Pandas Demystified: A Comprehensive Handbook for Data Enthusiasts

python.plainenglish.io/pandas-demystified-a-comprehensive-handbook-for-data-enthusiasts-part-1-136127e407f

RaviTeja G September 29, 2023

Part 1: Exploring Essential Functions for Effective Data Analysis

Image Generated with Ideogram.ai

Welcome to the world of Pandas, we all have worked with Excel at some point, right? Working with pivot tables and exploring the data! Now, Imagine being able to effortlessly shape, transform, and analyze your data. That's precisely what Pandas, the Python library, brings to the table.

In this comprehensive guide, we'll dive deep into Pandas, exploring its various features and functionalities. You'll learn how to wield Pandas to your advantage, turning raw data into valuable knowledge. Whether you're a seasoned data scientist or just beginning your data science odyssey, the Pandas Library is your trusty wand for your data cleaning and data manipulations. Buckle UP!

Table of Contents

> Let's get started on this data-driven adventure!

1. Data Structures

Pandas provides two fundamental data structures: **Series and DataFrame, which are the building blocks of data manipulation and analysis in Python.** Understanding these data structures is essential for effective data handling with Pandas.

1.1 Series

A Series is a one-dimensional labeled array that can hold various data types, such as integers, floats, strings, or even custom objects. It's similar to a column in an Excel spreadsheet or a single column in a SQL table. Key features of the Series include:

- Each element in a Series has a label or an index, which allows for easy access and manipulation of data.

- Unlike lists in Python, Series typically stores data of the same data type, ensuring consistency.
- You can perform vectorized operations on Series, making it efficient for element-wise calculations. This feature allows you to efficiently perform operations on entire columns or Series without the need for explicit loops. You can add, subtract, and multiply the series(columns of a data frame) with a series or scalar.

Creating a Series:

```
import pandas as pd
data = [1, 2, 3, 4, 5]
series = pd.Series(data, name="MySeries")

series = series *
```

1.2 DataFrame

A DataFrame is a two-dimensional, tabular data structure with labeled axes (rows and columns). **It resembles a spreadsheet or an SQL table** and is the primary data structure for data analysis in Pandas. Key features of DataFrames include:

- Each column in a DataFrame is a Series, which means it can hold different data types.
- DataFrames have both row and column indexes, allowing for flexible data selection.
- Like Series, DataFrames can align data based on labels, making operations easy and intuitive.
- You can merge, join, and concatenate DataFrames to combine and analyze data from various sources.

Creating a Data Frame and Renaming the columns

```
import pandas as pd
data = {
"Name": ["Alice", "Bob", "Charlie", "David"],
"Age": [25, 30, 35, 40],
"City": ["New York", "San Francisco", "Los Angeles", "Chicago"]
}
df = pd.DataFrame(data)

   df.rename(columns={:, inplace=}
```

Understanding these two core data structures sets the foundation for efficient data manipulation and analysis using Pandas. They enable you to load, clean, explore, and transform data in various ways, making Pandas a powerful tool in the data scientist's toolkit.

2. Data Loading and Data Inspection

Pandas provides a wide range of functions and methods for efficiently loading data **from various sources and formats into DataFrames.**

2.1 Reading Data from Different Sources (CSV, Excel, SQL, etc.)

```
import pandas as pd

# Load data from csv file with the name - data.csv
df_csv = pd.read_csv('data.csv')

# Load data from excel file with the name - data.xlsx
df_excel = pd.read_excel('data.xlsx')

# You can specify a specific sheet using the sheet_name parameter
df_sheet1 = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# Create a SQLAlchemy engine
from sqlalchemy import create_engine
engine = create_engine('sqlite:///mydatabase.db')

# Load data from a SQL database table
query = 'SELECT * FROM mytable'
df_sql = pd.read_sql_query(query, engine)

# Load data from an HTML table on a webpage
url = 'https://example.com/data-table.html'
df_html_table = pd.read_html(url)

# Load data from a JSON file
df_json = pd.read_json('data.json')

df_parquet = pd.read_parquet()
```

Important Parameters to keep in mind for commonly used read_csv:

1. `filepath` specifies the path to the CSV file you want to read. You can provide a file path (as a string), a URL, or a file-like object.
2. To Override column names, You have to use `names` a parameter with the list of new column names.
3. `sep` stands for "separator" and defines the character used to separate fields in the CSV file. The default is a comma (`,`), but you can specify other characters such as tabs (`'\t'`), semicolons (`';'`), or any custom delimiter.
4. The `index_col` parameter specifies which column(s) should be used as the data frames index. You can pass either a column name or column index (0-based) to this parameter.
5. `skiprows` allows you to skip a specific number of rows at the beginning of the CSV file. It can be useful if there are metadata or comments at the start of the file that you want to ignore.

2.2 Displaying DataFrames

It’s nice that we loaded the data, but how do we see it, right? Displaying a data frame is the first step in understanding its contents. **you can just type the data frame name** and execute the cell to see the top 5 and bottom 5 rows. And Pandas offers several othermethods to display different portions of your data frame:

- **`.head(n)`** This method displays the first `n` rows of the data frame. It's useful for getting a quick overview of the data's structure without overwhelming yourself with too much information, or if you just want to see the column names, you can use `.columns`
- **`.tail(n)`** Similar to `.head()`, this method shows the last `n` rows of the DataFrame. It's handy for checking the end of the dataset.
- **`.sample(n)`** If you want to see random rows from the DataFrame, use this method. This is useful for exploring diverse parts of the dataset.

2.3 Data Exploration: shape, info, describe

Pandas provides methods for obtaining fundamental insights into your data. These are the first things you need to check while exploring your data.

- **`.shape`** This function gives a set where the first element specifies the no.of samples/rows in the data and the second element specifies the no.of columns.
- **`.info()`** This method provides a concise summary of the DataFrame, including the data types, non-null counts, and memory usage.It's an excellent starting point for understanding the data's structure, or if you just want to see the data types, you can use `.dtypes`
- **`.describe()`**The methodgenerates basic statistics for each numeric columnin the DataFrame, such as count, mean, standard deviation, minimum, and maximum values.

2.4 Unique Values, Value Counts, and Basic Statistics

For categorical or discrete data, you can explore unique values and their frequencies:

- **`.nunique()`** This method calculates the number of unique values in each column. It's handy for understanding the diversity of data in categorical columns.
- **`.column_name or ['column_name']`**To access a specific column in the DataFrame. You can only use the second approach when the column name has spaces.

```
df.Citydf[]
```

`.value_counts()` Use this method on a specific column to count the occurrences of each unique value. It's particularly useful for categorical columns.

```
# By default value counts will not return the count of missing values
df['City'].value_counts()

df[].value_counts(dropna=)
```

You can calculate additional statistics for specific columns, such as the sum, max, min, mean, median, or mode, using Pandas' mathematical functions:

```
df.Age.mean() df.Salary.median() df.Gender.mode()
```

3. Data Selection and Indexing

Data selection and indexing are fundamental operations in Pandas, allowing you to extract specific subsets of data from a DataFrame.

3.1 Selecting Columns and Rows

You can select specific columns and rows from a DataFrame using square brackets `[]`, `.loc[]`, and `.iloc[]` indexing methods:

`[]` To select one or more columns by their names, you can use square brackets with the column names as a list.

```
selected_columns = df[[, ]]
```

`.loc[]` The `.loc[]` indexer allows you to select rows and columns by label. You can specify both row and column labels.

```
selected_data = df.loc[:, [, ]]
```

If you specify multiple rows or columns using index slicing, the inner and outer indices both are inclusive. Hence, 3,4,5,6 all the rows are included.

`.iloc[]` The `.iloc[]` indexer lets you select rows and columns by integer location, which is useful for numeric indexing.

```
selected_data = df.iloc[:, :]
```

If you specify multiple rows or columns using index slicing, only the inner is inclusive, and the outer is exclusive. Hence only 1,2,3 rows will be shown and 0,1 columns will be shown.

3.2 Filtering / Conditional Selection

Conditional selection enables you to filter rows based on specific criteria. You can use boolean indexing to achieve this. When you pass a list of booleans (length = length of samples/rows) to a data frame, the data frame selects the specific rows where the index of the boolean list is True.

Create a boolean mask by applying a condition to a column, and then use this mask to filter rows for the True Condition.

```
boolean_mask = df[] >  filtered_data = df[boolean_mask]
```

Combine multiple conditions using logical operators (`&` for AND, `|` for OR) and use parentheses for clarity. or you can also use `.isin()` a method of pandas when you want to check if a value from a list of things

```
boolean_mask = (df['Age'] > 25) & (df['Salary'] > 50000)
filtered_data = df[boolean_mask]

boolean_mask = (df['Age'] > 60) | (df['Age'] > 18)
filtered_data = df[boolean_mask]

drama_action_movies = movies.genre.isin([,])
```

3.3 Indexing Methods

Pandas provides various methods for customizing and resetting the DataFrame's index:

`set_index()` This method allows you to set one or more columns as the DataFrame's index. It's useful when you want to perform operations on a specific column. To save it, we have to modify the existing dataframe with the updated one.**`inplace=True`**

```
df = df.set_index() df.set_index(,inplace=)
```

`reset_index()` The `reset_index()` method resets the index to the default integer index and optionally removes the existing index.

```
df = df.reset_index(drop=)
```

4. Data Cleaning

Data cleaning is a critical step in the data preparation process. It involves identifying and addressing issues in your dataset to ensure its quality and reliability.

4.1 Handling Missing Data

Missing data is a common issue in real-world datasets. Pandas offers methods to handle missing values effectively.

`.isna().notna()` These methods allow you to identify missing (`NaN`) and non-missing values, respectively, in your DataFrame. Applying this method for a column will return the boolean list with True for the indices where there is a missing value. And passing this list to a dataframe will return the rows where that column values are null.

```
missing_data = df[df['Column'].isna()]

# To get the sum of null values in a column
df.Column.isna().sum()

df.isna().()
```

`.fillna()` You can replace missing values with a specified value or a calculated value using the `.fillna()` method. The below example replaces all the null values with zeros and directly modifies the data as we used in place = True.

```
# specify the value by which you want to fill the missing value
# In the below example we are replacing the missing value with zero.
df['Column'].fillna(value=0, inplace=True)

# If you want to fill mising data with the mean of the numeric column
df['Column'].fillna(value=df['Column'].mean(), inplace=True)

# If you want to fill mising data with the median of the numeric column
df['Column'].fillna(value=df['Column'].median(), inplace=True)

df[].fillna(value=df[].mode(), inplace=)
```

`.dropna()` Use this method to remove rows or columns containing missing values. By default, it will remove the rows (axis = 0) where any of the column values is missing (how = ‘any’).

```
# Default axis = 0, how= 'any'. Drops all rows where any columns is missing
df.dropna()

# If you want it to be checked only for certain columns, use subset.
# Drops the rows where any of column1 or column2 value is missing.
df.dropna(subset=['Column1', 'Column 2'], how = 'any', inplace=True)

df.dropna(axis=)
```

- By Using how = ‘any’, it will drop the rows where any of the column values are missing.
- By using how = ‘all’, it will drop the rows where all of the specified column values are missing.

4.2 Removing Duplicates

Duplicate rows can skew your analysis results. Pandas offers a simple way to remove duplicates:

`.duplicated()`This method identifies duplicate rows in a DataFrame.

```
# Results the duplicated columns
# when there is an any other row with exact mathc of all the columns.
duplicates = df[df.duplicated()]

duplicates = df[df.duplicated(subset=[,])]
```

1. By Default duplicated, uses `keep='first'` , which keeps the first observed row in the dataframe and marks the later observed ones as True, which specifies they are duplicated ones.
2. If you want to keep the last observed duplicated row in the dataframe then you can give `keep='last'` .
3. If you want to see all the duplicates, then you can give `keep='False'`
4. If you want to check duplicates based on specific columns, then you need to give

> **`.drop_duplicates()`** Use this method to remove duplicate rows from the DataFrame.

```
# Before we drop the duplicated rows,
# if you want to check the count of duplicated rows.
df.duplicated(subset=['column1','column2']).sum()

# Drop duplicated rows, below line will just show the data after dropping
df.drop_duplicates()

# It is advisable to check the shape after dropping before saving it
df.drop_duplicates().shape

df.drop_duplicates(inplace=)
```

4.3 Data Type Conversion

Correct data types are crucial for data analysis. Pandas provides methods to convert data types as needed.

> **`.astype():`** You can change the data type of a specific column using the `.astype()` method. Consider a case, where item_price is saved in object data type, then you would have to change it to float.

astype is also useful to convert booleans to int, basically for all True values it will replace with 1, and the False to be 0.

```
# To see the data types of all the columns
df.dtypes

# To change a column data type
df['Column'] = df['Column'].astype('float')

df.Sex = pd.Series(df.Sex == ).astype()
```

4.4 String Operations

When working with text data, Pandas offers string operations through `.str` an accessor to apply for the entire column which is of object data type.

- **`.str.lower().str.upper()`**These methods convert strings to lowercase or uppercase for the entire column values.
- **`.str.replace()`** Use this method to replace substrings within strings.
- **`.str.Contains()`**This method allows you to check if a specific substring or pattern exists within a string. It returns a boolean Series indicating whether each element contains the specified pattern.
- **`.str.slice()`** You can extract a substring from each string in a Series using the `.str.slice()` method. Specify the start and end positions to define the slice.

```
# Converts all the column values of name to lower case
df['Name'] = df['Name'].str.lower()

# Converts all the column values of name to upper case
df['Name'] = df['Name'].str.upper()

# Finding a value from the column
contains_pattern = df['Text'].str.contains('keyword')
filtered_data = df[contains_pattern]

# String slice
df['Substring'] = df['Text'].str.slice(start=2, stop=5)

# String Replace, say you want to convert $5.4 object to float
df.price.str.replace('$','').astype('float')

df.item_name..contains().astype()
```

5. Data Manipulation

Data manipulation is a core task in data analysis and involves transforming and modifying your data to derive insights or prepare it for further analysis. Pandas provides a rich set of methods for data manipulation that empower you to shape your data to meet your specific needs.

5.1 Applying Functions to DataFrames

There is one hack to do element-wise operations for the dataframe using python `.iterrows()` . However, it's important to note that Pandas is optimized for vectorized operations, and iterating through a DataFrame row by row is generally **not the most efficient way to work with data in Pandas.** It's recommended to use vectorized operations whenever possible.

```
import pandas as pd

# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
'Age': [25, 30, 35]}



df = pd.DataFrame(data)

 index, row  df.iterrows():    ()
```

As an efficient way you can below functions to columns or rows of a DataFrame to perform element-wise operations:

> **`.apply()`** Use this method to apply a custom function to a series or to the entire dataframe. - when you use this on series, Each element of the original column will be passed to the function.- when you use this for the entire dataframe, based on the axis (1 - row, 0-column), the entire row or the entire column will be passed to the function.

```
# Applying a custom function to a column
defsquare(x):
return x ** 2

     df[] = df[].apply(square)


# Sample DataFrame
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Define a function to calculate the average of a row
defaverage_row(row):
return row.mean()



# Apply the function row-wise (axis=0). For column wise, use (axis = 1)
df['Row_Average'] = df.apply(average_row, axis=1)

df
--------------------------------------------------------------------------
output:

   A  B  Row_Average
```

> **`.map()`** This method applies a function to each element of a Series. It's particularly useful for transforming one column based on values from another.

```
mapping_dict = {: , : , : }df[] = df[].(mapping_dict)
```

`.applymap()` When you want to apply a function to each element in the entire DataFrame, you can use `.applymap()`.

```
# Sample DataFrame
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Define a function to add 10 to a value
defadd_10(x):
return x + 10



# Apply the function to the entire DataFrame
df = df.applymap(add_10)

df----------------------------------------------------------------------------output:
A   B
```

5.2 Adding and Removing Columns

You can add and remove columns to tailor your DataFrame for analysis:

- To add a new column or replace an existing one, simply assign values to it.
- Use the `.drop()` method to remove columns. Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns'). You can use axis=1/ 'columns' to drop column, or use axis =0/ 'index' to drop the row.

```
# Creates a new column with the name New_Column
df['New_Column'] = [1, 2, 3, 4]

df.drop([, ], axis=, inplace=)
```

5.3 Combining DataFrames (Concatenation, Joining, Merging)

Pandas offers powerful methods to combine DataFrames:

You can concatenate DataFrames vertically or horizontally using `pd.concat()`. axis=0 will concatenate them in the rows, axis=1 will concatenate them in the columns. It will check for the common columns between both the dataframes and for the matched columns, it will concatenate in the rows.

```
import pandas as pd

# Sample DataFrames with the same column names
data1 = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C':[1,2,3]}
data2 = {'A': [7, 8, 9], 'B': [10, 11, 12], 'D':[1,2,3]}
df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Concatenate df1 and df2 horizontally (along columns) with same column names
result = pd.concat([df1, df2], axis=0)

# Display the concatenated DataFrame
print(result)

--------------------------------------------------------------------------output:   A
B    C    D         NaN         NaN         NaN      NaN        NaN        NaN
```

- You can perform SQL-like joins on DataFrames using the `.merge()` method.
- Pandas allows you to merge DataFrames based on common columns. Using merge you can perform various joins such as inner, outer, left, and right. - An inner join will only keep where the cells of common columns have matched. - An outer join will keep every row from both data frames. Left will keep all the rows from the left table, similarly right for all the rows from the right table.- If you want to match a table column with the index of another table, then for the table you want to match it with the index, specify `left_index=True` or `right_index=True` accordingly. And for the other table on which column you want to match it with, you have to specify it as `left_on=column_name` or `right_on=column_name` accordingly.

```
# Inner Join
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Age': [25, 30, 22]})

# Inner join on 'ID'
result = pd.merge(df1, df2, on='ID', how='inner')

# Display the merged DataFrame
print("Inner Join")
print(result)

# Left join on 'ID'
result = pd.merge(df1, df2, on='ID', how='left')

# Display the merged DataFrame
print("Left Join")
print(result)

# If matching column names are different from both data frames
# you can specify them manually
# Considering, left table has ID1 and right table has ID2
pd.merge(df1,df2, left_on="ID1", right_on="ID2")

# Here it will try to match the left dataframe index with right ID column
pd.merge(df1,df2, left_index=True,right_on="ID")
--------------------------------------------------------------------------
output:
Inner Join
   ID     Name  Age
02      Bob   25
13  Charlie   30

      Left Join   ID     Name   Age       Alice   NaN         Bob       Charlie
```

Conclusion

We've explored the intricacies of data structures, data loading, inspection, selection, cleaning, and manipulations— the essential tools every data enthusiast should wield.

This marks the end of Part 1, but remember, this is just the beginning. **is on the horizon, promising a deeper dive into Advanced Techniques, visualizations, and intricate data manipulations.**

If you are interested in Pandas, Numpy, Matplolib, Seaborn, and Plotly, then look no further, I also have detailed articles on these essential libraries for Data Science, Here's a curated List.

RaviTeja G

Detailed Guides for all the Essential Data Science Libraries

8 stories

I will be posting more interesting articles related to Machine learning as well. Follow me for more and Subscribe to not miss any exciting data science articles!

Exciting Announcement

Day 1/100
I am thrilled to announce that I am embarking on a journey of 100 days of machine learning and deep learning code challenge. Throughout this challenge, I'll be sharing my newfound insights with our amazing community. Each day, I'll revisit these topics from the basics of pandas to the advanced theory of machine learning, and create articles to teach what I've learned.

I invite you all to join me on this exhilarating journey! Whether you're a fellow data enthusiast or just curious about the world of machine learning, there's something here for everyone. Let's learn, grow, and inspire each other. By the way, the pandas in the above picture were cute, right? Thank you for reading, **Happy learning, and Have a good day :)**

In Plain English

Thank you for being a part of our community! Before you go: