Pandas

Notes by Mannan Ul Haq (BDS-3C)

Introduction to Pandas Library in Python

Pandas is a popular Python library for data manipulation and analysis. It provides easy-to-use data structures and functions for working with structured data, making it an essential tool for data scientists, analysts, and anyone working with tabular data.

What is Pandas?

Pandas stands for **"Panel Data"** and is built on top of the NumPy library. It introduces two primary data structures:

1. **Series**: A one-dimensional labeled array capable of holding any data type. It's like a single column from a DataFrame.
2. **DataFrame**: A two-dimensional, size-mutable, and heterogeneous tabular data structure with labeled axes (rows and columns). Think of it as a spreadsheet in Python.

Usage of Pandas:

Pandas is incredibly versatile and can help you with a wide range of tasks:

- **Data Cleaning**: Pandas makes it easy to handle missing data, duplicate records, and outliers. You can filter, fill, or drop data as needed.
- **Data Exploration**: You can quickly summarize and explore your data by calculating statistics, plotting charts, and aggregating information.
- **Data Transformation**: Transforming data for analysis or machine learning is a breeze with Pandas. You can reshape, pivot, or merge datasets effortlessly.
- **Data Analysis**: Perform data analysis tasks like grouping, sorting, and filtering data to gain insights from your dataset.

Topic Outline for Learning Basics of Pandas:

1. **Series in Pandas**
2. **DataFrames in Pandas**
3. **Pandas Operations on DataFrames**
4. **Pandas with Functions**
5. **Pandas with CSV and HTML Data**

1. Series in Pandas

Pandas, a powerful data manipulation library in Python, offers a fundamental data structure known as a **Series**. Think of a Series as a versatile container that combines data values with labels (or indices) for easy access and manipulation. This is especially handy when you want to work with one-dimensional data.

Let's dive into the key aspects of Series with detailed examples:

1. Creating a Series

You can create a Series in Pandas by passing a sequence of data (like a Python list) to the `Series()` constructor. Optionally, you can specify an index for the Series. If you don't provide an index, Pandas will generate a default integer-based index.

Example 1: Creating a Simple Series

```
import pandas as pd

# Creating a Series without specifying an index
data = [4, 7, -5, 3]
obj = pd.Series(data)

print(obj)
```

Output:

```
0    4
1    7
2   -5
3    3
dtype: int64
```

Example 2: Creating a Series with Custom Index

```
data = [4, 7, -5, 3]
custom_index = ['d', 'b', 'a', 'c']

# Creating a Series with specifying an index
obj2 = pd.Series(data, index=custom_index)

print(obj2)
```

Output:

```
d    4
b    7
a   -5
c    3
dtype: int64
```

2. Accessing Data in a Series

You can access data in a Series using either the index label or the position.

Example: Accessing Data by Index Label

```
print(obj2['a'])  # Accessing data with label 'a'
```

Output:

```
-5
```

Example: Accessing Data by Position

```
print(obj2[1])  # Accessing data at position 1 (zero-based)
```

Output:

```
7
```

In Pandas, index labels are not limited to integers or strings; they can be any data type, including numbers, strings, or even dates.

3. Operations Preserve Index

When you perform operations on a Series, the association between index labels and data values is preserved.

Example: Applying Operations on a Series

```
# Doubling the values in obj3
doubled_obj = obj2 * 2

print(doubled_obj)
```

Output:

```
d    8
b    17
a   -10
c    6
dtype: int64
```

4. Naming Your Series

Both the Series itself and its index can have names, which can be useful for documentation and organization.

Example: Naming a Series and Its Index

```
obj4 = pd.Series([10, 20, 30, 40], name='sample_data')
obj4.index.name = 'observation'

print(obj4)
```

Output:

```
observation
0    10
1    20
2    30
3    40
Name: sample_data, dtype: int64
```

2. DataFrames in Pandas

A **DataFrame** in pandas represents a versatile and powerful data structure that resembles a rectangular table of data. It is a fundamental component for data manipulation and analysis. In this guide, we'll explore DataFrames in depth, covering their characteristics, creation, data access, modification, and more.

Characteristics of DataFrames

A DataFrame possesses several key characteristics:

- **Rectangular Structure**: DataFrames are organized into rows and columns, creating a grid-like structure.
- **Row and Column Indexes**: A DataFrame includes both row and column indexes. These indexes allow for efficient data retrieval and manipulation. You can think of a DataFrame as a collection of Series, all sharing the same index.

Creating a DataFrame

You can create a DataFrame from a dictionary of equal-length lists or NumPy arrays. Here's an example:

```
import pandas as pd

data = {
    "state": ["Ohio", "Ohio", "Ohio", "Nevada", "Nevada", "Nevada"],
    "year": [2000, 2001, 2002, 2001, 2002, 2003],
    "pop": [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]
}

frame = pd.DataFrame(data)
print(frame)
```

This code creates a DataFrame called `frame` with the given data. The resulting DataFrame will look like this:

```
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
5  Nevada  2003  3.2
```

We can also specify a custom index:

```
# Specify a custom index using the 'index' parameter
custom_index = ["a", "b", "c", "d", "e", "f"]

frame2 = pd.DataFrame(data, index=custom_index)
print(frame2)
```

The output will look like this:

```
    state  year  pop
a    Ohio  2000  1.5
b    Ohio  2001  1.7
c    Ohio  2002  3.6
d  Nevada  2001  2.4
e  Nevada  2002  2.9
f  Nevada  2003  3.2
```

Accessing Data in a DataFrame

You can access data within a DataFrame using various methods:

Accessing by Column Name:

```
state_column = frame["state"]
print(state_column)
```

This code retrieves the **"state"** column as a Series. Similarly, you can access other columns by their names.

```
# Output:
0      Ohio
1      Ohio
2      Ohio
3    Nevada
4    Nevada
5    Nevada
Name: state, dtype: object
```

Accessing by Row and Column:

```
row = frame.loc[1]  # Get the second row (index 1)
print(row)
print('\n')
specific_row = frame.loc[1,["state"]]  # Get the specified second row of state column
print(specific_row)
```

This code uses `.loc` to access a specific row by its label (index). You can replace `1` with any row label you want.

```
# Output:
state    Ohio
year     2001
pop       1.7
Name: 1, dtype: object

state    Ohio
Name: 0, dtype: object
```

Slicing:

```
subset = frame.loc[2:4, ["state", "pop"]]
print(subset)
```

This code slices the DataFrame:

- `2:4` specifies the row slice. It includes rows with indices 2, 3, and 4 (inclusive). This is a range-based slicing of rows.

- `["state", "pop"]` specifies the columns you want to include in the subset. You're selecting only the **"state"** and **"pop"** columns.

```
state  pop
2    Ohio  3.6
3  Nevada  2.4
4  Nevada  2.9
```

Modifying DataFrames

DataFrames are mutable, and you can modify them in various ways:

Adding New Columns:

```
frame["debt"] = 16.5  # Add a new column "debt" with a constant value
print(frame)
```

This code adds a new column **"debt"** with the value 16.5 for all rows.

Deleting Columns:

```
del frame["debt"]  # Delete the "debt" column
print(frame)
```

This code removes the **"debt"** column from the DataFrame.

3. Pandas Operations on DataFrames:

Pandas is a popular data manipulation and analysis library in Python that provides versatile data structures and functions for working with structured data. In this response, we'll explore some of the key operations you can perform with pandas:

1. Selecting Columns:

You can access specific columns in a DataFrame using square brackets or the dot notation.

```
import pandas as pd

# Creating a DataFrame from a dictionary
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [25, 30, 35],
    "City": ["New York", "San Francisco", "Los Angeles"]
}
df = pd.DataFrame(data)

names = df["Name"]
ages = df.Age
```

2. Filtering Data:

You can filter rows based on specific conditions using boolean indexing.

```
adults = df[df["Age"] >= 30]
print(adults)
```

```
# Output:
Name  Age           City
1      Bob   30  San Francisco
2  Charlie   35    Los Angeles
```

3. Sorting Data:

Use the `sort_values()` method to sort the DataFrame based on one or more columns.

```
sorted_df = df.sort_values(by="Age", ascending = False)
```

In the code above, we are sorting the DataFrame `df` by the **"Age"** column in descending order (`ascending=False`).

Sorting a DataFrame

- `by`: This parameter specifies the column or columns by which you want to sort the DataFrame. It can accept either a single column name as a string or a list of column

names for multi-level sorting. If you provide multiple columns, pandas will sort first by the first column, then by the second column, and so on.

- `ascending`: By default, this parameter is set to `True`, which sorts the data in ascending order. If you want to sort in descending order, set it to `False`.

```
# Sorting by multiple columns
sorted_df = df.sort_values(by=["City", "Age"], ascending=[True, False])
```

Sorting Index

By default, when you sort a DataFrame using `sort_values()`, the index labels remain unchanged. If you want to sort by the index, you can use `sort_index()` instead.

```
sorted_by_index = df.sort_index(ascending=False)
```

In-Place Sorting

By default, `sort_values()` returns a new DataFrame with the sorted data, leaving the original DataFrame unchanged. If you want to sort the DataFrame in-place (i.e., modify the original DataFrame), you can use the `inplace=True` parameter.

```
df.sort_values(by="Age", ascending=False, inplace=True)
```

4. Adding and Dropping Columns:

In pandas, you can easily add and drop columns from a DataFrame, which allows you to manipulate and tailor your data as needed.

Adding Columns

To add a new column to a DataFrame, you can simply assign a new column of data to it. You can create a new column from a Python list or a NumPy array. Here's how to add columns:

```
import pandas as pd

# Sample DataFrame
data = {
```

```
    "Name": ["Alice", "Bob", "Charlie", "David", "Eve"],
    "Age": [25, 30, 22, 35, 28]
}

df = pd.DataFrame(data)

# Adding a new column
df["City"] = ["New York", "San Francisco", "Los Angeles", "Chicago", "Boston"]
```

In the code above, we added a new column **"City"** to the DataFrame `df` with values provided as a Python list.

Dropping Columns

To drop one or more columns from a DataFrame, you can use the `drop()` method. Specify the column name(s) or label(s) you want to drop and set `axis=1` to indicate that you're dropping columns (as opposed to rows).

```
# Dropping a single column
df = df.drop("City", axis=1)
```

To drop multiple columns, provide a list of column names:

```
# Dropping multiple columns
columns_to_drop = ["Age", "Birth Year"]
df = df.drop(columns=columns_to_drop, axis=1)
```

Alternatively, you can use the `del` statement to remove a column in-place:

```
# In-place column deletion
del df["City"]
```

When you drop columns, pandas returns a new DataFrame with the specified columns removed. If you want to modify the original DataFrame in-place, you can use the `inplace=True` parameter:

```
# Drop "Age" column in-place
df.drop("Age", axis=1, inplace=True)
```

Keep in mind that the `drop()` method does not modify the DataFrame in-place by default; it returns a new DataFrame with the specified columns dropped.

5. Handling Missing Data:

- `isnull()` checks if each element in a DataFrame or Series is null (missing).
- It returns a DataFrame or Series of Boolean values (`True` if null, `False` otherwise).

```
import pandas as pd

data = {'A': [1, 2, None, 4],
        'B': [None, 5, 6, 7]}
df = pd.DataFrame(data)

# Check for null values in the DataFrame
null_mask = df.isnull()
print(null_mask)
```

Output:

```
       A      B
0  False   True
1  False  False
2   True  False
3  False  False
```

- `isnull().sum()` calculates the total number of missing values in each column of a DataFrame.
- It returns a Series with the count of missing values per column.

```
# Calculate the total number of null values in each column
null_counts = df.isnull().sum()
print(null_counts)
```

Output:

```
A    1
B    1
```

```
dtype: int64
```

In this example, 'A' has 1 missing value, and 'B' has 1 missing value.

Pandas provides methods for dealing with missing values. You can use `dropna()` to remove rows with missing values and `fillna()` to fill missing values.

```
# DataFrame.dropna(axis = 0, subset = None, inplace = True)

df.dropna(axis = 0)  # Removes rows with missing values
df.dropna(subset = ["Age"])  # Drop rows with null values in the 'Age' column

# df["specific_column_name"].fillna(fill_value, inplace = True)
df.fillna(0)  # Fills missing values with 0

# To Drop Specific Values
# DataFrame.drop(axis = 0, index = None, columns = None, inplace = False)
```

6. Grouping and Aggregating Data:

In pandas, you can perform grouping and aggregating operations on your DataFrame to summarize and analyze your data effectively. Grouping allows you to group rows of data based on one or more columns, and aggregation functions help you compute summary statistics for each group. Let's dive into the details of grouping and aggregating data in pandas.

Grouping Data

To group your data, you can use the `groupby()` method in pandas. This method allows you to specify one or more columns by which you want to group your data. Here's how you can group data by a single column:

```
import pandas as pd

# Sample DataFrame
data = {
    "Category": ["A", "B", "A", "B", "A", "B"],
    "Value": [10, 15, 12, 18, 14, 20]
}

df = pd.DataFrame(data)
```

```
# Grouping by a single column
grouped = df.groupby("Category")
```

In this example, we grouped the DataFrame `df` by the **"Category"** column. Now, we can perform aggregation operations on each group.

Aggregating Data

Once you've grouped your data, you can apply various aggregation functions to compute summary statistics for each group. Some common aggregation functions include `sum()`, `mean()`, `median()`, `count()`, `max()`, `min()`, and more. Here are some examples:

```
# Calculating the sum for each group
sums = grouped["Value"].sum()

# Calculating the mean for each group
means = grouped["Value"].mean()

# Counting the number of elements in each group
counts = grouped["Value"].count()

print(sums)
print(means)
print(counts)
```

```
# Output:
Category
A    36
B    53
Name: Value, dtype: int64

Category
A    12.000000
B    17.666667
Name: Value, dtype: float64

Category
A    3
B    3
Name: Value, dtype: int64
```

You can also apply multiple aggregation functions simultaneously using the `agg()` method:

```
# Applying multiple aggregation functions
result = grouped["Value"].agg(["sum", "mean", "count"])
```

Grouping by Multiple Columns

You can also group data by multiple columns by passing a list of column names to the `groupby()` method:

```
# Grouping by multiple columns
grouped = df.groupby(["Category", "Subcategory"])
```

7. Merging and Joining Data:

Merging and joining data in pandas involves combining multiple DataFrames based on common columns or indices. These operations are essential when you have data distributed across different tables that you want to consolidate for analysis. In pandas, you can perform merging and joining using various methods and options.

Types of Joins:

1. **Inner Join (default):** An inner join returns only the rows that have matching values in both DataFrames. It retains only the common keys.
2. **Outer Join:** An outer join returns all rows from both DataFrames, filling in missing values with NaN where there is no match.
3. **Left Join:** A left join returns all rows from the left DataFrame and the matching rows from the right DataFrame. Unmatched rows from the left DataFrame are filled with NaN.
4. **Right Join:** A right join is similar to a left join but returns all rows from the right DataFrame and the matching rows from the left DataFrame.

Merging DataFrames:

The `merge()` function in pandas is used for merging DataFrames. Here's how you can use it:

```
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
                    'value': [1, 2, 3, 4]})

df2 = pd.DataFrame({'key': ['B', 'D', 'E', 'F'],
                    'value': [5, 6, 7, 8]})

# Inner join (default)
inner_merged = pd.merge(df1, df2, on='key')

# Outer join
outer_merged = pd.merge(df1, df2, on='key', how='outer')

# Left join
left_merged = pd.merge(df1, df2, on='key', how='left')

# Right join
right_merged = pd.merge(df1, df2, on='key', how='right')

print(inner_merged)
print(outer_merged)
print(left_merged)
print(right_merged)
```

In the code above, `on='key'` specifies the common column to merge on. You can use multiple columns by passing a list to the `on` parameter.

```
# Output:
key  value_x  value_y
0   B        2        5
1   D        4        6

key  value_x  value_y
0   A      1.0      NaN
1   B      2.0      5.0
2   C      3.0      NaN
3   D      4.0      6.0
4   E      NaN      7.0
5   F      NaN      8.0

key  value_x  value_y
0   A        1      NaN
1   B        2      5.0
2   C        3      NaN
3   D        4      6.0
```

```
key  value_x  value_y
0   B      2.0        5
1   D      4.0        6
2   E      NaN        7
3   F      NaN        8
```

Joining DataFrames:

The `join()` method is another way to combine DataFrames. It is more convenient when you want to join DataFrames based on their indices. Here's how you can use it:

```
import pandas as pd

# Sample DataFrames with labeled indices
df1 = pd.DataFrame({'value1': [1, 2, 3]}, index = ['A', 'B', 'C'])
df2 = pd.DataFrame({'value2': [4, 5, 6]}, index = ['B', 'C', 'D'])

# Inner join (default)
inner_joined = df1.join(df2, how='inner')

# Outer join
outer_joined = df1.join(df2, how='outer')

print(inner_joined)
print(outer_joined)
```

In this example, `how` specifies the type of join, which can be **'inner'** (default) or **'outer'**.

```
# Output:
   value1  value2
B       2       4
C       3       5

   value1  value2
A     1.0     NaN
B     2.0     4.0
C     3.0     5.0
D     NaN     6.0
```

8. Concatenation of Data:

Concatenation in pandas involves combining DataFrames along a particular axis (either rows or columns). Pandas provides the `concat()` function for this purpose.

Here's how you can use the `concat()` function to concatenate DataFrames:

```
import pandas as pd

# Sample DataFrames
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3']})

df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7']})

# Concatenating along rows (axis=0)
result = pd.concat([df1, df2])
```

In this example, we concatenate `df1` and `df2` along rows (axis=0), effectively stacking `df2` below `df1`.

You can also concatenate along columns (axis=1):

```
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3']})

df2 = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']})

# Concatenating along columns (axis=1)
result = pd.concat([df1, df2], axis=1)
```

In this case, `df2` is concatenated next to `df1` along columns. Make sure the index values match when concatenating along columns, or you may end up with misaligned data.

9. Analyzing DataFrames:

Analyzing data in a DataFrame often involves examining the beginning and end of the DataFrame to get a quick overview of its structure and content. Two commonly used methods for this purpose in pandas are `head()` and `tail()`.

1. `head()`

The `head()` method in pandas allows you to view the first few rows of a DataFrame. It's useful for quickly inspecting the beginning of the DataFrame and understanding its

structure. By default, `head()` displays the first 5 rows, but you can specify the number of rows to display as an argument.

Syntax:

```
DataFrame.head(n=5)
```

- `n` (optional): The number of rows to display. By default, it's set to 5.

Example:

```
import pandas as pd

# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
        'Age': [25, 30, 35, 40, 45]}

df = pd.DataFrame(data)

# Display the first 3 rows
print(df.head(3))
```

Output:

```
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
```

2. `tail()`

The `tail()` method, similar to `head()`, allows you to view the last few rows of a DataFrame. It's helpful for examining the end of the DataFrame, especially when dealing with large datasets. By default, `tail()` displays the last 5 rows, but you can specify the number of rows to display as an argument.

Syntax:

```
DataFrame.tail(n=5)
```

- `n` (optional): The number of rows to display. By default, it's set to 5.

Example:

```
import pandas as pd

# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
        'Age': [25, 30, 35, 40, 45]}

df = pd.DataFrame(data)

# Display the last 2 rows
print(df.tail(2))
```

Output:

```
    Name  Age
3  David   40
4    Eve   45
```

10. Reset Indexing:

In pandas, the `reset_index()` function is used to reset the index. This operation is often necessary when you perform data manipulations that leave the DataFrame's index in a state that is not suitable for further analysis, and you want to revert to the default integer-based index.

Here's how to use `reset_index()`:

```
import pandas as pd

# Create a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 40]}

df = pd.DataFrame(data)

# Set a custom index
df.set_index('Name', inplace=True)
```

```
# Reset the index
df = df.reset_index()
```

In this example, we first set the **'Name'** column as the index using `set_index()`. Later, we use `reset_index()` to revert to the default integer-based index.

11. Getting Info of DataFrame:

The `info()` method provides a concise summary of a DataFrame, including information about the data types of each column and the number of non-null values. It's a quick way to get an overview of your dataset.

```
import pandas as pd

# Create a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'San Francisco', 'Los Angeles']}
df = pd.DataFrame(data)

# Display information about the DataFrame
df.info()
```

```
# Output:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    3 non-null      object
 1   Age     3 non-null      int64
 2   City    3 non-null      object
dtypes: int64(1), object(2)
memory usage: 200.0+ bytes
```

12. Drop Duplicate values of DataFrame:

The `drop_duplicates()` method allows you to remove duplicate rows from a DataFrame, keeping only the first occurrence.

```
DataFrame.drop_duplicates(subset = None, keep = 'first', inplace = False, ignore_index
= False)
```

In pandas, when you're dealing with duplicate rows in a DataFrame, you can control which duplicates to keep based on the order of appearance using the `keep` parameter within the `drop_duplicates()` method. The `keep` parameter accepts three possible values: `'first'`, `'last'`, and `False`. Here's what each of these options means:

1. **`keep = 'first'`**: This option keeps the first occurrence of a duplicated row and removes all subsequent duplicates.

   ```
   df = df.drop_duplicates(keep='first')
   ```

   In this case, the first occurrence of a duplicated row is retained, and any subsequent duplicates are dropped.

2. **`keep = 'last'`**: This option keeps the last occurrence of a duplicated row and removes all previous duplicates.

   ```
   df = df.drop_duplicates(keep='last')
   ```

   Here, the last occurrence of a duplicated row is retained, and any previous duplicates are dropped.

3. **`keep = False`**: This option removes all duplicated rows entirely, keeping none of them.

   ```
   df = df.drop_duplicates(keep=False)
   ```

   With `keep = False`, all instances of duplicated rows are removed, leaving only unique rows in the DataFrame.

12. Check Column Names:

To print the column names of a DataFrame, you can use the `columns` attribute.

```
# Print column names
print(df.columns)

# Index(['Name', 'Age', 'City'], dtype='object')
```

13. Rename Column Names:

You can rename columns using the `rename()` method or by directly assigning new column names to the `columns` attribute.

```
# Rename columns using the rename() method
df = df.rename(columns={'Name': 'Full Name', 'Age': 'Years'})

# Rename columns by directly assigning new names
df.columns = ['Name', 'Age', 'City']
```

Lowercase All Column Names: To convert all column names to lowercase, you can use a list comprehension or the `str.lower()` method.

```
# Convert all column names to lowercase
df.columns = [col.lower() for col in df.columns]
```

14. Getting Summary of DataFrame

The `describe()` method provides summary statistics of the numerical columns in a DataFrame, including count, mean, std (standard deviation), min, 25%, 50%, 75%, and max values.

```
# Generate summary statistics for numerical columns
df.describe()

# Output:
Age
count   3.0
mean   30.0
std     5.0
min    25.0
25%    27.5
50%    30.0
```

```
75%    32.5
max    35.0
```

15. Frequency of Unique values

The `value_counts()` method is used to determine the frequency of unique values in a column. It is particularly useful for categorical data.

```
# Get the frequency of values in a column
city_counts = df['City'].value_counts()

# Output:
New York         1
San Francisco    1
Los Angeles      1
Name: City, dtype: int64
```

You can also use `unique()` function that returns an array of unique values found in the specified column.

```
# Get unique values in a column
unique_cities = df['City'].unique()

# Output:
array(['New York', 'San Francisco', 'Los Angeles'], dtype=object)
```

16. idxmax

- `idxmax` is a pandas function used to find the index (row label) of the maximum value in a Series or DataFrame.
- It's like asking pandas, "Which row has the highest value for a particular column?"

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Score': [85, 92, 78]}
df = pd.DataFrame(data)

# Find the index (row label) of the maximum score
```

```
max_index = df['Score'].idxmax()
print(max_index)
```

Output:

```
1
```

In this example, `idxmax` tells us that the row with index 1 (Bob) has the highest score (92).

17. Appending Rows in Pandas

You can use the `append()` method to add rows to an existing DataFrame. This is useful when you want to combine data from different sources or add new observations to your DataFrame.

Here's how you can do it step by step:

1. **Create an initial DataFrame**:

   ```
   import pandas as pd

   data = {'Name': ['Alice', 'Bob'],
           'Age': [25, 30]}
   df = pd.DataFrame(data)
   ```

2. **Prepare the new data**:

   ```
   new_data = {'Name': ['Charlie', 'David'],
               'Age': [35, 28]}
   new_df = pd.DataFrame(new_data)
   ```

3. **Append the new data**:

   Use the `append()` method to add the new rows to the existing DataFrame. Make sure to set `ignore_index=True` to reindex the resulting DataFrame.

   ```
   combined_df = df.append(new_df, ignore_index=True)
   ```

```
print(combined_df)
```

Output:

```
   Name  Age
0   Alice   25
1     Bob   30
2  Charlie   35
3    David   28
```

4. Pandas with Functions

Working with functions in pandas involves applying custom or built-in functions to Series or DataFrame objects to perform data transformations, aggregations, or other operations.

Applying Functions to Series:

You can apply a function to each element of a Series using the `apply()` method. Here's an example where we apply a custom function to a Series:

```
import pandas as pd

# Create a sample Series
data = [1, 2, 3, 4, 5]
series = pd.Series(data)

# Define a custom function
def square(x):
    return x ** 2

# Apply the function to the Series
squared_series = series.apply(square)
```

Applying Functions to DataFrames:

You can also apply functions to DataFrames along rows or columns using `apply()`. Here's an example:

```
# Create a sample DataFrame
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
```

```
df = pd.DataFrame(data)

# Apply a function to each column
def double_column(col):
    return col * 2

# For whole DataFrame
df = df.apply(double_column)

# For one Column of DataFrame
df = df['A'].apply(double_column)
```

5. Pandas with CSV and HTML Data

In pandas, you can easily read and write data from/to CSV and HTML files. This is a common and essential part of data analysis, as many datasets are stored in CSV (Comma-Separated Values) format, and you might want to share your analysis results in HTML format. Here's how you can work with CSV and HTML data in pandas:

Reading Data from CSV Files:

You can use the `pd.read_csv()` function to read data from a CSV file into a pandas DataFrame. Here's an example:

```
import pandas as pd

# Read data from a CSV file into a DataFrame
df = pd.read_csv('data.csv')
```

In this example, replace `'data.csv'` with the path to your CSV file.

Writing Data to CSV Files:

You can use the `to_csv()` method to write a pandas DataFrame to a CSV file. Here's an example:

```
# Write a DataFrame to a CSV file
df.to_csv('output.csv', index=False)
```

In this example, replace `'output.csv'` with the desired output file path. Setting `index=False` will omit writing the DataFrame index to the CSV file.

Reading Data from HTML Tables:

Pandas can also read data from HTML tables found on webpages. You can use the `pd.read_html()` function to scrape HTML tables and store them in a list of DataFrames. Here's an example:

```
import pandas as pd

# Read HTML tables from a webpage into a list of DataFrames
tables = pd.read_html('<https://example.com/mytable.html>')

# Select the desired DataFrame from the list (if there are multiple tables)
df = tables[0]
```

In this example, replace `'<https://example.com/mytable.html'>` with the URL of the webpage containing the HTML table you want to scrape.

Writing Data to HTML:

You can use the `to_html()` method to convert a pandas DataFrame to an HTML table. Here's an example:

```
# Convert a DataFrame to an HTML table and save it to a file
html_table = df.to_html('output.html', index=False)
```

In this example, replace `'output.html'` with the desired output HTML file path. Setting `index=False` will omit rendering the DataFrame index in the HTML table.