Introduction to Pandas

Overview

In this handout, we will cover `pandas` basic data types and explore how to work with them. We will also explore sorting and filtering data, and finally, we will look at basic data exploration methods and how to save data to a file.

1 Introduction to `pandas`

`pandas` is Python's most widely-used library for data analysis and manipulation. It is popular among data analysts and machine learning practitioners because it provides a wide range of functions to quickly handle, clean, and analyze data. Many real-world datasets are messy, containing missing values and a mix of data types (e.g., names as strings, ages as numbers, dates, and more). `pandas` makes it easy to work with such data.

`pandas` is the go-to library for working with mixed data types!

The name `pandas` comes from the term *panel data*, which refers to data collected over time. Today, `pandas` is an essential part of the data analysis toolkit in Python, often used alongside other libraries like `matplotlib` for visualization or `scikit-learn` for machine learning.

This handout will introduce many of the key `pandas` functions and methods commonly used in data analysis. Keep in mind that `pandas` offers even more functionality, and as you gain experience using it, you will discover additional features that can help with more advanced analysis. You can explore over 3,700 documentation pages at https://pandas.pydata.org/pandas-docs/version/1.4.4/pandas.pdf.
To get started with `pandas`, you need to import the library like this:

Importing `pandas`

```
import pandas as pd
```

`pandas` offers two powerful data structures

- `Series`
- `DataFrame`

2 Introducing the `Series` Data Structure

A `Series` in `pandas` is a one-dimensional labeled (indexed) array capable of holding any data type (integers, strings, floats, etc.). You can think of a `Series` as a column in a table or a list of values with labels, called the `index`, that correspond to each value. Here is a structure of a `Series`:

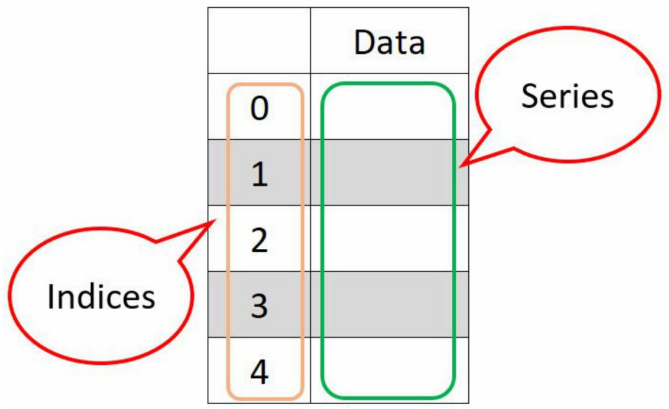


Figure 1: Structure of a `Series`

2.1 General syntax of `Series()` Function

`Series` can be created from a variety of inputs, such as Python lists, dictionaries, or even scalar values. This can be done using `Series()` function. The general syntax of the `Series` function is:

```
pd.Series(data, index=None, dtype=None, name=None)
```

The following table outlines describes each of these common arguments:

Argument	Required	Description
`data`	Yes	The list (or array-like structure) containing the values for the `Series`.
`index`	No	Custom labels to use for each element in the `Series`.
`dtype`	No	Specifies the data type for the elements in the `Series`.
`name`	No	Provides a name for the `Series` object.

Table 1: Explanation of arguments used in creating a `Series`.

`pandas Series`

A **pandas Series** is a list of values with labels, called an **index**, for each value. It's like a single column in a table and is useful for working with data because each value is easy to access by its label.

2.2 Creating a `Series` from a `list`

The simplest way to create a `Series` is from a `list` of values. `pandas` provides the `Series` function to do this.

Here's an example that represents game scores from a recent competition:

Creating a `Series` from a `list`

```
game_scores = pd.Series([100, 85, 78, 92, 88])
print(game_scores)
```

The output will look like this:

```
0 100
1 85
2 78
3 92
4 88
dtype:  int64
```

In the output, the left column shows the index (label), and the right column shows the values in the `Series`. By default, pandas assigns an integer index starting from `0`. Later, we will see how to customize our own indices.

2.3 Creating a `Series` from a Dictionary

You can also create a `Series` from a dictionary, where the dictionary keys become the index and the values become the `Series` data.

Here's an example of stock prices for different companies:

Creating a `Series` from a Dictionary

```
stock_prices = pd.Series({'Apple': 233.5,
                          'Google': 171.1,
                          'Amazon': 190.8,
                          'Tesla': 260
                         }
                        )
print(stock_prices)
```

The output will look like this:

```
Apple            233.5
Google           171.1
Amazon           190.8
Tesla            260.0
dtype:  float64
```

Here, the dictionary keys become the index for the `Series`, and the corresponding values become the data.

2.4 Custom Indices for a `Series`

You can specify custom indices when creating a `Series`. This is useful when your data has meaningful labels, like regions or categories.

Let's look at a sales data example:

Creating a `Series` with Custom Indices

```
sales_data = pd.Series([23000, 18500, 22000], index=['N. America', 'Europe', 'Asia'])
print(sales_data)
```

The output will look like this:

```
N. America     23000
Europe         18500
Asia           22000
dtype:  int64
```

As you can see, the custom labels (`'N. America'`, `'Europe'`, and `'Asia'`) are used as the `index` for the `Series`.

2.5 Accessing Data in a `Series`

You can access elements in a `Series` by using their index.[1] You can either use the default integer index or the custom labels.

Here is an example of accessing by default integer `index`:

Accessing Data by Integer Index

```
game_scores = pd.Series([100, 85, 78, 92, 88])

print(game_scores[1])  # The output would be 85
```

And here we access values by custom label `index`:

Accessing Data by Custom Label

```
sales_data = pd.Series(data = [23000, 18500, 22000],
                       index = ['N. America', 'Europe', 'Asia']
                      )
print(sales_data['Europe'])  # The output would be 18500
```

[1] This is very similar to accessing values of a dictionary using its keys.

Accessing Data in a `pandas Series`

You can access data in a **pandas Series** by using the label (index) or the position of the item. Use `Series[label]` to get a value by label, or `Series[position]` to get a value by position. This makes it easy to find specific data quickly.

2.6 Attributes of a `Series`

As a reminder, an `attribute` is a property or characteristic of an object in Python. In the case of a `Series`, attributes are built-in properties that provide useful information about the `Series`, such as its values or labels. A `Series` has two important attributes:

- `values`: Returns the values of the `Series`.
- `index`: Returns the labels of the `Series`.

  Let's look at the attributes of the `sales_data` `Series`:

Attributes of a `Series`

```
sales_data = pd.Series(data = [23000, 18500, 22000],
                       index = ['N. America', 'Europe', 'Asia'])

print('sales data values:')
print(sales_data.values)
print()  # added an empty line

print('sales data index:')
print(sales_data.index)
```

The output will be:

```
sales data values:
[23000 18500 22000]

sales data index:
Index(['N. America', 'Europe', 'Asia'], dtype='object')
```

3 Basic Operations on `Series`

Once you create a `Series`, you can perform basic operations on it. These operations allow you to manipulate and analyze the data efficiently.

3.1 Arithmetic Operations on `Series`

Arithmetic operations can be applied directly to a `Series`, performed element-wise.

Here's an example using game scores where we will add 10 points to each score:

Arithmetic Operations on `Series`

```
game_scores = pd.Series([100, 85, 78, 92, 88], name="Competition Scores")

updated_scores = game_scores + 10
print(updated_scores)
```

The output will be:

```
0 110
1 95
2 88
3 102
4 98
Name:  Competition Scores, dtype:  int64
```

In this example, 10 points were added to each score. Notice that we gave the `Series` a `name`. While this is optional, it is a good practice, as we may use it later.[2]

Arithmetic operations such as `addition`, `subtraction`, `multiplication`, and `division` are applied to each element in the `Series`.

3.2 Summarizing Data with `Series`

`pandas` provides many built-in methods (functions) to summarize data in a `Series`. One of the most useful is `.describe()`, which provides basic descriptive statistics like `count`, `mean`, `standard deviation`, and more.

Here is an example:

Using `.describe()` to Summarize a `Series`

```
game_scores = pd.Series([100, 85, 78, 92, 88], name="Competition Scores")
print(game_scores.describe())
```

The output will look like this:

```
count    5.000000
mean    88.600000
std      8.602325
min     78.000000
25%     85.000000
50%     88.000000
75%     92.000000
max    100.000000
Name:  Competition Scores, dtype:  float64
```

[2] One use of the `name` is that if you add a `Series` to a `DataFrame` as a column, the `Series` `name` will automatically become the column name in the `DataFrame`.

The `.describe()` method is a quick way to get a summary of your data.

`Series.describe()` Method

The `Series.describe()` method provides a quick summary of the data in a **pandas Series**. It returns key statistics like **count**, **mean**, **standard deviation**, **min**, **max**, and **quartiles**. This is helpful for understanding the distribution and spread of numerical data.

3.3 Converting a `Series` to a List or a Dictionary

So far, we have learned how to create a `Series` from a `list` or a `dict`. Similarly, we can easily convert a `Series` to other formats, such as a `list` or `dict`, which is useful when working with libraries or Python functions that require standard data structures like `list` or `dict`.

3.3.1 `.tolist()` Method

The `.tolist()` method converts the values in a `Series` to a `list`.

Converting a `Series` to a `list`

```
sales_data = pd.Series(data = [23000, 18500, 22000],
                       index = ['N. America', 'Europe', 'Asia'])
sales_list = sales_data.tolist()
print(sales_list)
```

```
[23000, 18500, 22000]
```

3.3.2 `.to_dict()` Method

The `.to_dict()` method converts a `Series` to a `dictionary`, with the `index` as the keys and the values of the `Series` as the dictionary values.

Converting a `Series` to a Dictionary

```
sales_dict = sales_data.to_dict()
print(sales_dict)
```

```
{'N. America': 23000, 'Europe': 18500, 'Asia': 22000}
```

3.4 From `Series` to `DataFrame`

A `Series` in `pandas` is a powerful data structure for representing one-dimensional data with labels (indices). However, most real-world datasets contain multiple columns, which is why we'll transition to the `DataFrame`, a two-dimensional data structure in `pandas` designed to handle multi-column datasets.

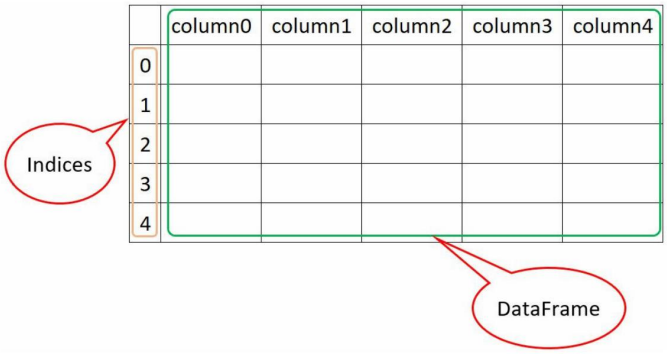


Figure 2: Structure of a `DataFrame`

4 Introduction to `DataFrame`

In real-world data, most datasets have more than one column. For such datasets, `pandas` offers an efficient two-dimensional data structure called a `DataFrame`. A `DataFrame` is essentially a collection of `Series` objects, where each column is a `Series`. It has both rows and columns, and each column in a `DataFrame` can contain different types of data (e.g., numbers, text, date, etc.).

The structure of a `DataFrame` looks like this:
Each row in the `DataFrame` has an index (label), and each column has a name (header). A `DataFrame` is powerful because it allows you to work with tabular data in a flexible and efficient way.

`DataFrame` works similarly to a `spreadsheet` in `MS Excel`:

- Columns in a `DataFrame` are like the columns in an Excel sheet, each having a column name (or header).
- Rows in a `DataFrame` are like the rows in Excel, each identified by an index number.
- You can perform operations such as sorting, filtering, and summarizing data in a `DataFrame`, just as you would in Excel.

4.1 General Syntax of the `DataFrame()` Function

A `DataFrame` in `pandas` can be created from multiple data sources such as dictionaries, lists of dictionaries, or arrays. The general syntax for creating a `DataFrame` is:

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

The table below describes each of these key arguments:

4.2 Creating a `DataFrame` from a Dictionary

Let's start by creating a `DataFrame` from a dictionary. In this example, we will store the closing stock prices of several companies over four days:

Creating a `DataFrame` from a Dictionary

```
price_dict = {
    "MSFT": [222.59, 223.94, 221.02, 222.75],
```

Argument	Required	Description
`data`	Yes	The primary data for the `DataFrame`. This can be a dictionary, list of dictionaries, two-dimensional array, or another `DataFrame`.
`index`	No	Custom row labels for the `DataFrame`. If not provided, integer indices are automatically assigned.
`columns`	No	Labels for each column in the `DataFrame`. If not specified, default column labels are assigned.
`dtype`	No	Specifies the data type for each column. If omitted, data types are inferred automatically.

Table 2: Explanation of arguments used in creating a `DataFrame`.

```
    "BABA": [260.43, 255.83, 256.18, 222.00],
    "META": [272.79, 267.09, 268.11, 267.40],
    "AMZN": [3206.18, 3206.52, 3185.27, 3172.69],
    "TSLA": [649.86, 640.34, 645.98, 661.77],
    "AAPL": [128.23, 131.88, 130.96, 131.97]
}
prices_df = pd.DataFrame(price_dict)
display(prices_df)
```

The output will look like:

	MSFT	BABA	META	AMZN	TSLA	AAPL
0	222.59	260.43	272.79	3206.18	649.86	128.23
1	223.94	255.83	267.09	3206.52	640.34	131.88
2	221.02	256.18	268.11	3185.27	645.98	130.96
3	222.75	222.00	267.40	3172.69	661.77	131.97

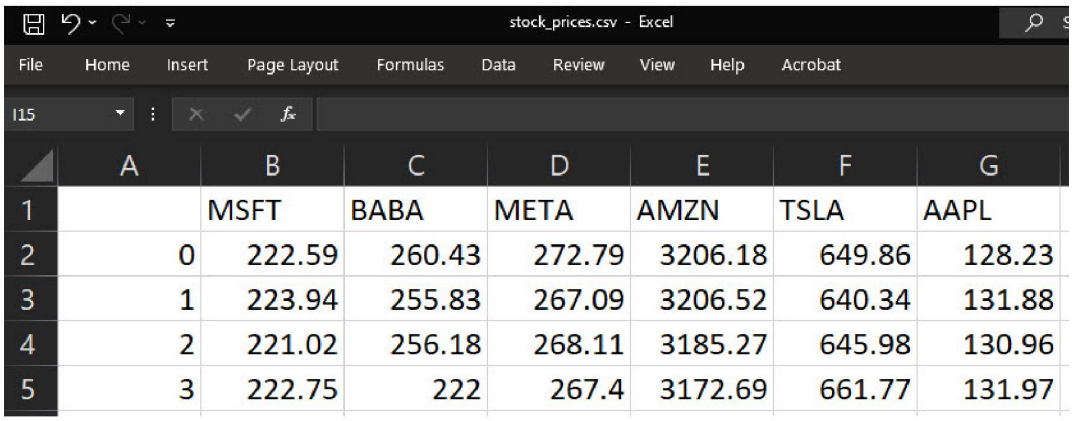
Note: We used `display()` function instead of `print()` because `display()` function provides a more visually formatted view of a `DataFrame`, especially in `Jupyter` environment.

pandas DataFrame

A **pandas DataFrame** is a two-dimensional table with labeled rows and columns, similar to a spreadsheet. Each column can hold different types of data (like numbers, text, or dates). DataFrames are widely used for data manipulation and analysis, as they allow for easy access, filtering, and modification of data.

4.3 Creating a `DataFrame` from a CSV File

One of the most common ways to create a `DataFrame` is by loading data from a `CSV` file. Below is a screenshot of a small sample `CSV` file named `stock_prices.csv`:



	A	B	C	D	E	F	G
1		MSFT	BABA	META	AMZN	TSLA	AAPL
2	0	222.59	260.43	272.79	3206.18	649.86	128.23
3	1	223.94	255.83	267.09	3206.52	640.34	131.88
4	2	221.02	256.18	268.11	3185.27	645.98	130.96
5	3	222.75	222	267.4	3172.69	661.77	131.97

Figure 3: Screenshot of `stock_prices.csv`

To load this file as a `DataFrame` in Python, use the `read_csv()` method from `pandas`, as shown below:

Creating a `DataFrame` from a `CSV` file

```
prices_df = pd.read_csv('stock_prices.csv')

display(prices_df.head())
```

This code will import the CSV data into a `DataFrame` named `prices_df`.

A Note on `csv` Files

One of the most common ways to create a `DataFrame` is by reading data from a `csv` file.

What is a `csv` File?

`csv` stands for **Comma Separated Values**. It's a file format designed to store tabular data as plain text, where each line corresponds to a row, and a comma separates each value. `csv` files are widely used for data exchange they are compatible with many tools and software.

Example of a `csv` File Structure:

```
Column1, Column2, Column3
Value1, Value2, Value3
Value4, Value5, Value6
```

You can open a `csv` file in any text editor, such as Notepad or MS Word, or import it into spreadsheet applications like Excel. Here are a few important properties of `csv` files:

- **Simple and Universal**: `csv` files are plain text and compatible across all platforms (Windows, Mac, etc.).
- **Data Types**: `csv` files store only basic data types: `strings` and `numbers`. They cannot retain complex features like formulas, graphs, or special formatting commonly found in Excel files. In other words, if you save an Excel file containing formulas as a `csv`, the formula calculations will not be preserved—only the values at the time of export will remain.
- **Headers**: By default, `pandas` assumes that the first row in a `csv` file represents the column names for the `DataFrame`. If the `csv` file does not have headers, you can specify `header=None` in `pd.read_csv()` to instruct `pandas` to treat the first row as regular data rather than as column names.
- When importing a `csv` into a `DataFrame`, `pandas` automatically tries to detect data types.

`csv` versus `xlsx` File Formats

While `csv` files are common for data analysis, they serve a different purpose than Excel files (`.xlsx` format). An Excel file may contain multiple worksheets, each with its own formatting, complex data types, and features like graphs, formulas, and macros. By contrast, a `csv` file is a simple, flat text file with only raw data. This makes `csv` files ideal for data storage and transfer, but it lacks the formatting flexibility and additional features that Excel offers.

4.4 Creating a `DataFrame` from an Online Data Repository

`pandas` can read data from web links, providing access to datasets from popular online repositories such as:

- **UCI Machine Learning Repository** (https://archive.ics.uci.edu/): A well-established source for machine learning datasets across multiple disciplines.
- **Kaggle** (https://www.kaggle.com/datasets): An extensive collection of datasets in fields such as finance,

healthcare, and social media.

- **data.gov** (https://data.gov/): The U.S. Government's open data portal, featuring extensive datasets on a range of topics, from education to transportation.
- **Google Dataset Search** (https://datasetsearch.research.google.com/): While not a repository, this website aggregates datasets from various sources such as government, academic, and private institutions, for public use.

Here's an example of initializing a `DataFrame` by directly importing a dataset from a URL:

Creating a `DataFrame` from `data.gov`

```
URL = 'https://data.cityofnewyork.us/api/views/c3uy-2p5r/rows.csv?accessType=DOWNLOAD'
nyc_air_quality_df = pd.read_csv(URL)
```

This code snippet creates a `DataFrame` named `nyc_air_quality_df` from the `data.gov` repository, containing New York City air quality surveillance data.

5 Customizing `pandas` Display Options

While the `prices_df` used in this handout is small, with only 5 rows and fewer than 10 columns, real-world datasets are often much larger. Viewing large data in a structured and readable format can be challenging. `pandas` provides several options to control the display of data to make working with extensive datasets easier. Here are some of the most common display settings:

5.1 `max_rows` and `max_columns`

When working with large DataFrames, `max_rows` and `max_columns` control the number of rows and columns displayed in the output. By default, `pandas` shows up to 60 rows and 20 columns. You can adjust these settings to display more or fewer rows and columns as needed.

Setting `max_rows` and `max_columns`

```
import pandas as pd

pd.options.display.max_rows = 10    # Display up to 10 rows
pd.options.display.max_columns = 5  # Display up to 5 columns
```

This setting is useful to prevent excessive scrolling when working with large datasets.

5.2 `float_format` and `precision`

`precision` controls the number of decimal places displayed for floating point values. For instance, you might want to limit floating-point numbers to two decimal places for a cleaner display:

Setting `precision`

```
pd.options.display.precision = 2  # Set precision to 2 decimal places
```

These options help display financial data, scientific measurements, or any data where the precision level affects readability. Please note that these display options only affect how data is visually represented in the output; they do not alter the actual data or the underlying numerical values stored in the `DataFrame`. The original precision of the data remains unchanged for calculations and analysis. You can always reset these options back to their defaults by setting them to `None`.

6 Accessing Data in a `DataFrame`

So far, we have covered the two data structures in `pandas`: the `Series` and the `DataFrame`. In this section, we'll learn the structure of a `DataFrame` followed by ways to access and interact with its various components.

6.1 Understanding `index` and `columns` Attributes

In a `DataFrame`, the `index` and `columns` attributes allow us to access and view the row and column labels.

- The `index` attribute stores row labels (or indices). By default, when we create a `DataFrame`, the index starts from 0 and increments by 1.
- The `columns` attribute stores the names of each column. The column names are derived from the data or assigned manually.

  Here's are a few examples using the `prices_df` `DataFrame` we created earlier:

	MSFT	BABA	META	AMZN	TSLA	AAPL
0	222.59	260.43	272.79	3206.18	649.86	128.23
1	223.94	255.83	267.09	3206.52	640.34	131.88
2	221.02	256.18	268.11	3185.27	645.98	130.96
3	222.75	222.00	267.40	3172.69	661.77	131.97

  To view the `index` and `columns` of the `DataFrame`:

`index` and `columns` attributes

```
print('index:')
print(prices_df.index)
print()                 # an empty line
print('columns:')
print(prices_df.columns)
```

  This will return:

```
index:
RangeIndex(start=0, stop=4, step=1)

columns:
Index(['MSFT', 'BABA', 'FB', 'AMZN', 'TSLA', 'AAPL'], dtype='object')
```

note that

- The `RangeIndex` represents a default integer-based index starting from 0 and a step of 1.
- The `Index` lists the column names of the `DataFrame` as string labels, indicating each column's name.

As you may have noticed, `prices_df.index` and `prices_df.columns` each return an `Index` data type that functions similarly to a Python `list`. This means you can access and manipulate them just like a standard Python `list`.

To customize the row `index`, you can assign specific labels like `day_1, day_2, ...` or even dates such as `Dec_20, Dec_21, ...`. Here is an example:

Changing the index to custom labels

```
prices_df.index = ['day_'+str(i) for i in range(1,5)]
display(prices_df)
```

This will output:

	MSFT	BABA	META	AMZN	TSLA	AAPL
day_1	222.59	260.43	272.79	3206.18	649.86	128.23
day_2	223.94	255.83	267.09	3206.52	640.34	131.88
day_3	221.02	256.18	268.11	3185.27	645.98	130.96
day_4	222.75	222.00	267.40	3172.69	661.77	131.97

In a similar way, you can set or change column headers using the `columns` attribute. For example, to rename columns in `prices_df`:

Renaming Multiple Columns

```
prices_df.columns = ['Microsoft', 'AliBaba', 'META', 'Amazon', 'Tesla', 'Apple']
```

This code directly updates the column names in the `prices_df` DataFrame. When renaming columns, ensure that the new column names match the original order of the columns in the `DataFrame`.

Accessing Data in a `pandas DataFrame`

You can access data in a **pandas DataFrame** by selecting rows, columns, or specific cell values. Use

- `DataFrame['column_name']` to access a column,
- `DataFrame.loc[row_label]` for a row by label, or
- `DataFrame.iloc[row_position, column_position]` for a specific cell by position. This allows for flexible and efficient data retrieval.

6.2 Accessing Columns

Assume the following `DataFrame` called `prices_df`:

	MSFT	BABA	META	AMZN	TSLA	Apple Inc
day_1	222.59	260.43	272.79	3206.18	649.86	128.23
day_2	223.94	255.83	267.09	3206.52	640.34	131.88
day_3	221.02	256.18	268.11	3185.27	645.98	130.96
day_4	222.75	222.00	267.40	3172.69	661.77	131.97

6.2.1 Accessing one column

Each column in a `DataFrame` is a `Series` and we can access a column in one of the following two ways:

- **Using `dot` notation**: in the form of `DataFrame's name.column name`
- **Using square brackets `[]`**[3]

Using the `dot` notation:

You can access a column in a `DataFrame` by using the dot notation, as long as the column name is a valid Python identifier (i.e., it doesn't contain spaces or special characters and doesn't begin with a number).

Here's an example accessing the `AMZN` column using dot notation:

Accessing the `'AMZN'` Column using Dot Notation

```
prices_df.AMZN
```

This produces the following output:

```
day_1  3206.18
day_2  3206.52
day_3  3185.27
day_4  3172.69
Name:  AMZN, dtype:  float64
```

This method has three limitations:

- **Column name must be a valid Python identifier.** In practical datasets, column names often contain spaces or special characters. This can cause issues when using dot notation. In the above `DataFrame`:

[3] Remember that we used `[]` for accessing components of a `str`, `list`, `tuple`, and a `dictionary`.

Dot Notation with Column Name with a Space (`Apple Inc`)

```
prices_df.Apple Inc
```

leads to

```
SyntaxError: invalid syntax
```

- **If the column name is a Python keyword, dot notation will not work.** For example, if a column is named `class` or `return` (reserved keywords in Python), dot notation will raise:

```
SyntaxError: invalid syntax
```

- **If the column name you are trying to access is stored in a variable**: This method will not work. Here is an example:

Using dot notation with a variable name

```
company_name = 'MSFT'
prices_df.company_name
```

This will lead to

```
AttributeError: 'DataFrame' object has no attribute 'company_name'
```

Because `pandas` will look for a column with the literal name of the `company_name`, rather than its value.

To avoid these issues, the fail-safe method of accessing a column is using square bracket `[]` notation.

Using square brackets `[]`

The general syntax is `DataFrame['column name']`. In the above `DataFrame`:

Accessing a Column using `[ ]`

```
prices_df['Apple Inc']
```

The output will look like:

```
day_1  128.23
day_2  131.88
day_3  130.96
day_4  131.97
Name:  Apple Inc, dtype:  float64
```

6.2.2 Accessing multiple columns

We can easily access multiple columns in any order from a `DataFrame` by providing a `list` of column names. The general syntax for accessing multiple columns is:

```
DataFrame[['column_name1', 'column_name2', ...]]
```

For instance, in the previously defined `prices_df`:

Accessing `Apple Inc`, `MSFT`, and `BABA` in this order

```
selected_columns = prices_df[['Apple Inc', 'MSFT', 'BABA']]
display(selected_columns)
```

produces

	Apple Inc	MSFT	BABA
day_1	128.23	222.59	260.43
day_2	131.88	223.94	255.83
day_3	130.96	221.02	256.18
day_4	131.97	222.75	222.00

6.3 Accessing Rows

You can access rows using the `loc[]` or `iloc[]` indexing attributes[4]

- `loc[]` uses custom labels (like dates).
- `iloc[]` uses integer-based indexing (zero-based).

In the previously defined `price_df`:

Accessing a row using `loc[]`

```
prices_df.loc['day_2']  # Access using custom label (index)
```

yields the following `Series`

```
MSFT         223.94
BABA         255.83
META         267.09
AMZN        3206.52
TSLA         640.34
Apple Inc    131.88
Name:  day_2, dtype:  float64
```

You can also use slicing notation with `loc[]` to access consecutive rows, as shown below:

[4] Notice that `loc` and `iloc` are not functions; if they were, they would require parentheses like `()` after their names.

Accessing multiple consecutive rows using `loc[ ]`

```
prices_df.loc['day_2':'day_4']
```

yields the following `DataFrame`

	MSFT	BABA	META	AMZN	TSLA	Apple Inc
day_2	223.94	255.83	267.09	3206.52	640.34	131.88
day_3	221.02	256.18	268.11	3185.27	645.98	130.96
day_4	222.75	222.00	267.40	3172.69	661.77	131.97

Note: When using slices with labels in `loc[]`, the specified range ***includes*** the end label (in this example, `'day_4'`).
You can also access non-consecutive rows by including their indices in a `list`, as shown below:

Accessing non-consecutive rows using `loc[ ]`

```
prices_df.loc[['day_2','day_4']]
```

yields the following `DataFrame`

	MSFT	BABA	META	AMZN	TSLA	Apple Inc
day_2	223.94	255.83	267.09	3206.52	640.34	131.88
day_4	222.75	222.00	267.40	3172.69	661.77	131.97

6.3.1 Accessing Specific Columns of Certain Rows with `loc[ ]`

In large datasets with hundreds of columns, you may need to access specific columns for certain rows. This can be achieved using `loc[]`, which allows you to select specific columns by providing both row and column labels.

For example, to access only the `MSFT` and `AMZN` columns for `day_2` through `day_4`:

Selecting Specific Rows and Columns with `loc[ ]`

```
prices_df.loc['day_2':'day_4', ['MSFT', 'AMZN']]
```

	MSFT	AMZN
day_2	223.94	3206.52
day_3	221.02	3185.27
day_4	222.75	3172.69

6.3.2 Conditional Filtering with `loc[ ]`

`loc[]` can be used with conditional statements to filter rows based on certain criteria.

For example, to select rows where the value of `TSLA` is greater than 650:

Filtering Rows with Condition using `loc[ ]`

```
filtered_df = prices_df.loc[prices_df['TSLA'] > 650]
display(filtered_df)
```

This will yield only the rows where `TSLA` exceeds 650; yielding:[5]

	MSFT	BABA	META	AMZN	TSLA	Apple Inc
3	222.75	222.00	267.40	3172.69	661.77	131.97

6.3.3 Accessing Values in One Column Based on a Condition in Another Column

The `loc[ ]` method allows you to access values in one column based on specific conditions in another column.

For example, to retrieve `Apple Inc` prices for days where `TSLA` is above 650:

Accessing Values Based on Condition in Another Column

```
high_tsla_apple = prices_df.loc[prices_df['TSLA'] > 650, 'Apple Inc']
display(high_tsla_apple)
```

This will return only the `Apple Inc` prices where `TSLA` is greater than 650, yielding:

	Apple Inc
day_1	128.23
day_4	131.97

This approach enables you to retrieve targeted data from one column while applying filters to another column.

You can also apply multiple conditions. For instance, to access `AMZN` prices where `TSLA` is above 640 and `BABA` is below 260:

Accessing Values with Multiple Conditions

```
specific_amzn_prices = prices_df.loc[(prices_df['TSLA'] > 640) &
                                     (prices_df['BABA'] < 260), 'AMZN']
display(specific_amzn_prices)
```

The output will display only the `AMZN` prices meeting both conditions:

	AMZN
day_3	3185.27

[5] We will explore filtering in more detail in the section on Boolean indexing.

6.3.4 Modifying Data with `loc[ ]`

`loc[ ]` is also useful for modifying specific entries or subsets in a `DataFrame`. For example, to update the value of `BABA` on `day_3`:

Modifying a Value using `loc[ ]`

```
prices_df.loc['day_3', 'BABA'] = 260.00
display(prices_df)
```

This will replace the existing value of `BABA` on `day_3` with 260.00.

6.4 Accessing Rows with `iloc[ ]`

Accessing rows using `iloc` is similar to using `loc`, but instead of custom labels, you use the 0-based index positions of the rows.

Try the following example to see the results:

Accessing row(s) using `iloc[ ]`

```
display(prices_df.iloc[1])
display(prices_df.iloc[1:4])
display(prices_df.iloc[[1, 3]])
```

Note: When using slices with 0-based indexes in `iloc[ ]`, the specified range ***excludes*** the end index.
One more reminder that when using slices with `loc[ ]`, the specified range ***includes*** the end label. In contrast, `iloc[ ]` uses 0-based indexing, and the specified range ***excludes*** the end index.

6.5 Accessing Specific Cells

You can access a specific cell using `at[ ]`, and `iat[ ]` indexing attributes

- `at[ ]` for custom indexes (labels)
- `iat[ ]` for integer indexes

The row and column indices must be separated in each case by a comma `,`. The first index (or label) is for the row and the next one is for the column.

Here are a few examples

Accessing Specific Cells

```
print(f"At day_2, price of MSFT: {prices_df.at['day_2', 'MSFT']}")
print(f"Price on 3rd row and 5th column: {prices_df.iat[2, 4]}")
```

yields

```
At day_2, price of MSFT: 223.94
Price on 3rd row and 5th column:  645.98
```

7 Modifying a `DataFrame`

Assume the following `DataFrame` called `sales_df`:

	Item	Type	Order_ID	Unit Price	Quantity	Discount
0	Laptop	Electronics	A	2200	1	0.0
1	Chair	Furniture	B	350	5	0.1
2	Desk	Furniture	C	450	3	0.0
3	Phone	Electronics	D	1400	2	0.15
4	Jacket	Clothing	E	120	10	0.2

This `DataFrame` represents a sample sales record with details for individual orders, including *Item*, *Type*, *Order_ID*, *Unit Price*, *Quantity*, and *Discount*. In this section, we'll learn how to modify the content of a `DataFrame` using this example as a reference.

7.1 Renaming Columns using `rename()` Method

Consistency in column names helps make your `DataFrame` easier to work with, especially when dealing with larger datasets or collaborating with others. Using a uniform format—such as avoiding spaces in column names, using underscores (_) instead of spaces, and avoiding special characters—makes code cleaner and more readable.

In the example below, we will rename columns to follow these guidelines, changing *Item* to *Product*, *Type* to *Category*, and *Unit Price* to *Unit_Price*.

Renaming Columns in `sales_df`

```
sales_df = sales_df.rename(columns={'Item': 'Product',
                                    'Type': 'Category',
                                    'Unit Price': 'Unit_Price'
                                   }
                           )
display(sales_df)
```

As you can see, the old and new column names are provided as key-value pairs in a dictionary, formatted as `'old_name': 'new_name'`. The updated `DataFrame` with the renamed columns is shown below:

	Product	Category	Order_ID	Unit_Price	Quantity	Discount
0	Laptop	Electronics	A	2200	1	0.0
1	Chair	Furniture	B	350	5	0.1
2	Desk	Furniture	C	450	3	0.0
3	Phone	Electronics	D	1400	2	0.15
4	Jacket	Clothing	E	120	10	0.2

7.2 Setting the `index` Column using `set_index()` Method

In the `sales_df` DataFrame above, the first column is a default 0-based index without a column header, which doesn't add much context to the data. Instead, we can set the `Order_ID` column as the index, allowing for more

intuitive access to rows based on specific order numbers, enhancing readability and usability.

Note: While `pandas` does not require the index column to contain unique values, having unique indices is generally recommended, as it simplifies data manipulation and access, especially when filtering or merging `DataFrame`.[6]

To set the `Order_ID` column as the index, we can use `set_index()` method as following:

Setting the `index` column

```
sales_df = sales_df.set_index('Order_ID')
display(sales_df)
```

After setting `Order_ID` as the index, the updated `DataFrame` will look like this:

Order_ID	Product	Category	Quantity	Unit_Price	Discount
A	Laptop	Electronics	1	2200	0.0
B	Chair	Furniture	5	350	0.1
C	Desk	Furniture	3	450	0.0
D	Phone	Electronics	2	1400	0.15
E	Jacket	Clothing	10	120	0.2

`pandas` automatically places the `index` column on the far left, making it easier to reference and identify rows by their index values.

`DataFrame.set_index()` Method

The `DataFrame.set_index()` method allows you to set one or more columns as the index (labels) of a **pandas DataFrame**. This changes how rows are labeled, making it easier to locate data based on those labels. Use `inplace=True` to modify the DataFrame directly.

7.3 Rename `index` Values using `rename()` Method

Similar to renaming `columns`, you can rename `index` values using `rename()` method and referencing `index` parameter.

For instance, suppose we wish to replace the current index labels `A`, `B`, `C`, `D`, and `E` with numeric order identifiers. The following code achieves this:

Renaming Row Indexes

```
sales_df = sales_df.rename(index={'A': 1001,
                                  'B': 1002,
                                  'C': 1003,
                                  'D': 1004,
                                  'E': 1005
                                 }
```

[6] more on this later

```
                            )
display(sales_df)
```

After renaming the rows, the updated `DataFrame` is as follows:

Order_ID	Product	Category	Quantity	Unit_Price	Discount
1001	Laptop	Electronics	1	2200	0.0
1002	Chair	Furniture	5	350	0.1
1003	Desk	Furniture	3	450	0.0
1004	Phone	Electronics	2	1400	0.15
1005	Jacket	Clothing	10	120	0.2

7.4 Replacing Values in a `DataFrame` with `replace()`

So far, we have used the `rename()` method primarily for renaming values in the index and column headers. To replace specific values within the data cells of a `DataFrame`, the `replace()` method is more appropriate. This method is highly versatile and can take a variety of inputs, with one of the most common formats being a dictionary in the form of `old_value: new_value`.

The following example shows replacing values in `Category` column.

Replacing Values in the 'Category' Column

```
sales_df['Category'] = sales_df['Category'].replace({'Electronics': 'Tech',
                                                     'Furniture': 'Office Supplies'
                                                    }
                                                   )
display(sales_df)
```

This will produce a new `DataFrame` with the updated values in the `Category` column:

Order_ID	Product	Category	Quantity	Unit_Price	Discount
1001	Laptop	Tech	1	2200	0.0
1002	Chair	Office Supplies	5	350	0.1
1003	Desk	Office Supplies	3	450	0.0
1004	Phone	Tech	2	1400	0.15
1005	Jacket	Clothing	10	120	0.2

7.5 Reordering Columns in a `DataFrame`

To rearrange the columns of a `DataFrame`, you can specify a new order by listing the columns in the desired sequence. The syntax for reordering columns is

```
DataFrame = DataFrame[[new_column_order]]
```

In this example, we'll reorder the columns so they appear as `Order_ID`, `Category`, `Product`, `Unit_Price`, `Quantity`, and `Discount`.

Reordering Columns in a `DataFrame`

```
new_column_order = ['Order_ID', 'Category',
                    'Product', 'Unit_Price',
                    'Quantity', 'Discount'
                   ]
sales_df = sales_df[new_column_order]
display(sales_df)
```

The `DataFrame` with the reordered columns is shown below:

Order_ID	Category	Product	Unit_Price	Quantity	Discount
1001	Tech	Laptop	2200	1	0.0
1002	Office Supplies	Chair	350	5	0.1
1003	Office Supplies	Desk	450	3	0.0
1004	Tech	Phone	1400	2	0.15
1005	Clothing	Jacket	120	10	0.2

7.6 Adding a Column

You can add a new column to a `DataFrame` by assigning it directly using the syntax:

`DataFrame_name['new_column_name'] = calculation_for_new_column`

For example, we can add a new column for Total Before Tax (`Total_B_Tax`) by calculating the total cost before tax with the discount applied. To do this, we multiply `Quantity` by `Unit_Price` and then apply the `Discount` for each row. Here is how:

Adding `Total_B_Tax` with Discount

```
sales_df['Total_B_Tax'] = sales_df['Quantity'] * \
                          sales_df['Unit_Price'] * \
                          (1 - sales_df['Discount'])
display(sales_df)
```

Note: As we have seen in week 1's material, to split a long line in Python, we can use a backslash `\` for line continuation.
This element-wise operation calculates the total cost before tax and after applying the discount for each order, yielding:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_B_Tax
1001	Tech	Laptop	2200	1	0.0	2200.0
1002	Office Supplies	Chair	350	5	0.1	1575.0
1003	Office Supplies	Desk	450	3	0.0	1350.0
1004	Tech	Phone	1400	2	0.15	2380.0
1005	Clothing	Jacket	120	10	0.2	960.0

You can also perform arithmetic operations on one single column.

For example, to calculate the total cost after an 8% tax, we can multiply the `Total_B_Tax` column by 1.08. This will create a new column called `Total_A_Tax`.

Adding `Total_A_Tax` with Discount

```
sales_df['Total_A_Tax'] = sales_df['Total_B_Tax'] * 1.08
display(sales_df)
```

This calculation is performed element-wise, applying the tax rate to each row's `Total_B_Tax` and yields:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_B_Tax	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2200	2376.0
1002	Office Supplies	Chair	350	5	0.1	1575	1701.0
1003	Office Supplies	Desk	450	3	0.0	1350	1458.0
1004	Tech	Phone	1400	2	0.15	2380	2570.4
1005	Clothing	Jacket	120	10	0.2	960	1036.8

7.7 Conditional Updates

In many cases, you may want to modify values in a `DataFrame` based on certain conditions. For instance, adjusting discount values for specific product categories or applying updates to prices above a certain threshold. You can accomplish this with using `loc[]`, which allows access to rows, as we have seen before.

For example, let's say we want to increase the `Discount` for all `Office Supplies` items to 0.2. We can achieve this by specifying the condition within `loc[]`, as shown below:

Updating `Discount` for Office Supplies Items

```
sales_df.loc[sales_df['Category'] == 'Office Supplies', 'Discount'] = 0.2
display(sales_df)
```

This code updates the `Discount` column for all rows where the `Category` is `Office Supplies` yielding

Order_ID	Product	Category	Unit_Price	Quantity	Discount
1001	Laptop	Tech	2200	1	0.00
1002	Chair	Office Supplies	350	5	0.20
1003	Desk	Office Supplies	450	3	0.20
1004	Phone	Tech	1400	2	0.15
1005	Jacket	Clothing	120	10	0.20

Another example: Suppose we want to apply a 5% discount to items with a `Unit_Price` above 1000. We can do so by adding this condition within `loc[]`:

Applying `Discount` to Expensive Items

```
sales_df.loc[sales_df['Unit_Price'] > 1000, 'Discount'] = 0.05
display(sales_df)
```

This command selects rows where the `Unit_Price` is greater than 1000 and updates the `Discount` value in those rows to 0.05 yielding

Order_ID	Product	Category	Unit_Price	Quantity	Discount
1001	Laptop	Tech	2200	1	0.05
1002	Chair	Office Supplies	350	5	0.20
1003	Desk	Office Supplies	450	3	0.20
1004	Phone	Tech	1400	2	0.05
1005	Jacket	Clothing	120	10	0.20

We will revisit this topic in more detail under Boolean indexing.

7.8 Dropping Column(s) using `drop()` Method

Sometimes, you may want to remove a column from your `DataFrame` if it's no longer needed. You can do this easily using the `drop()` method, specifying the column name.

For instance, let's remove the `Total_B_Tax` column:

Dropping a Column

```
sales_df = sales_df.drop(columns = ['Total_B_Tax'])
display(sales_df)
```

yielding:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1002	Office Supplies	Chair	350	5	0.1	1701.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	2570.4
1005	Clothing	Jacket	120	10	0.2	1036.8

Sometimes, you may want to drop multiple columns based on a range or pattern. This can be done by slicing the `columns`. `DataFrame_name.drop(columns = DataFrame_name.columns[slice])` drops one or more columns based on a slice of columns.

Here is an example:

To drop the `Category` and `Product` columns, we can use:

Dropping Columns with a Slice

```
sales_df.drop(columns = sales_df.columns[1:3])
```

yielding:

Order_ID	Unit_Price	Quantity	Discount	Total_A_Tax
1001	2200	1	0.0	2376.0
1002	350	5	0.1	1701.0
1003	450	3	0.0	1458.0
1004	1400	2	0.15	2570.4
1005	120	10	0.2	1036.8

7.9 Dropping Row(s) using `drop()` Method

Sometimes, you may also want to remove a specific row from your `DataFrame` if it's no longer needed. This can be done easily using the `drop()` method, specifying the row index.

For instance, let's remove the row with `Order_ID` 1002:

Dropping a Row

```
sales_df = sales_df.drop(index = ['1002'])
display(sales_df)
```

yielding:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	2570.4
1005	Clothing	Jacket	120	10	0.2	1036.8

We can also drop multiple rows based on a range or pattern by slicing the `index`. Using

```
DataFrame_name.drop(index=DataFrame_name.index[slice])
```

allows you to remove one or more rows based on specific row index slices. Alternatively, to keep only a subset of rows, you can use `DataFrame_name[slice]` to retain a specific range directly.

Here are some examples:

Dropping Rows with a Slice

```
sales_df.drop(index = sales_df.index[::2])  % Drops every other row
```

Yielding:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1002	Office Supplies	Chair	350	5	0.1	1701.0
1004	Tech	Phone	1400	2	0.15	2570.4

and the following example will keep only rows with indexes from 1 to 2.

Dropping Rows with a Slice

```
sales_df_slice = sales_df[1:3]
display(sales_df_slice)
```

yielding

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1002	Office Supplies	Chair	350	5	0.1	1701.0
1003	Office Supplies	Desk	450	3	0.0	1458.0

8 Boolean Indexing and Filtering in `pandas`

8.1 Introduction

Boolean indexing in `pandas` allows for efficient selection of data based on specific conditions. We will use the following `DataFrame`, named `sales_df`, as a reference for the Boolean indexing examples below.

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1002	Office Supplies	Chair	350	5	0.1	1701.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	3024.0
1005	Clothing	Jacket	120	10	0.2	1036.8

8.2 Filtering rows using Boolean indexing

Boolean indexing allows you to select rows where specific conditions are met. Here are a few examples:

Assume we are interested in rows with the `Total_A_Tax` greater than or equal to 2000:

Selecting Rows with `Total_A_Tax >= 2000`

```
high_value_sales = sales_df[sales_df['Total_A_Tax'] >= 2000]
display(high_value_sales)
```

yielding:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1004	Tech	Phone	1400	2	0.15	3024.0

While using `>=` as shown above works for filtering rows based on a condition, `pandas` offers comparison methods such as `.gt()` (greater than), `.lt()` (less than), `.ge()` (greater than or equal), and `.le()` (less than or equal), which provide a clearer syntax and are particularly helpful when chaining multiple conditions[7]. Here is how we can achieve the same filtering with the `.gt()` method:

Selecting Rows with `Total_A_Tax >= 2000` using `.gt()`

```
high_value_sales = sales_df[sales_df['Total_A_Tax'].gt(2000)]
display(high_value_sales)
```

This produces the same result:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1004	Tech	Phone	1400	2	0.15	3024.0

Boolean indexing also supports multiple conditions. You can combine multiple conditions using logical operators such as

- `&` (and)
- `|` (or)
- `~` (not)

[7] More on this soon

When using these operators, make sure to enclose each condition in parentheses `()` to ensure correct evaluation.

For instance, to select rows where `Category` is `Tech` and `Quantity` is greater than 1:

Selecting Tech with `Quantity > 1`

```
Tech_high_quantity = sales_df[(sales_df['Category'].eq('Tech')) &
                              (sales_df['Quantity'].gt(1))
                             ]
display(Tech_high_quantity)
```

yielding:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1004	Tech	Phone	1400	2	0.15	3024.0

Here is an example of using `~` (not) operator

Selecting Rows where Product is NOT Chair

```
non_chair_sales = sales_df[~(sales_df['Product'] == 'Chair')]
display(non_chair_sales)
```

yielding:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	3024.0
1005	Clothing	Jacket	120	10	0.2	1036.8

You can select complex conditions involving multiple columns as well. Here is an example:

Assume we are interested in rows where `Category` is either `Office Supplies` or `Tech` and `Total_A_Tax` is above 1500:

Selecting Multiple Categories and a Total Condition

```
selected_sales = sales_df[(
                           (sales_df['Category'] == 'Office Supplies') |
                           (sales_df['Category'] == 'Tech')
                          )
                            &
                          (sales_df['Total_A_Tax'] > 1500)
                         ]
display(selected_sales)
```

yielding:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1002	Office Supplies	Chair	350	5	1701.0	
1004	Tech	Phone	1400	2	0.15	3024.0

Note: While you can write all your conditions above in one line, it's a good idea to break the code into multiple lines for better readability, especially when dealing with complex conditions. By organizing each condition on its own line, it's easier to understand the logic, see each condition clearly, and avoid errors, as each condition is visually distinct. This approach also helps ensure that parentheses are properly matched, reducing the risk of syntax issues.

Boolean Indexing in `pandas DataFrame`

Boolean indexing lets you filter data in a `pandas DataFrame` based on conditions. By applying a condition, like `DataFrame['column'] > value`, you create a `True/False` mask. Use this mask to select only the rows where the condition is `True`, making it easy to work with specific parts of your data.

8.3 Using Comparison Functions: `.eq()`, `.gt()`, `.lt()`, `.ge()`, `.eq()`

As you saw in the examples above, `pandas` offers built-in comparison functions that can be used as an alternative to operators like `==`, `>=`, or `<`. These functions include `.eq()` (equal to), `.gt()` (greater than), `.lt()` (less than), `.ge()` (greater than or equal), and `.le()` (less than or equal). They provide a more readable syntax and make it easier to chain methods in complex queries. The following table, summarizes these methods and their usage.

Function	Equivalent Operator	Description
`.eq(value)`	`==`	Checks if values in the column are equal to the specified value.
`.gt(value)`	`>`	Checks if values in the column are greater than the specified value.
`.lt(value)`	`<`	Checks if values in the column are less than the specified value.
`.ge(value)`	`>=`	Checks if values in the column are greater than or equal to the specified value.
`.le(value)`	`<=`	Checks if values in the column are less than or equal to the specified value.

8.4 Chaining Methods for Complex Filters

In `pandas`, method chaining allows you to apply multiple operations in a single command. This technique is particularly helpful when filtering data based on several conditions. Instead of breaking down each step into

separate variables, chaining provides a clean and organized way to structure complex queries. Let's see a few examples:

Suppose you want to select rows from `sales_df` where the `Total_A_Tax` is greater than 1500, `Quantity` is more than 2, and `Category` is `Office Supplies`. Using chained comparison functions, you can achieve this in a single line:

Chaining Comparison Functions for Multiple Conditions

```
filtered_df = sales_df[sales_df['Total_A_Tax'].gt(1500)
                       & sales_df['Quantity'].gt(2)
                       & sales_df['Category'].eq('Office Supplies')]
display(filtered_df)
```

8.4.1 SettingWithCopyWarning in Method Chaining

While chaining methods is powerful for querying and filtering, chaining assignments—especially when using Boolean indexing—can sometimes lead to a `SettingWithCopyWarning` in `pandas`. This warning occurs because `pandas` may create a view of the data rather than referencing the original `DataFrame`, meaning that assignments might not always modify the original data as intended.

To understand this warning, let's look at the following snippet

Example of `SettingWithCopyWarning`

```
sales_df[sales_df['Quantity'] > 2]['Discount'] = 0.1  # This may trigger the warning
```

`pandas` might be working with a temporary copy of the filtered subset rather than directly referencing `sales_df`.

To avoid this, separate the filtering and assignment steps or use `loc[]` for clarity:

Using `loc[]` to Avoid Warning

```
sales_df.loc[sales_df['Quantity'] > 2, 'Discount'] = 0.1
```

8.5 Using `isin()` for Set-Based Filtering

The `isin()` function in `pandas` allows you to filter rows based on whether a column's values are present in a `list` of specified values. This method is particularly useful for checking against multiple values without needing multiple conditions with the `|` (or) operator.

For example, to select rows where `Category` is either `Tech` or `Office Supplies`:

Using `isin()` to Filter Rows Based on Multiple Values

```
filtered_sales = sales_df[sales_df['Category'].isin(['Tech', 'Office Supplies'])]
display(filtered_sales)
```

This would yield:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1002	Office Supplies	Chair	350	5	0.1	1701.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	3024.0

9 Sorting Data in `pandas`

Sorting is a fundamental operation in data analysis that helps organize data into meaningful patterns, making it easier to analyze and interpret. Common use cases for sorting include:

- **Ranking by performance metrics**: Sorting allows for the ranking of items, such as identifying top-performing products, departments, or employees based on specific metrics.
- **Time-based sorting for trend analysis**: Sorting chronologically (e.g., by date) is essential for visualizing trends, detecting seasonal patterns, and observing time-based behaviors in the data.
- **Categorical sorting for reporting**: Sorting by categories can help organize data for structured reporting, allowing insights into groups such as departments, regions, or product lines.

In `pandas`, we can sort data in a `DataFrame` by one or more columns, either in ascending or descending order. Sorting can be performed in place (modifying the original `DataFrame`) or by creating a new sorted `DataFrame`. We will learn these different sorting techniques using the example `DataFrame`, `sales_df`.

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1002	Office Supplies	Chair	350	5	0.1	1701.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	3024.0
1005	Clothing	Jacket	120	10	0.2	1036.8

9.1 The `sort_values()` Method

The primary method for sorting in `pandas` is `sort_values()`, which allows us to sort data by one or more columns.

9.1.1 Sorting by a Single Column

To sort the `DataFrame` by a single column, we specify the column name as a string within `sort_values()`. By default, the data is sorted in ascending order.

Sorting by a Single Column: `Unit_Price`

```
sorted_sales = sales_df.sort_values(by='Unit_Price')
display(sorted_sales)
```

This code sorts `sales_df` by `Unit_Price`, displaying rows in order from the lowest to highest price and yield:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1005	Clothing	Jacket	120	10	0.2	1036.8
1002	Office Supplies	Chair	350	5	0.1	1701.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	3024.0
1001	Tech	Laptop	2200	1	0.0	2376.0

9.1.2 Sorting in Descending Order

To sort data in descending order, we can set the `ascending` parameter to `False`.

For example, to sort by `Total_A_Tax` in descending order:

Sorting in Descending Order: `Total_A_Tax`

```
sorted_sales_desc = sales_df.sort_values(by='Total_A_Tax', ascending = False)
display(sorted_sales_desc)
```

This snippet sorts `sales_df` from the highest to the lowest `Total_A_Tax` value. The output would look like:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1004	Tech	Phone	1400	2	0.15	3024.0
1001	Tech	Laptop	2200	1	0.0	2376.0
1002	Office Supplies	Chair	350	5	0.1	1701.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1005	Clothing	Jacket	120	10	0.2	1036.8

9.1.3 Sorting by Multiple Columns

We can sort by multiple columns by passing a list of column names to the `by` parameter.

For example, to sort by `Category` first and then by `Quantity` within each category:

Sorting by Multiple Columns: `Category, Quantity`

```
multi_sorted_sales = sales_df.sort_values(by=['Category', 'Quantity'])
display(multi_sorted_sales)
```

In this case, `sales_df` is first sorted by `Category` alphabetically, and then within each `Category`, it's sorted by `Quantity` in ascending order. The output would be:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1005	Clothing	Jacket	120	10	0.2	1036.8
1003	Office Supplies	Desk	450	3	0.0	1458.0
1002	Office Supplies	Chair	350	5	0.1	1701.0
1001	Tech	Laptop	2200	1	0.0	2376.0
1004	Tech	Phone	1400	2	0.15	3024.0

9.1.4 Sorting with Mixed Order: Ascending and Descending

We can also sort by multiple columns with mixed order, where one column is sorted in ascending order and another in descending order. To do this, pass a list of Booleans to the `ascending` parameter, where each Boolean corresponds to the sort order of the respective column.

For example, to sort by `Category` in ascending order and by `Quantity` in descending order within each `Category`:

Sorting with Mixed Order: `Category` Ascending, `Quantity` Descending

```
mixed_sorted_sales = sales_df.sort_values(by=['Category', 'Quantity'],
                                          ascending=[True, False]
                                         )
display(mixed_sorted_sales)
```

Here, `Category` is sorted alphabetically in ascending order, and within each category, `Quantity` is sorted in descending order. The output would look like:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1005	Clothing	Jacket	120	10	0.2	1036.8
1002	Office Supplies	Chair	350	5	0.1	1701.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	3024.0
1001	Tech	Laptop	2200	1	0.0	2376.0

`DataFrame.sort_values()` Method

The `DataFrame.sort_values()` method allows you to sort the rows of a **pandas DataFrame** based on the values in one or more columns. You can specify the column to sort by, the order (`ascending=True` for ascending or `ascending=False` for descending), and whether to modify the DataFrame in place. This makes it easy to organize data for analysis.

9.2 Sorting by Index with `sort_index()`

In addition to sorting by columns, we can also sort data by the `DataFrame` index using the `sort_index()` method. This is useful when the index itself provides meaningful order (e.g., time series data or ordered categories).

Sorting by Index

```
sorted_by_index = sales_df.sort_index()
display(sorted_by_index)
```

yields

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1002	Office Supplies	Chair	350	5	0.1	1701.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	3024.0
1005	Clothing	Jacket	120	10	0.2	1036.8

By default, `sort_index()` sorts in ascending order, but we can set `ascending=False` to sort in descending order.

9.3 In-Place Sorting

Both `sort_values()` and `sort_index()` have an `inplace` parameter, which, if set to `True`, sorts the `DataFrame` directly without creating a new `DataFrame`. Be cautious when using `inplace=True` as it modifies the original data.

In-Place Sorting by `Unit_Price`

```
sales_df.sort_values(by='Unit_Price', inplace=True)
```

Here, `sales_df` itself is permanently sorted by `Unit_Price`.

10 Exploring Data in `pandas`

While your next handout will cover data exploration in greater depth, this section introduces foundational methods to explore a `DataFrame` quickly and effectively. `pandas` provides several methods to inspect, summarize, and analyze data in a `DataFrame`, allowing users to gain essential insights right from the start. We will explore some of the most commonly used methods, using `sales_df` as our reference.

10.1 Basic Properties: `len()`, `shape`, and `type`

Before looking into detailed data exploration, it's helpful to quickly check basic properties of a `DataFrame`:

- `len()`: Returns the number of rows in the `DataFrame`.
- `shape`: Returns a tuple containing the number of rows and columns.
- `type`: Provides the type of the object, which is useful for verifying that you are indeed working with a `DataFrame`.

Checking Basic Properties

```
print(f"Number of rows: {len(sales_df)}")
print(f"Shape of DataFrame: {sales_df.shape}")
print(f"Type of object: {type(sales_df)}")
```

This output provides a quick overview of the `DataFrame`'s structure, helping you confirm its dimensions and type before proceeding with further analysis. This yields

```
Number of rows:  5
Shape of DataFrame:  (5, 7)
Type of object:  <class 'pandas.core.frame.DataFrame'>
```

10.2 `info()` Method

The `info()` method provides a concise summary of the `DataFrame`, including the number of rows, columns, column names, data types, and non-null counts. This method is particularly helpful for getting an overview of the structure and data types within your dataset, as well as identifying any missing values.

Using `info()`

```
sales_df.info()
```

The output is as follows:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex:  5 entries, 0 to 4
Data columns (total 7 columns):
```

#	Column	Non-Null Count	Dtype
0	Order_ID	5 non-null	object
1	Category	5 non-null	object
2	Product	5 non-null	object
3	Unit_Price	5 non-null	int64
4	Quantity	5 non-null	int64
5	Discount	5 non-null	float64
6	Total_A_Tax	5 non-null	float64

```
dtypes:  float64(2), int64(2), object(3)
memory usage:  412.0 bytes
```

10.3 `head()` and `tail()` Methods

The `head()` and `tail()` methods display the first or last few rows of the `DataFrame`, respectively. By default, both methods display 5 rows, but you can specify a different number of rows to display as an argument. These methods are especially useful for quickly examining the structure and values in the beginning or end of a `DataFrame`.

Displaying the First Few Rows with `head()`

```
sales_df.head()
```

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1001	Tech	Laptop	2200	1	0.0	2376.0
1002	Office Supplies	Chair	350	5	0.1	1701.0
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	3024.0
1005	Clothing	Jacket	120	10	0.2	1036.8

Similarly, `tail()` displays the last few rows of the `DataFrame`:

Displaying the Last Few Rows with `tail()`

```
sales_df.tail(3)
```

This yields:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	3024.0
1005	Clothing	Jacket	120	10	0.2	1036.8

`DataFrame.info()` Method

The `DataFrame.info()` method provides a summary of a **pandas DataFrame**. It displays the number of rows, column names, data types, and the number of non-null values in each column. This is useful for quickly understanding the structure and completeness of your data.

10.4 `describe()` Method

The `describe()` method provides a statistical summary of each numerical column, including count, mean, standard deviation, minimum, maximum, and quartiles. This summary is a quick way to understand the distribution and central tendency of the data in your `DataFrame`, helping you spot outliers and trends.

Using `describe()` for Summary Statistics

```
sales_df.describe()
```

	Unit_Price	Quantity	Discount	Total_A_Tax
count	5.0	5.0	5.0	5.0
mean	904.0	4.2	0.09	1919.36
std	911.53	3.19	0.088	806.12
min	120.0	1.0	0.0	1036.8
25%	350.0	2.0	0.0	1458.0
50%	450.0	3.0	0.1	1701.0
75%	1400.0	5.0	0.15	2376.0
max	2200.0	10.0	0.2	3024.0

The `describe()` output provides a snapshot of key statistics for each column, which is helpful for understanding data range, variation, and any potential skewness.

10.5 Finding Averages, Sums, and Other Aggregates

In addition to `describe()`, we can compute individual summary statistics for columns. Common aggregations include finding the mean, sum, minimum, and maximum of columns.

10.5.1 Calculating the Mean of Columns

To find the average (mean) of a specific column, use the `mean()` method:

Finding the Mean of `Total_A_Tax`

```
average_total = sales_df['Total_A_Tax'].mean()
print(average_total)
```

```
1919.36
```

10.5.2 Calculating the Sum of Columns

Similarly, to calculate the sum of values in a column, use the `sum()` method:

Calculating the Sum of Quantity

```
total_quantity = sales_df['Quantity'].sum()
print(total_quantity)
```

```
21
```

This result shows the total number of items sold across all orders.

10.5.3 Finding the Minimum and Maximum Values

We can use `min()` and `max()` to find the minimum and maximum values in a column, respectively. For example, to find the lowest and highest `Unit_Price`:

Finding Minimum and Maximum of Unit_Price

```
min_price = sales_df['Unit_Price'].min()
max_price = sales_df['Unit_Price'].max()
print(f"Minimum Price: {min_price}")
print(f"Maximum Price: {max_price}")
```

```
Minimum Price:  120
Maximum Price:  2200
```

11 Resetting the Index with `reset_index()`

In `pandas`, the `reset_index()` method allows us to reset the index of a `DataFrame` to a default integer-based index, making it especially useful when the current index no longer aligns with the intended data structure. This method creates a clean, ordered index column.

11.1 Basic Usage of `reset_index()`

When we use `reset_index()` without any arguments, the method resets the index to a default integer-based sequence starting from 0. The existing index is added as a new column in the `DataFrame`:

Resetting the `Index`

```
reset_sales_df = sales_df.reset_index()
display(reset_sales_df)
```

In the resulting `DataFrame`, the original index becomes a new column named `index`:

	Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
0	1001	Tech	Laptop	2200	1	0.0	2376.0
1	1002	Office Supplies	Chair	350	5	0.1	1701.0
2	1003	Office Supplies	Desk	450	3	0.0	1458.0
3	1004	Tech	Phone	1400	2	0.15	3024.0
4	1005	Clothing	Jacket	120	10	0.2	1036.8

11.2 Dropping the Old Index

In some cases, retaining the old index as a column is unnecessary, such as before exporting the data to a CSV file. Setting the `drop` parameter to `True` discards the original index:

Resetting the Index without Keeping the Old Index

```
reset_sales_df = sales_df.reset_index(drop = True)
display(reset_sales_df)
```

With `drop = True`, the `DataFrame` retains only the reset integer index:

	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
0	Tech	Laptop	2200	1	0.0	2376.0
1	Office Supplies	Chair	350	5	0.1	1701.0
2	Office Supplies	Desk	450	3	0.0	1458.0
3	Tech	Phone	1400	2	0.15	3024.0
4	Clothing	Jacket	120	10	0.2	1036.8

11.3 The Importance of `reset_index()`

Resetting the index is often a crucial step before exporting data, especially when filtering or modifying rows. After operations such as sorting, filtering, or grouping, the index might no longer be a clean sequence of numbers, which can lead to confusion or errors. By using `reset_index()`, you ensure that the index is consistent and suitable for further analysis or sharing.

`reset_index()` is also helpful when combining `DataFrames` where mismatched indices may cause errors. We will cover combining `DataFrames` later.

DataFrame.reset$_index()Method$

The DataFrame.reset$_index()methodresetstheindexofa$**pandas DataFrame**$backtothedefaultintegerindex.Ifgouprevi

12 Working with MultiIndex in `pandas`

In many real-world datasets, multi-level indexing (MultiIndex) is common as it allows for organizing data hierarchically. This structure is valuable in business and data analysis when managing attributes across multiple categories, such as sales data segmented by region, product type, or customer demographics.

12.1 Creating a MultiIndex DataFrame

To illustrate, let's expand our `sales_df` to track sales across multiple regions by setting a MultiIndex on `Region` and `Order_ID`:

Creating a MultiIndex DataFrame

```
import pandas as pd

data = {
    'Region': ['East', 'East', 'West', 'West', 'South'],
```

```
    'Order_ID': ['1001', '1002', '1003', '1004', '1005'],
    'Category': ['Tech', 'Office Supplies', 'Office Supplies', 'Tech', 'Clothing'],
    'Product': ['Laptop', 'Chair', 'Desk', 'Phone', 'Jacket'],
    'Unit_Price': [2200, 350, 450, 1400, 120],
    'Quantity': [1, 5, 3, 2, 10],
    'Discount': [0.0, 0.1, 0.0, 0.15, 0.2],
    'Total_A_Tax': [2376.0, 1701.0, 1458.0, 3024.0, 1036.8]
}
multi_sales_df = pd.DataFrame(data).set_index(['Region', 'Order_ID'])
display(multi_sales_df)
```

The resulting `DataFrame` has a hierarchical index, making it easier to organize and filter based on region and order.

		Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
Region	**Order_ID**						
East	**1001**	Tech	Laptop	2200	1	0.0	2376.0
	1002	Office Supplies	Chair	350	5	0.1	1701.0
West	**1003**	Office Supplies	Desk	450	3	0.0	1458.0
	1004	Tech	Phone	1400	2	0.15	3024.0
South	**1005**	Clothing	Jacket	120	10	0.2	1036.8

12.2 Understanding the `index` Attribute in a MultiIndex DataFrame

In a `MultiIndex` DataFrame, the index is stored as a list of tuples, where each tuple represents a unique combination of the values across the levels of the index. For `multi_sales_df`, each index entry is a tuple of the form `(Region, Order_ID)`, allowing us to represent hierarchical relationships within the data. Here

- `Region` is known as **level 0** of the index
- `Order_ID` is **level 1**.

To inspect the index structure of a `MultiIndex` DataFrame, we can look at its `index` attribute:

Viewing the Index Structure

```
print(multi_sales_df.index)
```

This will output:

```
MultiIndex([( 'East', '1001'),
            ( 'East', '1002'),
            ( 'West', '1003'),
            ( 'West', '1004'),
            ('South', '1005')],
           names=['Region', 'Order_ID'])
```

In this representation:

- Each tuple is a unique key that identifies a row in the `DataFrame`.
- The **names** of the index levels (in this case, `Region` and `Order_ID`) correspond to **level 0** and **level 1**, respectively, providing clear labeling for each level.

Understanding the hierarchy of `MultiIndex levels` (levels **0** and **1**) provides flexibility in operations like slicing, filtering, or sorting data based on specific criteria in hierarchical data structures.

12.3 Accessing Data in a MultiIndex DataFrame

`MultiIndex` allows filtering and data selection based on specific levels, enabling more advanced data exploration.

12.3.1 Using `loc[ ]`

To access data for a specific region, such as `West`, use `loc[]` with the region name:

Accessing Data for a Specific Region

```
west_sales = multi_sales_df.loc['West']
display(west_sales)
```

This returns all records where `Region` is `West`:

Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
1003	Office Supplies	Desk	450	3	0.0	1458.0
1004	Tech	Phone	1400	2	0.15	3024.0

To access specific rows across both levels of the index, you can specify values for each level, noting that each MultiIndex label is treated as a tuple representing the multiple levels of indexing:

Accessing a Specific Row

```
west_desk = multi_sales_df.loc[('West', '1003')]
display(west_desk)
```

This returns the specific entry for the `West` region and `Order_ID` 1003:

Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
Office Supplies	Desk	450	3	0.0	1458.0

12.3.2 Using `at[ ]`

You can also use `at[]` for access to a specific cell when working with a MultiIndex.

Accessing a Single Value with `at[ ]`

```
discount_west_desk = multi_sales_df.at[('West', '1003'), 'Discount']
print(discount_west_desk)
```

This will output:

```
0.0
```

12.3.3 Boolean Indexing with MultiIndex

Boolean indexing can also be used with MultiIndex, allowing you to filter based on multiple criteria.

For example, to find records where `Quantity` is greater than 2:

Boolean Indexing with MultiIndex

```
high_quantity_sales = multi_sales_df[multi_sales_df['Quantity'] > 2]
display(high_quantity_sales)
```

Region	Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
East	1002	Office Supplies	Chair	350	5	0.1	1701.0
West	1003	Office Supplies	Desk	450	3	0.0	1458.0
South	1005	Clothing	Jacket	120	10	0.2	1036.8

12.4 Sorting with MultiIndex

MultiIndex `DataFrames` can be sorted at multiple levels. By default, the `sort_index()` method sorts on all levels in ascending order, but specific levels can be targeted.

When specifying a level, you can use the level index (0 for `Region` and 1 for `Order_ID`) or level names directly:

Sorting by MultiIndex Levels

```
sorted_multi_sales_df = multi_sales_df.sort_index(level=1, ascending=False)
display(sorted_multi_sales_df)
```

This sorts by `Order_ID` in descending order within each `Region`:

Region	Order_ID	Category	Product	Unit_Price	Quantity	Discount	Total_A_Tax
East	**1002**	Office Supplies	Chair	350	5	0.1	1701.0
	1001	Tech	Laptop	2200	1	0.0	2376.0
West	**1004**	Tech	Phone	1400	2	0.15	3024.0
	1003	Office Supplies	Desk	450	3	0.0	1458.0
South	**1005**	Clothing	Jacket	120	10	0.2	1036.8

13 Saving a `DataFrame` to CSV

Once data has been processed or analyzed within `pandas`, it is often useful to save the results for future use or to share with others. The `to_csv()` method provides a convenient way to export a `DataFrame` to a CSV file, which can then be used by other applications or reloaded for additional analysis.

13.1 Basic Usage of `to_csv()`

The basic syntax for saving a `DataFrame` to a CSV file is as follows:

```
DataFrame.to_csv('filename.csv')
```

For example, to save `sales_df` to a file named `'sales_data.csv'`:

Saving `sales_df` to CSV

```
sales_df.to_csv('sales_data.csv')
```

By default, this command saves all columns and rows of the `DataFrame`, along with the `index`.

13.2 Controlling the `index` Parameter

By default, `to_csv()` includes the `DataFrame`'s index in the CSV file, which can sometimes be useful for preserving row labels. However, in many cases, the index may not be necessary and can be excluded by setting the `index` parameter to `False`:

Saving CSV without Index

```
sales_df.to_csv('sales_data_no_index.csv', index=False)
```

In this example, the resulting CSV file will contain only the columns of `sales_df`, excluding the index.

13.3 Specifying a Different Delimiter

Although CSV files traditionally use commas to separate values, `to_csv()` allows for different delimiters using the `sep` parameter.

For example, to save `sales_df` with a tab separator:

Saving `CSV` with `Tab` Separator

```
sales_df.to_csv('sales_data_tab.tsv', sep='\t')
```

The `sep='\t'` argument saves the file in a tab-separated format, commonly used when data includes commas within fields.

13.4 Saving Selected Columns

In some cases, you may wish to save only specific columns from the `DataFrame`. The `columns` parameter allows you to specify a list of columns to save:

Saving Selected Columns to `CSV`

```
sales_df.to_csv('sales_data_selected_columns.csv',
                columns=['Order_ID', 'Product', 'Total_A_Tax'],
                index=False)
```

Here, only the `Order_ID`, `Product`, and `Total_A_Tax` columns are saved to the file.

13.5 Saving with Different Encoding

Computers work with binary (0s and 1s) and do not inherently understand text. To represent letters, numbers, and symbols, each character must be stored as a unique number that computers can interpret. `Encoding` is the process of mapping each character to a unique number. Different encodings exist to cover various characters from languages around the world.

Unicode is the most widely used system, assigning a unique number to every character and allowing consistent display across devices and programs. The most common encoding formats in `pandas` are **UTF-8** and **UTF-16**, which determine how many bytes each character occupies:

- **UTF-8**: This is the default encoding for `to_csv()`, and it uses one to four bytes per character. UTF-8 is efficient for English and many Latin-based languages, making it a good choice for most applications.
- **UTF-16**: This encoding uses two bytes for most characters, making it effective for texts with a wider range of characters, such as Asian languages. It requires more space for English text but can improve compatibility for complex character sets.

If your CSV file includes special characters (like accents or symbols), selecting the right encoding is essential to ensure that they appear correctly. Without proper encoding, software may misinterpret the data, leading to unreadable or mixed-up text.

Here's a basic example of saving a CSV file using the default UTF-8 encoding:

Saving `CSV` with Default `UTF-8` Encoding

```
sales_df.to_csv('sales_data_default.csv')
```

The above command saves `sales_df` in UTF-8 encoding, which is sufficient for most cases. However, if

special characters or specific language support is required, you may want to specify another encoding, like UTF-16:

Saving `CSV` with `UTF-16` Encoding

```
sales_df.to_csv('sales_data_utf16.csv', encoding='utf-16')
```

This ensures compatibility with software that requires non-UTF-8 encoding or that needs support for specific characters, improving file portability and readability.

Next Topic Preview

In the next handout, we will dive into `Exploring Data with Pandas`, where we'll focus on analyzing and summarizing datasets. Key topics include examining the structure of data, calculating basic statistics, handling missing values, and applying functions to transform and enrich data. By the end, you'll be able to confidently explore and gain insights from data using `pandas`.

14 Exercises

1. **Creating a Series for Product Prices**
   (a) A company has a list of prices for five products: `[250, 500, 750, 1000, 1250]`. Create a `pandas Series` for these prices.
   (b) Assign product names as the index for the Series: `['Product A', 'Product B', 'Product C', 'Product D', 'Product E']`.
   (c) Access the price of `Product C`.

2. **Series Operations on Product Data**
   (a) Sort the Series by product names (index) in alphabetical order.
   (b) Sort the Series by product prices in descending order to find the most expensive product.
   (c) Use `describe()` to get a quick statistical summary of the product prices.

3. **Series Creation**
   (a) Create a Series for a list of items in stock: `['Laptop', 1200, 'Tablet', 800, 'Monitor', 300]`.

4. **Creating a DataFrame for Customer Orders**
   (a) Given the dictionary `{ 'Customer': ['Alice', 'Bob', 'Charlie'], 'Order Amount': [200, 450, 300], 'Location': ['NY', 'LA', 'Chicago']}`, create a `pandas DataFrame`.
   (b) Display the `DataFrame` to review the customer order information.

5. **DataFrame Index Operations for Customer Orders**
   (a) Reset the index of the DataFrame to the default numerical index, and confirm the change by displaying the DataFrame.
   (b) Set the `Customer` column as the index of the DataFrame.
   (c) Reset the index back to the default numerical index again.

6. **Sorting and Describing Customer Orders**
   (a) Sort the DataFrame by the `Order Amount` in ascending order to see the smallest orders first.
   (b) Sort the DataFrame by the `Location` column in alphabetical order to group orders by city.
   (c) Use `describe()` to obtain a summary of the `Order Amounts`.

7. **Filtering Data Using Boolean Indexing**
   (a) Using the `Order Amount` column, filter the DataFrame to show only orders where the amount is greater than 300.
   (b) Filter the DataFrame to display only the orders placed by customers located in 'NY'.

15 References

References and Resources

The following references and resources were used in the preparation of these materials:

(a) Official Python website at https://www.python.org/.

(b) *Introduction to Computation and Programming Using Python*, John Guttag, The MIT Press, 2nd edition, 2016.

(c) *Python for Data Science Handbook: Essential Tools for Working with Data*, Jake VanderPlas, O'Reilly Media, 1st edition, 2016.

(d) *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, Wes McKinney, O'Reilly Media, 2nd edition, 2017.

(e) *Introduction to Python for Computer Science and Data Science*, Paul J. Deitel, Harvey Deitel, Pearson, 1st edition, 2019.

(f) *Data Visualization in Python with Pandas and Matplotlib*, David Landup, Independently published, 2021.

(g) *Python for Programmers with Introductory AI Case Studies*, Paul Deitel, Harvey Deitel, Pearson, 1st edition, 2019.

(h) *Effective Pandas: Patterns for Data Manipulation (Treading on Python)*, Matt Harrison, Independently published, 2021.

(i) *Introduction to Programming in Python; An Interdisciplinary Approach*, Robert Sedgewick, Kevin Wayne, Robert Dondero, Pearson, 1st edition, 2015.

(j) Python tutorials at https://betterprogramming.pub/.

(k) Python learning platform at https://www.learnpython.org/.

(l) Python resources at https://realpython.com/.

(m) Python courses and tutorials at https://www.datacamp.com/.