# PySpark

Quick python refresher (this is where you are coming from )

Install PySpark/Findspark

Find Pyspark

Import pyspark

Import find spark and configure it

1-Broadcast

2-Element by element operation, element-wise operation

3-Filtering in PySpark

4-modifying elements

Select \* from employees where salary > 100,000 where \* id = manager

Join

# Pandas:

# Introduction to pandas Series vs DataFrames

**1. Series: The 1D Powerhouse**

python

Copy

Download

import pandas as pd

# Create Series from list

s = pd.Series([10, 20, 30, 40], name='ages')

print(s)

**Output**:

text

Copy

Download

0 10

1 20

2 30

3 40

Name: ages, dtype: int64

**Key Operations**:

python

Copy

Download

# Vectorized math

s \* 2

# Filter

s[s > 20]

# Null handling

s\_with\_nulls = pd.Series([1, None, 3])

s\_with\_nulls.fillna(0)

**2. DataFrame: The 2D Workhorse**

python

Copy

Download

# Create DataFrame from dict

data = {'name': ['Alice', 'Bob'], 'age': [25, 30]}

df = pd.DataFrame(data)

print(df)

**Output**:

text

Copy

Download

name age

0 Alice 25

1 Bob 30

**Key Operations**:

python

Copy

Download

# Add column

df['score'] = [90, 85]

# Filter rows

df[df['age'] > 25]

# Rename columns

df.rename(columns={'age': 'years'}, inplace=True)

**3. Series vs. DataFrame: Key Differences**

python

Copy

Download

# Series = Single column

ages = df['age'] # Returns Series

# DataFrame = Multiple Series

type(ages) # pandas.core.series.Series

type(df) # pandas.core.frame.DataFrame

# Convert Series to DataFrame

ages.to\_frame()

**4. Pro Move: Memory Efficiency**

python

Copy

Download

# Check memory usage

df.info(memory\_usage='deep')

# Optimize dtypes

df['age'] = df['age'].astype('int8')

**Rule of Thumb**:

* Use **Series** for single-column ops.
* Use **DataFrame** for tables.

# Pandas Data Loading & Inspection (CSV, Json)

Lesson:

**Lesson: Pandas Data Loading & Inspection**

**1. Load CSV Files**

python

Copy

Download

import pandas as pd

# Basic CSV load

df = pd.read\_csv('data.csv')

# Custom options

df = pd.read\_csv(

'data.csv',

sep=',', # Custom delimiter

header=0, # Row to use as column names

skiprows=2, # Skip first 2 rows

na\_values=['NA'] # Treat 'NA' as NaN

)

**2. Load JSON Files**

python

Copy

Download

# Standard JSON load

df\_json = pd.read\_json('data.json')

# Nested JSON (flatten)

df\_nested = pd.json\_normalize(

data,

record\_path='records',

meta=['id', 'date']

)

**3. Quick Inspection**

python

Copy

Download

# First 5 rows

df.head()

# Last 3 rows

df.tail(3)

# Structure info

df.info()

# Stats summary (numeric cols)

df.describe()

**4. Advanced Inspection**

python

Copy

Download

# Count missing values

df.isna().sum()

# Unique values in a column

df['column'].unique()

# Value counts

df['column'].value\_counts()

**5. Save Data**

python

Copy

Download

# Save to CSV

df.to\_csv('output.csv', index=False)

# Save to JSON

df.to\_json('output.json', orient='records')

**Key Takeaways**

* **CSV**: Use read\_csv() with sep, skiprows, na\_values.
* **JSON**: Use read\_json() or json\_normalize() for nested data.
* **Inspect**: head(), tail(), info(), describe().
* **Save**: to\_csv(), to\_json().

# Pandas Data Selection and Filtering

Lesson:

**Pandas Data Selection & Filtering (Beginner to Pro)**

**1. Basic Selection**

**Select Columns**

# As Series

df['column\_name']

# As DataFrame (double brackets)

df[['column\_name']]

# Multiple columns

df[['col1', 'col2', 'col3']]

**Select Rows (by index)**

# By integer position (iloc)

df.iloc[0] # First row

df.iloc[0:5] # Rows 0 to 4

df.iloc[[1, 3, 5]] # Specific rows

# By label (loc)

df.loc['row\_label']

df.loc[['row1', 'row3']]

**2. Boolean Filtering**

**Basic Conditions**

# Single condition

df[df['column'] > 50]

# Multiple conditions (AND: &, OR: |, NOT: ~)

df[(df['col1'] > 50) & (df['col2'] == 'value')]

# Using query()

df.query("col1 > 50 and col2 == 'value'")

**isin() for Multiple Values**

df[df['column'].isin(['val1', 'val2', 'val3'])]

**str.contains() for Text**

df[df['text\_column'].str.contains('substring', case=False)]

**3. Advanced Selection**

**Combining loc & Conditions**

# Select specific columns after filtering

df.loc[df['col1'] > 50, ['col2', 'col3']]

# Modify filtered data

df.loc[df['col1'] > 50, 'col2'] = 'new\_value'

**where() vs mask()**

# Keep values where condition is True, else NaN

df.where(df['col'] > 50)

# Opposite of where() (mask where condition is True)

df.mask(df['col'] > 50)

**select\_dtypes()**

# Select numeric columns

df.select\_dtypes(include=['int64', 'float64'])

# Exclude strings

df.select\_dtypes(exclude=['object'])

**4. Pro-Level Techniques**

**Filter with query() + Variables**

threshold = 50

df.query("col1 > @threshold")

**nlargest() / nsmallest()**

df.nlargest(5, 'column') # Top 5 values

df.nsmallest(3, 'column') # Bottom 3 values

**Filter with between()**

python

Copy

Download

df[df['col'].between(10, 20)]

**at & iat (Fast Scalar Access)**

df.at['row\_label', 'column'] # Label-based

df.iat[0, 1] # Position-based (row 0, col 1)

**5. Performance Tips**

**Avoid Chained Indexing**

# ❌ Bad (may cause SettingWithCopyWarning)

df[df['col1'] > 50]['col2'] = 'new\_value'

# ✅ Good (use loc)

df.loc[df['col1'] > 50, 'col2'] = 'new\_value'

**Use isin() Instead of Multiple ORs**

python

Copy

Download

# ❌ Slow

df[(df['col'] == 'A') | (df['col'] == 'B') | (df['col'] == 'C')]

# ✅ Faster

df[df['col'].isin(['A', 'B', 'C'])]

**Summary**

* **Basic:** [], loc, iloc
* **Filtering:** Boolean conditions, isin(), query()
* **Pro:** where(), mask(), at, iat, nlargest()
* **Performance:** Avoid chained indexing, prefer isin()

# Pandas Data Cleaning & Transformation

Lesson:

**Pandas Data Cleaning & Transformation (Beginner to Pro)**

**1. Handling Missing Data**

**Detect Missing Values**

df.isna() # Boolean mask of missing values

df.isna().sum() # Count missing per column

**Drop Missing Data**

df.dropna() # Drop rows with ANY NaN

df.dropna(axis=1) # Drop columns with ANY NaN

df.dropna(subset=['col1', 'col2']) # Drop rows with NaN in specific cols

**Fill Missing Data**

df.fillna(0) # Fill with scalar

df.fillna({'col1': 0, 'col2': 'missing'}) # Column-specific

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

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

df.interpolate() # Linear interpolation

**2. Removing Duplicates**

df.drop\_duplicates() # Drop all exact duplicates

df.drop\_duplicates(subset=['col1']) # Check specific columns

df.drop\_duplicates(keep='last') # Keep last occurrence

**3. Type Conversion & Cleaning**

**Convert Dtypes**

df['col'] = df['col'].astype('int32') # Change dtype

df['col'] = pd.to\_numeric(df['col'], errors='coerce') # Force numeric, invalid → NaN

df['col'] = pd.to\_datetime(df['col']) # Convert to datetime

**String Cleaning**

df['col'] = df['col'].str.strip() # Trim whitespace

df['col'] = df['col'].str.lower() # Lowercase

df['col'] = df['col'].str.replace('old', 'new') # Replace text

df['col'] = df['col'].str.extract(r'(\d+)') # Extract numbers

**4. Column Operations**

**Rename Columns**

df.rename(columns={'old\_name': 'new\_name'}, inplace=True)

df.columns = ['col1', 'col2', 'col3'] # Bulk rename

df.columns = df.columns.str.upper() # Uppercase all

**Add/Delete Columns**

df['new\_col'] = df['col1'] + df['col2'] # New column

df.insert(1, 'inserted\_col', values) # Insert at position

df.drop('col\_to\_drop', axis=1, inplace=True) # Delete column

**5. Advanced Transformations**

**Apply Custom Functions**

# Apply to entire column

df['col'] = df['col'].apply(lambda x: x \* 2)

# Apply row-wise

df['new\_col'] = df.apply(lambda row: row['col1'] + row['col2'], axis=1)

# Vectorized operations (faster)

df['col'] = np.log(df['col']) # Using NumPy

**Binning & Discretization**

# Numeric → Categories

bins = [0, 18, 35, 60, 100]

labels = ['child', 'young', 'adult', 'senior']

df['age\_group'] = pd.cut(df['age'], bins=bins, labels=labels)

**One-Hot Encoding (Dummy Variables)**

pd.get\_dummies(df['category\_col']) # Basic

pd.get\_dummies(df, columns=['col1', 'col2']) # Multiple columns

**6. Pro-Level Cleaning**

**Conditional Replacement**

df.loc[df['col'] > 100, 'col'] = 100 # Cap values

df['col'] = np.where(df['col'] < 0, 0, df['col']) # Replace negatives

**Group-wise Imputation**

# Fill NaN with group mean

df['col'] = df.groupby('group\_col')['col'].transform(lambda x: x.fillna(x.mean()))

**Outlier Handling**

# Remove rows where 'col' is beyond 3 standard deviations

mean = df['col'].mean()

std = df['col'].std()

df = df[(df['col'] > mean - 3\*std) & (df['col'] < mean + 3\*std)]

**7. Performance Tips**

**Use**inplace=True**to Save Memory**

df.dropna(inplace=True) # Instead of df = df.dropna()

**Optimize Dtypes**

df = df.astype({

'col1': 'int32',

'col2': 'category' # Saves memory for strings

})

**Summary**

* **Missing Data:** isna(), dropna(), fillna()
* **Duplicates:** drop\_duplicates()
* **Type Conversion:** astype(), to\_numeric(), to\_datetime()
* **String Cleaning:** str.strip(), str.replace()
* **Column Ops:** rename(), drop(), insert()
* **Advanced:** apply(), pd.cut(), get\_dummies()
* **Pro Tips:** Group imputation, outlier handling, vectorization

# 10-05 Pandas Data Joining and Merging

Lesson:

**Pandas Data Joining & Merging (Beginner to Pro)**

**1. Basic Concatenation**

**Vertical Stacking (Same Columns)**

pd.concat([df1, df2]) # Stack vertically

pd.concat([df1, df2], ignore\_index=True) # Reset index

**Horizontal Stacking (Same Index)**

pd.concat([df1, df2], axis=1) # Side-by-side

**2. Merging (SQL-Style Joins)**

**Basic Inner Join**

pd.merge(left\_df, right\_df, on='key') # Single key

pd.merge(left\_df, right\_df, on=['key1', 'key2']) # Multiple keys

**Join Types**

pd.merge(left\_df, right\_df, how='left') # Keep all left

pd.merge(left\_df, right\_df, how='right') # Keep all right

pd.merge(left\_df, right\_df, how='outer') # Keep all (union)

pd.merge(left\_df, right\_df, how='inner') # Keep only matches

**Different Column Names**

pd.merge(left\_df, right\_df, left\_on='key1', right\_on='key2')

**3. Joining (Index-Based Merging)**

df1.join(df2) # Left join on index

df1.join(df2, how='inner') # Inner join on index

df1.join(df2, on='key\_col') # Join on column (left) and index (right)

**4. Advanced Merging Techniques**

**Indicator Column (Track Merge Source)**

pd.merge(left\_df, right\_df, indicator=True) # Adds '\_merge' column

**Suffixes for Overlapping Columns**

pd.merge(left\_df, right\_df, on='key', suffixes=('\_left', '\_right'))

**Merge with Conditions (Non-Key Joins)**

pd.merge(

left\_df,

right\_df,

how='cross' # Cartesian product (all combinations)

)

**5. Pro-Level Merging**

**Merge on Binned Columns**

# Create matching bins

left\_df['bin'] = pd.cut(left\_df['value'], bins=[0, 50, 100])

right\_df['bin'] = pd.cut(right\_df['value'], bins=[0, 50, 100])

pd.merge(left\_df, right\_df, on='bin')

**Fuzzy Merging (Approximate Matching)**

from fuzzywuzzy import fuzz

# Add best match column

left\_df['best\_match'] = left\_df['name'].apply(

lambda x: max(right\_df['name'], key=lambda y: fuzz.ratio(x, y))

)

pd.merge(left\_df, right\_df, left\_on='best\_match', right\_on='name')

**Memory-Efficient Merging**

# Reduce dtype before merge

right\_df['key'] = right\_df['key'].astype('int32')

pd.merge(left\_df, right\_df, on='key')

**6. Performance Tips**

**Use**merge()**Over**join()**for Complex Logic**

# Pandas Time Series Operations

Lesson:

**Pandas Time Series Operations (Beginner to Pro)**

**1. Basic Time Handling**

**Convert to Datetime**

python

Copy

Download

df['date'] = pd.to\_datetime(df['date\_string']) # Auto-convert

df['date'] = pd.to\_datetime(df['date\_string'], format='%Y-%m-%d') # Force format

**Set as Index**

python

Copy

Download

df = df.set\_index('date') # Make datetime the index

df.index = pd.to\_datetime(df.index) # Convert existing index

**Basic Filtering**

python

Copy

Download

df.loc['2023'] # All of 2023

df.loc['2023-05'] # May 2023

df.loc['2023-05-01':'2023-05-15'] # Date range

**2. Time-Based Selection**

**Partial String Indexing**

python

Copy

Download

df.loc['2023-Q2'] # 2023 Q2

df.loc['2023-05'] # May 2023

df.loc['2023-05-15'] # Specific day

**Between Time**

python

Copy

Download

df.between\_time('09:00', '17:00') # Filter by time of day

**First/Last N Entries**

python

Copy

Download

df.first('2W') # First 2 weeks

df.last('5D') # Last 5 days

**3. Resampling & Aggregation**

**Downsampling (Daily → Monthly)**

python

Copy

Download

df.resample('M').mean() # Monthly means

df.resample('Q').sum() # Quarterly sums

df.resample('W').agg({'sales': 'sum', 'price': 'mean'}) # Custom

**Upsampling (Daily → Hourly)**

python

Copy

Download

df.resample('H').asfreq() # Insert NaNs

df.resample('H').ffill() # Forward fill

df.resample('H').interpolate() # Linear interpolation

**Rolling Windows**

python

Copy

Download

df.rolling(window=7).mean() # 7-day moving average

df.rolling(window=30, min\_periods=10).std() # 30-day std (min 10 obs)

**4. Shifting & Differencing**

**Time Shifts**

python

Copy

Download

df.shift(1) # Shift forward 1 period

df.shift(-1) # Shift backward 1 period

df.tshift(7, freq='D') # Shift index (not data) by 7 days

**Differencing (Change Over Time)**

python

Copy

Download

df.diff() # 1-period difference

df.diff(7) # 7-day difference

df.pct\_change() # Percentage change

**5. Time Zone Handling**

**Localize & Convert**

python

Copy

Download

df.tz\_localize('UTC') # Set timezone

df.tz\_convert('US/Eastern') # Convert timezone

**Working with Timestamps**

python

Copy

Download

pd.Timestamp.now() # Current time

pd.Timestamp('2023-05-15', tz='UTC') # Timezone-aware

pd.Timestamp(2023, 5, 15) # From components

**6. Advanced Time Series**

**Custom Business Days**

python

Copy

Download

from pandas.tseries.offsets import BDay

df + BDay(3) # Add 3 business days

**Holiday Calendars**

python

Copy

Download

from pandas.tseries.holiday import USFederalHolidayCalendar

cal = USFederalHolidayCalendar()

holidays = cal.holidays(start='2023-01-01', end='2023-12-31')

**Periods (Fixed-Frequency)**

python

Copy

Download

pd.Period('2023-05', freq='M') # Represents May 2023

df.to\_period('M') # Convert index to periods

**7. Pro-Level Techniques**

**Group by Time Components**

python

Copy

Download

df.groupby(df.index.year).sum() # Yearly sums

df.groupby([df.index.year, df.index.month]).mean() # Monthly means

**Time-Based Joins**

python

Copy

Download

pd.merge\_asof(df1, df2, on='time', direction='nearest') # Nearest match

**Rolling Apply (Custom Functions)**

python

Copy

Download

df.rolling(5).apply(lambda x: x.max() - x.min()) # Rolling range

**Handling Missing Time Points**

python

Copy

Download

full\_range = pd.date\_range(start=df.index.min(), end=df.index.max(), freq='D')

df.reindex(full\_range) # Insert missing dates

**Performance Tips**

1. **Use**pd.to\_datetime()**once** instead of repeatedly parsing dates
2. **Set datetime as index** for faster slicing
3. **Prefer**resample()**over**groupby() for time-based operations
4. **Use**offsets (BDay, MonthEnd) instead of custom logic
5. **Downsample early** when working with high-frequency data