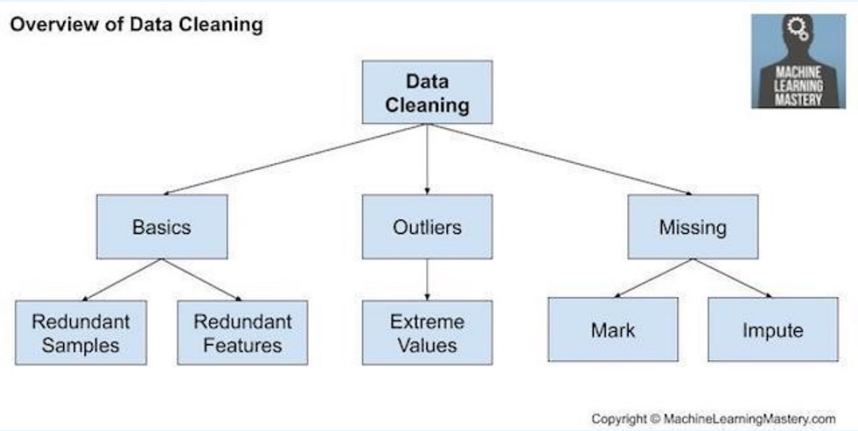
Some common tasks in data cleaning and preprocessing include:

1. Removal of missing values
2. Treatment of duplicate data
3. Data type conversion
4. Treatment of inconsistent data
5. Data normalization
6. Data grouping



Removal of missing values: Use the following commands to handle missing values

Missing data can affect model performance negatively. Methods to handle missing data include removal of missing data or filling missing values with mean, median, or mode values as a part of imputation.

Definition of Imputation: Imputation is the process of replacing missing data with substituted values in a dataset. It helps retain the majority of the data and information by estimating and filling in missing entries, thereby allowing for complete and robust analysis without discarding valuable information. Imputation can be performed at the level of an entire data point (unit imputation) or for individual feature values (item imputation)

Why is Imputation Important?

- Prevents data loss: Avoids discarding entire rows/columns due to missing values.
- Reduces bias: Minimizes bias introduced by listwise deletion.
- Enables robust analysis: Allows the use of standard analytical and machine learning methods on complete datasets.

Types of Imputation Methods

Here are the most common imputation techniques, along with Python examples for each:

1. Mean Imputation : Replace missing values with the mean of the non-missing values in the column.

```
import pandas as pd
import numpy as np

df = pd.DataFrame({'A': [1, 2, np.nan, 4, 5]})
df['A'].fillna(df['A'].mean(), inplace=True)
print(df)
```

2. Median Imputation : Use the median value for imputation (useful for skewed data)

```
df = pd.DataFrame({'A': [1, 2, np.nan, 4, 100]})
df['A'].fillna(df['A'].median(), inplace=True)
print(df)
```

3. Mode Imputation : Impute missing values with the most frequent value (mode), typically for categorical data

```
df = pd.DataFrame({'Gender': ['Male', 'Female', np.nan, 'Female']})
df['Gender'].fillna(df['Gender'].mode()[0], inplace=True)
print(df)
```

4. Forward Fill (Next/Previous Value) :Use the previous or next value in a time series.

```
df = pd.DataFrame({'Value': [1, np.nan, 3, np.nan, 5]})
df['Value'].fillna(method='ffill', inplace=True)  # Forward fill
print(df)
```

5. Backward Fill : Fill missing values using the next value.

```
df = pd.DataFrame({'Value': [1, np.nan, 3, np.nan, 5]})
df['Value'].fillna(method='bfill', inplace=True)  # Backward fill
print(df)
```

6. Fixed Value Imputation: Replace missing values with a fixed value (e.g., zero or a string like “Unknown”)

```
df = pd.DataFrame({'A': [1, np.nan, 3]})
df['A'].fillna(0, inplace=True)
print(df)
```

Imputation Techniques Table View :

Method	Suitable for	Python Command/Function
Mean Imputation	Numeric	`fillna(df.mean())`
Median Imputation	Numeric (skewed)	`fillna(df.median())`
Mode Imputation	Categorical	`fillna(df.mode())`
Forward/Backward Fill	Time Series	`fillna(method='ffill'/'bfill')`
KNN Imputation	Numeric/Categorical	`KNNImputer()`
Regression Imputation	Numeric	`LinearRegression()`
MICE	Numeric/Categorical	`IterativeImputer()`
Fixed Value	Any	`fillna(value)`

2. **Handling Duplicated Values in Data:** Duplicate values are common in datasets and can distort analysis or lead to incorrect conclusions. Here are the main techniques for handling duplicates, with explanations and Python code examples.

1**. Identifying Duplicate Values:** Using `duplicated()`method, The `duplicated()` method returns a Boolean Series indicating whether each row is a duplicate of a previous row.

```
import pandas as pd

data = pd.DataFrame({
    'name': ['John', 'Emily', 'John', 'Jane', 'John'],
    'age': [25, 28, 25, 30, 25],
    'salary': [50000, 60000, 50000, 70000, 50000]
})

# Identify duplicate rows
duplicates = data.duplicated()
print(data[duplicates])
```

```
   name  age  salary
2  John   25   50000
4  John   25   50000
```

2. Removing Duplicate Rows : `drop_duplicates()`, Removes duplicate rows from a DataFrame. By default, it keeps the first occurrence and removes subsequent duplicates. We can specify columns to consider, and whether to keep the first or last occurrence

```
# Remove all duplicate rows
data_cleaned = data.drop_duplicates()
print(data_cleaned)

# Remove duplicates based on specific columns, keeping the last occurrence
data_cleaned = data.drop_duplicates(subset=['name', 'age'], keep='last')
print(data_cleaned)

```

```
   name  age  salary
0  John   25   50000
1 Emily   28   60000
3  Jane   30   70000
   name  age  salary
1 Emily   28   60000
3  Jane   30   70000
4  John   25   50000
```

3. **Keeping Specific Occurrences :** Using `keep` Parameter in `drop_duplicates()`, by using above parameter we can specify whether to keep the first or last occurrence of a duplicate.

```
# Keep only the last occurrence of each duplicate
data_cleaned = data.drop_duplicates(keep='last')
print(data_cleaned)
```

4. **Replacing Duplicate Values :** Instead of simply removing duplicates, we may want to aggregate them (e.g., take the mean or median of a column for duplicate entries) and replace them

```
# Replace duplicate salaries with the median salary for each name
data['salary'] = data.groupby('name')['salary'].transform('median')
print(data)

```

```
    name  age   salary
0   John   25  50000.0
1  Emily   28  60000.0
2   John   25  50000.0
3   Jane   30  70000.0
4   John   25  50000.0
```

5. **Manual Correction :** Sometimes duplicates occur due to data entry errors. After identifying duplicates, we may manually correct the erroneous values.

```
# Suppose index 1 is a duplicate with a wrong name, correct it manually
data.loc[1, 'name'] = 'Dio'
print(data)
```

Techniques for Handling Duplicates in a Table view :

Technique	Use Case	Python Example/Function
Identify duplicates	DataFrames	`duplicated()`
Remove duplicates	DataFrames	`drop_duplicates()`
Keep first/last	DataFrames	`drop_duplicates(keep='last')`
Aggregate duplicates	DataFrames	`groupby().transform()`
Manual correction	DataFrames	Direct assignment
Remove duplicates (list)	Python lists	`set()`

3.**Data type conversion :** Data type conversion, also known as type casting, is a crucial step in EDA. It ensures that each variable in your dataset has the appropriate data type for analysis, visualization, and modelling.

Why Data Type Conversion Is Important

- Ensures correct calculations: Numeric operations require numeric types.
- Enables proper visualization: Categorical variables must be labeled as such.
- Prevents errors: Many functions expect specific data types.
- Optimizes memory usage: Using the right data type can reduce memory consumption.

Common Data Types in Python (Pandas)

- int: Integer numbers
- float: Floating-point numbers
- str/object: Text or mixed types
- bool: Boolean values (True/False)
- category: Efficient storage for categorical data
- datetime: Date and time values

Methods for Data Type Conversion : Using `astype()` Method, we can convert a single column or multiple columns at once.

```python
import pandas as pd

df = pd.DataFrame({
    'A': ['1', '2', '3'],
    'B': ['4.5', '5.2', '6.8'],
    'C': ['True', 'False', 'True']})

# Convert column 'A' to integer
df['A'] = df['A'].astype(int)

# Convert column 'B' to float
df['B'] = df['B'].astype(float)

# Convert column 'C' to boolean
df['C'] = df['C'].astype(bool)

print(df.dtypes)

```

```
A      int64
B    float64
C       bool
dtype: object
```

```python
df = df.astype({'A': int, 'B': float, 'C': bool})
print(df)
```

```
   A    B     C
0  1  4.5  True
1  2  5.2  True
2  3  6.8  True
```

2. Using `pd.to_numeric()`, `pd.to_datetime()`, and `pd.to_timedelta()`: These functions help convert columns to numeric, datetime, or timedelta types, handling errors gracefully.

```
# Convert to numeric (handles errors)
df['A'] = pd.to_numeric(df['A'], errors='coerce')
print(df["A"])
```

```
0    1
1    2
2    3
Name: A, dtype: int64
```

3. Using `convert_dtypes()` : Automatically converts columns in a DataFrame to the best possible data types.

```
df = df.convert_dtypes()
print(df.dtypes)

```

```
A      Int64
B    Float64
C    boolean
dtype: object
```

4. Using `infer_objects()`, Converts columns of type `object` to a more specific type (e.g., int, float) if possible

5. Type Conversion for Python Built-in Types : For variables (not DataFrames), use built-in functions: `int()`, `float()`, `str()`, `bool()`, etc.

```
x = "123"
y = int(x)        # Converts string to integer
type(y)

```

```
int
```

```
z = 45.67
w = str(z)        # Converts float to string
type(w)

```

```
str
```

6. Converting to Categorical Data : Efficient for columns with repeated values (e.g., gender, product category).

```
df['gender'] = df['gender'].astype('category')
```

Here is the summary of conversions we can do in a table view

Method	Use Case	Example Command
`astype()`	General type conversion	`df['col'].astype(int)`
`pd.to_numeric()`	Convert to numeric	`pd.to_numeric(df['col'])`
`pd.to_datetime()`	Convert to datetime	`pd.to_datetime(df['col'])`
`convert_dtypes()`	Auto-convert all columns	`df.convert_dtypes()`
`infer_objects()`	Infer better types for objects	`df.infer_objects()`
Built-in functions	Python variables	`int(x)`
Categorical	Efficient storage for categories	`df['col'].astype('category')`

5: **Treating inconsistent Data :** Inconsistent data refers to values in a dataset that do not follow a standard format, contain typos, use different units, or represent the same thing in multiple ways. Examples include different date formats, inconsistent capitalization, extra whitespace, or varying representations of the same category (e.g., “NY”, “New York”, “new york”).

Inconsistent data can lead to incorrect analysis, unreliable models, and misleading insights. Cleaning and standardizing such data is a crucial step in data preprocessing.

Common Types of Inconsistencies

- Textual inconsistencies: Typos, varying cases, extra spaces, abbreviations.
- Date/time formats: Different date representations.
- Categorical variations: Multiple labels for the same category.
- Unit inconsistencies: Different measurement units.
- Invalid values: Out-of-range or logically impossible entries.

Techniques and Python Code Examples :

1. Standardizing Text Data

a. Lowercasing and Stripping Whitespace

```
1  import pandas as pd
2
3  df = pd.DataFrame({'city': [' New York ', 'new york', 'NY', 'Los Angeles', 'los
   angeles ', 'LA']})
4  df['city'] = df['city'].str.lower().str.strip()
5  print(df)
6
```

```
          city
0     new york
1     new york
2           ny
3  los angeles
4  los angeles
5           la
```

b. Replacing Abbreviations and Typos :

```
mapping = {'ny': 'new york', 'la': 'los angeles'}
df['city'] = df['city'].replace(mapping)
print(df)

```

```
         city
0    new york
1    new york
2    new york
3  los angeles
4  los angeles
5  los angeles
```

c. Removing Punctuation :

```
df['city'] = df['city'].str.replace(r'[^\w\s]', '', regex=True)
print(df['city'])
```

```
0       new york
1       new york
2       new york
3    los angeles
4    los angeles
5    los angeles
Name: city, dtype: object
```

2. **Standardizing Date Formats :**

```
df = pd.DataFrame({'date': ['2025-07-08', '08/07/2025', 'July 8, 2025']})
df['date'] = pd.to_datetime(df['date'], errors='coerce')
print(df)

```

```
        date
0 2025-07-08
1        NaT
2        NaT
```

3. Handling Categorical Inconsistencies

a. Checking Unique Values

```
print(df['category'].unique())
```

b. Value Mapping

```
mapping = {'M': 'Male', 'F': 'Female', 'male': 'Male', 'female': 'Female'}
df['gender'] = df['gender'].replace(mapping)
```

Technique	Use Case	Example Command/Function
Lowercase/strip	Text standardization	`str.lower().str.strip()`
Replace mapping	Abbreviations/typos	`replace(mapping)`
Remove punctuation	Text cleaning	`str.replace(r'[^\w\s]', '', regex=True)`
Standardize dates	Date format consistency	`pd.to_datetime()`
Harmonize units	Measurement unit consistency	`str.extract()`
Fuzzy matching	Correcting typos	`fuzzywuzzy.process.extract()`
Value mapping	Categorical standardization	`replace(mapping)`
Validation rules	Logical/data domain checks	Filtering with conditions
Data type conversion	Ensuring correct data types	`astype()`

5. Data Normalization : Data normalization is a key preprocessing step in data analysis and machine learning. It ensures that numerical features are on a similar scale, which improves the performance and convergence speed of algorithms.

1. **Understanding Data Normalization**

Normalization transforms data to a common scale without distorting differences in the ranges of values. Common techniques include:

- Min-Max Normalization: Scales data to a fixed range, usually .
- Z-score Normalization (Standardization): Canters data with mean 0 and standard deviation 1.
- Decimal Scaling: Scales values by powers of 10.

2. **General Steps for Data Normalization**

1. Import Required Libraries
2. Load or Create the Dataset
3. Select Numerical Features to Normalize
4. Choose a Normalization Technique
5. Apply the Normalization
6. Inspect the Result

Examples :

```
from sklearn.datasets import load_iris
import pandas as pd

# Load the Iris dataset
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
print(df.head())
```

A. Min-Max Normalization : All features are will be converted between 0 and 1

```
[1]
from sklearn.datasets import load_iris
import pandas as pd

# Load the Iris dataset
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
print(df.head())

```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2
```

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
df_min_max_scaled = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
print(df_min_max_scaled.head())

```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0           0.222222          0.625000           0.067797          0.041667
1           0.166667          0.416667           0.067797          0.041667
2           0.111111          0.500000           0.050847          0.041667
3           0.083333          0.458333           0.084746          0.041667
4           0.194444          0.666667           0.067797          0.041667
```

B. Z-score Normalization (Standardization): Canters features with mean 0 and standard deviation 1

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df_standard_scaled = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)
print(df_standard_scaled.head())

```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0          -0.900681          1.019004          -1.340227         -1.315444
1          -1.143017         -0.131979          -1.340227         -1.315444
2          -1.385353          0.328414          -1.397064         -1.315444
3          -1.506521          0.098217          -1.283389         -1.315444
4          -1.021849          1.249201          -1.340227         -1.315444
```

C. Decimal Scaling Normalization : Divides values by a power of 10 based on the maximum absolute value.

```
import numpy as np

max_abs_val = np.max(np.abs(df.values), axis=0)
df_decimal_scaled = df / 10 ** np.ceil(np.log10(max_abs_val))
print(df_decimal_scaled.head())

```

```
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0               0.51              0.35               0.14              0.02
1               0.49              0.30               0.14              0.02
2               0.47              0.32               0.13              0.02
3               0.46              0.31               0.15              0.02
4               0.50              0.36               0.14              0.02
```

Summary Table: Normalization Methods

Method	Python Command Example	Effect on Data
Min-Max Scaling	`MinMaxScaler().fit_transform(X)`	Scales to [1]
Z-score Standardization	`StandardScaler().fit_transform(X)`	Mean 0, Std 1
Decimal Scaling	`X / 10 ** np.ceil(np.log10(np.max(np.abs(X))))`	Scales by power of 10
Vector Norm	`X / np.linalg.norm(X, axis=0)`	Each column norm = 1

Data Grouping Techniques in Data Preprocessing

Definition:

Data grouping is the process of dividing a dataset into subgroups or categories based on one or more variables. This helps in summarizing, analysing, and visualizing large datasets more effectively, and is a vital step in data preprocessing and analysis.

Why Use Data Grouping?

- Identify patterns and trends in data.
- Summarize large datasets into manageable chunks.
- Perform statistical analysis on specific subgroups.
- Visualize data in a more interpretable way.

Common Data Grouping Techniques

1. **Grouping by Categorical Variables**
   - Groups data based on discrete categories (e.g., gender, region, product type).
   - Useful for summarizing data by labels or classes.
2. **Grouping by Numerical Variables**
   - Divides continuous data into bins or ranges (e.g., age groups, income brackets).
   - Techniques include:
   - Binning: Dividing data into equal-sized intervals.
   - Quantiles: Dividing data based on percentiles (quartiles, deciles).
   - Clustering: Grouping based on similarity (e.g., k-means clustering).
3. **Hierarchical Grouping**
   - Groups data at multiple levels (e.g., country → city → neighbourhood).
   - Useful for nested data analysis.
4. **Composite Grouping**
   - Groups data using multiple variables at once (e.g., age group and gender)

Examples :

```python
import pandas as pd

# Sample data
data = {
    'Team': ['A', 'A', 'B', 'B', 'C', 'C'],
    'Player': ['John', 'Mike', 'Sara', 'Anna', 'Tom', 'Nick'],
    'Points': [10, 15, 20, 25, 30, 35],
    'Position': ['Forward', 'Guard', 'Forward', 'Guard', 'Forward', 'Guard']
}
df = pd.DataFrame(data)
print(df)

```

```
  Team Player  Points Position
0    A   John      10  Forward
1    A   Mike      15    Guard
2    B   Sara      20  Forward
3    B   Anna      25    Guard
4    C    Tom      30  Forward
5    C   Nick      35    Guard
```

1.Group by a Single Column :

```python
# Group by Team and calculate total points per team
grouped = df.groupby('Team')['Points'].sum()
print(grouped)

```

```
Team
A    25
B    45
C    65
Name: Points, dtype: int64
```

2. Group by Multiple Columns

```python
# Group by Team and Position, calculate average points
grouped_multi = df.groupby(['Team', 'Position'])['Points'].mean()
print(grouped_multi)

```

```
Team  Position
A     Forward     10.0
      Guard       15.0
B     Forward     20.0
      Guard       25.0
C     Forward     30.0
      Guard       35.0
Name: Points, dtype: float64
```

3. Applying Aggregation Functions

```python
# Group by Position and get multiple summary statistics
agg_stats = df.groupby('Position')['Points'].agg(['mean', 'sum', 'count'])
print(agg_stats)

```

```
          mean  sum  count
Position
Forward   20.0   60      3
Guard     25.0   75      3
```

4. **Grouping Numerical Data (Binning)**

```
# Group ages into bins
df['Age'] = [23, 25, 31, 35, 40, 45]
bins = [20, 30, 40, 50]
labels = ['20-30', '31-40', '41-50']
df['AgeGroup'] = pd.cut(df['Age'], bins=bins, labels=labels)
print(df.groupby('AgeGroup')['Points'].mean())

```

```
AgeGroup
20-30    12.5
31-40    25.0
41-50    35.0
Name: Points, dtype: float64
```

Summary Table: Data Grouping Techniques

Technique	Python Command Example	Use Case/Effect
Categorical Grouping	`df.groupby('category')`	Summarize by labels/classes
Multi-level Grouping	`df.groupby(['A', 'B'])`	Nested/grouped analysis
Binning (Numerical)	`pd.cut(df['num'], bins)`	Convert continuous to categorical
Aggregation	`groupby().agg(['sum', 'mean'])`	Summary statistics per group
Clustering	`KMeans().fit_predict(X)`	Group by similarity, unsupervised