

Becoming a Data Analyst

A beginner's guide to kickstarting your data analysis journey

Early Access



Kedeisha Bryan Maaike van Putten

Becoming a Data Analyst

Copyright © 2023 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Early Access Publication: Becoming a Data Analyst

Early Access Production Reference: B21107

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80512-641-6

www.packt.com

Table of Contents

1. [Becoming a Data Analyst: A beginner's guide to kickstarting your data analysis journey](#)
2. [1 Understanding the Business Context of Data Analysis](#)
 - I. [Join our book community on Discord](#)
 - II. [A Data Analyst's Role in the Data Analytics Lifecycle](#)
 - i. [Business Understanding](#)
 - ii. [Data Inspection](#)
 - iii. [Data Pre-processing & Preparation](#)
 - iv. [Exploratory Data Analysis](#)
 - v. [Data Validation](#)
 - vi. [Explanatory Data Analysis](#)
 - III. [Summary](#)
3. [2 Introduction to SQL](#)
 - I. [Join our book community on Discord](#)
 - II. [SQL and its use cases](#)
 - i. [Brief History of SQL](#)
 - ii. [SQL and data analysis](#)
 - III. [Different Databases](#)
 - i. [Relational vs. Non-Relational Databases](#)
 - ii. [Popular DBMS's](#)
 - IV. [SQL Terminology](#)
 - i. [Query](#)
 - ii. [Statement](#)
 - iii. [Clause](#)
 - iv. [Keyword](#)
 - v. [View](#)
 - V. [Setting up your environment](#)
 - i. [Choosing a DBMS](#)
 - ii. [Installing necessary software](#)
 - iii. [Creating a sample database](#)
 - VI. [Writing Basic SQL Queries](#)
 - i. [SELECT Statement](#)
 - ii. [Structure of a Query](#)

- iii. [INSERT Statements](#)
 - iv. [UPDATE Statements](#)
 - v. [DELETE Statements](#)
 - vi. [SQL basic rules and syntax](#)
 - VII. [Filtering and organizing data with clauses](#)
 - i. [WHERE Clause](#)
 - ii. [ORDER BY Clause](#)
 - iii. [DISTINCT Clause](#)
 - iv. [LIMIT Clause](#)
 - VIII. [Using operators and functions](#)
 - i. [Comparison operators](#)
 - ii. [Logical operators \(AND, OR\)](#)
 - iii. [LIKE operator](#)
 - iv. [Arithmetic operators](#)
 - v. [Functions for calculations](#)
 - vi. [Functions for text manipulation](#)
 - vii. [Date functions](#)
 - IX. [Summary](#)
4. [3 Joining Tables in SQL](#)
- I. [Join our book community on Discord](#)
 - II. [Table relations](#)
 - i. [Implementing Relationships in SQL](#)
 - ii. [SQL Joins](#)
 - iii. [Best Practices for Using JOIN in SQL](#)
 - III. [Summary](#)
5. [4 Creating Business Metrics with Aggregations](#)
- I. [Join our book community on Discord](#)
 - II. [Aggregations in Business Metrics](#)
 - i. [Aggregations in SQL to Analyze Data](#)
 - ii. [GROUP BY Clause](#)
 - iii. [HAVING clause](#)
 - iv. [Best Practices for Aggregations](#)
 - III. [Summary](#)
6. [5 Advanced SQL](#)
- I. [Join our book community on Discord](#)
 - II. [Working with subqueries](#)
 - i. [Types of subqueries](#)

- ii. [Non-code explanation of subquery](#)
 - iii. [Using a basic subquery on our library database](#)
 - iv. [Subquery vs joining tables](#)
 - v. [Rules of subquery usage](#)
 - vi. [More advanced subquery usage on our library database](#)
- III. [Common Table Expressions](#)
- i. [Use cases for CTEs](#)
 - ii. [Examples with the library database](#)
- IV. [Window functions: A panoramic view of your data](#)
- i. [Example with the Library Database](#)
- V. [Understanding date time manipulation](#)
- i. [Date and time functions](#)
 - ii. [Examples with the library database](#)
- VI. [Understanding text manipulation](#)
- i. [Text functions](#)
 - ii. [Text functions in the library database](#)
- VII. [Best practices: bringing it all together](#)
- i. [Write readable SQL Code](#)
 - ii. [Be careful with NULL values](#)
 - iii. [Use subqueries and CTEs wisely](#)
 - iv. [Think about performance](#)
 - v. [Test your queries](#)
- VIII. [Summary](#)
7. [6 SQL for Data Analysis Case Study](#)
- I. [Join our book community on Discord](#)
 - II. [Setting up the database](#)
- III. [Performing data analysis with SQL](#)
- IV. [Exploring the data](#)
- i. [General data insights](#)
- V. [Analyzing the data](#)
- i. [Examining the clothing category](#)
 - ii. [Determining the number of customers](#)
 - iii. [Researching the top payment methods](#)
 - iv. [Gathering customer feedback](#)
 - v. [Exploring the relationship between ratings and sales](#)
 - vi. [Finding the percentage of products with reviews](#)
 - vii. [Effectiveness of discounts](#)

- viii. [Identifying the top customers](#)
- ix. [Top-selling clothing products](#)
- x. [Top 5 high-performing products](#)
- xi. [Most popular product by country](#)
- xii. [Researching the performance of delivery](#)
- xiii. [Future projections with linear regression](#)

VI. [Summary](#)

- 8. [7 Fundamental Statistical Concepts](#)
 - I. [Join our book community on Discord](#)
 - II. [Descriptive statistics](#)
 - i. [Levels of measurement](#)
 - ii. [Measures of central tendency](#)
 - iii. [Measures of variability](#)
 - III. [Inferential statistics](#)
 - i. [Probability theory](#)
 - ii. [Probability distributions](#)
 - iii. [Correlation vs causation](#)
 - IV. [Summary](#)

- 9. [8 Testing Hypotheses](#)
 - I. [Join our book community on Discord](#)
 - II. [Technical requirements \(H1 – Section\)](#)
 - III. [Introduction to Hypothesis Testing](#)
 - i. [Role of Hypothesis Testing in Data Analysis](#)
 - ii. [Null and Alternative Hypothesis](#)
 - iii. [Step by Step Guide to Performing Hypothesis Testing](#)
 - IV. [One Sample t-Test](#)
 - V. [Conditions for Performing a One-Sample T-Test](#)
 - i. [Case Study: Average Exam Scores](#)
 - VI. [Two Sample t-Test](#)
 - i. [Case Study: Comparing Exam Scores Between Two Schools](#)
 - VII. [Chi Square Test](#)
 - i. [Case Study: Effect of Tutoring on Passing Rates](#)
 - VIII. [Analysis of Variance \(ANOVA\)](#)
 - i. [Case Study: Comparing Exam Scores Among Three Schools](#)
 - IX. [Summary](#)
- 10. [9 Business Statistics Case Study](#)
 - I. [Join our book community on Discord](#)

- II. [Technical requirements \(H1 – Section\)](#)
 - III. [Case Study Overview](#)
 - i. [Learning Objectives:](#)
 - ii. [Questions:](#)
 - iii. [Solutions:](#)
 - IV. [Additional Topics to Explore](#)
 - i. [Text Analytics](#)
 - ii. [Big Data](#)
 - iii. [Time Series Analysis](#)
 - iv. [Predictive Analytics](#)
 - v. [Prescriptive Analytics & Optimization](#)
 - vi. [Database Management](#)
 - V. [Where to practice](#)
 - VI. [Summary](#)
- 11. [10 Data analysis and programming](#)
 - I. [Join our book community on Discord](#)
 - II. [The role of programming and our case](#)
 - III. [Different programming languages](#)
 - i. [Python](#)
 - ii. [R](#)
 - iii. [SQL](#)
 - iv. [Julia](#)
 - v. [MATLAB](#)
 - IV. [Working with the Command Line Interface \(CLI\)](#)
 - i. [Command Line Interface \(CLI\) vs Graphical User Interface \(GUI\)](#)
 - ii. [Accessing the CLI](#)
 - iii. [Typical CLI tasks](#)
 - iv. [Using the CLI for programming](#)
 - V. [Setting up your system for Python programming](#)
 - i. [Check if Python is installed](#)
 - ii. [MacOS](#)
 - iii. [Linux](#)
 - iv. [Windows](#)
 - v. [Browser \(cloud-based\)](#)
 - vi. [Testing the Python setup](#)
 - VI. [Python use cases for CleanAndGreen](#)

- i. [Data Cleaning and Preparation](#)
- ii. [Data Visualization](#)
- iii. [Statistical Modeling](#)
- iv. [Predictive Modeling/Machine Learning](#)
- v. [General remarks on Python](#)

VII. [Summary](#)

12. [11 Introduction to Python](#)

I. [Join our book community on Discord](#)

II. [Understanding the Python Syntax](#)

- i. [Print Statements](#)
- ii. [Comments](#)
- iii. [Variables](#)
- iv. [Operations on variables](#)
- v. [Operators and Expressions](#)

III. [Exploring Data Types in Python](#)

- i. [Strings](#)
- ii. [Integers](#)
- iii. [Floats](#)
- iv. [Booleans](#)
- v. [Type Conversion](#)

IV. [Indexing and Slicing in Python](#)

V. [Unpacking Data Structures](#)

- i. [Lists](#)
- ii. [Dictionaries](#)
- iii. [Sets](#)
- iv. [Tuples](#)

VI. [Mastering Control Flow Structures](#)

- i. [Conditional Statements in Python](#)
- ii. [Looping in Python](#)

VII. [Functions in Python](#)

- i. [Creating Your Own Functions](#)
- ii. [Python Built-In Functions](#)

VIII. [Summary](#)

13. [12 Analyzing data with NumPy & Pandas](#)

I. [Join our book community on Discord](#)

II. [Introduction to NumPy](#)

- i. [Installing and Importing NumPy](#)

- ii. [Basic NumPy Operations](#)
 - III. [Statistical and Mathematical Operations](#)
 - i. [Mathematical Operations with NumPy Arrays](#)
 - IV. [Multi-dimensional Arrays](#)
 - i. [Creating Multi-dimensional Arrays](#)
 - ii. [Accessing elements in Multi-dimensional Arrays](#)
 - iii. [Reading Data from a CSV File](#)
 - V. [Introduction to Pandas](#)
 - i. [Series and DataFrame](#)
 - ii. [Loading Data with Pandas](#)
 - iii. [Data Analysis with Pandas](#)
 - iv. [Data Analysis](#)
 - VI. [Summary](#)
14. [13 Introduction to Exploratory Data Analysis](#)
- I. [Join our book community on Discord](#)
 - II. [The Importance of EDA](#)
 - i. [The EDA Process](#)
 - ii. [Tools and Techniques](#)
 - III. [Univariate Analysis](#)
 - i. [Analyzing Continuous Variables](#)
 - ii. [Analyzing Categorical Variables](#)
 - IV. [Bivariate Analysis](#)
 - i. [Understanding bivariate analysis](#)
 - ii. [Correlation vs Causation](#)
 - iii. [Visualizing relationships between two continuous variables](#)
 - V. [Multivariate analysis](#)
 - i. [Heatmaps](#)
 - ii. [Pair plots](#)
 - VI. [Summary](#)
15. [14 Data Cleaning](#)
- I. [Join our book community on Discord](#)
 - II. [Technical requirements](#)
 - III. [Importance of data cleaning](#)
 - i. [Impact on data quality](#)
 - ii. [Relevance to business decisions](#)
 - IV. [Common data cleaning challenges](#)

- i. [Inconsistent formats](#)
 - ii. [Misspellings and Inaccuracies](#)
 - iii. [Duplicate records](#)
 - V. [Dealing with missing values](#)
 - i. [Causes of missing values](#)
 - ii. [Strategies for handling missing values](#)
 - iii. [Types of missing data](#)
 - VI. [Dealing with duplicate values](#)
 - i. [Causes of duplicate data](#)
 - ii. [Identification and removal](#)
 - VII. [Dealing with outliers](#)
 - i. [Types of outliers](#)
 - ii. [Impact on analysis](#)
 - iii. [Techniques for identifying and handling outliers](#)
 - VIII. [Cleaning and transforming data](#)
 - i. [Handling inconsistencies](#)
 - ii. [Converting categorical data](#)
 - iii. [Normalizing numerical features](#)
 - IX. [Data validation](#)
 - i. [Validation methods](#)
 - X. [Summary](#)
16. [17 Exploratory Data Analysis Case Study](#)
- I. [Join our book community on Discord](#)
 - II. [Technical Requirements](#)
 - III. [E-commerce Sales Optimization Case Study](#)
 - i. [Time Series Analysis](#)
 - ii. [Customer Segmentation](#)
 - iii. [Product Analysis](#)
 - iv. [Payment and Returns](#)
 - v. [Case Study Answers](#)
 - IV. [Summary](#)

Becoming a Data Analyst: A beginner's guide to kickstarting your data analysis journey

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time. You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1. Chapter 1: Understanding the business context of data analysis
2. Chapter 2: Introduction to SQL
3. Chapter 3: Joining Tables
4. Chapter 4: Creating Business Metrics with Aggregations
5. Chapter 5: Advanced SQL
6. Chapter 6: SQL for Data Analysis Case Study
7. Chapter 7: Fundamental statistics concepts
8. Chapter 8: Testing hypotheses
9. Chapter 9: Business Statistics Case Study
10. Chapter 10: Data analysis and programming
11. Chapter 11: Introduction to Python
12. Chapter 12: Analyzing data in NumPy & Pandas
13. Chapter 13: Introduction to Exploratory Data Analysis
14. Chapter 14: Data cleaning
15. Chapter 15: Univariate Analysis
16. Chapter 16: Bivariate Analysis
17. Chapter 17: Exploratory Data Analysis Case Study
18. Chapter 18: Introduction to Data Visualization
19. Chapter 19: Choosing the right Visualization

1 Understanding the Business Context of Data Analysis

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



In a research paper written by David Becker in 2017, some of the leading causes of failed data projects relate to failures relating to the management of a project. These include **incorrect business objectives, improper scope of a project, incorrect project structure, and poor communication**. There is only so much you can control regarding the success of a project, but it's important to know how to manage your portion of the project. In this chapter we will cover the roles and responsibilities of a data analyst in each phase of a data project, helpful tools for project success, and a typical technical tools a data analyst can be expected to use.

A Data Analyst's Role in the Data Analytics Lifecycle

The data analytics lifecycle is developed from the cross-industry standard for data mining. Also known as CRISP-DM. The major phases include business understanding, data understanding, data preparation, modeling, evaluation, and deployment. As our focus will not be creating models, the data analytics lifecycle is similar except the substitution of exploratory data analysis, data validation, and presentation.

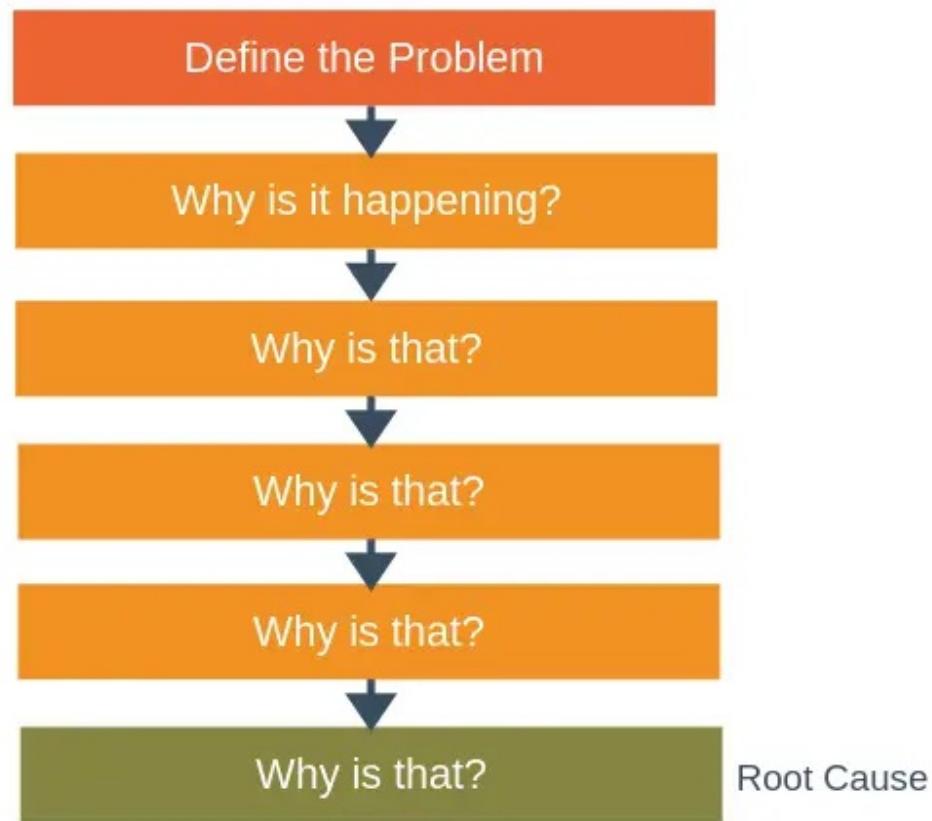
Business Understanding

The first and probably most important phase is business understanding as your work here will set the direction for the rest of the project. Mistakes here result in performing unnecessary work or providing a solution that does not solve the intended problem. There are about 3 main areas of this phase:

1. **Defining business objectives:** Here you will be defining the actual goal of the data project. The basis for every project is to provide a solution that solves a problem or improve a process. An important concept is to understand **symptoms** vs **the root cause**. The symptoms will be the visual signs or effects of an issue of a system. Symptoms trigger the investigation of an issue or the need for a solution.

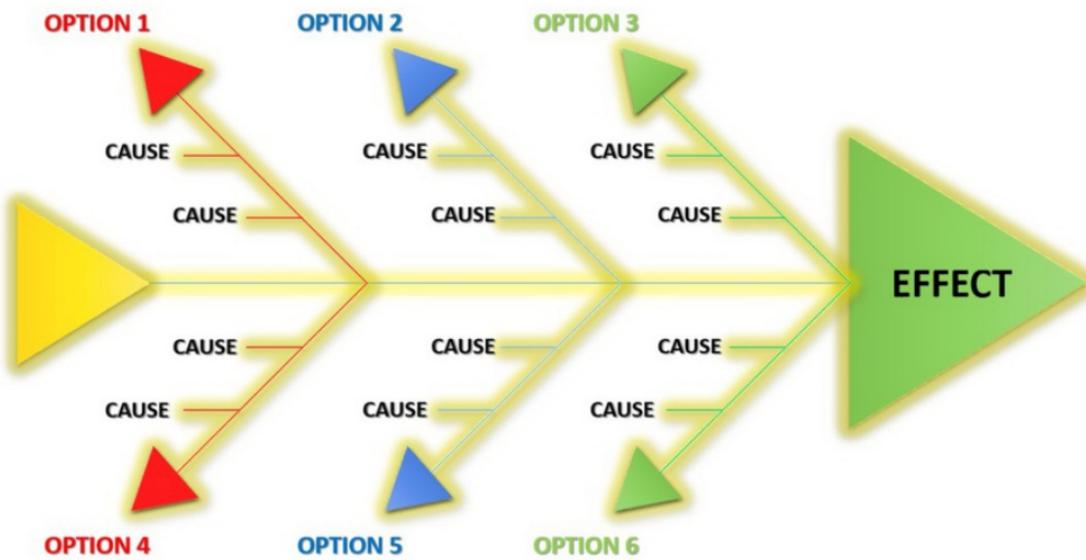
The root cause is the underlying issue that all the symptoms stem from. Unlike symptoms, root causes are often not visible or apparent without a thorough investigation. Addressing the symptoms will only provide temporary fixes, while addressing the root cause will resolve the problem more permanently. When speaking with stakeholders, often they may spend most of their time speaking about the symptoms. Many times, what they say is the problem really isn't the problem. As a data analyst, you must know how to ask the right questions to sift through symptoms to figure out the root cause and the correct business problem. **Tools for success:** Five Whys and Fishbone diagrams. The Five Whys is a quick and effective technique to uncover a root cause. Where you begin with a problem statement and follow with asking “why” five times or any amount needed to land at the underlying root cause. Below is a diagram the visually depicts the process.

The 5 Whys



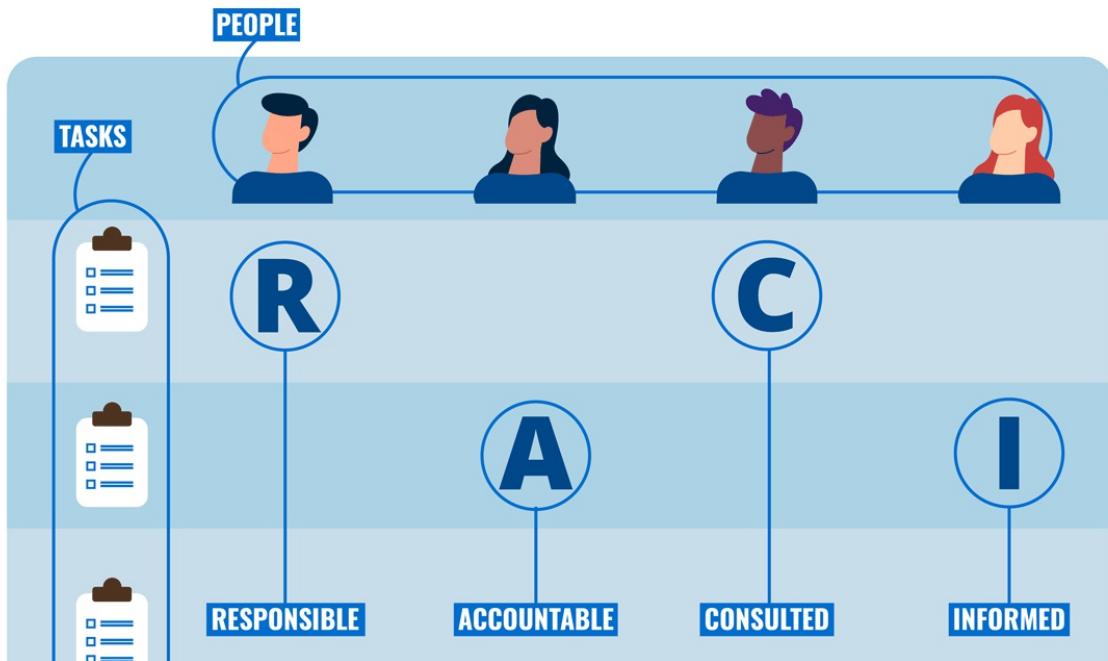
The Fishbone Diagram, as pictured below, is a visual aid to identify and organize the possible causes and effects of a problem. This is also a great method for prioritizing the different root problems to solve.

FISHBONE DIAGRAM



1. **Gather relative information:** Once you determine your business objective, your next task is to gather the data and any additional information regarding the project. Typical task includes identifying your data sources. This will be internal sources and may include external data as well. The data sources can include company databases, reports, or third-party research. Other data sources can include additional interviews with stakeholders, web scraping, or surveys.

Tools for success: An important skill for a data analyst is the ability to navigate an organization to source all the information and data that you need. This will be done through interviews and emails. Its also important to know that you will work with other people who will have different roles and responsibilities within the project. A great way to keep all this information organized is a **RACI matrix**, pictured below. A RACI matrix helps you keep track of the responsibilities of others in your project and whom you consulted and informed. This is a great way to ensure a high level of communication in a project.



Determine key performance indicators (KPIs) and metrics: KPIs and metrics are used to measure and evaluate a business process. A **metric** is a measure used to track and monitor a business process. Examples include website traffic and profit margin. A **KPI** is a measurable value that's often tied to business goals and strategy. Examples include net promoter score, conversion rates, and customer lifetime value. While they are very similar in nature, KPIs are more closely aligned with business goals while metrics are used to track any business activity. When speaking with stakeholders, you would also like to define the **critical success factors**, the essential activities that must go well in order for the objective to be achieved. Based on those success factors, you will develop your KPIs and metrics. These measures will be part of the essential data used for business decisions.

1. **Clarify scope of work:** A project can go in many directions as there can be many problems to investigate. To avoid scope creep, where a project encounters uncontrolled changes over time, you want to establish clear goals and priorities.

Tools for Success: A Scope of Work (SOW) or project charter are excellent tools to ensure a clear scope for your project. It is a document that can summarize all the information gathered in the previous steps that would

include the project overview, your tasks, expected results, timeline, and expected deliverable. The expected deliverable is one of the most important elements as it defines the work you will present at the end of the project. Deliverables can include a dashboard, setting up an automated process to maintain the dashboard, a simple report, or a PowerPoint presentation.

Data Inspection

Once the business objective has been established, a data analyst will then set out to understand their data. This phase will introduce more technical work involving the initial data collection. There are three major areas:

1. **Collect initial data:** You already identified your data sources, here you will explore your company's databases, reports, or extract external data through web scraping to build your initial dataset.
2. **Determine data availability:** Identify how often this data is gathered or updated. Also, how you would be able to access it for future use.
3. **Explore data and characteristics:** Where you take a first look to identify important variables, data types, and the format of your data. You will also determine if you need to gather more data, **data enrichment**, and identify the initial data that will be used for analysis.

Tools for success: To collect the initial data, a data analyst can typically use SQL to explore a database. If there is data that needs to be scraped from the web or even a PDF file, programming tools such as python or R can be used.

Data Pre-processing & Preparation

Most likely your data will be messy. In this phase you will be correcting, formatting, and transforming your data for your analysis. Here, we will provide a brief overview as more detail will be presented in chapter 14. The major elements of this phase can include:

- **Data validation:** You want to ensure that your data follows the business logic or rules. This can be investigated by exploring the ranges or formatting of your variables. Mistakes can occur, especially when data is entered manually. Essentially, you verify whether your data makes

sense. For example, if you have a customer table with an age of 150, this indicates an error.

- **Missing value treatment:** Missing data is a common issue when data cleaning. There are multiple methods that include deletion, ignoring the missing data, assigning a missing data column, and imputation. More details on these methods will be explained in
- **Removing duplicates:** Duplicates will lead to misleading numbers and will cause incorrect conclusions.
- **Outlier treatment:** Where you will identify outliers and determine how you will treat them. An outlier is a data point that is significantly different from most of the other data points. They are normally the result of normal variation of a process or errors. There are multiple treatment methods including ignoring them, imputation, deletion, or transformation.
- **Data normalization:** When data is moved through different tools and phases during the data pipeline, the data types may involuntarily convert. Here you will fix the formatting, convert units of measurements, or standardize categorical data.
- **Feature engineering:** Where you transform variables to better represent the data. This can involve binning, aggregations, or combining variables.

Exploratory Data Analysis

After successfully cleaning the data, now it is time to explore the data. Here, we will provide a brief overview as more detail will be presented in chapters 7, 13, 15, 16, and 17. In this phase, a data analyst will conduct univariate, bivariate, and multivariate analyses. The analyses will identify patterns, trends, and other information to uncover insights that support the business objective.

Tools for success: For EDA, a data analyst can use tools such as SQL, Excel, or business intelligence tools such as Tableau, Power BI, or Looker.

Knowledge of descriptive statistics is necessary to understand how to summarize and measure your data. To provide more advanced analysis for decision making, hypothesis testing is helpful for building and testing assumptions. It can enhance the credibility of your findings to support your recommendations.

Data Validation

After exploring your data, you want to ensure your analyses make sense according to the business logic. Like in the data preparation phase, you will perform a second data validation step to verify your numbers make sense before you present them in your deliverable. It is often helpful to include data quality checks within your process. **Tools for success:** Many different tools can be used to perform data validation or quality checks. Microsoft Excel can be used to quickly compare expected and actual values. If you are building and automating a data pipeline, implementing unit tests or error handling is essential to ensure errors will be caught and dealt with for future data.

Explanatory Data Analysis

The last step is presenting your analyses. This can be done through a dashboard, PowerPoint, or a simple report. More detail will be discussed in chapters 18 and 19. **Tools for success:** If a dashboard or report is your deliverable, normally a business intelligence tool such as Tableau, Power BI, or Looker can be used. Reports can also be created in Microsoft Excel. If a presentation is needed, a data analyst would need to be skilled in building slide decks and oral communication. No matter what format the deliverable will be, data storytelling skills will be necessary.

Summary

In this chapter we went over the importance of understanding the business context of a project. This included an overview of each phase of the data analytics lifecycle while including the typical roles and responsibilities of a data analyst in each phase. Different tools were provided to aid a data analyst's understanding of the business problem of a project. We introduced certain topics such as statistics, SQL, and exploratory data analysis that we will cover in more detail in future chapters.

2 Introduction to SQL

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



As we embark on our journey to become data analysts it's crucial to equip ourselves with the right tools and knowledge to succeed in this exciting field. Our first stop will be SQL. In this chapter and the next, we'll introduce you to the world of SQL. For a data analyst, it's a must-have to grasp the fundamentals that underpin the SQL language for managing and manipulating data. SQL enables data analysts to interact with databases. With the use of SQL, we can get, insert, update, and delete data from the database. Therefore, mastering SQL is essential for any aspiring data analyst, and understanding its core concepts will help you avoid common mistakes. In this chapter, we will explore the various aspects of SQL. Here are the topics we'll cover:

- SQL and its use cases
- Different databases and database management systems
- SQL terminology
- Setting up your environment
- Writing basic SQL queries
- Filtering and organizing data with clauses
- Using operators and functions

Let's dive into the fascinating world of SQL and kickstart our journey toward becoming a data analyst!

SQL and its use cases

SQL is the abbreviation of Structured Query Language. It is the language we need to interact with the database. A database is a collection of organized information. This information is also known as data. Typically, when we talk about databases, we mean electronically stored databases. Let's think of an example for a database. It usually helps me to picture something that we use on a daily basis. For example, the database of the local library. They need to keep track of a number of things, amongst others:

- The books they have and information about these books.
- The members they have and information about them.
- Who borrowed which book when.

This information can all be stored in a database. A database like this would consist of different tables:

- Books
- Members
- Loans

And these tables would have columns to represent certain pieces of information. The book table could have the following columns:

- Title
- Author
- ISBN
- Publication year
- Publisher

In *Figure 2.1* you can see an example of what that table could look like. Please mind that this is a simplified version, we'll make this bit by bit slightly more complicated as we learn more.

title	author	isbn	publication_year	publisher
Data Analytics for	Emeka	978-	2020	DataPress

Beginners	Okafor	1111111 1	Publishing
The Art of Data Analysis	Mei Lin	978- 2222222 2	Analytical Press
Unleashing the Data Analyst Within	Jana Nováková	978- 3333333 3	Insightful Books
Data-Driven Decisions	Femi Adeyemi	978- 4444444 4	Pinnacle Publications
Mastering Data Visualization Techniques	Surya Chaudhary	978- 5555555 5	Graphical Press
A Data Analyst's Guide to Machine Learning	Sofia Álvarez	978- 6666666 6	ML Books Inc.
Exploring the World of Data Science	Eun Ji Park	978- 7777777 7	Data Science Press

Figure 2.1 – Table of books

As you can see, every row of the table represents a book. And don't get too excited about these really cool titles, it's unfortunately dummy data. So, at this point you may wonder, great, what part about this is SQL? And that would be a great question! We can use SQL to do the following things:

- Create the database;
- Create the tables and their columns;
- Get, add, update, and delete rows from the tables.

Since we are focusing on the data analytics purposes, the last use case is the most important for us. We will need to get data from a table and do all sorts of things with it. But let's start with the origin of SQL first.

Brief History of SQL

SQL is a language with an origin that dates half a century back already. In the early 1970s, IBM researchers Donald D. Chamberlin and Raymond F. Boyce developed a language called SEQUEL (Structured English Query Language) to interact with relational databases. They based this on the relational model of Edgar F. Codd. SEQUEL was later renamed to SQL, which stands for Structured Query Language. There are two accepted ways of pronouncing SQL. You can spell the word and pronounce it like separate letter (S-Q-L), or you can call it “sequel” like the original version. (You should probably just pick one and don’t be like me and constantly mix it up.) Over the years, SQL has evolved to become the standard language for managing and manipulating data in relational databases. SQL is now both an ANSI (American National Standards Institute) and ISO (International Organization for Standardization) standard, ensuring consistency and compatibility across various database management systems (DBMS). SQL is an international standard. It is widely used by data analysts, developers, and database administrators to interact with databases and perform a range of tasks. These tasks range from getting data to generate a report to controlling user access. For us the data analyst purposes are most interesting. Let’s have a look at SQL’s role in data analysis and what we can do with it.

SQL and data analysis

SQL plays an important role by enabling analysts to access data in the database. They can use it to manipulate and analyze the data. As already mentioned, SQL is universally accepted as the standard language for database interaction. Since SQL is used a lot, it works with very many different databases, such as MySQL, Oracle, SQL Server, and PostgreSQL, and many more. The querying abilities of SQL are rather extensive. We can write queries that can be used to get, filter, sort and manipulate database data. This can be used to answer specific questions and to generate reports. Not only can we get that with SQL, it also gives us the power to insert, update, and delete records in a database. These operations are crucial for maintaining data accuracy and organization. With SQL we can also set and update the structure of the database. For example, we can create and alter tables. This option is essential for setting up databases tailored to specific business requirements. Admittedly, this is not the most common thing you’ll do with SQL as a data analyst, but it can be occasionally what you need to do. As you can see, we

can do a lot with SQL, and it has an important role in data analysis. That's why it's an absolute must-have to master SQL and set a strong foundation for your data analyst career. Let's talk briefly about the different types of databases before learning more about SQL.

Different Databases

We use SQL to interact with a database. There are different types of databases out there. Databases typically have a Database Management Systems (DBMS) available to inspect and manage data. Each DBMS has its own set of features and strengths, making them suitable for specific applications and use cases. Let's first explore the differences between relational and non-relational databases and then discuss some popular database management systems.

Relational vs. Non-Relational Databases

The main distinction we need to make is relational and non-relational databases. SQL deals with the relational databases, so let's start with that one.

Relational Databases

Relational databases use a schema to define the structure of the data, organizing it into tables with rows and columns. Each row in a table represents a record, and each column represents an attribute or field of that record. The key concept in relational databases is that tables can have relations. Every table works with a special column id, and other tables can refer to an item of that element with the specified id. This might sound a little vague, so let's upgrade our previous table example a bit. In *Figure 2.2* you can see that we now have a table with authors.

i	name	country	birth_yea	email
d			r	
1	Mei Lin	China	1978	mei.lin@example.com
2	Jana Nováková	Czech Republic	1990	jana.novakova@example.com

3	Femi Adeyemi	Nigeria	1986	femi.adeyemi@example.com
4	Sofia Álvarez	Spain	1980	sofia.alvarez@example.com
5	Eun Ji Park	South Korea	1985	eunji.park@example.com
6	Emeka Okafor	Nigeria	1982	emeka.okafor@example.com
7	Surya Chaudhary	India	1992	surya.chaudhary@example.com

In *Figure 2.3* we see the book table. And as you can see, we now use the id of the author to represent the author.

i	title	autho r_id	isbn	publication _year	publisher
1	Data Analytics for Beginners	6	978- 11111111 1	2020	DataPress Publishing
2	The Art of Data Analysis	1	978- 22222222 2	2019	Analytical Press
3	Unleashing the Data Analyst Within	2	978- 33333333 3	2021	Insightful Books
4	Data-Driven Decisions	3	978- 44444444 4	2018	Pinnacle Publications
5	Mastering Data Visualization Techniques	7	978- 55555555 5	2020	Graphical Press
6	A Data Analyst's Guide to Machine Learning	4	978- 66666666 6	2022	ML Books Inc.
7	Exploring the World of Data Science	5	978- 77777777 7	2019	Data Science Press

We call the id column in the table the primary key. A primary key is a unique

identifier for each record in a table, ensuring that no two records have the same value for the primary key attribute. Primary keys help enforce data integrity and consistency, and they are used to establish relationships between tables. When we use this key in another table to refer to the record in the other table, it's called a foreign key. SQL is the standard language for interacting with relational databases. Relational databases are very common. Some of the advantages of relational databases are data consistency, support for complex queries, and ease of data retrieval. There are also some downsides, it's hard to work with unstructured data and it becomes slow for large data volumes. In these cases, non-relational databases might be a better option.

Non-Relational Databases

Non-relational databases are commonly called NoSQL databases. They do not rely on a fixed schema or table-based structure. Instead, they store data in various formats. Some examples of these formats include key-value pairs, documents, column families, and graphs. Non-relational databases are often used when dealing with large volumes of unstructured or semi-structured data. They might also be the better option when high write and read performance is required. Their flexibility and ability to handle diverse data types make them an attractive choice for many big data situations. Dealing with NoSQL databases is not in the scope of this book. Let's see some of the popular database management systems that you might work with instead.

Popular DBMS's

There are several popular DBMS options available. They all have their own set of features and strengths. Let's list the most popular ones.

- **MySQL:** MySQL is an open-source, cross-platform relational database management system. It is widely used for web applications. It is an excellent choice for many small to medium-sized applications. It is easy to use and set up.
- **PostgreSQL:** PostgreSQL is an open-source, object-relational database system. It supports advanced data types and indexing, making it suitable for large-scale applications and complex queries.

- **SQL Server:** Developed by Microsoft, SQL Server is an enterprise-grade relational database management system. It offers a range of tools and features, such as advanced analytics, reporting, and integration with other Microsoft products. It's a great option for organizations already using the Microsoft ecosystem.
- **Oracle Database:** Oracle Database is a high-performance relational database management system designed for large-scale and mission-critical applications. It offers advanced features such as partitioning, data warehousing, and real application clusters. It is a popular choice for enterprise environments.
- **SQLite:** SQLite is a lightweight, self-contained, serverless relational database management system. It is used a lot in embedded systems and mobile applications. It's simple and reliable, that's why it's ideal for situations where a full-fledged DBMS might be overkill.

As a new data analyst, you're probably not going to be the person choosing a certain database. However, in your data analyst journey, you will likely encounter and work with various databases and DBMS. So, knowing these common ones and understanding their strength, will help you along the way. Next up, we'll be dealing with some SQL terminology that is going to be necessary for your success as a data analyst.

SQL Terminology

Let's familiarize ourselves with key SQL terminology to better understand the structure of relational databases and the language used to interact with them. In this section, we'll go through a brief explanation of what they are and what their role in data management is. Let's start with one of the most central terms: query.

Query

A **query** is a request for specific data or information from a database. In SQL, queries are used to get, update, insert, or delete data stored in tables. Queries are written using SQL statements, which are composed of clauses and keywords. Let's see what statements are next.

Statement

An SQL **statement** is a text string composed of SQL commands, clauses, and keywords. It is used to perform the various query tasks such as creating, updating, deleting, or getting data in a relational database. SQL statements are the building blocks of SQL queries, and they typically contain one or more clauses. Which brings us to the next key concept: clause.

Clause

A **clause** is a part of an SQL statement that performs a specific function or operation. Clauses can for example be used to define conditions, specify sorting, group data, or join tables. Some common SQL clauses include SELECT, FROM, WHERE, and ORDER BY. Each clause is often associated with specific SQL keywords.

Keyword

Keywords are reserved words in SQL that have a predefined meaning and are used to construct SQL statements. They help define the structure and syntax of a query. Examples of SQL keywords include SELECT, INSERT, UPDATE, DELETE, and CREATE. When writing more complex SQL queries, you'll often use keywords in conjunction with views.

View

A **view** is a virtual table that is based on the result of an SQL query. It does not store data itself but provides a way to access and manipulate data from one or more underlying tables. Views can be used to simplify complex queries and customize data presentation for specific users. Let's move on to setting up our environment, so that we can get some practice with writing SQL soon.

Setting up your environment

Setting up your environment is typically a tough job. And it's hard to write

general instructions that will work for all the systems out there. That's why we'll keep these instructions somewhat superficial, and why I have some online options for you if all else fails. Your first place to start would be the official documentation of the database. However, if that's too complicated, you and Google (or Bing or DuckDuckGo or whatever search engine you prefer) should be fine. This is something that is done a lot. I would search for example for: Setting up mysql macos Setting up postgres windows Tutorial installing postgres macos Etc. In this section, we will guide you through the process of choosing a DBMS, installing the necessary software, and creating a sample database. It's expected that you'll need to find some additional information yourself for your own system. After that, we're ready to start exploring SQL in-depth.

Choosing a DBMS

Selecting a suitable database management system is the first step in setting up your environment. There are several popular options out there such as MySQL, PostgreSQL, SQL Server, Oracle, and SQLite. For beginners, MySQL or SQLite are excellent choices due to their simplicity. In the examples, we'll be using MySQL, so that would be my recommendation to go with for now.

Installing necessary software

Once you have chosen your preferred DBMS, you'll need to install the appropriate software. This typically includes:

- the database server itself;
- a client for interacting with the server;
- a graphical user interface (GUI) tool for managing databases and running SQL queries.

So, let's assume you'll be installing MySQL, this is what you'll need to install:

- MySQL Server
- MySQL Workbench

You can follow the official installation guides provided by the respective database management system to ensure a smooth setup. Here's the one for MySQL:

- Windows: <https://dev.mysql.com/doc/mysql-installation-excerpt/8.0/en/macos-installation.html>
- MacOS: <https://dev.mysql.com/doc/mysql-installation-excerpt/8.0/en/macos-installation.html>
- Linux: <https://dev.mysql.com/doc/mysql-installation-excerpt/8.0/en/linux-installation.html>

Once you've managed to do this, it's time to set up a sample database that we can use.

Creating a sample database

Now, we'll walk through the steps of creating a sample database in MySQL. We need to start SQL server. On Windows, you might need to open up services, search for MySQL server and start it. On my MacOS system I type:

```
brew services start mysql
```

After this, we can open MySQL Workbench. This brings us to the start screen.

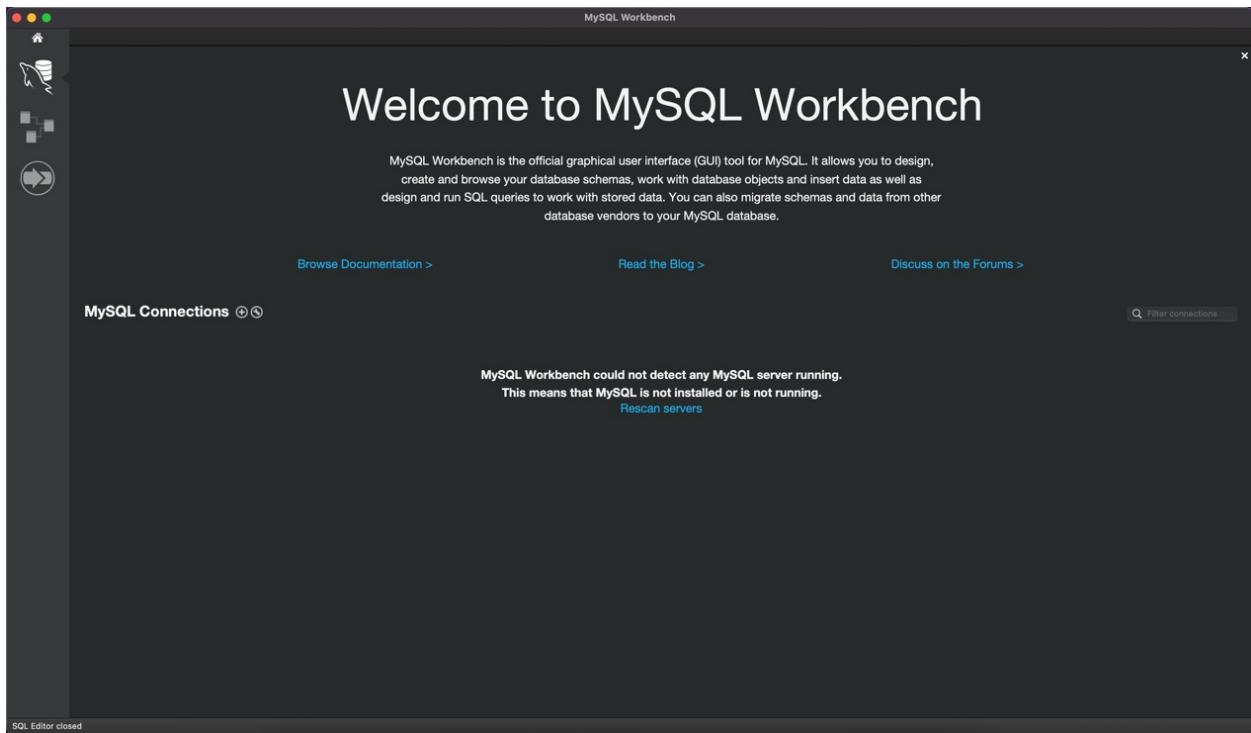


Figure 2.4 – Welcome to MySQL Workbench

In order to connect to the database, we need to add a MySQL connection. We can do this by clicking on the + next to MySQL connection in *Figure 2.4*. This brings us to the screen in *Figure 2.5*.

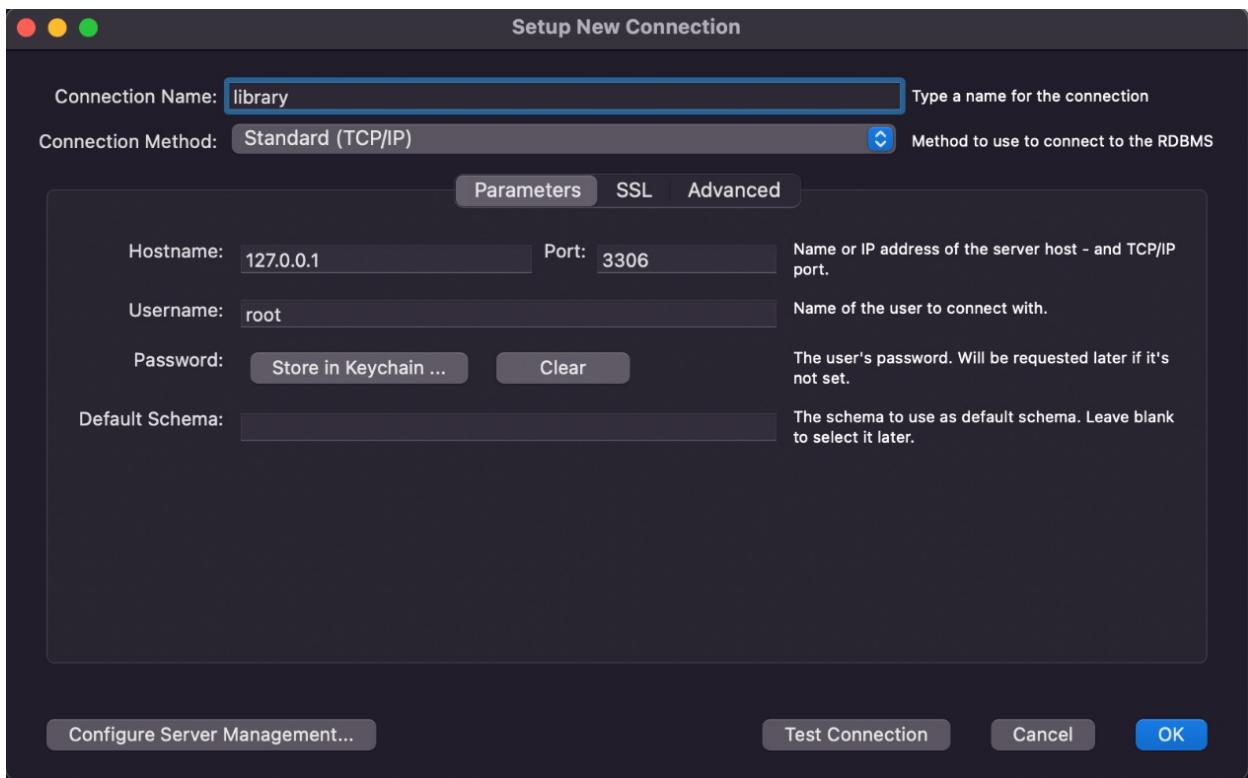


Figure 2.5 – Setting up a connection

Fill out the form as above, and click on OK. The connection library should now appear. We can double click it, and the connection should open. It should look like *Figure 2.6*.

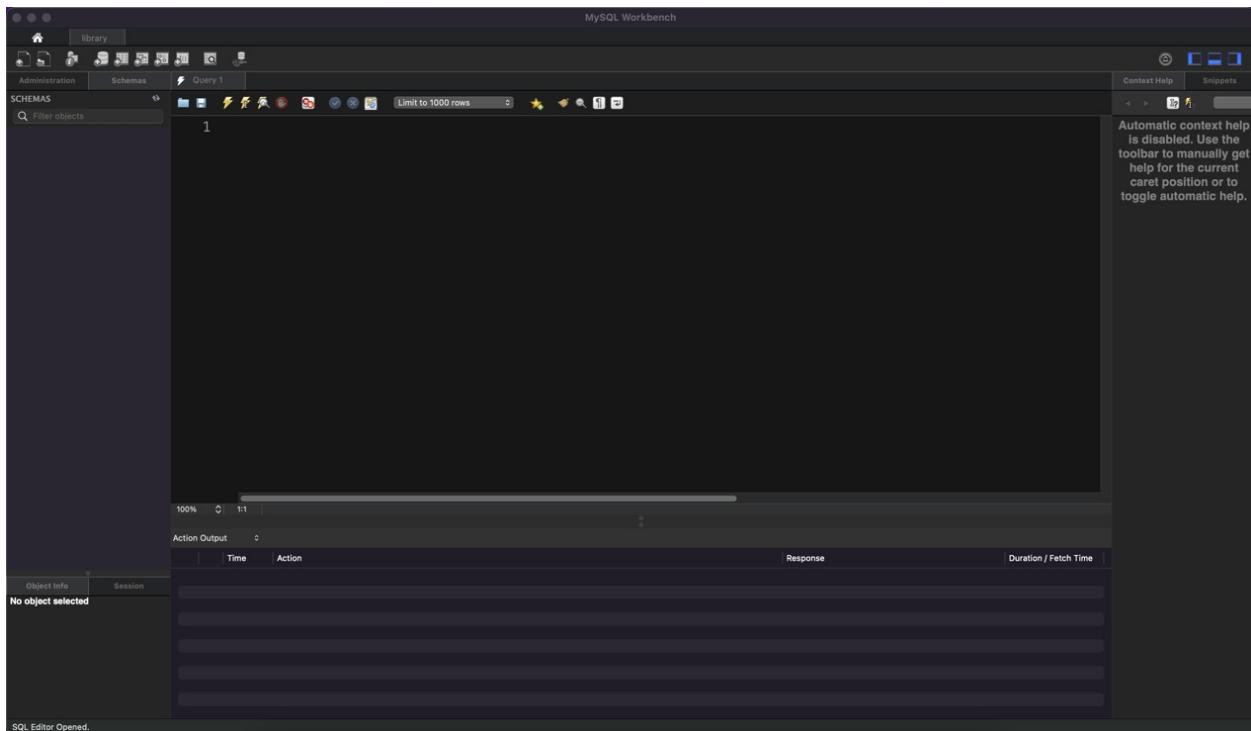


Figure 2.6 – Opened connection

Alright, if you're here, that's a great place to be! Let's add in a schema and some tables. For that, you can use the following code snippet (you can find this in the associated GitHub folder):

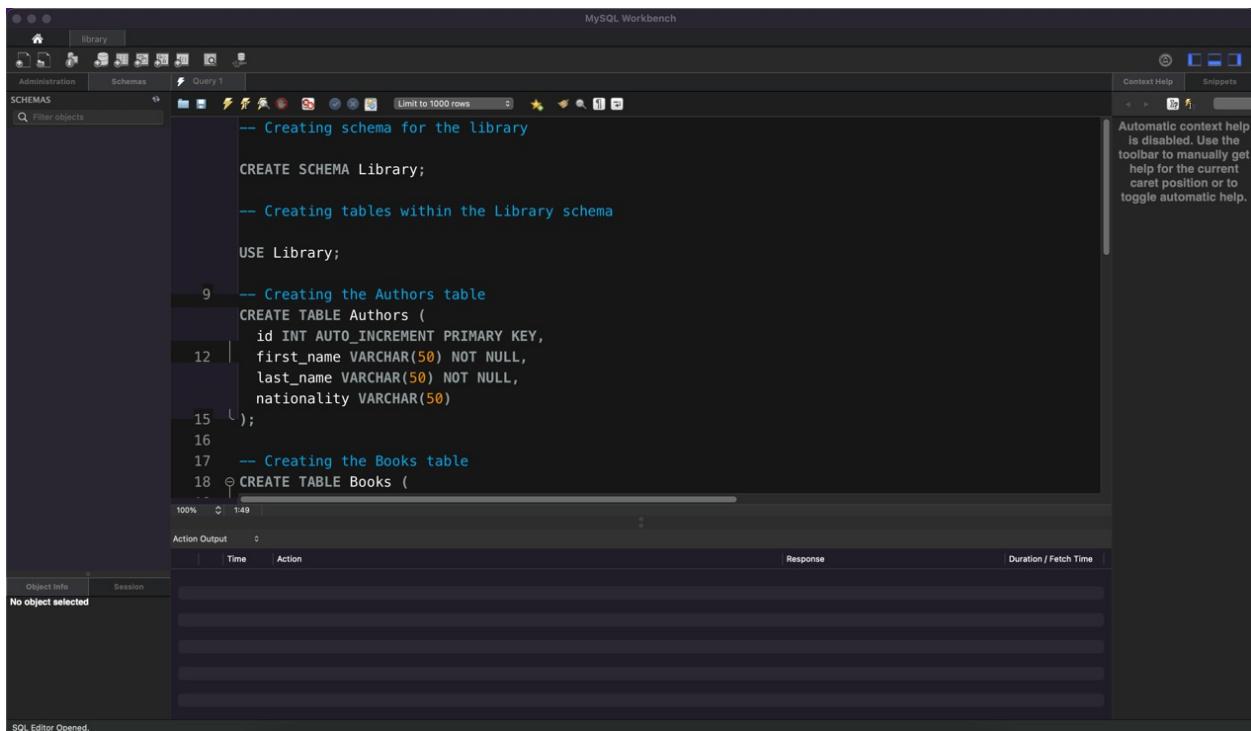
```
-- Creating schema for the library
CREATE SCHEMA Library;
-- Creating tables within the Library schema
USE Library;
-- Creating the Authors table
CREATE TABLE Authors (
    id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    nationality VARCHAR(50)
);
-- Creating the Books table
CREATE TABLE Books (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    author_id INT,
    isbn VARCHAR(20),
    publication_year INT,
    publisher VARCHAR(50),
```

```

    FOREIGN KEY (author_id) REFERENCES Authors(id)
);
-- Creating the Members table
CREATE TABLE Members (
    id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    join_date DATE,
    email VARCHAR(100) UNIQUE
);
-- Creating the BorrowedBooks table
CREATE TABLE BorrowedBooks (
    id INT AUTO_INCREMENT PRIMARY KEY,
    book_id INT,
    member_id INT,
    borrow_date DATE,
    due_date DATE,
    return_date DATE,
    FOREIGN KEY (book_id) REFERENCES Books(id),
    FOREIGN KEY (member_id) REFERENCES Members(id)
);

```

And paste it in the query editor, like shown in *Figure 2.7*:



The screenshot shows the MySQL Workbench interface with the SQL editor tab active. The editor contains the following SQL code:

```

-- Creating schema for the library
CREATE SCHEMA Library;

-- Creating tables within the Library schema
USE Library;

-- Creating the Authors table
CREATE TABLE Authors (
    id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    nationality VARCHAR(50)
);

-- Creating the Books table
CREATE TABLE Books (

```

The code is being typed into the editor, with line numbers 9 through 18 visible on the left. The right pane of the interface displays a message about context help being disabled.

Figure 2.7 – Query editor with the SQL statements for schema and table

creation

We now need to hit the lightning icon (the most left one). Make sure you didn't select a part of the SQL, because then it will only try and execute that part. It should show green checks and success messages in the bottom part. The tables won't show until you hit the refresh icon next to schemas (upper left part). You can see the newly created tables on the left in *Figure 2.8*.

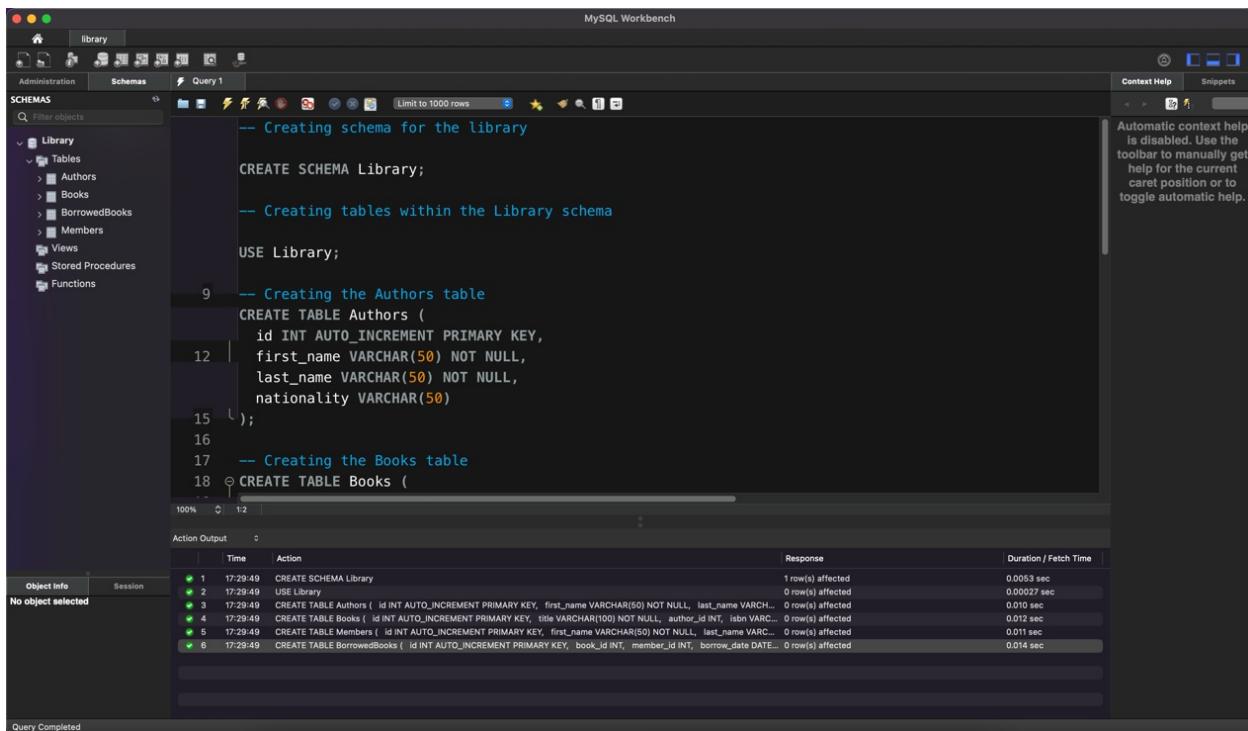


Figure 2.8 - Tables

The tables are still empty. In order to add some data in, we need to execute another set of statements. Go ahead and clear the query editor. Then copy the below code snippet in:

```
-- Populating the Authors table
INSERT INTO Authors (id, first_name, last_name, nationality) VALUES
(1, 'Akira', 'Suzuki', 'Japanese'),
(2, 'Amara', 'Diop', 'Senegalese'),
(3, 'Johannes', 'Müller', 'German'),
(4, 'Ying', 'Li', 'Chinese'),
(5, 'Aisling', 'O'Brien', 'Irish'),
(6, 'Carlos', 'Fernandez', 'Spanish'),
```

```

(7, 'Adeola', 'Adeyemi', 'Nigerian'),
(8, 'Anastasia', 'Ivanova', 'Russian'),
(9, 'Sofia', 'Silva', 'Portuguese'),
(10, 'Vivek', 'Patel', 'Indian');

-- Populating the Books table
INSERT INTO Books (id, title, author_id, isbn, publication_year,
publisher) VALUES
(1, 'Data Analysis with SQL', 1, '1234567890123', 2022, 'TechPress'),
(2, 'SQL for Data Analysts', 2, '2345678901234', 2021, 'DataBooks'),
(3, 'Mastering SQL for Data Analysis', 3, '3456789012345', 2020, 'Analytics Publishing'),
(4, 'Efficient Data Analytics with SQL', 4, '4567890123456', 2019, 'TechPress'),
(5, 'Advanced SQL Techniques for Data Analysts', 5, '5678901234567', 2018, 'DataBooks'),
(6, 'SQL for Business Intelligence', 6, '6789012345678', 2021, 'Analytics Publishing'),
(7, 'Data Wrangling with SQL', 7, '7890123456789', 2020, 'TechPress'),
(8, 'SQL for Big Data', 8, '8901234567890', 2019, 'DataBooks')

',
(9, 'Data Visualization Using SQL', 9, '9012345678901', 2018, 'Analytics Publishing'),
(10, 'SQL for Data Science', 10, '0123456789012', 2021, 'TechPress');

-- Populating the Members table
INSERT INTO Members (id, first_name, last_name, email, join_date
) VALUES
(1, 'Alice', 'Johnson', 'alice.johnson@example.com', '2021-01-01'),
(2, 'Bob', 'Smith', 'bob.smith@example.com', '2021-02-15'),
(3, 'Chiara', 'Rossi', 'chiara.rossi@example.com', '2021-05-10'),
(4, 'David', 'Gonzalez', 'david.gonzalez@example.com', '2021-07-20'),
(5, 'Eve', 'Garcia', 'eve.garcia@example.com', '2021-09-30'),
(6, 'Femi', 'Adeyemi', 'femi.adeyemi@example.com', '2021-11-10'),
(7, 'Grace', 'Kim', 'grace.kim@example.com', '2021-12-15'),
(8, 'Henrik', 'Jensen', 'henrik.jensen@example.com', '2022-03-05'),
(9, 'Ingrid', 'Pettersson', 'ingrid.pettersson@example.com', '2022-04-20'),
(10, 'Jia', 'Wang', 'jia.wang@example.com', '2022-06-01');

-- Populating the Borrowed_Books table
INSERT INTO BorrowedBooks (id, book_id, member_id, borrow_date,

```

```

due_date) VALUES
(1, 1, 1, '2022-04-01', '2022-05-01'),
(2, 2, 2, '2022-04-05', '2022-05-05'),
(3, 3, 3, '2022-04-10', '2022-05-10'),
(4, 4, 4, '2022-04-15', '2022-05-15'),
(5, 5, 5, '2022-04-20', '2022-05-20'),
(6, 6, 6, '2022-04-25', '2022-05-25'),
(7, 7, 7, '2022-04-30', '2022-05-30'),
(8, 8, 8, '2022-05-02', '2022-06-02'),
(9, 9, 9, '2022-05-05', '2022-06-05'),
(10, 10, 10, '2022-05-08', '2022-06-08');

```

Run this by clicking on the lightning icon. Again, you should get green confirmation messages that the data is in there. Without writing any SQL ourselves, we can verify that the data is in there. Right click on the “BorrowedBooks” on the left, and then click on “Select Rows – Limit 1000”, as displayed in *Figure 2.9*.

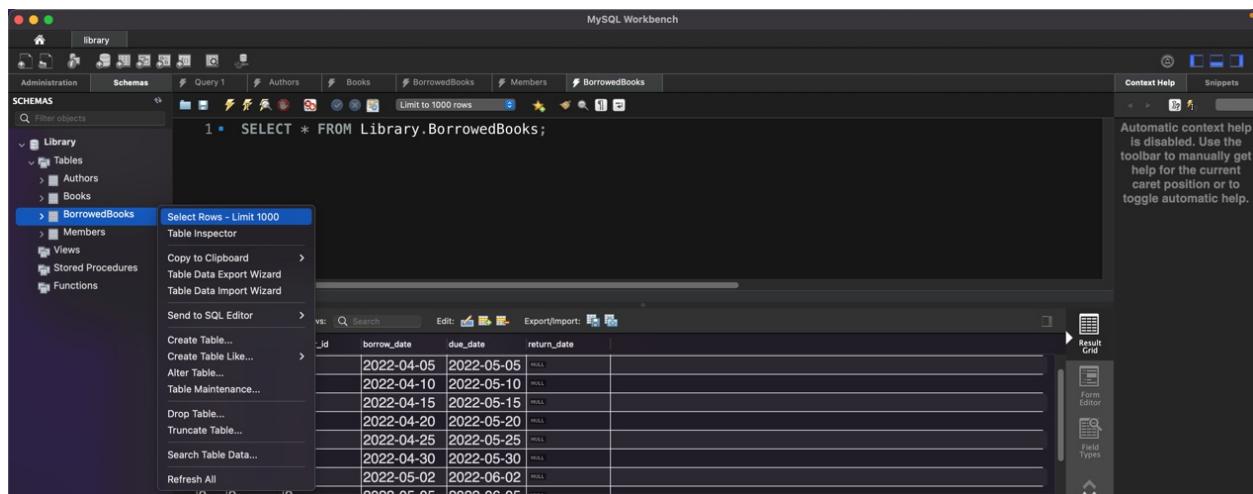


Figure 2.9 – Select the first 1000 records

It should then give the output shown in *Figure 2.10*.

id	book_id	member_id	borrow_date	due_date	return_date
1	1	1	2022-04-01	2022-05-01	NULL
2	2	2	2022-04-05	2022-05-05	NULL
3	3	3	2022-04-10	2022-05-10	NULL
4	4	4	2022-04-15	2022-05-15	NULL
5	5	5	2022-04-20	2022-05-20	NULL
6	6	6	2022-04-25	2022-05-25	NULL
7	7	7	2022-04-30	2022-05-30	NULL

Figure 2.10 – Data in table BorrowedBooks

And that's it! You have your system up and running. Let's finally get to it: writing our own SQL statements.

Writing Basic SQL Queries

Now that you have set up your environment and created a sample database, it's time to dive into writing SQL queries. We will introduce the most common SQL statements used for getting and updating data. We will continue to work with our library database here. We will cover SELECT, INSERT, UPDATE, and DELETE statements, and explore the structure of a query.

SELECT Statement

The **SELECT** statement is the foundation of data retrieval in SQL. We use it to fetch data from one or more tables within a database. The basic structure of a SELECT query is as follows:

```
SELECT column_name(s)
FROM table_name
WHERE conditions;
```

For example, to retrieve all books in our library catalog, the query would look like:

```
SELECT *
FROM books;
```

With the * we say we'd like to get all the columns. And we specify the table books. This will return all the books in our table, as you can see in *Figure 2.11*.

The screenshot shows a SQL query interface with the following details:

- Query:** 1 • `SELECT * FROM books;`
- Result Grid:** A table displaying 10 rows of book data.
- Columns:** id, title, author_id, isbn, publication_year, publisher.
- Data:** The table contains 10 rows of data corresponding to the books listed in the figure.

	id	title	author_id	isbn	publication_year	publisher
▶	1	Data Analysis with SQL	1	1234567890123	2022	TechPress
	2	SQL for Data Analysts	2	2345678901234	2021	DataBooks
	3	Mastering SQL for Data An...	3	3456789012345	2020	Analytics Publishing
	4	Efficient Data Analytics wit...	4	4567890123456	2019	TechPress
	5	Advanced SQL Technique...	5	5678901234567	2018	DataBooks
	6	SQL for Business Intellige...	6	6789012345678	2021	Analytics Publishing
	7	Data Wrangling with SQL	7	7890123456789	2020	TechPress
	8	SQL for Big Data	8	8901234567890	2019	DataBooks
	9	Data Visualization Using S...	9	9012345678901	2018	Analytics Publishing
	10	SQL for Data Science	10	0123456789012	2021	TechPress

*Figure 2.11 – The result of the `SELECT * from books` query*

We could also have specified only the title and isbn column:

```
SELECT title, isbn
FROM books;
```

That would have given us the output that you can see in *Figure 2.12*.

The screenshot shows a SQL query being run in a terminal window. The query is:

```
1 • SELECT title, isbn FROM books;
```

The results are displayed in a "Result Grid" table:

	title	isbn
▶	Data Analysis with SQL	1234567890123
	SQL for Data Analysts	2345678901234
	Mastering SQL for Data An...	3456789012345
	Efficient Data Analytics wit...	4567890123456
	Advanced SQL Technique...	5678901234567
	SQL for Business Intellige...	6789012345678
	Data Wrangling with SQL	7890123456789
	SQL for Big Data	8901234567890
	Data Visualization Using S...	9012345678901
	SQL for Data Science	0123456789012

Figure 2.12 – Select statement for only two columns

By the way, SQL is case insensitive. That means that we could spell the column and table names in both upper- and lowercase, and it would still know what column or table we're referring to.

Structure of a Query

A SQL query is composed of several clauses, each serving a specific purpose. The main clauses in a SELECT query are:

- **SELECT:** Specifies the columns to be retrieved.
- **FROM:** Indicates the table(s) from which to fetch the data.

- **WHERE:** Filters the data based on the specified condition(s).

The Where clause is optional, we didn't use it in the examples before. Without a Where clause, it will simply return all the data. We already inserted data into our table, but we didn't discuss that code yet. Let's have a closer look at the INSERT statement.

INSERT Statements

INSERT statements are used for adding new data to a table. The basic structure of an INSERT query is:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

For example, to add a new book to our library catalog, the query would look like:

```
INSERT INTO books (title, author_id, publication_year)
VALUES ('How to SQL book', 1, 2015);
```

Please note that we're not specifying all the columns here. Depending on how the database is created, this can or cannot be done. In our case, we can. It will add the value `null` for the missing columns. Sometimes you need to change some values of a row after adding it. This can be done with the UPDATE statement.

UPDATE Statements

With the **UPDATE** statements we can modify existing data in a table. The basic structure of an UPDATE query is:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE conditions;
```

For example, to update the publication year of a book in our library catalog, the query would look like:

```
UPDATE books
```

```
SET publication_year = 2016  
WHERE id = 11;
```

Using the WHERE clause for other columns than the id might require you to disable safe update mode.

```
SET SQL_SAFE_UPDATES = 0;  
UPDATE books  
SET publication_year = 2017  
WHERE title = 'How to SQL book';
```

We now use the first line to turn off safe update mode. And after that, we can update based on non-id fields. We can also delete records from the table. Let's explore how to do that next.

DELETE Statements

This might be a surprise, but the **DELETE** statements are used for removing data from a table. The basic structure of a DELETE query is:

```
DELETE FROM table_name  
WHERE conditions;
```

For example, to remove a book from our library catalog, the query would look like:

```
DELETE FROM books  
WHERE title = 'How to SQL book';
```

This will remove the book that we just added. We really need to master these basic SQL statements to interact with data. Before we do so, let's see some basic syntax rules.

SQL basic rules and syntax

When writing SQL queries, it's essential to follow certain rules and conventions to ensure that your queries are correct and easy to ready. You've seen the ultimate basics of SQL, so it's time to discuss some of the basic syntax rules in SQL.

General syntax rules

SQL queries are composed of keywords, clauses, and expressions. The keywords are reserved words that have a specific meaning in SQL (e.g., SELECT, FROM, WHERE). If you have a column that is called from, you have to enclose it in backticks.

```
SELECT `from`  
FROM table_name;
```

Here we only select the column named “from”. And since that is a keyword, we have to enclose it in backticks in order to use it as a literal value.

Capitalization

While SQL is not case-sensitive when it comes to keywords and identifiers (such as table and column names), it is the convention to capitalize SQL keywords. This makes queries more readable. For example:

```
SELECT title, author  
FROM books  
WHERE publication_year > 2010;
```

In this query, the keywords SELECT, FROM, and WHERE are capitalized, while the table and column names remain in lowercase. However, the following also works:

```
SeLeCT tITLE, autHor_ID  
From BOOKS  
WheRE publiCatiON_yeaR > 2010;
```

Please don't. Thank you.

Single quotes

We use single quotes are used to enclose string literals, such as text or date values. For example:

```
INSERT INTO books (title, author)
```

```
VALUES ('The Art of Data Analysis', 'Jane Doe');
```

Here, the title and author values are enclosed in single quotes since they are string literals. We have seen the queries end with semicolons so far. Let's talk about it.

Semicolons

The semicolon is used to mark the end of an SQL statement. While some database management systems do not require semicolons, it's a good practice to include them for clarity and to avoid potential errors. For example:

```
SELECT * FROM books;  
UPDATE books SET price = price * 1.04;
```

In this example, semicolons are used to mark the end of each SQL statement. This helps to make it clear where one statement ends and the next begins. In this particular example we are increasing all the prices. The basic rules are important whenever you're writing SQL. That should be enough to be ready for becoming more proficient at filtering the data next.

Filtering and organizing data with clauses

One of the most common things to do is to filter and organize data according to specific criteria. That's why our next topic is just that: filtering data. We are going to discuss various clauses that will help to write queries for retrieving the exact information we need. We will cover the WHERE, ORDER BY, DISTINCT, and LIMIT clauses while working with our library catalog example. The first one we have seen briefly already, but let's have a closer look.

WHERE Clause

The **WHERE** clause filters records based on specified conditions. Above we filtered for the condition equal to a certain id or title. We use WHERE in conjunction with SELECT, UPDATE, and DELETE statements. For example, to find all books published before 2019 in our library catalog, the

query would look like:

```
SELECT *
FROM books
WHERE publication_year < 2019;
```

This will return two results. The books with id 5 and 9. Let's have a look at how we can sort our results.

ORDER BY Clause

We can sort our results with the **ORDER BY** clause. We can specify one or multiple columns to sort on. We can also choose to sort the results in ascending (ASC) or descending (DESC) order. For example, to retrieve books sorted by publication year in ascending order, the query would look like:

```
SELECT *
FROM books
WHERE publication_year < 2019
ORDER BY publication_year ASC;
```

We can also specify the next column to sort for. In our current example nothing changes, since both the results have publication year 2018:

5	Advanced SQL Techniques for Data Analysts	5	5678901234567
	2018 DataBooks		
9	Data Visualization Using SQL	9	9012345678901
	2018 Analytics Publishing		

So let's change our query:

```
SELECT *
FROM books
WHERE publication_year < 2019
ORDER BY publication_year, publisher ASC;
```

Now this will swap around the results, since when the publication year is the same, it will sort by publisher from A-Z.

9	Data Visualization Using SQL	9	9012345678901	2
018	Analytics Publishing			

5 Advanced SQL Techniques for Data Analysts 5 567890123456
7 2018 DataBooks

We can do more things with the clauses. Let's see how we can use **DISTINCT** to only get unique values.

DISTINCT Clause

The **DISTINCT** clause is used to eliminate duplicate records from the result set of a query. For example, to retrieve a list of unique publishers from our library catalog, the query would look like:

```
SELECT DISTINCT publisher  
FROM books;
```

This will return a list of the unique publishers in our code; this is the result:

```
TechPress  
DataBooks  
Analytics Publishing
```

Since we only select the publisher column, that's the only one we'll get. **DISTINCT** is often used for inner SELECT statements, something we'll learn about in the later chapters. Let's for now have a look at the **LIMIT** clause.

LIMIT Clause

We use the **LIMIT** clause to limit the number of records returned by a query. This can be particularly useful when working with large datasets or when you need to get only a specific number of records. For example, to fetch the top 5 oldest books in our library catalog, the query would look like:

```
SELECT *  
FROM books  
ORDER BY publication_year ASC  
LIMIT 5;
```

We select all the columns from the books table, and order them by publication year. We then only get the first 5 results, as indicated by **LIMIT 5**. These filtering clauses really add to what we can do and the

questions we can answer with the use of SQL. Let's add some other great tools in our SQL toolbox and learn how to work with operators and functions.

Using operators and functions

One of the reasons that SQL is so great for data analysis are the operators and functions. We can use them to perform complex operations and calculations. We can even do clever things with data manipulation. Let's use our library example again to make acquaintance with various types of operators and functions that you can use in your SQL queries. The first topic is the comparison operators. We have seen this already above, so let's hear about the details.

Comparison operators

The **comparison operators** are used in the WHERE clause to filter records based on the specified conditions. We have seen = already. This is for checking whether a certain field equals a certain value. We also saw the <, to check for books before a certain publication year. The common comparison operators include =, <>, >, <, >=, and <=. For example, to find books published in or after 2020 you would use the following query:

```
SELECT *
FROM books
WHERE publication_year >= 2020;
```

This will return the books with id 1, 2, 3, 4, 7 and 10. The other comparison operators work in a similar way. Sometimes you'd like to specify multiple conditions. This can be achieved with the logical operators,

Logical operators (AND, OR)

With the **logical operators** we can combine multiple conditions in a WHERE clause. The AND operator returns true if both conditions are true, while the OR operator returns true if either condition is true. For example, to find books published between 1990 and 2018 published by DataBooks, you would use the following query:

```
SELECT *
FROM books
WHERE publication_year >= 1990 AND publication_year <= 2018
AND publisher = 'DataBooks';
```

This will return only one result (the id is 5). When we check for a certain text value, we surround this text value with single quotes. As you can see in the example above. It is looking for an exact match. We can also look for the text occurring in the field. This can be done with the **LIKE** operator.

LIKE operator

The **LIKE** operator is used to search for a specified pattern in a column. A pattern could be:

- The text starts with a certain combination of characters
- The text ends with a certain combination of characters
- The text contains a certain combination of characters

We use the percentage symbol (%) to represent zero or more characters and the underscore (_) to represent a single character. For example, to find books with titles starting with "The", we would use the following query:

```
SELECT *
FROM books
WHERE title LIKE 'The%';
```

To find books with titles containing the word "Data", we could do this:

```
SELECT *
FROM Books
WHERE title LIKE '%Data%';
```

We can also do more things with the numeric data fields. This can be done with the use of the arithmetic operators.

Arithmetic operators

The **Arithmetic operators** (+, -, *, /, %) add the ability to perform calculations on numeric columns. Let's imagine that our books table has a

price column. We can calculate the discounted price of each book, assuming a 10% discount:

```
SELECT title, price, price * 0.9 AS discounted_price  
FROM books;
```

In this SQL query, an alias is used. This is the `AS discounted_price`. We call the result of the calculation `discounted_price`. The alias is used to give a temporary name to a calculated column, this makes it easier to understand the result and it also enables us to reuse the value. This is a common thing to do when working with arithmetic operators. Let's see another example, let's calculate the costs of buying two books (assuming they have the same price):

```
SELECT title, price, price + price AS total_cost_for_two_books  
FROM books;
```

We can also work with multiple arithmetic operators. Let's calculate the price difference between the original price and a discounted price of 15%:

```
SELECT title, price, price - (price * 0.85) AS price_difference  
FROM books;
```

Or we could get the value of a single page with the following query where we calculate the price per page of each book:

```
SELECT title, price, page_count, price / page_count AS price_per  
_page  
FROM books;
```

And lastly, I had to be a bit creative with my example here. Let's assume we have a special lucky 7 discount campaign and we want to find the remainder when the price is divided by 7. Here's how to do that:

```
SELECT title, price, price % 7 AS remainder  
FROM books;
```

We can do a lot with the arithmetic operators, it basically creates a new temporary column for our records. We can also work with function to get information about the entire data set.

Functions for calculations

SQL includes several functions to perform calculations, such as COUNT, SUM, AVG, MIN, and MAX. For example, to find the total number of books in our library catalog, you would use the following query:

```
SELECT COUNT(*)
FROM books;
```

Let's say that our books table had a column page count (it doesn't so this won't work without adding it). We can calculate the average number of pages per book by using the AVG function:

```
SELECT AVG(page_count)
FROM books;
```

The total of all the pages of all the books in our library could be calculated like this:

```
SELECT SUM(page_count)
FROM books;
```

And we could find the publication year of the oldest book with the use of the MIN function:

```
SELECT MIN(publication_year)
FROM books;
```

There are also functions to manipulate text values. We call a text values commonly a **string**.

Functions for text manipulation

We can use **Text manipulation functions** to transform string data. Some of the popular functions include CONCAT, UPPER, LOWER, and LENGTH. For example, to display book titles in uppercase, you would use the following query:

```
SELECT UPPER(title)
FROM books;
```

And to get them in lowercase we would say:

```
SELECT LOWER(title)  
FROM books;
```

We can use the LENGTH function to get the number of characters in string. The following SQL will give back a number for each title, representing the length.

```
SELECT LENGTH(title)  
FROM books;
```

The CONCAT function is used to combine two or more strings together. Let's use the CONCAT function to display the author's first name and last name as a single column called `full_name` in the query results:

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name  
FROM authors;
```

In this example, we use the CONCAT function to combine the `first_name` and `last_name` columns from the authors table, separated by a space (' '). Next, we use the alias `full_name` to give a meaningful name to the combined column in the query results. Let's see the last type of functions we're going to discuss: date functions.

Date functions

Date functions are used to manipulate and perform calculations on date and time values. Some common date functions include `CURRENT_DATE`, `CURRENT_TIMESTAMP`, `DATE_ADD`, and `DATE_DIFF`. For example, to find borrow sessions that were due before today, we can say:

```
SELECT *  
FROM BorrowedBooks  
WHERE due_date < CURRENT_DATE
```

And to get the number of days that books were allowed to be borrowed we could say:

```
SELECT book_id, DATEDIFF(due_date, borrow_date) AS allowed_borrow_duration  
FROM BorrowedBooks  
WHERE due_date < CURRENT_DATE
```

Here we use the `DATEDIFF` function. This will come up with the number of days between two dates. We show the result in the output with the name `allowed_borrow_duration`. These functions and operators we just covered are enabling us to answer yet more complicated question with the use of SQL and a dataset. That's it for this introductory chapter!

Summary

You've made your way through the first chapter on SQL, well done! We introduced the fundamental concepts of SQL and its role in data analysis. We began by exploring the history and importance of SQL, followed by the difference between relational and non-relational databases. Then we saw an overview of different database management systems, such as MySQL, PostgreSQL, SQL Server, Oracle, and SQLite. We then delved into essential SQL terminologies, such as queries, statements, clauses, keywords, and views. We also provided guidance on setting up your environment, choosing a suitable database management system, and creating a sample database. Next, we covered basic SQL query writing, including the use of `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements. We also explained how to filter data using various clauses like `WHERE`, `ORDER BY`, `DISTINCT`, and `LIMIT`. Lastly, we discussed the use of operators and functions in SQL, including comparison operators, logical operators (`AND`, `OR`), the `LIKE` operator, arithmetic operators, and various functions for calculations, text manipulation, and date manipulation. At this point, you should have a solid understanding of the basic SQL concepts and be able to write simple queries to interact with databases. In the next chapters, we will expand on these concepts and explore how to join tables, work with groupings, and create more complex queries for data analysis.

3 Joining Tables in SQL

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



After learning about the SQL basics and understanding how databases work, it's time to dive a little deeper. We'll discuss table relationships and joins. Joining tables is typically considered a tough topic, but it's one that you'll really need to become comfortable with to be a data analyst. Don't worry, we'll build it up carefully. The topics we'll cover in this chapter are:

- Table relationships and why they're important
- Types of table relationships
- How to implement these relationships in SQL
- What joins are and why we need them
- Different types of joins
- How to write join queries

So please *join* me on this journey to learn about table joins. We'll continue to work with our library example.

Table relations

Imagine we tried to put all the data in our library database into one big table. What would happen? Well, we would quickly find that we're repeating a lot of the information. For example, if a book gets borrowed by multiple

members, we'd have to repeat its details each time it is borrowed. This repetition is not efficient to say the least. Because we have repetition, the data becomes harder to maintain, and it increases the likelihood of errors. Imagine that a detail of the member changes, we'd have to change it in all the rows where the member borrowed a book. Chances are that we forget to change it at some spots, and we'll lose data integrity as a result. This is why we separate our data into multiple tables, and to make this practical, we establish relationships between these tables. Let's revisit our library database, which is already divided into tables as per the previous chapter: `authors`, `books`, `borrowed_books`, and `members`. We have the following relationships between these tables:

- Each book is written by one author, and an author can write multiple books. This is a **one-to-many** relationship between the `authors` and `books` tables.
- Similarly, we can reverse the relationship between `authors` and `book` and call it a **many-to-one** relationship.
- Each member can borrow multiple books, and each book can be borrowed by multiple members. This is a **many-to-many** relationship between the `members` and the `books` tables. This relationship is represented using the `borrowed_books` table.
- There's also the concept of a **one-to-one** relationship that we don't have in our database. However, let's suppose we have a table for `rare_books` in the library, where each rare book has a unique insurance policy. This forms a one-to-one relationship between the `rare_books` and `insurance_policies` tables. One rare book only has one insurance policy. And one insurance policy is only for one rare book.

Implementing Relationships in SQL

Remember how we spoke about **primary** and **foreign keys** in the previous chapter? Well, these keys are central to establishing relationships in SQL. Just a quick recap. A **primary key** is a unique identifier for individual records in a table. Each record in the same table will have a unique primary key. A **foreign key** in one table is a field containing another (hence the name 'foreign') table's primary key. This is how we link records between tables. Let's illustrate this with an example. Here is a snippet from our `authors`

table:

id	name	country	birth_year	email
1	Mei Lin	China	1978	mei.lin@example.com
...
6	Emeka Okafor	Nigeria	1982	emeka.okafor@example.com

And here are the first two entries of our books table:

i	title	author_id	isbn	publication_year	publisher
1	Data Analytics for Beginners	6	978-1111111111	2020	DataPress Publishing
2	The Art of Data Analysis	1	978-2222222222	2019	Analytical Press

In the `books` table, `author_id` is a foreign key that links each book to an author in the `authors` table. The value of `author_id` in the `books` table matches the value of `id` (the primary key) in the `authors` table.

SQL Joins

So, when we want to retrieve data from multiple related tables, we perform a **join**. A join combines records from two or more tables based on a related column between them. This allows you to create a new result set that includes data from multiple tables. Here are some examples of questions that can be answered with the use of SQL joins:

- Which books have been borrowed by Member A?
- What books has Author B written that have been borrowed?
- What are the names of members who have borrowed Book C?
- List the names of all authors and their books that are currently in the library.
- Who is the author of the book that Member D currently has borrowed?
- List the members who have borrowed books written by Author E.
- Which books written by Author F have never been borrowed?

- Who borrowed the most recent book, and what was the title of the book?
- What is the complete list of books that a certain member has ever borrowed?
- Which authors' books have never been borrowed?

There are several types of joins. We'll discuss the most common ones here, starting with inner join. The joins that we will discuss deal with NULL values differently. We're going to explain the differences by using the following data:

	id	name	country	birth_year	email
1	Emeka Okafor	Nigeria	1982		emeka.okafor@example.com
2	Mei Lin	China	1978		mei.lin@example.com
3	Sophia Martin	USA	1985		sophia.martin@example.com
4	Juan Rodriguez	Spain	1970		juan.rodriguez@example.com
5	Aya Morimoto	Japan	1990		aya.morimoto@example.com

And here are the first five entries of our books table:

i	title	author	isbn	publication	publisher
d		_id		_year	
1	Data Analytics for Beginners	1	978-1111111111	2020	DataPress Publishing
2	The Art of Data Analysis	2	978-2222222222	2019	Analytical Press
3	Mastering Python	1	978-3333333333	2021	Python Publishers
4	SQL for Data Science	3	978-4444444444	2022	Science Publishing House
5	The Basics of Quantum Computing	NULL	978-5555555555	2023	Quantum Press

Okay, so this is as good an opportunity as any to mention a few key things you must know about SQL and deleting unwanted data from tables or **drop** tables. There are two statements we will briefly discuss and which you will use to modify your tables with the demands of this chapter.

DROP TABLE Statement

The **DROP TABLE** statement is used in SQL to remove an entire table from a database. It permanently deletes the table and all of its data, indexes, triggers, and other associated objects. The `DROP TABLE` statement is primarily used when you no longer need a table and want to remove it from the database altogether. The basic syntax of the `DROP TABLE` statement is as follows:

```
DROP TABLE table_name;
```

Let's apply this statement to our existing schema:

```
DROP TABLE authors;
DROP TABLE BorrowedBooks;
```

Unlike the `DROP TABLE` statement, which removes the entire table, the `DELETE` statement focuses on deleting specific data within the table.

Other details needed for this chapter

If you recall from the last chapter, not specifying any conditions and not using a `WHERE` clause leads to deleting all records in that table. Below is an example where the statements will clear all data from the tables:

```
DELETE FROM books;
DELETE FROM members;
```

After that, let's re-create the tables we dropped above.

```
-- Creating the Authors table
CREATE TABLE authors (
```

```

    id INT PRIMARY KEY,
    name VARCHAR (255) NOT NULL,
    country VARCHAR (255),
    birth_year INT,
    email VARCHAR (255)
);
-- Creating the Borrowed Books table
CREATE TABLE borrowed_books (
    id INT AUTO_INCREMENT PRIMARY KEY,
    book_id INT,
    member_id INT,
    borrow_date DATE,
    due_date DATE,
    return_date DATE,
    FOREIGN KEY (book_id) REFERENCES Books(id),
    FOREIGN KEY (member_id) REFERENCES Members(id)
)

```

Next, let's focus on populating all the tables we deleted records from or just created.

Comments in SQL

With `--` we can add a comment to SQL. This will ignore the text that comes after. We use it in these examples to indicate what the SQL snippet is doing.

```

-- Populating the Authors table
INSERT INTO authors VALUES (1, 'Emeka Okafor', 'Nigeria', 1982,
'emeka.okafor@example.com'),
(2, 'Mei Lin', 'China', 1978, 'mei.lin@example.com'),
(3, 'Sophia Martin', 'USA', 1985, 'sophia.martin@example.com'),
(4, 'Juan Rodriguez', 'Spain', 1970, 'juan.rodriguez@example.com'),
(5, 'Aya Morimoto', 'Japan', 1990, 'aya.morimoto@example.com'),
(6, "Maria Low", "Germany", 1981, "maria.low@example.com");
-- Re-Populating the Books table
INSERT INTO books VALUES (1, 'Data Analytics for Beginners', 1,
'978-1111111111', 2020, 'DataPress Publishing'),
(2, 'The Art of Data Analysis', 2, '978-2222222222', 2019, 'Analytical Press'),
(3, 'Mastering Python', 1, '978-3333333333', 2021, 'Python Publishers'),
(4, 'SQL for Data Science', 3, '978-4444444444', 2022, 'Science Publishing House'),
(5, 'The Basics of Quantum Computing', NULL, '978-5555555555', 20

```

```

23, 'Quantum Press'),
(6, 'The Basics of Blockchain', 5, '978-666666666', 2022, 'Block
chain Press'),
(7, 'SQL for Data Analysis', NULL, '978-777777777', 2023, 'Data
Analysis Press'),
(8, NULL, 4, '978-888888888', 2019, 'AI Publishing');

-- Populating the Members table
INSERT INTO members (first_name, last_name, email, join_date) VA
LUES
    ('Alice', 'Johnson', 'alice.johnson@example.com', '2021-01-01'
),
    ('Bob', 'Smith', 'bob.smith@example.com', '2021-02-15'),
    ('Chiara', 'Rossi', 'chiara.rossi@example.com', '2021-05-10'),
    ('David', 'Gonzalez', 'david.gonzalez@example.com', '2021-07-2
0'),
    ('Eve', 'Garcia', 'eve.garcia@example.com', '2021-09-30'),
    ('Femi', 'Adeyemi', 'femi.adeyemi@example.com', '2021-11-10'),
    ('Grace', 'Kim', 'grace.kim@example.com', '2021-12-15'),
    ('Henrik', 'Jensen', 'henrik.jensen@example.com', '2022-03-05
),
    ('Ingrid', 'Pettersson', 'ingrid.pettersson@example.com', '202
2-04-20'),
    ('Jia', 'Wang', 'jia.wang@example.com', '2022-06-01');

-- Populating the Borrowed_Books table
INSERT INTO borrowed_books (id, book_id, member_id, borrow_date,
due_date) VALUES
(1, 1, 1, '2022-04-01', '2022-05-01'),
(2, 2, 2, '2022-04-05', '2022-05-05'),
(3, 3, 3, '2022-04-10', '2022-05-10'),
(4, 4, 4, '2022-04-15', '2022-05-15'),
(5, 5, 5, '2022-04-20', '2022-05-20'),
(6, 6, NULL, '2022-04-25', '2022-05-25'),
(7, NULL, 7, '2022-04-30', '2022-05-30'),
(8, 4, 8, '2022-05-02', '2022-06-02'),
(9, NULL, NULL, '2022-05-05', '2022-06-05'),
(10, 7, 10, '2022-05-08', '2022-06-08'),
(11, 3, 6, NULL, NULL, NULL);

```

Note that we did not specify the column names when inserting data in the authors and books table. Why? Because we're adding data to all columns in the order their columns are specified, so it makes no difference if you leave the columns unspecified. Similarly, we did specify the columns for the members table and left out the `id` column because that's set to `AUTO_INCREMENT`. Copy and paste each of these statements individually and execute them. Once you have successfully done everything above, we can

move to the next section and learn about the types of joins. You can verify you have the correct data if you perform a select all statement and you'll get the following output:

```
mysql> SELECT * FROM authors;
+----+-----+-----+-----+-----+
| id | name | country | birth_year | email |
+----+-----+-----+-----+-----+
| 1 | Emeka Okafor | Nigeria | 1982 | emeka.okafor@example.com |
| 2 | Mei Lin | China | 1978 | mei.lin@example.com |
| 3 | Sophia Martin | USA | 1985 | sophia.martin@example.com |
| 4 | Juan Rodriguez | Spain | 1970 | juan.rodriguez@example.com |
| 5 | Aya Morimoto | Japan | 1990 | aya.morimoto@example.com |
| 6 | Maria Low | Germany | 1981 | maria.low@example.com |
+----+-----+-----+-----+-----+
6 rows in set (0.00 sec)

mysql> SELECT * FROM books;
+----+-----+-----+-----+-----+
| id | title | author_id | isbn | publication_year | publisher |
+----+-----+-----+-----+-----+
| 1 | Data Analytics for Beginners | 1 | 978-1111111111 | 2020 | DataPress Publishing |
| 2 | The Art of Data Analysis | 2 | 978-2222222222 | 2019 | Analytical Press |
| 3 | Mastering Python | 1 | 978-3333333333 | 2021 | Python Publishers |
| 4 | SQL for Data Science | 3 | 978-4444444444 | 2022 | Science Publishing House |
| 5 | The Basics of Quantum Computing | NULL | 978-5555555555 | 2023 | Quantum Press |
| 6 | The Basics of Blockchain | 5 | 978-6666666666 | 2022 | Blockchain Press |
| 7 | SQL for Data Analysis | NULL | 978-7777777777 | 2023 | Data Analysis Press |
| 8 | NULL | 4 | 978-8888888888 | 2019 | AI Publishing |
+----+-----+-----+-----+-----+
8 rows in set (0.00 sec)

mysql> SELECT * from members;
+----+-----+-----+-----+-----+
| id | first_name | last_name | join_date | email |
+----+-----+-----+-----+-----+
| 1 | Alice | Johnson | 2021-01-01 | alice.johnson@example.com |
| 2 | Bob | Smith | 2021-02-15 | bob.smith@example.com |
| 3 | Chiara | Rossi | 2021-05-10 | chiara.rossi@example.com |
| 4 | David | Gonzalez | 2021-07-20 | david.gonzalez@example.com |
| 5 | Eve | Garcia | 2021-09-30 | eve.garcia@example.com |
| 6 | Femi | Adeyemi | 2021-11-10 | femi.adeyemi@example.com |
| 7 | Grace | Kim | 2021-12-15 | grace.kim@example.com |
| 8 | Henrik | Jensen | 2022-03-05 | henrik.jensen@example.com |
| 9 | Ingrid | Pettersson | 2022-04-20 | ingrid.pettersson@example.com |
| 10 | Jia | Wang | 2022-06-01 | jia.wang@example.com |
+----+-----+-----+-----+-----+
10 rows in set (0.00 sec)
```

```
[mysql]> SELECT * FROM borrowed_books;
+----+-----+-----+-----+-----+-----+-----+-----+
| id | book_id | member_id | borrow_date | due_date | return_date | fine_amount |
+----+-----+-----+-----+-----+-----+-----+-----+
| 1 | 1 | 1 | 2022-04-01 | 2022-05-01 | NULL | NULL |
| 2 | 2 | 2 | 2022-04-05 | 2022-05-05 | 2022-10-11 | 1000 |
| 3 | 3 | 3 | 2022-04-10 | 2022-05-10 | NULL | NULL |
| 4 | 4 | 4 | 2022-04-15 | 2022-05-15 | NULL | NULL |
| 5 | 5 | 5 | 2022-04-20 | 2022-05-20 | NULL | NULL |
| 6 | 6 | NULL | 2022-04-25 | 2022-05-25 | NULL | NULL |
| 7 | NULL | 7 | 2022-04-30 | 2022-05-30 | NULL | NULL |
| 8 | 4 | 8 | 2022-05-02 | 2022-06-02 | 2022-12-15 | NULL |
| 9 | NULL | NULL | 2022-05-05 | 2022-06-05 | NULL | NULL |
| 10 | 7 | 10 | 2022-05-08 | 2022-06-08 | NULL | NULL |
| 11 | 3 | 6 | NULL | NULL | NULL | NULL |
+----+-----+-----+-----+-----+-----+-----+-----+
11 rows in set (0.00 sec)
```

Figure 3.0 – Select all statements and their output

Types of Joins in SQL

Several types of joins in SQL allow you to combine data from multiple tables based on specified conditions:

1. INNER JOIN
2. LEFT JOIN (or LEFT OUTER JOIN)
3. RIGHT JOIN (or RIGHT OUTER JOIN)
4. FULL JOIN (or FULL OUTER JOIN)
5. CROSS JOIN (or Cartesian Join)
6. SELF JOIN

Now you are ready to learn about each of these. We will begin exploring inner Join first.

Inner Join

An **inner join** retrieves data that exists in both tables being joined. It returns records that have matching values in both tables. Let's see a practical example. If you want to find all members who have borrowed a book, you would use an inner join.

```
SELECT members.first_name, members.last_name, borrowed_books.boo
```

```

k_id
FROM borrowed_books
INNER JOIN members ON borrowed_books.member_id = members.id;

```

This query will return the names of the members who have borrowed a book and the ID of the book they borrowed. Here's how the query works:

- It starts from the `borrowed_books` table which records all book borrowings.
- For each record in `borrowed_books`, it matches the `member_id` with the `id` from the `members` table.
- It then retrieves the `first_name` and `last_name` from the `members` table, along with the `book_id` from the `borrowed_books` table.

We can make a slightly more complex example; if you want to find all books that have been borrowed and by whom, you will use a double inner join. Here's an example using the library database (schema):

```

SELECT books.title, members.first_name, members.last_name
FROM borrowed_books AS bb
INNER JOIN books ON bb.book_id = books.id
INNER JOIN members ON bb.member_id = members.id;

```

This query will return the title of each borrowed book and the name of the member who borrowed it.

title	first_name	last_name
Data Analytics for Beginners	Alice	Johnson
The Art of Data Analysis	Bob	Smith
Mastering Python	Chiara	Rossi
SQL for Data Science	David	Gonzalez
SQL for Data Science	Henrik	Jensen
The Basics of Quantum Computing	Eve	Garcia
SQL for Data Analysis	Jia	Wang

7 rows in set (0.01 sec)

Figure 3.1 - Results of Inner Join Query

This is how the query works internally:

- It joins three tables (`borrowed_books`, `books`, and `members`) and retrieves the book's title from the `books` table and the `first_name` and `last_name` of the member from the `members` table.
- For each record in `borrowed_books`, it matches the `book_id` with the `id` from the `books` table and `member_id` with the `id` from the `members` table.

As seen above in *Figure 3.1*, the borrowed books are returned with the first and last names of the members who borrowed them.

Left Join

A **left join** retrieves all the records from the left table and any matching records from the right table. If there is no match, the result is `NULL` on the right side. For example, you may want to find all authors and any books they've written that are in the library, whether the books have been borrowed or not.

```
SELECT authors.name, books.title
FROM authors
LEFT JOIN books ON authors.id = books.author_id;
```

This query will return a list of all authors and the titles of any books they've written. If an author hasn't written any books in the library, the `books.title` field for that author will be `NULL`.

name	title
Emeka Okafor	Data Analytics for Beginners
Emeka Okafor	Mastering Python
Mei Lin	The Art of Data Analysis
Sophia Martin	SQL for Data Science
Juan Rodriguez	NULL
Aya Morimoto	The Basics of Blockchain
Maria Low	NULL

7 rows in set (0.01 sec)

Figure 3.2 Results of Inner Join Query

In this case, the result shows that `Emeka Okafor` is the author of two books while `Juan Rodriguez` is the author of a book without a title, hence the `NULL` value in the `title` column. Moreover, `Maria Low` is not the author of any books in our library database and that's why there's a `NULL` value for her. This query joins two tables (`books`, `authors`) and retrieves the author's name from the `authors` table and the book's title from the `books` table.

Right Join

A **right join** retrieves all the records from the right table and the matching records from the left. The result is `NULL` on the left side if there is no match. If a value in the left table doesn't occur in the right table, this value is left out (pun intended). When might such a join be useful? For instance, when we want a list of all books and who wrote them (if anyone).

```
SELECT authors.name, books.title
FROM authors
RIGHT JOIN books
ON authors.id = books.author_id;
```

name	title
Emeka Okafor	Data Analytics for Beginners
Mei Lin	The Art of Data Analysis
Emeka Okafor	Mastering Python
Sophia Martin	SQL for Data Science
NULL	The Basics of Quantum Computing
Aya Morimoto	The Basics of Blockchain
NULL	SQL for Data Analysis
Juan Rodriguez	NULL

8 rows in set (0.00 sec)

Figure 3.3 - Results of Right Join Query

All books would be listed here, "The Basics of Quantum Computing", appearing with a `NULL` author name, indicating that the book has no associated author in the `authors` table. The author Juan Rodriguez has written a book, but this book has a `NULL` value for the title. He was included because he was mentioned as a foreign id in the book table. Maria Low doesn't have a book, and that's why her name doesn't occur when we perform a RIGHT JOIN. Here's another example. Let's say you want to find all members and any books they've borrowed:

```
SELECT members.first_name, members.last_name, books.title
FROM members
RIGHT JOIN borrowed_books ON members.id = borrowed_books.member_id
JOIN books ON borrowed_books.book_id = books.id;
```

This query will return all members, even those who have yet to borrow any books. The filtering criteria here is that all records without `NULL` in `book_id` and `member_id` will be shown.

first_name	last_name	title
Alice	Johnson	Data Analytics for Beginners
Bob	Smith	The Art of Data Analysis
Chiara	Rossi	Mastering Python
Femi	Adeyemi	Mastering Python
David	Gonzalez	SQL for Data Science
Henrik	Jensen	SQL for Data Science
Eve	Garcia	The Basics of Quantum Computing
NULL	NULL	The Basics of Blockchain
Jia	Wang	SQL for Data Analysis

9 rows in set (0.00 sec)

Figure 3.4 - Results of Right Join Query

Full Join

A **full join**, also known as **full outer join**, retrieves all records where there is a match in either the left or the right table. If there is no match, the result is `NULL` on either side. Here's an example where we want to find all books and authors, whether the book has an author in the authors table or not, and whether the author has a book in the books table or not:

```
SELECT authors.name, books.title
FROM authors
FULL JOIN books
ON authors.id = books.author_id;
```

However, this gives an error because MySQL doesn't support the `FULL JOIN` keyword (other flavors of SQL like PostgreSQL do support this keyword, so this query would work in that instance). Hence, we will have to try another approach. We will have to "simulate" the results of a `FULL JOIN`.

```
SELECT a.name, b.title
FROM authors a
LEFT JOIN books b ON a.id = b.author_id
UNION
SELECT a.name, b.title
FROM authors a
```

```
RIGHT JOIN books b ON a.id = b.author_id  
WHERE a.id IS NULL;
```

name	title
Emeka Okafor	Data Analytics for Beginners
Emeka Okafor	Mastering Python
Mei Lin	The Art of Data Analysis
Sophia Martin	SQL for Data Science
Juan Rodriguez	NULL
Aya Morimoto	The Basics of Blockchain
Maria Low	NULL
NULL	The Basics of Quantum Computing
NULL	SQL for Data Analysis

9 rows in set (0.01 sec)

Figure 3.5 - Results of "Full" Join Query

Let's summarize what happens in this "simulated" FULL JOIN query. Basically, this query first performs a LEFT JOIN to fetch all matching records from the authors and books tables. After that, it performs a RIGHT JOIN to include the unmatched records from the books table, giving the semblance of a "full" join. The condition specified in the WHERE clause ensures that only the unmatched records from the books table are included because the matched ones are already retrieved; we don't want duplicate records to be displayed. At this point, a valid question would be, what is the UNION keyword? Let's understand that. In simple terms, the UNION operator combines the results of the LEFT JOIN and RIGHT JOIN queries into a single result set. The result of the LEFT JOIN, including the matched records from the authors and books table, is merged with the unmatched records from the books table resulting from the RIGHT JOIN. Phew! With that out of the way, we can discuss the output of this query now. In this case, all authors and all books are listed. Juan Rodriguez appears with a NULL book title because he is the author of a book whose title is NULL, while Maria Low is not the author of any book in the library database, and "The Basics of Quantum Computing" and "SQL for Data Analysis" appear with a NULL author name.

because their `author_id` values are `NULL` in the `books` table. What's important to note here is that these results still do show, unlike in `RIGHT JOIN`, where the unmatched results from the `authors` table aren't retrieved. Remember, you must practice these joins yourself if you want to learn how they work.

Cross Join

The **CROSS JOIN** or **Cartesian join** basically results in each row of the first table being matched with every row in the second table (known as the **cartesian product** in mathematics). This type of join is useful for retrieving all possible table record combinations. Here's an example to better understand this:

```
SELECT a.name, b.title  
FROM authors AS a  
CROSS JOIN books AS b;
```

As seen below in *Figure 3.6*, the result contains 48 rows of all possible combinations for the author names and book titles.

name	title
Maria Low	Data Analytics for Beginners
Aya Morimoto	Data Analytics for Beginners
Juan Rodriguez	Data Analytics for Beginners
Sophia Martin	Data Analytics for Beginners
Mei Lin	Data Analytics for Beginners
Emeka Okafor	Data Analytics for Beginners
Maria Low	The Art of Data Analysis
Aya Morimoto	The Art of Data Analysis
Juan Rodriguez	The Art of Data Analysis
Sophia Martin	The Art of Data Analysis
Mei Lin	The Art of Data Analysis
Emeka Okafor	The Art of Data Analysis
Maria Low	Mastering Python
Aya Morimoto	Mastering Python
Juan Rodriguez	Mastering Python
Sophia Martin	Mastering Python
Mei Lin	Mastering Python
Emeka Okafor	Mastering Python
Maria Low	SQL for Data Science
Aya Morimoto	SQL for Data Science
Juan Rodriguez	SQL for Data Science
Sophia Martin	SQL for Data Science
Mei Lin	SQL for Data Science
Emeka Okafor	SQL for Data Science
Maria Low	The Basics of Quantum Computing
Aya Morimoto	The Basics of Quantum Computing
Juan Rodriguez	The Basics of Quantum Computing
Sophia Martin	The Basics of Quantum Computing
Mei Lin	The Basics of Quantum Computing
Emeka Okafor	The Basics of Quantum Computing
Maria Low	The Basics of Blockchain
Aya Morimoto	The Basics of Blockchain
Juan Rodriguez	The Basics of Blockchain
Sophia Martin	The Basics of Blockchain
Mei Lin	The Basics of Blockchain
Emeka Okafor	The Basics of Blockchain
Maria Low	SQL for Data Analysis
Aya Morimoto	SQL for Data Analysis
Juan Rodriguez	SQL for Data Analysis
Sophia Martin	SQL for Data Analysis
Mei Lin	SQL for Data Analysis
Emeka Okafor	SQL for Data Analysis
Maria Low	NULL
Aya Morimoto	NULL
Juan Rodriguez	NULL
Sophia Martin	NULL
Mei Lin	NULL
Emeka Okafor	NULL

48 rows in set (0.01 sec)

Figure 3.6 - Results of Cross Join Query

Next, let's explore the **SELF JOIN**.

Self Join

As evident by its name, **SELF JOIN** is used to join a table to itself based on a condition. For example, an employee's table would contain a column to specify whether a specific employee is a manager. Here's the query:

```
SELECT e.employee_name, m.employee_name AS manager_name  
FROM employees e  
JOIN employees m ON e.manager_id = m.employee_id;
```

Note that the same table has two aliases, "e" and "m", and is treated as two individual tables joined on the `manager_id` column of the table. The output returns an employee's name and the respective manager's name. If there is a `NULL` value in the manager's name column, it indicates that the employee is a manager. Congratulations! You have learned all about the types of joins in SQL, let's learn about the best practices and performance tips to optimize your SQL query results now.

Best Practices for Using JOIN in SQL

Joins are an essential part of SQL regarding relations between databases. They allow us to access data from across multiple tables easily. However, note that using joins is only sometimes preferred because they are the most expensive operation in SQL (not in terms of money, but processing and the time that takes). Care must be taken to use them sparingly and only when necessary. Hence, let's look at a few practices you should follow on your journey to mastering SQL.

Use Explicit Join Syntax

There are multiple ways to write joins, but using the explicit JOIN syntax, rather than just separating your tables with commas, leads to clearer and maintainable code. Another benefit of explicit JOIN syntax is separating your

join logic from your filtering logic. Consider this example from our library database:

```
-- Implicit JOIN Syntax
SELECT members.name, books.title
FROM members, borrowed_books, books
WHERE members.id = borrowed_books.member_id AND books.id = borrowed_books.book_id;
```

And then this one where we use the explicit syntax:

```
-- Explicit JOIN Syntax
SELECT members.name, books.title
FROM members
JOIN borrowed_books ON members.id = borrowed_books.member_id
JOIN books ON books.id = borrowed_books.book_id;
```

The explicit JOIN syntax is easier to read and separates our join conditions from any other filter conditions we might have.

Be Mindful of your Join Conditions

Being mindful of your join conditions is crucial to obtaining the correct data. Careless join conditions can result in unintended cross-joins (cartesian products) or just outright incorrect data. Consider these two join conditions in our library database:

```
-- Incorrect JOIN condition, resulting in a Cartesian product
SELECT books.title, authors.name
FROM books
JOIN authors ON books.id = authors.id;
```

And this one where we do it right:

```
-- Correct JOIN condition
SELECT books.title, authors.name
FROM books
JOIN authors ON books.author_id = authors.id;
```

The first query will join books and authors whose ids match, which isn't correct in our data model. The second query correctly joins books and authors using the author_id foreign key in the books table, which is the

correct way.

Order Matters in Left and Right Joins

Remember that in a `LEFT JOIN`, all rows from the first (left) table will be included in the result set, even if there's no matching record in the second (right) table. Conversely, in a `RIGHT JOIN`, all rows from the second (right) table will be included, even if there's no matching record in the first (left) table. For instance, if you want to find all authors and their related books, you might use a `LEFT JOIN`, with `authors` as the first table. This would ensure that authors who have yet to write any books are included in the result. Using a second table in the same statement and calling `LEFT JOIN` will still refer to the first table mentioned.

Using Aliases in Join Statements

Aliases help make the SQL query easier to read and help the code look cleaner. As an additional tip, you only have to type that alias in your query and not the full name of the table repeatedly. This saves time and helps prevent errors when a table name is complex. Here's the same example as we used in `RIGHT JOIN` above:

```
SELECT a.name, b.title
FROM authors AS a
RIGHT JOIN books AS b
ON a.id = b.author_id;
```

Summary

Great job on completing this chapter! This chapter was dedicated to deepening your understanding of SQL by introducing the concept of joining tables. Joining tables in SQL is crucial for fetching data spread across different tables and stitching them together logically. This way, we can perform a more complex and insightful analysis. We started this chapter by emphasizing the importance and purpose of joins in SQL. From there, we took a comprehensive dive into various types of joins: `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `OUTER JOIN`, `CROSS JOIN`, and `SELF JOIN`, each

with its unique application in data analysis. In each section, we broke down the structure and syntax of the join types, followed by real-world examples. This approach aims to illustrate how each join works and provides you with ample practice to cement your understanding. We clarified how `INNER JOIN` is used to fetch common records from multiple tables while `LEFT JOIN`, `RIGHT JOIN`, and `OUTER JOIN` help retrieve expected and exclusive records from either or all tables. We also discussed the less commonly used `SELF JOIN` and `CROSS JOIN` and demonstrated their usage in specific scenarios. The chapter wrapped up with best practices for using joins to ensure that you write efficient and clean SQL join queries. At this point, you should feel comfortable using various SQL joins to combine data from different tables effectively. It must have been tough! But remember, mastering SQL joins is a significant step towards becoming a proficient data analyst. In the upcoming chapters, we'll learn about creating business metrics with aggregations, advanced SQL techniques, and much more.

4 Creating Business Metrics with Aggregations

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



Now that you know joins, you are ready for the next step in your journey to become a data analyst. We have arrived at aggregations and their significance in creating business metrics. With our current SQL knowledge, we are ready to dig deeper into data and use this data to gain actionable insights. This chapter will teach you about aggregations and how to use them to generate meaningful business metrics. This chapter will arm you with the knowledge and practical skills to use aggregations effectively in SQL, enabling you to extract valuable business metrics from large datasets. Here's the overview of what we're going to cover:

- Why we need aggregations in business metrics
- Important aggregate functions in SQL
- Combining aggregation functions with WHERE and GROUP BY
- Interacting with the HAVING clause and its comparison with WHERE
- Exploring the concept of nested aggregations
- Illustrating best practices and tips for effective use of aggregations

Let's level up our SQL game and dive into SQL aggregations and re their role in shaping business metrics!

Aggregations in Business Metrics

In the field of business analysis, data is the key to gaining valuable insights. However, the sheer volume of data can sometimes be overwhelming. This is where aggregations come into play. They can be used for summarizing large datasets into digestible metrics, thus enabling us to make data-driven strategic decisions. Imagine our library database - it's rich (or at least could be, right?) with data about books, authors, members, and borrowed books. While this data is valuable in its raw form, it gains greater significance when we perform aggregations. For example, we can aggregate data to determine the total number of books in the library, find the average number of books borrowed by the members, or calculate the maximum and minimum number of times a book has been borrowed. In short, aggregations help us convert raw data into meaningful metrics that answer specific business questions. Now that we have a basic understanding of the role of aggregations, let's explore some essential aggregation functions in SQL.

Aggregations in SQL to Analyze Data

Aggregations in SQL are operations that allow us to perform calculations across sets of rows and condense a large volume of data into a single value, such as a sum, average, maximum, or minimum of something. We need this when we're dealing with complex analytical problems, like finding the average borrowing frequency of books, the most popular author in our library, or the highest count of a specific book borrowed.

Using COUNT in SQL

The **COUNT** function in SQL provides the number of rows matching a specified condition. It is a popular function in data analysis when we need to know the number of records that meet specific criteria. The basic syntax of the **COUNT** function is as follows:

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

Let's use an example from our library database to demonstrate how the **COUNT** function is applied. Suppose we want to know how many books are

currently borrowed. We'd use the `COUNT` function on the borrowed books table for this.

```
SELECT COUNT(*) AS "Currently Borrowed"  
FROM borrowed_books  
WHERE return_date IS NULL;
```

In *Figure 4.1*, we can see that the result is a single aggregate value. The result is 10 because we set `NULL` for all 10 `return_date` records. The result informs us that 10 books are currently borrowed.

Currently Borrowed	
	10
1 row in set (0.00 sec)	

Figure 4.1 - Results of the COUNT function to find the number of borrowed books

Similarly, we can use `COUNT` to identify how many members are registered in the library:

```
SELECT COUNT(*) AS "TOTAL NUMBER OF MEMBERS"  
FROM members;
```

TOTAL NUMBER OF MEMBERS
10

1 row in set (0.01 sec)

Figure 4.2 - Results of the COUNT function to find the total number of members

With the COUNT function, we've added the ability to quantify data sets based on specific conditions to our SQL toolbox. Now, let's transition to a function that goes beyond counting – the SUM function.

Using the SUM Function in SQL

The **SUM** function in SQL allows us to add all the values in a specific column. Its syntax is similar to that of COUNT :

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

Imagine if we have a fine_amount column in the borrowed_books table, representing the amount to be paid for late returns. If you want to practice along, use the ALTER TABLE statement to add a new column called fine_amount , and then you can use the UPDATE statement to add data in that column for the rows you want. This would be the query to alter the table:

```
ALTER TABLE borrowed_books ADD COLUMN fine_amount INT;
```

After that, here's the update statement to add a fine to a specific row:

```
UPDATE borrowed_books
SET fine_amount = 1000
WHERE id=2;
```

```
UPDATE borrowed_books  
SET return_date= '2022-10-11'  
WHERE id=2;
```

If we want to calculate the total amount of fines collected, we can use the `SUM` function as follows:

```
SELECT SUM(fine_amount)  
FROM borrowed_books  
WHERE return_date IS NOT NULL;
```

SUM(fine_amount)
1000

1 row in set (0.00 sec)

Figure 4.3 - Results of the `SUM` function to find the total amount of fines collected

This query summarizes the `fine_amount` for all records where `return_date` is not `NULL` (meaning the book has been returned) and a fine is specified. Note that if a return date is not `NULL`, but the `fine_amount` is set to `NULL` for that row, then it won't affect the result. By utilizing the `SUM` function, we can quickly perform calculations that would be time-consuming to do manually. It might be obvious but let's make sure this is clear: `SUM` works on numerical data types only. Now, let's move on to a function that performs calculations and provides insights about the data distribution. I'm talking about the `AVG` function.

Using AVG in SQL

The **AVG** function is used in SQL to find the average value of a numeric column. The syntax for the **AVG** function is also similar to the above aggregate functions:

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

Suppose we want to calculate the average birth years of the authors in our library to gain an estimate of their ages. We can use the **AVG** function as follows:

```
SELECT AVG(birth_year)
FROM authors
WHERE birth_year IS NOT NULL;
```

avg(birth_year)
1981.0000

1 row in set (0.01 sec)

Figure 4.4 - Results of the **AVG** function to find the average year of birth

This query calculates the average `birth_year` for all records where `birth_year` is not `NULL`. It is worth observing that the average value is in decimal. Using the **AVG** function, we can understand the central tendency of our data, which can provide valuable insights in various scenarios. After dealing with **AVG**, let's look at how we can identify the smallest and largest values in our data set using the **MIN** and **MAX** functions.

Using MIN and MAX Functions in SQL

The **MIN** and **MAX** functions in SQL return the smallest and largest values in a column, respectively. These functions are beneficial for identifying the range of our data. Here is the basic syntax for the **MIN** and **MAX** functions:

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

Let's illustrate their use with our library database. We will use the `join_date` column in the `members` table, showing each member's joining date. If we want to find out the shortest and longest period the members have been visiting our library, we can use the **MIN** and **MAX** functions as follows:

```
SELECT MIN(join_date) FROM members;
SELECT MAX(join_date) FROM members;
```

min(join_date)
2021-01-01
1 row in set (0.00 sec)

Figure 4.5 - Results of the **MIN** function on the `join_date` column

The first query will return the minimum join date. This means the join date for the oldest member. The result is displayed in *Figure 4.5*.

```
+-----+
| max(join_date) |
+-----+
| 2022-06-01      |
+-----+
1 row in set (0.01 sec)
```

Figure 4.6 - Results of the MAX function on the join_date column

The second query will provide the latest `join_date`. This is the join date for the most recent member. The result is displayed in *Figure 4.6*. By using the `MIN` and `MAX` functions, we can get a sense of the boundaries or extremities in our data. This informs us about our dataset's overall spread of values. Having covered individual row calculations, you might wonder how we can perform these aggregations over data groups rather than the entire dataset. For instance, what if we wanted to know each member's total fines or average loan duration? This is where the power of the `GROUP BY` clause in SQL comes into play.

GROUP BY Clause

The **GROUP BY** clause is used in SQL to group rows with the same values in specified columns into aggregated data, which can be helpful when we want to perform computations on categorized data. Here's a basic syntax of `GROUP BY`:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s);
```

When you group results by a specific column, it essentially means that the output will be arranged so that rows with identical values in the specified column will be clustered together. Let's use the library database again to illustrate this. If we wanted to find out how many books each member has borrowed, we could group the results by the id of the member in the borrowed_books table:

```
SELECT member_id, COUNT(book_id)
FROM borrowed_books
GROUP BY member_id;
```

member_id	COUNT(book_id)
NULL	1
1	1
2	1
3	1
4	1
5	1
7	0
8	1
10	1

9 rows in set (0.03 sec)

Figure 4.9 - Results of the GROUP BY clause, grouped by the member id.

This SQL query will return a list of member ids along with the count of books each member has borrowed (*Figure 4.9*). The GROUP BY clause is something you'll need a lot because it allows us to analyze our data at more granular levels. It's like zooming in from a high-level overview to the details

of each category.

Aggregations with GROUP BY

We can take the `GROUP BY` clause a step further by using it in conjunction with various aggregate functions like `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, and others. This allows us to perform specific calculations on each group of data. Please note that any column specified in the `SELECT` clause must be used to group by or an error will occur. For example, if we wanted to calculate the total fines incurred by each member, we could use the `SUM` function with `GROUP BY` as follows:

```
SELECT member_id, SUM(fine_amount)
FROM borrowed_books
GROUP BY member_id;
```

member_id	SUM(fine_amount)
NULL	NULL
1	NULL
2	1000
3	NULL
4	NULL
5	NULL
7	NULL
8	NULL
10	NULL

9 rows in set (0.00 sec)

Figure 4.10 - Results of the Group By clause, grouped by the member id to find the total amount of fines collected.

This query will return each member's id and the total fines they've incurred, as displayed in *Figure 4.10*. Using aggregate functions with `GROUP BY` allows us to perform more specific and detailed analysis, which could reveal meaningful patterns and insights within our data. While `GROUP BY` allows us to segment our data for analysis, we may want to filter the results of our groupings. SQL achieves this through the `HAVING` clause, which we will discuss in the next section. Just as the `WHERE` clause is used to filter rows, the `HAVING` clause is used to filter groups. This is especially useful when finding groups that match certain conditions.

HAVING clause

The **HAVING** clause was added to SQL because the **WHERE** keyword could not be used with aggregate functions. **HAVING** is typically used with the **GROUP BY** clause to filter the group's results by operation. Here's the basic syntax of **HAVING**:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition;
```

The key difference between **HAVING** and **WHERE** is that **WHERE** filters individual rows before the aggregation operation while **HAVING** filters the results after the **GROUP BY** operation. This is why the order of these clauses also matters. In other words, **WHERE** applies conditions on individual rows before they're even grouped, whereas **HAVING** filters groups after aggregate functions form them. For example, using our library database, let's say we want to find out which members have borrowed one or more books. We could use the **HAVING** clause as follows:

```
SELECT member_id, COUNT(book_id) as "Number of Borrowed Books"
FROM borrowed_books
GROUP BY member_id
HAVING COUNT(book_id) >= 1;
```

member_id	Number of Borrowed Books
NULL	1
1	1
2	1
3	1
4	1
5	1
8	1
10	1

8 rows in set (0.01 sec)

Figure 4.11 - Results of the HAVING clause to find which members have borrowed one or more books.

This query will return a list of members who have borrowed one or more books, as displayed in *Figure 4.11*.

Combining GROUP BY and HAVING with Aggregations

By combining `GROUP BY` , `HAVING` , and aggregate functions, you can perform complex data analysis tasks. For example, we could find the members who have incurred total fines greater than \$50:

```
SELECT member_id, SUM(fine_amount) as TotalFine
FROM borrowed_books
GROUP BY member_id
HAVING SUM(fine_amount) > 500;
```

member_id	TotalFine
2	1000

1 row in set (0.01 sec)

Figure 4.12 - Result of a complex query, with aggregate functions, group by and having clauses.

This will list all members who have incurred a fine over \$500. As you can see there is one match, as displayed in *Figure 4.12*. Here are a few general rules when it comes to `GROUP BY` and `HAVING` with aggregations:

- any column that appears in the `SELECT` clause must be either part of the `GROUP BY` clause or used in an aggregation function;
- you must use the `HAVING` clause to apply conditions to the groups themselves;
- the `GROUP BY` clause usually comes after the `WHERE` clause and before the `ORDER BY` clause.

Let's look at some best practices for aggregations before we wrap up this chapter.

Best Practices for Aggregations

There are some best practices that you should keep in mind when working with aggregations. These are not strict rules, but common best practices that apply to most situations.

Be Specific in Your Aggregations

It's important only to aggregate the data that you need for your analysis. Unnecessary aggregations not only complicate your query, but also impact performance. For instance, if we are interested in finding the average fine amount by member, we don't also need to calculate the sum and count unless these are directly related to our analysis. Here's how to do it properly:

```
SELECT member_id, AVG(fine_amount) as AverageFine  
FROM borrowed_books  
GROUP BY member_id;
```

member_id	AverageFine
NULL	NULL
1	NULL
2	1000.0000
3	NULL
4	NULL
5	NULL
7	NULL
8	NULL
10	NULL

9 rows in set (0.00 sec)

Figure 4.12 - Results of a specific aggregate query.

This query gives us precisely what we need without any superfluous calculations or extra processing.

Choose the Right Column for Aggregation

The column that you choose to aggregate can significantly impact the results of your analysis. Therefore, picking a column that makes sense in the context of your question or task is crucial. In our library database, what should we do if we want to find out how many books each member has borrowed?

Aggregating on the `member_id` column would make sense, as it provides a count per member:

```
SELECT member_id, COUNT(book_id) as BooksBorrowed  
FROM borrowed_books  
GROUP BY member_id;
```

On the other hand, aggregating on `book_id` would not provide helpful information, as it would simply tell us how many times each individual book has been borrowed, not how many books each member has borrowed.

Consider Performance in SQL

Aggregations can be resource-intensive, especially on large datasets, so it's important to consider performance implications. One way to improve performance is to filter your data as much as possible before performing the aggregation using the `WHERE` clause. For example, if we only want to calculate the total fines for active members, we could filter our borrowed books table before aggregation (this assumes a column called `IsActive` exists in our table):

```
SELECT member_id, SUM(fine_amount) as TotalFine  
FROM borrowed_books  
WHERE IsActive = 1  
GROUP BY member_id;
```

This way, we aggregate only the relevant data, reducing the computational load.

Keep Readability in Mind

Clear, understandable SQL queries are essential for both your future self and others who might interact with your code. This includes using clear aliases for your aggregated columns and correctly formatting and indenting your SQL queries. For instance, `BooksBorrowed` is a much more meaningful name than `COUNT(book_id)` and would make it easier for others (and future you) to understand your query:

```
SELECT
    member_id,
    COUNT(book_id) as BooksBorrowed
FROM borrowed_books
GROUP BY member_id;
```

Aggregations are complex. Keeping these best practices in mind will help the readability and understandability of your analyses. And that's it for this chapter. Let's go over what we've learnt.

Summary

In the chapter "Creating Business Metrics with Aggregations", we learned to analyze data by using aggregations and `GROUP BY` and `HAVING`. We start the journey by emphasizing the importance of aggregations in analyzing data, explaining how they provide valuable insights by summarizing large volumes of data into meaningful metrics. Next, we deep dive into various aggregate functions, including `COUNT`, `SUM`, `AVG`, `MIN`, and `MAX`. Each function is introduced with its definition, syntax, and practical examples from our library database. We discuss how `COUNT` can help determine the total number of rows in a table, the number of non-`NULL` values in a given column, and how `SUM` can be used to total numerical data. We explore the `AVG` function to calculate average values and discuss how `MIN` and `MAX` can find the smallest and largest values. We then introduce the `GROUP BY` clause, which allows you to group results by a specific column and pair it with aggregate functions to derive meaningful summaries from the data. We distinguish between the roles of the `HAVING` and `WHERE` clauses, explaining that while `WHERE` filters records before grouping and aggregation, `HAVING` filters after. We continue to link theory with practice by providing use cases from our library database,

demonstrating how to combine `GROUP BY`, `HAVING`, and various aggregation functions to answer complex questions and filter aggregated results effectively. Lastly, we touch upon the best practices when using aggregations. We underscore the importance of selecting columns carefully when performing aggregations and provide guidelines to improve query performance and readability. The journey continues, so let's move on to even more exciting SQL concepts!

5 Advanced SQL

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



You have a solid SQL foundation, congrats! But that's not where we're going to stop. We're going to take your SQL expertise to the next level. You have the basic building blocks you need to create queries. This chapter will add even more building blocks for using SQL. This will enhance your data analysis capabilities and help you with transforming data into actionable insights. Here's what this chapter will cover:

- The concept of subqueries and their applications in complex queries
- The usage and benefits of Common Table Expressions
- The power of Window Functions in advanced data analysis
- Techniques for Date Time Manipulation in SQL
- Tools for Text Manipulation to work with string data

As you can tell, this chapter is called *Advanced SQL* for a reason. But fear not, we'll take it slow. And by the end of this chapter, you will have a solid understanding of these advanced SQL features and you'll be able to utilize them effectively in your daily data analysis tasks. So, let's get started with subqueries to kick off this chapter!

Working with subqueries

Subqueries are basically queries within SQL queries. Sometimes they are called **inner queries** or **nested queries**. Subqueries are commonly used to help you level up your data analysis game. They are essentially a query embedded within another SQL query. They can return data that will be used in the main query as a condition to restrict the data further to be retrieved. Subqueries can be used in `SELECT`, `WHERE`, and `FROM` clauses and can return a variety of formats. Just like normal queries, the result can be a single value, a single row, a single column, and a table. The subqueries are used to dynamically calculate a value to be used in the main query. This might sound vague, so let's see them for ourselves. The basic syntax of a subquery is as follows:

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator
(
    SELECT column_name
    FROM table_name
    WHERE condition
);
```

The first `SELECT` is the main query, the second `SELECT` between the parentheses is the subquery. Subqueries are always written inside a pair of parentheses. You might use subqueries when the result of the entire query depends on the unknown dynamic data. If a simple comparison can't solve these problems, you may need a subquery. You'll need to know how to navigate advanced SQL queries, because there are situations where you simply need them. However, subqueries come with a cost. They can be slower than regular queries because they require multiple passes over the data. The database has to execute the inner query before it can execute the outer query. This can be quite slow if the inner query returns a large amount of data. That's why we should not overuse subqueries and be mindful of their cost. You should always look for ways to rewrite your queries to avoid subqueries if possible. For example, in some cases, you can use a `JOIN` instead of a subquery, which is faster. We'll see a more concrete example soon, but let's first discuss the different types of subqueries in more detail.

Types of subqueries

Depending on the data they return, subqueries can be distinguished between four types:

- **Scalar subquery**: This type of subquery returns a single value. It's used where you need only one value. This can, for example, be the result of a `COUNT`.
- **Row subquery**: This type returns a single row of data. This is usually helpful when inserting values in a table when you must fetch the data from a temporary table.
- **Column subquery**: It produces a single column of data.
- **Table subquery**: This type of subquery returns an entire table. This is used when you want to create a temporary table for the runtime of a query—this subquery takes place after `FROM`.

Before we see a real example, let's make sure that you understand the concept by comparing the SQL subquery to a real-life subquery.

Non-code explanation of subquery

Let's use our library to understand and see how subqueries work in practice. We're going to first think about a non-analyst example to better understand the concept of subqueries. Imagine you want to find a specific book from our library database. When you're writing a simple query, think of it being directed to a "librarian". However, sometimes even librarians need a helper. You can think of this helper as the POS (point-of-sale) software. So, we query the librarian, and the librarian has to query the POS software. The result of what the POS software says, is what the librarian will use to give us our answer. The usage of the POS software can be considered the inner query, it is used to ensure the book is available. Based on the results, the librarian responds to our initial (or main) query.

Using a basic subquery on our library database

Alright, let's transfer back to our data analyst role and start with a basic example of a subquery in action. Suppose you're assigned to find the authors who have published a book after 2019. You can't find all this information in the books or authors tables.

```
SELECT name
FROM authors
WHERE id IN
(
    SELECT author_id
    FROM books
    WHERE publication_year > 2019
);
```

The following screenshot shows the results of this subquery. As you can see, there are three authors who have published books after 2019.

name
Emeka Okafor
Sophia Martin
Aya Morimoto

3 rows in set (0.01 sec)

Fig 5.1 - Results of the subquery which returns only the name of the authors

Note that the preceding query could also be run using an `INNER JOIN`. This would be the query in that case:

```
SELECT a.name
FROM authors a
JOIN books b ON b.author_id = a.id
WHERE publication_year > 2019;
```

The subquery approach returns three rows while `JOIN` will return four rows

because Emeka Okafor has published twice after 2019. That's why we get her name twice, and that's something we don't want. This can be fixed using a `DISTINCT` keyword before `a.name`, in which case, the `JOIN` query will perform better because it has to make only one pass through the data. Next, let's see some differences between using subquery and `JOIN` and when to choose which one.

Subquery vs joining tables

In many situations, both the subquery and the `JOIN` approach will lead to the same result. The approach you choose depends on several factors. The first factor that's often considered is performance. Joins can usually be more efficient (a.k.a better performing) if the dataset is small. The `JOIN` logic needs to pass through the data only once, while subqueries make multiple data passes and often involve implicit joins, which are worse in terms of performance costs. Next, readability is quite a thing. Subqueries improve the readability of SQL queries as they break a problem into smaller parts which are easier to read and understand. It's more the human way of solving the question. Lastly, we'll consider flexibility. Subqueries can be used in multiple clauses, such as the `SELECT`, `WHERE`, `HAVING`, or `FROM` clauses, while `JOIN` is only used with the `FROM` clause. There are some rules to how to use subqueries for each of these clauses though. Let's have a look at these rules.

Rules of subquery usage

In order to use subqueries, it's a must-have to be aware of the rules of subqueries:

- A subquery must be enclosed in parentheses.
- A subquery can return a maximum of one column and one row if used in the `SELECT`, `WHERE`, or `HAVING` clauses.
- A subquery can return multiple rows with `IN`, `ANY`, or `ALL` operators.
- A subquery that returns multiple rows can be used with `EXISTS` or `NOT EXISTS`.

You're doing great! With these basics out of the way, let's discuss more complex questions we can answer with subqueries.

More advanced subquery usage on our library database

Alright, leveling up... What if you were to find the number of books each author has written? Here's a subquery in the `SELECT` clause using an aggregate function to create a temporary column for the outer query's runtime. (You may need to read this twice, sorry about that. Look at the following example, that helps too!)

```
SELECT name,
(
    SELECT COUNT(*)
    FROM books
    WHERE author_id = authors.id
) AS book_count
FROM authors;
```

This is the breakdown of the preceding query:

- The inner query that executes first uses an aggregate function `COUNT()` to retrieve the count of books published by each author.
- The order of execution works such that first, the outer query fetches all rows from the `authors` table.
- The inner query is executed for each row to find the count of books for each author filtered by the id from the `authors` table.

As you can see, the results in the following screenshot depict all authors and their book count:

name	book_count
Emeka Okafor	2
Mei Lin	1
Sophia Martin	1
Juan Rodriguez	1
Aya Morimoto	1
Maria Low	0

6 rows in set (0.01 sec)

Fig 5.2 – Result of the subquery with COUNT

Okay, don't look at the following code snippet but see if you can come up with the solution to the following task yourself. Here's the task: find the book titles borrowed by members who joined the library between 1st May 2021 and 31st December 2022. Try to write your own query for this problem and then compare it to the solution here. Ready? Let's have a look.

```

SELECT id, title
FROM books
WHERE id IN (
    SELECT book_id
    FROM borrowed_books
    WHERE return_date IS NULL
    AND book_id IS NOT NULL
);
AND id IN (
    SELECT book_id
    FROM borrowed_books
    WHERE member_id IN (
        SELECT id
    )
);
```

```

        FROM members
    WHERE join_date BETWEEN '2021-05-01' AND '2022-12-31'
)
;

```

It might look daunting, but when you see what's going on, you'll find it simple. Before we explain the query, let's see the result:

id	title
3	Mastering Python
4	SQL for Data Science
5	The Basics of Quantum Computing
7	SQL for Data Analysis

4 rows in set (0.01 sec)

Fig 5.3 – Book titles borrowed by members who joined the library between 1st May and 31st December 2021

Okay, so how did we get there? Let's break it down. There are two inner queries. The first one selects all the ids of the books from the `borrowed_books` table for which the corresponding `return_date` is `NULL` and `book_id` is `NOT NULL` – books that are currently borrowed. The second subquery selects all ids from the `members` table who joined between May 2021 and December 2022. And then the outer query uses the ids returned from these subqueries as the condition to select the book titles and their ids from the `books` table. At this point, you're ready to face an even more complex subquery involving a `JOIN` that retrieves data from across three tables. Suppose we want to find out the books borrowed by the member who joined the library most recently. This task is not straightforward and requires two pieces of information. We first need to determine the most recent join date from the `members` table, and then we need to find the books borrowed by the member who joined on that date. Next, we need to query the `books` table for these books. Here's how to do it. Using a subquery, we are able to

accomplish this within a single SQL statement:

```
SELECT b.title
FROM books AS b
RIGHT JOIN borrowed_books AS bb ON b.id = bb.book_id
WHERE bb.member_id = (
    SELECT id
    FROM members
    WHERE join_date = (
        SELECT MAX(join_date)
        FROM members
    )
    LIMIT 1
);
```

This example has two nested inner queries as we select all ids from the `members` table based on their `join_date`. Note that we want to find the members who joined the library most recently, so we can use the `MAX(join_date)` function via a subquery within a subquery to find the `join_date` with the highest value, indicating the member who joined most recently. And finally, the main query retrieves the titles of the books borrowed by this member. We need the `LIMIT 1` at the end, because it's possible that there are multiple members who joined on the same latest date. (Not the case in our dataset though.)

```
+-----+
| title |
+-----+
| SQL for Data Analysis |
+-----+
1 row in set (0.01 sec)
```

Fig 5.4 – Result of books borrowed by most recent member

Now, isn't that neat? We solved a complex question with subqueries by breaking it down into smaller, more manageable queries. This is the true beauty of subqueries and something you will probably appreciate over time when dealing with complex problems. Next, we will explore another technique for dealing with complex problems and to help better organize our queries and subqueries: the **Common Table Expressions (CTEs)**.

Common Table Expressions

It's time to discuss CTEs. Just like subqueries, CTEs help us manage complexity by breaking down complex queries into smaller, more manageable parts. A CTE is a named temporary result set. It only exists within the scope of a single statement. Within this statement, it can be referenced multiple times. A CTE can be considered a temporary view, lasting the duration of a query only.

Use cases for CTEs

The advantage of using CTEs is that they simplify queries with multiple subqueries and make your SQL code easier to read. Instead of nesting multiple subqueries, you can declare CTEs at the beginning of your query. After that, you reference them just like you would with a regular table. CTEs eliminate the need to repeat complex subqueries. They also give us the opportunity to give our subquery a sensible name, making the code even more readable. The basic syntax for a CTE is as follows:

```
WITH cte_name AS (
    -- SQL query here
)
SELECT *
FROM cte_name;
```

The `WITH cte_name AS` in the preceding code is part of the definition of a CTE. The `cte_name` is the name that we use to refer to it. Within the following parentheses, you can write any SQL query, like you would with a subquery. We can use the `SELECT`, `INSERT`, `UPDATE`, `DELETE` statements. Once done, you can treat this CTE like any table in our database and fetch data from it multiple times simply by calling its name. Let's use our library

database and illustrate the use of CTEs with an example.

Examples with the library database

What approach would you take if you were tasked with finding out the book details of all authors from the USA? Oftentimes there are multiple ways to solve a problem – what you choose depends on what you want to optimize: performance, readability, or functionality. Here is how you would tackle this problem with a CTE:

```
WITH USA_authors AS (
    SELECT * FROM authors
    WHERE country = 'USA'
)
SELECT *
FROM books
WHERE author_id IN (
    SELECT id FROM USA_authors
);
```

With this CTE, we created a *named* subquery at the top and then called that temporary data result set in the sub(sequent) query when we needed it. The result of this query is shown in the following screenshot:

id	title	author_id	isbn	publication_year	publisher
4	SQL for Data Science	3	978-4444444444	2022	Science Publishing House

1 row in set (0.01 sec)

Fig 5.5 CTE to find book details written by USA authors

The CTE contains all rows of records from the `authors` table for any author born in the USA. Later on, when we need to find book details for all authors from the USA, we use the CTE to match the `id` with the `books` table and return matching records. This next example will emphasize the usefulness of CTEs in SQL. Imagine you want to find the member details of those who have borrowed books by the author with name Emeka Okafor. Take a moment to ground your approach before you dive into the following query:

```
WITH Emeka_books AS (
    SELECT id
```

```

        FROM books
        WHERE author_id = (
            SELECT id
            FROM authors
            WHERE name = 'Emeka Okafor'
        )
    ),
Emeka_borrowers AS (
    SELECT member_id
    FROM borrowed_books
    WHERE book_id IN (SELECT id FROM Emeka_books)
)
SELECT *
FROM members
WHERE id IN (SELECT member_id FROM Emeka_borrowers);

```

There are two CTEs in this query. In a way, it might remind you of nested subqueries. So, the first CTE finds out the ids of all books written by Emeka Okafor. The resulting temporary data set is assigned to be called `Emeka_books`. After that, another CTE is defined. This one is called `Emeka_borrowers` and it finds the member ids of library members who have borrowed books with the id range of Emeka Okafor's books. Note that we call upon our first CTE at this point instead of rewriting the whole subquery again. Finally, we select all the columns from the `members` table with ids matched by the ids of members who have borrowed books written by Emeka Okafor. The second CTE is used here to match the ids. Phew. The following figure shows the results:

id first_name last_name join_date email				
+-----+-----+-----+-----+-----+				
1 Alice Johnson 2021-01-01 alice.johnson@example.com				
3 Chiara Rossi 2021-05-10 chiara.rossi@example.com				
6 Femi Adeyemi 2021-11-10 femi.adeyemi@example.com				
+-----+-----+-----+-----+-----+				

3 rows in set (0.07 sec)

Fig 5.6 CTE query to find the members who borrowed books written by Emeka Okafor

Let's do something a bit different in the next query. Assume we want to determine the number of books each member borrowed in April 2022. We can do this by first creating a CTE that filters the `borrowed_books` table for

the relevant date range, and then joining this CTE with the `members` table:

```
WITH april_borrowings AS (
    SELECT member_id, COUNT(*) AS books_borrowed
    FROM borrowed_books
    WHERE borrow_date >= '2022-04-01' AND borrow_date < '2022-05
-01'
    GROUP BY member_id
)
SELECT m.first_name, m.last_name, ab.books_borrowed
FROM members AS m
JOIN april_borrowings AS ab ON m.id = ab.member_id;
```

In the CTE `april_borrowings`, we count the number of books each member borrowed in April 2022 using the techniques we have learned in previous chapters. We filter our results using the `WHERE` clause and then group them by each `member_id`. Then, in the main query, we join this CTE with the `members` table to retrieve the members' names and the number of books each of them borrowed in April 2022. You can see the results in the screenshot:

first_name	last_name	books_borrowed
Alice	Johnson	1
Bob	Smith	1
Chiara	Rossi	1
David	Gonzalez	1
Eve	Garcia	1
Grace	Kim	1

6 rows in set (0.00 sec)

Fig 5.7 CTE query to find the number of books borrowed by each member in April 2022

With (yup, pun intended) a CTE, we've made the query easier to read and understand. We've separated the logic for counting the books borrowed in April 2022 from the logic for retrieving the member details, resulting in a cleaner query. Quite some advanced new topics, right? Hold on, there's still

more to come! Next, we will learn about **window functions**.

Window functions: A panoramic view of your data

At this point, we have learned a great deal about SQL, but we're about to enter a new territory that will add a dynamic dimension to our SQL capabilities. We're about to learn about window functions. They might be a bit tough at first; but we promise you, they're worth it. Window functions in SQL are functions that perform a calculation across a set of rows that are related to the current row. In other words, window functions allow you to create a *window* of rows with which we can perform calculations. This window, also known as a frame, can be defined in various ways: it could be all rows in the table, the rows with the same value in a specific column, or a range of rows before or after the current row. The great thing about window functions is that they don't cause rows to become grouped into a single output row like some other functions do. Instead, each row retains its individual identity. Here is the general syntax for a window function:

```
<function>(<expression>) OVER (
    [PARTITION BY column1, ..., columnN]
    [ORDER BY column1 [ASC|DESC] [NULLS {FIRST|LAST}]]
    [ROWS BETWEEN start AND end]
)
```

The `<function>` indicates the window functions to be used, such as `SUM`, `RANK`, and `ROW_NUMBER`. The `<expression>` refers to the columns or expressions upon which the window function is called. `OVER (...)` indicates the window frame. The square brackets inside the parentheses of `OVER` indicate optional parameters. We may define partitions or groups from the dataset over columns. The window function applies separately to each partition. We may also order the results by specific columns in any order. And we can define the ordering for `NULL` values too. This can be done with `NULLS FIRST` or `NULLS LAST` (however, MySQL does not support this). Lastly, another optional clause, `ROWS BETWEEN start AND end`, defines each partition's window frame and specifies the range of rows to be used. Often, we can replace the start and end with keywords such as `UNBOUNDED PRECEDING`, `CURRENT ROW`, or numeric offsets, as we will see.

Example with the Library Database

Let's illustrate the use of window functions with a real example using our library database. We will find the first and last borrowing dates for each of the members in our library. How do we do that? Well... here it is:

```
SELECT member_id, MIN(borrow_date) OVER (PARTITION BY member_id)
    AS first_borrow_date,
    MAX(borrow_date) OVER (PARTITION BY member_id) AS last_borrow_date
FROM borrowed_books;
```

We're selecting `member_id` from the `borrowed_books` table. (This will return the `member_id` for every row in the `borrowed_books` table.) The `MIN(borrow_date) OVER (PARTITION BY member_id) AS first_borrow_date` expression uses a window function to find the earliest `borrow_date` for each `member_id`. The `PARTITION BY member_id` clause tells SQL to treat each unique `member_id` as a separate group, or *partition*. Then, within each partition, the `MIN(borrow_date)` function identifies the earliest `borrow_date`. The result of this operation is assigned the alias `first_borrow_date`. Similarly, the `MAX(borrow_date) OVER (PARTITION BY member_id) AS last_borrow_date` expression finds the latest `borrow_date` for each `member_id` (the most recent date on which each member borrowed a book). Again, the `PARTITION BY member_id` clause groups the data by `member_id`, and the `MAX(borrow_date)` function identifies the latest `borrow_date` within each group. The alias `last_borrow_date` is given to the result. So, for each row in the `borrowed_books` table, this query will return the `member_id`, the date of the first book that member borrowed (`first_borrow_date`), and the date of the last book that member borrowed (`last_borrow_date`). Your results should be similar to this:

member_id	first_borrow_date	last_borrow_date
NULL	2022-04-25	2022-05-05
NULL	2022-04-25	2022-05-05
1	2022-04-01	2022-04-01
2	2022-04-05	2022-04-05
3	2022-04-10	2022-04-10
4	2022-04-15	2022-04-15
5	2022-04-20	2022-04-20
6	NULL	NULL
7	2022-04-30	2022-04-30
8	2022-05-02	2022-05-02
10	2022-05-08	2022-05-08

11 rows in set (0.00 sec)

Fig 5.8 Window Function query to find the first and last borrowing date of each member

Multiple entries for a member

Please note that this query could return multiple rows for each `member_id`, because it will return a row for each entry in `borrowed_books`. If you'd like to see just one row for each member, with their first and last borrow dates, you could modify the query to use the `GROUP BY` clause:

```
SELECT member_id, MIN(borrow_date) AS first_borrow_date, MAX(borrow_date) AS last_borrow_date
FROM borrowed_books
GROUP BY member_id;
```

In this version of the query, instead of using window functions, you're using aggregation functions (`MIN` and `MAX`) with a `GROUP BY` clause to find the earliest and latest borrow dates for each member.

Okay, a new example! What if you want to find the ranks of the books based on the number of times they have been borrowed? This is a question that the

window function specializes in, using its `RANK` function. We are going to solve this using a window function and a CTE. (Let's hope you still remember CTEs. If not, nothing to be ashamed about. You're learning so much new stuff! Just glance at them once more, they're waiting for your return a few pages back.)

```
WITH book_borrow_counts AS (
    SELECT book_id, COUNT(member_id) AS total_borrows
    FROM borrowed_books
    GROUP BY book_id
)
SELECT b.title, bbc.book_id, bbc.total_borrows,
RANK() OVER (
    ORDER BY total_borrows DESC
) AS borrow_rank
FROM book_borrow_counts bbc
JOIN books b on b.id = bbc.book_id;
```

The preceding query has two parts: a CTE and a window function. In the CTE (the `WITH` clause), we find the number of times each book has been borrowed by a member and then use this information in the main query to find the rank of books based on how many times they have been borrowed in the window function. This is done by joining the `book_borrow_counts` CTE with the `books` table on the id of the books. Your result should look something like this:

title	book_id	total_borrows	borrow_rank
Mastering Python	3	2	1
SQL for Data Science	4	2	1
Data Analytics for Beginners	1	1	3
The Art of Data Analysis	2	1	3
The Basics of Quantum Computing	5	1	3
SQL for Data Analysis	7	1	3
The Basics of Blockchain	6	0	7

7 rows in set (0.05 sec)

Fig 5.9 Window Function query to find the ranks of books.

Moving on, imagine that we want to find out the cumulative number of books borrowed by each member in chronological order. For this, we can use the

`SUM` function as a window function and the `OVER` clause to define our window. Here's how we can do it:

```
SELECT
    m.first_name,
    m.last_name,
    bb.borrow_date,
    COUNT(*) OVER (
        PARTITION BY bb.member_id
        ORDER BY bb.borrow_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS cumulative_books_borrowed
FROM
    members AS m
    JOIN borrowed_books AS bb ON m.id = bb.member_id
ORDER BY
    bb.member_id,
    bb.borrow_date;
```

In this query, the `COUNT(*) OVER (...)` construct is our window function. The `PARTITION BY bb.member_id` clause means that we're creating a separate window for each member. The `ORDER BY bb.borrow_date` clause means that the rows are ordered by the borrow date within each window. Finally, the

`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` clause means that for each row, the window includes all preceding and current rows. The output should look like this:

first_name	last_name	borrow_date	cumulative_books_borrowed
Alice	Johnson	2022-04-01	1
Bob	Smith	2022-04-05	1
Chiara	Rossi	2022-04-10	1
David	Gonzalez	2022-04-15	1
Eve	Garcia	2022-04-20	1
Femi	Adeyemi	NULL	1
Grace	Kim	2022-04-30	1
Henrik	Jensen	2022-05-02	1
Jia	Wang	2022-05-08	1

9 rows in set (0.05 sec)

Fig 5.10: Window Function query to find the cumulative number of books

borrowed by each member in chronological order

As you can see, this query returns a list of all books borrowed, along with the members who borrowed them. For each row, the cumulative number of books that the member had borrowed up to and including the borrow date of that row. Phew, that was quite the topic. We're not done yet! Next, we're going to talk about date and time manipulation in SQL. We scratched the surface of this in *Chapter 1* already. It's a critical skill for any data analyst, so let's dive a bit deeper. It is a little easier to digest, your brain might appreciate that after all the hard work it had to do in this chapter so far!

Understanding date time manipulation

Every data analyst will find themselves at some point needing to manipulate date and time data. With SQL, this task is not only possible but also very enjoyable (sure, enjoyable is subjective, but don't worry... these authors do know a thing or two about fun(ctions)). SQL has a lot of built-in functions that allow us to extract and manipulate date and time data. Let's explore the date-time functions available in MySQL.

Date and time functions

There are a lot of functions for dealing with dates and times. These include functions to extract parts of a date (such as the year, month, or day), to calculate the difference between two dates, or to format a date as a string. Some of the most commonly used date and time functions in SQL are:

- `NOW()` : Returns the current date and time.
- `CURDATE()` : Returns the current date only.
- `DAY()` , `MONTH()` , `YEAR()` : Extracts the day, month, or year from a date.
- `DATEDIFF()` : Calculates the difference between two dates.
- `DATE_ADD()` , `DATE_SUB()` : Adds or subtracts a specified time interval from a date.
- `DAYOFWEEK()` , `DAYOFMONTH()` , `DAYNAME()` , `MONTHNAME()` , `DAYOFYEAR()` : Find the day (number) and day/month names for any given date.

Before we return to our library database for more practical examples, try out the `NOW()` or `CURDATE()` functions. Simply type in:

```
SELECT NOW();
```

or

```
SELECT CURDATE();
```

The result of the first one will show you the current date with a timestamp, while the second query will only show the current date without the time.

Examples with the library database

Alright, back to the library database example. Let's say we want to find out how many days after the due date was each book returned. We can calculate this by subtracting the due date from the return date using the `DATEDIFF()` function. Here's how we do it:

```
SELECT
    b.title,
    m.first_name,
    m.last_name,
    bb.due_date,
    bb.return_date,
    DATEDIFF(bb.return_date, bb.due_date) AS days_late
FROM books AS b
JOIN borrowed_books AS bb ON b.id = bb.book_id
JOIN members AS m ON bb.member_id = m.id
WHERE
    bb.return_date IS NOT NULL;
```

This query will return a list of all books that have been returned, along with the member who borrowed them, the due date, the return date, and the number of days late. Cool, right? Let's first look at the output we get before we explain it in more detail.

title	first_name	last_name	due_date	return_date	days_late
The Art of Data Analysis	Bob	Smith	2022-05-05	2022-10-11	159
SQL for Data Science	Henrik	Jensen	2022-06-02	2022-12-15	196

2 rows in set (0.02 sec)

Fig 5.11 DATEDIFF() Function to find the number of days late on book returns.

The `DATEDIFF` function takes two inputs, the later and the earlier date. It subtracts these dates to find the number of days of difference. Then, it finds the difference between these two given dates in days. We also link the `books` table with the `members` and `borrowed_books` tables using `JOIN` twice and filter the results based on the condition that `return_date` is `NOT NULL`. Let's look at one more example. This time, first see if you can come up with the solution before sneak peaking at the solution. Here's the exciting question: can you create a table in the query result to find each book's year, month, and day from the `borrow_date`? Once you've tried that out yourself, compare it with the solution below.

```
SELECT book_id,
       YEAR(borrow_date) AS borrow_year,
       MONTH(borrow_date) AS borrow_month,
       DAY(borrow_date) AS borrow_day
  FROM borrowed_books
 WHERE borrow_date IS NOT NULL;
```

The preceding query is selecting `book_id` from the `borrowed_books` table. It will return the `book_id` for every row in the table where `borrow_date` is not null. The `YEAR(borrow_date) AS borrow_year` expression uses the `YEAR` function to extract the year part from the `borrow_date` field. We call this result `borrow_year`. Next, we do something very similar for the day and the month. Finally, the `WHERE borrow_date IS NOT NULL` clause filters out any rows where `borrow_date` is null. We need this because the `YEAR`, `MONTH`, and `DAY` functions can't operate on null values. You can see results similar to the following screenshot:

book_id	borrow_year	borrow_month	borrow_day
1	2022	4	1
2	2022	4	5
3	2022	4	10
4	2022	4	15
5	2022	4	20
6	2022	4	25
NULL	2022	4	30
4	2022	5	2
NULL	2022	5	5
7	2022	5	8

10 rows in set (0.00 sec)

Fig 5.12 - YEAR(), MONTH(), DAY() functions applied on the borrowing date.

Let's do one more example. What if we want to determine how many books are currently borrowed and overdue? We can do this by comparing the due date with the current date and finding out which books have a `NULL` in `return_date` (not yet returned) and the due date of returning has passed:

```

SELECT
    COUNT(*) AS Books_Borrowed_Currently
FROM
    borrowed_books
WHERE
    due_date < CURDATE() AND return_date IS NULL;

```

So, this query will count all the books in the `borrowed_books` table that are currently overdue and have not yet been returned. The `Books_Borrowed_Currently` in the result will tell you the total number of such books. You can see the result in the following screenshot:

Books_Borrowed_Currently
8

1 row in set (0.01 sec)

Fig 5.13 - Find the number of books currently borrowed and overdue.

Here's another challenging yet fun task for you. Instead of finding only the year, month, and day of the books borrowed, let's find the day of the week, month and year, and the day and month names for the books borrowed. You might have to explore the functions required for this problem.

```
SELECT book_id,
       DAYOFWEEK(borrow_date) AS DayOfWeek,
       DAYOFMONTH(borrow_date) AS DayOfMonth,
       DAYNAME(borrow_date) AS DayName,
       MONTHNAME(borrow_date) AS MonthName,
       DAYOFYEAR(borrow_date) AS DayOfYear
  FROM borrowed_books
 WHERE borrow_date IS NOT NULL;
```

This is doing something very similar to the day, month and year functions. You can see the results here:

book_id	DayOfWeek	DayOfMonth	DayName	MonthName	DayOfYear
1	6	1	Friday	April	91
2	3	5	Tuesday	April	95
3	1	10	Sunday	April	100
4	6	15	Friday	April	105
5	4	20	Wednesday	April	110
6	2	25	Monday	April	115
NULL	7	30	Saturday	April	120
4	2	2	Monday	May	122
NULL	5	5	Thursday	May	125
7	1	8	Sunday	May	128

10 rows in set (0.01 sec)

Fig 5.14 Results of finding each book's borrowing days and day names.

This is probably enough for `NOW()`. (Yup, definitely for me.) Date and time functions are something we typically use quite a bit in our data analysis tasks. And we've made our way through! We're ready to move to the next section, where we will look at text manipulation functions.

Understanding text manipulation

Textual data is omnipresent in every database and therefore vital to data analysis. SQL doesn't let us down and provides a lot of built-in functions to interact with textual data. We can extract, concatenate, replace, and perform a range of other operations on strings in SQL. Let's discuss them!

Text functions

So, SQL offers numerous text functions. These include the following:

- `LENGTH()` : Returns the length of a string.
- `UPPER()`, `LOWER()` : Converts a string to uppercase or lowercase.
- `CONCAT()` : Concatenates two or more strings into one.
- `SUBSTRING()` : Extracts a substring from a string.
- `REPLACE()` : Replaces occurrences of a substring within a string.
- `TRIM()` : Removes leading and trailing spaces from a string.

Let's see these functions in action in some examples.

Text functions in the library databaseU

You might be surprised, but we will use the library database (of course) to see these functions in action. Suppose we want to generate a list of all members. Instead of displaying their first and last names separately, we want to display their full names in one column, in uppercase. We can accomplish this by using the `CONCAT()` and `UPPER()` functions as seen here:

```
SELECT
    UPPER(CONCAT(first_name, ' ', last_name)) AS full_name
FROM
    members;
```

We're assuming that you're a bit comfortable with functions already. We wrap a function inside a function. First the `CONCAT` function is executed. This is going to concat the three strings: the first name, a space, and the last name. After that, the result is the input for the `UPPER` function, which is going to change the string containing the name to uppercase. Lastly, we give it the alias `full_name`. You can see the result in *Figure 5.15*.

full_name
ALICE JOHNSON
BOB SMITH
CHIARA ROSSI
DAVID GONZALEZ
EVE GARCIA
FEMI ADEYEMI
GRACE KIM
HENRIK JENSEN
INGRID PETTERSSON
JIA WANG

10 rows in set (0.05 sec)

Fig 5.15: Results of concatenating the first and last names of the members and converting it to uppercase

Next, let's see how to replace a specific text portion in strings with something else you want. Here's the query to do that:

```
SELECT title,
```

```
REPLACE(publisher, 'Publishing', 'Analytics') AS Analytics_pub  
FROM books;
```

The output of this query can be seen in *Figure 5.16*. It's a table with two columns: `title` (which simply contains the title of each book as listed in the `books` table) and `Analytics_pub` (which contains the name of the publisher for each book, but with every instance of the word 'Publishing' replaced with 'Analytics').

title	Analytics_pub
Data Analytics for Beginners	DataPress Analytics
The Art of Data Analysis	Analytical Press
Mastering Python	Python Publishers
SQL for Data Science	Science Analytics House
The Basics of Quantum Computing	Quantum Press
The Basics of Blockchain	Blockchain Press
SQL for Data Analysis	Data Analysis Press
NULL	AI Analytics

8 rows in set (0.00 sec)

Fig 5.16 Results of the Replace function.

Let's take another example to clarify the other text functions we listed. Suppose we want to know how many books each member has borrowed. Moreover, we want to return the respective member name with their borrowed book count, but the member names in the `members` table are recorded in two different columns. And let's say we're not sure about the casing of these columns (although in our example, it's actually nicely done already). We want to make sure they're all title case, which means that the first letters of the words are in uppercase. We can achieve this by using the `CONCAT()`, `UPPER()`, `LOWER()`, and `SUBSTRING()` functions:

```
WITH member_names AS (  
    SELECT id, CONCAT(UPPER(SUBSTRING(first_name, 1, 1)), LOWER(  
        SUBSTRING(first_name, 2))) AS FirstName,  
        CONCAT(UPPER(SUBSTRING(last_name, 1, 1)), LOWER(SUBSTRING(la  
st_name, 2))) AS LastName  
    FROM members  
)
```

```

SELECT
    CONCAT(FirstName, ' ', LastName) AS FullName,
    COALESCE(COUNT(b.member_id), 0) AS num_books
FROM member_names AS m
LEFT JOIN borrowed_books AS b ON m.id = b.member_id
GROUP BY m.id;

```

This query will return a list of all authors, with their names in the title case, along with the number of books each has written:

FullName	num_books
Alice Johnson	1
Bob Smith	1
Chiara Rossi	1
David Gonzalez	1
Eve Garcia	1
Femi Adeyemi	1
Grace Kim	1
Henrik Jensen	1
Ingrid Pettersson	0
Jia Wang	1

10 rows in set (0.01 sec)

Fig 5.17: Results of the query to show the full member names in title case and the number of books they have borrowed.

We combined a CTE and `LEFT JOIN` to solve this problem. The CTE works to create a temporary data set that stores three columns from the `members`

table: id, title case first name, and title case last name. Also, we used the `SUBSTRING()`, `UPPER()`, `LOWER()` and `CONCAT()` functions to create the columns to store title case first and last names.

`CONCAT(UPPER(SUBSTRING(first_name, 1, 1)), LOWER(SUBSTRING(first_name, 2, LENGTH(first_name) - 1)))` is quite the expression. But don't be intimidated, let's break it down into smaller chunks. This expression takes the `first_name` field, extracts the first character (`SUBSTRING(first_name, 1, 1)`) and converts it to uppercase (`UPPER`). It then extracts the remainder of the string starting from the second character (`SUBSTRING(first_name, 2)`) and converts it to lowercase (`LOWER`). These two parts are then concatenated together using `CONCAT`, resulting in a `first_name` that is in title case. This is assigned the alias `FirstName`. We repeat the same process for the last name. After that, we call this CTE in the query and find the count of books borrowed by each member from the `borrowed_books` table with a `LEFT JOIN` to include the `NULL`s in the member id. The `COALESCE()` function replaces a `NULL` value with `0`, hence the `0` after "Ingrid Pettersson". That's enough on text functions. You're almost there! In the next section, we will wrap up our discussion on advanced SQL by sharing some best practices.

Best practices: bringing it all together

Having learned all these advanced SQL concepts, we should ensure that we're using them in the best way possible. The following best practices can help us write queries that are not only functional but also easy to read and high performing.

Write readable SQL Code

As always, writing SQL code that you and your colleagues can easily understand is vital. Even more so when this is advanced SQL! This includes formatting your SQL code consistently and using clear and descriptive names for tables and columns. It also helps to comment on your code to explain what it does and why you wrote it that way so anyone (including you) can understand what's going on if you need to work on it months later. For example, consider the following query:

```
select CONCAT(m.first_name, ' ', m.last_name) as fn from members
```

```
as m;
```

It could be made more readable by writing SQL keywords in uppercase, adding some formatting, a better name for `fn`, and a comment:

```
-- Get a list of full names for all members
SELECT
    CONCAT(m.first_name, ' ', m.last_name) AS full_name
FROM
    members AS m;
```

The second approach is quite easier to read. While proper formatting, uppercase keywords, and sensible naming are not mandatory in SQL, it's a great practice to follow to gain professionalism.

Be careful with NULL values

NULL values can often lead to unexpected results, so handling them correctly is crucial.

Use subqueries and CTEs wisely

While subqueries and CTEs can make your SQL queries much more flexible, they can also make them more complex and complicated. Avoid nesting subqueries too deeply, as this can make your query hard to follow and debug. Think of it like a maze, and the more turns and twists you add, the harder it is to find your way back. Also, be aware that subqueries and CTEs can sometimes lead to inefficient queries, as they can cause the database to process more data than necessary. And that brings me to our next point...

Think about performance

Consider the performance implications of your SQL queries. This includes being mindful of the amount of data you're working with, the complexity of your queries, and the indexes available on your tables. Try to write efficient queries that process as little data as necessary to get the desired results. This is why often you'll want to use a `JOIN` instead of a subquery or vice versa, depending on many factors such as dataset size, indexes, number of `JOINS`, etc. Prefix your queries with the `EXPLAIN` keyword to view the underlying performance.

Test your queries

Finally, always test your SQL queries to ensure they give you the expected results. This includes testing with different inputs and edge cases and checking your results for accuracy and completeness. Sometimes a query might provide you with the correct results but has an underlying failure point. And that's it! By applying these advanced SQL concepts and following these best practices, you're now well-equipped to tackle complex data analysis tasks and generate insights from your data. With that, let's wrap up this chapter.

Summary

In this chapter, we tackled the more complex aspects of SQL. You should now be equipped with the ability to complete difficult data analysis tasks. The key lies in mastering advanced SQL techniques such as subqueries, CTEs, window functions, and date-time and text manipulation techniques. We started this advanced journey by exploring subqueries, a great tool allowing you to nest queries within queries. They contribute to solving tough data problems in manageable parts. Next, we introduced CTEs to simplify complex queries by breaking them into smaller, more digestible sections. This makes the code more readable which then leads to better maintainable code. We then delved into the concept of window functions which enables you to perform calculations over sets of rows related to the current row. This advanced feature has opened new horizons for data analysis, allowing for complex calculations like running totals and rankings. After this, we ventured into the realm of date-time manipulation. We provided you with the tools and techniques to manage and manipulate date and time data effectively, these will help you to get valuable time-based insights. And in the final section, we explored text manipulation. The text functions are a must-have for dealing with string data. We covered various SQL functions to process and manipulate textual data. To bring these concepts to life, we used practical examples from our library database. And now we're ready for the next step: test all this SQL knowledge in practice with a case study! Spoiler that might make you happy: after quite some chapters, we're going to let go of the library example!

6 SQL for Data Analysis Case Study

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



At this point, you're finally prepared with the necessary tools to deal with a more complex project from scratch. And that's exactly what we're going to be doing in this chapter. In a nutshell, we'll:

- Set up our database
- Familiarize ourselves with the data
- Write queries to analyze the sales of EcommerceHub store

ECommerceHub is an (imaginary) international e-commerce store run by the (imaginary) famous Umar Ali with a wide variety of inventory and a diverse customer base. He needs our help analyzing his data. No need to keep him waiting any longer, let's get started!

Setting up the database

In the GitHub repository, you'll find a file `import.sql` for this chapter. This is the file you need to run before continuing with this chapter. It takes care of the database creation and insertion queries you'll need. Make sure you have run the queries before going further. You can simply download and import, or copy and run the SQL. After that, you have the database ready to analyze and you will try to solve some analytical questions from our client

EcommerceHub. The database is conveniently called EcommerceHub as well. There are seven tables in this database; we should understand what each table does before moving on to the analyzation phase. This will be crucial as the owner of this store expects you to answer some questions related to understanding and boosting the efficiency of his business in a data-friendly way. We will help Umar, the owner of this store, decide whether it is profitable to keep running ECommerceHub or not. Before we actually dive in, let's talk about how to approach a problem like this in general (no worries, I'm dying to get started too; we'll keep it short).

Performing data analysis with SQL

There's not just one way to answer complex questions. However, we do try to give you something to hold on to. Here's a basic roadmap you can use when answering questions with SQL:

- **Exploring the data:** Make sure that you are familiar with the data in the database. Understand what kind of tables there are, what sort of data there is and how the tables are related.
- **Know the dataset:** Find out the essential words you must focus on because this will help you know what you need to fetch or where from.
- **Map to their respective table(s):** For instance, customer details are in the `customer`, `orders` and `shopping_information` tables; suppose you need to find the largest purchase, then you figure out that there is a column called `total_cost` which is in the `orders` table. That's some great insights to get started with!
- **Determine and choose a primary table to fetch data:** Here, you need to ensure the table you choose has the most relevant information, and you can join other tables to it later on for secondary information.
- **Determine if a JOIN might be necessary:** Sometimes, you need to gain some other insightful information for this question and other times, you only need a single table.
- **Identify whether you need to use aggregate functions, subqueries, CTEs or other complex elements of SQL queries:** According to what information is required, you might need to use more complex features if the required data is derivative or not directly present in any table.

We are going to make the distinction between exploring the data and analyzing the data. Let's first explore our dataset.

Exploring the data

Normally, you'd have to figure this out yourself. So please take a little break and look at the tables and see if you can understand what's going on by looking at the columns and the data. After that, go ahead and have a look at this summary to compare this to what you thought when looking at the tables that exist in ECommerceHub:

- `product_category` contains data for each product category in the e-commerce store.
- `payment_method` contains data for each payment method type accepted by the e-commerce store.
- `customer` has the necessary details about the customers who have signed up.
- `product` is an entire product catalogue with the unit cost price of each product.
- `orders` contains the complete information about orders, order dates, and total costs after (a possible) discount.
- `shipping_information` has all the relevant shipping information necessary for delivery and tracking.
- `product_reviews` contains reviews and ratings from customers who have purchased products from the store.

Now that we know this, let's see if we can help Umar with some of his queries by writing some queries.

General data insights

Umar has hired many product administrators who were adding new products from diverse places. They probably did some amazing work, but he completely lost the overview. He wants your help with analyzing the data of his store and has prepared a few questions. This will help him decide the future of the store. First of all, Umar wants to see all the product categories that ECommerceHub offers to gain some insight into the many types of

categories that the store offers. This is simple to tackle. It only requires a single `SELECT` statement to fetch data from the `product_category` table. Try to do it without looking at the answer. And after that, here is how you can do it:

```
SELECT * FROM product_category;
```

ADD

category_id	category_name
1	Electronics
2	Clothing
3	Home & Kitchen
4	Beauty & Personal Care
5	Sports & Outdoors
6	Books
7	Toys & Games
8	Health & Wellness
9	Automotive
10	Office Supplies
11	Appliances
12	Furniture
13	Jewelry
14	Baby & Kids
15	Grocery & Gourmet Food
16	Pet Supplies
17	Music & Instruments
18	Movies & TV Shows
19	Computers & Accessories
20	Home Improvement
21	Garden & Outdoor
22	Tools & Hardware
23	Sports & Fitness
24	Fashion Accessories
25	Crafts & Sewing
26	Art & Collectibles
27	Industrial & Scientific
28	Party Supplies
29	Travel & Luggage
30	Office Electronics
31	Software & Video Games

31 rows in set (0.01 sec)

Figure 6.1: The list of categories in the store

As you can see in the preceding screenshot, there are 31 product categories offered by the e-store. All (in this case, only two) columns are shown because of the asterisk (*) after the `SELECT` keyword. It's important to remember that this is still a new business of his and that it might not look like a lot of data because of that. Now that we have explored the data roughly, it's time to start answering Umar's questions and take the step towards analyzing the data.

Analyzing the data

Umar has a lot of questions. In order to provide him with answers, we'll have to analyze the data for him. His first request is to dive into the clothing category.

Examining the clothing category

After seeing the output in *Figure 6.1*, Umar has an idea and wants to find out all products under the `Clothing` category in the store to know if their clothing catalogue is big enough to be placed in a better position on their home page.

```
SELECT p.product_id, p.product_name, p.description, p.product_price, p.discounted_price
FROM product p
JOIN product_category c ON p.category_id_fk = c.category_id
WHERE c.category_name = 'Clothing';
```

The preceding query will list the details of each product in the `Clothing` category, as you can see in the following screenshot:

product_id	product_name	description	product_price	discounted_price
4	Adidas Running Shoes	Comfortable and stylish running shoes	88.00	88.00
19	Nike Women's Running Shoes	Durable and comfortable running shoes	100.00	100.00
22	Calvin Klein Men's Slim Fit Suit	High fashion men's suit	300.00	300.00
35	Levi's Men's Jeans	Classic and comfortable men's jeans	60.00	60.00
39	Louis Vuitton Speedy 30 Bag	Luxury women's handbag	1200.00	1200.00
45	Adidas Men's Soccer Shoes	Durable and comfortable soccer shoes	85.00	85.00
51	Tommy Hilfiger Men's Polo Shirt	Comfortable and fashionable polo shirt	50.00	50.00
58	Champion Men's Powerblend Fleece Pullover Hoodie	Comfortable and warm pullover hoodie	48.00	34.00
70	Under Armour Men's HeatGear Compression Shirt	Performance compression shirt	38.00	27.00
73	The North Face Men's Resolve Jacket	Waterproof and breathable jacket	98.00	90.00
86	Nike Women's Revolution 5 Running Shoe	Comfortable and supportive running shoes	60.00	60.00
94	Levi's Men's 501 Original Fit Jeans	Classic straight-fit jeans	68.00	54.00
112	The North Face Women's Gotham Jacket II	Insulated and water resistant jacket	230.00	230.00

Figure 6.2: Product listings under the “Clothing” category

The query joins the `product` and `product_category` tables to return all the products in the `Clothing` category along with their `product_price` and `discounted_price` to present a fuller picture. The result in *Figure 6.2* shows that there are only 13 clothing products in the store. This means Umar is short-supplied if he wants to promote these products more.

Determining the number of customers

Moving on, Umar has been told the store doesn't have a lot of unique customers, and it's the same people who usually make purchases from the ECommerceHub. Since it's a relatively new business, Umar wants to know how many customers he has pulled in to reinforce or shake up the marketing strategy and answer the critics. Here's the query and its results:

```
SELECT COUNT(*) AS Num_Unique_Customers FROM customer;
```

```
+-----+
| Num_Unique_Customers |
+-----+
|          114          |
+-----+
1 row in set (0.04 sec)
```

6.3 The number of unique customers of ECommerceHub

As you can see, he has 114 unique customers. Let's see how they typically pay.

Researching the top payment methods

It's important for Umar to figure out the top payment methods currently offered that customers most commonly use. Knowing the top 5 payment methods will help him reduce the inefficiencies in payment methods by removing extra payment methods and enhancing the speed of the transactions on the remaining ones. Here's the query to do this:

```
SELECT payment_method_id_fk, COUNT(*) AS Count
FROM orders
GROUP BY payment_method_id_fk
ORDER BY Count DESC
LIMIT 5;
```

The following screenshot reflects the query output and shows that the payment method with id 3 is the most popular among the store's customers. Meanwhile, 1, 2, and 6 come in at second, followed by payment method with id 10 being the third most popular.

payment_method_id_fk	Count
3	9
1	7
2	7
6	7
10	6

5 rows in set (0.00 sec)

Figure 6.4: The most popular payment methods

The query fetches the `payment_method_id_fk` from the `orders` table because it stores transaction information. An aggregate function, `COUNT(*)`, is also used which counts all the records in the table without any conditions and helps retrieve the number of orders for each payment method. This is done using the `GROUP BY` clause applied on the `payment_method_id_fk` part of the query, which ensures the count for each payment method is shown in the table. Then, a `LIMIT` of 5 combined with the `ORDER BY Count DESC` ensures only the details of the five most popular methods are displayed. You could find additional information if you `JOIN` the `payment_method_id` table to display the corresponding `payment_method_name` of the id in the same query. Why don't you run this query and see what the output is?

```
SELECT o.payment_method_id_fk, p.payment_method_name, COUNT(*) AS Count
FROM orders o
JOIN payment_method p
ON o.payment_method_id_fk = p.payment_method_id
GROUP BY o.payment_method_id_fk, p.payment_method_name
ORDER BY Count DESC
LIMIT 5;
```

The preceding query uses an `INNER JOIN` on the `payment_method_id` to ensure that only results matched in both tables are obtained. The `JOIN` is used to get the `payment_method_name` from the `payment_method` table. Subsequently, the results in the following screenshot are now grouped by the payment method id and payment method name:

<code>payment_method_id_fk</code>	<code>payment_method_name</code>	<code>Count</code>
3	PayPal	9
1	Credit Card	7
2	Debit Card	7
6	Bank Transfer	7
10	Skrill	6

5 rows in set (0.00 sec)

Figure 6.5: Enhanced top payment methods query with the payment method name

This gives us sufficient insights into the payment methods. Let's help Umar with his questions regarding customer feedback.

Gathering customer feedback

So, another thing that Umar is curious about is customer feedback on the products sold because it is the key to a thriving business, and he wants you to find which of the store's products have been most liked by the customers who bought them. Let's find out how:

```
SELECT product_id_fk, AVG(rating) AS Average_Rating
FROM product_reviews
GROUP BY product_id_fk
ORDER BY Average_Rating DESC;
```

This query `SELECT`s the `product_id_fk` from the `product_reviews` table and the `AVG` ratings, and also uses a `GROUP BY` clause on the ids so that the result shows the average rating of each `product_id` and ordered with the highest average rating in descending order. This provides a higher-level view

of the information to quickly glance at and identify which products are the customers' favorite.

product_id_fk	Average_Rating
34	4.7500
35	4.5714
27	4.5556
15	4.5000
43	4.5000
12	4.4286
19	4.4286
3	4.3750
21	4.2857
11	4.0000
17	4.0000
38	3.8333
16	3.6364
8	3.5000
40	3.4000
13	3.0000

16 rows in set (0.01 sec)

Figure 6.6: Brief customer feedback query results showing average ratings for top products.

However, the output in the preceding screenshot might not satisfy what Umar Ali seeks when asking for customer feedback. It gives a general overview,

but it does not present the actual feedback. We can improve the query and fetch more relevant information to help make the results more meaningful. Here's the updated query:

```
SELECT product_id_fk, AVG(rating) AS average_rating, GROUP_CONCAT(review SEPARATOR '|| ') AS reviews
FROM product_reviews
GROUP BY product_id_fk
HAVING AVG(rating) > 4
ORDER BY average_rating DESC;
```

The preceding query results in the following screenshot is more granular and shows the individual reviews. It treats each unique id and review as a separate group, and filters out the records with an average rating lower than 4.

product_id_fk	average_rating	reviews
34	4.7500	Impressive video quality and good battery life. Outstanding video quality. Very satisfied! Crystal clear video quality, impressive battery. Great video quality and battery life. Good video quality, could use better battery. Great video quality, impressive battery life. Impressive video quality, decent battery life. Great video quality, impressive battery life.
35	4.5714	Fits well and extremely comfortable. Highly recommended. The perfect fit and comfort. My new favorite! The best fit I have found yet! Comfortable and fits well, but color faded after washing. Perfect fit, very comfortable. Fits well, but color is not as shown. Comfortable, but color faded quickly.
27	4.5556	Sound quality is top-notch. It's a joy to play. Great sound quality. Makes playing music a pleasure. Sound is good, but the battery life is not impressive. Excellent sound quality. I love this instrument! Excellent sound quality. Absolutely a pleasure to play. Excellent sound quality. Highly recommended. Excellent sound quality, worth the price. Good sound quality, a bit overpriced. Good sound quality, but a bit pricey.
16	4.5000	Sleek design and accurate timekeeping. Love it! Love the design! And it keeps time perfectly. Nice design, but a bit hard to read at times. Sleek design, hard to read in sunlight.
43	4.5000	Good camera and screen quality. Value for money. Impressive camera quality. Definitely a good buy. Good camera, screen resolution could be better. Decent camera quality, value for money. Excellent camera quality, worth the price. Great camera quality, definitely worth the money.
12	4.4286	Comfortable chair. Great for long hours at the office. The chair is extremely comfortable, even after long hours. Comfortable, could sit in it for hours. Very comfortable, excellent for working long hours. Comfortable for working long hours. Very comfortable, great for working long hours. Perfect for long hours, very comfortable.
19	4.4286	Comfortable shoes for everyday wear. Great for running but a bit tight around the toes. These shoes are perfect for my daily jogging routine. Great for running, perfect fit. Great fit for running, very comfortable. Comfortable for running, nice fit. Great for running, a bit tight.
3	4.3750	Solid phone. Very happy with my purchase. The best phone I've ever used. Highly recommended. Sturdy phone with good battery life. Fast and efficient phone. Good value for money. Good phone, could use more features. Sturdy phone, long battery life. Decent phone, could use more features. Decent phone, long battery life.
21	4.2857	Really immersive experience! Changed the way I game. The VR experience is quite good, but setup was complex. Really fun, but setup is tricky. Good VR experience, but the setup was complicated. Great VR experience, setup was a bit complicated. Immersive VR experience, worth every penny. Immersive VR, but needs better instructions.

Figure 6.7: Detailed customer feedback alongside combined reviews and ratings for highest ratings

ADD

Exploring the relationship between ratings and sales

You can use this information about the products with the best average ratings to determine for Umar if these reviews also indicate sales of these products. This would help determine whether these products yielded the same performance and indicate whether the positive reviews converted into sales.

The following is the query for that:

```
SELECT p.product_id, p.product_name,
SUM(o.quantity) as total_sold
FROM product p
JOIN orders o ON p.product_id = o.product_id_fk
WHERE p.product_id IN (34,35,27,15,43,12,19,3,21,11,17,38,16,8,4
0,13)
GROUP BY p.product_id, p.product_name
ORDER BY total_sold DESC;
```

The preceding query uses a `JOIN` to connect the product and orders table to obtain the `product_id`, `product_name` and `SUM` of the number of orders from the orders table. Additionally, it filters the results using the `WHERE` and `IN` clauses to check for specific product ids to return (only if they exist). The product ids retrieved here were the same as our last query when we determined the highest average ratings.

product_id	product_name	total_sold
16	PlayStation 5	54
15	Casio G-Shock Watch	46
35	Levi's Men's Jeans	45
11	Mobil 1 Engine Oil	44
27	Taylor Acoustic Guitar	40
19	Nike Women's Running Shoes	37
21	Oculus Quest 2	37
12	Staples Office Chair	36
40	Oster Blender	27
43	Google Pixel 7	24
34	GoPro HERO9	23
38	Sony WH-1000XM5 Headphones	19
17	Samsung Galaxy S23	11
13	HP Pavilion Gaming Laptop	8
3	Apple iPhone 14	2

15 rows in set (0.04 sec)

Figure 6.8: Query results to find best-selling products from ECommerceHub

The critical insight here is that you cannot decide which products sell well just based on their average ratings because there isn't any correlation. Note that the product with `product_id` 8 doesn't have sales (as seen by it missing in *Figure 6.8*) despite showing up in the prior query for average ratings, indicating review(s) found despite purchase from Umar Ali's store. This is an area to improve for Umar so that only people who have purchased a specific product can review it. This would prevent the spread of misinformation through fake product reviews, which hurt customer trust and market reputation. As a next step, we're going to expand the previous query and combine the average rating with the `total_sold` table:

```
SELECT p.product_id, p.product_name, o.total_sold, r.average_rating
FROM product p
JOIN (
    SELECT o.product_id_fk, SUM(o.quantity) AS total_sold
    FROM orders o
    WHERE o.product_id_fk IN (
        SELECT product_id
        FROM product
        WHERE product_id IN      (34,35,27,15,43,12,19,3,21,11,17,38,16
,8,40,13)
    )
    GROUP BY o.product_id_fk
)
AS o ON p.product_id= o.product_id_fk
JOIN (
    SELECT product_id_fk,
    AVG(rating) AS average_rating
    FROM product_reviews
    GROUP BY product_id_fk
)
AS r ON p.product_id= r.product_id_fk
ORDER BY o.total_sold DESC;
```

Here are the results:

product_id	product_name	total_sold	average_rating
16	PlayStation 5	54	3.6364
15	Casio G-Shock Watch	46	4.5000
35	Levi's Men's Jeans	45	4.5714
11	Mobil 1 Engine Oil	44	4.0000
27	Taylor Acoustic Guitar	40	4.5556
19	Nike Women's Running Shoes	37	4.4286
21	Oculus Quest 2	37	4.2857
12	Staples Office Chair	36	4.4286
40	Oster Blender	27	3.4000
43	Google Pixel 7	24	4.5000
34	GoPro HERO9	23	4.7500
38	Sony WH-1000XM5 Headphones	19	3.8333
17	Samsung Galaxy S23	11	4.0000
13	HP Pavilion Gaming Laptop	8	3.0000
3	Apple iPhone 14	2	4.3750

15 rows in set (0.01 sec)

Figure 6.9: Total product items sold and their average rating

The preceding screenshot should tell Umar more about the relation between the number of products sold and their rating. The role of the rating is not super clear looking at this table, but let's see how many products even have reviews before we draw too many conclusions.

Finding the percentage of products with reviews

A review tends to be a meaningful indicator of bonding with an item, positively or negatively. Hence, Umar thinks it's valuable to find out what fraction of the products sold to customers have distinct reviews. Here's an SQL query:

```
SELECT ROUND(
  (
    SELECT COUNT(DISTINCT product_id_fk)
    FROM product_reviews
  )
  / COUNT(*)
  * 100, 2
) as percentage_prods_with_reviews
FROM product;
```

The preceding query might look a little intimidating. But let's break it down. It uses 3 aggregate functions:

- `DISTINCT`: unique `product_id` values are selected only, and multiple instances of the same id are considered as one;
- `COUNT`: finds the number of all records from the rows returned by the subquery while the `COUNT(*)` returns the number of all records in the product table;
- `ROUND`: This one takes two inputs, namely the value and the number of decimal places to round to. In this instance, the answer is rounded to 2 decimal places as it's sufficient for our purposes.

The results as shown in the following screenshot returns a miserly percentage of 14.16 which means, on average, out of 100 products, roughly 14 are reviewed:

percentage_prods_with_reviews
14.16

1 row in set (0.00 sec)

Figure 6.10: Percentage of products with reviews in the store

These results are not great for an e-commerce store because reviews are essential for building trust with the customer base as a relatively new store. Umar is a little shocked and sees a great opportunity for improvement.

Effectiveness of discounts

On a similar note, the next step in the sales analysis of this store is determining whether the discounted products have done well with respect to

total sales per product or not. Doing so would help Umar conclude if it's wise to keep the current approach or try a new tactic. Let's help him answer that question with the following query:

```
SELECT p.product_id, p.product_name,
SUM(o.quantity) as total_sold
FROM product p
JOIN orders o ON p.product_id = o.product_id_fk
WHERE p.is_on_discount = TRUE
GROUP BY p.product_id, p.product_name
ORDER BY total_sold DESC;
```

You can see the results in the following screenshot:

product_id	product_name	total_sold
48	Razer Huntsman Gaming Keyboard	57
21	Oculus Quest 2	37
40	Oster Blender	27
34	GoPro HERO9	23
64	iRobot Roomba 675 Robot Vacuum	21
44	Nikon D850 DSLR Camera	14
17	Samsung Galaxy S23	11
13	HP Pavilion Gaming Laptop	8
26	Six Times Forever	7
81	Timex Men's Expedition Scout 40 Watch	5
3	Apple iPhone 14	2

11 rows in set (0.01 sec)

Figure 6.11: Total sales for discounted products, grouped by product id and name

The preceding query uses a filter on the Boolean column of `is_on_discount` to return the discounted products only in the output. You could also improve the result shown by determining the total revenue generated from each discounted product sold to present a more comprehensive picture to Umar to help with the decision. ADD

```
SELECT p.product_id, p.product_name, p.discounted_price,
SUM(o.quantity) as total_sold,
SUM(o.quantity * p.discounted_price) as total_revenue
```

```

FROM product p
JOIN orders o ON p.product_id = o.product_id_fk
WHERE p.is_on_discount = TRUE
GROUP BY p.product_id, p.product_name, p.discounted_price
ORDER BY total_sold DESC, total_revenue DESC;

```

The preceding query adds a few things: another `SUM` column which returns the total revenue by multiplying the quantity and discounted prices at which the discounted products are sold, uses the `GROUP BY` clause on the discounted price, too, and adds the total revenue as one of the factors for the `ORDER BY` clause. The revenue shown in the following screenshot is not symmetric because each product is priced differently.

product_id	product_name	product_price	discounted_price	total_sold	total_revenue
48	Razer Huntsman Gaming Keyboard	150.00	127.50	57	7267.50
21	Oculus Quest 2	300.00	285.00	37	10545.00
40	Oster Blender	90.00	81.00	27	2187.00
34	GoPro HERO9	400.00	360.00	23	8280.00
64	iRobot Roomba 675 Robot Vacuum	280.00	238.00	21	4998.00
44	Nikon D850 DSLR Camera	2800.00	2660.00	14	37240.00
17	Samsung Galaxy S23	700.00	595.00	11	6545.00
13	HP Pavilion Gaming Laptop	800.00	720.00	8	5760.00
26	Six Times Forever	9.25	5.55	7	38.85
81	Timex Men's Expedition Scout 40 Watch	45.00	40.50	5	202.50
3	Apple iPhone 14	899.00	809.10	2	1618.20

11 rows in set (0.00 sec)

Figure 6.12: Query result of total sales and revenue generated from the discounted products

So, despite not selling even half the number of items as the Razer Huntsman Gaming Keyboard, the Nikon D850 DSLR Camera still generates many folds more revenue because the price of one Nikon D850 DSLR Camera is more than 20x the price of the Razer Huntsman Gaming Keyboard. It would be interesting to compare these results with the non-discounted products and see how that reflects on the sales revenue generated. The query for that is the same except for one small change. The Boolean value will be revised to `FALSE` for the `p.is_on_discount` column:

```

SELECT p.product_id, p.product_name, p.product_price,
SUM(o.quantity) as total_sold,
SUM(o.quantity * p.product_price) as total_revenue
FROM product p
JOIN orders o ON p.product_id = o.product_id_fk
WHERE p.is_on_discount = FALSE
GROUP BY p.product_id, p.product_name, p.discounted_price

```

```
ORDER BY total_sold DESC, total_revenue DESC;
```

Overall, non-discounted products do pretty well in sales and revenue, as shown in the following screenshot:

product_id	product_name	product_price	total_sold	total_revenue
16	PlayStation 5	500.00	54	27000.00
15	Casio G-Shock Watch	200.00	46	9200.00
35	Levi's Men's Jeans	60.00	45	2700.00
11	Mobil 1 Engine Oil	50.00	44	2200.00
27	Taylor Acoustic Guitar	1200.00	40	48000.00
19	Nike Women's Running Shoes	100.00	37	3700.00
12	Staples Office Chair	180.00	36	6480.00
43	Google Pixel 7	600.00	24	14400.00
38	Sony WH-1000XM5 Headphones	350.00	19	6650.00
80	Nintendo Switch Pro Controller	70.00	18	1260.00
45	Adidas Men's Soccer Shoes	85.00	12	1020.00
4	Adidas Running Shoes	80.00	11	880.00
10	Fitbit Charge 4	120.00	5	600.00
54	Apple iPad Pro	1000.00	2	2000.00

14 rows in set (0.00 sec)

Figure 6.13: Query result of total sales and revenue generated from the non-discounted products

Looking closer, the number of items sold and the total revenue generated seem higher for non-discounted products. Now, this doesn't paint a complete picture or even implies a correlation because there are other factors to consider, such as the total number of discounted and non-discounted products, but it's an indication that Umar should dive into this.

Identifying the top customers

This leads us to an exciting and challenging question which is important for ECommerceHub: what are the top customers? It can be vital to identify the top customers and provide unique benefits to them to ensure they keep buying from you and show them appreciation. This way, they could even turn into ambassadors. In short, Umar wants to advertise a loyalty program for the store's top customers but he wants to know which customers to target for this program. Here's the query to find this information:

```
SELECT customer_id_fk, CONCAT(c.first_name, ' ', c.last_name) AS CustomerName, SUM(total_cost) as total_spent
FROM orders o
```

```

JOIN customer c
ON o.customer_id_fk = c.customer_id
GROUP BY customer_id_fk
HAVING total_spent >= 5000
ORDER BY total_spent DESC;

```

In this query, a `JOIN` must fetch each customer's name corresponding to the `id`, which we can retrieve from the `orders` table. But to present it in a way that makes more *sense*, we use a `CONCAT` function to return the full name of each customer alongside their respective `customer_id` values. The `SUM` of the `total_cost` column returns the amount spent by each customer on transactions in the store. The results in the following screenshot are ordered based on the total amount (in descending order) spent on purchases, grouped by their ids, and filtered using the `HAVING` clause so only customers who have spent more than 5000 on purchases are returned in the query results.

customer_id_fk	CustomerName	total_spent
69	Carter Reed	44200.00
56	Emily Garcia	30000.00
85	Akiko Tanaka	21600.00
39	Chloe Scott	8000.00
88	Oscar Lopez	7700.00
70	Evelyn Ward	7400.00
57	Ethan Rodriguez	6780.00
30	Alejandro Perez	6580.00
102	Mason Nguyen	6400.00
43	Amelia Hughes	5000.00
48	Michael Baker	5000.00
5	Ali Khan	5000.00

12 rows in set (0.01 sec)

Figure 6.14: Top customers based on the total amount spent

ADD

Top-selling clothing products

Do you remember when we found the product line for the `Clothing` category? We are now ready to dive deeper into it and expand on that to show more meaningful information. We're going to combine the information obtained from the `Clothing` category, with the `orders` table and the `product` table:

```
SELECT p.product_id, p.product_name, p.description, p.product_price, p.discounted_price, SUM(o.quantity) AS sales
FROM product p
JOIN product_category c ON p.category_id_fk = c.category_id
JOIN orders o ON p.product_id = o.product_id_fk
WHERE c.category_name = 'Clothing'
GROUP BY p.product_id, p.product_name, p.description, p.product_price, p.discounted_price
ORDER BY sales DESC;
```

In the preceding query, three tables are joined together. The `product_category` table is only used to filter the query results on `Clothing`. You can see this in the following results. No data is fetched from the `product_category` table.

product_id	product_name	description	product_price	discounted_price	sales
35	Levi's Men's Jeans	Classic and comfortable men's jeans	60.00	60.00	45
19	Nike Women's Running Shoes	Durable and comfortable running shoes	100.00	100.00	37
45	Adidas Men's Soccer Shoes	Durable and comfortable soccer shoes	85.00	85.00	12
4	Adidas Running Shoes	Comfortable and stylish running shoes	80.00	80.00	11

Figure 6.15: Top-selling clothing products in the store

The `orders` table is used in the preceding query because we need to find the sum of sales for each respective product. It's striking that none of the products returned by this query are discounted. Perhaps Umar can continue to research why that is. In the meantime, he asked us to inspect the top-performing products. Let's have a look!

Top 5 high-performing products

Looking at the results of the last query has made Umar really curious about

the top 5 products in any category in terms of generating revenue. Moreover, it presents an opportunity to highlight these products even more in the store with targeted advertising and brand exposure. This is the query to do that:

```
SELECT p.product_name, p.product_id, c.category_id, c.category_name, SUM(o.total_cost) AS Total_Revenue, SUM(o.quantity) AS Total_Sales
FROM product p
JOIN product_category c ON p.category_id_fk = c.category_id
JOIN orders o ON p.product_id = o.product_id_fk
GROUP BY p.product_name
ORDER BY Total_Revenue DESC, Total_Sales DESC
LIMIT 5;
```

This query also uses two `JOINS` to link three tables, as it's important to fetch product, category and sales information to return the columns shown in the following screenshot:

product_name	product_id	category_id	category_name	Total_Revenue	Total_Sales
Taylor Acoustic Guitar	27	17	Music & Instruments	48000.00	40
Nikon D850 DSLR Camera	44	1	Electronics	39200.00	14
PlayStation 5	16	19	Computers & Accessories	27000.00	54
Google Pixel 7	43	1	Electronics	14400.00	24
Oculus Quest 2	21	1	Electronics	11100.00	37

5 rows in set (0.01 sec)

Figure 6.16: Top 5 best-selling products and their details, including total revenue and sales

The preceding query selects the product names, category names, and ids, the sum of the `total_cost` column, and the `quantity` column to give revenue and the number of orders, respectively. These results are grouped by each unique combination of product name and id, and ordered in descending order with respect to both total revenue and total sales. The results are limited to 5 rows because Umar only wants information for the top 5 selling products. You could enhance the result of the last query by adding information about the categories to understand which of the categories these top products belong to. In terms of business context, this serves as a key point to better highlight these categories or take the alternative and focus on low-selling categories and products instead to improve their performance. The updated query will then look like:

```

SELECT p.product_name, p.product_id, c.category_id, c.category_name, SUM(o.total_cost) AS Total_Revenue, SUM(o.quantity) AS Total_Sales
FROM product p
JOIN product_category c ON p.category_id_fk = c.category_id
JOIN orders o ON p.product_id = o.product_id_fk
GROUP BY p.product_name, p.product_id, c.category_id, c.category_name
ORDER BY Total_Revenue DESC, Total_Sales DESC
LIMIT 5;

```

The preceding query expands on the previous one, selects the `category_id` and `category_name`, and uses a subquery to return the sales concerning each product category:

product_name	product_id	category_id	category_name	Total_Revenue	Total_Sales
Taylor Acoustic Guitar	27	17	Music & Instruments	48000.00	40
Nikon D850 DSLR Camera	44	1	Electronics	39200.00	14
PlayStation 5	16	19	Computers & Accessories	27000.00	54
Google Pixel 7	43	1	Electronics	14400.00	24
Oculus Quest 2	21	1	Electronics	11100.00	37

5 rows in set (0.01 sec)

Figure 6.17: Top 5 best-selling products with their categories and number of items sold

Next let's tackle a slightly more complex problem related to the most popular products.

Most popular product by country

Since this is an international e-commerce store, it makes sense to find the sales and the most popular products in each country based on the total quantities sold. Doing so isn't easy and requires a more comprehensive grasp of the SQL techniques that we have learned so far, using multiple CTEs, JOINs and subqueries:

```

WITH country_product_sales AS (
    SELECT s.country, p.product_name, SUM(o.quantity) AS total_quantity
    FROM orders o
    JOIN shipping_information s ON s.orders_id_fk = o.orders_id
    JOIN product p ON p.product_id = o.product_id_fk

```

```
        GROUP BY s.country, p.product_name
)
SELECT cps.country, cps.product_name, cps.total_quantity
FROM country_product_sales cps
RIGHT JOIN (
    SELECT s.country, MAX(cps.total_quantity) AS max_quantity
    FROM country_product_sales cps
    JOIN shipping_information s ON s.country = cps.country
    GROUP BY s.country
) AS max_qty ON cps.country = max_qty.country AND cps.total_quantity = max_qty.max_quantity
ORDER BY cps.country;
```

Let's break down this complex query into parts. The CTE (`WITH` statement) creates a temporary table to hold the records for the country names, product names and their total sales using the `SUM` function on the `total_quantity` column. These results are grouped by both country and product names. Then, in the `SELECT` query, the CTE table (created above) is used to fetch the data. Additionally, a subquery selects the maximum number of items sold for each country and product, and these results are ordered in ascending order for the country names. It's a great sign for Umar to see such a diverse “heat map” of customers for the international store:

country	product_name	total_quantity
Australia	Razer Huntsman Gaming Keyboard	24
Brazil	PlayStation 5	1
China	Levi's Men's Jeans	17
Egypt	GoPro HERO9	3
France	iRobot Roomba 675 Robot Vacuum	9
Germany	Mobil 1 Engine Oil	10
Hong Kong	Staples Office Chair	15
Italy	Nike Women's Running Shoes	8
Japan	Taylor Acoustic Guitar	18
Mexico	Samsung Galaxy S23	11
New Zealand	PlayStation 5	16
Pakistan	Casio G-Shock Watch	13
Poland	Razer Huntsman Gaming Keyboard	9
Portugal	Mobil 1 Engine Oil	13
Russia	Oculus Quest 2	7
South Korea	Levi's Men's Jeans	10
Spain	Razer Huntsman Gaming Keyboard	7
Sweden	Oster Blender	12
United Kingdom	Nike Women's Running Shoes	14
United States	Google Pixel 7	21
Vietnam	GoPro HERO9	16

21 rows in set (0.00 sec)

Figure 6.18: Best-selling product for each country the products have been sold in

ADDResearching the performance of delivery

The reputation of an e-commerce business often depends on its delivery commitment and the arrival of the products. As such, Umar wants to know if the store has been delivering the orders on time. If not, he can improve it to ensure the customers keep ordering and build a stable market reputation.

ADD

```
SELECT AVG(
    DATEDIFF(actual_delivery, expected_delivery)
) AS Late_Delivery_Days
FROM shipping_information;
```

The `DATEDIFF` date-time function calculates the difference between one

order's expected and actual delivery dates. We use the `AVG` function for nesting the `DATEDIFF` function to calculate the average delay in delivery time for all orders. As seen in the following screenshot, the average delay time amounts to 14.31 days, which is not a positive indicator for the business:

Late_Delivery_Days
14.3146

1 row in set (0.00 sec)

Figure 6.19: Average late delivery time for each shipment

This business model will not sustain revenue as it will lead to many disgruntled customers if they order something urgent and it arrives, on average, 2 weeks later. This way, they will not be willing to make more purchases from the store. Naturally, it begs the question, has this affected the sales revenue generated by the store in the last year then? Not only does it help ascertain the effect of the late deliveries, but it also helps analyze the bigger picture of store performance in the last year. Here is the query to find out:

```
SELECT YEAR(order_date) as year, MONTH(order_date) as month, SUM(total_cost) as monthly_sales
FROM orders
WHERE order_date >= DATE_SUB(CURRENT_DATE, INTERVAL 1 YEAR)
GROUP BY year, month
ORDER BY year DESC, month DESC;
```

Historical data is important when you're analyzing sales as it provides contextual information used to extract patterns or project future sales. The

preceding query relies on date-time and aggregate functions. The `YEAR` obtains the year from a date, `MONTH` extracts the month, `SUM` finds the monthly sales, and the `DATE_SUB` function sets the interval by subtracting one year from the current date.

year	month	monthly_sales
2023	5	218.85
2023	4	1890.00
2023	2	2600.00
2023	1	350.00
2022	12	300.00
2022	11	450.00
2022	9	500.00
2022	8	2250.00
2022	7	2100.00
2022	6	2600.00
2022	4	90.00
2022	3	1020.00
2022	2	1600.00
2022	1	39200.00
2021	11	600.00
2021	9	7160.00
2021	7	5000.00
2021	6	900.00
2021	5	8000.00
2021	4	1480.00
2021	3	21600.00
2021	2	70.00
2021	1	5600.00
2020	12	1080.00
2020	10	1020.00
2020	9	2520.00
2020	8	6400.00
2020	7	1800.00
2020	6	800.00

Figure 6.20: Monthly sales for each month in the last three years, excluding months without sales

Based on the result shown in the preceding screenshot, Umar is curious to learn about the customers who haven't purchased any products in the last year. Figuring this out would help Umar better target this audience.

```
SELECT customer_id FROM customer
WHERE customer_id NOT IN
(
    SELECT DISTINCT customer_id_fk
    FROM orders
    WHERE order_date > DATE_SUB(CURRENT_DATE, INTERVAL 1 YEAR)
);
```

The result of the preceding query returns more than 100 rows. It highlights the fact that ECommerceHub has lost many of its customers, and only a few remain. Why could this be? We have already figured out one possible reason, long delivery delay times. Considering all this information, can you identify future sales projections in the next 6 months? Let's see how this can be done.

Future projections with linear regression

Yes, we can make future sales projections too. We'll do so with the use of a simple linear regression model. We can analyze and project the revenue for the next 6 months to help Umar decide whether to shut down the store. This query follows a simple linear regression approach which finds the total revenue generated in the last 6 months and multiplies it by 6 to project for the coming 6 months:

```
SELECT (
    SELECT SUM(total_cost)
    FROM orders
    WHERE order_date BETWEEN
        DATE_SUB(CURRENT_DATE, INTERVAL 6 MONTH)
        AND CURRENT_DATE)
    * 6 as projected_revenue;
```

The inner query in the `SELECT` clause executes first, given that it's a subquery. The subquery finds the sum of all rows to find the total cost where the `order_date` is in the last 6 months, as displayed in the following

screenshot:

projected_revenue
32153.10

1 row in set (0.04 sec)

Figure 6.21: Project revenue for the next 6 months

The `DATE_SUB` function is a function we haven't discussed before. What does it do? It does exactly what it implies: subtracting a specific interval from a date. In this example, we subtract 6 months from the `current_date`. The result of this query is subsequently (not with subquery) multiplied by 6. How does this help Umar? The projected revenue helps him see what the near future might hold for the store. So, it is up to Umar to digest all this information we provided and conclude. Umar Ali is grateful for your help with ECommerceHub!

Summary

In this chapter, we helped Umar with performing data analysis on his ECommerHub store data. We used the newly acquired SQL skills we obtained in the last few chapters to solve a hypothetical real-world example of data analysis with MSQL. We started by importing the data into our database, and after that, we explored the different tables and their structure. Next, Umar took us by the hand asking for insights in his ECommerceHub dataset. Now that you have worked your way through this chapter, you should have an idea of how to use your SQL skills to perform data analysis on a

project that resembles real-world examples. You could come across a case like this in your career. Right now, you will have gained some confidence in solving real-world data analysis projects in SQL. Let's round up this pillar of data analysis before moving on to the *Data Cleaning & Exploratory Data Analysis* pillar.

7 Fundamental Statistical Concepts

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



Statistics is the science of collecting, analyzing, interpreting, and presenting data. Data Analysts must be able to extract valuable and meaningful patterns from data. This is part of the process of uncovering insights. But, to accomplish this goal, you must be able to accurately measure and interpret the data you see. This is where the knowledge of the fundamentals of statistics comes into play. In this chapter, we will focus on the important concepts of statistics that data analysts should absolutely know about. This chapter will equip you with the necessary tools to confidently tackle data analysis tasks to make informed decisions and recommendations. We will cover the following:

- Introduction to descriptive and inferential statistics.
- Measures of central tendency and variability.
- Introduction to probability distributions

Descriptive statistics

Descriptive statistics provide a summary of historical data or what happened in a business process. The ability to uncover accurate insights comes from knowing how to properly measure and interpret your data using statistics. Creating metrics, KPIs, or OKRs are also born out of your descriptive

analysis. In this section, we will cover the measurements of scale and the measures of central tendency variability.

Levels of measurement

Data can be categorized as **qualitative** or **quantitative**. Qualitative data refers to information that is descriptive in nature about qualities. They cannot be directly measured and are normally categories or attributes. Examples include:

- Marital status: Married, single, or divorced.
- Job title: Manager, data Analyst, or director
- Customer satisfaction: Low, medium, or high.

Qualitative data can be further categorized as **nominal** or **ordinal**. Nominal data is used only for the means of classification or grouping data, such as gender or race. There is no order or ranking system, whereas ordinal data does have a ranking system or order. Examples include letter grades (A, B, C, D, and F).

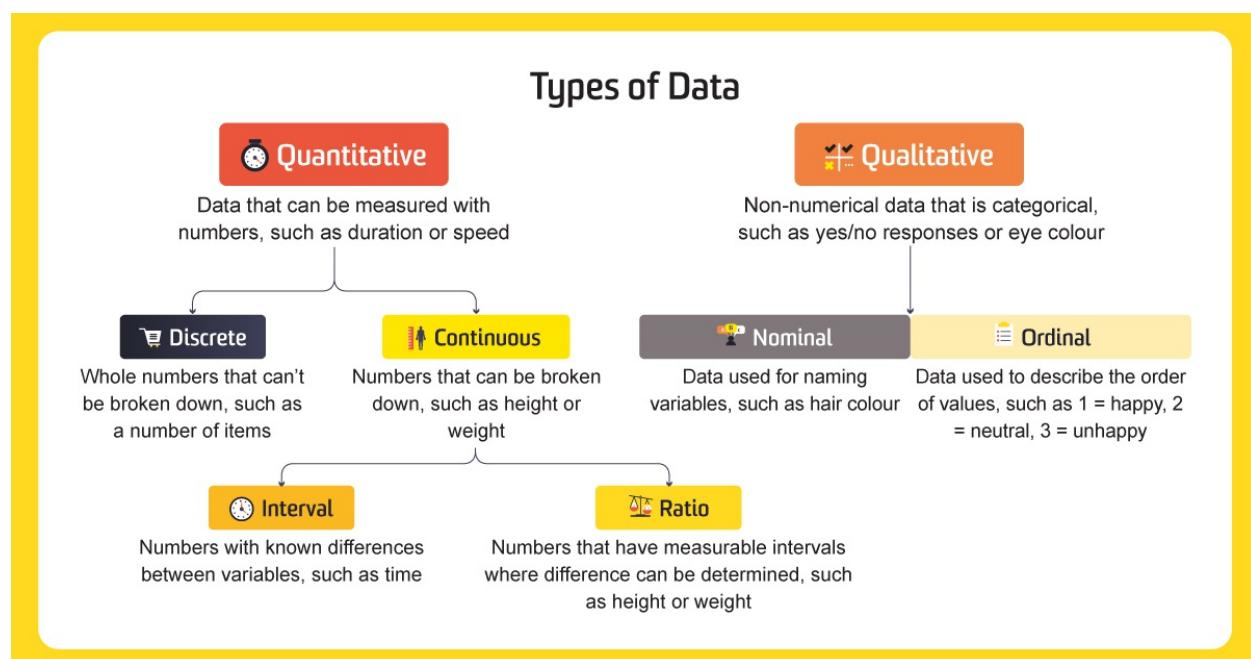


Figure 1. Levels and scales of measurement

Quantitative data is numeric in nature and can be measured or counted. You can also perform direct calculations such as addition, subtraction, etc. Data that is numeric, but not quantitative would be unique numeric IDs. Examples of quantitative data includes:

- Age: 25, 35, 45
- Salary: \$120,000, \$98,000, \$230,000
- Test scores: 89, 86, 92

Quantitative data can be further characterized into **discrete** or **continuous**. Discrete values can only be reported in whole numbers. Examples include the number of siblings you may have: 0, 1, 2, etc. You cannot have 1.3 siblings. Continuous are values that can be reported as decimals which allows for a high level of precision during measurement. Examples include height measurements: 65 inches, 65.7 inches, etc. Continuous data can further be categorized as **interval** or **ratio**. Interval data contains values in a given range and have no true zero. Examples include IQ scores and temperatures. For example, an IQ score of 0 does not mean a person has no intelligence, but rather a very low intelligence. Ratio data is like interval data but can possess a true 0 value where it means an absence of something. Examples include income and weight. For example, someone can truly have no income at all. ADD

Measures of central tendency

Measures of central tendency describe the typical value of your dataset or where majority of your data is distributed. Another term for measures of central tendency is measures of central location. Data analysts must have a strong understanding of these terms due to their significance in summarizing and interpreting large amounts of data. The following are three most common measures of central tendency: Summary statistics are values that help describe where your data is mostly located.

Average (Arithmetic Mean)

The average is calculated by the sum of all values divided by the number of observations in the dataset. While the average is a commonly used statistic, it

is highly sensitive to **outliers** (abnormally high or low values in the dataset). Outliers can be from the result of errors or natural events. If your dataset has a lot of outliers influencing the average, it is best to report the median (discussed next). If you determine that the outliers are the result of errors, you can remove them in your calculation and then report the average. Advantages of the Average

- Utilizes every value in the dataset.
- Widely popular and can be reported without much explanation.

Disadvantages of the Average

- Heavily influenced by outliers. If your dataset has many outliers, your mean can be a poor representation of where most of your data is located unless you perform outlier handling.
- Not best to use with a limited dataset or a skewed dataset, as it will not accurately represent where most of the data is located.

Data analyst's use case for the average: A data analyst can use the mean to calculate the average sales revenue per month for a company. By analyzing the average revenue, the analyst can identify trends, patterns, and seasonality in sales, allowing them to make informed decisions regarding inventory management, marketing strategies, and financial forecasting. Additionally, the average revenue can be compared to industry benchmarks or historical data to evaluate the company's performance relative to its competitors or past performance.

Median

A median is the middle number in an ordered dataset. If there is an even number of data points, the median is the average of the two middle numbers. Unlike the mean, the median is considered a robust measurement as it's not influenced by outliers. Advantages of the median:

- also known as a robust statistic, meaning that it is not heavily influenced by the presence of outliers
- Can be used in small or large datasets, and skewed datasets

Disadvantages of the median

- Does not take all the collected data into account in the calculation, and only uses a small portion of the dataset

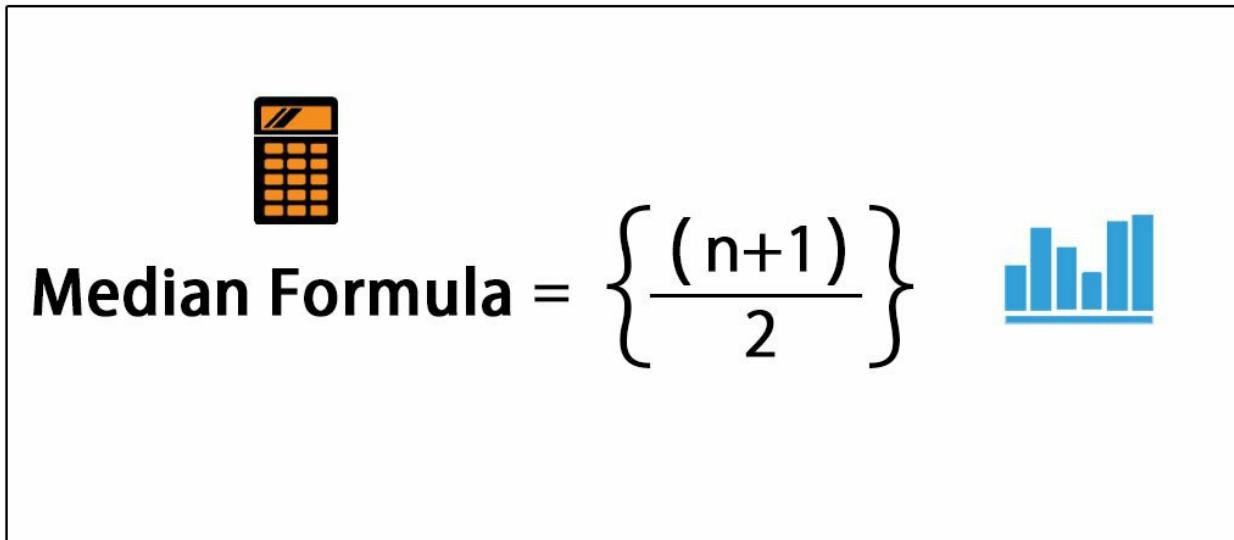


Figure 2 - This picture displays the equation to calculate the median of a dataset

Data analyst's use case for the median: A data analyst is examining the salaries of employees in a large organization. Instead of using the average, which can be heavily influenced by outliers or extreme values, the analyst calculates the median salary. This measure provides a better representation of the central tendency of the salary distribution, making it useful for understanding the typical earning potential within the organization.

Mode

The mode is the number that occurs the most often in a dataset. This measure is often used for summarizing categorical data. Advantages of the mode

- No calculation involved.
- Not influenced by outliers

Disadvantages of the mode

- A dataset may not have a mode or may have multiple modes
- Weaker measure than the mean and median

Data analyst's use case for the mode: For an e-commerce platform, the mode can be used to identify the most frequently purchased product category. This information can be leveraged to optimize inventory management, marketing strategies, and product recommendations to enhance customer satisfaction and maximize sales. By identifying the mode, the data analyst can understand customer preferences and tailor the platform's offerings accordingly. This insight can drive strategic decisions, such as prioritizing stock availability and promoting popular product categories to increase customer engagement and drive revenue. The following are less common and variations of the previous measures:

Weighted mean

This measure is calculated by assigning different weights to different values. The weighted mean is particularly valuable in situations where different groups have different levels of influence or importance within a dataset. Advantages of the weighted mean:

- **Reflects importance:** All values in a business process may not carry the same importance or priority. Weighted means allows a data analyst to emphasize the business significance of certain observations.
- **Imbalanced data:** Weighted means accounts for imbalances in your data as one can add more significance to underrepresented groups to decrease bias.

Disadvantages of the weighted mean:

- **Subjectivity:** Assigning weights to certain observations can be challenging and can also result in some subjectivity. Bias can also result from this measure. This calculation should be used with ethics and the business process in mind.
- **Outliers:** The weight assignments can make this measure more sensitive to outliers than the normal mean. Especially if the outlier observations are weighted.

- **Challenges in interpretability:** Unlike the arithmetic mean, the weighted mean can be complicated to interpret on its own. It should be reported with an explanation of how the measure was calculated. Stating the weights for each observation or group and why.

Data Analyst's use case for the weighted mean: A data analyst in the education sector can use the weighted mean to calculate the average test scores of students, considering the weightage assigned to different sections of the exam. This measure allows for a more accurate representation of student performance by accounting for variations in the importance of different test components. For example, if certain sections of the exam carry higher weights, the weighted mean provides a more comprehensive understanding of students' overall mastery of the subject. By analyzing the weighted mean, educators can identify areas of strength and weakness, design targeted interventions, and provide personalized feedback to improve student learning outcomes.

Trimmed mean

The trimmed mean is a variation of the arithmetic mean, but with a percentage of the extreme values or outliers removed on both ends of the dataset. The trimming is a method to report the average while removing the influencing outliers. Advantages of the trimmed mean:

- **Robust to outliers:** Because the extreme values are cut off from the calculation, this measure is not sensitive to the presence of outliers. If outliers are determined to be errors or nonrepresentative of the business process, this measure would be preferred.

Disadvantages of the trimmed mean:

- **Loss of data:** Deleting collecting data must be done with caution and is not always a preferred action. Important information may be lost.
- **Challenges in interpretability:** Additional explanation of how the calculation was performed may be needed to avoid confusion or misrepresentation of the data.
- **Subjectivity:** There is no hard rule on the percentage to remove in the

calculation. This can lead to bias and should be done with careful consideration of ethics and relation to the business process.

Data analyst's use case for the trimmed mean: In sentiment analysis of customer reviews, the trimmed mean can be used to analyze ratings by removing extreme outliers. This measure provides a robust estimate of the overall sentiment expressed in the reviews while mitigating the impact of outliers that may skew the results. By trimming the extreme values, the data analyst can focus on the general sentiment of customers and identify trends and patterns in their feedback. This helps businesses gain valuable insights into customer satisfaction, product improvements, and areas that require attention, enabling them to enhance their offerings and make data-driven decisions based on reliable sentiment analysis.ADD

Measures of variability

Measures of variability describe the spread, dispersion, or variability of the data points in a dataset. These measures summarize how the data points are distributed around the previously mentioned measures of central tendency (mean, median, mode, etc.) They allow data analysts to assess the reliability of results, identify outliers to investigate, and manage uncertainty. The following are common measures of variability that data analysts should know:

- **Range:** This measure is the difference between the largest and smallest numbers in a dataset. Data analysts can use this number to get a quick understanding of the spread of the data. A use case would include quality control to assess the variability of product measurements to ensure consistency.

Data analyst's use case for range: A data analyst working in quality control for a manufacturing company can use the range to assess the variability in product dimensions. By calculating the difference between the maximum and minimum values, the analyst can determine the acceptable range within which products must fall to meet quality standards. This also helps identify any deviations from specifications and assists in identifying potential issues in the manufacturing process or material quality. By monitoring the range

over time, the analyst can track process improvements and ensure consistency in product quality.

- **Interquartile Range (IQR):** The range between the first and third quartiles. Also known as the 25th and 75th quartiles. Best visually represented in a box plot as pictured below.

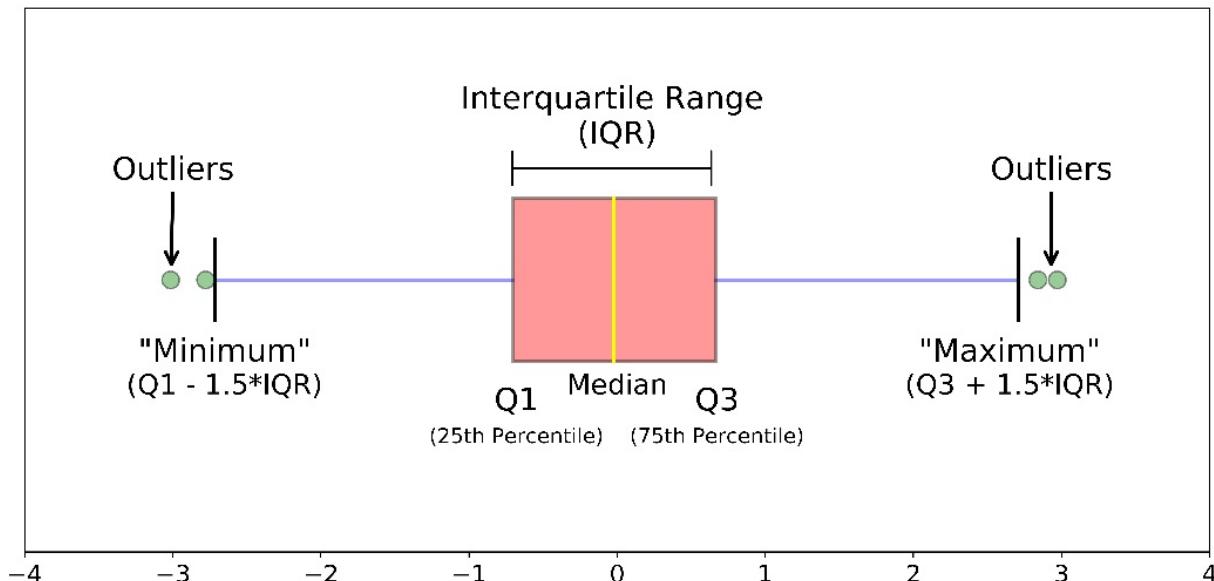


Figure 3. This photo displays a box plot labeled with each quartile with outliers displayed at the ends.

Data analyst's use case for IQR: A data analyst in a retail company is analyzing sales performance across different product categories and stores. To understand variability and identify outliers, the analyst calculates the IQR for each category. The IQR provides a measure of data spread, helping identify categories with consistent sales and those with higher variability. It also assists in detecting potential outliers, enabling further investigation and targeted interventions for improved performance and decision-making in inventory management, pricing, promotions, and product assortment.

- **Variance:** Measures how far data points are from the mean. It is calculated by averaging the squared differences from the mean.

Data Analyst's use case for variance: In financial portfolio analysis, the

variance can be used to measure the volatility of different investments. By quantifying the dispersion of returns around the mean, the analyst can compare and assess the risk associated with various investment options. A higher variance indicates higher volatility and potential fluctuations in investment returns, while a lower variance implies greater stability. Portfolio managers and investors can utilize variance to optimize their investment strategies, diversify their portfolios, and balance risk and return based on their risk tolerance and investment objectives.

- **Standard Deviation:** This is the most common measure used to describe variation. This measures the average distance between each data point and the mean. Use cases involve measuring the volatility or fluctuations of an investment portfolio. Additionally, this measure is the square root of the variance.

Data analyst's use case for standard deviation: In market research, the standard deviation can be used to measure the variability in customer ratings for a product or service. This helps identify the consistency of customer perceptions, enabling businesses to focus on areas that need improvement. A higher standard deviation indicates greater variability in ratings, suggesting that customer opinions are more dispersed. With this analysis, a business can pinpoint specific features or aspects of their offerings that contribute to customer satisfaction or dissatisfaction. This information helps prioritize product enhancements, customer service improvements, and targeted marketing campaigns to address customer needs and preferences effectively.

- **Skewness:** Refers to the measurement of distortion in the distribution of a dataset. In the presence of outliers, skewness will be most apparent either being negatively or positively skewed.

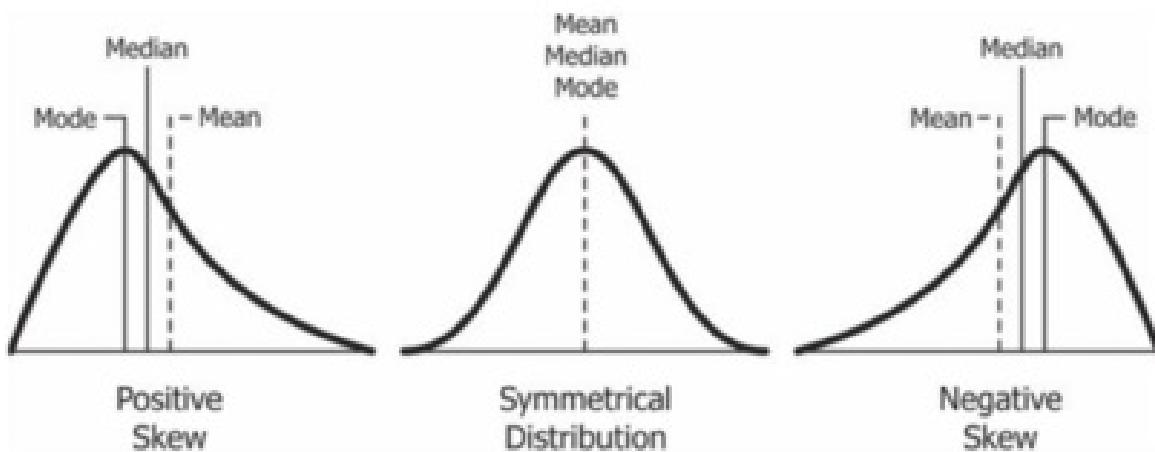


Figure 4. This photo displays the different types of skews along with a normal distribution for comparison.

Data Analyst's use case for skewness: A data analyst is studying the distribution of customer ages in a retail database. By calculating the skewness of the age distribution, the analyst can identify if the data is skewed towards younger or older customers. This measure provides insights into customer demographics, enabling targeted marketing campaigns or the development of age-specific products.

- **Outliers:** These are extreme values that are very different from the majority of the dataset. There is always a story behind an outlier. They can arise due to various reasons including measurement errors, data entry errors, or part of the natural business process. Because there is variation in every process, the presence of outliers can be expected.

Data analyst's use case for outliers: A data analyst is working for a transportation company and is analyzing the fuel efficiency of a fleet of vehicles. As part of the analysis, the analyst wants to identify any outliers in the fuel efficiency data that may indicate unusual or erroneous measurements. By examining the dataset and applying outlier detection techniques, the analyst can pinpoint vehicles that have exceptionally high or low fuel efficiency compared to most of the fleet. These outliers may represent vehicles with mechanical issues, measurement errors, or other factors influencing their fuel efficiency. And with that, we have explored the world of descriptive statistics, where we learned how to summarize and analyze data

using various measures of central tendency and variation. Descriptive statistics provided us with a comprehensive understanding of the characteristics of a dataset. However, our analytics job doesn't stop there. We will now explore the powers of inferential statistics to support more advanced decision making.

Inferential statistics

Inferential statistics allow us to go beyond the specific dataset at hand and generalize or make conclusions. In this section we will begin by introducing the concepts of probability, then dive into sampling, estimation, and end with the concept of correlation vs causation.

Probability theory

Probability refers to the likelihood of a particular event occurring. It provides a framework for quantifying uncertainty and making informed decisions in the face of randomness. Whether you're predicting customer behavior, analyzing stock market trends, or evaluating the effectiveness of a marketing campaign, understanding probability is essential for accurate and reliable data analysis. To reiterate, probability deals with the likelihood of events occurring. By assigning numerical values to these likelihoods, we can express them quantitatively, enabling us to make probabilistic statements and predictions. We will explore basic concepts and the different kinds of distributions that allow us to support more advanced decision making. Data analysts can conduct **experiments** to draw more advanced and meaningful insights from data. It represents any controlled and repeatable process that generates outcomes. Every outcome from an experiment is represented as an **event**. The collection of all events from an experiment is called a **sample space**. Experiments allow a data analyst to gather data, validate assumptions, and make inferences. Allowing the evaluation of different features, assessing effectiveness of different strategies, and exploring the strength of relationships between variables. The following are examples of use cases where a data analysts can perform experiments:

1. **A/B testing in marketing:** Data analysts conduct A/B tests to determine the probability of an event occurring, such as the likelihood of a user

clicking on an advertisement or making a purchase. By randomly assigning users to different groups (A and B) and exposing each group to different variations of a marketing campaign, analysts can measure the response rates and calculate the probability of specific events, such as click-through rates or conversion rates. This information helps optimize marketing strategies and make data-driven decisions about campaign effectiveness.

2. **Clinical trials in healthcare:** In clinical trials, data analysts conduct experiments to determine the probability of a particular event or outcome, such as the effectiveness of a new drug in treating a specific condition. By randomly assigning participants to control and treatment groups, analysts can measure the occurrence of desired outcomes and calculate the probability of success. This information is crucial for evaluating treatment efficacy, determining side effects, and making informed decisions about patient care.
3. **Quality control in manufacturing:** Data analysts conduct experiments in manufacturing settings to define the probability of events related to product quality. For example, analysts may test samples of products to determine the probability of defects occurring during production. By conducting experiments and collecting data, analysts can calculate probabilities of specific events, such as the likelihood of a defective product being produced. This information helps manufacturers identify areas for improvement, optimize processes, and enhance overall product quality.
4. **Fraud detection in banking:** Data analysts conduct experiments to define the probability of fraudulent events occurring in banking transactions. By analyzing historical data and conducting experiments with known fraudulent patterns, analysts can develop models to calculate the probability of a transaction being fraudulent. This information helps identify suspicious activities, establish risk thresholds, and implement fraud detection systems to protect customers and financial institutions.

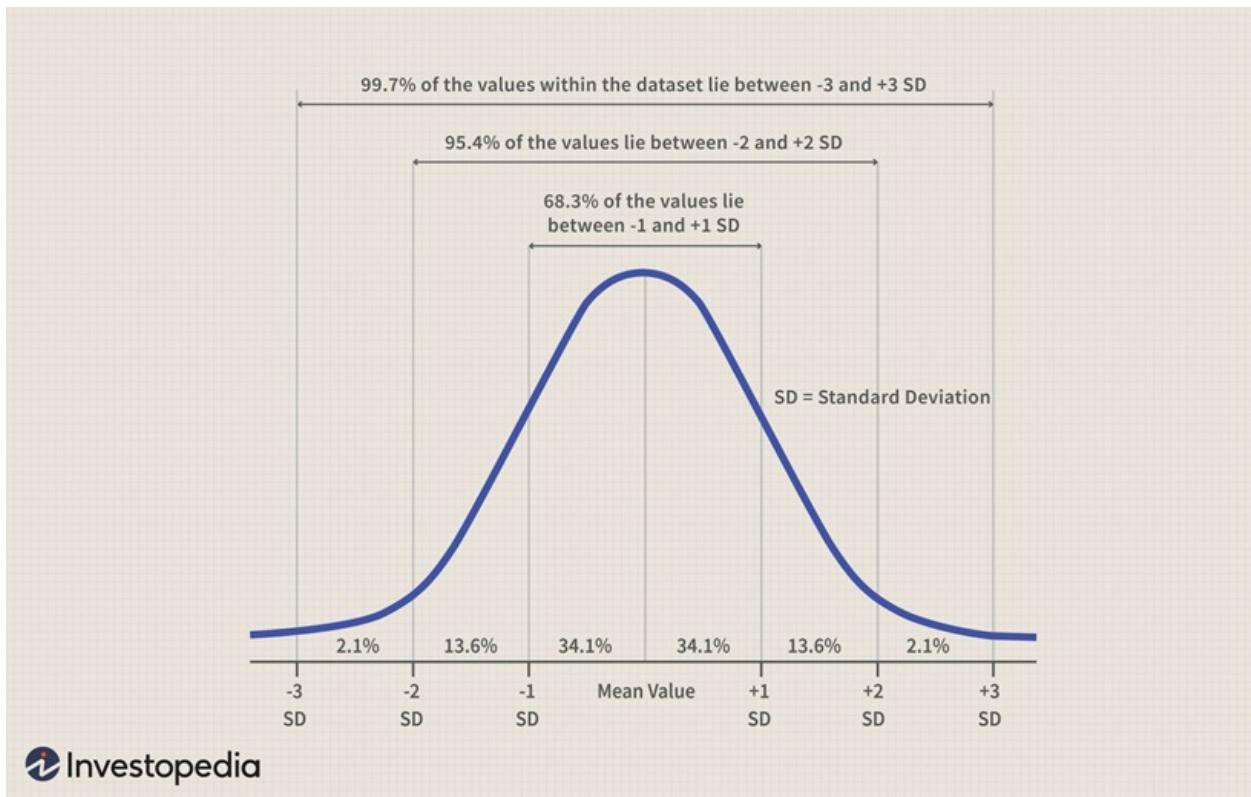
ADD

Probability distributions

Building upon the concepts of sample spaces and experiments, probability distributions enable us to study and quantify the likelihoods associated with various events and outcomes. By examining the probability distribution of a random variable, we gain insights into the likelihoods of different values it can take on. Here are the types of probability distributions:

- The **normal distribution**, also known as the Gaussian distribution, is one of the most important and widely used probability distributions. It is symmetric and bell-shaped, characterized by its mean and standard deviation. Many natural phenomena and measurement errors tend to follow a normal distribution. It is used for analyzing continuous data, such as heights, weights, test scores, or when sample sizes are large and we can make assumptions about the data being approximately normally distributed.

It follows the 68-95-99 rule where **68%** of the data lies between 1 standard deviation from the mean, **95%** of the data lies between 2 standard deviations from the mean, and **99.7%** of the data lies between 3 standard deviations from the mean.



 Investopedia

Figure 5 - A normal distribution with each section of the 68-95-99 rule labeled

- The **binomial distribution** models the number of successes in a fixed number of independent trials, where each trial has only two possible outcomes (success or failure). It is used when analyzing events with binary outcomes or proportions, such as the success rate of a marketing campaign, the pass/fail rates in quality control, or the probability of winning a game.

 *Figure 6 - Example of a binomial distribution*

- The **uniform distribution** represents a continuous random variable with a constant probability density function over a specific interval. It is often used when there is no specific bias or preference for any value within the given interval. The uniform distribution is commonly used in simulations, random number generation, and when there is equal likelihood for each value within a range.

Figure 7. Example of a uniform distribution

- The **exponential distribution** models the time between events occurring in a Poisson process. It is often used to model waiting times, lifetimes, or failure rates in systems where events occur randomly and independently over time. The exponential distribution is continuous and has a memoryless property, meaning that the probability of an event occurring is the same regardless of how much time has elapsed since the last event.

Figure 8 - Example of 3 exponential distributions with different rates

- The **log-normal distribution** is a continuous probability distribution where the logarithm of the random variable follows a normal distribution. It is often used to model variables that are naturally skewed and have positive values, such as stock prices, income data, or any variable that exhibits exponential growth. The log-normal distribution is commonly applied when data is expected to have a positive skew.

Figure 9. Example of a log normal distribution

- The **Student's t-distribution** is used when making inferences about the mean of a population when the sample size is small or when the population standard deviation is unknown. It is like the normal distribution but has heavier tails which means there's a larger possibility of very large values. It is commonly used in hypothesis testing and confidence interval estimation when dealing with small sample sizes. It provides more robust results when data does not meet the assumptions of normality or when the population standard deviation is unknown.

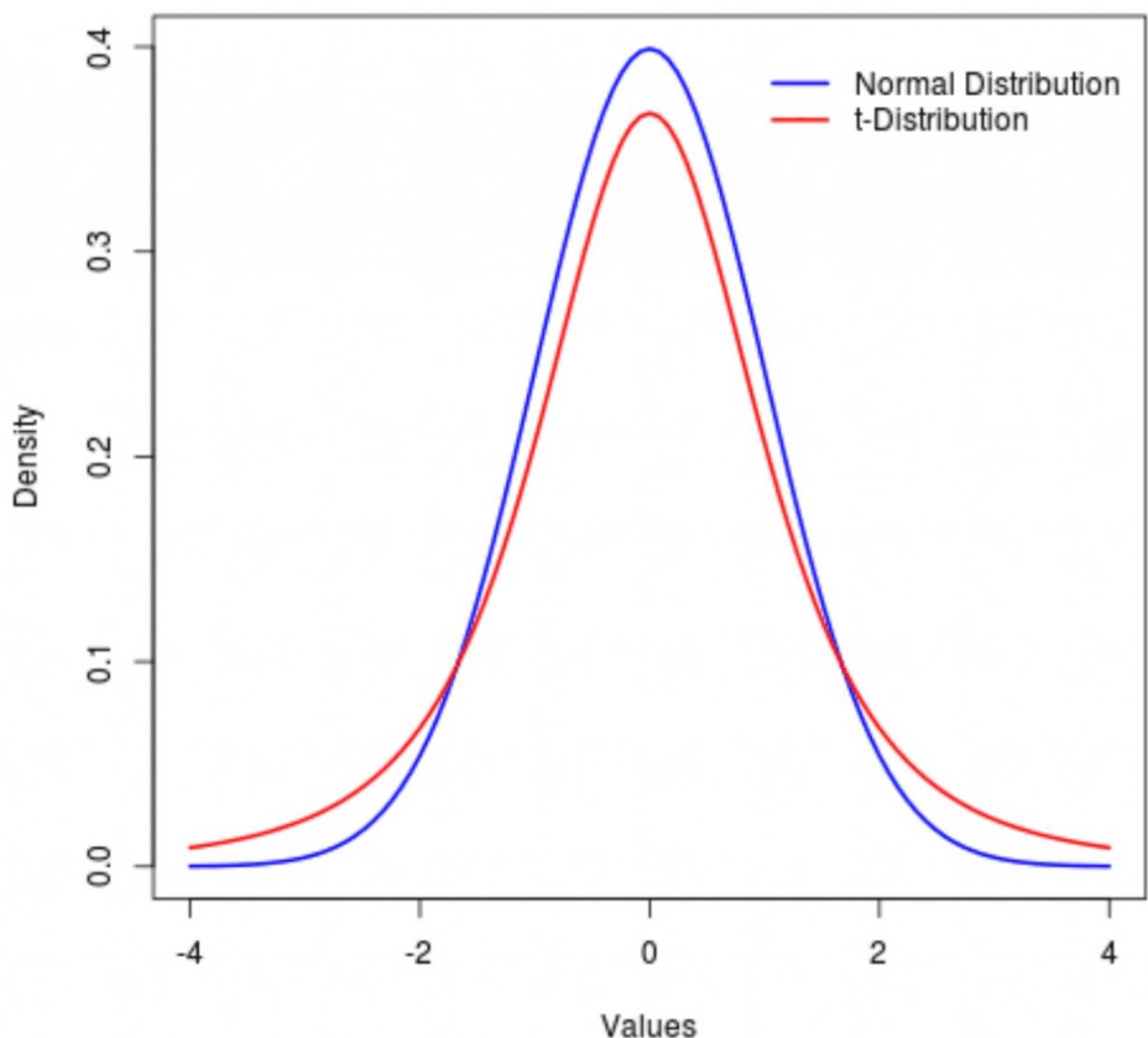


Figure 10. Example of a Student's t-distribution.

ADD

Correlation vs causation

Correlation refers to the coefficient that references the relationship between variables. The coefficient is represented as a number between -1 and 1. A mistake that can easily be made is to imply one event leads to / causes another event just because of a high correlation coefficient. Let's consider the following example:

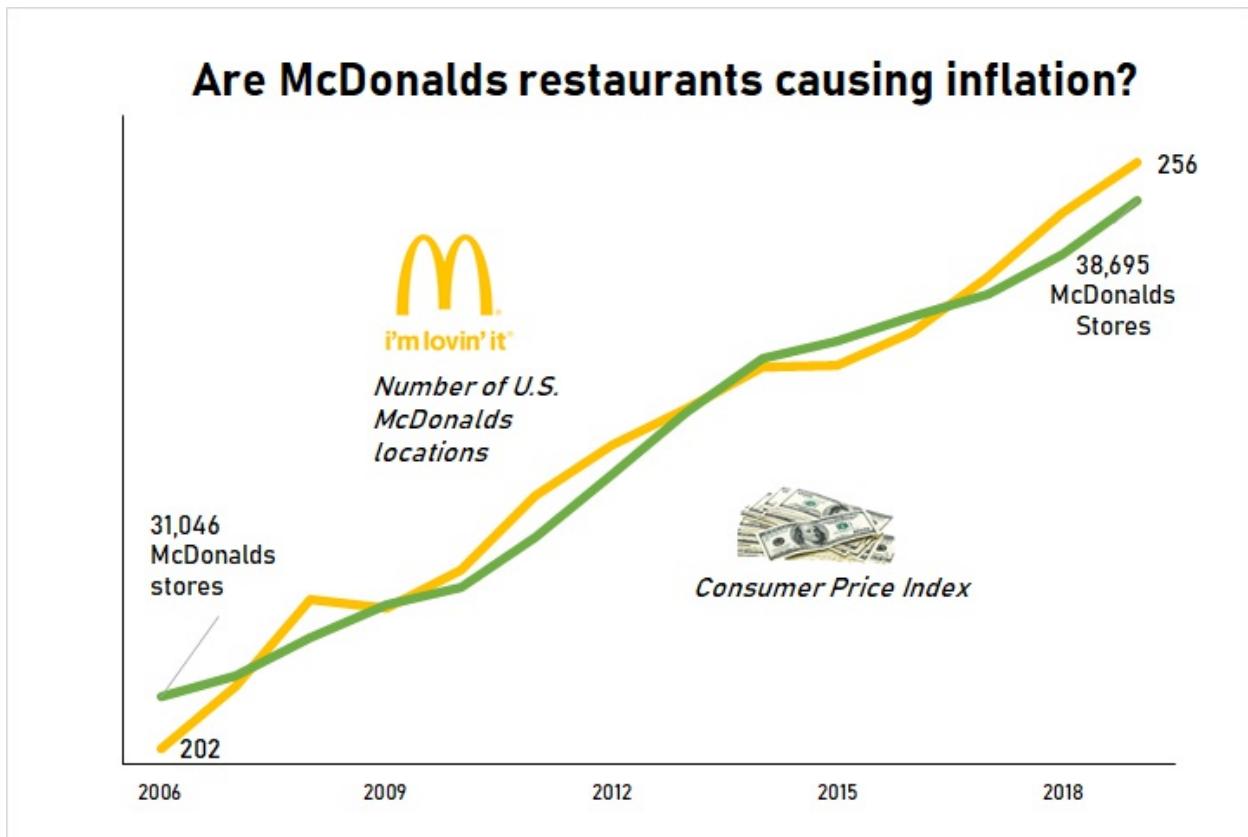


Figure 11. Chart showing a positive correlation between the number of McDonalds restaurants and the rise of inflation over time. Shows that two unrelated events can have a high correlation.

As pictured in the preceding graph, the number of McDonalds branches has a positive relationship with the increase in inflation. However, it would be an incomplete conclusion to state that either causes the other. There would be multiple other factors to investigate. This is another example where a data analysts need to consider of the business process to apply the context to the numbers. Numbers by themselves, don't tell the whole story.ADD

Summary

In this chapter, we introduced various descriptive and inferential statistical concepts that data analysts should know of. An in-depth understanding of descriptive statistics is necessary to perform an effective exploratory data analysis. While there are many more concepts and advanced terms, entry level data analysts do not need to know a lot of statistics beyond descriptive

and inferential. In the next chapter, we will build upon the concepts here to learn how to perform hypothesis testing. Hypothesis testing will allow a data analyst to conduct their own experiments allowing for deeper business insights.

8 Testing Hypotheses

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



Welcome to the world of hypothesis testing! Here, we investigate assumptions, challenge propositions, and seek statistical evidence to back our claims. It plays a pivotal role in data analysis, helping us make educated inferences based on a sample of data. We'll unpack the concept of hypothesis testing, discussing its role in data analysis and clarifying terms like null and alternate hypothesis. We'll also introduce the one-sample and two-sample t-tests, statistical tools that detect significant differences in means. Our step-by-step guides will show you how to perform these tests in Excel, and we'll also cover the Chi-square test used to determine significant associations between categorical variables. Finally, we'll explore Analysis of Variance (ANOVA), a technique to compare means of more than two groups. With practical examples along the way, you'll learn to uncover meaningful insights from your data, no matter your industry. Let's test some hypotheses!

Technical requirements (H1 – Section)

We will use the following files that can be found in the GitHub repository:

- One Sample t-test.xlsx
- Two Sample t-test.xlsx
- Chi Square Test.xlsx

- ANOVA in Excel.xlsx

Introduction to Hypothesis Testing

Hypothesis testing is a statistical method that enables us to make inferences about a population by analyzing a sample of data. Essentially, we create two opposing hypotheses about a population parameter, like the mean or proportion, and gather sample data to determine which hypothesis bears more accuracy. We will always start with a null hypothesis, or the "status quo," and seek out evidence against it. Upon finding such evidence, we may choose to refute the null hypothesis in favor of the alternative hypothesis.

Role of Hypothesis Testing in Data Analysis

Hypothesis testing is a crucial aspect of the data analysis process that provides a structured approach to decision-making based on data. It allows us to test assumptions derived from statistical models, making it an essential tool for assessing the significance of relationships in data. Descriptive statistics, on the other hand, only tell us what the data is, and inferential statistics, including hypothesis testing, are used to determine what the data signifies. By analyzing the results of hypothesis tests, we can learn whether observed differences in sample data are statistically significant or occurred by chance due to random variation. Below are a few real world situations where a data analyst would use hypothesis testing on the job:

Evaluating a Marketing Campaign: Assuming that you are a data analyst for an e-commerce company that launched a marketing campaign aimed at increasing the average order value (AOV), the company is keen to determine the campaign's effectiveness. The analyst has the option of establishing a hypothesis test in which the null hypothesis suggests that the marketing campaign had no impact on AOV, while the alternative hypothesis states that the campaign increased AOV. Data from both pre- and post-campaigns would be used alongside a hypothesis test like a one-sample t-test, and the findings would then be interpreted.

Exploring the Relationship between Variables: In a healthcare organization, you may be tasked with investigating the relationship between smoking and lung cancer. You could set up a hypothesis test where the null hypothesis states there is no

relationship between smoking and lung cancer. The alternative hypothesis is that there is a relationship. The analyst would use data on individuals' smoking habits and lung cancer diagnoses, conduct a chi-square test of independence to see if the variables are related, and draw conclusions from the results to whether smoking causes an effect on lung cancer or not. **Testing a New Product Feature:** Imagine you're a data analyst at a tech company that just introduced a new feature in its mobile app. The company wants to know if the new feature has led to an increase in user engagement, defined as time spent in the app. To measure the impact of a new feature on user engagement, a hypothesis test can be carried out. In this test, the null hypothesis is that the feature hasn't had any effect on engagement, while the alternative hypothesis is that it has increased engagement. The analyst then gathers data on user engagement before and after introducing the feature, conducts a suitable hypothesis test (such as a two-sample t-test), and makes a decision based on the findings.

Null and Alternative Hypothesis

In any hypothesis test, two opposing hypotheses are formulated: The **Null Hypothesis (H_0)** is the status quo that we want to investigate. Representing no difference or effect. In the case of testing a new drug, the H_0 could indicate that either the drug has no effect or that there's no difference in efficacy between the new drug and the existing one. The **Alternative Hypothesis (H_1 or H_a)** is the assertion we embrace if the proof is compelling enough to discard the null hypothesis. It represents an effect or difference. For instance, in our drug scenario, the alternative hypothesis could be that the new drug generates a disparate effect compared to the existing one. The goal of hypothesis testing is to provide evidence supporting the rejection or non-rejection of the null hypothesis. We do not outright accept either the null hypothesis or the alternative hypothesis. Rather, non-significant results are taken as a failure to reject the null hypothesis, whereas significant results lead to the rejection of the null hypothesis in favor of the alternative. The strength of the evidence is often expressed in terms of a p-value, a concept which we will cover in forthcoming sections. With a fundamental understanding of hypothesis testing and its importance in data analysis, let's explore various tests, starting with the one-sample t-test. This test is a great way to analyze sample means, so put it on your list of tests to learn.

Step by Step Guide to Performing Hypothesis Testing

Step 1: Formulate the Hypotheses Hypothesis testing begins with the establishment of a null hypothesis (H_0) and an alternative hypothesis (H_a). The former asserts that there is no difference or effect while the latter statement posits the opposite. This approach is crucial in determining the validity of the alternative hypothesis by deriving evidence against the null hypothesis. For example, if we want to test if the average height of a population is 5.5 feet, the null hypothesis would be $H_0: \mu = 5.5$, and the alternative hypothesis would be $H_a: \mu \neq 5.5$.

Step 2: Choose the Significance Level The significance level (often denoted by α) is the probability of rejecting the null hypothesis when it is true. It's essentially the risk of making a Type I error (false positive). The most common significance level is 0.05, which means there's a 5% risk of concluding that a difference exists when there is no actual difference.

Step 3: Select the Appropriate Test Depending on the data and the nature of your research question, you'll need to select an appropriate statistical test. Common tests include t-tests for comparing means, chi-square tests for testing relationships between categorical variables, and ANOVA for comparing means across more than two groups. The nature of your data (i.e., normal or not, dependent or independent) will guide your choice.

Step 4: Collect and Analyze the Data Collect the data that you need for your test, then use a statistical software package (such as Excel, R, or Python's `scipy.stats` library) to perform the test. The output will depend on the specific test you're performing, but you'll generally get a test statistic (like a t-value or F-value) and a p-value.

Step 5: Make a Decision The p-value represents the probability of obtaining an outcome as or more extreme than what was observed, under the condition that the null hypothesis is true. If the p-value is less than or equal to the significance level, you reject the null hypothesis and assume that the effect in your sample is a reflection of the actual population effect. Conversely, if the p-value is greater than the significance level, you fail to reject the null hypothesis. In the example of height, if our p-value is 0.03, we would reject the null hypothesis (since 0.03 is less than the significance level of 0.05) and infer that the average height of the population is not 5.5 feet. If we could not reject the null hypothesis, it does not necessarily mean the null hypothesis is correct - there just may not be enough evidence either way. Simultaneously, rejecting the null hypothesis doesn't mean the alternative hypothesis is automatically correct - it simply

suggests that it has more weight than the null hypothesis.

Step 6: Report the Results

Finally, report your results, including the test statistic, p-value, and whether you rejected or failed to reject the null hypothesis. Also, describe the practical significance of your findings. This step is critical for transparency and reproducibility in research. Keep in mind that this is a broad guide and the specific steps may vary slightly depending on the exact nature of your hypothesis test.

One Sample t-Test

A one-sample t-test is a statistical analysis that determines whether the mean of a single sample of scores differs from a known or hypothetical population mean. Determining if an event may have occurred by chance or not. For example, let's say we want to investigate whether the average height of individuals in a sample significantly differs from a specified value. The one-sample t-test allows us to compare the single sample with the population mean, enabling us to uncover if a significant difference exists between the two groups. This statistical test is predominantly used in research to determine if results of an experiment are significant, or if these results are merely due to chance. By comparing a single sample's mean score to a specific population mean, researchers can draw meaningful insights from the data and make informed decisions about the likelihood of that sample being a true representation of the underlying population.

Conditions for Performing a One-Sample T-Test

Before performing a one-sample t-test, certain assumptions need to be met:

1. **Independence of Observations:** Each observation should be independent of others. In practical terms, this means that the occurrence of one event has no influence on the next event.
2. **Normality:** The data should be approximately normally distributed. For large sample sizes (greater than 30), thanks to the Central Limit Theorem, this assumption can be somewhat relaxed.
3. **Scale of Measurement:** The scale of measurement should be continuous (interval/ratio).

Step-by-step Guide to Performing a One-Sample T-Test in Excel

1. Organize Your Data: Your data should be organized in a single column in Excel.
2. Navigate to the Appropriate Function: Click on the 'Data' tab, then navigate to 'Data Analysis' in the 'Analysis' group. If you don't see 'Data Analysis', you will need to load the Analysis ToolPak add-in.
3. Select T-Test: In the Data Analysis box, scroll down and select 't-Test: One-Sample Assuming Equal Variances', then click 'OK'.
4. Input Your Data: In the t-Test dialog box, input the range for your data sample in the 'Variable 1 Range' box. Check the 'Labels' box if you have included the column header.
5. Set Hypothesized Mean and Alpha: Input your hypothetical population mean in the 'Hypothesized Mean' box. The 'Alpha' is your chosen significance level, which is commonly set at 0.05.
6. Choose Output Range: Click the 'Output Range' option and select a cell where you want the output data to appear.
7. Run the Test: Click 'OK' to run the test.

Case Study: Average Exam Scores

Objective: The principal at Data Analytics High School believes that the mean exam score of all students in the school is around 75%. However, as an analyst, you have been given the task to statistically validate this claim. You will use a one-sample t-test to check if the mean exam score is indeed 75%.

Scenario: You have been given a random sample of exam scores of 20 students from Data Analytics High School. You have to determine whether there's statistical evidence that the actual mean (μ) of exam scores is different from the stated 75%.

- Null hypothesis (H_0): $\mu = 75$
- Alternative hypothesis (H_1): $\mu \neq 75$
- Significance: 0.05

Instructions to Perform One-Sample T-Test in Excel Using the Analysis ToolPak:

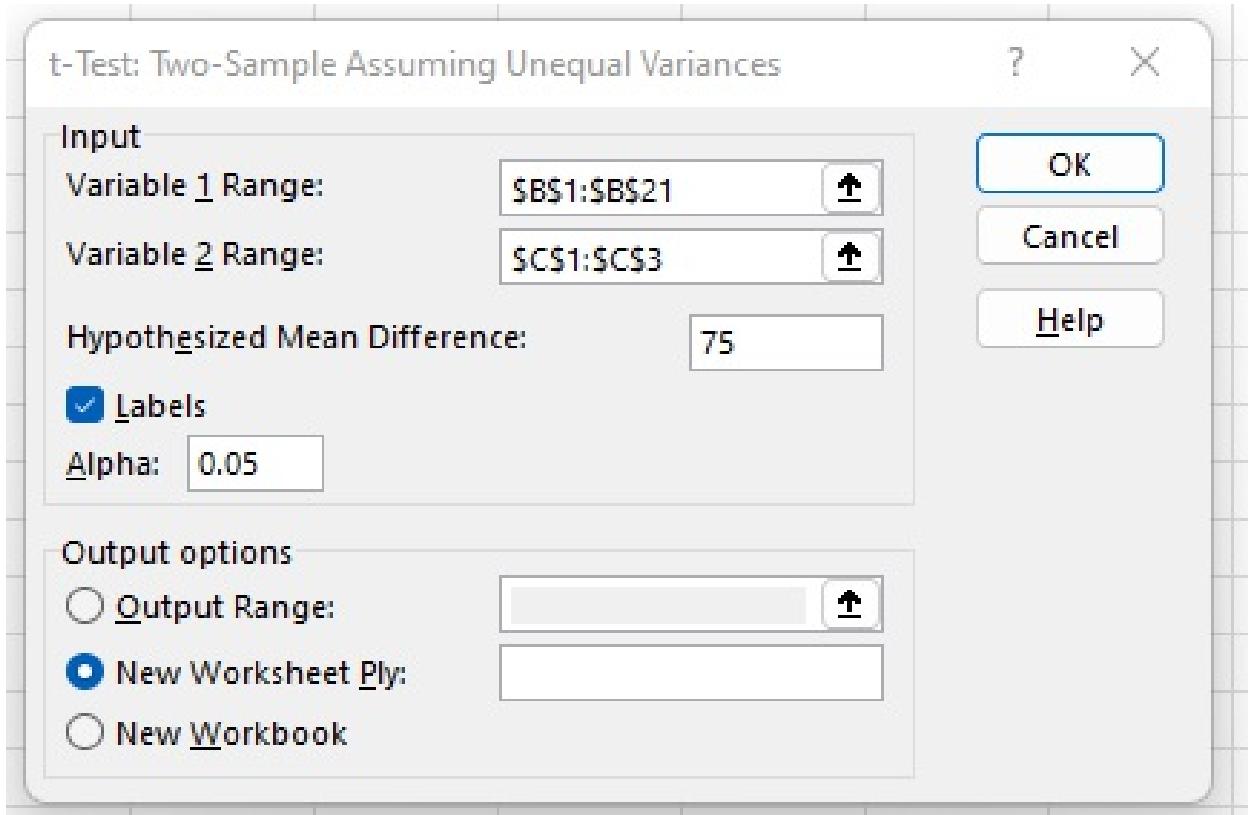


Figure 1. Inputs to perform the one sample t test

1. Upload the file called 'One Sample t Test.xlsx' into Excel.
2. You need to make sure the Analysis ToolPak is enabled. Go to **File > Options > Add-Ins**. In the 'Manage' box, select **Excel Add-ins** and then click **Go**. In the 'Add-Ins' box, check **Analysis ToolPak** and then click **OK**.
3. The ToolPak does not have a one sample t-test, so we will trick Excel to perform it for us using the Two-Sample test. We have created a dummy variable column for this purpose.
4. Go to the **Data** tab and then select **Data Analysis**.
5. In the Data Analysis box, scroll down and select **t-Test: Two-Sample Assuming Unequal Variances**, then click **OK**.
6. In the 't-Test: Two-Sample Assuming Unequal Variances' box:
 - For the 'Variable 1 Range' field, select your range of scores (B1:B21).
 - For the 'Variable 2 Range', input the dummy variable range (C1:C3)
 - For 'Hypothesized Mean Difference', input 75.

- You can leave 'Labels', 'Alpha', and 'Output Range' fields as is unless you have specific preferences.
7. Click 'OK'. Excel will return the t-test results in a new window.

t-Test: Two-Sample Assuming Unequal Variances		
	Score (%)	Dummy
Mean	75.85	0
Variance	13.92368	0
Observations	20	2
Hypothesized Mean Difference	75	
df	19	
t Stat	1.018725	
P(T<=t) one-tail	0.160566	
t Critical one-tail	1.729133	
P(T<=t) two-tail	0.321131	
t Critical two-tail	2.093024	

Figure 2. Results of the One sample t test

Interpreting the Results The output table will give you the sample mean, variance, the observed t Statistic, and the P($T \leq t$) one-tail and two-tail values. The two key values to look at are the P($T \leq t$) two-tail and the t Statistic:

- **P($T \leq t$) Two-tail:** This is the p-value. If this value is less than your chosen alpha level (often 0.05), then you reject the null hypothesis and conclude that there is a significant difference between the sample mean and the hypothesized population mean.
- **t Statistic:** This value can be positive or negative, indicating whether the sample mean is greater or less than the hypothesized mean.

After conducting an investigation using the one-sample t-test, we have gained a better understanding of comparing sample means to known population

means. The results of the test offer valuable insights into whether the sample meaningfully deviates from the population. Real-world situations often call for comparison between two distinct groups, rather than a sample against a known population. Examples may include contrasting the success rates of different marketing campaigns, the efficacy of varied medications, or sales figures in distinct regions. Next up, let's dive into the two-sample t-test, a fundamental statistical tool. It helps us identify if the means of two groups differ significantly.

Two Sample t-Test

A two-sample t-test, commonly referred to as the independent t-test, is a statistical method employed to determine whether two independent groups' means differ significantly. It's a parametric test that requires the populations from which the samples are drawn to be normally distributed. Consider wanting to discern whether spending averages among customers differ significantly between two distinct supermarket branches, or comparing average test scores in two separate classes. In both cases, a two-sample t-test would be the most appropriate tool for statistical analysis. **Conditions for Performing a Two-Sample T-Test** Before we carry out a two-sample t-test, we need to ensure the data meet certain conditions:

- **Independence:** The two groups being compared must be independent of each other. That is, what happens in one group does not influence what happens in the other group.
- **Normality:** The data from each group should follow a normal distribution. If the sample size is large enough (>30), thanks to the Central Limit Theorem, this assumption can be relaxed.
- **Equal Variance:** It is assumed that the variances of the populations the two samples come from are equal. However, if this is not the case, we can adjust the test for unequal variances (Welch's t-test)

Case Study: Comparing Exam Scores Between Two Schools

Objective: The principal at Dawson High School believes that students at his school perform better than students at Rivertown High School. To investigate this claim, you are tasked with conducting a two-sample t-test to compare the

average exam scores of the two schools. **Scenario:** You have been given a random sample of exam scores of 25 students each from Dawson High School and Rivertown High School. The objective is to determine whether there's statistical evidence that the actual mean exam score at Dawson High School is different from that at Rivertown High School.

- Null hypothesis (H_0): $\mu_1 = \mu_2$ (The mean score for Dawson High is equal to the mean score for Rivertown High)
- Alternative hypothesis (H_1): $\mu_1 \neq \mu_2$ (The mean score for Dawson High is not equal to the mean score for Rivertown High)
- Significance: 0.05

Instructions to Perform Two-Sample T-Test in Excel Using the Analysis ToolPak:

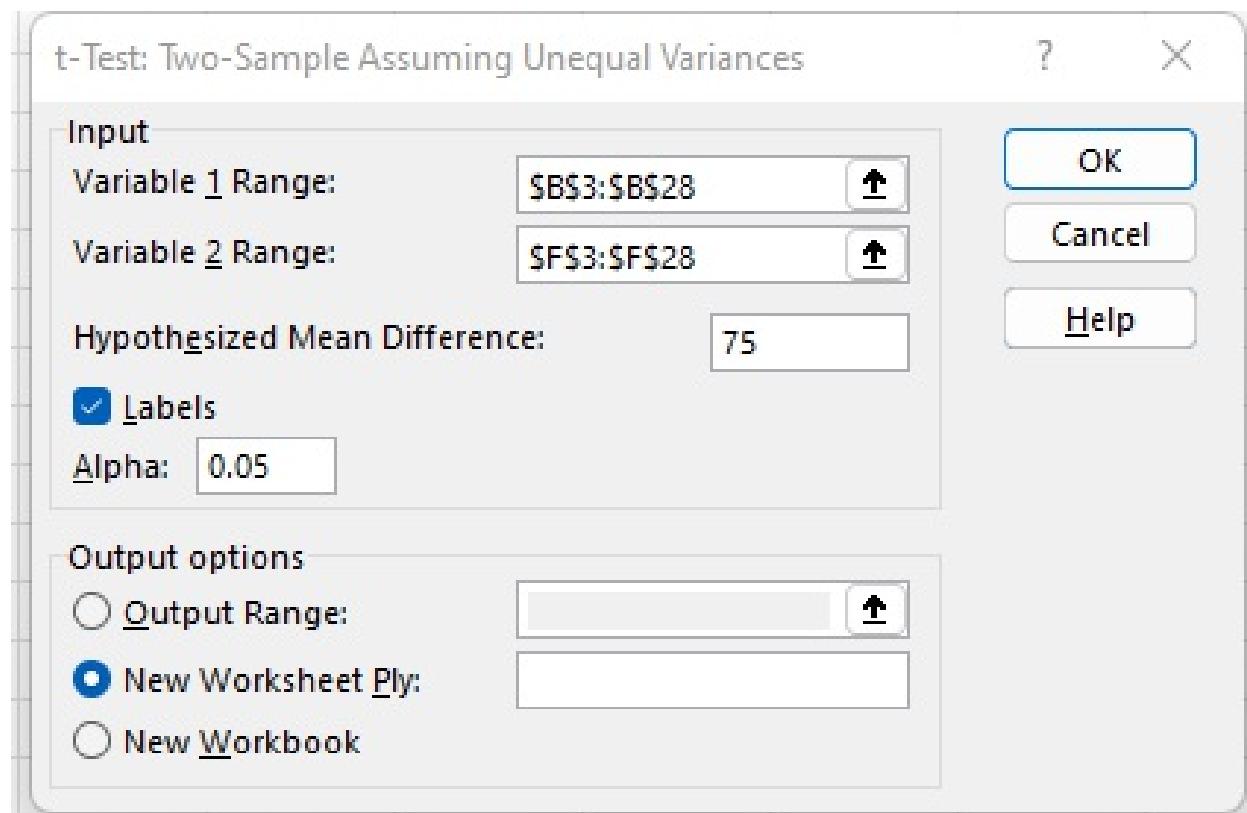


Figure 3. Inputs to perform the two sample t test in Excel

1. Upload the file called 'Two Sample t Test.xlsx' into Excel.
2. First, you need to make sure the Analysis ToolPak is enabled. Go to **File**

- > **Options > Add-Ins.** In the 'Manage' box, select **Excel Add-ins** and then click **Go**. In the 'Add-Ins' box, check **Analysis ToolPak** and then click **OK**.
3. Go to the **Data** tab and then select **Data Analysis**.
 4. In the Data Analysis box, scroll down and select **t-Test: Two-Sample Assuming Unequal Variances**, then click **OK**.
 5. In the '**t-Test: Two-Sample Assuming Unequal Variances**' box:
 - For the 'Variable 1 Range' field, select your range of the Data Analyst High School scores (B3:B38).
 - For the 'Variable 2 Range' field, select your range of Data Scientist High School scores (F3:F38).
 - For 'Hypothesized Mean Difference', input 75.
 - You can leave 'Labels', 'Alpha', and 'Output Range' fields as is unless you have specific preferences.
 1. Click 'OK'. Excel will return the t-test results in a new window.

t-Test: Two-Sample Assuming Unequal Variances		
	Score (%)	Score (%)
Mean	75.72	71.16
Variance	12.04333	2.64
Observations	25	25
Hypothesized Mean Difference	75	
df	34	
t Stat	-91.913	
P(T<=t) one-tail	1.21E-42	
t Critical one-tail	1.690924	
P(T<=t) two-tail	2.42E-42	
t Critical two-tail	2.032245	

Figure 4. Results of the two sample t test in Excel

Interpreting the Results The output table will give you the means and variances of both groups, the observed t Statistic, and the P($T \leq t$) one-tail

and two-tail values. The key values to look at are the $P(T \leq t)$ two-tail and the t Statistic:

- **$P(T \leq t)$ Two-tail:** This is the p-value. If this value is less than your chosen alpha level (often 0.05), then you reject the null hypothesis and conclude that there is a significant difference between the two group means.
- **t Statistic:** This value can be positive or negative, indicating which sample has the larger mean.

While the t-tests we've examined up until now are adept at comparing means and require the data to be somewhat normally distributed, we face a different issue when our data are categorical. Specifically, when the relationship between two variables is of interest, rather than just comparing means. What do we do then? This leads us into our next topic: the Chi-Square test. The Chi-Square test is a non-parametric method used in statistics to determine if there's a significant association between two categorical variables in a sample. It is particularly useful when analyzing data that are categorized into groups, not numbers. In the next section, we'll delve into the Chi-Square test, including how to implement it in Excel and interpret the results.

Chi Square Test

The chi-square test is a widely used non-parametric statistical test that plays an essential role in research by helping to determine whether there is a significant association between two categorical variables in a sample. Researchers can use this to evaluate relationships between different demographic and categorical variables. There are two fundamental types of chi-square tests: the chi-square test of independence and the chi-square goodness of fit test. The test of independence tests whether two categorical variables are independent, while the goodness of fit test determines if a set of observed frequencies match the expected frequencies. In this section, we will mainly focus on the chi-square test of independence, which is a vital statistical method to determine the extent of the association between categorical variables, providing insights into the interrelationship between them. By understanding the results of this test, researchers can make more informed decisions, ultimately improving their research findings and the

impact of their work. There are certain conditions to meet before performing a chi-square test:

- Categorical Variables: Both variables under consideration must be categorical (ordinal or nominal).
- Independence: Observations should be independent of each other, meaning that the occurrence of one event does not influence the occurrence of another.
- Sample Size: All cells should have an expected count greater than five. This is called the "five-count rule".

Case Study: Effect of Tutoring on Passing Rates

Objective: The principal at Data Analyst High School believes that students who receive tutoring are more likely to pass their exams than those who do not. As an analyst, you have been given the task to statistically validate this claim using a Chi-Square test for independence. **Scenario:** You have been given a sample of exam results for 100 students. Half of the students received tutoring while the other half did not. The objective is to determine whether there's statistical evidence that tutoring affects pass rates.

- Null hypothesis (H_0): Tutoring is independent of passing (no relationship)
- Alternative hypothesis (H_1): Tutoring is not independent of passing (there is a relationship)
- Significance: 0.05

Performing Chi Square test in Excel

1. Compute the expected values for each cell to create the expected values table. The formula for each expected value is **(Row Total * Column Total) / Grand Total**.

In this case:

- Expected Passed & Tutored = $(50 * 63) / 100 = 31.5$
- Expected Failed & Tutored = $(50 * 37) / 100 = 18.5$
- Expected Passed & Not Tutored = $(50 * 63) / 100 = 31.5$

- Expected Failed & Not Tutored = $(50 * 37) / 100 = 18.5$
1. Add these expected values to your Excel spreadsheet, preferably in an area adjacent to the observed values. Let's say you put these in cells F2 to G3, so your setup looks like:

Figure 5. Expected values table after calculations

2. Go to the **Formulas** tab and select **More Functions > Statistical**.
3. Scroll down and select **CHISQ.TEST**.
4. For the 'Actual_range' field, select your observed values (B2:C3).
5. For the 'Expected_range' field, select your expected frequencies (B8:C9).
6. Press 'OK'. Excel will return the p-value for the Chi-Square test in the next selected cell.

Interpreting the Results

1. The CHISQ.TEST function returns the p-value of 0.07 chi-square test.
2. If the p-value is less than your chosen alpha level (often 0.05), then you reject the null hypothesis and conclude that there is a significant relationship between the two categorical variables.
3. If the p-value is greater than your chosen alpha or significance level, then you fail to reject the null hypothesis and conclude that there is not enough evidence to suggest a significant relationship between the variables.

By exploring the Chi-Square test, we have broadened our statistical capabilities to analyze categorical data. This non-parametric method has provided us with the tools to understand the association between two categorical variables, shedding light on relationships that numerical tests such as the t-test might miss. It's a crucial test in a wide array of fields, helping researchers and analysts alike to understand the interplay between different categorical factors. However, thus far, we have mostly examined tests that allow us to compare two groups or examine the relationship between two categorical variables. What if we have more than two groups and we want to compare their means? For instance, suppose we're interested in comparing the

effectiveness of more than two teaching methods, or we want to compare sales across several stores. This leads us to the next statistical method we will cover: Analysis of Variance, or ANOVA. ANOVA is a statistical technique that extends the two-sample t-test to compare means across more than two groups. It's especially useful when we want to test the effect of some categorical independent variable on a numerical dependent variable across multiple groups. In the next section, we will introduce the concept of ANOVA, learn how to implement it in Excel, and discuss how to interpret the results. With ANOVA in our toolkit, we'll be equipped to draw even more robust insights from our data, giving us the ability to make effective, data-driven decisions across a variety of scenarios.

Analysis of Variance (ANOVA)

Analysis of Variance (ANOVA) is a statistical tool that helps you test for disparities between two or more means. You may wonder why the technique isn't called "Analysis of Means." That's because ANOVA analyzes the variance in the data to determine whether the means are significantly distinct. By illuminating differences between groups, ANOVA saves you time, enabling you to unearth meaningful insights from your data. One significant advantage of ANOVA is the ability to concurrently compare multiple groupings. For instance, if one seeks to evaluate the existence of a significant difference in the average height amongst several age groups, ANOVA is a fitting analytical method to apply. **Conditions for Performing an ANOVA** Before conducting an ANOVA, the following assumptions need to be satisfied:

1. Independence of Observations: Each sample is independent from the others.
2. Normality: The data within each group are approximately normally distributed.
3. Homogeneity of Variance: The variance among the groups should be roughly equal. This is also known as the assumption of homoscedasticity.

Case Study: Comparing Exam Scores Among Three Schools

Objective: The principal at Data Analyst High School believes that students at his school perform differently than students at Data Scientist High School and Data Engineer High School. To investigate this claim, you are tasked with conducting an Analysis of Variance (ANOVA) test to compare the average exam scores of the three schools. **Scenario:** You have been given a random sample of exam scores of 30 students each from Dawson High School, Rivertown High School, and Lakeside High School. The objective is to determine whether there's statistical evidence that the actual mean exam score is different among the three schools.

- Null hypothesis (H_0): $\mu_1 = \mu_2 = \mu_3$ (The mean scores for all schools are equal)
- Alternative hypothesis (H_1): At least one school's mean score is different

Performing ANOVA in Excel

1. Go to the **Data** tab and then select **Data Analysis**.
2. In the Data Analysis box, scroll down and select **Anova: Single Factor**, then click **OK**.
3. In the 'Anova: Single Factor' box:
 - For the 'Input Range' field, select your range of scores (A2:C31 in this example).
 - Check 'Grouped By Columns' since your data is arranged in columns.
 - For 'Alpha', input your significance level (commonly 0.05).
 - You can leave 'Labels' and 'Output Range' fields as is unless you have specific preferences.
4. Click 'OK'. Excel will return the ANOVA test results in a new window.

Anova: Single Factor					
SUMMARY					
Groups	Count	Sum	Average	Variance	
Data Analyst High School Score (%)	30	2389	79.63333	3.067816092	
Data Scientist High School Score (%)	30	2158	71.93333	1.512643678	
Data Engineer High School Score (%)	30	2297	76.56667	1.288505747	

ANOVA						
Source of Variation	SS	df	MS	F	P-value	F crit
Between Groups	901.6222	2	450.8111	230.438112	1.72346E-35	3.101296
Within Groups	170.2	87	1.956322			
Total	1071.822	89				

Figure 6. Results of ANOVA

Interpreting the Results The output table will include the 'Between Groups' and 'Within Groups' variance, the F statistic, and the P-value.

- F statistic: This is the test statistic. It's a ratio of the variance between groups to the variance within groups.
- P-value: This is the probability of getting an F statistic as extreme as, or more extreme than, the observed value, assuming the null hypothesis is true. If this value is less than your chosen alpha level (often 0.05), then you reject the null hypothesis and conclude that there is a significant difference between at least two of the group means.

Summary

In this chapter, we embarked on a journey through four fundamental statistical tests: One-Sample T-Test, Two-Sample T-Test, Chi-Square Test, and ANOVA. Each test has its unique capabilities and contexts in which it is most appropriate. We started with the One-Sample T-Test, a technique that allows us to compare a sample mean to a known population mean. This test helps us to understand whether our sampled data deviates significantly from the known population mean. Through an interactive case study, we practiced using Excel to perform a One-Sample T-Test, strengthening our practical

understanding of statistical analysis. Moving forward, we explored the Two-Sample T-Test. Unlike the One-Sample T-Test, this test allows us to compare the means of two independent groups. It's particularly useful when we need to understand differences between two groups based on sampled data. Next, we delved into the Chi-Square test, a non-parametric method used to determine if there is a significant association between two categorical variables. This statistical tool helped us analyze relationships between categorical variables, adding a new layer of complexity to our data analysis capabilities. Finally, we explored Analysis of Variance or ANOVA, a powerful statistical technique that extends the two-sample t-test to compare means across more than two groups. With the help of Excel, we learned how to implement ANOVA and how to interpret its results, empowering us to make data-driven decisions across a variety of scenarios. Together, these tests provide a solid foundation for statistical analysis. By understanding when and how to use these tests, we can make informed decisions based on our data, drive meaningful insights, and answer complex questions about the world around us.

9 Business Statistics Case Study

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



This chapter will dive into the practical application of business statistics through a comprehensive case study. This case study gives prospective data analysts a practical setting to use the descriptive and inferential concepts that we've covered in this book so far. After the case study, we will provide sample interview questions that will help you prepare for your interviews. For further optional study, additional advanced topics in analytics will be provided along with resources. By the end of this chapter, you will have a firm grasp of using statistical methods to examine business data and make relevant inferences. Additionally, you will develop real-world experience processing data from the actual world and developing business-related questions.

Technical requirements (H1 – Section)

This chapter utilizes `employee_data.csv` which can be found at https://github.com/PacktPublishing/Becoming-a-Data-Analyst-First-Edition/blob/main/Chapter9%20-%20Business-Statistics-Case-Study/employee_data.csv. The answers can be found in the file titled `Case Study Answers.pdf`.

Case Study Overview

Assume you work as a data analyst for the international company “Data in Motion Tech”. Sales, HR, IT, and Finance are the company's four main departments, which together make up Data in Motion's diversified workforce. The HR department has recently seen several trends and oddities in employee compensation and performance ratings. They believe that unobserved causes might cause these observations, but they are unsure. The HR department has contacted you to assist them in better comprehending these tendencies. They give you access to a dataset including data on 500 employees, including each employee's ID, gender, age, department, years of experience, salary, and performance score. You can complete the case study in any tool of your choice (Excel, Python, R, Tableau, Power BI, etc).

Learning Objectives:

1. Perform descriptive statistics to summarize and understand the data
2. Conduct t-tests to compare means between two groups
3. Perform chi-square tests to examine associations between categorical variables
4. Conduct ANOVA tests to compare means among more than two groups
5. Calculate correlation coefficients to measure the strength and direction of relationships between two variables

Questions:

1. **Descriptive Statistics:** The HR department is curious about the basic statistics of salaries and experience across different departments. They ask you:
 - What is the average, median, and range of salaries for each department?
 - Also, how does the average years of experience vary for each performance score category?
2. **T-tests:** During a meeting, the HR manager raises a concern about gender pay equity. They ask you:
 - Is there a significant difference in the average salary between male

- and female employees?
- Also, we have been investing heavily in training for the Sales department. Is their experience significantly different from the IT department?
3. **Chi-Square Test:** The HR department is planning some initiatives to promote diversity and inclusion. They ask you:
- Is there a significant association between gender and department?
 - Also, does the department an employee works in influence their performance score?"
4. **ANOVA:** The finance department wants to ensure that the budget allocation for salaries is fair across all departments. They ask you:
- Is there a significant difference in the average salary among the different departments?
 - Also, does the average years of experience vary significantly among the different performance score categories?
5. **Correlation:** The HR manager is interested in understanding the factors that influence an employee's salary. They ask you:
- Is there a correlation between years of experience and salary?
 - What about performance score and salary?

Solutions:

Additional Topics to Explore

It's critical to broaden our knowledge and investigate numerous aspects of this diverse profession as we go further into the fascinating realm of data analytics. We strive to introduce and provide resources for several important issues that are becoming increasingly important in the data analytics environment in this area, "Additional Topics to Explore." These subjects include time series analysis, big data, predictive analytics, prescriptive analytics, database management, and text analytics. Each component includes a synopsis of the subject and links to more reading. These topics will offer insightful information on the changing dynamics of data analytics, whether you want to broaden your skill set or focus on a specific sector.

Text Analytics

Text analytics, often known as text mining, draws essential knowledge, intelligence, and insights from unstructured text data. Usually, it entails structuring the input text, finding patterns in the structured data, and assessing and interpreting the results. Natural Language Processing (NLP), sentiment analysis, categorization, and entity recognition are a few methods it uses. These methods are drawn from linguistics, computer science, and machine learning. Text analytics can be used in various situations by a data analyst. An example is in customer service to examine customer reviews and feedback to find recurring themes and feelings. This may assist a company in understanding what clients are saying about their goods or services and making adjustments as necessary. Text analytics can be used in a social media setting to understand better the attitudes and trends surrounding a specific subject or brand, offering helpful information for marketing and strategic planning. Resources to learn and practice text analytics:

- *Mastering Text Mining with R* by Avish Paul and Kumar Ashish.
- *Hands-On Python Natural Language Processing* by Aman Kedia and Mayank Rasu.
- *Machine Learning for Text* by Nikos Tsourakis.
- Data in Motion Data Career Academy monthly and weekly skill challenges.

Learning Roadmap:

1. Start with understanding basics of Natural Language Processing (NLP).
2. Learn about text preprocessing techniques such as tokenization, stemming, and lemmatization.
3. Familiarize yourself with techniques for feature extraction from text, like Bag of Words and TF-IDF.
4. Learn about sentiment analysis and how it can be used to understand opinions and attitudes in text.
5. Study advanced NLP techniques like named entity recognition, topic modeling, and word embeddings.

Key Concepts: Tokenization, stemming, lemmatization, stop words, n-grams, feature extraction, sentiment analysis, named entity recognition, topic modeling, word embeddings (like Word2Vec and GloVe).

Big Data

Big data is a term that describes extensive and complex data sets that traditional data processing software can't manage. These data sets are characterized by their volume, variety, velocity, and veracity. Big data analytics uses advanced analytic techniques to extract valuable information from these data sets. Understanding big data is becoming increasingly crucial for a data analyst as the volume of data generated by businesses and individuals grows. Use cases for big data analytics in a data analyst role are vast. They range from analyzing customer behavior to improving marketing strategies to using machine logs for predictive maintenance in manufacturing industries. In healthcare, big data analytics can help predict disease outbreaks or improve patient care by analyzing large volumes of patient data. Resources to learn about big data:

- *Hands on Big Data Analytics with PySpark* by James Cross , Rudy Lai , and Bartłomiej Potaczek.
- *Big Data Analysis with Python* by Ivan Marin , Ankit Shukla, and Sarang VK.
- *Practical Big Data Analytics* by Nataraj Dasgupta.

Learning Roadmap:

1. Understand the basics of big data, including the 3Vs.
2. Learn about distributed storage and processing frameworks like Hadoop and Spark.
3. Familiarize yourself with NoSQL databases like MongoDB, Cassandra, and HBase.
4. Learn about big data processing and analytics tools like Hive and Pig.
5. Explore real-time processing with tools like Storm and Flink.

Key Concepts: Hadoop, MapReduce, Spark, NoSQL databases, distributed storage, real-time processing.

Time Series Analysis

Time series analysis involves the study of data points collected or recorded in

time order. The primary objective of time series analysis is to carefully collect and scrutinize data to discern the structure and functions that produce the observed data. Time series analysis is often used for forecasting, where one uses information collected in the past to predict future values. A data analyst might use time series analysis in various scenarios. For instance, time series analysis is used in finance to examine historical stock prices and project future prices. It is used in economics to comprehend and forecast factors like unemployment, GDP, and inflation rates. A data analyst in retail may use time series analysis to estimate sales to make sure there are enough stock levels for the foreseeable future. Time series analysis aids in making wise decisions in each of these situations based on trends that have been noticed throughout time. Resources to learn and practice time series analysis:

1. *Modern Time Series Forecasting with Python* by Manu Joseph.
2. *Machine Learning for Time Series* by Ben Auffarth.
3. *Hands on Time Series Analysis with R* by Rami Krispin.
4. Data in Motion Data Career Academy monthly and weekly skill challenges.

Resources to learn about big data:

- *Hands on Big Data Analytics with PySpark* by James Cross , Rudy Lai , and Bartłomiej Potaczek.
- *Big Data Analysis with Python* by Ivan Marin , Ankit Shukla, and Sarang VK.
- *Practical Big Data Analytics* by Nataraj Dasgupta.

Learning Roadmap:

1. Understand what a time series is and the unique challenges it presents.
2. Learn about basic components of a time series: trend, seasonality, and noise.
3. Familiarize yourself with methods for visualizing time series data.
4. Learn about different models for time series forecasting, such as ARIMA and state space models.
5. Explore advanced topics like multivariate time series and machine learning for time series.

Key Concepts: Time series components, stationarity, autocorrelation, ARIMA models, seasonality, trend analysis, forecasting.

Predictive Analytics

Using data, statistical algorithms, and machine learning approaches, predictive analytics determines the likelihood of future events based on historical data. Predictive analytics seeks to deliver the most accurate forecast of what will occur by going beyond simply knowing what has already occurred. Predictive analytics is a potent tool for data analysts that may assist businesses in making data-driven decisions. It requires a number of stages, including designing the project, gathering data, analyzing data, developing a predictive model, validating and putting the model into use, and tracking the model over time. Predictive analytics can be used in various fields and industries. Here are a few examples:

- **Marketing:** Predictive analytics can help businesses identify trends and predict customer behavior, which can be used to drive more effective marketing campaigns. For instance, predictive analytics can help identify which customers are most likely to respond to a particular offer, or which customers are at risk of churning.
- **Banking:** Predictive analytics can be used to assess the likelihood of a customer defaulting on a loan.
- **Healthcare:** Predictive analytics can be used to predict disease outbreaks or to identify patients at risk of developing certain conditions. This can help healthcare providers intervene earlier and improve patient outcomes.

Resources to learn and practice predictive analytics:

1. *Machine Learning with R* by Brett Lantz.
2. *Hands on Predictive Analytics with Python* by Alvaro Fuentes.
3. Data in Motion Data Career Academy monthly and weekly skill challenges.

Learning Roadmap:

1. Begin by understanding the basics of predictive modeling and its role in

decision-making.

2. Familiarize yourself with regression models, a common method used in predictive analytics.
3. Learn about classification techniques such as logistic regression, decision trees, and random forests.
4. Explore advanced machine learning methods for prediction, including neural networks and support vector machines.
5. Study the principles of model evaluation and validation, including concepts like overfitting, underfitting, cross-validation, and ROC curves.

Key Concepts: Supervised learning, unsupervised learning, overfitting, underfitting, model validation, model performance metrics, and deep learning.

Prescriptive Analytics & Optimization

A subset of business analytics called prescriptive analytics, commonly called decision science, focuses on giving advice or suggestions. It provides recommendations on potential outcomes using optimization and simulation methods. Prescriptive analytics seeks to not only forecast future events but also to suggest courses of action that will benefit from the anticipated future. The discipline of optimization, which is the process of selecting the optimal option from among a range of feasible alternatives, is highly reliant on prescriptive analytics. It entails identifying an objective function that must be maximized or minimized, such as maximizing profit or minimizing costs, and then determining the values of the decision variables that, given to a set of restrictions, yield the best value of the objective function. Prescriptive analytics can be used in various fields and industries. Here are a few examples:

- **Supply Chain Optimization:** Prescriptive analytics can be used to optimize many elements of the supply chain, such as inventory levels, shipping routes, and warehouse locations, in order to reduce costs while still satisfying client demand.
- **Energy Management:** Prescriptive analytics can assist in determining the best combination of energy sources to employ in the energy sector in order to minimize costs, lower emissions, or accomplish other

objectives.

- **Marketing Strategy:** To maximize return on investment, prescriptive analytics can be used in marketing to optimize the distribution of marketing resources across various channels, goods, or client categories.
- **Healthcare:** To enhance patient outcomes and lower costs, prescriptive analytics can be used to optimize staffing levels, operating room schedules, or treatment regimens.
- **Finance:** Prescriptive analytics are frequently used in the financial sector to determine the optimum investment portfolio to maximize return or minimize risk given a set of limitations. This process is known as portfolio optimization.

Resources to learn and practice predictive analytics:

- "Operations Research: Applications and Algorithms" by Wayne L. Winston
- Python's PuLP library for linear programming

Learning Roadmap:

1. Learn about linear programming and its use in optimization problems.
2. Study integer programming and mixed-integer programming for dealing with discrete decision variables.
3. Explore more complex optimization techniques like nonlinear programming and stochastic optimization.
4. Learn about software tools used for optimization, such Arena.

Key Concepts: Linear programming, integer programming, nonlinear programming, stochastic optimization, decision variables, objective function, and constraints.

Database Management

The tasks involved with administering a database, which is a structured set of data, are referred to as database management. This can entail creating the database's structure, entering data, implementing security controls, preserving the data's integrity, and retrieving data as required. In order to collect and analyze data, a database management system (DBMS) communicates with

applications, end users, and the database itself. An analogy to a file manager that handles data in a database as opposed to saving files in a file system is a DBMS. There are various DBMS kinds, including the most popular, Relational Database Management Systems (RDBMS), NoSQL DBMS, In-Memory DBMS, Columnar DBMS, and others. Database management can be used in various fields and industries. Here are a few examples:

- **Banking and finance:** To keep track of customer data, balances, loans, and transactions, banks employ databases. Additionally, they use databases for fraud detection, client profiling, and risk management.
- **E-commerce:** E-commerce companies utilize databases to keep track of their product inventories, customer data, order information, and payment details. Additionally, they leverage databases for customer behavior research, recommendation engines, and personalization.
- **Telecommunication:** To manage client information, call history, billing, and problem management, telecommunications businesses employ databases.

Resources to learn and practice predictive analytics:

- *SQL for Data Analytics* by Jun Shan , Matt Goldwasser, and Upom Malik.
- *SQL Query Design Patterns and Best Practices* by Steve Hughes , Dennis Neer, and Dr. Ram Babu Singh.

Learning Roadmap:

1. Learn about relational databases and SQL (Structured Query Language).
2. Understand the principles of database design, including normalization and indexing.
3. Familiarize yourself with the concepts of transactions, concurrency, and database security.
4. Explore non-relational databases, also known as NoSQL databases.

Key Concepts: Databases, DBMS, SQL, database design, normalization, indexing, transactions, concurrency, database security, and NoSQL.

Where to practice

Hands-on practice is essential to acquiring data analysis and associated skills. While it is important to learn and comprehend concepts, nothing can replace the experience of working with real data and the challenges that come with it. The Data Career Academy, a program provided by Data in Motion LLC, is a fantastic source for hands-on education. This online academy offers a systematic path for learning and honing important skills, and it is intended exclusively for people who want to work in professions related to data. Students at the Data Career Academy participate in weekly and monthly skill challenges that let them put their knowledge of important topics like SQL, data analysis, data science, and Python to use. These exercises are a great method to be ready for your career because they are made to look like the kinds of issues you could encounter in a real-world data job. The school gives you the option to learn at your own pace while also providing a steady framework to help you stay on course. You can develop your skill set and get expertise in a variety of activities and scenarios thanks to the challenges' relevance, practicality, and variety. The Data Career Academy is a fantastic resource for anyone who are serious about developing a career in data and looking to study, practice, and advance their skills. It could be a crucial step toward developing into a self-assured, competent data expert.

Summary

In this chapter, we conducted a comprehensive business statistics case study using a dataset of 500 employees from a multinational corporation. The dataset included each employee's ID, gender, age, department, years of experience, salary, and performance score. This provided a practical application of the statistical concepts discussed in *Fundamental Statistics Concepts and Testing Hypothesis*. We also provided sample interview questions to aid your preparation for the job search and additional analytics topics to explore. In the next chapter, Data Analysis and Programming, we will learn the applications and importance of programming languages for data analytics.

10 Data analysis and programming

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



It's time to take you beyond the world of spreadsheets and dashboards, into the dynamic world of programming. I'm super excited and honored to be your guide. Let's be honest: for some people this is the scariest pillar. And that's okay, it's going to be my personal mission to show you that this is the most fun part of the job (which aligns with my motto: aim high!). And even if I can't convince you it's fun, you'll absolutely end up needing it and appreciate the role of programming in the bigger picture. You can definitely get started as a data analyst without programming skills. But eventually, there's no other option than to embrace it. Programming is an important skill for anyone involved in data analysis. Honestly, there's no need to run away or hide. Everybody can do it! Just like almost anything new, it's tough in the beginning. But programming can eventually really become a second nature. Here's what we'll be doing:

- Discuss why we need programming in data analysis
- Talk about the most important programming languages in data analysis
- Familiarize ourselves with the command line interface
- Set up our environment for Python programming
- Run our first Python program
- Understand what we can do with Python

Throughout this programming pillar, we'll be using the hypothetical real

world case of CleanAndGreen, a sustainability focused startup that needs your help solving diverse data analytics tasks for them. By the end of this entire pillar, you'll be equipped with the basics of Python for data analysis tasks. Enough talking, let's see why we exactly need to learn programming and how this is going to help solving data analysis problems!

The role of programming and our case

In order to illustrate the need for programming in data analysis, let's introduce you to this pillar's case: CleanAndGreen. CleanAndGreen is a startup. They strive to curb waste in sprawling urban jungles. In their mission, they're confronted with vast amounts of data on waste generation, disposal, and recycling rates. In analyzing this data, they can evaluate their impact, guide their decisions, and propel their cause, strategic decisions that are vital to their survival as a startup. But without having programming skills, this seems like an impossible task. The sheer volume of data and data formats becomes overwhelming, and the complexity of analysis exceeds beyond measure. However, CleanAndGreen transforms this daunting task into an efficient process by adding a programming language to the mix of tools. They can automate data collection, efficiently manage colossal datasets in different formats, and execute complex analyses to extract valuable insights. The value of programming for data analysis is visible in any slightly more complex case. For example, consider a healthcare institution aiming to improve patient outcomes. They can use programming to analyze patient data and predict disease risk, thus enabling early interventions and saving many lives! Or picture a retail giant seeking to optimize its supply chain. Programming allows them to parse through sales data, market data and forecast demand, aiding them in avoiding stockouts or surpluses. There's so many programming languages. We will introduce you to various programming languages commonly used for data analysis before getting started with setting up our environments for coding.

Different programming languages

When choosing a programming language, you need to keep in mind what to do with it. For creating a website, you'll probably choose something different

than when you want to program a part of the functionality of a car, and yet another language for doing data analysis. Clearly, that last one is going to be our focus. We need to be able to solve diverse complex tasks. From automating tedious manual data analysis tasks to predicting future trends using machine learning. There are still several options for choosing a language, and the skills you'll acquire are helpful and cannot be missed in today's data-driven world. So let's see what logical options we're currently having. Please mind, this world is changing constantly. And learning one language, will help you with whatever future language you'll need to learn as well. Let's start with the one will end op focusing on: Python.

Python

Python is a very obvious one to put on the list. It's a widely used multipurpose language that is used for many things, amongst which data analysis. (Fun fact, just like one of your authors, it's from the early 90s and the Netherlands.) Python comes with a lot of libraries that are great for data analysis, including NumPy, Pandas, Matplotlib and a lot more. A library is some sort of add on, written in that language, that can be easily used to perform certain tasks. In most cases, you just need to know which statistical thing you need to do, and then if you give the function of the library the correct parameters, the slightly harder math is handled inside the library and it just gives you back the correct result. Let's have a look at how to create a list and iterate over it in Python (and do the same for the languages below so you can see strong similarities between different languages). Here's how Python does it:

```
numbers = [1, 2, 3, 4, 5]
for i in numbers:
    print(i)
```

Python heavily relies on the correct indentation level to group code blocks, where other languages use curly brackets to create code blocks. An example of such a language is R. Let's see R next.

R

R is primarily used for statistical analysis and data visualization. It is a

common choice for academic research, and many people know the basics of R after completing a program at a university. But that's not the only reason that it's a very popular choice among advanced statisticians and data scientists. It offers a rich library of statistical and graphical methods. Here is an example of a basic R syntax for creating and iterating through a vector (like lists in Python):

```
my_vector <- c(1, 2, 3, 4, 5)
for (i in my_vector) {
  print(i)
}
```

However, R's syntax can be less intuitive for beginners, and it is less versatile than Python for tasks outside of pure statistical computing. Let's not forget to mention a very important one that we've seen already.

SQL

We have seen SQL (Structured Query Language), and at this point you're probably quite adequate with SQL. But it shouldn't miss from this small summation of great choices of programming languages for data analysis. SQL, is a domain-specific language used in programming for managing and manipulating databases. SQL is great for querying and extracting data, and it's used by most companies due to the ubiquity of SQL databases. It is often used in combination with other languages like Python for database-related functions, using connectors. Here is an example of a basic SQL query that displays a table of all records in the Employees table where an employee is more than 30 years old:

```
SELECT * FROM Employees WHERE Employee_Age > 30;
```

Despite its efficiency in handling databases, SQL is confined to database-related tasks. Unlike Python, it's not a general-purpose language, and unlike R, it's not suitable for complex statistical analysis, and thus it cannot be used for a wide array of programming tasks. However, you'll end up needing it often when you encounter SQL databases.

Julia

Julia is a high-performance programming language for technical computing that addresses performance requirements for numerical and scientific computing while also being effective for general-purpose programming. Its syntax is similar to those of other technical computing languages. Here is how you might create a and iterate over a list in Julia:

```
numbers = [1, 2, 3, 4, 5]
for item in numbers
    println(item)
```

However, Julia is a newer language, and its ecosystem is less developed than other languages like Python and R. Additionally, it might be too advanced to use for simple data analysis.

MATLAB

Matlab is a high-level language and interactive environment for numerical computation and programming. It is prevalent in academia and engineering, where it is used for tasks such as signal processing, image processing, and simulations. This is how you can iterate through a vector with a loop in Matlab:

```
my_vector = [1, 2, 3, 4, 5];
for i = 1:length(my_vector)
    disp(my_vector(i));
```

However, Matlab's cost can be a disadvantage compared to open-source options like Python and R. Furthermore, it can be limited outside its defined use cases. So what's the best language? Well, there is no such thing as a "best programming language". The choice of programming language largely depends on the specific task at hand, your and your team's expertise, and the existing technological infrastructure of your organization. We'll choose Python here, because it's very popular in the data analysis field and as beginner friendly as it gets. Before we dive into Python, let's explore the command line interface first. This is an essential tool to "look like a cool (ethical) hacker". Just kidding (though you will look cool), but it's widely used by programmers because it's simple yet powerful to use and it helps you to automate certain tasks as well.

Working with the Command Line Interface (CLI)

The Command Line Interface (CLI), often called the terminal on MacOS and Linux or the command prompt on Windows, is a potent tool for interacting with your computer. It allows you to execute commands by typing and directly communicating with your computer's operating system using a keyboard instead of the mouse. You can see an example of the command prompt in Figure 10.1.

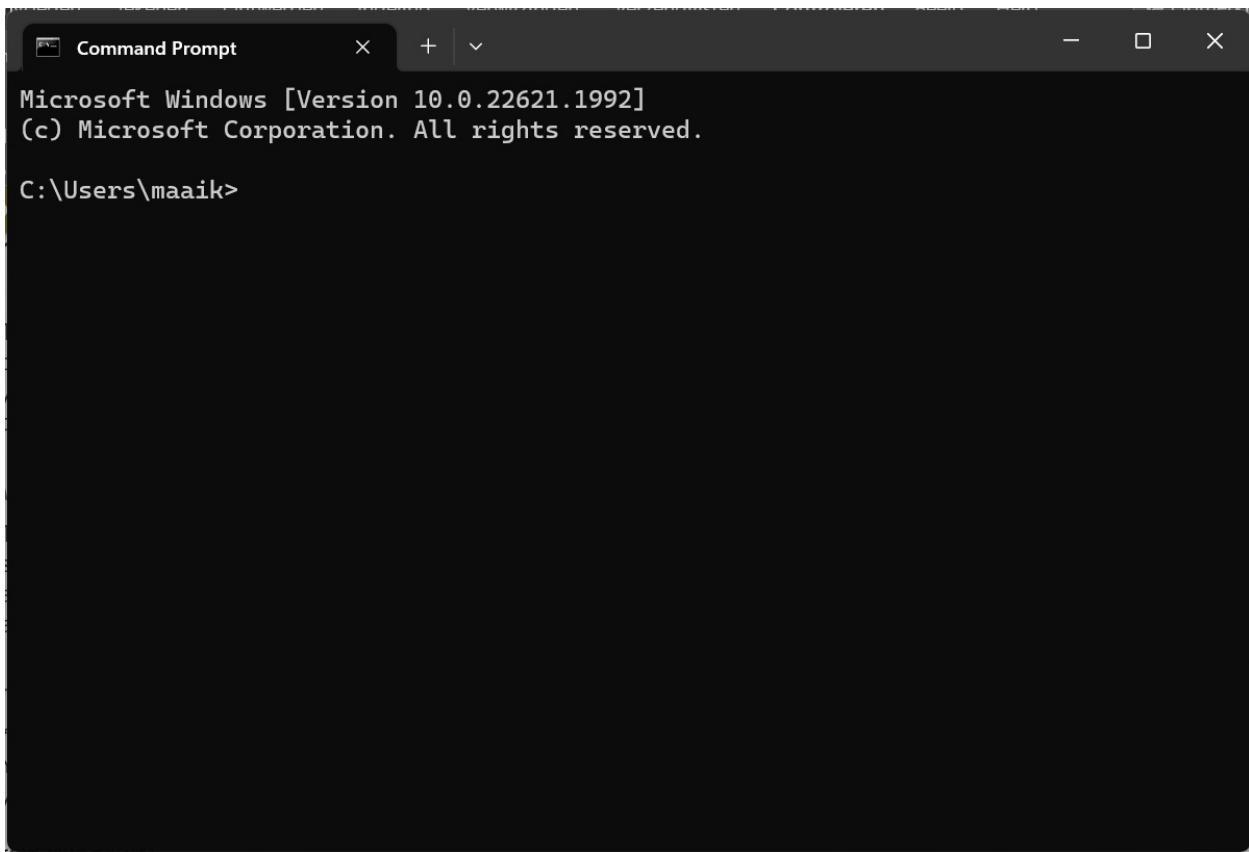


Figure 10.1 – Command prompt (CLI) on Windows

Command Line Interface (CLI) vs Graphical User Interface (GUI)

The CLI is a user interface where users can type commands to perform specific tasks. This starkly contrasts the Graphical User Interface (GUI) applications (anything from your desktop apps to browsers and even games)

that most computer users are familiar with. On the GUI tasks are performed by interacting with visual elements like buttons and menus typically with the use of the mouse (though commonly possible with the keyboard).

Accessing the CLI

Let's make sure that you can open the CLI on your machine. How to access it depends on your operating system and what you've installed exactly, but let's give you some general clues:

- Windows: You can use the Command Prompt or PowerShell, both of which serve similar purposes. You can find these by searching in the start menu or by typing 'cmd' or 'PowerShell' in the search box or run dialog box (windows button + r).
- MacOs: You can open the Terminal application, found in the "Utilities" directory in "Applications", but you can also quickly search for it with Spotlight (CMD + Space) by typing: "terminal".
- Linux: Typically, you should be able to type ctrl + alt + t to open up the terminal.

Now that we've opened up the CLI, let's see what we can use it for.

Typical CLI tasks

The CLI can perform almost any task that can be done through the GUI, including but not limited to the following common tasks:

- Navigating the file system
- Creating, deleting, and modifying files and directories
- Running scripts and applications
- Installing and managing software
- System administration
- Network troubleshooting
- Opening and editing files

It's very common to use the CLI for and during programming. Of course, the GUI is user friendly and chances are that you have a lot of experience with it

as opposed to the CLI, but let's see why we'd be using the CLI instead, especially during programming.

Using the CLI for programming

There's quite some good reasons to interact with your computer using the CLI during programming tasks. Let's go over them. First and foremost, efficiency. Tasks that take multiple click in a GUI, can often be done with one single command in the CLI. Next one is automation. If you need to do series of manual tasks often, you can group those together as CLI commands in a script. This ways, you can automate repetitive tasks to save time and enhance workflow efficiency. Versatility is also a major one. The CLI has a broad range of utilities that aren't available or are harder to access in a GUI, like package management, network diagnostics, and text manipulation. And the last one to mention here is control. The CLI allows for more precise control over tasks, as it doesn't need to abstract or simplify commands like a GUI. Understanding the CLI can be challenging at first, but it is a fundamental programming and data analysis skill. In the following sections, you'll see how we'll use the CLI to set up our Python environment. We're not going to talk about using the CLI in general, but this is definitely something to consider as a next step of learning outside of what's covered in this book.

Setting up your system for Python programming

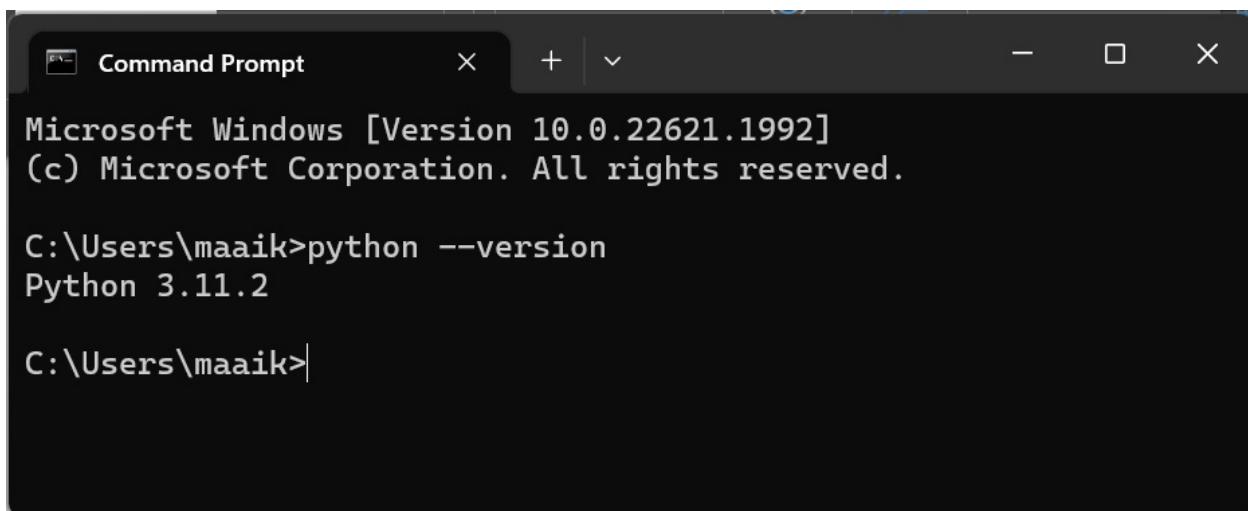
Setting up a Python programming environment is quite simple on any operating system. Below, we'll walk through the setup process for MacOS, Linux, Windows, and cloud-based development environments, such as Colab or Jupyter Notebook. Before we set up the system, let's make sure Python is not already installed.

Check if Python is installed

On some systems, especially Linux and MacOS, Python is installed already. You can check this with the following steps:

1. Open the terminal or command prompt.

2. Type: `python --version`
3. Hit enter.
4. Check the output. If it shows a Python version starting with a 3 (or in the future possibly higher), you don't need to set up your system. If it shows version 2 or something like command not recognized, you will probably have to setup your system. Let's make this sure with step 5.
5. If it didn't show a Python version starting with a 3, make sure that typing `python3 --version` and enter, is also not printing. Especially for MacOS and Linux this can be the case. You can see examples of systems with python installed for windows on *Figure 10.2*, for MacOS and Linux this is very similar.



A screenshot of a Windows Command Prompt window titled "Command Prompt". The window shows the following text:
Microsoft Windows [Version 10.0.22621.1992]
(c) Microsoft Corporation. All rights reserved.

C:\Users\maaik>python --version
Python 3.11.2

C:\Users\maaik>

Figure 10.2 – Command prompt output when Python 3 is installed

Depending on the outcome, let's set up your system or skip ahead to the testing setup section.

MacOS

If Python 3 is not installed on a MacOS, then follow these steps:

1. Install Homebrew if you didn't ever install that. Open your Terminal and type the following command and hit enter:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

The terminal should return a success message once Homebrew is installed.

1. The next step is to install Python using Homebrew. This can be done by typing the following command on your Terminal (after installing Homebrew):

```
brew install python
```

It will download for a minute or maybe two, and then displays a success message. Close the terminal and open it again, and see if this time `python --version` or `python3 --version` does show the correct version.

Linux

On Linux (depending on the exact Linux version), use the following commands:

```
sudo apt-get update
```

This is to update the package manager itself. After that, install the version of Python you want, for example:

```
sudo apt-get install python3.11
```

After installation, you can check the Python version again to confirm that the installation was successful.

Windows

1. Install Python. Download the latest version of the Python installer from the official website: (<https://www.python.org/downloads/windows/>). For example 3.11.4 in *Figure 10.3*. During the installation process, make sure to check the box that says "Add Python to PATH".
2. Once installed, open the command prompt and check the Python version to confirm the installation with the following command, followed by an enter:

```
python --version
```

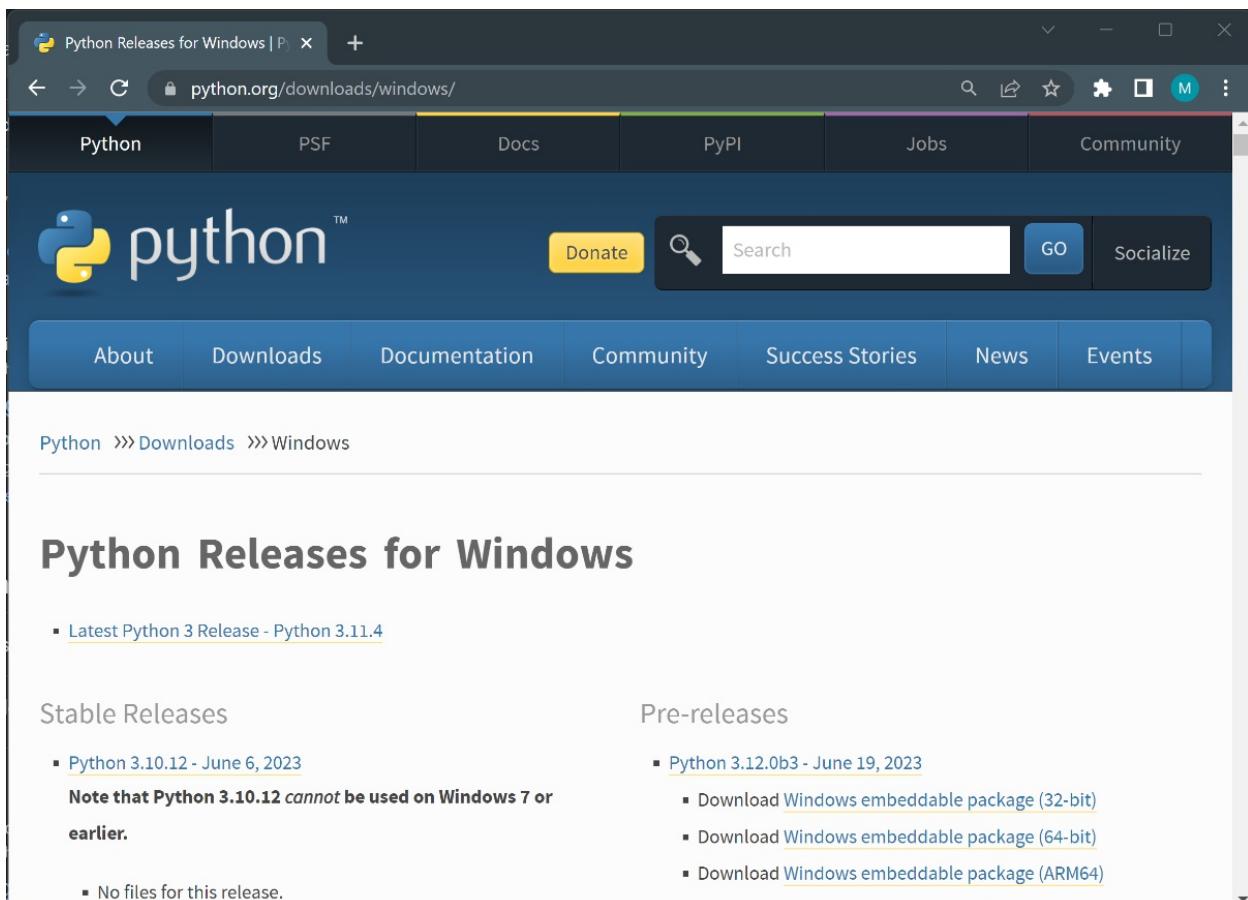


Figure 10.3 – Windows Python download page

Browser (cloud-based)

If you don't have a computer with Linux, MacOs or Windows, not to worry. There are great options for those with only access to cloud-based Python platforms. Some great options are Google Colab or Jupyter Notebook via Binder. They require no installation and allow direct access to Python from the browser instantly with no processor or storage overhead.

Google Colab

Go to Google Colab's website (<https://colab.research.google.com/>) and sign in with your Google account. This is the only requirement. You can start writing Python code immediately by creating a new notebook.

Jupyter Notebook via Binder

Visit the Binder website (<https://mybinder.org/>), enter the URL of a GitHub repository with a Jupyter notebook, and click "Launch". Binder will create a virtual environment where you can run the notebook.

Testing the Python setup

Let's go ahead and see if we managed to successfully install Python to our system. In order to do that we're going to create a file. You can do this with the command line, make sure to navigate to the right folder and use the command for the type of terminal that you have (different for command prompt on windows than for the terminal on MacOS). Or just anywhere on your computer, you can create a file with the GUI, call the file "helloworld.py". Open the file, for example with Notepad orTextEdit, but you can also use a more advanced editor such as Visual Studio Code. Add the following content:

```
print("Hello world")
```

After this, make sure to save file.

Hello world

It is common to always call the first program you write in a language hello world. All it does, is output the text hello world on the screen. This is to test the setup and typically the first milestone in a new programming language.

Next, we need to get to this folder in the command line. On Windows, and MacOs, you can right-click on the folder that contains the "helloworld.py", and select an option like "open in terminal". This opens the folder that contains the file in the terminal. In *Figure 10.4* you can see what this looks like for Windows.

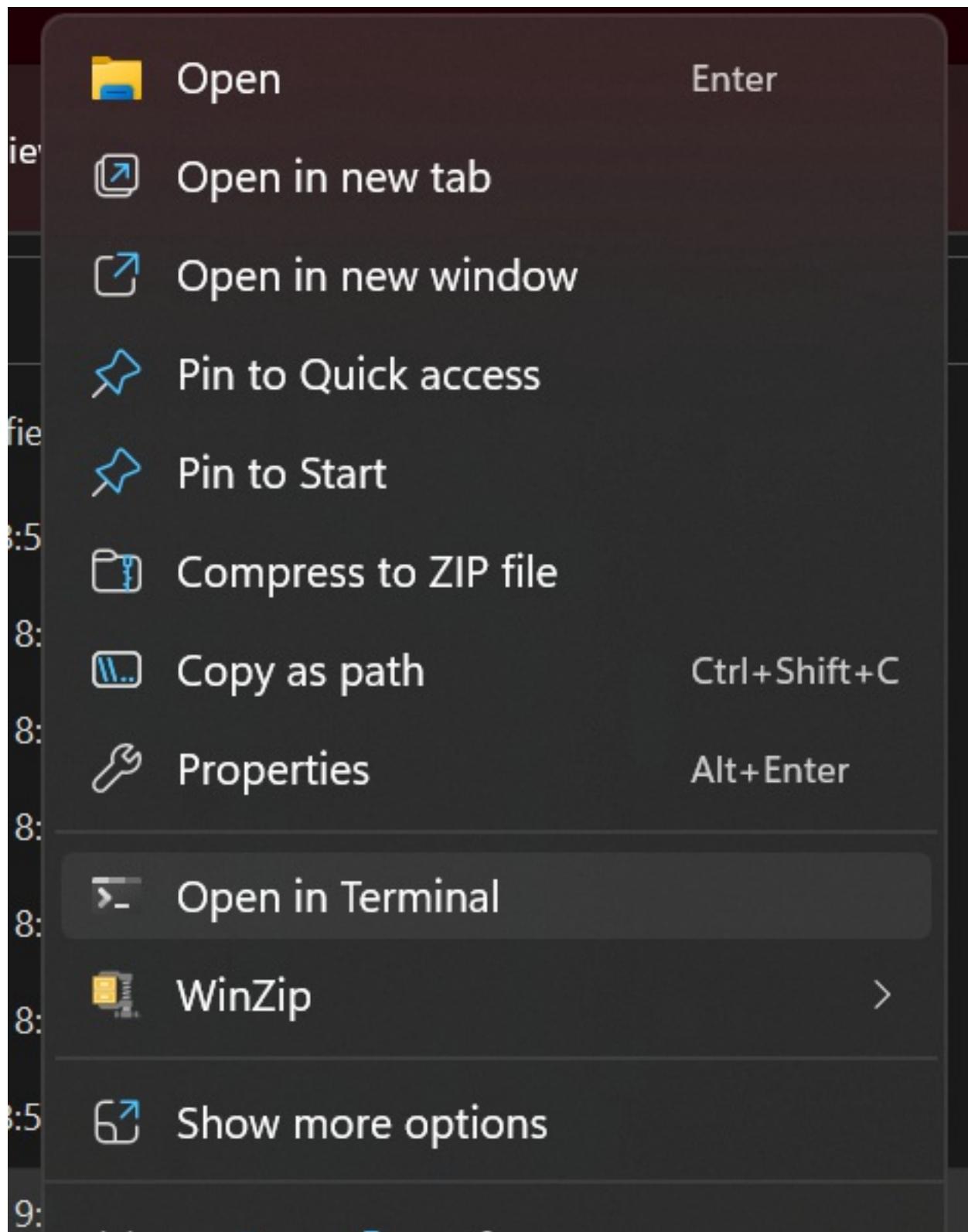


Figure 10.4 – Open in Terminal (third from the bottom)

You can check the path that shows in the command prompt on windows, and type “pwd” and hit enter on Linux and MacOS. If that’s indeed the right folder, go ahead and type one of the two following options followed by an enter:

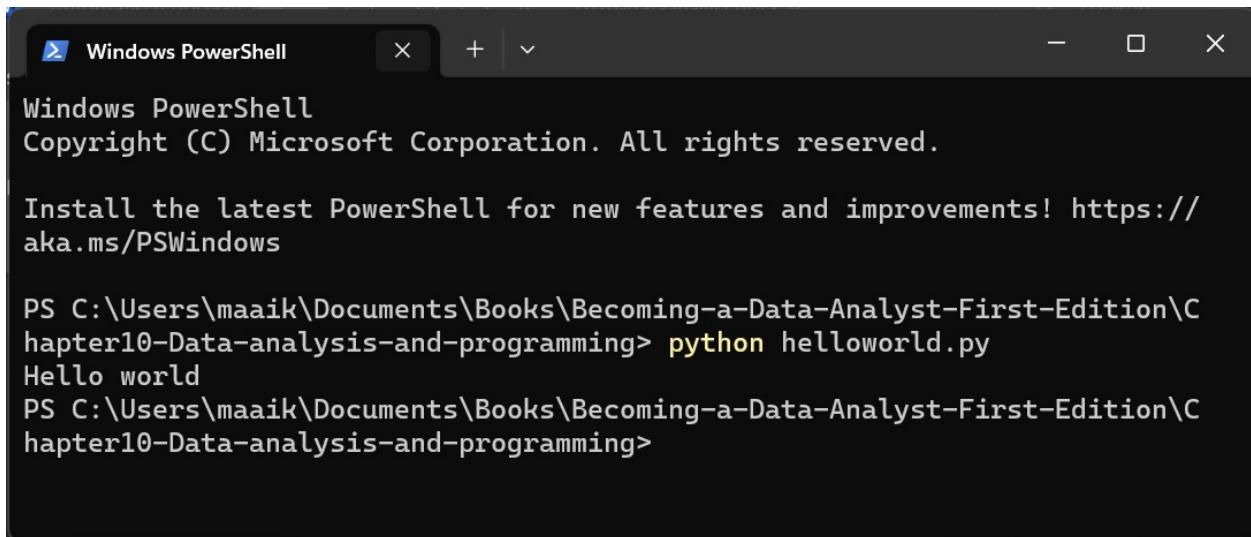
- If you got Python version 3 with `python --version`:

```
python helloworld.py
```

- If you got Python version 3 with `python3 --version`:

```
python3 helloworld.py
```

- The output should be, “Hello world”, like in *Figure 10.5*.



A screenshot of a Windows PowerShell window titled "Windows PowerShell". The window shows the standard PowerShell welcome message and then executes the command `python helloworld.py`. The output is "Hello world".

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\maaik\Documents\Books\Becoming-a-Data-Analyst-First-Edition\Chapter10-Data-analysis-and-programming> python helloworld.py
Hello world
PS C:\Users\maaik\Documents\Books\Becoming-a-Data-Analyst-First-Edition\Chapter10-Data-analysis-and-programming>
```

Figure 10.5 – Hello world output on Windows

If it says something like: no such file or directory. You’re not in the folder that contains `helloworld.py`. Make sure to right click on the folder that contains the file when you open the terminal, or navigate there using the command line (you can google the `cd` command to learn how to change directory (hence, `cd`) this if necessary).

Python use cases for CleanAndGreen

Now that you went to the effort of setting up your environment, let's assure you it was worth it. Python has earned its reputation as the cornerstone data analysis language for several reasons. The fundamentals of its syntax are easy to grasp, even for beginners. Next to that, Python has an amazing ecosystem of libraries specifically designed for data analysis and scientific computing, such as Pandas, NumPy, and SciPy. These libraries provide pre-built functions that simplify tasks like data cleaning, transformation, and statistical modelling. Python has an amazing strong community base. If you run into issues with Python and search for them on the internet, you'll find that other people had that issue and there are plenty of possible solutions for your issues out there. Let's explore four use cases to highlight the versatility and features of Python, which would excel (pun intended) in the data analysis for CleanAndGreen.

Data Cleaning and Preparation

In Python, the Pandas library provides numerous functions to clean and prepare data. In the case of CleanAndGreen, data collected on waste generation and recycling might be messy or inconsistent. You couldn't analyze this data in this form. You have to use Python to standardize the data formats, handle missing data, and detect outliers – often called data preprocessing. So that step, can be done with Python!

Data Visualization

CleanAndGreen might want to visualize the waste generation and recycling trends over time across different cities. In Python, there are many libraries for data visualization, some famous ones are Matplotlib and Seaborn, which can be used to create a variety of informative plots and graphs. We actually will not see how to use those in this book, but this is a very relevant use case that we will definitely recommend when you're passed your rookie stage in Python!

Statistical Modeling

CleanAndGreen wants to determine the factors that affect recycling rates in different neighborhoods. With Python, you can use several libraries for this

purpose. Some common choices would be statsmodels or scikit-learn. These libraries can be used to build statistical models and perform hypothesis tests.

Predictive Modeling/Machine Learning

Due to the growing inflation, CleanAndGreen might want to predict future recycling rates based on past trends. This is much more complex to do than the above use cases. Still, Python's powerful machine learning libraries, such as sci-kit-learn and TensorFlow, can be used to train predictive models efficiently and without much effort. Again, this is beyond the scope of the book. But definitely something that is a familiar friend for the more seasoned data analyst.

General remarks on Python

We love Python and we choose it for a reason. It's a great place to start with programming! Python is a multi-purpose language that works with everything, from scripting and automation to web development and machine learning. It is a one-stop shop for data analysts who may need to wear many hats. However, Python may not always be the ideal choice. It is an interpreted language, which means it can be slower to execute (in terms of performance) than compiled languages such as R, Java, C# and C++. Compiled means that the computer (sort of) processes all of the script and turns it into something executable before starting to run the program. A Python program is translated to something that the computer understands and runs at the same time. Therefore, Python is a little slower than a compiled counterpart. Definitely not our concern for now, but it might not be suitable for high-performance computing or real-time analytics applications.

Summary

In this chapter, we explored the need for programming in data analysis, illuminating its necessity in our data-driven age. You should now understand why programming is crucial for any data analyst, and in later sections, we will show how to work with Python a leading language in this field. With the help of the CleanAndGreen startup example, we outlined how programming

aids in data collection, handling large volumes of data and performing intricate data analysis to derive insights. By mastering programming, one can achieve the potential of the data and convert it into actionable, data-driven decisions. After that, we provided an overview of the different programming languages utilized in data analysis. We compared and contrasted popular languages for data analysis, such as Python, R, SQL, Julia, and MATLAB, highlighting the pros and cons of each and offering very small code snippets to showcase their syntax and usability differences. Our focus then shifted to Python – one of the most popular programming languages. We explained why Python is often the choice for data analysis due to its simplicity, flexibility, and its vast library support. Next, we transitioned into the practical aspect of setting up your Python programming environment. We provided step-by-step instructions for all operating systems as well as modern cloud-based development environments like Google Colab. After each step, we ensured that you understood the expected results, setting you up for success in your Python programming journey. And, of course, we also introduced you to Command Line Interface (CLI) and drew comparisons between the CLI and GUI, highlighting the advantages of using the CLI for programming tasks. We ended with some practical use cases we could use Python for in the light of supporting CleanAndGreen. As we conclude this chapter, we've equipped you with the foundational knowledge you need to apply in the upcoming chapters, where we will use Python to solve real-world data problems, given our CleanAndGreen example. Get ready to code!

11 Introduction to Python

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



Welcome to the next phase of your learn to code journey. At this point, you've already understood why programming is necessary for data analysis. It's time to get your hands dirty with some actual coding. We've chosen Python as a first language for you. It's a relatively easy syntax and it is extremely common in the data analysis field. We could write a whole series of books on Python to learn you all the ins and outs. The approach we chose here is to give you a quick overview of the language and how you can use it for data analysis. This is by far not a comprehensive guide, but rather a first look. This is what we're going to cover in this chapter:

- The basic syntax of Python
- Data types in Python
- Indexing and slicing in Python
- Control flow structures
- Python's data structures
- Functions

Indeed, there's a lot to cover. Not to worry, we will take it step-by-step, and by the end of this chapter, you will have a solid foundation of Python. At that point you're ready to look at some libraries of Python to unlock new levels of data analysis and problem-solving capabilities. We'll continue to lean on our greenFuture case study to bring these concepts to life. So, let's go!

Understanding the Python Syntax

The **syntax** refers to the set of rules that specify how to write the code for the program. It includes how to structure your programs and how different parts of your program should be arranged. It's the basic rules we need to write our Python programs. A program consists of **statements**. And every statement should go on its own line in the Python file. Let's look at one of the most basic statements first, the print statements.

Python files

In order to follow along with the code snippets in this chapter, you'll have to type them in a Python file. You can run this file as demonstrated in the previous chapter.

Print Statements

The print statement in Python is our go-to tool for displaying output to the console. Let's start with a basic example. Imagine if you just started your day at greenFuture and you want to greet your team with a little Python script. You can use a print statement for that.

```
print("Good morning, greenFuture team!")
```

When you run this script, the output will be as follows:

```
Good morning, greenFuture team!
```

And that's how to do the most basic print statement. In your code, you can include comments to explain what the code is doing and make it easier to read. Let's see how to do that next.

Comments

It's a good practice to explain what your code is doing. Especially when the code and logic gets more complicated and you couldn't tell what it does at first glance. We can use comments to do this.

Comments are ignored by the Python interpreter, and they can be used to explain difficult snippets in the code and to add some documentation to it. So comments don't influence the outcome of the program. Our previous code snippet is of course very easy, but for demonstration purposes we'll add a comment on top, so that you can see what that looks like.

```
# Printing a greeting for the team
print("Good morning, greenFuture team!")
```

The outcome will remain exactly the same. We'll use comments in our code snippets to explain certain parts every now and then throughout the examples. This Python script will always print the exact same thing. It would be great if it would "vary" a bit more. Well, that's where variables come in the mix! Let's have a look at them.

Variables

Variables are like containers for data. They have a name and they can hold a specific value. Variables allow us to store and manipulate data in Python programs. For instance, let's say you're tracking the amount of plastic waste greenFuture has recycled in a week. Here's how you might use a variable to do that.

```
plastic_waste_recycled = 1300 # weight in kilograms
print(plastic_waste_recycled)
```

In this example the name of the variable is `plastic_waste_recycled`. The value of the variable is 1300. After that, we have a comment that explains what the variable holds. Then on the next line, you can see that we can print the variable as well. When we print a variable, it will not print the name of the variable, but it will be evaluated to its value. That's why the output will be:

```
1300
```

Because the variable holds the value of 1300. We can also perform operations on variables, let's see a few basic examples next.

Operations on variables

Let's say the team at greenFuture manages to recycle an additional 700 kilograms of plastic waste. We can update our variable like so:

```
plastic_waste_recycled = 1300 # initial weight in kilograms  
plastic_waste_recycled += 700 # additional weight recycled  
print(plastic_waste_recycled)
```

On the second line, we use the `+=` symbol. This means that we increment the variable with what is on the right side of the operation. It's the same as:

```
plastic_waste_recycled = plastic_waste_recycled + 700
```

And hence the output will be:

2000

Now, let's make it a bit more interesting. Suppose you want to print the progress of a waste recycling task. You can include variable values in your print statements combined with regular text. You can do it like this:

```
task_percent_completed = 45  
print("The waste recycling task is", task_percent_completed, "% completed.")
```

Output:

The waste recycling task is 45% completed.

Now, let's take it a step further. Suppose we want to print a formatted string that includes variable values. We can use f-strings for that. You might be wondering, but why do I need to use f-strings if I can use the syntax above, which is more straightforward? It's because it allows you to insert multiple variables into a string seamlessly. You can achieve it with the previous syntax as well. However, the below one is very common and you need to be able to read it:

```
task_name = "Aluminium recycling"  
task_percent_completed = 45  
print(f"The {task_name} task is {task_percent_completed}% completed.")
```

We start with the f for format before the quotes. That tells Python that anything between { and } needs to be replaced with the value of a variable. And that's why the output will be:

```
The Aluminium recycling task is 45% completed.
```

Suppose we want to keep track of different types of waste recycled. This is how we could write it.

```
plastic = 1300  
glass = 700  
aluminium = 800  
print(plastic, glass, aluminium)
```

This will output:

```
1300 700 800
```

There is a common alternative. Python allows us to assign multiple variables at once. Here's how we could do it:

```
plastic, glass, aluminium = 1300, 700, 800 # weights in kilograms  
print(plastic, glass, aluminium)
```

Above, three different variables are assigned three different values, separated by commas. Note that the result is the exact same as when you declare and assign each variable in separate lines. We have already seen that we can add two numbers, but there are more allowed operations. Let's see some more of them.

Operators and Expressions

In Python, **operators** are symbols that carry out arithmetic or logical computations. The value or variable that the operator works on is called the **operand**. We can use these operators to create **expressions**. Expressions are combinations of variables, literals (direct values such as 5, "hello", 3.2), and operators. There are different types of operators, such as mathematical, comparison and logical operators. We'll start with the basic mathematical operators.

Mathematical operators

The basic **mathematical operators** in Python are: +, -, *, / for addition, subtraction, multiplication, and division, respectively. Let's suppose at greenFuture, you are tasked with calculating the total amount of waste recycled in a week. Here's how you might do that in Python:

```
# weights in kilograms
plastic = 1300
glass = 700
aluminium = 800
total_recycled = plastic + glass + aluminium
print(total_recycled)
```

At the end we print it. This would not be necessary for the task of calculating the total. However, it helps us to verify the result. Here's the output:

```
2800
```

This operation added the values stored in the variables `plastic`, `glass`, and `aluminium` together to give us the total weight of recycled waste. Let's try something more complex. Suppose the weight of waste to be recycled for the next week is expected to increase by a factor of 1.15. How could we calculate the expected weight for each type of waste?

```
# weights in kilograms
plastic = 1300
glass = 700
aluminium = 800
# increase factor
increase_factor = 1.15
# calculating expected weights of each type of waste
expected_plastic = plastic * increase_factor
expected_glass = glass * increase_factor
expected_aluminium = aluminium * increase_factor
print(expected_plastic, expected_glass, expected_aluminium)
```

These calculations use the multiplication operator (*) to increase each weight by a factor of 1.15. These numbers can help the greenFuture team understand the composition of their recycling impact. There are some surprises in the output though:

```
1494.999999999998 804.999999999999 919.999999999999
```

You can see that there are a lot more decimals after the floating point than you would expect. For example, $1.15 * 1300 = 1495$. But we get 1494.999999999998. What is going on? This is a problem that all programming languages have to deal with. It's related to translating a decimal number to a binary number that the computer inherently works with. Think of it like writing $1/3$ in decimal numbers; you can never be 100% as precise as stating $1/3$, no matter how many decimal points you add, such as 0.33333333333333333333. In Python, there's a library called decimal that helps mitigate these issues by providing arbitrary precision arithmetic. Now, to make our output more readable, we can format the floating-point numbers to show only two decimal places, we can use this print statement in order to do that:

```
print(f'{expected_plastic:.2f}, {expected_glass:.2f}, {expected_aluminium:.2f}')
```

The output will now be:

```
1495.00, 805.00, 920.00
```

We also have operators to compare operands. Let's have a look at those now.

Comparison operators

Moving on to **comparison operators**. Comparison operators are crucial in decision-making within your programs. They compare values and return a boolean result. The basic comparison operators in Python are: == (equal to), != (not equal to), < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to). Let's continue with our GreenFuture example. Suppose you have a target to recycle at least 3000 kilograms of waste every week. You can use a comparison operator to check if you've met the target:

```
# weights in kilograms
plastic = 1300
glass = 700
aluminium = 800
# calculate total
total_recycled = plastic + glass + aluminium
```

```
# checking if the target is met
is_target_met = total_recycled >= 3000
print(is_target_met)
```

This will output `False` since 2800 is less than 3000. These operators are fundamental in controlling the flow of your program and making decisions based on certain conditions. Let's make it more interesting and add logical operators to our skillset next.

Logical operators

Logical operators allow us to connect two expressions. The result of the logical operator is a **boolean**, meaning that it's a true or false value. You might have encountered some of them in basic math classes already. The primary logical operators in Python are `and`, `or`, and `not`. Below, this example expands on the one above and adds logical operators to make things more interesting. Using logical operators, you can check if certain conditions are met. In this case, we apply conditions on the weight of each type of waste.

```
# weights in kilograms
plastic = 1300
glass = 700
aluminium = 800
# increase factor
increase_factor = 1.15
# calculating expected weights of each type of waste
expected_plastic = plastic * increase_factor
expected_glass = glass * increase_factor
expected_aluminium = aluminium * increase_factor
# checking if any waste type has a weight greater than or equal
# to 1500 or a weight less than or equal to 600
check_condition_1 = expected_plastic >= 1500 or expected_glass <
= 600 or expected_aluminium >= 1500
print(check_condition_1) # Output: False
# checking if any waste type has a weight between 1000 and 1200
check_condition_2 = 1000 <= expected_plastic <= 1200 or 1000 <=
expected_glass <= 1200 or 1000 <= expected_aluminium <= 1200
print(check_condition_2) # Output: False
# checking if all waste types have an expected weight greater than 800
check_condition_3 = expected_plastic > 800 and expected_glass >
```

```
800 and expected_aluminium > 800
print(check_condition_3) # Output: True
# checking if not all waste types have an expected weight greater than 800
check_condition_4 = not (expected_plastic > 800 and expected_glass > 800 and expected_aluminium > 800)
print(check_condition_4) # Output: False
```

This is what the output looks like:

```
False
False
True
False
```

In this case, we use logical (or, and, not) and comparison operators (\leq , \geq) in conjunction with the arithmetic operators we already discussed. The print statements output the result of the logical expressions, which are boolean values (`True` or `False`). As you can see, understanding Python's syntax and using print statements, variables, operators, and expressions is powerful. These are the foundational concepts that we'll be building upon in the upcoming sections. You might have noticed the dynamicity of Python in choosing data types as it chose float for some operators despite an integer value. So, let's dive more into the data types available in Python to help store data more efficiently.

Exploring Data Types in Python

In Python, data can be stored in different types of “containers”. These different types of containers are referred to as datatypes. So far, we've worked with integers, floats, strings and booleans. Let's dive deeper into these types.

Strings

Strings are sequences of characters. They are used to represent the textual data in Python. To understand better, let's see an example:

```
city = "New York"
print(city)
```

Output:

New York

We have created a string variable `city` with the value `New York`. Now, let's use this in a slightly more complex situation. Suppose we want to display a message about the city where `greenFuture` operates:

```
city = "New York"
message = "GreenFuture operates in " + city
print(message)
```

Output:

GreenFuture operates in New York

In this example, we used the `+` operator to concatenate two strings. There are many more things you can do to manipulate strings in Python which you will explore if you continue your Python journey.

Integers

In the previous sections, we've already used **integers**, which represent whole numbers. But let's create another integer variable representing the number of `greenFuture` employees to illustrate this data type:

```
employees = 50
print(employees)
```

This will output:

50

As you can see, integers are for whole numbers. We can also have decimal point numbers of course, that's where the float data type comes in.

FLOATS

FLOATS represent real numbers, that means numbers with a decimal point. Suppose we want to represent the average amount of waste generated per

person in the city as part of the data collection by greenFuture:

```
average_waste_per_person = 0.87 # in kilograms  
print(average_waste_per_person)
```

Output:

```
0.87
```

As you can see, the value is a floating point number. Floats can also be the result of an operation on two integers. For example, $2 / 3$ will result in the float 0.6666666666666666.

Booleans

In Python, there is also another data type known as **boolean**. It stores values as either true or false. For example, expanding on the example above for integers, we can use a boolean data type to check whether the employee count is above a specific number.

```
employees = 50  
is_greater = employees > 40  
print(is_greater)
```

And since 50 is bigger than 40, the output will be:

```
True
```

Booleans are used a lot for decision making in the code, and we'll use them for control flow statements that we will discuss soon, such as if else statements and loops. Let's talk about how to change the data type first.

Type Conversion

Python also supports explicit type conversion of data types. Before we demonstrate that, it would be wise to print out the existing data type of your variable first. This can be done with `type()`.

```
average_waste_per_person = 0.87  
print(type(average_waste_per_person))
```

Output:

```
<class 'float'>
```

As you can see, 0.87 is of type float. Now, let's improve on this example and convert the variable's data type. Here's how you can do that:

```
average_waste_per_person_int = int(average_waste_per_person)
print(average_waste_per_person_int)
print(type(average_waste_per_person_int))
```

Output:

```
0
<class 'int'>
```

In the examples above, the first one prints the data type of the variable that stores the average waste per person. The second example goes a step further and explicitly converts the data type to an integer, stores the result in a new variable, and then prints the value and type of this new variable. This value is now of type int.

Something else happened though, the value changed from 0.87 to 0. Let's talk about this. When a float value is converted to an integer in Python using the `int()` function, the decimal portion of the float is truncated, not rounded. This means that everything after the decimal point is simply discarded, and only the whole number part is kept. In the case of 0.87, the whole number part is 0, and the decimal part .87 is discarded, resulting in 0. This behavior is consistent regardless of whether the decimal part is less or more than .5. It's important to be aware of this behavior as it can lead to unexpected results if you're used to work with rounding instead of truncation. A popular use case of type conversion is when you take a user input for a numerical value. In such a case, Python defaults to storing the results as a "string" type rather than a numerical value. So, you must convert the type of data to use the actual numerical value in different operations.

Here's a code snippet that demonstrates that:

```
# Collecting daily recycling input from the user
daily_recycling_input = input("Enter the number of plastic bottles recycled today: ")
# Printing the type of daily_recycling_input to show it's a str
```

```
ng
print(type(daily_recycling_input)) # Output: <class 'str'>
# Converting the string input to an integer
daily_recycling_int = int(daily_recycling_input)
# Now we can perform numerical operations
weekly_recycling_total = daily_recycling_int * 7
# Printing the weekly recycling total
print(f"The total number of plastic bottles recycled in a week is: {weekly_recycling_total}")
```

This will output:

Enter the number of plastic bottles recycled today:

After this, it pauses. It waits for the user input. Let's suppose we enter 1 as input. It then stores 1 in the `daily_recycling_input`. And after that it prints:

```
<class 'str'>
The total number of plastic bottles recycled in a week is: 7
```

Let's go over what happens in the code snippet. We first collect the daily recycling number using the `input()` function. The `input` function pauses and waits for user input. We store the entered value in the `daily_recycling_input` variable. We then print the type of `daily_recycling_input` to demonstrate that it's a string. Next, we convert `daily_recycling_input` to an integer using the `int()` function, allowing us to perform numerical operations on it. Finally, we calculate the weekly recycling total by multiplying `daily_recycling_int` by 7, and print the result.

There are other types of conversion as well:

- `float()`: The `float()` function is used to convert a specified value into a floating-point number. This is useful when you need to perform decimal arithmetic.
- `str()`: The `str()` function is used to convert a specified value into a string. This can be necessary when you want to concatenate number data or when you need to output numeric data as a string.
- `bool()`: This function is used to convert a value into a boolean value. Any non-zero numeric value will be converted to True. The zero values

(0, 0.0) will be converted to False. Any non-empty string will be converted to True, and any empty strings will be converted to False.

- Now, let's demonstrate these conversions with a GreenFuture example:

```
# GreenFuture data
daily_recycling_kg = 350.5 # daily recycling in kilograms
weekly_goal_kg = 2450 # weekly recycling goal in kilograms
# Convert daily_recycling_kg to string for concatenation
daily_recycling_str = str(daily_recycling_kg)
print("Daily recycling: " + daily_recycling_str + " kg")
# Output: Daily recycling: 350.5 kg
# Convert weekly_goal_kg to float for decimal arithmetic
weekly_goal_float = float(weekly_goal_kg)
print("Weekly goal: " + str(weekly_goal_float) + " kg")
# Output: Weekly goal: 2450.0 kg
# Check if daily recycling goal is met and convert the result to
# boolean
is_daily_goal_met = bool(daily_recycling_kg >= (weekly_goal_flo
t / 7))
print("Is daily goal met? " + str(is_daily_goal_met))
# Output: Is daily goal met? True
```

This will output:

```
Daily recycling: 350.5 kg
Weekly goal: 2450.0 kg
Is daily goal met? True
```

This is how to convert data types from one type to another. There's more you likely want to do with data. Suppose you want to extract specific values from a string or textual data type in Python. How can you do that? With indexing and slicing, you can access, modify, and manipulate the data stored in strings and other sequence types. And that's exactly what we're going to talk about next!

Indexing and Slicing in Python

Indexing is used to access individual elements of data structures, like strings. Let's access the first letter of the city string variable we used above:

```
city = "New York"
first_letter = city[0]
```

```
print(first_letter)
```

As you can see, we need the square brackets to access the values in the string. This is what it will output:

N

You might be surprised that the N is at position 0. Python uses zero-based indexing, meaning the first element begins at index 0. Python also supports negative indexing so that you can print from the reverse side: -1 refers to the last character, -2 to the second last, and so on.

```
city= "New York"
second_last_letter = city[-2]
print(second_last_letter)
```

Output:

r

When you use negative indexing, you must remember that -1 refers to the last character (on the right-most side of the string). Here we take the second last character with index -2. That's the indexing part. With the use of the index, we can also perform slicing. **Slicing** is used to extract specific portions of a string. It's useful when you only need certain parts of a string and can dispose of the rest. The syntax for this uses a colon in between the starting character and ending character on either side of it. If no end index is specified, it continues until the end of the string:

```
city = "New Jersey"
selected_text = city[4:]
print(selected_text)
```

In this example we start at index 4 (the fifth character) and continue until the end of the string. This will be the output:

Jersey

Slicing and indexing can also be used on lists. Let's learn about lists and other complex data structures next.

Unpacking Data Structures

In Python, we often deal with data structures that can hold multiple items, like lists, dictionaries, sets, and tuples. This is great whenever you want to bundle different data records. Such as all the users, all products, and more. Let's explore each type in detail, starting with lists.

Lists

A **list** is an ordered collection of items. It's a mutable data structure, meaning you can change its elements after declaration. It also allows duplicate elements and elements of different data types in a single list. Like strings, access to list elements uses zero-based indexing. Let's create a list of the types of waste that greenFuture recycles:

```
waste_types = ["plastic", "glass", "aluminium"]
print(waste_types)
```

This will output:

```
["plastic", "glass", "aluminium"]
```

You can access individual elements of a list by their index, just like the characters of strings:

```
# Get the first item
first_item = waste_types[0]
print(first_item) # Output: 'plastic'
# Get the last item
last_item = waste_types[-1]
print(last_item) # Output: 'aluminium'
```

Discovering the Length of a List

In our journey with GreenFuture, it's often essential to know how many types of waste materials we are dealing with. Python makes this easy with the `len()` function. This handy function tells us the number of items in a list. Let's see it in action:

```
list_length = len(waste_types)
print(list_length) # Output: 3
```

With just a simple function, we now know there are three types of waste materials in our list. There are a lot of methods built-in on the list data type that . Let's explore them.

Commonly Used List Methods: A Closer Look

Lists in Python are equipped with a variety of methods to make our life easier. Let's delve into some of these methods.

Appending Items

When GreenFuture starts recycling a new material, we need to add it to our list. The `append()` method is made for exactly this:

```
waste_types.append("paper")
print(waste_types)
```

Here is the output:

```
['plastic', 'glass', 'aluminium', 'paper']
```

As you can see, after appending the list got extended with paper. This is how to add a single element. There's also a way of adding a full list of elements to the list.

Extending the List

Suppose we have a list of additional materials. We can add all these items to our existing list with the `extend()` method:

```
additional_waste_types = ["organic", "e-waste"]
waste_types.extend(additional_waste_types)
print(waste_types)
```

This will output:

```
['plastic', 'glass', 'aluminium', 'paper', 'organic', 'e-waste']
```

As you can see, both elements got added to our list, without having a list inside our list. This is how to make the list bigger, we can also make it smaller by removing elements.

Removing Items

If for some reason, we stop recycling glass, we can remove it from our list using the `remove()` method:

```
waste_types.remove("glass")
print(waste_types)
```

This is what will be printed:

```
['plastic', 'aluminium', 'paper', 'organic', 'e-waste']
```

As you can see, glass has been removed from our list. Our list is a little unorganized right now, let's see how to fix that with sort.

Sorting the List

Keeping our list of materials sorted helps in quicker access and better organization. The `sort()` method is here to help:

```
waste_types.sort()
print(waste_types)
```

And this is what that prints:

```
['aluminium', 'e-waste', 'organic', 'paper', 'plastic']
```

The list is now sorted A-Z, making it easy for us to see whether a certain waste type is present or not.

Counting Occurrences

Curious to know how many times a particular material appears in our list? The `count()` method has got us covered:

```
count_plastic = waste_types.count("plastic")
print(count_plastic)
```

And this is what the output will be:

1

Since all the elements are only present once in our list, the result will either be 1 or 0. Since plastic is present, it is 1. If we'd look for something that is not on the list, it would be 0. We can also figure out on which position an element on the list is on.

Finding the Index

If we want to know the position of a material in our list, we can use the `index()` method:

```
index_aluminium = waste_types.index("aluminium")
print(index_aluminium)
```

And this outputs:

0

Since aluminium is the first element on the list, the index of aluminium is 0. If the element is not on the list, it will throw a `ValueError`. When the error is not handled, the program will stop and crash. You'll learn about how to handle errors later.

Reversing the List

For a different perspective, we might want to look at our list in reverse. The `reverse()` method makes this an easy task:

```
waste_types.reverse()
print(waste_types)
```

This will print:

```
['plastic', 'paper', 'organic', 'e-waste', 'aluminium']
```

The output shows that the list is now in reversed order. It was sorted A-Z and now it's sorted Z-A. It simply takes the last element of the list and makes it the first element, and the second last the second, etc. These methods are just the tip of the iceberg, but they provide a solid foundation for manipulating and querying lists, making our data analysis at GreenFuture more insightful. While lists are one of Python's most widely used data structures due to their versatility, there are other data structures that you'll encounter often. The next one we'll discuss is the dictionary. This data structure stores values in key-value pairs.

Dictionaries

A dictionary is an unordered collection of key-value pairs. The key for each pair must be unique, and it is immutable after declaration. You can store values with any data type and access is provided through the keys, not indexing. For example, we might use a dictionary to store the amount of waste greenFuture recycles in each category:

```
waste_recycled_kg = {"plastic": 1200, "glass": 800, "aluminium":  
1500}  
print(waste_recycled_kg)
```

Output:

```
{"plastic": 1200, "glass": 800, "aluminium": 1500}
```

We can also access the amount of plastic recycled by using the key "plastic":

```
plastic_recycled = waste_recycled_kg["plastic"]  
print(plastic_recycled)
```

Output:

```
1200
```

Let's have a look some of the other common things we need to do with dictionaries in our day-to-day tasks as data analysts.

Discovering the Size of a Dictionary

Just like with lists, it's often crucial to know the size of our dictionary. The `len()` function comes to the rescue again:

```
dict_size = len(waste_recycled_kg)  
print(dict_size) # Output: 3
```

This will print 3. With a simple function call, we now know there are three key-value pairs in our dictionary. Let's see some built-in methods on dictionaries.

Commonly Performed Tasks for Dictionaries

Dictionaries in Python are equipped with a variety of methods to make our data analysis tasks smoother. Not everything we would need to do is done with a built-in method. Let's explore some of these common tasks and see how to do it.

Adding Items

When GreenFuture starts recycling a new material, we need to add it to our dictionary. Here's how:

```
waste_recycled_kg["paper"] = 900  
print(waste_recycled_kg)
```

Here is the output:

```
{'plastic': 1200, 'glass': 800, 'aluminium': 1500, 'paper': 900}
```

The dictionary is updated to also contain a key-value pair for paper. It is also possible to update an existing key-value pair.

Updating Items

Suppose the amount of aluminium recycled changes, we can easily update this value in our dictionary using the key and setting the new value. Here's how that is done:

```
waste_recycled_kg["aluminium"] = 1600
```

```
print(waste_recycled_kg)
```

This will output:

```
{'plastic': 1200, 'glass': 800, 'aluminium': 1600, 'paper': 900}
```

The value for aluminium now changed to 1600. Another common modification to a dictionary is removing a key-value pair. Let's see that next.

Removing Items

If for some reason, we stop recycling glass, we can remove it from our dictionary using the `pop()` method:

```
waste_recycled_kg.pop("glass")
print(waste_recycled_kg)
```

This is what will be printed:

```
{'plastic': 1200, 'aluminium': 1600, 'paper': 900}
```

The glass: 800 key-value pair has been removed. And our dictionary now consists of three key-value pairs. If you try to pop a key that is not in the dictionary, you'll get an error. You can check whether a certain key exists as well.

Checking for a Key

Before attempting to access or remove an item, it's wise to check if the key exists to avoid the error. The `in` keyword helps us here:

```
print("glass" in waste_recycled_kg) # Output: False
```

This will print `False`. The next step would be to use an if-else statement, we'll see how to create those soon!

Getting All Keys and Values

Sometimes, we might want to take a look at all the keys or all the values in

our dictionary. The `keys()` and `values()` methods are perfect for this:

```
print(waste_recycled_kg.keys())
# Output: dict_keys(['plastic', 'aluminium', 'paper'])
print(waste_recycled_kg.values())
# Output: dict_values([1200, 1600, 900])
```

These come in handy once we've seen some more control flow statements. What we've seen now is just a glimpse into the capabilities of dictionaries, but it's definitely a good place to start managing and querying data. While dictionaries are incredibly useful due to their key-value pair structure, there are other data structures in Python that offer different advantages. As we continue our exploration, we'll delve into sets next.

Sets

A **set** is an unordered collection of unique items. It is also a mutable data structure that supports mathematical set operators like **Union** but does not allow indexing or slicing. Suppose greenFuture operates in multiple cities, and we want to keep track of them. The cities will be unique items and we don't want duplicates, so we store them in a set:

```
cities = {"New York", "San Francisco", "Chicago", "New York"}
print(cities)
```

Output:

```
{"New York", "San Francisco", "Chicago"}
```

Although we added "New York" twice, it only appears once in the set because all items in a set must be unique, and it doesn't support duplicate values like in lists. Hence, it's useful for eliminating duplicates. Note the difference in the type of parentheses used. Sets use curly braces, while lists use square brackets. Let's talk about what we can do with sets.

Discovering the Size of a Set

Just like with lists and dictionaries, it's often necessary to know the size of our set. The `len()` function is our go-to for this again:

```
set_size = len(cities)
print(set_size) # Output: 3
```

With a simple function call, we now know there are three unique cities in our set. Let's see some methods that we'll commonly use for our data analysis tasks on sets.

Commonly Performed Tasks on Sets

Sets in Python are equipped with a variety of methods to make our data analysis tasks smoother. Let's explore some of these methods and keywords. As you'll see, a part of what we often need to do is similar to what we want for dictionaries and lists, but the way we need to do it differs a little bit.

Adding Items

When GreenFuture expands to a new city, we need to add it to our set. Here's how:

```
cities.add("Los Angeles")
print(cities)
```

Here is the output:

```
{'New York', 'San Francisco', 'Chicago', 'Los Angeles'}
```

As you can see, Los Angeles has been added to our set. In a similar way, we can remove cities as well.

Removing Items

If for some reason, GreenFuture ceases operations in a city, we can remove it from our set using the `remove()` method:

```
cities.remove("Chicago")
print(cities)
```

This is what will be printed:

```
{'New York', 'San Francisco', 'Los Angeles'}
```

In the output you can see that Chicago is not longer on our set. When we try to remove something that's not in our set, we'll get an error. Luckily, we can check if an element exists in our set first.

Checking for an Item

We don't want to get errors. That's why it's wise to check if the item exists, before attempting to remove it. The `in` keyword helps us here:

```
print("Chicago" in cities)
```

Output:

```
False
```

In this case, Chicago is not in the set and the statement evaluates to False. Sets allow for some special mathematical operations. Let's see those next.

Set Operations

Sets support various mathematical operations that can be very useful in data analysis. Let's explore a few of the available operations.

Union

The first one we'll discuss is union. Union creates a new set combining all unique items from two sets. In order to make sure you understand it, we'll repeat the current values of the sets.

```
cities = {"New York", "San Francisco", "Los Angeles"}  
other_cities = {"Boston", "Miami", "New York"}  
all_cities = cities.union(other_cities)  
print(all_cities)
```

This is what it will print:

```
{'New York', 'San Francisco', 'Los Angeles', 'Boston', 'Miami'}
```

In the output are all the cities that are in the `other_cities` and `cities` sets. We can also only get the elements they have in common. This is done with the intersection method.

Intersection

We have just seen how to use union to get all the unique elements in both sets. We can also find the common elements in two sets. This is done with the intersection method. Here's an example of how to do that:

```
cities = {"New York", "San Francisco", "Los Angeles"}  
other_cities = {"Boston", "Miami", "New York"}  
common_cities = cities.intersection(other_cities)  
print(common_cities)
```

We repeated the values of the sets for clarity. This is the output:

```
{'New York'}
```

Since the `other_cities` and the `cities` only have New York in common, that's going to be the only element in the `common_cities` set. We can also find the elements that they don't have in common.

Difference

In order to find the elements in one set that are not in the other, we can use the difference method. Here's how to do that.

```
cities = {"New York", "San Francisco", "Los Angeles"}  
other_cities = {"Boston", "Miami", "New York"}  
unique_cities = cities.difference(other_cities)  
print(unique_cities)
```

This will give the values that are present in `cities`, but not present in `other_cities`. And this is what the output will be:

```
{'Los Angeles', 'San Francisco'}
```

As you can see, `cities` has Los Angeles and San Francisco. These values are not in `other_cities`. New York is in both, and that's why it's not in the

output. There's a lot more possible, but these will give you a great foundation to start with sets. The last data structure we'll discuss are tuples.

Tuples

A **tuple** is an ordered collection of items that cannot be changed once created, i.e., an immutable data type. It also allows zero-based indexing and duplicate values to be stored. Moreover, tuples can be used as the keys in dictionaries or elements in sets. For instance, we might use a tuple to store the latitude and longitude of greenFuture's office:

```
office_location = (40.7128, -74.0060) # coordinates for New York  
print(office_location)
```

This will output:

```
(40.7128, -74.0060)
```

We can access the latitude (the first item) just like we would in a list:

```
latitude = office_location[0]  
print(latitude)
```

Output:

```
40.7128
```

Tuples are similar to lists except that they are immutable and use parenthesis, while lists use square brackets. These things are the same:

- Getting the size with the `len()` function
- Using the `count()` method to see how often a value occurs
- Finding the index with the `index()` method
- One cool feature of tuples is that they can be used as keys in dictionaries. Don't be intimidated by the syntax and have a good look at this example. We're storing city waste data in a dictionary with tuple keys:

```
waste_data = {
```

```
(40.7128, -74.0060): {"plastic": 1200, "glass": 800, "aluminium": 1500}, # New York
(34.0522, -118.2437): {"plastic": 900, "glass": 700, "aluminium": 1200} # Los Angeles
}
# Accessing data for New York
new_york_data = waste_data[(40.7128, -74.0060)]
print(new_york_data)
```

In this example, we use tuples to represent the coordinates of different cities, and we use these tuples as keys in a dictionary to store waste data for each city. Here's what it will print:

```
{'plastic': 1200, 'glass': 800, 'aluminium': 1500}
```

And that's it: list, dictionary, set and tuple. These are the basic data types and structures in Python will be the foundation for many of the tasks you'll have to perform. They provide a flexible way to represent real-world data and manipulate it to derive insights. The topics we're going to be discussing next are going to open a new world in terms of tasks you can perform with Python. Using your knowledge of data structures and combining them with your near future knowledge of control flow structures is going to be a great tool for solving your data analysis tasks. So, let's go!

Mastering Control Flow Structures

Control flow structures are fundamental programming constructs that refer to how individual statements, instructions, or function calls are executed within a program. They dictate the order of operations in the program and which code blocks execute when. In Python, there are several types of control flow structures such as the conditional statements, for loops, while loops, and more. You cannot do a lot with Python without those. Hence, it's essential to understand how these work if you want to develop your programs. One key thing to note here is that while other languages use curly braces for scoping, Python uses indentation. The indentations in the examples here are not optional or for layout purposes, changing them breaks the code! Let's start by looking at a control flow structure that is used for decision making.

Conditional Statements in Python

Conditional statements execute based on specified conditions. If a certain condition is true, a specified code block will be executed. If it's false, that code block will not be executed. The if statement is defined by using the word if, followed by the expression that evaluates to a boolean, followed by a colon. Here's a basic example:

```
x = 3
```

```
y = 10
if x > y:
    print("x is greater than y!")
```

This will output nothing. Why? Well the print statement is only executed if $x > y$, and since x is not bigger than y . It is not executed. Let's look at this example:

```
x = 30
y = 10
if x > y:
    print("x is greater than y!")
```

This will output:

```
x is greater than y!
```

Why? Well, because now the x is greater than y . So it does execute the code block (in this case the print line) associated with the if. The code block could consist of multiple lines, they would all have to be indented. Otherwise it's considered outside of the if statement. So far, we've only seen if statements and nothing that needs to happen only if the condition is false. If the condition is false, the program can execute an optional else block. Let's explore this with an example:

```
x = 3
y = 10
if x > y:
    print("x is greater than y!")
else:
    print("x is smaller than or equal to y!")
```

The else doesn't specify a condition. It doesn't need one, it's when the if statement is not true, it will end up in the else. Again, the code in the else block needs to be indented in order to belong to the else block. Here's what

this will output:

```
x is smaller than or equal to y!
```

Since the statement `x > y` evaluates to false, the `else` block is executed. We can even specify multiple if conditions in one if statement. The second one is an “else if” written as `elif`. This second statement will only be evaluated if the first one is false.

```
x = 3
y = 10
z = 5
if x > y:
    print("x is greater than y!")
elif x > z:
    print("x is greater than z!")
else:
    print("x is smaller than or equal to z!")
```

The flow works from top to bottom through the `if-elif-else` statement. The conditions are checked; first, the `if` statement, when found false, the interpreter checks if the `elif` statement is true; and lastly, when it finds the `elif` to be false too, it prints out the statement inside the `else` block. Everything that evaluates to `True` or `False` can be the expression of the `if` statement. Now, let’s continue using our greenFuture example and tackle the problem of finding data on specified waste materials:

```
waste_recycled_kg = {"plastic": 1200, "glass": 800, "aluminium": 1500}
if "plastic" in waste_recycled_kg:
    plastic_kg = waste_recycled_kg["plastic"]
    print(f"Plastic: {plastic_kg} kg")
elif "glass" in waste_recycled_kg:
    glass_kg = waste_recycled_kg["glass"]
    print(f"Glass: {glass_kg} kg")
elif "aluminium" in waste_recycled_kg:
    aluminium_kg = waste_recycled_kg["aluminium"]
    print(f"Aluminium: {aluminium_kg} kg")
else:
    print("No data available for the specified materials.")
```

Remember, the `else` statement is only evaluated if the preceding `if` or `elif` conditions were false. That’s why the output is:

Plastic 1200 kg

Let's make sure you understand what is going on. What do you think this code snippet will print?

```
waste_recycled_kg = {"plastic": 1200, "glass": 800, "aluminium": 1500}
if "plastic" in waste_recycled_kg:
    plastic_kg = waste_recycled_kg["plastic"]
    print(f"Plastic: {plastic_kg} kg")
if "glass" in waste_recycled_kg:
    glass_kg = waste_recycled_kg["glass"]
    print(f"Glass: {glass_kg} kg")
if "aluminium" in waste_recycled_kg:
    aluminium_kg = waste_recycled_kg["aluminium"]
    print(f"Aluminium: {aluminium_kg} kg")
```

Since these are three separate if statements, all three conditions are evaluated. Since all conditions are true, all the code blocks will be executed. This is what it prints:

```
Plastic: 1200 kg
Glass: 800 kg
Aluminium: 1500 kg
```

You'll find yourself using if statements a lot. Another very commonly used control flow structure is the loop. Let's talk about looping next.

Looping in Python

Loops are a way for us to execute a code block repeatedly until the specified condition(s) remain true. Python has two main types of loops: for loops and while loops. We will look at each of them below.

For Loops

A for loop is used for iterating over a sequence (such as a list, tuple, or string) or other iterable objects. It's used in almost every program you will come across. Let's say greenFuture has a list of cities they plan to expand to, and

we want to print out each city's name. Now, there's another, more cumbersome, way to do this apart from the for loop. You could access each element by their indexes in the list and then print them. Here's what that would look like.

```
cities = ["Boston", "Denver", "Los Angeles", "Seattle"]
print(cities[0])
print(cities[1])
print(cities[2])
print(cities[3])
```

You can achieve the same thing with the below code snippet that uses a for-in loop.

```
cities = ["Boston", "Denver", "Los Angeles", "Seattle"]
for city in cities:
    print(city)
```

Do you see how the example with the loop is a lot less code? We need to specify a temporary name for each element in the list. Here we choose `city`. We could have chosen anything, this would have done the same thing:

```
cities = ["Boston", "Denver", "Los Angeles", "Seattle"]
for x in cities:
    print(x)
```

But `city` is more descriptive than `x`, so that's the better pick. Here's what it will output (in both cases):

```
Boston
Denver
Los Angeles
Seattle
```

Without the loops, we would have to add a print statement if we add an element to our cities. This is not needed for a loop, since it loops over all the elements in `cities`. We can use the for loop in a similar way on sets and tuples. But we could also use a for loop with a dictionary and iterate through its elements. Before we show you the next example, you should remember from earlier in this chapter there are various methods for accessing dictionary elements, such as `keys()`, `values()`, and `items()`. The first two we have seen,

and the third one returns all the key-value pairs. Now, for instance, if we want to print out each category of waste and the amount greenFuture has recycled, this is how you would do that:

```
waste_recycled = {"plastic": 1200, "glass": 800, "aluminium": 1500}
for category, amount in waste_recycled.items():
    print(f"greenFuture has recycled {amount} kilograms of {category}.")
```

As you can see, we need to define a name for the key (category) and the value (amount). Here is the output:

```
greenFuture has recycled 1200 kilograms of plastic.
greenFuture has recycled 800 kilograms of glass.
greenFuture has recycled 1500 kilograms of aluminium.
```

That's how we can use for loops. Let's talk about while loops next.

While Loops

While loops allow us to execute a code block as long as the condition(s) remains true. This is useful when you are unsure how often you need to run a loop. For example, let's say greenFuture has a goal to recycle 5000 kilograms of waste, and they're tracking their progress:

```
total_recycled = 0
goal = 5000
while total_recycled < goal:
    # let's say we recycle 1000 kilograms each week
    total_recycled += 1000
    print(f"Total recycled so far: {total_recycled} kilograms")
print("Goal reached!")
```

This is what it outputs:

```
Total recycled so far: 1000 kilograms
Total recycled so far: 2000 kilograms
Total recycled so far: 3000 kilograms
Total recycled so far: 4000 kilograms
Total recycled so far: 5000 kilograms
Goal reached!
```

On each iteration, the while loop checks if the condition statement is true or false; when it returns false, the loop exits to the code outside the while code block. In this case we use 1000 every time, but it could also be possible that the number 1000 would vary every time and would be coming from an external source. In that case, we really wouldn't know how often to execute the loop. All we know is that our goal is 5000 and that we need to continue until we reached our goal. At this point, there's something we need to discuss when talking about loops, and that would be the concept break and continue.

Utilizing Break and Continue

You might encounter situations where you need to exit a loop prematurely or skip an iteration. Python provides two keywords for these scenarios: **break** and **continue**. Let's have a look at each of them with examples from our greenFuture data analysis tasks.

The Break Statement

The **break** statement allows you to exit the loop prematurely when a certain condition is met. This can be particularly useful when you've found what you're looking for and there's no need to continue the loop. If you'd continue to execute the loop that would be a waste of computing power. Suppose greenFuture has a list of `cities` where it operates, and we want to check if a particular city is in the list. Once we find the city, there's no need to continue checking the rest of the list, because we've found our answer.

```
cities = ["New York", "Los Angeles", "Chicago", "Houston", "Phoenix"]
target_city = "Chicago"
for city in cities:
    print(f"Currently looping over {city}.")
    if city == target_city:
        print(f"{target_city} is in the list of cities.")
        break # Exit the loop when the target city is found
```

This will output:

```
Currently looping over New York.
Currently looping over Los Angeles.
```

```
Currently looping over Chicago.  
Chicago is in the list of cities.
```

In this code, as soon as the `target_city` is found, the `break` statement is executed, exiting the loop prematurely. You can see that it doesn't print "Currently looping over Houston" (or Phoenix). That's how to break out of the loop completely. You can also stop the current iteration and move on to the next, this is done with the `continue` statement. So let's continue, with `continue`!

The Continue Statement

The **continue** statement allows you to skip the rest of the current iteration and proceed to the next iteration. This can be useful when you want to skip certain items in a loop. Let's say greenFuture has a list of waste materials, and we want to print out the types of waste, skipping any entry that is glass.

```
waste_types = ["plastic", "glass", "aluminium", "paper", "organic"]  
for waste in waste_types:  
    if waste == "glass":  
        continue # Skip rest of the code in this iteration  
    print(waste)
```

This will output:

```
plastic  
aluminium  
paper  
organic
```

The iteration of glass is skipped. Because in this code, whenever the `waste` is 'glass', the `continue` statement is executed. This means that the `print(waste)` statement in that iteration will be skipped. Instead, the code execution proceeds to the next iteration of the loop. These two control flow tools, `break` and `continue`, provide additional control over how your loops execute. Loops are a great building block for automating repetitive tasks and let our program run until certain conditions are met. Another very useful basic Python building block that will help us to automate tasks and structure our code are functions. Let's explore them next.

Functions in Python

A function is a block of code that can be used to perform a single action. Functions provide modularity for your application and a high degree of code reusability. Whenever you feel the need to copy-paste, chances are you might need a function instead (or a class, but that's out of scope for this book). When you use functions, you follow the D-R-Y principle (Don't Repeat Yourself) in programming. This principle aims to eliminate duplicate code. Reusing code makes the application more flexible, easier to read and easier to maintain.

That must sound pretty good! So let's see how we can define functions to get started with this.

Creating Your Own Functions

In Python, you can define a function using the keyword `def` followed by the function name and a colon. Let's create a simple function that calculates the amount of waste left after recycling for greenFuture:

```
def calculate_waste_left(total_waste, waste_recycled):
    return total_waste - waste_recycled
```

This function takes two input parameters (`total_waste` and `waste_recycled`) and returns the result of subtracting them. We can call the function by stating its name, two parentheses, and the arguments that we need for the function between the parentheses. Here's an example:

```
waste = 5000
recycled = 3200
waste_left = calculate_waste_left(waste, recycled)
print(f"After recycling, there will be {waste_left} kilograms of
waste left.")
```

Output:

```
After recycling, there will be 1800 kilograms of waste left.
```

The cool thing about functions, is that we can call them as often as we like, with different arguments. So we can also say:

```
waste_left1 = calculate_waste_left(300, 250)
waste_left2 = calculate_waste_left(1000, 200)
waste_left3 = calculate_waste_left(14, 7)
```

This will store the values 50, 800 and 7 in `waste_left1`, `waste_left2` and `waste_left3` respectively. Whenever we write a snippet of code that we need to reuse, we can create a function for it. Python has a lot of functions built-in, let's explore those next.

Python Built-In Functions

We have been using some of Python's built-in functions already, for example:

- `len()`
- `print()`
- `input()`
- `type()`

These are highly useful in many programs. For instance, you will almost always find a `print` statement in any program as one of its function is also to help the debugging. Similarly, the `len()` function helps find the number of elements in an iterable object. And in contrast, the `type()` function, as we discussed in type conversion before, contributes to finding out the data type of existing objects. Of course, many more built-in functions exist, and we are only scratching the surface here.

The range function

Let's talk about a very often used built-in function that we haven't seen yet: the `range()` function. The `range()` function is used in Python to generate a sequence of numbers. It's often used in loops to control the number of iterations. The `range()` function takes up to three arguments: start, stop, and step. The start argument is the beginning number of the sequence, the stop argument is 1 past the end of the sequence, and the step argument is the difference between each number in the sequence. Its syntax is as follows:

```
range(start, stop, step)
```

Let's look at a practical example. We could use the `range()` function to iterate through the years and calculate the projected amount of waste recycled.

```
# Assume greenFuture is currently operating in 3 cities
current_cities = 3
# They plan to expand to 10 cities over the next 7 years
target_cities = 10
# Each city recycles 1000 kg of waste per year
recycle_per_city_per_year = 1000
# We'll project the total waste recycled over the next 7 years
for year in range(1, 8):
    # Assume greenFuture expands to 1 new city each year
    current_cities += 1
    total_recycled_this_year = current_cities * recycle_per_city_per_year
    print(f"Year {year}:")
    print(f"  Cities operating: {current_cities}")
    print(f"  Total waste recycled this year: {total_recycled_this_year} kg")
    print("-" * 45)
```

In this example, the `range()` function is used to simulate the passage of time (years) as greenFuture expands its operations to more cities. Each iteration of the loop represents a new year, with an additional city added to the `current_cities` count, and the total waste recycled for that year is calculated and printed. Here's the output:

```
Year 1:
  Cities operating: 4
  Total waste recycled this year: 4000 kg
-----
Year 2:
  Cities operating: 5
  Total waste recycled this year: 5000 kg
-----
Part omitted to keep the output shorter.
-----
Year 6:
  Cities operating: 9
  Total waste recycled this year: 9000 kg
-----
```

```
Year 7:  
    Cities operating: 10  
    Total waste recycled this year: 10000 kg
```

This is one example how to use the range for loops. It's often used. Let's see a few more popular built-in functions next.

More built-in functions

There are a lot of built-in functions. Discussing all of them is out of scope. But let's have a look at an example that combines a lot of them. You must be very familiar with the `print()` function right now, but do you know remember `input()` that we saw earlier in this chapter? Here's an example that calculates the total waste left, in whole numbers, after taking user input for the amount of total waste and waste recycled. We'll explain the different functions used after:

```
def calculate_waste_left(total_waste, waste_recycled):  
    return abs(total_waste - waste_recycled)  
total_waste = int(input("Enter the total amount of waste: "))  
waste_recycled = int(input("Enter the amount of waste recycled: "))  
waste_left = calculate_waste_left(total_waste, waste_recycled)  
# Round waste left to 2 decimal places  
rounded_waste_left = round(waste_left, 2)  
output_message = "After recycling, there will be {} kilograms of  
    waste left.".format(  
        rounded_waste_left  
)  
print(output_message)
```

It pauses twice. And I entered the values 100 and 80 here. Here is what it outputs:

```
Enter the total amount of waste:  
100  
Enter the amount of waste recycled:  
80  
After recycling, there will be 20 kilograms of waste left.
```

Note that this example incorporates a few built-in functions (in program

order): `abs()`, `int()`, `input()`, `round()`, `format()`, and `print()`. Here's a quick summary of all these functions: We have already discussed `print()`, and as evident, `input()` helps with engaging users by taking the input from the console. The `int()` function, as seen previously in type conversion, converts the input string into an integer to perform calculations on. The `abs()` function finds the absolute value of the result of the difference in total waste and waste recycled. Lastly, the `round()` function takes two arguments: the value and the number of decimal places to which you want to round the value. As you can see, functions, whether they are pre-built or defined by you, are very helpful for coding. They let you encapsulate blocks of code that perform specific tasks into one place, and then call that code whenever you want to perform the task. And that's it for the basics of Python! Let's reiterate what we've learnt before we move on to some more complex topics.

Summary

In this chapter, we started Python programming and got a basic understanding of its syntax and many of its components. Right now, you should have a comprehensive understanding of Python's syntax, its data types, and the concept of indexing, control flow structures, data structures, and functions. We began with the basics of Python's syntax: `print` statements and variables which are essential for output and data storage, respectively. We then discussed operators and expressions, fundamental elements for performing operations and constructing expressions in Python. Next, we explored the various data types you can utilize in Python. We looked into the usage and properties of strings, integers, floats and booleans. We have also seen how to manipulate these data types in Python to extract information and efficiently conduct operations. After that, we addressed the topic of indexing in Python, an invaluable technique for accessing elements in a data structure. Using various examples, we depicted the significance of correct indexing to handle data effectively, as it's vital to understand the concept behind it. When it comes to unpacking data structures, we looked at lists, dictionaries, sets, and tuples. For each data structure, we outlined characteristics, demonstrated how to work with them, and discussed their uses in Python programming. Our focus then shifted to control flow structures, emphasizing conditional statements and loops. Lastly, we discussed functions in Python. We started by creating our own functions, highlighting how to encapsulate reusable

pieces of code. We also reviewed some of Python's built-in functions, like print, input, and type, that Python natively provides for various tasks. Throughout this chapter, we utilized our greenFuture example to make each topic more practical to help you learn. This exploration of Python's fundamentals sets the stage for more advanced topics and future projects. In the next chapter, you'll see the basics of some popular libraries for data analysis with Python.

12 Analyzing data with NumPy & Pandas

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



After getting to know the Python basics, it's time to dive into some data analysis with Python. The language primarily preferred for data analysis has been Python. This is due to its rich ecosystem of libraries. Python's built-in data structures, like lists, tuples, etc., do not offer satisfactory performances and functionalities when it comes to numerical analysis of large and complex datasets. This is where two very popular libraries that are widely used in the data community come in: NumPy and Pandas. NumPy is a library that provides a high-performance multidimensional array object for efficient computation of arrays and matrices. Comparatively, lists in Python are not good enough for numerical operations, especially when the data size is large. Pandas provides high-level data structures like Series and DataFrame, built on top of NumPy arrays, providing many tools for data analysis tasks, such as handling missing data, merging/joining, reshaping, and pivoting datasets. The standard built-in Python data structures would require significant custom code for similar tasks. Our foundation of Python is still very fragile, so we will not cover these libraries in great depth. Again, these topics each deserve their own books and we're only doing one chapter to familiarize you with them and get you curious to learning more about them. This is what we'll cover:

- Fundamentals of NumPy

- Basic NumPy operations
- Statistical and mathematical operations
- Multi-dimensional Arrays
- Fundamentals of Pandas
- Series and DataFrame
- Loading data with Pandas
- Data cleaning and preparation
- Data analysis and visualization

You probably can't wait any longer, so let's get started with NumPy right away!

Introduction to NumPy

NumPy is a Python library that is short for Numerical Python. This library is the cornerstone for numerical computing in Python. It provides support for arrays, matrices and many mathematical functions to operate on these data structures. It gives us highly optimized arrays and mathematical functions. Unlike Python lists, NumPy arrays are homogenous and allow for vectorized operations, making computations faster. You would not need this for small amounts of data and simple operations, but it is necessary when dealing with large datasets or performing complex mathematical operations. And since this is a common scenario in data analysis, let's see how we can start using it.

Installing and Importing NumPy

Before we are able to make use of the features of NumPy, we need to install it on our computer. Luckily, installing NumPy is a breeze with pip:

```
pip install numpy
```

You should type this on the command line, press enter and ta-da. Now we have NumPy installed. We can go ahead and use it in our Python programs by importing it. It's a common practice to import NumPy as np for ease of use:

```
import numpy as np
```

The `as np` gives an alias for our numpy module. Whenever you need to use the module, you'll only have to type `np` and not the full word `numpy`. With NumPy now at our fingertips, we're all set to delve into some basic operations.

Basic NumPy Operations

We can do very advanced things with NumPy, but we will not go there just yet. Let's kick it off with some basic operations to get some insights in the recycle scores of greenFuture. We'll start with one of the core tasks we can do with NumPy: create an array.

Creating Arrays

Creating arrays in NumPy is straightforward, but a little different than without NumPy. An **array** is pretty much a container for more than one item, similar to lists. Let's create a simple array representing the amount of plastic waste greenFuture recycled over a week:

```
plastic_recycled = np.array([200, 300, 400, 250, 600, 350, 275])  
print(plastic_recycled)
```

We first create it and after that we print it. This is the output:

```
[200 300 400 250 600 350 275]
```

Here, we've just created a one-dimensional array. The `np.array()` function creates a NumPy array for with the values of the list that we gave as input.

Array Indexing and Slicing

We have seen that we can use indexing and slicing on Python lists. Of course we can do this on the NumPy arrays as well. Accessing elements in a NumPy array is similar to indexing a list in Python. Let's retrieve the amount of plastic recycled on the first and last day of the week:

```
first_day = plastic_recycled[0]  
last_day = plastic_recycled[-1]
```

```
print(f"First day: {first_day}, Last day: {last_day}")
```

Output:

```
First day: 200, Last day: 275
```

With NumPy, we can also slice and dice the data to get specific portions of the array. Here's how to slice the array to get the data for the weekdays only:

```
weekdays_recycled = plastic_recycled[:5]
print(weekdays_recycled)
```

Just like Python lists, when we don't specify the start value, it starts at the beginning. The end index is 5, meaning the last index it will print is index 4:

```
[200 300 400 250 600]
```

There are also operations that were not available in Python that we can use easily with NumPy arrays. Let's explore those next!

Statistical and Mathematical Operations

NumPy's true power shows when it comes to performing mathematical and statistical operations. Let's calculate the total and average amount (mean) of plastic recycled over the week:

```
total_recycled = np.sum(plastic_recycled)
average_recycled = np.mean(plastic_recycled)
print(f"Total recycled: {total_recycled}, Average recycled: {average_recycled}")
```

And this is what it will output:

```
Total recycled: 2375, Average recycled: 339.2857142857143
```

With just a two simple lines of code, we've computed the total and average recycling for the week. Let's see another example with some more statistical operations. We'll calculate some basic statistics for the amount of glass recycled over a week:

```
glass_recycled = np.array([150, 200, 250, 300, 200, 100, 50])
```

```
# Calculating mean, median and variance
mean_glass = np.mean(glass_recycled)
median_glass = np.median(glass_recycled)
variance_glass = np.var(glass_recycled)
print(f"Mean: {mean_glass}, Median: {median_glass}, Variance: {variance_glass}")
```

We are calculating the mean, median (middle value) and the variance (degree of spread). Here is the output:

```
Mean: 178.57142857142858, Median: 200.0, Variance: 6326.53061224  
4898
```

With just a few lines of code, we've managed to compute some fundamental statistics. We would have to check with NumPy, but it's probably statistically significant how easy that was! There are other cool features of NumPy that are worth mentioning. The basic mathematical operations can also be applied to complete arrays. Let's see how to do that next.

Mathematical Operations with NumPy Arrays

NumPy makes mathematical operations on arrays very intuitive and easy to do. Let's say we want to compare the amount of plastic and glass recycled over a week. We can easily perform element-wise addition, subtraction, and multiplication:

```
plastic_recycled = np.array([200, 300, 400, 250, 600, 350, 275])
glass_recycled = np.array([150, 200, 250, 300, 200, 100, 50])
# Element-wise addition
total_recycled_daily = plastic_recycled + glass_recycled
# Element-wise subtraction
difference_recycled_daily = plastic_recycled - glass_recycled
# Element-wise multiplication
multiplied_recycled_daily = plastic_recycled * glass_recycled
print(f"Total recycled daily: {total_recycled_daily}")
print(f"Difference recycled daily: {difference_recycled_daily}")
print(f"Multiplied recycled daily: {multiplied_recycled_daily}")
```

We've added the NumPy arrays on top of this code snippet again to make it a bit easier to process what is happening. As you can see, it adds, subtracts and multiplies the values on the same indices of each array. To be fair, we're not

sure how we would need the multiplication in this concrete example. It's just there to show you what it does and how easy it is to use. This is the output:

```
Total recycled daily: [350 500 650 550 800 450 325]  
Difference recycled daily: [ 50 100 150 -50 400 250 225]  
Multiplied recycled daily: [ 30000 60000 100000 75000 120000  
35000 13750]
```

To sum this up, we dare to say that math with NumPy is as easy as `np.array([1, 2, 3])`. Let's look at something that's typically more complex to work with (mainly because your brains need to keep track of what is happening): multidimensional arrays.

Multi-dimensional Arrays

Let's spice it up a bit and see what we can do with multidimensional arrays. These are simply said: an array of arrays. For a 2D array, this means that the outer array contains arrays and the inner arrays contain non-array elements. They allow us to represent different kinds of data in a structured form. Let's see how we can create one.

Creating Multi-dimensional Arrays

Creating multi-dimensional arrays is similar to creating a 1D array. Let's create a 2D array representing the amount of waste recycled in two cities over a week:

```
recycled_2d = np.array([[200, 300, 400, 250, 600, 350, 275],  
# City 1  
#           [150, 200, 250, 300, 200, 100, 50]]) #  
city 2  
print(recycled_2d)
```

This is what the output will be:

```
[[200 300 400 250 600 350 275]  
[150 200 250 300 200 100 50]]
```

As you can see, it's simply two arrays inside one array. The first inner array represents the first city and the second inner array represents the second city.

Let's see how to access the elements in this 2D array.

Accessing elements in Multi-dimensional Arrays

Accessing elements or slices of multi-dimensional arrays is a walk in the park. Let's fetch the data for the first city:

```
city_1_recycled = recycled_2d[0]  
print(city_1_recycled)
```

This is the output:

```
[200 300 400 250 600 350 275]
```

And here is the value of recycled material on the third day of the second city:

```
city_2_day_3_recycled = recycled_2d[1][2]  
print(city_2_day_3_recycled)
```

This will output the value:

```
250
```

As you can see, with the first [1] we access the elements of the outer array, which is an array in itself. With the second [2] we access the elements of this inner array. Let's see how we can read the data from a CSV file and analyze this with the use of NumPy.

Reading Data from a CSV File

Often, you'll have to read certain files and analyze the data inside it. We're going to Now, let's put our newfound knowledge to the test by analyzing greenFuture's monthly recycling data.

Loading and Analyzing Data

Suppose we have the weekly recycle data stored in a CSV file. Here's what

the file `weekly_recycling_data.csv` could look like:

```
Week,Plastic,Glass,Aluminium
1,200,150,300
2,250,180,350
3,220,160,310
4,240,170,320
5,230,155,305
6,210,145,290
7,205,140,285
8,235,165,315
9,245,175,325
10,225,160,310
```

We can easily load this data into a NumPy array using the `np.loadtxt()` function. It could be a lot bigger and it would work just the same.

```
weekly_data = np.loadtxt('weekly_recycling_data.csv', delimiter=',', skiprows=1, usecols=(1, 2, 3))
print(weekly_data)
```

This will output the array loaded as a NumPy array:

```
[[200. 150. 300.]
 [250. 180. 350.]
 [220. 160. 310.]
 [240. 170. 320.]
 [230. 155. 305.]
 [210. 145. 290.]
 [205. 140. 285.]
 [235. 165. 315.]
 [245. 175. 325.]
 [225. 160. 310.]]
```

With the data loaded, let's calculate the total and average for each type (plastic, glass and aluminium) per week:

```
total_recycling_weekly = np.sum(weekly_data, axis=0)
average_recycling_weekly = np.mean(weekly_data, axis=0)
print(f"Total recycling weekly: {total_recycling_weekly}")
print(f"Average recycling weekly: {average_recycling_weekly}")
```

Output:

```
Total recycling weekly: [2260. 1600. 3110.]  
Average recycling weekly: [226. 160. 311.]
```

And that was just a few seconds work! At this point, we've only just scratched the surface of what NumPy can do. With a bit of practice, you'll find it to be an ally in your data analysis work. For now this is enough. Let's move from the structured world of NumPy to the tabular world of Pandas! These libraries are very commonly combined and you'll see how they complement each other in doing your data analysis tasks. So let's go!

FileNotFoundException

We use a relative path in our code example, namely 'weekly_recycling_data.csv'. A relative path means that it's not the full path starting from the root of the machine (for example: C:\Users\maaike\Documents\Books\Becoming-a-Data-Analyst-First) but only how to get there from the current location.

If the current location is different, the relative path won't work. If you try to run this script from a different folder you can get issues. Therefore, make sure to be in the folder that the script is in when you run it, and that in this folder the weekly_recycling_data.csv is located as well. If you can't figure out how to do that, use the full path as a workaround.

Introduction to Pandas

Now that we're familiar with the basics of NumPy, let's take the next step. Let's learn about Pandas. Pandas is great for data manipulation and analysis. With Pandas we have access to many functions and structures to support our data analysis tasks. It is a high-level data manipulation tool, built on the Numpy package. Its key data structure is called **DataFrame**, which you can think of as an in-memory 2D table (like a spreadsheet), with labeled axes (rows and columns). This not only allows for the storage of data but also the manipulation and analysis of it. We're not really exaggerating when we state that it's a data analyst's best friend. Let's see how to get our system ready for using Pandas.

Installing and Importing Pandas

Installing Pandas is similar to installing Numpy. Just a simple command and you're good to go:

```
pip install pandas
```

Once installed, importing Pandas into your script is equally straightforward:

```
import pandas as pd
```

The alias that is the convention to use for Pandas is `pd`. Now that we are all set, let's create some data structures!

Series and DataFrame

The two main data structures in Pandas are Series and DataFrames. A **Series** is like a column in a table, a one-dimensional array, if you will. A **DataFrame** is a multi-dimensional table made up of a collection of Series. These structures come with many methods to manipulate and analyze data. It can be intimidating to start, but it's really not too complicated. Let's get it over with and create them.

Creating Series and DataFrame from Scratch

Creating a Series or DataFrame from scratch is simple. We'll create a Series to represent the amount of plastic recycled by greenFuture over a week:

```
plastic_recycled_series = pd.Series([200, 300, 400, 250, 600, 350, 275])
print(plastic_recycled_series)
```

Here is the output:

```
0    200
1    300
2    400
3    250
4    600
```

```
5    350
6    275
dtype: int64
```

As you can see we have all the values in our list, and they're all given a row number. Now, let's create a DataFrame (which is pretty much a collection of Series) to represent the amount of different materials recycled over a week:

```
recycled_data = {
    'Plastic': [200, 300, 400, 250, 600, 350, 275],
    'Glass': [150, 200, 250, 300, 200, 100, 50],
    'Aluminium': [100, 150, 200, 150, 300, 200, 175]
}
recycled_df = pd.DataFrame(recycled_data)
print(recycled_df)
```

We have to create an object that has a key (column name) and an array (values for each row in the column). We could have done that directly in the line, as we did with `pd.Series()`. Instead, we split it up and create the object separately for readability. Here is what the above code snippet will output:

	Plastic	Glass	Aluminium
0	200	150	100
1	300	200	150
2	400	250	200
3	250	300	150
4	600	200	300
5	350	100	200
6	275	50	175

And that's it already. We have successfully created a Series and DataFrame from scratch. The DataFrame gives us a neat tabular representation of our data, making it easier to analyze and manipulate. It's not very common to create the Series and DataFrames from Scratch. More often, we read a file with data and load it as a DataFrame. Let's see how to do that next before we start analyzing and manipulating our data.

Loading Data with Pandas

Now that we have acquainted ourselves with the basic data structures in Pandas, it's time to learn how to load external data into our environment.

Often, the data you need will not be created manually or come from within your program, but will be external sources like a CSV file or an Excel spreadsheet. Pandas has a plethora of functions to read data from such files, making it a breeze to load data for analysis. Let's "panda" to our curiosity and explore how we can load data with Pandas!

Reading CSV, Excel files

Reading data from files is straightforward with Pandas. Let's assume we have a CSV file named `recycling_data.csv`. Here's the content of the CSV file:

```
Date,Plastic,Glass,Aluminium,Paper  
2023-01-01,200,150,,300  
2023-01-08,250,,350,400  
2023-01-15,220,160,310,  
2023-01-22,240,170,320,450  
2023-01-29,230,155,305,420  
2023-02-05,,145,290,410  
2023-02-12,205,140,285,400  
2023-02-19,235,165,315,430  
2023-02-26,245,175,325,440  
2023-03-05,225,160,310,425
```

And here's how to load it with Pandas:

```
recycling_data_csv = pd.read_csv('recycling_data.csv')
```

If it was an Excel file named `recycling_data.xlsx` we could have loaded it like this:

```
recycling_data_excel = pd.read_excel('recycling_data.xlsx')
```

These commands will load the data from the specified files into DataFrame objects. Let's see how we can view and inspect the content.

Viewing and Inspecting Data

Once you have loaded your data, it's often a good practice to take a peek at it to understand its structure, contents, and how the data is organized. Pandas provides several methods to do just that. Here's how you can see the first 5

rows:

```
print(recycling_data_csv.head())
```

And this will output:

	Date	Plastic	Glass	Aluminium	Paper
0	2023-01-01	200.0	150.0	NaN	300.0
1	2023-01-08	250.0	NaN	350.0	400.0
2	2023-01-15	220.0	160.0	310.0	NaN
3	2023-01-22	240.0	170.0	320.0	450.0
4	2023-01-29	230.0	155.0	305.0	420.0

As you can see, we have rows and columns. There are also a few NaN (not a number). This is missing data. We can also view the last 5 rows. Here's how to do that:

```
print(recycling_data_csv.tail())
```

This will output:

	Date	Plastic	Glass	Aluminium	Paper
5	2023-02-05	NaN	145.0	290.0	410.0
6	2023-02-12	205.0	140.0	285.0	400.0
7	2023-02-19	235.0	165.0	315.0	430.0
8	2023-02-26	245.0	175.0	325.0	440.0
9	2023-03-05	225.0	160.0	310.0	425.0

You can tell by the row numbers that these are the last rows. If we want to get a quick summary, we can use the info() method:

```
print(recycling_data_csv.info())
```

The info() method provides a summary of the DataFrame including the number of non-null entries and data types for each column. Here's the output:

```
-----  
0  Date        10 non-null      object  
1  Plastic     9 non-null      float64  
2  Glass        9 non-null      float64  
3  Aluminium   9 non-null      float64  
4  Paper        9 non-null      float64  
dtypes: float64(4), object(1)  
memory usage: 532.0+ bytes
```

We can also get the statistical summary:

```
print(recycling_data_csv.describe())
```

The `describe()` method gives a statistical summary of the DataFrame, which is particularly useful for numerical columns as it provides count, mean, standard deviation, and other statistical measures. Here's the output:

	Plastic	Glass	Aluminium	Paper
count	9.000000	9.000000	9.000000	9.000000
mean	227.777778	157.777778	312.222222	408.333333
std	17.159384	11.486707	19.220938	44.017042
min	200.000000	140.000000	285.000000	300.000000
25%	220.000000	150.000000	305.000000	400.000000
50%	230.000000	160.000000	310.000000	420.000000
75%	240.000000	165.000000	320.000000	430.000000
max	250.000000	175.000000	350.000000	450.000000

Amazing what you can do with Pandas and a few basic methods. The ease with which we loaded and inspected our data is just a teaser of the power of Pandas. Let's see a few more neat things we can do with Pandas to up our data manipulation and analysis game.

Data Analysis with Pandas

Real-world data is often messy and incomplete, we'll encounter the need to clean and prepare our data before deriving insights from it. The good news is that Pandas provides a robust set of tools to handle such data inconsistencies. Let's "bear" down and see how to do that!

Data Cleaning and Preparation

Before being able to use data, the data needs to be cleaned up. Let's deal with a very common step of cleaning up data: handling the missing data.

Handling Missing Data

Missing data is present in most datasets you'll ever get to work with. Pandas

makes it easy to identify and handle such missing data. We have some missing values in our recycling data. Let's first find the missing data:

```
missing_data = recycling_data_csv.isnull()  
print(missing_data)
```

In the snippet above , `isnull()` identifies missing data. When we print it, you can see it has the value `True` for the data that is missing:

```
   Date  Plastic  Glass  Aluminium  Paper  
0  False     False  False      True  False  
1  False     False   True     False  False  
2  False     False  False     False  True  
3  False     False  False     False  False  
4  False     False  False     False  False  
5  False      True  False     False  False  
6  False     False  False     False  False  
7  False     False  False     False  False  
8  False     False  False     False  False  
9  False     False  False     False  False
```

We can fill the missing data with the preceding values. Here how to do that:

```
filled_data = recycling_data_csv.ffill()  
print(filled_data)
```

This is what it prints:

```
   Date  Plastic  Glass  Aluminium  Paper  
0  2023-01-01    200.0  150.0        NaN  300.0  
1  2023-01-08    250.0  150.0      350.0  400.0  
2  2023-01-15    220.0  160.0      310.0  400.0  
3  2023-01-22    240.0  170.0      320.0  450.0  
4  2023-01-29    230.0  155.0      305.0  420.0  
5  2023-02-05    230.0  145.0      290.0  410.0  
6  2023-02-12    205.0  140.0      285.0  400.0  
7  2023-02-19    235.0  165.0      315.0  430.0  
8  2023-02-26    245.0  175.0      325.0  440.0  
9  2023-03-05    225.0  160.0      310.0  425.0
```

This solves the problem for the second row in Glass column and for the third value in the Paper column, but not for the Aluminium column. This is because the first entry is missing in the Aluminium column and there is no preceding value. We could also choose to fill the empty data with the value

that follows. Here's how to do that:

```
filled_data_b = recycling_data_csv.bfill()  
print(filled_data_b)
```

And here's what that prints:

	Date	Plastic	Glass	Aluminium	Paper
0	2023-01-01	200.0	150.0	350.0	300.0
1	2023-01-08	250.0	160.0	350.0	400.0
2	2023-01-15	220.0	160.0	310.0	450.0
3	2023-01-22	240.0	170.0	320.0	450.0
4	2023-01-29	230.0	155.0	305.0	420.0
5	2023-02-05	205.0	145.0	290.0	410.0
6	2023-02-12	205.0	140.0	285.0	400.0
7	2023-02-19	235.0	165.0	315.0	430.0
8	2023-02-26	245.0	175.0	325.0	440.0
9	2023-03-05	225.0	160.0	310.0	425.0

We now set the missing values to the value that follows. That can be one approach to deal with missing data. You could also opt to drop the rows with missing that entirely, here's how:

```
clean_data = recycling_data_csv.dropna()
```

Row 0, 1 and 2 are gone, because they all had missing data:

	Date	Plastic	Glass	Aluminium	Paper
3	2023-01-22	240.0	170.0	320.0	450.0
4	2023-01-29	230.0	155.0	305.0	420.0
6	2023-02-12	205.0	140.0	285.0	400.0
7	2023-02-19	235.0	165.0	315.0	430.0
8	2023-02-26	245.0	175.0	325.0	440.0
9	2023-03-05	225.0	160.0	310.0	425.0

It is worth noting that these methods don't alter the original object, and that's why we assign it to new variables (`filled_data`, `filled_data_b` and `clean_data`). Which one you will choose, depends on what you need. We will continue with the `bfill()` data. We can also add columns and transform the data with Pandas. Let's go there next.

Data Transformation

Transforming data is a bit like sculpting; we start with a rough raw block and chisel away and reveal the form within. Here's how you can transform data with Pandas. We'll start by adding a new column. This new column should represent the total recycling of the materials for that week. So we want to add the values. In order to add the values, we need to exclude the Date column:

```
numeric_data = recycling_data_csv.drop(columns='Date')
print(numeric_data)
```

Here is what it looks like:

	Plastic	Glass	Aluminium	Paper
0	200.0	150.0	350.0	300.0
1	250.0	160.0	350.0	400.0
2	220.0	160.0	310.0	450.0
3	240.0	170.0	320.0	450.0
4	230.0	155.0	305.0	420.0
5	205.0	145.0	290.0	410.0
6	205.0	140.0	285.0	400.0
7	235.0	165.0	315.0	430.0
8	245.0	175.0	325.0	440.0
9	225.0	160.0	310.0	425.0

We then use this `numeric_data` to calculate the sum for each week and add it as a column total to our `recycling_data_csv` object.

```
recycling_data_csv['Total'] = numeric_data.sum(axis=1)
print(recycling_data_csv)
```

Please mind that this doesn't change the csv file, only the object in memory that represented the CSV. This is the renewed table:

	Date	Plastic	Glass	Aluminium	Paper	Total
0	2023-01-01	200.0	150.0	350.0	300.0	1000.0
1	2023-01-08	250.0	160.0	350.0	400.0	1160.0
2	2023-01-15	220.0	160.0	310.0	450.0	1140.0
3	2023-01-22	240.0	170.0	320.0	450.0	1180.0
4	2023-01-29	230.0	155.0	305.0	420.0	1110.0
5	2023-02-05	205.0	145.0	290.0	410.0	1050.0
6	2023-02-12	205.0	140.0	285.0	400.0	1030.0
7	2023-02-19	235.0	165.0	315.0	430.0	1145.0
8	2023-02-26	245.0	175.0	325.0	440.0	1185.0
9	2023-03-05	225.0	160.0	310.0	425.0	1120.0

If we decide that we want to rename the column Total, we can. Here's how to do it:

```
recycling_data_csv.rename(columns={'Total': 'Total Recycled'}, inplace=True)
```

The last column will have the name “Total Recycled” now. The `inplace=True` makes the operation happen directly on the `recycling_data_csv` and that's why we don't need to assign it to another (or the same) variable in order for the modification to have effect. This is more efficient on the memory if you only need the modified version of the DataFrame, because the memory then only needs to keep one. At this point, the data happens to be sorted by date and given the row numbers 0 to 9 accordingly. We can also choose to sort it differently. Let's say we want to sort it based on the Total Recycled value:

```
sorted_data = recycling_data_csv.sort_values(by='Total Recycled')
print(sorted_data)
```

This creates a new DataFrame with an altered order of the rows in the DataFrame, this is what it looks like:

	Date	Plastic	Glass	Aluminium	Paper	Total	Recycled
0	2023-01-01	200.0	150.0	350.0	300.0		1000.0
6	2023-02-12	205.0	140.0	285.0	400.0		1030.0
5	2023-02-05	205.0	145.0	290.0	410.0		1050.0
4	2023-01-29	230.0	155.0	305.0	420.0		1110.0
9	2023-03-05	225.0	160.0	310.0	425.0		1120.0
2	2023-01-15	220.0	160.0	310.0	450.0		1140.0
7	2023-02-19	235.0	165.0	315.0	430.0		1145.0
1	2023-01-08	250.0	160.0	350.0	400.0		1160.0
3	2023-01-22	240.0	170.0	320.0	450.0		1180.0
8	2023-02-26	245.0	175.0	325.0	440.0		1185.0

The rows are now sorted in the new way. It is important to note that this doesn't rename the number of the row. At this point we have cleaned and transformed our data. This cleaning and transforming will help in better understanding and analyzing the data. So far, our cleaned up data only exists in the memory of the application. Let's see how we can save these results.

Saving Your Results

In order to continue to work on the data in a later stage, we need to save our results into accessible formats. Here's how you can save a DataFrame to a new CSV file.

```
sorted_data.to_csv('cleaned_recycling_data.csv', index=False)
```

With the `to_csv()` method, we inscribe our cleaned data onto a new CSV file, ready to be shared with the greenFuture council. The `index=False` parameter ensures that the index column doesn't tag along uninvited. Here's the content of the newly created CSV file:

```
Date,Plastic,Glass,Aluminium,Paper,Total Recycled  
2023-01-01,200.0,150.0,350.0,300.0,1000.0  
2023-02-12,205.0,140.0,285.0,400.0,1030.0  
2023-02-05,205.0,145.0,290.0,410.0,1050.0  
2023-01-29,230.0,155.0,305.0,420.0,1110.0  
2023-03-05,225.0,160.0,310.0,425.0,1120.0  
2023-01-15,220.0,160.0,310.0,450.0,1140.0  
2023-02-19,235.0,165.0,315.0,430.0,1145.0  
2023-01-08,250.0,160.0,350.0,400.0,1160.0  
2023-01-22,240.0,170.0,320.0,450.0,1180.0  
2023-02-26,245.0,175.0,325.0,440.0,1185.0
```

Now we know how to store our DataFrames, let's proceed to some basic data analysis with Pandas.

Data Analysis

It's time to do some basic data analysis with Python. Of course, this is the very moment we've been building towards. Let's start with a more elaborate CSV file `recycle_data_city.csv` for this example:

```
Date,Plastic,Glass,Aluminium,Paper,City  
2023-01-01,120.0,200.0,,300.0,New York  
2023-01-08,150.0,250.0,100.0,350.0,New York  
2023-01-15,130.0,,90.0,310.0,Los Angeles  
2023-01-22,140.0,240.0,80.0,320.0,Los Angeles  
2023-01-29,160.0,260.0,110.0,360.0,Chicago
```

We will use this CSV to work with. Let's start by adding the Total column again:

```
recycling_data_csv = pd.read_csv('recycling_data.csv')
recycling_data_csv['Total'] = recycling_data_csv[['Plastic', 'Glass', 'Aluminium', 'Paper']].sum(axis=1)
```

Here's what the DataFrame looks like:

```
Date    Plastic   Glass   Aluminium   Paper      City  Total
0  2023-01-01     120.0   200.0        NaN   300.0  New York   62
0.0
1  2023-01-08     150.0   250.0     100.0   350.0  New York   85
0.0
2  2023-01-15     130.0     NaN     90.0   310.0 Los Angeles   53
0.0
3  2023-01-22     140.0   240.0     80.0   320.0 Los Angeles   78
0.0
4  2023-01-29     160.0   260.0     110.0   360.0 Chicago   89
0.0
```

As you can see it has some missing values. We can fill them or drop the rows. In this case, we choose to leave it as is, which is something to keep in mind when looking at the results in the next few steps if you were to interpret them. The next step, is that we're going to group the data.

Grouping and Aggregation

We can group the data by columns. In our example, it would make sense to group the data by city. The code for this is quite intuitive:

```
grouped_data = recycling_data_csv.groupby('City')
```

This has grouped the data by city. In order to make sense of it, we need to aggregate the data. Here's how to do that:

```
aggregated_data = grouped_data.sum()
print(aggregated_data)
```

And here's the output:

per	Total	Date	Plastic	Glass	Aluminium	Pa
City						
Chicago		2023-01-29	160.0	260.0	110.0	36
0.0	890.0					
Los Angeles	2023-01-15	2023-01-22	270.0	240.0	170.0	63
0.0	1310.0					
New York	2023-01-01	2023-01-08	270.0	450.0	100.0	65
0.0	1470.0					

You can see that it grouped the data by city. It added the numbers to display the totals. For the dates it did something that's less intuitive, it concatenated them. It makes sense to drop the date column altogether. Grouping and aggregation allow us to slice and dice the data to uncover patterns and insights. We can now easily find the city with the most recycling (but please keep in mind there were missing values that we left as is):

```
top_city = aggregated_data['Total'].idxmax()
print(f'The city with the highest recycling is {top_city}.')
```

This is the output:

The city with the highest recycling is New York.

In the snippets in this section, we grouped our data by city and then aggregated it to find the total recycling per city. This way, we can easily compare the recycling rates of different cities. Grouping and aggregation really help us make sense of the data! There's one more topic to discuss at this point. In order to represent your data, data visualization is the way to go. Of course, this can be done with Python too. Let's see how we can get started with data visualization with Pandas.

Data Visualization

A picture is worth a 1000 words, and a well-crafted plot is worth a thousand data points. Pandas can do a lot, and it can be used for plotting our data too. Here's an example:

```
import pandas as pd
recycling_data_csv = pd.read_csv('recycle_data_city.csv')
recycling_data_csv['Total'] = recycling_data_csv[['Plastic', 'Gl
```

```

ass', 'Aluminium', 'Paper']] .sum(axis=1)
grouped_data = recycling_data_csv.groupby('City')
aggregated_data = grouped_data.sum()
ax = aggregated_data['Total'].plot(kind='bar', color='skyblue',
figsize=(10, 6))
ax.set_title('Total Recycled per City')
ax.set_xlabel('City')
ax.set_ylabel('Total Recycled (kg)')

```

In order to visualize our data Pandas uses Matplotlib beneath the surface. In order to run the above snippet, we need to make sure that this is installed. Here's how to do that:

```
pip install matplotlib
```

This installs matplotlib, and now the above code snippet can run. There won't be any output if you run this code in VS Code. If you run it in jupyter notebook, it will automatically show it to you. In order to get some visuals, we'll need to be using Matplotlib in our code. Matplotlib is a Python library specialized at plotting data. Here's the updated version that results in a visual:

```

import pandas as pd
import matplotlib.pyplot as plt # Importing Matplotlib
recycling_data_csv = pd.read_csv('recycle_data_city.csv')
recycling_data_csv['Total'] = recycling_data_csv[['Plastic', 'Glass',
ass', 'Aluminium', 'Paper']].sum(axis=1)
grouped_data = recycling_data_csv.groupby('City')
aggregated_data = grouped_data.sum()
ax = aggregated_data['Total'].plot(kind='bar', color='skyblue',
figsize=(10, 6))
ax.set_title('Total Recycled per City')
ax.set_xlabel('City')
ax.set_ylabel('Total Recycled (kg)')
plt.show() # Displaying the plot

```

In the snippet above, we used Matplotlib, a plotting library, to create a bar plot of the total recycling per city. Visualizing data not only makes analysis helps in communicating insights effectively. The following figure will pop up in a separate window:

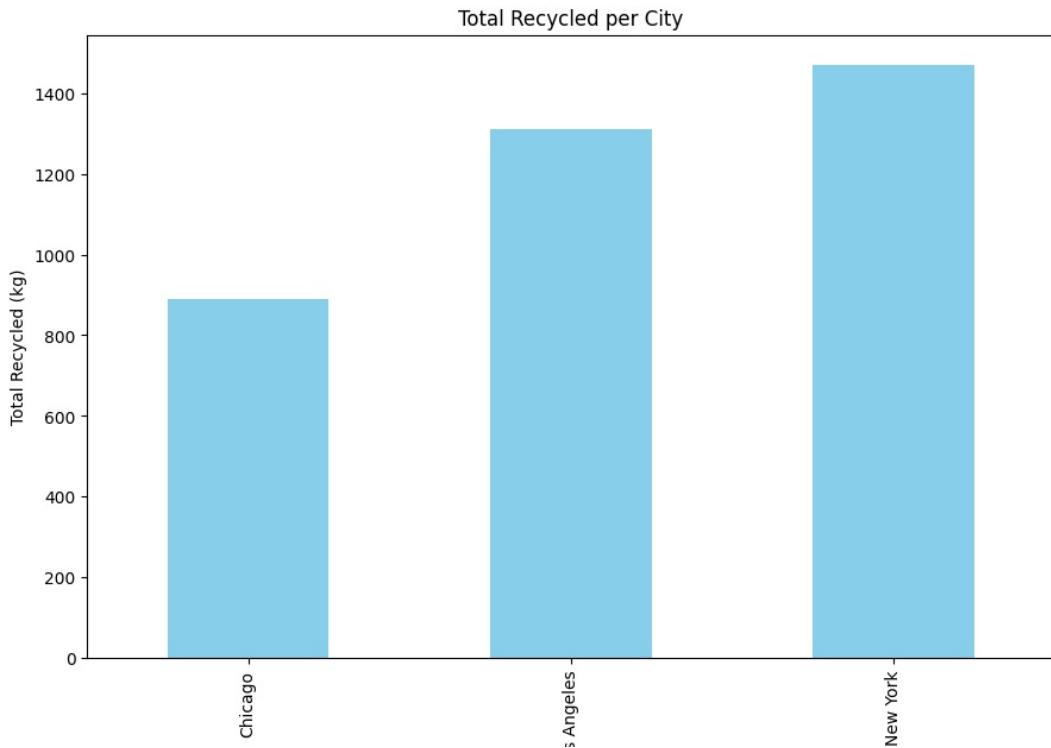


Fig 12.1 – Displaying the total recycled per city

Again, this is just a very quick introduction showing only a very small part of what you can do with Pandas, NumPy and Matplotlib. And only that very small part already gives us an amazing tool of creating descriptive statistics, grouping, aggregating, visualizing the data. You're really at the next level right now. Let's end this chapter with a summary.

Summary

In this chapter, we learnt about data analysis with NumPy and Pandas, two of Python's most powerful libraries. We kicked things off with NumPy, diving into its fundamentals and exploring the ease of creating arrays and performing basic operations. We've also seen how to use NumPy for statistical and mathematical operations. After that, we touched upon multidimensional arrays. At that point, we were ready for Pandas. As we transitioned to Pandas, started to work with Series and DataFrames. These are the core data structures that make Pandas a joy to work with. Loading data

with Pandas is easy, and it got us ready for the process of data cleaning and preparation. We've seen how to use Pandas to handle missing data and transform our data. We ended with a very basic example of how to use Pandas combined with Matplotlib for visualization. This chapter was not trying to teach you all the ins and outs of these libraries, but mainly to tell you enough to understand how excited you should be about them. You can probably imagine that using these libraries is an invaluable skill for data analysts.

13 Introduction to Exploratory Data Analysis

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



Welcome to the exploration of Exploratory Data Analysis (EDA), a crucial skill in the arsenal of data analytics. We frequently lose ourselves in the world of algorithms and intricate statistical models, but a well-done EDA is the foundation of any robust analysis. Using various graphical and statistical techniques, EDA enables us to identify patterns, identify anomalies, test hypotheses, and validate assumptions. In essence, it aids in our familiarization with the dataset by exposing its underlying structure and providing critical information that frequently guides all subsequent analytical steps. In this chapter, we'll delve deep into the mechanics of EDA, starting with its significance and the overall EDA process and moving on to the most popular tools and techniques. The various approaches to data analysis—univariate, bivariate, and multivariate—and their visualization and interpretation, will also be covered. The goal of this chapter is to provide you with the fundamental knowledge and abilities needed to conduct thorough EDA on a variety of datasets. In this chapter, we're going to cover the following main topics:

- Examine continuous variables' distributions by analyzing them with statistical methods.
- Choosing the right methods and tools for conducting EDA.
- Implementing multivariate analysis to understand more complex

relationships in your data.

The Importance of EDA

Without a solid foundation in exploratory data analysis (EDA), beginning a data analytics project is comparable to setting sail on choppy waters without a compass; you're likely to drift aimlessly and may eventually veer off course. Your analytical compass, EDA will lead you through the complexity of your data and direct you toward more complex analyses. This section aims to give readers a thorough understanding of the crucial role that EDA plays in data analysis, the methodologies used to carry it out, and the tools and techniques that are useful when doing so. First, we'll talk about the EDA Process, outlining the steps you would usually take when conducting an EDA exercise. This is a framework that you can modify according to the data at hand and the questions you're attempting to answer rather than a rigid manual. EDA's iterative process ensures you're constantly improving your assumptions and understanding of the dataset. Finally, the section on Tools and Techniques for EDA will acquaint you with the various programs, libraries, and statistical techniques that are frequently employed in this industry. Understanding these tools is essential because a data analyst's proficiency with EDA tools and techniques directly correlates to their effectiveness, just as a craftsman is only as good as his tools. By the end of this section, you will thoroughly understand why EDA is frequently regarded as the foundation of data analytics and how to implement it successfully using various tools and techniques.

The EDA Process

The Exploratory Data Analysis (EDA) process is a flexible framework that will help you understand your dataset's structure, peculiarities, and patterns rather than being a rigid set of guidelines. Typically, the process starts with formulating your interest-driven questions and determining the data required to provide answers. The real journey begins once you have the data in your possession.

1. **Data cleaning** typically comes first in the process. You will fill in any missing values, eliminate duplicates, and fix any errors here. If you skip

this step, your subsequent analyses may contain noise and error. It's important to remember that data cleaning can be an iterative process revisited as your investigation progresses.

2. **Data summarization**, the next step, involves using descriptive statistics like mean, median, variance, and standard deviation. This is a great place to start looking for the first patterns or anomalies that require additional investigation.
3. The next stage is **data visualization**, which is essential for comprehending the underlying structure of the data. When compared to just numerical summaries, visuals like histograms, box plots, and scatter plots help the reader understand the data more quickly and intuitively. They enable you to support or refute initial hypotheses and support the development of new ones.
4. **Insight Generation** is the last step, bringing everything together. You ought to be able to resolve your initial queries by fusing the numerical summaries and the visual depictions, and you ought to be able to come up with actionable or further-researchable insights. Additionally, you can point out areas where more information might be required for a more complete understanding.

It's crucial to keep track of your conclusions, presumptions, and any decisions you make throughout this process. In addition to keeping your analysis transparent, proper documentation leaves a trail for anyone who might later review or expand on your work. Knowing how the EDA works is like having a map for a challenging journey. Using the appropriate tools and a growth mindset guarantees that you examine all the essential aspects of your data. With this foundational knowledge in place, let's switch gears and discuss the tools and methods that will enable you to perform EDA efficiently.

Tools and Techniques

Your toolkit can be as varied in exploratory data analysis as the data you're exploring. The depth and breadth of your analyses can be greatly improved by understanding the various tools and techniques at your disposal, much like how a well-stocked toolbox helps a craftsman do their best work. Let's start by discussing software and libraries. Due to their robust data manipulation libraries, languages like Python and R have established themselves as

industry standards in the data analysis industry. Python provides libraries for data manipulation, data visualization, and advanced statistical modeling, including Statsmodels and Pandas. Like Python offers a wide range of packages, R offers ggplot2 and dplyr for data manipulation and visualization, respectively. Additionally, you might come across specialized software like Tableau, **Power BI, or Excel**, which enables interactive data visualization and is especially helpful for presenting findings to stakeholders. Second, understanding the statistical methods that make up the foundation of EDA is crucial. The dataset's shape, dispersion, and central tendency are all summarized by descriptive statistics. Using inferential statistics, you can apply the findings from your data sample to a larger population. This segment covers understanding distributions, testing hypotheses, and working with confidence intervals. Data transformation techniques like normalization and standardization become crucial when working with variables of various scales or units. Additionally, feature engineering—the process of developing new variables that capture aspects of the data—is frequently used. Dimensionality reduction methods, like Principal Component Analysis (PCA), can also be applied to make the data more manageable and more comprehensible. Being adaptable and willing to try new methods as your data comprehension develops is key. You are now equipped to delve into the specifics of actual data analysis because you comprehensively understand the available techniques and tools. After that, we will explore univariate analysis, where you will discover how to analyze each variable to comprehend its distribution, **behavior**, and impact on your dataset.

Univariate Analysis

It's time to explore the various layers of data analysis after navigating the Exploratory Data Analysis (EDA) fundamentals and becoming familiar with its basic tools and techniques. First up is a type of analysis called a univariate analysis, which looks at just one variable at a time. Though it might seem simple, don't undervalue its strength. Each variable's behavior, distribution, and summary statistics can be thoroughly understood to provide profound insights and direct subsequent, more intricate analyses. This section will define Univariate Analysis and discuss why it's an essential first step in any data analysis process. After that, we will divide our discussion into two main groups: continuous variables and categorical variables. You'll discover

analysis and visualization methods for each category, each one completing the picture your dataset represents. By the end of this section, you'll be skilled at examining single variables to discover their unique traits, behaviors, and patterns. This crucial step in the EDA process lays the foundation for more complex analyses like bivariate and multivariate analysis, which we will examine later.

Analyzing Continuous Variables

Continuous variables have an infinite number of possible values within a specified range. Consider elements that can change continuously, such as age, income, or temperature. These are not limited to fixed, distinct categories. To predict stock prices, analyze patient metrics in healthcare, and even study social science trends like voter turnout based on age, it is crucial to analyze continuous variables. One of the most popular tools for analyzing continuous variables is the histogram. It involves counting how many data points fall into each bin after dividing the entire range of values into a series of intervals, or bins. Use histograms to determine the distribution's shape, detect skewness, and locate potential outliers in your data. If you're looking at user engagement on a website, for instance, a histogram might show that most users only stay for 5 to 10 minutes per session, which is important information for improving website design or ad placement. The following visual depicts multiple histograms with different types of distributions.

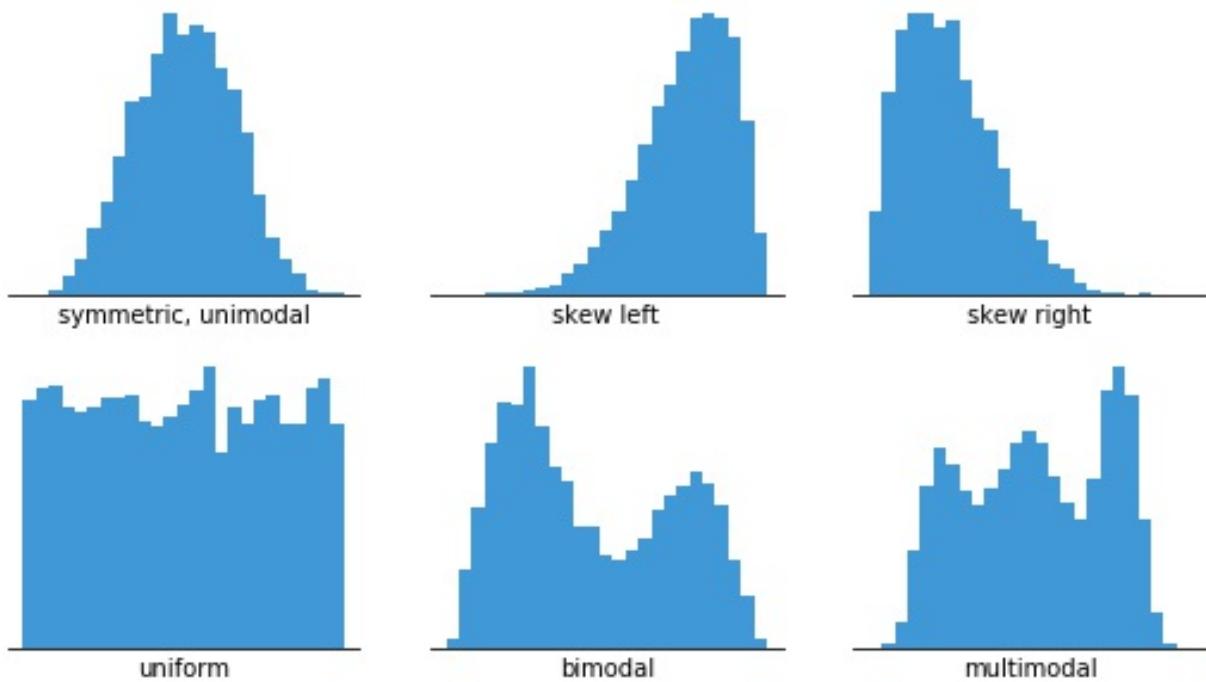


Figure 13.1: Histograms with various examples of distributions

By concentrating on quartiles, box plots, also referred to as box-and-whisker plots, provide a different perspective on your data. While the "whiskers" denote variability outside the interquartile range (IQR), the "box" represents the IQR, which contains the middle 50% of the data. The spread and skewness of your data can be understood using box plots, which are also great for spotting outliers. In a retail scenario, for instance, a box plot could quickly display the range of transaction amounts, allowing you to determine whether a particular range is unusually high or low and necessitates further investigation. The following visual displays a box plot with each section labelled.

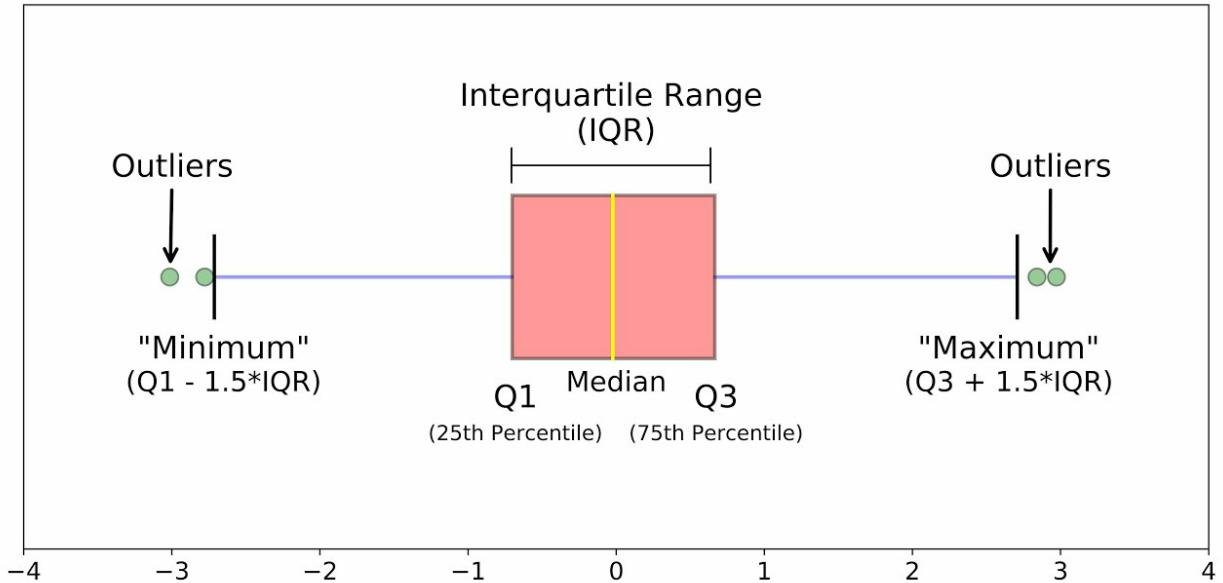


Figure 13.2: Box plot

Density plots, which are slick iterations of histograms, are used to see how the data distribution is shaped. Density plots provide a more continuous view of the data distribution than histograms, which are constrained by the rigid structure of bins. This makes comparing the distribution of several variables at once simpler. Density plots can be used in market research to compare the distribution of customer satisfaction ratings across various product categories and pinpoint areas that require improvement. The following picture displays an example of a visual with multiple density plots.

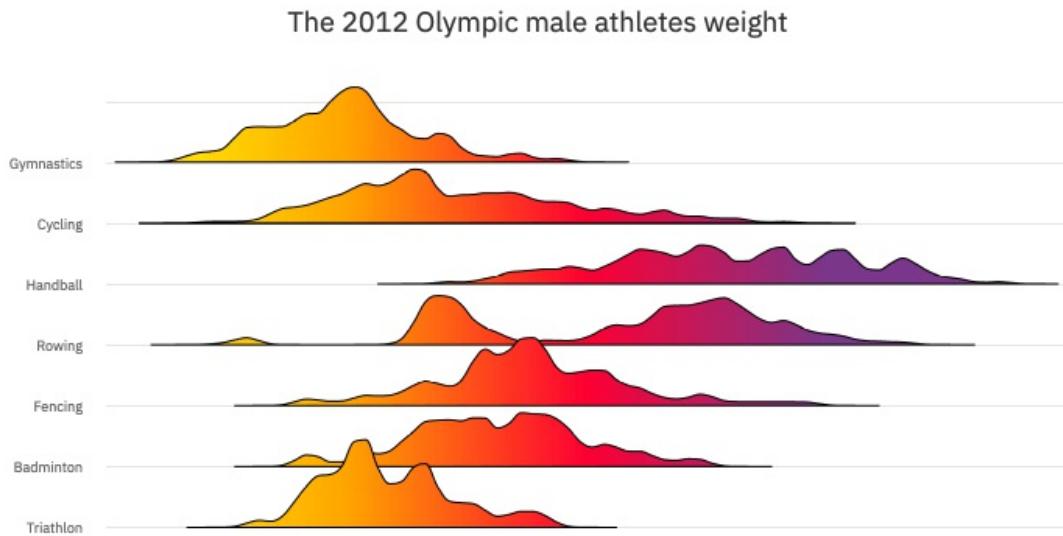


Figure 13.3: Distribution of weights of different categories visualized by density plots

Histograms, box plots, and density plots are three tools that each provide a different perspective on your continuous variables. The quality and interpretability of your analysis can be significantly impacted by knowing when to use which tool. After examining continuous variables, let's now look at categorical variables, a different species in the data zoo. Understanding them is equally important as they are frequently key to significant classifications and groupings in your data.

Analyzing Categorical Variables

Variables with discrete categories or labels are known as categorical variables. They frequently stand in for qualitative characteristics like gender, product type, or levels of customer satisfaction (such as "happy," "neutral," or "unhappy"). These variables play a crucial role in a variety of disciplines, including marketing for segmenting customer groups, healthcare for categorizing patient outcomes, and social sciences for gathering survey responses. One of the simplest and most efficient ways to visualize categorical variables is with **bar charts**. They show the actual categories along the horizontal axis and the frequency of each category along the vertical axis. When comparing the sizes of various groups in your data, bar charts come in

very handy. To determine which marketing channels are most successful, a marketing analyst, for instance, might use a bar chart to compare the number of customers acquired through various channels, such as social media, email, organic search, etc.

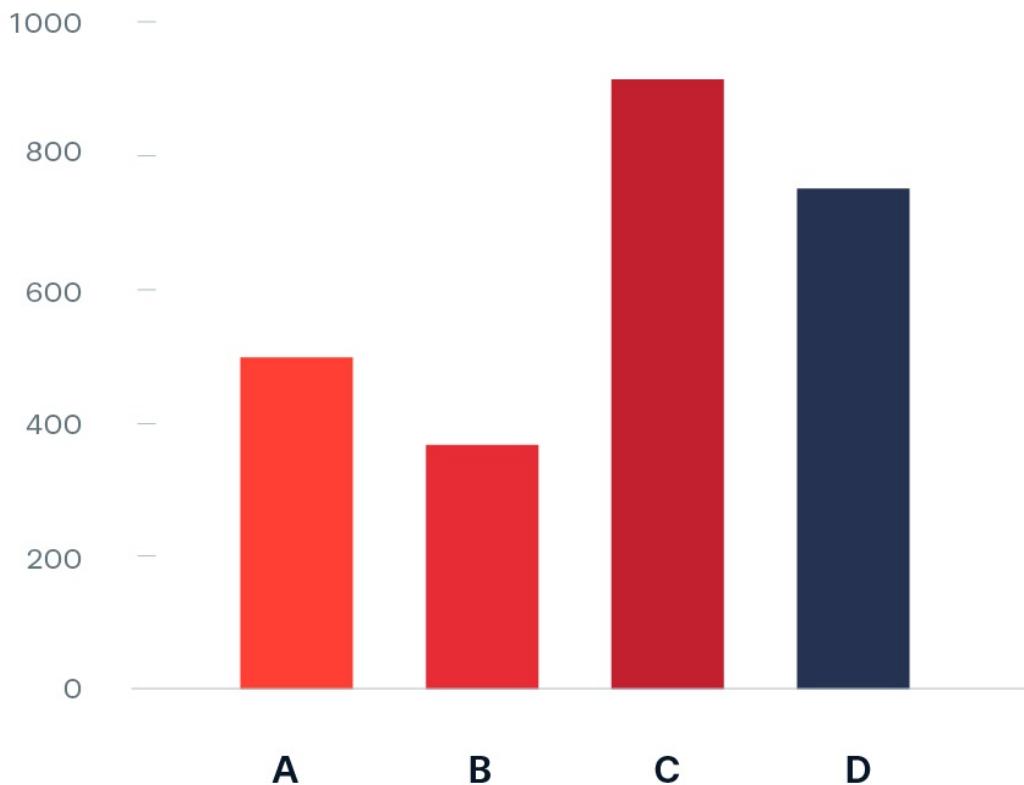


Figure 13.4: Example of a bar chart

Pie charts, which show each category's share of the total, provide a comprehensive view of your categorical data. They work best when you have

a small number of types, although they are visually appealing. A pie chart, for instance, could show the proportion of customer feedback broken down into three categories: positive, neutral, and negative. This would provide stakeholders with a quick overview of the general customer mood.

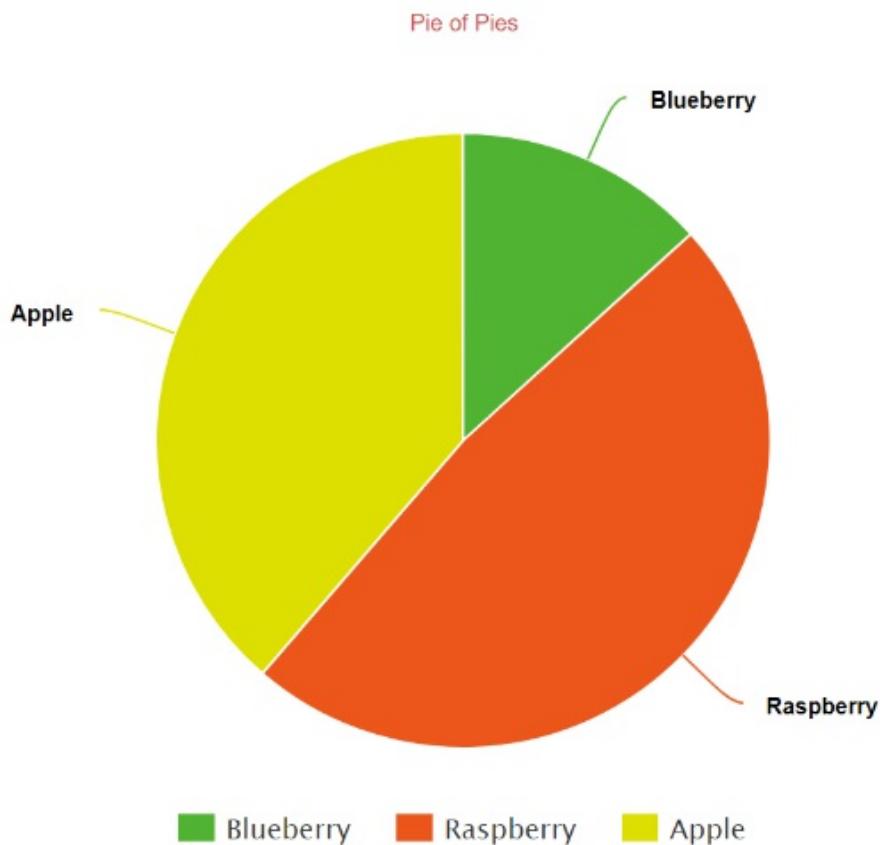


Figure 13.5: Pie chart with three sections

The distribution of each category is numerically depicted in **frequency tables**, which frequently include percentages. Frequency tables are incredibly informative and enable precise comparisons, despite not having the same visual appeal as bar or pie charts. For instance, frequency tables in educational research could show the proportion of students who achieved in various grade categories (A, B, C, etc.), giving teachers a detailed understanding of class performance.

Age (years)	Frequency	Percentage (%)
below 20	1	0.3
21-25	2	0.7
26-30	75	24.6
31-35	66	21.6

Figure 13.6: Example of frequency table with percentages

Your specific requirements, the complexity of your data, and the message you want to convey will determine which of the following to use: bar charts, pie charts, frequency tables, etc. Your overall data analysis skills will increase in depth and breadth when you learn how to use these tools to analyze categorical variables. We are well-equipped to advance our exploratory journey now that we have delved into the nuances of analyzing categorical variables. The next step is bivariate analysis, where we'll look at the connections between two variables and draw conclusions that one-variable analyses could not.

Bivariate Analysis

After learning the foundations of exploratory data analysis (EDA) and diving deep into univariate analysis, you now have the knowledge necessary to comprehend each variable separately. It's time to broaden our focus and investigate the relationships between various variables. Welcome to the world of bivariate analysis, where each variable interacts with the others to reveal insights that are frequently hidden in analyses with only one variable. We will examine what bivariate analysis is in this section and why it is an essential component of EDA. We'll discuss important ideas like Correlation vs.

Causation and explore several methods for observing and analyzing relationships between two types of variables, including Continuous-Continuous, Categorical-Categorical, and Continuous-Categorical. By the time you finish reading this section, you will be able to comprehend each variable independently and investigate how they interact to create patterns or trends. You'll discover how to recognize correlations among variables and develop theories for additional research, laying the groundwork for even more complex multivariate analysis.

Understanding bivariate analysis

In bivariate analysis, two variables are compared and analyzed to find patterns, trends, and relationships between them. Bivariate analysis enables you to comprehend how one variable influences or is correlated with another, as opposed to univariate analysis, which examines each variable separately. This type of analysis is fundamental to many industries, including healthcare, where it's used to investigate the link between interventions and results, finance, where it's used to comprehend how various economic indicators affect stock prices, and marketing, where it's used to assess how various distribution channels affect consumer engagement. You might wonder why it's so crucial to consider two variables at once. The depth of understanding that such an analysis offers holds the key to the solution. Bivariate Analysis, for instance, may show that a marketing campaign primarily attracted users between the ages of 18 and 25 while Univariate Analysis might indicate that a marketing campaign attracted many new users. When making decisions, these types of insights are essential because they help you focus your efforts more efficiently. Furthermore, multivariate analysis, which involves three or more variables and is more complex, is a stepping stone from bivariate analysis. Understanding the fundamentals of bivariate analysis prepares you for more sophisticated and nuanced data explorations by teaching you to see beyond the obvious and unearth hidden relationships in your data. After establishing the basics of bivariate analysis and why it is so important, let's explore one of the most crucial aspects of this type of analysis: knowing the distinction between correlation and causation. You'll be able to steer clear of common pitfalls and interpret your data more precisely as a result.

Correlation vs Causation

The distinction between correlation and causation is one of the most important concepts to grasp in bivariate analysis. Correlation denotes a statistical relationship between two variables, indicating that the other variable tends to change similarly when one variable changes. Nevertheless, correlation does not prove causation. Simply because two variables are correlated does not imply that one is the cause of the other. The standard error of mistaking correlation for causation can lead to erroneous conclusions and misguided actions. Suppose, for instance, that you observe a strong positive correlation between ice cream sales and the number of drowning incidents over several years. While it may be tempting to believe that ice cream sales cause an increase in drownings, it is more likely that both are influenced by a third factor: the weather, specifically hot summer temperatures. This is known as a "confounding variable," and when interpreting correlations, it is essential to consider such possibilities. The example below depicts a correlated relationship between two logically unrelated events, divorce rates and margarine consumption. Despite the statistical significance, one event does not cause the other.

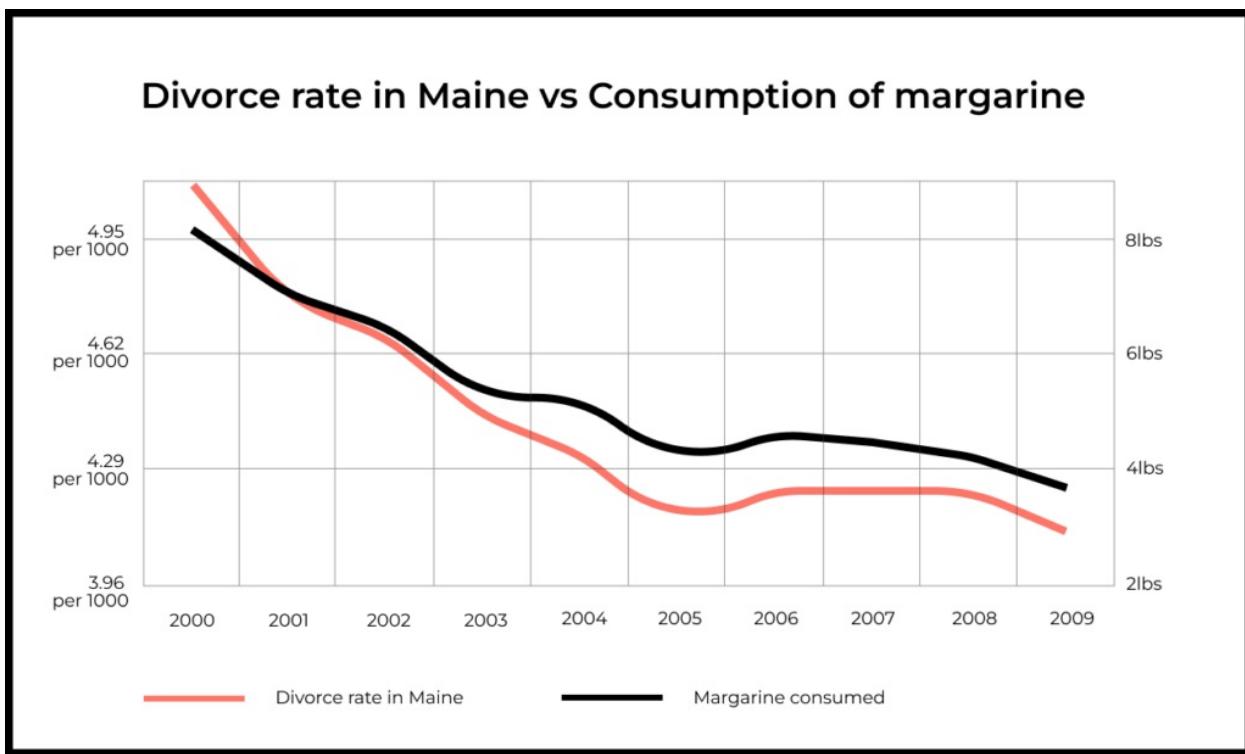


Figure 13.7: Example of unrelated events having high correlation

Understanding the distinction between correlation and causation is crucial because it influences how you interpret your data and the questions you can ask. Correlation, for instance, can point you toward potential relationships worth investigating, but proving causation typically requires more rigorous, controlled experiments or advanced statistical methods such as regression analysis. As a result of elucidating the crucial distinction between correlation and causation, we are now better equipped to investigate the various techniques for visualizing the relationships between two continuous variables. This will enhance our Bivariate Analysis toolkit and help us interpret such visual representations with greater nuance.

Visualizing relationships between two continuous variables

Understanding the relationship between two continuous variables can be particularly enlightening in bivariate Analysis. While summary statistics such as the correlation coefficient provide a numerical measure of the relationship, visualizations provide a more intuitive depiction of these relationships.

Scatter plots, line charts, and heatmaps are three techniques commonly used to visualize the relationship between two continuous variables. **Scatter Plots:** A scatter plot positions each data point on a two-dimensional plane according to its values for the compared variables. This plot is handy for determining a relationship's type (linear, exponential, etc.) and strength (strong, weak, etc.).

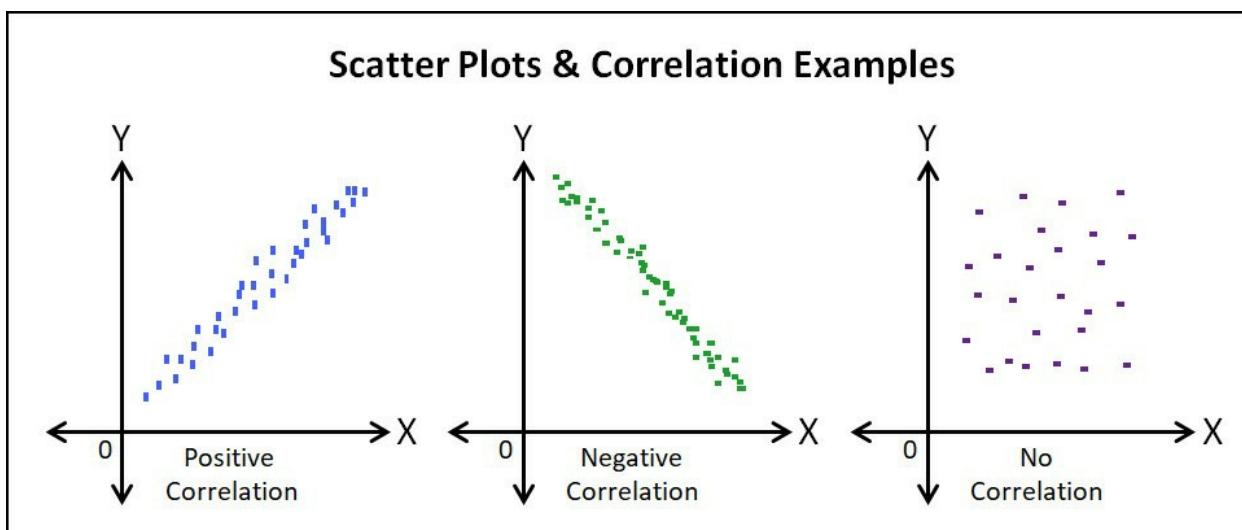


Figure 13.8: Example of multiple scatterplots depicting the different types of relationships between variables

Line charts are most valuable when the data points can be ordered meaningfully, typically chronologically. For example, line charts are ideal for displaying trends over time, such as a company's quarterly revenue and advertising spending.

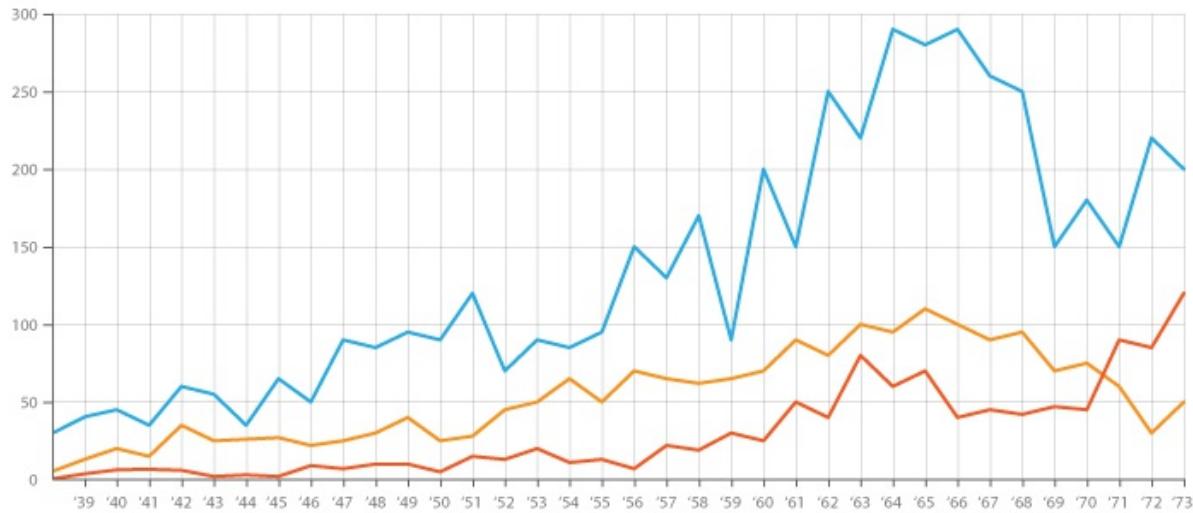


Figure 13.9: Example of multiple line charts in one graph

Mastering these visualization techniques will allow you to emphasize crucial relationships between continuous variables, thereby enhancing your overall analysis. Not only do they facilitate comprehension of your findings, but they also help you generate hypotheses for further investigation or testing. Equipped with the knowledge and tools necessary to analyze and visualize the relationship between two continuous variables, let's move on to the more advanced multivariate analysis.

Multivariate analysis

With a solid foundation in Univariate and Bivariate Analysis under our belts, we are now at the crossroads of more complex data relationships: the realm of Multivariate Analysis. As its name suggests, this method investigates relationships between more than two variables, allowing us to explore and decode complex interdependencies within our data sets. Multivariate Analysis is not merely an extension of the bivariate techniques covered thus far. It is a rich, multifaceted domain that can help uncover insights that would otherwise

remain concealed if only two variables were considered. Multivariate Analysis provides the tools and methodologies necessary to segment customers into distinct clusters based on their purchasing habits or to understand how multiple factors simultaneously impact sales revenue. In this section, we will traverse the expansive terrain of Multivariate Analysis, revealing its techniques and applications. We will discover how to manage multiple variables without becoming overwhelmed and how to extract concise, actionable insights from complex data scenarios.

Heatmaps

In multivariate analysis, heatmaps are a potent tool for illustrating the relationships between more than two variables in an easily digestible format. In a heatmap, data values are displayed as colors, providing a quick visual summary that can reveal patterns, trends, and correlations in complex datasets. Essentially, it is a table-like representation in which individual cells are colored according to the variables' magnitude, with the color's intensity typically denoting the value's relative prominence.

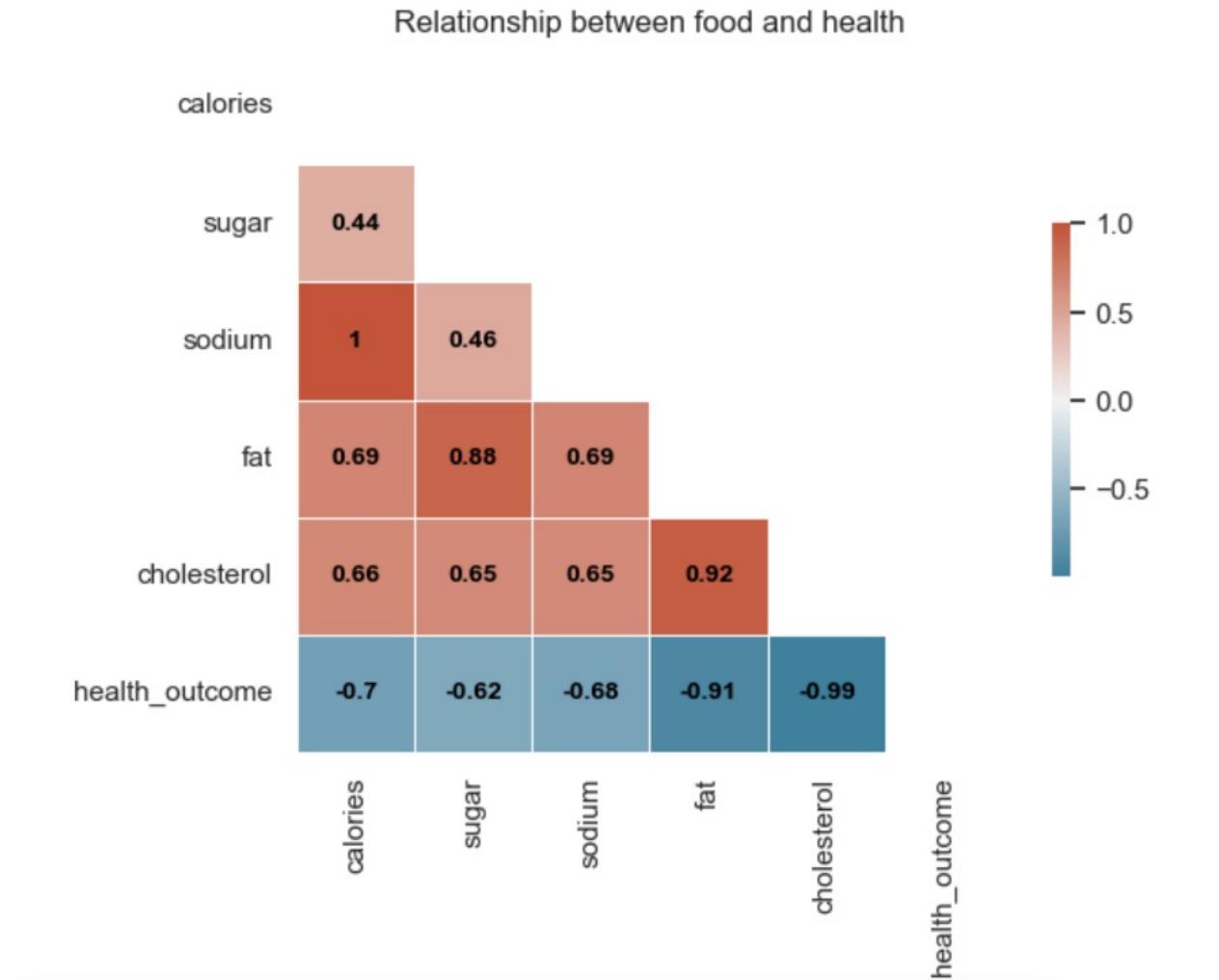


Figure 13.10: Heatmap depicting the relationships between food and health

In e-commerce analytics, for instance, a heatmap could be used to investigate the relationship between various customer metrics (such as age, frequency of purchases, and average spending) and product categories. A glance at the heatmap would immediately reveal the demographic characteristics that are most strongly associated with each product category, thereby guiding marketing strategies. Similarly, heatmaps are frequently used in healthcare analytics to examine the correlation between age, BMI, and lifestyle factors, and the incidence of health conditions. They are also commonly used in finance to visualize the correlations between various sectors or assets, providing insights into portfolio risk and diversification. One of the primary benefits of heatmaps is their ability to condense large amounts of multivariate

data into a single visual format, enabling analysts to quickly comprehend intricate relationships. It is an indispensable tool for exploratory analysis, mainly when dealing with high-dimensional data. Now that we've discussed the utility of heatmaps, let's move on to another graphical tool that is essential for understanding pairwise relationships in a multivariate context: scatterplot matrices or pair plots. This method provides a broader perspective, allowing simultaneous examination of relationships between multiple pairs of variables.

Pair plots

Pair plots, also known as scatterplot matrices, are indispensable in multivariate analysis for examining the pairwise relationships between variables. Each variable in your dataset is paired with another variable in a pair plot grid. A histogram or kernel density plot displaying the distribution of the variable in question is typically displayed on the diagonal. Off the diagonal, you will find scatter plots between pairs of variables, which reveal any correlation between them.

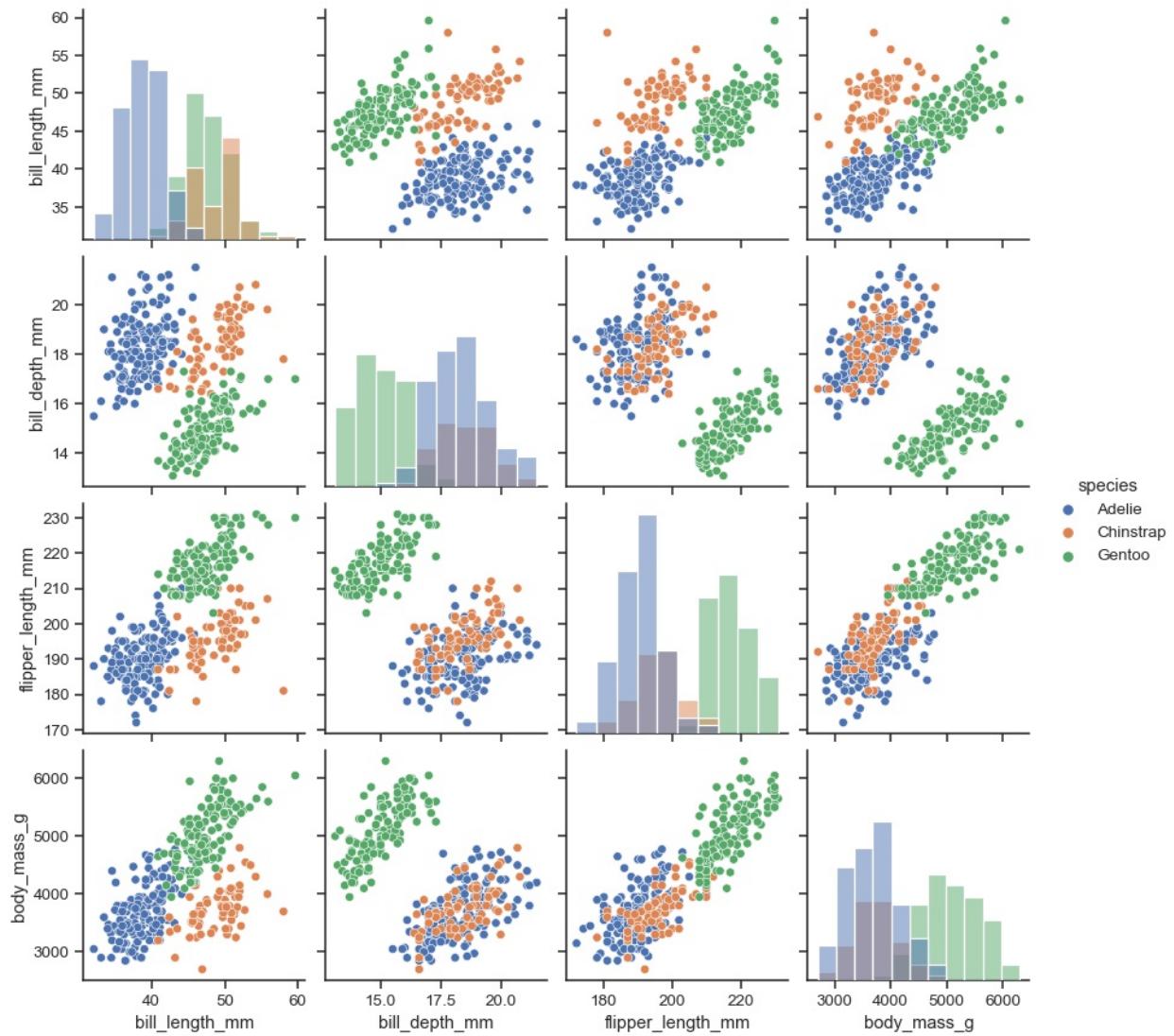


Figure 13.11: Pair plot example done in python

Pair plots are especially useful when you have a moderate number of variables and want to quickly determine whether any potential relationships warrant further investigation. They are an excellent starting point before diving into more detailed and specific analyses, such as regression models or more complex multivariate techniques.

Summary

Exploratory Data Analysis (EDA), a cornerstone of data analytics, was explored in this chapter. Understanding how EDA turns raw data into

actionable insights was our first step. The EDA process revealed a structured approach to complex data sets. EDA tools and techniques include statistical and graphical methods for data exploration. Univariate Analysis—studying continuous and categorical variables—was thoroughly examined.

Histograms, box plots, and bar charts help us understand data features. We advanced to Bivariate Analysis to understand the relationships between continuous and categorical variables. The chapter concluded with Multivariate Analysis, where heatmaps and pair plots helped us understand complex multivariable relationships. After finishing this chapter, we should remember that EDA is a narrative process that lets us meaningfully interact with data. The goal is to find the data's story, which can inform decisions, policy, or even lives. We are prepared to put theory into practice now that we have mastered the foundations of EDA. The Exploratory Data Analysis Case Study in the following chapter will give us a hands-on introduction to data storytelling.

14 Data Cleaning

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



Data cleaning is a fundamental step that guarantees the accuracy and dependability of the insights drawn from data in the large and developing field of data analytics. Even the most advanced analytical models may produce false or misleading conclusions without clean and well-structured data. Data cleaning is a rigorous procedure that molds the quality of data, thereby affecting the conclusions reached and the decisions taken considering those conclusions. It is not merely a preliminary stage. Analysts must complete several complex activities and overcome several obstacles to ensure that the data is prepared for analysis. Each phase in the data cleaning process has a specific purpose and involves many subtle nuances, such as handling missing numbers, coping with inconsistent formats, and integrating multiple data sources. This chapter will examine the crucial facets of data cleansing, thoroughly explaining its significance, difficulties, and methods. The following are the primary learning goals for this chapter:

- Understanding the Importance of Data Cleaning
- Identify common data cleaning challenges
- Dealing with missing values
- Handling duplicate and missing data

By the end of this chapter, you should have a good grasp of what data cleaning is and how important it is to the process of data analytics. With the

help of real-world examples and Python code snippets, you will be prepared to take on the obstacles that data cleaning poses in the real world. You will be able to convert unorganized, unclean data into a resource that is ready for intelligent analysis. This chapter includes insightful information that will improve your capacity to handle data precisely and confidently, regardless of your level of experience.

Technical requirements

You can find all materials for this chapter in the following address:
<https://github.com/PacktPublishing/Becoming-a-Data-Analyst-First-Edition/blob/main/Chapter14-Data-Cleaning/Chapter-14-Data-Cleaning.ipynb>

Importance of data cleaning

To improve data quality, errors and inconsistencies are found and corrected (or removed). This process is known as data cleaning. It covers various topics, such as handling missing numbers, eliminating duplicates, handling outliers, changing data types, and more. Data cleansing is an essential stage in the data analysis process, not only preliminary. Its significance derives from the requirement to guarantee that the data is correct, consistent, and analytically ready, serving as a strong foundation for any data-driven project.

Impact on data quality

Any analytics project must ensure data quality and data cleansing is essential to achieving this quality. Dirty or unclean data is full of flaws, inaccuracies, and discrepancies that can produce false findings. Analysts carefully clean the data to remove these flaws, improving the accuracy and integrity of the data. This procedure ensures reliable data support following analyses and produces insights that accurately depict the underlying patterns and phenomena. In other words, data cleaning turns unrefined, untrustworthy data into a polished resource.

Relevance to business decisions

In the context of business, data-driven decisions frequently play a crucial role in determining strategies, directing operations, and impacting results. The quality of the underlying data will determine how well these decisions turn out. Poorly cleaned data may result in mistaken tactics and incorrect conclusions, potentially squandering time, money, and a competitive edge. On the other hand, well-cleansed data enables reliable and exact analysis, producing conclusions that are consistent with reality. Businesses may make educated, self-assured decisions that foster success and innovation by ensuring clean data. Although it is occasionally disregarded, the data cleansing process is essential to the development of actionable intelligence and the establishment of a business culture that is data-driven.

Common data cleaning challenges

Use (P – Regular) numbered steps for sequential instructions. Remember, readers will often be practically implementing your lessons, looking between your content and their own computer screen. Numbered steps help them to keep their place.

1. Step one. (L – Numbers)
2. Step two.
3. Step three.

Inconsistent formats

Dealing with mismatched formats is one of the frequent difficulties in data cleaning. There are many different sources of data, each with its own format for dates, monetary symbols, and other rules. Confusion can be caused by these inconsistencies, which can also prevent data from integrating smoothly. For data to be adequately processed and comprehended, it must be aligned to a standard format. Early correction of format issues makes a guarantee that the data behaves consistently, allowing for more efficient and precise analysis. The following are examples of fixing inconsistent dates in python: Standardizing dates

```
import pandas as pd  
# Create a DataFrame with inconsistent date formats
```

```
df = pd.DataFrame({'date': ['2022-08-01', '01/08/2022', 'August 1, 2022']})
# Convert the date column to a uniform format
df['date'] = pd.to_datetime(df['date'])
print(df)
```

Instructions:

1. **Import Pandas:** Use `import pandas as pd` to import the Pandas library.
2. **Create DataFrame:** Create a DataFrame with inconsistent date formats.
3. **Convert Dates:** Use `pd.to_datetime` to convert the date column to a uniform format.

Converting currency symbols

```
# Create DataFrame with inconsistent currency formats
df = pd.DataFrame({'price': ['$100', '€200', '£300']})
# Remove currency symbols and convert to float
df['price'] = df['price'].replace({'$': '', '€': '', '£': ''}, regex=True).astype(float)
print(df)
```

Instructions:

1. **Create DataFrame:** Create a DataFrame with inconsistent currency formats.
2. **Remove Currency Symbols:** Use the `replace` method with a regex pattern to remove currency symbols.
3. **Convert to Float:** Use `astype(float)` to convert the price column to a floating-point number.

Standardize phone numbers

```
# Create DataFrame with inconsistent phone number formats
df = pd.DataFrame({'phone': ['123-456-7890', '(123) 456-7890', '123.456.7890']})
# Standardize phone numbers to a consistent format
df['phone'] = df['phone'].replace(r'[^\d]', '', regex=True)
print(df)
```

Instructions:

- Create DataFrame:** Create a DataFrame with inconsistent phone number formats.
- Standardize Phone Numbers:** Use the `replace()` method with regex to remove all non-digit characters, standardizing the format.

Misspellings and Inaccuracies

Misspelled words, typographical errors, and other irregularities can significantly impact data quality. These inaccuracies may result from human error during data entry or flawed automated procedures. Such errors may result in improper classification, inappropriate aggregation, or other analytical blunders. Maintaining data integrity depends on spotting and fixing these mistakes. By meticulously addressing typos and other errors, analysts ensure that the data accurately conveys the intended information, enabling reliable analysis. The following are examples of fixing misspellings in python: Misspelled categories

```
# Create a DataFrame with misspelled categories
df = pd.DataFrame({'category': ['Appl', 'Orang', 'Bnan', 'App1']}
)
# Define a mapping to correct the spellings
corrections = {'Appl': 'Apple', 'Orang': 'Orange', 'Bnan': 'Bana
na'}
# Apply the mapping to correct the misspellings
df['category'] = df['category'].map(corrections).fillna(df['cate
gory'])
print(df)
```

Instructions:

- Import Pandas:** If not already imported, use `import pandas as pd`.
- Create DataFrame:** Create a DataFrame containing a column with misspelled categories.
- Define Corrections:** Create a dictionary that maps misspelled words to their correct forms.
- Apply Corrections:** Use the `map()` method along with `fillna()` to apply the corrections to the misspelled categories.

Using fuzzy matching

```
from fuzzywuzzy import process
# Create DataFrame with misspelled cities
df = pd.DataFrame({'city': ['Nw York', 'Sn Francisco', 'Chcago']
})
# List of correct city names
correct_names = ['New York', 'San Francisco', 'Chicago']
# Correct misspellings using fuzzy matching
df['city'] = df['city'].apply(lambda x: process.extractOne(x, correct_names)[0])
print(df)
```

Instructions

- Import FuzzyWuzzy:** Import the process module from FuzzyWuzzy .
- Create DataFrame:** Create a DataFrame with misspelled cities.
- List Correct Names:** Create a list of correct city names.
- Apply Fuzzy Matching:** Use the apply method along with FuzzyWuzzy's extractOne to correct misspellings.

Misspelled product names

```
# Create DataFrame with incorrect product names
df = pd.DataFrame({'product': ['Laptoop', 'Smarttphone', 'Tablet t']}
)
# Define specific corrections
corrections = {'Laptoop': 'Laptop', 'Smarttphone': 'Smartphone',
 'Tablett': 'Tablet'}
# Replace incorrect product names with correct ones
df['product'] = df['product'].replace(corrections)
print(df)
```

Instructions

- Create DataFrame:** Create a DataFrame with incorrect product names.
- Define Corrections:** Create a dictionary mapping incorrect names to correct ones.
- Replace Misspellings:** Use the replace() method to apply the corrections.

Duplicate records

An additional frequent problem in data cleaning is duplicate records. These

identical entries may result from incorrect system settings, data merging from many sources, or repeated data entering. By overrepresenting particular data points, duplicate records might skew analysis and produce biased results. Careful investigation and applying specific tools or algorithms are frequently required to identify and eliminate duplicates. Addressing duplicate records correctly ensures that each piece of information is appropriately and individually represented, which helps create a more accurate and objective picture of the data environment. The following are examples of fixing duplicate records in python:

Example 1: Removing all duplicates

```
# Create a DataFrame with duplicate records
df = pd.DataFrame({'A': [1, 2, 2, 3], 'B': [5, 6, 6, 7]})
# Remove duplicate rows
df = df.drop_duplicates()
print(df)
```

Instructions

1. **Import Pandas:** If not already imported, use `import pandas as pd`.
2. **Create DataFrame:** Create a DataFrame that contains duplicate rows.
3. **Remove Duplicates:** Use the `drop_duplicates` method to remove duplicate rows.

Example 2: Removing duplicates based on a specific column

```
# Create DataFrame with duplicate records in specific column
df = pd.DataFrame({'A': [1, 2, 2, 3], 'B': [5, 6, 6, 7]})
# Remove duplicates based on column 'A'
df = df.drop_duplicates(subset=['A'])
print(df)
```

Instructions

1. **Create DataFrame:** Create a DataFrame with duplicate records in a specific column.
2. **Remove Duplicates:** Use the `drop_duplicates()` method with the `subset()` argument to remove duplicates based on a specific column.

Ex 3: Last occurrence

```
df = pd.DataFrame({'A': [1, 2, 2, 3], 'B': [5, 6, 6, 7]})
```

```
# Remove duplicates but keep the last occurrence
df = df.drop_duplicates(keep='last')
print(df)ample 3: Keeping last occurrence of duplicates
```

Instructions

1. **Create DataFrame:** Create a DataFrame with duplicate records.
2. **Remove Duplicates:** Use the `drop_duplicates()` method with the `keep='last'` argument to keep the last occurrence of duplicates.

Formatting errors, misspellings, duplicate records, and integration issues threaten data integrity and usability. Techniques, tools, and data context knowledge are needed to solve these problems. Managing these challenges efficiently improves dataset quality and reliability, enabling insightful and accurate analysis. After these foundational cleaning tasks, we must address missing values, another important part of data preparation. Missing values introduce ambiguity and bias into an analysis, so their proper handling requires careful consideration of the causes and mitigation strategies. In the next section, we will discuss these complexities and how to manage missing data for a robust analytical process.

Dealing with missing values

In data analysis, missing values are a common problem often hard to avoid. When there are missing values in a dataset, it can have a significant effect on the quality and reliability of statistical analyses. This can lead to biased results or even make some analyses impossible. Taking care of missing values in the right way is, therefore, an essential part of the data cleaning process that requires careful thought and expertise. There is no one way to deal with missing values, and different strategies can be used depending on the type and cause of the missing data. In this section, we'll talk about the main reasons why there are missing values, look at different ways to deal with them, and think about how different approaches can be changed to fit the needs and limits of a data analysis project. Analysts can ensure that the conclusions they draw from the data are strong, fair, and reliable by understanding and handling missing values well.

Causes of missing values

Missing values in a dataset are a common problem every data analyst has to deal with. Data might be missing for several reasons:**Data collection errors:** Errors in the data collection process, equipment failures, and human error can all result in missing data occasionally. Incomplete information can result from data entry mistakes or mistakes made by the people collecting the data. Depending on the kind of failure, these errors may be random or systematic.**Non-response:** This phenomenon, which is particularly prevalent in survey data, happens when participants altogether forego participating in the study or fail to respond to specific questions. Non-response can be selective, meaning the missing data is relevant to the topic under investigation, which could skew the findings.**Data integration:** Combining data from different sources, such as various databases or file formats, may result in information misalignment or inconsistencies. Gaps in the integrated dataset may result from variations in standards, units, or coding between various sources.**System limitations:** Data loss or incompleteness may be caused by technical issues, compatibility issues, or restrictions within a data storage or retrieval system. Understanding the underlying reason for missing data is essential because it aids in selecting the best approach for handling it and reducing potential biases.

Strategies for handling missing values

The handling of missing values is an important step in data preparation that can have a big impact on the validity and reliability of the analysis. Missing data poses a serious challenge that can skew the results, reduce statistical power, or make the data impossible to interpret. It is not just an inconvenience. There are many ways to deal with missing values, from straightforward removal to sophisticated imputation techniques. The type of missing data and the dataset's underlying relationships influence the choice of an appropriate strategy. Maintaining the integrity of the data and ensuring that the subsequent analysis produces accurate and useful insights require an understanding of the types and causes of missing data as well as the appropriate technique to handle them.**Removal** of records with missing values might be the easiest method, but it's frequently tricky. The potential loss of important data is the main worry. The removal may not introduce bias

if the missing values are distributed randomly, but it will shrink the dataset and potentially reduce statistical power. Removal can introduce serious bias and misrepresent the underlying relationships in the data if the missingness is systematic (related to the phenomenon being studied). In the following code, a Pandas DataFrame is created with some missing values (None) for 'Age' and 'Income'. Using the `dropna()` method without specifying the axis, rows containing any missing values are removed. This method can be useful when the dataset is large, and the missing data is random, not causing a significant loss of information.

```
import pandas as pd
# Creating data with missing values
data = {
    'Age': [25, 30, None, 22],
    'Income': [50000, 70000, 80000, None]
}
df = pd.DataFrame(data)
# Removing rows with any missing values
df_removed = df.dropna()
print(df_removed)
```

A more sophisticated method that keeps the size of the original dataset is **imputing**, or filling in missing values. Various methods can be used, depending on the type of data:**Mean/median/mode imputation:** Simple statistical measures like mean, median, or mode can be used to impute missing values. Although simple to use, this method may alter the distribution of the original data, particularly if the missingness is not random. Here, instead of removing rows, the columns containing any missing values are removed using `dropna(axis=1)`. This is often applied when specific features have many missing values, and their removal won't affect the analysis significantly.

```
import pandas as pd
from sklearn.impute import SimpleImputer
# Creating data with missing values
data = {
    'Age': [25, 30, None, 22],
    'Income': [50000, 70000, 80000, None]
}
df = pd.DataFrame(data)
# Using mean imputation for missing values
mean_imputer = SimpleImputer(strategy="mean")
```

```
df_mean_imputed = pd.DataFrame(mean_imputer.fit_transform(df), columns=df.columns)
print(df_mean_imputed)
```

Interpolation: Missing values in time series or sequential data can be estimated using nearby observations. While doing so uses data trends that already exist, if the interpolations are inaccurate, noise may be introduced. This code will estimate the missing temperature values based on a linear relationship between the surrounding known temperatures.

Interpolation is particularly useful when the data follows a pattern or trend, and the missing values can be estimated by considering the existing data points. This method can provide more accurate estimates for missing values than mean or median imputation, mainly when the data follows a specific temporal or spatial pattern.

```
import pandas as pd
import numpy as np
# Creating a sample time series data with missing values
date_rng = pd.date_range(start='2021-01-01', end='2021-01-10', freq='D')
data = {
    'date': date_rng,
    'temperature': [25, 28, np.nan, 27, 24, np.nan, 23, 22, 20,
19]
}
df = pd.DataFrame(data)
# Setting the date column as the index
df.set_index('date', inplace=True)
# Using linear interpolation to fill missing values
df_interpolated = df.interpolate(method='linear')
print(df_interpolated)
```

Using machine learning models: Predictive modeling-based imputation can be highly accurate, but it needs careful tuning and validation. Additionally, it is predicted that the relationships used to forecast missing values are applicable across the entire dataset. This code uses the `KNNImputer` class from scikit-learn to perform imputation based on the k-nearest neighbors algorithm. The missing values are predicted using the average of the 2 nearest neighbors (as specified by `n_neighbors=2`). This approach considers the relationships between features and can provide more accurate imputations when the data is missing at random (MAR).

```

from sklearn.impute import KNNImputer
# Creating data with missing values
data_with_missing = {
    'Feature1': [2, 4, None, 5],
    'Feature2': [3, 1, 2, None]
}
df_missing = pd.DataFrame(data_with_missing)
# Imputing missing values using K-Nearest Neighbors
knn_imputer = KNNImputer(n_neighbors=2)
df_knn_imputed = pd.DataFrame(knn_imputer.fit_transform(df_missing),
                                columns=df_missing.columns)
print(df_knn_imputed)

```

These code examples show different ways to deal with missing data. The method you choose should match the type and nature of the missing data and the goals of the analysis. In some situations, simple methods like removal or mean/median imputation may be enough, but in others, more advanced methods like K-Nearest Neighbors or iterative imputation may be needed. Understanding the structure of the data and why values are missing is key to choosing the right method and making sure that the way missing values are handled doesn't introduce bias or change the results of the analysis.

Types of missing data

When analyzing data, missing values are not an exception, but rather the rule. These missing entries are not just blank spaces; they often mean something important and should be dealt with carefully. But not every set of missing data is the same. Different types of data can be made based on the reason and pattern of the missing data. Understanding these types is important because it lets you know how to deal with missing values and ensures that the conclusions you draw from the data are still correct. This section goes into detail about the three main types of missing data, including what they are, how they work, and what you can do about them.

- **Missing completely at random (MCAR):** MCAR is a situation in which the missing data has no relationship with any of the variables that have been observed or that have not been observed. For example, think about a hospital where some random temperature readings are missing because of broken thermometers. Since the missing temperature data has nothing to do with any specific patient traits or health problems, it's just

a random thing that happens. In this case, removal might be a good idea because it's unlikely to bring in bias. Alternatively, the mean or median of the remaining data could be used to fill in missing temperature values. This would keep the size of the dataset the same without changing the relationships.

- **Missing at random (MAR):** MAR happens when the probability of missing data is linked to some data that has been seen but not to the missing data. In real life, an example could be a survey about income levels, where people with higher incomes are likelier to skip the question about income. Here, the income data is missing because of something else, like education level or job type, but not because of the income data itself. In this case, regression or other predictive models that use these variables to estimate the missing income values can be very useful. Multiple imputation, which creates several imputed datasets that show how uncertain the real missing income values are, can also make an analysis stronger.
- **Missing not at random:** The third type, MNAR, is when the missingness is related to the missing data that wasn't seen. Consider a study on mental health in which people with more severe symptoms are less likely to answer questions about their health. In this case, the missing mental health data is directly linked to the conditions at play. Sensitivity analysis can be used to figure out how different assumptions about the missing mental health data would affect the study's conclusions and evaluate how strong the results are. A more nuanced way to fill in missing values is to use "informed imputation," which uses outside information like general trends in mental health studies or domain-specific knowledge.

In data cleaning, figuring out what kind of data is missing and how to handle it is not just an important step. It's a key part of the analysis process that can have a big effect on the conclusions that can be drawn from the data. Whether the missing data is completely random or tied to things that can be seen or even things that can't be seen, knowing what it is helps analysts deal with it in a way that keeps the analysis's integrity. This nuanced way of dealing with missing data is important for turning raw, imperfect data into insights that can be used.

Dealing with duplicate values

Duplicate data, or entries that are repeated in a dataset, may seem like nothing important, but they can lead to wrong conclusions and bad decisions. Duplicates can cause numbers to be inflated, distributions to be off, and relationships to be misrepresented. Human mistakes, system bugs, or inconsistent data integration can cause them. To keep data accurate and reliable, it is essential to understand why duplicates happen and set up strong ways to find and get rid of them. This section talks about where duplicate data usually comes from and how to find and get rid of it.

Causes of duplicate data

Duplicate data doesn't just happen by chance; there are often specific reasons why it happens. When analysts know about these sources, they can plan for possible problems and adjust their data cleaning efforts accordingly. Duplicates can get into datasets in different ways, such as when people make mistakes when entering data or when something goes wrong with the technology. Each of these problems requires a different solution.

- **Human mistakes:** Duplicate entries often happen when people enter data by hand, and mistakes or misunderstandings lead to the same information being entered twice. For example, a customer might be listed twice in a sales database because their name was spelled differently, or their address was formatted differently.
- **System problems:** Duplicates can also be caused by problems with how data collection systems work. If a system glitch causes a transaction to be sent again, the same record could show up more than once in a financial database.
- **Inconsistent Data Integration:** When merging data from different sources or combining multiple databases, inconsistent labelling, formatting, or matching criteria can lead to duplicate records. In a healthcare setting, a patient might have different records in different departmental systems. If these records aren't correctly linked, they might be treated as separate entries.

Identification and removal

It's not as simple as finding exact matches and deleting them to find and get rid of duplicates. To make sure that valuable information doesn't get lost in the process, you have to have a deep understanding of the data, think about the context, and do it carefully. Combining manual inspection with automated algorithms is a smart way to find duplicates, even when there are differences and inconsistencies. This is followed by a careful removal process that keeps the data's integrity. The following are different example scenarios of finding and removing duplicates.

Scenario 1: Removing Duplicates Based on a Specific Column In this scenario, we want to identify duplicates based solely on the 'Customer_ID' column. This situation might arise when 'Customer_ID' is a unique identifier, and any repetition indicates a mistake.

```
import pandas as pd
# Creating a DataFrame with duplicate data
data = {
    'Customer_ID': [101, 102, 103, 101, 104, 102],
    'Name': ['Alice', 'Bob', 'Charlie', 'Alice', 'David', 'Bob']
}
    'Address': ['123 Main St', '456 Pine St', '789 Oak St', '123
Main St', '321 Cedar St', '456 Pine St']
}
df = pd.DataFrame(data)
# Removing duplicates based on the 'Customer_ID' column
df_no_duplicates_by_id = df.drop_duplicates(subset=['Customer_ID'])
print("\nDataFrame after Removing Duplicates Based on 'Customer_
ID':")
print(df_no_duplicates_by_id)
```

Here, we only consider the 'Customer_ID' column to identify duplicates, meaning that any repeated 'Customer_ID' will be considered a duplicate, regardless of the other columns. This approach might be useful in a context where 'Customer_ID' is expected to be unique across all records, and any repetition represents an error.

Scenario 2: Keeping the Last Occurrence of Duplicates Sometimes, duplicates may contain updated information, and we may want to keep the most recent occurrence. In this scenario, we'll identify duplicates based on all columns and keep the last occurrence. Using the same data in scenario one.

```
# Removing duplicates but keeping the last occurrence of each du  
plicate  
df_keep_last = df.drop_duplicates(subset=['Customer_ID', 'Name',  
'Address'], keep='last')  
print("\nDataFrame after Removing Duplicates, Keeping Last Occur  
rence:")  
print(df_keep_last)
```

By setting the `keep` parameter to '`last`', we ensure that the last occurrence of each duplicate is retained in the DataFrame. This approach might be appropriate when handling time-stamped data where the most recent entry represents the most accurate or relevant information. These scenarios show different ways to deal with duplicate data, which is similar to how real-world data often presents different problems. Whether they want to focus on specific columns or choose which occurrences to keep, these methods give analysts the power to clean data in a way that fits the needs of their analysis. These techniques give us more ways to deal with duplicate data and help us understand how data cleaning works in a more complete way.

Dealing with outliers

Outliers are data points that are very different from the other data points in the set. Even though they may seem like oddities, they can have a significant impact, affecting statistical measures, skewing distributions, and leading to wrong conclusions from an analytical point of view. Outliers are essential to understand, find, and deal with when cleaning up data, so pay close attention to this step. This section explains what outliers are, how they affect data analysis, and how to deal with them in the best way possible.

Types of outliers

In the vast world of data, outliers can show up in many ways. Each presents its own challenges and needs a different way to be found and dealt with. Understanding the different kinds of outliers is not just a matter of putting them into groups; it is also a necessary step in figuring out how complicated the data is. In this section, we look at the different kinds of outliers, such as univariate, multivariate, and contextual outliers, laying the groundwork for finding and treating them effectively.

- **Univariate Outliers:** These are the most extreme values in the distribution of a single variable. For example, consider a dataset containing the heights of adult humans. Most values will typically fall within a range of 150 to 200 centimeters. However, a data entry error leads to one record showing a height of 500 centimeters. This value is a univariate outlier, as it's an extreme value in a single variable's distribution. Univariate outliers are easy to find and usually come from mistakes in data entry or rare but normal events. In this case, the outlier is probably a mistake, and statistical measures like the Z-score or graphs like a histogram can help find it quickly.
- **Multivariate outliers** are more complicated because they involve more than one variable and a combination of them. Imagine a dataset containing the weights and heights of a group of individuals. Most data points might form a consistent pattern where weight increases with height. However, a record showing a weight of 50 kilograms for a height of 180 centimeters might be considered a multivariate outlier. This outlier is not extreme in either variable individually, but the combination of weight and height is unusual. Detecting multivariate outliers often requires more sophisticated techniques like Mahalanobis distance, which considers the relationship between multiple variables.
- **Global and Contextual Outliers:** Global outliers differ from the rest of the data set in a big way, while contextual outliers are unusual within a certain subgroup or context.
 - In a dataset of monthly temperatures for a city, a value of -10 degrees Celsius in summer would be considered a global outlier, as it deviates significantly across the entire dataset. Now consider the temperatures for different seasons. A value of -10 degrees Celsius in winter might be normal for that context, but if it were recorded in summer, it would be a contextual outlier.

These examples show that outliers can look different and that it's essential to know what kind of outlier you're dealing with so you can handle it correctly. Even though univariate outliers might be easy to spot, multivariate and contextual outliers often require a deeper understanding of how the data is organized and how it relates to other data.

Impact on analysis

The presence of outliers is more than just a statistical curiosity; it can have a big effect on how an analysis turns out. Outliers have a lot of different effects on analysis, from simple statistical measures to complex predictive models. This section goes into detail about how outliers can change results, confuse visualizations, and affect how well a model works. This shows how important it is to recognize and deal with these unusual observations.

- **Skewing results:** Outliers can have a significant effect on the mean, standard deviation, and other statistical measures, which can lead to skewed results.
- **Misleading visualizations:** Outliers can make graphs and plots show the wrong patterns or relationships in the data.
- **Affecting model performance:** In machine learning, outliers can lower the performance of algorithms, making predictions less accurate and reliable.

The following provides a visual of how the presence of outliers can effect the skew of a distribution.

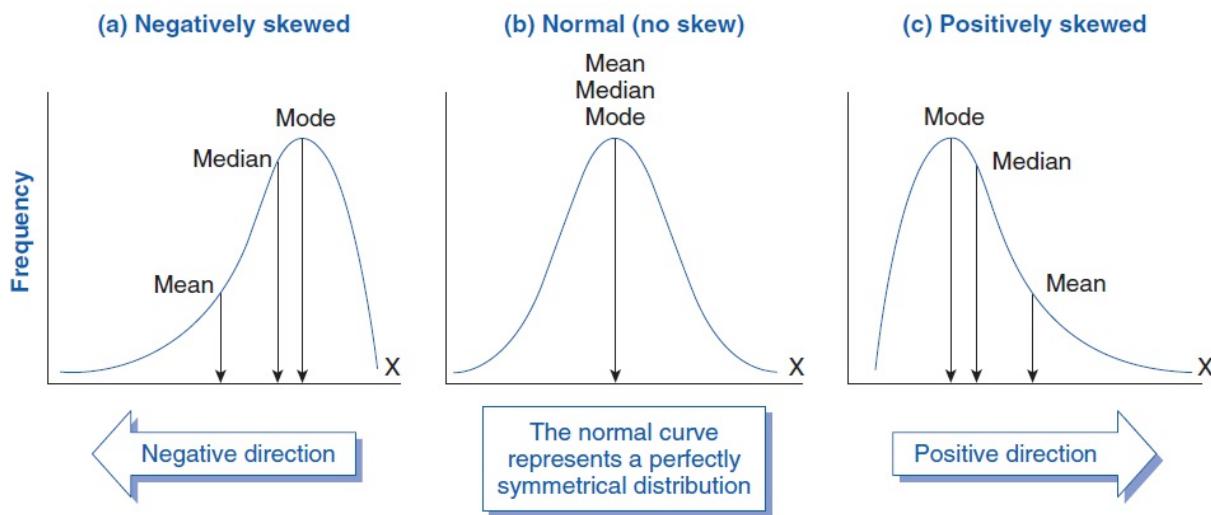


Figure 14.1: Distribution examples of normal vs skewed data with outliers.

Techniques for identifying and handling outliers

Because dealing with outliers is hard, you need a variety of tools that can be used in different situations. From finding unusual data points to figuring out

how to handle them, each step needs careful thought and execution. In this section, we'll show you how to find and deal with outliers using statistical methods, visualizations, and modeling strategies, among other things. Understanding these methods is important for keeping the integrity and reliability of the data analysis and making sure that the conclusions drawn are valid and can be used. There are multiple ways to identify outliers including the z-score and Mahalanobis distance. The following are example scenarios:**Scenario 1:** You want to identify outliers in a dataset containing exam scores using the Z-score.

```
from scipy.stats import zscore
# Example data
data = {'score': [50, 55, 40, 100, 45]}
df = pd.DataFrame(data)
# Calculating Z-scores
df['z_score'] = zscore(df['score'])
# Identifying outliers (e.g., |z_score| > 2)
outliers = df[np.abs(df['z_score']) > 2]
```

Explanation: The Z-score is a measure of how many standard deviations a data point is from the mean. Outliers are typically identified as data points with Z-scores below a lower threshold (e.g., -2) or above an upper threshold (e.g., 2).**Scenario 2:** You want to identify multivariate outliers in a dataset containing height and weight.

```
from scipy.spatial import distance
import numpy as np
# Example data
data = {'height': [170, 180, 150, 200], 'weight': [70, 80, 50, 30]}
df = pd.DataFrame(data)
# Computing Mahalanobis distance
cov_matrix = np.cov(df, rowvar=False)
inv_cov_matrix = np.linalg.inv(cov_matrix)
mean = df.mean()
mahalanobis_dist = df.apply(lambda x: distance.mahalanobis(x, mean, inv_cov_matrix), axis=1)
```

Explanation: Mahalanobis distance considers the correlations between variables, making it suitable for identifying multivariate outliers. Depending on the nature and cause of the outliers, different strategies may be employed: If an outlier is the result of an error, it might be best to remove it.

- **Scenario:** You have a dataset containing heights, and any height below 100 cm or above 250 cm is considered an outlier. This code filters the DataFrame by retaining only the rows with height values within the desired range, effectively removing the outliers.

```
import pandas as pd
# Example data
data = {'height': [150, 160, 50, 270, 180]}
df = pd.DataFrame(data)
# Removing outliers
df = df[(df['height'] >= 100) & (df['height'] <= 250)]
```

Transforming: Applying mathematical transformations can mitigate the influence of outliers.

- **Scenario:** You want to reduce the impact of outliers in a dataset containing salaries using a log transformation. The log transformation can reduce the impact of extreme values (outliers), making the data more suitable for many statistical analyses.

```
import numpy as np
# Example data
data = {'salary': [30000, 40000, 2000000, 50000]}
df = pd.DataFrame(data)
# Log transforming the 'salary' column
df['log_salary'] = np.log(df['salary'])
```

Imputing: Replacing outliers with statistical measures like the median or mean is another approach.

- **Scenario:** You have identified outliers in a dataset of ages and want to replace them with the median age. This code identifies outliers as ages below 18 or above 100 and replaces them with the median age of the dataset.

```
# Example data
data = {'age': [25, 30, 120, 22, 40]}
df = pd.DataFrame(data)
# Replacing outliers with median
median_age = df['age'].median()
df['age'] = np.where((df['age'] < 18) | (df['age'] > 100), median_age, df['age'])
```

Dealing with outliers is more than just a statistical exercise; it's a deep dive into the nature, cause, and importance of unusual data points in a dataset. If you can find outliers and handle them carefully, you can be sure that the insights you get from the data are not only statistically sound but also useful in real-world situations. As we continue to learn about the complexities of data cleaning, this understanding of outliers makes it possible for us to learn more about converting and formatting data types. There, we'll look into the subtleties of data representation and manipulation, making sure that our data is not only clean but also in the right format for analysis.

Cleaning and transforming data

The inherent complexities and inconsistencies present in raw data frequently make it difficult to derive actionable insights. Similar to refining ore to extract precious metals, cleaning and transforming data requires careful handling of various flaws and deliberate reshaping to reveal the hidden value. We will walk you through the critical steps of handling contradictions, converting categorical data, normalizing numerical features, and comprehending the significance of these transformations in revealing underlying patterns in this section of our exploration.

Handling inconsistencies

In the realm of data, inconsistencies are almost inevitable. They may stem from human errors, technological glitches, or even well-intended variations in data entry. These inconsistencies can disrupt the harmony of the dataset and lead to misleading results. Consider the following scenario: A customer database where country names are entered in different formats like "USA," "U.S.A.," and "United States." These inconsistencies can lead to confusion and incorrect analysis.

```
import pandas as pd
# Example data
data = {'country': ['USA', 'U.S.A.', 'United States', 'UK', 'U.K
.']}
df = pd.DataFrame(data)
# Standardizing the country names
df['country'] = df['country'].replace(['USA', 'U.S.A.'], 'United
```

```
States').replace(['UK', 'U.K.'], 'United Kingdom')
```

The code uses the `replace()` method to standardize the country names to a consistent format, so the analysis treats all the variations as the same entity. Addressing such inconsistencies involves standardizing the data to ensure that it adheres to a standard format or structure. This may include text cleaning, date format standardization, or handling variations in categorical values.

Converting categorical data

Categorical data, such as gender or product category, frequently needs to be converted into a machine-readable format. This transformation fills the gap between what people can understand and what computers need to do. The following is an example: In a marketing dataset, the column 'Response' contains values 'Yes' and 'No' indicating whether a customer responded to a campaign.

```
# Example data
data = {'Response': ['Yes', 'No', 'Yes', 'No']}
df = pd.DataFrame(data)
# Converting 'Yes' and 'No' to 1 and 0
df['Response'] = df['Response'].map({'Yes': 1, 'No': 0})
```

The code utilizes the `map()` method to convert categorical responses into a numerical binary format, making them suitable for statistical modeling. This data can be converted into a binary format (0 and 1) using encoding techniques like one-hot encoding or label encoding. Such conversion allows algorithms to process the categorical data efficiently.

Normalizing numerical features

Direct comparisons can be difficult because numerical data frequently varies in scale and unit. Without distorting the differences in the range of values, normalizing these characteristics brings them to a common scale.

```
from sklearn.preprocessing import MinMaxScaler
# Example data
data = {'price': [5, 100, 20], 'quantity': [10, 1000, 100]}
df = pd.DataFrame(data)
```

```
# Applying Min-Max scaling
scaler = MinMaxScaler()
df[['price', 'quantity']] = scaler.fit_transform(df[['price', 'q
uantity']])
```

This code leverages the `MinMaxScaler()` from the `scikit-learn` library to scale the features between 0 and 1, aligning them on a common scale. Applying normalization techniques like Min-Max scaling or Z-score normalization helps in transforming these features onto a similar scale, making them more suitable for analysis and machine learning algorithms. These code snippets serve as concrete examples of how data cleaning and transformation is performed in practice. By standardizing, encoding, normalizing, and cleaning the data, analysts prepare the raw information for subsequent analysis, enhancing the quality and reliability of insights derived from it.

Data validation

The accuracy of data analysis depends critically on data validation. Each data point that enters our analytical pipelines must be authenticated and verified. Without it, our findings and insights could be at risk of having errors and inconsistencies that compromise their quality. We will explore the complex facets of data validation in this section, highlighting its importance and going in-depth on its various methodologies. Fundamentally, data validation is the process of confirming and making sure that a dataset satisfies the required quality standards and is error-free. Consider it as data quality assurance, the process of double-checking the text and figures before approving them for further analysis. This step's significance cannot be overstated. The 'new oil' of decision-making, strategy-shaping, and pattern-illuminating is often referred to as data. The machinery of decision-making may stall out or even veer off course if this oil is tainted. By ensuring that data is valid before use, one can prevent erroneous inferences and preserve the integrity of data processing steps that come after.

Validation methods

The methods used to validate data are just as varied as the data landscape

itself. Let's examine some popular methods for validation and then immerse ourselves in scenarios that illustrate them. **Range checks** are the protective barriers ensuring data values stay within expected bounds. They're akin to setting minimum and maximum temperature thresholds on a thermostat. Scenario: Imagine a school database storing students' grades. Logically, these grades should lie between 0 and 100. If a data entry lists a student's grade as 150, it's a clear indication of an anomaly. Range checks would flag this, ensuring such errors are highlighted for correction.

```
import pandas as pd
# Example data
data = {'grades': [95, 50, 150, 88, -10]}
df = pd.DataFrame(data)
# Applying range checks to flag invalid grades
df['invalid_grade'] = (df['grades'] < 0) | (df['grades'] > 100)
```

This code snippet adds a new column 'invalid_grade' that will be **True** if the grade is outside the 0-100 range. This can help in identifying and correcting anomalous entries. In order to make sure that the format and layout of the data conform to expectations, **pattern matching** examines the data's structure. It is the skill of identifying a rhythm in data and ensuring that each data point moves in time with it. Scenario: Consider an international organization collecting phone numbers from different countries. A U.S. phone number has a distinct pattern, different from a U.K. number. By employing pattern matching, the organization ensures each number adheres to its country-specific format, ensuring accuracy and uniformity.

```
import re
# Example data
data = {'email': ['john@example.com', 'sarah.example', 'david@example']}
df = pd.DataFrame(data)
# Defining a pattern for valid email addresses
pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}+$'
# Applying pattern matching to flag invalid emails
df['invalid_email'] = df['email'].apply(lambda x: not bool(re.match(pattern, x)))
```

The code defines a regular expression pattern that matches valid email addresses. It then applies this pattern to the 'email' column and creates a new column 'invalid_email' to flag any addresses that do not meet the pattern. The

cornerstone of data trust is consistency. **Consistency checks** guarantee the harmony of the data and make sure it doesn't convey conflicting messages. Scenario: In a retail inventory, if the total number of items sold exceeds the initial stock without any replenishment, there's an inconsistency. Such discrepancies can skew analysis and projections, underscoring the need for rigorous consistency checks.

```
# Example data
data = {'initial_stock': [100, 50, 200], 'sales': [50, 60, 150],
        'replenished': [0, 20, 50]}
df = pd.DataFrame(data)
# Checking if sales exceed stock levels
df['inconsistent_data'] = (df['sales'] > (df['initial_stock'] +
                                             df['replenished']))
```

This code checks whether the sales for each product exceed the sum of the initial stock and replenished stock. If so, it flags the row as having inconsistent data in a new column 'inconsistent_data'. These code examples provide practical guidance on implementing essential data validation techniques. They serve as robust tools for safeguarding the data's quality and reliability, ensuring that it's primed for insightful analysis.

Summary

Data cleaning is a broad term for several tasks that are essential to the success of data analytics projects. Every step, from dealing with missing values and outliers to validating and normalizing data, has a big effect on the quality of the insights that can be drawn from it. Focusing on the best ways to clean data can lead to more useful and actionable results, which can help businesses and organizations make better decisions. Cleaning data is more important than just getting it ready. It is at the heart of what makes data analytics such a useful tool in today's data-driven world.

17 Exploratory Data Analysis Case Study

Join our book community on Discord

<https://packt.link/EarlyAccessCommunity>



Welcome to the Exploratory Data Analysis Case Study. This chapter will provide you with hands-on experience in conducting a comprehensive exploratory data analysis (EDA) on a real-world dataset. EDA is a fundamental skill for any data analyst, as it enables you to comprehend the subtleties of your data, recognize patterns, and make informed decisions. This chapter will guide you through a case study that simulates a typical business scenario. You will be provided with a dataset and business questions to answer. This exercise lets you utilize various EDA techniques we've previously covered to inform business strategies. At the conclusion of this chapter, you will have a solid understanding of how to analyze data and effectively communicate your findings. In this chapter we're going to cover the following main topics:

- Compute descriptive statistics to summarize the distribution's central tendency, dispersion, and shape.
- Deriving valuable inferences from your EDA and realizing the ways in which these findings can address business inquiries.

Technical Requirements

The dataset for the case study can be found in this book's repository at the

following link: https://github.com/PacktPublishing/Becoming-a-Data-Analyst-First-Edition/blob/main/Chapter-17-EDA-Case-Study/MotionEase_Sales_Data.csv

E-commerce Sales Optimization Case Study

You work as a data analyst for MotionEase, an online retailer that offers everything from fashion to electronics. For the last six months, the company's sales have been declining, and the management is worried about the viability of the enterprise going forward. The management wants to understand the underlying factors contributing to the decline in sales. They are particularly interested in:

1. Identifying the categories of products that are performing well or poorly.
2. Understanding customer behavior, including spending patterns and frequency of purchases.
3. Evaluating the effectiveness of various sales channels.

Your task is to conduct an exploratory data analysis to uncover insights that can help the company reverse the declining sales trend.

Time Series Analysis

1. The sales at MotionEase have been like a rollercoaster, going up and down every month. Can you chart this fluctuating journey and identify any seasonal patterns that might explain these ups and downs?
2. It's a busy week at MotionEase, but is every day equally busy? Your task is to find out if weekdays bring in more sales than weekends or vice versa. This could help the company in planning their staffing and marketing strategies.

Customer Segmentation

1. MotionEase wants to roll out a VIP program but isn't sure who to invite. Your task is to identify the top 10% of customers based on their spending. What makes these customers special?
2. Customer retention is a big deal at MotionEase. The company wants to

reward customers who come back to make multiple purchases. Can you identify these loyal customers and find out what keeps them coming back?

Product Analysis

1. In the vast inventory of MotionEase, some products are stars while others are not. Can you spotlight the top 5 best-selling products in each category and suggest why they might be the customer favorites?
2. Returns are a headache for any retail business. MotionEase is no exception. Your challenge is to identify products that are frequently returned and hypothesize why this might be happening.

Payment and Returns

1. People have different preferences when it comes to parting with their money. Can you find out if there's a preferred payment method for higher-value orders? This could influence future payment options offered by MotionEase.
2. Returns are like a leak in a boat, and MotionEase wants to plug it. Are there specific categories or payment methods that are more prone to returns? Your findings could help the company tighten its return policy.

Case Study Answers

You can find all the answers with explanations in this book's GitHub repository at the following link:

https://github.com/PacktPublishing/Becoming-a-Data-Analyst-First-Edition/blob/main/Chapter-17-EDA-Case-Study/EDA_Case_Study_Answers.ipynb

Summary

In this chapter, we conducted an in-depth case study to practice Exploratory Data Analysis (EDA) skills. Following introducing a business scenario involving MotionEase, we provided a dataset for investigation. We covered

various aspects of EDA through hands-on exercises. We answered questions regarding, among other things, monthly sales trends, the impact of weekdays on sales, and customer segmentation. Each question was presented in a narrative format, and solutions included code explanations to ensure a comprehensive understanding of the techniques employed. As we transition into the next chapter, "Introduction to Data Visualization," we'll build upon the foundational skills acquired in this chapter. Data visualization is a crucial aspect of data analysis, as it provides a graphical representation of data to make insights easier to comprehend. We will investigate various visualization types and learn how to create them using Python libraries. Therefore, let's add another layer of sophistication to our data analysis toolbox!