Pandas Advanced Cheat Sheet

1. Performance & Memory

Tools for speeding up operations and reducing memory footprint.

‘query()‘ and ‘eval()‘

Use string expressions for faster and more readable filtering and column creation, especially on large DataFrames.

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.rand(1000, 3),
                  columns=['A', 'B', 'C'])

# Instead of: df[(df['A'] > 0.5) & (df['B'] < 0.5)]
# Use .query() for cleaner, faster filtering:
df.query('A > 0.5 and B < 0.5')

# Instead of: df['D'] = df['A'] + df['B']
# Use .eval() to create new columns:
df.eval('D = A + B', inplace=True)
```

Memory Optimization

Downcast numeric types to use less memory.

```
df.info(memory_usage='deep') # Check memory
# Use to_numeric to safely downcast
df['int_col'] = pd.to_numeric(df['int_col'],
                              downcast='integer')
df['float_col'] = pd.to_numeric(df['float_col'],
                                downcast='float')
df.info(memory_usage='deep') # Check again
```

2. Advanced Window Operations

Beyond simple rolling windows.

‘expanding()‘ Windows

Calculates a cumulative value from the start of the series.

```
s = pd.Series([1, 2, 3, 4, 5])
# Cumulative sum
s.expanding().sum() # Returns: [1., 3., 6., 10.,
    15.]
```

‘ewm()‘ (Exponentially Weighted)

Gives more weight to more recent observations. Common in finance and signal processing.

```
# Exponentially weighted moving average
s.ewm(span=3).mean() # span is decay factor
```

3. Efficient I/O for Large Data

Handle files that are too large to fit in memory.

Reading CSVs in Chunks

Process a large file piece by piece.

```
chunk_iter = pd.read_csv('huge_file.csv',
                         chunksize=10000)
for chunk in chunk_iter:
    # Process each chunk DataFrame here
    print(chunk['value'].mean())
```

Using Parquet Format

A fast, compressed, columnar storage format. Often much faster and smaller than CSV. Requires ‘pyarrow‘ or ‘fastparquet‘.

```
# Save to Parquet
df.to_parquet('my_data.parquet')

# Read from Parquet
df_loaded = pd.read_parquet('my_data.parquet')
\end{stlisting}

% --- SECTION 4: THE STYLER API ---
\section*{4. The Styler API}
Format and style DataFrames for better visualization
    and reporting in notebooks. Returns a ‘Styler‘
    object.
\begin{lstlisting}
df = pd.DataFrame(np.random.randn(5, 4),
                  columns=['A', 'B', 'C', 'D'])

# Chain styling methods
(df.style
  .background_gradient(cmap='viridis')
  .format('{:.2f}')
  .highlight_max(axis=0, color='red')
  .set_caption('Styled DataFrame'))
\end{stlisting}

% --- SECTION 5: EXTENDING PANDAS ---
\section*{5. Extending Pandas with Accessors}
Create your own custom namespaces on Pandas objects,
    similar to ‘.str‘ or ‘.dt‘.
\begin{lstlisting}
from pandas.api.extensions import
    register_dataframe_accessor

@register_dataframe_accessor("geo")
class GeoAccessor:
    def __init__(self, pandas_obj):
        self._validate(pandas_obj)
        self._obj = pandas_obj

    @staticmethod
    def _validate(obj):
        if 'lat' not in obj.columns or 'lon' not in
    obj.columns:
            raise AttributeError("Must have 'lat'
    and 'lon'.")

    def plot(self):
        # Custom plotting logic using self._obj
        return self._obj.plot.scatter(x='lon', y='
    lat')

# Usage (assuming df has 'lat' and 'lon' columns)
# df.geo.plot()
```

6. Advanced String Manipulation

Use regular expressions with ‘.str.extract()‘ to pull structured data from strings.

```
s = pd.Series(['Name: Alice, Age: 25',
               'Name: Bob, Age: 30'])
pat = r'Name: (?P<Name>\w+), Age: (?P<Age>\d+)'

# .extract() returns a DataFrame with named groups
# as columns
extracted_df = s.str.extract(pat)
#   Name Age
# 0  Alice  25
# 1    Bob  30
```

Pandas Advanced Cheat Sheet

7. Out-of-Core with Dask

Use Dask to apply Pandas-like operations on datasets larger than RAM. Dask builds a task graph and executes it lazily.

```
import dask.dataframe as dd

# Dask mimics the Pandas API
# No data is loaded here (lazy operation)
ddf = dd.read_csv('massive_file.csv')

# Build a computation graph
result_graph = ddf.groupby('category')['value'].mean
    ()

# Trigger the computation
final_result = result_graph.compute()
```

8. Global Options & Settings

Configure Pandas behavior for your entire session.

```
# Set max rows to display
pd.set_option('display.max_rows', 100)

# Set float precision for display
pd.set_option('display.precision', 4)

# Stop truncating long strings in columns
pd.set_option('display.max_colwidth', None)

# Reset an option to its default
pd.reset_option('display.max_rows')

# Use a context manager for temporary settings
with pd.option_context('display.precision', 2):
    print(df.head())
```