THE OUTCOMES

Week 1: Basics of python programming
Installation of anaconda
variables and keywords
operators
conditional statements
looping statements
inbuilt and user defined functions

Week 2: basic data structures in python
List
Strings
tuple
dictionary
set

Week 3: Classes and objects
Creating classes and objects,Accessing attributes
Class Inheritance
Data hiding,Overriding and overloading
file Handling
Exception Handling

Week 4: Numpy and Pandas
Numpy arrays, Built in methods, array attribute and methods
array indexing and selection, broadcasting, fancy indexing, statistical operations on array

Introduction to pandas, Series, Dataframe
indexing of dataframe, operations on dataframes(count,unique, sum, describe, info, statistical operations)
reading and writing csv files using data frames, analysis of data using dataframe operations

Week 5: Matplotlib and seaborn

Introduction to matplotlib, plotting graphs , subplots
plotting bar charts, piecharts, histograms and scatterplots for data visualization
Introduction to seaborn, plotting using seaborn, distplot, pairplot
barplot, boxplot, countplot, whiskerplot using csv files
correlation matrix, heatmap , pairgrids

Week 6: Project Submission based on Concepts learned in python

ASSIGNMENT 1

Program 1:

Take a **string** and a **single character** as the inputs from the user. Your task is to search for the given character in the provided string. If the character is present in the string it should print its place value otherwise it should not return anything.

Note: If there are multiple characters repeating themselves in the given string then the output should be **all the place values** where the element is found.

Sample Test case 1:

```
anaconda
n
2
6
```

Sample Test case 2:

```
constructor
z
```

Explanation :

In test case 1, String = **"anaconda"** and the character to be searched is **'n'** and we can see n repeats itself in places 2 and 6. So the output is 2 and 6.

In test case 2, z is not present in the string **"constructor"**, so it didn't give any output.

Instructions:

- Your input and output must follow the input and output layout mentioned in the visible sample test case.

```python
def search_character(string, character):
    positions = []
    for i in range(len(string)):
        if string[i] == character:
            positions.append(i+1)

    return positions

string = input("Enter a string: ")
character = input("Enter a character to search: ")

result = search_character(string, character)


if result:
    for position in result:
        print(position)
else:
    print("Character not found in the string.")
```

```
Enter a string: constructor
Enter a character to search: z
Character not found in the string.
```

This code defines a Python function **search_character(string, character)** that takes two parameters: a string (**string**) and a character (**character**) to search for within that string. The purpose of this function is to find all the positions (indices) at which the given character appears in the string.

Function Definition:

def search_character(string, character):

This line defines a function named **search_character** that takes two parameters: **string** (the input string to search) and **character** (the character to search for within the string).

Initializing an Empty List:

positions = []

This line creates an empty list called **positions**. This list will be used to store the positions (indices) at which the given character is found in the string.

Loop Through the String:

for i in range(len(string)):

This **for** loop iterates through each index (**i**) in the range from 0 to the length of the input string (**string**).

Check if Character Matches:

```
if string[i] == character:
```

Inside the loop, this **if** statement checks if the character at the current index (**i**) in the string matches the specified **character**.

Append Positions:

```
positions.append(i+1)
```

If the character matches, the current index **i+1** (adding 1 because positions are usually 1-based, not 0-based) is appended to the **positions** list. This records the position of the matching character.

Return Positions:

```
return positions
```

Once the loop finishes, the function returns the **positions** list containing the positions of the matching character within the input string.

Input String and Character:

```
string = input("Enter a string: ") character = input("Enter a character to search: ")
```

These lines prompt the user to enter a string and a character to search for.

Function Call and Result Handling:

```
result = search_character(string, character)
```

The **search_character** function is called with the provided string and character, and the result is stored in the **result** variable.

Print Results:

```
if result:
    for position in result:
        print(position)
else:
    print("Character not found in the string.")
```

If the **result** list is not empty (meaning the character was found in the string), it iterates through the positions in **result** and prints each position. If the **result** list is empty, it prints a message indicating that the character was not found in the string.

In summary, this code defines a function that searches for a given character in a provided string and returns the positions where that character appears. It then takes user input for the string and character, performs the search, and prints the positions where the character was found or a message if it wasn't found.

Program 2

Prepare a system to manage a zoo. The zoo consists of different types of animals, including mammals, birds, and reptiles. Each animal has a name, an age, and a type-specific characteristic (e.g., mammals have fur, birds have feathers, and reptiles have scales).
Your task is to implement the following classes:

1. A class called "Animal" that serves as the base class for all types of animals. It should have the following methods:
    - A constructor that takes the animal's name and age as parameters and initializes the respective instance variables.
    - A method called "get_description" that returns a string describing the animal, including its name and age.

2. A class called "Mammal" that inherits from the "Animal" class. It should have an additional instance variable for fur color. Implement the following methods:
   - A constructor that takes the animal's name, age, and fur color as parameters. Initialize the instance variables accordingly.
   - Override the "get_description" method to include the fur color.
3. A class called "Bird" that inherits from the "Animal" class. It should have an additional instance variable for wing span. Implement the following methods:
   - A constructor that takes the animal's name, age, and wing span as parameters. Initialize the instance variables accordingly.
   - Override the "get_description" method to include the wing span.
4. A class called "Reptile" that inherits from the "Animal" class. It should have an additional instance variable for the number of legs. Implement the following methods:
   - A constructor that takes the animal's name, age, and number of legs as parameters. Initialize the instance variables accordingly.
   - Override the "get_description" method to include the number of legs.

```python
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def get_description(self):
        return f"Name: {self.name}, Age: {self.age}"


class Mammal(Animal):
    def __init__(self, name, age, fur_color):
        super().__init__(name, age)
        self.fur_color = fur_color

    def get_description(self):
        return f"Name: {self.name}, Age: {self.age}, Fur Color: {self.fur_color}"


class Bird(Animal):
    def __init__(self, name, age, wing_span):
        super().__init__(name, age)
        self.wing_span = wing_span

    def get_description(self):
        return f"Name: {self.name}, Age: {self.age}, Wing Span: {self.wing_span}"


class Reptile(Animal):
    def __init__(self, name, age, num_legs):
        super().__init__(name, age)
        self.num_legs = num_legs

    def get_description(self):
        return f"Name: {self.name}, Age: {self.age}, Number of Legs: {self.num_legs}"
```

```python
mammal = Mammal("Lion", 5, "Brown")
bird = Bird("Eagle", 3, 180)
reptile = Reptile("Snake", 2, 0)

print(mammal.get_description())
print(bird.get_description())
print(reptile.get_description())
```

```
Name: Lion, Age: 5, Fur Color: Brown
Name: Eagle, Age: 3, Wing Span: 180
Name: Snake, Age: 2, Number of Legs: 0
```

This code defines a basic object-oriented hierarchy of animal classes. Each class is a subclass of the **Animal** class, and they have additional attributes and methods specific to their respective categories (Mammals, Birds, and Reptiles). Let's break down the code:

1. **Animal Class:**
   - This is the base class that represents a generic animal.
   - It has an **__init__** method to initialize the **name** and **age** attributes of the animal.
   - It also has a **get_description** method that returns a string containing the name and age of the animal.
2. **Mammal Class (subclass of Animal):**
   - This class inherits from the **Animal** class.
   - It has an additional attribute **fur_color**.
   - Its **__init__** method extends the parent's **__init__** method and sets the **fur_color** attribute.
   - The **get_description** method is overridden to include the fur color in the description.
3. **Bird Class (subclass of Animal):**
   - This class also inherits from the **Animal** class.
   - It has an additional attribute **wing_span**.
   - Its **__init__** method extends the parent's **__init__** method and sets the **wing_span** attribute.
   - The **get_description** method is overridden to include the wing span in the description.
4. **Reptile Class (subclass of Animal):**
   - This class inherits from the **Animal** class.
   - It has an additional attribute **num_legs**.
   - Its **__init__** method extends the parent's **__init__** method and sets the **num_legs** attribute.

- The **get_description** method is overridden to include the number of legs in the description.

These classes follow the principles of object-oriented programming, particularly inheritance. Each subclass inherits attributes and methods from its parent class, and they can add new attributes and behaviors or override existing ones.

Overall, this code defines a simple class hierarchy representing different types of animals and demonstrates the use of inheritance and method overriding in Python's object-oriented programming paradigm.

Program 3:

Your task is to :

Create a Parent class **Polygon** with init method to initialize **no_of_sides** as data member. Define the methods **inputSides** and **dispSides** in the parent **class Polygon**.

inputSides: Asks the user to enter sides for n times, n represents the side of the selected polygon.

dispSides: Displays the sides entered by the user.

Derive classes **Triangle and Pentagon** from the base parent class **Polygon.**

In the init method (constructor) of the child class, call the init method of the parent class by passing **No. of sides** as an argument.

Constraints:

The sides must be integers.

Sample Test case:

```
1. Triangle ----> Display the menu
2. Pentagon
1 -----> Enter the option
Enter side 1 : 3 ------> Enter the sides
Enter side 2 : 4
Enter side 3 : 5
Side 1 is 3 -----> Display the sides
Side 2 is 4
Side 3 is 5
```

```python
class Polygon:
    def __init__(self, no_of_sides):
        self.no_of_sides = no_of_sides
        self.sides = []

    def inputSides(self):
        for i in range(self.no_of_sides):
            side = int(input(f"Enter side {i+1}: "))
            self.sides.append(side)

    def dispSides(self):
        for i in range(len(self.sides)):
            print(f"Side {i+1} is {self.sides[i]}")


class Triangle(Polygon):
    def __init__(self):
        super().__init__(3)


class Pentagon(Polygon):
    def __init__(self):
        super().__init__(5)

print("1. Triangle")
print("2. Pentagon")
option = int(input("Enter the option: "))


if option == 1:
    triangle = Triangle()
    triangle.inputSides()
    triangle.dispSides()
elif option == 2:
    pentagon = Pentagon()
    pentagon.inputSides()
    pentagon.dispSides()
else:
    print("Invalid option.")
```

```
1. Triangle
2. Pentagon
Enter the option: 1
Enter side 1: 3
Enter side 2: 4
Enter side 3: 5
Side 1 is 3
Side 2 is 4
Side 3 is 5
```

1. **Polygon Class:**
    - **This class is a base class representing a polygon.**
    - **It initializes with the number of sides and an empty list to store side lengths.**
    - **The inputSides method allows the user to input the lengths of each side.**
    - **The dispSides method displays the lengths of all sides.**
2. **Triangle Class (subclass of Polygon):**
    - **This class inherits from the Polygon class with a specific number of sides (3).**
    - **It does not have its own __init__ method, as it uses the parent's __init__ method with the specific number of sides (3).**
3. **Pentagon Class (subclass of Polygon):**
    - **Similar to the Triangle class, this class inherits from the Polygon class with a specific number of sides (5).**
4. **User Input:**
    - **The program prints options for the user to choose: Triangle or Pentagon.**

  - **The user inputs an option.**

5. **Object Creation and Interaction:**
   - **If the user chooses 1 (Triangle), a Triangle object is created.**
   - **If the user chooses 2 (Pentagon), a Pentagon object is created.**
   - **For the chosen object, the inputSides method is called to input side lengths.**
   - **The dispSides method is then called to display the entered side lengths.**
6. **Invalid Option:**
   - **If the user chooses an option other than 1 or 2, an "Invalid option" message is printed.**

In summary, this code defines a basic class hierarchy for polygons. The Polygon class provides a generic structure for any polygon with a variable number of sides. The Triangle and Pentagon classes inherit from the Polygon class, specifying the number of sides they have. The user can choose between creating a triangle or a pentagon, input their side lengths, and then display those side lengths. The code demonstrates inheritance, constructor extension (using super()), and method usage.

ASSIGNMENT 2

In this Exercise you will be given some Fake Data about some purchases done through Amazon! Just go ahead and follow the directions and try your best to answer the questions and complete the tasks. Feel free to reference the solutions. Most of the tasks can be solved in different ways. For the most part, the questions get progressively harder.

Please excuse anything that doesn't make "Real-World" sense in the dataframe, all the data is fake and made-up.

Also note that all of these questions can be answered with one line of code

Ques1 Check the head of the DataFrame.

Ques2 How many rows and columns are there?

Ques 3 What is the average Purchase Price?

Ques 4 What were the highest and lowest purchase prices

Ques 5 How many people have English 'en' as their Language of choice on the website?

Ques 6 How many people have the job title of "Lawyer" ?

Ques 7 How many people made the purchase during the AM and how many people made the purchase during PM ?

Ques 8 What are the 5 most common Job Titles?

Ques 9 Someone made a purchase that came from Lot: "90 WT" , what was the Purchase Price for this transaction?

Ques 10 What is the email of the person with the following Credit Card Number: 4926535242672853

Ques 11 *How many people have American Express as their Credit Card Provider *and* made a purchase above $95 ?

Ques 12 How many people have a credit card that expires in 2025?

Ques 13 What are the top 5 most popular email providers/hosts (e.g. gmail.com, yahoo.com, etc...)

```
df['Purchase Price'].mean()
```

```
50.347302
```

```
df['Purchase Price'].max()
df['Purchase Price'].min()
```

```
0.0
```

```
df[df['Language'] == 'en'].shape[0]
```

```
1098
```

```
df[df['Job'] == 'Lawyer'].shape[0]
```

```
30
```

```
df['AM or PM'].value_counts()
```

```
AM or PM
PM    5068
AM    4932
Name: count, dtype: int64
```

```
df['Job'].value_counts().head(5)
```

```
Job
Interior and spatial designer    31
Lawyer                           30
Social researcher                28
Purchasing manager               27
Designer, jewellery              27
Name: count, dtype: int64
```

```
import pandas as pd
```

```
file_path ="/Users/likhitayerra/Downloads/Ecommerce Purchases.csv"
```

```
df = pd.read_csv(file_path)
```

```
df.head()
```

1	9374 Jasmine Spurs Suite 508\nSouth John, TN 8...	28 rn	PM	Opera/8.93. (Windows 98; Win 9x 4.90; en-US) Pr...	Fletcher, Richards and Whitaker	3337758169645356	11/18	561	Mastercard	anthony41@reed.com	Drilling engineer	15.160.
2	Unit 0065 Box 5052\nDPO AP 27450	94 vE	PM	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT ...	Simpson, Williams and Pham	675957666125	08/19	699	JCB 16 digit	amymiller@morales-harrison.com	Customer service manager	132.207.1
3	7780 Julia Fords\nNew Stacy, WA 45798	36 vm	PM	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_0 ...	Williams, Marshall and Buchanan	6011578504430710	02/24	384	Discover	brent16@olson-robinson.info	Drilling engineer	30.250.
4	23012 Munoz Drive Suite 337\nNew Cynthia, TX 5...	20 IE	AM	Opera/9.58. (X11; Linux x86_64; it-IT) Presto/2...	Brown, Watson and Andrews	6011456623207998	10/25	678	Diners Club / Carte Blanche	christopherwright@gmail.com	Fine artist	24.140.

```
df.shape
```

```
(10000, 14)
```

```
df[df['Lot'] == '90 WT']['Purchase Price']
```

```
513    75.1
Name: Purchase Price, dtype: float64
```

```
df[df['Credit Card'] == 4926535242672853]['Email']
```

```
1234    bondellen@williams-garza.com
Name: Email, dtype: object
```

```
df[(df['CC Provider'] == 'American Express') & (df['Purchase Price'] > 95)].shape[0]
```

```
39
```

```
df[df['CC Exp Date'].apply(lambda x: x.split('/')[1] == '25')].shape[0]
```

```
1033
```

```
df['Email'].apply(lambda x: x.split('@')[1]).value_counts().head(5)
```

```
Email
hotmail.com     1638
yahoo.com       1616
gmail.com       1605
smith.com         42
williams.com      37
Name: count, dtype: int64
```

1. To check the head of the DataFrame, we can use the **.head()** method, which displays the first few rows of the DataFrame.

2. we can find the number of rows and columns in the DataFrame using the **.shape** attribute. The number of rows is the first element of the tuple, and the number of columns is the second element.

3. Calculate the average purchase price by using the **.mean()** function on the 'Purchase Price' column.

4. Find the highest and lowest purchase prices using the **.max()** and **.min()** functions on the 'Purchase Price' column, respectively.

5. Count the number of occurrences of 'en' in the 'Language' column using the **.value_counts()** function.

6. Count the number of occurrences of 'Lawyer' in the 'Job' column using the **.value_counts()** function.

7. Count the occurrences of 'AM' and 'PM' in the 'AM or PM' column using the **.value_counts()** function.

8. Use the **.value_counts()** function on the 'Job' column and select the top 5 values.

9. Filter the DataFrame for rows where the 'Lot' column matches '90 WT' and retrieve the 'Purchase Price' value.

10. Use boolean indexing to filter the DataFrame for rows where the 'Credit Card' column matches the given credit card number, then retrieve the corresponding 'Email' value.

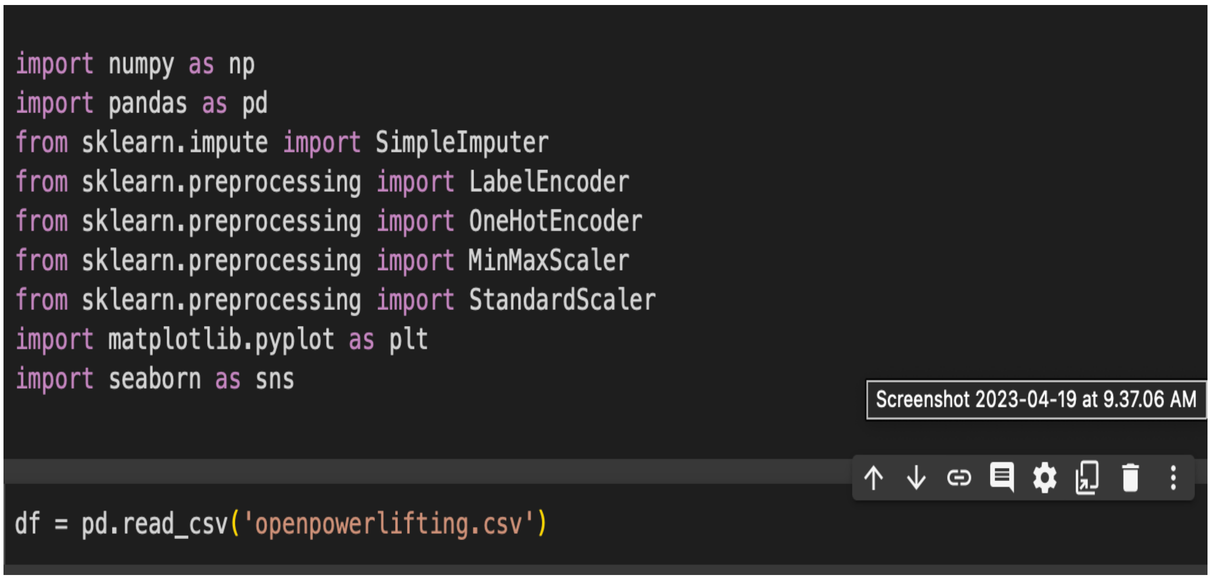
11. Use boolean indexing to filter the DataFrame for rows where the 'CC Provider' column is 'American Express' and the 'Purchase Price' is above $95, then count the number of occurrences.

12. Use boolean indexing to filter the DataFrame for rows where the 'CC Exp Date' column contains '25', then count the number of occurrences.

13. Extract the email domain (portion after '@') from each email address using string manipulation, then use **.value_counts()** to find the top 5 most common email domains.

ASSIGNMENT 3-THE PROJECT

FOR A POWERLIFTING DATABASE ,
HANDLING MISSING DATA,
CATEGORICAL DATA,
FEATURE SCALING,
DATAVISUALIZATION,
APPLYING MODEL



```
import numpy as np
import pandas as pd
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import seaborn as sns
```

```
df = pd.read_csv('openpowerlifting.csv')
```

Importing several libraries and modules commonly used in data preprocessing and analysis tasks. Here's a brief overview of what each of these libraries and modules does:

1. **numpy (import numpy as np):** NumPy is a library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a variety of mathematical functions to operate on these arrays.

2. **pandas (import pandas as pd):** Pandas is a powerful data manipulation and analysis library in Python. It provides data structures like Series and DataFrame, which allow you to

efficiently work with structured data, perform data cleaning, transformation, and analysis.

3. **SimpleImputer (from sklearn.impute import SimpleImputer):** SimpleImputer is a class from the scikit-learn library. It provides methods to fill missing values in a dataset using various strategies like mean, median, most frequent, etc.

4. **LabelEncoder (from sklearn.preprocessing import LabelEncoder):** LabelEncoder is another class from scikit-learn. It's used to encode categorical labels with numerical values, which can be useful when working with machine learning algorithms that require numerical inputs.

5. **OneHotEncoder (from sklearn.preprocessing import OneHotEncoder):** OneHotEncoder, also from scikit-learn, is used to convert categorical data into a binary (0/1) matrix representation. This is especially useful when dealing with categorical variables in machine learning models.

6. **MinMaxScaler (from sklearn.preprocessing import MinMaxScaler):** MinMaxScaler is used for feature scaling. It scales features to a specified range (usually between 0 and 1) by transforming them based on the minimum and maximum values of the feature.

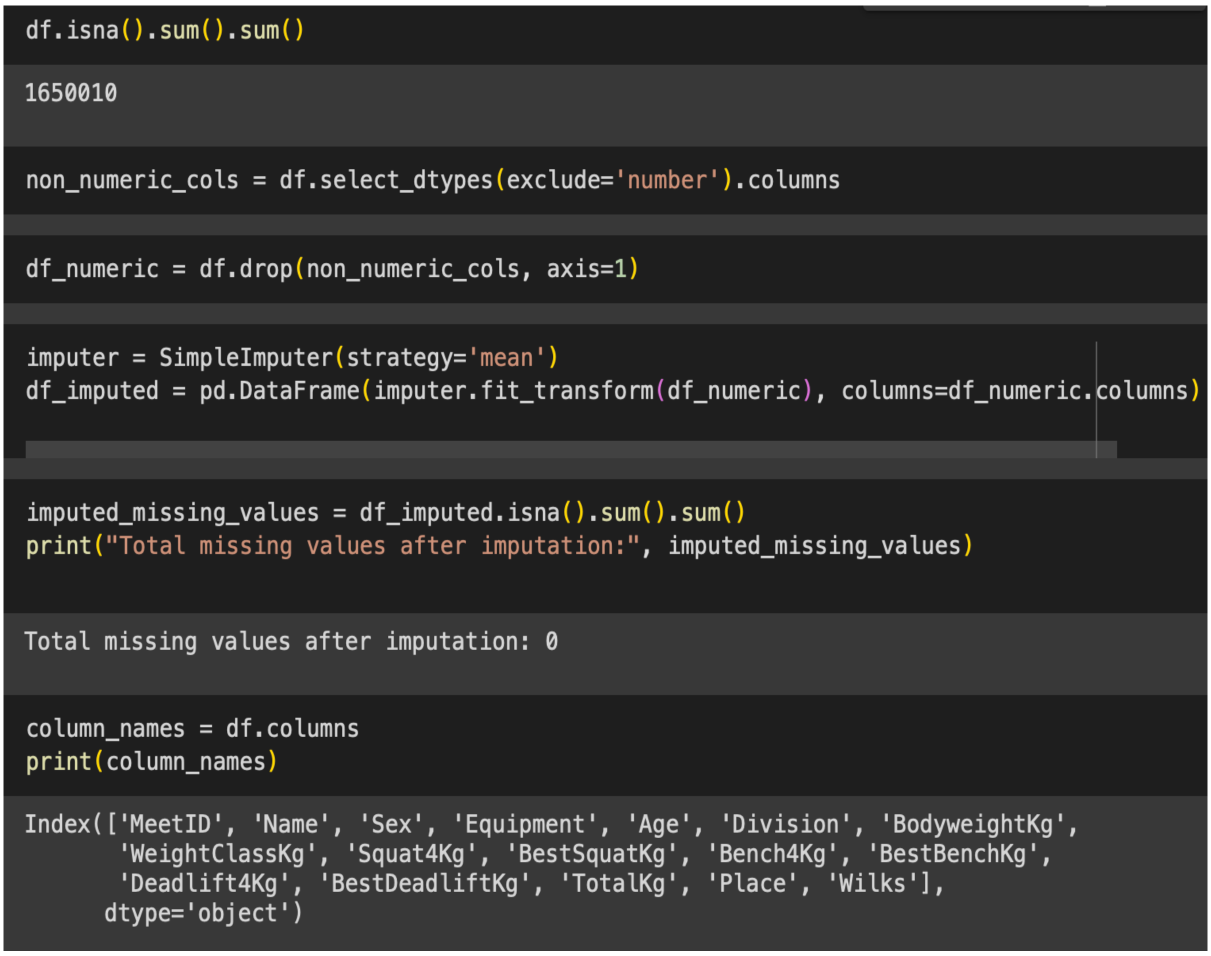
7. **StandardScaler (from sklearn.preprocessing import StandardScaler):** Similar to MinMaxScaler, StandardScaler is used for feature scaling. It transforms features to have a mean of 0 and a standard deviation of 1.

8. **matplotlib.pyplot (import matplotlib.pyplot as plt):** Matplotlib is a widely used plotting library in Python. The **pyplot** module provides functions that allow you to create various types of plots and visualizations.

9. **seaborn (import seaborn as sns):** Seaborn is a statistical data visualization library built on top of Matplotlib. It provides a higher-level interface for creating attractive and informative statistical graphics.

These libraries/modules are essential tools for data preprocessing, analysis, visualization, and preparing data for machine learning tasks. Depending on your project's requirements, you can use these tools to clean, transform, and visualize your data effectively.

Using the **pandas** library to read data from a CSV file named "openpowerlifting.csv" and stored it in a DataFrame named **df**. The **read_csv()** function in pandas is used to read data from a CSV file and create a DataFrame from it.



```
df.isna().sum().sum()
```

```
1650010
```

```
non_numeric_cols = df.select_dtypes(exclude='number').columns
```

```
df_numeric = df.drop(non_numeric_cols, axis=1)
```

```
imputer = SimpleImputer(strategy='mean')
df_imputed = pd.DataFrame(imputer.fit_transform(df_numeric), columns=df_numeric.columns)
```

```
imputed_missing_values = df_imputed.isna().sum().sum()
print("Total missing values after imputation:", imputed_missing_values)
```

```
Total missing values after imputation: 0
```

```
column_names = df.columns
print(column_names)
```

```
Index(['MeetID', 'Name', 'Sex', 'Equipment', 'Age', 'Division', 'BodyweightKg',
       'WeightClassKg', 'Squat4Kg', 'BestSquatKg', 'Bench4Kg', 'BestBenchKg',
       'Deadlift4Kg', 'BestDeadliftKg', 'TotalKg', 'Place', 'Wilks'],
      dtype='object')
```

1. The code snippet **df.isna().sum().sum()** calculates the total number of missing values in the entire DataFrame **df**.

- **df.isna()** creates a DataFrame of the same shape as **df**, where each cell contains a boolean value indicating whether the corresponding cell in **df** is missing (True) or not (False).

- **.sum()** is then called twice to first sum up the missing values for each column (which results in a Series containing the sum of missing values for each column), and then sum up the values in that Series, resulting in the total number of missing values in the entire DataFrame.

- This code can be useful to quickly get an overview of how much missing data is present in the dataset. If the result is 0, it means there are no missing values in the DataFrame. If it's greater than 0, then there are missing values that need to be addressed through techniques like imputation or further analysis.

2. The code **non_numeric_cols = df.select_dtypes(exclude='number').columns** selects all columns from the DataFrame **df** that have non-numeric (categorical or textual) data types and stores their column names in the **non_numeric_cols** variable.
   Here's what each part of the code does:

- **df.select_dtypes(exclude='number')** filters the columns in the DataFrame **df** based on their data types. The parameter **exclude='number'** specifies that only columns with non-numeric data types should be selected.

- **.columns** returns the column labels (names) of the filtered columns.

- After executing this code, the **non_numeric_cols** variable will contain a list of column names that correspond to the non-numeric columns in the DataFrame. This can be useful for further analysis or preprocessing tasks specific to non-numeric data.

3. The code **df_numeric = df.drop(non_numeric_cols, axis=1)** creates a new DataFrame named **df_numeric** by dropping the columns with non-numeric (categorical or textual) data types from the original DataFrame **df**.

- **df.drop(non_numeric_cols, axis=1)** drops the columns specified in the **non_numeric_cols** list from the DataFrame **df**. The parameter **axis=1** indicates that columns are being dropped, not rows.

- After executing this code, the **df_numeric** DataFrame will contain only the columns that have numeric data types. This can be helpful when you want to perform numerical calculations, statistical analysis, or machine learning on the data that consists of numeric values.

4. using the **SimpleImputer** from scikit-learn to fill missing values in the **df_numeric** DataFrame with the mean value of each column. Here's how the code works:

- **imputer = SimpleImputer(strategy='mean')**: This creates an instance of the **SimpleImputer** class with the strategy set to **'mean'**. This means that missing values will be replaced with the mean value of each column.

- **imputer.fit_transform(df_numeric)**: The **fit_transform** method of the **SimpleImputer** class is applied to the **df_numeric** DataFrame. This fits the imputer on the data and then transforms the data by filling in missing values with the mean value of each column. The result is a NumPy array.

- **pd.DataFrame(..., columns=df_numeric.columns)**: This converts the imputed NumPy array back to a pandas DataFrame. The **columns** parameter ensures that the column names are retained from the **df_numeric** DataFrame.

- After running this code, the **df_imputed** DataFrame will contain the same columns as the **df_numeric** DataFrame, but with missing values replaced by the mean value of each respective column. This imputation process can help ensure that the data is ready for analysis or modeling without the bias that could arise from dropping missing values

5. calculates and prints the total number of missing values in the **df_imputed** DataFrame after the imputation process. Here's how the code works:

- **df_imputed.isna().sum().sum()**: This code first uses the **.isna()** method to create a DataFrame of the same shape as **df_imputed**, where each cell contains a boolean value indicating whether the corresponding cell is missing (True) or not (False). Then **.sum()** is applied twice: the first **.sum()** calculates the sum of missing values for each column (resulting in a Series), and the second **.sum()** calculates the sum of missing values in that Series. This gives you the total number of missing values in the entire DataFrame after imputation.

- **print("Total missing values after imputation:", imputed_missing_values)**: This line prints out the total number of missing values in the **df_imputed** DataFrame after the imputation process.

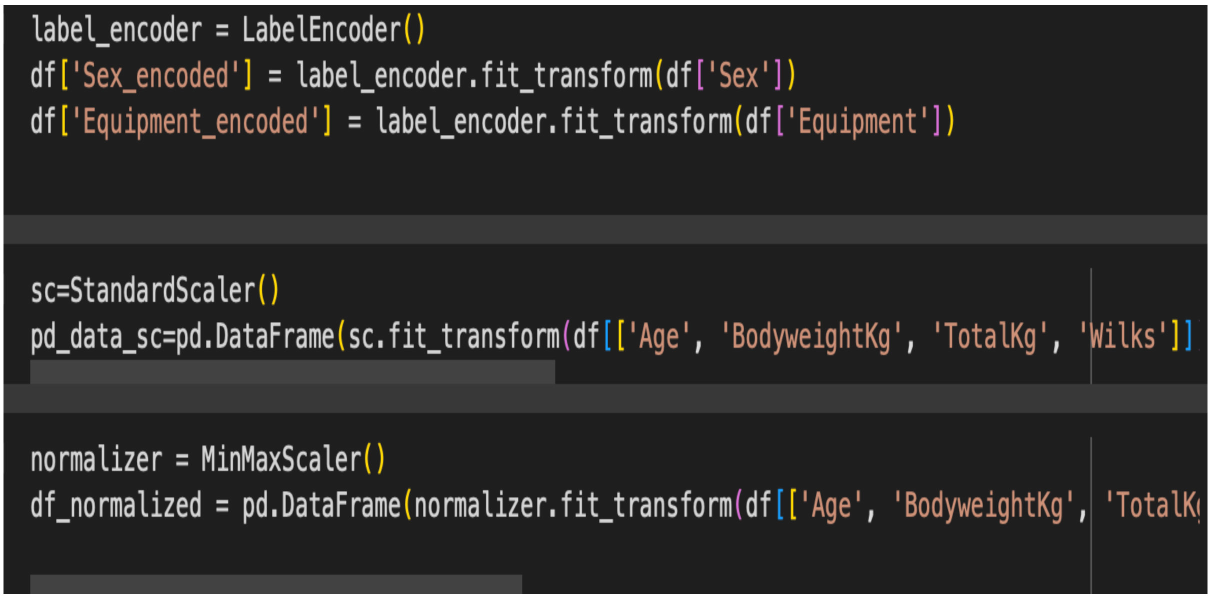
- The purpose of this code is to verify the effectiveness of the imputation process. If the printed total is 0, it indicates that all missing values have been successfully imputed. If it's greater than 0, there might still be missing values present in the dataset, which could require further investigation or action.

6. prints out the column names of the DataFrame **df**. Here's how the code works:

- **column_names = df.columns**: This line retrieves the column names of the DataFrame **df** and stores them in the **column_names** variable.

- **print(column_names)**: This line prints the contents of the **column_names** variable, which are the column names of the DataFrame **df**.

- Running this code will display a list of column names, providing you with an overview of the different attributes or features present in your dataset. This can be helpful for understanding the structure of your data and planning any further analysis or preprocessing tasks.



```
label_encoder = LabelEncoder()
df['Sex_encoded'] = label_encoder.fit_transform(df['Sex'])
df['Equipment_encoded'] = label_encoder.fit_transform(df['Equipment'])
```

```
sc=StandardScaler()
pd_data_sc=pd.DataFrame(sc.fit_transform(df[['Age', 'BodyweightKg', 'TotalKg', 'Wilks']]
```

```
normalizer = MinMaxScaler()
df_normalized = pd.DataFrame(normalizer.fit_transform(df[['Age', 'BodyweightKg', 'TotalK
```

- using the **LabelEncoder** from scikit-learn to encode categorical columns 'Sex' and 'Equipment' in the DataFrame **df**. Here's what each part of the code does:

1. **label_encoder = LabelEncoder()**: This creates an instance of the **LabelEncoder** class, which is used to encode categorical labels (strings) into numerical labels.

2. **df['Sex_encoded'] = label_encoder.fit_transform(df['Sex'])**: This line applies the label encoder to the 'Sex' column of the DataFrame **df**. The **.fit_transform()** method first fits the encoder to the values in the 'Sex' column and then transforms those values into encoded numerical labels. The encoded labels are stored in a new column called 'Sex_encoded'.

3. **df['Equipment_encoded'] = label_encoder.fit_transform(df['Equipment'])**: Similar to the previous line, this line applies the label encoder to the 'Equipment' column of the DataFrame **df**. The encoded labels are stored in a new column called 'Equipment_encoded'.

After running this code, the DataFrame **df** will have two new columns, 'Sex_encoded' and 'Equipment_encoded', containing the encoded numerical values for the corresponding categorical columns. This encoding is necessary when working with machine learning algorithms that require numerical inputs, as it converts categorical data into a format that the algorithms can process.

- using the **StandardScaler** from scikit-learn to standardize specific columns in the DataFrame **df**. Here's what each part of the code does:

1. **sc = StandardScaler()**: This creates an instance of the **StandardScaler** class, which is used to standardize features by removing the mean and scaling to unit variance.

2. **df[['Age', 'BodyweightKg', 'TotalKg', 'Wilks']]**: This part selects a subset of columns from the DataFrame **df**, namely 'Age', 'BodyweightKg', 'TotalKg', and 'Wilks'.

3. **pd_data_sc = pd.DataFrame(sc.fit_transform(...), columns=['Age_normalized', 'BodyweightKg_normalized', 'TotalKg_normalized', 'Wilks_normalized'])**: This line applies the standard scaler to the selected columns and creates a new DataFrame **pd_data_sc** with the standardized values. The **.fit_transform()** method first fits the scaler to the selected columns and then transforms those columns into standardized values. The resulting DataFrame contains the standardized columns with names 'Age_normalized', 'BodyweightKg_normalized', 'TotalKg_normalized', and 'Wilks_normalized'.

4. After running this code, the **pd_data_sc** DataFrame will contain the standardized versions of the selected columns. Standardization is useful for preparing the data for machine learning algorithms that benefit from having features on similar scales. It helps prevent features with larger scales from dominating the algorithm's behavior.

- using the **MinMaxScaler** from scikit-learn to normalize specific columns in the DataFrame **df**. Here's how the code works:

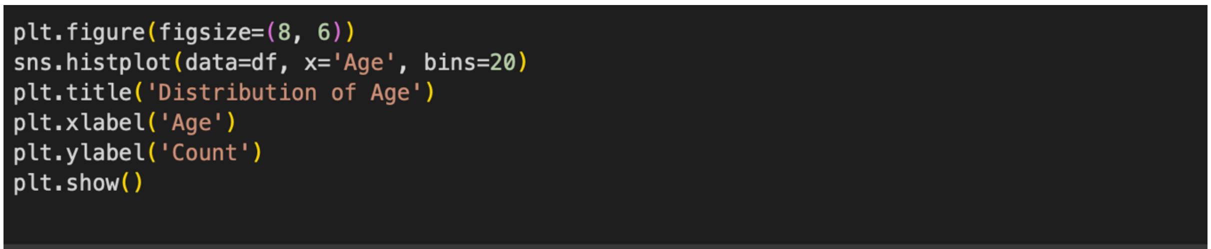
1. **normalizer = MinMaxScaler()**: This creates an instance of the **MinMaxScaler** class, which is used to normalize features by scaling them to a specified range, usually between 0 and 1.

2. **df[['Age', 'BodyweightKg', 'TotalKg', 'Wilks']]**: This part selects a subset of columns from the DataFrame **df**, namely 'Age', 'BodyweightKg', 'TotalKg', and 'Wilks'.

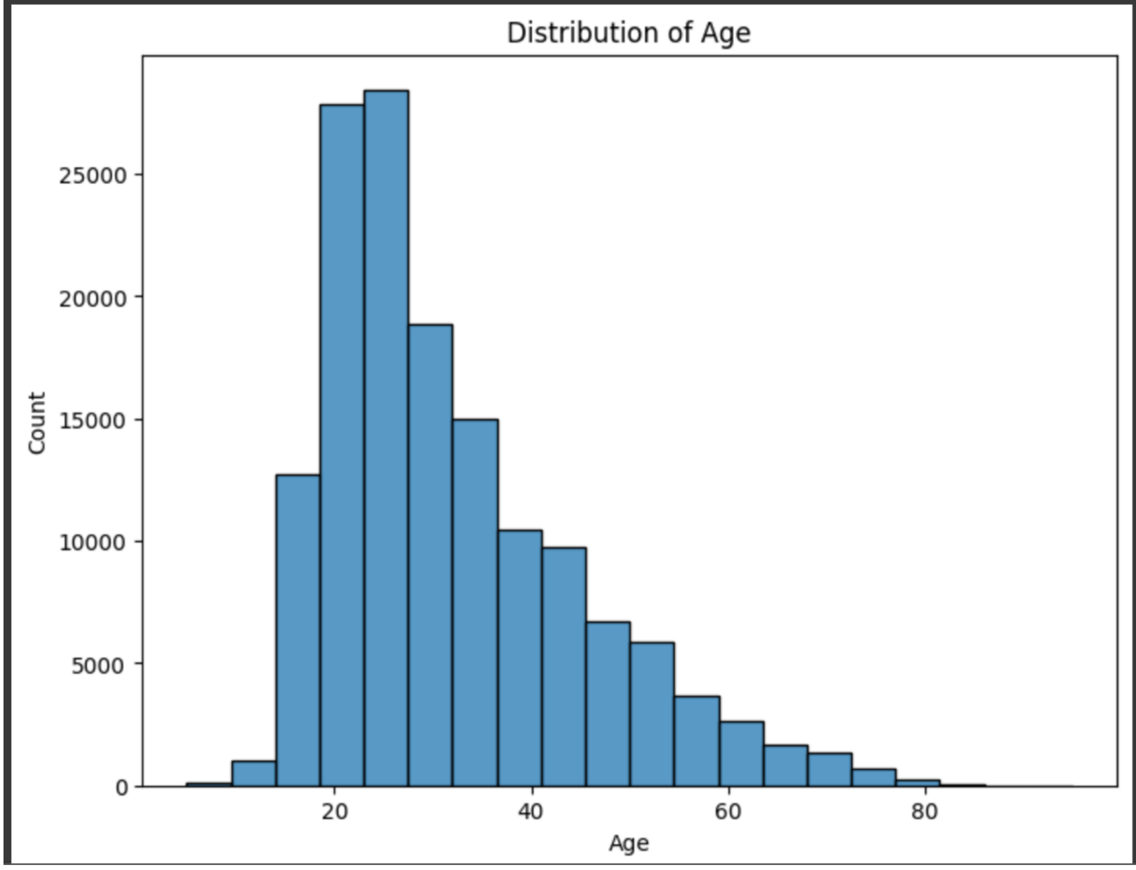
3. **pd_data_normalized = pd.DataFrame(normalizer.fit_transform(...), columns=['Age_normalized', 'BodyweightKg_normalized', 'TotalKg_normalized', 'Wilks_normalized'])**: This line

applies the min-max scaler to the selected columns and creates a new DataFrame **df_normalized** with the normalized values. The **.fit_transform()** method first fits the scaler to the selected columns and then transforms those columns into normalized values. The resulting DataFrame contains the normalized columns with names 'Age_normalized', 'BodyweightKg_normalized', 'TotalKg_normalized', and 'Wilks_normalized'.

4. After running this code, the **df_normalized** DataFrame will contain the normalized versions of the selected columns. Normalization is useful for bringing all features onto a similar scale, which can improve the performance of machine learning algorithms that rely on distance or magnitude calculations.



```
plt.figure(figsize=(8, 6))
sns.histplot(data=df, x='Age', bins=20)
plt.title('Distribution of Age')
plt.xlabel('Age')
plt.ylabel('Count')
plt.show()
```



Using **matplotlib** and **seaborn** to create a histogram plot of the distribution of ages in the DataFrame **df**. Here's what each part of the code does:

1. **plt.figure(figsize=(8, 6))**: This line creates a new figure for the plot with a specific figsize of 8x6 inches.

2. **sns.histplot(data=df, x='Age', bins=20)**: This line uses Seaborn's **histplot** function to create a histogram plot. The **data** parameter specifies the DataFrame to use, the **x** parameter specifies the column to plot ('Age' in this case), and the **bins** parameter specifies the number of bins (bars) in the histogram.

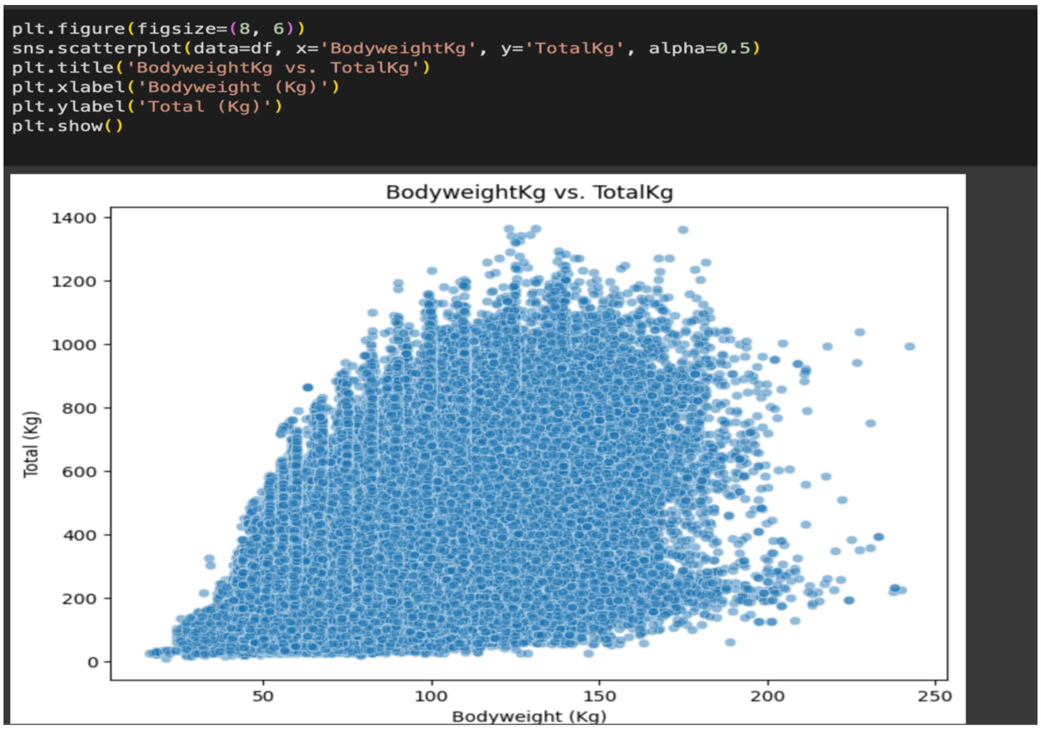
3. **plt.title('Distribution of Age')**: This line sets the title of the plot to 'Distribution of Age'.

4. **plt.xlabel('Age')**: This line sets the label for the x-axis to 'Age'.

5. **plt.ylabel('Count')**: This line sets the label for the y-axis to 'Count'.

6. **plt.show()**: This line displays the plot.

Overall, the code generates a histogram that visualizes the distribution of ages in the dataset. The x-axis represents the age values, the y-axis represents the count of occurrences, and the bars (bins) show how many data points fall into each age range. This type of plot is helpful for understanding the distribution and central tendency of a numerical variable in a dataset.



```
plt.figure(figsize=(8, 6))
sns.scatterplot(data=df, x='BodyweightKg', y='TotalKg', alpha=0.5)
plt.title('BodyweightKg vs. TotalKg')
plt.xlabel('Bodyweight (Kg)')
plt.ylabel('Total (Kg)')
plt.show()
```

Using **matplotlib** and **seaborn** to create a scatter plot of the relationship between 'BodyweightKg' and 'TotalKg' in the DataFrame **df**. Here's how the code works:

1. **plt.figure(figsize=(8, 6))**: This line creates a new figure for the plot with a specific figsize of 8x6 inches.

2. **sns.scatterplot(data=df, x='BodyweightKg', y='TotalKg', alpha=0.5)**: This line uses Seaborn's **scatterplot** function to create a scatter plot. The **data** parameter specifies the DataFrame to use, the **x** parameter specifies the column for the x-axis ('BodyweightKg' in this case), the **y** parameter specifies the column for the y-axis ('TotalKg' in this case), and the **alpha** parameter sets the transparency level of the points.

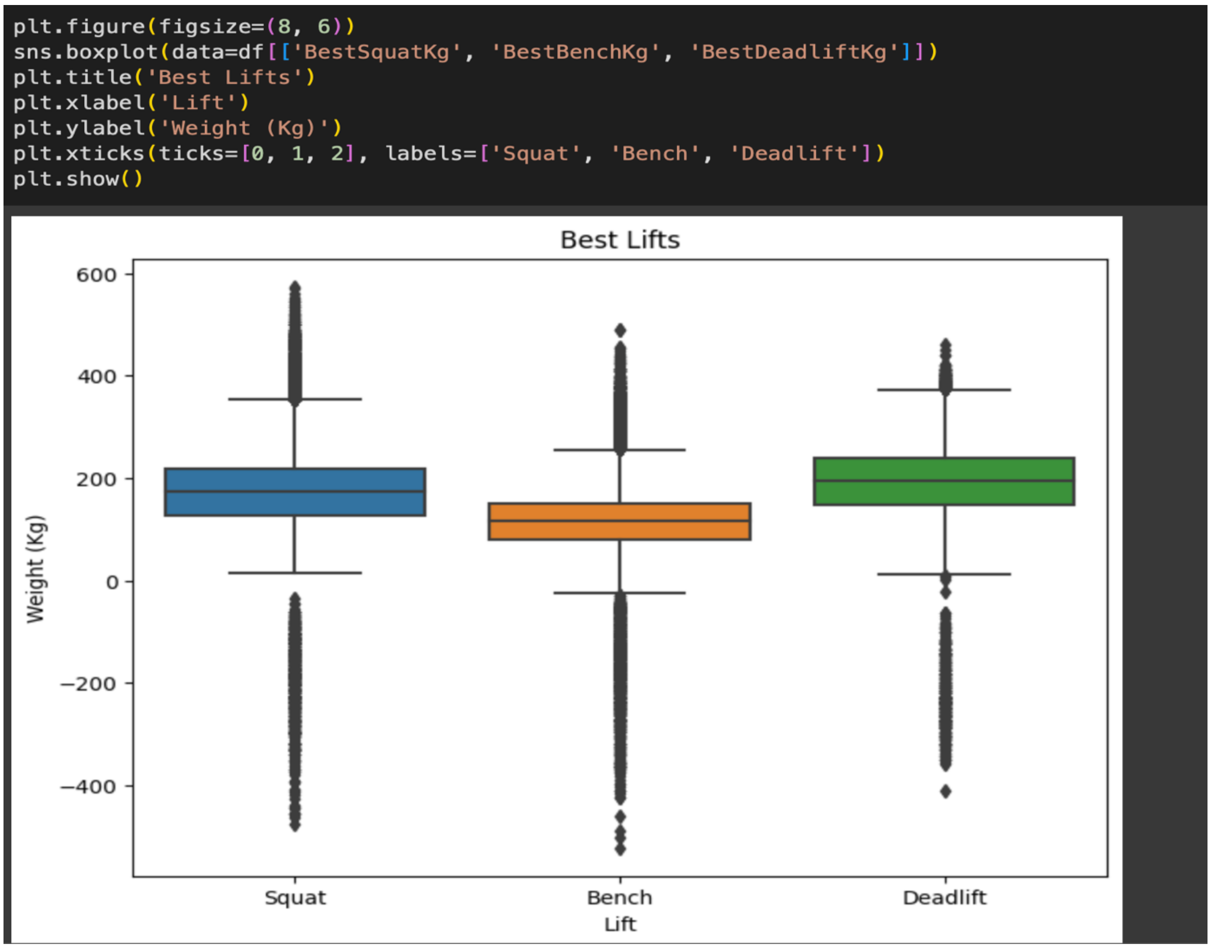
3. **plt.title('BodyweightKg vs. TotalKg')**: This line sets the title of the plot to 'BodyweightKg vs. TotalKg'.

4. **plt.xlabel('Bodyweight (Kg)')**: This line sets the label for the x-axis to 'Bodyweight (Kg)'.

5. **plt.ylabel('Total (Kg)')**: This line sets the label for the y-axis to 'Total (Kg)'.

6. **plt.show()**: This line displays the plot.

The scatter plot shows individual points where the x-axis represents 'BodyweightKg' and the y-axis represents 'TotalKg'. Each point represents a data point in the dataset. The **alpha** parameter has been set to 0.5 to make the points slightly transparent, which can be useful when there are many overlapping points. This type of plot helps visualize the relationship and distribution between two numerical variables.



```
plt.figure(figsize=(8, 6))
sns.boxplot(data=df[['BestSquatKg', 'BestBenchKg', 'BestDeadliftKg']])
plt.title('Best Lifts')
plt.xlabel('Lift')
plt.ylabel('Weight (Kg)')
plt.xticks(ticks=[0, 1, 2], labels=['Squat', 'Bench', 'Deadlift'])
plt.show()
```

Using **matplotlib** and **seaborn** to create a box plot of the best lifts (Squat, Bench, Deadlift) in the DataFrame **df**. Here's how the code works:

1. **plt.figure(figsize=(8, 6))**: This line creates a new figure for the plot with a specific figsize of 8x6 inches.

2. **sns.boxplot(data=df[['BestSquatKg', 'BestBenchKg', 'BestDeadliftKg']])**: This line uses Seaborn's **boxplot** function to create a box plot. The **data** parameter specifies the DataFrame to use, and the list of column names (**['BestSquatKg', 'BestBenchKg', 'BestDeadliftKg']**) specifies the columns for which box plots should be created.

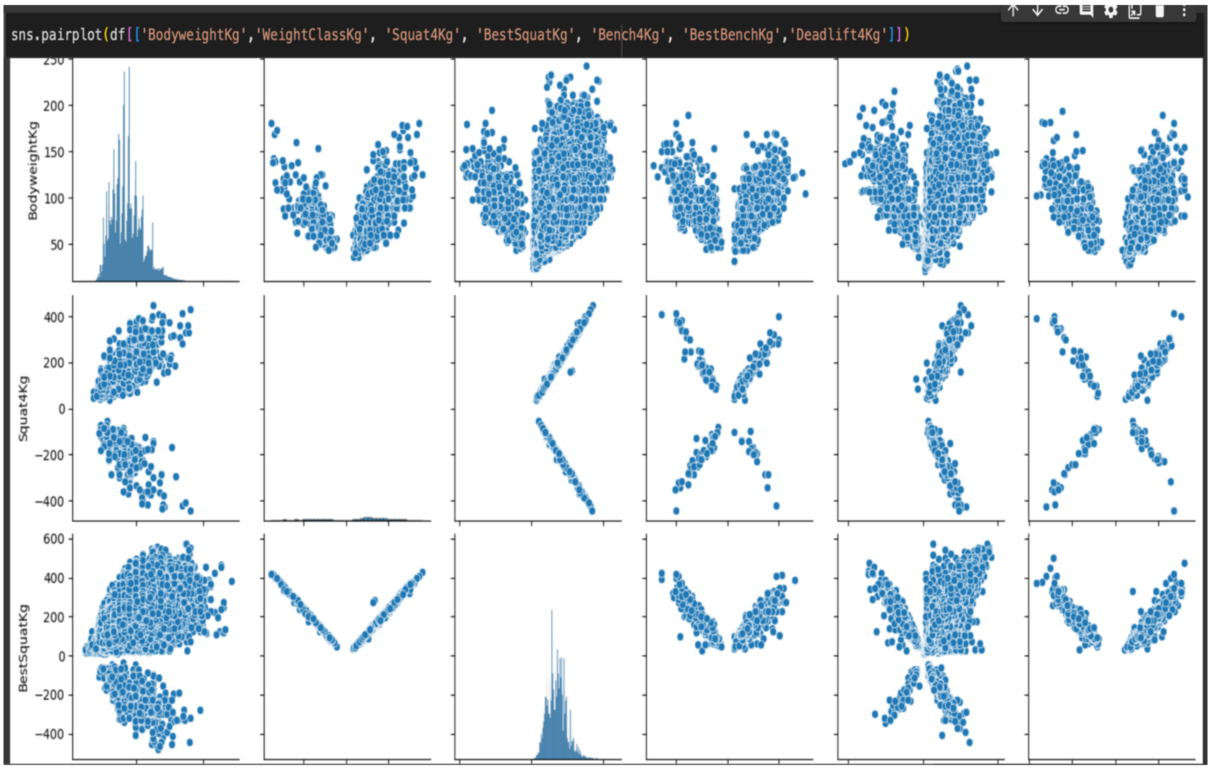
3. **plt.title('Best Lifts')**: This line sets the title of the plot to 'Best Lifts'.
4. **plt.xlabel('Lift')**: This line sets the label for the x-axis to 'Lift'.

5. **plt.ylabel('Weight (Kg)')**: This line sets the label for the y-axis to 'Weight (Kg)'.

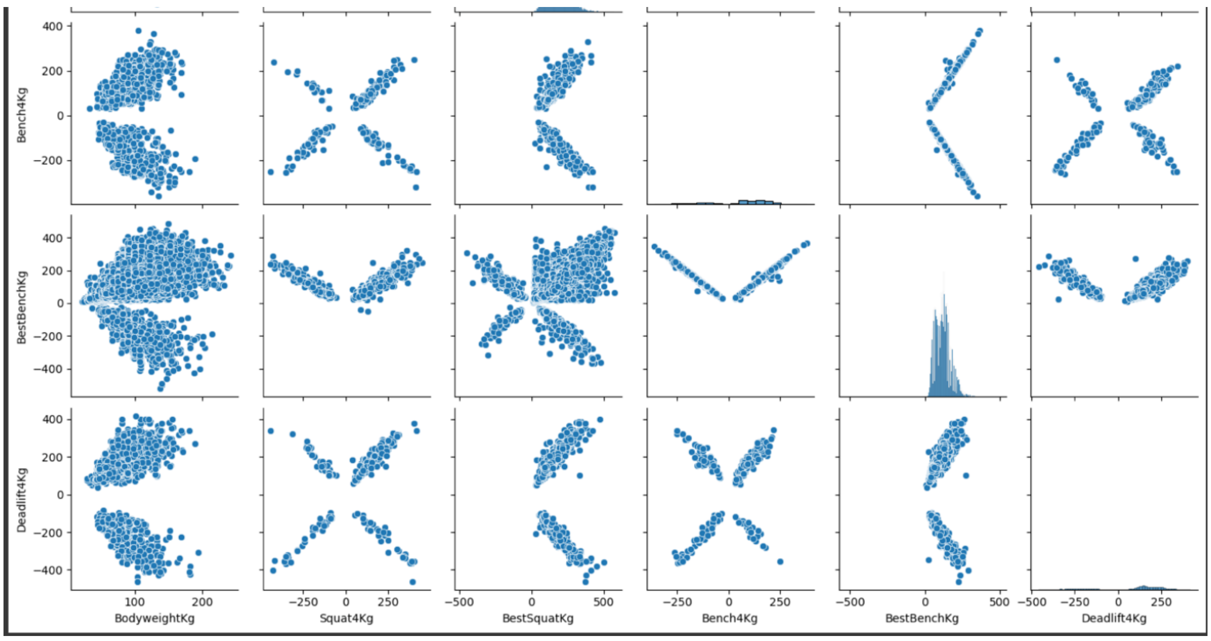
6. **plt.xticks(ticks=[0, 1, 2], labels=['Squat', 'Bench', 'Deadlift'])**: This line customizes the x-axis tick positions and labels. It maps the tick positions to the list of labels ['Squat', 'Bench', 'Deadlift'], so each box on the plot corresponds to the respective lift type.

7. **plt.show()**: This line displays the plot.

The box plot shows the distribution of best lift weights for each lift type (Squat, Bench, Deadlift). The boxes represent the interquartile range (IQR) of the data, the horizontal lines inside the boxes represent the medians, and the whiskers extend to show the data range within a certain distance from the median. This type of plot is useful for comparing the distribution of a numerical variable across different categories.



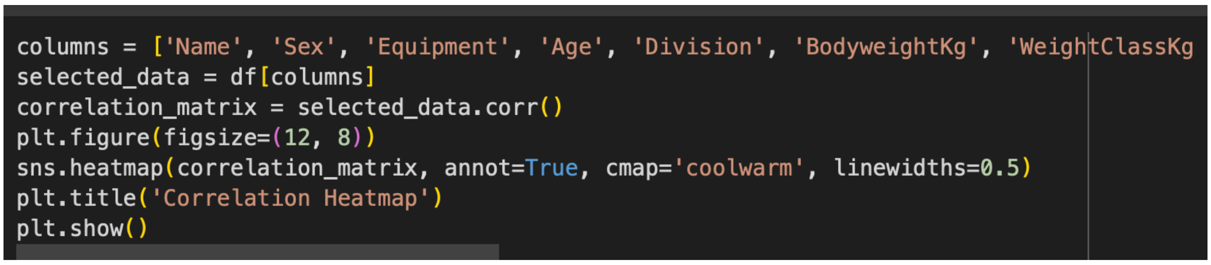
```
sns.pairplot(df[['BodyweightKg','WeightClassKg', 'Squat4Kg', 'BestSquatKg', 'Bench4Kg', 'BestBenchKg','Deadlift4Kg']])
```



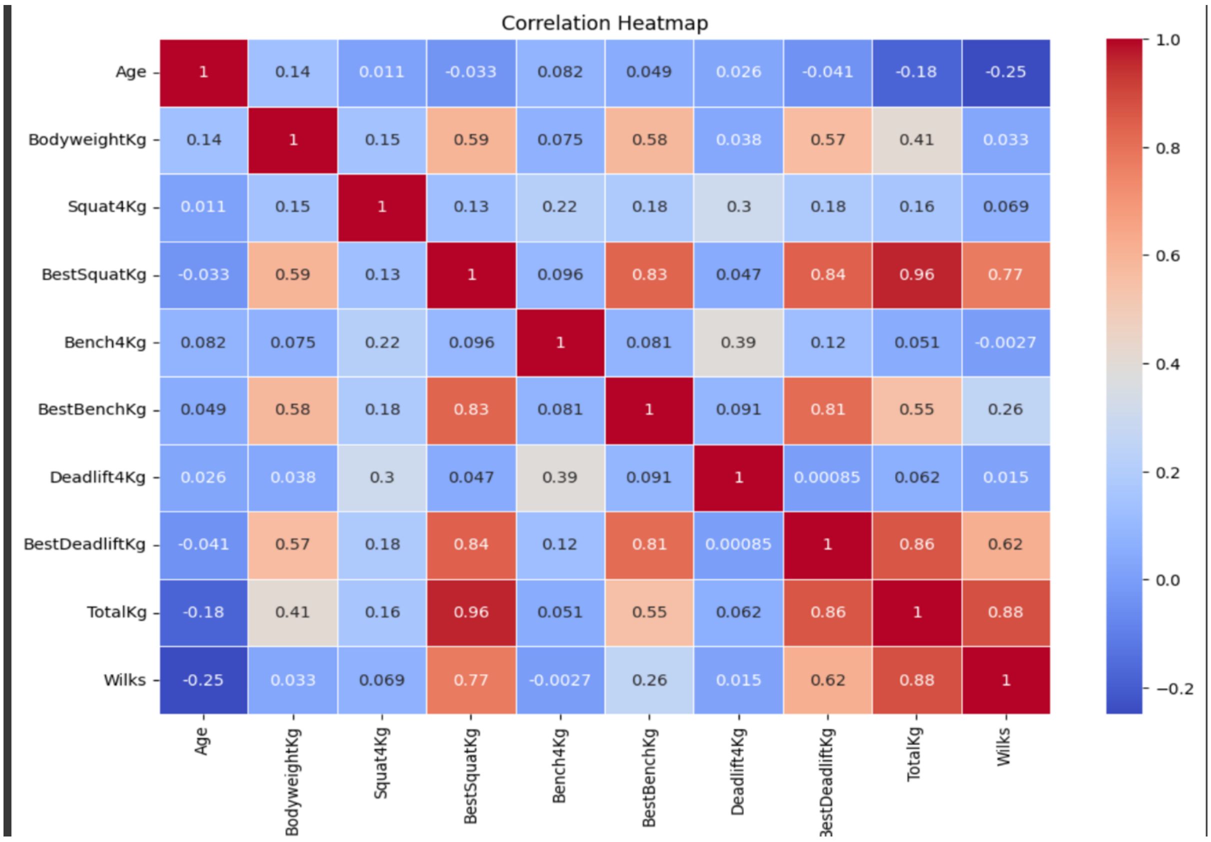
Using **seaborn** to create a pair plot for selected columns in the DataFrame **df**. Here's what the code does:

1. **sns.pairplot(df[['BodyweightKg','WeightClassKg', 'Squat4Kg', 'BestSquatKg', 'Bench4Kg', 'BestBenchKg','Deadlift4Kg']])**: This line uses Seaborn's **pairplot** function to create a matrix of scatter plots. The **data** parameter specifies the DataFrame to use, and the list of column names within double square brackets specifies the columns for which scatter plots should be created.

A pair plot creates scatter plots between all pairs of the selected columns. It's useful for visually understanding relationships and distributions among multiple numerical variables. Each scatter plot in the matrix shows how one variable changes with respect to another variable.



```
columns = ['Name', 'Sex', 'Equipment', 'Age', 'Division', 'BodyweightKg', 'WeightClassKg
selected_data = df[columns]
correlation_matrix = selected_data.corr()
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()
```



Generating a correlation heatmap using **seaborn** to visualize the correlation coefficients between selected columns in the DataFrame **df**. Here's how the code works:

1. **columns = ['Name', 'Sex', 'Equipment', ...]**: This line defines a list of column names that you want to include in the correlation analysis. These columns will be used to create the correlation matrix.

2. **selected_data = df[columns]**: This line creates a new DataFrame **selected_data** by selecting only the columns specified in the **columns** list from the original DataFrame **df**.
3. **correlation_matrix = selected_data.corr()**: This line calculates the correlation coefficients between the columns in the **selected_data** DataFrame, creating a correlation matrix.

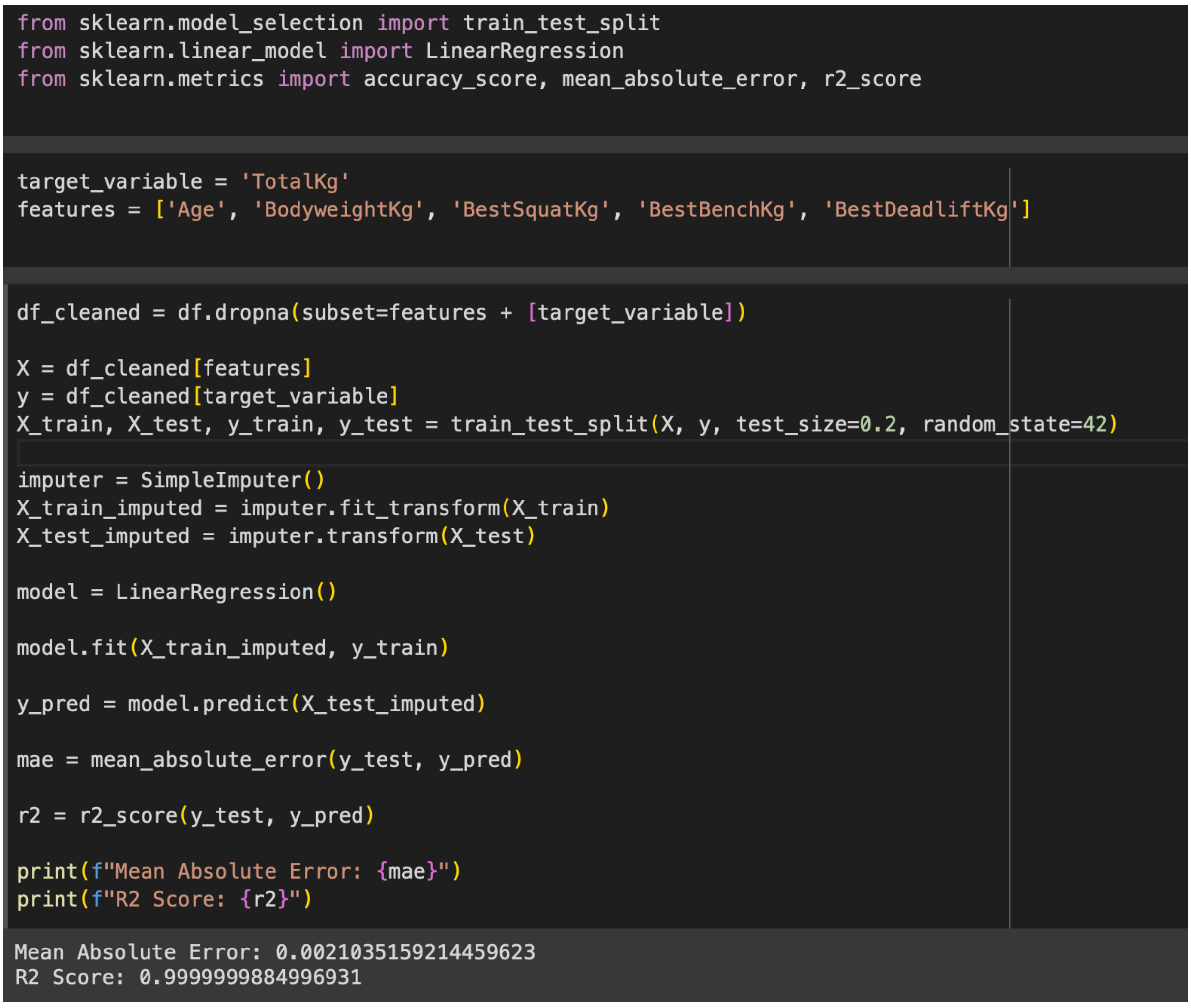
4. **plt.figure(figsize=(12, 8))**: This line creates a new figure for the plot with a specific figsize of 12x8 inches.

5. **sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', linewidths=0.5)**: This line uses Seaborn's **heatmap** function to create a heatmap of the correlation matrix. The **correlation_matrix** is provided as input, and **annot=True** adds the correlation coefficients to the cells, **cmap='coolwarm'** sets the color map, and **linewidths=0.5** adds white lines between cells for better readability.

6. **plt.title('Correlation Heatmap')**: This line sets the title of the plot to 'Correlation Heatmap'.

7. **plt.show()**: This line displays the plot.

The heatmap visually represents the correlation between pairs of numerical columns. Positive correlations are indicated by warmer colors (tending towards red), while negative correlations are indicated by cooler colors (tending towards blue). The intensity of the color reflects the strength of the correlation. This type of plot is useful for understanding relationships between numerical variables and identifying patterns in the data.



```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import accuracy_score, mean_absolute_error, r2_score
```

```python
target_variable = 'TotalKg'
features = ['Age', 'BodyweightKg', 'BestSquatKg', 'BestBenchKg', 'BestDeadliftKg']
```

```python
df_cleaned = df.dropna(subset=features + [target_variable])

X = df_cleaned[features]
y = df_cleaned[target_variable]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

imputer = SimpleImputer()
X_train_imputed = imputer.fit_transform(X_train)
X_test_imputed = imputer.transform(X_test)

model = LinearRegression()

model.fit(X_train_imputed, y_train)

y_pred = model.predict(X_test_imputed)

mae = mean_absolute_error(y_test, y_pred)

r2 = r2_score(y_test, y_pred)

print(f"Mean Absolute Error: {mae}")
print(f"R2 Score: {r2}")
```

```
Mean Absolute Error: 0.0021035159214459623
R2 Score: 0.9999999884996931
```

1.

- **train_test_split (from sklearn.model_selection import train_test_split):** This function is used to split your dataset into training and testing sets. It helps you evaluate the performance of a model on unseen data by creating a separation between the data used for training and the data used for testing.

- **LinearRegression (from sklearn.linear_model import LinearRegression): LinearRegression** is a class from scikit-learn that implements linear regression, a statistical method used for modeling the relationship between a dependent variable and one or more independent variables. It's commonly used for predicting numerical values.

- **accuracy_score (from sklearn.metrics import accuracy_score):** The **accuracy_score** function is used to evaluate the accuracy of classification models. Since you're performing linear regression, this metric might not be directly applicable, as it's more suited for classification problems.

- **mean_absolute_error (from sklearn.metrics import mean_absolute_error): mean_absolute_error** calculates the mean absolute error between the actual values and the predicted values. It's a measure of how well the predictions of the model match the actual data.

- **r2_score (from sklearn.metrics import r2_score): r2_score**, also known as the coefficient of determination, measures the proportion of the variance in the dependent variable that is predictable from the independent variables. It's a measure of how well the model fits the data.

- These modules are essential for performing regression analysis and evaluating the performance of a linear regression model using various metrics. If you have a specific dataset and modeling task in mind, you can use these tools to train a linear regression model, make predictions, and assess its performance.

2. **target_variable:** This is the variable for trying to predict for model. In this case, target variable is 'TotalKg', which suggests that we're trying to predict the total weight lifted by an individual.

features: These are the independent variables or attributes that you believe might influence the target variable. You've specified a list of features that include 'Age', 'BodyweightKg', 'BestSquatKg', 'BestBenchKg', and 'BestDeadliftKg'. These are likely attributes that you think could contribute to predicting the 'TotalKg' of weight lifted.

With these definitions, we're setting up our linear regression analysis to use these features to predict the 'TotalKg' target variable. The goal is to build a model that can learn the relationship between these features and the target variable, allowing us to make predictions on new data.

3.

- **df_cleaned = df.dropna(subset=features + [target_variable]):** This line drops rows from the DataFrame **df** where there are missing values in the specified features and the target variable. This prepares a cleaned dataset for your analysis by removing instances with missing data.

- **X = df_cleaned[features] and y = df_cleaned[target_variable]:** These lines separate the cleaned dataset into the feature matrix **X** (independent variables) and the target vector **y** (dependent variable).

- **X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42):** This line splits the data into training and testing sets using the **train_test_split** function. **test_size=0.2** specifies that 20% of the data will be used for testing, and **random_state=42** sets the random seed for reproducibility.

- **imputer = SimpleImputer():** This creates an instance of the **SimpleImputer** class, which will be used to impute any missing values in the training and testing data.

- **X_train_imputed = imputer.fit_transform(X_train) and X_test_imputed = imputer.transform(X_test):** These lines use the imputer to fill missing values in the training and testing feature matrices.

- **model = LinearRegression():** This creates an instance of the **LinearRegression** class, which is the linear regression model you will be using.

- **model.fit(X_train_imputed, y_train):** This line fits the linear regression model to the imputed training data.

- **y_pred = model.predict(X_test_imputed):** This line uses the trained model to make predictions on the imputed testing data.

- **mae = mean_absolute_error(y_test, y_pred):** This calculates the mean absolute error between the actual target values (**y_test**) and the predicted values (**y_pred**).

- **r2 = r2_score(y_test, y_pred):** This calculates the R-squared score, which represents the goodness of fit of the model on the test data.

- **print(f"Mean Absolute Error: {mae}") and print(f"R2 Score: {r2}"):** These lines print out the calculated mean absolute error and R-squared score.

- Overall, our code trains a linear regression model using cleaned and imputed data, makes predictions, and then evaluates the model's performance using mean absolute error and R-squared score. These metrics help assess how well our model is performing in predicting the 'TotalKg' target variable.