

# ✓ Pandas

## Agenda

### 1. Introduction to Pandas

- What is Pandas?
- Installation
- Importing Pandas

### 2. Pandas Data Structures

- Series
- DataFrame

### 3. Basic Operations on DataFrames

- Viewing Data
- Selection
- Filtering Data

### 4. DataFrame Manipulations

- Adding/Deleting Columns
- Renaming Columns
- Handling Missing Data

## ✓ Introduction to Pandas



## What is Pandas?

- Pandas is a powerful and flexible open-source data analysis and manipulation library for Python. It is built on top of the NumPy library and provides data structures like Series and DataFrame, which are designed to work efficiently with structured data.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

## ✓ Installation

```
!pip install pandas
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.1.4)  
Requirement already satisfied: numpy<2,>=1.22.4 in /usr/local/lib/python3.10/dist-packages (1.24.4)  
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (2.9.0)  
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (2024.1)  
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (2024.1)  
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (1.17.0)
```

## ✓ Importing Pandas

```
import pandas as pd
```

## ✓ Pandas Data Structures

- Pandas provides two primary data structures for managing data: Series and DataFrame. Both are built on top of NumPy arrays, but they provide more powerful features, particularly for data analysis tasks.

## ✓ Series

- A Series is a one-dimensional labeled array capable of holding any data type (integers, strings, floats, Python objects, etc.). It is similar to a column in a spreadsheet or a single column in a DataFrame.
- Key Characteristics:
  - Indexing: Each element in a Series has an associated label, known as its index.
  - Homogeneous Data: A Series is homogeneous, meaning it can hold data of only one type.

```
import pandas as pd
```

```
# Creating a Series from a list  
s = pd.Series([1, 2, 3, 4, 5])
```

```
# Creating a Series with custom index  
s = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
```

- Accessing Data:
  - By Position: `s[0]` or `s[:3]`
  - By Index: `s['a']` or `s[['a', 'c']]`

```
s = pd.Series([10, 20, 30], index=['a', 'b', 'c'])  
print(s)
```

```
↵ a    10  
  b    20  
  c    30  
  dtype: int64
```

## ▼ DataFrame

- A DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). It is similar to a spreadsheet or SQL table, or a dictionary of Series objects.
- Key Characteristics:
  - Indexing: Both rows and columns are indexed, allowing for easy access and manipulation.
  - Heterogeneous Data: Each column in a DataFrame can contain different data types (e.g., integers, floats, strings).
  - Size-Mutable: The size of the DataFrame can be changed (adding/removing rows or columns).

```
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)
```

```
# Creating a DataFrame from a list of lists
df = pd.DataFrame([[1, 2], [3, 4]], columns=['A', 'B'])
```

- Accessing Data:
  - By Column Name: `df['Name']` or `df[['Name', 'Age']]`
  - By Row Index: `df.iloc[0]` or `df.loc[0]`
  - Slicing

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

```
➡
```


	Name	Age	Salary
0	Alice	25	50000
1	Bob	30	60000
2	Charlie	35	70000

## ✓ Index Object

- Both Series and DataFrame use an Index object to label their axes. The Index object ensures that labels are unique and immutable, allowing for efficient alignment and joining of data.

```
# Accessing index of a DataFrame
print(df.index)
# Output: RangeIndex(start=0, stop=3, step=1)

# Accessing columns of a DataFrame
print(df.columns)
# Output: Index(['Name', 'Age', 'Salary'], dtype='object')
```



```
RangeIndex(start=0, stop=3, step=1)
Index(['Name', 'Age', 'Salary'], dtype='object')
```

## ✓ Basic Operations on DataFrames

### ✓ Viewing Data

- Viewing data in a DataFrame is crucial for understanding its structure, content, and summary statistics. Pandas provides several methods to quickly inspect your data:

#### 1. df.head()

- Purpose: Displays the first few rows of the DataFrame.
- Default Behavior: Shows the first 5 rows.

```
import pandas as pd

# Sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Edward'],
        'Age': [25, 30, 35, 40, 45],
        'Salary': [50000, 60000, 70000, 80000, 90000]}
df = pd.DataFrame(data)

# Display the first 5 rows
df.head()
```



	Name	Age	Salary
0	Alice	25	50000
1	Bob	30	60000
2	Charlie	35	70000
3	David	40	80000
4	Edward	45	90000

## 2. df.tail()

- Purpose: Displays the last few rows of the DataFrame.
- Default Behavior: Shows the last 5 rows.

```
# Display the last 5 rows
df.tail()
```



	Name	Age	Salary
0	Alice	25	50000
1	Bob	30	60000
2	Charlie	35	70000
3	David	40	80000
4	Edward	45	90000

## 3. df.info()

- Purpose: Provides a concise summary of the DataFrame.
- Details:
  - The number of entries (rows).
  - The index range.
  - Column names and their data types.
  - Non-null counts (useful for identifying missing data).
  - Memory usage.

```
df.info()
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
#   Column  Non-Null Count  Dtype
#
```

```

-----
0   Name      5 non-null    object
1   Age       5 non-null    int64
2   Salary    5 non-null    int64
dtypes: int64(2), object(1)
memory usage: 248.0+ bytes

```

#### 4. df.describe()

- Purpose: Generates descriptive statistics of the DataFrame's numerical columns.
- Details:
  - Count: Number of non-null entries.
  - Mean: Average of the data.
  - Standard Deviation: Dispersion of the data.
  - Min/Max: Minimum and maximum values.
  - Percentiles: 25th, 50th (median), and 75th percentiles.

df.describe()



	Age	Salary
<b>count</b>	5.000000	5.000000
<b>mean</b>	35.000000	70000.000000
<b>std</b>	7.905694	15811.388301
<b>min</b>	25.000000	50000.000000
<b>25%</b>	30.000000	60000.000000
<b>50%</b>	35.000000	70000.000000
<b>75%</b>	40.000000	80000.000000
<b>max</b>	45.000000	90000.000000

#### ✓ Selection

- Selecting data from a Pandas DataFrame is a fundamental operation that allows you to filter, slice, and manipulate your data based on specific criteria. Pandas provides various ways to select data from a DataFrame, including by column, row, index, and using conditions.

##### 1. Using .loc[] (Label-based Indexing):

- Selects rows based on label/index names.

- `df.loc[row_label]`

## 2. Using `.iloc[]` (Position-based Indexing):

- Selects rows based on integer location/position.
- `df.iloc[row_position]`

## ✓ Filtering Data

- Filtering data in Pandas involves selecting subsets of data that meet specific conditions. This is a powerful way to focus on the data that is most relevant for analysis, based on criteria like column values, data types, or custom logic.

### 1. Basic Filtering with Conditions

- Filter Rows Based on a Single Condition:
  - You can filter rows by applying a condition directly to a DataFrame column.

```
df[df['Age'] > 30] # Selects rows where 'Age' is greater than 30
```



	Name	Age	Salary
2	Charlie	35	70000
3	David	40	80000
4	Edward	45	90000

- Filter Rows Based on Multiple Conditions:
  - Use logical operators like `&` (AND), `|` (OR), and `~` (NOT) to combine multiple conditions. Enclose each condition in parentheses.

```
df[(df['Age'] > 30) & (df['Salary'] > 60000)] # Both conditions must be True
df[(df['Age'] < 35) | (df['Name'] == 'Alice')] # Either condition can be True
df[~(df['Name'] == 'Bob')] # Selects all rows where 'Name' is not 'Bob'
```





	Name	Age	Salary
0	Alice	25	50000
2	Charlie	35	70000
3	David	40	80000
4	Edward	45	90000

## 2. Filtering Using isin()

- Filter Rows with Specific Values:

- The `isin()` method allows you to filter rows where a column's value matches any value in a list.

```
df[df['Name'].isin(['Alice', 'Bob'])] # Selects rows where 'Name' is 'Alice' or 'Bob'
```



	Name	Age	Salary
0	Alice	25	50000
1	Bob	30	60000

## 3. Filtering Using between()

- Filter Rows Between Two Values:

- The `between()` method allows you to filter rows where a column's value falls within a specified range.

```
df[df['Age'].between(30, 40)] # Selects rows where 'Age' is between 30 and 40 (inclusive)
```



	Name	Age	Salary
1	Bob	30	60000
2	Charlie	35	70000
3	David	40	80000

## 4. Filtering Using String Methods

- Filter Rows Based on String Matching:

- You can use string methods like `str.contains()`, `str.startswith()`, `str.endswith()` to filter rows based on string patterns.

```
df[df['Name'].str.contains('li')] # Selects rows where 'Name' contains 'li'
df[df['Name'].str.startswith('A')] # Selects rows where 'Name' starts with 'A'
df[df['Name'].str.endswith('e')] # Selects rows where 'Name' ends with 'e'
```



	Name	Age	Salary
0	Alice	25	50000
2	Charlie	35	70000

## 5. Filtering Using `query()`

- Use SQL-Like Queries:
  - The `query()` method allows you to filter DataFrames using a string expression that looks like a SQL query.

```
df.query('Age > 30') # Same as df[df['Age'] > 30]
df.query('Age > 30 & Salary > 60000') # Same as df[(df['Age'] > 30) & (df['Salary'] > 60000)]
```



	Name	Age	Salary
2	Charlie	35	70000
3	David	40	80000
4	Edward	45	90000

## ✓ DataFrame Manipulations

- Manipulating DataFrames in Pandas involves a range of operations that allow you to modify, reshape, and transform your data. Here's a comprehensive guide to some common DataFrame manipulations:

### ✓ Adding and Dropping Columns

```
# Add a New Column:
df['New Salary'] = df['Salary'] * 1.1 # Adding a new column with a calculated value
```

```
# Drop a Column:
```

```
df.drop('New Salary', axis=1, inplace=True) # Drop the 'New Salary' column
```

## ✓ Renaming Columns and Index

Rename Columns:

```
df.rename(columns={'Salary': 'Annual Salary'}, inplace=True) # Renaming 'Salary' to 'Annual
```

Rename Index:

```
df.rename(index={0: 'Row_0'}, inplace=True) # Renaming index 0 to 'Row_0'
```

```
df_reindexed = df.reindex([0, 1, 2, 3, 4, 5]) # Reindexing rows with a new index list
```