Pandas

Reading and Writing Data:

- **pd.read_csv():** Read data from a CSV file into a DataFrame.
- **to_dict():** Convert DataFrame to a dictionary.

DataFrame Creation and Manipulation:

- **pd.DataFrame():** Create a DataFrame from an existing object (in this case, x).
- **to_csv():** Write DataFrame to a CSV file.
- Accessing unique values in a column: df["Manufacturer"]**.unique()**.
- Filtering DataFrame based on conditions: **df**[df.Manufacturer=="Toyota"], **df.query**("Manufacturer=='Toyota'"), etc.

Missing Data Handling:

- **isnull(), notna(), notnull():** Check for missing values.
- **fillna():** Fill missing values with specified values or methods (e.g., forward fill, backward fill).
- **dropna():** Drop rows or columns with missing values.
- **drop_duplicates():** Remove duplicate rows.

Grouping and Aggregation:

- **groupby():** Group DataFrame using a mapper or by a Series of columns.
- **Aggregating grouped data:** ["Sales_in_thousands"].min().

Sorting Data:

- **sort_values():** Sort DataFrame by one or more columns.

Data Exploration:

- **value_counts():** Count unique values in a column.
- **corr():** Compute pairwise correlation of columns.
- Crosstabulation: **pd.crosstab()**.

Data Transformation:

- **replace():** Replace values in a column with other values.
- **map():** Map values of Series using input correspondence (e.g., dictionary).

Setting DataFrame Options:

- **pd.set_option():** Set options for controlling the display of DataFrame.

Resetting Index:

- data_one = **data.reset_index():** This line resets the index of the DataFrame data and assigns the result to a new DataFrame data_one. This operation adds a new column named "index" containing the old index values.

Counting Non-Null Values:

- **df.count():** This function returns the number of non-null values for each column in the DataFrame df.

Checking for Non-Null Values:

- **notnull().sum():** This code seems to be incomplete. It should be df.notnull().sum(). It counts the number of non-null values for each column in the DataFrame df.

Pandas

Checking for Duplicate Rows:

- **df.duplicated():** This function identifies duplicate rows in the DataFrame df. By default, it marks duplicates as True except for the first occurrence.
- **df.duplicated(keep="first"):** This is the same as the previous line. It marks duplicates as True except for the first occurrence.
- **df.duplicated(keep="last"):** This marks duplicates as True except for the last occurrence.
- **df.duplicated(keep=False):** This marks all duplicates as True.

```
import pandas as pd
#Before you can use Pandas, you need to import it into your Python environment
```

Data Structures:

Pandas primarily works with two main data structures: Series and DataFrame.

1. **Series:** A one-dimensional labeled array capable of holding any data type.
2. **DataFrame:** A two-dimensional labeled data structure with columns of potentially different types. It is like a spreadsheet or SQL table.

Creating DataFrames:

You can create a DataFrame from various data sources such as dictionaries, lists, NumPy arrays, CSV files, Excel files, SQL queries, etc.

Example:

```
import pandas as pd

# Create a DataFrame from a dictionary
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 40],
        'Salary': [50000, 60000, 70000, 80000]}

df = pd.DataFrame(data)
print(df)

# Create a DataFrame from lists
names = ['Alice', 'Bob', 'Charlie', 'David']
ages = [25, 30, 35, 40]
salaries = [50000, 60000, 70000, 80000]

df = pd.DataFrame({'Name': names, 'Age': ages, 'Salary': salaries})

print(df)
```

IO tools:

These tools facilitate reading data from various sources and writing data to different formats.

1. **pd.read_csv():** Read data from a CSV file into a pandas DataFrame.
2. **DataFrame.to_csv():** Write DataFrame to a CSV file.
3. **pd.read_excel():** Read data from an Excel file into a pandas DataFrame.

4. **DataFrame.to_excel():** Write DataFrame to an Excel file.
5. **pd.read_json():** Read data from a JSON file into a pandas DataFrame.
6. **DataFrame.to_json():** Write DataFrame to a JSON file.
7. **pd.read_sql():** Read data from a SQL database into a pandas DataFrame.
8. **DataFrame.to_sql():** Write DataFrame to a SQL database.
9. **pd.read_html():** Read HTML tables from a webpage into a list of pandas DataFrames.
10. **DataFrame.to_dict():** Convert DataFrame to a dictionary.

Example:

```
import pandas as pd
# Read data from a CSV file into a DataFrame
df = pd.read_csv('data.csv')

# Display the DataFrame
print(df)
```

Viewing Data:

1. head(): This method displays the first n rows of the DataFrame. By default, it shows the first 5 rows.
   - **df.head()** # Displays the first 5 rows
   - **df.head(n)** # Displays the first n rows

2. tail(): Similar to head(), but displays the last n rows of the DataFrame.
   - **df.tail()** # Displays the last 5 rows
   - **df.tail(n)** # Displays the last n rows

3. sample(): This method returns a random sample of the DataFrame. The number of rows to return can be specified.
   - **df.sample()** # Returns a single random row
   - **df.sample(n)** # Returns n random rows

4. info(): This method provides a concise summary of the DataFrame, including the data types of each column and the number of non-null values.
   - **df.info()**

5. describe(): This method generates descriptive statistics that summarize the central tendency, dispersion, and shape of the dataset's distribution.
   - **df.describe()**

6. shape: This attribute returns a tuple representing the dimensionality of the DataFrame (number of rows, number of columns).
   - **df.shape**

7. query() method allows you to filter rows from a DataFrame using a query expression. This method provides a more concise and readable way to filter data compared to boolean indexing or other methods.
   - **DataFrame.query(expr, inplace=False, **kwargs)**
     - **expr:** A string containing the query expression to filter rows. It can reference column names directly without needing to specify the DataFrame name.

- **inplace:** A boolean indicating whether to modify the DataFrame in place or return a new DataFrame with the filtered rows. The default is False, which means it returns a new DataFrame.
- ****kwargs:** Additional keyword arguments that are passed to the expr namespace.

Example:

```
import pandas as pd

# Creating a sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emily'],
    'Age': [25, 30, 35, 40, 45],
    'Salary': [50000, 60000, 70000, 80000, 90000],
    'Department': ['HR', 'IT', 'Finance', 'Marketing', 'Sales']
}

df = pd.DataFrame(data)

# View the first 3 rows of the DataFrame
print("Head:\n", df.head(3))

# View the last 2 rows of the DataFrame
print("\nTail:\n", df.tail(2))

# Get the dimensions of the DataFrame (rows, columns)
print("\nShape:", df.shape)

# Get concise summary information about the DataFrame
print("\nInfo:")
df.info()

# Get descriptive statistics for numeric columns in the DataFrame
print("\nDescription:")
print(df.describe())

# Using query() to filter rows where Age is greater than 30 and Salary is less than 80000
filtered_df = df.query('Age > 30 and Salary < 80000')
print("\nFiltered DataFrame:")
print(filtered_df)

# Using query() with inplace=True to modify the DataFrame in place
df.query('Age > 30', inplace=True)
print("\nDataFrame after inplace modification:")
print(df)

# Using query() with additional keyword arguments
threshold = int(input("Enter threshold value: "))
result = df.query('Age > @threshold')
print("\nDataFrame filtered with additional argument:")
```

```
print(result)
```

Output:

```
Head:
      Name  Age  Salary Department
0   Alice   25   50000         HR
1     Bob   30   60000         IT
2  Charlie   35   70000    Finance

Tail:
    Name  Age  Salary Department
3  David   40   80000  Marketing
4  Emily   45   90000      Sales

Shape: (5, 4)

Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Name        5 non-null      object
 1   Age         5 non-null      int64
 2   Salary      5 non-null      int64
 3   Department  5 non-null      object
dtypes: int64(2), object(2)
memory usage: 288.0+ bytes

Description:
             Age        Salary
count   5.000000      5.000000
mean   35.000000  72000.000000
std     8.660254  15811.388301
min    25.000000  50000.000000
25%    30.000000  60000.000000
50%    35.000000  70000.000000
75%    40.000000  80000.000000
max    45.000000  90000.000000

Filtered DataFrame:
    Name  Age  Salary Department
2  Charlie   35   70000    Finance

DataFrame after inplace modification:
    Name  Age  Salary Department
2  Charlie   35   70000    Finance
3    David   40   80000  Marketing
4    Emily   45   90000      Sales
```

```
Enter threshold value: 35

DataFrame filtered with additional argument:
   Name  Age  Salary Department
3  David   40   80000  Marketing
4  Emily   45   90000      Sales
```

Indexing and Selection:

Bracket notation: You can access a single column using square brackets and the column name as a string

- df['column_name']

Dot notation: If the column name is a valid Python identifier and doesn't conflict with DataFrame methods, you can also use dot notation.

- df.column_name

Label-based indexing with .loc[]: Use labels to slice rows and columns.

- df.loc[row_label, column_label]
- df.loc[row_label]['column_name']

Integer-based indexing with .iloc[]: Use integer positions to slice rows and columns.

- df.iloc[row_index, column_index]
- df.loc[row_label, 'column_name']

Boolean indexing: Select rows based on a condition.

- df[df['column_name'] > value]

Setting values with .loc[] or .iloc[]:

- df.loc[row_label, 'column_name'] = new_value

Example 1:

```
import pandas as pd

# Creating a sample DataFrame
data = {
   'A': [1, 2, 3, 4, 5],
   'B': [6, 7, 8, 9, 10],
   'C': ['a', 'b', 'c', 'd', 'e']
}
df = pd.DataFrame(data)

# Bracket notation: Accessing a single column
column_A_bracket = df['A']
print("Bracket notation:", column_A_bracket)

# Dot notation: Accessing a single column
```

```
column_B_dot = df.B
print("Dot notation:", column_B_dot)

# Label-based indexing with .loc[]: Slicing rows and columns
row_2_col_B_loc = df.loc[2, 'B']
print("Label-based indexing with .loc[]:", row_2_col_B_loc)

# Chained indexing with .loc[] and bracket notation
row_3_col_C_chain = df.loc[3]['C']
print("Chained indexing with .loc[] and bracket notation:", row_3_col_C_chain)

# Integer-based indexing with .iloc[]: Slicing rows and columns
row_0_col_1_iloc = df.iloc[0, 1]
print("Integer-based indexing with .iloc[]:", row_0_col_1_iloc)

# Boolean indexing: Selecting rows based on a condition
conditioned_rows = df[df['A'] > 3]
print("Boolean indexing:", conditioned_rows)

# Setting values with .loc[]
df.loc[0, 'A'] = 100
print("DataFrame after setting value with .loc[]:")
print(df)
```

Output:

```
Bracket notation: 0    1
1    2
2    3
3    4
4    5
Name: A, dtype: int64
Dot notation: 0     6
1     7
2     8
3     9
4    10
Name: B, dtype: int64
Label-based indexing with .loc[]: 8
Chained indexing with .loc[] and bracket notation: d
Integer-based indexing with .iloc[]: 6
Boolean indexing:    A   B  C
3  4   9  d
4  5  10  e
DataFrame after setting value with .loc[]:
     A   B  C
0  100   6  a
1    2   7  b
2    3   8  c
3    4   9  d
4    5  10  e
```

Example 2:

```
import pandas as pd

# Creating a sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emily'],
    'Age': [25, 30, 35, 40, 45],
    'Salary': [50000, 60000, 70000, 80000, 90000],
    'Department': ['HR', 'IT', 'Finance', 'Marketing', 'Sales']
}

df = pd.DataFrame(data)

# Selecting a single column
print("Selecting a single column using df['column_name']:\n", df['Name'])
print("\nSelecting a single column using df.column_name:\n", df.Name)

# Selecting rows by label
print("\nSelecting rows by label using df.loc:\n", df.loc[2])  # Selects the row with label 2

# Selecting rows by position
print("\nSelecting rows by position using df.iloc:\n", df.iloc[3])  # Selects the row at position 3

# Slicing rows
print("\nSlicing rows using df[start:end]:\n", df[1:3])  # Selects rows 1 to 2 (end index exclusive)

# Conditional selection
print("\nConditional selection using df[df['column_name'] > value]:\n", df[df['Age'] > 30])  # Selects rows where Age > 30
```

Output:

```
Selecting a single column using df['column_name']:
 0      Alice
1        Bob
2    Charlie
3      David
4      Emily
Name: Name, dtype: object

Selecting a single column using df.column_name:
 0      Alice
1        Bob
2    Charlie
3      David
4      Emily
Name: Name, dtype: object

Selecting rows by label using df.loc:
 Name         Charlie
Age             35
```

```
Salary         70000
Department    Finance
Name: 2, dtype: object
```

Selecting rows by position using df.iloc:

```
 Name         David
Age            40
Salary        80000
Department      Marketing
Name: 3, dtype: object
```

Slicing rows using df[start:end]:

```
      Name  Age  Salary Department
1      Bob   30   60000         IT
2  Charlie   35   70000    Finance
```

Conditional selection using df[df['column_name'] > value]:

```
    Name  Age  Salary Department
2  Charlie   35   70000    Finance
3    David   40   80000  Marketing
4    Emily   45   90000      Sales
```

Data Manipulation:

1. Adding a new column: df['new_column'] = value
2. Removing columns: df.drop(columns=['column_name'], inplace=True)
3. Renaming columns: df.rename(columns={'old_name': 'new_name'}, inplace=True)
4. Sorting data: df.sort_values(by='column_name', ascending=True)

Aggregation and Grouping:

1. df.groupby('column_name').agg({'agg_column': 'agg_function'})