# Pandas Exploration I

pandas

## Introduction

Pandas stands out as the preeminent library within the Python ecosystem for data manipulation and analysis. It boasts speed, power, flexibility, ease of use, and the open-source advantage.

Pandas was conceived and developed by **Wes McKinney (A.K.A GOD MCKINNEY)**, a financial analyst turned software developer. Wes recognized the need for a tool that could effectively address the challenges of data analysis in the financial industry. His vision was to create a Python library that could provide the same data manipulation capabilities found in popular spreadsheet software and relational databases.

In 2008, Wes McKinney began working on Pandas while at AQR Capital Management. His passion and dedication led to the release of the first version of Pandas in 2009. It didn't take long for Pandas to gain traction in the Python community, and it quickly became an essential tool for data analysts and scientists.

**Key Features of Pandas**:Pandas offers an array of compelling features, including:

* **DataFrame (core)**: A swift and efficient DataFrame object for seamless data manipulation, complete with built-in indexing.
* **Data I/O**: Streamlined data reading and writing in various formats, such as Microsoft Excel, CSV, SQL databases, and more.
* **Robust Data Manipulation**: Integrated and efficient methods for a wide spectrum of data manipulations, including handling missing data, subsetting, merging, and more.
* **Temporary Data Handling**: Pandas excels in managing temporary data, making it the preferred choice for working with panel data (hence its name).
* **Integration**: Smooth integration with other data analysis and machine learning libraries like scikit-learn, scipy, seaborn, and plotly.
* **Widespread Usage**: Pandas enjoys widespread adoption across both private and academic sectors, making it a go-to tool for data enthusiasts.

**Empowering Data Analysis**: Pandas provides high-level data structures and functions tailored to expedite your work with structured or tabular data. Since its debut in 2010, Pandas has played a pivotal role in elevating Python as a robust and efficient data analysis environment.

The primary Pandas objects you'll encounter in this guide are the DataFrame—a column-oriented, tabular data structure with intuitive row and column labels—and the Series—a labeled, one-dimensional array.

Pandas marries the high-performance principles of NumPy with the versatile data manipulation capabilities of spreadsheets and relational databases, such as SQL. It introduces sophisticated indexing features that simplify reshaping, slicing, aggregating, and selecting data subsets.

In summary, Pandas empowers data analysts, scientists, and engineers to wield Python as a potent tool for a wide array of data-related tasks, revolutionizing the way structured data is managed and analyzed.

image

Source: [Forbes](https://www.forbes.com/sites/gilpress/2016/03/23/data-preparation-most-time-consuming-least-enjoyable-data-science-task-survey-says/#1ba071616f63)

## Installation

The first thing you should do will always be pip install pandas, conda install pandas

import pandas as pd  
import numpy as np

## Introduction to pandas data structures

To get started with pandas, you'll need to get comfortable with its two working data structures: Series and DataFrame. Although they are not a universal solution to all problems, they provide a solid and easy-to-use foundation for most applications.

### Series

A Series is like a one-dimensional array, but with a twist. It holds a sequence of values, similar to what you'd find in NumPy arrays, but it also pairs each value with a label known as an index. This combination of data values and labels gives Series its power, making it a versatile tool for efficient data storage and retrieval.

Here's how to create a basic Series using a list of numbers:

data = [10, 20, 30, 40, 50]  
series = pd.Series(data)  
print(series)  
  
# All methods implicit in serie  
print([i for i in dir(series) if "\_" not in i])

0 10  
1 20  
2 30  
3 40  
4 50  
dtype: int64  
['T', 'abs', 'add', 'agg', 'aggregate', 'align', 'all', 'any', 'apply', 'argmax', 'argmin', 'argsort', 'array', 'asfreq', 'asof', 'astype', 'at', 'attrs', 'autocorr', 'axes', 'backfill', 'between', 'bfill', 'bool', 'clip', 'combine', 'compare', 'copy', 'corr', 'count', 'cov', 'cummax', 'cummin', 'cumprod', 'cumsum', 'describe', 'diff', 'div', 'divide', 'divmod', 'dot', 'drop', 'droplevel', 'dropna', 'dtype', 'dtypes', 'duplicated', 'empty', 'eq', 'equals', 'ewm', 'expanding', 'explode', 'factorize', 'ffill', 'fillna', 'filter', 'first', 'flags', 'floordiv', 'ge', 'get', 'groupby', 'gt', 'hasnans', 'head', 'hist', 'iat', 'idxmax', 'idxmin', 'iloc', 'index', 'info', 'interpolate', 'isin', 'isna', 'isnull', 'item', 'items', 'keys', 'kurt', 'kurtosis', 'last', 'le', 'list', 'loc', 'lt', 'map', 'mask', 'max', 'mean', 'median', 'min', 'mod', 'mode', 'mul', 'multiply', 'name', 'nbytes', 'ndim', 'ne', 'nlargest', 'notna', 'notnull', 'nsmallest', 'nunique', 'pad', 'pipe', 'plot', 'pop', 'pow', 'prod', 'product', 'quantile', 'radd', 'rank', 'ravel', 'rdiv', 'rdivmod', 'reindex', 'rename', 'repeat', 'replace', 'resample', 'rfloordiv', 'rmod', 'rmul', 'rolling', 'round', 'rpow', 'rsub', 'rtruediv', 'sample', 'searchsorted', 'sem', 'shape', 'shift', 'size', 'skew', 'squeeze', 'std', 'struct', 'sub', 'subtract', 'sum', 'swapaxes', 'swaplevel', 'tail', 'take', 'transform', 'transpose', 'truediv', 'truncate', 'unique', 'unstack', 'update', 'values', 'var', 'view', 'where', 'xs']

The string representation of a Series displayed interactively shows the index on the left and the values ​​on the right. Since we didn't specify an index for the data, a default one consisting of the integers 0 to N - 1 (where N is the length of the data) is created. You can get the array representation and the index object of the Series through its values ​​and index attributes, respectively:

Another way to think of a Series is as a fixed-length ordered dict, since it is a mapping of index values ​​to data values. It can be used in many contexts where a dictionary could be used. If you have data contained in a Python dict, you can create a Series from it by passing the dict:

# Data type for pandas (class)  
type(series)  
  
# You can get the length too  
len(series)  
  
# Index of the series, can we use for loops here?  
series.index  
  
# The values on the series  
series.values  
  
# access by the index (as before)  
series[3]

40

When only one dict is passed, the resulting String index will have the keys of the dict in order. You can override this by passing the keys of the dict in the order you want them to appear in the resulting String:

# Let's create a random dictionary  
some\_data = {  
 "Ohio":4567,  
 "Texas": 5678,  
 "Oregon": 45678,  
 "Utah": 56789,  
 "something else": 43567  
}  
  
# Generate the type series  
my\_series = pd.Series(some\_data)  
my\_series

Ohio 4567  
Texas 5678  
Oregon 45678  
Utah 56789  
something else 43567  
dtype: int64

# Do you think key and index are the same?  
list(some\_data.keys()) == list(my\_series.index)

True

# Get the value  
my\_series.values  
  
# Get the index  
my\_series.index

Index(['Ohio', 'Texas', 'Oregon', 'Utah', 'something else'], dtype='object')

**Handling Missing Data**: When constructing a Series with provided data and a custom index, Pandas will align the data based on the index labels. Any missing values in the data corresponding to the index will be marked as NaN (not a number). In Pandas, NaN represents missing or undefined values.

For instance, let's consider creating a Series using predefined data and a custom index. In the following example, we have data for some states, but not all:

# Initial data  
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000}  
states = ['Ohio', 'Texas', 'Oregon', 'Utah', 'California']  
  
# Create a Series with provided data and a custom index  
my\_series = pd.Series(data=sdata, index=states)  
  
# The resulting Series contains the data for 'Ohio', 'Texas', and 'Oregon'  
# 'Utah' and 'California' are included in the index but have no data associated, hence they appear as NaN  
print(my\_series)  
  
print('\n')  
print(my\_series.values) # Displays the values of the Series  
print(my\_series.index) # Displays the index labels of the Series

Ohio 35000.0  
Texas 71000.0  
Oregon 16000.0  
Utah NaN  
California NaN  
dtype: float64  
  
  
[35000. 71000. 16000. nan nan]  
Index(['Ohio', 'Texas', 'Oregon', 'Utah', 'California'], dtype='object')

In this example, the Series my\_series is created with data from sdata and a custom index states. The values corresponding to 'Ohio', 'Texas', and 'Oregon' are placed accordingly. However, 'Utah' and 'California' are present in the index but have no associated data, leading to NaN values in the Series.

You can access the values and index of a Series using the values and index attributes.

When working with NaN values, it's important to handle them appropriately in your data analysis, as operations on NaN may result in unexpected outcomes. You can use functions like isna() or fillna() to detect and manage missing values in your Series.

print(my\_series.isna()) # Returns a boolean Series indicating NaN values  
my\_series.fillna(0, inplace=True) # Fills NaN values with a specified value (e.g., 0) in-place

Ohio False  
Texas False  
Oregon False  
Utah True  
California True  
dtype: bool

### Dataframes

In Pandas, a DataFrame is a two-dimensional, size-mutable, and potentially heterogeneous tabular data structure with labeled axes (rows and columns). It is often compared to a spreadsheet or SQL table, as it provides a convenient way to store and manipulate data.

**Understanding Dataframes**

* **Series**: Before diving into DataFrames, it's essential to understand the concept of Series. A Series is a one-dimensional array-like object that can hold various data types. DataFrames are essentially collections of Series objects, each representing a column.
* **Columns**: In a DataFrame, each Series represents a column. These columns can contain different types of data, such as integers, floats, or strings.
* **Rows**: Rows in a DataFrame are organized by index labels. Each row corresponds to a specific entry, and you can access rows using their index labels.

DataFrames are a powerful tool for data manipulation, analysis, and cleaning. They offer a structured way to work with data, making it easier to filter, sort, and compute statistics on datasets. We'll explore various DataFrame operations and functionalities in this guide.

#### From data types

In Pandas, you can create DataFrames from various data sources, including dictionaries, lists, CSV files, SQL databases, and more. One common way to create a DataFrame is from a dictionary, where each key represents a column name, and the associated value is a list or array of data for that column.

from dictionaries with lists as values

# Create a dictionary with columns and data  
dict\_states = {  
 "state": ["Oregon", "Utah", "New Mexico", "Nebraska"],  
 "year": [1900, 1898, 2000, 1900],  
 "something\_else": [456, "ssdsd", 0.023, np.nan]  
}  
  
# Create a DataFrame from the dictionary  
df = pd.DataFrame(dict\_states)  
df

state year something\_else  
0 Oregon 1900 456  
1 Utah 1898 ssdsd  
2 New Mexico 2000 0.023  
3 Nebraska 1900 NaN

In this example, we first import Pandas and NumPy libraries. Then, we create a dictionary dict\_states, where each key represents a column name, and the associated value is a list of data for that column. We pass this dictionary to pd.DataFrame() to create a DataFrame named df. Finally, we display the DataFrame.

When using Jupyter Notebook, Pandas DataFrame objects are displayed as more browser-friendly HTML tables, making it easier to view and explore the data interactively. You can find more options and customization details for DataFrame display in the [Pandas documentation](https://pandas.pydata.org/pandas-docs/stable/user_guide/options.html).

from list of dictionaries If you create a DataFrame from a list of dictionaries:

* Each dictionary within the list represents a row in the DataFrame.
* The keys within each dictionary become the column names of the DataFrame.
* All dictionaries in the list must have the same structure in terms of keys to ensure consistency.

Here are a few examples:

# Example 1: Creating a DataFrame with consistent keys in each dictionary  
dict\_states = {  
 "state": ["Oregon", "Utah", "New Mexico", "Nebraska"],  
 "year": [1900, 1898, 2000, 1900],  
 "something\_else": [456, "ssdsd", 0.023, np.nan]  
}  
  
list\_of\_dictionaries = [  
 {"state":["Oregon", "Utah", "New Mexico", "Nebraska"]}, # Each dictionary represents a row  
 {"year": [1900, 1898, 2000, 1900]}, # Corresponding values for the 'year' column  
 {"something\_else": [456, "ssdsd", 0.023, np.nan]}, # Corresponding values for the 'something\_else' column  
]  
  
# Creating a DataFrame from the list of dictionaries  
df = pd.DataFrame(list\_of\_dictionaries)  
df

state year \  
0 [Oregon, Utah, New Mexico, Nebraska] NaN   
1 NaN [1900, 1898, 2000, 1900]   
2 NaN NaN   
  
 something\_else   
0 NaN   
1 NaN   
2 [456, ssdsd, 0.023, nan]

# Example 2: Creating a DataFrame with varying dictionary structures  
list\_of\_dictionaries = [  
 {"state": "Oregon", "year": 1900, "something\_else": "oiuyghj"}, # Each dictionary represents a row  
 {"state": "Utah", "year": 1989, "something\_else": 678}, # Varying structures among dictionaries  
 {"state": "New Mexico", "year": 456, "something\_else": 87, "extra": 98765} # Extra key 'extra' in one dictionary  
]  
  
# Creating a DataFrame from the list of dictionaries  
df = pd.DataFrame(list\_of\_dictionaries)  
df

state year something\_else extra  
0 Oregon 1900 oiuyghj NaN  
1 Utah 1989 678 NaN  
2 New Mexico 456 87 98765.0

In the first example, we create a DataFrame df using a list of dictionaries list\_of\_dictionaries, where each dictionary represents a row with columns matching the keys. In the second example, we show that you can have variations in the structure of each dictionary as long as they have common keys.

For more details, you can refer to the [pandas documentation on from\_dict](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.from_dict.html).

#### From path

.csv

In this example, we load data into a DataFrame df using the pd.read\_csv() function. Loading data from CSV files is a common operation when working with Pandas, as it allows you to bring external data into a DataFrame for analysis and manipulation. The resulting DataFrame, df, contains the structured data from the CSV file, making it accessible for further exploration and analysis.

**Avocado Prices Dataset**: The "Avocado Prices" dataset, sourced from Kaggle, is a widely used dataset for data analysis and machine learning projects. It provides historical data on avocado prices and sales in various regions across the United States. This dataset is valuable for understanding trends in avocado pricing, sales volumes, and their relationship with different factors.

**Key Attributes**:

* **Columns**: The dataset includes several columns of information. Some of the key columns typically found in this dataset include:
  + **Date**: The date of observation.
  + **AveragePrice**: The average price of avocados.
  + **Total Volume**: Total volume of avocados sold.
  + **4046**: Volume of small Hass avocados sold.
  + **4225**: Volume of large Hass avocados sold.
  + **4770**: Volume of extra-large Hass avocados sold.
  + **Total Bags**: Total bags of avocados sold.
  + **Small Bags**: Bags of small avocados sold.
  + **Large Bags**: Bags of large avocados sold.
  + **XLarge Bags**: Bags of extra-large avocados sold.
  + **Type**: The type of avocados, often categorized as conventional or organic.
  + **Region**: The region or city within the United States where the data was recorded.
* **Date Range**: The dataset covers a range of dates, enabling time-series analysis. You can examine how avocado prices and sales change over different seasons and years.
* **Regions**: Information is provided for various regions or cities across the United States, allowing for the analysis of price and sales variations in different markets.
* **Types**: The dataset distinguishes between different types of avocados, such as conventional and organic, which can be useful for comparing price trends between these categories.
* **Volume**: Data on the total volume of avocados sold is available. This volume metric is often used to analyze market demand.
* **Average Price**: The dataset contains the average price of avocados, a fundamental metric for understanding price trends.

**Use Cases**:

* This dataset is commonly used for learning and practicing data analysis, data visualization, and regression modeling in data science and machine learning projects.
* It serves as a valuable resource for understanding how to work with real-world data, draw insights, and make data-driven decisions.

# load dataset  
df = pd.read\_csv("datasets/avocado\_kaggle.csv")  
  
# Show dataset  
df

Unnamed: 0 Date AveragePrice Total Volume 4046 4225 \  
0 0 2015-12-27 1.33 64236.62 1036.74 54454.85   
1 1 2015-12-20 1.35 54876.98 674.28 44638.81   
2 2 2015-12-13 0.93 118220.22 794.70 109149.67   
3 3 2015-12-06 1.08 78992.15 1132.00 71976.41   
4 4 2015-11-29 1.28 51039.60 941.48 43838.39   
... ... ... ... ... ... ...   
18244 7 2018-02-04 1.63 17074.83 2046.96 1529.20   
18245 8 2018-01-28 1.71 13888.04 1191.70 3431.50   
18246 9 2018-01-21 1.87 13766.76 1191.92 2452.79   
18247 10 2018-01-14 1.93 16205.22 1527.63 2981.04   
18248 11 2018-01-07 1.62 17489.58 2894.77 2356.13   
  
 4770 Total Bags Small Bags Large Bags XLarge Bags type \  
0 48.16 8696.87 8603.62 93.25 0.0 conventional   
1 58.33 9505.56 9408.07 97.49 0.0 conventional   
2 130.50 8145.35 8042.21 103.14 0.0 conventional   
3 72.58 5811.16 5677.40 133.76 0.0 conventional   
4 75.78 6183.95 5986.26 197.69 0.0 conventional   
... ... ... ... ... ... ...   
18244 0.00 13498.67 13066.82 431.85 0.0 organic   
18245 0.00 9264.84 8940.04 324.80 0.0 organic   
18246 727.94 9394.11 9351.80 42.31 0.0 organic   
18247 727.01 10969.54 10919.54 50.00 0.0 organic   
18248 224.53 12014.15 11988.14 26.01 0.0 organic   
  
 year region   
0 2015 Albany   
1 2015 Albany   
2 2015 Albany   
3 2015 Albany   
4 2015 Albany   
... ... ...   
18244 2018 WestTexNewMexico   
18245 2018 WestTexNewMexico   
18246 2018 WestTexNewMexico   
18247 2018 WestTexNewMexico   
18248 2018 WestTexNewMexico   
  
[18249 rows x 14 columns]

.xlsx, xls, xlsm, xlsb, odf, ods, odt

Pandas, a powerful data manipulation library in Python, provides robust functionality for reading and writing data to and from Excel files with various extensions. Excel files are commonly used to store structured data, making them a popular format for sharing and analyzing data in both business and research settings. (sometimes is too slow...)

Pandas simplifies the process of handling Excel files, allowing you to seamlessly integrate data from spreadsheets into your data analysis workflows. Whether you're dealing with classic .xls files, modern .xlsx workbooks, or other Excel-compatible formats like .xlsm, .xlsb, .odf, .ods, and .odt, Pandas offers versatile tools to import and export data.

**Key Features**:

* **Read Excel Data**: Pandas provides functions to read Excel data into DataFrames, preserving the structure and formatting of worksheets. You can efficiently read data from multiple sheets within a workbook.
* **Write Excel Data**: Pandas enables you to write DataFrames back to Excel files, allowing you to save your data along with any modifications or analyses you've performed.
* **Compatibility**: Pandas supports various Excel file extensions, including .xls, .xlsx, .xlsm, .xlsb, .odf, .ods, and .odt, ensuring compatibility with different Excel versions and formats.
* **Data Preservation**: When reading Excel files, Pandas retains data types, formulas, cell styles, and other attributes, ensuring data integrity.
* **Data Manipulation**: Once data is loaded into Pandas DataFrames, you can use Pandas' extensive data manipulation and analysis capabilities to explore, clean, transform, and visualize your data.

**Use Cases**:

* Data Extraction: Extract structured data from Excel files to perform analysis, reporting, or visualization.
* Data Integration: Combine data from multiple Excel sheets or workbooks into a single consolidated dataset.
* Data Export: Save the results of data analysis conducted with Pandas back into Excel files for sharing or further processing.
* Automation: Automate data extraction and manipulation tasks by incorporating Pandas into your data pipeline or workflow.

Pandas makes working with Excel files a breeze, allowing you to harness the power of Python for your data analysis projects while seamlessly interacting with Excel data.

To get started, explore Pandas' extensive documentation on [Excel File I/O](https://pandas.pydata.org/docs/reference/io.html#excel) to learn about the various methods and options available for reading and writing Excel files.

#!pip install openpyxl  
# Another dataset  
df\_from\_excel = pd.read\_excel("datasets/Online Retail.xlsx", engine="openpyxl", nrows=5)  
df\_from\_excel.head()

InvoiceNo InvoiceNo.1 InvoiceNo.2 \  
0 536365 2010-12-01 08:26:00 85123A   
1 536373 2010-12-01 09:02:00 85123A   
2 536375 2010-12-01 09:32:00 85123A   
3 536390 2010-12-01 10:19:00 85123A   
4 536394 2010-12-01 10:39:00 85123A   
  
 InvoiceNo.3 InvoiceNo.4 InvoiceNo.5 InvoiceNo.6 \  
0 CREAM HANGING HEART T-LIGHT HOLDER 6 2.55 15.3   
1 CREAM HANGING HEART T-LIGHT HOLDER 6 2.55 15.3   
2 CREAM HANGING HEART T-LIGHT HOLDER 6 2.55 15.3   
3 CREAM HANGING HEART T-LIGHT HOLDER 64 2.55 163.2   
4 CREAM HANGING HEART T-LIGHT HOLDER 32 2.55 81.6   
  
 InvoiceNo.7 InvoiceNo.8   
0 17850 United Kingdom   
1 17850 United Kingdom   
2 17850 United Kingdom   
3 17511 United Kingdom   
4 13408 United Kingdom

Reading different sheeets

# Import the Pandas library  
import pandas as pd  
  
# Define the path to the Excel file  
excel\_file\_path = "datasets/Online Retail.xlsx"  
  
# Reading the Default Tab (First One)  
# Use the read\_excel function to read the first sheet of the Excel file (default behavior)  
df\_default\_tab = pd.read\_excel(excel\_file\_path, engine="openpyxl", nrows=5)  
  
# Display the first 5 rows of the DataFrame  
df\_default\_tab.head()

InvoiceNo InvoiceNo.1 InvoiceNo.2 \  
0 536365 2010-12-01 08:26:00 85123A   
1 536373 2010-12-01 09:02:00 85123A   
2 536375 2010-12-01 09:32:00 85123A   
3 536390 2010-12-01 10:19:00 85123A   
4 536394 2010-12-01 10:39:00 85123A   
  
 InvoiceNo.3 InvoiceNo.4 InvoiceNo.5 InvoiceNo.6 \  
0 CREAM HANGING HEART T-LIGHT HOLDER 6 2.55 15.3   
1 CREAM HANGING HEART T-LIGHT HOLDER 6 2.55 15.3   
2 CREAM HANGING HEART T-LIGHT HOLDER 6 2.55 15.3   
3 CREAM HANGING HEART T-LIGHT HOLDER 64 2.55 163.2   
4 CREAM HANGING HEART T-LIGHT HOLDER 32 2.55 81.6   
  
 InvoiceNo.7 InvoiceNo.8   
0 17850 United Kingdom   
1 17850 United Kingdom   
2 17850 United Kingdom   
3 17511 United Kingdom   
4 13408 United Kingdom

# Reading Another Tab (e.g., "new\_tab")  
# Specify the sheet name as a string to read a specific sheet from the Excel file  
df\_new\_tab = pd.read\_excel(excel\_file\_path, engine="openpyxl", sheet\_name="new\_tab", nrows=5)  
  
# Display the first 5 rows of the DataFrame from the "new\_tab" sheet  
df\_new\_tab.head()

column1 column2  
0 example hello

#### From databases

sql: [docs](https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html)

from sqlite3 import connect  
  
conn = connect(':memory:')  
df = pd.read\_sql('SELECT column\_1, column\_2 FROM sample\_data', conn)  
  
df.to\_sql('test\_data', conn)

mongodb

import pymongo  
from pymongo import MongoClient  
  
client = MongoClient()  
db = client.database\_name  
collection = db.collection\_name  
data = pd.DataFrame(list(collection.find()))

## Exploratory analysis of a dataframe

In this section, we'll dive into the world of exploratory data analysis (EDA) using Pandas. EDA is a crucial step in the data analysis process, where we get to know our data, understand its characteristics, and uncover initial insights. We'll use a DataFrame, df, loaded from the "Advertising.csv" dataset as an example to perform various exploratory analyses. Let's begin by loading the dataset and getting a glimpse of its contents.

# load the dataset  
df = pd.read\_csv("datasets/Advertising.csv")  
  
# Check first rows  
df.head()

TV Radio Newspaper Sales  
0 230.1 37.8 69.2 22.1  
1 44.5 39.3 45.1 10.4  
2 17.2 45.9 69.3 9.3  
3 151.5 41.3 58.5 18.5  
4 180.8 10.8 58.4 12.9

### Metadata and information

shape, columns, dtypes, info, describe

When working with data in a DataFrame, it's crucial to understand the basic characteristics and structure of the dataset. To gain insights into the data, we can retrieve meta information about the DataFrame. This information includes details such as the shape of the DataFrame, column names, data types, general information, and a statistical summary of the data.

In this section, we'll explore how to use various Pandas functions to obtain essential meta information about your DataFrame. This knowledge will help you better understand and prepare your data for analysis and visualization.

# Shape of the DataFrame (rows, columns)  
data\_shape = df.shape  
print(f"\nShape of the DataFrame: {data\_shape}")  
  
# Column names  
column\_names = df.columns  
print(f"\nColumn Names: {column\_names}")  
  
# Data types of each column  
data\_types = df.dtypes  
print(f"\nData Types:\n{data\_types}")

Shape of the DataFrame: (200, 4)  
  
Column Names: Index(['TV', 'Radio', 'Newspaper', 'Sales'], dtype='object')  
  
Data Types:  
TV float64  
Radio float64  
Newspaper float64  
Sales float64  
dtype: object

# General Information about the DataFrame  
data\_info = df.info()

<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 200 entries, 0 to 199  
Data columns (total 4 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 TV 200 non-null float64  
 1 Radio 200 non-null float64  
 2 Newspaper 200 non-null float64  
 3 Sales 200 non-null float64  
dtypes: float64(4)  
memory usage: 6.4 KB

# Statistical Summary  
data\_summary = df.describe()  
print("\nStatistical Summary:")  
print(data\_summary)

Statistical Summary:  
 TV Radio Newspaper Sales  
count 200.000000 200.000000 200.000000 200.000000  
mean 147.042500 23.264000 30.554000 14.022500  
std 85.854236 14.846809 21.778621 5.217457  
min 0.700000 0.000000 0.300000 1.600000  
25% 74.375000 9.975000 12.750000 10.375000  
50% 149.750000 22.900000 25.750000 12.900000  
75% 218.825000 36.525000 45.100000 17.400000  
max 296.400000 49.600000 114.000000 27.000000

### Previsualization

Before diving into in-depth analysis and manipulation of your dataset, it's often helpful to get a quick glimpse of what the data looks like. Pandas provides two useful methods, head() and tail(), to help you previsualize the beginning and end of your DataFrame.

In this section, we'll explore how to use these methods to display a subset of your data, making it easier to understand the structure and content of your DataFrame. These simple yet powerful tools are the first step in getting acquainted with your data, allowing you to identify any immediate patterns or issues.

head and tail

df.head()  
  
df.tail()

TV Radio Newspaper Sales  
195 38.2 3.7 13.8 7.6  
196 94.2 4.9 8.1 9.7  
197 177.0 9.3 6.4 12.8  
198 283.6 42.0 66.2 25.5  
199 232.1 8.6 8.7 13.4

By default head shows me the first 5 rows, I can see some more or less by passing a number as a parameter

### Sort and organize a dataframe

Organizing and sorting your data is a fundamental part of data analysis. In this section, we'll explore how to order a DataFrame using the Pandas library. By sorting data, you can gain valuable insights, identify trends, and make your data more accessible for analysis.

We'll cover various scenarios, such as sorting by one or more columns, in ascending or descending order, and selecting specific columns to view. Understanding how to arrange your data effectively can significantly enhance your ability to extract meaningful information from it.

Let's dive into the different ways to order and arrange your data using Pandas.

# Load the DataFrame from a CSV file  
df = pd.read\_csv("datasets/avocado\_kaggle.csv")  
  
# Sorting by a single column in descending order (most recent year first)  
df.sort\_values(by="year", ascending=False)

Unnamed: 0 Date AveragePrice Total Volume 4046 \  
9124 10 2018-01-14 0.90 950954.60 463945.73   
8883 9 2018-01-21 0.78 1315329.83 613600.56   
8905 7 2018-02-04 0.91 3316494.76 1609104.94   
8906 8 2018-01-28 0.99 2533937.56 1387712.40   
8907 9 2018-01-21 1.18 2208596.12 1051836.46   
... ... ... ... ... ...   
10278 8 2015-11-01 1.36 52815.97 13987.25   
10279 9 2015-10-25 1.41 48799.38 13800.24   
10280 10 2015-10-18 1.62 33737.71 10370.68   
10281 11 2015-10-11 1.72 28062.63 8093.67   
0 0 2015-12-27 1.33 64236.62 1036.74   
  
 4225 4770 Total Bags Small Bags Large Bags XLarge Bags \  
9124 188126.02 11227.47 287655.38 125408.69 162040.02 206.67   
8883 246703.53 13332.26 441693.48 210780.39 226079.75 4833.34   
8905 827998.81 9830.10 869560.91 678696.23 190573.14 291.54   
8906 515698.93 6371.30 624154.93 474701.92 149437.52 15.49   
8907 479599.90 7659.28 669500.48 562989.08 106461.22 50.18   
... ... ... ... ... ... ...   
10278 20741.07 0.00 18087.65 17485.60 602.05 0.00   
10279 21060.95 0.00 13938.19 12992.81 945.38 0.00   
10280 16004.06 3.18 7359.79 6393.48 966.31 0.00   
10281 14759.53 3.18 5206.25 5146.25 60.00 0.00   
0 54454.85 48.16 8696.87 8603.62 93.25 0.00   
  
 type year region   
9124 conventional 2018 WestTexNewMexico   
8883 conventional 2018 PhoenixTucson   
8905 conventional 2018 Plains   
8906 conventional 2018 Plains   
8907 conventional 2018 Plains   
... ... ... ...   
10278 organic 2015 LosAngeles   
10279 organic 2015 LosAngeles   
10280 organic 2015 LosAngeles   
10281 organic 2015 LosAngeles   
0 conventional 2015 Albany   
  
[18249 rows x 14 columns]

# Sorting by multiple columns in descending order (year and region)  
df.sort\_values(by=["year", "region"], ascending=False)

Unnamed: 0 Date AveragePrice Total Volume 4046 \  
9114 0 2018-03-25 0.84 965185.06 438526.12   
9115 1 2018-03-18 0.88 855251.17 457635.79   
9116 2 2018-03-11 0.94 897607.12 467501.55   
9117 3 2018-03-04 0.88 935934.10 454269.43   
9118 4 2018-02-25 0.88 895671.55 431217.01   
... ... ... ... ... ...   
9173 47 2015-02-01 1.83 1228.51 33.12   
9174 48 2015-01-25 1.89 1115.89 14.87   
9175 49 2015-01-18 1.93 1118.47 8.02   
9176 50 2015-01-11 1.77 1182.56 39.00   
9177 51 2015-01-04 1.79 1373.95 57.42   
  
 4225 4770 Total Bags Small Bags Large Bags XLarge Bags \  
9114 199585.90 11017.42 316055.62 153009.89 160999.10 2046.63   
9115 137597.04 8422.08 251596.26 151191.85 98535.60 1868.81   
9116 154130.63 11380.26 264594.68 152380.60 110322.16 1891.92   
9117 164856.57 9907.85 306900.25 164965.35 138399.68 3535.22   
9118 171532.79 10590.75 282331.00 125973.31 147040.26 9317.43   
... ... ... ... ... ... ...   
9173 99.36 0.00 1096.03 1096.03 0.00 0.00   
9174 148.72 0.00 952.30 952.30 0.00 0.00   
9175 178.78 0.00 931.67 931.67 0.00 0.00   
9176 305.12 0.00 838.44 838.44 0.00 0.00   
9177 153.88 0.00 1162.65 1162.65 0.00 0.00   
  
 type year region   
9114 conventional 2018 WestTexNewMexico   
9115 conventional 2018 WestTexNewMexico   
9116 conventional 2018 WestTexNewMexico   
9117 conventional 2018 WestTexNewMexico   
9118 conventional 2018 WestTexNewMexico   
... ... ... ...   
9173 organic 2015 Albany   
9174 organic 2015 Albany   
9175 organic 2015 Albany   
9176 organic 2015 Albany   
9177 organic 2015 Albany   
  
[18249 rows x 14 columns]

# Sorting by a single column in descending order (region)  
df.sort\_values(by="region", ascending=False)

Unnamed: 0 Date AveragePrice Total Volume 4046 \  
9124 10 2018-01-14 0.90 950954.60 463945.73   
8426 1 2017-12-24 0.93 769970.84 402476.91   
8428 3 2017-12-10 0.86 751137.79 384349.01   
8429 4 2017-12-03 0.85 799879.00 359419.00   
8430 5 2017-11-26 0.91 623094.00 354268.00   
... ... ... ... ... ...   
9174 48 2015-01-25 1.89 1115.89 14.87   
9175 49 2015-01-18 1.93 1118.47 8.02   
9176 50 2015-01-11 1.77 1182.56 39.00   
9177 51 2015-01-04 1.79 1373.95 57.42   
0 0 2015-12-27 1.33 64236.62 1036.74   
  
 4225 4770 Total Bags Small Bags Large Bags XLarge Bags \  
9124 188126.02 11227.47 287655.38 125408.69 162040.02 206.67   
8426 150206.86 7966.08 209320.99 96489.71 112764.62 66.66   
8428 133201.98 5774.97 227811.83 104486.24 123222.25 103.34   
8429 224516.00 6690.00 209254.00 70520.00 138586.00 148.00   
8430 104956.00 5465.00 158405.00 74684.00 83606.00 114.00   
... ... ... ... ... ... ...   
9174 148.72 0.00 952.30 952.30 0.00 0.00   
9175 178.78 0.00 931.67 931.67 0.00 0.00   
9176 305.12 0.00 838.44 838.44 0.00 0.00   
9177 153.88 0.00 1162.65 1162.65 0.00 0.00   
0 54454.85 48.16 8696.87 8603.62 93.25 0.00   
  
 type year region   
9124 conventional 2018 WestTexNewMexico   
8426 conventional 2017 WestTexNewMexico   
8428 conventional 2017 WestTexNewMexico   
8429 conventional 2017 WestTexNewMexico   
8430 conventional 2017 WestTexNewMexico   
... ... ... ...   
9174 organic 2015 Albany   
9175 organic 2015 Albany   
9176 organic 2015 Albany   
9177 organic 2015 Albany   
0 conventional 2015 Albany   
  
[18249 rows x 14 columns]

# Sorting by a single column in descending order (year)  
df.sort\_values(by="year", ascending=False)

Unnamed: 0 Date AveragePrice Total Volume 4046 \  
9124 10 2018-01-14 0.90 950954.60 463945.73   
8883 9 2018-01-21 0.78 1315329.83 613600.56   
8905 7 2018-02-04 0.91 3316494.76 1609104.94   
8906 8 2018-01-28 0.99 2533937.56 1387712.40   
8907 9 2018-01-21 1.18 2208596.12 1051836.46   
... ... ... ... ... ...   
10278 8 2015-11-01 1.36 52815.97 13987.25   
10279 9 2015-10-25 1.41 48799.38 13800.24   
10280 10 2015-10-18 1.62 33737.71 10370.68   
10281 11 2015-10-11 1.72 28062.63 8093.67   
0 0 2015-12-27 1.33 64236.62 1036.74   
  
 4225 4770 Total Bags Small Bags Large Bags XLarge Bags \  
9124 188126.02 11227.47 287655.38 125408.69 162040.02 206.67   
8883 246703.53 13332.26 441693.48 210780.39 226079.75 4833.34   
8905 827998.81 9830.10 869560.91 678696.23 190573.14 291.54   
8906 515698.93 6371.30 624154.93 474701.92 149437.52 15.49   
8907 479599.90 7659.28 669500.48 562989.08 106461.22 50.18   
... ... ... ... ... ... ...   
10278 20741.07 0.00 18087.65 17485.60 602.05 0.00   
10279 21060.95 0.00 13938.19 12992.81 945.38 0.00   
10280 16004.06 3.18 7359.79 6393.48 966.31 0.00   
10281 14759.53 3.18 5206.25 5146.25 60.00 0.00   
0 54454.85 48.16 8696.87 8603.62 93.25 0.00   
  
 type year region   
9124 conventional 2018 WestTexNewMexico   
8883 conventional 2018 PhoenixTucson   
8905 conventional 2018 Plains   
8906 conventional 2018 Plains   
8907 conventional 2018 Plains   
... ... ... ...   
10278 organic 2015 LosAngeles   
10279 organic 2015 LosAngeles   
10280 organic 2015 LosAngeles   
10281 organic 2015 LosAngeles   
0 conventional 2015 Albany   
  
[18249 rows x 14 columns]

# Selecting specific columns (year and region) after sorting  
df.sort\_values(by="year", ascending=False)[["year", "region"]]

year region  
9124 2018 WestTexNewMexico  
8883 2018 PhoenixTucson  
8905 2018 Plains  
8906 2018 Plains  
8907 2018 Plains  
... ... ...  
10278 2015 LosAngeles  
10279 2015 LosAngeles  
10280 2015 LosAngeles  
10281 2015 LosAngeles  
0 2015 Albany  
  
[18249 rows x 2 columns]

## Access the df's columns  
print(df.columns)  
# Creating a list of column names to create a subset of columns  
subset\_columns = list(df.columns)[1:4]  
  
# Selecting a subset of columns using the list of column names  
df[subset\_columns]

Index(['Unnamed: 0', 'Date', 'AveragePrice', 'Total Volume', '4046', '4225',  
 '4770', 'Total Bags', 'Small Bags', 'Large Bags', 'XLarge Bags', 'type',  
 'year', 'region'],  
 dtype='object')

Date AveragePrice Total Volume  
0 2015-12-27 1.33 64236.62  
1 2015-12-20 1.35 54876.98  
2 2015-12-13 0.93 118220.22  
3 2015-12-06 1.08 78992.15  
4 2015-11-29 1.28 51039.60  
... ... ... ...  
18244 2018-02-04 1.63 17074.83  
18245 2018-01-28 1.71 13888.04  
18246 2018-01-21 1.87 13766.76  
18247 2018-01-14 1.93 16205.22  
18248 2018-01-07 1.62 17489.58  
  
[18249 rows x 3 columns]

sample

In data analysis, it's often essential to work with a representative subset of your dataset for various purposes, such as data exploration, testing, or model training. Pandas provides the sample method to facilitate random sampling of rows from a DataFrame. This method allows you to obtain random rows or a specific fraction of your data, making it a valuable tool for statistical analysis and machine learning tasks. In this section, we'll explore how to use the sample method to extract random samples from a DataFrame and understand its various options and applications.

# Sampling a random single row from the DataFrame  
df.sample()

Unnamed: 0 Date AveragePrice Total Volume 4046 4225 \  
17100 27 2017-06-25 1.67 17275.62 191.83 5205.01   
  
 4770 Total Bags Small Bags Large Bags XLarge Bags type year \  
17100 117.4 11761.38 9122.35 2639.03 0.0 organic 2017   
  
 region   
17100 SouthCarolina

# Sampling a random fraction (20%) of rows from the DataFrame  
df.sample(frac=0.2)

Unnamed: 0 Date AveragePrice Total Volume 4046 \  
5216 16 2016-09-04 0.96 4804279.06 2776884.75   
4085 29 2016-06-05 0.99 786071.68 474319.60   
2564 16 2015-09-06 1.36 48358.73 675.27   
17557 7 2017-11-12 1.80 15670.44 1743.38   
7851 9 2017-10-29 1.56 716567.05 197107.52   
... ... ... ... ... ...   
4785 1 2016-12-18 1.15 174209.10 64259.62   
6039 52 2017-01-01 1.21 217051.50 47765.32   
15581 45 2017-02-19 0.90 226000.27 4432.89   
6584 14 2017-09-24 1.25 949769.71 470511.08   
16005 45 2017-02-19 0.56 10571.30 0.00   
  
 4225 4770 Total Bags Small Bags Large Bags XLarge Bags \  
5216 664439.79 12528.81 1350425.71 771445.27 574732.66 4247.78   
4085 149114.65 252.33 162385.10 56515.95 98746.65 7122.50   
2564 37504.92 97.12 10081.42 10081.42 0.00 0.00   
17557 2795.92 0.00 11131.14 10827.23 303.91 0.00   
7851 428002.14 4543.85 86913.54 85850.59 44.00 1018.95   
... ... ... ... ... ... ...   
4785 54299.21 2892.26 52758.01 51586.60 1171.41 0.00   
6039 94571.69 15036.44 59678.05 45920.26 13711.12 46.67   
15581 66308.19 0.00 155259.19 29892.25 125366.94 0.00   
6584 145808.34 1812.37 331637.92 262287.19 69350.73 0.00   
16005 1277.78 0.00 9293.52 287.77 9005.75 0.00   
  
 type year region   
5216 conventional 2016 Southeast   
4085 conventional 2016 MiamiFtLauderdale   
2564 conventional 2015 Syracuse   
17557 organic 2017 WestTexNewMexico   
7851 conventional 2017 SanFrancisco   
... ... ... ...   
4785 conventional 2016 RichmondNorfolk   
6039 conventional 2017 Charlotte   
15581 organic 2017 GreatLakes   
6584 conventional 2017 Houston   
16005 organic 2017 Louisville   
  
[3650 rows x 14 columns]

display When working with Jupyter Notebook or JupyterLab, you can use the display function to render Pandas DataFrames in a more visually appealing and interactive format. While Pandas' default tabular display is informative, the display function provides additional flexibility and customization options.

By using display, you can take advantage of the enhanced table formatting capabilities of Jupyter environments, including sortable columns, responsive design, and improved visual representation of your data. It's especially useful when dealing with larger datasets or when you want to present your data in a cleaner and more interactive way for data exploration or reporting.

# Display the DataFrame using the display function (equivalent to print)  
display(df)

Unnamed: 0 Date AveragePrice Total Volume 4046 4225 \  
0 0 2015-12-27 1.33 64236.62 1036.74 54454.85   
1 1 2015-12-20 1.35 54876.98 674.28 44638.81   
2 2 2015-12-13 0.93 118220.22 794.70 109149.67   
3 3 2015-12-06 1.08 78992.15 1132.00 71976.41   
4 4 2015-11-29 1.28 51039.60 941.48 43838.39   
... ... ... ... ... ... ...   
18244 7 2018-02-04 1.63 17074.83 2046.96 1529.20   
18245 8 2018-01-28 1.71 13888.04 1191.70 3431.50   
18246 9 2018-01-21 1.87 13766.76 1191.92 2452.79   
18247 10 2018-01-14 1.93 16205.22 1527.63 2981.04   
18248 11 2018-01-07 1.62 17489.58 2894.77 2356.13   
  
 4770 Total Bags Small Bags Large Bags XLarge Bags type \  
0 48.16 8696.87 8603.62 93.25 0.0 conventional   
1 58.33 9505.56 9408.07 97.49 0.0 conventional   
2 130.50 8145.35 8042.21 103.14 0.0 conventional   
3 72.58 5811.16 5677.40 133.76 0.0 conventional   
4 75.78 6183.95 5986.26 197.69 0.0 conventional   
... ... ... ... ... ... ...   
18244 0.00 13498.67 13066.82 431.85 0.0 organic   
18245 0.00 9264.84 8940.04 324.80 0.0 organic   
18246 727.94 9394.11 9351.80 42.31 0.0 organic   
18247 727.01 10969.54 10919.54 50.00 0.0 organic   
18248 224.53 12014.15 11988.14 26.01 0.0 organic   
  
 year region   
0 2015 Albany   
1 2015 Albany   
2 2015 Albany   
3 2015 Albany   
4 2015 Albany   
... ... ...   
18244 2018 WestTexNewMexico   
18245 2018 WestTexNewMexico   
18246 2018 WestTexNewMexico   
18247 2018 WestTexNewMexico   
18248 2018 WestTexNewMexico   
  
[18249 rows x 14 columns]

### NaN values

In data analysis, missing data is a common occurrence and can significantly impact the accuracy and reliability of your analyses. One way to represent missing data in Pandas and many other data analysis libraries is by using the special value "NaN," which stands for "Not A Number."

NaN is essentially a placeholder for missing or undefined data points, and it is typically treated as a floating-point value. This allows Pandas to work with missing data while maintaining data types within a DataFrame.

Handling NaN values is a crucial aspect of data cleaning and preprocessing, as it can affect statistical calculations, visualizations, and machine learning models. In this section, we'll explore various techniques and functions in Pandas for dealing with missing data and ensuring that your data analysis yields accurate and meaningful results.

# Check the dataset info  
df.info()  
  
pd.isnull(df)

<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 18249 entries, 0 to 18248  
Data columns (total 14 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 Unnamed: 0 18249 non-null int64   
 1 Date 18249 non-null object   
 2 AveragePrice 18249 non-null float64  
 3 Total Volume 18249 non-null float64  
 4 4046 18249 non-null float64  
 5 4225 18249 non-null float64  
 6 4770 18249 non-null float64  
 7 Total Bags 18249 non-null float64  
 8 Small Bags 18249 non-null float64  
 9 Large Bags 18249 non-null float64  
 10 XLarge Bags 18249 non-null float64  
 11 type 18249 non-null object   
 12 year 18249 non-null int64   
 13 region 18249 non-null object   
dtypes: float64(9), int64(2), object(3)  
memory usage: 1.9+ MB

Unnamed: 0 Date AveragePrice Total Volume 4046 4225 4770 \  
0 False False False False False False False   
1 False False False False False False False   
2 False False False False False False False   
3 False False False False False False False   
4 False False False False False False False   
... ... ... ... ... ... ... ...   
18244 False False False False False False False   
18245 False False False False False False False   
18246 False False False False False False False   
18247 False False False False False False False   
18248 False False False False False False False   
  
 Total Bags Small Bags Large Bags XLarge Bags type year region   
0 False False False False False False False   
1 False False False False False False False   
2 False False False False False False False   
3 False False False False False False False   
4 False False False False False False False   
... ... ... ... ... ... ... ...   
18244 False False False False False False False   
18245 False False False False False False False   
18246 False False False False False False False   
18247 False False False False False False False   
18248 False False False False False False False   
  
[18249 rows x 14 columns]

# Check for missing values in the DataFrame  
missing\_values = pd.isnull(df)  
  
# Count missing values in each column  
missing\_counts = missing\_values.sum()  
  
# Count columns with missing values  
columns\_with\_missing = missing\_counts[missing\_counts > 0].count()  
  
# Check if all columns have missing values  
all\_columns\_missing = missing\_counts.all()  
  
# Calculate the total number of missing values  
total\_missing\_values = missing\_counts.sum()  
  
# Display the results  
print("Missing Values in Each Column:\n", missing\_counts)  
print("\nNumber of Columns with Missing Values:", columns\_with\_missing)  
print("All Columns Have Missing Values:", all\_columns\_missing)  
print("\nTotal Missing Values in the DataFrame:", total\_missing\_values)

Missing Values in Each Column:  
 Unnamed: 0 0  
Date 0  
AveragePrice 0  
Total Volume 0  
4046 0  
4225 0  
4770 0  
Total Bags 0  
Small Bags 0  
Large Bags 0  
XLarge Bags 0  
type 0  
year 0  
region 0  
dtype: int64  
  
Number of Columns with Missing Values: 0  
All Columns Have Missing Values: False  
  
Total Missing Values in the DataFrame: 0

## Data Manipulation

### Index & Columns

What does an index mean?

What properties must they obey?

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Index.html>

import numpy as np  
  
# Lets take a step back and see what they are made of  
#### DATAFRAMES ####  
  
# Define a df  
myarray = np.random.random((10,5))  
print(myarray)  
  
a = pd.DataFrame(myarray)  
#a = pd.DataFrame([['joao',2,3],['Melissa',8,7],['CR7',9,5]])  
  
# Define it's column names  
a.columns = ["first","second","third","fourth","fifth"]  
  
# Same with rows  
print(list(a.index))  
a.head()  
  
  
#### SERIES ####  
# Dataframes are actually a 2D Array of SERIES  
a = np.random.random(3)  
myseries = pd.Series(a)  
  
# An index in a pandas dataframe/dataseries a unique key that completely identifies a row (in a distinct way)  
myseries  
  
# Two fundamental objects in pandas  
  
#series  
a = pd.Series(np.random.random(20))  
  
#dataframes  
a = pd.DataFrame(np.random.random((3,5)))

[[0.43353257 0.2573954 0.02288007 0.3089654 0.51594592]  
 [0.48870204 0.93742799 0.66144714 0.14893895 0.59236944]  
 [0.07378025 0.75708394 0.85239105 0.08708699 0.47942168]  
 [0.27962889 0.50473708 0.59414165 0.72464508 0.33448434]  
 [0.20707984 0.23936101 0.14698872 0.61358973 0.5181471 ]  
 [0.90651441 0.72964431 0.67606184 0.71140239 0.56583905]  
 [0.26775285 0.22988121 0.91664434 0.95113142 0.19971368]  
 [0.97457099 0.04673149 0.92673728 0.06078074 0.58368429]  
 [0.34594432 0.73481806 0.59516902 0.81976896 0.66207883]  
 [0.74299487 0.63727392 0.69116209 0.2928931 0.18260807]]  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Columns  
print(a.columns)  
print(list(a.columns))  
  
#what is i want to change all column names?  
a.columns = ['fourth','second','third','first','fifth']  
a.head()

RangeIndex(start=0, stop=5, step=1)  
[0, 1, 2, 3, 4]

fourth second third first fifth  
0 0.607726 0.519731 0.824067 0.951831 0.876547  
1 0.751746 0.698252 0.329275 0.462447 0.839133  
2 0.711967 0.792454 0.810246 0.908971 0.367002

# Change some columns names  
a = a.rename(columns = {'first': 'FIRST'})  
a  
a = a.rename(columns = {'FIRST': 'first'})  
a

fourth second third first fifth  
0 0.607726 0.519731 0.824067 0.951831 0.876547  
1 0.751746 0.698252 0.329275 0.462447 0.839133  
2 0.711967 0.792454 0.810246 0.908971 0.367002

# Re Order the columns  
#a.columns = ['first','second','third','fourth','fifth']  
  
#or  
  
column\_order = ['first','second','third','fourth','fifth']  
a = a[column\_order]  
  
a

first second third fourth fifth  
0 0.951831 0.519731 0.824067 0.607726 0.876547  
1 0.462447 0.698252 0.329275 0.751746 0.839133  
2 0.908971 0.792454 0.810246 0.711967 0.367002

# INDEX  
# Explain what index is  
  
a.index  
list(a.index)

[0, 1, 2]

# INDEX: imutable -> change all or none  
a.index = ['first-row', 'second-row', 'third-row']  
#a.index[1] = '2nd line'  
  
# See how index is imutable -> Be carefull here: you can change the indexes, if you change it all at once  
a

first second third fourth fifth  
first-row 0.951831 0.519731 0.824067 0.607726 0.876547  
second-row 0.462447 0.698252 0.329275 0.751746 0.839133  
third-row 0.908971 0.792454 0.810246 0.711967 0.367002

# INDEX: typical assignements  
#what types of indices are there?  
  
a.index = ['melissa', 'joao', 'cr7']  
# By order of importance in the class  
a  
  
import datetime as dt  
  
#new\_index = pd.date\_range(end = dt.date.today(), periods = 10,)  
new\_index = pd.date\_range(end = dt.date.today(), periods = 3,)  
new\_index  
  
a.index = new\_index  
a

first second third fourth fifth  
2024-09-13 0.951831 0.519731 0.824067 0.607726 0.876547  
2024-09-14 0.462447 0.698252 0.329275 0.751746 0.839133  
2024-09-15 0.908971 0.792454 0.810246 0.711967 0.367002

### LOC & ILOC

iloc

#### LOC & ILOC ####  
# Slicing dataframes will be something very frequent  
  
## INDEXING  
#indexing dataframes  
a['first'][0:2]  
  
#indexing dataframes  
print(a[:][1:2])

first second third fourth fifth  
2024-09-14 0.462447 0.698252 0.329275 0.751746 0.839133

# Lets increase the size of a  
a = pd.DataFrame(np.random.random((100,5)))  
a.columns = ["first","second","third","fourth","fifth"]  
  
# Lets not use List Comprehension, as the legilibility is also important  
index\_list = []  
for i in range(100):  
 index\_list.append('row '+str(i))  
  
a.index = index\_list  
a.head(20)

first second third fourth fifth  
row 0 0.074721 0.752058 0.300459 0.993859 0.725156  
row 1 0.072121 0.789893 0.284558 0.879853 0.197394  
row 2 0.220153 0.245554 0.141805 0.788514 0.888601  
row 3 0.706425 0.028381 0.511108 0.627935 0.511819  
row 4 0.376100 0.854190 0.095048 0.623959 0.766986  
row 5 0.125241 0.105692 0.812553 0.680620 0.881675  
row 6 0.989110 0.162407 0.840786 0.749083 0.975522  
row 7 0.494970 0.478653 0.537667 0.354120 0.678645  
row 8 0.140798 0.890224 0.898581 0.482208 0.855371  
row 9 0.917860 0.514851 0.997214 0.659416 0.292319  
row 10 0.483697 0.598249 0.479843 0.403859 0.824221  
row 11 0.213738 0.709171 0.427798 0.934839 0.983904  
row 12 0.240587 0.498229 0.388655 0.345337 0.376929  
row 13 0.629541 0.897132 0.971506 0.232291 0.570331  
row 14 0.880729 0.374034 0.303513 0.199294 0.232371  
row 15 0.419881 0.844819 0.844841 0.175124 0.173865  
row 16 0.986087 0.272481 0.185979 0.432591 0.700952  
row 17 0.617134 0.486061 0.197791 0.417290 0.771816  
row 18 0.289666 0.554600 0.825303 0.090145 0.417987  
row 19 0.712314 0.887088 0.351095 0.207521 0.602982

## THIS IS VERY IMPORTANT  
  
## LOC  
# or using the locate method  
  
a.loc['row 0'] # -> This returns a series  
#a.loc['row 0','row 1'] # What does this return? an error  
a.loc[['row 0','row 1']]  
  
## ILOC  
# or using the index-locate method  
# Same as Lists: from a[x:y], will return from a[x] including, to a[y] excluding  
  
#a.iloc[0:2]  
a.iloc[[x for x in range (50) if x%2==0]]

first second third fourth fifth  
row 0 0.074721 0.752058 0.300459 0.993859 0.725156  
row 2 0.220153 0.245554 0.141805 0.788514 0.888601  
row 4 0.376100 0.854190 0.095048 0.623959 0.766986  
row 6 0.989110 0.162407 0.840786 0.749083 0.975522  
row 8 0.140798 0.890224 0.898581 0.482208 0.855371  
row 10 0.483697 0.598249 0.479843 0.403859 0.824221  
row 12 0.240587 0.498229 0.388655 0.345337 0.376929  
row 14 0.880729 0.374034 0.303513 0.199294 0.232371  
row 16 0.986087 0.272481 0.185979 0.432591 0.700952  
row 18 0.289666 0.554600 0.825303 0.090145 0.417987  
row 20 0.983036 0.505713 0.439537 0.020183 0.219120  
row 22 0.506787 0.288277 0.921966 0.776790 0.609087  
row 24 0.593648 0.170077 0.389784 0.272424 0.899756  
row 26 0.240759 0.437778 0.630988 0.784724 0.435176  
row 28 0.768368 0.177000 0.297286 0.854256 0.729827  
row 30 0.724361 0.212551 0.359651 0.138772 0.529116  
row 32 0.875871 0.523875 0.343060 0.047918 0.827851  
row 34 0.737511 0.850978 0.179975 0.086165 0.832980  
row 36 0.701504 0.304384 0.087779 0.448407 0.230118  
row 38 0.299115 0.564502 0.172536 0.719585 0.821128  
row 40 0.913396 0.488864 0.493516 0.219289 0.821193  
row 42 0.636435 0.768368 0.275599 0.442125 0.732907  
row 44 0.265226 0.371440 0.750245 0.025511 0.413761  
row 46 0.080262 0.230308 0.176187 0.765657 0.019742  
row 48 0.797970 0.522980 0.095253 0.972341 0.352232

### Filter by Line

#### FILTER ####  
# Filter by Categorical data  
  
### OVERVIEW ON DATAFRAMES ###  
# Dont copy for now - I will repeat all this  
# pip install xlrd  
import pandas as pd  
  
orders = pd.read\_excel("datasets/Sample - Superstore.xls")  
display(orders.head())  
  
condition = orders['Category'] == "Office Supplies"  
condition  
#orders[condition]

Row ID Order ID Order Date Ship Date Ship Mode Customer ID \  
0 1 CA-2018-152156 2018-11-08 2018-11-11 Second Class CG-12520   
1 2 CA-2018-152156 2018-11-08 2018-11-11 Second Class CG-12520   
2 3 CA-2018-138688 2018-06-12 2018-06-16 Second Class DV-13045   
3 4 US-2017-108966 2017-10-11 2017-10-18 Standard Class SO-20335   
4 5 US-2017-108966 2017-10-11 2017-10-18 Standard Class SO-20335   
  
 Customer Name Segment Country/Region City ... \  
0 Claire Gute Consumer United States Henderson ...   
1 Claire Gute Consumer United States Henderson ...   
2 Darrin Van Huff Corporate United States Los Angeles ...   
3 Sean O'Donnell Consumer United States Fort Lauderdale ...   
4 Sean O'Donnell Consumer United States Fort Lauderdale ...   
  
 Postal Code Region Product ID Category Sub-Category \  
0 42420.0 South FUR-BO-10001798 Furniture Bookcases   
1 42420.0 South FUR-CH-10000454 Furniture Chairs   
2 90036.0 West OFF-LA-10000240 Office Supplies Labels   
3 33311.0 South FUR-TA-10000577 Furniture Tables   
4 33311.0 South OFF-ST-10000760 Office Supplies Storage   
  
 Product Name Sales Quantity \  
0 Bush Somerset Collection Bookcase 261.9600 2   
1 Hon Deluxe Fabric Upholstered Stacking Chairs,... 731.9400 3   
2 Self-Adhesive Address Labels for Typewriters b... 14.6200 2   
3 Bretford CR4500 Series Slim Rectangular Table 957.5775 5   
4 Eldon Fold 'N Roll Cart System 22.3680 2   
  
 Discount Profit   
0 0.00 41.9136   
1 0.00 219.5820   
2 0.00 6.8714   
3 0.45 -383.0310   
4 0.20 2.5164   
  
[5 rows x 21 columns]

0 False  
1 False  
2 True  
3 False  
4 True  
 ...   
9989 False  
9990 False  
9991 False  
9992 True  
9993 True  
Name: Category, Length: 9994, dtype: bool

# Filter by numerical data (intervals)  
# order [ condition ]  
#orders[ orders['Sales'] == 22638.48 ]  
orders[ orders['Sales'] > 5000 ]

Row ID Order ID Order Date Ship Date Ship Mode \  
165 166 CA-2016-139892 2016-09-08 2016-09-12 Standard Class   
509 510 CA-2017-145352 2017-03-16 2017-03-22 Standard Class   
683 684 US-2019-168116 2019-11-04 2019-11-04 Same Day   
2505 2506 CA-2016-143917 2016-07-25 2016-07-27 Second Class   
2623 2624 CA-2019-127180 2019-10-22 2019-10-24 First Class   
2697 2698 CA-2016-145317 2016-03-18 2016-03-23 Standard Class   
4098 4099 CA-2016-116904 2016-09-23 2016-09-28 Standard Class   
4190 4191 CA-2019-166709 2019-11-17 2019-11-22 Standard Class   
4277 4278 US-2018-107440 2018-04-16 2018-04-20 Standard Class   
6340 6341 CA-2019-143112 2019-10-05 2019-10-09 Standard Class   
6425 6426 CA-2018-143714 2018-05-23 2018-05-27 Standard Class   
6520 6521 CA-2019-138289 2019-01-16 2019-01-18 Second Class   
6626 6627 CA-2016-145541 2016-12-14 2016-12-21 Standard Class   
6826 6827 CA-2018-118689 2018-10-02 2018-10-09 Standard Class   
7666 7667 US-2018-140158 2018-10-04 2018-10-08 Standard Class   
8153 8154 CA-2019-140151 2019-03-23 2019-03-25 First Class   
8488 8489 CA-2018-158841 2018-02-02 2018-02-04 Second Class   
8858 8859 CA-2019-135909 2019-10-13 2019-10-20 Standard Class   
9039 9040 CA-2018-117121 2018-12-17 2018-12-21 Standard Class   
  
 Customer ID Customer Name Segment Country/Region \  
165 BM-11140 Becky Martin Consumer United States   
509 CM-12385 Christopher Martinez Consumer United States   
683 GT-14635 Grant Thornton Corporate United States   
2505 KL-16645 Ken Lonsdale Consumer United States   
2623 TA-21385 Tom Ashbrook Home Office United States   
2697 SM-20320 Sean Miller Home Office United States   
4098 SC-20095 Sanjit Chand Consumer United States   
4190 HL-15040 Hunter Lopez Consumer United States   
4277 BS-11365 Bill Shonely Corporate United States   
6340 TS-21370 Todd Sumrall Corporate United States   
6425 CC-12370 Christopher Conant Consumer United States   
6520 AR-10540 Andy Reiter Consumer United States   
6626 TB-21400 Tom Boeckenhauer Consumer United States   
6826 TC-20980 Tamara Chand Corporate United States   
7666 DR-12940 Daniel Raglin Home Office United States   
8153 RB-19360 Raymond Buch Consumer United States   
8488 SE-20110 Sanjit Engle Consumer United States   
8858 JW-15220 Jane Waco Corporate United States   
9039 AB-10105 Adrian Barton Consumer United States   
  
 City ... Postal Code Region Product ID \  
165 San Antonio ... 78207.0 Central TEC-MA-10000822   
509 Atlanta ... 30318.0 South OFF-BI-10003527   
683 Burlington ... 27217.0 South TEC-MA-10004125   
2505 San Francisco ... 94122.0 West OFF-SU-10000151   
2623 New York City ... 10024.0 East TEC-CO-10004722   
2697 Jacksonville ... 32216.0 South TEC-MA-10002412   
4098 Minneapolis ... 55407.0 Central OFF-BI-10001120   
4190 Newark ... 19711.0 East TEC-CO-10004722   
4277 Lakewood ... 8701.0 East TEC-MA-10001047   
6340 New York City ... 10035.0 East TEC-MA-10001047   
6425 Philadelphia ... 19120.0 East TEC-CO-10004722   
6520 Jackson ... 49201.0 Central OFF-BI-10004995   
6626 New York City ... 10024.0 East TEC-MA-10001127   
6826 Lafayette ... 47905.0 Central TEC-CO-10004722   
7666 Providence ... 2908.0 East TEC-CO-10001449   
8153 Seattle ... 98115.0 West TEC-CO-10004722   
8488 Arlington ... 22204.0 South TEC-MA-10001127   
8858 Sacramento ... 95823.0 West OFF-BI-10003527   
9039 Detroit ... 48205.0 Central OFF-BI-10000545   
  
 Category Sub-Category \  
165 Technology Machines   
509 Office Supplies Binders   
683 Technology Machines   
2505 Office Supplies Supplies   
2623 Technology Copiers   
2697 Technology Machines   
4098 Office Supplies Binders   
4190 Technology Copiers   
4277 Technology Machines   
6340 Technology Machines   
6425 Technology Copiers   
6520 Office Supplies Binders   
6626 Technology Machines   
6826 Technology Copiers   
7666 Technology Copiers   
8153 Technology Copiers   
8488 Technology Machines   
8858 Office Supplies Binders   
9039 Office Supplies Binders   
  
 Product Name Sales Quantity \  
165 Lexmark MX611dhe Monochrome Laser Printer 8159.952 8   
509 Fellowes PB500 Electric Punch Plastic Comb Bin... 6354.950 5   
683 Cubify CubeX 3D Printer Triple Head Print 7999.980 4   
2505 High Speed Automatic Electric Letter Opener 8187.650 5   
2623 Canon imageCLASS 2200 Advanced Copier 11199.968 4   
2697 Cisco TelePresence System EX90 Videoconferenci... 22638.480 6   
4098 Ibico EPK-21 Electric Binding System 9449.950 5   
4190 Canon imageCLASS 2200 Advanced Copier 10499.970 3   
4277 3D Systems Cube Printer, 2nd Generation, Magenta 9099.930 7   
6340 3D Systems Cube Printer, 2nd Generation, Magenta 5199.960 4   
6425 Canon imageCLASS 2200 Advanced Copier 8399.976 4   
6520 GBC DocuBind P400 Electric Binding System 5443.960 4   
6626 HP Designjet T520 Inkjet Large Format Printer ... 6999.960 4   
6826 Canon imageCLASS 2200 Advanced Copier 17499.950 5   
7666 Hewlett Packard LaserJet 3310 Copier 5399.910 9   
8153 Canon imageCLASS 2200 Advanced Copier 13999.960 4   
8488 HP Designjet T520 Inkjet Large Format Printer ... 8749.950 5   
8858 Fellowes PB500 Electric Punch Plastic Comb Bin... 5083.960 5   
9039 GBC Ibimaster 500 Manual ProClick Binding System 9892.740 13   
  
 Discount Profit   
165 0.4 -1359.9920   
509 0.0 3177.4750   
683 0.5 -3839.9904   
2505 0.0 327.5060   
2623 0.2 3919.9888   
2697 0.5 -1811.0784   
4098 0.0 4630.4755   
4190 0.0 5039.9856   
4277 0.0 2365.9818   
6340 0.0 1351.9896   
6425 0.4 1119.9968   
6520 0.0 2504.2216   
6626 0.0 2239.9872   
6826 0.0 8399.9760   
7666 0.0 2591.9568   
8153 0.0 6719.9808   
8488 0.0 2799.9840   
8858 0.2 1906.4850   
9039 0.0 4946.3700   
  
[19 rows x 21 columns]

# Let's review what happens when we filter by row  
  
# We create a boolean vector with the same size as the rows in the data set  
condition = orders['Sales'] == 9449.950   
print(condition)  
display(orders[condition])  
  
print(orders['Sales'].max())  
#orders[orders['Sales'] == orders['Sales'].max()]

0 False  
1 False  
2 False  
3 False  
4 False  
 ...   
9989 False  
9990 False  
9991 False  
9992 False  
9993 False  
Name: Sales, Length: 9994, dtype: bool

Row ID Order ID Order Date Ship Date Ship Mode \  
4098 4099 CA-2016-116904 2016-09-23 2016-09-28 Standard Class   
  
 Customer ID Customer Name Segment Country/Region City ... \  
4098 SC-20095 Sanjit Chand Consumer United States Minneapolis ...   
  
 Postal Code Region Product ID Category Sub-Category \  
4098 55407.0 Central OFF-BI-10001120 Office Supplies Binders   
  
 Product Name Sales Quantity Discount \  
4098 Ibico EPK-21 Electric Binding System 9449.95 5 0.0   
  
 Profit   
4098 4630.4755   
  
[1 rows x 21 columns]

22638.48

# The condition might be an interval of things (not a fixed value)  
condition = (orders['Sales'] > 5000) & (orders['Sales'] < 10000)  
#condition = orders['Sales'] > orders['Sales'].max()\*0.5  
orders[condition]

Row ID Order ID Order Date Ship Date Ship Mode \  
165 166 CA-2016-139892 2016-09-08 2016-09-12 Standard Class   
509 510 CA-2017-145352 2017-03-16 2017-03-22 Standard Class   
683 684 US-2019-168116 2019-11-04 2019-11-04 Same Day   
2505 2506 CA-2016-143917 2016-07-25 2016-07-27 Second Class   
4098 4099 CA-2016-116904 2016-09-23 2016-09-28 Standard Class   
4277 4278 US-2018-107440 2018-04-16 2018-04-20 Standard Class   
6340 6341 CA-2019-143112 2019-10-05 2019-10-09 Standard Class   
6425 6426 CA-2018-143714 2018-05-23 2018-05-27 Standard Class   
6520 6521 CA-2019-138289 2019-01-16 2019-01-18 Second Class   
6626 6627 CA-2016-145541 2016-12-14 2016-12-21 Standard Class   
7666 7667 US-2018-140158 2018-10-04 2018-10-08 Standard Class   
8488 8489 CA-2018-158841 2018-02-02 2018-02-04 Second Class   
8858 8859 CA-2019-135909 2019-10-13 2019-10-20 Standard Class   
9039 9040 CA-2018-117121 2018-12-17 2018-12-21 Standard Class   
  
 Customer ID Customer Name Segment Country/Region \  
165 BM-11140 Becky Martin Consumer United States   
509 CM-12385 Christopher Martinez Consumer United States   
683 GT-14635 Grant Thornton Corporate United States   
2505 KL-16645 Ken Lonsdale Consumer United States   
4098 SC-20095 Sanjit Chand Consumer United States   
4277 BS-11365 Bill Shonely Corporate United States   
6340 TS-21370 Todd Sumrall Corporate United States   
6425 CC-12370 Christopher Conant Consumer United States   
6520 AR-10540 Andy Reiter Consumer United States   
6626 TB-21400 Tom Boeckenhauer Consumer United States   
7666 DR-12940 Daniel Raglin Home Office United States   
8488 SE-20110 Sanjit Engle Consumer United States   
8858 JW-15220 Jane Waco Corporate United States   
9039 AB-10105 Adrian Barton Consumer United States   
  
 City ... Postal Code Region Product ID \  
165 San Antonio ... 78207.0 Central TEC-MA-10000822   
509 Atlanta ... 30318.0 South OFF-BI-10003527   
683 Burlington ... 27217.0 South TEC-MA-10004125   
2505 San Francisco ... 94122.0 West OFF-SU-10000151   
4098 Minneapolis ... 55407.0 Central OFF-BI-10001120   
4277 Lakewood ... 8701.0 East TEC-MA-10001047   
6340 New York City ... 10035.0 East TEC-MA-10001047   
6425 Philadelphia ... 19120.0 East TEC-CO-10004722   
6520 Jackson ... 49201.0 Central OFF-BI-10004995   
6626 New York City ... 10024.0 East TEC-MA-10001127   
7666 Providence ... 2908.0 East TEC-CO-10001449   
8488 Arlington ... 22204.0 South TEC-MA-10001127   
8858 Sacramento ... 95823.0 West OFF-BI-10003527   
9039 Detroit ... 48205.0 Central OFF-BI-10000545   
  
 Category Sub-Category \  
165 Technology Machines   
509 Office Supplies Binders   
683 Technology Machines   
2505 Office Supplies Supplies   
4098 Office Supplies Binders   
4277 Technology Machines   
6340 Technology Machines   
6425 Technology Copiers   
6520 Office Supplies Binders   
6626 Technology Machines   
7666 Technology Copiers   
8488 Technology Machines   
8858 Office Supplies Binders   
9039 Office Supplies Binders   
  
 Product Name Sales Quantity \  
165 Lexmark MX611dhe Monochrome Laser Printer 8159.952 8   
509 Fellowes PB500 Electric Punch Plastic Comb Bin... 6354.950 5   
683 Cubify CubeX 3D Printer Triple Head Print 7999.980 4   
2505 High Speed Automatic Electric Letter Opener 8187.650 5   
4098 Ibico EPK-21 Electric Binding System 9449.950 5   
4277 3D Systems Cube Printer, 2nd Generation, Magenta 9099.930 7   
6340 3D Systems Cube Printer, 2nd Generation, Magenta 5199.960 4   
6425 Canon imageCLASS 2200 Advanced Copier 8399.976 4   
6520 GBC DocuBind P400 Electric Binding System 5443.960 4   
6626 HP Designjet T520 Inkjet Large Format Printer ... 6999.960 4   
7666 Hewlett Packard LaserJet 3310 Copier 5399.910 9   
8488 HP Designjet T520 Inkjet Large Format Printer ... 8749.950 5   
8858 Fellowes PB500 Electric Punch Plastic Comb Bin... 5083.960 5   
9039 GBC Ibimaster 500 Manual ProClick Binding System 9892.740 13   
  
 Discount Profit   
165 0.4 -1359.9920   
509 0.0 3177.4750   
683 0.5 -3839.9904   
2505 0.0 327.5060   
4098 0.0 4630.4755   
4277 0.0 2365.9818   
6340 0.0 1351.9896   
6425 0.4 1119.9968   
6520 0.0 2504.2216   
6626 0.0 2239.9872   
7666 0.0 2591.9568   
8488 0.0 2799.9840   
8858 0.2 1906.4850   
9039 0.0 4946.3700   
  
[14 rows x 21 columns]

## Find out the top 5 orders  
  
orders = orders.sort\_values(by=['Sales'], ascending=False)  
#orders.head()  
  
top\_five = orders.iloc[0:5]  
top\_five

Row ID Order ID Order Date Ship Date Ship Mode \  
2697 2698 CA-2016-145317 2016-03-18 2016-03-23 Standard Class   
6826 6827 CA-2018-118689 2018-10-02 2018-10-09 Standard Class   
8153 8154 CA-2019-140151 2019-03-23 2019-03-25 First Class   
2623 2624 CA-2019-127180 2019-10-22 2019-10-24 First Class   
4190 4191 CA-2019-166709 2019-11-17 2019-11-22 Standard Class   
  
 Customer ID Customer Name Segment Country/Region City \  
2697 SM-20320 Sean Miller Home Office United States Jacksonville   
6826 TC-20980 Tamara Chand Corporate United States Lafayette   
8153 RB-19360 Raymond Buch Consumer United States Seattle   
2623 TA-21385 Tom Ashbrook Home Office United States New York City   
4190 HL-15040 Hunter Lopez Consumer United States Newark   
  
 ... Postal Code Region Product ID Category Sub-Category \  
2697 ... 32216.0 South TEC-MA-10002412 Technology Machines   
6826 ... 47905.0 Central TEC-CO-10004722 Technology Copiers   
8153 ... 98115.0 West TEC-CO-10004722 Technology Copiers   
2623 ... 10024.0 East TEC-CO-10004722 Technology Copiers   
4190 ... 19711.0 East TEC-CO-10004722 Technology Copiers   
  
 Product Name Sales Quantity \  
2697 Cisco TelePresence System EX90 Videoconferenci... 22638.480 6   
6826 Canon imageCLASS 2200 Advanced Copier 17499.950 5   
8153 Canon imageCLASS 2200 Advanced Copier 13999.960 4   
2623 Canon imageCLASS 2200 Advanced Copier 11199.968 4   
4190 Canon imageCLASS 2200 Advanced Copier 10499.970 3   
  
 Discount Profit   
2697 0.5 -1811.0784   
6826 0.0 8399.9760   
8153 0.0 6719.9808   
2623 0.2 3919.9888   
4190 0.0 5039.9856   
  
[5 rows x 21 columns]

## Business Challenge: Analyzing Avocado Sales

Business Challenge: Analyzing Avocado Sales

In this exercise, we'll delve into a real-world business scenario involving avocado sales data. Avocado consumption has surged in recent years, and you've been hired by a regional grocery store chain to gain insights from their sales data. The store wants to understand the trends, pricing strategies, and factors influencing avocado sales to make informed decisions and improve profitability.

### Introduction to the Avocado Dataset

The dataset you'll be working with contains information about avocado sales across different regions in the United States. The data includes details like date, average price, total volume sold, region, and more.

**Your Mission:** Using Pandas, perform a comprehensive analysis to answer critical business questions. Here are some of the tasks you'll need to accomplish:

### Tasks:

1. **Data Loading:** Begin by loading the Avocado dataset (avocado.csv) into a Pandas DataFrame.
2. **Data Exploration:** Conduct an exploratory data analysis to understand the dataset's structure, including the number of rows and columns, data types, and any missing values.
3. **Time-Series Analysis:** Analyze the avocado sales over time. Identify seasonal trends, and determine if there are any patterns related to pricing and volume.
4. **Regional Analysis:** Investigate avocado sales by region. Which regions are the top performers in terms of sales volume and pricing? Are there any regions that require specific attention?
5. **Price and Volume Trends:** Determine how changes in avocado prices affect sales volume. Are there price points that drive higher or lower sales?
6. **Price Elasticity:** Calculate the price elasticity of demand for avocados. This will help the store understand how sensitive sales are to price changes.
7. **Recommendations:** Based on your analysis, provide actionable recommendations to the grocery store chain. What pricing strategies should they consider? Are there specific regions where they can improve sales?

### Getting Started:

To get started, load the Avocado dataset and begin your data exploration. Utilize Pandas for data cleaning, visualization, and analysis. As you progress through the tasks, document your findings and insights to present to the grocery store chain's management.

Remember, Pandas is a powerful tool that can help businesses make data-driven decisions. This exercise will give you hands-on experience in data analysis and showcase the valuable insights that can be extracted from real-world data.

Now, let's dive into the world of avocado sales and start making data-driven recommendations to boost profitability!

# your code here

## Recap: Key Points About Pandas

RECAP

* **Pandas is a Powerful Library:** Pandas is a versatile Python library used for working with structured data efficiently.
* **Built on NumPy:** It's built on top of NumPy, which makes it incredibly fast and efficient for data manipulation.
* **Widely Used:** Pandas is widely adopted by other data libraries and tools, making it a fundamental part of the data ecosystem.
* **Tabular Data:** Pandas excels at handling tabular data, which consists of rows and columns.
* **Python and Pandas Together:** When working with data, you often use both Python and Pandas in tandem to achieve your goals.

### Exploratory Data Analysis (EDA)

* **Data Preview:** You can quickly preview your data using methods like head(), tail(), and sample() to inspect the beginning, end, or random portions of your dataset.
* **Data Overview:** The info() and describe() methods provide an overview of your data, including data types and descriptive statistics.
* **Unique Values:** Use unique() or value\_counts() to find unique values or count occurrences in a column.
* **Sorting:** You can sort your data using the sort\_values() method, which is helpful for arranging data by specific columns.
* **Subsetting Data:** Subsetting allows you to select specific rows or columns. Use square brackets ([]) to extract columns or rows based on conditions.

### Creating DataFrames

* **From Lists of Dictionaries:** You can create DataFrames from lists of dictionaries, where each dictionary represents a row.
* **From Dictionaries with Lists:** Alternatively, DataFrames can be created from dictionaries with lists as values. Be mindful of ensuring lists have the same length.
* **Reading Data:** Pandas provides functions like read\_csv() for reading data from various sources, such as CSV files, URLs, SQL databases, Excel files, and more. You can use both absolute and relative paths, and even load data from remote sources like GitHub repositories.

These are some of the essential concepts and operations in Pandas, allowing you to manipulate and explore your data effectively.

## Pandas usual methods

df.head() # prints the head, default 5 rows  
df.tail() # set the tail, default 5 rows  
df.describe() # statistical description  
df.info() # df information  
df.columns # show column  
df.index # show index  
df.dtypes # show column data types  
df.plot() # make a plot  
df.hist() # make a histogram  
df.col.value\_counts() # counts the unique values ​​of a column  
df.col.unique() # returns unique values ​​from a column  
df.copy() # copies the df  
df.drop() # remove columns or rows (axis=0,1)  
df.dropna() # remove nulls  
df.fillna() # fills nulls  
df.shape # dimensions of the df  
df.\_get\_numeric\_data() # select numeric columns  
df.rename() # rename columns  
df.str.replace() # replace columns of strings  
df.astype(dtype='float32') # change the data type  
df.iloc[] # locate by index  
df.loc[] # locate by element  
df.transpose() # transposes the df  
df.T  
df.sample(n, frac) # sample from df  
df.col.sum() # sum of a column  
df.col.max() # maximum of a column  
df.col.min() # minimum of one column  
df[col] # select column  
df.col  
df.isnull() # null values  
df.isna()  
df.notna() # not null values  
df.drop\_duplicates() # remove duplicates  
df.reset\_index(inplace=True) # reset the index and overwrite

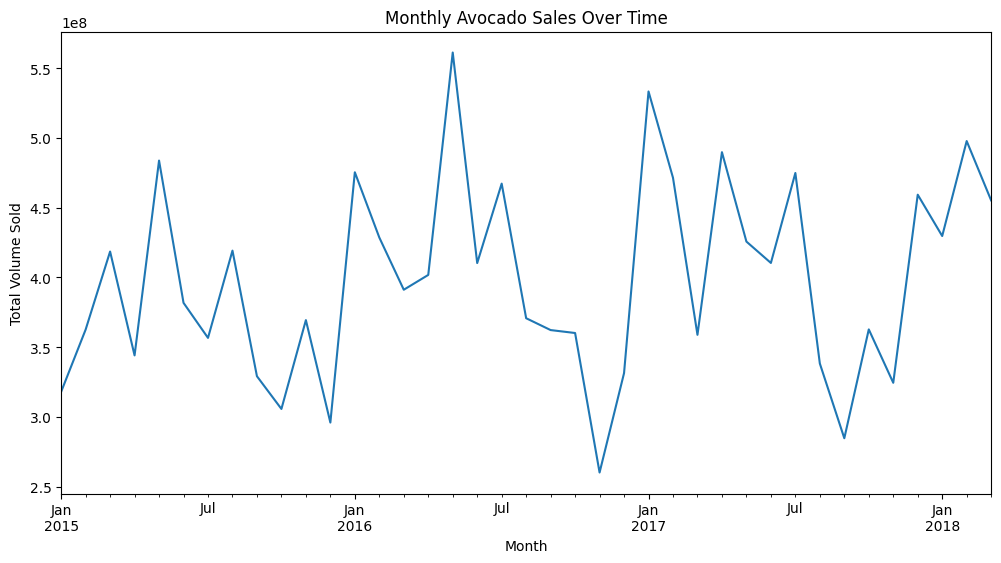
## Further materials

* [Read the docs!](https://pandas.pydata.org/pandas-docs/stable/index.html)
* [Cheatsheet](https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf)
* [Exercises to practice](https://github.com/guipsamora/pandas_exercises)
* [More on merge, concat, and join](https://realpython.com/pandas-merge-join-and-concat/#pandas-join-combining-data-on-a-column-or-index). And [even more!](https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html)

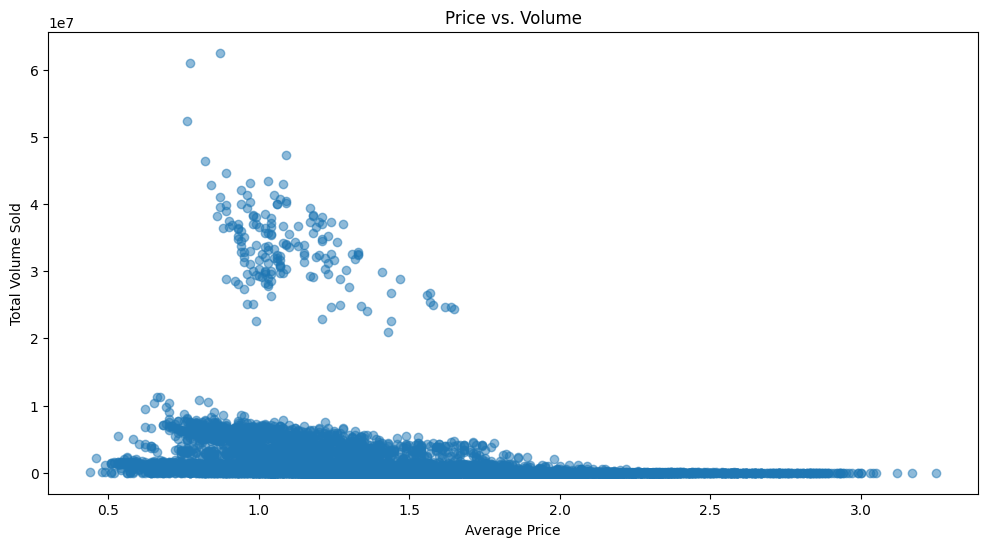
### Solution

# Import necessary libraries  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
  
# Load the Avocado dataset into a Pandas DataFrame  
df = pd.read\_csv("datasets/avocado\_kaggle.csv")  
  
# Data Exploration  
# Display basic information about the dataset  
print(df.info())  
  
# Time-Series Analysis  
# Convert the 'Date' column to a datetime format  
df['Date'] = pd.to\_datetime(df['Date'])  
  
# Group data by date to analyze trends  
monthly\_sales = df.groupby(df['Date'].dt.to\_period("M"))['Total Volume'].sum()  
monthly\_sales.plot(figsize=(12, 6))  
plt.title('Monthly Avocado Sales Over Time')  
plt.xlabel('Month')  
plt.ylabel('Total Volume Sold')  
plt.show()  
  
# Regional Analysis  
# Identify the top-performing regions in terms of sales volume and pricing  
top\_regions\_volume = df.groupby('region')['Total Volume'].sum().nlargest(5)  
top\_regions\_price = df.groupby('region')['AveragePrice'].mean().nlargest(5)  
  
print("Top 5 Regions by Sales Volume:")  
print(top\_regions\_volume)  
print("\nTop 5 Regions by Average Price:")  
print(top\_regions\_price)  
  
# Price and Volume Trends  
# Analyze how changes in avocado prices affect sales volume  
plt.figure(figsize=(12, 6))  
plt.scatter(df['AveragePrice'], df['Total Volume'], alpha=0.5)  
plt.title('Price vs. Volume')  
plt.xlabel('Average Price')  
plt.ylabel('Total Volume Sold')  
plt.show()  
  
# Price Elasticity  
# Calculate the price elasticity of demand  
df['Price Elasticity'] = df['Total Volume'] / df['AveragePrice']  
df['Price Elasticity'].describe()  
  
# Recommendations  
# Provide recommendations based on analysis  
print("\nRecommendations:")  
print("1. Focus on regions with high sales volume, such as", top\_regions\_volume.index[0])  
print("2. Monitor price elasticity closely and consider adjusting prices strategically.")  
print("3. Analyze seasonal trends for potential marketing campaigns.")

<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 18249 entries, 0 to 18248  
Data columns (total 14 columns):  
 # Column Non-Null Count Dtype   
--- ------ -------------- -----   
 0 Unnamed: 0 18249 non-null int64   
 1 Date 18249 non-null object   
 2 AveragePrice 18249 non-null float64  
 3 Total Volume 18249 non-null float64  
 4 4046 18249 non-null float64  
 5 4225 18249 non-null float64  
 6 4770 18249 non-null float64  
 7 Total Bags 18249 non-null float64  
 8 Small Bags 18249 non-null float64  
 9 Large Bags 18249 non-null float64  
 10 XLarge Bags 18249 non-null float64  
 11 type 18249 non-null object   
 12 year 18249 non-null int64   
 13 region 18249 non-null object   
dtypes: float64(9), int64(2), object(3)  
memory usage: 1.9+ MB  
None



Top 5 Regions by Sales Volume:  
region  
TotalUS 5.864740e+09  
West 1.086779e+09  
California 1.028982e+09  
SouthCentral 1.011280e+09  
Northeast 7.132809e+08  
Name: Total Volume, dtype: float64  
  
Top 5 Regions by Average Price:  
region  
HartfordSpringfield 1.818639  
SanFrancisco 1.804201  
NewYork 1.727574  
Philadelphia 1.632130  
Sacramento 1.621568  
Name: AveragePrice, dtype: float64



Recommendations:  
1. Focus on regions with high sales volume, such as TotalUS  
2. Monitor price elasticity closely and consider adjusting prices strategically.  
3. Analyze seasonal trends for potential marketing campaigns.