Pandas Intermediate Cheat Sheet

1. GroupBy and Aggregation

The "split-apply-combine" pattern is a cornerstone of data analysis. Use `groupby()` to group data and perform aggregate calculations.

```
import pandas as pd
import numpy as np

data = {'Team': ['A', 'B', 'A', 'B', 'A'],
        'Player': ['P1','P2','P3','P4','P5'],
        'Points': [12, 15, 18, 10, 11],
        'Assists': [4, 8, 3, 9, 5]}
df = pd.DataFrame(data)

# Group by one column and get the sum
df.groupby('Team').sum()

# Group by 'Team' and get the mean of 'Points'
df.groupby('Team')['Points'].mean()

# Perform multiple aggregations with .agg()
df.groupby('Team').agg(
    total_points=('Points', 'sum'),
    avg_assists=('Assists', 'mean'),
    player_count=('Player', 'count')
)
```

2. Merging & Joining DataFrames

Combine multiple DataFrames using database-style joins.

'pd.merge()'

The primary function for joining.

```
df_left = pd.DataFrame({'key': ['K0', 'K1'], 'A': ['
    A0', 'A1']})
df_right = pd.DataFrame({'key': ['K0', 'K1'], 'B': [
    'B0', 'B1']})

# Inner join (default): keeps only common keys
pd.merge(df_left, df_right, on='key', how='inner')

# Left join: keeps all keys from the left frame
pd.merge(df_left, df_right, on='key', how='left')

# Joining on different column names
# pd.merge(df_l, df_r, left_on='lkey', right_on='
    rkey')
```

'pd.concat()'

Stack DataFrames vertically or horizontally.

```
pd.concat([df_left, df_right], axis=0) # Stack rows
pd.concat([df_left, df_right], axis=1) # Stack
    columns
```

3. Reshaping Data: Pivot & Melt

Transform data between "wide" and "long" formats.

'$\mathbf{pivot}_table()$'$(LongtoWide)$

Creates a spreadsheet-style pivot table.

```
data = {'Date': ['2023-01-01', '2023-01-01', '
    2023-01-02'],
        'Type': ['A', 'B', 'A'],
        'Value': [10, 20, 30]}
df_long = pd.DataFrame(data)

# Create a pivot table
df_long.pivot_table(values='Value',
                    index='Date',
                    columns='Type',
                    aggfunc='sum')
```

'melt()' (Wide to Long)

Unpivots a DataFrame from wide to long format.

```
df_wide = pd.DataFrame({'A': {0: 'a', 1: 'b'},
                        'B': {0: 1, 1: 3},
                        'C': {0: 2, 1: 4}})

# Melt the DataFrame
pd.melt(df_wide, id_vars=['A'], value_vars=['B', 'C'
    ])
```

4. Applying Custom Functions

Apply a function to your data.

'.apply()' (on Rows/Columns)

```
df = pd.DataFrame([[4, 9], [1, 5]], columns=['A', 'B
    '])

# Apply a function to each column (axis=0)
df.apply(np.sqrt)

# Apply a function to each row (axis=1)
df.apply(np.sum, axis=1)
```

'.map()' (on a Series)

Used for element-wise transformation of a Series, often with a dictionary or function.

```
s = pd.Series(['cat', 'dog', 'rabbit'])
s.map({'cat': 'kitty', 'dog': 'puppy'})
```

'.applymap()' (on a DataFrame)

Apply a function element-wise to the entire DataFrame.

```
df.applymap(lambda x: f'{x:.2f}')
```

5. Time Series Analysis

Pandas has powerful tools for working with dates and times.

Creating a DatetimeIndex

```
# Create a date range
dates = pd.date_range('20230101', periods=6, freq='D
    ')
ts_df = pd.DataFrame(np.random.randn(6, 1),
                     index=dates, columns=['Value'])

# Convert a column to datetime objects
# pd.to_datetime(df['date_column'])
```

Pandas Intermediate Cheat Sheet

Resampling

Change the frequency of your time series data (e.g., from daily to monthly).

```
# Downsample to monthly, taking the sum
ts_df.resample('M').sum()

# Upsample to 12-hour frequency, forward-filling
ts_df.resample('12H').ffill()
```

Rolling Windows

Calculate moving statistics.

```
# Calculate a 3-day rolling mean
ts_df.rolling(window=3).mean()
```

6. Multi-Indexing (Hierarchical)

An index with multiple levels, allowing you to store and manipulate higher-dimensional data in a 2D DataFrame.

Creating a MultiIndex

```
arrays = [['A', 'A', 'B', 'B'], [1, 2, 1, 2]]
index = pd.MultiIndex.from_arrays(
    arrays, names=('Class', 'ID'))
df_multi = pd.DataFrame(np.random.randn(4, 2),
                        index=index,
                        columns=['Val1', 'Val2'])
```

Selecting from a MultiIndex

Use `.loc` with tuples to select data.

```
# Select all data for Class 'A'
df_multi.loc['A']

# Select a specific row
df_multi.loc[('A', 1)]

# Slicing with pd.IndexSlice
idx = pd.IndexSlice
df_multi.loc[idx[:, 1], :] # Select all rows where
    ID is 1
```

7. Categorical Data Type

A memory-efficient data type for columns with a limited number of unique values.

```
df = pd.DataFrame({'grade': ['A','B','A','C','B','A'
    ]})
df['grade_cat'] = df['grade'].astype('category')

# .cat accessor provides category-specific methods
df['grade_cat'].cat.categories # ['A', 'B', 'C']
df['grade_cat'].cat.codes       # [0, 1, 0, 2, 1, 0]

# Benefits: uses less memory, can improve
# performance in some operations (e.g., groupby).
```

8. Method Chaining

Write cleaner, more readable code by chaining operations together instead of creating intermediate variables. Wrap chains in parentheses '()'.

```
# Standard way
df1 = df[df['Team'] == 'A']
df2 = df1.sort_values('Points')
result = df2['Player']

# Chained way (more readable)
result_chained = (df
    [df['Team'] == 'A']
    .sort_values('Points')
    ['Player']
)
```