

AI & ML





Table of Contents

[**Chapter One: Python Refresher 2**](#_heading=h.90wvikz4j5p1)

[1. Python Basics: Variables, Data Types, and Control Structures 2](#_heading=h.8ot7nr4vx7k)

[2. Functions and Modules 7](#_heading=h.90vecjggh0nk)

[3. Introduction to NumPy for Numerical Operations 10](#_heading=h.1ysli4x95ufw)

[4. Introduction to Pandas for Data Manipulation 12](#_heading=h.3pbpnkk9uaxu)

[5. Basic Data Visualization with Matplotlib 14](#_heading=h.hih502yt24yk)

[6. Writing and Using Python Scripts and Modules 15](#_heading=h.5uxu2ooul0qc)

[7. Common Python Pitfalls and Best Practices 17](#_heading=h.2x27bpy67isa)

[**Chapter Two: Data Preprocessing and Feature Engineering 18**](#_heading=h.z33q04qwpqa4)

[1. Data manipulation and viewing with pandas 18](#_heading=h.wyx4om19ug0c)

[2. Handling Missing Data: Deletion, Imputation Techniques 19](#_heading=h.b1lo2iutiyu8)

[3. Feature Scaling: Normalization and Standardization 23](#_heading=h.am3bkhr44cyw)

[4. Encoding Categorical Variables: One-Hot Encoding, Label Encoding 25](#_heading=h.cjjsezghjf3o)

[5. Basic Feature Engineering: Creating New Features, Handling Outliers 26](#_heading=h.9zfkhwafs95)

[6. Detecting and Treating Outliers in Data 27](#_heading=h.d3sp0haqlgjb)

[7. Data Transformation Techniques for Improving Model Performance 28](#_heading=h.z21ibv1frjtg)

# 

# 

# **Chapter One: Python Refresher**

## 1. Python Basics: Variables, Data Types, and Control Structures

**Introduction to Python**

Python is a high-level, interpreted programming language known for its readability and versatility. It is used in various domains such as web development, data science, machine learning, automation, and more.

**Variables and Data Types:**

* **Variables:** Containers for storing data values. In Python, you don’t need to declare a variable’s type explicitly. The interpreter infers the type based on the value assigned to the variable.

x = 10 # integer

y = 15.5 # float

name = "Alice" # string

is\_active = True # boolean

**Data Types:**

* **Integers**: Whole numbers, positive or negative.

a = 5

b = -3

* **Floats**: Numbers with a decimal point.

pi = 3.14159

gravity = 9.8

* **Strings**: Sequences of characters.

greeting = "Hello, World!"

* **Booleans**: `True` or `False`.

is\_open = True

* **Complex** **Numbers**: Numbers with a real and imaginary part.

complex\_num = 2 + 3j

**Lists**

* **Introduction:** Lists are ordered, mutable collections that can hold items of different data types.

fruits = ["apple", "banana", "cherry"]

* **Accessing Elements:** Use indices to access list elements.

print(fruits[0]) # Output: apple

* **Modifying** **Elements**: Lists are mutable, so elements can be changed.

fruits[1] = "blueberry"

* **List** **Methods**:

fruits.append("orange") # Adds an item to the end

fruits.remove("banana") # Removes an item by value

fruits.pop(1) # Removes an item by index

* **List** **Comprehensions**: Concise way to create lists.

squares = [x\*\*2 for x in range(10)]

**Tuples**

* **Introduction:** Tuples are ordered, immutable collections.

point = (10, 20)

* **Accessing** **Elements**: Use indices to access tuple elements.

print(point[0]) # Output: 10

* **Immutability**: Tuples cannot be changed after creation.

# point[1] = 30 # This will raise an error

* **Tuple Packing and Unpacking:**

coordinates = 1, 2

x, y = coordinates

**Sets**

* **Introduction:** Sets are unordered collections of unique elements.

fruits = {"apple", "banana", "cherry"}

* **Adding Elements:** Use `add()` method.

python

fruits.add("orange")

* **Set Operations:** Union, intersection, difference.

python

set1 = {1, 2, 3}

set2 = {3, 4, 5}

union\_set = set1.union(set2) # {1, 2, 3, 4, 5}

intersection\_set = set1.intersection(set2) # {3}

**Dictionaries**

* **Introduction:** Dictionaries are collections of key-value pairs.

python

person = {"name": "Alice", "age": 25}

* **Accessing Elements:** Use keys to access values.

print(person["name"]) # Output: Alice

* **Modifying Elements:** Dictionaries are mutable.

person["age"] = 26

* **Dictionary Methods:**

person.keys() # Returns all keys

person.values() # Returns all values

person.items() # Returns all key-value pairs

* **Dictionary Comprehensions**: Concise way to create dictionaries.

squares = {x: x\*\*2 for x in range(10)}

**Type Conversion:** Python allows for type conversion using functions like `int()`, `float()`, `str()`, etc.

x = 5 # integer

y = 3.2 # float

z = x + y # automatically converted to float

print(z) # Output: 8.2

# Explicit conversion

x = 5

y = "10"

z = x + int(y) # converting string to int

print(z) # Output: 15

**Control Structures:** Control structures allow you to control the flow of your program.

* **Conditional Statements:**

age = 20

if age >= 18:

print("You are an adult.")

elif age < 18 and age >= 13:

print("You are a teenager.")

else:

print("You are a child.")

**Loops:**

* **For Loop:**

for i in range(5): # 0 to 4

print(i)

* **While Loop:**

count = 0

while count < 5:

print(count)

count += 1

**Loop Control Statements:**

for i in range(10):

if i == 3:

continue # skips the rest of the loop for i = 3

if i == 8:

break # exits the loop when i = 8

print(i)

**List Comprehensions:** A concise way to create lists.

squares = [x\*\*2 for x in range(10)]

print(squares) # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

**Exercises:**

1. Create a variable `radius` with a value of 7.5, calculate the area of a circle using this radius, and print the result. (Use the formula: `Area = π \* radius^2`).
2. Write a program that prints all the numbers from 1 to 50 that are divisible by 3.
3. Write a list comprehension that creates a list of the squares of all even numbers between 1 and 20.
4. Create a list of your favorite fruits and print each fruit.
5. Create a tuple with your top three favorite numbers and print the second number.
6. Create a set of your favorite hobbies and add a new hobby to the set.
7. Create a dictionary with keys as subjects and values as your scores in those subjects. Print the score of a particular subject.

## 2. Functions and Modules

**Functions**

Functions are blocks of reusable code that perform a specific task. They help to make your code more organized and modular.

**Defining a Function:**

def greet(name):

return f"Hello, {name}!"

print(greet("Alice")) # Output: Hello, Alice!

**Function Arguments:**

* **Positional Arguments**: Arguments that are passed in a specific order.

def add(a, b):

return a + b

print(add(5, 3)) # Output: 8

* **Keyword Arguments:** Arguments passed by explicitly naming the parameter.

def introduce(name, age):

return f"My name is {name} and I am {age} years old."

print(introduce(age=25, name="Bob")) # Output: My name is Bob and I am 25 years old.

* **Default Arguments:** Arguments that have a default value if not provided.

def greet(name, message="Hello"):

return f"{message}, {name}!"

print(greet("Charlie")) # Output: Hello, Charlie!

print(greet("Charlie", "Hi")) # Output: Hi, Charlie!

* **Variable-Length Arguments:** Allows you to pass an arbitrary number of arguments.

def sum\_all(\*args):

return sum(args)

print(sum\_all(1, 2, 3, 4)) # Output: 10

**Returning Values:** Functions can return a value using the `return` statement.

def multiply(x, y):

return x \* y

result = multiply(6, 7)

print(result) # Output: 42

```

**Anonymous Functions (Lambda Expressions):** Lambda functions are small, unnamed functions defined using the `lambda` keyword.

add = lambda x, y: x + y

print(add(2, 3)) # Output: 5

**Recursion:** A function that calls itself to solve a problem.

def factorial(n):

if n == 1:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5)) # Output: 120

**Modules**

Modules are files containing Python code (functions, variables, etc.) that can be imported into other Python programs.

**Importing Modules:**

import math

print(math.sqrt(16)) # Output: 4.0

**Custom Modules:** You can create your own module by saving a `.py` file and importing it into another script.

# In my\_module.py

def greet(name):

return f"Hello, {name}!"

# In another script

import my\_module

print(my\_module.greet("Diana")) # Output: Hello, Diana!

**Exercises:**

1. Write a function `is\_even()` that checks if a number is even. Use this function to filter all even numbers from a list of numbers.
2. Create a custom module `calculator.py` with functions for addition, subtraction, multiplication, and division. Import this module into another script and use its functions.

## 3. Introduction to NumPy for Numerical Operations

**Introduction to NumPy**

NumPy is a powerful library for numerical computing in Python. It provides support for arrays, matrices, and a wide range of mathematical functions.

**Creating Arrays:**

import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr) # Output: [1 2 3 4 5]

matrix = np.array([[1, 2, 3], [4, 5, 6]])

print(matrix)

# Output:

# [[1 2 3]

# [4 5 6]]

**Array Operations:** NumPy allows for element-wise operations on arrays.

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

print(arr1 + arr2) # Output: [5 7 9]

print(arr1 \* arr2) # Output: [ 4 10 18]

**Broadcasting:** Broadcasting allows you to perform operations on arrays of different shapes.

arr = np.array([1, 2, 3])

print(arr + 5) # Output: [6 7 8]

**Array Reshaping:** Reshaping allows you to change the shape of an array without changing its data.

arr = np.array([1, 2, 3, 4, 5,

6])

reshaped\_arr = arr.reshape((2, 3))

print(reshaped\_arr)

# Output:

# [[1 2 3]

# [4 5 6]]

**Statistical Operations:**

data = np.array([1, 2, 3, 4, 5])

print(np.mean(data)) # Output: 3.0

print(np.std(data)) # Output: 1.4142135623730951

**Exercises:**

1. Create a NumPy array containing numbers from 1 to 50. Reshape this array into a 5x10 matrix.
2. Use NumPy to create an array of random numbers and compute the mean, median, and standard deviation.

## 4. Introduction to Pandas for Data Manipulation

**Introduction to Pandas**

Pandas is a powerful data analysis library that provides data structures like Series and DataFrames, which are essential for handling and analyzing structured data.

**Creating DataFrames:**

import pandas as pd

data = {

'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [24, 27, 22, 32],

'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']

}

df = pd.DataFrame(data)

print(df)

**DataFrame Operations:** Accessing data, modifying data, and performing operations.

print(df['Name']) # Accessing a column

print(df.iloc[1]) # Accessing a row by index

print(df.loc[df['Age'] > 25]) # Filtering data based on a condition

df['Age'] += 1 # Modifying data

print(df)

**Data Cleaning:** Handling missing values, removing duplicates, and transforming data.

df = pd.DataFrame({

'A': [1, 2, np.nan, 4],

'B': [5, np.nan, np.nan, 8],

'C': [10, 11, 12, 13]

})

df.fillna(0, inplace=True) # Replace NaN with 0

df.dropna(inplace=True) # Drop rows with NaN

print(df)

**Grouping and Aggregation:** Grouping data and performing aggregate functions.

df = pd.DataFrame({

'Department': ['HR', 'Engineering', 'HR', 'Engineering'],

'Employee': ['Alice', 'Bob', 'Charlie', 'David'],

'Salary': [50000, 60000, 55000, 65000]

})

grouped = df.groupby('Department').mean()

print(grouped)

**Exercises:**

1. Create a Pandas DataFrame from a CSV file. Filter the data to show only rows where a certain condition is met.
2. Group data in a DataFrame by a categorical column and calculate the mean for another column.

## 5. Basic Data Visualization with Matplotlib

**Introduction to Matplotlib**

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

**Creating Basic Plots:**

import matplotlib.pyplot as plt

# Line Plot

plt.plot([1, 2, 3, 4], [10, 20, 25, 30])

plt.title("Simple Line Plot")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.show()

# Scatter Plot

plt.scatter([1, 2, 3, 4], [10, 20, 25, 30])

plt.title("Simple Scatter Plot")

plt.show()

**Customizing Plots:** Adding titles, labels, legends, and customizing the appearance.

plt.plot([1, 2, 3, 4], [10, 20, 25, 30], marker='o', linestyle='--', color='r')

plt.title("Customized Line Plot")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.grid(True)

plt.show()

**Saving Plots:**

plt.plot([1, 2, 3, 4], [10, 20, 25, 30])

plt.savefig("plot.png") # Save as PNG

plt.savefig("plot.pdf") # Save as PDF

**Exercises:**

1. Create a bar plot that shows the number of students in different classes.
2. Create a scatter plot of two variables from a dataset and customize the plot with labels, title, and a grid.

## 6. Writing and Using Python Scripts and Modules

**Writing Python Scripts**

Scripts are files containing Python code that can be executed from the command line.

**Basic Script:**

# script.py

def greet(name):

return f"Hello, {name}!"

if \_\_name\_\_ == "\_\_main\_\_":

print(greet("World"))

**Run this script from the command line:**

python script.py

**Using Python Modules and Packages**

**Creating a Custom Module:** Save a Python file (e.g., `my\_module.py`) with some functions or classes, and import it in another script.

# my\_module.py

def add(a, b):

return a + b

# main.py

import my\_module

print(my\_module.add(5, 3)) # Output: 8

**Virtual Environments:** Virtual environments are isolated Python environments that allow you to manage dependencies for different projects.

**Creating and Activating a Virtual Environment:**

python -m venv myenv

source myenv/bin/activate # On Windows: myenv\Scripts\activate

**Installing Packages in a Virtual Environment:**

pip install numpy pandas matplotlib

**Exercises:**

1. Create a Python script that reads a CSV file, processes the data, and saves the result to a new file.
2. Set up a virtual environment for a project, install the necessary packages, and create a simple script that uses those packages.

## 7. Common Python Pitfalls and Best Practices

**Common Pitfalls**

**Mutable vs Immutable Data Types:** Understand how mutable (e.g., lists, dictionaries) and immutable (e.g., tuples, strings) types behave in Python.

# Mutable example

list\_a = [1, 2, 3]

list\_b = list\_a

list\_b.append(4)

print(list\_a) # Output: [1, 2, 3, 4]

# Immutable example

str\_a = "Hello"

str\_b = str\_a

str\_b += " World"

print(str\_a) # Output: "Hello"

**Best Practices:**

**Writing Clean Code:**

* Follow PEP 8 guidelines for code style.
* Use meaningful variable names.
* Comment your code where necessary.

**Exercises:**

1. Write a Python script that includes both mutable and immutable data types. Modify the variables and observe the differences.

# 

# **Chapter Two: Data Preprocessing and Feature Engineering**

## 1. Data manipulation and viewing with pandas

To replace a value in a DataFrame, you can use the replace() function or directly assign a new value to a specific cell using loc[]. Below is an example where we replace the NaN value in the 'Magic Power' column for 'Legolas' with a specific value, say 85.

import pandas as pd

import numpy as np

# Create a sample dataset

data = {

'Character': ['Gandalf', 'Legolas', 'Gimli', 'Saruman', 'Frodo', np.nan, 'Aragorn'],

'Magic Power': [95, np.nan, 10, 90, np.nan, 75, 50],

'Agility': [80, 95, 60, 70, 85, np.nan, 80],

'Wisdom': [100, 85, 70, 95, np.nan, 60, 90]

}

df = pd.DataFrame(data)

# Replace NaN value in 'Magic Power' column for 'Legolas' with 85

df.loc[df['Character'] == 'Legolas', 'Magic Power'] = 85

print(df)

**Simple Example of groupby**

You can group the data by a specific column and apply aggregation functions. Here's an example where we group by the 'Character' column and calculate the mean of the other columns:

# Group by 'Character' and calculate the mean of 'Magic Power', 'Agility', and 'Wisdom'

grouped\_df = df.groupby('Character').mean()

print(grouped\_df)

**Aggregate Examples**

You can use the agg() function to apply multiple aggregation functions at once. Here's an example where we group by the 'Character' column and apply different aggregation functions:

# Group by 'Character' and apply different aggregation functions

aggregate\_df = df.groupby('Character').agg({

'Magic Power': ['mean', 'max'],

'Agility': ['mean', 'min'],

'Wisdom': ['mean', 'sum']

})

print(aggregate\_df)

These examples should help you understand how to replace values, group data, and perform aggregation operations using pandas.

## 

## 2. Handling Missing Data: Deletion, Imputation Techniques

**Understanding Missing Data**

In datasets related to fantasy characters, missing data can occur due to incomplete records or information that was never collected. Just like in any other dataset, it is important to understand the nature of missing data and handle it appropriately to avoid biased models.

For example, suppose we have a dataset of characters from different fantasy races (Wizards, Elves, Dwarves) with attributes such as Magic Power, Agility, and Wisdom. Missing data in this context could mean missing values for some characters’ abilities.

**Techniques for Handling Missing Data**

* **Deletion Methods**

Deletion is one approach to handling missing data. In our fantasy characters dataset, we might want to remove characters with incomplete data, but this could lead to a loss of valuable information, especially if many records are missing data.

**Example:** Continuing with our fantasy characters dataset:

import pandas as pd

import numpy as np

# continuing with the previously created dataframe (df)

# Load the CSV to demonstrate deletion

df = pd.read\_csv('fantasy\_characters.csv')

# Listwise deletion: Remove rows with any missing values

listwise\_deleted\_df = df.dropna()

print("Listwise Deletion:\n", listwise\_deleted\_df)

# Pairwise deletion: Drop specific columns with too many missing values

pairwise\_deleted\_df = df.dropna(axis=1)

print("\nPairwise Deletion:\n", pairwise\_deleted\_df)

**Imputation Techniques**

Imputation is a more sophisticated method where missing data is filled with substituted values. In our fantasy dataset, we can use various imputation techniques to estimate the missing Magic Power, Agility, or Wisdom of characters.

* **Imputation using mean and median**

We will work with certain columns here

**Example:** Continuing with our fantasy characters dataset:

from sklearn.impute import SimpleImputer

from sklearn.impute import KNNImputer

# Mean Imputation for 'Magic Power'

mean\_imputer = SimpleImputer(strategy='mean')

df['Magic Power'] = mean\_imputer.fit\_transform(df[['Magic Power']])

# Median Imputation for 'Agility'

median\_imputer = SimpleImputer(strategy='median')

df['Agility'] = median\_imputer.fit\_transform(df[['Agility']])

print("After Mean and Median Imputation:\n", df)

* **Imputation using KNN**

Let us try & work by imputing the entire dataframe at one go, but KNNImputer in scikit-learn works only with numerical data like strings (e.g., the 'Character' column in your dataframe). To resolve this issue, you can either drop the non-numeric columns before applying the KNN imputer or separately handle the imputation for numeric and non-numeric columns. Here’s how you can do it:

**Approach 1: Drop the Non-Numeric Columns for KNN Imputation**

import pandas as pd

import numpy as np

from sklearn.impute import KNNImputer

# Create a sample dataset

data = {

'Character': ['Gandalf', 'Legolas', 'Gimli', 'Saruman', 'Frodo', np.nan, 'Aragorn'],

'Magic Power': [95, np.nan, 10, 90, np.nan, 75, 50],

'Agility': [80, 95, 60, 70, 85, np.nan, 80],

'Wisdom': [100, 85, 70, 95, np.nan, 60, 90]

}

df = pd.DataFrame(data)

# Separating numeric data

numeric\_cols = df.select\_dtypes(include=[np.number])

# KNN Imputation for numeric columns

knn\_imputer = KNNImputer(n\_neighbors=2)

df\_knn = pd.DataFrame(knn\_imputer.fit\_transform(numeric\_cols), columns=numeric\_cols.columns)

# Combine the imputed numeric data with the original non-numeric data

df\_final = df.copy()

df\_final.update(df\_knn)

print("\nAfter KNN Imputation:\n", df\_final)

**Approach 2: Impute Non-Numeric Data Separately**

If you want to keep the 'Character' column intact and handle the imputation only for numeric columns:

import pandas as pd

import numpy as np

from sklearn.impute import KNNImputer

# Create a sample dataset

data = {

'Character': ['Gandalf', 'Legolas', 'Gimli', 'Saruman', 'Frodo', np.nan, 'Aragorn'],

'Magic Power': [95, np.nan, 10, 90, np.nan, 75, 50],

'Agility': [80, 95, 60, 70, 85, np.nan, 80],

'Wisdom': [100, 85, 70, 95, np.nan, 60, 90]

}

df = pd.DataFrame(data)

# Separating numeric data

numeric\_cols = df.select\_dtypes(include=[np.number])

# KNN Imputation for numeric columns

knn\_imputer = KNNImputer(n\_neighbors=2)

df\_knn = pd.DataFrame(knn\_imputer.fit\_transform(numeric\_cols), columns=numeric\_cols.columns)

# Combine the imputed numeric data with the original non-numeric data

df\_combined = df.drop(columns=numeric\_cols.columns).join(df\_knn)

print("\nAfter KNN Imputation:\n", df\_combined)

In both approaches, the non-numeric columns like 'Character' are preserved as is, while the numeric columns are imputed using the KNN method.

## 

## 3. Feature Scaling: Normalization and Standardization

**Importance of Feature Scaling**

In machine learning, feature scaling is critical for ensuring that all features contribute equally to the model. For example, in a dataset of fantasy characters, Magic Power might range from 0 to 100, while Wisdom might have a narrower range. Without scaling, Magic Power could dominate the model, leading to biased predictions.

**Normalization**

Normalization scales features to a fixed range, typically [0, 1]. This is particularly useful for algorithms that rely on distance metrics, such as KNN or neural networks.

**Example:** Let's normalize the features in our modified fantasy characters dataset:

from sklearn.preprocessing import MinMaxScaler

# Normalization

scaler = MinMaxScaler()

df[['Magic Power', 'Agility', 'Wisdom']] = scaler.fit\_transform(df[['Magic Power', 'Agility', 'Wisdom']])

print("After Normalization:\n", df)

In this example, all numerical features are scaled to the range [0, 1].

**Standardization**

Standardization centers the data around zero with a standard deviation of one. This is especially important for algorithms like SVM or linear regression that assume normally distributed data.

**Example:** Standardizing the fantasy characters dataset:

from sklearn.preprocessing import StandardScaler

# Standardization

scaler = StandardScaler()

df[['Magic Power', 'Agility', 'Wisdom']] = scaler.fit\_transform(df[['Magic Power', 'Agility', 'Wisdom']])

print("After Standardization:\n", df)

In this example, the features are standardized so that they have a mean of 0 and a standard deviation of 1.

## 4. Encoding Categorical Variables: One-Hot Encoding, Label Encoding

**Understanding Categorical Variables**

Categorical variables in a fantasy dataset might include character types (Wizard, Elf, Dwarf) or weapon types (Sword, Bow, Staff). These need to be converted into a numerical format for machine learning models to process them.

**One-Hot Encoding**

One-Hot Encoding is useful for converting categorical variables into binary columns. For instance, in our fantasy dataset, we might want to create binary columns for each character type.

**Example:** Encoding character types:

# Add a categorical column for 'Character Type'

df['Character Type'] = ['Wizard', 'Elf', 'Dwarf', 'Wizard', 'Hobbit', 'Elf', 'Human']

# One-Hot Encoding

df = pd.get\_dummies(df, columns=['Character Type'])

print("After One-Hot Encoding:\n", df)

Here, each character type is converted into a binary column, ensuring that the model treats each type independently.

**Label Encoding**

Label Encoding assigns an integer to each category. This is suitable for ordinal data where the categories have a meaningful order.

**Example:** Encoding the ranks of fantasy characters:

from sklearn.preprocessing import LabelEncoder

# Add a categorical column for 'Rank'

df['Rank'] = ['A', 'B', 'C', 'A', 'B', 'C', 'A']

# Label Encoding

le = LabelEncoder()

df['Rank'] = le.fit\_transform(df['Rank'])

print("After Label Encoding:\n", df)

In this example, the ranks are converted into integers.

## 

## 5. Basic Feature Engineering: Creating New Features, Handling Outliers

**Creating New Features**

Feature engineering involves creating new features that capture additional information. For instance, combining Magic Power and Wisdom might give us a new feature called "Magical Intelligence," which could be a better predictor of a character's effectiveness.

Example: Creating new features on our modified dataset (df\_combined):

Note: We are using the dataset after KNN imputation (df\_combined)

# Create a new feature 'Magical Intelligence' by combining 'Magic Power' and 'Wisdom'

df\_combined['Magical Intelligence'] = df\_combined['Magic Power'] \* df\_combined['Wisdom']

print("New Feature - Magical Intelligence:\n", df\_combined)

**Handling Outliers**

Outliers in a fantasy dataset could be characters with extreme abilities that differ significantly from the rest. Detecting and handling outliers is important to prevent them from skewing the model.

**Example:** Handling outliers in the dataset continued from earlier example (df\_combined):

# Detecting Outliers using IQR

Q1 = df\_combined['Magic Power'].quantile(0.25)

Q3 = df\_combined['Magic Power'].quantile(0.75)

IQR = Q3 - Q1

outliers = df\_combined[(df\_combined['Magic Power'] < (Q1 - 1.5 \* IQR)) | (df\_combined['Magic Power'] > (Q3 + 1.5 \* IQR))]

print("Outliers Detected:\n", outliers)

# Handling Outliers by Capping

df\_combined['Capped Magic Power'] = np.where(df\_combined['Magic Power'] > df\_combined['Magic Power'].quantile(0.95),

df\_combined['Magic Power'].quantile(0.95), df\_combined['Magic Power'])

print("\nAfter Capping Outliers:\n", df\_combined)

## 6. Detecting and Treating Outliers in Data

**Advanced Outlier Detection Methods**

Advanced outlier detection methods, such as Isolation Forest, are useful in datasets with complex interactions between features, like in fantasy characters where a combination of abilities might create outliers.

**Example:** Using Isolation Forest to detect outliers in the dataset continued from earlier example (df\_combined):

from sklearn.ensemble import IsolationForest

# Detecting Outliers using Isolation Forest

iso\_forest = IsolationForest(contamination=0.1)

df\_combined['Outlier'] = iso\_forest.fit\_predict(df\_combined[['Magic Power', 'Agility', 'Wisdom']])

print("Outliers Detected by Isolation Forest:\n", df\_combined)

## 7. Data Transformation Techniques for Improving Model Performance

**Logarithmic Transformation**

Logarithmic transformation can reduce skewness in data, such as when dealing with highly skewed features like Magic Power in a dataset of fantasy characters.

**Example:** Applying logarithmic transformation in the dataset continued from earlier example (df\_combined):

# Logarithmic Transformation of 'Magic Power'

df\_combined['Log Magic Power'] = np.log(df\_combined['Magic Power'] + 1)

print("After Logarithmic Transformation:\n", df\_combined)

**Box-Cox Transformation**

The Box-Cox transformation can stabilize variance and make the data more normal, which is useful for improving the performance of models like linear regression.

**Example:** Applying Box-Cox transformation in the dataset continued from earlier example (df\_combined):

from scipy.stats import boxcox

# Box-Cox Transformation of 'Magic Power'

df\_combined['Magic Power BoxCox'], lam = boxcox(df\_combined['Magic Power'] + 1)

print("After Box-Cox Transformation:\n", df\_combined)

print("Lambda Value:", lam)

**Feature Interactions**

Creating interaction terms between features can capture complex relationships. For instance, multiplying Magic Power by Agility could create a feature representing "Combat Effectiveness."

**Example:** Creating interaction terms in the dataset continued from earlier example (df\_combined):

# Create an interaction term between 'Magic Power' and 'Agility'

df\_combined['Combat Effectiveness'] = df\_combined['Magic Power'] \* df\_combined['Agility']

print("Interaction Term - Combat Effectiveness:\n", df\_combined)

**Binning and Discretization**

Binning continuous variables into categories can make the data easier to interpret. For example, binning Magic Power into "Low", "Medium", and "High" categories.

**Example:** Binning power levels in the dataset continued from earlier example (df\_combined):

# Binning 'Magic Power' into categories

bins = [0, 30, 70, np.inf]

labels = ['Low', 'Medium', 'High']

df\_combined['Magic Power Category'] = pd.cut(df\_combined['Magic Power'], bins=bins, labels=labels)

print("Binned Magic Power:\n", df\_combined)