1. What is *Pandas* in *Python* and why is it used for data analysis?

Pandas is a powerful Python library for data analysis. In a nutshell, it's designed to make the manipulation and analysis of structured data intuitive and efficient.

Key Features

- Data Structures: Offers two primary data structures: `Series` for one-dimensional data and `DataFrame` for two-dimensional tabular data.
- Data Munging Tools: Provides rich toolsets for data cleaning, transformation, and merging.
- Time Series Support: Extensive functionality for working with time-series data, including date range generation and frequency conversion.
- Data Input/Output: Facilitates effortless interaction with a variety of data sources, such as CSV, Excel, SQL databases, and REST APIs.
- Flexible Indexing: Dynamically alters data alignments and joins based on row/column index labeling.

Ecosystem Integration

Pandas works collaboratively with several other Python libraries like:

- Visualization Libraries: Seamlessly integrates with Matplotlib and Seaborn for data visualization.
- Statistical Libraries: Works in tandem with statsmodels and SciPy for advanced data analysis and statistics.

Performance and Scalability

Pandas is optimized for fast execution, making it reliable for small to medium-sized datasets. For large datasets, it provides tools to optimize or work with the data in chunks.

Common Data Operations

- Loading Data: Read data from files like CSV, Excel, or databases using the built-in functions.
- Data Exploration: Get a quick overview of the data using methods like `describe`, `head`, and `tail`.
- Filtering and Sorting: Use logical indexing to filter data or the `sort_values` method to order the data.
- Missing Data: Offers methods like `isnull`, `fillna`, and `dropna` to handle missing data efficiently.
- Grouping and Aggregating: Group data by specific variables and apply aggregations like sum, mean, or count.
- Merging and Joining: Provide several merge or join methods to combine datasets, similar to SQL.
- Pivoting: Reshape data, often for easier visualization or reporting.
- Time Series Operations: Includes functionality for date manipulations, resampling, and time-based queries.
- Data Export: Save processed data back to files or databases.

Code Example

Here is the Python code:

```python
import pandas as pd

# Create a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'Age': [25, 30, 35, 40],
    'Department': ['HR', 'Finance', 'IT', 'Marketing']
}
df = pd.DataFrame(data)

# Explore the data
print(df)
print(df.describe())  # Numerical summary

# Filter and sort the data
filtered_df = df[df['Department'].isin(['HR', 'IT'])]
sorted_df = df.sort_values(by='Age', ascending=False)

# Handle missing data
df.at[2, 'Age'] = None  # Simulate missing age for 'Charlie'
df.dropna(inplace=True)  # Drop rows with any missing data

# Group, aggregate, and visualize
```

```
grouped_df = df.groupby('Department')['Age'].mean()
grouped_df.plot(kind='bar')

# Export the processed data
df.to_csv('processed_data.csv', index=False)
```

2. Explain the difference between a *Series* and a *DataFrame* in *Pandas*.

Pandas, a popular data manipulation and analysis library, primarily operates on two data structures: Series, for one-dimensional data, and DataFrame, for two-dimensional data.

Series Structure

- Data Model: Each Series consists of a one-dimensional array and an associated array of labels, known as the index.
- Memory Representation: Data is stored in a single ndarray.
- Indexing: Series offers simple, labeled indexing.
- Homogeneity: Data is homogeneous, meaning it's of a consistent data type.

DataFrame Structure

- Data Model: A DataFrame is composed of data, which is in a 2D tabular structure, and an index, which can be a row or column header or both. Extending its table-like structure, a DataFrame can also contain an index for both its rows and columns.
- Memory Representation: Internally, a DataFrame is made up of one or more Series structures, in one or two dimensions. Data is 2D structured.
- Indexing: Data can be accessed via row index or column name, favoring loc and iloc for multi-axes indexing.
- Columnar Data: Columns can be of different, heterogeneous data types.
- Missing or NaN Values: DataFrames can accommodate missing or NaN entries.

Common Features of Series and DataFrame

Both Series and DataFrame share some common characteristics:

- Mutability: They are both mutable in content, but not in length of the structure.
- Size and Shape Changes: Both the Series and DataFrame can change in size. For the Series, you can add, delete, or modify elements. For the DataFrame, you can add or remove columns or rows.
- Sliceability: Both structures support slicing operations and allow slicing with different styles of indexers.

3. How can you read and write data from and to a *CSV file* in *Pandas*?

Pandas makes reading from and writing to CSV files straightforward.

Reading a CSV File

You can read a CSV file into a DataFrame using the `.read_csv()` method. Here is the code:

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

Configuring the Read Operation

- Header: By default, the first row of the file is used as column names. If your file doesn't have a header, you should set `header=None`.
- Index Column: Select a column to be used as the row index. Pass the column name or position using the `index_col` parameter.
- Data Types: Let Pandas infer data types or specify them explicitly through `dtype` parameter.
- Text Parsing: Handle non-standard delimiters or separators using `sep` and `delimiter`.
- Date Parsing: Specify date formats for more efficient parsing using the `parse_dates` parameter.

Writing to a CSV File

You can export your DataFrame to a CSV file using the `.to_csv()` method.

```
df.to_csv('output.csv', index=False)
```

- Index: If you don't want to save the index, set `index` to `False`.
- Specifying Delimiters: If you need to use a different delimiter, e.g., tabs, use `sep` parameter.
- Handling Missing Values: Choose a representation for missing values, such as `na_rep='NA'`.
- Encoding: Use the `encoding` parameter to specify the file encoding, such as 'utf-8' or 'latin1'.
- Date Format: When writing dates to the file, choose 'ISO8601', 'epoch', or specify your custom format with `date_format`.
- Compression: If your data is large, use the `compression` parameter to save disk space, e.g., `compression='gzip'` for compressed files.

Example: Writing a Dataframe to CSV

Here is a code example:

```
import pandas as pd

data = {
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [25, 22, 28]
}

df = pd.DataFrame(data)

df.to_csv('people.csv', index=False, header=True)
```

4. What are *Pandas indexes*, and how are they used?

In Pandas, an index serves as a unique identifier for each row in a DataFrame, making it powerful for data retrieval, alignment, and manipulation.

Types of Indexes

1. Default, Implicit Index: Automatically generated for each row (e.g., 0, 1, 2, ...).

2. Explicit Index: A user-defined index where labels don't need to be unique.
3. Unique Index: All labels are unique, typically seen in primary key columns of databases.
4. Non-Unique Index: Labels don't have to be unique, can have duplicate values.
5. Hierarchical (MultiLevel) Index: Uses multiple columns to form a unique identifier for each row.

Key Functions for Indexing

- Loc: Uses labels to retrieve rows and columns.
- Iloc: Uses integer indices for row and column selection.

Operations with Indexes

- Reindexing: Changing the index while maintaining data integrity.
- Set operations: Union, intersection, difference, etc.
- Alignment: Matches rows based on index labels.

Code Example: Index Types

```
# Creating DataFrames with different types of indexes
import pandas as pd

# Implicit index
df_implicit = pd.DataFrame({'A': ['a', 'b', 'c'], 'B': [1, 2, 3]})

# Explicit index
df_explicit = pd.DataFrame({'A': ['a', 'b', 'c'], 'B': [1, 2, 3]}, index=['X',
'Y', 'Z'])

# Unique index
df_unique = pd.DataFrame({'A': ['a', 'b', 'c'], 'B': [1, 2, 3]}, index=['1st',
'2nd', '3rd'])

# Non-unique index
df_non_unique = pd.DataFrame({'A': ['a', 'b', 'c', 'd'], 'B': [1, 2, 3, 4]},
index=['E', 'E', 'F', 'G'])

# Hierarchical index
df_hierarchical = df_explicit.set_index('A')

print(df_implicit, df_explicit, df_unique, df_non_unique, df_hierarchical,
sep='\n\n')
```

Code Example: Key Operations

```
# Setting up a DataFrame for operation examples
import pandas as pd

data = {
    'A': [1, 2, 3, 4, 5],
    'B': [10, 20, 30, 40, 50],
    'C': [100, 200, 300, 400, 500]
}

df = pd.DataFrame(data, index=['a', 'b', 'c', 'd', 'e'])

# Reindexing
new_index = ['a', 'e', 'c', 'b', 'd']  # Reordering index labels
df_reindexed = df.reindex(new_index)

# Set operations
index1 = pd.Index(['a', 'b', 'c', 'f'])
index2 = pd.Index(['b', 'd', 'c'])
print(index1.intersection(index2))  # Intersection
print(index1.difference(index2))  # Difference

# Data alignment
data2 = {'A': [6, 7], 'B': [60, 70], 'C': [600, 700]}
df2 = pd.DataFrame(data2, index=['a', 'c'])
print(df + df2)  # Aligned based on index labels
```

5. How do you handle *missing data* in a *DataFrame*?

Dealing with missing data is a common challenge in data analysis. Pandas offers flexible tools for handling missing data in DataFrames.

Common Techniques for Handling Missing Data in Pandas

Dropping Missing Values

This is the most straightforward option, but it might lead to data loss.

- Drop NaN Rows: `df.dropna()`

- Drop NaN Columns: `df.dropna(axis=1)`

Filling Missing Values

Instead of dropping missing data, you can choose to fill it with a certain value.

- Fill with a Constant Value: `df.fillna(value)`
- Forward Filling: `df.fillna(method='ffill')`
- Backward Filling: `df.fillna(method='bfill')`

Interpolation

Pandas gives you the option to use various interpolation methods like linear, time-based, or polynomial.

- Linear Interpolation: `df.interpolate(method='linear')`

Masking

You can create a mask to locate and replace missing data.

- Create a Mask: `mask = df.isna()`
- Replace with a value if NA: `df.mask(mask, value=value_to_replace_with)`

Tagging

Categorize or "tag" missing data to handle it separately.

- Select Missing Data: `missing_data = df[df['column'].isna()]`

Advanced Techniques

Pandas supports more refined strategies:

- Apply a Function: `df.apply()`
- Use External Libraries for Advanced Imputation: Libraries like `scikit-learn` offer sophisticated imputation techniques.

Visualizing Missing Data

The `missingno` library provides a quick way to visualize missing data in pandas dataframes using heatmaps. This can help to identify patterns in missing data that might not be immediately obvious from summary statistics. Here is a code snippet to do that:

6. Discuss the use of `groupby` in *Pandas* and provide an example.

GroupBy in Pandas lets you aggregate and analyze data based on specific features or categories. This allows for powerful split-apply-combine operations, especially when combined with `agg` to define multiple aggregations at once.

Core Functions in GroupBy

- Split-Apply-Combine: This technique segments data into groups, applies a designated function to each group, and then merges the results.
- Lazy Evaluation: GroupBy operations are not instantly computed. Instead, they are tracked and implemented only when required. This guarantees efficient memory utilization.
- In-Memory Caching: Once an operation is computed for a distinctive group, its outcome is saved in memory. When a recurring computation for the same group is demanded, the cached result is utilized.
  Beneath this sleek surface, GroupBy functionalities meld simplicity with robustness, furnishing swift and accurate outcomes.

Key Methods

- `groupby()`: Divides a dataframe into groups based on specified keys, often column names. This provides a `groupby` object which can be combined with additional aggregations or functions.
- Aggregation Functions: These functions generate aggregated summaries once the data has been divided into groups. Commonly used aggregation functions include `sum`, `mean`, `median`, `count`, and `std`.
- Chaining GroupBy Methods: `.filter()`, `.apply()`, `.transform()`.

Practical Applications

- Statistical Summary by Category: Quickly compute metrics such as average or median for segregated data.
- Data Quality: Pinpoint categories with certain characteristics, like groups with more missing values.
- Splitting by Predicate: Employ `.filter` to focus on particular categories that match user-specified criteria.
- Normalized Data: Deploy `.transform()` to standardize or normalize data within group partitions.

Code Example: GroupBy & Aggregation

Consider this dataset of car sales:

Car	Category	Price	Units Sold
Honda	Sedan	25000	120
Honda	SUV	30000	100
Toyota	Sedan	23000	150
Toyota	SUV	28000	90
Ford	Sedan	24000	110
Ford	Pickup	35000	80

We can compute the following using GroupBy and Aggregation:

1. Category-Wise Sales:

    - Sum of "Units Sold"
    - Average "Price"
2. General computations:
    - Total car sales
    - Maximum car price.

7. Explain the concept of *data alignment* and *broadcasting* in *Pandas*.

Data alignment and broadcasting are two mechanisms that enable pandas to manipulate datasets with differing shapes efficiently.

Data Alignment

Pandas operations, such as addition, are designed to generate new series that adhere to the index of both source series, avoiding any NaN values. This process, where the data aligns based on the index labels, is known as data alignment.

This behavior is particularly useful when handling data where values may be missing.

How It Works

Take two DataFrames, `df1` and `df2`, each with different shapes and sharing only partial indices:

- `df1`:
    - Index: A, B, C
    - Column: X
    - Values: 1, 2, 3
- `df2`:
    - Index: B, C, D
    - Column: Y
    - Values: 4, 5, 6

When you perform an addition (`df1 + df2`), pandas join values that have the same index. The resulting DataFrame has:

- Index: A, B, C, D

- Columns: X, Y
- Values: NaN, 6, 8, NaN

Broadcasting

Pandas efficiently manages operations between objects of different dimensions or shapes through broadcasting.

It employs a set of rules that allow operations to be performed on datasets even if they don't perfectly align in shape or size.

Key broadcasting rules:

1. Scalar: Any scalar value can operate on an entire Series or DataFrame.
2. Vector-Vector: Operations occur pairwise—each element in one dataset aligns with the element in the same position in the other dataset.
3. Vector-Scalar: Each element in a vector gets the operation with the scalar.

Example: Add a Scalar to a Series

Consider a Series `s`:

```None
           A
           -
Index: 0, 1, 2
Value: 3, 4, 5
```

Now, perform `s + 2`. This adds 2 to each element of the Series, resulting in:

```None
           A
           -
Index: 0, 1, 2
Value: 5, 6, 7
```

8. What is *data slicing* in *Pandas*, and how does it differ from *filtering*?

Data slicing and filtering are distinct techniques used for extracting subsets of data in Pandas.

Key Distinctions

- Slicing entails the selection of contiguous rows and columns based on their order or the position within the DataFrame. This is more about locational reference.
- Filtering, however, involves selecting rows and columns conditionally based on specific criteria or labels.

Code Example: Slicing vs Filtering

Here is the Python code:

```python
import pandas as pd

# Create a simple DataFrame
data = {'A': [1, 2, 3, 4, 5],
        'B': [10, 20, 30, 40, 50],
        'C': ['foo', 'bar', 'baz', 'qux', 'quux']}
df = pd.DataFrame(data)

# Slicing
sliced_df = df.iloc[1:4, 0:2]
print("Sliced DataFrame:")
print(sliced_df)
# 'iloc' is a positional selection method.

# Filtering
filtered_df = df[df['A'] > 2]
print("\nFiltered DataFrame:")
print(filtered_df)
# Here, we use a conditional expression within the brackets to filter rows.
```

9. Describe how *joining* and *merging* data works in *Pandas*.

Pandas provides versatile methods for combining and linking datasets, with the two main approaches being `join` and `merge`. Let's explore how they operate.

Join: Relating DataFrames on Index or Column

The `join` method is a convenient way to link DataFrames based on specific columns or their indices, aligning them either through intersection or union.

Types of Joins

- Inner Join: Retains only the common entries between the DataFrames.
- Outer Join: Maintains all entries, merging based on where keys exist in either DataFrame.

These types of joins can be performed both along the index (`df1.join(df2)`) as well as specified columns (`df1.join(df2, on='column_key')`). The default join type is an Inner Join.

Example: Inner Join on Index

```
# Inner join on default indices
result_inner_index = df1.join(df2, lsuffix='_left')

# Inner join on specified column and index
result_inner_col_index = df1.join(df2, on='key', how='inner', lsuffix='_left',
rsuffix='_right')
```

Merge: Handling More Complex Join Scenarios

The `merge` function in pandas provides greater flexibility than `join`, accommodating a range of keys to combine DataFrames.

Join Types

- Left Merge: All entries from the left DataFrame are kept, with matching entries from the right DataFrame. Unmatched entries from the right get `NaN` values.

- Right Merge: Correspondingly serves the right DataFrame.
- Outer Merge: Unites all entries from both DataFrames, addressing any mismatches with `NaN` values.
- Inner Merge: Selects only entries with matching keys in both DataFrames.

These merge types can be assigned based on the data requirement. You can use the `how` parameter to specify the type of merge.

Code Example

```
# Perform a left merge aligning on 'key' column
left_merge = df1.merge(df2, on='key', how='left')

# Perform an outer merge on 'key' and 'another_key' columns
outer_merge = df1.merge(df2, left_on='key', right_on='another_key',
how='outer')
```

10. How do you *apply* a function to all elements in a *DataFrame* column?

You can apply a function to all elements in a DataFrame column using Pandas' `.apply()` method along with a lambda function for quick transformations.

Using .apply() and Lambda Functions

The `.apply()` method works as a vectorized alternative for element-wise operations on a column. This mechanism is especially useful for complex transformation and calculation steps.

Here is the generic structure:

```
# Assuming 'df' is your DataFrame and 'col_name' is the name of your column
df['col_name'] = df['col_name'].apply(lambda x: your_function(x))
```

You can tailor this approach to your particular transformation function.

Example: Doubling Values in a Column

Let's say you want to double all values in a `scores` column of your DataFrame:

```
import pandas as pd

# Sample DataFrame
data = {'names': ['Alice', 'Bob', 'Charlie'], 'scores': [80, 90, 85]}
df = pd.DataFrame(data)

# Doubling the scores using .apply() and a lambda function
df['scores'] = df['scores'].apply(lambda x: x*2)

# Verify the changes
print(df)
```

Considerations

- Efficiency: Depending on the nature of your function, a traditional `for` loop might be faster, especially for simple operations. However, in many scenarios, using vectorized operations such as `.apply()` can lead to improved efficiency.
- In-Place vs. Non In-Place: Specifying `inplace=True` in the `.apply()` method directly modifies the DataFrame, to save you from the need for reassignment.

11. Demonstrate how to handle *duplicate rows* in a *DataFrame*.

Dealing with duplicate rows in a DataFrame is a common data cleaning task. The Pandas library provides simple, yet powerful, methods to identify and handle this issue.

Identifying Duplicate Rows

You can use the `duplicated()` method to identify rows that are duplicated.

```
import pandas as pd

# Create a sample DataFrame
data = {'A': [1, 1, 2, 2, 3, 3], 'B': ['a', 'a', 'b', 'b', 'c', 'c']}
df = pd.DataFrame(data)

# Identify duplicate rows
print(df.duplicated())  # Output: [False, True, False, True, False, True]
```

Dropping Duplicate Rows

You can use the `drop_duplicates()` method to remove duplicated rows.

```
# Drop duplicates
unique_df = df.drop_duplicates()

# Alternatively, you can keep the last occurrence
last_occurrence_df = df.drop_duplicates(keep='last')

# To drop in place, use the `inplace` parameter
df.drop_duplicates(inplace=True)
```

Carrying Out Aggregations

For numerical data, you can aggregate using functions such as mean or sum. This is beneficial when duplicates may have varying values in other columns.

```
# Aggregate using mean
mean_df = df.groupby('A').mean()
```

Counting Duplicates

To count the occurrences of duplicates, use the `duplicated()` method in conjunction with `sum()`. This provides the number of duplicates for each row.

```
# Count duplicates
num_duplicates = df.duplicated().sum()
```

Keeping the First or Last Occurrence

By default, `drop_duplicates()` keeps the first occurrence of a duplicated row.

If you prefer to keep the last occurrence, you can use the `keep` parameter.

```
# Keep the last occurrence
df_last = df.drop_duplicates(keep='last')
```

Leverage Unique Identifiers

Identifying duplicates might require considering a subset of columns. For instance, in an orders dataset, two orders with the same order date might still be distinct because they involve different products. Set the subset of columns to consider with the `subset` parameter.

```python
# Consider only the 'A' column to identify duplicates
df_unique_A = df.drop_duplicates(subset=['A'])
```

12. Describe how you would convert *categorical data* into *numeric format*.

Converting categorical data to a numeric format, a process also known as data pre-processing, is fundamental for many machine learning algorithms that can only handle numerical inputs.

There are two common approaches to achieve this: Label Encoding and One-Hot Encoding.

Label Encoding

Label Encoding replaces each category with a unique numerical label. This method is often used with ordinal data, where the categories have an inherent order.

For instance, the categories "Low", "Medium", and "High" can be encoded as 1, 2, and 3.

Here's the Python code using scikit-learn's `LabelEncoder`:

```python
from sklearn.preprocessing import LabelEncoder
import pandas as pd

data = {'Size': ['S', 'M', 'L', 'XL', 'M']}
df = pd.DataFrame(data)

le = LabelEncoder()
df['Size_LabelEncoded'] = le.fit_transform(df['Size'])
print(df)
```

One-Hot Encoding

One-Hot Encoding creates a new binary column for each category in the dataset. For every row, only one of these columns will have a value of 1, indicating the presence of that category.

This method is ideal for nominal data that doesn't indicate any order.

Here's the Python code using scikit-learn's `OneHotEncoder` and pandas:

```
from sklearn.preprocessing import OneHotEncoder

# Avoids SettingWithCopyWarning
df = pd.get_dummies(df, prefix=['Size'], columns=['Size'])

print(df)
```

13. How can you *pivot* data in a *DataFrame*?

In Pandas, you can pivot data using the `pivot` or `pivot_table` methods. These functions restructure data in various ways, such as converting unique values into column headers or aggregating values where duplicates exist for specific combinations of row and column indices.

Key Methods

- `pivot`: It works well with a simple DataFrame but cannot handle duplicate entries for the same set of index and column labels.
- `pivot_table`: Provides more flexibility and robustness. It can deal with data duplicates and perform varying levels of aggregation.

Code Example: Pivoting with `pivot`

Here is the Python code:

```
import pandas as pd

# Sample data
data = {
    'Date': ['2020-01-01', '2020-01-01', '2020-01-02', '2020-01-02'],
    'Category': ['A', 'B', 'A', 'B'],
    'Value': [10, 20, 30, 40]
}

df = pd.DataFrame(data)

# Pivot the DataFrame
pivot_df = df.pivot(index='Date', columns='Category', values='Value')
print(pivot_df)
```

Output

The pivot DataFrame is a transformation of the original one:

Date	A	B
2020-01-01	10	20
2020-01-02	30	40

Code Example: Pivoting with `pivot_table`

Here is the Python code:

```
import pandas as pd

# Sample data
data = {
    'Date': ['2020-01-01', '2020-01-01', '2020-01-02', '2020-01-02'],
    'Category': ['A', 'B', 'A', 'B'],
    'Value': [10, 20, 30, 40]
}

df = pd.DataFrame(data)

# Pivoting the DataFrame using pivot_table
pivot_table_df = df.pivot_table(index='Date', columns='Category',
values='Value', aggfunc='sum')
print(pivot_table_df)
```

Output

The pivot table presents the sum of values for unique combinations of 'Date' and 'Category'. Notice how duplicate entries for some combinations have been automatically aggregated.

Date	A	B
2020-01-01	10	20
2020-01-02	30	40

14. Show how to apply *conditional logic* to columns using the `where()` method.

The `where()` method in Pandas enables conditional logic on columns, providing a more streamlined alternative to `loc` or `if-else` statements.

The method essentially replaces values in a DataFrame or Series based on defined conditions.

`where()` Basics

Here are some important key points:

- Import: It works directly on data obtained via Pandas modules (`import pandas as pd`).
- Parameters: Specify conditions like `cond` for when to apply replacements and `other` for the values to replace when the condition is False.

Code Example: `where()`

Here is the Python code:

```
# Importing Pandas
import pandas as pd

# Sample Data
```

```
data = {'A': [1, 2, 3, 4, 5], 'B': [10, 20, 30, 40, 50]}
df = pd.DataFrame(data)

# Applying 'where'
result = df.where(df > 2, df * 10)

# Output
print(result)

# Output:   A      B
#           0    NaN    NaN
#           1    NaN    NaN
#           2    3.0   30.0
#           3    4.0   40.0
#           4    5.0   50.0
```

In this example:

- Values less than or equal to 2 are replaced with `original_value * 10`.
- `NaN` values are returned for all the cells that did not meet the condition.

15. What is the purpose of the `apply()` function in *Pandas*?

The `apply()` function in Pandas is utilized to apply a given function along either the rows or columns of a DataFrame for advanced data manipulation tasks.

Practical Applications

- Row or Column Wise Operations: Suitable for applying element-wise or aggregative functions, like sum, mean, or custom-defined operations, across rows or columns.
- Aggregations: Ideal for multiple computations, for example calculating totals and averages simultaneously.
- Custom Operations: Provides versatility for applying custom functions across data; for example, calculating the interquantile range of two columns.
- Scalability: Offers performance improvements over element-wise iterations, particularly in scenarios with significantly large datasets.

Syntax

The `apply()` function typically requires specification regarding row or column iteration:

```
# Row-wise function application
df.apply(func, axis=1)

# Column-wise function application (default; '0' or 'index' yields the same behavior)
df.apply(func, axis='columns')
```

Here `func` denotes the function to apply, which can be a built-in function, lamba function, or user-constructed function.

Code Example: `apply()`

Let's look at a simple code example to show how `apply()` can be used to calculate the difference between two columns.

```
import pandas as pd

# Sample data
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Define the function to calculate the difference
def diff_calc(row):
    return row['B'] - row['A']

# Apply the custom function along the columns
df['Diff'] = df.apply(diff_calc, axis='columns')

print(df)
```

The output will be:

```
None
   A  B  Diff
0  1  4     3
1  2  5     3
2  3  6     3
```