Panda 2.0

Welcome to Panda 2.0!

If you're just starting your journey with Python and data analysis, you've come to the right place. This book will guide you through the fundamentals of Pandas, a powerful library for working with data. We'll be using Google Colab, a fantastic environment for writing and running Python code right in your browser.

Figure 1: Pandas can't code but we can do a lot with pandas framework.

1 Combining Datasets - A Real-World Scenario

Imagine you're running an online store. You have two separate spreadsheets: one contains information about your products (like their ID, brand, and price), and the other contains information about your sales (like the product ID, the date of sale, and the quantity sold).

You realize that to gain better insights – like knowing which brands are selling the most or whether higher-priced items have lower sales – you need to combine these two spreadsheets. This is where Pandas comes to the rescue! Pandas helps you bring these separate tables together, clean up any inconsistencies, and analyze the combined data to answer your business questions.

The core of this process is something called "merging" dataframes. Let's dive deeper and explore how Pandas lets us achieve this.

2 Understanding DataFrames

Before we start, let's quickly recap what a Pandas DataFrame is. Think of it as a table in a spreadsheet. It has rows and columns, where each column can hold a different type of data (numbers, text, etc.). You can create a DataFrame from various sources, like a list of dictionaries or even directly from a CSV file (more on that later!).

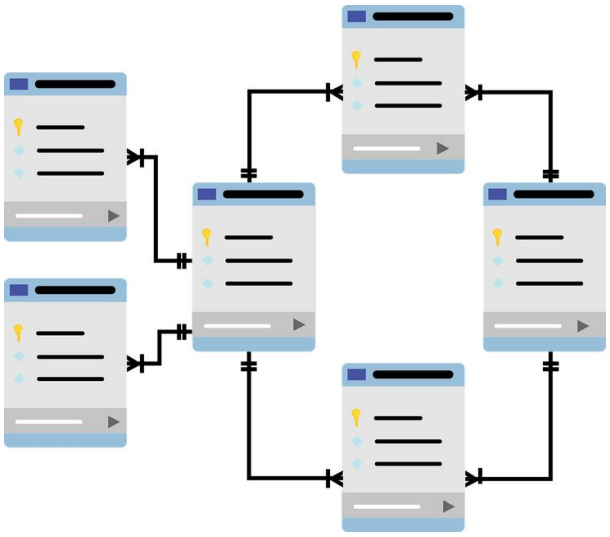

Figure 2: Data Frames can be merged using pandas.

3 Merging DataFrames - Different Approaches

When merging DataFrames, we need a common column (or columns) that connects the two tables. In our online store example, the "product ID" is the common link between the product data and the sales data. But what happens if one table has a product ID that doesn't exist in the other? That's where different types of merges come into play.

Imagine you are combining the following two tables/dataframes:

Product Data (DataFrame 1):

Product ID	Brand	Price
1	Brand A	20
2	Brand B	30
3	Brand C	25
4	Brand D	40
5	Brand E	50

Sales Data (DataFrame 2):

- **Inner Merge:** This keeps only the rows where the "Product ID" exists in *both* DataFrames. In our example, it would only include products with IDs 1, 2, 3, and

Product ID	Sales	Quantity
1	100	5
2	150	8
3	120	6
4	200	10
6	80	4

4. Product ID 5 will be discarded from the first data frame and Product ID 6 will be discarded from the second dataframe.
Example Code:

```
inner_merge = pd.merge(df1, df2, how='inner', on='CustomerID'
    )
print(inner_merge)
```

- **Outer Merge:** This keeps all rows from *both* DataFrames. If a "Product ID" exists in one DataFrame but not the other, the missing values will be filled with "NaN" (Not a Number), representing missing data. So, it would have Product IDs from 1 to 6.
Example Code:

```
outer_merge = pd.merge(df1, df2, how='outer', on='CustomerID'
    )
print(outer_merge)
```

- **Left Merge:** This keeps all rows from the *left* DataFrame (the first one you specify) and the matching rows from the *right* DataFrame (the second one). If a "Product ID" in the left DataFrame doesn't exist in the right, the corresponding columns from the right DataFrame will be filled with "NaN". So, you will have product IDs from 1 to 5.
Example Code:

```
    left_merge = pd.merge(df1, df2, how='left', on='
       CustomerID')
    print(left_merge)
```

- **Right Merge:** This keeps all rows from the *right* DataFrame and the matching rows from the *left* DataFrame. If a "Product ID" in the right DataFrame doesn't exist in the left, the corresponding columns from the left DataFrame will be filled with "NaN". You will have product IDs from 1, 2, 3, 4, and 6.
Example Code:

```
    right_merge = pd.merge(df1, df2, how='right', on='
       CustomerID')
    print(right_merge)
```

Essentially:

- **Inner:** Intersection

- **Outer:** Union

The choice of which merge to use depends on the analysis you're trying to do and the questions you're trying to answer. If you only want information for products that have both product details *and* sales data, an inner merge is the way to go. If you want *all* product details, even if there are no sales recorded, a left merge is appropriate.

Example Code:

```
import pandas as pd
df = pd.read_csv("name_of_your_data_file.csv")


# First DataFrame: Customer info
df1 = pd.DataFrame({
    'CustomerID': [1, 2, 3, 4],
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, 30, 35, 40]
})

# Second DataFrame: Orders
df2 = pd.DataFrame({
    'CustomerID': [1, 2, 5],
    'OrderID': [101, 102, 103],
    'Amount': [500, 300, 700]
})

print("df1:\n", df1)
print("df2:\n", df2)
```

Output:

```
df1:
   CustomerID      Name  Age
0           1     Alice   25
1           2       Bob   30
2           3   Charlie   35
3           4     David   40

df2:
   CustomerID  OrderID  Amount
0           1      101     500
1           2      102     300
2           5      103     700
```

4 Descriptive Statistics - Understanding Your Data

Once you have your combined DataFrame, you'll want to understand the data better. Pandas provides a handy function called `.describe()` that gives you a summary of the numerical columns.

For each numerical column, `.describe()` calculates:

- **Count:** The number of non-missing values.
- **Mean:** The average value.
- **Standard Deviation:** A measure of how spread out the values are.
- **Minimum:** The smallest value.
- **25th Percentile (Q1):** The value below which 25% of the data falls.
- **50th Percentile (Q2):** The median value.
- **75th Percentile (Q3):** The value below which 75% of the data falls.
- **Maximum:** The largest value.

This information helps you quickly get a sense of the distribution and range of your data.

5 Visualizing Your Data

Pandas also integrates well with plotting libraries like Matplotlib and Seaborn. You can use the `.plot()` function to create various types of charts directly from your DataFrame, to see how the data changes over the dataset.'

```
#if df is our dataframe we can plot a graph like this
df.plot()
```

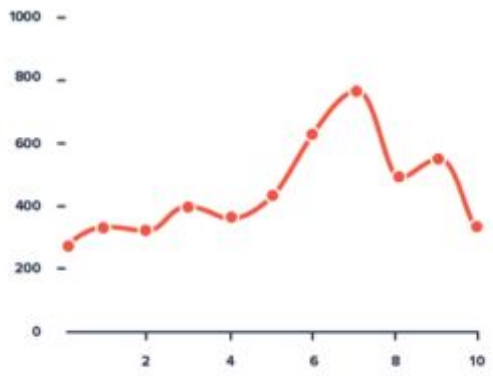

Figure 3: Graph by using df.plot() .

6 Handling Missing Data

Real-world data is often messy and contains missing values. Pandas provides tools for handling these "NaN" values.

- **Identifying Missing Data:** You'll see "NaN" in your DataFrame where data is missing.
- **Filling Missing Data:** You can fill "NaN" values with different strategies:
  - **Filling with a Constant Value:** Replace all "NaN"s with a specific value, like 0 or the average.
  - **Filling with the Mean/Median/Mode:** Replace "NaN"s with the mean, median, or mode of the column.
  - **Forward Fill:** Replace "NaN"s with the value from the previous row.
  - **Backward Fill:** Replace "NaN"s with the value from the next row.
  - **Interpolation:** Estimate missing values based on the surrounding data. You could take the average of the rows before and after the NaN values, or use statistical techniques to estimate the data.
  - **Drop NA:** Remove all of the rows with NaN values.
- **Dropping Missing Data:** If you have a lot of missing data in a particular row, you can simply remove those rows from your DataFrame.

The choice of which method to use depends on the nature of your data and the analysis you're trying to perform. Be careful when filling missing data, as you don't want to introduce bias or distort your results.

Code From Class

```
import pandas as pd
import numpy as np

# Sample DataFrame with NaN values
data = {
    'item_id': [101, 102, 103, 104, 105],
    'item': ['Apple', 'Banana', 'Orange', np.nan, 'Grapes'],
    'price': [50, np.nan, 30, 40, np.nan],
    'sales': [200, 150, np.nan, 180, 220]
}

df = pd.DataFrame(data)

print("Original␣DataFrame␣with␣NaN␣values:")
print(df)

# 1. Fill NaN with a specific value
df1 = df.fillna(0)

# 2. Fill NaN in a specific column with a fixed value
df2 = df.copy()
```

```
df2['price'] = df2['price'].fillna(df2['price'].mean())  #
    Filling NaN with mean value

# 3. Fill NaN using forward fill (propagate last valid value
    forward)
df3 = df.ffill()

# 4. Fill NaN using backward fill (propagate next valid value
    backward)
df4 = df.bfill()

# 5. Fill NaN with the median value of the column
df5 = df.copy()
df5['price'] = df5['price'].fillna(df5['price'].median())

# 6. Fill NaN with the mode (most frequent value) of a column
df6 = df.copy()
df6['item'] = df6['item'].fillna(df6['item'].mode()[0])

# 7. Fill NaN using interpolation
df7 = df.copy()
df7['sales'] = df7['sales'].interpolate()

# 8. Fill NaN using a dictionary of values for each column
df8 = df.fillna({'item': 'Unknown', 'price': 45, 'sales': 95 })

# 9. Drop rows containing NaN values
df9 = df.dropna()

# Displaying results
print("\nFill␣NaN␣with␣0:\n", df1)
print("\nFill␣NaN␣in␣'price'␣with␣mean:\n", df2)
print("\nForward␣Fill␣(ffill):\n", df3)
print("\nBackward␣Fill␣(bfill):\n", df4)
print("\nFill␣NaN␣in␣'price'␣with␣median:\n", df5)
print("\nFill␣NaN␣in␣'item'␣with␣mode:\n", df6)
print("\nInterpolate␣missing␣values␣in␣'sales':\n", df7)
print("\nFill␣NaN␣with␣a␣dictionary:\n", df8)
print("\nDrop␣rows␣with␣NaN:\n", df9)
```

Output:

```
Original DataFrame with NaN values:
   item_id    item  price  sales
0      101   Apple   50.0  200.0
1      102  Banana    NaN  150.0
2      103  Orange   30.0    NaN
3      104     NaN   40.0  180.0
4      105  Grapes    NaN  220.0

Fill NaN with 0:
    item_id    item  price  sales
```

```
0       101    Apple   50.0  200.0
1       102   Banana    0.0  150.0
2       103   Orange   30.0    0.0
3       104        0   40.0  180.0
4       105   Grapes    0.0  220.0

Fill NaN in 'price' with mean:
    item_id     item  price  sales
0       101    Apple   50.0  200.0
1       102   Banana   40.0  150.0
2       103   Orange   30.0    NaN
3       104      NaN   40.0  180.0
4       105   Grapes   40.0  220.0

Forward Fill (ffill):
    item_id     item  price  sales
0       101    Apple   50.0  200.0
1       102   Banana   50.0  150.0
2       103   Orange   30.0  150.0
3       104   Orange   40.0  180.0
4       105   Grapes   40.0  220.0

Backward Fill (bfill):
    item_id     item  price  sales
0       101    Apple   50.0  200.0
1       102   Banana   30.0  150.0
2       103   Orange   30.0  180.0
3       104   Grapes   40.0  180.0
4       105   Grapes    NaN  220.0

Fill NaN in 'price' with median:
    item_id     item  price  sales
0       101    Apple   50.0  200.0
1       102   Banana   40.0  150.0
2       103   Orange   30.0    NaN
3       104      NaN   40.0  180.0
4       105   Grapes   40.0  220.0

Fill NaN in 'item' with mode:
    item_id     item  price  sales
0       101    Apple   50.0  200.0
1       102   Banana    NaN  150.0
2       103   Orange   30.0    NaN
3       104    Apple   40.0  180.0
4       105   Grapes    NaN  220.0

Interpolate missing values in 'sales':
    item_id     item  price  sales
0       101    Apple   50.0  200.0
1       102   Banana    NaN  150.0
2       103   Orange   30.0  165.0
```

```
3       104      NaN   40.0  180.0
4       105   Grapes    NaN  220.0

Fill NaN with a dictionary:
    item_id      item  price  sales
0       101     Apple   50.0  200.0
1       102    Banana   45.0  150.0
2       103    Orange   30.0   95.0
3       104   Unknown   40.0  180.0
4       105    Grapes   45.0  220.0

Drop rows with NaN:
    item_id    item  price  sales
0       101   Apple   50.0  200.0
```

7 Loading Data from CSV Files

Instead of defining a dataframe using the `pd.DataFrame()` command, you can also load data from your computer directly. CSV, or comma-separated values, is a common file format for storing tabular data, such as spreadsheets or databases.

You can upload the CSV files using Google Colab. After the data is in Colab, use the `read_csv` command to read the files.

```
import pandas as pd
df = pd.read_csv("name_of_your_data_file.csv")
print(df)
```

8 Bringing It All Together - A Case Study

Let's revisit our online store example. You now have the product data and sales data combined into a single DataFrame. You've cleaned up any missing values and calculated descriptive statistics.

Now you can start answering interesting questions:

- What is the average price of products sold in each quantity range?
- Which brands have the highest sales?
- Is there a correlation between price and quantity sold?
- Which industry had the fastest growth adoption rate over any consecutive three-year period?

By using Pandas to manipulate and analyze your data, you can gain valuable insights that can help you improve your business.

Conclusion

Pandas is an indispensable tool for anyone working with data in Python. By understanding the concepts we've covered in this book, you'll be well-equipped to tackle a wide range of data analysis tasks. So, get out there, experiment with your own datasets, and discover the power of Pandas! Good luck!