Day 4 - Session 2:

Pandas - Deep Dive

This notebook provides a beginner-friendly deep dive into Pandas, an essential library for working with structured (tabular) data in Python. Pandas builds on NumPy and provides powerful, flexible, and easy-to-use data structures called DataFrames and Series. These are indispensable for tasks like telemetry analysis, experimental data processing, and preparing data for machine learning. We will cover every fundamental concept with clear explanations and simple, focused examples suitable for anyone learning Pandas for the first time.

1. Introduction to Pandas: DataFrames and Series

Imagine your data is organized like a spreadsheet with rows and columns. That's exactly what Pandas helps you work with!

- **Series**: Think of a `Series` as a single column from a spreadsheet. It's a list of values, but each value also has a label (called an "index").
- **DataFrame**: Think of a `DataFrame` as an entire spreadsheet or a table. It's a collection of columns, where each column is a `Series`. DataFrames are the most common way to work with tabular data in Pandas.

Why Use Pandas?

- **Easy Tables**: It makes working with data in rows and columns very simple.
- **Labeled Data**: You can use names (like column headers) instead of just numbers to find your data.
- **Missing Data**: It has built-in tools to easily find and fix missing information.
- **Powerful Tools**: It offers quick ways to filter, sort, group, and combine data.

Conceptual Approach:

We'll start by importing pandas (we usually shorten it to pd) and numpy (shortened to np) because Pandas uses NumPy behind the scenes.

```python
import pandas as pd
import numpy as np

print("--- Introduction to Pandas ---")

# 🔰 Motivation:
# Let's say we receive sensor data like:
# - Sensor A = 25.5°C (temperature)
# - Sensor B = 101.2 kPa (pressure)
# For multiple readings, managing this manually becomes messy.
# Pandas helps organize this kind of tabular data efficiently.

print("Pandas helps organize data like a table (DataFrame) or a single column (Series).")

# ➤ Think of a DataFrame like an Excel table (rows + columns).
# ➤ Think of a Series like a single column from that table.

# We'll now explore:
# - Series: For 1D labeled data
# - DataFrame: For 2D tabular data
```

2. Pandas Series: One-Dimensional Labeled Arrays

A `Series` is a single column of data. Each item in the column has a value and a label (index).

2.1 Series Creation

You can create a `Series` from simple Python lists, NumPy arrays, or dictionaries.

`pd.Series()`: How to Make a Series

- **Purpose**: Creates a one-dimensional array-like object with an index.

- **Syntax**: pd.Series(data, index=None, dtype=None, name=None)
    - **data**: The values for your Series (e.g., a list or NumPy array).
    - **index**: (Optional) The labels for each value. If not given, Pandas uses numbers (0, 1, 2...).
    - **dtype**: (Optional) The type of data (e.g., **int**, **float**, **str**).
    - **name**: (Optional) A name for your Series.

```python
print("--- Pandas Series: Basic Creation Examples ---")

import pandas as pd

# ✅ Example 1: Create a Series from a Python list (default index)
numbers = [10, 20, 30, 40]
series1 = pd.Series(numbers)
print("Series from a list (default index):")
print(series1)
print()

# ✅ Example 2: Create a Series with custom labels (index)
marks = [80, 85, 90]
subjects = ['Math', 'Science', 'English']
series2 = pd.Series(marks, index=subjects)
print("Series with custom index (subject names):")
print(series2)
print()

# ✅ Example 3: Create a Series from a dictionary (keys become index)
fruit_prices = {'Apple': 30, 'Banana': 10, 'Mango': 50}
series3 = pd.Series(fruit_prices)
print("Series from a dictionary:")
print(series3)
print()

# ✅ Example 4: Series with a name (label the data)
temperatures = pd.Series([25.0, 26.5, 27.8], name="Room Temperatures")
print("Series with a name:")
print(temperatures)
```

2.2 Series Attributes (Information about your Series)

These are like properties that tell you about your Series, such as its labels, values, or type.

Common Series Attributes

Attribute	What it tells you	Example (for a Series s)
`.index`	The labels (names) for each item.	`s.index`
`.values`	Just the data values (as a NumPy array).	`s.values`
`.dtype`	The type of data stored (e.g., int64, float64).	`s.dtype`
`.name`	The name you gave the Series.	`s.name`
`.shape`	The size of the Series (e.g., (4,) for 4 items).	`s.shape`
`.size`	The total number of items.	`s.size`

Python

```python
print("\n--- Pandas Series: Basic Attributes ---")

import pandas as pd

# Create a simple Series with sensor readings
sensor_readings = pd.Series(
    [10.5, 11.2, 9.8],
    index=['SensorA', 'SensorB', 'SensorC'],
    name='Current_Readings'
)

# Show the Series
print("Sensor Readings Series:")
print(sensor_readings)
```

```
print()

# Check useful attributes
print("📌 Index Labels:", sensor_readings.index.tolist())
print("📌 Values Only:", sensor_readings.values)
print("📌 Data Type:", sensor_readings.dtype)
print("📌 Series Name:", sensor_readings.name)
print("📌 Shape (rows,):", sensor_readings.shape)
print("📌 Total Number of Items:", sensor_readings.size)
```

2.3 Series Indexing and Slicing (Picking Parts of your Series)

You can pick specific items from a Series using their label or their position (like in a list).

Conceptual Approach:

- Use `[ ]` with either a label or a number (position).
- Slicing works like Python lists: `[start:end]` for numbers, but `[start_label:end_label]` for labels (where end_label is included!).

Python

```
print("\n--- Pandas Series: Indexing and Slicing (Simple Examples) ---")

import pandas as pd

# Create a Series with fruit prices
fruit_prices = pd.Series(
    [30, 40, 25, 50, 35],
    index=['Apple', 'Banana', 'Orange', 'Mango', 'Grapes']
)

print("Fruit Prices:")
print(fruit_prices)
print()

# Accessing by label
print("Price of Orange:", fruit_prices['Orange'])
```

```python
# Accessing by position
print("First fruit's price:", fruit_prices[0])

# Slicing by position
print("Prices of 2nd and 3rd fruits (positions 1 to 2):")
print(fruit_prices[1:3])  # Banana, Orange

# Slicing by labels
print("Prices from Banana to Mango:")
print(fruit_prices['Banana':'Mango'])  # Includes Mango

# Selecting multiple fruits by label
print("Prices of Apple and Grapes:")
print(fruit_prices[['Apple', 'Grapes']])

# Boolean Indexing
print("Fruits priced above ₹30:")
print(fruit_prices[fruit_prices > 30])
```

3. Pandas DataFrame: Two-Dimensional Labeled Data Structures

A `DataFrame` is like a whole spreadsheet: it has rows and columns, both with labels (names).

3.1 DataFrame Creation (Making your Spreadsheet)

DataFrames are usually made from dictionaries where keys are column names and values are lists/Series for each column. Or from NumPy arrays.

`pd.DataFrame()`: How to Make a DataFrame

- **Purpose**: Creates a two-dimensional tabular data structure with labeled rows and columns.

- Syntax: `pd.DataFrame(data, index=None, columns=None, dtype=None)`
    - **data**: The data for your DataFrame. Common options:
        - Dictionary of lists/NumPy arrays (keys are column names).
        - List of dictionaries (each dictionary is a row).
    - **index**: (Optional) Labels for the rows.
    - **columns**: (Optional) Names for the columns.

Python

```python
import pandas as pd
import numpy as np

print("--- Pandas DataFrame: Easy Examples ---")

# 🌟 Example 1: Create DataFrame from a dictionary of lists
# Each key becomes a column, and each list holds the values
student_data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Maths_Marks': [85, 90, 78],
    'Science_Marks': [88, 82, 91]
}

df_students = pd.DataFrame(student_data)
print("📊 Students DataFrame (from dictionary of lists):")
print(df_students)
print()

# 🌟 Example 2: Create DataFrame from a list of dictionaries
# Each dictionary represents one row of data
product_data = [
    {'Product': 'Pen', 'Price': 10},
    {'Product': 'Notebook', 'Price': 30},
    {'Product': 'Pencil', 'Price': 5}
]

df_products = pd.DataFrame(product_data)
print("🛒 Products DataFrame (from list of dictionaries):")
print(df_products)
print()
```

```python
# 🌟 Example 3: Create DataFrame from a NumPy array
# Rows are data, columns are labeled
array_data = np.array([
    [100, 200],
    [300, 400]
])

df_sales = pd.DataFrame(array_data, columns=['Online_Sales', 'Offline_Sales'],
index=['Jan', 'Feb'])
print("💰 Sales DataFrame (from NumPy array):")
print(df_sales)
```

3.2 DataFrame Inspection (Getting to Know your Data)

After making a DataFrame, you'll want to quickly see what's inside it and its overall structure.

Common DataFrame Inspection Methods/Attributes

Method/Attribute	What it tells you	Example (for a DataFrame df)
`.head(n)`	Shows the first n rows (default is 5).	`df.head()`
`.tail(n)`	Shows the last n rows (default is 5).	`df.tail(2)`
`.info()`	Gives a summary: column names, how many non-empty values, data types, memory usage.	`df.info()`
`.describe()`	Gives basic math info (count, average, min, max) for number columns.	`df.describe()`
`.shape`	The size of the table as (rows, columns).	`df.shape`

`.dtypes`	The data type for each column.	`df.dtypes`
`.index`	The labels for the rows.	`df.index`
`.columns`	The names of the columns.	`df.columns`
`.values`	Just the data values (as a NumPy array).	`df.values`
`.isnull().sum()`	Counts how many missing values are in each column.	`df.isnull().sum()`

Python

```python
import pandas as pd
import numpy as np

print("\n--- Easy DataFrame Inspection Example ---")

# 🥝 Simple fruit data
fruit_data = {
    'Fruit': ['Apple', 'Banana', 'Mango', 'Orange'],
    'Quantity': [10, 15, 8, 12],
    'Price': [30, 10, 50, 25]
}

df_fruits = pd.DataFrame(fruit_data)
print("Our Fruit DataFrame:\n", df_fruits, "\n")

# 👀 Show first 2 rows
print("First 2 rows:\n", df_fruits.head(2), "\n")

# 👀 Show last 1 row
print("Last row:\n", df_fruits.tail(1), "\n")

# 📋 Quick info about DataFrame
print("Info about DataFrame:")
df_fruits.info()
print()

# 📊 Describe only number columns
print("Summary Statistics:\n", df_fruits.describe(), "\n")
```

```python
# 📐 Shape (rows, columns)
print("Shape of DataFrame:", df_fruits.shape)

# 📚 Data types of each column
print("Data Types:\n", df_fruits.dtypes, "\n")

# 🔤 Column names
print("Column Names:", df_fruits.columns.tolist())

# 🔢 Row labels (index)
print("Row Labels (Index):", df_fruits.index, "\n")

# 🌧️ Add some missing values
df_with_missing = df_fruits.copy()
df_with_missing.loc[1, 'Quantity'] = np.nan
df_with_missing.loc[3, 'Price'] = np.nan
print("DataFrame with missing values:\n", df_with_missing, "\n")

# ❓ Check how many missing values in each column
print("Missing values count:\n", df_with_missing.isnull().sum())
```

3.3 Data Selection (Picking Specific Data)

This is how you get specific rows, columns, or cells from your DataFrame.

Conceptual Approach:

- To get a **column**, use `df['ColumnName']`.
- To get **rows by label** (their names), use `.loc[]`.
- To get **rows by position** (their row number starting from 0), use `.iloc[]`.
- To get **rows based on a condition** (like "Temperature > 25"), use boolean indexing `df[condition]`.

Common Data Selection Methods

Python

```python
import pandas as pd

# 🍎 Sample data: fruit shop
data = {
    'Fruit': ['Apple', 'Banana', 'Mango', 'Orange'],
    'Quantity': [10, 15, 8, 12],
    'Price': [30, 10, 50, 25],
    'Available': ['Yes', 'Yes', 'No', 'Yes']
}

df = pd.DataFrame(data)
print("--- Fruit Shop Data ---")
print(df)
```

Method	What it does	Example
`df['ColName']`	Select one column (returns Series)	`df['Price']`
`df[['Col1', 'Col2']]`	Select multiple columns (returns DataFrame)	`df[['Fruit', 'Price']]`
`df.loc[row, col]`	Access by label (row name/index, column name)	`df.loc[1, 'Fruit']`
`df.iloc[row, col]`	Access by position (row number, column number)	`df.iloc`
`df[condition]`	Filter rows based on condition	`df[df['Price'] > 25]`

Python

```python
print("\n--- Easiest DataFrame Selection Examples ---")
```

```python
# 1. Select one column
print("Fruit column:\n", df['Fruit'], "\n")

# 2. Select multiple columns
print("Fruit and Price columns:\n", df[['Fruit', 'Price']], "\n")

# 3. Use .loc[row_label, column_name]
print("Fruit at row label 1:", df.loc[1, 'Fruit'])  # Banana

# 4. Use .iloc[row_pos, col_pos]
print("Fruit at row 2, column 0:", df.iloc[2, 0])  # Mango

# 5. Select multiple rows and columns using .loc[]
print("Rows 0-2, Fruit and Quantity:\n", df.loc[0:2, ['Fruit', 'Quantity']],
"\n")

# 6. Filter: Show rows where Price > 25
print("Fruits with price > 25:\n", df[df['Price'] > 25], "\n")

# 7. Filter: Show fruits available AND Quantity > 10
filtered_df = df[(df['Available'] == 'Yes') & (df['Quantity'] > 10)]
print("Available fruits with quantity > 10:\n", filtered_df, "\n")

# 8. Update a value using .loc
df_copy = df.copy()
df_copy.loc[2, 'Available'] = 'Yes'
print("After marking Mango as Available:\n", df_copy)
```

3.4 Data Manipulation: Adding/Modifying Columns (Changing your Spreadsheet)

You can easily add completely new columns or change data in existing ones.

Conceptual Approach:

- To add a new column, simply assign data to `df['NewColumnName']`.
- You can use existing columns in math operations to create new ones.

Python

```python
import pandas as pd

# 🍓 Simple fruit shop data
data = {
    'Fruit': ['Apple', 'Banana', 'Mango'],
    'Price': [30, 10, 50],
    'Quantity': [5, 8, 3]
}

df = pd.DataFrame(data)
print("--- Original Fruit Shop Data ---")
print(df, "\n")
```

Python

```python
# 1. Add a new column with the same value for all rows
df['Shop_Name'] = 'FreshFruitMart'
print("Added 'Shop_Name' column:\n", df, "\n")

# 2. Add a new column based on calculation
# Total cost = Price × Quantity
df['Total_Cost'] = df['Price'] * df['Quantity']
print("Added 'Total_Cost' column:\n", df, "\n")

# 3. Change values based on a condition
# If Price > 20, mark as 'Expensive', else 'Cheap'
df['Price_Tag'] = df['Price'].apply(lambda x: 'Expensive' if x > 20 else
'Cheap')
```

```
print("Added 'Price_Tag' column:\n", df, "\n")

# 4. Apply a function for custom logic
def stock_status(qty):
    if qty >= 5:
        return 'In Stock'
    else:
        return 'Low Stock'

df['Stock_Status'] = df['Quantity'].apply(stock_status)
print("Added 'Stock_Status' column:\n", df, "\n")
```

Fruit	Price	Quantity	Shop_Name	Total_Cost	Price_Tag	Stock_Status
Apple	30	5	Fres...	150	Expe...	In St...
Banana	10	8	Fres...	80	Cheap	In St...
Mango	50	3	Fres...	150	Expe...	Low ...

3.5 Handling Missing Data (NaN values)

Sometimes your data has holes (missing values), which Pandas shows as NaN (Not a Number). You need to decide how to handle them.

Conceptual Approach:

First, find where the NaN values are. Then, you can either:

- Drop rows or columns that have NaNs.
- Fill NaNs with a specific value (like 0) or by using a method (like filling with the value from the previous row).

Common Missing Data Methods

Method	What it does	Example (for a DataFrame df)
`.isnull()`	Shows `True` where data is missing, `False` otherwise.	`df.isnull()`
`.isnull().sum()`	Counts the number of missing values (NaNs) in each column.	`df.isnull().sum()`
`.dropna(axis=0/1)`	Removes rows (`axis=0`) or columns (`axis=1`) containing any missing values.	`df.dropna()` (drops rows) `df.dropna(axis=1)` (drops columns)
`.fillna(value)`	Fills missing values with a specified `value`.	`df.fillna(0)` (fills with 0)
`.fillna(method='ffill')`	Fills missing values with the value from the row directly above it (forward fill).	`df.fillna(method='ffill')`
`.fillna(method='bfill')`	Fills missing values with the value from the row directly below it (backward fill).	`df.fillna(method='bfill')`

Python

```python
import pandas as pd
import numpy as np

# 🎓 Simple student marks data (some values are missing)
data = {
    'Student': ['Alice', 'Bob', 'Charlie', 'David'],
    'Maths': [90, np.nan, 85, 78],
    'Science': [88, 92, np.nan, np.nan]
}

df = pd.DataFrame(data)
print("--- Original Data ---")
```

```
print(df)
```

Python

```
df = pd.DataFrame(data)
print("\nOriginal Data:")
print(df)

# Step 2: Check where missing values are
print("\nWhere values are missing (True means missing):")
print(df.isnull())

# Step 3: Count how many missing values per column
print("\nCount of missing values per column:")
print(df.isnull().sum())

# Option A: Drop rows with any missing values
df_dropna = df.dropna()
print("\nDrop rows with missing values:")
print(df_dropna)

# Option B: Fill missing values with 0
df_fill_zero = df.fillna(0)
print("\nFill missing values with 0:")
print(df_fill_zero)

# Option C: Fill missing values with column average (mean)
df_fill_mean = df.copy()
df_fill_mean['Maths']                                                    =
df_fill_mean['Maths'].fillna(df_fill_mean['Maths'].mean())
df_fill_mean['Science']                                                  =
df_fill_mean['Science'].fillna(df_fill_mean['Science'].mean())
print("\nFill missing values with column mean:")
print(df_fill_mean)

# Option D: Forward fill (copy value from above)
df_ffill = df.fillna(method='ffill')
print("\nForward fill (fill with previous value):")
print(df_ffill)
```

3.6 Data Cleaning and Transformation (Making your Data Tidy)

This involves making your data consistent and ready for analysis, like renaming columns, changing data types, or correcting values.

Common Cleaning/Transformation Methods

Method	What it does
`.rename(columns={...})`	Changes column names.
`.astype(dtype)`	Changes the data type of a column (e.g., str to float).
`.replace(old_value, new_value)`	Finds and replaces specific values.
`.map(dictionary)`	Replaces values in a column based on a dictionary.
`pd.to_datetime()`	Converts a column to proper date/time objects.

Python

```python
import pandas as pd
import numpy as np

print("\n--- Easy Example: Data Cleaning and Transformation ---")

# Step 1: Create a simple DataFrame with "dirty" data
data = {
    'id': ['S1', 'S2', 'S3', 'S4'],
    'score': ['85', '90', 'NA', '75'],  # 'NA' should be treated as missing
    'unit': ['marks', 'marks', 'marks', 'marks'],
    'status': ['ok', 'ok', 'warn', 'error']
}
df = pd.DataFrame(data)
print("\nOriginal Data:")
print(df)
```

```
# Step 2: Rename columns to be clearer
df = df.rename(columns={
    'id': 'Student_ID',
    'score': 'Exam_Score',
    'unit': 'Score_Unit',
    'status': 'Result_Status'
})
print("\nAfter renaming columns:")
print(df)

# Step 3: Convert 'Exam_Score' to numeric (turn 'NA' to actual missing value)
df['Exam_Score'] = df['Exam_Score'].replace('NA', np.nan)
df['Exam_Score'] = pd.to_numeric(df['Exam_Score'])
print("\nAfter converting 'Exam_Score' to numbers:")
print(df)
print("Data type of 'Exam_Score':", df['Exam_Score'].dtype)

# Step 4: Clean 'Result_Status' - make everything uppercase
df['Result_Status'] = df['Result_Status'].str.upper()
print("\nAfter cleaning 'Result_Status':")
print(df)
```

3.7 Grouping and Aggregation (Summarizing your Data)

This is a very powerful feature to group rows based on certain categories and then calculate summaries (like average, sum, count) for each group.

Conceptual Approach:

It's like using "Group By" in a spreadsheet or SQL. You tell Pandas: "Group these rows by Sensor_ID, then give me the average Temperature for each group."

Common GroupBy/Aggregation Methods

Method	What it Does	Example (from our code)
`df.groupby('Col')`	Groups data by unique values	`df.groupby('Student')`
`.mean()`	Average of each group	`df.groupby('Student')['Score'].mean()`
`.sum()`	Total sum per group	`df.groupby('Subject')['Score'].sum()`
`.count()`	Count of non-NA values per group	`df.groupby('Student')['Score'].count()`
`.min()`/`.max()`	Minimum/maximum per group	Used in `.agg()` below
`.agg({'Col': 'func'})`	Apply multiple functions and rename result	`agg(Max_Score=('Score', 'max'), ...)`

Python

```python
import pandas as pd

print("\n--- Easy GroupBy and Aggregation Example ---")

# Step 1: Create a simple DataFrame
df = pd.DataFrame({
    'Student': ['Alice', 'Bob', 'Alice', 'Bob', 'Charlie'],
    'Subject': ['Math', 'Math', 'Science', 'Science', 'Math'],
    'Score': [85, 90, 80, 88, 75]
})
print("Original Data:\n", df, "\n")

# Example 1: Average score per student
avg_score = df.groupby('Student')['Score'].mean()
print("Average Score per Student:\n", avg_score, "\n")

# Example 2: Total score per subject
```

```
total_per_subject = df.groupby('Subject')['Score'].sum()
print("Total Score per Subject:\n", total_per_subject, "\n")

# Example 3: Count of entries per student
count_scores = df.groupby('Student')['Score'].count()
print("Number of Scores per Student:\n", count_scores, "\n")

# Example 4: Multiple aggregation per student
summary = df.groupby('Student').agg(
    Max_Score=('Score', 'max'),
    Min_Score=('Score', 'min'),
    Total_Score=('Score', 'sum')
)
print("Summary per Student (Max, Min, Total):\n", summary, "\n")
```

3.8 Merging and Joining DataFrames (Combining Spreadsheets)

When your data is split across different tables (DataFrames), you often need to combine them. This is like doing a "lookup" or "join" in a database.

Conceptual Approach:

You join DataFrames based on a shared column (like a common "ID" column).

Common Merging Methods

Method	What it does	Analogy
`pd.merge(df1, df2, on='key_col')`	Inner Join (default): Keeps only rows where the `key_col` exists in BOTH DataFrames.	Find matching customer IDs in two lists.

`pd.merge(..., how='left')`	Left Join: Keeps ALL rows from the LEFT DataFrame, and adds matching rows from the RIGHT. If no match, fills with `NaN`.	Keep all customer orders, add addresses if available.
`pd.merge(..., how='right')`	Right Join: Keeps ALL rows from the RIGHT DataFrame, and adds matching rows from the LEFT. If no match, fills with `NaN`.	Keep all product inventory, add supplier details if available.
`pd.merge(..., how='outer')`	Outer Join: Keeps ALL rows from BOTH DataFrames. Fills `NaN` where there's no match in either.	Combine all customers and all products, show if they've ordered.
`pd.merge(..., left_on='key1', right_on='key2')`	Merge when the shared column has different names in each DataFrame.	Join `cust_id` from one table with `customer_id` from another.

Python

```python
import pandas as pd

print("\n--- Simple DataFrame Merging Example ---")

# Table 1: Student Marks
marks_df = pd.DataFrame({
    'Student': ['Alice', 'Bob', 'Charlie'],
    'Maths_Marks': [85, 90, 78]
})

# Table 2: Student Grades
grades_df = pd.DataFrame({
    'Student': ['Alice', 'Bob', 'David'],  # 'David' not in marks_df
    'Grade': ['A', 'A+', 'B']
})

print("Marks DataFrame:\n", marks_df, "\n")
print("Grades DataFrame:\n", grades_df, "\n")
```

```python
# Example 1: Inner Merge - only matching students in both tables
inner_merged = pd.merge(marks_df, grades_df, on='Student', how='inner')
print("Inner Merge (only common students):\n", inner_merged, "\n")

# Example 2: Left Merge - keep all from marks_df
left_merged = pd.merge(marks_df, grades_df, on='Student', how='left')
print("Left Merge (all students from marks_df):\n", left_merged, "\n")

# Example 3: Outer Merge - include all from both
outer_merged = pd.merge(marks_df, grades_df, on='Student', how='outer')
print("Outer Merge (all students from both):\n", outer_merged, "\n")
```

4. Time Series Handling (Working with Dates and Times)

Pandas is excellent for data that has timestamps, like sensor readings taken over time.

Conceptual Approach:

First, make sure your time column is a proper "datetime" type. Then, you can group or summarize data by specific time periods (like daily or hourly averages).

Common Time Series Methods

Method	What it does	Example
`pd.to_datetime()`	Converts strings to datetime objects	`pd.to_datetime(df['Date'])`
`.set_index('Date')`	Uses the date column as row index (helps with resampling)	`df.set_index('Date')`
`.resample('W').mean()`	Groups data by week (W) and calculates the weekly average	`df.resample('W').mean()`
`.index.day`	Extracts just the day from each datetime index	`df.index.day`

Python

```python
import pandas as pd

print("\n--- Time Series Handling (Easiest Example) ---")

# Step 1: Create a small dataset with date strings and temperature values
data = {
    'Date': ['2025-07-01', '2025-07-02', '2025-07-03', '2025-07-04'],
    'Temperature': [30.5, 32.0, 31.2, 29.8]
}
df = pd.DataFrame(data)
print("Original DataFrame:\n", df, "\n")

# Step 2: Convert the 'Date' column to datetime format
df['Date'] = pd.to_datetime(df['Date'])
print("After converting 'Date' to datetime:\n", df, "\n")

# Step 3: Set the 'Date' column as the index (helps with time-based operations)
df.set_index('Date', inplace=True)
print("DataFrame with 'Date' as index:\n", df, "\n")

# Step 4: Resample to Weekly Average (though here we only have 4 days)
weekly_avg = df.resample('W').mean()
print("Weekly Average Temperature:\n", weekly_avg, "\n")

# Step 5: Extract Day values using `.index.day`
print("Day of each reading:", df.index.day.tolist())
```

5. Categorical Data (Efficiently Storing Limited Choices)

When a column has a limited number of unique text values (like 'OK', 'WARNING', 'ERROR' statuses), Pandas can store this more efficiently using a "categorical" data type. This saves memory and speeds up some operations.

Conceptual Approach:

If your column has text data with repeated, limited choices, tell Pandas it's a "category" type.

Common Categorical Data Methods

Method	What it does	Example (for a Series s)
`.astype('category')`	Changes a Series to a Categorical data type.	`df['Status'].astype('category')`
`.cat.categories`	Shows all the unique categories in the Series.	`df['Status'].cat.categories`
`.cat.codes`	Shows the hidden number that Pandas uses for each category.	`df['Status'].cat.codes`
`.value_counts()`	Counts how many times each unique value appears.	`df['Status'].value_counts()`

Python

```python
import pandas as pd

print("\n--- Pandas Categorical Data Handling ---")

# Step 1: Create a simple DataFrame with text-based status values
df_cat = pd.DataFrame({
    'Device_Type': ['Engine', 'Cabin', 'Engine', 'Cabin', 'Engine'],
    'Status': ['OK', 'WARNING', 'OK', 'ERROR', 'OK'],
    'Fault_Code': [101, 201, 101, 301, 101]
})
print("Original DataFrame:\n", df_cat, "\n")
print("Data type of 'Status' before conversion:", df_cat['Status'].dtype, "\n")
```

```
# Step 2: Convert the 'Status' column to a categorical data type
df_cat['Status'] = df_cat['Status'].astype('category')
print("After converting 'Status' to categorical type:\n", df_cat, "\n")
print("New data type of 'Status':", df_cat['Status'].dtype, "\n")

# Step 3: List all the unique categories (labels) Pandas recognized
print("Unique Categories in 'Status':", df_cat['Status'].cat.categories, "\n")

# Step 4: View internal integer codes used for each category (under the hood)
print("Internal Category Codes for each row in 'Status':",
df_cat['Status'].cat.codes.tolist(), "\n")

# Step 5: Get count of each category — works efficiently with categorical types
print("Frequency count of each Status value:\n",
df_cat['Status'].value_counts(), "\n")
```

6. Practical Lab: Analyzing Simplified Telemetry Data

This lab puts everything you've learned into practice by analyzing a simple set of telemetry data.

Problem Statement:

You have received a small telemetry log file (simple_telemetry.csv). Your task is to:

1. Load the data into a Pandas DataFrame.
2. Clean the data:
    - Convert 'Timestamp' column to a proper date/time format.
    - Fill any missing temperature readings with the average temperature.
3. Analyze the data:
    - Calculate the **daily average** temperature.
    - Find the **highest** pressure recorded.
    - Count how many 'ERROR' status entries there are.
4. Generate a simple summary report.

Conceptual Approach:

1. **Create Dummy CSV File:** We'll first create a small simple_telemetry.csv file right here in the notebook to work with.
2. **Load Data:** Use pd.read_csv().
3. **Inspect:** Use head() and info().
4. **Clean:** Apply pd.to_datetime() and fillna() with .mean().
5. **Analyze:** Use resample() for daily average, max() for highest pressure, and boolean indexing with count() for errors.
6. **Summary:** Print the results clearly.

6.1 Code Implementation: Simple Telemetry Data Analysis Lab

Python

```python
import pandas as pd
import numpy as np
import os  # Used to remove the dummy CSV file at the end

print("\n--- Lab: Analyzing Simple Telemetry Data ---")

# ------------------------------------------------------
# Step 1: Create a dummy CSV file for practice
# ------------------------------------------------------

# Multiline string that represents CSV content
telemetry_csv_content =
"""Timestamp,Sensor_ID,Temperature_C,Pressure_kPa,Status
2025-07-01 10:00:00,TS-001,25.0,100.0,OK
2025-07-01 10:00:00,PS-001,26.0,98.5,OK
2025-07-01 10:30:00,TS-001,25.2,100.5,OK
2025-07-01 10:30:00,PS-001,27.0,np.nan,WARNING
2025-07-02 09:00:00,TS-001,28.0,101.0,OK
2025-07-02 09:00:00,PS-001,np.nan,102.5,ERROR
2025-07-02 09:30:00,TS-001,28.5,101.2,OK
2025-07-02 09:30:00,PS-001,29.0,103.0,OK
"""

# Save the content to a CSV file (replace 'np.nan' string with actual NaN)
```

```
with open("simple_telemetry.csv", "w") as f:
    f.write(telemetry_csv_content.replace("np.nan", str(np.nan)))

print("Dummy CSV file 'simple_telemetry.csv' created.\n")

# ------------------------------------------------------
# Step 2: Load the data into a Pandas DataFrame
# ------------------------------------------------------

try:
    df = pd.read_csv("simple_telemetry.csv")
    print("CSV successfully loaded!\n")
    print("First few rows of the DataFrame:\n", df.head(), "\n")
except FileNotFoundError:
    print("File not found. Ensure the CSV file exists.")
    exit()

# Show column info and data types
print("Data types and non-null counts (df.info()):")
df.info()
print("\n")

# ------------------------------------------------------
# Step 3: Clean the Data
# ------------------------------------------------------

print("--- Cleaning Data ---")

# Convert Timestamp column to datetime objects
df['Timestamp'] = pd.to_datetime(df['Timestamp'])
print("Converted 'Timestamp' to datetime.\n")

# Set Timestamp as index (for time-based analysis)
df.set_index('Timestamp', inplace=True)
print("Set 'Timestamp' as index.\n")

# Ensure temperature values are numeric, then fill missing with column mean
df['Temperature_C'] = pd.to_numeric(df['Temperature_C'], errors='coerce')
avg_temp = df['Temperature_C'].mean()
df['Temperature_C'].fillna(avg_temp, inplace=True)
print(f"Filled missing 'Temperature_C' with mean value: {avg_temp:.2f}\n")
```

```python
# Ensure pressure values are numeric, then fill missing with column mean
df['Pressure_kPa'] = pd.to_numeric(df['Pressure_kPa'], errors='coerce')
avg_pressure = df['Pressure_kPa'].mean()
df['Pressure_kPa'].fillna(avg_pressure, inplace=True)
print(f"Filled missing 'Pressure_kPa' with mean value: {avg_pressure:.2f}\n")

# Check if any missing values remain
print("Remaining missing values (should be 0):\n", df.isnull().sum(), "\n")

# ------------------------------------------------------
# Step 4: Analyze the Data
# ------------------------------------------------------

print("--- Analyzing Data ---")

# 1. Daily average temperature
daily_avg_temp = df.resample('D')['Temperature_C'].mean()
print("Daily Average Temperatures:\n", daily_avg_temp.round(2), "\n")

# 2. Highest pressure recorded
max_pressure = df['Pressure_kPa'].max()
print(f"Highest Pressure Recorded: {max_pressure:.2f} kPa\n")

# 3. Count how many times status was 'ERROR'
error_count = df[df['Status'] == 'ERROR'].shape[0]
print(f"Total 'ERROR' status entries: {error_count}\n")

# 4. Average pressure per sensor
avg_pressure_by_sensor = df.groupby('Sensor_ID')['Pressure_kPa'].mean()
print("Average Pressure by Sensor:\n", avg_pressure_by_sensor.round(2), "\n")

# ------------------------------------------------------
# Step 5: Summary Report
# ------------------------------------------------------

print("--- Summary Report ---")
print(f"Overall Average Temperature: {df['Temperature_C'].mean():.2f} °C")
print(f"Overall Maximum Pressure: {max_pressure:.2f} kPa")
print(f"Total Error Entries: {error_count}")
print("Daily Temperature Trends:\n", daily_avg_temp.round(2))

# ------------------------------------------------------
```

```
# Step 6: Clean Up (Remove Dummy File)
# ------------------------------------------------------

os.remove("simple_telemetry.csv")
print("\nDummy file 'simple_telemetry.csv' has been deleted.")
```

7. Key Takeaways

This session provided a hands-on introduction to Pandas, equipping you with the fundamental skills for working with structured data, which is essential for engineers and scientists.

- **Pandas Basics**: Understood the core Series (single column) and DataFrame (table) structures.
- **Creating & Inspecting DataFrames**: Learned how to build DataFrames and use methods like head(), info(), and describe() to quickly understand your data.
- **Picking Your Data**: Mastered how to select specific columns, rows, or cells using simple column names, .loc (by label/name), .iloc (by position/number), and powerful boolean conditions.
- **Changing Your Data**: Learned to add new columns, modify existing ones, and clean data by renaming columns, changing data types, and replacing values.
- **Handling Missing Data**: Gained crucial skills to identify, remove (dropna()), or fill (fillna()) missing values (NaN), a common step in real-world data.
- **Summarizing with GroupBy**: Understood the powerful groupby() method to categorize data and calculate summaries (like averages or counts) for each group.
- **Working with Time Data**: Learned how to convert text timestamps to proper date/time objects and use resample() to get summaries for different time periods (like daily averages).
- **Efficient Categorical Data:** Understood how to use the 'category' data type for memory efficiency with columns that have limited, repeated text values.

- **Practical Workflow**: Applied a sequence of these Pandas operations in a conceptual lab to analyze a simple telemetry dataset, demonstrating a typical data analysis workflow.

By mastering these fundamental Pandas concepts, you are now well-equipped to start cleaning, transforming, analyzing, and gaining insights from structured datasets in your engineering and scientific projects.

ISRO URSC – Python Training | Analog Data | Rajath Kumar

📩 **For queries:** rajath@analogdata.ai| 📱 (+91) 96633 53992 | 🌐 https://analogdata.ai

---
