2. Data Manipulation

These data manipulation techniques are crucial for data preparation and analysis in various domains, from business analytics to data science and machine learning. They help ensure data quality, consistency, and usability for decision-making and reporting.

2.1 Import, store, and export data:

Fundamental understanding of ETL (Extract, Transform, Load): ETL is a process used in data management to extract data from various sources, transform it into a consistent format, and load it into a destination such as a data warehouse or a database. It involves the following steps:

- **Extract:** This step involves extracting data from various sources like databases, flat files, APIs, or web scraping.
  example:
  df.to_csv(“D:/MyEduSolve/tugas_cleansing.csv”)

  extract the file into .csv format

  ```
  df.to_csv("D:/MyEduSolve/tugas_cleansing.csv")
  ```

- **Transform:** In the transformation step, data is cleaned, standardized, and converted into a format that can be used for analysis. This might include data cleansing, data enrichment, and the creation of new derived variables.
  example :

  ```
  df['Referal'] = df['Referal'].astype(str)
  df['Referal'] = df['Referal'].replace({'1.0':'use Referal code', '0.0':'not use Referal code'})
  ```

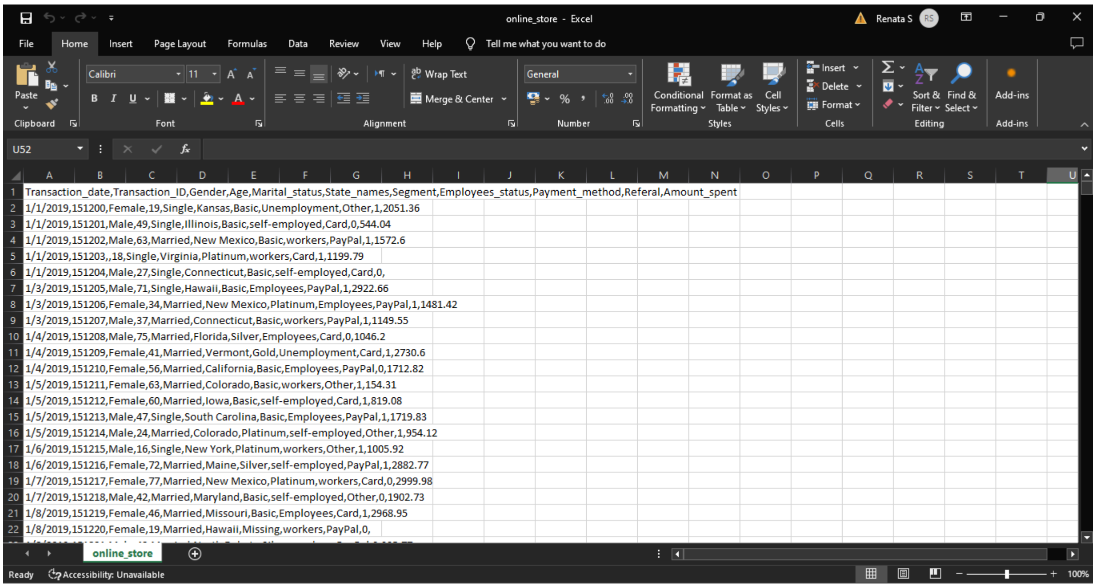
- **Load:** The final step is to load the transformed data into a target storage system, like a database or a data warehouse, making it available for analysis.

  Load a CSV file containing datasets from online stores obtained from Kaggle

  ```
  In [3]: df = pd.read_csv('online_store.csv')
  ```

- **Common data storage file formats:** These include delimited data files (e.g., CSV), XML (Extensible Markup Language), and JSON (JavaScript Object Notation) for storing structured data.
  Here we use a CSV format file (Comma Separated Values), which means that each value in the data row in the file is separated by commas. Files in this format are generally used to store datasets.

  ```
  In [3]: df = pd.read_csv('online_store.csv')
  ```



2.2 Clean data:

Purpose and common practices:

- **Handling NULL values:** Dealing with missing or NULL values, which may involve imputing missing data or excluding rows with missing values.

  example:

  Check data condition

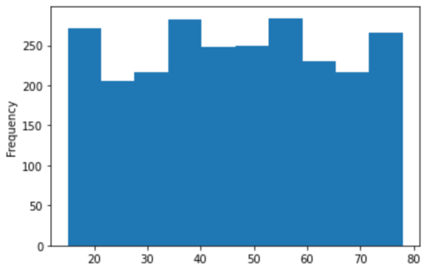
  df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2512 entries, 0 to 2511
Data columns (total 11 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Transaction_date  2512 non-null   object 
 1   Transaction_ID    2512 non-null   int64  
 2   Gender            2484 non-null   object 
 3   Age               2470 non-null   float64
 4   Marital_status    2512 non-null   object 
 5   State_names       2512 non-null   object 
 6   Segment           2512 non-null   object 
 7   Employees_status  2486 non-null   object 
 8   Payment_method    2512 non-null   object 
 9   Referal           2357 non-null   float64
 10  Amount_spent      2270 non-null   float64
dtypes: float64(3), int64(1), object(7)
memory usage: 216.0+ KB
```

  Display a visualization of the columns Age.

  df.Age.plot(kind='hist')

```
<AxesSubplot:ylabel='Frequency'>
```



  Because the Age column has a skewness distribution

  Then we will do imputation on the Age column using the median

  val = df.Age.median()

  df['Age'] = df.Age.fillna(val)

  Display dataset info to see whether the Age column has been imputed df.info()

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2512 entries, 0 to 2511
Data columns (total 11 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Transaction_date  2512 non-null   object 
 1   Transaction_ID    2512 non-null   int64  
 2   Gender            2484 non-null   object 
 3   Age               2512 non-null   float64
 4   Marital_status    2512 non-null   object 
 5   State_names       2512 non-null   object 
 6   Segment           2512 non-null   object 
 7   Employees_status  2486 non-null   object 
 8   Payment_method    2512 non-null   object 
 9   Referal           2357 non-null   float64
 10  Amount_spent      2270 non-null   float64
dtypes: float64(3), int64(1), object(7)
memory usage: 216.0+ KB
```

From the dataset info above, it can be seen that the Age column has changed

- **Special characters:** Removing or encoding special characters that can cause data processing issues.
  example :
  You have the file name
  df = pd.read_csv('online_store!!.csv')
  And the double exclamation mark special character (!!) can cause problems when trying to read or process such files. You can remove these special character to make the file name cleaner and more easily accessible.
  After removing special characters, the filename will become
  df = pd.read_csv('online_store.csv')
- **Trimming spaces:** Trimming leading and trailing white spaces from text data to ensure consistency.
  example:
  Suppose you have text " jupyter notebook " that has extra spaces at the front and at the back. By trimming the extra spaces, the text will become:
  "jupyter notebook"
  This ensures that the text does not have unnecessary spaces at the beginning or end, thereby ensuring consistency in text formatting and avoiding problems that can occur when searching or processing data. Example:

```
In [35]: import pandas as pd

         data = {'Fruits': ['  Apple  ', 'Banana  ', '  Cherry  ', 'Date  ']}
         print(data)

         {'Fruits': ['  Apple  ', 'Banana  ', '  Cherry  ', 'Date  ']}

In [36]: df = pd.DataFrame(data)

In [37]: df['Fruits'] = df['Fruits'].str.strip()

In [38]: df.head()

Out[38]:
```

	Fruits
0	Apple
1	Banana
2	Cherry
3	Date

- **Inconsistent formatting:** Standardizing data formats, such as date formats, to make them consistent.Example, change the data format in the date column to YYYY-MM-DD for consistency

```
In [47]: df['Transaction_date'] = pd.to_datetime(df['Transaction_date'])
```

```
In [48]: df['Transaction_date'] = df['Transaction_date'].dt.strftime('%Y-%m-%d')
```

```
In [49]: df.head(10)
```

Out[49]:

	Transaction_date	Transaction_ID	Gender	Age	Marital_status	State_names	Segment	Employees_status	Payment_method	Referal	Amount_spent
0	2019-01-01	151200	Female	19.0	Single	Kansas	Basic	Unemployment	Other	Use	2051.360
1	2019-01-01	151201	Male	49.0	Single	Illinois	Basic	self-employed	Card	Not use	544.040
2	2019-01-01	151202	Male	63.0	Married	New Mexico	Basic	workers	PayPal	Use	1572.600
3	2019-01-01	151203	Female	18.0	Single	Virginia	Platinum	workers	Card	Use	1199.790
4	2019-01-01	151204	Male	27.0	Single	Connecticut	Basic	self-employed	Card	Not use	1341.435
5	2019-01-03	151205	Male	71.0	Single	Hawaii	Basic	Employees	PayPal	Use	2922.660
6	2019-01-03	151206	Female	34.0	Married	New Mexico	Platinum	Employees	PayPal	Use	1481.420
7	2019-01-03	151207	Male	37.0	Married	Connecticut	Basic	workers	PayPal	Use	1149.550
8	2019-01-04	151208	Male	75.0	Married	Florida	Silver	Employees	Card	Not use	1046.200
9	2019-01-04	151209	Female	41.0	Married	Vermont	Gold	Unemployment	Card	Use	2730.600

- **Removing duplicates:** Identifying and removing duplicate records to ensure data accuracy.
  df.duplicated().sum()
  df_new = df.drop_duplicates()
  df_new.duplicated().sum()

Check for duplicate data in the table

```
In [13]: df.duplicated().sum()
Out[13]: 12
```

Removing the duplicates

```
In [14]: df_new = df.drop_duplicates()
```

Check data in the table after duplicate data has been deleted

```
In [15]: df_new.duplicated().sum()
Out[15]: 0
```

- **Imputing data:** Filling in missing data with appropriate values based on rules or algorithms.
  df.Gender[df.Gender.isnull()]
  df.Gender.value_counts()
  val = df.Gender.mode().values[0]
  df['Gender'] = df.Gender.fillna(val)
  df.Gender.value_counts()

Check the amount of data/values in the categories in the Gender column

```
In [7]: df.Gender.value_counts()
Out[7]: Female    1356
        Male      1128
        Name: Gender, dtype: int64
```

From the proportion of the Gender column, Female is the data that appears most often, so Female is the mode

```
In [8]: val = df.Gender.mode().values[0]
        df['Gender'] = df.Gender.fillna(val)
```

After imputation, it can be seen that the proportions have changed

```
In [9]: df.Gender.value_counts()
Out[9]: Female    1384
        Male      1128
        Name: Gender, dtype: int64
```

- **Validating data:** Checking data for correctness and validity to ensure it meets predefined criteria or constraints
  Before starting validation, it is important to examine the data by understanding the structure, data types, and any potential problems. It can use the head(), info(), and describe() methods of pandas DataFrame for this purpose.
  df.head()
  df.info()

```
In [62]: df.head()
Out[62]:
```

	Transaction_date	Transaction_ID	Gender	Age	Marital_status	State_names	Segment	Employees_status	Payment_method	Referal	Amount_spent
0	1/1/2019	151200	Female	19.0	Single	Kansas	Basic	Unemployment	Other	1.0	2051.360
1	1/1/2019	151201	Male	49.0	Single	Illinois	Basic	self-employed	Card	0.0	544.040
2	1/1/2019	151202	Male	63.0	Married	New Mexico	Basic	workers	PayPal	1.0	1572.600
3	1/1/2019	151203	Female	18.0	Single	Virginia	Platinum	workers	Card	1.0	1199.790
4	1/1/2019	151204	Male	27.0	Single	Connecticut	Basic	self-employed	Card	0.0	1341.435

```
In [28]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2512 entries, 0 to 2511
Data columns (total 11 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Transaction_date  2512 non-null   object
 1   Transaction_ID    2512 non-null   int64
 2   Gender            2512 non-null   object
 3   Age               2512 non-null   float64
 4   Marital_status    2512 non-null   object
 5   State_names       2512 non-null   object
 6   Segment           2512 non-null   object
 7   Employees_status  2512 non-null   object
 8   Payment_method    2512 non-null   object
 9   Referal           2512 non-null   float64
 10  Amount_spent      2512 non-null   float64
dtypes: float64(3), int64(1), object(7)
memory usage: 216.0+ KB
```

Based on the results of the data inspection, it was found that the referral column used the float data type, this was deemed unsuitable, the data would be easier to understand if converted into a string, where "1.0" means using a referral code, while "0.0" means not using a code referral. This will help people who read the data so that it is easier to understand. (Data Type Validating)
df['Referal'] = df['Referal'].astype(str)
df['Referal'] = df['Referal'].replace({1.0: 'Use', 0.0: 'Not use'})

Check the data in the table after validating the data type

```
In [42]: df.head()
Out[42]:
```

	Transaction_date	Transaction_ID	Gender	Age	Marital_status	State_names	Segment	Employees_status	Payment_method	Referal	Amount_spent
0	1/1/2019	151200	Female	19.0	Single	Kansas	Basic	Unemployment	Other	Use	2051.360
1	1/1/2019	151201	Male	49.0	Single	Illinois	Basic	self-employed	Card	Not use	544.040
2	1/1/2019	151202	Male	63.0	Married	New Mexico	Basic	workers	PayPal	Use	1572.600
3	1/1/2019	151203	Female	18.0	Single	Virginia	Platinum	workers	Card	Use	1199.790
4	1/1/2019	151204	Male	27.0	Single	Connecticut	Basic	self-employed	Card	Not use	1341.435

```
In [43]: df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2512 entries, 0 to 2511
Data columns (total 11 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   Transaction_date  2512 non-null   object
 1   Transaction_ID    2512 non-null   int64
 2   Gender            2512 non-null   object
 3   Age               2512 non-null   float64
 4   Marital_status    2512 non-null   object
 5   State_names       2512 non-null   object
 6   Segment           2512 non-null   object
 7   Employees_status  2512 non-null   object
 8   Payment_method    2512 non-null   object
 9   Referal           2512 non-null   object
 10  Amount_spent      2512 non-null   float64
dtypes: float64(2), int64(1), object(8)
memory usage: 216.0+ KB
```

2.3 Organize data:

Purpose and common practices:

- **Sorting:** Reordering data based on one or more columns, usually in ascending or descending order.
  example:

```
# Sample data
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
        'Age': [25, 30, 22, 35, 28],
        'Salary': [50000, 60000, 45000, 70000, 55000]}

# Sorting: Reorder data based on the 'Age' column in ascending order
sorted_data = sorted(zip(data['Name'], data['Age'], data['Salary']), key=lambda x: x[1])
print("Sorted data by Age:", sorted_data)
```

```
In [1]: # Sample data
        data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
                'Age': [25, 30, 22, 35, 28],
                'Salary': [50000, 60000, 45000, 70000, 55000]}

In [2]: # Sorting: Reorder data based on the 'Age' column in ascending order
        sorted_data = sorted(zip(data['Name'], data['Age'], data['Salary']), key=lambda x: x[1])
        print("Sorted data by Age:", sorted_data)

        Sorted data by Age: [('Charlie', 22, 45000), ('Alice', 25, 50000), ('Eve', 28, 55000), ('Bob', 30, 60000), ('David', 35, 7000
        0)]
```

- **Filtering:** Selecting a subset of data based on specified criteria.
  example:

```
# Filtering: Selecting records with Age greater than 25
filtered_data = [(name, age, salary) for name, age, salary in zip(data['Name'], data['Age'], data['Salary']) if age > 25]
print("Filtered data:", filtered_data)
```

```
In [3]: # Filtering: Selecting records with Age greater than 25
        filtered_data = [(name, age, salary) for name, age, salary in zip(data['Name'], data['Age'], data['Salary']) if age > 25]
        print("Filtered data:", filtered_data)

        Filtered data: [('Bob', 30, 60000), ('David', 35, 70000), ('Eve', 28, 55000)]
```

- **Slicing:** Extracting a specific range or portion of the data.
  example:

```
# Slicing: Extracting the second and third records
sliced_data = (data['Name'][1:3], data['Age'][1:3], data['Salary'][1:3])
print("Sliced data:", sliced_data)
```

```
In [4]: # Slicing: Extracting the second and third records
        sliced_data = (data['Name'][1:3], data['Age'][1:3], data['Salary'][1:3])
        print("Sliced data:", sliced_data)

        Sliced data: (['Bob', 'Charlie'], [30, 22], [60000, 45000])
```

- **Transposing:** Changing the orientation of data, such as converting rows to columns or vice versa.
  example:

```
# Transposing: Changing rows to columns using zip
transposed_data = {'Name': data['Name'], 'Age': data['Age'], 'Salary': data['Salary']}
print("Transposed data:", list(zip(*transposed_data.values())))
```

```
In [5]: # Transposing: Changing rows to columns using zip
        transposed_data = {'Name': data['Name'], 'Age': data['Age'], 'Salary': data['Salary']}
        print("Transposed data:", list(zip(*transposed_data.values())))

        Transposed data: [('Alice', 25, 50000), ('Bob', 30, 60000), ('Charlie', 22, 45000), ('David', 35, 70000), ('Eve', 28, 55000)]
```

- **Appending:** Combining or adding new data to an existing dataset.
  example:
  # Appending: Adding new data to the existing dataset
  new_data = {'Name': ['Frank', 'Grace'], 'Age': [29, 32], 'Salary': [52000, 60000]}
  appended_data = {key: data[key] + new_data[key] for key in data}
  print("Appended data:", appended_data)

```
In [6]: # Appending: Adding new data to the existing dataset
        new_data = {'Name': ['Frank', 'Grace'], 'Age': [29, 32], 'Salary': [52000, 60000]}
        appended_data = {key: data[key] + new_data[key] for key in data}
        print("Appended data:", appended_data)

        Appended data: {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank', 'Grace'], 'Age': [25, 30, 22, 35, 28, 29, 32], 'Sa
        lary': [50000, 60000, 45000, 70000, 55000, 52000, 60000]}
```

- **Truncating:** Reducing the data to a specific length or number of rows, often to create smaller subsets of data.
  example:
  # Truncating: Reducing the dataset to the first three records
  truncated_data = {key: data[key][:3] for key in data}
  print("Truncated data:", truncated_data)

```
In [7]: # Truncating: Reducing the dataset to the first three records
        truncated_data = {key: data[key][:3] for key in data}
        print("Truncated data:", truncated_data)

        Truncated data: {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 22], 'Salary': [50000, 60000, 45000]}
```

2.4 Aggregate data:

Purpose and common practices:

- **Grouping:** Grouping data by one or more columns to perform operations on subsets of data, often used with aggregation functions like SUM, COUNT, AVG.
  example:
  1. SUM

```
import pandas as pd

data = {
    'Category': ['Electronics', 'Clothing', 'Electronics', 'Clothing', 'Electronics'],
    'Amount': [500, 200, 300, 150, 700]
}

df = pd.DataFrame(data)

# Grouping by 'Category' and calculating the total sales (SUM)
grouped_data = df.groupby('Category')['Amount'].sum()
print(grouped_data)
```

```
In [8]: import pandas as pd

data = {
    'Category': ['Electronics', 'Clothing', 'Electronics', 'Clothing', 'Electronics'],
    'Amount': [500, 200, 300, 150, 700]
}

df = pd.DataFrame(data)

# Grouping by 'Category' and calculating the total sales (SUM)
grouped_data = df.groupby('Category')['Amount'].sum()
print(grouped_data)

Category
Clothing        350
Electronics    1500
Name: Amount, dtype: int64
```

2. COUNT
   count_data = df.groupby('Category')['Amount'].count()
   print(count_data)

```
In [9]: count_data = df.groupby('Category')['Amount'].count()
        print(count_data)

Category
Clothing       2
Electronics    3
Name: Amount, dtype: int64
```

3. AVG
   average_data = df.groupby('Category')['Amount'].mean()
   print(average_data)

```
In [10]: average_data = df.groupby('Category')['Amount'].mean()
         print(average_data)

Category
Clothing       175.0
Electronics    500.0
Name: Amount, dtype: float64
```

- **Joining/Merging:** Combining data from multiple sources or tables using keys or common columns.
  example:

```
customer_data = {
    'CustomerID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
}

purchase_data = {
    'CustomerID': [2, 1, 3],
    'Product': ['Laptop', 'Phone', 'Tablet']
}

customers = pd.DataFrame(customer_data)
purchases = pd.DataFrame(purchase_data)

# Merge the two DataFrames on 'CustomerID'
merged_data = pd.merge(customers, purchases, on='CustomerID')
print(merged_data)
```

```
In [11]: customer_data = {
             'CustomerID': [1, 2, 3],
             'Name': ['Alice', 'Bob', 'Charlie']
         }

         purchase_data = {
             'CustomerID': [2, 1, 3],
             'Product': ['Laptop', 'Phone', 'Tablet']
         }

         customers = pd.DataFrame(customer_data)
         purchases = pd.DataFrame(purchase_data)

         # Merge the two DataFrames on 'CustomerID'
         merged_data = pd.merge(customers, purchases, on='CustomerID')
         print(merged_data)

            CustomerID     Name Product
         0           1    Alice   Phone
         1           2      Bob  Laptop
         2           3  Charlie  Tablet
```

- **Summarizing:** Creating summary statistics or aggregations to get an overview of the data, such as calculating totals, averages, or counts.
  example:
  summary_stats = df.describe()
  print(summary_stats)

```
In [12]: summary_stats = df.describe()
         print(summary_stats)

                    Amount
         count    5.000000
         mean   370.000000
         std    228.035085
         min    150.000000
         25%    200.000000
         50%    300.000000
         75%    500.000000
         max    700.000000
```

- **Pivoting:** Restructuring data to transform rows into columns or vice versa, often used for creating summary tables or pivot tables.
  example:
  pivoted_data = df.pivot(index='Category', columns='Amount', values='Amount')
  print(pivoted_data)

```
In [13]: pivoted_data = df.pivot(index='Category', columns='Amount', values='Amount')
         print(pivoted_data)

         Amount         150    200    300    500    700
         Category
         Clothing     150.0  200.0    NaN    NaN    NaN
         Electronics    NaN    NaN  300.0  500.0  700.0
```