Python Data Analysis Code Cheatsheet

This cheatsheet is your post-program reference toolkit, this generic template is designed to help you quickly recall and apply the key code patterns you've learned throughout your data analysis journey.

How to use it:

1. During the program: Use this as your personal playbook. When working on projects, solving problems, revisiting concepts later, come back to this sheet to copy, tweak, and run the code you need.

2. When in doubt: Think of this as your first stop before Googling or digging through notes.

Use this cheatsheet as your go-to companion whenever you need to work with data in Python. You've done the learning - now here's the code to get it done.

Data Preparation Process	Workflow	Code

Getting Started	Import Python Modules	`# Import the required Python Modules`
		
		`import pandas as pd`
		`import numpy as np`
		`import matplotlib.pyplot as plt`
		`import seaborn as sns`

	loading Dataset	`# Load the dataset`
		`df = pd.read_csv("DATASETNAME.csv")`
		
		`# Note: DF is the name of the data frame. If you are working with multiple data frames, you will need to use the appropriate data frame name based on how you defined them.`

Exploring the data	Print Table	`# Display the first 5 rows`

frame		df.head() # Display the first 5 rows of specific COLUMNS. Note you can apply this concept of selecting specific columns for subsequent codes. df[['COLUMN1', 'COLUMN2']].head()

	Row and column	# Check the shape of the dataset
		print(f"Dataset contains {df.shape[0]} rows and {df.shape[1]} columns.")
		
		# Note: [0] refers to rows, while [1] refers to columns.

	Identify missing values - using.info()	df.info()

Data Cleaning	Identify missing values - using isnull()	# Check for missing values
		print("Missing Values Count:\n", df.isnull().sum())
		
		# Display rows with missing values
		df[df.isnull().any(axis=1)]

	Handling missing values - Removal method	#Remove rows with missing values in the COLUMN. Save this as a new variable called df2
		df2 = df.dropna(subset=['COLUMN'])
		
		#Verify if the process was completed by checking the number of remaining missing values
		print("Missing values after cleaning:")
		print(df2.isnull().sum())
		
		#note that it is best practice to create new data frames when we make

		any changes so that we can go back to it

	Handling missing values - Imputation method	# Fill missing values in the 'Price' column with the mean. Save this as a new variable called df2
		df2['COLUMN'] = df['COLUMN'].fillna(df['COLUMN'].mean())
		
		# Note that you can change to median by changing .mean to .median
		
		# Verify the missing values are handled
		print("Missing values after performing imputation:")
		print(df2['Price'].isnull().sum())

	Handling duplicates in data frames	#Check for duplicates
		duplicates = df_cleaned2.duplicated().sum()
		print(f"Number of duplicate rows: {duplicates}")
		
		#Display duplicate rows if any exist
		if duplicates > 0:
		    print("Duplicate rows found:")
		    display(df_cleaned2[df_cleaned2.duplicated(keep=False)]) # Show all duplicate rows (including first occurrence)
		
		# Remove duplicates if necessary
		df_cleaned2 = df_cleaned2.drop_duplicates()
		
		# Verify removal
		print(f"Number of duplicate rows after removal: {df_cleaned2.duplicated().sum()}")

	Export the cleaned datasets for secondary storage	# Export data frame into CSV
		df2.to_csv('DATASET NAME.csv', index=False)

Data Manipulation	Creating New Columns	# Creating new columns in the table based on another column df['NEW_COLUMN'] = df['OLD_COLUMN']

		# Note: You can personalise the formulas accordingly. Ie: If we want to apply a 10% discount, you can multiply the OLD COLUMN by 0.9

	Encoding Categorical Data into Numerical Data	# Define your custom mapping for categories
		category_mapping = {
		    'CATEGORY 1': 0,
		    'CATEGORY 2': 1
		}
		
		# Apply the mapping
		df['NEW_COLUMN'] = df['OLD_COLUMN'].map(category_mapping)

	Extracting from datetime columns	# Convert the 'DATE_COLUMN' into datetime format df['DATE_COLUMN'] = pd.to_datetime(df['DATE_COLUMN']) # Create new columns that extracts the YEAR, MONTH, DAY, and DAY_OF_WEEK from the DATE_COLUMN df['YEAR'] = df['DATE_COLUMN'].dt.year df['MONTH'] = df['DATE_COLUMN'].dt.month df['DAY'] = df['DATE_COLUMN'].dt.day df['DAY_OF_WEEK'] = df['DATE_COLUMN'].dt.day_name() # Display the transformed dataset df[['DATE_COLUMN', 'YEAR', 'MONTH', 'DAY', 'DAY_OF_WEEK']].head()
Data Transformation	Converting data types in a new column	# Create a new column and changes it into int data type. df['COLUMN_INT'] = df['COLUMN'].astype(int) # Note: You can convert to other data types by changing ‘int’ with others (ie: float, str)
	Convert date and time to string format	df['Date_of_Visit'] = pd.to_datetime(df['Date_of_Visit'], errors='coerce', dayfirst=True).dt.strftime('%d-%m-%Y')

		`df.head()`

	Ensure consistent formatting of date & time	`# Convert 'DATE_COLUMN' into datetime format and displays it as a DD-MM-YYYY format`
		`df['DATE_COLUMN'] = pd.to_datetime(df['DATE_COLUMN'], errors='coerce').dt.strftime('%d-%m-%Y')`
		
		`# Note: You can personalise it based on the format of date that you want to display.`

	Ensure consistent formatting of categorical data	`# Standardise categorical data in a column with lower case`
		`df[COLUMN] = df['COLUMN'].str.lower().str.strip()`
		
		`# Verify unique values after data transformation`
		`df['Weather'].unique()`

	Information Range checks	`# Replace null or negative prices with median price`
		`df['COLUMN'] = df['COLUMN'].apply(lambda x: x if pd.notnull(x) and x > 0 else df['COLUMN'].median())`
		
		`# Verify if any prices are still missing or invalid`
		`df[df[‘COLUMN’] <= 0]`
		
		`# Note: You can replace the highlighted parts with any range of your choice, and replace median with your preferred choice of handling data that is out of range.`

Data Joining	GroupBy - on Numerical Data	`# Standardise a categorical column`
		`# Replace <column_name> with the actual column header you want to clean`
		`df['<column_name>'] = df['<column_name>'].str.lower().str.strip()`

<table>
<tr>
<td></td>
<td></td>
<td>

```python
# Quick check
print(df['<column_name>'].unique())
```

</td>
</tr>
<tr>
<td></td>
<td>Merge</td>
<td>

```python
# Merge two DataFrames
# Replace <left_df>, <right_df>, and <join_key> with your actual names
merged_df = <left_df>.merge(<right_df>, on="<join_key>", how="left")

# Inspect the result
merged_df.head()
```

</td>
</tr>
<tr>
<td>Exploring Data Analysis (EDA)</td>
<td>Summarise Numerical variable- describe ( )</td>
<td>

```python
# STEP 1: Drop identifier or helper columns that you don’t want in the summary
numerical_data = <df>.drop(columns=<id_columns>)

# STEP 2: Keep only numeric columns
numerical_columns = numerical_data.select_dtypes(include="number")

# STEP 3: Generate descriptive statistics
numerical_summary = numerical_columns.describe()

print(numerical_summary)
```

</td>
</tr>
<tr>
<td></td>
<td>Explore categorical variable</td>
<td>

```python
# STEP 1 ► Identify all categorical columns (dtype "object")
categorical_columns = <df>.select_dtypes(include="object").columns

print("\nFrequency distribution for categorical variables:")

# STEP 2 ► Loop through each categorical column
for col in categorical_columns:

    # STEP 3 ► Display absolute frequencies for that column
    print(f"\nColumn: {col}")
    print(<df>[col].value_counts())

    # OPTIONAL ► Uncomment to display relative frequencies
```

</td>
</tr>
</table>

<table>
<tr><td></td><td></td><td><pre>(percentages)
    # print(<df>[col].value_counts(normalize=True))</pre></td></tr>
<tr><td></td><td>Display the frequency of categories</td><td><pre>import pandas as pd

# STEP 1 ► Identify categorical columns (dtype "object")
categorical_columns = <DF>.select_dtypes(include="object").columns

print("\nFrequency distribution for categorical variables:")

# STEP 2 ► Loop through each categorical column and print the counts
for col in categorical_columns:
    print(f"\nColumn: {col}")
    print(<DF>[col].value_counts())   # absolute frequencies

    # OPTIONAL ► Uncomment to show relative frequencies (percentages)
    # print(<DF>[col].value_counts(normalize=True))</pre></td></tr>
<tr><td>Data Visualisation</td><td>Visualising Trends : Line plot</td><td><pre># STEP 1 ► Group data by <group_col> to calculate total visits (row
count)
visits_by_category = (
    <DF>
        .groupby('<group_col>')
        .size()   # counts rows in each
group
        .reset_index(name='<count_col>')   # renames the count column
)

# OPTIONAL ► Sort categories in a custom order (e.g. calendar order)
custom_order = ['Jan', 'Feb', 'Mar', 'Apr', 'May']   # edit to your
needs
visits_by_category['<group_col>'] = pd.Categorical(
    visits_by_category['<group_col>'],
    categories=custom_order,
    ordered=True</pre></td></tr>
</table>

<table>
<tr><td></td><td></td><td>

```
)
visits_by_category = visits_by_category.sort_values('<group_col>')

# STEP 2 ► Plot the trend with a line chart
plt.figure(figsize=(10, 5))
plt.plot(
    visits_by_category['<group_col>'],
    visits_by_category['<count_col>'],
    marker='o',
    linestyle='-'              # solid line
    # color parameter is optional; Matplotlib uses a default palette
)

plt.xlabel("<group_col>")
plt.ylabel("<count_col>")
plt.title(f"Total {<count_col>} by {<group_col>}")
plt.xticks(rotation=45)
plt.grid(True)
plt.tight_layout()
plt.show()
```

</td></tr>
<tr><td></td><td>Visualising comparison: Bar Chart</td><td>

```
# STEP 1 ► Group by <group_col> to calculate the average of <agg_col>
df_grouped = (
    <DF>
        .groupby('<group_col>')[ '<agg_col>' ]
        .mean()
        .reset_index()
)

# STEP 2 ► Sort the result in descending order of the average value
df_grouped_sorted = df_grouped.sort_values(by='<agg_col>',
ascending=False)

# STEP 3 ► Plot the bar chart
plt.figure(figsize=(8, 5))
```

</td></tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td><pre>plt.bar(
    df_grouped_sorted['<group_col>'],
    df_grouped_sorted['<agg_col>'],
)

# STEP 4 ► Customise the chart
plt.xlabel('<group_col>')
plt.ylabel(f'Average <agg_col>')
plt.title(f'Average <agg_col> by <group_col>')
plt.xticks(rotation=45)
plt.tight_layout()

STEP 5 ► Display the plot
plt.show()</pre></td>
</tr>
<tr>
<td></td>
<td>Visualising distribution:
Histogram</td>
<td><pre># Step 1: Create histogram
plt.figure(figsize=(8, 5))
sns.histplot(<DF>['<numeric_col>'], bins=20, kde=True, color='green')

# Step 2: Customize the plot
plt.xlabel("<numeric_col>")
plt.ylabel("Frequency")
plt.title("Distribution of <numeric_col>")

# Step 3: Show the plot
plt.tight_layout()
plt.show()</pre></td>
</tr>
<tr>
<td></td>
<td>Visualising composition:
Pie Chart</td>
<td><pre># Step 1: Group by <group_col> and sum <value_col>
grouped_data = (
    <DF>
        .groupby('<group_col>')['<value_col>']
        .sum()
        .reset_index()
)</pre></td>
</tr>
</table>

```python
# Step 2: Create a pie chart
plt.figure(figsize=(8, 8))
plt.pie(
    grouped_data['<value_col>'],
    labels=grouped_data['<group_col>'],
    autopct='%1.2f%%',
    startangle=140
)

# Step 3: Customize the chart
plt.title(f"Total {<value_col>} by {<group_col>}")
plt.tight_layout()

# Step 4: Display the chart
plt.show()
```