Unveiling the Power of Pandas for Panel Data

Minor In AI, IIT Ropar

19th June 2025

Welcome, future AI enthusiasts, to a crucial step in your data journey! We've dipped our toes into the world of numbers with NumPy, learning how to handle arrays and perform speedy calculations. But what happens when our data isn't just a simple grid of numbers? What happens when we collect information over time, perhaps from different places or different subjects?

Imagine you're tracking something like, say, the daily study habits of a group of students preparing for a big exam. For each student, over several days, you might note things like the number of cups of chai they drank, how many lines of code they wrote, how many hours they spent actively studying, and perhaps a 'miscellaneous' category for everything else – sleep, food, relaxation, etc.

This kind of data, collected over a period of time for multiple individuals or entities, is often called **Panel Data**. It's rich, it's layered, and it requires tools that can handle this structure efficiently. While NumPy is great for numerical operations on arrays, it doesn't intuitively understand things like column names ("Cups of Chai", "Hours of Study") or row labels ("Day 1", "Day 2").

This is where **Pandas** swoops in! And no, we're not talking about the adorable, bamboo-chomping bears (though we might wish our data tasks were as chill!). We're talking about the incredibly powerful Python library specifically designed for working with structured data, especially panel data. Pandas builds upon NumPy but adds layers of intelligence, allowing us to easily organize, manipulate, clean, and analyze data that has labels for rows and columns, much like a spreadsheet or a database table.

In this chapter, we'll uncover how Pandas does its magic, starting from building a simple data structure to tackling real-world data challenges like missing values and preparing data for predictive models.

The Heart of Pandas: The DataFrame

At the core of Pandas is a structure called the **DataFrame**. Think of a DataFrame as a superpowered spreadsheet right inside your Python code. It has rows and columns, with labels for both. This structure is particularly well-suited for our panel data because we have different attributes (columns) like 'Cups of Chai', and values for these attributes collected across different instances (rows), like 'Day 1', 'Day 2', etc.

How does Pandas create this magical DataFrame? It cleverly combines fundamental Python data structures we've already met: **dictionaries** and **lists**.

Remember how dictionaries store information as `key: value` pairs? And how lists store ordered collections of items? Pandas uses a dictionary where the *keys* become your column names (like 'Days', 'Cups of Chai'), and the *values* for each key are *lists* containing all the data points for that specific column across the different rows.

Let's recreate our "Student Exam Survival Log" example using this structure, just like we did in the lecture.

First, we need to import the Pandas library. It's standard practice to import it and give it the shorthand name `pd`, just like we use `np` for NumPy.

```
1 import pandas as pd
```

Now, let's build our data using a Python dictionary:

```python
# Data structured as a dictionary
# Keys are column names, Values are lists of data for each column
student_data = {
    'Days': ['First', 'Second', 'Third', 'Fourth', 'Fifth'],
    'Cups of Chai': [7, 5, 8, 9, 4],
    'Code Written': [1300, 800, 600, 1800, 0], # Let's add 0 lines for the last day before exam
    'Hours of Study': [8.5, 4.5, 5.5, 8.5, 6.5],
    'MISC': [3.5, 2.5, 6.5, 3.5, 5.5] # Miscellaneous hours (sleep, etc.)
}
```

See how `Days` is the key, and its value is a list of strings? `Cups of Chai` is another key, and its value is a list of numbers? This dictionary structure perfectly captures our panel data intuition: for each feature (key), we have a list of values collected over time (the days).

Now, we convert this dictionary into a Pandas DataFrame using `pd.DataFrame()`, passing our dictionary to it:

```python
# Create a DataFrame from the dictionary
df_study = pd.DataFrame(student_data)

# Let's see what our DataFrame looks like
print(df_study)
```

If you run this code, you'll see a beautifully formatted table:

```
     Days  Cups of Chai  Code Written  Hours of Study  MISC
0   First             7          1300             8.5   3.5
1  Second             5           800             4.5   2.5
2   Third             8           600             5.5   6.5
3  Fourth             9          1800             8.5   3.5
4   Fifth             4             0             6.5   5.5
```

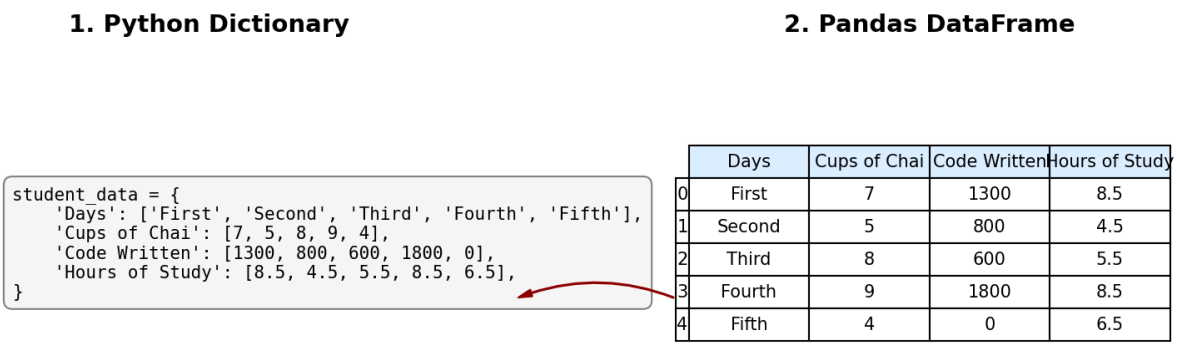

Figure 1: A visualization showing how a Python dictionary (left) is transformed into a Pandas DataFrame (right). The dictionary keys become the column headers.

Notice how Pandas automatically added a numbered index (0, 1, 2, 3, 4) for the rows. This is a default index, and each row represents a specific instance (a day in our case). This structure makes accessing and manipulating data incredibly intuitive, much easier than trying to manage multiple lists or nested data structures manually.

Loading Data from the Real World (CSV Files)

Building small DataFrames from scratch is useful for understanding the concept, but in reality, our data usually comes from files – often CSV (Comma Separated Values) files, like the `students.csv` dataset we discussed in the lecture.

Loading data from a CSV file into a Pandas DataFrame is incredibly simple thanks to the `pd.read_csv()` function. You just need to tell it the path or name of your file.

Assuming you have the `students.csv` file (which contains 'id', 'age', and 'score' for 1000 students, with some missing values) in the same directory as your Python script or Colab notebook, you can load it like this:

```
# Load the students.csv file into a DataFrame
df_students = pd.read_csv('students.csv')

# Let's see the first few rows to get a feel for the data
print(df_students.head())
```

The `.head()` function is super useful! When dealing with large datasets, printing the whole DataFrame can be overwhelming. `.head()` (by default) shows you just the first 5 rows, giving you a quick peek at the column names and the data types.

Now that our data is loaded, we might want to know how many students (rows) are in our dataset. We can easily get the number of rows using the `len()` function on the DataFrame, or by accessing its `.index` property:

```
# Get the total number of students (rows)
total_students = len(df_students)
# or using the index property: total_students = len(df_students.index)

print(f"Total number of students in the dataset: {total_students}")
```

This should tell us we have 1000 students.

Powerful Data Selection and Filtering

One of the most common tasks with data is finding specific pieces of information or filtering rows based on certain conditions. Pandas DataFrames make this incredibly easy.

Suppose we want to see the scores of all students. We can select a single column (which Pandas calls a **Series**) by using square brackets `[]` with the column name:

```
# Select the 'score' column
scores = df_students['score']
print(scores.head())
```

Now, let's apply a condition. What if we only want to see the data for students who scored *more than 80*? This is where Pandas' powerful **boolean indexing** comes in.

First, let's create a condition: `df_students['score'] > 80`. What does this do? For *every* row in the DataFrame, it checks if the value in the 'score' column is greater than 80. It returns a **Boolean Series** – a Series of `True` and `False` values, where `True` indicates the condition was met for that row, and `False` indicates it wasn't.

```python
# Create a boolean Series: True if score > 80, False otherwise
high_score_condition = df_students['score'] > 80
print(high_score_condition.head())
```

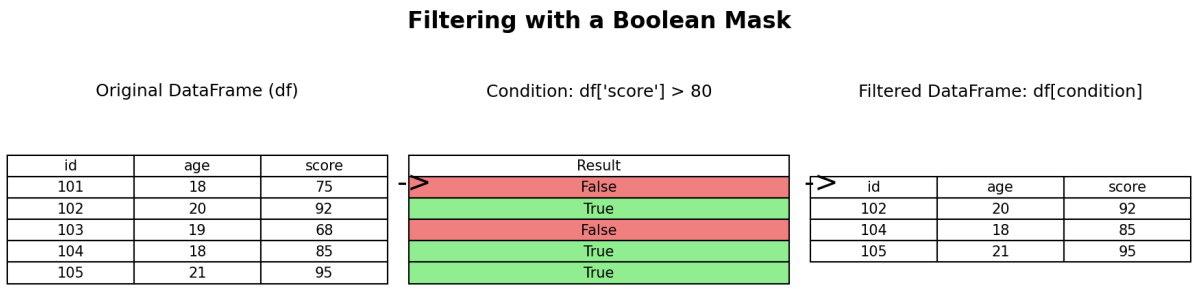

Figure 2: Boolean indexing in action. A condition is applied to a column, creating a ”mask” of True/False values. This mask is then used to filter the original DataFrame, keeping only the rows corresponding to `True`.

Now, here's the clever part. You can pass this Boolean Series *back into* the DataFrame's square brackets! When you do `df_students[high_score_condition]`, Pandas uses the `True`/`False` values to decide which rows to keep. It keeps only the rows where the value in the Boolean Series was `True`.

Combining these steps, we can filter the DataFrame in one go:

```python
# Filter the DataFrame to keep only students with score > 80
df_high_scorers = df_students[df_students['score'] > 80]

# Let's see how many students scored more than 80
total_high_scorers = len(df_high_scorers)

print(f"Total students with score > 80: {total_high_scorers}")

# Look at the first few rows of the filtered data
print(df_high_scorers.head())
```

This should show us that 207 students had a score greater than 80, and display their 'id', 'age', and 'score'. This double bracket syntax (`df[...]` where `...` is a condition involving `df`) is a fundamental and powerful way to filter DataFrames in Pandas. The inner part evaluates the condition for each row, and the outer part selects the rows that meet the condition.

Cleaning Your Data: Handling Missing Values

Looking at the `students.csv` data, you might have noticed some spots where the age or score is missing. These often appear as `NaN` (Not a Number) in Pandas. Missing data is a common challenge in real-world datasets and can cause problems for analysis and machine learning models. Pandas provides simple ways to handle this. We have two main strategies:

1. **Drop the rows (or columns) with missing values.** This is suitable when you have a large dataset and only a few rows have missing data, or if the missingness is critical and cannot be reasonably estimated.

2. **Fill the missing values.** This is useful when you want to preserve your data points, and you can replace the missing values with a reasonable estimate, like the average (mean) of the existing values, the median, or a specific value like zero.

Let's see how to do both using our `df_students` DataFrame.

Method 1: Dropping Missing Values

The `.dropna()` function removes rows (by default, though you can set an axis to drop columns) that contain any `NaN` values.

```
# Create a new DataFrame by dropping rows with any missing values
df_clean_dropped = df_students.dropna()

# See how many students are left after dropping missing values
students_after_drop = len(df_clean_dropped)

print(f"Total students after dropping rows with missing values: {
    students_after_drop}")
```

You should see that the number of students has reduced from 1000 to 810. This means 190 rows had at least one missing value in the 'id', 'age', or 'score' columns.

Method 2: Filling Missing Values

Instead of dropping, we can fill. The `.fillna()` function replaces missing values. We can specify what to fill them with. A common approach, especially for numerical data like 'score' or 'age', is to fill with the mean of the existing values in that column.

To fill the missing scores with the average score:

```
# Calculate the mean score from the non-missing values
mean_score = df_students['score'].mean()

# Create a new DataFrame by filling missing values in the original DataFrame
# with the calculated mean score.
# The lecture showed df.fillna(df['score'].mean()), which fills ALL NaNs
# with the score mean. For simplicity mirroring the lecture, we use this
# syntax, understanding it applies the score mean everywhere there's a NaN.
df_filled = df_students.fillna(mean_score)

# See how many students are in the DataFrame after filling
students_after_fill = len(df_filled)

print(f"Total students after filling missing values with mean: {
    students_after_fill}")

# Look at the first few rows again - you should see the mean value where NaNs
    were
print(df_filled.head())
```

The student count remains 1000, but if you look at the `.head()` output or inspect rows that previously had `NaN`s, you'll see the missing values replaced by the mean score. Filling with the mean is just one method. You could also use the median (`.median()`), a constant value (`.fillna(0)`), or more complex techniques like interpolation. The best method depends on your data and the specific problem you're trying to solve.

Grouping and Summarizing Data

Analyzing data often involves grouping it by categories and then calculating summary statistics (like mean, sum, count) for each group. For instance, in our `students.csv` data, we might want to know the *average score for each age group*.

Pandas' `groupby()` function is perfect for this. You specify the column(s) you want to group by, then select the column(s) you want to analyze within those groups, and finally apply an aggregation function.

To find the mean score for each age:

```
# Group the DataFrame by the 'age' column
# Select the 'score' column for aggregation
# Calculate the mean score for each age group
df_age_group = df_students.groupby('age')['score'].mean()

# Let's see the result
print(df_age_group)
```

The output will show a list of ages (the unique values in the 'age' column) as the index, and the corresponding mean score for students of that age as the value.

Preparing for Predictive Modeling

Ultimately, in the world of AI, we often use cleaned and processed data to train machine learning models that can make predictions. Whether it's predicting a numerical value (regression) or a category (classification), the first step is always to prepare your data by separating it into **features** (the input variables, often denoted by `X`) and the **target** (the variable we want to predict, often denoted by `y`).

Since the `students.csv` file doesn't contain a 'Passed' column, let's quickly create a small DataFrame for demonstrating the X/y split:

```
# Creating a small sample DataFrame for modeling demo
modeling_data = {
    'Hours of Study': [8, 4, 5, 8, 6, 9, 7],
    'Score': [90, 55, 65, 88, 75, 95, 80],
    'Passed': [1, 0, 0, 1, 1, 1, 1] # 1 for pass, 0 for fail
}
df_model = pd.DataFrame(modeling_data)
print(df_model)
```

Now, we can separate our features (`X`) and target (`y`). `X` will be a DataFrame containing the 'Hours of Study' and 'Score' columns. `y` will be a Series containing the 'Passed' column. We select multiple columns for `X` by passing a list of column names inside the double square brackets `[[]]`.

```
# Define features (X) and target (y)
X = df_model[['Hours of Study', 'Score']] # Features
y = df_model['Passed']                    # Target

print("\nFeatures (X):")
print(X)
print("\nTarget (y):")
print(y)
```

Now that we have our data in the standard `X` and `y` format, we are ready to feed it into a machine learning model! Since our target variable `y` ('Passed') is categorical (either 0 or 1), the appropriate type of model for this prediction task is **Logistic Regression**.

```
# Import the Logistic Regression model
from sklearn.linear_model import LogisticRegression

# Create a Logistic Regression model instance
model = LogisticRegression()

# Train the model using our features (X) and target (y)
model.fit(X, y)

# Evaluate the model's accuracy
accuracy = model.score(X, y)

print(f"\nModel Accuracy: {accuracy}")
```

This brief example demonstrates how Pandas helps you prepare your data (`df_model`, selecting `X` and `y`) to be used by powerful machine learning libraries like scikit-learn.

Wrapping Up

Congratulations! You've taken a significant leap in your data manipulation skills by diving into Pandas. You've learned how DataFrames are structured, how to load data from common file types like CSVs, how to select and filter data based on conditions, how to handle pesky missing values by dropping or filling them, and how to summarize data by grouping. Finally, you've seen how Pandas DataFrames prepare your data beautifully for the next stage: building predictive models.

Pandas is an indispensable tool for anyone working with data in AI and machine learning. The concepts covered here are just the tip of the iceberg, but they form a strong foundation for tackling more complex data challenges. Keep practicing, explore the extensive Pandas documentation, and soon you'll be wrangling data like a pro!