Pandas syntax

```python
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:

```python
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)
```

```python
# From a NumPy array
import numpy as np
data = np.array([['Alice', 25, 'New York'], ['Bob', 30, 'Los Angeles'], ['Charlie', 35, 'Chicago']])
df = pd.DataFrame(data, columns=['Name', 'Age', 'City'])
```

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.

Pandas syntax

10. **DataFrame.to_dict():** Convert DataFrame to a dictionary.

```
# From SQL database
import sqlite3
conn = sqlite3.connect('database.db')
df = pd.read_sql_query("SELECT * FROM table_name", conn)
```

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.

Indexing and Selection:

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

- df['column_name']
- df['column_name1', 'column_name2']

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

Filtering Data:

Filtering based on conditions

1. print(df[df['Age'] > 25])

Multiple conditions

2. print(df[(df['Age'] > 25) & (df['City'] == 'New York')])

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= ["column1" , "column2"], ascending= [True, False] , inplace=True)

Aggregation and Grouping:

Pandas syntax

1. df.groupby('column_name').agg({'agg_column': 'agg_function'})
2. df.groupby("column1")["column2"].sum()

Data Cleaning

Handling Missing Data:

Missing values are common in datasets and can adversely affect analysis. Pandas provides methods to detect and handle missing values.

Check for missing values # **isnull(), notna(), notnull():**

1. print(df.isnull())

Drop rows with missing values

2. df.dropna(inplace=True)
3. df.dropna(how = "any" , inplace = True)
4. df.dropna(how = "all", subset = ["column1","column2"])

Fill missing values with a specified value

5. df.fillna(0, inplace=True)
6. df["column"].fillna(value, inplace=True)

Removing Duplicates:

Duplicate rows can skew analysis results. Pandas makes it easy to identify and remove duplicates.

Identify duplicates

7. print(df.duplicated())

Remove duplicates

8. df.drop_duplicates(inplace=True) or df.drop_duplicates(keep = False, inplace=True)
    - keep=["first", "last". False]

Handling Outliers:

Outliers are extreme values that deviate from other observations. Detecting and dealing with outliers is essential for accurate analysis.

```
# Define a function to detect outliers
def detect_outliers(df, column):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    return df[(df[column] < lower_bound) | (df[column] > upper_bound)]

# Detect and handle outliers
outliers_df = detect_outliers(df, 'Age')
df = df[~df.index.isin(outliers_df.index)]
```

Data Transformation:

Data often needs to be transformed into a suitable format for analysis. Pandas offers various methods for data transformation, including converting data types, applying functions to columns, and reshaping data.

Convert data types

Pandas syntax

```
df['Age'] = df['Age'].astype(int)

# Apply a function to a column
df['Income'] = df['Income'].apply(lambda x: x * 0.8)  # Adjust income by 20%
df["column"]=df["column"].apply(lambda x:x,replace("-",""))
df["column"] = df["column"].str.replace("&", "and")
df["new_column"] = df["column"].str.extract(r'(\d+)')

#Remove Leading and Trailing Whitespace from String Columns:
df["column"] = df["column"].str.strip()

# Reshape data
# For example, pivot tables, melting, stacking, etc.
```

Handling Categorical Data:

Categorical variables need to be encoded numerically for analysis. Pandas provides methods for encoding categorical variables and handling text data.

```
# Encoding categorical variables
df = pd.get_dummies(df, columns=['City'])

# Convert Continuous Variable to Categorical Variable:
df["categorical_column"] = pd.cut(df["numeric_column"], bins, labels=labels)

#Convert String Column to Lowercase:
df["column"] = df["column"].str.lower()

#Convert String Column to Uppercase:
df["column"] = df["column"].str.upper()

# Handling text data
# For example, removing punctuation, converting to lowercase, tokenization, etc.
```

Handling Date and Time Data:

If your dataset includes date and time information, pandas offers functionality to handle and manipulate datetime objects.

```
# Convert to datetime
df['Date'] = pd.to_datetime(df['Date'])

# Extract components (e.g., year, month, day)
df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
df['Day'] = df['Date'].dt.day

df["column"] = df["column"].dt.strftime("%d-%m-%y)
```

Scaling and Normalization:

In some cases, it's necessary to scale or normalize numerical data to ensure all features contribute equally to the analysis.

Pandas syntax

```
from sklearn.preprocessing import MinMaxScaler
# Initialize the scaler
scaler = MinMaxScaler()

# Fit and transform the data
df[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
```

Pandas syntax

Data Exploration:

- Accessing unique values in a column: df["Manufacturer"]**.unique()**.
- **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.