Pandas Cheat Sheet for Quick Basic Fundamentals

Welcome to your comprehensive quick reference guide for **Pandas**! This cheat sheet covers the fundamental concepts you need to get started with Pandas for data manipulation and analysis, enhanced with extended examples for better understanding and quick revision.

Table of Contents

Introduction

Overview: Pandas is a powerful open-source data analysis and manipulation library for Python. It provides data structures like **Series** and **DataFrame** that make handling structured data intuitive and efficient.

Key Features:

- Easy handling of missing data
- Size mutability
- Automatic and explicit data alignment
- Powerful, flexible group by functionality
- Time series-specific functionality

Installation and Setup

Overview: Set up Pandas on your local machine to start data analysis seamlessly.

Using pip

If you have Python installed, you can install Pandas using `pip`.

```
pip install pandas
```

Using Anaconda

Anaconda distribution comes with Pandas pre-installed.

1. **Download Anaconda:**
    - Visit Anaconda Downloads and download the installer for your operating system.
2. **Install Anaconda:**
    - Follow the installation instructions specific to your OS.
3. **Verify Installation:**

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

Importing Pandas

```
import pandas as pd
```

Data Structures

Overview: Understand the primary data structures in Pandas: **Series** and **DataFrame**.

Series

A one-dimensional labeled array capable of holding any data type.

```
import pandas as pd

# Creating a Series from a list
data = [10, 20, 30, 40, 50]
s = pd.Series(data)
print(s)

# Creating a Series with custom index
s = pd.Series(data, index=['a', 'b', 'c', 'd', 'e'])
print(s)
```

DataFrame

A two-dimensional labeled data structure with columns of potentially different types.

```
# Creating a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)
print(df)

# Creating a DataFrame with custom index
df = pd.DataFrame(data, index=['id1', 'id2', 'id3'])
print(df)
```

Reading and Writing Data

Overview: Import and export data between Pandas and various file formats.

Reading Data

From CSV

```
df = pd.read_csv('data.csv')
print(df.head())
```

From Excel

```
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
print(df.head())
```

From JSON

```
df = pd.read_json('data.json')
print(df.head())
```

From SQL

```
import sqlite3

# Establishing a connection
conn = sqlite3.connect('database.db')

# Reading from SQL
df = pd.read_sql_query("SELECT * FROM table_name", conn)
print(df.head())

# Closing the connection
conn.close()
```

Writing Data

To CSV

```
df.to_csv('output.csv', index=False)
```

To Excel

```
df.to_excel('output.xlsx', sheet_name='Sheet1', index=False)
```

To JSON

```
df.to_json('output.json', orient='records', lines=True)
```

To SQL

```
import sqlite3

# Establishing a connection
conn = sqlite3.connect('database.db')

# Writing to SQL
df.to_sql('table_name', conn, if_exists='replace', index=False)

# Closing the connection
conn.close()
```

Basic Operations

Overview: Perform fundamental operations to explore and understand your data.

Viewing Data

- **Head and Tail:**

```
print(df.head())  # First 5 rows
print(df.tail(3)) # Last 3 rows
```

- **Info and Describe:**

```
print(df.info())        # Summary of DataFrame
print(df.describe())    # Statistical summary
```

- **Shape and Size:**

```
print(df.shape)  # (rows, columns)
print(df.size)   # Total number of elements
```

Selecting Columns

```
# Single column
print(df['Name'])

# Multiple columns
print(df[['Name', 'City']])
```

Selecting Rows

- **By Position:**

```
print(df.iloc[0])        # First row
print(df.iloc[1:3])      # Second and third rows
```

- **By Label:**

```
print(df.loc['id1'])     # Row with index 'id1'
print(df.loc[['id1', 'id3']])
```

Summary Statistics

```
print(df['Age'].mean())   # Mean age
print(df['Age'].median()) # Median age
print(df['Age'].sum())    # Sum of ages
```

Indexing and Selecting Data

Overview: Efficiently access and manipulate subsets of your data.

Setting and Resetting Index

```
# Setting a column as index
df.set_index('Name', inplace=True)
print(df)

# Resetting index
df.reset_index(inplace=True)
print(df)
```

Selecting Data with loc and iloc

- **loc (Label-based):**

```
# Single label
print(df.loc['Alice'])

# Multiple labels
print(df.loc[['Alice', 'Bob']])

# Label slicing
print(df.loc['Alice':'Charlie'])
```

- **iloc (Position-based):**

```
# Single position
print(df.iloc[0])

# Multiple positions
print(df.iloc[[0, 2]])

# Position slicing
print(df.iloc[0:2])
```

Boolean Indexing

```
# Filter rows where Age > 30
filtered_df = df[df['Age'] > 30]
print(filtered_df)

# Multiple conditions
filtered_df = df[(df['Age'] > 25) & (df['City'] == 'Chicago')]
print(filtered_df)
```

Selecting with Conditions

```
# Using query
filtered_df = df.query('Age > 25 and City == "Chicago"')
print(filtered_df)
```

Handling Missing Data

Overview: Identify, handle, and clean missing data in your dataset.

Identifying Missing Data

```
# Detect missing values
print(df.isnull())

# Count missing values per column
print(df.isnull().sum())
```

Dropping Missing Data

```
# Drop rows with any missing values
df_clean = df.dropna()

# Drop columns with any missing values
df_clean = df.dropna(axis=1)

# Drop rows where specific columns are missing
df_clean = df.dropna(subset=['Age', 'City'])
```

Filling Missing Data

```
# Fill missing values with a specific value
df_filled = df.fillna(0)
```

```
# Fill missing values with mean of the column
df['Age'] = df['Age'].fillna(df['Age'].mean())

# Forward fill
df_filled = df.fillna(method='ffill')

# Backward fill
df_filled = df.fillna(method='bfill')
```

Replacing Missing Values

```
# Replace NaN with a specific value
df.replace(to_replace=np.nan, value=0, inplace=True)
```

Data Cleaning

Overview: Clean and prepare your data for analysis by handling inconsistencies and formatting issues.

Renaming Columns

```
# Rename columns using a dictionary
df.rename(columns={'Name': 'Full Name', 'Age': 'Years'}, inplace=True)
print(df)
```

Changing Data Types

```
# Convert column to integer
df['Years'] = df['Years'].astype(int)

# Convert column to string
df['City'] = df['City'].astype(str)

# Convert column to datetime
df['Join Date'] = pd.to_datetime(df['Join Date'])
```

Removing Duplicates

```
# Remove duplicate rows
df.drop_duplicates(inplace=True)

# Remove duplicates based on a specific column
df.drop_duplicates(subset=['Name'], inplace=True)
```

String Manipulation

```
# Convert to lowercase
df['City'] = df['City'].str.lower()

# Remove leading/trailing whitespace
df['Name'] = df['Name'].str.strip()

# Replace substrings
df['City'] = df['City'].str.replace('new york', 'NYC')
```

Handling Outliers

```
# Identify outliers using IQR
Q1 = df['Age'].quantile(0.25)
Q3 = df['Age'].quantile(0.75)
IQR = Q3 - Q1

# Filter out outliers
df_filtered = df[(df['Age'] >= (Q1 - 1.5 * IQR)) & (df['Age'] <= (Q3 + 1.5 *
IQR))]
print(df_filtered)
```

Parsing Dates

```
# Convert string to datetime
df['Join Date'] = pd.to_datetime(df['Join Date'], format='%Y-%m-%d')

# Extract year, month, day
df['Year'] = df['Join Date'].dt.year
df['Month'] = df['Join Date'].dt.month
df['Day'] = df['Join Date'].dt.day
```

Data Manipulation

Overview: Transform and reshape your data to suit your analysis needs.

Merging DataFrames

```
# Merge on a common column
df_merged = pd.merge(df1, df2, on='ID', how='inner')
print(df_merged)
```

Concatenating DataFrames

```
# Concatenate vertically
df_concat = pd.concat([df1, df2], axis=0)
print(df_concat)

# Concatenate horizontally
df_concat = pd.concat([df1, df2], axis=1)
print(df_concat)
```

Joining DataFrames

```
# Join using index
df_joined = df1.join(df2, how='left')
print(df_joined)
```

Pivot Tables

```
# Create a pivot table
pivot = pd.pivot_table(df, values='Sales', index='Region', columns='Product',
aggfunc='sum')
print(pivot)
```

Melting DataFrames

```
# Melt DataFrame from wide to long format
df_melted = pd.melt(df, id_vars=['ID', 'Name'], value_vars=['Sales_Q1',
'Sales_Q2'], var_name='Quarter', value_name='Sales')
print(df_melted)
```

Stacking and Unstacking

```
# Stack DataFrame
df_stacked = df.stack()
print(df_stacked)

# Unstack DataFrame
df_unstacked = df_stacked.unstack()
print(df_unstacked)
```

Adding and Removing Columns

```
# Add a new column
df['Total Sales'] = df['Sales_Q1'] + df['Sales_Q2']

# Remove a column
df.drop('Total Sales', axis=1, inplace=True)
```

Grouping and Aggregation

Overview: Perform group-wise operations to summarize and analyze data.

Group By

```
# Group by a single column and calculate mean
grouped = df.groupby('City')['Age'].mean()
print(grouped)

# Group by multiple columns and aggregate
grouped = df.groupby(['City', 'Gender']).agg({'Age': 'mean', 'Sales': 'sum'})
print(grouped)
```

Aggregation Functions

```
# Multiple aggregation functions
agg_functions = df.groupby('City').agg({
    'Age': ['mean', 'min', 'max'],
    'Sales': ['sum', 'count']
})
print(agg_functions)
```

Transformations

```
# Standardize sales within each group
df['Sales_Standardized'] = df.groupby('City')['Sales'].transform(lambda x: (x -
x.mean()) / x.std())
print(df)
```

Filtering Groups

```
# Filter groups with total sales > 1000
filtered_groups = df.groupby('City').filter(lambda x: x['Sales'].sum() > 1000)
print(filtered_groups)
```

Sorting and Filtering

Overview: Arrange your data and extract subsets based on specific criteria.

Sorting

```
# Sort by a single column
df_sorted = df.sort_values(by='Age')
print(df_sorted)

# Sort by multiple columns
df_sorted = df.sort_values(by=['City', 'Sales'], ascending=[True, False])
print(df_sorted)

# Sort by index
df_sorted = df.sort_index()
print(df_sorted)
```

Filtering

```
# Filter rows where Sales > 500
filtered_df = df[df['Sales'] > 500]
print(filtered_df)

# Filter using isin
cities = ['New York', 'Chicago']
filtered_df = df[df['City'].isin(cities)]
print(filtered_df)
```

Selecting Specific Rows and Columns

```
# Select specific rows and columns
subset = df.loc[10:20, ['Name', 'Sales']]
print(subset)

# Select using iloc
subset = df.iloc[0:5, 1:3]
print(subset)
```

Applying Functions

Overview: Enhance your data manipulation by applying custom functions to your data.

Using apply()

```
# Apply a function to a column
df['Sales_Doubled'] = df['Sales'].apply(lambda x: x * 2)
print(df)

# Apply a function to each row
def categorize_age(row):
    if row['Age'] < 30:
        return 'Young'
    elif row['Age'] < 50:
        return 'Middle-aged'
    else:
        return 'Senior'

df['Age_Category'] = df.apply(categorize_age, axis=1)
print(df)
```

Using map()

```
# Map values in a column
gender_map = {'M': 'Male', 'F': 'Female'}
df['Gender_Full'] = df['Gender'].map(gender_map)
print(df)
```

Using applymap()

```
# Apply a function to every element in the DataFrame
df_numeric = df.select_dtypes(include=['int64', 'float64'])
df_numeric = df_numeric.applymap(lambda x: x * 100)
print(df_numeric)
```

Using vectorized operations

```
# Perform operations directly on columns
df['Sales_Percentage'] = (df['Sales'] / df['Sales'].sum()) * 100
print(df)
```

Visualization with Pandas

Overview: Create quick and insightful visualizations directly from Pandas DataFrames.

Basic Plotting

```
import matplotlib.pyplot as plt

# Line plot
df['Sales'].plot(kind='line')
plt.title('Sales Over Time')
plt.xlabel('Index')
plt.ylabel('Sales')
plt.show()

# Bar plot
df.groupby('City')['Sales'].sum().plot(kind='bar')
plt.title('Total Sales by City')
plt.xlabel('City')
plt.ylabel('Total Sales')
plt.show()

# Histogram
df['Age'].plot(kind='hist', bins=10, edgecolor='black')
plt.title('Age Distribution')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()

# Scatter plot
df.plot(kind='scatter', x='Age', y='Sales')
plt.title('Sales vs Age')
plt.xlabel('Age')
plt.ylabel('Sales')
plt.show()
```

Advanced Plotting

```
# Box plot
df.boxplot(column='Sales', by='City')
plt.title('Sales Distribution by City')
plt.xlabel('City')
plt.ylabel('Sales')
plt.show()

# Pie chart
df.groupby('Gender')['Sales'].sum().plot(kind='pie', autopct='%1.1f%%')
plt.title('Sales Distribution by Gender')
plt.ylabel('')
plt.show()
```

Customizing Plots

```
# Customize plot size and style
plt.figure(figsize=(10, 6))
plt.style.use('ggplot')
df['Sales'].plot(kind='line', color='green', linestyle='--', marker='o')
plt.title('Customized Sales Line Plot')
plt.xlabel('Index')
plt.ylabel('Sales')
plt.legend(['Sales'])
plt.grid(True)
plt.show()
```

Time Series

Overview: Handle and analyze time series data effectively using Pandas.

Converting to Datetime

```
# Convert string to datetime
df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d')
print(df.dtypes)
```

Setting Datetime as Index

```
# Set datetime column as index
df.set_index('Date', inplace=True)
print(df.head())
```

Resampling

```
# Resample data to monthly frequency and sum sales
monthly_sales = df['Sales'].resample('M').sum()
print(monthly_sales)

# Plot resampled data
monthly_sales.plot(kind='line')
plt.title('Monthly Sales')
plt.xlabel('Month')
plt.ylabel('Sales')
plt.show()
```

Rolling Window Calculations

```
# Calculate 7-day rolling average
df['Sales_Rolling_Avg'] = df['Sales'].rolling(window=7).mean()
print(df[['Sales', 'Sales_Rolling_Avg']].head(10))

# Plot rolling average
df[['Sales', 'Sales_Rolling_Avg']].plot()
plt.title('Sales and 7-Day Rolling Average')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.show()
```

Shifting Data

```
# Shift data by one day
df['Sales_Shifted'] = df['Sales'].shift(1)
print(df[['Sales', 'Sales_Shifted']].head())
```

Time-based Indexing

```
# Select data for a specific month
january_data = df['2021-01']
print(january_data)

# Select data between two dates
subset = df['2021-01':'2021-03']
print(subset)
```

Advanced Topics

Overview: Dive into more complex features of Pandas to handle sophisticated data analysis tasks.

MultiIndex

```
# Creating a MultiIndex DataFrame
arrays = [
    ['Group1', 'Group1', 'Group2', 'Group2'],
    ['A', 'B', 'A', 'B']
]
index = pd.MultiIndex.from_arrays(arrays, names=('Group', 'Subgroup'))
df_multi = pd.DataFrame({'Value': [10, 20, 30, 40]}, index=index)
print(df_multi)

# Accessing data in MultiIndex
```

```
print(df_multi.loc['Group1'])
print(df_multi.loc[('Group2', 'B')])
```

Pivot and Pivot Table

```
# Pivot
df_pivot = df.pivot(index='Date', columns='City', values='Sales')
print(df_pivot)

# Pivot Table with aggregation
df_pivot_table = pd.pivot_table(df, values='Sales', index='Date', columns='City',
aggfunc='sum')
print(df_pivot_table)
```

Window Functions

```
# Cumulative sum
df['Cumulative_Sales'] = df['Sales'].cumsum()
print(df[['Sales', 'Cumulative_Sales']].head())

# Expanding mean
df['Expanding_Mean'] = df['Sales'].expanding().mean()
print(df[['Sales', 'Expanding_Mean']].head())
```

Advanced Merging

```
# Merging with multiple keys
df_merged = pd.merge(df1, df2, on=['ID', 'Date'], how='outer')
print(df_merged)

# Merging with indicator
df_merged = pd.merge(df1, df2, on='ID', how='outer', indicator=True)
print(df_merged)
```

Categorical Data

```
# Convert column to categorical
df['Category'] = df['Category'].astype('category')
print(df['Category'].cat.categories)

# Add category
df['Category'] = df['Category'].cat.add_categories(['New_Category'])
df['Category'].fillna('New_Category', inplace=True)
print(df['Category'])
```

Memory Optimization

```
# Optimize data types
df['Age'] = df['Age'].astype('int8')
df['Sales'] = df['Sales'].astype('float32')
print(df.dtypes)
```

Best Practices

Overview: Adopt best practices to write efficient, readable, and maintainable Pandas code.

Use Vectorized Operations

- **Avoid loops:** Utilize Pandas' built-in functions for better performance.

```
# Instead of looping
df['Sales_Double'] = 0
for i in range(len(df)):
    df.at[i, 'Sales_Double'] = df.at[i, 'Sales'] * 2

# Use vectorized operation
df['Sales_Double'] = df['Sales'] * 2
```

Chain Methods

- **Method chaining:** Perform multiple operations in a single, readable line.

```
df_clean = (df.dropna()
            .rename(columns={'Name': 'Full_Name'})
            .sort_values(by='Sales', ascending=False))
print(df_clean)
```

Handle Missing Data Early

- **Address missing values** at the beginning to prevent issues later in analysis.

Use Meaningful Variable Names

- **Clarity:** Use descriptive names for DataFrames and columns.

```
# Good
sales_data = pd.read_csv('sales.csv')
```

```
# Bad
df1 = pd.read_csv('sales.csv')
```

Document Your Code

- **Comments and Docstrings:** Explain complex operations and transformations.

```
# Calculate total sales per city
total_sales_city = df.groupby('City')['Sales'].sum()
```

Avoid SettingWithCopyWarning

- **Understand copying vs. referencing:** Use `.copy()` when necessary to prevent unexpected behavior.

```
subset = df[df['Sales'] > 500].copy()
subset['Sales'] = subset['Sales'] * 2
```

Profile Your Data

- **Understand your data** before performing transformations.

```
print(df.info())
print(df.describe())
print(df.head())
```

Utilize Pandas Documentation

- **Stay updated:** Refer to Pandas Documentation for comprehensive guides and references.

Tips for Beginners

Overview: Enhance your learning experience with these essential tips for new Pandas users.

Start with Tutorials

- **Official Pandas Tutorials:** Pandas Getting Started
- **Interactive Platforms:** DataCamp and Kaggle offer hands-on Pandas courses.

Practice with Real Datasets

- **Kaggle Datasets:** Explore and analyze datasets from Kaggle.
- **UCI Machine Learning Repository:** Access datasets at UCI Repository.

Learn by Projects

- **Mini Projects:** Start with small projects like data cleaning, analysis, and visualization.
- **Capstone Projects:** Gradually move to comprehensive projects integrating multiple Pandas functionalities.

Utilize Keyboard Shortcuts in IDEs

- **Boost Productivity:** Learn shortcuts in IDEs like Jupyter Notebook, VSCode, or PyCharm for efficient coding.

Explore Pandas Extensions

- **Additional Functionality:** Libraries like Dask for parallel computing or Pandas-Profiling for automated profiling.

Participate in Communities

- **Forums and Q&A:** Engage with communities on Stack Overflow and Reddit.
- **GitHub Repositories:** Explore and contribute to open-source Pandas projects.

Keep Learning and Updating

- **Stay Current:** Pandas is actively developed. Keep an eye on updates and new features.
- **Advanced Topics:** As you progress, delve into advanced topics like performance optimization and integration with other libraries.

Additional Resources

Overview: Access a curated list of resources to deepen your understanding of Pandas.

Official Documentation

- **Pandas Documentation:** https://pandas.pydata.org/docs/
- **Pandas Tutorials:** https://pandas.pydata.org/pandas-docs/stable/getting_started/tutorials.html

Tutorials and Guides

- **Real Python:** Pandas Tutorials
- **DataCamp:** Pandas Courses
- **Kaggle:** Pandas Exercises

Books

- **"Python for Data Analysis" by Wes McKinney:** Comprehensive guide by the creator of Pandas.
- **"Pandas Cookbook" by Theodore Petrou:** Practical recipes for mastering Pandas.

Online Communities

- **Stack Overflow:** Pandas Questions
- **Reddit:** r/pandas
- **GitHub:** Pandas Repository

Videos and Webinars

- **YouTube Tutorials:** Search for "Pandas Tutorial" for video guides.
- **Official Pandas YouTube Channel:** Pandas YouTube

Cheat Sheets

- **Pandas Cheat Sheet by DataCamp:** Download PDF
- **Pandas Cheat Sheet by Data School:** Download PDF

Tools and Extensions

- **Pandas Profiling:** https://pandas-profiling.github.io/pandas-profiling/docs/master/rtd/
- **Dask:** https://dask.org/ for scalable data analysis.
- **Vaex:** https://vaex.io/ for out-of-core DataFrames.

Happy Coding!

Leverage this cheat sheet to streamline your Pandas experience. Practice regularly, explore new features, and engage with the community to become proficient in using Pandas for your data analysis projects.