Data Analytics - Module I: Data Cleaning, Preparation and Analysis with Python

Objectives

- Understand data types and their significance for analysis and visualization.
- Master the Pandas library to import, explore, and manipulate datasets efficiently.
- Develop skills in data selection, slicing, and filtering to extract subsets based on specific criteria.
- Perform summary statistics and aggregation functions to analyze and summarize data sets effectively.
- Gain proficiency in merging DataFrames and saving data in various file formats for future use.
- Develop strategies to effectively address and remove missing values, outliers, and duplicate records from datasets.
- Master data reshaping techniques to enhance analysis and visualization processes.
- Gain proficiency in identifying inconsistencies and anomalies within datasets.
- Explore univariate analysis methods, including statistical measures and histograms.
- Extend analysis to bivariate scenarios to investigate relationships between two variables.

Introduction

In Machine Learning (ML), data analysis involves examining and preparing datasets to understand their patterns, trends, and potential biases. This step helps in selecting relevant features and cleaning the data for training ML models. Visualization, on the other hand, employs graphs, charts, and plots to represent data visually. It aids in intuitively grasping data distributions, relationships, and anomalies, making it easier to make informed decisions during model development and interpretation. Data analysis and visualization together empower ML practitioners to make better choices, improve model performance, and communicate results effectively.

1. Identify, Interpret, and Communicate with Data

Identifying, interpreting, and communicating with data involves a systematic approach that helps extract valuable insights and convey meaningful information. Here are some steps to take to achieve this:

1) Define clear objectives for data analysis, outlining the problem or decision to be addressed.
2) Gather relevant data from various sources, ensuring its accuracy and completeness.
3) Conduct thorough data cleaning and preprocessing to rectify errors, handle missing values, and ensure consistency.
4) Perform exploratory data analysis using visualizations and statistical techniques to understand data distribution, trends, and relationships.
5) Communicate insights effectively through data visualization, interactive dashboards, storytelling, and concise reports to facilitate informed decision-making and understanding.

Throughout this process, remember the importance of domain knowledge. Understanding the context of the data and the problem you're solving is crucial for accurate interpretation and effective communication. By following these steps, you'll be able to harness the power of data to make informed decisions, uncover hidden trends, and share valuable insights with others.

In our data introduction session, we've covered various data types such as structured, unstructured, and semi-structured data. For a quick reminder, you can always revisit the session content.

2. Learn Pandas

Pandas is an open-source Python library offering robust tools for both data manipulation and analysis. Its intuitive and efficient data structures, combined with a wide range of functions and methods, make it a valuable tool for individuals working with data. This includes data scientists, analysts, researchers, and engineers. Pandas is widely used for handling structured data, such as tabular data—similar information found in spreadsheets and databases. It supports various formats, including csv, xlsx, JSON, Parquet, HDF5, and more.

Note

To install Pandas using pip, you can open your command prompt or terminal and use the following commands

- pip install pandas

2.1. Know Your Data

The generated dataset is a synthetic example that simulates real-world data with various features. It consists of the following columns:

- Id: A unique identifier for each data entry, ranging from 1 to 500.
- Gender: Represents the gender of individuals, with values ‘M’, ‘F’, or missing (NaN) to simulate missing data or data inconsistencies.
- Age: Represents the age of individuals, ranging from 18 to 65, with some missing values introduced to simulate incomplete data.
- Income: Represents the income of individuals, ranging from 20,000 to 150,000, with missing values introduced to simulate missing or incomplete data.
- Height: Represents the height of individuals in centimeters, ranging from 150 to 190, with missing values introduced to simulate incomplete data.
- Weight: Represents the weight of individuals in kilograms, ranging from 50 to 100, with missing values introduced to simulate incomplete data.

The dataset has been intentionally created with missing values, inconsistencies, and unrealistic values in some cases to mimic the kind of imperfections often found in real-world datasets. The dataset has been saved to a CSV file named ‘sample_dataset_pandas.csv’. First few data samples are given below:

```
Id,Gender,Age,Income,Height,Weight
1,,24.0,,175.80577900049343,61.75595372098708
2,F,34.0,,188.5693286082469,84.08907266914984
3,F,29.0,91758.0,,
4,M,38.0,107861.0,167.1384894175738,75.66437539821675
5,M,35.0,137311.0,170.74060205382588,77.55824926347873
```

```
6,,60.0,148722.0,171.25568067326594,
```

Create a new directory called **“Data”** within the directory that contains your code and download the [sample_dataset_pandas.csv]() into the **“Data”** directory.

2.2. Load Pandas

Python doesn’t load all of the libraries available to it by default. We have to add an import statement to our code in order to use library functions. To import a library, we use the syntax **import libraryName**. If we want to give the library a nickname to shorten the command, we can add it **as nickNameHere**. An example of importing the **pandas** library using the common nickname **pd** is below.

Code:

```python
import pandas as pd
```

Whenever we invoke a function from a library, the syntax follows this pattern: **LibraryName.FunctionName**. The presence of the library name, separated by a dot before the function name, guides Python to locate and use that function. In the provided illustration, we’ve imported the Pandas library and designated it as pd. This smart trick allows us to avoid typing out the full “pandas” keyword every time we use a function from the Pandas library.

2.3. Read CSV file using Pandas

Pandas library in Python can be used to import data stored in a Comma-Separated Values (CSV) file format. CSV is a common and simple way of structuring tabular data, where each line corresponds to a row and the values within a line are separated by commas. Pandas, a popular data manipulation library, provides efficient tools for reading, processing, and analyzing such tabular data.

To read CSV data using Pandas, we typically use the **pd.read_csv()** function. This function takes the path to the CSV file as input and returns a Pandas DataFrame, which is a two-dimensional tabular data structure similar to a table in a database or a spreadsheet.

Here's an overview of the steps involved in reading CSV data using Pandas:

1. **Import Pandas:** Begin by importing the Pandas library with an **import pandas as pd** statement.
2. **Specify File Path:** Before you can read your CSV file using Pandas, you need to specify where the file is located in relation to your current working directory. For example:

    a. If your CSV file named data.csv is in the same directory as your Python script or notebook, you simply use the file name:

    Code:

    ```
    file_path = 'data.csv'
    ```

    b. If your CSV file is in a directory named data within your current working directory, the path would be:

    Code:

    ```
    file_path = 'data/data.csv'
    ```

    c. For an absolute path, which specifies the exact location on your file system, it might look something like this on a Windows system:

    Code:

    ```
    file_path =
    'C:/Users/YourUsername/Documents/data/data.csv'
    ```

    d. Or like this on a Unix/Linux system:

Code:

```
file_path =
'/home/yourusername/documents/data/data.csv'
```

3. **Read CSV:** Use the **pd.read_csv()** function to read the CSV file. Provide the file path as an argument. Additional optional parameters can be used to customize how Pandas reads the data, such as specifying delimiter, encoding, header row, etc.
4. **DataFrame:** The result of **pd.read_csv()** is a Pandas DataFrame. This DataFrame contains the CSV data in a tabular structure, with rows and columns.
5. **Data Exploration**: Once you have the data in a DataFrame, you can explore, manipulate, clean, and analyze it using various Pandas methods and functions.

Code:

```
import pandas as pd

# Read CSV data
file_path = 'Data/sample_dataset_pandas.csv'
pd.read_csv(file_path)
```

Output:

```
     Id   Gender Age  Income Height    Weight
0    1    NaN   24.0  NaN   175.805779 61.755954
1    2    F     34.0  NaN   188.569329 84.089073
2    3    F     29.0  91758.0     NaN  NaN
3    4    M     38.0  107861.0   167.138489 75.664375
4    5    M     35.0  137311.0   170.740602 77.558249
...  ...  ...   ...   ...   ...   ...
495  496  M     63.0  119736.0   162.803814 63.614329
496  497  F     37.0  NaN   164.546003 80.388584
497  498  NaN   24.0  71117.0    180.631835 54.797833
498  499  M     49.0  55912.0    158.822222 NaN
499  500  F     20.0  NaN   182.814738 51.138366
```

```
500 rows × 6 columns
```

We can see that 500 rows have been successfully parsed. Each of these rows consists of 6 distinct columns. Notably, the initial column is known as the DataFrame's index. This index serves to locate data positions, although it doesn't represent an actual DataFrame column.

> **Note**
>
> - While we utilized the **read_csv** function to load CSV datasets, it's worth noting that pandas offer versatile methods for loading various types of datasets, such as Excel spreadsheets using **read_excel**, JSON datasets using **read_json**, and SQL databases using **read_sql**.

The read_csv function from the Pandas library has appropriately processed our file. However, it's essential to recognize that this data isn't yet stored in memory for our manipulation. To work with this data, we must allocate the DataFrame to a variable. Let's say the variable name is 'df'

Code:

```python
import pandas as pd

# Read CSV data into a DataFrame
df = pd.read_csv('Data/sample_dataset_pandas.csv')
```

It's important to note that when we assign an imported DataFrame to a variable, Python doesn't generate any immediate output on the screen. To see the output of the df object, we can directly enter its name into the Python command prompt, and the output will be the same as the previous one.

Code:

```python
df
```

If the dataset contains so many samples then it is a good idea to use the **head()** function of Pandas to see the first few samples of the dataset.

Code:

```
df.head()
```

Output:

	Id	Gender	Age	Income	Height	Weight
0	1	NaN	24.0	NaN	175.805779	61.755954
1	2	F	34.0	NaN	188.569329	84.089073
2	3	F	29.0	91758.0	NaN	NaN
3	4	M	38.0	107861.0	167.138489	75.664375
4	5	M	35.0	137311.0	170.740602	77.558249

We can see what kind of thing is our 'df' using '**type()**' function

Code:

```
type(df)
```

Output:

```
pandas.core.frame.DataFrame
```

Yes, as expected, 'df' is a dataframe. In addition to this, we can also check what kind of things 'df' contains using '**dtypes**'.

Code:

```
df.dtypes
```

Output:

```
Id           int64
Gender      object
Age        float64
Income     float64
```

```
Height    float64
Weight    float64
dtype: object
```

It includes information about the name of each column (‘Id’, ‘Gender’, ‘Age’, ‘Income’, ‘Height’, ‘Weight’) and the corresponding data type for each column (‘int64’, ‘object’, ‘float64’, ‘float64’, ‘float64’, ‘float64’). The data types signify the nature of the values contained in each column, such as integers, strings (objects), and floating-point numbers. This information helps in understanding the structure of the DataFrame and the kind of data it holds

2.4. Explore the DataFrame Object

When exploring a DataFrame object in Pandas, we can use both attributes and methods to access information and perform operations. Here’s how we can distinguish between them using the **dot (.)** notation:

Methods are functions that we can apply to the DataFrame to perform specific operations. They usually require parentheses. If we wish to see the information of a dataframe, we can use the ‘**info()**’ function:

Code:

```
info = df.info()
print(info)
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500 entries, 0 to 499
Data columns (total 6 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Id      500 non-null    int64  
 1   Gender  340 non-null    object 
 2   Age     450 non-null    float64
 3   Income  392 non-null    float64
 4   Height  449 non-null    float64
```

```
 5   Weight  447 non-null    float64
dtypes: float64(4), int64(1), object(1)
memory usage: 23.6+ KB
None
```

The provided output is a summary description of a Pandas DataFrame. Let's break down the key information:

- Class: This DataFrame is an instance of the class pandas.core.frame.DataFrame.
- RangeIndex: The DataFrame has a range index, indicating that it starts from index 0 and goes up to 499. There are a total of 500 entries (rows) in the DataFrame.
- Data columns: The DataFrame has 6 columns in total.
- Column Information:
    - Id: An integer column with 500 non-null values.
    - Gender: An object (string) column with 340 non-null values.
    - Age: A floating-point column with 450 non-null values.
    - Income: A floating-point column with 392 non-null values.
    - Height: A floating-point column with 449 non-null values.
    - Weight: A floating-point column with 447 non-null values.
    - Data Types: The data types for each column are specified:
        - Four columns have a data type of float64 (floating-point numbers).
        - One column has a data type of int64 (integer).
        - One column has a data type of object (likely string or mixed data types).
- Memory Usage: The approximate memory usage of the DataFrame is given as 23.6+ KB.

This summary provides valuable information about the DataFrame's structure, data types, and the presence of missing values. It's a quick overview that helps you understand the content and characteristics of the DataFrame.

Again if we want to see the first 3 samples of the dataset:

Code:

```
# Display the first 3 rows using the head method
first_rows = df.head(3)
print(first_rows)
```

Output:

```
  Id Gender   Age   Income      Height     Weight
0   1    NaN  24.0      NaN  175.805779  61.755954
1   2      F  34.0      NaN  188.569329  84.089073
2   3      F  29.0  91758.0         NaN        NaN
```

Furthermore, attributes are properties of the DataFrame that provide information about its characteristics. They don't require parentheses.

Code:

```
# Get the shape of the DataFrame (attribute)
shape = df.shape
print(shape)  # Output: (rows, columns)
```

Output:

```
(500, 6)
```

Exercise 2.1

- What would be the output of the following command
    - df.tail()
    - df.columns

We can use **'unique()'** function to identify the distinct values within a column or an array.

Code:

```
pd.unique(df['Gender'])
```

Output:

```
array([nan, 'F', 'M'], dtype=object)
```

Exercise 2.2

- What would be the output of the following command
  - age_list = pd.unique(df['Age'])
    len(age_list)

2.5. Selecting Data Using Labels (Column Headings)

To select a single column, use the DataFrame's name followed by the column label in square brackets (**['ColumnLabel']**).

Code:

```
# Select the 'Gender' column
df['Gender']
```

Output:

```
 0       NaN
1         F
2         F
3         M
4         M
       ...
495       M
496       F
497     NaN
498       M
499       F
Name: Gender, Length: 500, dtype: object
```

Note

- We can also use column name as an 'attribute' to access data from that column
  - df.Gender

- The output will be the same as the previous one

To select multiple columns, enclose the column labels in double square brackets (**[['Column1', 'Column2']]**).

Code:

```
# Select the 'Age' and 'Income' columns
df[['Age', 'Income']]
```

Output:

```
      Age    Income
0    24.0       NaN
1    34.0       NaN
2    29.0   91758.0
3    38.0  107861.0
4    35.0  137311.0
..    ...       ...
495  63.0  119736.0
496  37.0       NaN
497  24.0   71117.0
498  49.0   55912.0
499  20.0       NaN

[500 rows x 2 columns]
```

We can also create a new object and store the result, and later we can access the result from the object.

Code:

```
# Select the 'Age' and 'Income' columns and store in a
object
age_income_columns = df[['Age', 'Income']]
print("\nSelected Age and Income columns:\n",
age_income_columns)
```

Output:

```
Selected Age and Income columns:
      Age    Income
0    24.0       NaN
1    34.0       NaN
2    29.0   91758.0
3    38.0  107861.0
4    35.0  137311.0
..    ...       ...
495  63.0  119736.0
496  37.0       NaN
497  24.0   71117.0
498  49.0   55912.0
499  20.0       NaN

[500 rows x 2 columns]
```

Exercise 2.3

- What happens if you ask for a column that doesn't exist?
    - df['Name']

2.6. Extracting Range-based Subsets (Slicing)

Slicing is a technique used to extract a portion or subset of elements from a sequence, such as a list, array, or string. It allows us to specify a range of indices to retrieve a subset of the data. The syntax for slicing is typically **start_index:end_index**, where **start_index** is the index of the first element we want to include, and **end_index** is the index immediately after the last element we want to include.

- Getting Specific Elements
    - We can use slicing to extract specific elements from a sequence. For instance, **sequence[start_index]** retrieves the element at the start_index.

- Getting a Set of Elements

- By using slicing, we can obtain a set of consecutive elements. For example, **sequence[start_index:end_index]** retrieves elements from start_index up to, but not including, end_index.

- Getting First Few Elements
    - To retrieve the first several elements, we can use slicing with a start index of 0. For example, **sequence[:end_index**] extracts elements from the beginning up to end_index.

- Getting Last Few Elements
    - Similarly, to retrieve the last few elements, use a slice-like **sequence[-num_elements:]**, which retrieves the last num_elements from the sequence.

Code:

```
# Sample list
my_list = [10, 20, 30, 40, 50, 60, 70, 80]

# Getting specific elements
element_at_index_2 = my_list[2]
print("Element at index 2:", element_at_index_2)

# Getting a set of elements
subset = my_list[2:5]
print("Subset from index 2 to 4:", subset)

# Getting first few elements
first_three_elements = my_list[:3]
print("First three elements:", first_three_elements)

# Getting last few elements
last_two_elements = my_list[-2:]
print("Last two elements:", last_two_elements)
```

Output:

```
Element at index 2: 30
```

```
Subset from index 2 to 4: [30, 40, 50]
First three elements: [10, 20, 30]
Last two elements: [70, 80]
```

Exercise 2.4

- What would be the output of the following command
  my_list[len(my_list)]

2.7. Slicing Rows and Columns

Slicing rows and columns simultaneously involves using **.loc** or **.iloc** and specifying the row indices and column labels or indices we want to include.

- **.loc** is **label-based** indexing, meaning we specify the row and column labels.
- **.iloc** is **integer-based** indexing, meaning we use integer indices for rows and columns.

Using .loc:

We can use .loc to slice rows and columns by specifying the row and column labels we want to include.

Code:

```
# Slice rows 1 to 3 and columns 'Gender' and 'Age'
sliced_rows_columns_loc = df.loc[1:3, ['Gender', 'Age']]
print("Sliced Rows and Columns using .loc:\n",
sliced_rows_columns_loc)
```

Output:

```
Sliced Rows and Columns using .loc:
   Gender   Age
1       F  34.0
2       F  29.0
3       M  38.0
```

Now, if we want to select 'Gender','Age', and 'Weight' columns with row labels "1, 3, 4", we can also do this using the below code

Code:

```
# Slice rows 1 to 3 and columns 'Gender' and 'Age'
sliced_rows_columns_loc2 = df.loc[[1, 3, 4], ['Gender',
'Age', 'Weight']]
print("Sliced Rows and Columns using .loc:\n",
sliced_rows_columns_loc2)
```

Output:

```
Sliced Rows and Columns using .loc:
 Gender  Age    Weight
1      F   34  84.089073
3      M   38  75.664375
4      F   35  77.558249
```

Using .iloc:

We can use .iloc to slice rows and columns by specifying integer indices for rows and columns.

Code:

```
# Slice rows 1 to 3 and columns at index 1 to 3
sliced_rows_columns_iloc = df.iloc[1:4, 1:4]
print("Sliced Rows and Columns using .iloc:\n",
sliced_rows_columns_iloc)
```

Output:

```
Sliced Rows and Columns using .iloc:
   Gender   Age    Income
1      F  34.0       NaN
2      F  29.0   91758.0
3      M  38.0  107861.0
```

In both cases, the first argument specifies the rows to include, and the second argument specifies the columns to include.

> **Note**
> - The slicing is inclusive for the starting index and exclusive for the ending index.

2.8. Subsetting Data using Criteria

Subsetting data using criteria involves selecting a subset of rows from a DataFrame based on specific conditions. This is often done to filter out rows that meet certain criteria or to focus on specific data points that are relevant to our analysis.

We can use conditional statements to filter rows based on specific criteria. The condition is typically applied to a column, and rows meeting the condition are retained.

For example, let's say we want to subset the DataFrame to only include individuals with an age greater than 25

Code:

```
# Subset data for individuals with age > 25
subset_age_gt_25 = df[df['Age'] > 25]
print("Subset of individuals with age > 25:\n",
subset_age_gt_25)
```

Output:

```
Subset of individuals with age > 25:
       Id Gender   Age    Income      Height     Weight
1      2      F  34.0       NaN  188.569329  84.089073
2      3      F  29.0   91758.0         NaN        NaN
3      4      M  38.0  107861.0  167.138489  75.664375
4      5      M  35.0  137311.0  170.740602  77.558249
5      6    NaN  60.0  148722.0  171.255681        NaN
..   ...    ...   ...       ...         ...        ...
```

```
490  491     NaN  48.0        NaN  178.828853  76.748475
493  494       F  29.0        NaN  166.562967  60.780068
495  496       M  63.0  119736.0  162.803814  63.614329
496  497       F  37.0        NaN  164.546003  80.388584
498  499       M  49.0   55912.0  158.822222        NaN

[375 rows x 6 columns]
```

Also, we can combine multiple criteria using logical operators such as & (AND) and | (OR) to create more complex conditions.

For instance, to subset the DataFrame for individuals with an age greater than 25 and an income greater than 60000:

Code:

```
# Subset data for individuals with age > 25 and income > 60000
subset_age_income = df[(df['Age'] > 25) & (df['Income'] > 60000)]
print("Subset of individuals with age > 25 and income > 60000:\n", subset_age_income)
```

Output:

```
Subset of individuals with age > 25 and income > 60000:
       Id Gender   Age    Income      Height     Weight
2      3      F  29.0   91758.0         NaN        NaN
3      4      M  38.0  107861.0  167.138489  75.664375
4      5      M  35.0  137311.0  170.740602  77.558249
5      6    NaN  60.0  148722.0  171.255681        NaN
6      7      F  31.0   71724.0  189.389367  98.780338
..   ...    ...   ...       ...         ...        ...
479  480      F  65.0  145404.0  151.032340  72.705696
480  481    NaN  40.0  100749.0         NaN  50.180693
482  483    NaN  49.0  122247.0  161.812798  71.048448
485  486    NaN  62.0   97018.0  161.654412  62.798616
495  496      M  63.0  119736.0  162.803814  63.614329
```

```
[217 rows x 6 columns]
```

We can also use the **~** symbol to negate a condition. For example, to subset the DataFrame for individuals with an age less than or equal to 25:

Code:

```
# Subset data for individuals with age <= 25
subset_age_le_25 = df[~(df['Age'] > 25)]
print("Subset of individuals with age <= 25:\n",
subset_age_le_25)
```

Output:

```
Subset of individuals with age <= 25:
       Id Gender   Age    Income      Height     Weight
0      1    NaN  24.0       NaN  175.805779  61.755954
9     10    NaN   NaN       NaN         NaN  95.598488
14    15      M   NaN  106906.0  167.097076        NaN
18    19      M  23.0   94731.0  181.581347  56.365664
19    20    NaN   NaN   83835.0  181.556427  61.367387
..   ...    ...   ...       ...         ...        ...
491  492      M  22.0   85040.0  160.230433  92.393771
492  493      M  20.0  113665.0  158.677897  50.582680
494  495    NaN  24.0   37457.0  170.864801  74.668351
497  498    NaN  24.0   71117.0  180.631835  54.797833
499  500      F  20.0       NaN  182.814738  51.138366

[125 rows x 6 columns]
```

The **isin()** function is used to filter data based on whether values are present in a specified list or iterable. It's a convenient way to subset data when we want to select rows that match specific values for a particular column.

Let's say we want to select rows where the 'Gender' column has values 'M' or 'F':

Code:

```
# Subsetting data using isin() function
subset_gender = df[df['Gender'].isin(['M', 'F'])]
print("Subset of data with 'Gender' values 'M' or 'F':\n",
subset_gender)
```

Output:

```
Subset of data with 'Gender' values 'M' or 'F':
       Id Gender   Age    Income      Height     Weight
1      2      F  34.0       NaN  188.569329  84.089073
2      3      F  29.0   91758.0         NaN        NaN
3      4      M  38.0  107861.0  167.138489  75.664375
4      5      M  35.0  137311.0  170.740602  77.558249
6      7      F  31.0   71724.0  189.389367  98.780338
..   ...    ...   ...       ...         ...        ...
493  494      F  29.0       NaN  166.562967  60.780068
495  496      M  63.0  119736.0  162.803814  63.614329
496  497      F  37.0       NaN  164.546003  80.388584
498  499      M  49.0   55912.0  158.822222        NaN
499  500      F  20.0       NaN  182.814738  51.138366

[340 rows x 6 columns]
```

isnull() and **notnull()** functions are used to detect missing (**NaN**) values in a DataFrame. isnull() returns a DataFrame of the same shape as the input, with True values indicating missing values. notnull() returns the opposite.

Let's say we want to select rows where the 'Age' column has missing values:

Code:

```
# Subsetting data using isnull() function
subset_missing_age = df[df['Age'].isnull()]
print("Subset of data with missing 'Age' values:\n",
subset_missing_age)
```

Output:

```
Subset of data with missing 'Age' values:
      Id Gender  Age    Income      Height     Weight
9     10    NaN  NaN       NaN         NaN  95.598488
14    15      M  NaN  106906.0  167.097076        NaN
19    20    NaN  NaN   83835.0  181.556427  61.367387
23    24      F  NaN       NaN         NaN  55.019491
27    28      M  NaN   22846.0  180.921226        NaN
37    38    NaN  NaN   68004.0  174.146601  66.151129
40    41      F  NaN   37461.0  156.848852  67.866031
43    44      M  NaN       NaN         NaN  62.241544
68    69    NaN  NaN       NaN  155.452964  73.922483
71    72      F  NaN   54227.0         NaN  98.990119
81    82      M  NaN   70620.0  173.478439  78.767451
84    85      F  NaN  104426.0  151.080330  80.598941
88    89      F  NaN   21166.0  177.467332  84.983588
93    94      M  NaN  112669.0  172.706326        NaN
96    97      F  NaN   43471.0  168.845029  57.264576
101  102    NaN  NaN       NaN         NaN  85.259432
118  119      M  NaN  147516.0  181.526756  74.276041
138  139      F  NaN  136341.0  187.995264  89.762415
159  160    NaN  NaN   54632.0  171.519415  93.906116
165  166      F  NaN  127918.0  174.153554  83.129017
171  172      M  NaN   45818.0  176.131824  99.055606
174  175    NaN  NaN       NaN  150.250917  55.949382
193  194      F  NaN   94674.0  159.718209  97.889626
196  197      F  NaN  148849.0         NaN        NaN
211  212      F  NaN       NaN  182.823699  68.666456
229  230      F  NaN  138253.0  165.057028  62.671794
240  241      M  NaN  140514.0  166.611281  90.630989
263  264    NaN  NaN  122688.0         NaN  63.895171
271  272      F  NaN       NaN  189.056653  99.960711
278  279    NaN  NaN  148185.0  160.823516  51.765436
292  293      F  NaN  146086.0  156.831644  97.892778
297  298      M  NaN       NaN  150.978536  51.052895
318  319      M  NaN  147747.0  155.758379  71.654287
324  325      F  NaN   66508.0  189.478568  70.055598
```

```
339  340     NaN  NaN   97646.0         NaN        NaN
343  344       F  NaN   76697.0  165.657829  50.391670
350  351       F  NaN   70545.0  152.391342  80.172444
352  353       M  NaN  132435.0  153.969261  73.650252
355  356       M  NaN  149482.0  166.794464  57.413299
369  370       M  NaN  121887.0  156.509895        NaN
370  371       M  NaN  108699.0  170.655172  52.003007
387  388       F  NaN   41839.0  170.453294  81.332171
412  413     NaN  NaN   87652.0  172.084179  64.169075
417  418     NaN  NaN   47230.0  182.209902  64.264326
431  432       M  NaN       NaN  155.052951        NaN
441  442       M  NaN       NaN  187.211594  75.858597
459  460       F  NaN   82170.0  153.308410        NaN
476  477       M  NaN   89638.0         NaN  66.998268
477  478     NaN  NaN  123423.0  168.679799  81.676143
488  489       F  NaN   72385.0  182.203778  62.183321
```

Exercise 2.5

- Create a new DataFrame that only contains observations with gender values that are not female or male, and print it. Later, verify your result with the number of rows where gender is null.

2.9. Calculating Statistics from Pandas DataFrame

We can use Pandas DataFrame's built-in methods to quickly generate summary statistics for our data. Such as, we can use the '**describe()**' function to get summary statistics for numerical columns like count, mean, standard deviation, minimum, and maximum.

Code:

```
summary = df.describe()
print(summary)
```

Output:

```
               Id         Age       Income      Height      Weight
count  500.000000  450.000000   392.000000  449.000000  447.000000
```

```
mean    250.500000  41.286667   88457.959184  169.352801  75.161284
std     144.481833  13.699611   36435.434393   11.212903  15.044107
min       1.000000  18.000000   20137.000000  150.057661  50.094478
25%     125.750000  30.000000   59945.000000  159.718209  62.112599
50%     250.500000  41.000000   87346.500000  170.197485  74.276041
75%     375.250000  52.000000  119528.250000  178.421450  89.008912
max     500.000000  65.000000  149869.000000  189.873394  99.968650
```

If we want to calculate the standard deviation of a numerical column we can use '**std()**` function

Code:

```
age_std = df['Age'].std()
print("Age Standard Deviation:", age_std)
```

Output:

```
Age Standard Deviation: 13.699610645874296
```

> Exercise 2.6
>
> - What would be the output of the following command
>   - gender_counts = df['Gender'].value_counts()
>   print(gender_counts)

2.10. Groups in Pandas

Frequently, there's a need to compute summary statistics based on subsets or specific attributes within our dataset. For instance, we might wish to find the average income of all individuals using the following code.

Code:

```
df['Income'].describe()
```

Output:

```
count      392.000000
mean     88457.959184
std      36435.434393
```

```
min        20137.000000
25%        59945.000000
50%        87346.500000
75%       119528.250000
max       149869.000000
Name: Income, dtype: float64
```

Again, we might also want to get only specific information, like the maximum using the following code

Code:

```
df['Income'].max()
```

Output:

```
149869.0
```

However, when the intention is to summarize data based on one or more variables, such as gender, the Pandas library offers the **.groupby** method. Once a DataFrame is grouped using this approach, we have the ability to compute summary statistics of the selected grouping.

Code:

```
# Group data by sex
grouped_data = df.groupby('Gender')

# Provide the mean for each numeric column by sex
grouped_data.mean(numeric_only=True)
```

Output:

Gender	Id	Age	Income	Height	Weight
F	245.228070	40.894040	89354.985612	169.320899	73.967383
M	251.502959	40.940789	90492.552239	169.937040	76.359571

2.11. Basic Math with Pandas

If desired, it's entirely possible to perform mathematical operations, such as addition or division, on an entire column of our dataset. Such as, we will multiply the weight column by 2.

Code:

```
# Multiply all weight values by 2
df['Weight']*2
```

Output:

```
0      123.511907
1      168.178145
2             NaN
3      151.328751
4      155.116499
          ...
495    127.228658
496    160.777168
497    109.595666
498           NaN
499    102.276731
Name: Weight, Length: 500, dtype: float64
```

2.12. Concatenating DataFrames

Concatenating DataFrames refers to combining two or more DataFrames along a particular axis (either rows or columns) to create a single larger DataFrame. This is useful when we have data split across multiple DataFrames and we want to consolidate them into one for analysis or processing.

- Data Split Across Multiple Sources: Data might be collected and stored in different files or databases.
- Time-Series Data: Data collected at different time intervals might need to be combined.
- Data Transformation: You might transform data and want to concatenate it back together.

- Comparison and Analysis: Concatenation helps in comparing and analyzing data from different sources.

In Pandas, we can use the **'concat()'** function to concatenate DataFrames. This function provides various options to control how the concatenation should be performed. Let's say we have two DataFrames, **df1** and **df2**, and we want to concatenate them vertically (along rows):

Code:

```
import pandas as pd

# Sample DataFrames
data1 = {'A': [1, 2, 3], 'B': [4, 5, 6]}
data2 = {'A': [7, 8, 9], 'B': [10, 11, 12]}
df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Concatenate DataFrames vertically
concatenated_df = pd.concat([df1, df2], ignore_index=True)
print("Concatenated DataFrame:\n", concatenated_df)
```

Output:

```
Concatenated DataFrame:
    A   B
0  1   4
1  2   5
2  3   6
3  7  10
4  8  11
5  9  12
```

Note

- In this example, **pd.concat()** is used to concatenate df1 and df2 vertically into concatenated_df. The **ignore_index=True** argument ensures that the index is reset after concatenation

We can also concatenate DataFrames **horizontally** by specifying **axis=1** as an argument to pd.concat(). This will merge the DataFrames along columns.

Code:

```
import pandas as pd

# Sample DataFrames
data1 = {'A': [1, 2, 3], 'B': [4, 5, 6]}
data2 = {'C': [7, 8, 9], 'D': [10, 11, 12]}
df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Concatenate DataFrames horizontally
concatenated_df_horizontal = pd.concat([df1, df2], axis=1)
print("Horizontally Concatenated DataFrame:\n",
concatenated_df_horizontal)
```

Output:

```
Horizontally Concatenated DataFrame:
    A  B  C   D
0  1  4  7  10
1  2  5  8  11
2  3  6  9  12
```

Note

- In this example, pd.concat() is used with **axis=1** to concatenate df1 and df2 horizontally, creating the concatenated_df_horizontal. The resulting DataFrame has columns from both df1 and df2.

Exercise 2.7

- Consider two DataFrames, df1 and df2, with the following data

  import pandas as pd

data1 = {'A': [1, 2, 3], 'B': [4, 5, 6]}
data2 = {'A': [7, 8, 9], 'B': [10, 11, 12]}
df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

What will be the output of the following code

result = pd.concat([df1, df2], axis=1)
print(result)

Select the correct answer

a) The concatenated DataFrame with columns A, B, A, B
b) An error will occur because columns A and B are duplicated
c) The concatenated DataFrame with columns A, B, C, D

2.13. Saving Pandas DataFrame

We can save a Pandas DataFrame to various file formats using different methods provided by Pandas. Before we move forward with saving a pandas dataframe, let's first create a new directory called "Results" within the directory that contains your code.

Here are some commonly used methods to save a DataFrame:

- CSV Format

  To save a DataFrame to a CSV file, we can use the to_csv() method:

Code:

```
# Save DataFrame to CSV file
output_path = 'Results/output.csv'
df.to_csv(output_path, index=False)
```

This will save the DataFrame to a CSV file named 'output.csv' inside a directory called "Results", without including the index.

- Excel Format

  To save a DataFrame to an Excel file, we can use the to_excel() method:

Code:

```
# Save DataFrame to Excel file
output_path = 'Results/output.xlsx'
df.to_excel(output_path, index=False)
```

This will save the DataFrame to an Excel file named 'output.xlsx' inside a directory called "Results", without including the index.

- Other Formats
  Pandas supports various other formats, including JSON, Parquet, HDF5, and more. We can use the appropriate method based on the desired format:

  - JSON: df.to_json("output.json", orient="records")
  - Parquet: df.to_parquet("output.parquet")
  - HDF5: df.to_hdf("output.h5", key="data")

  Make sure to replace 'output' with your desired file name and extension.

Remember to adjust the options and parameters based on your specific needs. The index parameter can be adjusted to control whether the DataFrame index is included in the saved file. Additionally, some formats may offer additional options to fine-tune the saving process.

Note

It's important to note that Pandas is a powerful library with a wide range of functions designed to handle various data manipulation tasks. In our discussions, we covered a subset of these functions that were relevant to our topics and objectives. However, the Pandas library offers much more functionality that you can explore based on your needs.

Remember that each function in Pandas often comes with numerous arguments and parameters that allow you to customize its behavior to suit your specific requirements. The official Pandas documentation is an invaluable resource to learn about the complete set of functions, their options, and how to use them effectively.

When working with Pandas, don't hesitate to refer to the official documentation for detailed information about each function, examples, and best practices. This will enable you to leverage the full potential of the library and efficiently handle various data analysis and manipulation tasks.

3. Real-world data preparation

Data Cleaning and Preparation, as well as Exploratory Data Analysis (EDA), are fundamental steps in the data analysis process, playing a pivotal role in ensuring the reliability and effectiveness of any data-driven project. Data Cleaning involves identifying and rectifying errors, inconsistencies, and inaccuracies within the dataset. By eliminating missing values, outliers, and redundant information, data quality is enhanced, leading to more accurate and reliable insights.

EDA, on the other hand, involves delving into the data to understand its structure, distribution, patterns, and relationships. It allows analysts to identify trends, correlations, and potential variables of interest. EDA aids in formulating hypotheses and designing appropriate analytical approaches, ensuring that subsequent analyses are well-informed and meaningful.

In this session, participants will engage in hands-on practice to master the art of data cleaning and preparation, as well as EDA. Through practical examples and exercises, attendees will learn the techniques to identify and address data quality issues and uncover insights that can drive informed decision-making. By the end of the session, participants will be equipped with essential skills to handle real-world datasets and derive meaningful insights from them. This dataset serves as a preliminary exploration for our data preprocessing and EDA session. It has been randomly generated to simulate a diabetes classification scenario. Please note that this dataset is entirely synthetic and is not based on real medical data.

3.1. Dataset Description

The dataset contains information on various individuals, encompassing features commonly associated with diabetes diagnosis. These features include:

- ID: A unique identifier assigned to each individual.

- Age: The age of the individual.
- Gender: The gender of the individual.
- BMI: The Body Mass Index, which relates to weight and height.
- Glucose: Blood glucose level, a crucial indicator in diabetes diagnosis.
- Insulin: Insulin levels in the body.
- HbA1c: Hemoglobin A1c level, another vital parameter for diabetes assessment.
- FamilyHistory: Whether the individual has a family history of diabetes.
- Diabetes: The target variable indicates whether the individual has diabetes.

3.2. Characteristics of the Dataset

- Random Generation: This dataset has been randomly generated for educational purposes and does not reflect real patient data.
- Duplicates: To introduce variability, a small number of duplicate entries have been intentionally inserted.
- Outliers: Outliers have been introduced in the 'Glucose' and 'Insulin' columns to mimic potential data anomalies.
- Missing Values: A subset of the dataset contains missing values, as is common in real-world data.
- Inconsistency Values: A subset of the dataset contains inconsistency values, as common in real-life data.

Please note that the dataset's primary purpose is to explore and practice data cleaning, preparation, visualization, and analysis techniques.

Note

- If you feel interested, check the [Python script] that has been used to generate the simulated dataset.

Note

Please use the following links to access the necessary resources for the class:

- Dataset: Download the **[required dataset]**. This dataset is crucial for the completion of the main program. Save it in a new Directory called "Data" inside the directory that will contain your code.
- Main Program: Download the **[main program]** that requires completion. It's important to note that the program is currently incomplete, and your task is to finalize the code as part of your exercise.

In order to fulfill the requirements of the exercise, you are expected to complete the provided main program. Your completion should encompass the missing elements to make the program fully functional. It's imperative that the final version of the program accomplishes its intended objectives.

4. Data Cleaning and Preparation

In this part, we'll explore important ways to work with data. We'll discover how to handle data that's missing, deal with numbers that are outliers, manage duplicates, change how data is arranged, and make sure things are all in the same format. With practical activities, we'll learn how to do all this step by step, so we can feel confident working with different kinds of data on our own.

Exercise 4.1

- Once you downloaded the dataset and the main program, please open the main program, and solve question number 1.

4.1. Handling Missing Values

Missing values in a dataset can hinder analysis and modeling. Pandas provides functions to handle missing values, such as ***fillna()***, which allows us to fill ***NaN*** values with a specific value or method. Here's how to do it:

Code:

```
import pandas as pd
import numpy as np

# Creating a DataFrame with missing values
```

```python
data = {
    'A': [1, 2, np.nan, 4, 5],
    'B': [10, 20, 30, 40, 50]
}
df = pd.DataFrame(data)

# Before filling missing value
print("DataFrame before Filling Missing Values:\n",df)

# Filling missing values with 0
df_filled = df.fillna(0)

print("DataFrame with Missing Values Filled:\n", df_filled)
```

Output:

```
DataFrame before Filling Missing Values:
     A   B
0  1.0  10
1  2.0  20
2  NaN  30
3  4.0  40
4  5.0  50
DataFrame with Missing Values Filled:
     A   B
0  1.0  10
1  2.0  20
2  0.0  30
3  4.0  40
4  5.0  50
```

Note

- An alternative to using the ***fillna()*** function in Pandas for handling missing values is to impute or interpolate the missing values using other methods.
- Interpolation is the process of estimating missing values based on the values of other data points. Pandas provides several interpolation methods through the ***interpolate()*** function.

- Both ***fillna()*** and ***interpolation*** have their use cases. If you want a simple and explicit way to fill missing values with a constant, ***fillna()*** is a good choice. On the other hand, if preserving data distribution and context are important, and you're comfortable with more advanced techniques, interpolation can provide more accurate results. The choice depends on the nature of your data and your analysis goals.

Exercise 4.2

- Please go to the main program you downloaded, and solve question number 2.

4.2. Handling Outliers

Outliers are extreme values that can skew analysis and modeling results. Pandas can help us identify and handle outliers. In this example, we identify outliers using the ***interquartile range (IQR)*** method and remove them:

Code:

```
import pandas as pd

# Creating a DataFrame with outliers
data = {
    'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 200, 50]
}
df = pd.DataFrame(data)

# Main Dataframe with Outliers
print("DataFrame with Outliers:\n", df)

# Identifying and handling outliers
q1 = df['B'].quantile(0.25)
q3 = df['B'].quantile(0.75)
iqr = q3 - q1
lower_bound = q1 - 1.5 * iqr
upper_bound = q3 + 1.5 * iqr
df_no_outliers = df[(df['B'] >= lower_bound) & (df['B'] <=
```

```
upper_bound)]

print("DataFrame with Outliers Removed:\n", df_no_outliers)
```

Output:

```
DataFrame with Outliers:
    A    B
0  1   10
1  2   20
2  3   30
3  4  200
4  5   50
DataFrame with Outliers Removed:
    A   B
0  1  10
1  2  20
2  3  30
4  5  50
```

Code Explanation

- ***q1*** and ***q3*** are calculated using the ***quantile()*** function, representing the first and third quartiles of column 'B'.
- ***q1*** represents the value below which 25% of the data lies. For column 'B', ***q1*** would be the median of the first half of the sorted values, which is 15.
- ***q3*** represents the value below which 75% of the data lies. For column 'B', ***q3*** would be the median of the second half of the sorted values, which is 50.
- ***iqr*** (Interquartile Range) is computed as the difference between ***q3*** and ***q1***.
- ***lower_bound*** and ***upper_bound*** are calculated to define the thresholds beyond which data points are considered outliers. These bounds are defined as 1.5 times the IQR below q1 and above q3.
- The line ***df_no_outliers = df[(df['B'] >= lower_bound) & (df['B'] <= upper_bound)]*** filters the DataFrame to keep only the rows where the values in column 'B' fall within the acceptable range, effectively removing the outliers.

Note

- The choice of method depends on the nature of your data and your specific goals. Always carefully consider the implications of outlier handling methods and their impact on your analysis or modeling. Other methods to handle outliers:
    - Z-Score Method
    - Winsorizing

Exercise 4.3

- Please go to the main program you downloaded, and solve question number 3.

4.3. Dealing with Duplicate Data

Duplicate data can lead to misleading analysis. Pandas provides functions to detect and remove duplicate rows. Here's how we can do it:

Code:

```
import pandas as pd

# Creating a DataFrame with duplicate data
data = {
    'A': [1, 2, 2, 3, 4, 4],
    'B': [10, 20, 20, 30, 40, 40]
}
duplicate_df = pd.DataFrame(data)

# Main DataFrame with duplicate data
print("DataFrame with Duplicates:\n", duplicate_df)

# Detecting and removing duplicated rows
duplicated_rows = duplicate_df[duplicate_df.duplicated()]

#Detecting and removing duplicated rows but keeping the first
duplicate
deduplicated_df = duplicate_df.drop_duplicates(keep = ‘first’)

print("Duplicated Rows:\n", duplicated_rows)
```

```
print("DataFrame after Dropping Duplicates:\n", deduplicated_df)
```

Output:

```
DataFrame with Duplicates:
   A   B
0  1  10
1  2  20
2  2  20
3  3  30
4  4  40
5  4  40
Duplicated Rows:
   A   B
2  2  20
5  4  40
DataFrame after Dropping Duplicates:
   A   B
0  1  10
1  2  20
3  3  30
4  4  40
```

Exercise 4.4

- Please go to the main program you downloaded, and solve question number 4.

4.4. Data Reshaping

Reshaping data is the process of transforming data from one format to another. In the context of data analysis and machine learning (ML), reshaping data often involves reorganizing it into a different structure that is better suited for analysis, visualization, or modeling. Reshaping can involve tasks such as pivoting, melting, stacking, unstacking, and more.

4.4.1. Wide to Long Format (Melting)

In this transformation, we convert a dataset from a wide format (many columns) to a long format (fewer columns) by melting or unpivoting it. This is useful when we have variables stored as columns and we want to gather them into a single column.

Melting data is useful for making it more suitable for analysis, especially when we want to compare or aggregate across different variables.

Code:

```
import pandas as pd

# Creating a Wide DataFrame
data = {
    'ID': [1, 2, 3],
    'Math': [90, 85, 78],
    'Science': [75, 88, 92]
}
df = pd.DataFrame(data)

# Main Wide DataFrame
print("Original Wide DataFrame:\n", df)

# Melting the DataFrame
df_long = pd.melt(df, id_vars=['ID'], value_vars=['Math',
'Science'], var_name='Subject', value_name='Score')

# After Melting the DataFrame
print("Long Format DataFrame:\n", df_long)
```

Output:

```
Original Wide DataFrame:
    ID  Math  Science
0   1    90       75
1   2    85       88
2   3    78       92
Long Format DataFrame:
    ID  Subject  Score
0   1     Math     90
1   2     Math     85
```

```
2   3      Math      78
3   1   Science      75
4   2   Science      88
5   3   Science      92
```

Code Explanation

- The ***pd.melt()*** function is used to transform the ***df*** DataFrame from wide format to long format.
- ***id_vars=['ID']*** specifies that the 'ID' column should be kept as an identifier for each observation.
- ***value_vars=['Math', 'Science']*** specifies the columns ('Math' and 'Science') whose values will be "melted" or transformed into a single column.
- ***var_name='Subject'*** specifies the name of the new column that will store the subject names ('Math' and 'Science').
- ***value_name='Score'*** specifies the name of the new column that will store the scores for each subject.

Note

- In this new format, each row represents a single observation (student's score in a subject), and the 'Subject' column indicates whether the score is for 'Math' or 'Science'. The 'ID' column uniquely identifies each student, and the 'Score' column contains the respective scores.
- The purpose of this transformation is to make it easier to work with and analyze the data, especially when you want to perform operations or calculations that involve multiple subjects. It's a common technique used in data analysis and preparation.

4.4.2. Long to Wide Format (Pivoting)

This transformation involves converting a long-format dataset back into a wide format by pivoting or spreading the values.

Pivoting is useful when we want to reshape data to make it easier to visualize or perform calculations on.

<u>Code:</u>

```
import pandas as pd
```

```python
# Creating a Long DataFrame
data = {
    'ID': [1, 1, 2, 2],
    'Subject': ['Math', 'Science', 'Math', 'Science'],
    'Score': [90, 75, 85, 88]
}
df_long = pd.DataFrame(data)

# Main Long DataFrame
print("Original Long DataFrame:\n", df_long)

# Pivoting the DataFrame
df_wide = df_long.pivot(index='ID', columns='Subject',
values='Score')
print("Wide Format DataFrame:\n", df_wide)
```

<u>Output:</u>

```
Original Long DataFrame:
    ID  Subject  Score
0   1      Math     90
1   1   Science     75
2   2      Math     85
3   2   Science     88
Wide Format DataFrame:
 Subject  Math  Science
ID
1           90       75
2           85       88
```

Code Explanation

- The ***df_long.pivot()*** function is used to transform the ***df_long*** DataFrame from a long format to a wide format.
- ***index='ID'*** specifies that the 'ID' column will be the index of the resulting pivoted DataFrame.
- ***columns='Subject'*** specifies that the unique values in the 'Subject' column will become the column headers of the pivoted DataFrame.

- ***values='Score'*** specifies that the values in the 'Score' column will be placed in the corresponding cells of the pivoted DataFrame.

Note

- In this new format, each row represents a unique 'ID', and each column represents a unique 'Subject'. The cell values represent the corresponding scores for each 'ID' and 'Subject' combination.
- The purpose of this transformation is to organize the data in a way that makes it easier to compare and analyze scores across different subjects for each student. It's particularly useful when you want to perform operations or calculations that involve comparing data across categories (subjects in this case).

4.4.3. Stacking and Unstacking

Stacking involves converting columns into rows, and unstacking is the reverse process. These operations can be useful for creating hierarchical indexes and dealing with multi-level data.

Stacking and unstacking can make data manipulation and analysis easier when dealing with multi-indexed data.

Code:

```
import pandas as pd

# Creating a DataFrame
data = {
    'ID': [1, 2],
    'Math': [90, 85],
    'Science': [75, 88]
}
df = pd.DataFrame(data)

# Original DataFrame
print("Original DataFrame:\n", df)

# Set the DataFrame index using the ID column
df.set_index('ID', inplace=True)
```

```
# Doing Stacking and Unstacking
stacked_df = df.stack()
unstacked_df = stacked_df.unstack()
print("Stacked DataFrame:\n", stacked_df)
print("Unstacked DataFrame:\n", unstacked_df)
```

Output:

```
Original DataFrame:
    ID  Math  Science
0   1    90       75
1   2    85       88
Stacked DataFrame:
 ID
1   Math       90
    Science    75
2   Math       85
    Science    88
dtype: int64
Unstacked DataFrame:
     Math  Science
ID
1     90       75
2     85       88
```

Note

Reshaping data is important in data analysis and ML because

- Analysis Requirements: Different analyses might require data to be organized in a specific way. Reshaping allows you to transform your data to meet the requirements of various analyses.
- Visualization: Certain visualization techniques work better with data in specific formats. Reshaping data can help create visualizations that are more informative and easier to understand.
- Feature Engineering: In ML, data structure can impact the performance of models. Reshaping can help create new features or representations that improve model performance.

- Algorithms and Models: Some algorithms and models might assume or perform better with certain data structures. Reshaping ensures that your data is compatible with these algorithms.

4.5. Handling Inconsistent Data and Standardizing

Handling inconsistent data is a crucial step in data preprocessing to ensure the accuracy and reliability of our analysis or modeling. Inconsistent data refers to values that do not adhere to the expected format or constraints. This can include typos, varying representations, or unexpected values in categorical variables.

Methods for Handling Inconsistent Data

- Standardization: Convert data to a consistent format. For example, converting all text to lowercase, removing leading/trailing spaces, or converting categorical values to a common category naming convention.
- Regular Expressions: Use regular expressions to identify and correct patterns within strings.
- Categorical Mapping: Create a mapping of inconsistent categories to a standardized set of categories.
- Data Validation Rules: Apply rules to validate data entries against predefined criteria.
- Domain Knowledge: Leverage domain knowledge to identify and correct inconsistencies.

Why Handling Inconsistent Data is Important

- Accuracy: Inconsistent data can lead to erroneous conclusions or inaccurate predictions when performing analysis or building models.
- Data Quality: Inaccurate data can negatively impact the quality of any downstream processes that rely on that data.
- Comparability: Inconsistent data makes it difficult to compare or combine datasets, leading to potential errors in merging and analysis.

Suppose we have a dataset with a "Gender" column that contains variations of the categories "Male" and "Female". To handle inconsistencies, we can standardize the values.

Code:

```
import pandas as pd

# Creating a DataFrame
data = {
    'ID': [1, 2, 3, 4],
    'Gender': ['Male', 'female', 'Male', 'femAle']
}
df = pd.DataFrame(data)

# Original DataFrame
print("Original DataFrame:\n", df)

# Convert gender values to lowercase and standardize
df['Gender'] =
df['Gender'].str.lower().str.strip().replace({'male': 'Male',
'female': 'Female'})

print("DataFrame with Consistent Gender Values:\n", df)
```

Output:

```
Original DataFrame:
    ID  Gender
0   1     Male
1   2   female
2   3     Male
3   4   Female
DataFrame with Consistent Gender Values:
    ID  Gender
0   1     Male
1   2   Female
2   3     Male
3   4   Female
```

Code Explanation

- ***df['Gender']*** selects the 'Gender' column from the DataFrame.
- ***.str.lower()*** is a string method that converts all the values in the 'Gender' column to lowercase. This ensures that all variations of 'male' and 'female' are in lowercase, making the replacement consistent.
- ***.replace({'male': 'Male', 'female': 'Female'})*** is used to replace specific values in the 'Gender' column. Here, it's specified that the value 'male' should be replaced with 'Male', the value 'female' should be replaced with 'Female', and the value 'other' should be replaced with 'Other'.
    - This replacement is case-insensitive due to the prior conversion to lowercase. For instance, 'Male' and 'male' will both be converted to 'Male'.
- **.str.strip()** helps in removing leading and trailing whitespaces from a string. When dealing with textual data in Python, especially from external sources like files or user input, it's common to encounter unwanted leading or trailing whitespaces. These spaces might seem harmless, but they can significantly impact data analysis, leading to inconsistencies and errors.
- The updated 'Gender' column, after performing the lowercase conversion and replacements, is assigned back to the original 'Gender' column in the DataFrame. This effectively updates the values in the DataFrame.

Now, suppose we have a dataset with a "Color" column that contains various color names, including some inconsistent spellings and synonyms. We want to standardize these color names.

Code:

```
import pandas as pd

# Creating a DataFrame
data = {
    'ID': [1, 2, 3, 4, 5],
    'Color': ['red', 'green', 'blue', 'green', 'Reddish']
}
df = pd.DataFrame(data)

# Original DataFrame
print("Original DataFrame:\n", df)
```

```python
# Define a mapping for inconsistent color names to standard names
color_mapping = {
    'red': 'Red',
    'green': 'Green',
    'blue': 'Blue',
    'reddish': 'Red'  # Handling a synonym
}

# Apply the mapping to the Color column
df['Color'] = df['Color'].str.lower().map(color_mapping)

print("DataFrame with Consistent Color Names:\n", df)
```

Output:

```
Original DataFrame:
   ID    Color
0  1       red
1  2     green
2  3      blue
3  4     green
4  5   Reddish
DataFrame with Consistent Color Names:
   ID  Color
0  1    Red
1  2  Green
2  3   Blue
3  4  Green
4  5    Red
```

Key Points

- Libraries need to be imported using the **import** statement.
- We can use aliases for libraries using **as** a keyword.

- Functions from libraries are accessed using the dot notation **LibraryName.FunctionName**.
- Importing libraries with an alias allows you to use the alias to call functions **AliasName.FunctionName**.
- Use pd.**read_csv**("filename.csv") to read CSV files and create DataFrames.
- Access column names using the columns attribute of the DataFrame.
- Slicing rows and columns can be done using **.iloc** and **.loc**.
- .iloc uses integer indices, while .loc uses label indices.
- Use pd.**concat**() to concatenate DataFrames along rows or columns.
- Specify **axis=0** for vertical concatenation (rows) and **axis=1** for horizontal concatenation (columns).
- Use conditional statements to filter rows based on specific criteria using DataFrame column values.
- We can use operators like >, <, ==, & (AND), | (OR), and ~ (NOT).
- **.isin()** filters data based on values present in a list or iterable.
- **.isnull()** and **.notnull()** detect missing (**NaN**) values in DataFrames.
- **fillna()**, allows filling **NaN** values with a specific value or method.
- Identify outliers using the **interquartile range (IQR)**.
- **drop_duplicates()** function can be used to remove duplicates from a dataframe.
- **melt()** function is used to transform a dataframe from a wide format to a long format.
- **pivot()** function can be used to transform a dataframe from a long format to a wide format.
- **.str.lower()** is a string method that converts all values in a column to lowercase.

Further Resources

Pandas by Official Documentation
Data Analysis and Visualization in Python for Ecologists by Data Carpentry
Pandas Tutorial by W3School
Mastering Data Cleaning & Data Preprocessing by Encord
Exploratory Data Analysis with an Example by Analyticsvidhya

Disclaimer: Some content of this lesson has been inspired and adapted from Capentaries Data Analysis and Visualization in Python for Ecologists.