# The Pandas Playbook: Data Analysis Made Easy

Pandas is an open-source Python library used for data manipulation and analysis.
It provides data structures like **DataFrame** and **Series**, which make it easy to clean, explore, and analyze structured data.

Pandas can be thought of as a collection of various functionalities and tools that make data manipulation and analysis easier.
Here's a breakdown of some of its key parts:

**Data Structures**

1. **Series**: A one-dimensional labeled array that can hold any data type.
2. **DataFrame**: A two-dimensional labeled data structure with columns of potentially different types, similar to a table in a database or an Excel spreadsheet.

**Data Input/Output**

- **Reading Data**: Functions like read_csv(), read_excel(), read_sql(), and read_json() allow you to read data from various file formats.
- **Writing Data**: Functions like to_csv(), to_excel(), to_sql(), and to_json() let you export data to different formats.

**Data Manipulation**

- **Indexing and Selection**: Use .loc and .iloc for label-based and position-based indexing, respectively.
- **Filtering**: Easily filter data based on conditions.
- **Sorting**: Sort data by labels or values using sort_index() and sort_values().
- **Merging**: Combine data from multiple DataFrames using merge(), join(), and concat().

**Data Cleaning**

- **Handling Missing Data**: Functions like isnull(), dropna(), and fillna() help in detecting and handling missing data.
- **Data Transformation**: Apply custom functions to data using .apply() and .map().

**Data Aggregation and Grouping**

- **Grouping**: Group data using groupby() and perform aggregate operations like sum, mean, and count.
- **Pivot Tables**: Create pivot tables using pivot_table() to summarize data.

**Time Series Analysis**

- **Date and Time Functions**: Work with date and time data using functions like pd.to_datetime().
- **Resampling**: Resample time series data using resample() for frequency conversion.

**Data Visualization**

- **Plotting**: Basic plotting capabilities using .plot(), integrated with Matplotlib.
- **Integration**: Seamless integration with other visualization libraries like Seaborn for more complex visualizations.

**Statistical Functions**

- **Descriptive Statistics**: Functions like mean(), median(), std(), and describe() provide summary statistics.
- **Correlation and Covariance**: Analyze relationships between data using corr() and cov().

Here's a simple example to show how some of these parts work together:

```
import pandas as pd

# Read data from a CSV file
df = pd.read_csv('your_file.csv')

# Display the first few rows of the dataframe
print(df.head())

# Filter data where a column 'A' has values greater than 50
filtered_df = df[df['A'] > 50]

# Group by a column 'B' and calculate the mean of column 'C'
grouped_df = filtered_df.groupby('B')['C'].mean()

# Plot the grouped data
grouped_df.plot(kind='bar')
```

# 1. Data Structures

## Series :

A Series is a one-dimensional labeled array that can hold data of any type (integer, float, string, etc.). It's similar to a column in a spreadsheet or a SQL table. Each element in a Series is assigned a unique index label, which can be used to access individual elements.

**Key Features of Series**:

- **Homogeneous Data**: All elements in a Series have the same data type.
- **Automatic Indexing**: Each element is assigned an index starting from 0.
- **Custom Indexing**: You can define your own index labels.

**Example**:

```
import pandas as pd

# Creating a Series from a list
data = [1, 2, 3, 4, 5]
series = pd.Series(data)
print(series)

# Creating a Series with custom index labels
data = [10, 20, 30, 40, 50]
index = ['a', 'b', 'c', 'd', 'e']
series = pd.Series(data, index=index)
print(series)
```

## DataFrame :

A DataFrame is a two-dimensional labeled data structure with columns of potentially different types. It’s like a table in a database or a spreadsheet with rows and columns. Each column in a DataFrame is a Series.

**Key Features of DataFrame**:

- **Heterogeneous Data**: Columns can have different data types (e.g., integers, floats, strings).
- **Labeled Axes**: Both rows and columns can be labeled.
- **Size-Mutable**: You can insert and delete columns from a DataFrame.

**Example**:

```
import pandas as pd

# Creating a DataFrame from a dictionary
data = {
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 30, 35],
   'Salary': [50000, 60000, 70000]
}
df = pd.DataFrame(data)
print(df)

# Accessing DataFrame columns
print(df['Name'])

# Accessing DataFrame rows using iloc
print(df.iloc[0])

# Accessing DataFrame rows using loc with custom index
df = df.set_index('Name')
print(df.loc['Alice'])
```

## Common DataFrame Operations:

- **Selection**: Selecting rows and columns using .loc (label-based) and .iloc (position-based).
- **Filtering**: Filtering data based on conditions.
- **Adding/Removing Columns**: Adding new columns or dropping existing ones.
- **Aggregation**: Performing aggregate operations like sum, mean, and count.
- **Merging**: Combining multiple DataFrames using merge(), join(), and concat().

**Example** of a few operations:

```
import pandas as pd

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

# Adding a new column
df['D'] = df['A'] + df['B']
```

```
print(df)

# Filtering rows where column 'A' > 1
filtered_df = df[df['A'] > 1]
print(filtered_df)

# Aggregating data
agg_df = df.agg({'A': 'sum', 'B': 'mean'})
print(agg_df)
```

## Practical Use Cases:

- **Data Cleaning**: Detect and handle missing data, remove duplicates, and apply transformations.
- **Exploratory Data Analysis (EDA)**: Summarize data, visualize distributions, and detect patterns.
- **Data Modeling**: Prepare datasets for machine learning models by manipulating and transforming data.
- **Reporting**: Create summary statistics and export results to various formats (CSV, Excel, etc.).

# 2. Data Input/Output

## Reading Data

1. read_csv(): Reads data from a CSV file.
    - **Definition**: Reads a comma-separated values (CSV) file into a DataFrame.
    - **Syntax**: pd.read_csv(filepath)
    - **Example**:

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.head())
```

2. read_excel(): Reads data from an Excel file.
    - **Definition**: Reads an Excel file into a DataFrame.
    - **Syntax**: pd.read_excel(filepath, sheet_name='Sheet1')
    - **Example**:

```
import pandas as pd
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
print(df.head())
```

3. read_sql(): Reads data from a SQL database.
    - **Definition**: Reads a SQL query or database table into a DataFrame.
    - **Syntax**: pd.read_sql(query, connection)
    - **Example**:

```
import pandas as pd
import sqlite3

conn = sqlite3.connect('database.db')
df = pd.read_sql('SELECT * FROM table_name', conn)
print(df.head())
```

# Writing Data

1. to_csv(): Writes data to a CSV file.
    - **Definition**: Writes DataFrame to a comma-separated values (CSV) file.
    - **Syntax**: df.to_csv(filepath, index=False)
    - **Example**:

```
df.to_csv('output.csv', index=False)
```

2. to_excel(): Writes data to an Excel file.
    - **Definition**: Writes DataFrame to an Excel file.
    - **Syntax**: df.to_excel(filepath, sheet_name='Sheet1', index=False)
    - **Example**:

```
df.to_excel('output.xlsx', sheet_name='Sheet1', index=False)
```

3. to_sql(): Writes data to a SQL database.
    - **Definition**: Writes DataFrame to a SQL database.
    - **Syntax**: df.to_sql(table_name, connection, if_exists='replace')
    - **Example**:

```
import pandas as pd
import sqlite3

conn = sqlite3.connect('database.db')
df.to_sql('table_name', conn, if_exists='replace')
```

# 3. Data Manipulation

## ➢ Aggregation Functions

**1. sum()**

- Computes the sum of all values in the selected column(s).

```
df['column_name'].sum()
```

**2. mean()**

- Computes the mean (average) of the selected column(s).

```
df['column_name'].mean()
```

**3. median()**

- Computes the median (middle value) of the selected column(s).

```
df['column_name'].median()
```

**4. min()**

- Computes the minimum value of the selected column(s).

```
df['column_name'].min()
```

**5. max()**

- Computes the maximum value of the selected column(s).

```
df['column_name'].max()
```

**6. std()**

- Computes the standard deviation of the selected column(s).

```
df['column_name'].std()
```

**7. var()**

- Computes the variance of the selected column(s).

```
df['column_name'].var()
```

**8. count()**

- Counts the number of non-null values in the selected column(s).

```
df['column_name'].count()
```

**9. nunique()**

- Counts the number of unique values in the selected column(s).

```
df['column_name'].nunique()
```

**10. first()**

- Returns the first value of the selected column(s).

```
df['column_name'].first()
```

**11. last()**

- Returns the last value of the selected column(s).

```
df['column_name'].last()
```

**12. agg()**

- Allows the application of multiple aggregation functions at once.

```
df['column_name'].agg([np.sum, np.mean, 'count'])
```

**GroupBy Aggregations**

You can combine aggregation functions with the groupby() method to perform aggregations on groups within your data:

```
# Group by a specific column and calculate aggregation functions
grouped = df.groupby('column_name').agg({
   'numeric_column': ['sum', 'mean', 'count'],
   'another_column': ['max', 'min']
})
```

**Example:**

```
import pandas as pd

data = {
   'Category': ['A', 'A', 'B', 'B', 'A', 'B'],
   'Value': [10, 20, 30, 40, 50, 60]
}

df = pd.DataFrame(data)

# Aggregating by 'Category'
result = df.groupby('Category').agg({
   'Value': ['sum', 'mean', 'max', 'min']
})
```

```
print(result)
```
Output:

```
       Value
         sum mean max min
Category
A         80   26.7  50  10
B        130   43.3  60  30
```

## ➢ Filtering Functions

**1. Boolean Indexing**
You can filter rows in a DataFrame based on a condition that results in True or False values.

```
df[df['column_name'] > value]
```
Example:

```
df[df['Age'] > 30]
```

**2. query() Method**
The .query() method allows you to filter a DataFrame using a query string. This is a more readable alternative to boolean indexing.

```
df.query('column_name > value')
```

Example:

```
df.query('Age > 30')
```
You can also use logical operators:

```
df.query('Age > 30 and Salary < 50000')
```

**3. loc[] Method**
The .loc[] method is used for label-based indexing, and it can be combined with boolean conditions to filter data.

```
df.loc[df['column_name'] > value]
```

Example:

df.loc[df['Age'] > 30]
You can filter multiple columns as well:

df.loc[(df['Age'] > 30) & (df['Salary'] < 50000)]

**4. iloc[] Method**
The .iloc[] method is used for integer-location based indexing. It can also be combined with boolean conditions if you want to filter by index position.

df.iloc[condition]

Example:

df.iloc[2:5]  # Rows from index position 2 to 4

**5. Using isin() Method**
The .isin() method is useful when you want to filter rows based on whether a column's value exists in a list or set of values.

df[df['column_name'].isin([value1, value2, value3])]
Example:

df[df['City'].isin(['New York', 'Los Angeles'])]

**6. Using notna() or isna()**

These methods are used to filter rows that are or are not missing (NaN) values.

- **notna()**: Returns True for non-missing values.
- **isna()**: Returns True for missing (NaN) values.

df[df['column_name'].notna()]
Example:

df[df['Age'].notna()]  # Filters rows where 'Age' is not NaN
To filter rows with missing values:

df[df['Age'].isna()]  # Filters rows where 'Age' is NaN

**7. Using between() Method**
The .between() method allows you to filter values within a specified range.

df[df['column_name'].between(min_value, max_value)]

df[df['Age'].between(30, 40)]

**8. str.contains() for String Matching**
If you're working with string columns and want to filter based on a substring, you can use the .str.contains() method.

df[df['column_name'].str.contains('substring')]
Example:

df[df['City'].str.contains('York')]  # Filters rows where 'City' contains 'York'

**9. Multiple Conditions with & (AND) and | (OR)**
You can combine multiple conditions using the & (AND) and | (OR) operators.

df[(df['column1'] > value1) & (df['column2'] < value2)]
Example:

df[(df['Age'] > 30) & (df['Salary'] < 50000)]
For OR condition:

df[(df['Age'] > 30) | (df['Salary'] < 50000)]

**10. filter() Method**
The .filter() method is mainly used to filter columns based on labels, like regex matching.

df.filter(items=['col1', 'col2'])  # Filter specific columns
For column names that match a regex pattern:

df.filter(regex='^A')  # Selects columns whose names start with 'A'

**Example of Filtering:**

```
import pandas as pd

data = {
   'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Edward'],
   'Age': [25, 35, 30, 45, 28],
   'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Chicago'],
   'Salary': [50000, 60000, 55000, 70000, 48000]
}

df = pd.DataFrame(data)

# Filtering rows where Age > 30
filtered_df = df[df['Age'] > 30]
print(filtered_df)
# Using .query() to filter
filtered_df_query = df.query('Age > 30 and Salary > 50000')
```

```
print(filtered_df_query)
```

Output:

```
   Name  Age        City  Salary
1   Bob   35  Los Angeles   60000
3  David   45     New York   70000

   Name  Age        City  Salary
1   Bob   35  Los Angeles   60000
3  David   45     New York   70000
```

## ➢ Transformation Functions

**1. apply() Method**

```
import pandas as pd

data = {
   'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Edward'],
   'Age': [25, 35, 30, 45, 28],
   'Salary': [50000, 60000, 55000, 70000, 48000]
}

df = pd.DataFrame(data)

# Apply function to 'Age' column
df['Age'] = df['Age'].apply(lambda x: x + 1)  # Add 1 to each age

print(df)
```

**Output**:

```
     Name  Age  Salary
0   Alice   26   50000
1     Bob   36   60000
2  Charlie   31   55000
3   David   46   70000
4  Edward   29   48000
```

---

**2. map() Method**

```
# Map function to 'City' column
df['City'] = df['City'].map({'New York': 'NYC', 'Los Angeles': 'LA', 'Chicago': 'CH'})

print(df)
```

**Output**:

```
     Name  Age  Salary      City
0   Alice   26   50000      NYC
1     Bob   36   60000       LA
2 Charlie   31   55000       CH
3   David   46   70000      NYC
4  Edward   29   48000       CH
```

---

**3. applymap() Method (for DataFrame)**

```
# Apply function to entire DataFrame
df[['Age', 'Salary']] = df[['Age', 'Salary']].applymap(lambda x: x * 1.1)  # Increase by 10%

print(df)
```

**Output**:

```
     Name   Age   Salary      City
0   Alice  28.6   55000.0      NYC
1     Bob  39.6   66000.0       LA
2 Charlie  34.1   60500.0       CH
3   David  50.6   77000.0      NYC
4  Edward  31.9   52800.0       CH
```

---

**4. transform() Method**

```
# Transform to subtract the mean of 'Salary' for each row
df['Salary'] = df['Salary'].transform(lambda x: x - x.mean())

print(df)
```

**Output**:

```
     Name   Age   Salary      City
0    Alice  28.6  -6750.0      NYC
1      Bob  39.6   750.0       LA
2  Charlie  34.1  -2500.0       CH
3    David  50.6  4500.0       NYC
4   Edward  31.9  -2700.0       CH
```

---

**5. fillna() Method**

```
# Fill NaN values in the 'Salary' column with the mean salary
df['Salary'] = df['Salary'].fillna(df['Salary'].mean())

print(df)
```

**Output**:

```
     Name   Age   Salary      City
0    Alice  28.6  -6750.0      NYC
1      Bob  39.6   750.0       LA
2  Charlie  34.1  -2500.0       CH
3    David  50.6   4500.0      NYC
4   Edward  31.9  -2700.0       CH
```

(Note: If there are no NaN values in Salary, this function would not modify the column.)

---

**6. replace() Method**

```
# Replace 'NYC' with 'New York' in the 'City' column
df['City'] = df['City'].replace({'NYC': 'New York'})

print(df)
```

**Output**:

sql

```
     Name   Age   Salary      City
0    Alice  28.6  -6750.0   New York
1      Bob  39.6   750.0       LA
2  Charlie  34.1  -2500.0       CH
3    David  50.6   4500.0   New York
4   Edward  31.9  -2700.0       CH
```

---

**7. cut() and qcut() Methods**

```
# Using cut() to categorize 'Age' into bins
df['Age_category'] = pd.cut(df['Age'], bins=[0, 30, 40, 50], labels=['Young', 'Middle-aged', 'Senior'])

print(df)
```

**Output**:
scss

```
     Name   Age   Salary      City     Age_category
0    Alice  28.6  -6750.0   New York          Young
1      Bob  39.6   750.0       LA   Middle-aged
2  Charlie  34.1  -2500.0       CH   Middle-aged
3    David  50.6   4500.0   New York          Senior
4   Edward  31.9  -2700.0       CH   Middle-aged
```

---

**8. rename() Method**

```
# Rename 'Age' column to 'Age_in_years'
df.rename(columns={'Age': 'Age_in_years'}, inplace=True)

print(df)
```

**Output**:
scss

```
     Name  Age_in_years   Salary      City     Age_category
0    Alice          28.6   -6750.0   New York          Young
1      Bob          39.6    750.0       LA   Middle-aged
2  Charlie          34.1   -2500.0       CH   Middle-aged
3    David          50.6    4500.0   New York          Senior
4   Edward          31.9   -2700.0       CH   Middle-aged
```

---

**9. set_index() and reset_index() Methods**

```
# Set 'Name' as index
df.set_index('Name', inplace=True)

# Reset index
df.reset_index(inplace=True)
print(df)
```

**Output**:
scss

```
     Name  Age_in_years   Salary      City     Age_category
0    Alice          28.6   -6750.0   New York          Young
1      Bob          39.6    750.0       LA   Middle-aged
2  Charlie          34.1   -2500.0       CH   Middle-aged
3    David          50.6    4500.0   New York          Senior
4   Edward          31.9   -2700.0       CH   Middle-aged
```

---

**10. sort_values() Method**

```
# Sort values by 'Age_in_years'
df.sort_values(by='Age_in_years', ascending=False, inplace=True)

print(df)
```

**Output**:
scss

```
     Name  Age_in_years   Salary      City     Age_category
3    David          50.6    4500.0   New York          Senior
1      Bob          39.6    750.0       LA   Middle-aged
2  Charlie          34.1   -2500.0       CH   Middle-aged
4   Edward          31.9   -2700.0       CH   Middle-aged
0    Alice          28.6   -6750.0   New York          Young
```

---

**11. pivot() and pivot_table() Methods**

```
# Using pivot_table to compute the mean salary by city
df_pivot_table = df.pivot_table(values='Salary', index='City', aggfunc='mean')

print(df_pivot_table)
```

**Output**:

```
              Salary
City
CH        -2600.000000
LA          750.000000
New York   -1125.000000
```

**12. astype() Method**

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

print(df)
```

**Output**:
scss

```
     Name  Age_in_years   Salary      City     Age_category
3   David          50.6    4500.0   New York          Senior
1     Bob          39.6     750.0        LA   Middle-aged
2 Charlie          34.1   -2500.0        CH   Middle-aged
4  Edward          31.9   -2700.0        CH   Middle-aged
0   Alice          28.6   -6750.0   New York          Young
```

# ➢ Sorting Functions

**1. sort_values() Method**
The .sort_values() method is used to sort a DataFrame or Series by one or more columns.

**Syntax:**

```
df.sort_values(by, axis=0, ascending=True, inplace=False, kind='quicksort',
na_position='last', ignore_index=False)
```

- **by**: The column or list of columns by which to sort.
- **axis**: 0 for sorting by rows (default), 1 for sorting by columns.
- **ascending**: Boolean or list of booleans. If True (default), sorts in ascending order. If False, sorts in descending order.
- **inplace**: If True, modifies the original DataFrame, otherwise returns a sorted copy.
- **kind**: The sorting algorithm ('quicksort', 'mergesort', 'heapsort').
- **na_position**: Whether to place NaN values at the beginning ('first') or at the end ('last').
- **ignore_index**: If True, resets the index.

**Example:**

```
import pandas as pd

data = {
   'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Edward'],
   'Age': [25, 35, 30, 45, 28],
   'Salary': [50000, 60000, 55000, 70000, 48000]
}

df = pd.DataFrame(data)

# Sorting by Age in ascending order
df_sorted_age = df.sort_values(by='Age')

# Sorting by Salary in descending order
df_sorted_salary_desc = df.sort_values(by='Salary', ascending=False)

print(df_sorted_age)
print("\n")
print(df_sorted_salary_desc)
```

**Output**:

```
     Name  Age  Salary
0   Alice   25   50000
4  Edward   28   48000
2 Charlie   30   55000
1     Bob   35   60000
3   David   45   70000

     Name  Age  Salary
3   David   45   70000
1     Bob   35   60000
2 Charlie   30   55000
0   Alice   25   50000
4  Edward   28   48000
```

**2. Sorting by Multiple Columns**

You can sort by multiple columns by passing a list of column names to the by parameter.

**Example:**

```
# Sorting by Age and then by Salary in descending order
df_sorted_multi = df.sort_values(by=['Age', 'Salary'], ascending=[True, False])

print(df_sorted_multi)
```

**Output**:

```
     Name  Age  Salary
0    Alice   25   50000
4   Edward   28   48000
2  Charlie   30   55000
1      Bob   35   60000
3    David   45   70000
```

**3. sort_index() Method**
The .sort_index() method is used to sort a DataFrame or Series by its index (row labels).

**Syntax:**

df.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', ignore_index=False)

- **axis**: 0 for sorting by row index (default), 1 for sorting by column index.
- **level**: Used if the DataFrame has multiple levels of indices (MultiIndex).
- **ascending**: Boolean indicating whether to sort in ascending or descending order.
- **inplace**: If True, sorts the DataFrame in place.
- **kind**: The sorting algorithm ('quicksort', 'mergesort', 'heapsort').
- **na_position**: Whether to place NaN values at the beginning ('first') or at the end ('last').

**Example:**

```
# Sorting by index in ascending order
df_sorted_index = df.sort_index(ascending=True)

print(df_sorted_index)
```

**Output**:

```
     Name  Age  Salary
0    Alice   25   50000
1      Bob   35   60000
2  Charlie   30   55000
3    David   45   70000
4   Edward   28   48000
```

---

**4. rank() Method**
The .rank() method is used to rank the elements in a Series or DataFrame. It assigns ranks to the data, with the lowest value getting rank 1. Ties are handled according to the method used (e.g., 'average', 'min', 'max', etc.).

**Syntax:**

df.rank(axis=0, method='average', ascending=True, na_option='keep', pct=False)

- **axis**: 0 for ranking rows (default), 1 for ranking columns.
- **method**: The ranking method ('average', 'min', 'max', 'first', 'dense').
- **ascending**: If True, ranks in ascending order; if False, ranks in descending order.
- **na_option**: How to rank NaN values ('keep', 'top', 'bottom').
- **pct**: If True, returns the rank as a percentage of the total number of elements.

**Example:**

```
# Ranking the 'Salary' column
df['Salary_rank'] = df['Salary'].rank(ascending=False)

print(df)
```

**Output**:

```
     Name  Age  Salary  Salary_rank
0   Alice   25   50000         4.0
1     Bob   35   60000         2.0
2 Charlie   30   55000         3.0
3   David   45   70000         1.0
4  Edward   28   48000         5.0
```

---

**5. nlargest() and nsmallest() Methods**

- **nlargest(n)**: Returns the top n largest values from a DataFrame or Series.
- **nsmallest(n)**: Returns the top n smallest values from a DataFrame or Series.

**Syntax:**

df.nlargest(n, columns, keep='first')
df.nsmallest(n, columns, keep='first')

- **n**: Number of items to return.
- **columns**: The column by which to sort the values.
- **keep**: 'first', 'last', or 'all' for handling duplicate values.

**Example (Top 3 Salaries):**

```
# Top 3 highest salaries
top_salaries = df.nlargest(3, 'Salary')

# Top 3 lowest salaries
bottom_salaries = df.nsmallest(3, 'Salary')
```

```
print("Top 3 Salaries:\n", top_salaries)
print("\nBottom 3 Salaries:\n", bottom_salaries)
```

**Output**:

```
Top 3 Salaries:
     Name  Age  Salary  Salary_rank
3    David   45   70000         1.0
1      Bob   35   60000         2.0
2  Charlie   30   55000         3.0

Bottom 3 Salaries:
     Name  Age  Salary  Salary_rank
4   Edward   28   48000         5.0
0    Alice   25   50000         4.0
```

---

**6. Sorting in Place (modifying original DataFrame)**
You can sort the DataFrame in place, modifying the original DataFrame, by setting the inplace parameter to True.

**Example:**

```
# Sorting by 'Age' in descending order and modifying the original DataFrame
df.sort_values(by='Age', ascending=False, inplace=True)

print(df)
```

**Output**:

```
     Name  Age  Salary  Salary_rank
3    David   45   70000         1.0
1      Bob   35   60000         2.0
2  Charlie   30   55000         3.0
4   Edward   28   48000         5.0
0    Alice   25   50000         4.0
```

---

**7. Sorting by Row/Column Index with sort_index()**
You can also sort by row or column indices using the sort_index() function.

**Example (sorting columns):**

```
# Sorting by column names
df_sorted_columns = df.sort_index(axis=1)
print(df_sorted_columns)
```

**Output**:

```
    Age     Name  Salary  Salary_rank
0   25.0    Alice   50000        4.0
1   35.0      Bob   60000        2.0
2   30.0  Charlie   55000        3.
```

# ➢ Merging and Joining Functions

**1. merge() Method**

The merge() function is used to combine two DataFrames based on one or more columns (or indices). It is similar to SQL joins (e.g., inner, outer, left, right).

**Syntax:**

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'))
```

- **left**: The first DataFrame.
- **right**: The second DataFrame.
- **how**: The type of join ('inner', 'outer', 'left', 'right').
- **on**: The column(s) to join on. Must be present in both DataFrames.
- **left_on**: The column(s) in the left DataFrame to join on.
- **right_on**: The column(s) in the right DataFrame to join on.
- **left_index** and **right_index**: If True, uses the index for joining instead of columns.
- **sort**: If True, sorts the result DataFrame by the join keys.
- **suffixes**: A tuple of suffixes to append to overlapping column names.

**Example:**

```
import pandas as pd

df1 = pd.DataFrame({
   'EmployeeID': [1, 2, 3],
   'Name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
   'EmployeeID': [1, 2, 4],
   'Salary': [50000, 60000, 70000]
})

# Merging DataFrames on 'EmployeeID' (inner join)
merged_df = pd.merge(df1, df2, on='EmployeeID', how='inner')
```

```
print(merged_df)
```

**Output**:

```
  EmployeeID     Name  Salary
0          1    Alice   50000
1          2      Bob   60000
```

---

**2. Join Types in merge()**

- **inner join**: Only returns rows that have matching keys in both DataFrames (default behavior).
- **outer join**: Returns all rows from both DataFrames, with NaN for missing values.
- **left join**: Returns all rows from the left DataFrame, and matching rows from the right DataFrame.
- **right join**: Returns all rows from the right DataFrame, and matching rows from the left DataFrame.

**Example (Outer Join):**

```
# Outer join (all rows from both DataFrames)
outer_merge = pd.merge(df1, df2, on='EmployeeID', how='outer')
print(outer_merge)
```

**Output**:

```
  EmployeeID     Name   Salary
0          1    Alice   50000.0
1          2      Bob   60000.0
2          3  Charlie       NaN
3          4      NaN   70000.0
```

---

**3. join() Method**

The .join() method is a simpler way to combine DataFrames by using indices. It's typically used for combining DataFrames based on their indices, although it can also join on columns.

**Syntax:**

```
df1.join(df2, on=None, how='left', lsuffix='', rsuffix='', sort=False)
```

- **df2**: The DataFrame to join with.
- **on**: Column or index level name(s) to join on. By default, the index is used.
- **how**: The type of join ('left', 'right', 'outer', 'inner').
- **lsuffix** and **rsuffix**: Suffixes to add to duplicate column names.

**Example (Join by Index):**

```
df1 = pd.DataFrame({
   'Name': ['Alice', 'Bob', 'Charlie'],
   'Age': [25, 35, 30]
}, index=[1, 2, 3])

df2 = pd.DataFrame({
   'Salary': [50000, 60000, 55000]
}, index=[1, 2, 4])

# Join using indices
joined_df = df1.join(df2, how='inner')
print(joined_df)
```

**Output**:

```
     Name  Age  Salary
1   Alice   25   50000
2     Bob   35   60000
```

---

**4. concat() Method**
The concat() function is used to concatenate two or more DataFrames along rows or columns.

**Syntax:**

```
pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None, levels=None,
names=None, verify_integrity=False, sort=False)
```

- **objs**: A sequence or mapping of DataFrames to concatenate.
- **axis**: 0 for row-wise concatenation, 1 for column-wise concatenation.
- **join**: 'outer' (default) for union of columns/rows, 'inner' for intersection.
- **ignore_index**: If True, the index will be reset.
- **keys**: Grouping labels for the resulting DataFrame.

**Example (Concatenate along Rows):**

```
df1 = pd.DataFrame({
   'Name': ['Alice', 'Bob'],
   'Age': [25, 35]
})
```

```
df2 = pd.DataFrame({
   'Name': ['Charlie', 'David'],
   'Age': [30, 45]
})

# Concatenate along rows (default is axis=0)
concatenated_df = pd.concat([df1, df2])
print(concatenated_df)
```

**Output**:

```
     Name  Age
0    Alice   25
1      Bob   35
0  Charlie   30
1    David   45
```

**Example (Concatenate along Columns):**

```
# Concatenate along columns (axis=1)
concatenated_columns = pd.concat([df1, df2], axis=1)
print(concatenated_columns)
```

**Output**:

```
     Name  Age     Name  Age
0    Alice   25  Charlie   30
1      Bob   35    David   45
```

---

**5. append() Method**

The .append() method is used to append rows of one DataFrame to another. It's similar to concat() with axis=0 and ignore_index=True.

**Syntax:**

```
df.append(other, ignore_index=False, verify_integrity=False, sort=False)
```

- **other**: The DataFrame or Series to append.
- **ignore_index**: If True, reindexes the resulting DataFrame.
- **verify_integrity**: If True, checks for duplicates.
- **sort**: If True, sorts columns.

**Example:**

```
# Append df2 to df1
appended_df = df1.append(df2, ignore_index=True)
print(appended_df)
```

**Output**:

```
     Name  Age
0   Alice   25
1     Bob   35
2 Charlie   30
3   David   45
```

---

**6. merge_asof() Method**

The merge_asof() function performs an as-of merge. It is typically used when you want to merge two DataFrames based on the closest match to a particular column.

**Syntax:**

```
pd.merge_asof(left, right, on, by=None, tolerance=None, direction='backward')
```

- **on**: The column to join on.
- **by**: Optional columns to group by.
- **tolerance**: Maximum allowed distance between matched values.
- **direction**: 'forward', 'backward', or 'nearest'.

**Example:**

```
df1 = pd.DataFrame({
   'Date': pd.to_datetime(['2022-01-01', '2022-01-03', '2022-01-05']),
   'Value1': [1, 2, 3]
})

df2 = pd.DataFrame({
   'Date': pd.to_datetime(['2022-01-02', '2022-01-04']),
   'Value2': [10, 20]
})

# Merge asof with 'Date' column
asof_merged_df = pd.merge_asof(df1, df2, on='Date')
print(asof_merged_df)
```

**Output**:

```
       Date  Value1  Value2
0 2022-01-01       1      10
1 2022-01-03       2      20
2 2022-01-05       3      20
```

**Conclusion**

- **merge()**: Great for SQL-like joins on columns or indices.
- **join()**: Easier way to join DataFrames by indices, but also allows joining on columns.
- **concat()**: Best for concatenating DataFrames along rows or columns.
- **append()**: Adds rows from one DataFrame to another.
- **merge_asof()**: Performs merges

# 4. Data Cleaning

## ➢ Handling Missing Data Functions

**1. Detecting Missing Data**
Pandas uses NaN (Not a Number) to represent missing values. You can use the following functions to detect missing data:

**isna() / isnull()**
These functions return a DataFrame or Series of boolean values, where True indicates missing values (NaN), and False indicates non-missing values.

```
df.isna()  # or df.isnull()
```

**notna() / notnull()**
These functions return the inverse of isna()/isnull(), where True indicates non-missing values.

```
df.notna()  # or df.notnull()
```

**Example:**

```
import pandas as pd
import numpy as np

df = pd.DataFrame({
   'A': [1, 2, np.nan, 4],
   'B': [np.nan, 2, 3, 4]
})

print(df.isna())  # Detect missing values
```

**Output**:

```
       A      B
0  False   True
1  False  False
2   True  False
3  False  False
```

---

**2. Filling Missing Data**
Pandas provides several methods to fill missing data:

**fillna()**
The .fillna() method is used to fill missing values with a specified value, method, or interpolation.

**Syntax:**

df.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None)

- **value**: The value to fill NaN values with (can be scalar, dict, Series, or DataFrame).
- **method**: The method to use for filling ('ffill' for forward fill, 'bfill' for backward fill).
- **axis**: The axis to fill (0 for rows, 1 for columns).
- **inplace**: If True, modifies the original DataFrame.
- **limit**: Maximum number of replacements.
- **downcast**: If True, downcast numeric types.

**Example (Filling with a Constant Value):**

```
# Filling missing values with 0
df_filled = df.fillna(0)
print(df_filled)
```

**Output**:

```
     A  B
0  1.0  0.0
1  2.0  2.0
2  0.0  3.0
3  4.0  4.0
```

**Example (Forward Fill):**

```
# Forward fill to propagate the previous value
df_ffill = df.fillna(method='ffill')
```

```
print(df_ffill)
```
**Output**:

```
   A    B
0  1.0  NaN
1  2.0  2.0
2  2.0  3.0
3  4.0  4.0
```

**Example (Backward Fill):**

```
# Backward fill to propagate the next value
df_bfill = df.fillna(method='bfill')
print(df_bfill)
```

**Output**:

```
   A    B
0  1.0  2.0
1  2.0  2.0
2  4.0  3.0
3  4.0  4.0
```

---

**3. Dropping Missing Data**

**dropna()**
The .dropna() method removes missing values from a DataFrame or Series. You can specify whether to drop rows or columns with missing values.

**Syntax:**

```
df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)
```

- **axis**: 0 to drop rows, 1 to drop columns.
- **how**: 'any' (drop if any NaN value is present) or 'all' (drop if all values are NaN).
- **thresh**: Require a minimum number of non-null values in a row/column.
- **subset**: Specifies columns or index levels to check for missing values.
- **inplace**: If True, modifies the original DataFrame.

**Example (Dropping Rows with Any Missing Value):**

```
# Drop rows with any missing value
df_dropped = df.dropna(axis=0, how='any')
print(df_dropped)
```

**Output**:

```
    A    B
1  2.0  2.0
3  4.0  4.0
```

**Example (Dropping Columns with Any Missing Value):**

```
# Drop columns with any missing value
df_dropped_columns = df.dropna(axis=1, how='any')
print(df_dropped_columns)
```

**Output**:

```
    A
0  1.0
1  2.0
2  NaN
3  4.0
```

---

**4. Replacing Missing Data**

**replace()**
The .replace() method allows you to replace NaN values with specified values or other replacements in the DataFrame.

**Syntax:**
df.replace(to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad')

- **to_replace**: The value to be replaced (NaN, or any value to be replaced).
- **value**: The replacement value.

**Example (Replacing NaN with a Specific Value):**

```
# Replacing NaN with a specific value (e.g., -1)
df_replaced = df.replace(np.nan, -1)
print(df_replaced)
```

**Output**:

```
    A  B
0  1.0 -1
1  2.0  2
2 -1.0  3
3  4.0  4
```

---

**5. Interpolating Missing Data**

**interpolate()**
The .interpolate() method performs linear interpolation on the missing data, filling gaps with estimated values based on neighboring data points.

**Syntax:**

df.interpolate(method='linear', axis=0, limit=None, inplace=False, limit_direction='forward', limit_area=None)

- **method**: The interpolation method ('linear', 'polynomial', 'spline', etc.).
- **axis**: 0 for rows, 1 for columns.
- **limit**: Maximum number of missing values to fill.
- **inplace**: If True, modifies the original DataFrame.

**Example (Linear Interpolation):**

```
# Interpolate missing values linearly
df_interpolated = df.interpolate(method='linear')
print(df_interpolated)
```

**Output**:

```
    A    B
0  1.0  NaN
1  2.0  2.0
2  3.0  3.0
3  4.0  4.0
```

---

**6. Filling Missing Data Based on a Condition**

**apply() with a Custom Function**
You can use the .apply() method along with a custom function to fill missing values based on specific conditions.

**Example:**

```
# Fill missing values in column 'A' with the mean of the column
df['A'] = df['A'].apply(lambda x: df['A'].mean() if pd.isna(x) else x)
print(df)
```

**Output**:

```
    A    B
0  1.0  NaN
1  2.0  2.0
2  2.33 3.0
3  4.0  4.0
```

---

**7. Checking for Missing Data**

**any() and all()**

You can check whether there are any missing values in a DataFrame or Series with .any() or .all().

- **.any()**: Returns True if any missing value is found.
- **.all()**: Returns True if all values are missing.

**Example**

```
# Check if any missing values exist in each column
print(df.isna().any())

# Check if all values are missing in each column
print(df.isna().all())
```

**Output**:

```
A     True
B     True
dtype: bool

A     False
B     False
dtype: bool
```

---

**Summary of Common Functions for Handling Missing Data:**

- **Detect Missing Data**: .isna(), .isnull(), .notna(), .notnull()
- **Fill Missing Data**: .fillna()
- **Drop Missing Data**: .dropna()
- **Replace Missing Data**: .replace()
- **Interpolate Missing Data**: .interpolate()
- **Check Missing Data**: .any(), .all()

# ➢ String Manipulation Functions

**1. str.lower() / str.upper() / str.title()**
These methods allow you to change the case of strings.

- **str.lower()**: Converts all characters in a string to lowercase.
- **str.upper()**: Converts all characters in a string to uppercase.
- **str.title()**: Converts the first character of each word to uppercase.

**Example:**

```
import pandas as pd

df = pd.DataFrame({
   'Name': ['Alice', 'bob', 'CHARLIE']
})

df['Name_lower'] = df['Name'].str.lower()
df['Name_upper'] = df['Name'].str.upper()
df['Name_title'] = df['Name'].str.title()

print(df)
```

**Output**:

```
     Name Name_lower Name_upper Name_title
0   Alice      alice      ALICE      Alice
1     bob        bob        BOB        Bob
2 CHARLIE    charlie    CHARLIE    Charlie
```

---

**2. str.strip() / str.lstrip() / str.rstrip()**
These methods remove leading and trailing characters (by default, whitespace).

- **str.strip()**: Removes leading and trailing spaces or specified characters.
- **str.lstrip()**: Removes leading (left) spaces or specified characters.
- **str.rstrip()**: Removes trailing (right) spaces or specified characters.

**Example:**

```
df = pd.DataFrame({
   'Name': ['  Alice  ', '  bob  ', 'CHARLIE ']
})

df['Name_stripped'] = df['Name'].str.strip()
df['Name_lstrip'] = df['Name'].str.lstrip()
df['Name_rstrip'] = df['Name'].str.rstrip()
print(df)
```

**Output**:

```
      Name Name_stripped Name_lstrip Name_rstrip
0   Alice     Alice        Alice        Alice
1     bob       bob          bob          bob
2  CHARLIE     CHARLIE      CHARLIE      CHARLIE
```

---

**3. str.replace()**
The str.replace() method is used to replace occurrences of a substring within each string with a new substring.

**Syntax:**

```
df['column_name'].str.replace(old, new, regex=False)
```

- **old**: The substring or regular expression to replace.
- **new**: The string to replace old with.
- **regex**: If True, treats old as a regular expression.

**Example:**

```
df = pd.DataFrame({
   'Name': ['Alice', 'Bob', 'Charlie']
})

df['Name_replaced'] = df['Name'].str.replace('o', '0')
print(df)
```

**Output**:

```
     Name Name_replaced
0   Alice        Alice
1     Bob          B0b
2  Charlie     Charlie
```

**Example with Regular Expression:**

```
df['Name_replaced_regex'] = df['Name'].str.replace(r'[aeiou]', 'X', regex=True)
print(df)
```

**Output**:

```
     Name Name_replaced_regex
0   Alice           XlXcX
1     Bob             BXb
2  Charlie        ChXrlXX
```

---

**4. str.contains()**
The str.contains() function checks whether a pattern or substring is present in each string of a Series.

**Syntax:**

df['column_name'].str.contains(pattern, case=False, regex=True)

- **pattern**: The substring or regular expression to look for.
- **case**: If True, performs case-sensitive matching.
- **regex**: If True, the pattern is treated as a regular expression.

**Example:**

```
df = pd.DataFrame({
   'Name': ['Alice', 'Bob', 'Charlie']
})

df['Contains_A'] = df['Name'].str.contains('A', case=False)
print(df)
```

**Output**:

```
     Name  Contains_A
0   Alice       True
1     Bob      False
2  Charlie       True
```

---

**5. str.startswith() / str.endswith()**
These methods check if a string starts or ends with a particular substring.

**Example:**

```
df = pd.DataFrame({
   'Name': ['Alice', 'Bob', 'Charlie']
})

df['Starts_with_A'] = df['Name'].str.startswith('A')
df['Ends_with_e'] = df['Name'].str.endswith('e')

print(df)
```

**Output**:

```
     Name  Starts_with_A  Ends_with_e
0   Alice          True         True
1     Bob         False        False
2  Charlie         False         True
```

---

**6. str.find() / str.index()**
Both str.find() and str.index() are used to find the index of a substring within a string.

- **str.find()**: Returns the lowest index of the substring or -1 if the substring is not found.
- **str.index()**: Similar to find(), but raises a ValueError if the substring is not found.

**Example:**

```
df = pd.DataFrame({
   'Name': ['Alice', 'Bob', 'Charlie']
})

df['Find_A'] = df['Name'].str.find('A')
df['Index_C'] = df['Name'].str.index('C')


print(df)
```

**Output**:

```
     Name  Find_A  Index_C
0   Alice       0        2
1     Bob      -1        2
2  Charlie       2        2
```

---

**7. str.split()**
The str.split() method splits each string in the Series at a specified delimiter and returns a list of strings.

**Syntax:**

```
df['column_name'].str.split(pat=None, n=-1, expand=False)
```

- **pat**: The delimiter (string or regex) to split the string.
- **n**: The maximum number of splits.
- **expand**: If True, splits into separate columns.

**Example:**

```
df = pd.DataFrame({
   'Name': ['Alice Johnson', 'Bob Smith', 'Charlie Brown']
})

df['Name_split'] = df['Name'].str.split()
df[['First_Name', 'Last_Name']] = df['Name'].str.split(expand=True)

print(df)
```

**Output**:

```
           Name          Name_split First_Name Last_Name
0   Alice Johnson     [Alice, Johnson]      Alice   Johnson
1       Bob Smith       [Bob, Smith]        Bob     Smith
2  Charlie Brown   [Charlie, Brown]    Charlie    Brown
```

---

**8. str.len()**

The str.len() function is used to get the length of each string in a Series.

**Example:**

```
df = pd.DataFrame({
   'Name': ['Alice', 'Bob', 'Charlie']
})

df['Name_length'] = df['Name'].str.len()
print(df)
```

**Output**:

```
     Name  Name_length
0   Alice           5
1     Bob           3
2  Charlie          7
```

---

**9. str.replace() with Regex**

You can use regular expressions to find and replace more complex patterns in your strings.

**Example:**

```
df = pd.DataFrame({
   'Text': ['The cat is on the mat', 'The dog is on the rug']
})
```

```
# Replace "cat" and "dog" with "animal"
df['Text_replaced'] = df['Text'].str.replace(r'\b(cat|dog)\b', 'animal', regex=True)
print(df)
```

**Output**:

```
                Text           Text_replaced
0  The cat is on the mat  The animal is on the mat
1  The dog is on the rug  The animal is on the rug
```

---

**10. str.extract()**
The str.extract() method extracts a pattern or a group of patterns from each string using a regular expression.

**Syntax:**

```
df['column_name'].str.extract(pattern, expand=True)
```

- **pattern**: The regular expression pattern to extract.
- **expand**: If True, returns a DataFrame with one column per capture group.

**Example:**

```
df = pd.DataFrame({
   'Text': ['2021-01-01', '2022-02-02', '2023-03-03']
})

# Extract the year
df['Year'] = df['Text'].str.extract(r'(\d{4})')
print(df)
```

**Output**:

```
         Text  Year
0   2021-01-01  2021
1   2022-02-02  2022
2   2023-03-03  2023
```

**Summary of Common String Manipulation Functions:**

- **Change Case**: .str.lower(), .str.upper(), .str.title()
- **Whitespace Handling**: .str.strip(), .str.lstrip(), `.str.rstrip

# 5. Time Series Analysis

## ➢ Date and Time Manipulation Functions

**1. to_datetime()**
The pd.to_datetime() function is used to convert a column or a series to datetime objects.

**Syntax:**

pd.to_datetime(arg, format=None, errors='raise', dayfirst=False)

- **arg**: The data to convert (e.g., a string, list, or Series).
- **format**: The format of the date/time string (optional).
- **errors**: If 'raise' (default), raises an error on invalid parsing. If 'coerce', invalid parsing will be set as NaT.
- **dayfirst**: If True, it treats the first element as the day (useful for European date formats).

**Example:**

```
import pandas as pd

# Convert a string to datetime
df = pd.DataFrame({
   'Date': ['2021-01-01', '2022-02-02', '2023-03-03']
})

df['Date'] = pd.to_datetime(df['Date'])
print(df)
```

**Output**:

```
      Date
0 2021-01-01
1 2022-02-02
2 2023-03-03
```

---

**2. datetime.now()**
This function returns the current date and time.

**Example:**

```
current_time = pd.to_datetime("now")
print(current_time)
```

**Output**:
2025-01-20 12:34:56.789123

---

**3. dt Accessor**
The .dt accessor allows you to extract different components from a datetime column such as year, month, day, hour, minute, second, weekday, and more.

**Commonly used attributes:**

- **dt.year**: Extracts the year.
- **dt.month**: Extracts the month.
- **dt.day**: Extracts the day.
- **dt.hour**: Extracts the hour.
- **dt.minute**: Extracts the minute.
- **dt.second**: Extracts the second.
- **dt.weekday()**: Returns the day of the week (Monday=0, Sunday=6).

**Example:**

```
df = pd.DataFrame({
   'Date': pd.to_datetime(['2021-01-01', '2022-02-02', '2023-03-03'])
})

df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
df['Day'] = df['Date'].dt.day
df['Weekday'] = df['Date'].dt.weekday

print(df)
```

**Output**:

```
      Date  Year  Month  Day  Weekday
0 2021-01-01  2021      1    1        4
1 2022-02-02  2022      2    2        2
2 2023-03-03  2023      3    3        4
```

---

**4. date_range()**
The pd.date_range() function is used to generate a range of dates.

**Syntax:**

pd.date_range(start=None, end=None, periods=None, freq='D', tz=None, normalize=False)

- **start**: The start date.
- **end**: The end date.
- **periods**: Number of periods to generate.
- **freq**: Frequency of the generated dates (e.g., 'D' for daily, 'M' for monthly).
- **tz**: Timezone.
- **normalize**: If True, it normalizes the start date to midnight.

**Example:**

```
date_range = pd.date_range(start='2025-01-01', end='2025-01-10')
print(date_range)
```

**Output**:

```
DatetimeIndex(['2025-01-01', '2025-01-02', '2025-01-03', '2025-01-04',
               '2025-01-05', '2025-01-06', '2025-01-07', '2025-01-08',
               '2025-01-09', '2025-01-10'],
              dtype='datetime64[ns]', freq='D')
```

---

**5. timedelta()**
You can use pd.Timedelta() to create timedeltas (differences between dates or times).

**Example:**

```
delta = pd.Timedelta(days=5, hours=3)
print(delta)
```

**Output**:
5 days 03:00:00

**Example: Adding Timedelta to a Date:**

```
df = pd.DataFrame({
    'Date': pd.to_datetime(['2021-01-01', '2022-02-02', '2023-03-03'])
})

df['New_Date'] = df['Date'] + pd.Timedelta(days=10)
print(df)
```

**Output**:

```
      Date   New_Date
0 2021-01-01 2021-01-11
1 2022-02-02 2022-02-12
2 2023-03-03 2023-03-13
```

---

**6. pd.DatetimeIndex**

A DatetimeIndex is a specialized index for datetime objects. You can create one from a list of datetime values or a date_range.

**Example:**

```
date_index = pd.DatetimeIndex(['2025-01-01', '2025-02-01', '2025-03-01'])
print(date_index)
```

**Output**:

```
DatetimeIndex(['2025-01-01', '2025-02-01', '2025-03-01'], dtype='datetime64[ns]',
freq=None)
```

---

**7. strftime() / strptime()**

- **strftime()**: Converts a datetime object to a string based on a specific format.
- **strptime()**: Converts a string to a datetime object based on a given format.

**Example:**

```
# Convert datetime to string using strftime
df['Date_str'] = df['Date'].dt.strftime('%Y-%m-%d')
print(df)
```

**Output**:

```
      Date    Date_str
0 2021-01-01  2021-01-01
1 2022-02-02  2022-02-02
2 2023-03-03  2023-03-03
```

---

**8. bdate_range()**
The bdate_range() function generates a range of business days.

**Syntax:**

pd.bdate_range(start, end, freq='B', holidays=None, weekmask=None, weekdays=None)

- **start**: The start date.
- **end**: The end date.
- **freq**: The frequency (default is 'B' for business days).
- **holidays**: A list of holiday dates to exclude.
- **weekmask**: A string that defines which weekdays to include (e.g., 'Mon Tue Wed').

**Example:**

```
business_days = pd.bdate_range(start='2025-01-01', end='2025-01-10')
print(business_days)
```

**Output**:

```
DatetimeIndex(['2025-01-01', '2025-01-02', '2025-01-05', '2025-01-06',  '2025-01-
07', '2025-01-08', '2025-01-09', '2025-01-10'],
          dtype='datetime64[ns]', freq='B')
```

---

**9. period_range()**
The period_range() function generates a range of periods (e.g., months, years, etc.).

**Syntax:**

pd.period_range(start, end, freq='M')

- **start**: The start date.
- **end**: The end date.
- **freq**: The frequency (e.g., 'M' for monthly, 'A' for yearly).

**Example:**

```
periods = pd.period_range(start='2025-01-01', end='2025-12-31', freq='M')
print(periods)
```

**Output**:

```
PeriodIndex(['2025-01', '2025-02', '2025-03', '2025-04', '2025-05', '2025-06',
          '2025-07', '2025-08', '2025-09', '2025-10', '2025-11', '2025-12'],
         dtype='period[M]', freq='M')
```

---

**10. timestamp()**
You can use pd.Timestamp() to create a single timestamp.

**Example:**

```
timestamp = pd.Timestamp('2025-01-01 12:00:00')
print(timestamp)
```

**Output**:

```
2025-01-01 12:00:00
```

---

**Summary of Common Date/Time Functions:**

- **Conversion**: pd.to_datetime(), pd.to_timedelta(), pd.Timestamp()
- **Datetime extraction**: .dt.year, .dt.month, .dt.day, .dt.hour
- **Datetime creation**: pd.date_range(), pd.bdate_range(), pd.period_range()
- **Formatting**: .strftime(), .strptime()
- **Business days**: pd.bdate_range()
- **Time delta**: pd.Timedelta()

# ➢ Reshaping and Pivoting Functions

**1. pivot()**
The pivot() function reshapes data where values in a column become new columns in the DataFrame. It is useful when you want to reorganize your data into a more convenient format.

**Syntax:**

df.pivot(index=None, columns=None, values=None)

- **index**: Column(s) to set as the new index (rows).
- **columns**: Column(s) to set as the new columns.
- **values**: Column(s) to use for populating the new table.

**Example:**

```
import pandas as pd

df = pd.DataFrame({
   'Date': ['2021-01-01', '2021-01-01', '2021-01-02', '2021-01-02'],
   'City': ['New York', 'Los Angeles', 'New York', 'Los Angeles'],
   'Temperature': [32, 75, 30, 74]
})

pivot_df = df.pivot(index='Date', columns='City', values='Temperature')
print(pivot_df)
```

**Output**:

| City | Los Angeles | New York |
|---|---|---|
| Date | | |
| 2021-01-01 | 75 | 32 |
| 2021-01-02 | 74 | 30 |

In this example, the Date column becomes the index, the City column becomes the new columns, and the Temperature values fill the table.

---

**2. pivot_table()**
The pivot_table() function is similar to pivot(), but it allows for more advanced features such as aggregating data. It is useful when there are multiple rows for each combination of index and column.

**Syntax:**

df.pivot_table(index=None, columns=None, values=None, aggfunc='mean')

- **index**: The column(s) to set as the new index.
- **columns**: The column(s) to set as the new columns.
- **values**: The column(s) to aggregate.
- **aggfunc**: The aggregation function to apply (e.g., mean, sum, count). The default is 'mean'.

**Example:**

```
df = pd.DataFrame({
   'Date': ['2021-01-01', '2021-01-01', '2021-01-02', '2021-01-02'],
   'City': ['New York', 'Los Angeles', 'New York', 'Los Angeles'],
   'Temperature': [32, 75, 30, 74],
   'Humidity': [80, 10, 78, 12]
})

pivot_table_df = df.pivot_table(index='Date', columns='City', values=['Temperature', 'Humidity'], aggfunc='mean')
print(pivot_table_df)
```

**Output**:

| | Temperature | | Humidity | |
|---|---|---|---|---|
| City | Los Angeles | New York | Los Angeles | New York |
| Date | | | | |
| 2021-01-01 | 75.0 | 32.0 | 10.0 | 80.0 |
| 2021-01-02 | 74.0 | 30.0 | 12.0 | 78.0 |

In this example, the pivot_table() aggregates the temperature and humidity data for each city by the date, using the mean aggregation function.

---

**3. melt()**
The melt() function is the reverse of pivot(). It unpivots or reshapes a DataFrame from wide format to long format. This is useful when you need to normalize your data by converting it into a tidy format.

**Syntax:**

df.melt(id_vars=None, value_vars=None, var_name=None, value_name='value')

- **id_vars**: Columns that will remain as identifier variables.
- **value_vars**: Columns to unpivot.
- **var_name**: The name of the variable column (optional).
- **value_name**: The name of the values column.

**Example:**

```
df = pd.DataFrame({
   'Date': ['2021-01-01', '2021-01-02'],
   'New York': [32, 30],
   'Los Angeles': [75, 74]
})

melted_df = df.melt(id_vars='Date', value_vars=['New York', 'Los Angeles'],
var_name='City', value_name='Temperature')
print(melted_df)
```

**Output**:

```
        Date         City  Temperature
0  2021-01-01     New York          32
1  2021-01-02     New York          30
2  2021-01-01  Los Angeles          75
3  2021-01-02  Los Angeles          74
```

Here, Date remains as an identifier, and the New York and Los Angeles columns are melted into a single column for cities, with their respective temperatures as values.

---

**4. stack()**
The stack() function compresses a level in the DataFrame's columns into rows. It is used to stack columns into a single column, which is useful for hierarchical indexing (multi-indexing).

**Syntax:**

df.stack(level=-1, dropna=True)

- **level**: The level of columns to stack.
- **dropna**: If True, removes missing values.

**Example:**

```
df = pd.DataFrame({
   'Date': ['2021-01-01', '2021-01-02'],
   'New York': [32, 30],
   'Los Angeles': [75, 74]
}).set_index('Date')

stacked_df = df.stack()
print(stacked_df)
```

**Output**:

```
Date
2021-01-01  New York       32
            Los Angeles    75
2021-01-02  New York       30
            Los Angeles    74

dtype: int64
```

Here, the stack() function combines the New York and Los Angeles columns into a single column with a hierarchical index.

---

**5. unstack()**

The unstack() function is the inverse of stack(). It pivots the level of a MultiIndex column (stacked format) into the columns.

**Syntax:**

```
df.unstack(level=-1, fill_value=None)
```

- **level**: The level to unstack (can specify a column).
- **fill_value**: Value to fill for missing values.

**Example:**

```
stacked_df = pd.DataFrame({
   'Date': ['2021-01-01', '2021-01-02'],
   'New York': [32, 30],
   'Los Angeles': [75, 74]
}).set_index('Date').stack()

unstacked_df = stacked_df.unstack()
print(unstacked_df)
```

**Output**:

City New York Los Angeles
Date
2021-01-01 32.0 75.0
2021-01-02 30.0 74.0
Here, the unstack() function converts the multi-index series back to a DataFrame with columns for each city.

---

**6. crosstab()**
The crosstab() function is used to create a cross-tabulation (contingency table) of two or more factors. It is similar to a pivot table but produces a frequency table.

**Syntax:**

pd.crosstab(index, columns, values=None, aggfunc=None, margins=False, margins_name='All')

- **index**: Values for rows.
- **columns**: Values for columns.
- **values**: Values to aggregate.
- **aggfunc**: Aggregation function (e.g., sum, count).
- **margins**: If True, adds a row/column with totals.

**Example:**

```
df = pd.DataFrame({
   'City': ['New York', 'Los Angeles', 'New York', 'Los Angeles', 'New York'],
   'Weather': ['Sunny', 'Cloudy', 'Sunny', 'Cloudy', 'Rainy']
})

crosstab_df = pd.crosstab(index=df['City'], columns=df['Weather'])
print(crosstab_df)
```

**Output**:

Weather Cloudy Rainy Sunny
City
Los Angeles 1 0 1
New York 0 1 2
This creates a frequency table showing how many times each weather type occurs in each city.

---

**7. wide_to_long()**

The wide_to_long() function is useful when you want to convert data from a wide format to a long format, particularly when you have multiple columns that share the same base name with different suffixes.

**Syntax:**

```
pd.wide_to_long(df, stubnames, i, j, sep='_', suffix='\d+')
```

- **stubnames**: The prefix of the column names that share a common base name.
- **i**: The identifier column.
- **j**: The column where the suffix is stored.
- **sep**: The separator between the base name and the suffix.
- **suffix**: The regular expression to match the suffixes.

**Example:**

```
df = pd.DataFrame({
   'id': [1, 2],
   'item_1': ['A', 'B'],
   'item_2': ['C', 'D'],
   'item_3': ['E', 'F']
})

long_df = pd.wide_to_long(df, stubnames='item', i='id', j='time')
print(long_df)
```

**Output**:

```
       item
id time
1  1    A
   2    C
   3    E
2  1    B
   2    D
   3    F
```

In this example, columns item_1, item_2, and item_3 are converted to a single item column with a new time-based index.

---

**Summary of Reshaping and Pivoting Functions:**

- **pivot()**: Reshapes data by converting unique values into columns.
- **pivot_table()**: Creates a pivot table with aggregation options.
- **melt()**: Converts wide format data to long format.
- **stack()**: Converts columns into rows for hierarchical indexing.
- **unstack()**: Converts rows back into columns.
- **crosstab()**: Generates a contingency table (cross-tabulation).
- **wide_to_long()**: Converts wide data into a long format.

# ➢ Joining/Concatenation Functions

**1. concat()**
The concat() function is used to concatenate multiple pandas objects (such as DataFrames or Series) along a particular axis (either rows or columns). This is often used to stack DataFrames vertically (row-wise) or horizontally (column-wise).

**Syntax:**

pd.concat(objs, axis=0, join='outer', ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, sort=False)

- **objs**: A sequence (list or tuple) of pandas objects (DataFrames or Series) to concatenate.
- **axis**: The axis to concatenate along. 0 for rows (default) and 1 for columns.
- **join**: How to handle indexes. 'outer' (default) takes the union of indexes, while 'inner' takes the intersection.
- **ignore_index**: If True, the resulting index will be reset to a default integer index.
- **keys**: Adds hierarchical indexing (multi-index).
- **sort**: If True, sorts the columns (useful if they have different column names).

**Example:**

```
import pandas as pd

df1 = pd.DataFrame({
   'A': [1, 2],
   'B': [3, 4]
})

df2 = pd.DataFrame({
   'A': [5, 6],
   'B': [7, 8]
})

concatenated_df = pd.concat([df1, df2], axis=0, ignore_index=True)
print(concatenated_df)
```

**Output**:

```
   A  B
0  1  3
1  2  4
2  5  7
3  6  8
```

In this example, two DataFrames df1 and df2 are stacked vertically (along rows), and the index is reset because ignore_index=True.

---

**2. merge()**
The merge() function is used to combine DataFrames by aligning them based on common columns or indexes. This is similar to SQL joins (e.g., inner, outer, left, right join).

**Syntax:**

pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False)

- **left**: The left DataFrame.
- **right**: The right DataFrame.
- **how**: Specifies the type of join: 'left', 'right', 'outer', or 'inner'. Default is 'inner'.
- **on**: Column(s) to join on (if the columns have the same name in both DataFrames).
- **left_on** and **right_on**: Columns in the left and right DataFrames to join on if they differ.
- **left_index** and **right_index**: If True, use the index for merging.

**Example (Inner Join):**

```
df1 = pd.DataFrame({
   'ID': [1, 2, 3],
   'Name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
   'ID': [1, 2, 4],
   'Score': [85, 92, 78]
})

merged_df = pd.merge(df1, df2, on='ID', how='inner')
print(merged_df)
```

**Output**:

```
   ID     Name  Score
0   1    Alice     85
1   2      Bob     92
```

In this example, an inner join is performed on the ID column, so only rows with matching ID values are returned.

**Example (Left Join):**

```
merged_df_left = pd.merge(df1, df2, on='ID', how='left')
print(merged_df_left)
```

**Output**:

```
   ID     Name  Score
0   1    Alice   85.0
1   2      Bob   92.0
2   3  Charlie    NaN
```

In this case, a left join is performed, so all rows from df1 are returned, with missing values (NaN) for columns from df2 where there is no match.

---

**3. join()**

The join() function is used to combine two DataFrames based on their indexes or a specific column. It is similar to merge(), but typically used when you want to join based on indexes.

**Syntax:**

```
df1.join(df2, on=None, how='left', lsuffix='', rsuffix='')
```

- **df2**: The DataFrame to join with df1.
- **on**: Column to join on (optional, if on is not specified, it will use the index).
- **how**: Type of join: 'left', 'right', 'outer', 'inner' (default is 'left').
- **lsuffix**: Suffix to append to overlapping columns from the left DataFrame.
- **rsuffix**: Suffix to append to overlapping columns from the right DataFrame.

**Example:**

```
df1 = pd.DataFrame({
   'ID': [1, 2, 3],
   'Name': ['Alice', 'Bob', 'Charlie']
}).set_index('ID')

df2 = pd.DataFrame({
   'Score': [85, 92, 78]
}, index=[1, 2, 4])

joined_df = df1.join(df2, how='left')
print(joined_df)
```

**Output**:

```
      Name  Score
ID
1     Alice   85.0
2       Bob   92.0
3   Charlie    NaN
```

In this example, the df1 DataFrame is joined with df2 on their indexes using a left join.

---

**4. append()**

The append() function is used to add rows to the end of a DataFrame. It can be used to append a single DataFrame or a list of DataFrames.

**Syntax:**

df.append(other, ignore_index=False, verify_integrity=False, sort=False)

- **other**: The DataFrame or list of DataFrames to append.
- **ignore_index**: If True, resets the index of the result.
- **verify_integrity**: If True, checks for duplicates in the new DataFrame.
- **sort**: If True, sorts the columns when appending.

**Example:**

```
df1 = pd.DataFrame({
   'A': [1, 2],
   'B': [3, 4]
})

df2 = pd.DataFrame({
   'A': [5, 6],
   'B': [7, 8]
})

appended_df = df1.append(df2, ignore_index=True)
print(appended_df)
```

**Output**:

```
   A  B
0  1  3
1  2  4
2  5  7
3  6  8
```

In this case, df2 is appended to df1, and the index is reset because ignore_index=True.

---

**5. combine_first()**
The combine_first() function is used to combine two DataFrames, filling missing values in the first DataFrame with values from the second DataFrame.

**Syntax:**

df1.combine_first(df2)

- **df2**: The DataFrame to fill missing values from.

**Example:**

```
df1 = pd.DataFrame({
   'A': [1, 2, None],
   'B': [None, 4, 5]
})

df2 = pd.DataFrame({
   'A': [None, None, 3],
   'B': [6, None, None]
})

combined_df = df1.combine_first(df2)
print(combined_df)
```

**Output**:

```
    A    B
0  1.0  6.0
1  2.0  4.0
2  3.0  5.0
```

Here, missing values in df1 are filled with the corresponding values from df2.

---

**6. merge_asof()**
The merge_asof() function performs an as-of merge, which is similar to an ordered join. It is useful when merging time-series or ordered data where you want to match the closest value to a given point.

**Syntax:**

pd.merge_asof(left, right, on=None, by=None, direction='forward', tolerance=None)

- **left** and **right**: The DataFrames to merge.
- **on**: The column to merge on, usually a time or ordered numeric column.
- **direction**: Specifies whether to merge in the 'forward', 'backward', or 'nearest' direction.
- **tolerance**: Maximum distance to search for matches.

**Example:**

```
df1 = pd.DataFrame({
   'Time': [1, 2, 3, 4],
   'Value': [10, 20, 30, 40]
})

df2 = pd.DataFrame({
   'Time': [2, 3],
   'Score': [100, 200]
})

asof_merged_df = pd.merge_asof(df1, df2, on='Time')
print(asof_merged_df)
```

**Output**:

```
   Time  Value  Score
0     1     10      0
1     2     20    100
2     3     30    200
3     4     40    200
```

Here, the merge_asof() function merges the closest Score values based on the Time column.

---

**Summary of Joining and Concatenation Functions:**

- **concat()**: Concatenates multiple DataFrames along rows or columns.
- **merge()**: Performs SQL-style joins based on columns or indexes.
- **join()**: Combines DataFrames based on their indexes (or columns).
- **append()**: Appends rows to a DataFrame.
- **combine_first()**: Fills missing values in the first DataFrame with values from the second.
- **merge_asof()**: Performs an as-of merge, ideal for ordered data or time series.

# ➢ Windowing and Rolling Functions

**1. rolling()**
The rolling() function provides a moving window view of your data, allowing you to perform calculations over a specified number of preceding rows. It is typically used for creating rolling statistics like moving averages.

**Syntax:**

df.rolling(window, min_periods=1, axis=0, closed=None)

- **window**: The size of the moving window (int). This can be a fixed number of periods or a time-based offset (e.g., '5D' for a 5-day window).
- **min_periods**: Minimum number of observations in the window required to have a value (default is 1).
- **axis**: The axis to calculate along (0 for rows, 1 for columns).
- **closed**: Which side of the window to include: 'right', 'left', or 'both'.

**Example (Moving Average):**

```
import pandas as pd

df = pd.DataFrame({
   'Value': [10, 20, 30, 40, 50, 60, 70]
})

# Rolling window size of 3
rolling_avg = df['Value'].rolling(window=3).mean()
print(rolling_avg)
```

**Output**:

```
0    NaN
1    NaN
2   20.0
3   30.0
4   40.0
5   50.0
6   60.0
Name: Value, dtype: float64
```

In this example, a rolling window of size 3 is used to calculate the moving average of the Value column. The first two values are NaN because there are not enough data points in the window to compute the average.

---

**2. expanding()**
The expanding() function is used to perform calculations over an expanding window. It calculates statistics over all previous values up to the current point. It's often used to compute cumulative statistics, like a cumulative sum or average.

**Syntax:**

df.expanding(min_periods=1, axis=0)

- **min_periods**: Minimum number of observations in the window required to have a value (default is 1).

**Example (Cumulative Sum):**

```
expanding_sum = df['Value'].expanding().sum()
print(expanding_sum)
```

**Output**:

```
0     10
1     30
2     60
3    100
4    150
5    210
6    280
Name: Value, dtype: int64
```

In this example, expanding().sum() calculates the cumulative sum of the Value column, where each value is the sum of all preceding values up to that point.

---

**3. ewm()**
The ewm() function is used to perform exponentially weighted moving averages. It is useful for giving more weight to recent observations and less weight to older ones. This is commonly used in financial data analysis (e.g., for stock prices or returns).

**Syntax:**

df.ewm(span=None, adjust=True, ignore_na=False, axis=0)

- **span**: The decay factor. A larger value means that more weight is given to the recent data.
- **adjust**: If True, the calculation will adjust the weights to ensure they sum to 1.
- **ignore_na**: If True, the method will ignore NaN values when calculating the average.
- **axis**: The axis along which the function will be applied (0 for rows, 1 for columns).

**Example (Exponential Moving Average):**

```
df['EWMA'] = df['Value'].ewm(span=3).mean()
print(df)
```

**Output**:

```
   Value       EWMA
0     10  10.000000
1     20  15.000000
2     30  22.500000
3     40  31.250000
4     50  40.625000
5     60  50.312500
6     70  60.156250
```

In this example, an exponential weighted moving average is computed with a span of 3, giving more weight to recent values.

---

**4. shift()**

The shift() function is used to shift the data in a DataFrame or Series by a specified number of periods. It can be useful for creating lag features, comparing data between periods, or calculating differences between adjacent values.

**Syntax:**

```
df.shift(periods=1, freq=None, axis=0, fill_value=None)
```

- **periods**: The number of periods to shift. A positive number shifts the data forward, and a negative number shifts it backward.
- **freq**: The frequency to use for shifting (for time series data).
- **axis**: The axis along which to shift.
- **fill_value**: The value to use for missing data after the shift.

**Example (Shifting Data):**

```
df['Shifted'] = df['Value'].shift(1)
print(df)
```

**Output**:

```
   Value  Shifted
0     10      NaN
1     20     10.0
2     30     20.0
3     40     30.0
4     50     40.0
5     60     50.0
6     70     60.0
```

In this example, shift(1) shifts the Value column by 1 period. The first value is NaN because there is no previous data for it.

---

**5. rolling().apply()**
The apply() function can be used with the rolling() window to apply custom functions over the rolling window. This is useful when you need to compute more complex statistics than the built-in ones like mean, sum, etc.

**Syntax:**

df.rolling(window).apply(func, raw=False, engine='cython', engine_kwargs=None)

- **window**: The size of the rolling window.
- **func**: The custom function to apply.
- **raw**: If True, passes the raw window data as a numpy array to the function; otherwise, a pandas Series is passed.

**Example (Custom Function with Rolling Apply):**

```
rolling_max = df['Value'].rolling(window=3).apply(lambda x: x.max())
print(rolling_max)
```

**Output**:

```
0     NaN
1     NaN
2    30.0
3    40.0
4    50.0
5    60.0
6    70.0
Name: Value, dtype: float64
```

Here, we use apply() with a rolling window of size 3 to calculate the maximum value in each window.

---

**6. window()**
The window() function is a more generic version of the rolling() function and provides additional functionality, such as calculating rolling statistics with a more complex window definition.

**Syntax:**

df.window(window, min_periods=1, axis=0)
This function is mainly used in the context of rolling operations or for specialized window definitions.

---

**Summary of Windowing and Rolling Functions:**

- **rolling()**: Performs rolling window operations such as moving averages, sums, etc.
- **expanding()**: Performs cumulative calculations over an expanding window.
- **ewm()**: Performs exponentially weighted calculations, such as exponential moving averages.
- **shift()**: Shifts data by a specified number of periods (useful for creating lagged features).
- **rolling().apply()**: Applies custom functions over a rolling window.
- **window()**: Generalized version for more complex window operations.

# ➢ Categorical Data Functions

**1. Categorical()**
The Categorical() constructor is used to convert a regular pandas Series into a categorical data type. This is useful for optimizing memory and computational efficiency when dealing with categorical variables.

**Syntax:**

pd.Categorical(values, categories=None, ordered=False, dtype=None)

- **values**: The list or array of values to convert into categorical data.
- **categories**: A list of categories to use (optional). If not provided, pandas will infer the categories.
- **ordered**: If True, the categories will be treated as ordered (ordinal). Default is False.
- **dtype**: Optional. The categorical dtype to use.

**Example:**

```
import pandas as pd

data = ['apple', 'banana', 'cherry', 'apple', 'banana']
categorical_data = pd.Categorical(data)
print(categorical_data)
```

**Output**:

```
[apple, banana, cherry, apple, banana]
Categories (3, object): [apple, banana, cherry]
```

In this example, the data is converted into a categorical data type, with three distinct categories: apple, banana, and cherry.

---

**2. get_dummies()**

The get_dummies() function is used to convert categorical variables into a series of binary (0 or 1) columns. This is typically used when preparing data for machine learning, where categorical data needs to be transformed into a numerical format.

**Syntax:**

```
pd.get_dummies(data, columns=None, drop_first=False, dtype=None)
```

- **data**: The DataFrame or Series containing categorical variables.
- **columns**: The columns to convert into dummy variables (optional). If not specified, all object-type columns are converted.
- **drop_first**: If True, the first category is dropped to avoid multicollinearity (dummy variable trap).
- **dtype**: The data type for the result.

**Example:**

```
df = pd.DataFrame({
   'Fruit': ['apple', 'banana', 'cherry', 'apple', 'banana'],
   'Color': ['red', 'yellow', 'red', 'green', 'yellow']
})

dummies = pd.get_dummies(df, drop_first=True)
print(dummies)
```

**Output**:

| | Fruit_banana | Fruit_cherry | Color_green | Color_red |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 0 |
| 4 | 1 | 0 | 0 | 0 |

In this example, the get_dummies() function converts the categorical columns Fruit and Color into separate binary columns. The first category for each column is dropped to avoid the dummy variable trap (e.g., Fruit_apple is dropped).

---

**3. cut()**
The cut() function is used to segment and sort data values into discrete bins or intervals. This is particularly useful for binning numerical data into categorical data.

**Syntax:**

pd.cut(x, bins, right=True, labels=False, retbins=False, precision=3, include_lowest=False)

- **x**: The data to bin.
- **bins**: The number of bins or the actual bin edges.
- **right**: Whether the bins should be closed on the right side (default is True).
- **labels**: If True, assigns labels to the bins; otherwise, returns integer indicators for the bins.
- **retbins**: If True, returns the bins used for the segmentation.
- **precision**: The precision of the bin edges.
- **include_lowest**: If True, the lowest value is included in the first bin.

**Example (Binning Continuous Data):**

```
import pandas as pd

data = [1, 7, 5, 3, 6, 8, 10, 12, 15]
bins = [0, 5, 10, 15]
labels = ['Low', 'Medium', 'High']

categories = pd.cut(data, bins=bins, labels=labels)
print(categories)
```

**Output**:

```
[Low, Medium, Medium, Low, Medium, Medium, High, High, High]
Categories (3, object): [Low < Medium < High]
```

In this example, the continuous data is divided into three categories: Low, Medium, and High, based on the specified bin edges [0, 5, 10, 15].

---

**4. qcut()**
The qcut() function is similar to cut(), but instead of specifying the bin edges, it divides the data into equal-sized quantiles (e.g., quartiles, percentiles). This is useful when you want to divide the data into categories based on the distribution of the data.

**Syntax:**

pd.qcut(x, q, labels=False)

- **x**: The data to bin.
- **q**: The number of quantiles (bins) to create.
- **labels**: If True, assigns labels to the bins; otherwise, returns integer indicators for the bins.

**Example (Binning Based on Quantiles):**

```
data = [1, 7, 5, 3, 6, 8, 10, 12, 15]
quantiles = pd.qcut(data, q=3, labels=['Low', 'Medium', 'High'])
print(quantiles)
```

**Output**:

```
[Low, Medium, Medium, Low, Medium, Medium, High, High, High]
Categories (3, object): [Low < Medium < High]
```

In this example, the data is divided into three equal-sized quantiles: Low, Medium, and High.

---

**5. factorize()**
The factorize() function is used to encode categorical variables as integers. This is useful for converting labels into a numerical format when performing machine learning tasks or analyzing categorical data.

**Syntax:**

pd.factorize(values, sort=False, na_sentinel=-1)

- **values**: The data to encode.
- **sort**: If True, the categories are sorted.
- **na_sentinel**: The value to use for missing data.

**Example:**

```
data = ['apple', 'banana', 'cherry', 'apple', 'banana']
encoded, unique = pd.factorize(data)
print(encoded)
print(unique)
```

**Output**:
[0 1 2 0 1]

Index(['apple', 'banana', 'cherry'], dtype='object')
In this example, factorize() encodes the unique values in the data list into integers (apple → 0, banana → 1, cherry → 2) and returns the unique categories.

---

**6. CategoricalDtype**
The CategoricalDtype function is used to create a specific categorical type with custom categories and order. This is useful when you want to define categorical variables with an ordered structure (e.g., low, medium, high) and use them for comparisons.

**Syntax:**

pd.CategoricalDtype(categories=None, ordered=False)

- **categories**: The categories to define.
- **ordered**: Whether the categories should be ordered (default is False).

**Example:**

```
dtype = pd.CategoricalDtype(categories=['low', 'medium', 'high'], ordered=True)
cat_series = pd.Series(['medium', 'low', 'high', 'medium'], dtype=dtype)
print(cat_series)
```

**Output**:

```
0    medium
1       low
2      high
3    medium
dtype: category
Categories (3, object): [low < medium < high]
```

In this example, a custom categorical type with ordered categories is created. The series is then defined using this ordered categorical type.

---

**7. add_categories()**
The add_categories() function allows you to add new categories to an existing categorical variable. This is useful when you need to extend the range of categories in a categorical column.

**Syntax:**
cat_series.add_categories(new_categories)

- **new_categories**: The categories to add.

**Example:**

```
cat_series = pd.Series(['medium', 'low', 'high', 'medium'], dtype='category')
cat_series = cat_series.add_categories(['very high'])
print(cat_series)
```

**Output**:

```
0     medium
1        low
2       high
3     medium
dtype: category
Categories (4, object): [low < medium < high < very high]
```

In this example, the category very high is added to the existing categorical series.

---

**Summary of Categorical Data Functions:**

- **Categorical()**: Converts a Series into a categorical data type.
- **get_dummies()**: Converts categorical variables into dummy (binary) variables.
- **cut()**: Bins numerical data into discrete categories based on bin edges.
- **qcut()**: Bins numerical data into quantiles.
- **factorize()**: Encodes categorical data as integers.
- **CategoricalDtype**: Creates custom categorical types with categories and order.
- **add_categories()**: Adds new categories to an existing categorical variable.