Introduction to Data Science in Python

Data science involves the analysis and interpretation of large datasets to uncover patterns, make predictions, and gain insights. Python is one of the most popular programming languages used in data science due to its rich ecosystem of libraries, simplicity, and flexibility.

In this tutorial, we will begin by importing the essential libraries that are commonly used for data analysis in Python: **Pandas** and **NumPy**.

Why These Libraries?

- **Pandas**: This library is highly efficient for data manipulation and analysis. It provides data structures like DataFrames, which are essential for handling and analyzing structured data.
- **NumPy**: It stands for "Numerical Python" and provides support for large multi-dimensional arrays and matrices, along with a wide range of mathematical functions to operate on these arrays.

Steps to Install the Libraries

Before we start, ensure that these libraries are installed in your environment. If not, you can install them using `pip`:

```
pip install pandas numpy
```

Alternatively, if you have a conda environment, you can install them using `conda`:

```
conda install pandas numpy
```

In [1]:
```
# Import necessary libraries
import pandas as pd
import numpy as np

# pd is a conventional alias for pandas
# np is a conventional alias for numpy
```

Introduction to DataFrames

A **DataFrame** is one of the most important data structures in the Pandas library. It is a two-dimensional, size-mutable, and heterogeneous tabular data structure with labeled axes (rows and columns). You can think of it as a spreadsheet or SQL table in Python, but much more powerful for handling and manipulating data.

DataFrames are primarily used to:

- Store and manipulate structured data
- Perform operations such as filtering, grouping, merging, and aggregating data
- Handle missing or incomplete data

In this section, we will learn how to create a toy DataFrame from scratch and perform basic manipulations to understand the fundamental concepts of working with DataFrames in Pandas.

Creating a Toy DataFrame

Let's create a simple DataFrame to get started. We will use the **pd.DataFrame()** constructor and provide some sample data in the form of a dictionary, where the keys are column names and the values are lists representing data for each column.

In [2]:
```
# Creating a toy DataFrame using a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva'],
    'Age': [25, 30, 35, 40, 28],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix'],
    'Salary': [70000, 80000, 120000, 90000, 85000]
}

df = pd.DataFrame(data)

# Display the DataFrame
df
```

Out[2]:

	Name	Age	City	Salary
0	Alice	25	New York	70000
1	Bob	30	Los Angeles	80000
2	Charlie	35	Chicago	120000
3	David	40	Houston	90000
4	Eva	28	Phoenix	85000

In [3]:
```
df['Name']
```

Out[3]:
```
0      Alice
1        Bob
2    Charlie
3      David
4        Eva
Name: Name, dtype: object
```

Understanding the Structure of a DataFrame

Once we have created the DataFrame, it is crucial to understand its structure. A DataFrame consists of:

- Rows: Represent individual records (in our case, people)
- Columns: Represent attributes or features (like Name, Age, City, and Salary)

In this DataFrame, each row contains data about a person, and each column holds a specific type of information.

In [4]:
```
df.tail()    # Checking first few rows of a dataframe.
```

Out[4]:

	Name	Age	City	Salary
0	Alice	25	New York	70000
1	Bob	30	Los Angeles	80000
2	Charlie	35	Chicago	120000
3	David	40	Houston	90000
4	Eva	28	Phoenix	85000

In [5]:
```
df.set_index('Name').loc["David"]
```

Out[5]:
```
Age            40
City      Houston
Salary      90000
Name: David, dtype: object
```

Accessing Data in a DataFrame

One of the key features of Pandas is its ability to allow easy access to data in a DataFrame. You can access columns or rows using various methods. Let's see some examples:

In [6]:
```
# Accessing the 'Name' column
df['Name']
```

Out[6]:
```
0      Alice
1        Bob
2    Charlie
3      David
4        Eva
Name: Name, dtype: object
```

In [7]:
```
# Accessing multiple columns
df[['Name', 'City']]
```

Out[7]:

	Name	City
0	Alice	New York
1	Bob	Los Angeles
2	Charlie	Chicago
3	David	Houston
4	Eva	Phoenix

In [8]:
```
# Accessing the first row using iloc (index-based)
df.iloc[0]
```

Out[8]:
```
Name         Alice
Age             25
City      New York
Salary       70000
Name: 0, dtype: object
```

In [9]:
```
# Accessing a row by its index using loc (label-based)
df.loc[2]
```

Out[9]:
```
Name      Charlie
Age            35
City      Chicago
Salary     120000
Name: 2, dtype: object
```

Basic Manipulations of DataFrames

Once the DataFrame is created, we can perform several manipulations to explore or clean the data. Here are a few common operations:

1. **Adding a New Column:**

We can easily add new columns to a DataFrame by assigning values to a new column label. Let's calculate the annual bonus as 10% of the salary.

In [10]:
```
# Adding a new column 'Bonus' that is 10% of the salary
df['Bonus'] = df['Salary'] * 0.1
df
```

Out[10]:

	Name	Age	City	Salary	Bonus
0	Alice	25	New York	70000	7000.0
1	Bob	30	Los Angeles	80000	8000.0
2	Charlie	35	Chicago	120000	12000.0
3	David	40	Houston	90000	9000.0
4	Eva	28	Phoenix	85000	8500.0

2. **Filtering Rows Based on Conditions:**

You can filter data based on specific conditions. For example, let's filter out all employees who earn more than $80,000.

In [11]:
```
# Filtering rows where Salary > 80000
high_salary = df[df['Salary'] > 80000]
high_salary
```

Out[11]:

	Name	Age	City	Salary	Bonus
2	Charlie	35	Chicago	120000	12000.0
3	David	40	Houston	90000	9000.0
4	Eva	28	Phoenix	85000	8500.0

3. ***Updating Values:***

You can also modify values in a DataFrame. Let's say David has moved to San Francisco, and we want to update his city in the DataFrame.

In [12]:
```
# Updating the city of 'David' to 'San Francisco'
df.loc[df['Name'] == 'David', 'City'] = 'San Francisco'
df
```

Out[12]:

	Name	Age	City	Salary	Bonus
0	Alice	25	New York	70000	7000.0
1	Bob	30	Los Angeles	80000	8000.0
2	Charlie	35	Chicago	120000	12000.0
3	David	40	San Francisco	90000	9000.0
4	Eva	28	Phoenix	85000	8500.0

4. ***Sorting the DataFrame:***

Sorting is often required to organize data. We can sort the DataFrame based on values in a specific column. For example, let's sort the DataFrame by salary in descending order.

In [13]:
```
# Sorting the DataFrame by 'Salary' in descending order
df_sorted = df.sort_values(by='Salary', ascending=False)
df_sorted
```

Out[13]:

	Name	Age	City	Salary	Bonus
2	Charlie	35	Chicago	120000	12000.0
3	David	40	San Francisco	90000	9000.0
4	Eva	28	Phoenix	85000	8500.0
1	Bob	30	Los Angeles	80000	8000.0
0	Alice	25	New York	70000	7000.0

5. ***Dealing with Missing Data:***

In real-world scenarios, data is often incomplete. Pandas provides tools to handle missing data. For demonstration purposes, let's add some missing values to the DataFrame.

In [14]:
```
# Introducing some missing data
df.loc[2, 'Salary'] = np.nan  # Set Charlie's salary to NaN (missing)
df
```

Out[14]:

	Name	Age	City	Salary	Bonus
0	Alice	25	New York	70000.0	7000.0
1	Bob	30	Los Angeles	80000.0	8000.0
2	Charlie	35	Chicago	NaN	12000.0
3	David	40	San Francisco	90000.0	9000.0
4	Eva	28	Phoenix	85000.0	8500.0

You can handle missing data by filling them with specific values or by removing the rows/columns with missing data.

In [15]:
```
# Filling missing values with a default value (e.g., 0)
df_filled = df.ffill()
df_filled
```

Out[15]:

	Name	Age	City	Salary	Bonus
0	Alice	25	New York	70000.0	7000.0
1	Bob	30	Los Angeles	80000.0	8000.0
2	Charlie	35	Chicago	80000.0	12000.0
3	David	40	San Francisco	90000.0	9000.0
4	Eva	28	Phoenix	85000.0	8500.0

In [16]:
```
# Alternatively, you can drop rows with missing data
df_dropped = df.dropna()
df_dropped
```

Out[16]:

	Name	Age	City	Salary	Bonus
0	Alice	25	New York	70000.0	7000.0
1	Bob	30	Los Angeles	80000.0	8000.0
3	David	40	San Francisco	90000.0	9000.0
4	Eva	28	Phoenix	85000.0	8500.0

Tutorial

Now, we will get started with the tutorial.

Exercise 1: Astronomical Statistical Unit

The **Astronomical Unit (AU)** is a fundamental concept in astronomy used to describe the distance between the Earth and the Sun. In this exercise, we are given data collected from various studies aiming to estimate the AU, provided in a CSV file named `AU.csv`.

a. Identifying the Statistical Units and Features

The statistical units in this dataset are the individual studies conducted to estimate the distance from the Earth to the Sun. The features for each statistical unit are:

- **Study name**: Name of the scientist or organization that conducted the study.
- **Year**: The year when the study was conducted.
- **Distance**: The distance from Earth to the Sun in millions of miles as estimated by the study.

b. Using the Data to Determine an AU in 1961

If you had lived in 1961, the available data up to that year would help you approximate the value of the AU by analyzing trends in the estimates. By reviewing and averaging the measurements provided by studies conducted before 1961, you could derive an estimate close to the actual distance.

c. Loading the Data and Determining the AU

Let's load the data from the file `AU.csv` and compute the average distance from Earth to the Sun based on the data provided. Then, we will compare our result to the modern AU value of **92,955,807.2730 miles**.

In [17]:
```
# Loading the data from AU.csv
au_data = pd.read_csv('./AU.csv')

# Displaying the first few rows of the dataset
au_data.head()
```

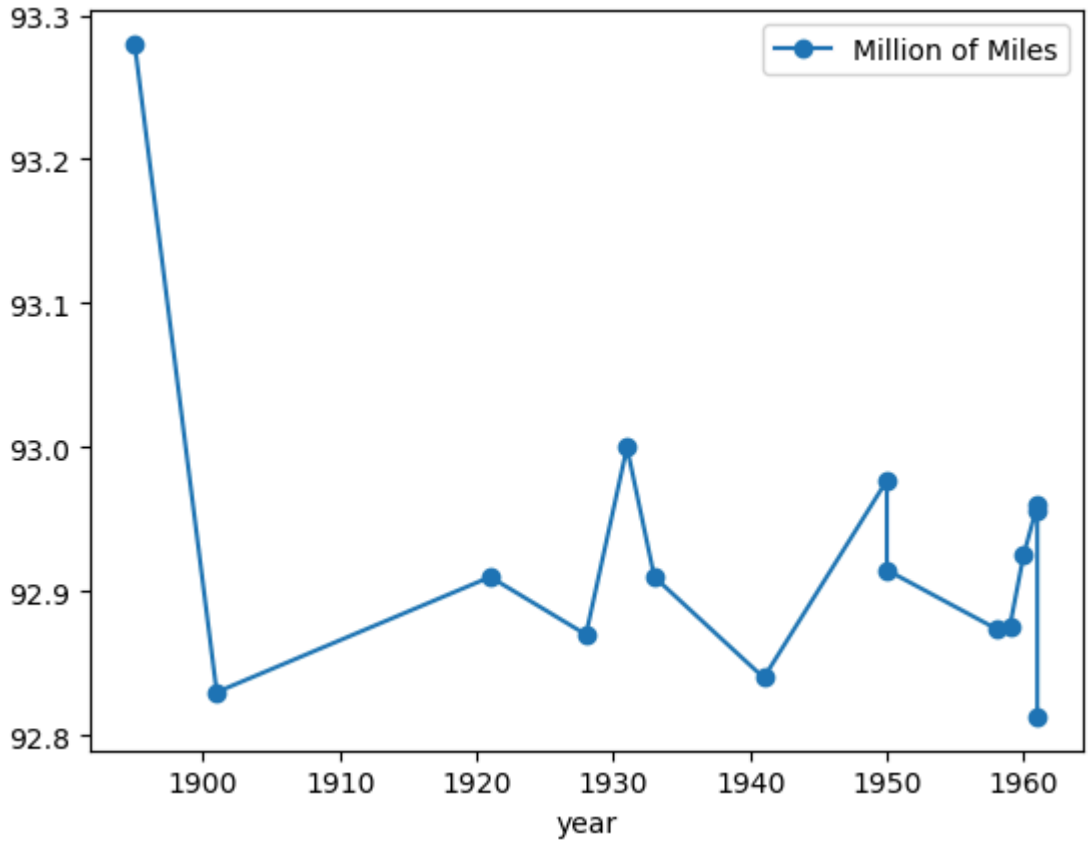
Out[17]:

	study	year	mmiles
0	newcombe	1895	93.28
1	hinks	1901	92.83
2	noteboom	1921	92.91
3	spencer1	1928	92.87
4	spencer2	1931	93.00

In [18]:
```
# Calculating the mean distance
mean_distance = au_data['mmiles'].mean()
mean_distance
```

Out[18]:
```
92.92905999999999
```

In [20]:
```
# Looking at the propagation of the studies in time.

import matplotlib.pyplot as plt
au_data.plot(x='year', y='mmiles', marker='o', label="Million of Miles")
plt.show()
```



In [21]:
```
# Modern value of AU in millions of miles
modern_au = 92_955_807.2730 / 1_000_000

# Difference between the modern AU and calculated mean
difference = modern_au - mean_distance
difference
```

Out[21]:
```
0.02674727300001223
```

Do the same exercise that we did with mean, but using the median. What differences do you observe? What might be the reason for that?

Exercise 2: Standard Soldier

In this exercise, we examine historical height standards for soldiers in the French army over time. The dataset `soldier.csv` contains information about the minimum required height of soldiers from different years.

a. Difference Between the Two Datasets

Both datasets deal with standards, but the key difference is the nature of what is being measured. The first dataset (AU) measured an astronomical unit, which is a physical constant, while the second dataset (soldier heights) deals with a human standard that changes over time, possibly due to societal, medical, or technological advancements.

b. What Can We Conclude from the Data?

By looking at the data, we can observe a decrease in the minimum height required for soldiers over time. This may be explained by changes in the population's average height, the needs of the army, or changes in military recruitment strategies.

Let's load the soldier data and visualize the trend.

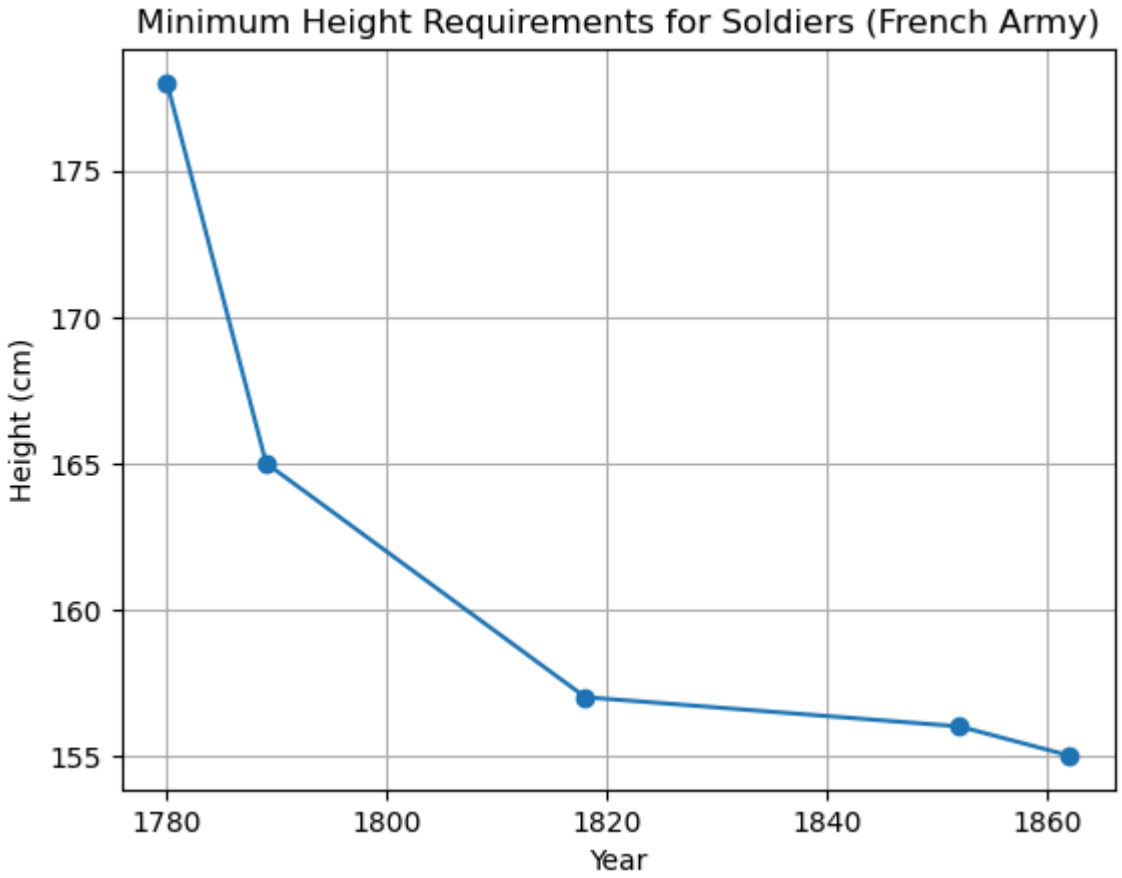
In [22]:
```
# Loading the soldier height data from soldier.csv
import matplotlib.pyplot as plt
soldier_data = pd.read_csv('./soldier.csv')

# Displaying the data
soldier_data
```

Out[22]:

	year	height
0	1780	178
1	1789	165
2	1818	157
3	1852	156
4	1862	155

In [23]:

```
plt.plot(soldier_data['year'], soldier_data['height'], marker='o')
plt.title('Minimum Height Requirements for Soldiers (French Army)')
plt.xlabel('Year')
plt.ylabel('Height (cm)')
plt.grid(True)
plt.show()
```


Minimum Height Requirements for Soldiers (French Army)
Height (cm)
175
170
165
160
155
1780
1800
1820
1840
1860
Year