above that the "Total Sales" column has been converted to an integer data type. We can now group the data and create a plot.

```
[4]: import matplotlib.pyplot as plt

# Group dataframe by items
grouped_sales = df.groupby("Item")["Total Sales"].sum()

# Plot the bar chart
plt.figure(figsize=(12,8))
plt.bar(grouped_sales.index, grouped_sales)
plt.xlabel("Products")
plt.ylabel("Revenue")
plt.title("Product Sales Analysis", fontsize=20)
plt.show()
```

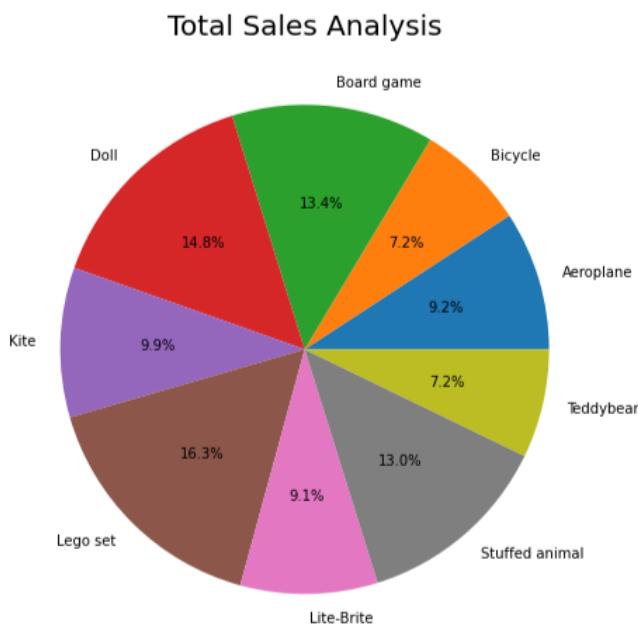


3. This challenge requires that we plot a pie plot showing the percentage of total sales for each item using Matplotlib. We will pass in the values for each item and specify the `autopct` parameter to show the percentages. First, we have to group the data by item and sum the total sales. Here is the code below:

```
[5]: # Group dataframe by items
grouped_sales = df.groupby("Item")["Total Sales"].sum()

# Plot the pie chart using matplotlib
plt.figure(figsize=(10,8))
plt.pie(grouped_sales, labels=grouped_sales.index, autopct='%1.1f%%')
plt.title("Total Sales Analysis", fontsize = 20)
plt.show()
```

This will output the following:



4. To create subplots of both bar and line plots, we can use the `subplots()` method of Matplotlib. We will specify the

number of rows and columns for the subplots. We will use the `bar()` method to create a bar plot on the first axis and the `plot()` method to create a line plot on the second axis. It is important to first group the data before plotting. So, we will first group the data using the `groupby()` method.

```
[6]: # Group the data
grouped_sales = df.groupby("Item")["Total Sales"].sum()

# Creating the subplots
fig, ax = plt.subplots(nrows=2, ncols=1, figsize=(15, 10))
# Plot the bar chart on the first axes
ax[1].bar(grouped_sales.index, grouped_sales)
# Plot the line chart on the second axes
ax[0].plot(grouped_sales.index, grouped_sales)
plt.xlabel("Products")
plt.ylabel("Revenue")
fig.suptitle("Sales Analysis", fontsize = 20)
plt.xticks(rotation = 90)
plt.show()
```

This will output:

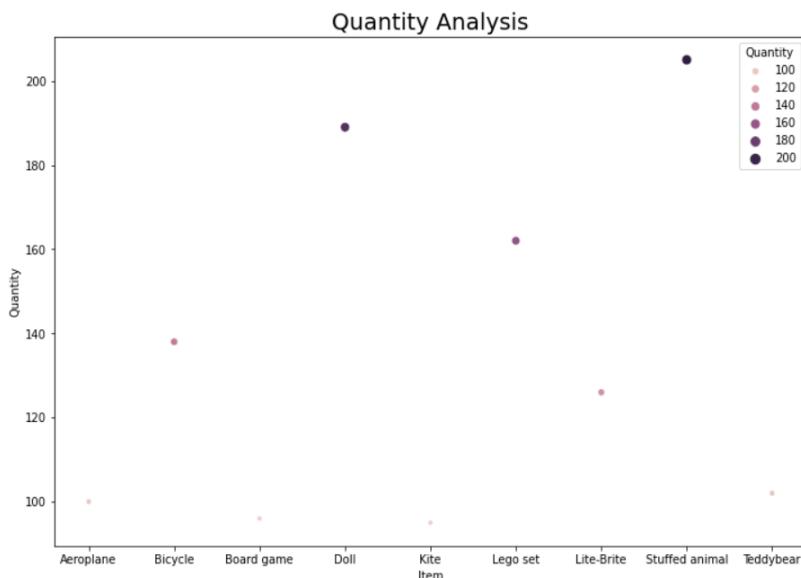


5. In this challenge, we are required to create a scatter plot showing the relationship between each item and the quantity sold. First, we will group the data by each item and sum the quantity. We will import Seaborn and use the `scatterplot()` function to create the plot. See the code below:

```
[7]: import seaborn as sns

# Group the sales data by items and sum quantity
grouped_sales = df.groupby("Item")["Quantity"].sum()

plt.figure(figsize=(12, 8))
sns.scatterplot(
    x=grouped_sales.index,
    y=grouped_sales, size = grouped_sales,
    hue = grouped_sales)
plt.title("Quantity Analysis", fontsize= 20)
plt.show()
```



Day 43: Time Series Data Analysis

For this challenge, you are going to explore the time series data. These challenges will teach you how to manipulate and visualize time series analysis using Python. You will import the `time_series_data` CSV file. Here is a sample of the data below:

	A	B
1	date	value
2	2022-01-01 00:00:00	0.897366
3	2022-01-01 01:00:00	0.389095
4	2022-01-01 02:00:00	0.849054
5	2022-01-01 03:00:00	0.526571
6	2022-01-01 04:00:00	0.403405

1. Resample the dataset at daily intervals and calculate the maximum value for each day.
2. Create a **line** plot of the dataset, where the **x-axis** is the date and time and the **y-axis** is the value.
3. Create a **bar** plot of the dataset, where the **x-axis** is the hour of the day (0–23), and the **y-axis** is the **mean** value for each hour.
4. The rolling average is a widely used statistical tool that smooths out short-term fluctuations in data and shows the longer-term trend of the data. Calculate the rolling average of the dataset using a window size of 3, and plot the original values and the rolling average on the same plot.

Day 43 - Answers

1. First we are going to import the **time_series_data** dataset using pandas.

```
[1]: import pandas as pd  
  
df = pd.read_csv("time_series_data.csv")  
df.head()
```

```
[1]:          date      value  
0 2022-01-01 00:00:00  0.897366  
1 2022-01-01 01:00:00  0.389095  
2 2022-01-01 02:00:00  0.849054  
3 2022-01-01 03:00:00  0.526571  
4 2022-01-01 04:00:00  0.403405
```

If we check the DataFrame data types, you will notice that the date column is of the "object" data type.

```
[2]: # Checking datatypes  
df.dtypes
```

```
[2]: date      object  
value     float64  
dtype: object
```

We have to convert the date to pandas datetime format and make the date the index. First, we will make a copy of the DataFrame.

```
[3]: # Create a copy of the DataFrame  
df_copy = df.copy()
```

Now, we convert the "date" column to datetime format and set it as an index.

```
[4]: # Convert date to datetime format  
df_copy['date'] = pd.to_datetime(df_copy.date)  
# Set date as index  
df_copy.set_index('date', inplace=True)
```

We are now going to use the `sample()` method to sample the DataFrame by "day" and return the `max()` value.

```
[5]: day_max = df_copy.resample('D').max()  
day_max
```

```
[5]:          value  
date  
-----  
2022-01-01  0.966448  
2022-01-02  0.922851
```

2. To create a line plot using Matplotlib, we will pass the index as the **x-axis** and the values as the **y-axis**.

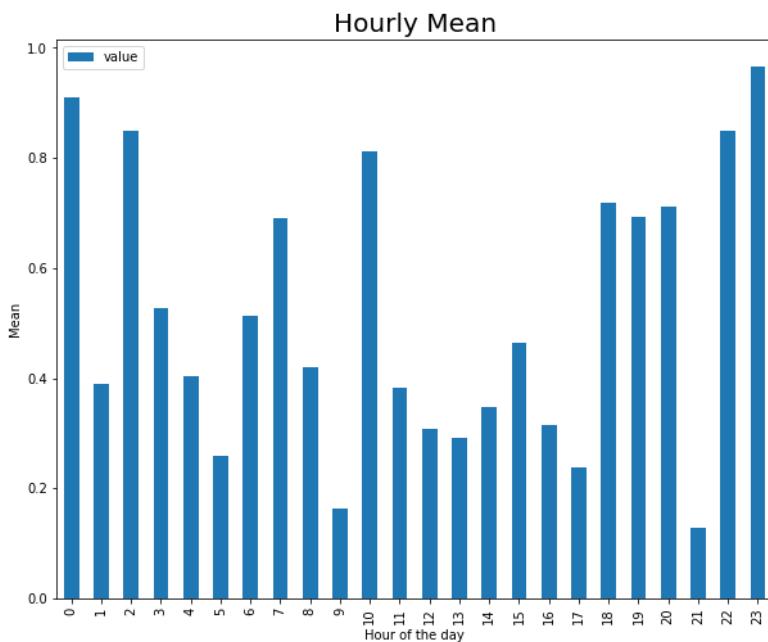
```
[6]: import matplotlib.pyplot as plt  
  
plt.figure(figsize=(10, 8))  
plt.plot(df_copy.index, df_copy['value'])  
plt.xlabel('Date')  
plt.ylabel('Value')  
plt.title("Price Movement", fontsize = 20)  
plt.show()
```



3. First, we use the **groupby()** method to group data by index and calculate the **mean** of each hour. We will use the pandas **plot()** method and Matplotlib to plot the bar plot.

```
[7]: # Grouping data by index and calculating mean
hour_mean = df_copy.groupby(df_copy.index.hour).mean()

hour_mean.plot(kind='bar', figsize=(10, 8))
plt.xlabel('Hour of the day')
plt.ylabel('Mean')
plt.title("Hourly Mean", fontsize = 20)
plt.show()
```

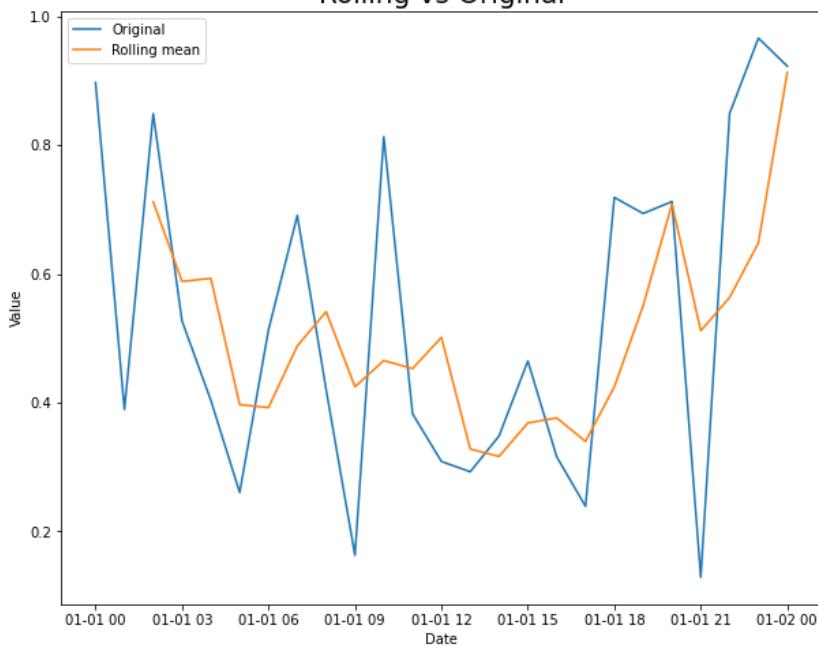


4. The **rolling()** method creates a rolling window of a specified size, and the **mean()** method calculates the mean value of the values in each window. The significance of the rolling average is that it can help filter out noise in a dataset, making it easier to see the underlying trend. We are going to calculate the rolling average, and we will set the window to 3. We will then plot the original values and the rolling average values on the same line plot.

```
[8]: rolling_mean = df_copy.rolling(window=3).mean()

plt.figure(figsize=(10, 8))
plt.plot(df_copy.index, df_copy['value'], label='Original')
plt.plot(rolling_mean.index, rolling_mean['value'], label='Rolling mean')
plt.xlabel('Date')
plt.ylabel('Value')
plt.title("Rolling vs Original", fontsize = 20)
plt.legend()
plt.show()
```

Rolling vs Original



Day 44: Sports Data Analysis

For this challenge, you are going to use the `sports_data` dataset. The file is a CSV file. Here is a sample of the dataset below:

	A	B	C	D	E	F	G	H	I
1	Name	Sport	Goals	Assists	Fouls	Minutes P	Yellow	Red Cards	Team
2	Alex	Basketbal	4	5	8	36	1	0	A
3	Bob	Soccer	0	2	1	31	0	0	B
4	Charlie	Basketbal	2	2	6	34	1	0	A
5	David	Soccer	1	7	6	9	0	0	B
6	Eve	Basketbal	8	2	6	29	0	0	A

1. Import the sports dataset and write code to check the first five rows to confirm that it has loaded properly. Now, using the pandas `items()` method, return all the names of the columns that have values of the "object" data type.
2. Using a pandas method, check the memory size of each column using the `df.memory_usage()` method. Set the deep parameter to True. What is the total memory usage of the DataFrame in kilobytes?
3. Using pandas convert all the columns with the "object" data type to the "categorical" data type. Create a new variable for the data frame after the conversation. Check the memory size of the resulting DataFrame using the `df.memory_usage()` method. Compare your results to the results in question 2. What conclusion can you draw?
4. Using the pandas method, rename the "Yellow" column to "Yellow Cards."
5. Using the pandas `query()` method, create a subset DataFrame of only players that got over six fouls. Reset your index (set it back to the default 0, 1, 2, etc.) and ensure that you remove the index as a column.
6. Save the new DataFrame as a CSV file. Give it a name of your choice.

Day 44 - Answers

1. We are going to import the **sports_data** dataset using pandas and view the first 5 rows.

```
[1]: import pandas as pd  
  
df = pd.read_csv("sports_data.csv")  
df.head()
```

	Name	Sport	Goals	Assists	Fouls	Minutes Played	Yellow	Red	Cards	Team
0	Alex	Basketball	4	5	8	36	1	0	0	A
1	Bob	Soccer	0	2	1	31	0	0	0	B
2	Charlie	Basketball	2	2	6	34	1	0	0	A
3	David	Soccer	1	7	6	9	0	0	0	B
4	Eve	Basketball	8	2	6	29	0	0	0	A

We are going to use a **for loop** statement with the pandas **items()** method (formerly known as the **iteritems()** method). This method returns a tuple of the column name and its content. We are going to use this method to loop through the DataFrame and return the names of all columns of the "object" data type.

```
[2]: columns_names = []  
  
# Looping over the DataFrame  
for (columns, values) in df.items():  
    if values.dtype == 'object':  
        columns_names.append(columns)  
  
print(columns_names)
```

['Name', 'Sport', 'Team']

This returns the "Names," "Sport," and "Team" columns.

2. The **df.memory_usage()** method returns the memory usage of each column in bytes. We set the deep parameter to True to get a more accurate view of the memory usage.

```
[3]: df.memory_usage(deep =True)
```

```
[3]: Index      128
Name       1235
Sport      1300
Goals      160
Assists    160
Fouls      160
Minutes Played 160
Yellow     160
Red Cards   160
Team       1160
dtype: int64
```

To get the total memory usage, we can sum the columns. Since the method returns the usage in bytes, we will divide the total by 1024 to get the usage in kb.

```
[4]: df.memory_usage(deep = True).sum()/1024
```

```
[4]: 4.6708984375
```

This puts the memory usage of the DataFrame at 4.67kb

3. We are going to use the `astype()` method to convert the columns of the "object" data type to the "category" data type.

```
[5]: # Changing the data type from object to category
df2 = df.astype({"Name": "category", "Sport": "category", "Team": "category"})
df2.dtypes
```

```
[5]: Name      category
Sport     category
Goals     int64
Assists   int64
Fouls     int64
Minutes Played  int64
Yellow    int64
Red Cards int64
Team      category
dtype: object
```

Now, let's check the memory being consumed by the DataFrame using the `df.memory_usage()` method.

```
[6]: df2.memory_usage(deep = True)
```

```
[6]: Index          128
      Name         1811
      Sport         258
      Goals         160
      Assists        160
      Fouls          160
      Minutes Played 160
      Yellow         160
      Red Cards      160
      Team           244
      dtype: int64
```

You can see below that the total memory usage has decreased from 4.67 to 3.32. We can conclude that the "category" data type is good for saving memory.

```
[7]: df2.memory_usage(deep = True).sum()/1024
```

```
[7]: 3.3212890625
```

4. We are going to use the `rename()` method to rename the column.

```
[8]: # Renaming the column from 'Yellow' to 'Yellow Cards'
df2 = df2.rename(columns={'Yellow':'Yellow Cards'})
df2.head()
```

```
[8]:   Name      Sport  Goals  Assists  Fouls  Minutes Played  Yellow Cards  Red Cards  Team
    0  Alex  Basketball    4       5       8          36            1            0            A
    1  Bob   Soccer      0       2       1          31            0            0            B
    2 Charlie Basketball   2       2       6          34            1            0            A
    3 David  Soccer      1       7       6             9            0            0            B
    4 Eve   Basketball   8       2       6          29            0            0            A
```

You can see that the column has been renamed to "Yellow Cards."

5. Using the `query()` method, we can query the rows of the DataFrame that meet a certain criteria. We want to return names that have committed over six fouls. We are going to query the "Fouls" column. We are going to create a new DataFrame.

```
[9]: # Using query to filter data by fouls
df3 = df2.query("Fouls > 6").reset_index(drop=True)
df3
```

	Name	Sport	Goals	Assists	Fouls	Minutes Played	Yellow Cards	Red Cards	Team
0	Alex	Basketball	4	5	8	36	1	0	A
1	George	Basketball	5	8	8	3	1	0	A
2	Harry	Soccer	9	6	8	13	1	0	B
3	Ivan	Basketball	7	9	8	30	1	0	A
4	Katie	Basketball	7	8	9	6	0	0	A
5	Nate	Soccer	0	0	7	44	1	0	B
6	Olivia	Basketball	3	0	8	5	1	0	A

6. To save the DataFrame in CSV format, we can use the `to_csv()` method. The method takes the name and the index. Here is the code below:

```
[10]: df3.to_csv("sports_data_modified.csv", index=False)
```

We set the `index` parameter to False so that an addition index is not added to the data.

Day 45: Medical Data Analysis

For this challenge, you are going to preprocess and analyze the medical data of patients. The name of the file is **medical_data**, and it is in CSV format.

	A	B	C	D	E	F
1	First Name	First Letter	Last Name	Condition	Gender	BMI
2	John	J	Smith	Healthy	Female 27	27.35
3	Mary	M	Jones	Cancer	Female 39	24.29
4	Amy	A	Candy	Heart Disease	Male 51	28.72
5	David	D	Tuss	Diabetes	Female 73	24.66

1. Import the **medical_data** dataset and view the first 5 rows. Use the **info()** method to get insights in the data.
2. What are the IQR ranges of the "BMI" column?
3. You must have noticed that there are two columns with names. As part of feature engineering, you are required to create a copy of your DataFrame (question 1). Combine the "First Name" column and the "Last Name" column into one column called "Names."
4. Now that you have combined the two columns, drop columns "First Name" and "Last Name" from the DataFrame and change the order of the columns to: **[Names, First Letter, BMI, Gender, Condition]**.
5. The "Gender" column is combined with the age. Your task is to create a new column for age by separating age from the "Gender" column. Once you do that, make the "Condition" column the last column of the DataFrame. Your columns' order must be: **[Name, First Letter, BMI, Gender, Age, Condition]**.

Day 45 - Answers

1. We are going to import the dataset using the pandas `read_csv()` function.

```
[1]: import pandas as pd  
  
df = pd.read_csv("medical_data.csv")  
df.head()
```

	First Name	First Letter	Last Name	Condition	Gender	BMI
0	John	J	Smith	Healthy	Female	27 27.35
1	Mary	M	Jones	Cancer	Female	39 24.29
2	Amy	A	Candy	Heart Disease	Male	51 28.72
3	David	D	Tuss	Diabetes	Female	73 24.66
4	Michael	M	Brookes	Asthma	Male	71 28.21

Now let's use the `info()` method to get some insights into our data. The `info()` method will tell us how many columns and rows are in the dataset. It will also check for null values in the dataset and the data types of the columns.

```
[2]: df.info()  
  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 10 entries, 0 to 9  
Data columns (total 6 columns):  
 #   Column      Non-Null Count  Dtype     
 ---  --          --          --  
 0   First Name  10 non-null    object    
 1   First Letter 10 non-null    object    
 2   Last Name   10 non-null    object    
 3   Condition   10 non-null    object    
 4   Gender      10 non-null    object    
 5   BMI         10 non-null    float64  
dtypes: float64(1), object(5)  
memory usage: 608.0+ bytes
```

2. To calculate the IQR ranges, we are going to use the `df.quantile()` method. The lower range is the 1st quarter (0.25), and the upper limit is the 3rd quarter (0.75).

```
[3]: lower_limit = df['BMI'].quantile(q = 0.25)
      upper_limit = df['BMI'].quantile(q = 0.75)

      print(f'The IQR range is {lower_limit:.2f} to {upper_limit:.2f}')
The IQR range is 23.71 to 27.66
```

3. In this challenge, we are required to create a copy of the DataFrame and combine the two columns, "First Name" and "Last Name," into one column. We are going to use the `pandas.str.cat` method to concatenate the two columns. This is because the two columns are of the str data type.

```
[4]: df_copy = df.copy()

# concatenate the two columns into one column
df_copy['Name'] = df_copy["First Name"].str.cat(df_copy["Last Name"], sep='')
df_copy.head()
```

	First Name	First Letter	Last Name	Condition	Gender	BMI	Name
0	John	J	Smith	Healthy	Female	27	27.35 John Smith
1	Mary	M	Jones	Cancer	Female	39	24.29 Mary Jones
2	Amy	A	Candy	Heart Disease	Male	51	28.72 Amy Candy
3	David	D	Tuss	Diabetes	Female	73	24.66 David Tuss
4	Michael	M	Brookes	Asthma	Male	71	28.21 Michael Brookes

You can see that the "Name" column has been added to the DataFrame.

4. In this question, we are required to drop the "First Name" and "Last Name" columns from the DataFrame. We are going to use the `pandas.drop()` method to remove the columns and the `.iloc` attribute to change the column order in the DataFrame. Here is the code below:

```
[5]: # Drop the two columns
df_copy.drop(columns=["First Name","Last Name"], axis=1, inplace=True)

# using iloc to change columns order
df_copy = df_copy.iloc[:, [4, 0, 3, 2, 1]]
df_copy.head()
```

	Name	First Letter	BMI	Gender	Condition
0	John Smith	J	27.35	Female 27	Healthy
1	Mary Jones	M	24.29	Female 39	Cancer
2	Amy Candy	A	28.72	Male 51	Heart Disease
3	David Tuss	D	24.66	Female 73	Diabetes
4	Michael Brookes	M	28.21	Male 71	Asthma

5. Now, let's split the gender column into two columns and rearrange the order of the columns. We are going to use the `pandas.str.split()` method and the `.iloc` attribute to change the columns' order. Here is the code below:

```
[6]: # Creating two columns from the Gender column using the split method
df_copy[['Gender', 'Age']] = df_copy.Gender.str.split(" ", expand=True)

# using iloc to change columns order
df_copy = df_copy.iloc[:, [0, 1, 2, 3, 5, 4]]
df_copy.tail()
```

	Name	First Letter	BMI	Gender	Age	Condition
5	Jessica kil	J	27.76	Male	61	Diabetes
6	Mike Storey	M	23.51	Female	28	Asthma
7	Sarah Mary	S	26.78	Male	39	Heart Disease
8	Chris Derick	C	21.50	Male	61	Healthy
9	Emily Cat	E	21.62	Male	61	Asthma

Day 46: Financial Data Analysis

For this challenge, you are going to analyze the **financial_data** CSV file. You will import the file using pandas and answer the questions below.

	A	B	C	D
1	Date	Revenue	Expenses	Profit
2	31 01 2022	19000	5000	14000
3	28 02 2022	16000	5500	10500
4	31 03 2022	10000	7000	3000
5	30 04 2022	18000	8000	10000
6	31 05 2022	25000	9000	16000

1. What is the average monthly revenue, expenses, and profit?
2. How does the profit margin vary over time? Plot a line plot of the changes in profit margin.
3. What is the difference between the revenue of the highest month and that of the lowest month? Which month has the highest revenue, and which month has the lowest revenue?
4. Which months had expenses above \$10,000?
5. How much profit was made in the last quarter of 2022?
6. What was the percentage change in revenue from quarter to quarter?

Day 46 - Answers

- First, we are going to load the **financial_data** dataset using pandas and view the first five rows to ensure that it has loaded correctly.

```
[1]: import pandas as pd  
  
df = pd.read_csv("financial_data.csv")  
df.head()  
  
[1]:
```

	Date	Revenue	Expenses	Profit
0	31 01 2022	19000	5000	14000
1	28 02 2022	16000	5500	10500
2	31 03 2022	10000	7000	3000
3	30 04 2022	18000	8000	10000
4	31 05 2022	25000	9000	16000

To calculate the average revenue, expenses, and profit, we are going to calculate the **mean** of each column.

```
[2]: # calculate the average monthly revenue, expenses, and profit  
avg_revenue = df['Revenue'].mean()  
avg_expenses = df['Expenses'].mean()  
avg_profit = df['Profit'].mean()  
  
print(f"Average monthly revenue: {avg_revenue:.2f}")  
print(f"Average monthly expenses: {avg_expenses:.2f}")  
print(f"Average monthly profit: {avg_profit:.2f}")  
  
Average monthly revenue: 35200.00  
Average monthly expenses: 15425.00  
Average monthly profit: 19775.00
```

- We are going to add a "Profit Margin" column to the DataFrame. We will calculate the margin by dividing the profit by the revenue. We will then convert the date to datetime format.

```
[3]: # calculate the profit margin  
df['Profit Margin'] = (df['Profit'] / df['Revenue']) * 100  
# Converting date to datetime format  
df["Date"] = pd.to_datetime(df["Date"], dayfirst=True)
```

Now let's plot a line plot using Matplotlib.

```
[4]: import matplotlib.pyplot as plt

# plot the profit margin over time
plt.figure(figsize=(10, 8))
plt.plot(df['Date'], df['Profit Margin'])
plt.xlabel('Date')
plt.ylabel('Profit Margin (%)')
plt.title('Profit Margin Over Time', fontsize = 20)
plt.show()
```



3. We are going to use the `.loc` attribute to find the highest and lowest revenues and calculate the difference.

```
[5]: highest_revenue = df.loc[df['Revenue'] == df['Revenue'].max(), 'Revenue'].values[0]
lowest_revenue = df.loc[df['Revenue'] == df['Revenue'].min(), 'Revenue'].values[0]

difference = highest_revenue - lowest_revenue
print(f'The difference in Revenue is {difference}')

The difference in Revenue is 48000
```

To return the months with the lowest and highest revenues, first we will extract the month from the "Date" column and use the `.loc` attribute to return the months with the lowest and highest revenue, respectively.

```
[6]: # Extract month from Date
df["Month"] = df["Date"].dt.month_name()

highest_month = df.loc[df['Revenue'] == df['Revenue'].max(), 'Month'].values[0]
lowest_month = df.loc[df['Revenue'] == df['Revenue'].min(), 'Month'].values[0]
print(f'The month with highest revenue is: {highest_month}')
print(f'The month with highest revenue is: {lowest_month}')

The month with highest revenue is: August
The month with highest revenue is: March
```

4. We are going to filter the data to return months with expenses above \$10,000.

```
[7]: month_expenses_above_10 = df.loc[df['Expenses'] > 10000, 'Month']
month_expenses_above_10

[7]: 6      July
      7     August
      8   September
      9    October
      10   November
      11  December
      12  January
      13  February
      14    March
      15    April
      16     May
      17    June
      18    July
      19    August
Name: Month, dtype: object
```

5. To answer this question, we have to group data by quarters using the `groupby()` method with a combination of the `dt.to_period()` method and the `sum()` function. Then we will access the revenue for the fourth quarter of 2022 using the `.loc` attribute.

```
[8]: # group the data by quarters
df2 = df.groupby(df["Date"].dt.to_period('Q'))["Revenue"].sum()
df2
```

```
[8]: Date
2022Q1    45000
2022Q2    65000
2022Q3    97000
2022Q4   112000
2023Q1   126000
2023Q2   150000
2023Q3   100000
Freq: Q-DEC, Name: Revenue, dtype: int64
```

6. To find the percentage change from one quarter to the next, we are going to use the `pct_change()` function.

```
[10]: # calculate the percentage change in revenue from quarter to quarter
quarterly_revenue_change = df2.pct_change()

print("Percentage change in revenue from quarter to quarter:")
print(f'{quarterly_revenue_change:}')
```

```
Percentage change in revenue from quarter to quarter:
Date
2022Q1      NaN
2022Q2     0.444444
2022Q3     0.492308
2022Q4     0.154639
2023Q1     0.125000
2023Q2     0.190476
2023Q3    -0.273333
Freq: Q-DEC, Name: Revenue, dtype: float64
```

Day 47: Text Data Preprocessing

In this challenge, you are going to work with text data. You will import the **Text_data** CSV file. Here is a sample of the data below:

	A	B	C	D	E	F	G	H
1	text							
2	The AI revolution is, here.							
3	seaborn and matplotlib are visualization tools.							
4	Pandas is a powerful% data analysis library.							
5	Data analysis is a crucial part of machine learning.							
6	Python is a popular, programming language for data science.							
7	Natural Language Processing (NLP) is a subfield of AI.							
8	Machine learning is changing the world.							

1. Import the **Text_data** dataset. Write a code that checks if there are duplicates in the "text" column.
2. Create a copy of the DataFrame and convert the text in the "text" column to lowercase letters.
3. Any character that is not a number or in the alphabet is considered a special character. Remove special characters from the "text" column.
4. Write a code that adds a column for the length of each row in the text column. Which row has the longest text?
5. Stopwords are common words that are typically removed from text data before performing natural language processing tasks, such as text classification, sentiment analysis, or information retrieval. Examples of stop words include "the," "and," "a," "an," "in," "of," "to," etc. Write code to remove stop words from the text using the **nltk** module.
6. For text data to be used in machine learning, it must be transformed into numeric vectors. Using Sklearn, tokenize the text into numeric vectors.

Day 47 - answers

1. First, we load the `Text_data` CSV file using pandas and view the first five rows:

```
[1]: import pandas as pd  
  
df = pd.read_csv("Text_data.csv")  
df.head()
```

	text
0	The AI revolution is, here.
1	seaborn and matplotlib are visualization tools.
2	Pandas is a powerful% data analysis library.
3	Data analysis is a crucial part of machine lea...
4	Python is a popular programming language for d...

Now, let's check for duplicates in the "text" column. If there are any duplicates, it will return True. You can see below that our code returns False. This means that we do not have any duplicates in the "Text" column.

```
[2]: #Checking for duplicates in text column  
df.duplicated(subset=['text']).any()
```

```
[2]: False
```

2. To convert the text into lowercase letters, we will use the `str.lower()` method. We will apply this method to the "text" column.

```
[3]: # Creating a copy of the DataFrame  
df_copy = df.copy()  
  
df_copy['text'] = df_copy['text'].str.lower()  
df_copy.head()
```

```
[3]:
```

	text
0	the ai revolution is, here.
1	seaborn and matplotlib are visualization tools.
2	pandas is a powerful% data analysis library.
3	data analysis is a crucial part of machine lea...
4	python is a popular programming language for d...

3. This challenge requires that we remove special characters from the text column. We will make use of the **string** module. The **translate()** method is used to replace specified characters with another set of characters or remove them altogether. In this case, an empty string is passed as the second argument to **maketrans()**, indicating that the characters should be removed rather than replaced. The **string.punctuation** attribute provides a string containing all the punctuation characters.

```
[4]: import string  
  
df_copy['text'] = df_copy['text'].str.translate(str.maketrans('', '', string.punctuation))  
df_copy.head()
```

```
[4]:
```

	text
0	the ai revolution is here
1	seaborn and matplotlib are visualization tools
2	pandas is a powerful data analysis library
3	data analysis is a crucial part of machine lea...
4	python is a popular programming language for d...

4. First, we are going to create a "text length" column for the length of each text in the row. We will then sort the DataFrame by this column in descending order and return the first row.

```
[5]: # Creating a text length column  
df_copy["text length"] = df_copy["text"].str.len()  
df_copy.head()
```

```
[5]:
```

	text	text length
0	the ai revolution is here	25
1	seaborn and matplotlib are visualization tools	46
2	pandas is a powerful data analysis library	42
3	data analysis is a crucial part of machine lea...	51
4	python is a popular programming language for d...	57

We are going to sort the DataFrame by the "text length" column and return the first row. The first row is the longest text.

```
[6]: # Sorting the DataFrame and returning the first row  
df_copy.sort_values(by = ["text length"], ascending=False).iloc[:1]
```

```
[6]:
```

	text	text length
0	big data is becoming increasingly important in...	59

5. The first thing we are going to do is install **nltk module**. You can run the following command to install the library.

```
pip install nltk
```

Stopwords are words such as "is," "and," and so forth. These words are usually less important to the text. When processing texts for machine learning, these words must be removed so that the analysis can concentrate on the important parts of the text. We are going to import "stopwords" from the nltk module and use it to remove stopwords from the text column.

The code below iterates over each word in each text entry and checks if it is present in the set of stopwords. If a word is found in the stopwords set, it is excluded from the resulting text entry. The **lambda** function and list comprehension construct are used to apply this filtering logic to each text entry in the "text" column.

```
[7]: from nltk.corpus import stopwords

# Accessing 'English' stopwords
stop_words = set(stopwords.words('english'))

# Returning words that are not in stopwords
df['text'] = df['text'].apply(lambda x: ' '.join([word for word in x.split() if word not in stop_words]))
df.head()
```

	text
0	The AI revolution is, here.
1	seaborn matplotlib visualization tools.
2	Pandas powerful% data analysis library.
3	Data analysis crucial part machine learning.
4	Python popular programming language data science.

You can see in the output that stopwords have been removed from the text.

6. We are going to use the CountVectorizer from Sklearn to convert the text into numeric vectors. The resulting DataFrame will have the words as columns and the counts of each word in each text as rows. This can be used as input to a machine learning model for further analysis or processing. Below is the output that you see when you process the data.

```
[8]: from sklearn.feature_extraction.text import CountVectorizer

# Instantiating the Vectorizer
cv = CountVectorizer()

# Fitting the text
word_count_matrix = cv.fit_transform(df.text)
# Returning a DataFrame of vectorized text
word_count_df = pd.DataFrame(word_count_matrix.toarray(), columns=cv.get_feature_names_out())
word_count_df.head()
```

	ai	analysis	becoming	big	changing	crucial	data	here	important	increasingly	...	science	seaborn
0	1	0	0	0	0	0	0	1	0	0	...	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	1
2	0	1	0	0	0	0	1	0	0	0	...	0	0
3	0	1	0	0	0	1	1	0	0	0	...	0	0
4	0	0	0	0	0	0	1	0	0	0	...	1	0

Day 48: Preprocess Data with Sklearn

- When training a model on a dataset, it is advisable to split the data into training and test sets. Why do we split data into training and test sets?
- Import the **flowers_dataset**, which is a CSV file.

	A	B	C	D	E	F
1	sepal_len	sepal_wid	petal_len	petal_wid	flower_type	
2	5.1	3.5	1.4	0.2	setosa	
3	4.9	3	1.4	0.2	setosa	
4	4.7	3.2	1.3	0.2	versicolor	
5	4.6	3.1	1.5	0.2	versicolor	
6	5	3.6	1.4	0.2	setosa	

Create a copy of the DataFrame. In supervised machine learning, the **target** variable is the variable that will be predicted using the other variables in the dataset. This target variable must be converted into a numeric data type before fitting a model. Convert this target variable (**flower_type**) into a numeric data type and separate the target column from the other variables. Write code to create two variables, **x** and **y**. **X** is the variable that will predict the target variable, **y**.

Use Sklearn **train_test_split()** function to split the data into training and test sets. Make the test size 20% of the dataset. Set the **random_state** parameter to 42. Check the shapes of the training and test sets. What is the purpose of the **random_state** parameter in the **train_test_split()** function?

- Why is it important to standardize the data before fitting? Use Sklearn **StandardScaler** to standardize the features in the dataset (training and test sets).

Day 48 - Answers

1. The purpose of training a machine learning model is to make predictions on new data, so it is important to assess the model's performance on data that it has not yet seen. We split the data into training and test sets in order to evaluate how well our model can generalize to new, unseen data.

By using a portion of the data to train the model and another portion to test the model, we can get an estimate of how well the model will perform on new, unseen data. The training data is used to fit the parameters of the model, while the test data is used to evaluate the performance of the model on data it has not yet seen.

If we did not split the data and use all of it to train the model, the model could potentially overfit to the training data and perform poorly on new data. By splitting the data, we can ensure that our model is not only memorizing the training data but is also generalizing well to new data.

2. First we import the **flower_dataset** using pandas.

```
[1]: import pandas as pd  
  
df = pd.read_csv("flower_dataset.csv")  
df.head()  
  
[1]:   sepal_length  sepal_width  petal_length  petal_width  flower_type  
      0         5.1        3.5         1.4        0.2    setosa  
      1         4.9        3.0         1.4        0.2    setosa  
      2         4.7        3.2         1.3        0.2  versicolor  
      3         4.6        3.1         1.5        0.2  versicolor  
      4         5.0        3.6         1.4        0.2    setosa
```

Now, we are going to create a copy of the DataFrame. Then we will import LabelEncoder from Sklearn. We will use LabelEncoder to convert the "flower_type" (target column) column into numeric format.

```
[2]: # Creating a copy of the DataFrame  
df_copy = df.copy()
```

```
[3]: from sklearn.preprocessing import LabelEncoder  
  
# Initializing Label encoder  
le = LabelEncoder()  
  
df_copy["flower_type"] = le.fit_transform(df_copy["flower_type"])  
df_copy.head()
```

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

Now, we are going to use the pandas `drop()` method to separate the target column from the rest of the columns. Then we will use `train_test_split()` to separate the data into training and test sets. We will set the test size to 20%, or 0.2. The `random_state` parameter is set to 42 to ensure that the random state is saved. It ensures that the data is split in a reproducible and consistent manner. Here is the code for how we create these two variables:

```
[4]: from sklearn.model_selection import train_test_split

# Dropping the target column
X = df_copy.drop(columns=["flower_type"])

# Separate target column from DataFrame
y = df_copy["flower_type"]

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=42)

# Checking the shape of training and test data set
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

[4]: ((16, 4), (16,), (4, 4), (4,))

3. The reason for this is that machine learning models can be sensitive to the scale of the input features. If the input features have different scales, the model may give more weight to features with larger scales, even if they are not more important. This can result in a biased model with poor performance. Standardization can help mitigate this issue by rescaling the input features to have a zero mean and unit variance, ensuring that each feature has equal importance in the model.
In addition, standardization can also help to improve the convergence rate of certain machine learning algorithms, such as gradient descent, which can be slow to converge if the input features have widely different scales.

Here is one way we can standardize our data using Sklearn:

```
[5]: from sklearn.preprocessing import StandardScaler

# Scale the data using StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
X_train

[5]: array([[-1.3782598 , -1.53281263, -0.23791548, -0.2773501 ],
       [ 0.94301987,  1.25411943,  1.66540833,  1.94145069],
       [-0.44974794, -0.1393466 ,  1.03096706, -0.2773501 ],
       [-0.91400387, -0.97542622,  0.39652579, -0.2773501 ],
       [ 1.63940377,  0.97542622,  1.66540833,  0.83205029],
       [ 0.94301987,  1.25411943, -0.87235674,  1.94145069],
       [-1.61038777, -1.25411943, -2.14123928, -1.38675049],
       [-0.6818759 , -0.69673301, -0.87235674, -0.2773501 ],
       [-0.21761997, -0.97542622,  0.39652579, -1.38675049],
       [ 0.24663596,  0.97542622,  0.39652579,  0.83205029],
       [ 0.014508 ,  0.41803981, -0.23791548, -0.2773501 ],
       [-0.44974794, -1.25411943, -0.23791548, -1.38675049],
       [ 0.014508 , -0.1393466 ,  0.39652579, -0.2773501 ],
       [ 0.94301987,  0.69673301,  0.39652579, -0.2773501 ],
       [ 1.87153173,  1.53281263, -1.50679801, -0.2773501 ],
       [-0.91400387, -0.1393466 , -0.23791548,  0.83205029]])
```

You can see that our output above has our data standardized to ones and zeros.

Day 49: End-to-End Regression Challenge

For this challenge, you are going to do an end-to-end regression project. You are going to train a model to predict the sale price of a house. This is a guided challenge that is meant to show how you can carry out an end-to-end regression problem. This is in no way exhaustive or the only way to carry out this challenge. Feel free to experiment with different methods.

1. What is meant by "regression" in supervised machine learning?
2. Load the **house_price_prediction_data** dataset using pandas. Check for missing values, the shape of the DataFrame, and duplicates.
3. In machine learning, what is the meaning of "continuous values"? Which columns in the DataFrame above have continuous values?
4. What is the main goal of Exploratory Data Analysis (EDA)?
 - a. Plot a scatter plot of the "**X2 house age**" column (x-axis) and the "**Y house price of unit area**" column (y-axis). This will reveal any relationship between this variable and the target column. (Use Matplotlib).
 - b. Using Seaborn, plot a bar plot of the "**X4 number of convenience stores**" column and the "**Y house price of unit area**." Is there a correlation between the number of stores in the location and the price of the house?
 - c. To check the distribution of data, using Matplotlib, create a histogram plot of the "**Y house price of unit area**" column (the target column). Calculate the **mean** of the column. Add a mean line to the plot. Do you notice any outliers in the data?
5. Create a copy of the DataFrame and drop all rows that have values of over 115 in the "**Y house price of unit area**" column. These are outliers.

6. As you may have noticed, the "No" column is an index column. Since our DataFrame has a numeric index, drop this column.
7. Extract the year from the "**X1 transaction date**" column and create a year column called "**X1 Transaction Year**." Drop the "**X1 transaction date**" from the DataFrame. The "**X1 Transaction Year**" should be the only date column.
8. Shuffling a DataFrame is important in machine learning to avoid any bias that may be introduced when the data is ordered in a certain way. Our data seems to have a particular order. The prices and sizes are ordered from smallest to largest. This may create bias. Using Sklearn, shuffle the DataFrame.
9. Write code to separate the DataFrame (shuffled DataFrame) into x and y variables. X for feature columns and y for target columns; split into training and test sets using the Sklearn `train_test_split()` function. Make the test set to 10% of the data.
10. You are going to use the Sklearn **LinearRegression** model to train. Ensure that your data is scaled before fitting. Use the **r2_score** and **MSE** (mean squared error) metrics to evaluate the model. What is the significance of these metrics?
11. Cross-validation is a technique used to assess the model's performance on unseen data by splitting the dataset into multiple subsets (folds) and evaluating the model on each fold. The cross-validation score is an average of the evaluation scores across all folds. Use the cross validation function to assess the performance of a model on unseen data. Use the **LinearRegression** model. Make the number of folds 10. Compare your cross-validation score to your r2-score (question 10). Ensure your data is scaled using StandardScaler. Is the model showing signs of overfitting?
12. Save the model you fitted in question 10 using the joblib module. Load the model and make predictions on X_test.

Day 49 - Answers

1. In supervised machine learning, regression means predicting a continuous numerical value for each input data point based on its features. For example, in this challenge, we are going to try to train the model to predict the sale price of the house based on the features of location, transaction date, age of the house, and so forth.
2. We will import most of the libraries that we need for this project. It is always recommended to have all the required libraries as the first line of code. However, since this is a guided project, you can import the necessary modules when required by the challenge.

```
[1]: import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.model_selection import learning_curve
from sklearn.model_selection import cross_val_score
import joblib
```

Importing the **house_price_prediction_data** dataset.

[2]:

```
# Importing dataset
df = pd.read_csv("house_price_prediction_data.csv")
df.head()
```

[2]:

No	X1 transaction date	X2 house age	X3 distance to the nearest MRT station	X4 number of convenience stores	X5 latitude	X6 longitude	Y house price of unit area
0	1	2012.917	32.0	84.87882	10	24.98298	121.54024
1	2	2012.917	19.5	306.59470	9	24.98034	121.53951
2	3	2013.583	13.3	561.98450	5	24.98746	121.54391
3	4	2013.500	13.3	561.98450	5	24.98746	121.54391
4	5	2012.833	5.0	390.56840	5	24.97937	121.54245

Checking for missing values:

[3]: # Checking for missing values
df.isnull().sum()

[3]: No 0
X1 transaction date 0
X2 house age 0
X3 distance to the nearest MRT station 0
X4 number of convenience stores 0
X5 latitude 0
X6 longitude 0
Y house price of unit area 0
dtype: int64

Checking the **shape** of the DataFrame.

[4]: # Checking shape
df.shape

[4]: (414, 8)

Let's check if we have any duplicates:

[5]: # Checking for duplicates
df.duplicated().sum()

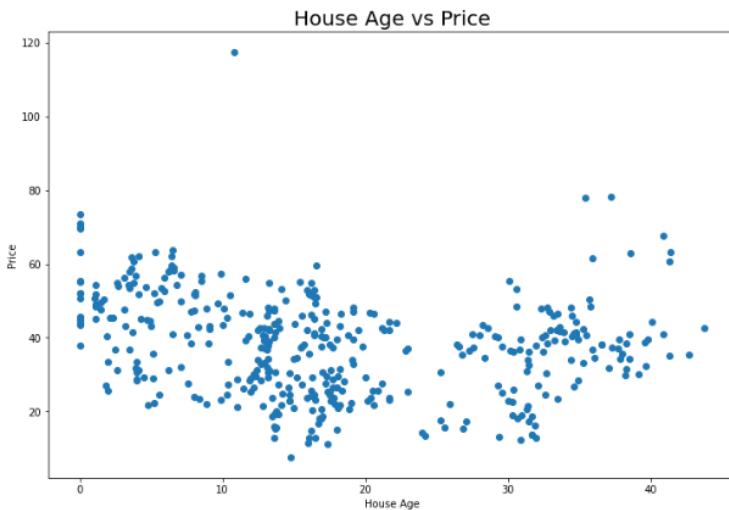
[5]: 0

3. A continuous numerical value is a type of numerical variable that can take on any value within a range, including decimal values. In other words, a continuous variable can take on an infinite number of values between any two points.

For example, the height of a person is a continuous number. This is because the height of a person can be 120.1298909090928, yes, continuous. For example, in the data above, the price of a house and the size (square foot) are continuous numerical values. They can be any amount between a certain minimum and maximum price, and both can be expressed as decimal values.

4. Exploratory Data Analysis (EDA) The main goal of EDA is to gain a deeper understanding of the data and identify any interesting or unexpected patterns or relationships. This can involve examining the distribution of the data, looking for outliers, and identifying trends and relationships between variables.
- a. A scatter plot helps to visualize the patterns and trends in the data. It can also help to visualize outliers in the data.

```
[6]: plt.figure(figsize = (12, 8))
plt.scatter(x = df["X2 house age"],
            y = df["Y house price of unit area"])
plt.xlabel("House Age")
plt.ylabel("Price")
plt.title( "House Age vs Price", fontsize = 20)
plt.show()
```



You can see that we have a huge concentration of data points around 0–20 years and between 30 and 40 years. The price of the house seems to drop as it advances in age. The price starts to rise again for houses 25 years and older. We can also see an outlier that is about 10 years old and has a price of about 120k.

- b. Bar plots are simple, making it easy to visualize, understand, and communicate categorical data. In this plot, we are going to visualize how the number of convenience stores in an area impacts the house sale price.

```
[7]: plt.figure(figsize=(12,8))
sns.barplot(data=df,
             x=df["X4 number of convenience stores"],
             y=df["Y house price of unit area"])
plt.title("House Price vs Number of Stores", fontsize = 20)
plt.show()
```



It looks like there is a correlation between the number of stores and the price of the house. As the number of stores increases, the price of the house rises.

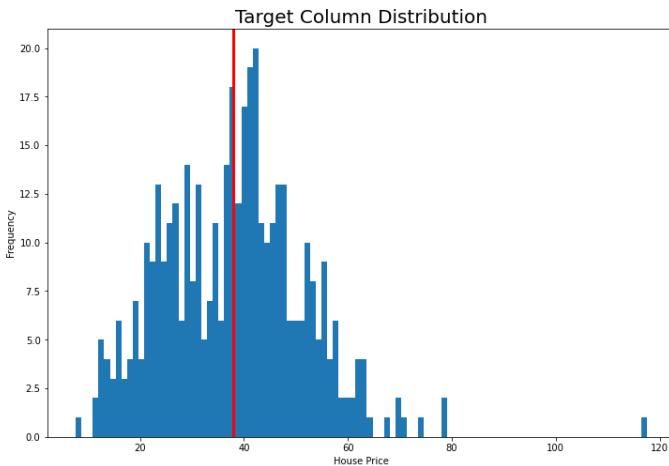
- c. Creating a histogram plot will help us visualize the distribution of data in the column. It shows how the data is spread across the range of values and how frequently it occurs. It also helps to identify outliers. We will plot the data and calculate the mean of each column. We will add a `mean()` line to the plot using the `plt.axvline()` method.

```
[8]: plt.figure(figsize=(12,8))
plt.hist(df["Y house price of unit area"],
         bins= 100)

print(f'Mean price is: {df["Y house price of unit area"].mean():.2f}')

plt.axvline(df["Y house price of unit area"].mean(),
            color='Red',
            linestyle='solid',
            linewidth=3)
plt.ylabel("Frequency")
plt.xlabel("House Price")
plt.title("Target Column Distribution", fontsize = 20)
plt.show()

Mean price is: 37.98
```



The red line is the mean of the column (37.98). We can see that there are data points around 120. These data points are separated from the majority of the data points. This makes them outliers.

5. We are going to create a copy of the DataFrame and use filtering to drop all rows that have values over 115 in the "**Y house price of unit area**" column.

```
[9]: # Create a copy of DataFrame
df_copy = df.copy()

[10]: # Dropping outliers
df_copy = df_copy[df_copy["Y house price of unit area"] < 115]
```

6. We are going to use the `drop()` method to drop the column.

```
[11]:
# Drop "No" columns
df_copy = df_copy.drop(columns=["No"])
df_copy.head()

[11]:
   X1          X2  X3 distance to the nearest MRT station  X4 number of convenience stores  X5  latitude  X6  longitude  Y house price of unit area
0  2012.917    32.0           84.87882                  10      24.98298  121.54024     37.9
1  2012.917    19.5           306.59470                  9      24.98034  121.53951     42.2
2  2013.583    13.3           561.98450                  5      24.98746  121.54391     47.3
3  2013.500    13.3           561.98450                  5      24.98746  121.54391     54.8
4  2012.833    5.0            390.56840                  5      24.97937  121.54245     43.1
```

You can see above that the column has been dropped.

7. We are going to create a function to extract the "year" from the "X1 transaction date" column. We will convert the "X1 Transaction Year" to an integer data type.

```
[12]: def convert_column(column):
    column = column.astype("str")
    # Split the column into two columns
    df_copy [[ "X1 Transaction Year", "Date" ]] = column.str.split(".", expand = True).astype(int)
    df_copy.drop(columns=[ "X1 transaction date", "Date" ], inplace=True)
    return df_copy.head()

convert_column(df_copy["X1 transaction date"])

[12]:
   X2  X3 distance to the nearest MRT station  X4 number of convenience stores  X5  latitude  X6  longitude  Y house price of unit area  X1 Transaction Year
0    32.0           84.87882                  10      24.98298  121.54024     37.9             2012
1    19.5           306.59470                  9      24.98034  121.53951     42.2             2012
2    13.3           561.98450                  5      24.98746  121.54391     47.3             2013
3    13.3           561.98450                  5      24.98746  121.54391     54.8             2013
4     5.0            390.56840                  5      24.97937  121.54245     43.1             2012
```

8. We are going to use **Shuffle** from **Sklearn**. We will create a new variable for the shuffled DataFrame. The **random_state** parameter ensures that the shuffling is reproducible. By specifying a fixed value for random_state, the same shuffling order will be maintained every time the code is run.

[13]:

```
df_shuffled = shuffle(df_copy, random_state = 42)
df_shuffled.head()
```

[13]:

	X2 house age	X3 distance to the nearest MRT station	X4 number of convenience stores	X5 latitude	X6 longitude	Y house price of unit area	X1 Transaction Year
395	21.2	512.5487	4	24.97400	121.53842	42.5	2012
350	13.2	492.2313	5	24.96515	121.53737	42.3	2013
401	7.6	2175.0300	3	24.96305	121.51254	27.7	2013
354	12.2	1360.1390	1	24.95204	121.54842	30.1	2013
181	11.6	201.8939	8	24.98489	121.54121	55.9	2013

You can see that the index of the DataFrame has been randomized.

9. We are going to use **train_test_split** from the Sklearn model. We will drop the target variable from the DataFrame. Once we create variables X and y, we split the data into training and test datasets.

```
[14]: # Creating X and y variables
X = df_shuffled.drop(columns=["Y house price of unit area"])
y = df_shuffled["Y house price of unit area"]

# Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.1,
                                                    random_state=23)
```

10. For this challenge, we are going to import the **LinearRegression** estimator and the **StandardScaler** for scaling the features before fitting. Then, we will score the model on the test sets using the **r2_score** and **MSE** metrics.

The `r2_score` measures how well the regression line (or curve) fits the data points. In other words, it measures how well the independent variables explain the variance in the dependent variable (target). The score can range from 0 to 1. A score of 1 indicates a perfect prediction (it fits all the data points), while a score of 0 indicates that the model does not fit at all. So, basically, we want this model to score close to 1.

MSE measures the average squared difference between the predicted values and the actual values. A lower MSE indicates better performance from the model.

```
[15]: # Scale the data using StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Instantiating model
reg = LinearRegression()
reg.fit(X_train, y_train)

# Scoring the model on test dataset
y_pred = reg.predict(X_test)
r2_score = r2_score(y_test, y_pred)
mse = mean_squared_error(y_test,y_pred)
print(f'The r2_score is: {r2_score:.2f}')
print(f'The mean squared error is: {mse:.2f}')
```

```
The r2_score is: 0.72
The mean squared error is: 41.18
```

The model scores a R2 score of 0.72 and a MSE score of 41.18. The model is not doing so well, but again, not so bad.

***Note that your scores will likely be different from the ones above.*

11. First, we import the `cross_val_score` from Sklearn.

model_selection. We will create a pipeline to scale and train the data. We will pass the **LinearRegression** model and StandardScaler to the pipeline. The number of folds is 10.

```
[16]: from sklearn.pipeline import make_pipeline

# Create a pipeline that first scales the data, then fits the model
pipeline = make_pipeline(StandardScaler(), LinearRegression())

scores = cross_val_score(pipeline, X, y, cv=10, scoring ='r2')
print(f'The accuracy score is {scores.mean():.2f}')

The accuracy score is 0.60
```

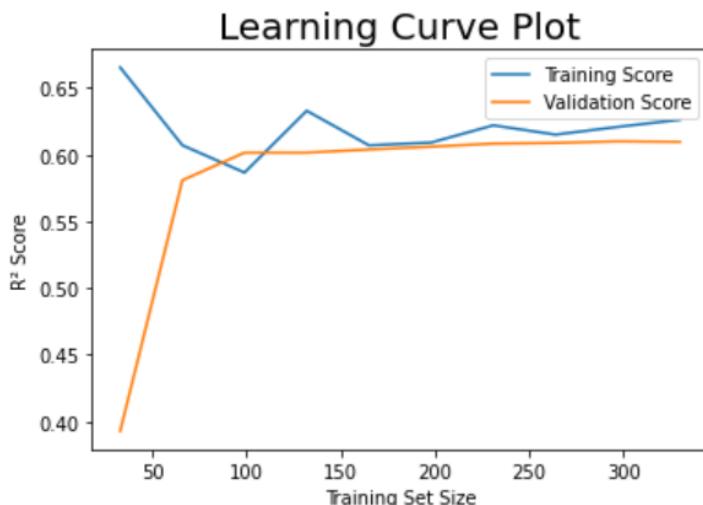
The performance of the model, as measured by the `r2_score` using `cross_val_score`, is 60%, while the `r2_score` on a single train-test split is 75%. This discrepancy could suggest that the model is overfitting to the training data, as it performs better on one specific test set but worse across multiple subsets of the data during cross-validation. Overfitting occurs when the model captures noise or random variations in the data, resulting in high performance on training data but poorer generalization to unseen data. We can confirm overfitting using the **learning_curve**. We use the pipeline as estimator to ensure data is scaled.

```
[17]: from sklearn.model_selection import learning_curve

train_sizes, train_scores, val_scores = learning_curve(
    pipeline, X, y, cv=5, scoring='r2', train_sizes=np.linspace(0.1, 1.0, 10))

# Calculating Mean and std for train/validation scores
train_scores_mean = np.mean(train_scores, axis=1)
val_scores_mean = np.mean(val_scores, axis=1)
# Plotting the data
plt.plot(train_sizes, train_scores_mean, label="Training Score")
plt.plot(train_sizes, val_scores_mean, label="Validation Score")
plt.xlabel("Training Set Size")
plt.ylabel("R2 Score")
plt.legend()
plt.title("Learning Curve Plot", fontsize=20)
plt.show()
```

This will output the plot:



The blue line represents the model's performance on the training set as the training set size increases. The orange line represents the model's performance on the validation set as the training set size increases. Here the training score increases while the validation score plateaus around 0.6. This suggests that the model might be memorizing the training data rather than learning general patterns, suggesting overfitting.

12. We are going to use the joblib library to save the model.
To install the library, you can run: **pip install joblib**.

Here is the code below:

```
[18]: # Saving the model to a file  
joblib.dump(reg, 'regression_model.joblib')
```

```
[18]: ['regression_model.joblib']
```

To load the model and make predictions, we are going to use the **load()** method from joblib and make predictions on the X_test data.

```
[19]: # Load the model from file
reg_loaded = joblib.load("regression_model.joblib")

# Making predictions with model
model_predictions = reg_loaded.predict(X_test)
model_predictions
```



```
[19]: array([27.66433978, 53.41815849, 36.30487344, 31.92014729, 13.04274426,
       15.42123066, 43.05381432, 27.7182633 , 47.02503307, 51.71265544,
       48.3814149 , 13.27596951, 33.16028647, 37.78334714, 37.95011699,
       34.11632905, 39.64077071, 42.89190858, 53.39119674, 48.52456306,
       44.52524061, 26.2983223 , 52.72395378, 45.63201692, 53.33727322,
       32.76701935, 31.15244789, 31.31189847, 49.27736471, 46.91914625,
       40.24893988, 48.23267027, 35.20968084, 45.73093231, 45.75789407,
       47.06020559, 13.07605357, 43.05381432, 40.28542852, 46.97265465,
       45.58001659, 35.96057733])
```

Day 50: End-to-End Classification Challenge

In this challenge, you are going to carry out an end-to-end classification problem. You are going to train a model to predict the risk of someone suffering from heart disease. You are going to import data, clean and process the data, train a model, and evaluate the model's performance. This is a guided challenge. Please note that this represents one approach of many you can take to tackle this issue. This guided challenge is meant to show the general steps that one can take to train a model. Training a machine learning model takes a lot of experimentation to find what works. So, feel free to carry out more experimentation.

1. Import the Cardiovascular Study dataset using pandas. Use the pandas `head()` method to check that your data has loaded properly.
 - a. Check if there are duplicates in the dataset. Check the `shape` of your dataset (number of columns and rows).
 - b. Check for data types in the DataFrame.
 - c. Is there any missing data in the DataFrame?
 - d. What are the `minimum` and `maximum` ages in the dataset? Create a histogram plot for the age column.
 - e. Identify the target column. Compare the value count of ones and zeros in the column. Plot this on a bar plot.
2. Now, create a copy of the DataFrame. Clean up the data by dropping all the rows that have null values. Write a code to confirm that the null values have been dropped.
3. Exploratory data analysis is important to ensure that we understand our data. Using the pandas `quantile()` method, count the number of outliers in each column with a numeric datatype. Using the `boxplot()` function from `Seaborn`, create a plot to visualize these outliers in the data. Your plot should only include columns with outliers.

Outliers are identified as data points that are below the lower fence ($Q1 - 1.5 * IQR$) or above the upper fence ($Q3 + 1.5 * IQR$).

4. You have decided to drop the outliers in the "BMI" column. Check the shape of the DataFrame before and after dropping outliers.
5. You want to know the correlation between smoking and the risk of heart disease. Using crosstab, compare the "**is_smoking**" column to the "**TenYearCHD**" column. Is there a link between smoking and heart disease? Create a bar plot using pandas.
6. How many females and males are in the dataset? Do a crosstab of the "sex" column and the target column (TenYearCHD) to find a relationship between gender and heart disease. Create a bar plot using pandas.
7. Is there a link between a person's cholesterol levels, their age, and heart disease? To answer this question, plot a scatter plot using Matplotlib of the "**totChol**" column and the "**age**" column for people that have heart disease in the "**TenYearCHD**" column (remember that 1 is for heart disease and 0 is for non-heart disease).
8. Is there a link between heart rate, age, and the risk of heart disease? To answer this question, plot a scatter plot using Matplotlib.
9. Let's consider the social aspect: Is there a link between a person's level of education and heart disease? Plot a bar plot comparing the number of people with heart disease to those without heart disease at all levels of education. Try using a crosstab. Use Matplotlib.
10. Using the Sklearn **LabelEncoder**, transform the non-numeric columns into numeric ones. To further understand the relationship between variables, create a **heatmap** using the Seaborn library.
11. Separate the DataFrame into **X** and **y** values. The **y** value will be the target column. The target column is **TenYearCHD**. The variables will be the remainder of the channels. Since the dataset has a numeric index, the "**id**"

column seems redundant; please drop it. Check the shapes of **x** and **y** and ensure that they have an equal number of rows.

12. Split the data into training and testing sets. 20% of the data will be testing data. Scale the data using StandardScaler from the Sklearn library. Train the logistic regression model with the data. Test the accuracy score of the model on the testing sets.
13. Now that we have our accuracy score on the test data, try hyperparameter tuning the logistic regression model using **RandomizedSearchCV**. Hyperparameter tuning is simply searching for the best combination of parameters for the best performance of the machine learning model. Use the best parameters obtained by **RandomizedSearchCV** to instantiate a **LogisticRegression** model and fit the training data. What is the accuracy score?
14. Above, you have evaluated the results using the accuracy score. The accuracy score does not give a complete picture of the performance of our model. We need to use other methods to evaluate or model.
 - a. Use the **confusion_matrix** to obtain an understanding of what the model is predicting right and what it is predicting wrong in the test data.
 - b. Generate a ROC curve plot for the results. Use the **predict_proba()** method to calculate the AUC. What is the significance of the Roc curve?
 - c. Generate the classification report using Sklearn. The classification report will evaluate the following matrices: **precision**, **recall**, and **F1-score**. What can you conclude from each metric? Using this report, can you conclude if there is a data imbalance in the dataset?
15. Is there another way we can obtain even better results? How can we deal with the imbalanced data? Explain.
16. Save the model you trained above (question 13). Save the model using the **pickle library**. After saving the model, load it.

Day 50 - Answers

- When doing a project, it is generally recommended to start by importing all the needed libraries. For this project, we will import all the libraries used in this project in the first cell. However, since this is a guided project, I expect you to import the libraries when they are needed (when answering a specific question). We are going to import the "cardiovascular_dataset" and use the `head()` method to view the first 5 rows of the dataset. Since this is an end-to-end project, we are going to import all the libraries that we are going to need for the project.

```
[1]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import classification_report
from sklearn import metrics
import pickle
```

```
[2]: df = pd.read_csv('cardiovascular_dataset.csv')
df.head()
```

```
[2]:   id age education sex is_smoking cigsPerDay BPMeds prevalentStroke prevalentHyp
  0   0    64          2.0    F      YES       3.0     0.0         0           0
  1   1    36          4.0    M       NO      0.0     0.0         0           1
  2   2    46          1.0    F      YES      10.0     0.0         0           0
  3   3    50          1.0    M      YES      20.0     0.0         0           1
  4   4    64          1.0    F      YES      30.0     0.0         0           0
```

- Checking for duplicates using the `duplicated()` method and the shape **attribute** of the DataFrame

```
[3]: df.duplicated().sum()  
[3]: 0  
[4]: # Checking the shape of DataFrame  
df.shape  
[4]: (3390, 17)
```

You can see that our DataFrame has no duplicates and has 3390 rows and 17 columns.

- b. To check the data types in the DataFrame, we are going to use the `dtypes` attribute. This attribute will return the data types of each column in the DataFrame.

```
[5]: df.dtypes  
[5]: id          int64  
age         int64  
education    float64  
sex          object  
is_smoking   object  
cigsPerDay   float64  
BPMeds       float64  
prevalentStroke int64  
prevalentHyp   int64  
diabetes      int64  
totChol       float64  
sysBP         float64  
diaBP         float64  
BMI          float64  
heartRate     float64  
glucose       float64  
TenYearCHD    int64  
dtype: object
```

- c. To check for missing numbers in the DataFrame, we can use the `isnull()` method of pandas.

```
[6]: # Check for missing values  
df.isnull().sum()
```

```
[6]: id          0  
age         0  
education   87  
sex         0  
is_smoking  0  
cigsPerDay  22  
BPMeds      44  
prevalentStroke 0  
prevalentHyp  0  
diabetes    0  
totChol     38  
sysBP       0  
diaBP       0  
BMI         14  
heartRate   1  
glucose     304  
TenYearCHD  0  
dtype: int64
```

You can see above that we have some missing values in some columns (education, cigsPerDay, BPMeds, totchol, BMI, heartrate and glucose).

- d. To find the minimum and maximum age of the age column, we can use the `min()` and `max()` methods.

```
[7]: # Finding the minimum age  
min_age = df.age.min()  
print(f'The minimum age is: {min_age}')
```

```
The minimum age is: 32
```

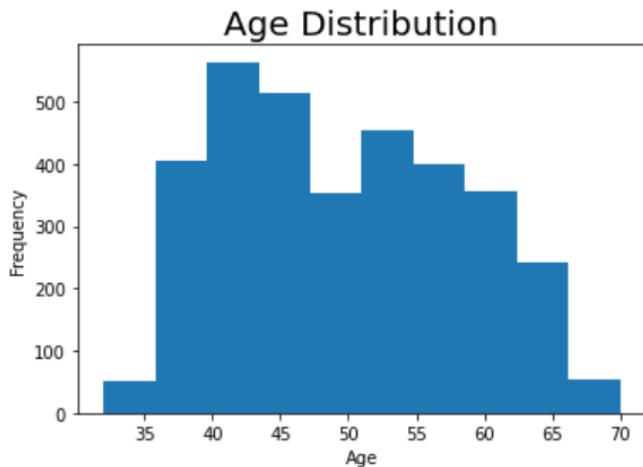
The maximum age is:

```
[8]: # Finding the maximum age  
max_age = df.age.max()  
print(f'The maximun age is: {max_age}')
```

```
The maximun age is: 70
```

Creating a histogram to visualize age distribution.

```
[9]: # Ploting a hist plot of the age column
df["age"].plot(kind="hist")
plt.xlabel(xlabel="Age")
plt.title("Age Distribution", fontsize = 20)
plt.show()
```



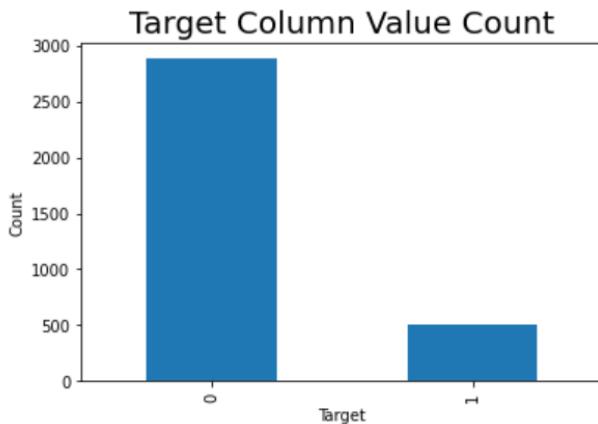
- e. The "TenYearCHD" is the target column. We are going to use the `value_counts()` method to compare the count of ones and zeros in the column.

```
[10]: # Using value_count to count uniques values in column
df.TenYearCHD.value_counts()
```

```
[10]: TenYearCHD
0    2879
1    511
Name: count, dtype: int64
```

Now let's visualize this with a barplot.

```
[11]: # Plotting the value count of the target column
df.TenYearCHD.value_counts().plot(kind= "bar")
plt.title("Target Column Value Count", fontsize = 20)
plt.xlabel(xlabel= "Target")
plt.ylabel(ylabel= "Count")
plt.show()
```



2. Remember that rows are on **axis-0** and columns are on **axis-1**. We are going to use the pandas `dropna()` method to drop all rows with missing values (null). We are going to pass **axis-0** as the argument to drop the rows. First, we will create a copy of the DataFrame.

```
[12]: # Creating a copy of the DataFrame  
df_2 = df.copy()  
  
[13]: # Dropping rows  
df_2 = df_2.dropna(axis=0)  
  
# Checking for null values in the DataFrame  
df_2.isnull().sum()  
  
[13]: id          0  
age          0  
education    0  
sex          0  
is_smoking   0  
cigsPerDay   0  
BPMed          0  
prevalentStroke 0  
prevalentHyp   0  
diabetes      0  
totChol        0  
sysBP          0  
diaBP          0  
BMI            0  
heartRate      0  
glucose         0  
TenYearCHD     0  
dtype: int64
```

You can see that we do not have any missing values in the DataFrame anymore.

3. To return the columns and the count of outliers in each numeric column, we can use the interquartile range (IQR) method. First, we will get all the columns that are of the numeric data type and pass them as arguments to the method. This function will return the count of the number of outliers in each column.

```
[14]: # Returning all columns with numeric dtypes
numeric_dtype_columns = []
for column, values in df_2.items():
    if values.dtype != "object":
        numeric_dtype_columns.append(column)

# Function to return a count of outliers in each column
def count_outliers_in_columns(df, columns):
    Q1 = df[columns].quantile(0.25)
    Q3 = df[columns].quantile(0.75)
    IQR = Q3 - Q1
    outliers = df[((df[columns] < (Q1 - 1.5*IQR)) | (df[columns] >
                                                               (Q3 + 1.5*IQR)))]
    return outliers[columns].count()

count_outliers_in_columns(df_2, numeric_dtype_columns)
```

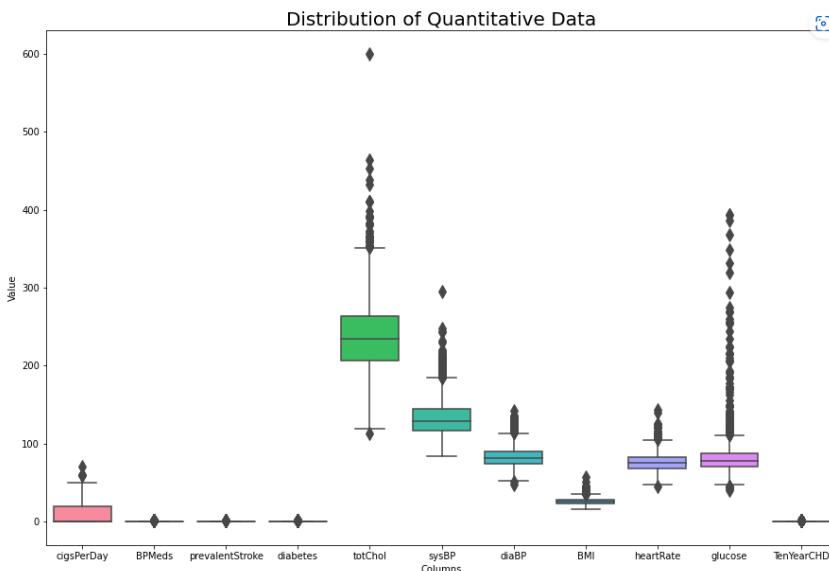
```
[14]: id          0
age         0
education   0
cigsPerDay  7
BPMeds      88
prevalentStroke 18
prevalentHyp  0
diabetes     79
totChol      34
sysBP        90
diaBP        50
BMI          72
heartRate    56
glucose      146
TenYearCHD   444
dtype: int64
```

A **boxplot** is the best way to visualize outliers in the DataFrame. We are going to create a **boxplot** with only columns that have outliers. Here is how we can plot one:

```
[15]: plt.subplots(figsize=(15,10))
sns.boxplot(data=df_2.drop(columns=[['id', 'age', 'education', 'prevalentHyp',
                                    'sex',
                                    'is_smoking']], inplace= False),
            saturation=0.95,
            fliersize=10,
            whis=1.5,
            width=0.8)

plt.xlabel(xlabel= "Columns")
plt.ylabel(ylabel = "Value")
plt.title("Distribution of Quantitative Data", fontsize= 20)
plt.show()
```

Because they display the median, interquartile range, and minimum and maximum values of the data, boxplots are helpful for visualization. In a boxplot, the first and third quartiles (the 25th and 75th percentiles) are represented by the lower and upper edges of a box. The median of the data is represented by a line within the box. Outliers are plotted as individual points outside of the whiskers. This will output the plot below:



4. We can use the interquartile range (IQR) method to identify the outliers in a column and drop them. Here is the data shape before dropping outliers:

```
[16]: # Checking the shape before dropping outliers  
df_2.shape
```

```
[16]: (2927, 17)
```

Now, we are going to drop the outliers from the BMI column and check the shape of the DataFrame.

```
[17]: # Dropping outliers in a column
Q1 = df_2.BMI.quantile(0.25)
Q3 = df_2.BMI.quantile(0.75)

IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Dropping the outliers in a column
df_2 = df_2[(df_2.BMI > lower_bound) & (df_2.BMI < upper_bound)]
df_2.shape
```

[17]: (2855, 17)

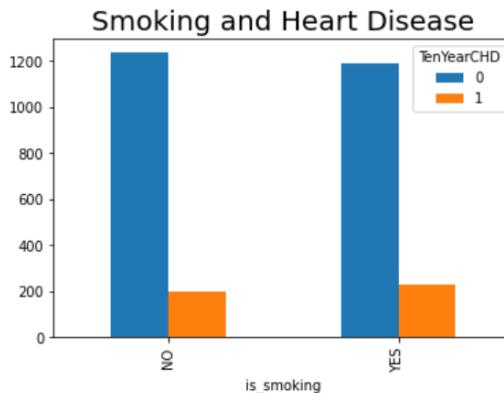
5. A crosstab will give us a clear picture of whether there is a link between smoking and heart disease.

```
[18]: pd.crosstab(df_2.is_smoking, df.TenYearCHD)
```

```
[18]: TenYearCHD      0      1
      is_smoking
      NO    1236   198
      YES   1192   229
```

We can see that there is a link between smoking and heart disease. There is a slightly higher rate of heart disease among smokers (229) than non-smokers (189). Now let's plot this data on a bar plot.

```
[19]: pd.crosstab(df_2.is_smoking, df_2.TenYearCHD).plot(kind="bar")
plt.title("Smoking and Heart Disease", fontsize = 20)
plt.show()
```



6. To find the number of females and males in the dataset, we can call the `value_counts()` method on the "sex" column.

```
[20]: df_2.sex.value_counts()
```

```
[20]: sex
      F    1561
      M    1294
      Name: count, dtype: int64
```

You can see above that we have more females than males in our dataset.

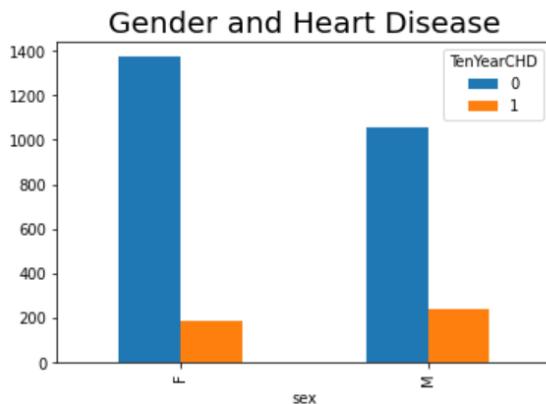
Now, let's crosstab this column with the target column to see if there is a link between gender and heart disease.

```
[21]: pd.crosstab(df_2.sex, df_2.TenYearCHD)
```

```
[21]: TenYearCHD      0      1
      sex
      -----
      F  1374  187
      M  1054  240
```

From the output, we can conclude that if you are a male, you are more likely to get a heart disease than a female. Let's plot this data.

```
[22]: pd.crosstab(df_2.sex, df_2.TenYearCHD).plot(kind='bar')
plt.title("Gender and Heart Disease", fontsize = 20)
plt.show()
```



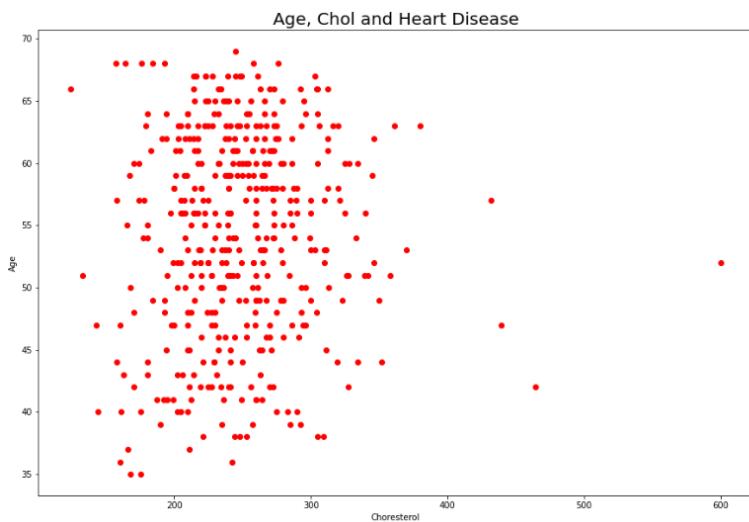
Even though there are fewer females in the dataset compared to males, we have more males with heart disease than females. This shows that the risk of heart disease is higher for males.

7. Scatter plots are a type of data visualization used to represent the relationship between two variables. We can use them to spot correlations in the data. Scatter plots can also reveal outliers and patterns in the variables that we are comparing. We are using the scatter plot below to visualize the correlation between cholesterol levels and the risk of heart disease.

```

plt.subplots(figsize=(15,10))
plt.scatter(df_2.totChol[df_2.TenYearCHD==1],
            df_2.age[df.TenYearCHD==1], c= "red")
plt.xlabel(xlabel ="Cholesterol")
plt.ylabel(ylabel ="Age")
plt.title("Age, Chol and Heart Disease", fontsize= 20)
plt.show()

```



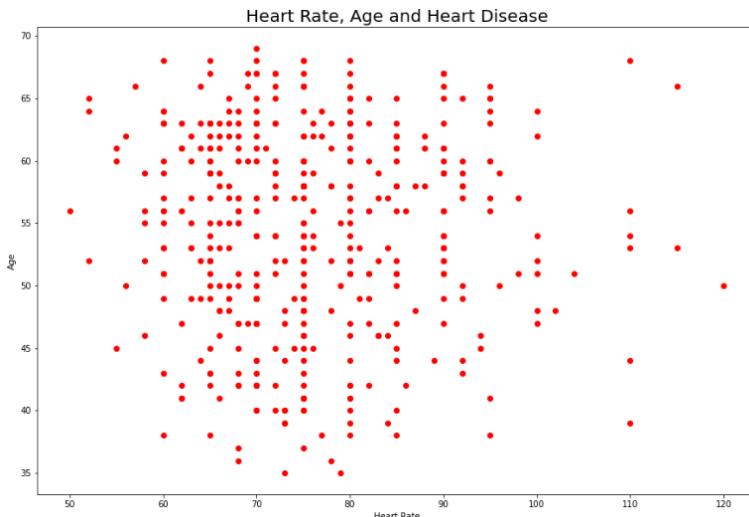
From the plot, we can conclude that there is a correlation between an increase in age, cholesterol, and the risk of heart disease. The density of the points is high after 55 years. The risk of getting a heart attack also increases when cholesterol levels hit 200.

8. Let's see if there is a relationship between heart rate, age and heart disease. We will plot another plot.

```

plt.subplots(figsize=(15,10))
plt.scatter(df_2.heartRate[df_2.TenYearCHD==1],
            df_2.age[df_2.TenYearCHD==1], c= "red")
plt.xlabel(xlabel ="Heart Rate")
plt.ylabel(ylabel ="Age")
plt.title("Heart Rate, Age and Heart Disease", fontsize = 20)
plt.show()

```



It looks like the chances of heart disease start to increase at the age of 40 and older. However, there doesn't seem to be a strong, linear correlation between heart rate and age among individuals with heart disease

- Using a bar plot, we can visualize the link between education and heart disease. We are going to use a crosstab to create a plot.

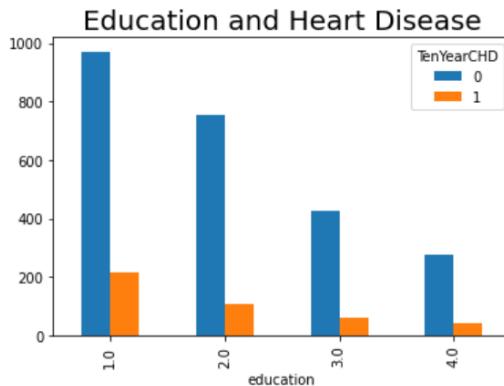
```

[25]: pd.crosstab(df_2.education, df_2.TenYearCHD)

[25]:
   TenYearCHD      0      1
   education
   1.0    971   216
   2.0    756   107
   3.0    424    60
   4.0    277    44

```

```
[26]: pd.crosstab(df_2.education, df_2.TenYearCHD).plot(kind='bar')
plt.title("Education and Heart Disease", fontsize = 20)
plt.show()
```



There seems to be a link between a person's level of education and their risk of heart disease. It seems that as the level of education increases, the risk of heart disease decreases.

10. We are going to import **LabelEncoder** from Sklearn. It is important that we convert non-numeric columns because machine learning algorithms can only read numeric values. We are going to convert the "sex" column and the "is_smoking" column. We will use the **head()** method to check if the columns have been processed.

```
[27]: le = LabelEncoder()
df_2['sex'] = le.fit_transform(df_2['sex'])
df_2['is_smoking'] = le.fit_transform(df_2['is_smoking'])
df_2.head(5)
```

```
[27]:   id  age  education  sex  is_smoking  cigsPerDay  BPMeds  prevalentStroke
  1    1    36          4.0    1            0        0.0      0.0          0
  2    2    46          1.0    0            1       10.0      0.0          0
  3    3    50          1.0    1            1       20.0      0.0          0
  4    4    64          1.0    0            1       30.0      0.0          0
  5    5    61          3.0    0            0        0.0      0.0          0
```

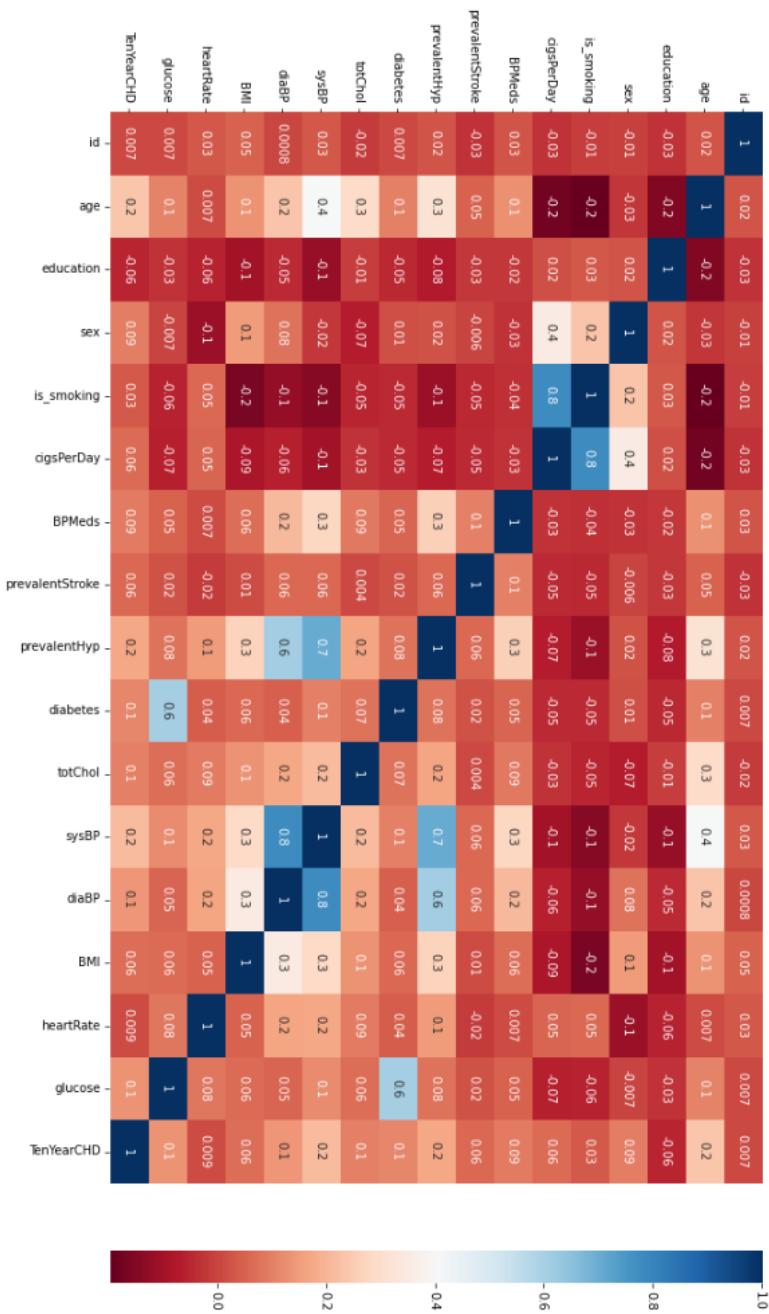
Now, let's create a heatmap using Seaborn.

```
[28]: # Calculate the correlation matrix
corr = df_2.corr()

# Plot the heatmap of the correlation matrix
plt.subplots(figsize=(20,10))
sns.heatmap(corr,
            annot=True,
            cmap="RdBu",
            fmt=".0g",
            vmax=1,
            )
plt.title("Correlation Matrix", fontsize = 20)
plt.show()
```

Find the plot on the next page.

Correlation Matrix



Positive numbers mean the feature has a strong correlation to the positive output (1), and negative numbers mean a negative correlation, meaning the more that number increases, the higher the chances that one will not have heart disease. For example, education has a negative correlation. This means that the more educated someone is, the less likely it is that they will have a heart problem. Age and totChol show a strong positive correlation (0.3), indicating that as age increases, total cholesterol tends to increase as well.

11. We are going to separate the target column from the rest of the columns and drop the target column and the id column from the variable columns (variable X).

```
[29]: # Extracting the target column from DataFrame  
y = df_2["TenYearCHD"]  
  
# Dropping the target and 'id' columns from variables  
X = df_2.drop(columns=["TenYearCHD", "id"])
```

We can now check the shape of X and y.

```
[30]: X.shape, y.shape  
  
[30]: ((2855, 15), (2855,))
```

12. Now, let's split the dataset into two sets: one for training (**X_train**, **y_train**) and another for testing (**X_test**, **y_test**). The test set will be 20% of the total data. We will scale the data using **StandardScaler** from Sklearn, and we will train the **LogisticRegression** model.

```
[31]: # Splitting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=0.2,
                                                    random_state=42)

# Scale the data using StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train the classifier
clf = LogisticRegression()
clf.fit(X_train, y_train)

# Evaluate the classifier on the test data
score_on_test_data = clf.score(X_test, y_test)

print(f'Test data accuracy: {score_on_test_data*100:.2f}%')
```

Test data accuracy: 85.81%

You can see that our model has accuracy score of 85.81%.

13. Hyperparameter tuning the model is to find the best combination of hyperparameters in **LogisticRegression** to optimize the performance of the model on our data. To do this, we can use **RandomizedSearchCV**.

RandomizedSearchCV will randomly search for a combination of hyperparameters in the model, and it will return the combination that gives the best results. We are going to use this combination to fit the **LogisticRegression** model.

```
[32]: distributions = {
    'penalty' : ['l1', 'l2', 'elasticnet'],
    'max_iter' : range(100, 800),
    'warm_start' : [True, False],
    'solver' : ['lbfgs', 'liblinear', 'newton-cg', 'newton-cholesky', 'sag'],
    'C' : np.logspace(-1, 1, 22)
}

clf = RandomizedSearchCV(estimator = LogisticRegression(),
                         param_distributions = distributions,
                         n_iter = 100,
                         scoring = 'accuracy',
                         n_jobs = -1,
                         verbose = 1,
                         random_state = 42)

# Fit the data with RandomizedSearchCV
clf.fit(X_train, y_train)

# Getting the best hyperparameters
best_params = clf.best_params_

# Instantiating the model using the best score
clf_best_dt = LogisticRegression(**best_params)

# Using the best params to fit the classifier
clf_best_dt.fit(X_train, y_train)

# Getting accuracy score
accuracy_score = clf_best_dt.score(X_test, y_test)

print(f'Test data accuracy: {accuracy_score*100:.2f}%')

Fitting 5 folds for each of 100 candidates, totalling 500 fits
Test data accuracy: 85.99%
```

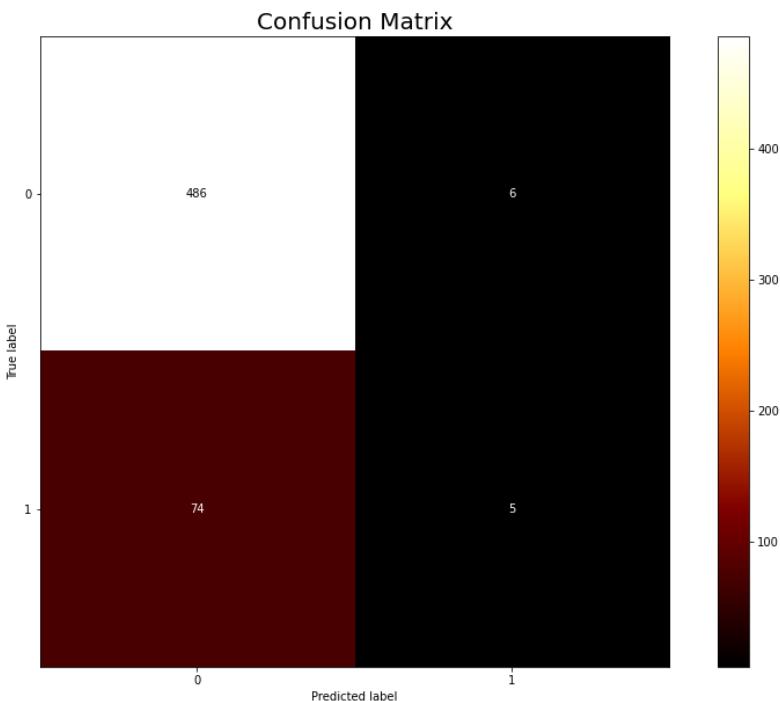
We can see that there is a tiny improvement in the results. Even though the results are not very drastic, you get the gist of how you can use **RandomizedSearchCV** to try to improve model performance.

14. This question requires that we evaluate our classification model.
 - a. To plot the confusion matrix, we are going to use the Sklearn library.

```
[33]: # predicting out on test data
y_pred = clf_best_dt.predict(X_test)

disp = ConfusionMatrixDisplay.from_predictions( y_test, y_pred,
                                              cmap=plt.cm.afmhot,
                                              display_labels = clf_best_dt.classes_)

fig = disp.ax_.get_figure()
fig.set_figwidth(15)
fig.set_figheight(10)
plt.title("Confusion Matrix", fontsize=20)
plt.show()
```



The confusion matrix has two labels: the True label and the predicted label. Here is how we can understand the results above: In the test dataset, there are 492 people who have no risk of a heart attack (label 0). Out of the 492, the trained model was able to predict 486 correctly (True negatives). It got six wrong. With 6, it predicted

that someone has a risk of heart disease when it should have predicted the opposite (a False Positive). The dataset has 79 people who have a risk of heart disease (label 1). The model was only able to predict 5 cases correctly (True positives), and it got 74 wrong (False negatives). So, our model has done quite well to predict if someone has no risk of heart disease, but it has done poorly to predict or detect the risk of heart disease.

- b. First, we are going to calculate `y_predict` for the Receiver Operating Characteristic (ROC) curve calculation. Then we will calculate the `roc_auc_score` (which represents the area under the ROC curve), and then we will plot the ROC curve. In a perfect world, the AUC is 1. The perfect 1 means the model can perfectly distinguish target 1 (risk of heart disease) from target 0 (no risk of heart disease). It means there are no false positives.

```
[34]: # Using proba to get class probabilities
y_predict = clf_best_dt.predict_proba(X_test)[:,1]

# Calculating AUC_score
auc_score = roc_auc_score(y_test, y_predict)
print(f'Auc score: {auc_score:.2f}'')
```

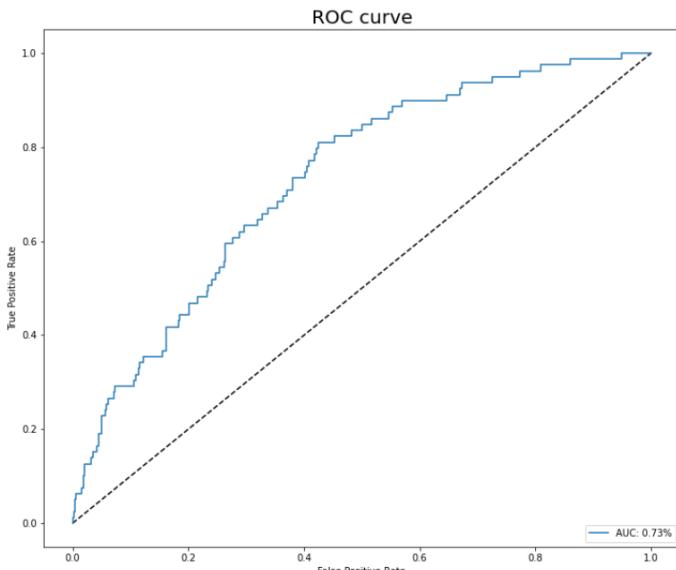
Auc score: 0.73

You can see that the ROC AUC score (0.73) is lower than the accuracy score (0.86). This indicates that the model is misclassifying negative samples more frequently than positive samples, as evident from the **confusion matrix** plot above. The model faces difficulty accurately capturing the underlying patterns and discriminating between the classes. One possible reason for this is the imbalanced dataset, where the class representing the risk of heart disease is underrepresented. As a result, the model struggles to make accurate predictions for individuals at risk of heart disease.

Now, let's view this area under the curve by plotting a ROC curve. So, in the plot below, the number of True positive rates (Tpr) is plotted against the False positive rates (Fpr). The closer the curve is to 1 on the y-axis (Tpr), the better the model can predict the risk of heart disease. The line is the 50% mark of accuracy.

```
[35]: fpr, tpr, thresholds = roc_curve(y_test, y_predict)

plt.figure(figsize=(12,10))
plt.plot([0,1],[0,1],"k--")
plt.plot(fpr,tpr, label=f'AUC: {auc_score:.2f}%')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve', fontsize= 20)
plt.legend(loc ='lower right')
plt.show()
```



So, the area under the curve represents the **roc_auc_score** that we calculated above (0.73). You can see that the curve is far from the 1.0 mark of the y-axis. This confirms the poor performance of our model.

c. Let's plot the classification report below using Sklearn.

```
[36]: # Target names
names = ['No Risk of Heart Disease', 'Risk of Heart Disease']
print(classification_report(y_test, y_pred, target_names=names))
```

	precision	recall	f1-score	support
No Risk of Heart Disease	0.87	0.99	0.92	492
Risk of Heart Disease	0.45	0.06	0.11	79
accuracy			0.86	571
macro avg	0.66	0.53	0.52	571
weighted avg	0.81	0.86	0.81	571

Our report has three main metrics: precision, recall, and f-1 score. Let's break down each of these scores:

Precision: This metric measures how accurately the trained model can find positive instances (risk of heart disease). In our model, the percentage is 45%. This means that when our label labels someone as having a risk of heart disease, it is right 45% of the time and wrong 55% of the time.

Recall: This measures how well our model can find positives (risks of heart disease) in our data. It is measured as (true positives/true positives plus (+) false negatives). So of all the cases that had a risk of heart disease, the model only caught 6% of the cases. So, the 6% represents the proportion of true positive cases (risks of heart disease) that the model identified correctly out of all the actual positive cases.

F1-score: Now, this is the weighted average between recall and precision. Our weighted average is only 11% for the minority class (Risk of Heart Disease) and 92% for the other class. This implies that our model performs much better at predicting the majority class compared to the minority class.

Support: This indicates how many samples of each class are in the test dataset. We can see that the "no risk of heart attack" class has more samples (492), and the "risk of heart attack" class has only 79 samples. This reveals something about our data: it is not balanced. This reveals that our data is imbalanced, which explains why our model has higher accuracy in predicting "no risk of heart disease" compared to "risk of heart disease." The low macroaverage further confirms the impact of data imbalances on model evaluation.

One way to detect data imbalance is by examining the precision, recall, and F1-score for each class. When the precision and recall scores for the minority class are low compared to the majority class, it indicates a potential data imbalance. Similarly, a low F1-score for the minority class can also signal an imbalanced dataset.

15. Training a machine learning model can be a complex process depending on the desired outcome. If we aim to enhance the results, there are several approaches we can explore. Firstly, acquiring more data can often lead to improved performance. Additionally, considering alternative models such as **RandomForestClassifier**, **KNeighborsClassifier**, **DecisionTreeClassifier**, etc. may yield better results compared to the **LogisticRegression** model. To fine-tune the model's hyperparameters, **GridSearchCV** can be used instead of **RandomizedSearchCV**. **GridSearchCV** exhaustively tries different combinations of hyperparameters to find the optimal configuration.

As discussed earlier, imbalanced datasets present challenges in building machine learning models since they can result in biased models that perform poorly on minority classes. To address this issue, we can employ the resampling techniques offered by the **imbalanced-learning**

library. This library provides a range of effective techniques that can help mitigate the impact of class imbalance and improve the model's performance for minority classes.

16. To save the model, we are going to use the pickle library.
Here is the code below.

```
[37]: # Save the model
with open('model.pkl', 'wb') as file:
    pickle.dump(clf_best_dt, file)
```

Here is how we can load and use our model:

```
[38]: # Loading the saved model
saved_model = pickle.load(open('model.pkl', 'rb'))
score = saved_model.score(X_test, y_test)
score
```



```
[38]: 0.8598949211908932
```

What's Next?

Congratulations on completing "**50 Days of Data Analysis with Python: The Ultimate Challenges Book for Beginners**"! By now, you've developed a strong foundation in data analysis and acquired essential skills using popular libraries like NumPy, pandas, Seaborn, Sklearn, and Matplotlib. As you reflect on your journey and look ahead, you may be wondering what steps you can take to further enhance your data analysis skills and explore exciting opportunities in the field. In this chapter, I will guide you through some suggestions to continue your growth as a data analyst and provide insights into the path towards becoming a data scientist or machine learning practitioner.

Undertake Data Analysis Projects

One of the best ways to solidify your skills and enhance your CV is by working on data analysis projects. Seek out real-world datasets or participate in platforms like Kaggle, where you can find diverse datasets and engage in competitive data analysis challenges. Taking on projects allows you to apply your skills in a practical context, tackle complex problems, and showcase your abilities to potential employers or collaborators.

Explore Advanced Topics and Techniques

To further expand your data analysis toolkit, consider delving into advanced topics and techniques. Explore machine learning algorithms and their application to predictive modeling, or dive into natural language processing (NLP) for text analysis. Learn about time series analysis, feature engineering, dimensionality reduction, and other advanced techniques that are relevant to your areas of interest. Online courses, tutorials, and textbooks

can be valuable resources for self-study and deepening your knowledge.

Expand Your Python Libraries Repertoire

While you've already gained proficiency in essential data analysis libraries, there is a vast ecosystem of Python libraries to explore. Depending on your specific interests, you may want to learn more about Sklearn for machine learning, TensorFlow or PyTorch for deep learning, or spaCy for NLP. Continually expanding your knowledge of these libraries can unlock new possibilities and enable you to tackle more complex data analysis tasks.

Develop Statistical Analysis Skills

To strengthen your data analysis capabilities, consider deepening your understanding of statistical concepts and methods. Familiarize yourself with hypothesis testing, regression analysis, ANOVA, and other statistical techniques commonly used in data analysis. This knowledge will enable you to make more informed decisions, effectively interpret data, and communicate insights with confidence.

Collaborate and Network

Engaging with the data analysis community can be invaluable for your growth. Join online forums, attend meetups or conferences, and connect with fellow data analysts and professionals in the field. Collaborating on projects, participating in data analysis competitions, or contributing to open-source projects can provide you with new perspectives, feedback, and opportunities to learn from others.

Consider Formal Education or Certifications

If you aspire to advance your career and take on more specialized roles in data science or machine learning, you

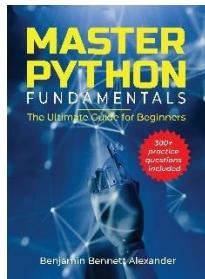
may consider pursuing formal education or certifications. Universities and online learning platforms offer degree programs, bootcamps, and certifications that provide in-depth knowledge and recognized credentials. Evaluate your career goals and research educational options that align with your aspirations.

Remember, the journey to becoming a skilled data analyst is ongoing and ever-evolving. Embrace a growth mindset, stay curious, and be proactive in exploring new challenges and opportunities. The field of data analysis is constantly evolving, and there is always more to learn and discover.

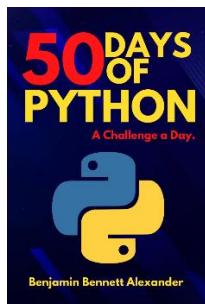
I hope "**50 Days of Data Analysis with Python**" has sparked your passion for data analysis and empowered you with the tools and confidence to embark on exciting data-driven endeavors. Now, it's up to you to continue your learning journey, apply your skills to real-world problems, and make a meaningful impact with data. Good luck, and may your future data analysis adventures be rewarding and fulfilling!

Other Books By Author

1. [Master Python Fundamentals: The Ultimate Guide for Beginners](#)



2. [50 Days of Python: A Challenge a Day](#)



3. [Python Tips and Tricks: A Collection of 100 Basic and Intermediate Tips and Tricks](#)

