PANDAS CHEATSHEET

A Beginners Guide

@apexiq.ai

Introduction

What is Pandas?

Pandas is a free library for Python that makes it easy to work with data. It provides two main data structures: Series (like a list) and DataFrame (like a table or spreadsheet). With Pandas, you can easily organize, analyze, and manipulate data.

Why use Pandas?

- User-Friendly: It has a simple and clear syntax, making it easy to learn and use.
- Data Handling: You can easily read and write data in different formats, like CSV or Excel.
- Data Manipulation: It offers powerful tools to filter, group, and reshape your data quickly.
- Integration: Pandas works well with other libraries like Matplotlib for plotting graphs and Scikit-learn for machine learning.

Installation

To install Pandas, open your terminal or command prompt and type:

```
!pip install pandas
```

If you’re using Anaconda, you can install it by typing:

```
!conda install pandas
```

1. Loading Data

Loading data is the first step in any data analysis workflow. Pandas provides several functions to read data from various file formats.

Import:

Import Pandas library:

```
import pandas as pd
```

Load CSV File:

```
df = pd.read_csv('file.csv')
```

Load Excel File:

```
df = pd.read_excel('file.xlsx')
or
df = pd.read_excel('file.xlsx', sheet_name='Sheet1')
```

Load JSON File:

```
df = pd.read_json('file.json')
```

2. Viewing Data

After loading the data, it's important to inspect it to understand its structure and content. Pandas provides several methods for this.

View First N Rows:

```
df.head(n=5)
```

View Last N Rows:

```
df.tail(n=5)
```

Random Sample of Rows:

```
df.sample(n=5)
```

Summary of DataFrame:

Display information about the DataFrame (data types, non-null counts)

```
df.info()
```

Display descriptive statistics for numerical columns (count, mean, std, min, max)

```
df.describe()
```

3. Selecting Data

Selecting specific data from a DataFrame is crucial for analysis. Pandas allows you to select columns and rows easily.

Select Column by Name:

Access a single column by name

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

Access multiple columns by names (returns a DataFrame)

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

Select Rows by Index:

Access the first row by integer index (position)

```
df.iloc[0]
```

Access the first row by label (if index is not integer)

```
df.loc[0]
```

Select Rows with Conditions:

Filter rows based on condition (e.g., column_name > value)

```
filtered_df = df[df['column_name'] > value]
```

4. Modifying Data

Modifying data in a DataFrame is essential for preparing your dataset for analysis.

Add New Column:

Create a new column that is double the values of an existing column

```
df['new_column'] = df['existing_column'] * 2
```

Rename Columns:

Rename a specific column

```
df.rename(columns={'old_name': 'new_name'}, inplace=True)
```

Drop Columns:

Drop specified column(s)

```
df.drop(columns=['column_to_drop'], inplace=True)
```

5. Handling Missing Values

Dealing with missing values is crucial to ensure the integrity of your analysis.

Check for Missing Values:

Count of missing values in each column

```
df.isnull().sum()
```

Drop Rows with Missing Values:

Drop any row with NaN values

```
df.dropna(inplace=True)
```

Drop rows where a specific column is NaN

```
df.dropna(subset=['column_name'], inplace=True)
```

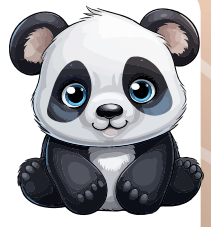
Fill Missing Values:

Fill NaN with a specified value (e.g., zero)

```
df.fillna(value=0, inplace=True)
```

Forward fill to propagate the last valid observation forward

```
df.fillna(method='ffill', inplace=True)
```



6. Removing Duplicates

Dealing with missing values is crucial to ensure the integrity of your analysis.

Remove Duplicate Rows:

Remove duplicate rows based on all columns

```
df.drop_duplicates(inplace=True)
```

Remove duplicates based on specific column(s)

```
df.drop_duplicates(subset=['col1'], inplace=True)
```

7. Sorting Data

Sorting data is essential for analysis and presentation. You can sort your DataFrame by one or more columns.

Sort by One Column:

Sort in ascending order

```
df.sort_values(by='column_name', ascending=True, inplace=True)
```

Sort in descending order

```
df.sort_values(by='column_name', ascending=False, inplace=True)
```

Sort by Multiple Columns:

Sort by col1 ascending and col2 descending

```
df.sort_values(by=['col1', 'col2'], ascending=[True, False], inplace=True)
```

8. Grouping and Aggregating Data

Grouping data allows you to perform operations on subsets of your data.

Group By One Column:

Group data by specified column(s)

```
grouped = df.groupby('column_name')
```

Aggregate Functions on Grouped Data:

Sum of grouped values in a specific column

```
grouped['value_column'].sum()
```

Mean of grouped values in a specific column

```
grouped['value_column'].mean()
```

Multiple aggregations

```
agg_df = grouped.agg({'value_column': ['sum', 'mean'], 'another_column': 'count'})
```

9. Merging and Joining DataFrames

Combining multiple DataFrames is often necessary when working with related datasets.

Merge Two DataFrames:

Merge on key column(s)

```
merged_df = pd.merge(df1, df2, on='key_column')
```

Outer Join Two DataFrames:

Outer join to include all records from both DataFrames

```
merged_outer = pd.merge(df1, df2, how='outer', on='key_column')
```

Concatenate Two DataFrames:

Concatenate along rows (axis=0)

```
concat_df = pd.concat([df1, df2], axis=0)
```

Concatenate along columns (axis=1)

```
concat_cols_df = pd.concat([df1, df2], axis=1)
```

10. Applying Functions

You can apply custom functions to your DataFrame or Series to manipulate or transform data.

Using apply() on DataFrame:

Apply a function to each element in a column

```
df['new_col'] = df['existing_col'].apply(lambda x: x + 1)
```

Using apply() on Series:

Square each element in the Series

```
s = pd.Series([1, 2, 3])
s_squared = s.apply(lambda x: x**2)
```

Using map() for Element-wise Operations:

Map values based on a dictionary

```
df['new_col'] = df['existing_col'].map({1: 'A', 2: 'B'})
```

11. String Methods

Pandas provides string methods that allow you to perform vectorized string operations on Series.

Converting Strings to Lowercase:

Convert all strings in the column to lowercase

```
df['string_column'] = df['string_column'].str.lower()
```

Checking for Substrings:

Check if 'text' is in each string

```
df['contains_text'] = df['string_column'].str.contains('text')
```

Replacing Substrings:

Replace 'old' with 'new' in strings

```
df['string_column'] = df['string_column'].str.replace('old', 'new')
```

12. Advanced Data Manipulation

Advanced data manipulation techniques allow for more complex transformations and reshaping of your DataFrame.

Melt Function:

The melt() function is used to transform wide-format data into long-format data.

```
df_melted = pd.melt(df, id_vars=['id'], value_vars=['col1', 'col2'])
```

Pivot Function:

The pivot() function reshapes the DataFrame by specifying index, columns, and values.

```
df_pivot = df.pivot(index='date', columns='category', values='value')
```

Stack and Unstack:

Stack: Convert columns into rows (long format).

```
stacked_df = df.stack()
```

Unstack: Convert rows back into columns (wide format).

```
unstacked_df = stacked_df.unstack()
```

13. Creating and Using Pivot Tables

Pivot tables allow you to summarize data in a flexible way.

Creating a Pivot Table:

Create a pivot table with specified values, index, columns, and aggregation function

```
df_melted = pd.melt(df, id_vars=['id'], value_vars=['col1', 'col2'])
```

Pivot Function:

The pivot() function reshapes the DataFrame by specifying index, columns, and values.

```
pivot_table = df.pivot_table(values='value', index='index_col', columns='column_col', aggfunc='sum')
```

Pivot Table with Multiple Aggregations:

Create a pivot table with multiple aggregation functions (sum and mean)

```
pivot_table_multi = df.pivot_table(values='value', index='index_col', aggfunc=[np.sum, np.mean])
```

14. Working with Categorical Data

Pandas provides support for categorical data, which can improve performance and memory usage.

Convert Column to Categorical:

Convert a column to categorical type

```
df['category_column'] = df['category_column'].astype('category')
```

Get Categories and Their Codes:

Get unique categories

```
pivot_table = df.pivot_table(values='value', index='index_col',
columns='column_col', aggfunc='sum')
```

Get integer codes for categories

```
codes = df['category_column'].cat.codes
```

Using Categorical Data for Grouping:

Group by categorical column and count occurrences

```
grouped = df.groupby('category_column').size()
```

15. Handling Date and Time Data

Pandas provides powerful tools for working with date and time data, making it easy to manipulate and analyze time series.

Convert Strings to Datetime:

Convert to datetime format

```
df['date_column'] = pd.to_datetime(df['date_column'])
```

Extracting Date Components:

Extract year

```
df['year'] = df['date_column'].dt.year
```

Extract month

```
df['month'] = df['date_column'].dt.month
```

Extract day

```
df['day'] = df['date_column'].dt.day
```

Setting a Date Column as Index:

Set date_column as the index

```
df.set_index('date_column', inplace=True)
```

Apex IQ

LIKE FOLLOW SHARE

THANK YOU!

@apexiq.ai