Pandas for Beginners

Connect with me: Youtube | LinkedIn | WhatsApp Channel | Web | Facebook | Twitter

- Download PDF
- To access the updated handouts, please click on the following link: https://yasirbhutta.github.io/python/docs/modules/pandas.html

📽 YouTube Playlists to Learn Python:

- 🔗 Python Tutorials for Beginners
- 🔗 Python Code Challenges | Quiz
- 🔗 Python Exercises

1. **Introduction to Pandas**

- What is Pandas?
- Installing Pandas (`pip install pandas`)
- Importing Pandas (`import pandas as pd`)

2. **Data Structures in Pandas**

- **Series**: A one-dimensional labeled array.
    - Creating a Series
    - Accessing elements in a Series
- **DataFrame**: A two-dimensional labeled data structure (like a spreadsheet or SQL table).
    - Creating a DataFrame from:
        - Lists of lists
        - Dictionaries
        - CSV/Excel files

3. **Basic Operations with DataFrames**

- Viewing data:
    - `head()`, `tail()`, `info()`, `describe()`
- Selecting data:
    - Columns (`df['column']`)
    - Rows (`iloc`, `loc`)
- Filtering rows based on conditions
- Adding and deleting columns

4. **Data Cleaning**

- Handling missing values:
    - `dropna()`
    - `fillna()`
- Renaming columns
- Changing data types (`astype()`)

5. **Data Manipulation**

- Sorting data (`sort_values()`)
- Grouping data (`groupby()`)
- Aggregations (`mean()`, `sum()`, `count()`, etc.)
- Merging and joining DataFrames (`merge()`, `concat()`)

6. **Reading and Writing Data**

- Reading from:
  - CSV (`read_csv()`)
  - Excel (`read_excel()`)
  - JSON (`read_json()`)
- Writing to:
  - CSV (`to_csv()`)
  - Excel (`to_excel()`)

7. **Data Visualization with Pandas**

- Basic plots with Pandas:
  - Line plot, bar plot, histogram, scatter plot (`df.plot()`)

8. **Useful Functions for Analysis**

- `value_counts()`
- `unique()`
- `nunique()`
- `pivot_table()`

9. **Practical Examples**

- Analyzing real-world datasets
- Cleaning messy data
- Simple data analysis projects

1. 📌 Introduction to Pandas

1.1 What is Pandas?

- **Definition:** pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. [1]

- **Key Features**:

  - Data alignment and missing data handling.
  - Support for importing/exporting data from various file formats (CSV, Excel, SQL, etc.).
  - Tools for reshaping, pivoting, and aggregating data.
  - Time series functionality.

For more details on Data alignment, see Appendix B.

1.2 **Why Use Pandas?**

- **Benefits**:
  - Simplifies data analysis tasks.
  - Efficient handling of large datasets.
  - Allows easy data cleaning and preprocessing.
  - Combines flexibility with powerful tools for slicing, filtering, and transforming data.

1.3 **Installing Pandas**

- Install Pandas using `pip`:

```
pip install pandas
```

- **Verifying Installation**:

```
import pandas as pd
print(pd.__version__)
```

1.4 **Importing Pandas**

- Import the Pandas library with a conventional alias:

```
import pandas as pd
```

- Why `pd`?
  It's a commonly used alias to shorten the code and improve readability.

2. Basic data structures in pandas

Pandas provides two types of classes for handling data:

- **Series**:
  - a one-dimensional labeled array holding data of any type such as integers, strings, Python objects etc. similar to a column in a spreadsheet. [2]
- **DataFrame**:
  - a two-dimensional data structure that holds data like a two-dimension array or a table with rows and columns. [2]

Creating a Simple Series

Creating a Series by passing a list of values, letting pandas create a default RangeIndex. [2]

```
import pandas as pd
import numpy as np

s = pd.Series([1, 3, 5, np.nan, 6, 8])
print(s)
```

`np.nan` stands for "Not a Number", representing a missing or undefined numerical value (similar to NaN in other contexts).

Indexing: Both Series and DataFrames have index labels to identify data.

Creating a Simple DataFrame

```
import pandas as pd

# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

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

Output:

```
     Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

Example: How to Create a Data Frame with Fruits and Colors Example

1. Loading CSV Datasets

You can load CSV files from your local system or directly from a URL into Pandas using pd.read_csv().

Example Titanic Dataset (from a URL)

```
import pandas as pd

# Loading the Titanic dataset from a URL
url =
"https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
```

```
titanic = pd.read_csv(url)
print(titanic.head())
```

Example: Local CSV File

Loading a local CSV file

```
titanic = pd.read_csv("path_to_your_file/titanic.csv")
print(titanic.head())
```

2. Loading Excel Files

Pandas can easily read Excel files using pd.read_excel().

Example: Tasks Trackers Dataset download

Loading an Excel file

```
import pandas as pd

tasks = pd.read_excel("tasks_ds.xlsx")
print(tasks.head())
```

3. **Basic Operations with DataFrames**

3.1 Viewing Data

When working with DataFrames in libraries like **pandas**, it's essential to quickly inspect your data to understand its structure. Below are some key functions to view data:

- `head(n)`: Displays the first *n* rows (default is 5).

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

- `tail(n)`: Displays the last *n* rows (default is 5).

  ```
  df.tail()
  ```

- `info()`: Provides a concise summary of the DataFrame, including column names, non-null counts, and data types.

  ```
  df.info()
  ```

- `describe()`: Generates summary statistics for numerical columns, such as mean, standard deviation, and quartiles.

  ```
  df.describe()
  ```

3.2 Selecting Data

Selecting specific parts of a DataFrame is a common operation. You can select **columns** and **rows** using different methods:

- **Selecting Columns**
  - Using the column name in square brackets:

    ```
    df['column_name']
    ```

  - Selecting multiple columns by providing a list of column names:

    ```
    df[['col1', 'col2']]
    ```

Loading the Titanic dataset (assuming it's available as a CSV file):

```
import pandas as pd

# Load the Titanic dataset
df = pd.read_csv('titanic.csv')
```

Example: Selecting a Single Column

- For example, to select the **"Age"** column:

  ```
  df['Age']
  ```

- This returns a **Series** representing the "Age" column.

Example: Selecting Multiple Columns

- To select **"Name"**, **"Age"**, and **"Survived"** columns:

  ```
  df[['Name', 'Age', 'Survived']]
  ```

- This returns a **DataFrame** with the specified columns.
- **Selecting Rows**
  - Using `iloc` (integer-location based): Select rows by their index position.

    ```
    df.iloc[0]       # First row
    df.iloc[1:4]     # Rows 2 to 4
    ```

  - Using `loc` (label-based): Select rows by their index labels.

    ```
    df.loc[0]                 # Row with index 0
    df.loc[0:3]               # Rows from index 0 to 3 (inclusive)
    df.loc[df['col'] > 10]    # Rows where 'col' > 10
    ```

Selecting Rows with the Titanic Dataset

1. **Loading the Titanic Dataset**

Let's load the dataset and inspect its structure:

```
import pandas as pd

# Load the Titanic dataset (assuming it's available as 'titanic.csv')
df = pd.read_csv('titanic.csv')

# Display the first few rows to understand the data
print(df.head())
```

This might display something like:

PassengerId	Name	Age	Survived	Pclass
1	Braund, Mr. Owen Harris	22.0	0	3
2	Cumings, Mrs. John Bradley	38.0	1	1
3	Heikkinen, Miss. Laina	26.0	1	3

PassengerId	Name	Age	Survived	Pclass
4	Futrelle, Mrs. Jacques Heath	35.0	1	1
5	Allen, Mr. William Henry	35.0	0	3

Selecting Rows Using `iloc` (Integer-location Based)

- **Select the First Row**:

```
first_row = df.iloc[0]
print(first_row)
```

Output:

```
PassengerId                          1
Name             Braund, Mr. Owen Harris
Age                               22.0
Survived                             0
Pclass                               3
Name: 0, dtype: object
```

- **Select Rows 2 to 4** (indices 1 to 3):

```
rows_2_to_4 = df.iloc[1:4]
print(rows_2_to_4)
```

Output:

```
   PassengerId                          Name   Age  Survived  Pclass
1            2     Cumings, Mrs. John Bradley  38.0         1       1
2            3         Heikkinen, Miss. Laina  26.0         1       3
3            4   Futrelle, Mrs. Jacques Heath  35.0         1       1
```

Selecting Rows Using `loc` (Label-based)

- **Select the Row with Index 0**:

```
row_index_0 = df.loc[0]
print(row_index_0)
```

Output:

```
PassengerId                        1
Name              Braund, Mr. Owen Harris
Age                               22.0
Survived                             0
Pclass                               3
Name: 0, dtype: object
```

- **Select Rows with Index 0 to 3** (inclusive):

```
rows_0_to_3 = df.loc[0:3]
print(rows_0_to_3)
```

Output:

```
   PassengerId                            Name   Age  Survived  Pclass
0            1       Braund, Mr. Owen Harris    22.0         0       3
1            2     Cumings, Mrs. John Bradley   38.0         1       1
2            3         Heikkinen, Miss. Laina   26.0         1       3
3            4  Futrelle, Mrs. Jacques Heath    35.0         1       1
```

- **Select Rows Where `Age` > 30**:

```
rows_age_above_30 = df.loc[df['Age'] > 30]
print(rows_age_above_30.head())
```

Output:

```
   PassengerId                            Name   Age  Survived  Pclass
1            2     Cumings, Mrs. John Bradley   38.0         1       1
3            4  Futrelle, Mrs. Jacques Heath    35.0         1       1
5            6        Moran, Mr. James          32.0         0       3
6            7        McCarthy, Mr. Timothy J   54.0         0       3
9           10  Nasser, Mrs. Nicholas           14.0         1       2
```

3.3 Filtering Rows Based on Conditions

Filtering rows allows you to extract data that meets certain criteria. Conditions can be combined using logical operators:

- **Filter based on a single condition**:

```
df[df['column_name'] > 50]
```

- **Filter based on multiple conditions**:
  - Using **AND (`&`)**:

    ```
    df[(df['col1'] > 50) & (df['col2'] == 'Value')]
    ```

  - Using **OR (`|`)**:

    ```
    df[(df['col1'] > 50) | (df['col2'] == 'Value')]
    ```

- **Filter based on text matching**:

  ```
  df[df['col'].str.contains('keyword')]
  ```

3.4 Adding and Deleting Columns

- **Adding a New Column**

  You can create new columns by assigning values or calculations based on existing columns:

  ```
  df['new_col'] = df['col1'] + df['col2']
  ```

- **Deleting a Column**

  Use the `drop` method to remove a column. Set `axis=1` to specify columns:

  ```
  df.drop('col_to_delete', axis=1, inplace=True)
  ```

  Alternatively, use the `del` statement:

  ```
  del df['col_to_delete']
  ```

1.9 **Key Pandas Terminology**

- **Index**: The labels for rows in a Series or DataFrame.
- **Column**: A named set of data within a DataFrame.

- **Row**: An individual record within a DataFrame.

Key Terms

True/False (Mark T for True and F for False)

Answer Key (True/False):

Multiple Choice (Select the best answer)

1. **Which function would you use to determine the type of a variable in Python?**
    - A) id()
    - B) type()
    - C) str()
    - D) isinstance()

Watch this video for the answer:

Answer key (Mutiple Choice):

Fill in the Blanks

Answer Key (Fill in the Blanks):

Exercises

1. Skill Level Categories Define clear categories based on skill levels, such as:

Beginner: Basic concepts and syntax. Intermediate: More complex problems involving data structures and algorithms. Advanced: Challenging problems that require in-depth understanding and optimization.

Review Questions

Answers to Review Questions:

References and Bibliography

- [1] Pandas, “Python Data Analysis Library,” Pydata.org, 2018. https://pandas.pydata.org/
- [2] Pandas, “User Guide — pandas 1.0.1 documentation,” Pydata.org, 2014. https://pandas.pydata.org/docs/user_guide/index.html

Appendices

Appendix A: Loading and Handling Datasets in Pandas

Pandas doesn't come with built-in datasets like some other libraries, but it offers many ways to load and handle external datasets. You can easily read data from CSV, Excel, SQL, JSON, and other formats using Pandas.

Here are common datasets you can load and work with in Pandas, along with some examples of reading them into your environment:

1. Loading CSV Datasets

You can load CSV files from your local system or directly from a URL into Pandas using pd.read_csv().

Example Titanic Dataset (from a URL)

```
import pandas as pd

# Loading the Titanic dataset from a URL
url =
"https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
titanic = pd.read_csv(url)
print(titanic.head())
```

Example: Local CSV File

Loading a local CSV file

```
titanic = pd.read_csv("path_to_your_file/titanic.csv")
print(titanic.head())
```

2. Loading Excel Files

Pandas can easily read Excel files using pd.read_excel().

#xample: Superstore Dataset

Loading an Excel file

```
superstore = pd.read_excel("path_to_your_file/superstore_sales.xlsx")
print(superstore.head())
```

3. Loading JSON Files

You can load JSON files using pd.read_json().

Example: JSON File Loading

```
# Loading a JSON file
json_data = pd.read_json("path_to_your_file/data.json")
print(json_data.head())
```

4. Loading SQL Databases

If you're working with databases, Pandas can directly query them using SQL queries.

Example: Loading Data from SQL

```
import sqlite3

# Create connection to your SQLite database
conn = sqlite3.connect('database_name.db')

# Query the database
data = pd.read_sql_query("SELECT * FROM table_name", conn)
print(data.head())
```

5. Loading HTML Tables

Pandas can extract tables from HTML web pages using pd.read_html().

Example: Loading Data from an HTML Table

```
# Loading data from a webpage with HTML tables
url = "https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)"
tables = pd.read_html(url)
print(tables[0].head())  # The first table on the page
```

6. Loading Data from APIs

You can load data from APIs that return JSON, CSV, or other formats. For example, using the Kaggle API, you can download datasets and load them into Pandas.

Example: Loading Kaggle Dataset (after downloading)

```
# After downloading a dataset from Kaggle
kaggle_data = pd.read_csv("path_to_downloaded_kaggle_file.csv")
print(kaggle_data.head())
```

7. Loading Data from Google Sheets

You can also read data from Google Sheets by exporting them as CSV and reading into Pandas.

Example: Loading Data from Google Sheets

```
# Google Sheets shared link with export format as CSV
sheet_url = "https://docs.google.com/spreadsheets/d/your_sheet_id/export?
format=csv"
google_sheets_data = pd.read_csv(sheet_url)
print(google_sheets_data.head())
```

8. Loading Data from Zip Files

Pandas can read CSVs from zipped files directly without unzipping them.

Example: Loading from a Zip File

```
# Loading CSV from a zipped file
zip_url = "https://your_url/file.zip"
zipped_data = pd.read_csv(zip_url, compression='zip')
print(zipped_data.head())
```

9. Loading Data from a Clipboard

You can even copy data from somewhere and paste it into Pandas using pd.read_clipboard().

Example: Loading Clipboard Data

```
# Assuming you've copied a table from a webpage or a document
clipboard_data = pd.read_clipboard()
print(clipboard_data.head())
```

10. Sample Datasets in Python Libraries

While Pandas itself doesn’t provide built-in datasets, you can use datasets from libraries like Seaborn and Scikit-learn and load them into Pandas:

Example: Seaborn's Titanic Dataset into Pandas

```
import seaborn as sns

# Load Titanic dataset from Seaborn and convert to Pandas DataFrame
titanic = sns.load_dataset('titanic')
print(titanic.head())

Example: Scikit-learn Iris Dataset into Pandas

from sklearn.datasets import load_iris

# Load Iris dataset and convert to Pandas DataFrame
```

```
iris = load_iris()
iris_df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
print(iris_df.head())
```

How to store MySQL results in a pandas DataFrame using Python

There are two primary ways to store MySQL results in a pandas DataFrame using Python:

1. Using pandas.read_sql()

This is the recommended approach as it's specifically designed for this purpose. Here's how it works:

```
import pandas as pd
import mysql.connector

# Establish connection
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)

# Define your SQL query
sql = "SELECT * FROM mytable"  # Replace with your specific query

# Read the results into a DataFrame
df = pd.read_sql(sql, mydb)  # mydb is the connection object

# Close the connection
mydb.close()

# Now you can work with the data in your DataFrame (df)
print(df.head())  # View the first few rows
```

Explanation:

- Import `pandas` and `mysql.connector`.
- Establish a connection to your MySQL database.
- Define your SQL query string (`sql`).
- Use `pd.read_sql(sql, mydb)` to execute the query and store the results in a pandas DataFrame named `df`. The `mydb` argument provides the connection object.
- Close the connection after reading the data.
- Now you can use the `df` DataFrame for further analysis or manipulation.

2. Using cursor.fetchall() and DataFrame constructor

This method involves fetching the results as a list of tuples and then constructing a DataFrame from it. Here's an example:

```
import pandas as pd
import mysql.connector

# Establish connection
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)

mycursor = mydb.cursor()

# Execute your SQL query
sql = "SELECT * FROM mytable"
mycursor.execute(sql)

# Fetch the results
data = mycursor.fetchall()  # data is a list of tuples

# Define column names (optional, but recommended for clarity)
column_names = [i[0] for i in mycursor.description]  # Get column names from cursor description

# Create the DataFrame
df = pd.DataFrame(data, columns=column_names)

# Close connection (same as previous method)
mycursor.close()
mydb.close()

# Now you can work with the DataFrame (df)
print(df.head())
```

Explanation:

- Import necessary libraries.
- Establish connection and create a cursor.
- Execute your SQL query using the cursor.
- Fetch the results using `fetchall()` which returns a list of tuples.
- Optionally, define column names based on the cursor description.
- Construct the DataFrame using `pd.DataFrame(data, columns=column_names)`.
- Close the connection.
- Now you can use the `df` DataFrame for further analysis.

Choosing the right approach:

- `pandas.read_sql()` is generally preferred as it's more concise and efficient, especially for larger datasets.

- The cursor-based approach might be useful if you need more control over the cursor object or want to perform additional operations before constructing the DataFrame.

Example #: Using SQLAlchemy Engine

```
from sqlalchemy import create_engine

# Construct the connection URL (replace with your credentials)
engine =
create_engine("mysql+mysqlconnector://yourusername:yourpassword@host/yourdatabase"
)

df = pd.read_sql(sql, engine)
```

```
from sqlalchemy import create_engine

# Construct the connection URL (replace with your credentials)
engine = create_engine("mysql+mysqlconnector://root:abc1234@localhost/library")

# Define your SQL query
sql = "SELECT * FROM books"  # Replace with your specific query

# Read the results into a DataFrame
df = pd.read_sql(sql, engine)  # mydb is the connection object

# Close the connection


# Now you can work with the data in your DataFrame (df)
print(df.head())  # View the first few rows
```

Example #: Using Database String URI

```
import pandas as pd

# Replace with your connection string details
connection_string =
"mysql+mysqlconnector://yourusername:yourpassword@host/yourdatabase"
df = pd.read_sql(sql, connection_string)
```

Appendix B: 📌 Data Alignment in Pandas

Data alignment refers to how Pandas handles operations between data structures (such as `Series` or `DataFrames`) with differing indexes. When performing operations like addition, subtraction, or merging, Pandas automatically aligns the data by their index labels to ensure that operations happen between corresponding elements.

This feature helps simplify data operations and avoid errors, especially when dealing with real-world datasets that may not always be perfectly aligned.

◇ Example of Data Alignment with Series

When performing operations between two `Series` with different indexes, Pandas aligns the data by the index labels and fills any missing values with `NaN` (Not a Number).

```
import pandas as pd

# First Series
s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# Second Series with different index
s2 = pd.Series([4, 5, 6], index=['b', 'c', 'd'])

# Adding the two Series
result = s1 + s2

print(result)
```

Output:

```
a    NaN
b    6.0
c    8.0
d    NaN
dtype: float64
```

Explanation:

- The elements with matching indexes (`b` and `c`) are added together.
- For indexes `a` and `d`, there are no corresponding values in the other Series, so the result is `NaN`.

◇ Data Alignment with DataFrames

When performing operations on `DataFrames`, Pandas aligns both rows and columns based on their respective indexes.

```
# First DataFrame
df1 = pd.DataFrame({
    'A': [1, 2],
    'B': [3, 4]
}, index=['row1', 'row2'])
```

```
# Second DataFrame with different columns and rows
df2 = pd.DataFrame({
    'B': [5, 6],
    'C': [7, 8]
}, index=['row2', 'row3'])

# Adding the two DataFrames
result = df1 + df2

print(result)
```

Output:

```
        A     B    C
row1  NaN   NaN  NaN
row2  NaN   9.0  NaN
row3  NaN   NaN  NaN
```

Explanation:

- The addition is performed where both rows and columns match (`row2` and `B`).
- Missing rows or columns result in `NaN`.

◇ Handling Missing Data During Alignment

You can handle missing data resulting from alignment by using methods like:

- **`fillna()`**: Replace `NaN` with a specific value.
- **`add()`, `sub()`, etc. with `fill_value`**: Provide a default value for missing entries.

Example using `fill_value`:

```
result = s1.add(s2, fill_value=0)
print(result)
```

Output:

```
a    1.0
b    6.0
c    8.0
d    6.0
dtype: float64
```

☑ Summary

- **Data Alignment** ensures operations occur between matching indexes.
- Non-matching indexes result in NaN unless specified otherwise.
- Pandas handles alignment automatically, making data manipulation intuitive and error-free.