

Data Analysis with Pandas:

Core Concepts 4

This guide builds on the foundational concepts of Pandas to explore more advanced data manipulation techniques. We will cover how to filter, sort, and group data, as well as how to combine multiple DataFrames through merging and concatenation. Finally, we'll look at how to add and remove columns and rows. All examples use a consistent dataset to demonstrate practical applications of these concepts.

Introduction to Pandas Data Manipulation

Concept Introduction: Data Manipulation

In real-world scenarios, raw data is rarely ready for analysis. Data manipulation involves a series of steps to clean, transform, and structure data to make it suitable for a specific task. Pandas provides a rich set of tools to perform these operations efficiently and intuitively.

1. DataFrame - Filtering

Concept Introduction: Filtering Data

Filtering is the process of selecting a subset of data from a DataFrame based on specific conditions. This is one of the most common and powerful

operations in data analysis, allowing you to focus on the information that is most relevant to your task.

Example 1: Filtering with a Single Condition

Let's find all the sales transactions where the `product_type` is 'Frozen'. We use a boolean mask to achieve this.

```
import pandas as pd

# Load the sales data
df = pd.read_csv('sales_transactions_50.csv')

# Create a boolean mask where the product_type is 'Frozen'
is_frozen_mask = df['product_type'] == 'Frozen'

# Use the mask to filter the DataFrame and display the first few results
frozen_items = df[is_frozen_mask]

print("Transactions for 'Frozen' items:")
print(frozen_items.head())
print("...")
print(f"Total number of 'Frozen' transactions: {len(frozen_items)}")
```

This code first creates a `Series` of `True / False` values (`is_frozen_mask`) by checking if each row's `product_type` is 'Frozen'. Passing this mask to the DataFrame `df[...]` returns only the rows where the condition is `True`.

Example 2: Filtering with Multiple Conditions

Now, let's find all sales of 'Dessert' items with a `quantity` of more than 3. We combine multiple conditions using logical operators (`&` for AND, `|` for OR, `~` for NOT).

```
# Filtering for 'Dessert' items with a quantity greater than 3
dessert_high_qty_items = df[(df['product_type'] == 'Dessert') &
(df['quantity'] > 3)]

print("Dessert items with quantity > 3:")
print(dessert_high_qty_items)
```

By enclosing each condition in parentheses and joining them with `&`, we create a combined boolean mask. The filtered DataFrame contains only the rows that satisfy both conditions simultaneously. Note that each condition must be enclosed in parentheses to ensure correct operator precedence.

Real-time Usage

A business manager might need to quickly identify all the sales of a particular product line to analyze its performance. For instance, they could filter for all transactions of a 'Frozen' item to see which flavors are most popular or if any are selling poorly. This helps in making informed decisions about inventory and marketing strategies.

2. Data Frame - Sorting

Concept Introduction: Sorting Data

Sorting a DataFrame by its values is essential for organizing data and making it easier to read and analyze. The `sort_values()` method allows you to sort the DataFrame based on one or more columns, in either ascending or descending order.

Example 1: Sorting by a Single Column

Let's sort our sales transactions by `quantity` in ascending order to see which items were sold least.

```
# Load the sales data
df = pd.read_csv('sales_transactions_50.csv')

# Sort the DataFrame by the 'quantity' column in ascending order
(default)
df_sorted_by_quantity = df.sort_values(by='quantity')

print("DataFrame sorted by quantity (ascending, first 5
records):")
print(df_sorted_by_quantity.head())
```

The `sort_values()` method, when given a column name, returns a new DataFrame sorted by that column. The default order is ascending, which means the lowest values appear first.

Example 2: Sorting by Multiple Columns (Descending)

To find the most expensive items that were sold in high quantities, we can sort by `price_per_unit_inr` and then by `quantity`, both in descending order.

```
# Sort by 'price_per_unit_inr' and then by 'quantity', both in
descending order
df_sorted_multi = df.sort_values(by=['price_per_unit_inr',
'quantity'], ascending=[False, False])

print("DataFrame sorted by price (descending) and then quantity
(descending, first 5 records):")
print(df_sorted_multi.head())
```

By passing a list of column names to the ``by`` parameter and a list of boolean values to the ``ascending`` parameter, we can define a multi-level sort. The data is first sorted by the first column in the list, and then any rows with equal values are sorted by the second column, and so on.

Real-time Usage

A data analyst might sort the data to generate a sales leaderboard for staff or to identify the most expensive products sold in a day. Sorting helps in quickly identifying patterns, outliers, and trends, which is a key part of any exploratory data analysis process.

3. Data Frame - GroupBy

Concept Introduction: Grouping Data

The ``groupby()`` method is one of the most powerful features of Pandas. It allows you to split a DataFrame into groups based on some criteria, apply a function (like ``sum()``, ``mean()``, or ``count()``) to each group, and then combine the results. This is similar to the GROUP BY clause in SQL.

Example 1: Grouping by a Single Column and Aggregating

Let's find the total quantity of each product `flavor` sold. We'll group the DataFrame by the ``flavor`` column and then calculate the sum of the ``quantity`` for each group.

```
import pandas as pd

# Load the sales data
df = pd.read_csv('sales_transactions_50.csv')
```

```
# Group by 'flavor' and calculate the sum of 'quantity'
flavor_sales = df.groupby('flavor')['quantity'].sum()

print("Total quantity sold per flavor:")
print(flavor_sales)
```

First, we group the DataFrame by `flavor`. Then, we select the `quantity` column and apply the `sum()` aggregation function. This returns a new Series with the unique flavors as the index and their corresponding total quantities as the values.

Example 2: Grouping by Multiple Columns and Applying Multiple Aggregations

To get a more detailed view, let's group by both `staff_id` and `product_type`, and then find the total `quantity` sold and the total `price_per_unit_inr` for each group. We'll use the `.agg()` method for multiple aggregations.

```
# Group by 'staff_id' and 'product_type', then aggregate
staff_sales_summary = df.groupby(['staff_id',
    'product_type']).agg(
    total_quantity=('quantity', 'sum'),
    average_price=('price_per_unit_inr', 'mean')
)

print("Sales summary by staff and product type:")
print(staff_sales_summary)
```

The `groupby()` method is used with a list of columns. The `agg()` method takes a dictionary where keys are the new column names and values are tuples of the original column and the aggregation function to apply. This creates a powerful summary table with a multi-index.

Real-time Usage

Grouping data is crucial for generating business intelligence reports. A manager might use this to calculate total revenue per staff member, average sales per hour, or to find the most popular product types on a monthly basis. This aggregated data provides the key insights needed to measure performance and make strategic decisions.

4. Merging or Joining

Concept Introduction: Merging DataFrames

Merging or joining is the process of combining two or more DataFrames based on a common column or index. This is a fundamental operation when your data is spread across multiple tables or files, a common scenario in real-world databases. Pandas provides the `merge()` function for this purpose.

Example 1: Merging DataFrames with an Inner Join

Let's combine our `sales_transactions_50` and `staff_details` DataFrames to see which employee made each transaction. We will join them on the common column, `staff_id`.

```
import pandas as pd

# Load the two DataFrames
df_sales = pd.read_csv('sales_transactions_50.csv')
df_staff = pd.read_csv('staff_details.csv')

# Perform an inner merge on the 'staff_id' column
merged_df = pd.merge(df_sales, df_staff, on='staff_id',
                     how='inner')

print("Merged DataFrame (sales with staff names, first 5 records):")
```

```
print(merged_df.head())
```

The `pd.merge()` function is used here. We specify the two DataFrames to merge, the column to join on (`on='staff_id'`), and the type of join (`how='inner'`). An inner join only keeps rows where the `staff_id` exists in both DataFrames, which is the default behavior.

Example 2: Merging with a Left Join

What if we wanted to keep all sales transactions, even if a `staff_id` in the sales data didn't have a match in the staff details? A left join would be appropriate. In this case, since all `staff_id`'s in our sales data have a match, the output will be the same as the inner join, but the principle is important.

```
# Perform a left merge on the 'staff_id' column
left_merged_df = pd.merge(df_sales, df_staff, on='staff_id',
                           how='left')

print("Left merged DataFrame:")
print(left_merged_df.head())
```

A left merge (`how='left'`) returns all rows from the "left" DataFrame (`df_sales`) and the matching rows from the "right" DataFrame (`df_staff`). If there's no match, the columns from the right DataFrame are filled with `NaN` for that row. This is useful for preserving all records from a primary dataset.

Pros and Cons of Merging

- **Pros:**
 - **Versatility:** Allows for powerful combinations of data using different join types (inner, left, right, outer).

- **Clarity:** The `on` parameter makes the join condition explicit and easy to understand.
- **Cons:**
 - **Key Dependency:** Requires a common key or column between the DataFrames to be effective.
 - **Memory Usage:** Creating a new, combined DataFrame can be memory-intensive with very large datasets.

Real-time Usage

Merging is essential for getting a complete picture of a business's operations. A sales manager might merge sales transactions with staff details to analyze sales performance per employee. They could also merge sales data with a separate product catalog to see how many units of a specific item were sold, regardless of the flavor.

5. DataFrame - Concat

Concept Introduction: Concatenating DataFrames

Concatenation is the process of stacking DataFrames on top of each other (vertically) or side by side (horizontally). It is used when you have multiple DataFrames with the same or similar structure that you want to combine into a single, larger DataFrame. The `pd.concat()` function is used for this.

Example 1: Concatenating Vertically

Let's add the new sales transactions from `new_transactions.csv` to our main `sales_transactions_50.csv` DataFrame. We will stack the new data below the old data.

```
import pandas as pd
```

```
# Load the two DataFrames
df_sales = pd.read_csv('sales_transactions_50.csv')
df_new_sales = pd.read_csv('new_transactions.csv')

# Vertically concatenate the two DataFrames
combined_sales = pd.concat([df_sales, df_new_sales],
                           ignore_index=True)

print("Original sales DataFrame tail:")
print(df_sales.tail(3))
print("\nNew transactions DataFrame:")
print(df_new_sales)
print("\nCombined DataFrame after concatenation (tail):")
print(combined_sales.tail(10))
```

The `pd.concat()` function takes a list of DataFrames to combine. By default, it concatenates vertically along the row axis (`axis=0`). We use `ignore_index=True` to reset the index of the final DataFrame, creating a clean, continuous index from 0 upwards.

Pros and Cons of Concatenation

- **Pros:**
 - **Simplicity:** It's a straightforward way to stack or join DataFrames with similar structures.
 - **Performance:** Generally faster than other join methods when simply stacking data.
- **Cons:**
 - **Schema Dependency:** Works best when the DataFrames have the same columns and data types.
 - **Index Duplication:** Without `ignore_index=True`, the final DataFrame can have duplicate index values, which can cause issues.

Real-time Usage

Concatenation is often used to combine daily, weekly, or monthly sales reports into a single, comprehensive dataset for long-term analysis. For example, a business might receive a new CSV file of transactions every day, and a script could use `pd.concat()` to append this new data to a master sales file.

6. DataFrame - Adding, Dropping Columns & Rows

Concept Introduction: Modifying DataFrame Structure

As part of data cleaning and preparation, you will often need to add, remove, or modify columns and rows in a DataFrame. These operations are essential for feature engineering, data normalization, and preparing a dataset for a specific analysis task.

Example 1: Adding a New Column

Let's add a new column, `total_price_inr`, to our sales DataFrame. This new column will be the result of a simple calculation: `quantity` multiplied by `price_per_unit_inr`.

```
import pandas as pd
df = pd.read_csv('sales_transactions_50.csv')

# Add a new column by multiplying existing columns
df['total_price_inr'] = df['quantity'] * df['price_per_unit_inr']

print("DataFrame with the new 'total_price_inr' column (first 5 records):")
print(df.head())
```

Adding a new column is as simple as assigning a new `Series` of values to a new column name. This new column is created and populated based on the

calculation of the two existing columns. This is a very common task for feature creation.

Example 2: Dropping Columns

Sometimes, columns are no longer needed. We can use the `drop()` method to remove one or more columns from a DataFrame. Let's drop the `product_type` column.

```
# Drop the 'product_type' column
df_dropped_col = df.drop(columns='product_type')

print("DataFrame after dropping the 'product_type' column (first 5
records):")
print(df_dropped_col.head())
```

The `drop()` method returns a new DataFrame with the specified column(s) removed. You must specify `axis=1` (or `columns=`) to tell Pandas you are dropping a column and not a row. By default, `drop()` returns a new DataFrame, leaving the original unchanged.

Example 3: Dropping Rows

We can also drop rows based on their index. Let's drop the first and the last rows of our DataFrame, which correspond to index labels 0 and 49.

```
# Drop rows at index 0 and 49
df_dropped_rows = df.drop(index=[0, 49])

print("DataFrame after dropping the first and last rows (head and
tail):")
print(df_dropped_rows.head(2))
print("...")
```

```
print(df_dropped_rows.tail(2))
```

To drop rows, we use the `index` parameter and pass a list of index labels. This method is useful when you want to remove specific records from the dataset, for example, removing a transaction that was found to be fraudulent or duplicated.

Real-time Usage

Data modification is a daily task for an analyst. They might add a new column for 'Profit_Margin' to a sales DataFrame to perform a new analysis. They might also drop a column like 'transaction_id' before training a machine learning model, as it is unlikely to have predictive value. Dropping rows is useful for removing outliers or duplicate entries from a dataset.