Pandas Cheatsheet

Essential operations for data manipulation and analysis

This cheatsheet provides a quick reference to fundamental Pandas operations, syntax, and advanced features, ideal for both beginners and experienced data scientists for efficient data processing.

Data Loading
Import data from various sources

Data Selection
Access and subset data

Data Cleaning
Prepare data for analysis

Data Analysis
Perform statistical operations

Data Visualization
Plot and explore data

Data Loading & Saving

Read CSV: `pd.read_csv()`

Load data from a CSV file into a DataFrame.

```
import pandas as pd
# Read a CSV file
df = pd.read_csv('data.csv')
# Set first column as index
df = pd.read_csv('data.csv', index_col=0)
# Specify a different separator
df = pd.read_csv('data.csv', sep=';')
# Parse dates
df = pd.read_csv('data.csv', parse_dates=['Date'])
```

Read Excel: `pd.read_excel()`

Load data from an Excel file.

```
# Read first sheet
df = pd.read_excel('data.xlsx')
# Read specific sheet
df = pd.read_excel('data.xlsx', sheet_name='Sheet2')
# Set row 2 as header (0-indexed)
df = pd.read_excel('data.xlsx', header=1)
```

Read SQL: `pd.read_sql()`

Read SQL query or table into a DataFrame.

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///my_database.db')
df = pd.read_sql('SELECT * FROM users', engine)
df = pd.read_sql_table('products', engine)
```

Save CSV: `df.to_csv()`

Write DataFrame to a CSV file.

```
# Exclude index column
df.to_csv('output.csv', index=False)
# Exclude header row
df.to_csv('output.csv', header=False)
```

Save Excel: `df.to_excel()`

Write DataFrame to an Excel file.

```
# Save to Excel
df.to_excel('output.xlsx', sheet_name='Results')
writer = pd.ExcelWriter('output.xlsx')
df1.to_excel(writer, sheet_name='Sheet1')
df2.to_excel(writer, sheet_name='Sheet2')
writer.save()
```

Save SQL: `df.to_sql()`

Write DataFrame to a SQL database table.

```
# Create/replace table
df.to_sql('new_table', engine, if_exists='replace',
index=False)
# Append to existing table
df.to_sql('existing_table', engine, if_exists='append')
```

DataFrame Info & Structure

Understand the structure and summary of your DataFrame.

01

Basic Info: `df.info()`

Prints a concise summary of a DataFrame, including data types and non-null values.

```
# Display DataFrame summary
df.info()
# Show data types of each column
df.dtypes
# Get the number of rows and
columns (tuple)
df.shape
# Get column names
df.columns
# Get row index
df.index
```

02

Descriptive Statistics: `df.describe()`

Generates descriptive statistics of numerical columns.

```
# Summary statistics for numerical
columns
df.describe()
# Summary for a specific column
df['column'].describe()
# Include all columns (object type
too)
df.describe(include='all')
```

03

View Data: `df.head()` / `df.tail()`

Display the first or last 'n' rows of the DataFrame.

```
# First 5 rows
df.head()
# Last 10 rows
df.tail(10)
# Random 5 rows
df.sample(5)
```

Data Cleaning & Transformation

Missing Values: `isnull()` / `fillna()` / `dropna()`

Identify, fill, or drop missing (NaN) values.

```
# Count missing values per column
df.isnull().sum()
# Fill all NaN with 0
df.fillna(0)
# Fill with column mean
df['col'].fillna(df['col'].mean())
# Drop rows with any NaN
df.dropna()
# Drop columns with any NaN
df.dropna(axis=1)
```

Duplicates: `duplicated()` / `drop_duplicates()`

Identify and remove duplicate rows.

```
# Boolean Series indicating duplicates
df.duplicated()
# Remove all duplicate rows
df.drop_duplicates()
# Remove based on specific columns
df.drop_duplicates(subset=['col1', 'col2'])
```

Data Types: `astype()`

Change the data type of a column.

```
# Change to integer
df['col'].astype(int)
# Change to string
df['col'].astype(str)
# Convert to datetime
df['col'] = pd.to_datetime(df['col'])
```

Apply Function: `apply()` / `map()` / `replace()`

Apply functions or replace values in DataFrames/Series.

```
# Apply lambda function to a column
df['col'].apply(lambda x: x*2)
# Map values using a dictionary
df['col'].map({'old': 'new'})
# Replace values
df.replace('old_val', 'new_val')
# Replace multiple values
df.replace(['A', 'B'], ['C', 'D'])
```

DataFrame Inspection

Unique Values: `unique()` / `value_counts()`

Explore unique values and their frequencies.

```
# Get unique values in a column
df['col'].unique()
# Get number of unique values
df['col'].nunique()
# Count occurrences of each unique value
df['col'].value_counts()
# Proportions of unique values
df['col'].value_counts(normalize=True)
```

Correlation: `corr()` / `cov()`

Calculate correlation and covariance between numerical columns.

```
# Pairwise correlation of columns
df.corr()
# Pairwise covariance of columns
df.cov()
# Correlation between two specific columns
df['col1'].corr(df['col2'])
```

Aggregations: `groupby()` / `agg()`

Group data by categories and apply aggregate functions.

```
# Mean for each category
df.groupby('category_col').mean()
# Group by multiple columns
df.groupby(['col1', 'col2']).sum()
# Multiple aggregations
df.groupby('category_col').agg({'num_col': ['min', 'max',
'mean']})
df.pivot_table(values='sales', index='region',
columns='product', aggfunc='sum')
```

Cross-Tabulations: `pd.crosstab()`

Compute a frequency table of two or more factors.

```
# Simple frequency table
pd.crosstab(df['col1'], df['col2'])
# With row/column sums
pd.crosstab(df['col1'], df['col2'], margins=True)
# With aggregate values
pd.crosstab(df['col1'], df['col2'], values=df['value_col'],
aggfunc='mean')
```

Memory Management

Optimize DataFrame memory usage for large datasets.

Memory Usage: `df.memory_usage()`

Display the memory usage of each column or the entire DataFrame.

```
# Memory usage of each column
df.memory_usage()
# Total memory usage in bytes
df.memory_usage(deep=True).sum()
# Detailed memory usage in info() output
df.info(memory_usage='deep')
```

Optimize Dtypes: `astype()`

Reduce memory by converting columns to smaller, appropriate data types.

```
# Downcast integer
df['int_col'] = df['int_col'].astype('int16')
# Downcast float
df['float_col'] = df['float_col'].astype('float32')
# Use categorical type
df['category_col'] = df['category_col'].astype('category')
```

Chunking Large Files: `read_csv(chunksize=...)`

Process large files in chunks to avoid loading everything into memory at once.

```
chunk_iterator = pd.read_csv('large_data.csv',
chunksize=10000)
for chunk in chunk_iterator:
    # Process each chunk
    print(chunk.shape)

# Concatenate processed chunks (if needed)
# processed_chunks = []
# for chunk in chunk_iterator:
#    processed_chunks.append(process_chunk(chunk))
# final_df = pd.concat(processed_chunks)
```

Data Import/Export

Transfer data using various formats and protocols.

Read JSON: `pd.read_json()`

Load data from a JSON file or URL.

```
# Read from local JSON
df = pd.read_json('data.json')
# Read from URL
df = pd.read_json('http://example.com/api/data')
# Read from JSON string
df = pd.read_json(json_string_data)
```

Read HTML: `pd.read_html()`

Parse HTML tables from a URL, string, or file.

```
tables =
pd.read_html('http://www.w3.org/TR/html401/sgml/entiti
es.html')
# Usually returns a list of DataFrames
df = tables[0]
```

To JSON: `df.to_json()`

Write DataFrame to JSON format.

```
# To JSON file
df.to_json('output.json', orient='records', indent=4)
# To JSON string
json_str = df.to_json(orient='split')
```

To HTML: `df.to_html()`

Render DataFrame as an HTML table.

```
# To HTML string
html_table_str = df.to_html()
# To HTML file
df.to_html('output.html', index=False)
```

Read Clipboard: `pd.read_clipboard()`

Read text from the clipboard into a DataFrame.

```
# Copy table data from web/spreadsheet and run
df = pd.read_clipboard()
```

Data Serialization

Store and retrieve Pandas objects efficiently.

Pickle: `df.to_pickle()` / `pd.read_pickle()`

Serialize/deserialize Pandas objects to/from disk.

```
# Save DataFrame as a pickle file
df.to_pickle('my_dataframe.pkl')
# Load DataFrame
loaded_df = pd.read_pickle('my_dataframe.pkl')
```

HDF5: `df.to_hdf()` / `pd.read_hdf()`

Store/load DataFrames using the HDF5 format, good for large datasets.

```
# Save to HDF5
df.to_hdf('my_data.h5', key='df', mode='w')
# Load from HDF5
loaded_df = pd.read_hdf('my_data.h5', key='df')
```

Data Filtering & Selection

Locate and extract specific data subsets.

Label-based: `df.loc[]` / `df.at[]`

Select data by explicit label of index/columns.

```
# Select row with index 0
df.loc[0]
# Select all rows for 'col1'
df.loc[:, 'col1']
# Slice rows and select multiple columns
df.loc[0:5, ['col1', 'col2']]
# Boolean indexing for rows
df.loc[df['col'] > 5]
# Fast scalar access by label
df.at[0, 'col1']
```

Position-based: `df.iloc[]` / `df.iat[]`

Select data by integer position of index/columns.

```
# Select first row by position
df.iloc[0]
# Select first column by position
df.iloc[:, 0]
# Slice rows and select multiple columns by position
df.iloc[0:5, [0, 1]]
# Fast scalar access by position
df.iat[0, 0]
```

Boolean Indexing: `df[condition]`

Filter rows based on one or more conditions.

```
# Rows where 'col1' is greater than 10
df[df['col1'] > 10]
# Multiple conditions
df[(df['col1'] > 10) & (df['col2'] == 'A')]
# Rows where 'col1' is NOT in list
df[~df['col1'].isin([1, 2, 3])]
```

Query Data: `df.query()`

Filter rows using a query string expression.

```
# Equivalent to boolean indexing
df.query('col1 > 10')
# Complex query
df.query('col1 > 10 and col2 == "A"')
# Use local variables with '@'
df.query('col1 in @my_list')
```

Performance Monitoring

Timing Operations: `%%timeit` / `time`

Measure execution time of Python/Pandas code.

```
# Jupyter/IPython magic command for timing a line/cell
%%timeit
df['col'].apply(lambda x: x*2) # Example operation

import time
start_time = time.time()
# Your Pandas code here
end_time = time.time()
print(f"Execution time: {end_time - start_time} seconds")
```

Optimized Operations: `eval()` / `query()`

Utilize these methods for faster performance on large DataFrames, especially for element-wise operations and filtering.

```
# Faster than `df['col1'] + df['col2']`
df['new_col'] = df.eval('col1 + col2')
# Faster filtering
df_filtered = df.query('col1 > @threshold and col2 ==
"value"')
```

Profiling Code: `cProfile` / `line_profiler`

Analyze where time is spent in your Python functions.

```
import cProfile
def my_pandas_function(df):
    # Pandas operations
    return df.groupby('col').mean()
cProfile.run('my_pandas_function(df)') # Run function
with cProfile

# For line_profiler (install with pip install line_profiler):
# @profile
# def my_function(df):
#    ...
# %load_ext line_profiler
# %lprun -f my_function my_function(df)
```

Pandas Installation & Setup

Install and manage the Pandas library for your Python environment.

Pip: `pip install pandas`

Standard Python package installer.

```
# Install Pandas
pip install pandas
# Upgrade Pandas to the latest
version
pip install pandas --upgrade
# Show installed Pandas
package information
pip show pandas
```

Conda: `conda install pandas`

Package manager for Anaconda/Miniconda environments.

```
# Install Pandas in current
conda env
conda install pandas
# Update Pandas
conda update pandas
# List installed Pandas package
conda list pandas
# Create new env with Pandas
conda create -n myenv pandas
```

Check Version / Import

Verify your Pandas installation and import it in your scripts.

```
# Standard import alias
import pandas as pd
# Check installed Pandas
version
print(pd.__version__)
# Display all columns
pd.set_option('display.max_colu
mns', None)
# Display more rows
pd.set_option('display.max_rows
', 100)
```

Configuration & Settings

Adjust Pandas options to control display, warning behavior, and more.

Display Options: `pd.set_option()`

Control how DataFrames are displayed in the console/Jupyter.

```
# Max rows to display
pd.set_option('display.max_rows', 50)
# Display all columns
pd.set_option('display.max_columns', None)
# Width of the display
pd.set_option('display.width', 1000)
# Format float values
pd.set_option('display.float_format', '{:.2f}'.format)
```

Reset Options: `pd.reset_option()`

Reset a specific option or all options to their default values.

```
# Reset specific option
pd.reset_option('display.max_rows')
# Reset all options to default
pd.reset_option('all')
```

Getting Options: `pd.get_option()`

Retrieve the current value of a specified option.

```
# Get current max_rows setting
print(pd.get_option('display.max_rows'))
```

Context Manager: `pd.option_context()`

Temporarily set options within a `with` statement.

```
with pd.option_context('display.max_rows', 10,
'display.max_columns', 5):
    print(df) # DataFrame displayed with temporary
options
print(df) # Options revert to previous settings outside the
block
```

Method Chaining

Chain multiple Pandas operations together for cleaner and more readable code.

Chaining Operations

Apply a sequence of transformations to a DataFrame.

```
(
    df.dropna(subset=['col1'])
    .assign(new_col = lambda x: x['col2'] * 2)
    .query('new_col > 10')
    .groupby('category_col')
    ['new_col']
    .mean()
    .reset_index()
)
```

Using `.pipe()`

Apply functions that take the DataFrame as their first argument, enabling custom steps in a chain.

```
def custom_filter(df, threshold):
    return df[df['value'] > threshold]

(
    df.pipe(custom_filter, threshold=50)
    .groupby('group')
    .agg(total_value=('value', 'sum'))
)
```

Reference: This cheatsheet covers essential Pandas commands and modern practices for efficient data manipulation and analysis in data science workflows.