Lecture 08 - Data Manipulation (`Pandas`)

Overview

What is `pandas`?

- `Pandas` is a powerful Python library used for **data manipulation** and **analysis**.
- It provides two main data structures: `Series` (1D) and `DataFrame` (2D), which are ideal for working with **structured data**, similar to Excel spreadsheets or SQL tables.

Why `pandas` for Finance?

- **Exploratory Data Analysis** (EDA): large and structured financial datasets that require cleaning, manipulation, and analysis.
- **Efficiency**: `Pandas` can handle millions of rows of financial data efficiently.
- **Integration with Data Science**: `Pandas` works seamlessly with other libraries like `NumPy` and `matplotlib` for numerical operations and visualization, essential for financial modeling.

This notebook covers:

- `DataFrame` and `Series` classes
- Basic operations with `Pandas`
- `GroupBy`, complex selection and data combinations

In [ ]:

```
import pandas as pd
import numpy as np
```

1. The `DataFrame` Class

At the core of `Pandas` is the `DataFrame`, a class designed to **efficiently handle data in tabular form** —i.e., data characterized by a columnar organization.

- A `DataFrame` in `pandas` is a **two-dimensional**, **labeled data structure** that organizes and manipulates structured data
- `DataFrames` consist of **rows** and **columns**, where each column can contain different types of data (`integers`, `floats`, `strings`, etc.).
    - Like a table in a relational database or an Excel sheet.

1.1 Creating a `DataFrame`

The `pd.DataFrame()` function in `pandas` creates `DataFrames`.

```
pd.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False)
```

- `data` : Data to populate the `DataFrame`
- `index` : Index labels for the rows (optional).
- `columns` : Column labels (optional).
- `dtype` : Data type for the elements (optional).
- `copy` : Whether to copy the input data (optional).

There are several ways to create a `DataFrame`.

- from **scratch** using `lists` or `dictionaries`
- from **reading** external files (CSV, Excel, SQL databases).

From a `list`

In [ ]:
```
# Creating a DataFrame from a list
data = [10, 20, 30, 40]
df = pd.DataFrame(data, columns=['numbers'], index=['a', 'b', 'c', 'd']
```

In [ ]:
```
# Displaying the DataFrame
print(df)
```

In this example:

- The `data` list contains values for a single column.
- The `columns` parameter labels the column, and the `index` parameter sets row labels.

Once a `DataFrame` is instantiated, one can observe its meta structure.

In [ ]:
```
df.index
```

In [ ]:
```
df.columns
```

From a `dictionary`

A `DataFrame` can be sourced from a `dictionary`, where `keys` are column `names` and values are lists of column data.

In [ ]:
```
data = {
    'Product': ['A', 'B', 'C', 'D'],
    'Price': [100, 150, 200, 250],
    'Stock': [50, 60, 70, 80]
}

df = pd.DataFrame(data)

# Displaying the DataFrame
print(df)
```

From external sources

`DataFrames` can also be created by reading data from **external files** such as CSVs, Excel files, or databases.

```
# Reading from a CSV file
df = pd.read_csv('file.csv')

# Reading from an Excel file
df = pd.read_excel('file.xlsx')
```

More on this in next lecture...

1.2 Accessing a `DataFrame`

Once a `DataFrame` is created, one can access

- columns
- rows
- specific subsets of data

Accessing columns

Access to columns can be done **directly** by referring the column like in a `dictionary`.

In [ ]:
```
# Accessing a single column
df['Product']
```

In [ ]:
```
# Accessing multiple columns
df[['Product', 'Price']]
```

Accessing rows

Access to rows can be done by index (`.loc[]`) or position (`.iloc[]`)

In [ ]:
```
df
```

In [ ]:
```
# Accessing by label
df.loc[1]  # Retrieves the row with index 1
```

In [ ]:
```
# Accessing by position
df.iloc[2]  # Retrieves the third row (index 2)
```

Selecting multiple rows

A range of rows or a specific set of rows can be selected by combining `.loc[]` or `.iloc[]` with slicing or `lists`.

In [ ]:
```
# Accessing multiple rows by label
df.loc[1:3]
```

In [ ]:
```
# Accessing multiple rows by position
df.iloc[0:2]
```

In [ ]:
```
# Accessing multiple rows by list
indeces = [1,3]
df.loc[indeces]
```

1.3 Editing a `DataFrame`

`DataFrames` are live objects which allow for adding, deleting or modifying data (columns or rows) on the fly.

Initialization

In [ ]:

```
data = {
    'Product': ['A', 'B', 'C', 'D'],
    'Price': [110, 160, 210, 260],
    'Stock': [50, 60, 70, 80],
}
df = pd.DataFrame(data)
```

Editing columns

- Adding
- Modifying
- Deleting

Adding columns

- **Internal product**

In [ ]:
```
# Adding a new column
df['Discounted_Price'] = df['Price'] * 0.9
print(df)
```

- **from `list`**
    - The length of the list should match the number of rows in the `DataFrame`.

In [ ]:
```
# Adding a new column 'Tax' using a list
df['Tax'] = [10.5, 15.0, 19.5, 24.0]
print(df)
```

In [ ]:
```
# Displaying the 'Tax' column
print(df['Tax'])
```

- **from `DataFrame`**
    - Indices must match between the original `DataFrame` and the new column.

In [ ]:
```
# Adding a new column 'Supplier' based on another DataFrame
df['Supplier'] = pd.DataFrame(['Supplier1', 'Supplier2', 'Supplier3', '
                               index=[0, 1, 2, 3])
```

After enlarging a `DataFrame`, it's also essential to check the data types of each column to ensure everything is in order.

In [ ]:
```
# Checking the data types of the DataFrame columns
print(df.dtypes)
```

Modifying columns

In [ ]:
```
# Modifying an existing column
df['Price'] = df['Price'] + 10
print(df)
```

Deleting columns

Removing columns can be done using the `.drop()` method.

In [ ]:
```
# Dropping the 'Discounted_Price' column
df_dropped = df.drop(columns=['Discounted_Price'])
print(df_dropped)
```

Editing rows

- Adding
- Modifying
- Deleting

Adding rows

- `append()`
    - Single row
    - However, note that `append()` is deprecated, and should be replaced using `pd.concat()` instead in future `pandas` versions.

In [ ]:

```
# Appending a new row to the DataFrame
df_appended = df.append({'Product': 'E', 'Price': 300, 'Stock': 90,
                         'Discounted_Price': 270.0, 'Tax': 30.0, 'Supplier': 'Su
                        ignore_index=True)
print(df_appended)
```

- `concat()`

In [ ]:
```python
# New row as a DataFrame
new_row = pd.DataFrame({
    'Product': ['E'],
    'Price': [300],
    'Stock': [90],
    'Discounted_Price': [270.0],
    'Tax': [30.0],
    'Supplier': ['Supplier5']
})

# Using pd.concat() to append the new row
df_concated = pd.concat([df, new_row], ignore_index=True)

# Display the updated DataFrame
print(df_concated)
```

More on `concat()` in section 6. Concatenation, Join, Merge and below on indexing

Modifying rows

In [ ]:
```
# Modifying an existing row
df.loc[1, 'Price'] = 180
df.loc[1, 'Stock'] = 200
```

In [ ]:
```
print(df)
```

Deleting rows

- Rows can be removed from a DataFrame using the `.drop()` method either by index or by condition.

In [ ]:
```
# Dropping a row by index
df_dropped = df.drop(index=2)  # Dropping the row with index 2 (Product
print(df_dropped)
```

In [ ]:
```
# Dropping rows where Stock is greater than 80
df_dropped = df[df['Stock'] <= 80]
print(df_dropped)
```

Indexing

When appending rows to a `DataFrame`, it's important to pay attention to the treatment of **indices**.

- **Indexing with `concat()`**
  - When concatenating two or more `DataFrames`, the default behavior is to **retain the original index** of the `DataFrames`, which can lead to duplicate indices.
  - The `ignore_index=True` parameter resets the index for the concatenated `DataFrame` to ensure a continuous sequence.

In [ ]:
```
# New DataFrame to concatenate
df_new = pd.DataFrame({
    'Product': ['F', 'G'],
    'Price': [350, 400],
    'Stock': [95, 100],
    'Discounted_Price': [315.0, 360.0],
    'Tax': [35.0, 40.0],
    'Supplier': ['Supplier6', 'Supplier7']
})
```

In [ ]:
```
# Concatenating the two DataFrames without resetting the index
df_concat = pd.concat([df, df_new])
print(df_concat)
```

In [ ]:
```
# Concatenating with index reset
df_concat_reset = pd.concat([df, df_new], ignore_index=True)
print(df_concat_reset)
```

- **Indexing with `append()`**
    - When appending, the new row needs to adopt the next available index by default (`ignore_index = True`), else does not proceed.

In [ ]:

```
# Appending a single row with a new product using append()
new_row = {
    'Product': 'H',
    'Price': 450,
    'Stock': 105,
    'Price_Squared': 202500,
    'Discounted_Price': 405.0,
    'Tax': 45.0,
    'Supplier': 'Supplier8'
}

df_appended = df.append(new_row, ignore_index = True)

print(df_appended)
```

1.4 Handling missing data

Missing data is common in real-world datasets, and `pandas` provides tools to handle them.

In [ ]:
```
# Adding missing values (NaN) to some cells
df.loc[1, 'Price'] = np.nan  # Missing price for product B
df.loc[2, 'Discounted_Price'] = np.nan  # Missing discounted price for
df.loc[3, 'Stock'] = np.nan  # Missing stock for product D
```

In [ ]:
```
print (df)
```

- **`df.isna()`**: Detects missing values

In [ ]:
```
df.isna()
```

- **`df.fillna(a)`**: Fils missing values with a default

In [ ]:
```
df.fillna(0)
```

- **`df.dropna()`** : Drops rows with missing values

In [ ]:
```
df.dropna()
```

Operations with missing data

`Pandas` method calls handle missing data

In [ ]:
```
df
```

In [ ]:
```
df[['Price', 'Stock']].mean()
```

In [ ]:
```
df[['Price', 'Stock']].std()
```

1.5 Sorting Data

`DataFrame` can be sorted based on any column using the `.sort_values()` method.

In [ ]:
```
# Sorting by 'Price' in ascending order
df_sorted = df.sort_values(by='Price')
```

In [ ]:
```
print (df_sorted)
```

In [ ]:
```
# Sorting by sequence of multiple columns: first stock, than price
df.loc[3,'Stock'] = df.loc[2,'Stock']
df_sorted = df.sort_values(by=['Stock', 'Price'], ascending=[False, Tru
```

In [ ]:
```
print (df_sorted)
```

1.6 Libraries integration

`Pandas` integrates the manipulation of objects from other classes such as `numpy` and `datetime` objects.

- **NumPy**

In [ ]:
```
import numpy as np
np.random.seed(100)
a = np.random.standard_normal((9,4))
a
```

In [ ]:
```
df_n = pd.DataFrame(a)
df_n.columns = ['No1', 'No2', 'No3', 'No4']
df_n
```

In [ ]:
```
df_n['No2'].mean()
```

- **DateTime**

In [ ]:
```
dates = pd.date_range('2019-1-1', periods = 9, freq = 'M')
dates
```

In [ ]:
```
df_n.index = dates
df_n
```

In [ ]:
```
df_n.values
```

🚦 Checkpoint

Task:

1. Create a `DataFrame` with the following columns: `Item`, `Price`, `Quantity` using the following dictionary

```
data = {
    'Item': ['Apple', 'Banana', 'Orange'],
    'Price': [1.0, 0.5, 0.75],
    'Quantity': [10, 5, 8]
        }
```

2. Add a new column `Total` that calculates the total price by multiplying `Price` and `Quantity`.
3. Add a column called `Discount` with values `[0.1, 0.05, 0.2]` to the existing DataFrame.
4. Update the `Total` column to apply the discount to the total price.

2. Basic Analytics

When working with data in `pandas`, the library provides several **built-in methods** that make data exploration and analysis efficient.

These methods help with

- Inspection
- Summaries
- Statistics
- Data operations
- Visualization

2.1 Inspection and summary

- **`df.info()`**: Provides a concise summary of the `DataFrame`, including the number of entries, column names, data types, and memory usage.

In [ ]:
```
df.info()
```

- **`df.head()`**: Returns the first few rows of the `DataFrame` (by default, the first 5 rows).

In [ ]:
```
df.head()  # By default, displays the first 5 rows
```

In [ ]:
```
df.head(10)  # Displays the first 10 rows
```

- **`df.tail()`**: Similar to `df.head()`, but returns the last few rows of the DataFrame.

In [ ]:
```
df.tail()  # By default, displays the last 5 rows
```

- **`df.describe()`**: Generates descriptive statistics for numerical columns, including count, mean, standard deviation, minimum, and maximum values, as well as the 25th, 50th, and 75th percentiles.

In [ ]:
```
df
```

In [ ]:
```
df.describe()
```

2.2 Statistics and operations

- **`df.sum()`**
- **`df.mean()`**

In [ ]:
```
df[['Price','Stock']].mean()  # By default, computes the mean of each c
```

In [ ]:
```
df[['Price','Stock']].mean(axis=0)  # Mean across columns (default beha
```

In [ ]:
```
df[['Price','Stock']].mean(axis=1)  # Mean across rows
```

- **`df.cumsum()`**

In [ ]:
```
df.cumsum()
```

- **`df.apply()`**: Custom functions can be applied to columns or rows using the `.apply()` method.

In [ ]:
```
# Applying a lambda function to square the values in 'Price' column
df['Price'].apply(lambda x: x ** 2)
```

2.3 Integrated `NumPy` Functions

`Pandas` integrates with `NumPy`, allowing the use of `NumPy` functions directly on `DataFrames`.

- **`np.mean(df)`**

In [ ]:
```
np.mean(df[['Price','Stock']])
```

- **`np.log(df)`**

In [ ]:
```
np.log(df)
```

- **`np.sqrt(abs(df))`**

In [ ]:
```
np.sqrt(abs(df))
```

In [ ]:
```
np.sqrt(abs(df)).sum()
```

2.4 Basic Visualization

`Pandas` integrates with visualization libraries like `matplotlib` to enable quick visualizations of data.

Setting up `Matplotlib` for Visualization

In [ ]:

```
from pylab import plt, mpl

# Setting the style to 'seaborn' for better aesthetics
plt.style.use('seaborn')

# Setting the default font to 'serif' for a clean look
mpl.rcParams['font.family'] = 'serif'

# Ensure plots are displayed inline in Jupyter notebooks
%matplotlib inline
```

Examples

In [ ]:
```
df.cumsum().plot(lw=2.0, figsize=(10, 6))  # Line width set to 2.0 and
```

In [ ]:
```
df.plot(kind='bar', figsize=(10, 6))  # Generates a bar plot with a fig
```

3. The `Series` Class

The `Series` class in `pandas` is a **one-dimensional labeled array** capable of holding any data type.

A `Series` is essentially a single column of data, making it a simpler, more specialized version of the `DataFrame` class. It shares many characteristics and methods with DataFrame and adds more specifics techniques.

3.1 Creating a `Series`

A `Series` object can be created directly or obtained by selecting a single column from a `DataFrame`.

- **From `list`**

In [ ]:
```
# Creating a Series with evenly spaced numbers between 0 and 15
s = pd.Series(np.linspace(0, 15, 7), name='series')
print(s)
```

In [ ]:
```
# Checking the type of the Series
type(s)
```

- **From `DataFrame`**

Selecting a single column from a `DataFrame` result in a `Series`.

In [ ]:
```
# Selecting a column from the DataFrame as a Series
s = df_n['No1']
print(s)
```

In [ ]:
```
# Checking the type
type(s)
```

3.2 Methods for `Series`

Inherited methods

Most of the methods available for `DataFrame` are also available for `Series`, such as `mean()`, or `plot()`.

In [ ]:
```
# Calculating the mean of the Series
s.mean()
```

In [ ]:
```
# Plotting the Series
s.plot(lw=2.0, figsize=(10, 6))  # Line width set to 2.0 and figure siz
```

`Series` specific methods

There are several methods that are specific to `Series` objects.

These methods are designed to leverage the **one-dimensional** nature of a `Series` and simplify certain operations that are less intuitive or applicable in a two-dimensional `DataFrame`.

- **`Series.value_counts()`**: Returns the count of unique values in a `Series`.
    - Useful when working with categorical data or needing to understand the frequency of certain values.

In [ ]:
```
# Example of value_counts
s = pd.Series(['A', 'B', 'A', 'C', 'B', 'A'])
s.value_counts()
```

- **`Series.unique()`**: Returns an array of the unique values in a `Series`.
    - It helps identify distinct values in a column of data.

In [ ]:
```
# Example of unique
s = pd.Series([1, 2, 2, 3, 4, 4, 4])
s.unique()
```

- **`Series.nunique()`**: Returns the number of unique values in a `Series`.
    - Similar to `value_counts()` but only provides the total number of unique elements, not their frequency.

In [ ]:
```
# Example of nunique
s = pd.Series([1, 2, 2, 3, 4, 4, 4])
s.nunique()
```

- **`Series.str` accessor:** performs string operations.
    - This feature is unique to Series and makes it easy to manipulate text data in bulk.
    - These include `.str.upper()`, `.str.contains()`, `.str.replace()`, and `.str.len()`

In [ ]:
```
# Sample Series of strings
s = pd.Series(['apple', 'banana', 'pear'])
```

In [ ]:
```
# Convert all strings to uppercase
s_upper = s.str.upper()
print(s_upper)
```

In [ ]:
```
# Check if each string contains the letter 'a'
s_contains = s.str.contains('a')
print(s_contains)
```

In [ ]:
```
# Replace 'a' with 'o' in each string
s_replace = s.str.replace('a', 'o')
print(s_replace)
```

In [ ]:
```
# Get the length of each string
s_len = s.str.len()
print(s_len)
```

- **`Series.dt` accessor:** performs datetime-specific operations.
    - This makes it easy to extract year, month, day, or other components from a datetime Series.
    - These include `dt.day`, `.dt.year`, `.dt.month`, and `.dt.weekday`.

In [ ]:
```
# Example of dt accessor
s = pd.Series(pd.date_range('2023-01-01', periods=3, freq='D'))
s.dt.day  # Extract the day from the datetime
```

- **`Series.isin()`**: Checks whether each element of the `Series` is in a given list of values and returns a boolean `Series`.

In [ ]:
```
# Example of isin
s = pd.Series(['A', 'B', 'C', 'D'])
s.isin(['B', 'C', 'E'])
```

- **`Series.idxmax()`** and **`Series.idxmin()`**: Return the index of the maximum or minimum value in the `Series`, respectively.

In [ ]:
```
# Example of idxmax and idxmin
s = pd.Series([1, 5, 3, 9, 2])
print(s.idxmax())  # Index of the maximum value
print(s.idxmin())  # Index of the minimum value
```

🚦 Checkpoint

Task:

1. Create a `pandas` Series from a list of stock prices: `[150.25, 153.50, 2800.50, 2830.75, 3400.00, 3450.50]` and name the Series `Stock_Prices`.

2. Using the `Stock_Prices` series:

    - Find the maximum price in the series.
    - Find the minimum price in the series.
    - Calculate the mean (average) price.
    - Find how many stock prices are above the mean.
    - Create a new Series where each stock price is increased by 5%.
    - Normalize the Series so that all values are between 0 and 1 (i.e., subtract the minimum and divide by the range).

3. Create another `pandas` Series from the list `[0.05, 0.03, 0.02, 0.04, 0.01, 0.06]` and name it `Stock_Changes`.
   This series represents the daily percentage changes for each stock (assume all changes are positive and range between 0.01 to 0.10).

4. `GroupBy` Operations

`Pandas` provides powerful and flexible **grouping** capabilities that function similarly to SQL groupings and pivot tables in Excel.

Grouping is often used when performing **aggregations** or **applying specific operations** to subsets of data.

Initialization

In [ ]:

```
# Creating a sample financial DataFrame
data = {
    'Stock': ['AAPL', 'AAPL', 'GOOGL', 'GOOGL', 'AMZN', 'AMZN', 'AAPL',
    'Quarter': ['Q1', 'Q1', 'Q1', 'Q2', 'Q2', 'Q2', 'Q3', 'Q3', 'Q3'],
    'Price': [150.25, 153.50, 2800.50, 2830.75, 3400.00, 3450.50, 155.3
    'Volume': [1000, 1100, 1500, 1600, 1700, 1800, 1200, 1700, 2000],
    'Market_Cap': [2.41e12, 2.45e12, 1.78e12, 1.82e12, 1.71e12, 1.75e12
}

df = pd.DataFrame(data)
df
```

4.1 Simple grouping

Simple grouping of a DataFrame using `.groupby()` is done on a specific column which then allows to perform basic operations on each group.

In [ ]:
```
groups = df.groupby('Quarter')
```

Inspection

- `.size()`: checks how many records belong to each group

In [ ]:
```
groups.size()
```

Basic Operations on Groups

Groups can perform simple aggregate functions

- `.mean()` : compute the average value of given columns for each group.

In [ ]:
```
groups[['Price', 'Volume']].mean()
```

- `.max()` : shows the maximum for given columns for each group.

In [ ]:
```
groups[['Price', 'Volume']].max()
```

Aggregating Multiple Functions at Once

The `.aggregate()` methods obtains multiple aggregate functions to summarize data.

In [ ]:
```
groups[['Price', 'Volume']].aggregate(['min', 'max']).round(2)
```

4.2 Grouping by multiple columns

In [ ]:
```
groups = df.groupby(['Quarter', 'Stock'])
```

In [ ]:
```
groups.size()
```

4.3 Advanced grouping example

Aggregations on Multiple Groups

After grouping by multiple column, one can perform multiple operations, such as summing and calculating the mean for specific columns

In [ ]:
```
groups[['Price', 'Volume']].aggregate(['sum', 'mean'])
```

Calculating market share

Goal: calculate each stock's market share in a specific quarter.

- Compute the market share of each stock by dividing its market cap by the total market cap of all stocks in that quarter.

In [ ]:

```
df['Market_Share'] = df.groupby('Quarter')['Market_Cap'].transform(lamb
df[['Stock', 'Quarter', 'Market_Share']]
```

Step by step

- Step 1: Grouping by `Quarter`

We are using the `groupby()` function to group the data by the `Quarter` column:

```
df.groupby('Quarter')
```

This groups the rows in the `DataFrame` based on the values in the `Quarter` column. This means that all stocks from the same quarter will be grouped together.

- Step 2: Applying the Aggregation

Next, we focus on the `Market_Cap` column within each quarter. This is done using:

```
df.groupby('Quarter')['Market_Cap']
```

Here, we are selecting the `Market_Cap` values for each group (each quarter). We now want to calculate the market share for each stock within its respective quarter.

- Step 3: Calculating Market Share with `transform()`

The custom operation is a `lambda` function that computes the market share:

```
lambda x: x / x.sum()
```

This `lambda` function divides each stock's market capitalization (`x`) by the total market capitalization of all stocks in the same quarter (`x.sum()`).

- `x` represents the `Market_Cap` values for the group (all stocks in a given quarter).
- `x.sum()` calculates the total market capitalization for that quarter.
- The `lambda` function then divides each stock's market capitalization by the total, which gives the proportion (or market share) of that stock relative to the total market capitalization in the quarter.

- Step 4: Assigning the Market Share to a New Column

The result of this operation is assigned to a new column in the DataFrame called Market_Share:

```
df['Market_Share'] = ...
```

Now, each row in the DataFrame will have a Market_Share value representing the percentage of the total market capitalization that each stock holds within its respective quarter.

🚦 Checkpoint

Task:

1. Create a `DataFrame` with columns: `Employee`, `Department`, `Salary` from

```
data = {
    'Employee': ['John', 'Jane', 'Peter', 'Lucy'],
    'Department': ['HR', 'HR', 'IT', 'IT'],
    'Salary': [50000, 60000, 70000, 80000]
    }
```

2. Calculate the average salary per department.

5. Complex Selection

In `pandas`, **data selection** involves formulating **conditions** based on column values and combining multiple conditions logically.

Initialization

In [ ]:

```
import numpy as np

# Creating a DataFrame with random financial data
data = {
    'Transaction_Amount': np.random.uniform(-500, 1500, 10),  # Random
    'Interest_Rate': np.random.uniform(0.01, 0.15, 10),       # Random
    'Loan_Amount': np.random.uniform(1000, 20000, 10),         # Random
}

df = pd.DataFrame(data)
```

In [ ]:
```
df.info()  # Display information about the DataFrame
```

In [ ]:
```
df.head()  # Display the first few rows
```

5.1 Conditions

Conditions based on the values in the `DataFrame` columns:

1. **Single Condition**: Selecting transactions greater than a specified amount:

In [ ]:
```
condition1 = df['Transaction_Amount'] > 0
```

2. **Multiple Conditions**: Using logical operators to combine conditions:

- **AND** Condition: Selecting rows where the transaction amount is positive and the interest rate is less than 0.1:

In [ ]:
```
condition2 = (df['Transaction_Amount'] > 0) & (df['Interest_Rate'] < 0.
```

- **OR** Condition: Selecting rows where the transaction amount is positive or the interest rate is less than 0.05:

In [ ]:
```
condition3 = (df['Transaction_Amount'] > 0) | (df['Interest_Rate'] < 0.
```

3. **Condition on All Values**: Checking for all positive values in the DataFrame:

In [ ]:
```
condition4 = df > 0
```

5.2 Conditional Selection

Condition-based selection of data from the DataFrame.

- **Select rows where `Transaction_Amount` is greater than 0**:

In [ ]:
```
positive_transactions = df[df['Transaction_Amount'] > 0]
print (positive_transactions)
```

In [ ]:
```
condition1
```

- **Select rows where `Transaction_Amount` is positive and `Interest_Rate` is less than 0.1:**

In [ ]:
```
filtered_transactions = df[(df['Transaction_Amount'] > 0) & (df['Intere
print (filtered_transactions)
```

In [ ]:
```
df[condition2]
```

- **Select rows where either `Transaction_Amount` is positive or `Interest_Rate` is less than 0.05:**

In [ ]:
```
selected_transactions = df[(df['Transaction_Amount'] > 0) | (df['Intere
```

- **Select all positive values from the DataFrame**:

In [ ]:
```
df[df > 0]
```

🚦 Checkpoint

Task:

1. Create a `DataFrame` with columns `Age`, `Income` from

```
data = {
        'Age': [25, 35, 45, 50],
        'Income': [40000, 60000, 70000, 30000]
    }
```

2. Select rows where `Age > 30` and `Income > 50000`.

6. Concatenation, Join, Merge

Data manipulation often involves the need to **combine multiple datasets**.

Initialization

In [ ]:
```
loan_data = pd.DataFrame({
    'Customer_ID': ['A001', 'A002', 'A003', 'A004'],
    'Loan_Amount': [10000, 20000, 15000, 25000]
})

credit_scores = pd.DataFrame({
    'Customer_ID': ['A002', 'A004', 'A005'],
    'Credit_Score': [720, 680, 710]
})
```

In [ ]:
```
loan_data
```

In [ ]:
```
credit_scores
```

6.1 Concatenation

Concatenation or appending means adding rows from one DataFrame to another.

In [ ]:
```
# Concatenating the DataFrames
pd.concat([loan_data, credit_scores], sort=False)
```

- Remember that when concatenating, the index values are maintained unless specified otherwise. They can be reset `ignore_index=True` (more on this here).

In [ ]:
```
pd.concat([loan_data, credit_scores], ignore_index=True, sort=False)
```

6.2 Join

Joining allows to **combine two DataFrames** based on their **indices**.

- It is similar to SQL joins
- useful when combining datasets with matching shared keys.

Let's join the `loan_data` and `credit_scores` using `Customer_ID`.

In [ ]:

```
# Setting the 'Customer_ID' column as index for join
loan_data.set_index('Customer_ID', inplace=True)
credit_scores.set_index('Customer_ID', inplace=True)
```

In [ ]:

```
# Performing a left join
loan_data.join(credit_scores, how='left')
```

Join methods

There are **4 different approaches** to join `DataFrames`.

Each approach leads to a different behavior with regard to how index values and the corresponding data rows are handled:

Join Method	Description
`left`	Preserves index values from the left DataFrame.
`right`	Preserves index values from the right DataFrame.
`inner`	Preserves index values found in both DataFrames.
`outer`	Preserves all index values from both DataFrames.

In [ ]:
```
# Different types of joins
loan_data.join(credit_scores, how='right')
```

In [ ]:
```
loan_data.join(credit_scores, how='inner')
```

In [ ]:
```
loan_data.join(credit_scores, how='outer')
```

6.3 Merge

Merging `DataFrames` is similar to joining except it can be achieved on specific columns instead of only indeces.

- Useful when financial datasets have overlapping columns.

In [ ]:
```
# Resetting index for merging
loan_data.reset_index(inplace=True)
credit_scores.reset_index(inplace=True)
```

In [ ]:
```
# Merging based on the 'Customer_ID' column
pd.merge(loan_data, credit_scores, on='Customer_ID', how='inner')
```

As for **joins**, mergers can be done with the four different options:

- `left`
- `right`
- `inner`
- `outer`

In [ ]:
```
# Merge with outer join
pd.merge(loan_data, credit_scores, on='Customer_ID', how='outer')
```

Merging can also be done using different columns from each `DataFrame`

- `left_on`
- `right_on`

In [ ]:

```
# Sample DataFrame for loan data
loan_data = pd.DataFrame({
    'Loan_ID': ['L001', 'L002', 'L003', 'L004'],
    'Amount': [5000, 7000, 8000, 6000],
    'Branch_ID': ['B001', 'B002', 'B003', 'B004']
})

# Sample DataFrame for branch data
branch_data = pd.DataFrame({
    'Branch_Name': ['Main Branch', 'East Branch', 'West Branch', 'North
    'Manager': ['John', 'Sally', 'Mike', 'Anna'],
    'ID': ['B001', 'B002', 'B003', 'B005']  # Different name for Branch
})

# Merging using different columns from each DataFrame
# Merging on loan_data['Branch_ID'] and branch_data['ID']
merged_df = pd.merge(loan_data, branch_data, left_on='Branch_ID', right

print(merged_df)
```

🚦 Checkpoint

Task:

1. Create two `DataFrames`, one with `Customer_ID` and `Loan_Amount` and another with `Customer_ID` and `Credit_Score` from

```
loan_data = pd.DataFrame({
    'Customer_ID': ['A001', 'A002', 'A003'],
    'Loan_Amount': [10000, 15000, 20000]
    })

    credit_data = pd.DataFrame({
        'Customer_ID': ['A001', 'A002', 'A004'],
        'Credit_Score': [720, 650, 700]
    })
```

2. Merge the two `DataFrames`.
