**Getting Started With Pandas**

## A PROJECT REPORT

***Submitted by,***

**Mr. Ankit Kumar - 20201ISE0089**

**Mr. Nataraj S - 20201ISE0065**

**Mr. Yash Sanjeev Bastawad -20201ISE0063**

**Mr. Gaurav Kumar - 20201ISE0088**

**Mr. HemanthKumar D M - 20201ISE0095**

**Mr. C H Surya Ragavendra -20201ISE0057**

**Mr. Sudarshan V - 20201ISE0049**

**Mr. Abhilash Y - 20201ISE0072**

**To Asst.professor,**

**Miss Poornima**

**BACHELOR OF TECHNOLOGY**

**IN**

**INFORMATION SCIENCE AND ENGINEERING**

**At**



**PRESIDENCY UNIVERSITY**

**ABSTRACT**

Getting started with Pandas involves understanding its core components and functionalities. The central data structures in Pandas are the DataFrame, akin to a table in a spreadsheet, and the Series, representing a single column or row of data. These structures facilitate easy indexing, slicing, and manipulation of data.Once you've loaded your data into a DataFrame, you can unleash Pandas' power to clean and transform it. Techniques such as handling missing data, applying functions across rows or columns, and merging multiple datasets are made simple with Pandas' extensive range of methods.

Pandas also excels in data exploration and analysis. You can aggregate data, calculate statistics, and generate insights using functions like `groupby()`, `pivot\_table()`, and `value\_counts()`. Visualization is seamless with Pandas, as it seamlessly integrates with libraries like Matplotlib and Seaborn, allowing you to create insightful plots and charts to convey your findings effectively.

Whether you're a beginner or an experienced data practitioner, Pandas offers a robust toolkit for data manipulation and analysis in Python, empowering you to extract valuable insights and make informed decisions from your datasets. Delving deeper, Pandas supports advanced operations such as time series analysis, categorical data handling, and text data manipulation, broadening its applicability across various domains. Its flexibility and efficiency make it a go-to choice for data professionals tackling real-world challenges. With a wealth of documentation, tutorials, and community support, mastering Pandas becomes not just a skill, but a gateway to unlocking the full potential of data analysis in Python. Whether you're wrangling messy datasets, conducting exploratory data analysis, or building pr

|  |
| --- |
|  |

**TABLE OF CONTENTS**

|  |  |  |  |
| --- | --- | --- | --- |
| **CHAPTER NO.** | | **TITLE** | **PAGE NO.** |
|  | | ABSTRACT | 2 |
|  | |
| 1. | | INTRODUCTION | 5 |
|  | | What is Pandas? |  |
|  | | Why Use Pandas |  |
|  |
| 2. | | SERIES | 8 |
|  | | Creating Pandas Series |  |
|  | | Accessing Pandas Series Elements |  |
|  | | Pandas Series Data Types Pandas Series Operations |  |
| 3. | | DATAFRAMES | 15 |
|  | | Creating Pandas DataFrames |  |
|  | | Accessing Pandas DataFrames Elements |  |
|  | | Pandas DataFrames Data Types Pandas DataFrames Operations |  |
| 4. | | DATACLEANING | 21 |
|  | | Handling Missing Data |  |
|  | | Data Imputation |  |
|  | | Data Transformation |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 5. | | DATAMANIPULATION  Filtering Data | 35 |
|  | | Sorting Data? |  |
|  | | Grouping Data |  |
|  | | Merging and Joining Data |  |
| 6. | | DATA ANALYSIS | 38 |
|  | | Basic Structural Analysis |  |
|  | | Time Series Analysis |  |
|  | | Descriptive Statistics Data Visualization |  |
| 7. | | DATA-IPUT/OUTPUT | 42 |
|  | | Reading And Writing Data Disk  Reading And Writing Data from other data formats |  |
|  |

|  |
| --- |
|  |
|  |

**Introduction:**

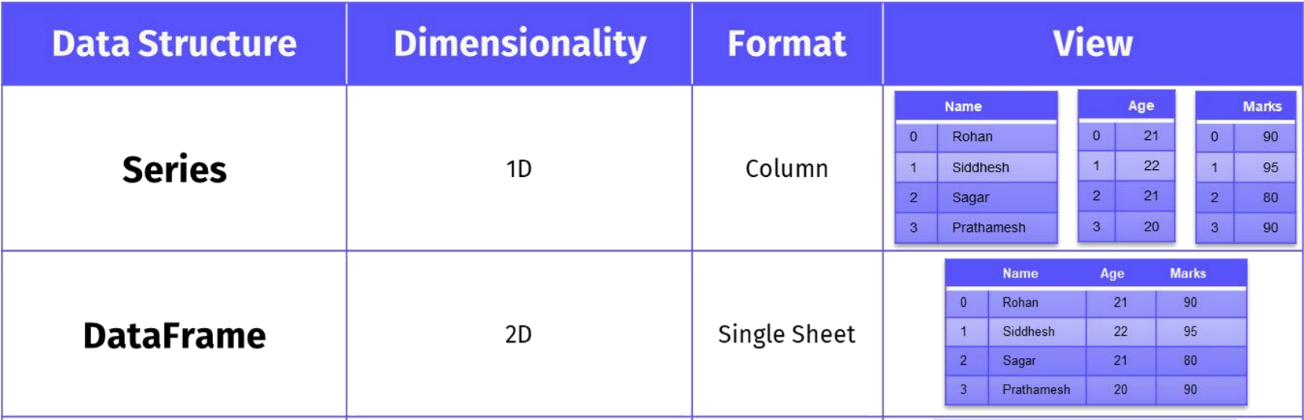
Pandas, a Python library, is a powerful tool for data manipulation and analysis.

Developed by Wes McKinney in 2008, it introduces two primary data structures: Series and DataFrame. Series is a one-dimensional labeled array, while DataFrame is a two-dimensional tabular structure resembling a spreadsheet. These structures offer versatile capabilities for handling structured data, making Pandas invaluable in various data-related tasks.The utility of Pandas spans across data cleaning, transformation, and analysis. It simplifies tasks such as filtering, sorting, and reshaping data, allowing users to focus on insights rather than mechanics. With Pandas, handling missing values, removing duplicates, and transforming data formats become straightforward processes. Furthermore, Pandas facilitates data analysis by providing functions for descriptive statistics, group-by operations, and time-series analysis.Pandas seamlessly integrates with other Python libraries like NumPy, Matplotlib, and Scikit-Learn, enhancing its functionality and interoperability. Installing Pandas is effortless using Python's package manager, pip. Once installed, users can leverage Pandas to extract valuable insights from their data efficiently. In essence, Pandas empowers users to manipulate, analyze, and visualize data with ease, making it an essential tool in the data science toolkit.

**What is Pandas?**

Pandas is a powerful and open-source Python library used for data manipulation and analysis. It provides data structures and functions to efficiently work with structured data. Here are some key points about Pandas:Data Manipulation: Pandas simplifies tasks related to data manipulation in Python. It’s particularly well-suited for working with tabular data, such as spreadsheets or SQL tables.

Essential Tool: Data analysts, scientists, and engineers often rely on Pandas when dealing with structured data. Its name is derived from “panel data,” and it’s also referred to as “Python Data Analysis.”



Pandas provides two primary data structures:

Series : A one-dimensional labeled array capable of holding data of various types. It nes the features of a list and a dictionary, offering both index-based and labelaccess to its elements.

DataFrame: A two-dimensional labeled data structure resembling a table or consisting of rows and columns. Each column in a DataFrame can have a a type, making it a versatile container for structured data.These data structures serve as the building blocks for data manipulation and analysis tasks in Pandas, enabling users to perform a wide range of operations with ease and

efficiency

Why Use Pandas?

The popularity and ubiquity of Pandas stem from its rich set of features and benefits, which cater to the needs of data scientists, analysts, and enthusiasts alike. the Pandas library is used in conjunction with other libraries that are used for data science.It is built on top of the NumPy library which means that a lot of the structures of NumPy are used or replicated in Pandas.The data produced by Pandas is often used as input for plotting functions in

Matplotlib, statistical analysis in SciPy, and machine learning algorithms in Scikit-learn.

Let's explore some of the key reasons why Pandas is the tool of choice for data manipulation and analysis:

**Data Manipulation:**

Pandas offers a comprehensive suite of functions and methods for manipulating data, enabling tasks such as filtering, sorting, merging, and reshaping. Its syntax and expressive API simplify complex operations, allowing users to focus on their analysis rather than the mechanics of data manipulation. Dealing with messy, inconsistent, or incomplete data is a common challenge in data analysis. Pandas provides robust tools for data cleaning, including handling missing values, outliers, and transforming data formats. Whether it's imputing missing values, encoding categorical variables, or detecting outliers, Pandas offers efficient solutions to ensure data quality and consistency. From basic descriptive statistics to advanced analytics, Pandas facilitates a wide range of data analysis tasks. Its built-in functions and methods enable users to compute summary statistics, perform group-by operations, conduct time-series analysis, and visualize trends in financial data, customer behavior, or any other dataset effortlessly. Its intuitive syntax and expressive API enable users to perform data manipulation tasks effortlessly, allowing them to focus on their analysis.

**Data Cleaning:**

Dealing with messy, inconsistent, or incomplete data is a common challenge in data analysis. Pandas provides robust tools for data cleaning, including handling missing values, removing duplicates, and efficiently transforming data formats. Whether it's imputing missing values, encoding categorical variables, or detecting outliers, Pandas offers efficient solutions to ensure data quality and consistency.

**Data Analysis:**

From basic descriptive statistics to building predictive models, Pandas provides the tools needed to extract actionable insights from data. Whether exploring summary statistics or building predictive models, Pandas facilitates various data analysis tasks, enabling users to derive insights and make informed decisions.

**Data Visualization:**

While Pandas is not a data visualization library itself, it integrates seamlessly with libraries such as Matplotlib and Seaborn. Users can leverage Pandas to manipulate data for visualization, allowing for the creation of insightful plots, charts, and graphs to communicate findings effectively. By integrating with other Python libraries and

frameworks, Pandas fosters a cohesive data analysis ecosystem. Whether it's NumPy for numerical computations, SciPy for scientific computing, or Scikit-Learn for machine learning, Pandas seamlessly integrates with these tools, enabling a seamless workflow

**Integration:**

Pandas plays a crucial role in the Python ecosystem for data analysis and model training. It seamlessly integrates with other libraries and frameworks, providing efficient data manipulation capabilities for scientific computing, machine learning, and data visualization. Whether it's loading data from various file formats or exporting results to different formats, Pandas provides flexible and efficient solutions for data ingestion and exportation, making it a preferred choice for data analysis tasks.

**Series**

**Creating Pandas Series**

Creating pandas Series is the first step in working with data using this powerful library. There are several methods available for creating Series, each suited to different scenarios and data types. Below, we'll explore various methods with detailed explanations, code examples, and illustrations.

1. Creating Series from Lists, Arrays, or Dictionaries:

-From Lists:You can create a Series from a Python list by passing the list as an argument to the `pd.Series()` constructor. The index is automatically generated, starting from 0.

import pandas as pd

data\_list = [10, 20, 30, 40, 50]

series\_from\_list = pd.Series(data\_list)

print(series\_from\_list)

-From Arrays:Similarly, you can create a Series from a NumPy array.

import numpy as np

data\_array = np.array([1.5, 2.5, 3.5, 4.5, 5.5])

series\_from\_array = pd.Series(data\_array)

print(series\_from\_array)

-From Dictionaries:You can create a Series from a Python dictionary, where keys become the index labels.

data\_dict = {'A': 10, 'B': 20, 'C': 30, 'D': 40, 'E': 50}

series\_from\_dict = pd.Series(data\_dict)

print(series\_from\_dict)

2. Using `pd.Series()` Constructor with Different Parameters:

-With Data and Index:You can pass both data and index as arguments to the `pd.Series()` constructor to create a Series with specific index labels.

data = [100, 200, 300, 400, 500]

index = ['A', 'B', 'C', 'D', 'E']

series\_with\_index = pd.Series(data, index=index)

print(series\_with\_index)

-With Scalar Value:You can create a Series with a scalar value repeated over a specified index.

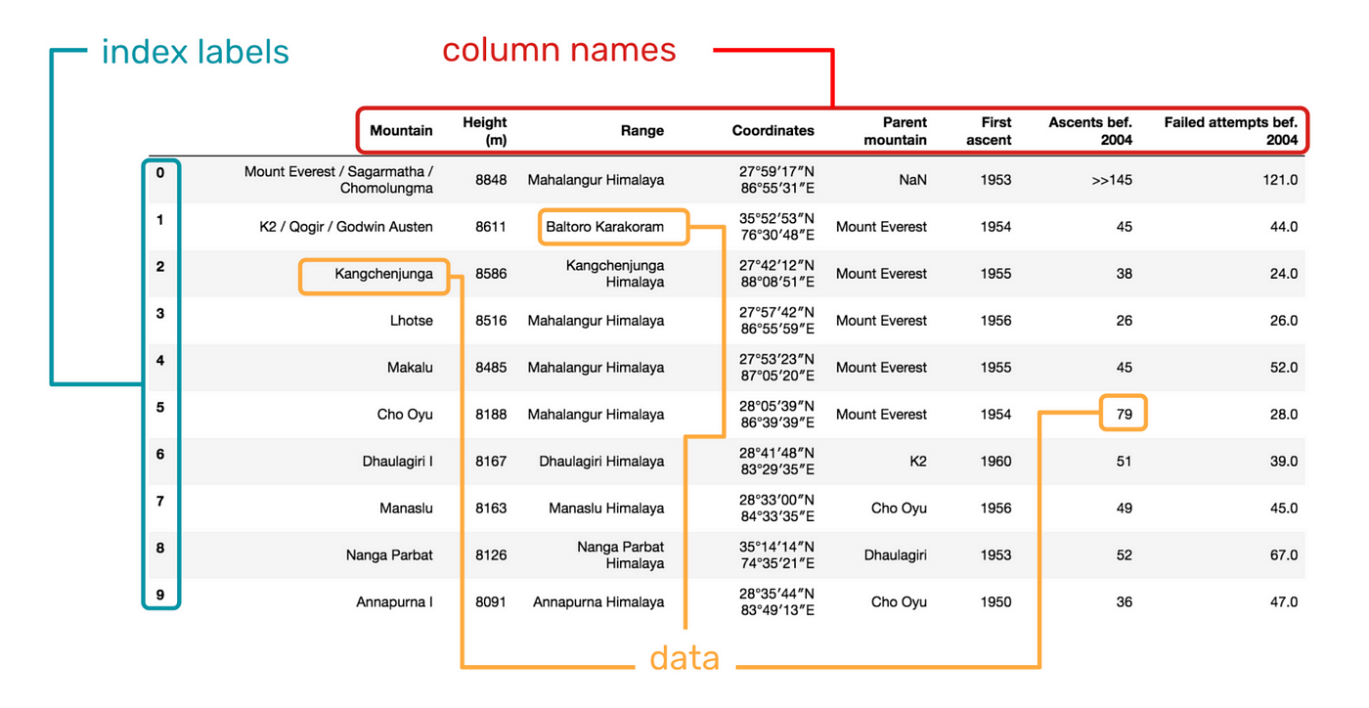
scalar\_value = 5

index = ['A', 'B', 'C', 'D', 'E']

series\_with\_scalar = pd.Series(scalar\_value, index=index)

print(series\_with\_scalar)

Each of these methods offers flexibility and convenience in creating pandas Series, allowing you to tailor the Series to your specific data and requirements. Experimenting with these methods will help you become proficient in data manipulation using pandas.



**Accessing Pandas Series Elements**

Accessing elements within a pandas Series is fundamental to data analysis and manipulation tasks. This page will cover various techniques for accessing elements, including indexing, slicing, and handling missing values and out-of-bound indices.

1. Indexing and Slicing Operations:

- Label-based Indexing: Pandas Series support label-based indexing, where you can access elements using their index labels.

- Position-based Indexing: You can also access elements using their position in the Series, similar to Python lists.

2. Demonstrations:

- Accessing Single Elements:

import pandas as pd

data = [10, 20, 30, 40, 50]

series = pd.Series(data, index=['A', 'B', 'C', 'D', 'E'])

# Label-based indexing

print(series['A']) # Output: 10

# Position-based indexing

print(series[0]) # Output: 10

- Accessing Multiple Elements:

# Label-based indexing

print(series[['A', 'C', 'E']]) # Output: A 10\nC 30\nE 50\n

# Position-based indexing

print(series[[0, 2, 4]]) # Output: A 10\nC 30\nE 50\n

- Slicing:

# Label-based slicing (inclusive)

print(series['A':'C']) # Output: A 10\nB 20\nC 30\n

# Position-based slicing (exclusive)

print(series[0:3]) # Output: A 10\nB 20\nC 30\n

3. Handling Missing Values and Out-of-Bound Indices:

- Pandas Series automatically aligns data based on their index labels. If an index label is not found, it returns `NaN` (Not a Number) for that element.

- You can use methods like `isnull()` and `fillna()` to handle missing values effectively.

- Accessing out-of-bound indices will raise an error unless you handle it explicitly.

By understanding these indexing and slicing operations and knowing how to handle missing values and out-of-bound indices, you can efficiently retrieve and manipulate data within pandas Series, facilitating various data analysis tasks.

**Pandas Series Data Types**

Understanding data types within pandas Series is essential for proper data manipulation, analysis, and interpretation. This page will cover the common data types supported by pandas Series, methods for checking and converting data types, and best practices for choosing appropriate data types based on data characteristics.

1. Overview of Common Data Types:

- Numeric Types: Pandas Series support numeric data types such as integers (`int`) and floating-point numbers (`float`), which represent whole numbers and decimal numbers, respectively.

- Object Type: Object type is a catch-all for any Python objects. It's commonly used for string data but can also represent mixed data types within a single Series.

- DateTime Types: Pandas provides robust support for date and time data, including

`datetime64` for specific dates and times and `timedelta64` for time differences or durations.

- Categorical Data: Categorical data type is useful for representing data with a fixed number of unique values, such as categories or labels.

2. Checking and Converting Data Types:

- Checking Data Types: You can check the data type of a pandas Series using the `dtype` attribute.

import pandas as pd

series = pd.Series([1, 2, 3, 4, 5])

print(series.dtype) # Output: int64

- Converting Data Types: Pandas provides methods like `astype()` to convert data types. For example, converting numeric data to categorical data.

categorical\_series = series.astype('category')

print(categorical\_series.dtype) # Output: category

3. Best Practices for Choosing Data Types:

- Memory Efficiency: Choose data types that occupy less memory without sacrificing precision. For example, using `int8` instead of `int64` for small integers.

- Data Integrity: Select data types that accurately represent the nature of your data. For example, use datetime data type for date and time information to facilitate date-based operations.

- Performance Considerations: Opt for data types that support efficient operations. Numeric types generally perform better in mathematical operations compared to object types.

- Categorical Data: Utilize categorical data type for columns with a limited number of unique values to optimize memory usage and enable efficient analysis.

By understanding the common data types supported by pandas Series, how to check and convert data types, and best practices for choosing appropriate data types, you can ensure efficient and effective data manipulation and analysis workflows.

**Pandas Series Operations**

Pandas Series offer a wide range of operations for data manipulation and analysis. This page will explore various operations that can be performed on pandas Series, including arithmetic operations, element-wise operations, statistical operations, sorting, filtering, and grouping operations. Each operation will be accompanied by examples and use cases to illustrate its functionality.

1. Arithmetic Operations:

- Pandas Series support standard arithmetic operations such as addition (`+`), subtraction (`-`), multiplication (`\*`), and division (`/`) with either scalar values or other Series.

import pandas as pd

# Creating two Series

series1 = pd.Series([1, 2, 3, 4, 5])

series2 = pd.Series([10, 20, 30, 40, 50])

# Addition

print(series1 + series2)

# Subtraction

print(series1 - series2)

# Multiplication

print(series1 \* series2)

# Division

print(series2 / series1)

2. Element-wise Operations:

- Element-wise operations allow applying functions to each element of a Series or broadcasting operations across Series.

# Applying a function to each element

result = series1.apply(lambda x: x \*\* 2)

print(result)

# Broadcasting

print(series1 + 10)

3. Statistical Operations:

- Pandas Series provide statistical methods such as mean, median, minimum, maximum, and standard deviation.

# Statistical operations

print(series1.mean())

print(series1.median())

print(series1.min())

print(series1.max())

print(series1.std())

4. Sorting, Filtering, and Grouping Operations:

- Sorting: Sort values in ascending or descending order.

- Filtering: Select subsets of data based on conditions.

- Grouping: Aggregate data based on specific criteria.

# Sorting

print(series1.sort\_values()) # Sort in ascending order

# Filtering

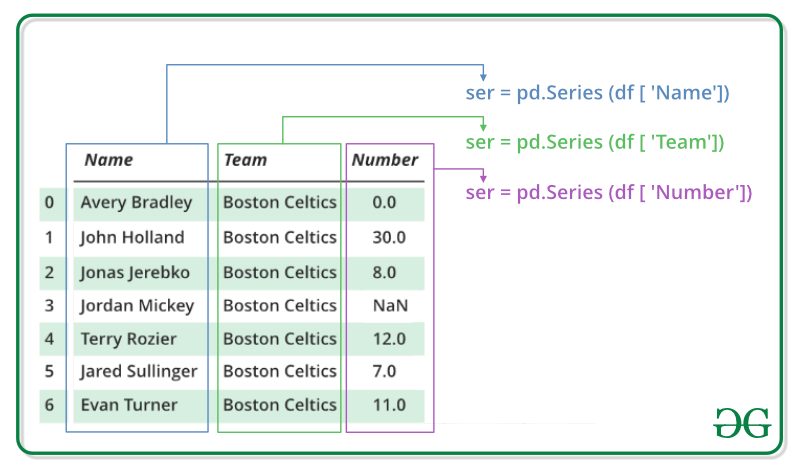
print(series1[series1 > 3]) # Select values greater than 3

# Grouping

grouped = series1.groupby(series1 % 2 == 0).sum() # Group by even and odd numbers

print(grouped)

These operations demonstrate the versatility of pandas Series for data manipulation and analysis. By mastering these operations, you can efficiently analyze and transform data to derive valuable insights for decision-making and problem-solving in various domains.



**DataFrames**

* Creating Pandas DataFrames:

Pandas is a powerful Python library for data manipulation and analysis, and one of its key features is the DataFrame object. A DataFrame is a 2-dimensional labeled data structure with columns of potentially different types. There are several ways to create DataFrames in Pandas, but two common methods are creating them from dictionaries and from lists.

**1. From Dictionaries:**

Creating a DataFrame from a dictionary is a straightforward process. In this method, the keys of the dictionary become the column names, and the values become the column data.

Column Names: The keys of the dictionary become the column names of the DataFrame. These column names must be unique identifiers for each column, and they can be strings or integers.

Data Alignment: When creating the DataFrame, Pandas aligns the data based on the column names. If there are missing values for certain keys in the dictionary, Pandas fills those positions with NaN (Not a Number) values.

Let's take an example to illustrate this:

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],

'Age': [25, 30, 35],

'City': ['New York', 'Los Angeles', 'Chicago']}

df = pd.DataFrame(data)

In this example, the keys of the data dictionary ('Name', 'Age', and 'City') become the column names, and the corresponding lists of values become the data in each column. Pandas automatically aligns the data based on the keys.

**2. From Lists:**

Creating a DataFrame from lists involves specifying column names separately and providing lists of data for each column. Each list corresponds to a row in the DataFrame.

Column Names: When creating a DataFrame from lists, you explicitly specify the column names using the columns parameter. This allows you to customize the column names according to your requirements.

Data Alignment: Similar to creating DataFrames from dictionaries, Pandas aligns the data based on the column names. Each list corresponds to a row, and the elements of the list populate the respective columns. If the lists have different lengths, Pandas fills missing values with NaN.

Creating DataFrames using both methods offers flexibility and control over the resulting DataFrame's structure. Depending on your data source and how it's organized, you can choose the most suitable method for creating DataFrames.

Here's how you can create a DataFrame from lists:

data = [['Alice', 25, 'New York'],

['Bob', 30, 'Los Angeles'],

['Charlie', 35, 'Chicago']]

df = pd.DataFrame(data, columns= ['Name', 'Age', 'City'])

In this example, the data list contains lists representing rows of data. The columns parameter specifies the column names explicitly. This method gives you more control over column names compared to using dictionaries.

Creating DataFrames from dictionaries or lists is just the beginning. Pandas also supports creating DataFrames from various other data sources such as CSV files, Excel files, SQL databases, and more. This flexibility makes Pandas a versatile tool for data manipulation and analysis.

* **Accessing DataFrame Elements:**

Once you have created a DataFrame, you may need to access its elements, whether it's columns, rows, or individual values. Pandas provides several methods for accessing DataFrame elements:

**1. Column Selection:**

Columns in a DataFrame are accessed using their column names. You can think of a DataFrame as a dictionary of Series objects, where each column name is a key mapping to a Series of values.

When you select a single column from a DataFrame, you get a Series object containing the

values of that column. This Series retains the index of the DataFrame, allowing you to align data from different columns.

Accessing a column in a DataFrame is as simple as using square brackets with the column name as the key.

For example:

df['Name']

This returns the entire column with the specified name as a Pandas Series object. You can also use dot notation to access columns if the column names are valid Python identifiers and don't conflict with DataFrame methods or attributes:

df.Name

Both methods return the same result.

**2. Row Selection:**

You can use the loc[] and iloc[] methods to access rows in a DataFrame:

* loc[] is label-based, which means you can access rows using their index labels.
* iloc[] is integer-based, which means you can access rows using their integer location.

Here's how you can use these methods:

# Accessing the first row using label-based indexing

df.loc[0]

# Accessing the first row using integer-based indexing

df.iloc[0]

Both methods return a Series object containing the values of the specified row.

**3. Element Selection:**

Individual elements in a DataFrame can be accessed using the combination of row and column labels or integer positions. This allows you to retrieve specific values from the DataFrame, similar to accessing elements in a 2-dimensional array

By specifying both the row and column labels (or positions), you can pinpoint the exact location of the element you want to retrieve. This level of granularity enables precise data extraction and manipulation.

Accessing DataFrame elements is essential for performing various data analysis tasks, such as filtering, aggregation, and visualization. Having a clear understanding of how to access different parts of the DataFrame enables efficient data exploration and manipulation.

To access individual elements in a DataFrame, you can use the loc[] or iloc[] methods with both row and column indices.

For example:

# Accessing the value in the first row and the 'Name' column

df.loc[0, 'Name']

# Accessing the value in the first row and the first column

df.iloc[0, 0]

These methods allow you to retrieve specific values from the DataFrame based on their positions.

Accessing DataFrame elements is fundamental for data analysis and manipulation tasks. Whether you're extracting specific columns, filtering rows, or retrieving individual values, Pandas provides efficient methods for accessing data in DataFrames.

* **Pandas DataFrame Data Types:**

One of the key features of Pandas DataFrames is that they can contain columns with different data types. When you create a DataFrame, Pandas automatically infers the data types of each column based on the input data. Understanding the data types of DataFrame columns is crucial for data analysis and manipulation tasks.

**Retrieving Data Types:**

Pandas automatically infers the data types of each column when you create a DataFrame. The dtypes attribute allows you to inspect the data types of all columns in the DataFrame. Understanding the data types is crucial for data analysis and manipulation because it determines how the data is stored, processed, and interpreted.

Here's a breakdown of common data types and their implications:

* **int64:** Represents integer values. These can be either signed or unsigned integers with a precision of 64 bits.
* **float64:** Represents floating-point values. These are numbers with decimal points and can represent both integer and fractional parts.
* **object:** Represents string values. This data type is used for columns containing text data, mixed data types, or categorical variables.
* **datetime64:** Represents date and time values. This data type is used for columns containing date and time information, allowing for convenient date/time manipulation and arithmetic.
* **bool:** Represents boolean values (True or False). This data type is used for columns containing binary or categorical data with only two possible values.

Understanding the data types of DataFrame columns is essential for data validation, type conversion, and data analysis. It allows you to perform appropriate operations and transformations on the data while maintaining data integrity and consistency.

**Pandas DataFrame Operations:**

Once you have created a DataFrame and understood how to access its elements and data types, you can perform various operations on the DataFrame to manipulate and analyze the

data. Pandas provides a wide range of functionalities for data manipulation, including descriptive statistics, filtering, sorting, adding new columns, grouping, and aggregation.

**1. Descriptive Statistics:**

Descriptive statistics are used to summarize the main characteristics of a dataset. Pandas provides the describe() method, which calculates summary statistics for numerical columns in the DataFrame. These statistics include count, mean, standard deviation, minimum, 25th percentile, median (50th percentile), 75th percentile, and maximum.

The describe () method provides a quick overview of the data, allowing you to identify outliers, assess data quality, and understand the data distribution.

This method provides valuable insights into the distribution and central tendency of numerical data in the DataFrame, allowing you to understand the data at a glance

**2. Filtering:**

Filtering allows you to select subsets of data from the DataFrame based on specific criteria. You can filter rows using boolean indexing, which involves creating a boolean mask indicating which rows satisfy the given condition.

Filtering is useful for data exploration, data cleaning, and subset selection, enabling you to

focus on relevant portions of the data for further analysis.

For example, to filter rows where the age is greater than 30:

df[df['Age'] > 30]

.

**3. Sorting:**

Sorting allows you to arrange the rows of the DataFrame based on the values in one or more columns. You can use the sort\_values() method to sort the DataFrame by one or more columns, specifying the column(s) to sort by and whether to sort in ascending or descending order.

Sorting is helpful for organizing data, identifying patterns, and facilitating data visualization. It allows you to identify trends, outliers, and relationships within the data.

For example, to sort the DataFrame by age in descending order:

df.sort\_values(by='Age', ascending=False)

This returns a new DataFrame with rows sorted by age in descending order

**4. Adding a New Column:**

You can add a new column to the DataFrame by assigning values to a new column name. This allows you to derive new information from existing columns or external sources and incorporate it into the DataFrame.

Adding a new column enhances the DataFrame's capabilities, enabling further analysis, visualization, and modeling of the data.

For example, to create a new column called 'New Column' with values twice the age:

df['New Column'] = df['Age'] \* 2

This adds a new column to the DataFrame containing values computed from existing columns.

**5. Grouping and Aggregation:**

Grouping involves splitting the DataFrame into groups based on one or more key variables and then applying aggregate functions to each group. Pandas provides the groupby() method for grouping data, allowing you to perform operations such as sum, mean, count, etc., on each group.

Grouping and aggregation are powerful tools for data analysis, enabling you to summarize and compare data across different categories or dimensions.

df.groupby('City').mean()

This groups the DataFrame by city and calculates the mean age for each city.

By combining these operations, you can perform a wide range of data manipulation and analysis tasks, gaining valuable insights from your data.

**Data Cleaning**

What Is Data Cleaning?

When working with multiple data sources, there are many chances for data to be incorrect, duplicated, or mislabeled. If data is wrong, outcomes and algorithms are unreliable, even though they may look correct. *Data cleaning* is the process of changing or eliminating garbage, incorrect, duplicate, corrupted, or incomplete data in a dataset. There’s no such absolute way to describe the precise steps in the data cleaning process because the processes may vary from dataset to dataset. Data cleansing, data cleansing, or data scrub is the general data preparation process initiative. Data cleaning plays an important part in developing reliable answers within the analytical process and is observed to be a basic feature of the info science basics. The motive of data cleaning services is to construct uniform and standardized data sets that enable easy access to data analytics tools and business intelligence and perceive accurate data for each problem.

Why Is Data Cleaning Essential?

Data cleaning is the most important task that should be done by a data science professional. Having wrong or bad-quality data can be detrimental to processes and analysis. Having clean data will ultimately increase overall productivity and permit the very best quality information in your decision-making.



## Data Cleaning Cycle

It is the method of analyzing, distinguishing, and correcting untidy, raw data. Data cleaning involves filling in missing values, handling outliers, and distinguishing and fixing errors present in the dataset. Whereas the techniques used for data cleaning might vary in step with different types of datasets. In this tutorial, we will learn how to clean data using pandas. The following are standard steps to map out data cleaning:

## Data Cleaning CycleData Cleaning With Pandas

Data scientists spend a huge amount of time cleaning datasets and getting them in the form in which they can work. It is an essential skill of Data Scientists to be able to work with messy data, missing values, and inconsistent, noisy, or nonsensical data. To work smoothly, python provides a built-in module, Pandas. Pandas is the popular Python library that is mainly used for data processing purposes like cleaning, manipulation, and analysis. Pandas stand for “Python Data Analysis Library”. It consists of classes to read, process, and write csv files. There are numerous Data cleaning tools present, but the Pandas library provides a really fast and efficient way to manage and explore data. It does that by providing us with Series and DataFrames, which help us represent data efficiently and manipulate it in various ways.

Let’s get started with data cleaning step by step.

To start working with Pandas, we need to first import it. We are using Google Colab as IDE, so we will import Pandas in Google Colab.

**#importing module**

**Import pandas as pd *Step 1: Import Dataset***

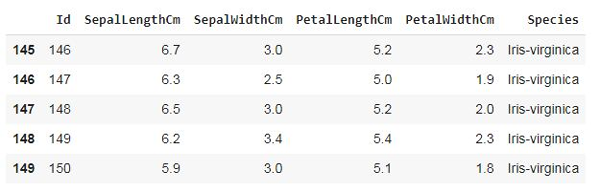
To import the dataset, we use the read\_csv() function of pandas and store it in the pandas DataFrame named as data. As the dataset is in tabular format, when working with tabular data in Pandas, it will be automatically converted into a DataFrame.

DataFrame is a two-dimensional, mutable data structure in Python. It is a combination of rows and columns like an excel sheet.

**Python Code:**

The head() function is a built-in function in pandas for the dataframe used to display the rows of the dataset. We can specify the number of rows by giving the number within the parenthesis. By default, it displays the first five rows of the dataset. If we want to see the last five rows of the dataset, we use the tail()function of the dataframe like this:

#displayinf last five rows of dataset

data.tail()

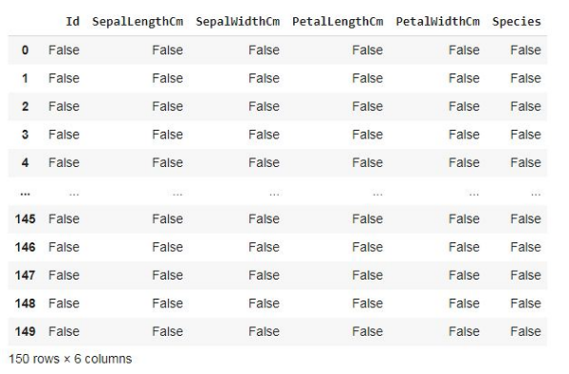
***Step 2: Merge Dataset***

Merging the dataset is the process of combining two datasets in one and lining up rows based on some particular or common property for data analysis. We can do this by using the merge() function of the dataframe. Following is the syntax of the merge function:

**DataFrame\_name.merge(right, how='inner', on=None, left\_on=None, right\_on=None, left\_index=False, right\_index=False, sort=False, suffixes=('\_x','\_y'), copy=True, indicator=False, validate=None**

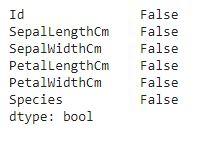
***Step 3: Rebuild Missing Data***

To find and fill in the missing data in the dataset, we will use another function. There are 4 ways to find the null values if present in the dataset. Let’s see them one by one:

**Using isnull() function:** **data.isnull()**

This function provides the boolean value for the complete dataset to know if any null value is present or not.

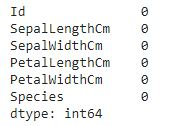
**Using isna() function:** **data.isna()**



This function also gives a boolean value if any null value is present or not, but it gives results column-wise, not in tabular format.

**Using isna(). sum():data.isna().sum()**

This function gives the sum of the null values preset in the dataset column-wise.

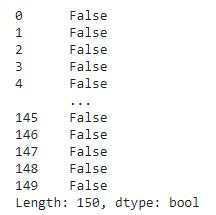
**Using Isna().any().sum():data.isna().any().sum()**

***Step 4: Standardization and Normalization***

Data Standardization and Normalization is a common practices in machine learning.Standardization is another scaling technique where the values are centered around the mean with a unit standard deviation. This means that the mean of the attribute becomes zero, and the resultant distribution has a unit standard deviation.Normalization is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1. It is also known as Min-Max scaling.

***Step 5: De-Duplicate Data***

De-Duplicate means removing all duplicate values. There is no need for duplicate values in data analysis. These values only affect the accuracy and efficiency of the analysis result. To find duplicate values in the dataset, we will use a simple dataframe function, i.e., duplicated(). Let’s see the example:

data.duplicated() This function also provides bool values for duplicate values in the dataset. As we can see, the dataset doesn’t contain any duplicate values. If a dataset contains duplicate values, it can be removed using the drop\_duplicates() function. Following is the syntax of this function:

DataFrame\_name.drop\_duplicates(subset=None, keep='first', inplace=False, ignore\_index=False)

#### Step 6: Verify and Enrich the Data

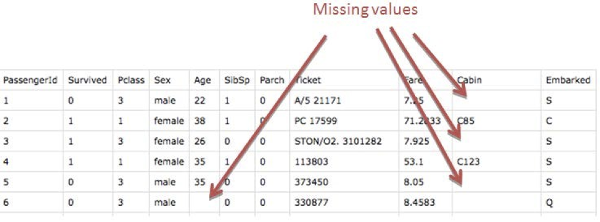
After removing null, duplicate, and incorrect values, we should verify the dataset and validate its accuracy. In this step, we have to check that the data cleaned so far is making any sense. If the data is incomplete, we have to enrich the data again by data gathering activities like approaching the clients again, re-interviewing people, etc. Completeness is a little more challenging to achieve accuracy or quality in the dataset.

*Step 7: Export Dataset*

This is the last step of the data-cleaning process. After performing all the above operations, the data is transformed into a clean dataset,

What Is a Missing Value?

Missing [data](https://www.analyticsvidhya.com/blog/2021/10/end-to-end-introduction-to-handling-missing-values/) is defined as the values or data that is not stored (or not present) for some variable/s in the given dataset. Below is a sample of the missing data from the Titanic dataset. You can see the columns ‘Age’ and ‘Cabin’ have some missing values.



## How to Handle Missing Data?

Missing data is a common headache in any field that deals with datasets. It can arise for various reasons, from human error during data collection to limitations of data gathering methods. Luckily, there are strategies to address missing data and minimize its impact on your analysis. Here are two main approaches:

1. **Deletion:** This involves removing rows or columns with missing values. This is a straightforward method, but it can be problematic if a significant portion of your data is missing. Discarding too much data can affect the reliability of your conclusions.
2. **Imputation:** This replaces missing values with estimates. There are various imputation techniques, each with its strengths and weaknesses. Here are some common ones:
   * **Mean/Median/Mode Imputation**
   * **K-Nearest Neighbors (KNN Imputation)**
   * **Model-based Imputation**

## Why Do We Need to Care About Handling Missing Data?

It is important to handle the [missing values](https://www.analyticsvidhya.com/blog/2021/10/a-complete-guide-to-dealing-with-missing-values-in-python/) appropriately.

* Many machine learning algorithms fail if the dataset contains missing values. However, algorithms like K-nearest and Naive Bayes support data with missing values.
* You may end up building a biased machine learning model, leading to incorrect results if the missing values are not handled properly.

List of Methods to handle missing values in a dataset

Here is a list of popular strategies to handle missing values in a dataset

* Deleting the Missing Values
* Imputing the Missing Values
* Imputing the Missing Values for Categorical Features
* Imputing the Missing Values using Sci-kit Learn Library
* Using “Missingness” as a Feature

What Is Data Imputation?

Data imputation is a method for retaining the majority of the dataset's data and information by substituting missing data with a different value. These methods are employed because it would be impractical to remove data from a dataset each time. Additionally, doing so would substantially reduce the dataset's size, raising questions about bias and impairing analysis. When it comes to data, Pandas is the single most useful library for handling them. After you start working with real-world data and start finding ambiguities in it, no one would come to rescue other than the Pandas library. In this article, we will learn to face missing values in our data and see how we actually deal with them with the help of different imputation techniques.

Need of Imputation

As we have already discussed that real-world data contains a lot of ambiguities, be it data of different types, missing values, etc. Missing values are one ambiguity that needs to be dealt with while working with real-world data in Machine Learning. There are multiple ways you can deal with these ambiguities.

What is the easiest way?

Probably to ignore such rows or columns or drop these columns. The tougher way is to fill in values in these columns. Why do we need to take the tougher road and not go the easy way? We all know that we are working on such huge amounts of data in order to derive a result, make an observation or prediction, etc.If we ignore or drop the rows and columns containing missing values, we might be missing some important test cases, and the prediction with incomplete data has a higher margin of error. This is where the need for imputation arises.

Importance of Data Imputation:

* Distorts Dataset: Large amounts of missing data can lead to anomalies in the variable distribution, which can change the relative importance of different categories in the dataset.
* Unable to work with the majority of machine learning-related Python libraries: When utilizing ML libraries (SkLearn is the most popular), mistakes may occur because there is no automatic handling of these missing data.
* Impacts on the Final Model: Missing data may lead to bias in the dataset, which could affect the final model's analysis.
* Desire to restore the entire dataset: This typically occurs when we don't want to lose any (or any more) of the data in our dataset because all of it is crucial. Additionally, while the dataset is not very large, eliminating a portion of it could have a substantial effect on the final model.

Data Imputation Techniques

After learning about what data imputation is and its importance, we will now learn about some of the various data imputation techniques. These are some of the data imputation techniques that we will be discussing in-depth:

* Next or Previous Value  Missing Value Prediction
* Most Frequent Value K Nearest Neighbors

Average or Linear Interpolation Maximum or Minimum Value

Next/Previous Value Imputation: This method substitutes missing values with the nearest available value in the ordered dataset. It's particularly useful for time-series or ordered data, where nearby values are often more comparable than distant ones.

**K Nearest Neighbors (KNN):** KNN imputation finds the k nearest examples with available values in relevant features and replaces missing values with the most frequent value within that group. It's effective for both nominal and numerical features.

Missing Value Prediction with Machine Learning: Machine learning models can be trained to predict missing values based on other features in the dataset. Various regression or classification models can be employed, depending on the type of feature.

Mean/Median/Rounded Mean Imputation: This technique replaces missing values with the mean, median, or rounded mean of the feature across the entire dataset. Median is preferred over mean when dealing with datasets containing significant outliers.

Fixed Value Imputation: In this approach, missing values are replaced with a predetermined fixed value. For example, in a survey dataset, missing values could be imputed with a label like "not answered." This method is applicable to all data types

**Data transformation**

Data transformation is the process of converting data from one format or structure to another. This can be done for a variety of reasons, such as to make the data easier to analyze, to improve its compatibility with other software, or to meet the requirements of a specific application.

Pandas is a powerful Python library for data analysis and manipulation. It provides a variety of tools for data transformation, including:

Adding and dropping columns: You can use the add() and drop() methods to add and drop columns from a DataFrame.

Filtering rows: You can use the loc() and iloc() methods to filter rows from a DataFrame based on their index or label.

Sorting rows: You can use the sort\_values() method to sort rows in a DataFrame by a specific column.

Grouping rows**:** You can use the groupby() method to group rows in a DataFrame by a specific column or set of columns.

Aggregating data: You can use the agg() method to aggregate data in a DataFrame by group.

Merging dataframes: You can use the merge() method to merge two or more DataFrames together.

Most real-world dataset is dirty. Before analyzing the dataset, you need to transform this dataset. This process is called data transformation. In this post, I’ll talk about data transformation. In short, I’ll cover the following topics:

1. Finding the duplicate values 5. Cutting

2. Mapping 6. Finding the specific values

3. Replacing 7. Selecting

4. Renaming 8. Creating dummy variable

How is data transformation used?

Data transformation works on the simple objective of extracting data from a source, converting it into a usable format and then delivering the converted data to the destination system. The extraction phase involves data being pulled into a central repository from different sources or locations, therefore it is usually in its raw original form which is not usable. To ensure the usability of the extracted data it must be transformed into the desired format by taking it through a number of steps. In certain cases, the data also needs to be cleaned before the transformation takes place. This step resolves the issues of missing values and inconsistencies that exist in the dataset.

The data transformation process is carried out in five stages.

**1. Discovery**

The first step is to identify and understand data in its original source format with the help of data profiling tools. Finding all the sources and data types that need to be transformed. This step helps in understanding how the data needs to be transformed to fit into the desired format.

**2. Mapping**

The transformation is planned during the data mapping phase. This includes determining the current structure, and the consequent transformation that is required, then mapping the data to understand at a basic level, the way individual fields would be modified, joined or aggregated.

**3. Code generation**

The code, which is required to run the transformation process, is created in this step using a data transformation platform or tool.

**4. Execution**

The data is finally converted into the selected format with the help of the code. The data is extracted from the source(s), which can vary from structured to streaming, telemetry to log files. Next, transformations are carried out on data, such as aggregation, format

**5. Review**

The transformed data is evaluated to ensure the conversion has had the desired results in terms of the format of the data.It must also be noted that not all data will need transformation, at times it can be used as is.

Benefits of data transformation

Data holds the potential to directly affect an organization’s efficiencies and its bottom line. It plays a crucial role in understanding customer behavior, internal processes, and industry trends. While every organization has the ability to collect an immense amount of data, the challenge is to ensure that this is usable. Data transformation processes empower organizations to reap the benefits offered by the data.

**Data utilization**

If the data being collected isn’t in an appropriate format, it often ends up not being utilized at all. With the help of data transformation tools, organizations can finally realize the true potential of the data they have amassed since the transformation process standardizes the data and improves its usability and accessibility.

**Data consistency**

Data is continuously being collected from a range of sources which increases the inconsistencies in metadata. This makes organization and understanding data a huge challenge. Data transformation helps making it simpler to understand and organize data sets.

**Better quality data**

Transformation process also enhances the quality of data which can then be utilized to acquire business intelligence.Compatibility across platformsData transformation also supports compatibility between types of data, applications and systems.Faster data accessIt is quicker and easier to retrieve data that has been transformed into a standardized format.More accurate insights and predictions

The transformation process generates data models which are then converted to metrics, dashboards and reports which enable organizations to achieve specific goals. The metrics and key performance indicators help businesses quantify their efforts and analyze their progress. After being transformed, data can be used for many use cases, including:

**Data Manipulation:**

In today's data-driven world, where information is abundant and varied, the ability to manipulate data effectively is a crucial skill. Whether you're a data scientist, analyst, or business professional, understanding how to filter, sort, group, and merge data sets is essential for extracting meaningful insights and making informed decisions. In this comprehensive guide, we will explore each of these techniques in detail, providing explanations, examples, and code snippets to help you master the art of data manipulation.

In Machine Learning, the model requires a dataset to operate, i.e. to train and test. But data doesn’t come fully prepared and ready to use. There are discrepancies like Nan/ Null / NA values in many rows and columns. Sometimes the data set also contains some of the rows and columns which are not even required in the operation of our model. In such conditions, it requires proper cleaning and modification of the data set to make it an efficient input for our model. We achieve that by practicing before giving data input to the model.

Today, we will get to know some methods using Pandas which is a famous library of Python. And by using it we can make out data ready to use for training the model and hence getting some useful insights from the results.

**1.** **Understanding Data Manipulation**

At its core, data manipulation involves the process of transforming raw data into a structured format that is conducive to analysis. This encompasses a myriad of tasks, including cleaning and preprocessing data, aggregating and summarizing information, as well as merging and reshaping datasets. Data manipulation serves as a crucial precursor to data analysis, enabling practitioners to derive meaningful insights and make informed decisions.

**2. Essential Techniques of Data Manipulation**

a. Filtering Data: Filtering involves selecting specific subsets of data based on predefined criteria. Whether it's removing outliers, isolating relevant information, or excluding unnecessary data points, filtering is indispensable for refining datasets.

This technique allows you to focus on relevant information while excluding irrelevant or unnecessary data points. Let's delve deeper into various methods and tools for filtering

•Filtering with Boolean Conditions: Boolean conditions enable you to specify logical expressions to determine whether a particular data point should be included in the filtered subset. For instance, in Python using pandas:

Python Code:

filtered\_data = df[df['Column'] > 10]

This code snippet selects rows where the value in the 'Column' is greater than 10.

b. Sorting Data:

in a specific order based on one or more attributes. This technique facilitates easier analysis and visualization of data. Let's explore some common methods for sorting data:

•Ascending and Descending Order: Sorting data in ascending order arranges values in increasing order, while sorting in descending order arranges them in decreasing order. For example:

Python code:

sorted\_data = df.sort\_values(by='Column', ascending=True)This code sorts the DataFrame 'df' based on the values in 'Column' in ascending order.

Multiple Column Sorting: Sorting data based on multiple attributes allows for a more nuanced analysis. For instance:

Python Code:

sorted\_data = df.sort\_values(by=['Column1', 'Column2'], ascending=[True, False])

This code sorts the DataFrame 'df' first by 'Column1' in ascending order, then by 'Column2' in descending order.

c. Grouping Data: Grouping involves aggregating rows of data that share common attributes into summary rows. By grouping data, practitioners can perform calculations and analysis at a higher level of abstraction, gaining insights into underlying trends and patterns.

Grouping data involves aggregating rows of data that share common attributes into summary rows. This technique facilitates higher-level analysis and summarization. Let's examine various methods for grouping data

Grouping by Single Column: Grouping data based on a single attribute allows for the creation of distinct groups or categories within the dataset. For example:

Python Code

grouped\_data = df.groupby('Column').sum()

This code groups the DataFrame 'df' by values in 'Column' and calculates the sum of other columns for each group.

Grouping by Multiple Columns: Grouping data based on multiple attributes enables the creation of nested or hierarchical groups. For instance:

Python Code

grouped\_data = df.groupby(['Column1', 'Column2']).mean()

This code groups the DataFrame 'df' by values in both 'Column1' and 'Column2', calculating the mean of other columns for each group.

d. Merging and Joining Data: Merging and joining data entail combining multiple datasets into a single dataset based on common attributes or keys. This technique is essential for integrating data from disparate sources, enabling comprehensive analysis and decision-making.

Let's explore different types of joins and their applications:

Inner Join: An inner join returns only the rows that have matching values in both datasets. For example:

Python Code

merged\_data = pd.merge(df1, df2, on='Key', how='inner')

This code performs an inner join between 'df1' and 'df2' based on the common key 'Key'.

Left Join: A left join returns all rows from the left dataset and the matching rows from the right dataset. For instance:

Python Code

merged\_data = pd.merge(df1, df2, on='Key', how='left')

This code performs a left join, retaining all rows from 'df1' and matching rows from 'df2' based on the common key 'Key'.

**3. Tools for Data Manipulation**

a. Python: Python, with its rich ecosystem of libraries such as Pandas and NumPy, has emerged as a powerhouse for data manipulation. Pandas, a powerful library built on top of NumPy, offers versatile data structures and functions for manipulating structured data. With Pandas, practitioners can easily filter, sort, group, and merge datasets using intuitive and expressive syntax. For example, filtering data based on specific conditions can be achieved with a single line of code using Pandas' DataFrame operations.

**Python Code:**

import pandas as pd

# Load data

df = pd.read\_csv('data.csv')

# Filter data

filtered\_data = df[df['column\_name'] > threshold\_value]

b. R: R is another popular programming language for data manipulation and analysisPackages like dplyr and tidyr provide a wide array of functions for manipulating and reshaping data. For instance, dplyr's filter() function allows users to select rows of data based on specified conditions, while the group\_by() and summarize() functions enable grouping and summarizing data efficiently.

**R Code:**

# Load libraries

library(dplyr)

# Load data

data <- read.csv('data.csv')

# Filter data

filtered\_data <- data %>% filter(column\_name > threshold\_value)

c. SQLStructured Query Language (SQL) remains a staple for data manipulation tasks, particularly for working with relational databases. SQL enables practitioners to perform a range of operations such as filtering, sorting, grouping, and joining tables. With SQL, users can write declarative queries to extract and manipulate data directly from databases.

**Sql Code:**

-- Filter data

SELECT \*

FROM table\_name

WHERE column\_name > threshold\_value;

**4. Best Practices for Effective Data Manipulation**

a. Data Cleaning: Data cleaning is a crucial prelude to data manipulation. It involves identifying and addressing issues such as missing values, outliers, and inconsistencies in the dataset. By cleaning the data upfront, practitioners can ensure the integrity and reliability of subsequent data manipulation tasks.

b. Modularization: Breaking down data manipulation tasks into smaller, modular components enhances code readability, maintainability, and reusability. By encapsulating specific data manipulation operations into functions or modules, practitioners can streamline their workflows and facilitate collaboration with team members.

c. Documentation: Documenting data manipulation workflows is essential for transparency and reproducibility. Documenting assumptions, transformations, and decisions made during the data manipulation process helps stakeholders understand the rationale behind data transformations and ensures that analyses can be replicated and verified.

d. Performance Optimization: Optimizing data manipulation workflows for efficiency and scalability is paramount, especially when dealing with large datasets. Techniques such as using appropriate data structures, leveraging parallel processing, and optimizing queries can significantly improve the performance of data manipulation tasks and reduce computational overhead.

**Data Analysis**

**1. Basic Statistical Analysis:**

Basic statistical analysis forms the cornerstone of data exploration, providing fundamental insights into the characteristics and distribution of the dataset. In this section, we delve into various statistical measures to uncover patterns and trends within the data.

**Measures of Central Tendency:**

Central tendency measures such as mean, median, and mode offer insights into the typical or central value of the dataset. The mean provides the arithmetic average, the median represents the middle value, and the mode signifies the most frequently occurring value. By computing these measures, we gain a holistic understanding of the central tendencies exhibited by the

**Measures of Variability:**

Understanding the variability within the dataset is crucial for assessing the spread or dispersion of data points. Standard deviation and variance quantify the degree of dispersion around the mean, highlighting the consistency or variability of observations. Analyzing these measures helps identify the range within which most data points lie and assesses the level of homogeneity or heterogeneity present in the dataset.

**Distribution Analysis:**

Visualizing the distribution of data through histograms, box plots, and density plots provides valuable insights into its shape and characteristics. Histograms illustrate the frequency distribution of numerical data, while box plots depict the quartiles, median, and potential outliers. Density plots offer a smoothed representation of the data distribution, aiding in identifying underlying patterns or deviations from normality.

**Correlation Analysis:**

Examining the relationships between variables through correlation analysis elucidates the degree and direction of association. Pearson correlation coefficient quantifies the linear relationship between two continuous variables, ranging from -1 to 1. A coefficient close to 1 indicates a strong positive correlation, while a value near -1 denotes a strong negative correlation. Understanding these correlations facilitates the identification of interdependencies and informs subsequent analytical approaches.

**Hypothesis Testing:**

Hypothesis testing enables us to make inferences about population parameters based on sample data. By formulating null and alternative hypotheses and conducting statistical tests such as t-tests or ANOVA, we assess the significance of observed differences or relationships within the data. This rigorous analytical approach provides statistical evidence to support or refute hypotheses, guiding decision-making processes.

**2. Time Series Analysis:**

Time series analysis focuses on understanding and forecasting sequential data points recorded over regular time intervals. By identifying patterns, trends, and seasonality within the time series data, we gain valuable insights into temporal dynamics and future projections.

**Trends Identification:**

Detecting trends within time series data is essential for uncovering underlying patterns or directional movements. Moving averages, such as simple moving average (SMA) or exponential moving average (EMA), smoothen out fluctuations to reveal long-term trends. Trend analysis enables us to discern upward, downward, or stationary trends, guiding strategic decision-making and resource allocation.

**Seasonality Detection:**

Seasonality refers to recurring patterns or fluctuations that occur at fixed intervals within the time series data. Seasonal decomposition techniques, such as seasonal-trend decomposition using LOESS (STL), decompose the time series into seasonal, trend, and residual components. Autocorrelation analysis assesses the similarity between observations at different time lags, elucidating the presence and magnitude of seasonal effects.

**Forecasting Techniques:**

Forecasting future values based on historical data is integral to informed decision-making and planning. Time series forecasting methods, including Autoregressive Integrated Moving Average (ARIMA) and seasonal ARIMA (SARIMA), model the sequential dependencies and seasonal variations present in the data. Exponential smoothing methods, such as simple exponential smoothing (SES) or Holt-Winters method, provide robust forecasting capabilities by incorporating trend and seasonality components.

**Evaluation Metrics:**

Assessing the accuracy and reliability of forecasting models requires the use of appropriate evaluation metrics. Mean absolute error (MAE), mean squared error (MSE), and root mean squared error (RMSE) quantify the magnitude of prediction errors, providing insights into model performance. Additionally, measures such as mean absolute percentage error (MAPE) and forecast bias assess the relative accuracy and directional bias of forecasts, facilitating model selection and refinement.

**3. Descriptive Statistics:**

Descriptive statistics offer a concise summary of the main features and characteristics of the dataset, facilitating data exploration and interpretation. By quantifying central tendencies, dispersion, and shape of the data distribution, descriptive statistics provide valuable insights into the underlying data structure.

**Measures of Central Tendency:**

Central tendency measures, including mean, median, and mode, provide a central reference point or typical value around which data points cluster. The mean represents the average value, the median denotes the middle value, and the mode signifies the most frequently occurring value. Analyzing these measures offers a comprehensive understanding of the central tendencies exhibited by the data.

**Measures of Dispersion:**

Dispersion measures, such as standard deviation, variance, and range, quantify the degree of spread or variability within the dataset. Standard deviation measures the average deviation of data points from the mean, while variance represents the average squared deviation. Range reflects the difference between the maximum and minimum values, providing a simple measure of data spread. Evaluating these measures enables us to assess the variability and consistency of observations.

**Shape of the Data Distribution:**

Visualizing the distribution of data through histograms, density plots, or quantile-quantile (Q-Q) plots elucidates its shape and characteristics. Symmetric distributions exhibit equal frequencies on both sides of the mean, while skewed distributions display asymmetric patterns. Kurtosis measures the degree of peakedness or flatness of the distribution, indicating the presence of outliers or extreme values. Analyzing the shape of the data distribution informs subsequent analytical approaches and modeling techniques.

**Interquartile Range (IQR) and Outliers:**

The interquartile range (IQR) represents the range within which the middle 50% of data points lie, offering a robust measure of data spread. Outliers, or extreme observations, fall outside the typical range of values and may indicate data anomalies or measurement errors. Detecting and managing outliers is essential for ensuring the accuracy and reliability of statistical analyses, as they can significantly influence summary statistics and inferential outcomes.

**4. Data Visualization:**

Data visualization is a powerful tool for conveying complex information and uncovering hidden patterns within the dataset. By transforming raw data into visually appealing charts, graphs, and maps, we enhance comprehension and facilitate decision-making processes.

**Histograms:**

Histograms represent the frequency distribution of numerical data by dividing the data range into intervals or bins and plotting the frequency of observations within each bin. Histograms provide insights into the shape, central tendencies, and variability of the data distribution, aiding in the identification of patterns or outliers.

**Scatter Plots:**

Scatter plots visualize the relationship between two continuous variables by plotting individual data points on a two-dimensional graph. Scatter plots reveal patterns such as linear, non-linear, or no correlation between variables, enabling the assessment of associations and dependencies. Additionally, trend lines or regression lines can be added to highlight underlying trends or directional movements within the data.

**Line Charts:**

Line charts depict trends or patterns over time by connecting data points with straight or curved lines. Line charts are particularly useful for visualizing time series data and illustrating temporal dynamics, such as trends, seasonality, or cyclical patterns.

**Data Input/Output**

Reading data into pandas DataFrames is often the first step when conducting data analysis in Python. The pandas package comes equipped with several data reading and writing functions that let you read data directly from common file formats like comma separated values files (CSV) and Microsoft Excel files. This lesson will focus on reading and writing data from these common file formats, but Python has packages available to work with just about every data format you encounter.

Python Working Directory and File Paths

Before we can jump into reading and writing data, we need to learn a little bit about Python's working directory and file paths. When you launch Python, it starts in a default location in your computer's file system (or the remote computer you are using) known as the working directory. You can check your current working directory by importing the os module and then using os.getcwd():

import os

os.getcwd()

'/kaggle/working'

The working directory acts as your starting point for accessing files on your computer from within Python. To load a data set, you either need to put the file in your working directory, change your working directory to the folder containing the data or supply the data file's file path to the data reading function.

You can change your working directory by supplying a new file path in quotes to the os.chdir() function:

os.chdir('/kaggle/')

os.getcwd()

'/kaggle'

You can list all of the objects in a directory by passing the file path to the os.listdir( ) function:

linkcode

os.listdir('/kaggle/input')

['titanic', 'draft2015']

Reading CSV and TSV Files

Data is commonly stored in simple flat text files consisting of values delimited(separated) by a special character like a comma (CSV) or tab (TSV).

You can read CSV files into a pandas DataFrame using the pandas function pd.read\_csv():

linkcode

import pandas as pd

titanic\_train = pd.read\_csv('input/titanic/train.csv')

titanic\_train.head(6)

The Internet gives you access to more data than you could ever hope to analyze. Data analysis often begins with getting data from the web and loading it into Python. Websites that offer data for download usually let you download it as CSV, TSV or excel files. Perhaps the easiest way load web data, is to simply download data to your hard drive and then use the functions we discussed earlier to load it into a DataFrame.

If you are running Python locally, reading from the clipboard is another quick and dirty option for reading web data and other tabular data. To read data from the clipboard, highlight the data you want to copy and use the appropriate copy function on your keyboard (typically control + C) as if you were going to copy and paste the data. Next, use the pd.read\_clipboard() function with the appropriate separator to load the data into a pandas DataFrame. Since we are using Kaggle's kernel environment for this guide we will not be reading from the clipboard.

Pandas also comes with a read\_html() function to read data directly from web pages. To use read\_html() you need the HTML5lib package. Install it by opening a command console and running "pip install HTLM5lib" (without quotes). Note that HTML can have all sorts of nested structures and formatting quirks, which makes parsing it to extract data troublesome. The read\_html() function does its best to draw out tabular data in web pages, but the results aren't always perfect. Again, since we are using the Kaggle kernel environment for this guide, we won't be using read\_html() as it does not seem to play well with Kaggle's notebook environment. When it comes to using outside data on Kaggle, your best bet is to download the data to your local machine and then upload it to Kaggle as a dataset that you can add to your project.

Data comes in all sorts of formats other than the ones we've discussed here. The pandas library has several other data reading functions to work with data in other common formats, like json, SAS and stata files and SQL databases.

**Writing Data**

Each of the data reading functions in pandas has a corresponding writer function that lets you write data back to into the format it came from. Most of the time, however, you'll probably

want to save your data in an easy-to-use format like CSV. Write a DataFrame to CSV in the working directory by passing the desired file name to the df.to\_csv() function:

linkcode

draft.to\_csv("draft\_saved.csv")

os.listdir('/kaggle/')

['input', 'draft\_saved.csv', 'lib', 'working', 'src']   
  
The collection of reader functions that can be used with Pandas.read\_csv and typically return pandas objects are called the pandas I/O API. Because of it, data can be loaded from files. Using series objects and DataFrame, data can be written on the clipboard. There are several methods for reading pandas files for CSV w have .read\_csv, for excel we have .read\_excel etc.

What is I/O API in Pandas?

The pandas I/O API is a collection of top-level reader functions that can be used with "pandas.read csv()" and typically return a pandas object. Similar to DataFrame .to\_csv, the corresponding writer functions are object methods (). The API for saving the contents of Series and DataFrame objects to various types of objects, including the clipboard and files, is called Pandas IO Tools. It also makes it possible to load data from files, objects, or the clipboard.

**Write Files**

Data and labels can be written to the clipboard or files using the methods available in Series and DataFrame objects. They have the naming convention . to file-type>(), where file-type> denotes the destination file's type like .to\_csv(), .to\_excel(), .to\_json() etc.

These methods provide parameters that specify the destination file directory where the data and labels were saved. It depends on the situation if this is required or optional. If this option is offered and you decide not to use it, the methods return objects (such as strings or iterables) that include the data from DataFrame instances.

Compression, an optional parameter, controls how the file with the data and labels will be compressed.

There are a few additional parameters, but most of them are unique to one or more methods.

**Read Files**

Pandas functions for reading file contents have names that follow the pattern ".read file-type>()," where "file-type>" denotes the type of the file to be read like read\_csv(), read\_excel(), read\_json(), read\_html() etc.

The path to the destination file is specified as a parameter for these functions. Whether on a local machine or in a URL, it can be any valid string that represents the path. Depending on the type of file, more items are also permitted.The compressed files' type of decompression will be determined by the optional parameter compression. There are additional arguments, but they are exclusive to one or more functions.

Working with Different File Types

The Pandas library provides a variety of options for loading data from files and saving data to files. Let's see how to use JSON, web pages, databases, and Python pickle files, among other file types.

**CSV**

CSV files(comma-separated files) are a simple way to store big data. Pandas can read CSV files since they are a widely used format that only contains plain text. csv file can be read by using .read\_csv(file\_path). Using the data.csv file that you gave as the first input, the Pandas read CSV()function returns a new DataFrame containing the information from that file. A valid path, including URLs, may be contained in this string.

Let's look at an example as follows:

**Code:**

import pandas as pd

data=pd.read\_csv("https://raw.githubusercontent.com/codeforamerica/ohana-api/master/data/sample-csv/addresses.csv")

print(data.head())

**Output:**

id location\_id address\_1 address\_2 city \

0 1 1 2600 Middlefield Road NaN Redwood City

1 2 2 24 Second Avenue NaN San Mateo

2 3 3 24 Second Avenue NaN San Mateo

3 4 4 24 Second Avenue NaN San Mateo

4 5 5 24 Second Avenue NaN San Mateo

state\_province postal\_code country

0 CA 94063 US

1 CA 94401 US

2 CA 94403 US

3 CA 94401 US

4 CA 94401 US

Explanation: In the above code example, Pandas are imported as pd. Here we use a GitHub file link for data and reading this CSV file using the pd.read\_csv() function. .head() is used to read only the starting rows of the data, by default starting 5 rows.

**JSON**

JSON stands for Javascript object Notation. It indicates that the data is stored and sent using a script file, which is composed of text written in a programming language. A pre-built module called JSON in Python provides support for JSON. The Python script imports the JSON package to make use of this feature. Text in JSON is represented as a quoted string that has the value from a key-value mapping inside. It is comparable to Python's dictionary.

**HTML**

The Python language may be used today to make our lives easier and has great applications in practically every sector. Getting the data output in an HTML file is one such Python application.

Creating an Html file and saving the input data into a file.

\* Creating an HTML file. Function\_Name = open("Complete\_File\_Name","File\_operation")

\* Adding input data in HTML format into the file.Function\_Name.write("Adding\_Input\_data\_using\_HTML\_Synatx\_separted\_by\_/n")

\* Saving the HTML file.Function\_Name.close()

Opening the HTML file from the saved location by using the following code.

Code:

Funcn = open("Scalertopics.html", "w")

# Adding input data to the HTML file

Funcn.write("<html>\n<head>\n<title> \nOutput Data in an HTML file using python \

</title>\n</head> <body><h1>Welcome to <u>Scaler By InterviewBit</u></h1>\

\n<h2> An online transformative <u>upskilling platform</u> for working tech professionals</h2> \n</body></html>")

# Saving the data into the HTML file

Funcn.close()

After creating the Html file, it is saved in the directory. I am using google colab here, so the file is saved in google colab. If you are using any of the python environments like pycharm, anaconda, Jupyter notebook, etc. then your created Html file will be stored on your computer.

**Output:**

Html file looks like:

To read the Html tables we have pandas built-in function .read\_html(). Let's see how it works.

Code:

import pandas as pd

Html\_Data = """

<table>

<thead>

<tr>

<th>S.No.</th>

<th>name</th>

<th>English\_Marks</th>

<th>Maths\_Marks</th>

</tr>

</thead>

<tbody>

<tr>

<td>1</td>

<td>Robert</td>

<td>91</td>

<td>86</td>

</tr>

<tr>

<td>2</td>

<td>Alice</td>

<td>84</td>

<td>79</td>

</tr>

<tr>

<td>3</td>

<td>Harry</td>

<td>94</td>

<td>93</td>

</tr>

</tbody>

</table>

"""

df = pd.read\_html(Html\_Data)

print("Type: ",type(df))

print()

print(df)

**Output:**

Type: <class 'list'>

[ S.No. name English\_Marks Maths\_Marks

0 1 Robert 91 86

1 2 Alice 84 79

2 3 Harry 94 93]

Explanation In the above code, pandas is imported as pd. An HTML code for creating tables is written and stored in Html\_Data.Then using the pd.read\_html() function, an HTML file can be read. type(df) is used to check the file type that is created.

SQL

For working with saved data in SQL we have to perform some complex queries. Pandas are specially built for data pre-processing and are more user-friendly than SQL . For this python has an inbuilt function to read the sql database pandas.read\_sql.

See this with the help of an example.

Code:

from sqlite3 import connect

import pandas as pd

connectivity = connect(':memory:')

df = pd.DataFrame(data=[[0, '10/09/22'], [1, '11/09/22']],

columns=['int\_column', 'date\_column'])

(df.to\_sql('Data\_File', connectivity))

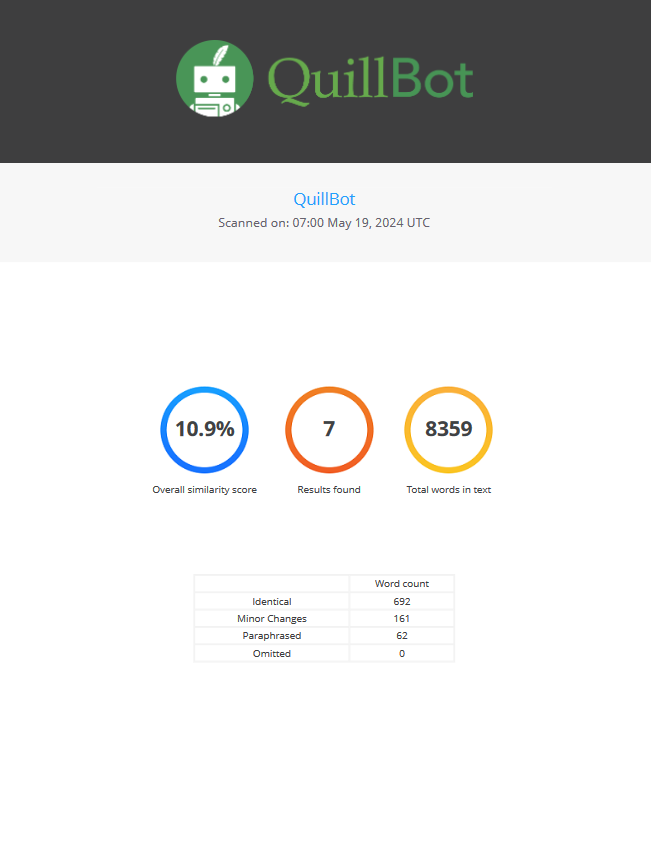
pd.read\_sql('SELECT int\_column, date\_column FROM Data\_File', connectivity)

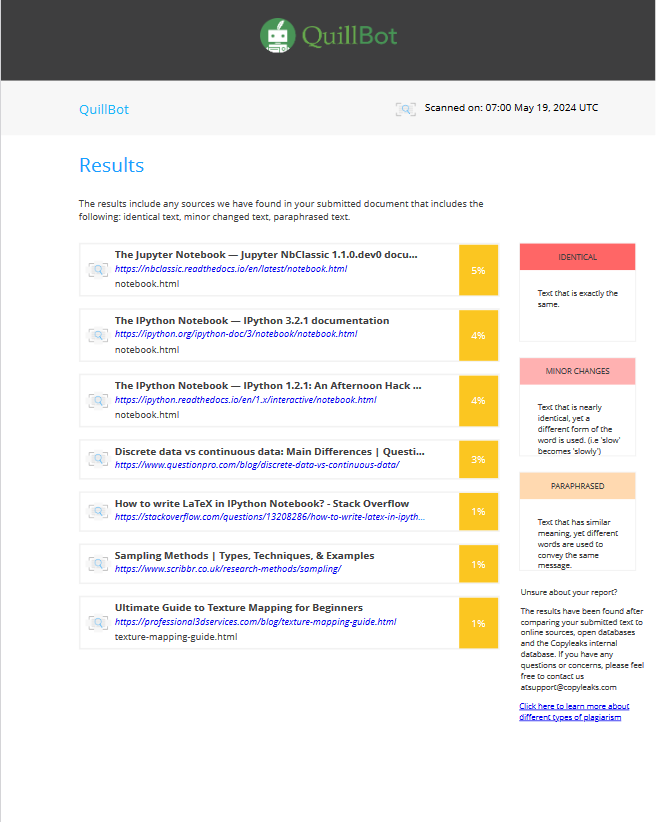
Output:

int\_column date\_column

0 0 10/09/22

1 1         11/09/22

****

****

**Reference**

McKinney, W. (2010). Data Structures for Statistical Computing in Python. In Proceedings of the 9th Python in Science Conference (pp. 51-56). Link to paper

Wes McKinney. (2013). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. O'Reilly Media. ISBN: 978-1-449-31979-3.

Pandas Documentation. (n.d.). Data Structures. Pandas Documentation - Data Structures

VanderPlas, J. (2016). Python Data Science Handbook: Essential Tools for Working with Data. O'Reilly Media. ISBN: 978-1-491-91205-8.

Chen, D., Li, T., Ross, N., & Zhu, Y. (2020). Pandas for Everyone: Python Data Analysis. Addison-Wesley. ISBN: 978-0134546933.

Jones, E., Oliphant, T., & Peterson, P. (2001). SciPy: Open source scientific tools for Python. SciPy Documentation

Raschka, S. (2015). Python Machine Learning. Packt Publishing Ltd. ISBN: 978-1783555130.

Ramalho, L. (2015). Fluent Python: Clear, Concise, and Effective Programming. O'Reilly Media. ISBN: 978-1491946008.

Mueller, A., & Massaron, L. (2015). Python for Data Science For Dummies. John Wiley & Sons. ISBN: 978-1118844182.

Heydt, M. (2018). Learning pandas: Python Data Discovery and Analysis Made Easy. Packt Publishing. ISBN: 978-1788620956.

Millman, K. J., & Aivazis, M. (2011). Python for Scientists and Engineers. Computing in Science & Engineering, 13(2), 9-12.

Grus, J. (2019). Data Science from Scratch: First Principles with Python. O'Reilly Media. ISBN: 978-1-492-03790-6.

Brownlee, J. (2020). "How to Use Pandas for Data Analysis." Machine Learning Mastery.

Deitel, P., & Deitel, H. (2020). Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data, and The Cloud. Pearson. ISBN: 978-0-134-05416-4.

Chen, D., Li, T., Ross, N., & Zhu, Y. (2020). Pandas for Everyone: Python Data Analysis. Addison-Wesley. ISBN: 978-0134546933.

Jones, E., Oliphant, T., & Peterson, P. (2001). SciPy: Open source scientific tools for Python. SciPy Documentation.

Raschka, S. (2015). Python Machine Learning. Packt Publishing Ltd. ISBN: 978-1783555130.

Ramalho, L. (2015). Fluent Python: Clear, Concise, and Effective Programming. O'Reilly Media. ISBN: 978-1491946008.

Mueller, A., & Massaron, L. (2015). Python for Data Science For Dummies. John Wiley & Sons. ISBN: 978-1118844182.

Heydt, M. (2018). Learning pandas: Python Data Discovery and Analysis Made Easy. Packt Publishing. ISBN: 978-1788620956.

Millman, K. J., & Aivazis, M. (2011). Python for Scientists and Engineers. Computing in Science & Engineering, 13(2), 9-12.

Brownlee, J. (2020). "How to Use Pandas for Data Analysis." Machine Learning Mastery.

Deitel, P., & Deitel, H. (2020).