Comprehensive Pandas Library Notes for Data Science Learners

Type	@datasciencebrain

Introduction to pandas

Definition

Pandas is an open-source Python library providing high-performance, easy-to-use data structures and data analysis tools. It is built on top of NumPy and is widely used for data manipulation, cleaning, and analysis in data science.

Origin

Developed by Wes McKinney in 2008, pandas was created to address the need for flexible and efficient data analysis tools in Python. It has become a cornerstone of the Python data science ecosystem, integrated with tools like NumPy, Matplotlib, and Scikit-learn.

Uses in Data Science

- **Data Cleaning**: Handling missing data, duplicates, and inconsistencies.
- **Data Exploration**: Summarizing and visualizing datasets.
- **Data Transformation**: Reshaping, merging, and aggregating data.
- **Time Series Analysis**: Working with temporal data for forecasting and trend analysis.
- **Integration**: Interfacing with various data formats and databases.

Installation

Install pandas using pip or conda:

```
pip install pandas
```

or

```
conda install pandas
```

Verify installation:

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

Core Data Structures

Series

A Series is a one-dimensional, labeled array capable of holding any data type. It is similar to a NumPy array but with an index for labeling.

Properties:

- **index**: Labels for the data.
- **values**: Underlying NumPy array of data.
- **dtype**: Data type of the Series.

Example: Creating a Series

```
import pandas as pd
# From a list
s1 = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])
print(s1)
# Output:
# a    1
# b    2
# c    3
# d    4
# dtype: int64

# From a dictionary
s2 = pd.Series({'a': 10, 'b': 20, 'c': 30})
print(s2)
# Output:
# a    10
# b    20
# c    30
# dtype: int64
```

Common Methods:

- `s.head(n)`: View first n rows.
- `s.tail(n)`: View last n rows.
- `s.value_counts()`: Count unique values.
- `s.mean()`, `s.sum()`, `s.min()`, `s.max()`: Basic statistical operations.

DataFrame

A DataFrame is a two-dimensional, tabular data structure with labeled axes (rows and columns). It is similar to a spreadsheet or SQL table.

Properties:

- **columns**: Column labels.

- **index**: Row labels.
- **shape**: Dimensions of the DataFrame.

Example: Creating a DataFrame

```
# From a dictionary
data = {'Name': ['Alice', 'Bob', 'Cathy'], 'Age': [25, 30, 22], 'City': ['New York',
'London', 'Paris']}
df = pd.DataFrame(data)
print(df)
# Output:
#     Name  Age     City
# 0   Alice   25  New York
# 1     Bob   30   London
# 2   Cathy   22    Paris

# From a list of dictionaries
data_list = [{'Name': 'Alice', 'Age': 25}, {'Name': 'Bob', 'Age': 30}]
df2 = pd.DataFrame(data_list)
print(df2)
# Output:
#     Name  Age
# 0   Alice   25
# 1     Bob   30
```

Common Methods:

- `df.head()`, `df.tail()`: View top/bottom rows.
- `df.info()`: Summary of DataFrame structure.
- `df.describe()`: Descriptive statistics for numeric columns.

Data Input and Output

Pandas supports reading and writing data in various formats.

Common File Formats

- **CSV**: `pd.read_csv()`, `df.to_csv()`
- **Excel**: `pd.read_excel()`, `df.to_excel()` (requires `openpyxl` or `xlrd`)
- **JSON**: `pd.read_json()`, `df.to_json()`
- **SQL**: `pd.read_sql()`, `df.to_sql()` (requires `SQLAlchemy`)
- **Parquet**: `pd.read_parquet()`, `df.to_parquet()` (requires `pyarrow` or `fastparquet`)

Examples:

```
# Reading CSV
df_csv = pd.read_csv('data.csv')

# Writing to CSV
df.to_csv('output.csv', index=False)

# Reading Excel
df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# Reading JSON
df_json = pd.read_json('data.json')

# Reading from SQL database
import sqlite3
conn = sqlite3.connect('database.db')
df_sql = pd.read_sql('SELECT * FROM table_name', conn)

# Writing to Parquet
df.to_parquet('output.parquet')
```

Data Selection and Indexing

.loc and .iloc

- **.loc**: Label-based indexing.
- **.iloc**: Integer-based indexing.

Example:

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['x', 'y', 'z'])
# Using .loc
print(df.loc['x', 'A'])  # Output: 1
print(df.loc['x':'y', 'B'])  # Output: x    4
                           #       y    5
                           # Name: B, dtype: int64

# Using .iloc
print(df.iloc[0, 0])  # Output: 1
print(df.iloc[0:2, 1])  # Output: x    4
                       #       y    5
                       # Name: B, dtype: int64
```

Conditional Selection

Filter rows based on conditions.

```
# Select rows where A > 1
print(df[df['A'] > 1])
# Output:
#    A  B
# y  2  5
# z  3  6
```

Multi-Indexing

Create hierarchical indices for advanced indexing.

```
# Creating a multi-index DataFrame
arrays = [['A', 'A', 'B', 'B'], [1, 2, 1, 2]]
index = pd.MultiIndex.from_arrays(arrays, names=('Group', 'Num'))
df_multi = pd.DataFrame({'Value': [10, 20, 30, 40]}, index=index)
print(df_multi)
# Output:
```

```
#             Value
# Group Num
# A     1      10
#       2      20
# B     1      30
#       2      40

# Selecting with multi-index
print(df_multi.loc['A'])
# Output:
#      Value
# Num
# 1      10
# 2      20
```

Data Exploration and Descriptive Statistics

Key Functions

- **.info()**: Displays DataFrame structure, including data types and non-null counts.
- **.describe()**: Summary statistics for numeric columns.
- **.value_counts()**: Frequency of unique values in a Series.

Example:

```
df = pd.DataFrame({'A': [1, 2, 2, 3], 'B': ['x', 'y', 'x', 'z']})
print(df.info())
# Output:
# <class 'pandas.core.frame.DataFrame'>
# RangeIndex: 4 entries, 0 to 3
# Data columns (total 2 columns):
#  #   Column  Non-Null Count  Dtype
# ---  ------  --------------  -----
#  0   A       4 non-null      int64
```

```
#  1   B       4 non-null      object
# dtypes: int64(1), object(1)
# memory usage: 192.0+ bytes

print(df.describe())
# Output:
#              A
# count  4.000000
# mean   2.000000
# std    0.816497
# min    1.000000
# 25%    1.750000
# 50%    2.000000
# 75%    2.250000
# max    3.000000

print(df['B'].value_counts())
# Output:
# x    2
# y    1
# z    1
# Name: B, dtype: int64
```

Best Practices

- Use `.info()` to understand data types and missing values.
- Use `.describe()` to identify outliers or unusual distributions.
- Check categorical columns with `.value_counts()` for imbalances.

Data Cleaning and Handling Missing Values

Identifying Missing Values

Use `.isna()` or `.isnull()` to detect missing values.

```
df = pd.DataFrame({'A': [1, None, 3], 'B': [4, 5, None]})
print(df.isna())
# Output:
#        A      B
# 0  False  False
# 1   True  False
# 2  False   True
```

Handling Missing Values

- **Drop**: `df.dropna()` removes rows/columns with missing values.
- **Fill**: `df.fillna(value)` replaces missing values with a specified value.
- **Impute**: Use statistical methods (mean, median) for imputation.

Example:

```
# Drop rows with any missing values
print(df.dropna())
# Output:
#      A    B
# 0  1.0  4.0

# Fill missing values with 0
print(df.fillna(0))
# Output:
#      A    B
# 0  1.0  4.0
# 1  0.0  5.0
# 2  3.0  0.0

# Impute with mean
df['A'] = df['A'].fillna(df['A'].mean())
print(df)
# Output:
#      A    B
```

```
# 0  1.0  4.0
# 1  2.0  5.0
# 2  3.0  NaN
```

Removing Duplicates

Use `df.drop_duplicates()` to remove duplicate rows.

```
df = pd.DataFrame({'A': [1, 1, 2], 'B': [4, 4, 5]})
print(df.drop_duplicates())
# Output:
#    A  B
# 0  1  4
# 2  2  5
```

Data Replacement

Replace specific values using `df.replace()`.

```
df = pd.DataFrame({'A': ['x', 'y', 'x'], 'B': [1, 2, 3]})
df['A'] = df['A'].replace('x', 'z')
print(df)
# Output:
#    A  B
# 0  z  1
# 1  y  2
# 2  z  3
```

Data Manipulation and Transformation

Creating, Renaming, Dropping Columns

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
# Add new column
df['C'] = df['A'] + df['B']
```

```
# Rename columns
df = df.rename(columns={'A': 'X', 'B': 'Y'})
# Drop column
df = df.drop('C', axis=1)
print(df)
# Output:
#    X  Y
# 0  1  4
# 1  2  5
# 2  3  6
```

Sorting Data

Sort by values or index using `df.sort_values()` or `df.sort_index()`.

```
# Sort by column 'X'
print(df.sort_values('X', ascending=False))
# Output:
#    X  Y
# 2  3  6
# 1  2  5
# 0  1  4
```

Reshaping Data

- **Melt**: Convert wide format to long format.
- **Pivot**: Convert long format to wide format.
- **Stack/Unstack**: Reshape using multi-index.

Example: Melt and Pivot

```
df = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Math': [90, 85], 'Science': [88, 9
2]})
# Melt
melted = pd.melt(df, id_vars=['Name'], value_vars=['Math', 'Science'], var_na
me='Subject', value_name='Score')
```

```
print(melted)
# Output:
#    Name  Subject  Score
# 0  Alice     Math     90
# 1    Bob     Math     85
# 2  Alice  Science     88
# 3    Bob  Science     92

# Pivot
pivoted = melted.pivot(index='Name', columns='Subject', values='Score')
print(pivoted)
# Output:
# Subject  Math  Science
# Name
# Alice      90       88
# Bob        85       92
```

Grouping, Aggregation, and Pivot Tables

GroupBy

Group data by one or more columns and apply aggregation functions.

```
df = pd.DataFrame({'Category': ['A', 'A', 'B', 'B'], 'Value': [10, 20, 30, 40]})
grouped = df.groupby('Category').agg({'Value': ['sum', 'mean']})
print(grouped)
# Output:
#          Value
#           sum  mean
# Category
# A          30  15.0
# B          70  35.0
```

Pivot Tables

Summarize data with flexible aggregation.

```
df = pd.DataFrame({'Name': ['Alice', 'Bob', 'Alice', 'Bob'], 'Subject': ['Math', 'M
ath', 'Science', 'Science'], 'Score': [90, 85, 88, 92]})
pivot_table = df.pivot_table(values='Score', index='Name', columns='Subject',
aggfunc='mean')
print(pivot_table)
# Output:
# Subject  Math  Science
# Name
# Alice     90       88
# Bob       85       92
```

Merging, Joining, and Concatenating DataFrames

Concatenation

Combine DataFrames vertically or horizontally using `pd.concat()`.

```
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})
# Vertical concatenation
print(pd.concat([df1, df2], ignore_index=True))
# Output:
#    A  B
# 0  1  3
# 1  2  4
# 2  5  7
# 3  6  8
```

Merging

Combine DataFrames based on keys using `pd.merge()`.

```
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Cathy']})
df2 = pd.DataFrame({'ID': [1, 2, 4], 'Score': [90, 85, 88]})
# Inner merge
```

```
print(pd.merge(df1, df2, on='ID', how='inner'))
# Output:
#    ID   Name  Score
# 0   1  Alice     90
# 1   2    Bob     85
```

Join Types:

- `inner`: Keep only matching rows.
- `left`: Keep all rows from left DataFrame.
- `right`: Keep all rows from right DataFrame.
- `outer`: Keep all rows from both DataFrames.

Joining

Join DataFrames on index using `df.join()`.

```
df1 = pd.DataFrame({'A': [1, 2]}, index=['x', 'y'])
df2 = pd.DataFrame({'B': [3, 4]}, index=['x', 'y'])
print(df1.join(df2))
# Output:
#    A  B
# x  1  3
# y  2  4
```

Working with Time Series Data

Date-Time Conversions

Convert strings to datetime using `pd.to_datetime()`.

```
df = pd.DataFrame({'date': ['2023-01-01', '2023-01-02'], 'value': [10, 20]})
df['date'] = pd.to_datetime(df['date'])
print(df)
# Output:
#         date  value
```

```
# 0 2023-01-01     10
# 1 2023-01-02     20
```

Indexing by Dates

Set datetime as index for time-based operations.

```
df.set_index('date', inplace=True)
print(df)
# Output:
#             value
# date
# 2023-01-01     10
# 2023-01-02     20
```

Resampling

Aggregate time series data over periods using `df.resample()`.

```
# Resample to monthly mean
df_resampled = df.resample('ME').mean()
print(df_resampled)
# Output:
#             value
# date
# 2023-01-31   15.0
```

Applying Functions and Custom Transformations

.apply(), .map(), .applymap()

- **.apply()**: Apply a function along an axis of a DataFrame or on a Series.
- **.map()**: Apply a function element-wise on a Series.
- **.applymap()**: Apply a function element-wise on a DataFrame (deprecated; use `map` for Series or `apply` with lambda).

Example:

```
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
# Apply on Series
df['A'] = df['A'].map(lambda x: x * 2)
print(df)
# Output:
#    A  B
# 0  2  4
# 1  4  5
# 2  6  6

# Apply on DataFrame
df['C'] = df.apply(lambda row: row['A'] + row['B'], axis=1)
print(df)
# Output:
#    A  B   C
# 0  2  4   6
# 1  4  5   9
# 2  6  6  12
```

Advanced pandas Techniques and Performance Optimization

Vectorization

Use vectorized operations instead of loops for performance.

```
# Slow: Using loop
df['A_squared'] = [x**2 for x in df['A']]
# Fast: Vectorized
df['A_squared'] = df['A'] ** 2
```

Categorization

Convert object columns to `category` type for memory efficiency.

```
df = pd.DataFrame({'Category': ['A', 'B', 'A', 'C']})
df['Category'] = df['Category'].astype('category')
print(df['Category'].memory_usage())
# Output: 152 (less memory than object type)
```

Best Practices

- Avoid chained indexing (`df['A']['B']`) to prevent SettingWithCopyWarning.
- Use `inplace=True` sparingly to avoid unexpected behavior.
- Pre-allocate DataFrame size when appending data in loops.

Common Errors and Troubleshooting

Key Errors

- **Error**: `KeyError: 'column_name'`
- **Solution**: Check column names with `df.columns`. Ensure exact spelling and case.

SettingWithCopyWarning

- **Error**: Occurs when modifying a slice of a DataFrame.
- **Solution**: Use `.loc` or `.copy()` to explicitly modify data.

```
# Problematic
df_subset = df[df['A'] > 1]
df_subset['B'] = 0  # Warning

# Solution
df.loc[df['A'] > 1, 'B'] = 0
```

Type Mismatch

- **Error**: Operations fail due to mismatched data types.

- **Solution**: Check types with `df.dtypes` and convert using `df.astype()` or `pd.to_numeric()`.

Additional Resources and Best Practices

Recommended Readings

- **Official Documentation**: pandas.pydata.org
- **Books**:
  - "Python for Data Analysis" by Wes McKinney
  - "Pandas for Everyone" by Daniel Y. Chen
- **Cheat Sheets**: Available on DataCamp or pandas website.

Community Resources

- **Stack Overflow**: For specific pandas questions.
- **Kaggle**: Datasets and tutorials for practice.
- **pandas-dev/pandas**: GitHub repository for contributing or reporting issues.

Best Practices

- Document your code and transformations for reproducibility.
- Use meaningful column names and consistent data types.
- Test on small datasets before scaling to large ones.
- Regularly update pandas to leverage performance improvements.